URGE: Motor para o Desenvolvimento de Jogos Eletrônicos

Alexandre A. Abdalla de O. Cardoso
DCC - UFRJ
abdalla@dcc.ufrj.br

Vitor Carneiro Maia
DCC - UFRJ
vcm@ufrj.br

Rodrigo de Toledo
DCC, PPGI - UFRJ
rtoledo@dcc.ufrj.br

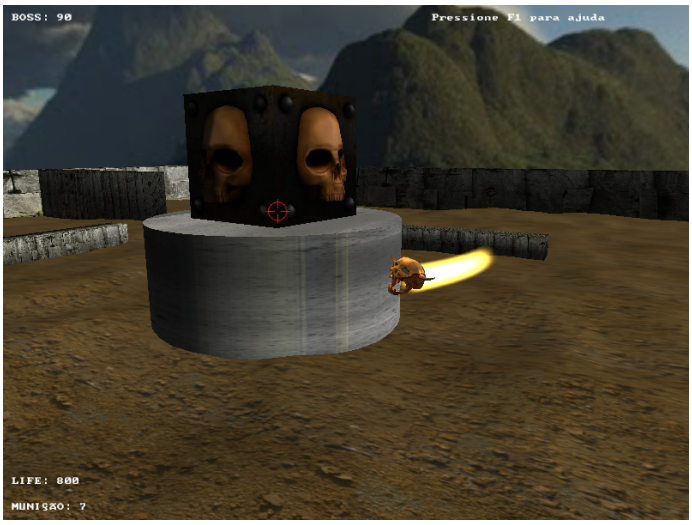

Figura 1. Exemplo de jogo criado com a URGE.

***Resumo*—URGE is a engine for the development of three-dimensional computer games.**

It is based on its two cores: The Visualization System and the Physics System. Both are responsible for simulating the reality of an extensive range of materials in nature, and all the dynamic interaction between bodies based on the laws of physics. In this work, we discuss the entire build process, including techniques involved in the development of each module, and the evolution of the project.

***Keywords*-game development; 3D game engine**

I. Introdução

Com as tecnologias hoje disponíveis, é possível desenvolver jogos eletrônicos com grande agilidade. Não é mais necessário criá-los a partir de recursos de programação em baixo nível, como bibliotecas ou APIs. 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 teleprocessamento em redes, existentes em qualquer aplicação interativa em tempo real. As *engines* encapsulam uma série de artifícios necessários na programação de qualquer jogo eletrônico, o que torna muito mais ágil e simples o esforço feito em sua criação.

Contribuições: Este artigo apresenta uma nova *engine*, especializada na criação de jogos 3D para computadores. A proposta da URGE (Unified Resources GDP/Game engine) baseia-se em dois objetivos: completude e simplicidade. Buscando atingir a completude, a ferramenta reúne as várias técnicas de computação gráfica aplicada em jogos eletrônicos, simulação física em tempo real e tele-processamento em redes, atentado ao fato de trabalhar em computadores de baixo desempenho. Para garantir a simplicidade, a engine foi adaptada para programadores sem experiência em desenvolvimento de jogos poder trabalhar, isso se traduz numa forma intuitiva de se programar.

A. Visão Geral

A URGE se divide em dois núcleos: *Rendering* e *Physics*. O núcleo *Rendering* engloba toda a parte de visualização, iluminação e mapeamentos, e o núcleo *Physics* engloba o tratamento de colisão, resposta de colisão e simulação física. Os dois núcleos são unificados pela entidade base da URGE: a classe Object. Alguns recursos que a ferramenta dispõe são: Simulação de terrenos através de *height maps*, integração com *shaders* (GLSL), iluminação pixel a pixel, geração automática de primitivas tri-dimensionais, grafo de cena, otimização por *octrees*, reunião de várias técnicas de mapeamentos de textura

(*Bump Mapping*, *Environment Cube Mapping*, *Parallax Occlusion Mapping*...), neblinas, sistemas de partículas, simulação física para corpos rigidos baseada em impulso, detecção e resposta de colisão e *skydomes*. A Figura 6 mostra uma versão simplificada do diagrama de classes da URGE.

II. 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. Por ter 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. Na Figura 3 estão representas as camadas operacionais.

O objetivo desta representação é enfatizar o processo de geração de um jogo a partir da URGE. Estruturas mais próximas do topo constituem níveis mais baixos de implementação, e estruturas mais próximas ao produto final (o jogo) representam partes de alto nível de programação, manipuláveis por usuários.

A. Bibliotecas de baixo nível

As bibliotecas de baixo nível utilizadas foram *OpenGL* para desenvolver a parte gráfica, *Allegro* para capturar comandos de entrada e som, *Pthread* para realizar *multi-threading* por parte do gerenciador de cena, e *Winsock* para desenvolver as classes de conexão (TCP ou UDP) com a internet.

B. Núcleos de visualização e de física

O núcleo de visualização (*Rendering*) e o sistema de simulação física (*Physics*), juntos, constituem o coração da URGE.

O núcleo de visualização engloba todas as classes e métodos responsáveis por exibir objetos na tela, com o máximo de proveito obtido na relação entre qualidade gráfica e performance (veja a Figura Figura 2). Para tal são considerados 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, carregamento de modelos 3D, mapeamentos, iluminação via *shaders* e sistemas de partículas para simular fogo ou rastros luminosos.

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

C. Object

Existe uma entidade básica que deve ser superclasse de qualquer elemento visual e/ou físico dentro do espaço de uma cena. Essa entidade, denominada *Object*, é o elo que unifica os dois núcleos principais da engine (*Rendering* e *Physics*).

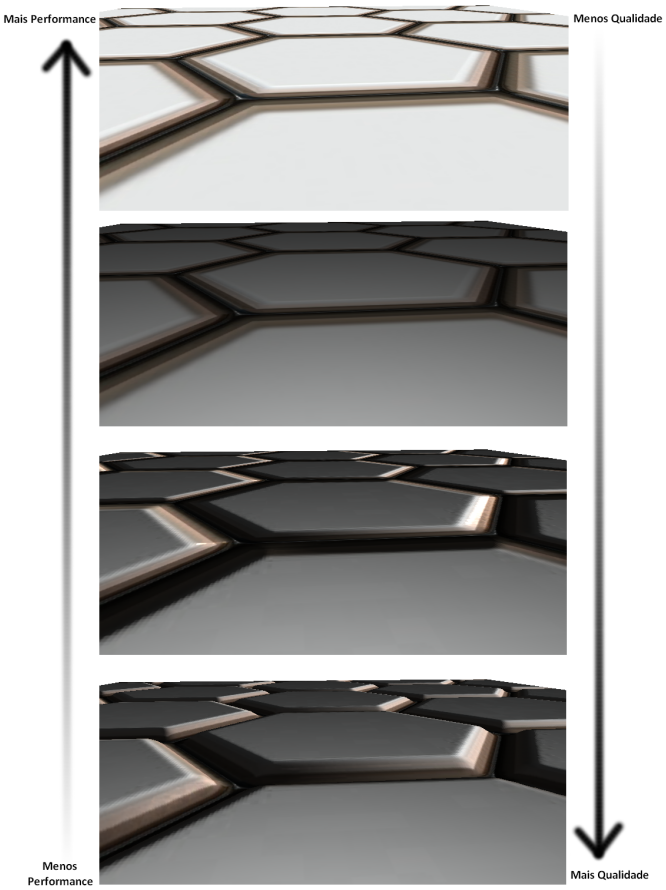

Figura 2. Qualidade X Performance

Entretanto, há um certo conjunto de classes que pertencem a cena do jogo, mas não são derivados da classe *Object*, tais classes constituem o módulo denominado *Environment*, observe seus detalhes no diagrama estrutural da URGE (Figura Figura 3).

D. Templates

Para facilitar a programação, 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.

A classe *Terrain* é capaz de gerar um terreno a partir de um *height map*, que por sua parte, é definido através de uma imagem carregada de um arquivo. Alguns outros parâmetros editáveis do terreno são suas três dimensões de tamanho e a textura difusa.

Há a possibilidade de simular oceanos, através de uma entidade nomeada *Ocean* a qual, a partir de dois mapas de normais, gera um ambiente semelhante a um oceano, com ondulações, reflexões e refrações. Todos estes parâmetros são também editáveis, assim como a região que será ocupada pela água.

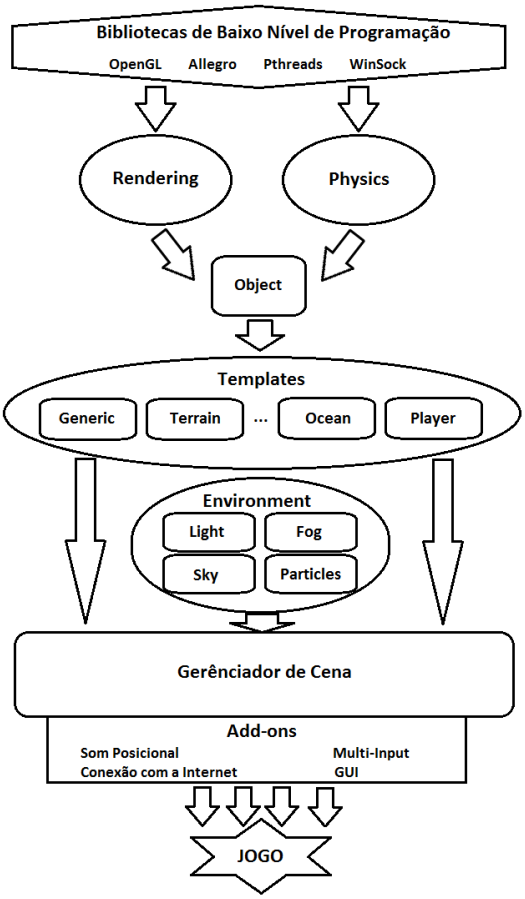

Figura 3. Modelo Estrutural.

Existe uma classe para vários propósitos, relativos à necessidade do usuário da engine. A *Generic* é capaz de se transformar em diferentes objetos de cena, como primitivas geométricas, modelos 3D animados ou estáticos e imagens 2D em um mundo 3D (*billboard sprites*). Todos sujeitos a simulação física.

O módulo Templates é bastante útil para programadores iniciantes, pois fornece o acesso a recursos de alta qualidade da URGE sem exigir conhecimento prévio das técnicas de programação que os compõem. Na Figura Figura 5 podemos observar o código de um projeto de um jogo 3D feito apenas com Templates da URGE apesar de ser apenas um módulo inicial, repare que os recursos são invocados de forma relativamente intuitiva.

E. Environment

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 ou físicos, porém são importantes no processo de criação de qualquer jogo.

O módulo *Environment* reúne entidades que também são administradas pelo gerenciador de cena. Luz, Neblina, Partículas

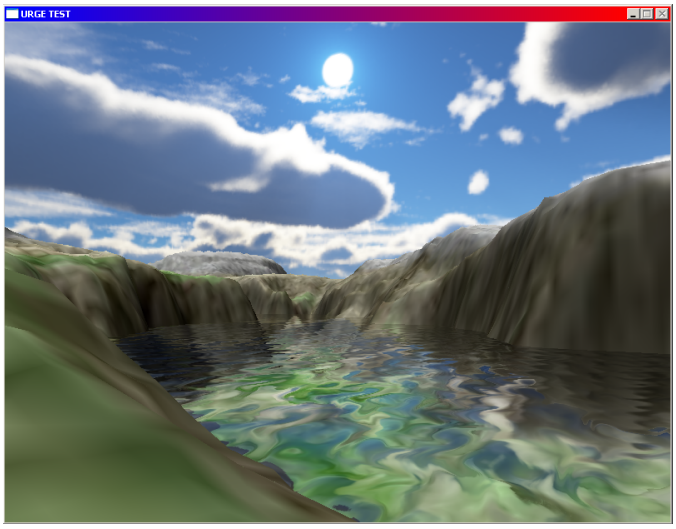

Figura 4. Exemplo das entidades *Terrain* e *Ocean*.

e Céu são alguns exemplos.

A classe *Light* representa a luz de um cenário, e é essencial para garantir a imersão e o realismo. A URGE suporta infinitas luzes dentro de uma mesma cena.

A classe *Particle Emitter* é responsável por vários efeitos visuais, tais como Fogo, fumaça e rastros luminosos.

F. Gerência de cena

É possível criar um jogo apenas com os dois grandes núcleos da *engine*, o que já é feito pela classe *Object* conforme visto. Todavia, se somente juntássemos ambos os núcleos, teríamos um sério problema de ineficiência causado por *overheads*, devido à falta de um gerenciamento na interação de todas as entidades do jogo. Por isso foi criada uma entidade de gerenciamento de cena. Seu objetivo é evitar *overheads* de visualização e de simulação física, ou seja, evitar desenhar objetos não vistos pelo jogador e evitar testes de colisão desnecessários. Para tal, o gerenciador de cena executa algoritmos como *Viewing Frustum Culling* e *Octrees*.

G. Recursos extras

A engine possui um sistema de recursos adicionais para potencializar a liberdade do usuário. Chamamos esses de *Add-ons*. 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. Há uma entidade chamada *Controller* que gerencia comandos de entrada.

A URGE conta com um dispositivo de som posicional, que ajusta o volume e o balanço do som em função da posição do mesmo.

É também possível criar jogos em que vários jogadores interagem por meio de uma conexão com a local ou com a Internet através do protocolo TCP ou UDP com suporte a conexão *multicast*.

```
URGE_BEGIN
{
    gimme_window(800,600);

    FirstPersonCamera player;
    Scenario scenario;
    Generic enemy;
    Sky sky;
    Light light;
    Terrain terrain;
    Ocean ocean;
    FireBall fireball;

    ocean.loadNormalMap("media/water/wave_n.png");
    ocean.color(5,60,120);
    ocean.scale(500);

    player.position(0,20,0);

    enemy.load("media/models/mask/mask.md2");

    terrain.load("media/terrain/heightmap.jpg");
    terrain.loadTexture("media/terrain/heightmap_diffuse.jpg");

    light.directional();
    light.direction(-1,1,-1);

    sky.loadTexture("media/sky/cloudy");

    scenario.insert(player);
    scenario.insert(sky);
    scenario.insert(enemy);
    scenario.insert(terrain);
    scenario.insert(light);
    scenario.insert(ocean);

    scenario.prepare();

    do
    {
        if (Keyboard::hit(Keyboard::ESC)) break;

        // Fazendo enemy "seguir" player
        enemy.velocity() += (player.position() - enemy.position()).normalize()*5.0;
        enemy.direction() = enemy.velocity();

        scenario.update();

        next_frame();
    }while(true);

    thanks_byebye();
}
URGE_END
```

Figura 5. Exemplo de uma versão inicial de jogo usando apenas Templates

Embora ainda não finalizado, é desejado que na versão final seja possível criar interfaces com usuário: botões, menus, caixas de opção e de texto, *comboboxes* e similares.

H. O Jogo

Todas essas funcionalidades e classes juntas, por fim, dão origem ao jogo. Este fluxo leva em consideração apenas a etapa de programação de um jogo.

III. Processo de Criação

Nesta sessão será mostrada a modelagem da URGE, e as vertentes de engenharia de software adotadas. A equipe utilizou uma adaptação da metodologia ágil como forma de trabalho. Alguns conceitos serão comentados a seguir.

A. Adaptação do Planning Poker

Etapas de desenvolvimento foram ponderadas em relação ao custo de desenvolvimento e sua prioridade, inspirando-se na técnica do *Planning Poker*.

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.

B. Adaptação de User Stories

A priorização dos recursos da engine foi adaptada para *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, o processo de priorização não pode ser mais feito em função de requisitos técnicos. Através das User Stories, os desenvolvedores passaram a priorizar o trabalho de acordo com o que seria mais interessante tanto para o jogador quanto para o usuário da URGE.

C. Planejamento Contínuo

User Stories eram criadas e então quebradas em histórias menores. Histórias grandes, chamadas de épicos, definem idéias gerais sobre um tema. “Recursos gráficos” é um exemplo de épico. Este épico pode ser quebrado em, por exemplo “Shadow Mapping” e “Bump Mapping”. Depois de quebrados os épicos, a equipe avaliava quais as histórias menores eram mais prioritárias, e as desenvolvia. Isto é chamado de Planejamento Contínuo.

D. UML simplificado

A última versão do diagrama UML de classes 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. Veja na Figura 6.

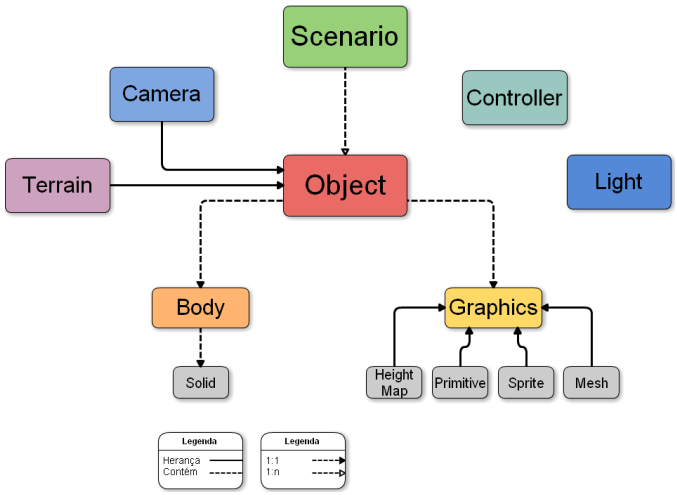

Figura 6. Diagrama de Classes - versão simplificada.

IV. Técnicas e conceitos envolvidos

Nesta sessão serão comentados superficialmente algumas tecnicas e algoritmos implementados pela URGE.

A. Programação de Shader

Um dos principais objetivos do sistema de visualização da URGE é gerar gráficos de alta qualidade. Isso implica em um problema: O *pipeline* padrão do OpenGL não dispõe de grande parte dos recursos para a programação de técnicas que possibilitam a geração de gráficos que atendam a esse objetivo. Foi necessário que os programadores da URGE migrassem para um pipeline secundário do OpenGL: O pipeline programável [1]. Essa diferença no pipeline é vantajosa, pois o programador agora pode implementar suas próprias técnicas de visualização usufruindo da aceleração do núcleo de processament gráfico do computador.

B. Iluminação de Materiais

Dado uma superfície e um foco de luz, o OpenGL é capaz de simular o efeito de iluminação nativamente (usando o *Gouraud Shading*) 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. Entretanto esse sistema apresenta falhas quando a iluminação é aplicada a superfícies com poucos vértices. Os programadores da URGE precisaram tirar proveito do uso de programas shaders para criar um efeito de iluminação chamado *Blinn-Phong Shading* [2], a qual a iluminação é interpolada pixel a pixel.

C. Normal Mapping

Em jogos, muitas vezes queremos representar de forma fiel matérias que possuem uma aparência áspera, irregular ou esburacada, como é o caso de tijolos, mármore e pedras. Também é comum encontrar modelos tridimensionais com um alto nível de detalhes. Para retratar tais detalhes com fidelidade, é preciso estender o número de vértices do objeto. Isso seria ineficiente, pois modelos com muitos vértices devem ser evitados em aplicações de tempo real por questão de performance.

O algoritmo *Bump Mapping* ou *Normal Mapping* [3] apresenta uma possível solução para esse problema. Ele distorce as normais de um modelo, pixel a pixel, de acordo com uma função ou textura que chamamos de mapa de normal (*normal map*), adicionando detalhes ao efeito de iluminação.

D. Parallax Mapping

O *Parallax Mapping*, considerado uma evolução do *Bump Mapping* convencional, que distorce também as coordenadas de textura com o objetivo de gerar relevos a nível de pixels. Isto fornece uma impressão tridimensional da própria texturização.

A versão conhecida como *Parallax Occlusion Mapping* (também implementada pela URGE) constitui-se em um aprimoramento do *Parallax Mapping* convencional. Nesse caso, são efetuados testes de oclusão partindo do principio do *Ray Casting*, adicionando uma noção de obstrução do relevo [4].

E. Environment Mapping

Environment Mapping é um conjunto de técnicas que buscam aproximar de forma convincente a aparência de superfícies reflexivas, como espelhos, materiais metálicos ou plásticos [5]. Podemos também estender seu uso para simulações de superfícies refrativas como vidro, água ou cristal. Na engine também são simulados a dispersão cromática e o efeito fresnel [6]. O funcionamento do *Environment Mapping* baseia-se em de uma textura pré-calculada chamada *environment map*. Tal textura deve representar a cena que a superfície em questão reflete. O último passo, está no procedimento de texturização. Diferentemente dos mapeamentos convencionais usados nas texturas difusas, o mapeamento, nesse caso, deve ser feito de forma dinamica, ou seja, deve ser feito de forma interativa, proporcional à normal da superficie e à visão do observador.

F. Renderização de Terrenos

A URGE considera um terreno, como sendo uma superfície tridimensional sem 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. [7]

A URGE possui suporte a visualização de terrenos baseado em técnicas simples programadas em CPU. O objetivo disso é adaptar o algoritmo para funcionar em computadores com dispositivos gráficos menos potentes. A visualização de terrenos da engine implementa técnicas de otimização de desenho como o uso de LODs (*Level of Detail*) [8] e VBOs (*Vertex Buffer Objects*) [9].

G. Sistema de Detecção de Colisões

A primeira etapa do núcleo de física é realizar testes de colisão em função de formas geométricas [10]. 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. As três diferentes estruturas geométricas suportadas pela engine são: Esferas, AABBs e OBBs.

A Detecção de colisão por esferas é o tipo mais simples de se efetuar. Basta comparar a distancia dos centros com a soma dos raios das mesmas.

A Detecção de colisão por AABB (*Axis Aligned Bounding Box*, ou caixa alinhada ao eixo de origem) é feita em função das situações em que não há colisão, caso todos eles falhem, a colisão foi detectada. Essa forma é bastante eficiente, pois para o espaço tri-dimensional existem apenas seis casos em que dois AABBs não se colidem.

No caso de OBBs (*Oriented Bounding Box*), 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 [11].

H. Resposta de Colisão

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 [12].

I. Simulação Física Baseada em Impulso

Infelizmente, efetuar resposta de colisão não leva em conta fatores físicos importantes, isto é, ao colidir precisamos modificar, segundo as leis da física, as velocidades dos corpos em função de atributos como massa e elasticidade. 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 [13]. 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.

J. View Frustum Culling

Uma técnica utilizada pelo gerenciador de cena com o intuito de eliminar processamento gráfico desnecessário é o *View Frustum Culling*. Ela consiste em evitar desenhar tudo que não está dentro do campo de visão do observador [14]. O campo de visão é denominado de *View Frustum* e o ato de evitar desenhar elementos fora do mesmo denomina-se *Culling*.

K. Octrees

Octrees são usadas pelo gerenciador de cena da engine para eliminar processamento desnecessário por parte do núcleo de física [6]. Seu funcionamento baseia-se na divisão espacial do cenário em oito sub-regiões cúbicas. Esse algoritmo é feito de forma recursiva, isto é, as sub-regiões são divididas em oito novas sub-regiões até atingir um nível máximo de profundidade ou satisfazer as condições de parada do algorimo. Evidentemente, o *output* dessa técnica é uma árvore a qual cada nó possui oito filhos (com exceção das folhas), afinal, cada nó representa uma partição do espaço em forma de AABB.

V. Conclusão

Este trabalho apresentou a URGE, uma nova engine, que busca atingir dois objetivos: completude e simplicidade.

Para isso a engine dispõe diversos recursos de visualização gráfica tri-dimensional, simulação física em tempo real e teleprocessamento em redes por parte de seus módulos internos. O acesso a tais recursos foi um ponto crucial para reduzir ao máximo o esforço de se programar um jogo, por isso a ferramenta foi feita para ser controlada da forma mais intuitiva possível.

Vimos que a URGE 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 realizar o desenho de materiais visíveis obtendo o maior proveito possível na relação entre qualidade gráfica e performance. 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.

Também 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.

Referências

[1] R. J. Rost, B. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen, *OpenGL Shading Language*, 3rd ed. Addison-Wesley Professional, 2009.

[2] J. F. Blinn, *Simulation of Wrinkled Surfaces*, 3rd ed. ACM, 1978, vol. 12.

[3] V. Krishnamurthy and M. Levoy, "Fitting smooth surfaces to dense polygon meshes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996.

[4] M. McGuire and M. McGuire, "Steep parallax mapping," *I3D 2005 Poster*, 2005.

[5] B. Cabral, M. Olano, and P. Nemec, "Reflection space image based rendering," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999.

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

[7] *Sibgrapi 2012, Proceedings of the XXV Brazilian Symposium on Computer Graphics and Image Processing*. Maceió, Brazil: IEEE, august 2012.

[8] H. Hoppe, "View-dependent refinement of progressive meshes," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997.

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

[10] C. Ericson, *Real-Time Collision Detection*. Elsevier, 2005. [Online]. Available: http://books.google.com.br/books?id=WGpL6Sk9qNAC

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

[12] J. Van Verth and L. Bishop, *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. Elsevier Science, 2008. [Online]. Available: http://books.google.com.br/books?id=zkEY9RIm4WkC

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

[14] U. Assarsson and T. Möller, "Optimized view frustum culling algorithms for bounding boxes," *J. Graph. Tools*.