UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE UM AMBIENTE 3D PARA JOGOS, UTILIZANDO A *ENGINE* CRYSTAL SPACE COM DIRECTX E LINGUAGEM C++

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

RODRIGO PETER

BLUMENAU, JUNHO/2002

2002/1-66

PROTÓTIPO DE UM AMBIENTE 3D PARA JOGOS, UTILIZANDO A *ENGINE* CRYSTAL SPACE COM DIRECTX E LINGUAGEM C++

RODRIGO PETER

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

____

Prof. Dalton Solano dos Reis — Orientador na FURB

____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

____

Prof. Dalton Solano dos Reis

____

Prof. Paulo César Rodacki Gomes

____

Prof. Roberto Heinzle

AGRADECIMENTOS

Agradeço primeira e acima de tudo à Deus, pela graça divina da vida e oportunidades nela oferecidas. À minha querida Clarice, pelo apoio e grande paciência despojados durante a elaboração deste trabalho, o qual nos privou de muitas coisas, mas que com certeza colheremos os frutos do tempo dedicado.

Meus agradecimentos estendem-se também ao meu orientador, professor Dalton Solano dos Reis, pelas muitas horas de dedicação. Muito obrigado à minha família e a todas as outras pessoas que não cito explicitamente, mas que contribuíram direta ou indiretamente para o desenvolvimento do trabalho.

SUMÁRIO

AGRADECIMENTOS III

LISTA DE FIGURAS VI

LISTA DE QUADROS VI

LISTA DE SIGLAS E ABREVIATURAS VIII

RESUMO IX

ABSTRACT X

1 INTRODUÇÃO 1

1.1 OBJETIVOS DO TRABALHO 5

1.2 ESTRUTURA DO TRABALHO 5

2 CRYSTAL SPACE 7

2.1 UMA APLICAÇÃO CRYSTAL SPACE 10

2.1.1 O SISTEMA SCF 10

2.1.2 CONTADOR DE REFERÊNCIAS (*REFERENCE COUNTER*) 11

2.1.3 REGISTRO DE OBJETOS (*THE OBJECT REGISTRY*) 11

2.1.4 GERENCIADOR DE *PLUGIN* (*THE PLUGIN MANAGER*) 12

2.1.5 FILA DE EVENTO (*THE EVENT QUEUE*) 12

2.1.6 O RELÓGIO VIRTUAL (*THE VIRTUAL CLOCK*) 12

2.1.7 ANÁLISE DE LINHA DE COMANDO (*THE COMMANDLINE PARSER*) 13

2.1.8 O GERENCIADOR DE CONFIGURAÇÃO (*THE CONFIG MANAGER*) 13

2.1.9 DRIVERS DE ENTRADA (*THE INPUT DRIVERS*) 13

2.1.10 A CLASSE CSINITIALIZER 14

2.2 CRIANDO UMA APLICAÇÃO 14

2.2.1 UM ARQUIVO SIMPLES 14

2.2.2 TRATAMENTO DE EVENTOS (*EVENT HANDLING)* 17
2.2.3 CRIANDO UM “MUNDO” 18
2.2.4 A CÂMARA 21
2.2.5 LOCOMOÇÃO (MOVENDO-SE NO AMBIENTE) 25
3 MICROSOFT DIRECTX 26
3.1 DIRECTDRAW 27
3.2 DIRECT3D 28
3.2.1 CAMADAS (*LAYERS*) 29
3.2.2 LOCAL 29
3.3 DIRECTX E COM 30
4 DESENVOLVIMENTO DO PROTÓTIPO 31
4.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 31
4.2 ESPECIFICAÇÃO E IMPLEMENTAÇÃO 31
4.2.1 ARQUIVO DE CONFIGURAÇÃO (.CFG) 32
4.2.2 O ARQUIVO “MUNDO” 32
4.2.3 A CLASSE SIMPLE 35
4.2.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO 43
5 CONCLUSÕES 49
5.1 EXTENSÕES 50
REFERÊNCIAS BIBLIOGRÁFICAS 51
ANEXO A: CÓDIGO FONTE DO ARQUIVO ‘`SIMPLE.CPP`’ 53
ANEXO B: CÓDIGO PARA ADIÇÃO DE ITENS À JANELA NO ARQUIVO ‘`SIMPLE.CPP`’ 57
ANEXO C: CÓDIGO FONTE ‘SIMPLEMAP.CPP’ 59

LISTA DE FIGURAS

FIGURA 1 – AMBIENTE CONSTRUÍDO COM O CS 8

FIGURA 2 – *ENGINE* 3D, INTERMEDIAÇÃO ENTRE A APLICAÇÃO E O *HARDWARE* 9

FIGURA 3 – LOCALIZAÇÃO DO DIRECT3D 30

FIGURA 4 – MODELO DA CLASSE SIMPLE 35

FIGURA 5 – SEQÜÊNCIA DA EXECUÇÃO DOS MÉTODOS DA CLASSE 'SIMPLE' .. 43

FIGURA 6 – PLANTA BAIXA DO PROTÓTIPO 44

FIGURA 7 – CENA INICIAL DO PROTÓTIPO 45

FIGURA 8 – FUMAÇA E FOGO 46

FIGURA 9 - TRANSPARÊNCIA 47

FIGURA 10 – SEM DETECÇÃO DE COLISÃO 48

LISTA DE QUADROS

QUADRO 1 – MACROS DO ARQUIVO 'SCF.H' 11

QUADRO 2 – CABEÇALHO DO ARQUIVO 'SIMPLE.H' 15

QUADRO 3 – MÉTODOS PRIVADOS DA CLASSE 'SIMPLE' 17

QUADRO 4 – CÓDIGO PARA FINALIZAR COM A TECLA 'ESC' 18

QUADRO 5 – INICIALIZAÇÃO DA FUNÇÃO TRATADORA DE EVENTO (*HANDLE EVENT*) 18

QUADRO 6 – ADIÇÃO DE UM PONTEIRO 19

QUADRO 7 – INSTANCIAÇÃO DA CLASSE 'IVIEW'. 22

QUADRO 8 – CLASSE 'IVIEW' NO CONSTRUTOR/DESTRUTOR 22

QUADRO 9 – CÓDIGO FONTE PARA ADIÇÃO DA CÂMARA....................................22
QUADRO 10 – CÓDIGO PARA DESENHAR NA TELA 23
QUADRO 11 – ALTERAÇÃO NO CÓDIGO DO TRATADOR DE EVENTOS (*EVENT HANDLER*)..24
QUADRO 12 – MOVIMENTANDO A CÂMARA 25
QUADRO 13 – FORMATO DE UM ARQUIVO DE CONFIGURAÇÃO.........................32
QUADRO 14 – TRECHOS DE UM ARQUIVO DE DEFINIÇÕES DE AMBIENTE.....33
QUADRO 15 – CRIAÇÃO DE UMA NOVA TEXTURA 34
QUADRO 16 – CRIAÇÃO DE UM NOVO MATERIAL 34
QUADRO 17 – ALTERAÇÃO DO CHÃO NO SETOR..34
QUADRO 18 – ESTRUTURA DA CLASSE ‘SIMPLE’ 35
QUADRO 19 – CÓDIGO DO CONSTRUTOR E DESTRUTOR DE ‘SIMPLE’ 36
QUADRO 20 – CÓDIGO DO MÉTODO ‘INITIALIZE()’...37
QUADRO 21 – CÓDIGO DO MÉTODO ‘START()’..39
QUADRO 22 – CÓDIGO DOS MÉTODOS ‘SIMPLEEVENTHANDLER()’ E ‘HANDLEEVENT()’..39
QUADRO 23 – CÓDIGO DO MÉTODO ‘SETUPFRAME()’...40
QUADRO 24 – CÓDIGO DO MÉTODO ‘FINISHFRAME()’...41
QUADRO 25 – CÓDIGO DO MÉTODO ‘LOADMAP()’ 41

LISTA DE SIGLAS E ABREVIATURAS

2D	Duas Dimensões
3D	Três Dimensões
6DOF	*Six Degrees-of-Freedom*
API	*Application Program Interface*
BSP	*Binary Space Partitioning*
COM	*Component Object Model*
CS	Crystal Space
FSF	*Free Software Foundation*
HAL	*Hardware Abstraction Layer*
HEL	*Hardware Emulation Layer*
KDS	*Kit* de Desenvolvimento de *Software*
LGPL	*Library General Public License*
RPG	*Role-Playing Game*
SCF	*Shared Class Facility*
SDK	*Software Development Kit*
VFS	*Virtual File System*
VOS	*Virtual Object System*

RESUMO

Este trabalho apresenta os fatores relacionados à construção de um ambiente 3D. Mais especificamente, como construir um ambiente em 3D utilizando a *engine* Crystal Space. São abordadas as principais funções, técnicas e modelos desta *engine*, necessários à geração do ambiente, visualização e aplicação de "efeitos", tais como iluminação colorida, objetos 3D, transparência (*alpha*) e espelhamento. O trabalho demonstra também, como a *engine* utiliza-se das funções da *Application Program Interface* (API) do DirectX, para a construção do ambiente e comunicação com as utilidades do sistema operacional, como a placa de vídeo.

ABSTRACT

This work presents factors related to the construction of an 3D environment. More specifically, how to construct an environment in 3D, utilizing the Crystal Space engine. Are approached the principal functions, techniques and models from this engine, necessary to generation of the environment, visualization and application of “effects”, such as colored illumination, 3D objects, alpha transparence and mirroring. The work also demonstrate, how the engine utilize the functions of Application Program Interface (API) of DirectX, for the environment construction and the communication with the operational system utilities, as the video plate.

1 INTRODUÇÃO

Conforme Rio Gráfica (1985), os jogos eletrônicos integram o mundo da computação porque divertem, instruem e desenvolvem. Fascinantes e irresistíveis, simulam situações da vida real ou da imaginação do ser humano.

Segundo Cardoso (2000?), depois da era dos videojogos simbólicos (1971 a 1984, onde os limites tecnológicos do *hardware* prevaleciam sobre a elaboração sugestiva dos neo-artistas), da era dos videojogos clássicos (1984 – 1993), em que se apresentava um equilíbrio entre a capacidade tecnológica e a elaboração sugestiva e da era dos videojogos românticos (1993 - ...), assiste-se agora ao surgimento de uma nova geração de jogos multimídia, a qual implica o desenvolvimento de novas dimensões de análise.

Estes novos tipos de jogos, tendem a afirmar-se como algo diferenciado dos modelos anteriores, que pode-se designar como jogos de realidade complexa, ou seja, fruto da aposta na construção de temas e *scripts* fortes, os quais buscam inspiração nas características da produção cinematográfica.

De acordo com Watt (2000), o mercado dos jogos eletrônicos é um mercado em grande ascendência. As possibilidades para os caminhos da evolução dos jogos e do próprio *hardware* são infinitos. Muitas empresas estão explorando novos caminhos para o uso da tecnologia 3D em aplicações que não somente jogos, mas também no comércio pela *web* e em *plugins* para aplicações comerciais, entre outros.

Segundo Lamothe (1999), ao implementar um jogo parte-se de um esqueleto que é utilizado pela maioria dos desenvolvedores.

Lamothe (1999) subdivide a execução de um jogo em 8 seções:

a) **Inicialização:** aqui se deve alocar memória, alocar recursos, carregar dados de arquivos, assim como a inicialização de um programa normal;
b) **Entrar no *Game loop*:** início do *loop*, todas as ações são controladas aqui até a saída do programa;
c) **Ler entrada do jogador:** os comandos executados pelo jogador são verificados para utilização nas seções de Inteligência Artificial e lógica;

d) **Executar algoritmos de inteligência artificial e lógica do jogo:** seção que detém a maior parte do código. A Inteligência Artificial, a Física e a lógica do jogo são executadas nessa parte. Os resultados obtidos são utilizados para se produzir o próximo quadro do jogo;
e) **Criar o próximo quadro do jogo:** os resultados obtidos nos itens “c” e “d” são graficamente criados, para serem exibidos na tela do computador;
f) **Sincronizar a Imagem:** devido à diferença de velocidade entre as máquinas e a complexidade de cada jogo, o objetivo dessa parte é exatamente a sincronização do jogo para que ele seja executado respeitando uma taxa de atualização. Um valor recomendado é 30 quadros por segundo;
g) **Final do Game Loop:** volta-se ao item “b”;
h) **Terminar o aplicativo:** quando o usuário deseja parar de jogar, deve-se liberar todos os recursos utilizados, e voltar ao sistema operacional.

Informações adicionais sobre jogos podem ser encontrados em Beltrão (2000).

Segundo Sully (1993), um espaço tridimensional é algo fácil de perceber porém, algo difícil para desenhar e projetar. Pelo motivo de um ambiente 3D ser muito complexo, é freqüentemente útil olhar para este ambiente de dois modos diferentes – uma visualização do topo e outra de frente ou de lado. Embora alguns projetistas possam captar e projetar mentalmente em três dimensões ao mesmo tempo, muitos optam em simplificar este espaço em um modo, focalizando o espaço em apenas duas dimensões em um determinado tempo.

Quando analisa-se um ambiente em três dimensões, estas dimensões referem-se às suas coordenadas geométricas. Este é um sistema para mensuração de um espaço em 3D pela utilização de três linhas imaginárias, ou *axis* (eixos). Pode-se imaginar estas três linhas da seguinte maneira: uma percorrendo de norte a sul, outra percorrendo de leste a oeste e a terceira percorrendo verticalmente através da interseção das duas primeiras linhas.

Mais informações sobre ambientes em três dimensões como um sistema de mensuração de um espaço, podem ser encontradas em Raitz (2001) e Silva (2000).

Existem diferentes conotações de "espaços em 3D" quando fala-se em gráficos 3D:

a) *Object space* (***3D space***): este sistema de coordenadas é local para um objeto, constituído por um conjunto de polígonos. Para ter várias instâncias de um mesmo objeto em diferentes locais no ambiente, é necessário o espaço de objeto (*object space*). O espaço de objeto também é o espaço dentro do qual são modelados os objetos;
b) *World space* (***3D space***): segundo Tremblay (1999?) este sistema de coordenadas é o mais importante. É neste espaço onde todos os objetos são posicionados, onde se faz a computação física, movimentos e detecção de colisão. Também é neste espaço que a iluminação é computada. Deve-se pensar no espaço do mundo (*space world*) como o ambiente do jogo;
c) *View space* (***3D space***): este sistema de coordenadas é relativo à câmara. Objetos no *world space* são transformados para o *view space* para saber o que será visível na tela. Este espaço também é chamado de espaço do olho (*eye space*) ou espaço da câmara (*camera space*);
d) *Screen space* (***2D space***): O espaço da tela (*screen space*) é de fato a representação homogênea das coordenadas na tela. Coordenadas não são *pixels*. O *viewport* projeta a *view space* para a *screen space*, e a origem na *screen space* é no centro da tela (para projeções em perspectiva).

Os espaços são representados utilizando-se a notação de matrizes. Em uma matriz 3x3 têm-se três vetores. Cada vetor define a direção de um eixo. O primeiro vetor é "direita" (eixo X positivo), o segundo vetor é o eixo Y positivo e o terceiro é o eixo Z positivo. Todos os vetores são ortogonais em unidades de tamanho.

Segundo O'Neill (1998), todos os dias da vida tende-se a observar elementos em elevação muito mais do que elementos planos (rentes ao solo). Ao olhar fixamente para o fim de um compartimento, encontrar-se-á na maioria das vezes, dependendo do local onde nos encontramos, mais de 80% da área principal no centro do campo de visão é tomado por superfícies verticais, enquanto os planos de solo e teto estão relegados às extremidades superior e inferior do campo de visão. Os elementos verticais disputam uma grande parte na determinação das características de um ambiente.

Os principais elementos de um ambiente segundo O'Neill (1998) são:

a) paredes;
b) aberturas em paredes;
c) portais (passagens de um ambiente ao outro);
d) janelas;
e) solo e teto do ambiente;
f) topografia e declives;
g) escadarias;
h) cantos (esquinas);
i) objetos;
j) paisagens;
k) iluminação e sombra.

Os elementos citados e diversos outros podem ser construídos com *engines*. Segundo Eberly (2001), para a produção e gerenciamento de imagens realísticas nos jogos, são utilizadas as *engines*. *Engines* 3D são *Application Programmer Interfaces* (API´s) para visualização 3D em tempo real. Uma *engine* para jogos precisa tratar da questão do gerenciamento dos gráficos de cena que forneça uma eficiente renderização, seja um *software* ou um *hardware* renderizador. A *engine* também precisa ter a capacidade de processar objetos complexos e o movimento dos objetos de forma realista. A *engine* precisa dar suporte à detecção de colisão, superfícies curvas bem como modelos poligonais, animação, plano geométrico dos detalhes, geração de terrenos e distribuição de espaços. Além disso, a *engine* precisa ser suficientemente abrangente, para dar suporte aos princípios da orientação à objetos.

Conforme Crystal Space (2000?a), a *engine* Crystal Space (CS) é um "*kit*" de desenvolvimento para jogos 3D com código aberto, escrita na linguagem de programação C++ e está baseada na licença de uso LGPL. O Crystal Space pode ser utilizado para várias tarefas de visualização em 3D. Ela implementa todos os itens citados acima e é multiplataforma. Suporta as bibliotecas gráficas OpenGL, Glide e DirectX. Levando-se em

consideração que o CS possui várias classes para componentes 2D e 3D que correspondem à API's proprietárias, optou-se pela utilização do DirectX para a renderização[1] do ambiente.

Segundo LaMothe (2001?), o DirectX foi criado para solucionar um conjunto de problemas que vem crescendo a cada dia, onde os fabricantes de *software* surgem em todo lugar, criando novas tecnologias como aceleradores 2D/3D, avançados sintetizadores de som e complicados dispositivos de entrada de dados.

A API DirectX, tecnicamente é um conjunto de funções da *Component Object Model* (COM) que implementam um número de *interfaces* que habilitam a comunicação com a placa de vídeo, placa de som, placa de rede, dispositivos de entrada de dados e algumas utilidades do sistema operacional. Porém, mais do que isso, abre um caminho ao que os programadores tanto desejam, que é a programação de jogos.

Tendo em vista os aspectos citados, procurou-se desenvolver um protótipo de um ambiente 3D para jogos, com a inserção de objetos e efeitos simples neste ambiente e visualização destes em primeira pessoa.

1.1 OBJETIVOS DO TRABALHO

O objetivo principal do trabalho é implementar um protótipo de um ambiente 3D utilizando a *engine* Crystal Space e DirectX.

Os objetivos específicos do trabalho são:

a) criar um ambiente em 3D, com objetos simples e a visualização destes em primeira pessoa;
b) implementar neste ambiente 3D, iluminação colorida, objetos 3D, 6DOF, transparência (*alpha*) e espelhamento.

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado conforme descrito abaixo.

---

[1] A conversão da descrição de um objeto-base de alto nível, dentro de uma imagem gráfica para exibição

O capítulo dois apresenta as principais características da *engine* Crystal Space, apresentando o histórico, evolução, conceitos, funções e restrições.

O capítulo três destina-se à apresentação do DirectX.

O capítulo quatro apresenta os requisitos, a especificação, implementação e o funcionamento do protótipo do ambiente em 3D, demonstrando as técnicas dos “efeitos” aplicados no ambiente pelo Crystal Space.

Como finalização, o capítulo cinco é utilizado para a apresentação das conclusões gerais, originadas durante o estudo e confecção do trabalho e apresentação de algumas possíveis extensões para futuros trabalhos correlatos.

2 CRYSTAL SPACE

Conforme Crystal Space (2000?a), o Crystal Space (CS) é uma *engine*[2] com seis graus de liberdade de visão (*six-degrees-of-liberty* - 6DOF) com código fonte aberto, que se enquadra na licença LGPL. Pode ser descrita como um pacote de componentes e bibliotecas que podem ser utilizadas na criação de jogos. O CS está em constante evolução, e ainda há muito o que desenvolver. Qualquer pessoa que tenha interesse, pode contribuir com o código fonte. A seguir serão descritos conceitos relacionados ao CS obtidos de Crystal Space (2000?a).

O CS é livre e deve permanecer livre, pois é licenciada à *Library General Public License* (LGPL), publicada pela *Free Software Foundation* (FSF), atualmente na versão 2. Como o código da biblioteca é livre (aberto), não há garantias quanto ao funcionamento da mesma.

O projeto Crystal Space foi iniciado por Jorrit Tyberghein, que escreveu a primeira versão do CS durante o tempo livre que tinha. O interesse foi despertado após a leitura de um artigo sobre portais. O projeto "decolou" quando Jorrit resolveu inscrever o projeto à LGPL e anunciou-o em *comp.graphics.algorithms*. Atualmente, já existem vários projetos utilizando o Crystal Space como API de desenvolvimento. Entre eles, alguns dos principais são:

a) VOS: *Virtual Object System – internet virtual reality*: é um sistema de objetos distribuídos. Uma infra-estrutura para comunicação em rede orientada à objetos que pode ser utilizada para construir componentes (clientes, servidores, nós ponto-a-ponto) para aplicações de rede. A meta deste projeto é utilizar esta infra-estrutura para a construção de ambientes 3D multiusuário colaborativos;
b) PLANESHIFT: o objetivo do Planeshift é criar um mundo virtual de fantasia. É um jogo *Role-Playing Game* (RPG) *multiplayer* para jogar na internet. Pode ser acessado no endereço <http://www.planeshift.it>, (figura 1);
c) TUNNEL FIGHTER: é um pequeno jogo totalmente feito com o CS, podendo ser jogado em 1ª ou 3ª pessoa.

---

[2] *Engines* 3D são *Application Programmer Interfaces* (API´s) para visualização 3D em tempo real.

FIGURA 1 – AMBIENTE CONSTRUÍDO COM O CS

O CS está dividido em várias bibliotecas de funções, sendo que algumas destas bibliotecas estão descritas abaixo:

a) *System Driver*: uma biblioteca de sistema que efetua as operações de dependência do sistema para o programa. Com isto, o programa é facilmente portável para qualquer outro sistema operacional suportado;
b) *Geometry Library*: uma biblioteca de matemática 3D, incluindo vetores 2D/3D, matrizes e polígonos. Esta biblioteca pode ser utilizada fora do Crystal Space;
c) *2D Canvas / 3D Renderer*: esta biblioteca habilita o desenho diretamente na tela e permite utilizar um *hardware* acelerador 3D. Esta biblioteca também é independente do Crystal Space;
d) *3D engine*: biblioteca com os seguintes recursos:
   - 6DOF – Seis graus de liberdade de visão;
   - *engine* de paisagem integrada;
   - movimentação de objetos e o controle dos movimentos;
   - iluminação colorida estática com sombras reais;
   - iluminação colorida dinâmica com suporte à sombras (detalhes das sombras são ignorados);

- portais são utilizados para uma fácil e eficiente visibilidade. Pode-se habilitar uma OCTREE ou BSP para alguns setores, para permitir uma definição do "mundo" cada vez mais detalhada. Com portais também cria-se espelhos transparentes e semitransparentes;
- suporte à múltiplas câmaras e capacidade de fazer câmaras em primeira e terceira pessoa;

e) *Utility Library*: biblioteca de utilitários como leitura de arquivos de configuração (.INI) para opções do programa, leitura e escrita de arquivos (.ZIP) como arquivos de dados, entre outros;
f) VFS (*Virtual File System*): sistema de arquivos virtual;
g) *Tool Library*: biblioteca de ferramentas. Contém várias funções que auxiliam na inicialização de uma aplicação. A classe mais importante desta biblioteca é `csInitializer` que cria tudo o que for necessário para tornar uma aplicação CS operacional;
h) *Sound Driver / Sound Renderer*: biblioteca para o sistema de som em geral;
i) *Network Driver*: biblioteca para suporte à rede. Atualmente suporta somente TCP/IP.

Para executar (compilar) o CS com sucesso, é necessário, além da utilização de um ambiente de desenvolvimento – o Microsoft Visual C++ é o recomendado – a utilização do DirectX *Software Development Kit* (SDK). A figura 2 apresenta a *engine* 3D como intermediária entre a aplicação e o *hardware* gráfico.

FIGURA 2 – *ENGINE* 3D, INTERMEDIAÇÃO ENTRE A APLICAÇÃO E O *HARDWARE*

2.1 UMA APLICAÇÃO CRYSTAL SPACE

Para escrever uma aplicação utilizando o Crystal Space, há vários conceitos que precisam ser explanados. Esta seção descreve um resumo dos principais conceitos que precisam ser conhecidos ao implementar uma aplicação para o Crystal Space.

2.1.1 O SISTEMA SCF

O sistema *Shared Class Facility* (SCF) é o núcleo da área de trabalho de memória de baixo nível para o CS. É necessário a inicialização do SCF antes de efetuar qualquer outra coisa (uma maneira é utilizar `csInitializer::InitializeSCF()`). A SCF faz parte da biblioteca `csUtil`. A SCF é algo parecido com a COM[3]. A principal diferença entre a COM e a SCF é devida a crescente necessidade de uma *interface* fácil de usar e talvez mais amigável do que a COM.

O principal paradigma da SCF é a *interface*. Define-se uma *interface*, isto é, um conjunto de métodos abstratos que serão acessados dentro de um objeto. As *interfaces* SCF são simples estruturas em C++ (*C++ structs*) ou classes (não há muita importância, exceto se estiverem sendo utilizadas classes, é necessário adicionar a palavra chave `'public:'`).

Para a utilização da SCF, é necessário apenas o arquivo *include* `'scf.h'`. Esse arquivo contém um número de macros e funções que serão necessárias para uma fácil utilização da SCF. Também há a necessidade de adicionar os arquivos `'scfimp.cpp'` e `'scf.cpp'`. Algumas funcionalidades básicas da SCF são fornecidas por um objeto central. Este objeto pode ser acessado como `'iSCF::SCF'` e é do tipo '`iSCF*'` (que também é uma *interface* válida).

Para simplificar as coisas, todas as classes exportadas são derivadas de uma *interface* básica chamada `'iBase'`. Atualmente, '`iBase`' define três funções de *interface*:

a) `void IncRef();`: esta função deve ser chamada toda vez que faz-se uma nova referência para um objeto, que armazena esta referência para um longo tempo de uso;

---

[3] Arquitetura de software livre da DEC e Microsoft, permitindo a interoperação entre ObjectBroker e OLE.

b) `void DecRef();`: função que deve ser chamada para decrementar a contagem de referências a objetos. Quando a contagem chegar a zero, o objeto é automaticamente excluído, desde que não seja uma *interface* embutida dentro de outro objeto;

c) `iBase* QueryInterface(scfInterfaceID iInterfaceID, int iVersion);`: este método retorna um ponteiro para uma *interface* ou para uma *interface* embutida. A `InterfaceID` é sinônimo para o nome da *interface*. A ID da *interface* pode ser obtida utilizando `'iSCF::SCF->GetInterfaceID (nome)'`.

Para simplificar, o arquivo `'scf.h'` disponibiliza várias macros que fornecem declarações e implementações dos três métodos citados acima. A macro `'SCF_DECLARE_IBASE'` declara estes métodos dentro de qualquer definição de classe que derive de `'iBase'`. A macro `'SCF_IMPLEMENT_IBASE'` adiciona as implementações destas funções no módulo, conforme quadro 1.

QUADRO 1 – MACROS DO ARQUIVO 'SCF.H'

```
Struct iTest : public iBase
{
  ---
};

class Test : public iTest
{
  SCF_DECLARE_IBASE;
};
SCF_IMPLEMENT_IBASE (Test)
```

2.1.2 CONTADOR DE REFERÊNCIAS (*REFERENCE COUNTER*)

O contador de referência é uma parte muito importante da estrutura do CS. Sempre deve-se ter certeza de que o gerenciamento das referências no código estão corretas. Faltando apenas um `DecRef()`, pode causar um grande vazamento de memória.

2.1.3 REGISTRO DE OBJETOS (*THE OBJECT REGISTRY*)

O registro de objeto é o repositório central para todos os objetos no CS. Tudo no Crystal Space utiliza o registro de objeto para obter o ponteiro para os objetos que se tem

interesse. Uma das primeiras tarefas que uma aplicação CS deveria fazer, é criar o registro de objeto. Um registro de objeto pode ser criado utilizando `csInitializer::CreateObjectRegistry()` ou `csInitializer::CreateEnvironment()`.

2.1.4 GERENCIADOR DE *PLUGIN* (*THE PLUGIN MANAGER*)

O gerenciador de *plugin* é responsável por carregar e descarregar *plugins*. Um *plugin* é uma biblioteca compartilhada de funções que podem ser carregadas dinamicamente dentro de uma aplicação CS em execução. No sistema operacional Unix, um *plugin* possui a extensão `'.so'` e no Windows, a extensão `'.dll'`. Quase tudo o que constitui a estrutura do CS são *plugins*. Portando, o gerenciador de *plugin* é uma parte importante. O gerenciador de *plugin* pode ser criado com `csInitializer::CreatePluginManager()` ou `csInitializer::CreateEnvironment()`.

2.1.5 FILA DE EVENTO (*THE EVENT QUEUE*)

O Crystal Space é um sistema dirigido a eventos. Portanto, uma aplicação CS também será "orientada e eventos". A fila de evento gerencia os eventos do sistema e envia-os para registradores. Todo módulo ou *plugin* que implementa `iEventHandler` pode se auto-registrar com a fila de evento. Assim, será notificado quando ocorrerem certos eventos. A fila de eventos pode ser criada com `csInitializer::CreateEventQueue()` ou `csInitializer::CreateEnvironment()`.

Uma aplicação CS também deverá criar um tratador de eventos próprio para ser capaz de fazer algo. Este tratador de eventos será responsável realmente pela renderização na tela (isto deve ser feito em resposta ao evento `cscmdProcess`) e também capturar a entrada do teclado e do *mouse*. Para inicializar o manuseador de eventos para a aplicação pode ser utilizado `csInitializer::SetupEventHandler()`.

2.1.6 O RELÓGIO VIRTUAL (*THE VIRTUAL CLOCK*)

Muitas aplicações Crystal Space são baseadas em tempo. O relógio virtual suporta a noção de TEMPO CORRENTE e TEMPO PASSADO. Ele é chamado de relógio virtual

porque não necessariamente corresponde com a hora real. Um jogo pode ser colocado em pausa, por exemplo. Isto é especialmente importante para cálculos de física e também para velocidade de movimentação dos objetos. O relógio virtual pode ser criado com `csInitializer::CreateVirtualClock()` ou `csInitializer::CreateEnvironment()`.

2.1.7 ANÁLISE DE LINHA DE COMANDO (*THE COMMANDLINE PARSER*)

O analisador de linha de comando (*commandline parser*) é responsável por analisar opções de uma linha de comando. Vários *plugins* e módulos do CS também solicitarão a linha de comando, pois isto é importante para ter este objeto corretamente inicializado. O *Commandline Parser* pode ser criado com `csInitializer::CreateCommandLineParser()` ou `csInitializer::CreateEnvironment()`. O *commandline parser* pode ser inicializado com `csInitializer::SetupCommandLineParser()`.

2.1.8 O GERENCIADOR DE CONFIGURAÇÃO (*THE CONFIG MANAGER*)

O gerenciador de configuração é responsável pela leitura de opções de um arquivo de configuração. Todos os arquivos de configuração estão inseridos em um repositório global que é gerenciado pelo gerenciador de configuração. Vários *plugins* e módulos do CS solicitarão opções do gerenciador de configuração e uma aplicação CS também pode fazer isto se quiser. O gerenciador de configuração pode ser criado com `csInitializer::CreateConfigManager()` ou `csInitializer::CreateEnvironment()`. Ele pode ser inicializado com `csInitializer::SetupConfigManager()`.

2.1.9 *DRIVERS* DE ENTRADA (*THE INPUT DRIVERS*)

Existem três *drivers* de entrada padrão inclusos com o CS: teclado, *mouse* e *joystick*. Chamando-se `csInitializer::CreateInputDrivers()` ou

`csInitializer::CreateEnvironment()` os *drivers* serão criados e editados através do registro de objetos. Estes *drivers* afixarão eventos na fila de eventos global.

2.1.10 A CLASSE CSINITIALIZER

Já foram mencionadas várias funções da classe `csInitializer` anteriormente. Esta classe (que pela definição pode ser encontrada em `'include\csTool\initapp.h'`) pode ser utilizada para auxílio na inicialização de tudo o que for necessário para fazer a aplicação rodar. Esta classe contém apenas funções membro estáticas.

2.2 CRIANDO UMA APLICAÇÃO

Para uma aplicação ser considerada propriamente "Aplicação Crystal Space", há alguns requisitos para isto, que são:

a) todo e qualquer arquivo fonte (`'.cpp'`) deve importar `'cssysdef.h'` como o primeiro arquivo de cabeçalho do CS a ser incluso. `'cssysdef.h'` nunca deverá ser incluído por um arquivo de cabeçalho;
b) `CS_IMPLEMENT_APPLICATION` precisa ser inserido no início do arquivo (após os `'include'`;
c) A função `main()` é obrigada a ser declarada com o seguinte protótipo: `int main(int argc, char *argv[])`.

2.2.1 UM ARQUIVO SIMPLES

Pode-se "adotar" uma boa prática de programação, que é sempre colocar as definições e declarações em arquivos de cabeçalho. Em alguns casos isto até é obrigatório. Será apresentado a seguir, um arquivo de cabeçalho (*header file*) para uma aplicação CS simples. Embora isto não seja restritamente necessário, utiliza-se uma classe para encapsular a lógica da aplicação. O arquivo de cabeçalho `'simple.h'` pode ser feito conforme quadro 2.

QUADRO 2 – CABEÇALHO DO ARQUIVO ‘SIMPLE.H’

```
#ifndef __SIMPLE_H__
#define __SIMPLE_H__

#include <stdarg.h>

struct iEngine;
struct iLoader;
struct iGraphics3D;
struct iKeyboardDriver;
struct iSector;
struct iView;
struct iVirtualClock;
struct iObjectRegistry;
struct iEvent;

class Simple
{
private:
     iObjectRegistry* object_reg;
     iEngine* engine;
     iLoader* loader;
     iGraphics3D* g3d;
     iKeyboardDriver* kbd;
     iVirtualKlock* vc;

public:
     simple();
     ~simple();

     bool Initialize(int argc, const char* const argv[]);
     void Start();
};

endif // __SIMPLE_H__
```

A classe `Simple` mantêm um número de referências à objetos importantes que serão muito utilizados. Com isto não existe a necessidade de obter estes objetos cada vez que for preciso. Após isto, têm-se um construtor que fará a inicialização destas variáveis, um destrutor que irá “limpar” a aplicação, uma função de inicialização que será responsável por iniciar todo o CS e a aplicação e por último, a função `Start()` para começar o manuseador de eventos. No Anexo A, encontra-se o arquivo fonte `'simple.cpp'`.

Esta é quase a aplicação mais simples possível e absolutamente “inútil”. Também não deve-se executar esta aplicação em um sistema operacional que não permita “matar” a aplicação que está rodando, porque não há outra maneira de parar a aplicação, uma vez iniciada. Embora que esta aplicação seja inútil no momento, ela já possui algumas funções

que serão muito úteis posteriormente. Segue abaixo um pequeno resumo das funcionalidades que a aplicação já possui:

a) abrir uma janela;
b) pode-se controlar o tamanho da janela e o *driver* de vídeo para esta janela com as opções de linha de comando (`-video` e `-mode`).
c) será apresentado uma pequena ajuda (help) quando utilizado `-help` na linha de comando;
d) possui os seguintes *plugins* inicializados e prontos para usar: *engine*, *3D renderer*, *canvas*, *reporter*, *reporter listener*, *font server*, *image loader*, *level loader* e VFS.

Após a inclusão dos arquivos de cabeçalho necessários, primeiramente é necessário utilizar algumas "macros". A macro `CS_IMPLEMENT_APPLICATION` é essencial para qualquer aplicação que utilize o CS. Isto faz com que a rotina `main()` seja corretamente "*linkada*" e chamada de qualquer plataforma.

A rotina `main()` primeiramente cria uma instância da classe `'simple'`. A instância desta classe é inserida dentro de uma variável global para tornar o acesso a esta classe mais fácil. O próximo passo é a inicialização. A primeira tarefa que a função `Initialize()` faz é criar o ambiente com `csInitializer::CreateEnvironment()`. Isto irá inicializar a SCF, criar o registro de objeto, e então criar um número de outras entidades úteis (*plugin manager, event queue*, ...). A função `csReport()` é, por conveniência, uma função para enviar mensagens (normalmente um erro ou notificação). O seu funcionamento é muito parecido com o da função '`printf()`', exceto pelo fato de ser necessário passar o nível exato e um identificador que pode dar uma idéia da origem da mensagem.

Após isto é chamada a função `csInitializer::SetupCommandLineParser()` para iniciar o analisador de linha de comando. Isto permite com que o restante da inicialização passe a ler instruções da linha de comando. A função `csInitializer::RequestPlugins()` utilizará o arquivo de configuração (que não é utilizado no exemplo anterior), a linha de comando e os *plugins* requeridos para encontrar quais *plugins* serão carregados. A linha de comando é de mais alta prioridade, seguido pelo arquivo de configuração e por último, os *plugins* requeridos.

A função csCommandLineHelper::CheckHelp() verifica se foi digitado -help na linha de comando e apresenta uma "ajuda" para todos os *plugins* carregados. Após isto, é solicitado ao registro de objetos para encontrar todos os objetos comuns que serão necessários posteriormente e armazenar uma referência a estes objetos na classe principal. No destrutor do objeto, é necessário liberar estas referências com DecRef().

Por último, quando efetuou-se toda a inicialização, é aberta uma janela com uma chamada à função csInitializer::OpenApplication(). Isto envia a mensagem cscmdSystemOpen para todos os componentes que estão listados para a fila de eventos. Um dos *plugins* que faz isto é o renderizador 3D (*3D renderer*) que então abrirá sua janela (ou habilitará os gráficos em um sistema operacional *non-windowing*). Isto conclui a passagem da inicialização.

Na função Simple::Start() inicia-se o laço padrão (*loop*) principal chamando-se csDefaultRunLoop(). Esta função somente retornará quando a aplicação finalizar (que o exemplo anterior ainda não faz). Basicamente, esta função iniciará o *loop* para os tratadores de eventos.

2.2.2 TRATAMENTO DE EVENTOS (*EVENT HANDLING)*

Para fazer desta simples aplicação algo mais fácil de testar, será adicionado um meio de finalizar a aplicação com a tecla ESC. Devem ser adicionados os métodos privados da classe 'simple' no arquivo 'simple.h', conforme quadro 3.

QUADRO 3 – MÉTODOS PRIVADOS DA CLASSE 'SIMPLE'

```
Static bool SimpleEventHandler (iEvent& ev);
Bool HandleEvent (iEvent& ev);
```

A função estática SimpleEventHandler() será registrada para a fila de eventos. É necessário que esta função seja uma função estática porque a fila de eventos espera chamar uma função normal no estilo C (isto também poderia ser implementado, fazendo uma subclasse de iEventHandler, mas isto seria mais complicado e não realmente necessário). A função HandleEvent() é quem faz o tratamento de eventos real. A função SimpleEventHandler() é utilizada apenas como uma sub-função para chamar HandleEvent().

Após isto, deve-se adicionar o código do quadro 4 antes de `Simple::Initialize()`.

QUADRO 4 – CÓDIGO PARA FINALIZAR COM A TECLA ‘ESC’

```
Bool Simple::HandleEvent(iEvent& ev)
{
     if (ev.Type == csevKeyDown && ev.Key.Code == CSKEY_ESC)
     {
          iEventQueue* q = CS_QUERY_REGISTRY(object_reg, iEventQueue);
          if (q)
          {
               q->GetEventOutlet()->Broadcast(cscmdQuit);
               q->DecRef();
          }
          return true;
     }
     return false;
}

bool Simple::SimpleEventHandler(iEvent& ev)
{
     return simple->HandleEvent(ev);
}
```

A função `HandleEvent()` verifica se a tecla `ESC` foi pressionada. Em caso afirmativo, será utilizado o registro de objetos para encontrar o objeto na fila de evento global. Utilizando-se `Broadcast` será transmitida a mensagem `cscmdQuit` para todas as “partes interessadas”. Isto causará a saída da aplicação pela finalização do *loop*. Também é necessário inicializar esta função tratadora de evento (*handle event*). Para fazer isto, adiciona-se o código do quadro 5 após a chamada à `csInitializer::RequestPlugins()`.

QUADRO 5 – INICIALIZAÇÃO DA FUNÇÃO TRATADORA DE EVENTO (*HANDLE EVENT*)

```
If (!csInitializer::SetupEventHandler(object_reg, SimpleEventHandler))
{
     csReport(object_reg, CS_REPORTER_SEVERITY_ERROR,
          "crystalspace.application.simple",
          "Can't initialize event handler !");
     return false;
}
```

2.2.3 CRIANDO UM “MUNDO”

Com o código adicionado até o momento, tem-se uma aplicação interessante que abre uma janela negra e aguarda pelo pressionamento da tecla `ESC`. Nesta seção, será criado um compartimento para visualização na janela. Adiciona-se um gerenciador de texturas, um compartimento (tecnicamente chamado de setor) e algumas luzes. Primeiramente, adiciona-se

um ponteiro para o setor principal da classe ´simple´ no arquivo de cabeçalho, conforme quadro 6.

QUADRO 6 – ADIÇÃO DE UM PONTEIRO

```
...
struct iSector;
...
class Simple
{
private:
     ...
     iSector* room;
     ...
```

Agora, adiciona-se os pedaços de código (gerenciador de texturas, compartimento, luzes) ao arquivo 'simple.cpp', conforme Anexo B.

Este código extra, primeiramente carrega uma textura com LoadTexture(). O primeiro parâmetro é o nome da textura, o segundo é como ela será conhecida na *engine* e o terceiro é o nome do arquivo atual no volume do VFS. Caso não exista a textura 'stone4.gif', pode ser utilizado qualquer outro arquivo '.gif'. O único requisito é que a textura precisa ter tamanhos que sejam potências de 2 (ex.: 64 x 64) (o CS fará a escala automaticamente caso este requisito não seja atendido, porém isto poderá reduzir a qualidade). Esta função retorna um objeto 'iTextureWrapper' que não é utilizado. Em seu lugar, é utilizado 'iMaterialWrapper' que é criado automaticamente com o método LoadTexture().

O compartimento é criado com o método CreateSector(). Este compartimento encontra-se vazio inicialmente. Um compartimento no Crystal Space é representado por 'iSector' que é basicamente um recipiente que pode manter objetos geométricos. Os objetos no Crystal Space são representados por objetos de malha (*MESH OBJECTS*) da classe 'iMeshObject'. Há vários tipos de objetos de malha no CS. Cada tipo de um objeto de malha representa um modo diferente de representar a geometria.

Após a criação do compartimento, são criadas seis paredes para o compartimento. Primeiramente cria-se um objeto de malha com 'csThing::MeshObject'. A criação de um objeto de malha com 'csThing::MeshObject' é um caso muito comum e por este motivo, há uma função na *engine*, chamada 'CreateSectorWallsMesh()' que cria uma

certa malha e a adiciona ao setor. Após isto, se faz necessário adicionar polígonos para esta malha. Para fazer isto, verifica-se a *interface* chamada iThingState para acessar os parâmetros internos do objeto de malha da classe 'csThing'. Utiliza-se a macro SCF_QUERY_INTERFACE que é parte da SCF. Isto verificará se o objeto de malha (*mesh object*) (que está encapsulado pelo empacotador de malha (*mesh wrapper*)) atualmente implementa iThingState e em caso positivo, retornará um ponteiro para a implementação de iThingState. Todos os objetos de malha implementam algum tipo de estado da *interface* que é utilizado para inicializar ou verificar o estado deste objeto. Todas as *interfaces* que são examinadas utilizando-se SCF_QUERY_INTERFACE deverão ser liberadas, chamando DecRef().

Com o retorno do estado de 'csthing', pode-se criar polígonos com uma chamada a CreatePolygon() para cada parede. Isto retornará um ponteiro para um polígono (iPolygon3D). Com isto, pode-se inicializar vários parâmetros como por exemplo, o material. Então são adicionados quatro vértices (no Crystal Space um polígono é visível se os seus vértices são orientados no sentido horário). A localização dada para CreateVertex() está em *object space* (em contraste com a área do mundo e a área da câmara). Para definir como a textura será projetada sobre o polígono é utilizado o método SetTextureSpace(). Existem várias versões para esta função porém, no código do anexo B é utilizada uma das versões mais simples, mas que oferece o menor controle. Neste caso particular pega-se os dois primeiros vértices do polígono para os eixos-u (*u-axis)* da textura. Os eixos-v (*v-axis*) serão calculados perpendicularmente aos eixos-u. O terceiro parâmetro indica que a textura será escalonada de modo que um ladrilho de textura seja exatamente uma unidade 3x3 do mundo, em tamanho.

Por último, cria-se algumas luzes no compartimento para certificar-se de que será possível ver as paredes. A *interface* 'iStatLight' representa uma iluminação estática que não pode ser movida ou alterada de intensidade. No exemplo (anexo B), são criadas três luzes semelhantes que são adicionadas ao compartimento utilizando o método AddLight(). Note-se que a especificação de luzes em um setor é apresentada por um objeto implementando iLightList. Para obter esta especificação é efetuada uma chamada à iSector::GetLights(). Verifica-se também que esta especificação de luz trabalha com

luzes do tipo `iLight`, que é a *interface* base para todas as luzes (iluminação) no Crystal Space. Para obter o `iLight` de uma instância de `iStatLight`, pode-se utilizar `iStatLight::QueryLight()`.

Quando cria-se uma iluminação são utilizados vários parâmetros. Primeiramente, têm-se o nome da iluminação. Isto não é utilizado freqüentemente e geralmente pode-se inicializar com `NULL`. O segundo parâmetro é a localização da iluminação no ambiente. Em seguida têm-se o raio. A luz não afetará os polígonos que estão fora da esfera definida pelo centro da luz e do raio. O parâmetro seguinte é a cor da iluminação no formato RGB (<1,1,1> significa o branco puro e <0,0,0> significa o preto puro). O último parâmetro indica onde deseja-se ter uma iluminação pseudo-dinâmica.

A chamada à `Prepare()`, prepara a *engine* para a renderização da cena. Serão apresentadas todas as texturas e criados todos os mapas de luzes (*lightmaps*) se necessário. Somente após esta chamada pode-se iniciar a renderização do ambiente, pois os mapas de luzes precisam ser convertidos para um formato mais adequado para o renderizador 3D escolhido.

A última parte do código adicionado aloca a paleta com o gerenciador de texturas. Neste ponto, cria-se e inicializa-se o compartimento. Ao compilar e executar esta aplicação, somente será visto uma janela preta, pois ainda não criou-se uma câmara através da qual poder-se-á ver o ambiente.

2.2.4 A CÂMARA

No Crystal Space, a câmara é uma *interface* chamada `iView` que encapsula as instâncias de `'iCamera'` e `'iClipper2D'`. A princípio pode-se utilizar estas classes diretamente, porém utilizar `iView` é mais fácil. Edita-se o arquivo `'simple.h'` para fazer uso da classe `iView`, conforme quadro 7.

QUADRO 7 – INSTANCIAÇÃO DA CLASSE 'IVIEW'.

```
Struct iView;
...
class Simple
{
private:
  ...
  iView* view;
  ...
  void SetupFrame ();
  void FinishFrame ();
  ...
```

Após isto, edita-se o arquivo `'simple.cpp'` efetuando alterações no construtor e no destrutor de `'simple'`, conforme quadro 8.

QUADRO 8 – CLASSE 'IVIEW' NO CONSTRUTOR/DESTRUTOR

```
Simple::Simple ()
{
  view = NULL;
  ...
}

Simple::~Simple ()
{
  if (view) view->DecRef ();
  ...
}
```

Ao final da função `Initialize()` (antes de `txtmgr->SetPalette()`), adiciona-se o código fonte do quadro 9.

QUADRO 9 – CÓDIGO FONTE PARA ADIÇÃO DA CÂMARA

```
bool Simple::Initialize (int argc, const char* const argv[])
{
...
  view = new csView (engine, g3d);
  view->GetCamera ()->SetSector (room);
  view->GetCamera ()->GetTransform ().SetOrigin (csVector3 (0, 5, -3));
  iGraphics2D* g2d = g3d->GetDriver2D ();
  view->SetRectangle (0, 0, g2d->GetWidth (), g2d->GetHeight ());

  txtmgr->SetPalette ();
  return true;
}
```

Conforme código apresentado, primeiramente cria-se uma vista para o ambiente e o renderizador gráfico 3D. A `view` tem o setor atual, que é passado para a câmara e

inicializado com o método `SetSector()`. A câmara também possui uma posição neste setor que pode ser inicializada da seguinte maneira: primeiramente obtêm-se a câmara com `GetCamera()` e então inicializa-se a posição (que é um `'csVector3'`) com `SetPosition()`. A *view* também mantém uma região de corte (*clipping region*) que corresponde a área da janela que está sendo utilizada para desenhar o ambiente. O CS suporta polígonos convexos para serem utilizados como áreas de visualização, mas no exemplo é utilizado um simples retângulo. A inicialização deste retângulo é feita com o método `SetRectangle()`.

Com todo o código apresentado até o momento, têm-se uma câmara mas ela não é utilizada. Deve-se adicionar o código que efetivamente desenha a tela. Isto é feito nas funções `SetupFrame()` e `FinishFrame()`, conforme o quadro 10. Note-se que o Crystal Space é orientado a eventos (*event driver*), portanto, o desenho atual precisa ser disparado pelo tratador de eventos.

QUADRO 10 – CÓDIGO PARA DESENHAR NA TELA

```
void Simple::SetupFrame ()
{
  // Tell 3D driver we're going to display 3D things.
  if (!g3d->BeginDraw(
    engine->GetBeginDrawFlags() | CSDRAW_3DGRAPHICS))
    return;

  // Tell the camera to render into the frame buffer.
  View->Draw ();
}

void Simple::FinishFrame ()
{
  g3d->FinishDraw ();
  g3d->Print (NULL);
}
```

Modifica-se o tratador de eventos, conforme o quadro 11.

QUADRO 11 – ALTERAÇÃO NO CÓDIGO DO TRATADOR DE EVENTOS (*EVENT HANDLER*)

```
bool Simple::HandleEvent (iEvent& ev)
{
  if (ev.Type == csevBroadcast && ev.Command.Code == cscmdProcess)
  {
    simple->SetupFrame ();
    return true;
  }
  else if (ev.Type == csevBroadcast &&
    ev.Command.Code == cscmdFinalProcess)
  {
    simple->FinishFrame ();
    return true;
  }
  else if (ev.Type == csevKeyDown && ev.Key.Code == CSKEY_ESC)
  ...
```

O processo de desenhar a tela é dividido em duas partes. A primeira parte é feita em `SetupFrame()`, onde efetivamente preenche-se a exibição. Neste caso, deixa-se a maioria do trabalho para a *engine* chamando `view->Draw()`.

Em `SetupFrame()`, primeiramente deve-se indicar ao rasterizador 3D que deseja-se iniciar o desenho de gráficos 3D. Esta chamada certifica-se de que todos os *buffers* necessários estão inicializados e realiza as inicializações necessárias. Freqüentemente a *engine* precisa de inicializações extras para isto, assim como é obrigatório a chamada à `engine->GetBeginDrawFlags()` para obter estes *flags* e/ou os parâmetros extras com aqueles que o usuário quer.

A segunda parte está em `FinishFrame()`, onde efetivamente descarrega-se o quadro para a tela. A razão para isto estar dividido, é que outros componentes (*plugins*) no Crystal Space podem escolher atender aos eventos para desenhar "coisas" adicionais no topo da visão renderizada 3D em `SetupFrame()`. Quando um quadro precisa ser renderizado, o CS enviará quatro mensagens:

a) `cscmdPreProcess` é enviado primeiro. Isto permite aos *plugins* à pré-processar algo antes do desenho na tela propriamente dito;
b) em seguida, é enviada a mensagem `cscmdProcess`. Nesta etapa a aplicação efetivamente fará a renderização;

c) após, é enviada a mensagem `cscmdPostProcess`. Nesta etapa isto pode ser utilizado por componentes externos para renderizar no topo da visualização renderizada pela aplicação;

d) por último, é enviada a mensagem `cscmdFinalProcess`. Nesta etapa a aplicação exibirá o quadro na tela.

Ao compilar o código, agora será apresentada uma parede sólida.

2.2.5 LOCOMOÇÃO (MOVENDO-SE NO AMBIENTE)

Simplesmente observar a parede gerada pela aplicação, torna-se muito tedioso após algum tempo. O problema é que não pode-se movimentar a câmara para alterar o ponto de visão. Adiciona-se algum código para fazer exatamente isto, no arquivo `'simple.cpp'`. Deve-se alterar `'SetupFrame()'`, conforme quadro 12.

QUADRO 12 – MOVIMENTANDO A CÂMARA

```
void Simple::SetupFrame ()
{
  // First get elapsed time from the virtual clock.
  CsTicks elapsed_time = vc->GetElapsedTicks ();

  // Now rotate the camera according to keyboard state
  float speed = (elapsed_time / 1000.0) * (0.03 * 20);

  iCamera* c = view->GetCamera();
  if (kbd->GetKeyState (CSKEY_RIGHT))
    c->GetTransform ().RotateThis (VEC_ROT_RIGHT, speed);
  if (kbd->GetKeyState (CSKEY_LEFT))
    c->GetTransform ().RotateThis (VEC_ROT_LEFT, speed);
  if (kbd->GetKeyState (CSKEY_PGUP))
    c->GetTransform ().RotateThis (VEC_TILT_UP, speed);
  if (kbd->GetKeyState (CSKEY_PGDN))
    c->GetTransform ().RotateThis (VEC_TILT_DOWN, speed);
  if (kbd->GetKeyState (CSKEY_UP))
    c->Move (VEC_FORWARD * 4 * speed);
  if (kbd->GetKeyState (CSKEY_DOWN))
    c->Move (VEC_BACKWARD * 4 * speed);
  ...
}
```

Após a inserção do código acima, pode-se rotacionar a câmara com as teclas seta para esquerda e seta para direita e movimentar a câmara para frente e para trás com as teclas seta para cima e seta para baixo. Para rotacionar a câmara, é utilizado `RotateThis()` que aguarda um vetor para rotacionar para frente e um ângulo passado em radianos (o parâmetro `speed`). Existem no CS, vários vetores pré-definidos que podem ser utilizados.

3 MICROSOFT DIRECTX

Segundo Joffe (2001), antes do lançamento do sistema operacional Windows, muitos jogos eram lançados para a plataforma DOS, geralmente utilizando algo como DOS4GW ou alguma outra extensão 32 bits para ter acesso ao modo protegido de 32 bits. Porém, com o surgimento do sistema operacional Windows, os desenvolvedores de jogos começaram a querer saber como eles iriam escrever jogos que rodassem nesta nova plataforma – um jogo tipicamente roda em modo de tela cheia *(full screen)* e precisa estar o mais adequado possível ao *hardware*. O Windows parecia ser a solução para isso. O DOS havia permitido uma programação o mais "fechada" possível, isto é, ir diretamente ao *hardware*, sem precisar passar por camadas de abstração e encapsulamento. Naqueles tempos, a sobrecarga (*overhead*) extra de uma API genérica teria feito os jogos muito lentos.

A resposta da Microsoft para este problema foi a um "*Kit* de Desenvolvimento de *Software*" (KDS) chamado DirectX. Originalmente o DirectX foi comprado de uma empresa de Londres chamada RenderMorphics e já encontrava-se mais ou menos na versão 2. Os vendedores de *hardware* rapidamente viram que seguir a Microsoft era a coisa mais prudente a fazer, e todos começaram a produzir *drivers* para o DirectX em seus *hardwares*. De qualquer forma, isto foi uma coisa boa para os desenvolvedores de jogos.

Um dos principais propósitos do DirectX é prover um estilo padrão de acesso à vários *hardwares* proprietários diferentes. Como exemplo, têm-se o Direct3D que provê uma *interface* de programação padrão que pode ser utilizada para acessar as características do *hardware* acelerador 3D de quase todos os cartões 3D disponíveis no mercado, que possuem *drivers* do Direct3D escritos para eles.

A API DirectX foi projetada principalmente para escrever jogos, mas pode muito bem ser utilizada em outros tipos de aplicações. Segundo Microsoft Corporation (2001?), o DirectX, em sua versão 8.1 possui os componentes descritos na tabela 1.

TABELA 1 – COMPONENTES DO DIRECTX (VERSÃO 8.1)

DirectX Graphics	Uma combinação do DirectDraw e Direct3D. Possui toda a parte de programação gráfica.
DirectX Audio	Uma combinação do DirectSound e DirectMusic, que habilita a programação de áudio.
DirectInput	Provê suporte para uma variedade de dispositivos de entrada (*input devices*), incluindo total suporte à tecnologia *force-feedback*.
DirectPlay	Suporte a jogos de rede multiusuário.
DirectShow	Para captura e reprodução de alta qualidade dos canais multimídia.

3.1 DIRECTDRAW

Segundo Soares (2001?), o DirectDraw é um dos componentes para programação da API DirectX que permite manipular diretamente a memória da placa de vídeo. Com isso, o aplicativo tem uma performance superior àquela utilizada na programação com a *interface* do Windows. O DirectDraw é essencialmente um gerenciador de memória de vídeo. Permite que o programador armazene e manipule mapas de bits (*bitmaps*), diretamente na memória de vídeo. A possibilidade de trocar informações da memória de vídeo para a própria memória de vídeo é muito mais rápido do que trocar informações da memória do sistema para a memória de vídeo. Outra vantagem é que, como o processo de manipulação de imagens é feito pela placa de vídeo, isto permite que o processador central ocupe-se em fazer outras coisas.

Uma aplicação que utiliza o DirectDraw, utiliza dois objetos nomeados `DirectDraw` e `DirectDrawSurface`. O objeto `DirectDraw` representa o cartão adaptador de exibição (*display adapter card*). O objeto `DirectDrawSurface` representa a memória da exibição (*display*), em que os dados a serem exibidos são renderizados (Patwardhan, 2001?).

Um método padrão da utilização do DirectDraw é apresentada abaixo:

a) Criação do *buffer* frontal e do *buffer* de fundo (para transferência de imagens);
b) Imagens a serem exibidas são escritas no *buffer* de fundo, em vez de diretamente na tela;
c) Ao final do desenho, a tela é atualizada jogando-se o *buffer* de fundo e o *buffer* frontal. Após esta operação, o *buffer* de fundo passa a ser o *buffer* frontal.

3.2 DIRECT3D

Conforme Joffe (2001), o Direct3D é parte do DirectX e é o componente que auxilia na integração de gráficos 3D para dentro de aplicações Windows. É utilizado para o desenvolvimento de aplicações em tempo real e iterativas. Para o desenvolvimento destas aplicações, o Direct3D possui as seguintes características:

a) independência de dispositivo: auxilia na proteção das aplicações das particularidades das diferentes plataformas de *hardware*. Como resultado disto, as aplicações tornam-se independentes do *hardware* e conseqüentemente mais portáveis;
b) modelo de *driver* comum para *hardware*: garante às aplicações, que todos os *drivers* que suportam o Direct3D, suportarão um conjunto de definições mínimas das características e capacidades. Devido a isto, as aplicações desenvolvidas que utilizam tais características, funcionarão sobre todas as plataformas de *hardware*. Adicionalmente, o Direct3D provê uma especificação para todos os desenvolvedores de *hardware*, que auxilia os cartões destes desenvolvedores a suportar as várias características do Direct3D. Aplicações que utilizam estas características terão um salto de performance;
c) facilita a adição de gráficos 3D nas aplicações: como o Direct3D provê um mecanismo padrão e um conjunto padrão de algoritmos para gráficos 3D, aplicações que requerem tais características podem ser desenvolvidas muito mais rapidamente;
d) acesso transparente ao acelerador de *hardware*: é uma das características mais importantes do Direct3D que utiliza o suporte ao *hardware*, se disponível. No caso de alguma plataforma de *hardware* não suportar algumas das características, o Direct3D provê uma implementação equivalente no *software*. Esta escolha da utilização das características de *hardware* (se disponível), é transparente ao usuário. A aplicação, em tempo de execução pode detectar as capacidades do *hardware* e utilizá-las.

Em adição à estas características, o Direct3D fornece um *software* ágil baseado na seqüência de renderização 3D completa. Aplicações que são desenvolvidas utilizando o

Direct3D, são fáceis de serem atualizadas em parte ou no todo da seqüência de renderização 3D, que pode estar no *hardware*, e o Direct3D pode fazer uso dela, caso detectada.

3.2.1 CAMADAS (*LAYERS*)

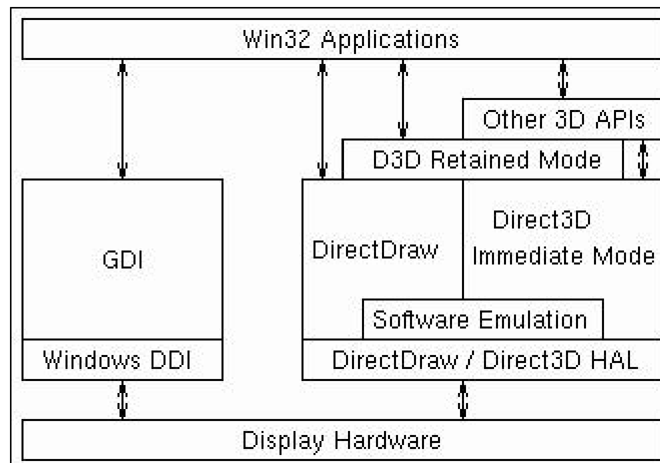
Ainda segundo Joffe (2001), o Direct3D utiliza duas camadas, nomeadas *Hardware Abstraction Layer* (HAL) e *Hardware Emulation Layer* (HEL). Todas as características do Direct3D são construídas no topo da HAL, que fornece independência ao *hardware* e torna as aplicações portáveis. A aplicação não comunica-se diretamente com o dispositivo, mas sim com a HAL. Se o dispositivo for capaz de atender ao comando enviado à HAL, este o envia ao dispositivo. Caso não seja, a HAL pode, dependendo da aplicação, retornar um erro para a aplicação do usuário ou emular em *software* o comando solicitado. Este conceito é conhecido como HEL.

O Direct3D vem acompanhado com a HEL, que fornece emulação em *software* das características da renderização 3D *pipeline*, não suportado pelo *hardware*. Esta camada é intimamente integrada com o DirectDraw, HAL e a *Graphics Device Interface* (GDI) do Windows. Esta camada auxilia no fornecimento de um modelo de *driver* unificado para aceleração 3D.

3.2.2 LOCAL

A figura 3 apresenta as diferentes partes do Direct3D, em relação aos outros módulos de um sistema Win32.

FIGURA 3 – LOCALIZAÇÃO DO DIRECT3D



Ao verificar a figura 3, observa-se que o modo retido (*retained mode*) utiliza o modo imediato (*immediate mode*), o que é transparente ao desenvolvedor utilizando o modo retido. O desenvolvedor não toma ciência desta utilização. O modo retido também utiliza algumas características do DirectDraw. O modo retido, o modo imediato e o Direct3D HAL, juntos, constituem o componente Direct3D do DirectX.

Embora muitos dos programas existentes para gráficos 3D sobre a plataforma Windows se comunicam com as diferentes partes do Direct3D diretamente, provavelmente os componentes DirectDraw e Direct3D do DirectX serão incorporados às futuras versões de sistemas Win32.

Outras informações sobre a API DirectX podem ser encontradas em Strube (2001).

3.3 DIRECTX E COM

O conjunto de módulos do DirectX está construído como objetos COM. Um objeto COM é um pouco semelhante a uma classe de C++, onde estão encapsulados um conjunto de métodos e atributos em um único módulo o qual fornece um tipo de modelo de herança por meio do qual um objeto COM pode ser construído para suportar todos os métodos de seus objetos "pai" (*parent object*) e então adicionar outros.

4 DESENVOLVIMENTO DO PROTÓTIPO

De acordo com os assuntos abordados nos capítulos anteriores, principalmente no que diz respeito à utilização do Crystal Space, pôde-se criar um ambiente em três dimensões utilizando a estrutura do mesmo e seguindo todas as instruções conforme descrito nas documentações que encontram-se em Crystal Space (2000?a).

A seção 5.1 apresenta os principais requisitos necessários para a construção do ambiente e a seção 5.2 apresenta a especificação e implementação do protótipo.

4.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Para a construção do protótipo, foram considerados alguns requisitos básicos:

a) possuir um sistema operacional (Windows 98);
b) possuir um ambiente de desenvolvimento em C++ (Microsoft Visual C++ 6.0): o CS precisa ser construído com um compilador C++;
c) o pacote principal do Crystal Space (versão 0.94r002). Contém: a biblioteca compilada e vários *drivers* (arquivos '`.dll`' e '`.so`');
d) microsoft DirectX 6.1 ou superior;
e) um ou mais arquivos mapa (*map files*);
f) bibliotecas externas `zlib`, `libjpeg` e `libpng`.

4.2 ESPECIFICAÇÃO E IMPLEMENTAÇÃO

Para uma melhor especificação do protótipo, a princípio pode-se fazer uma "divisão" do mesmo em três partes quase que independentes uma da outra, sendo:

a) arquivos de configuração (`.cfg`);
b) o arquivo "`mundo`", que contém todas as definições do ambiente 3D propriamente dito;
c) a classe '`Simple`', que possui dentre outros métodos, o método `Simple::Initialize()` para carregar/inicializar toda a aplicação, e o método `Simple::LoadMap()` para carregar o arquivo "mundo" e fixar a câmara.

4.2.1 ARQUIVO DE CONFIGURAÇÃO (.CFG)

Várias aplicações do Crystal Space possuem um arquivo de configuração acompanhando, que descrevem opções de configuração particulares para as aplicações. Similarmente, vários módulos *plugin* têm arquivos de configuração. Os arquivos de configuração são nomeados com a extensão de arquivo '`.cfg`' e tipicamente estão localizados dentro do diretório virtual '`/config/`' que refere-se geralmente à localização física '`/CS/data/config/`'.

O formato destes arquivos é simples: eles consistem em um número de chaves, cada uma definida com um nome, um valor e um comentário opcional. Nomes de chaves contêm pontos para manter um tipo de hierarquia, similar a um sistema de arquivos: '`Video.OpenGL.EnableDither`'. Isto é interpretado como: na chave '`EnableDither`', na sub-seção '`OpenGL.`', que encontra-se na seção '`Video.`' (nota-se que os pontos fazem parte dos nomes das seções). O quadro 13 apresenta um exemplo de como é um arquivo de configuração.

QUADRO 13 – FORMATO DE UM ARQUIVO DE CONFIGURAÇÃO

```
; Este é o comentário para Section.Keyname
Section.Keyname = Value
; Comentários podem ter
; várias linhas
; A opção 'Section.TestOption' não tem comentários anexos à ela
Video.OpenGL.EnableDither = true
Section.TestOption = false
; Comentários no fim do arquivo não pertencem à quaquer chave de
; configuração. Quando um arquivo de configuração é carregado e salvo
; novamente mais tarde, este comentário de fim-de-arquivo é recolocado no
; final do arquivo de configuração.
```

Quando uma aplicação é executada, todos os campos de configuração são combinados em uma grande base de dados de opções. Quando isto acontece, opções de domínios de maior prioridade sobrescrevem as opções de domínios de mais baixa prioridade.

4.2.2 O ARQUIVO "MUNDO"

O arquivo "mundo" é onde estão todas as definições do ambiente. Com isto, pode-se executar certas modificações em parâmetros do ambiente, sem alterar uma única linha de código no arquivo fonte. Este arquivo encontra-se no diretório físico '`C:\CS\data`'. Alguns trechos de um arquivo de definições do ambiente, encontram-se no quadro 14.

QUADRO 14 – TRECHOS DE UM ARQUIVO DE DEFINIÇÕES DE AMBIENTE

```
WORLD {
  TEXTURES {
    TEXTURE 'spark' (FILE (/lib/std/spark.png))
    TEXTURE 'flare_picir' (FILE (/lib/stdtex/flare_picir.jpg))
    TEXTURE 'muro_quarto_inicial' (FILE (/lib/stdtex/mosholes.png))
    TEXTURE 'flare_center' (FILE (/lib/stdtex/flare_center.jpg))
...
  }
  MATERIALS (
    MATERIAL 'wood' (TEXTURE ('andrew_wood.jpg'))
    MATERIAL 'abstract' (TEXTURE ('abstract_a032.jpg'))
    MATERIAL 'muro_quarto_inicial' (TEXTURE ('muro_quarto_inicial'))
    MATERIAL 'raindrop' (TEXTURE ('raindrop'))
...
SECTOR 'room' (
    MESHOBJ 'walls' (
        ZFILL ()
        PLUGIN ('thing')
        PARAMS (
          VERTEX (-20,-1,-20) VERTEX (-20,-1,20)
          VERTEX (20,-1,20) VERTEX (20,-1,-20)
...
          MATERIAL ('muro_quarto_inicial')
          TEXLEN (5)
          POLYGON 'down1' (VERTICES (0,1,10,15) MATERIAL
                 ('floors_1_d1n__128') COSFACT(0.8)
            TEXTURE (PLANE (floor)))
          POLYGON 'down2' (VERTICES (15,10,11,14) MATERIAL
                 ('floors_1_d1n__128') COSFACT(0.8)
            TEXTURE (PLANE (floor)))
          POLYGON 'down3' (VERTICES (14,11,2,3) MATERIAL
                 ('floors_1_d1n__128') COSFACT(0.8)
            TEXTURE (PLANE (floor)))
          POLYGON 'up1' (VERTICES (5,4,23,18) MATERIAL
                 ('wood') COSFACT(0.8)
            TEXTURE (PLANE (ceiling)))
          POLYGON 'up2' (VERTICES (18,23,22,19) MATERIAL
                 ('wood') COSFACT(0.8)
            TEXTURE (PLANE (ceiling)))
          POLYGON 'up3' (VERTICES (19,22,7,6) MATERIAL
                 ('wood') COSFACT(0.8)
              TEXTURE (PLANE (ceiling)))
...
```

Pode-se por exemplo querer alterar a textura do chão de um dos quartos. Primeiramente, cria-se uma nova textura, carregando um arquivo de imagem conforme o quadro 15.

QUADRO 15 – CRIAÇÃO DE UMA NOVA TEXTURA

```
...
  TEXTURES {
    TEXTURE 'chao_novo' (FILE (/lib/stdtex/mosholes.png))
    ...
  }
...
```

Em seguida, cria-se um material chamado ‘chao_novo’ (pode ser outro nome) e utiliza-se a textura inicialmente criada, confome o quadro 16.

QUADRO 16 – CRIAÇÃO DE UM NOVO MATERIAL

```
...
  MATERIALS {
    MATERIAL 'chao_novo' (TEXTURE ('chao_novo'))
    ...
  }
...
```

Após isto, simplesmente insere-se o material criado, no setor onde será feita a alteração, confome o quadro 17.

QUADRO 17 – ALTERAÇÃO DO CHÃO NO SETOR

```
...
  SECTOR ‘room’ {
  ...
    POLYGON 'down1' (VERTICES (0,1,10,15) MATERIAL ('chao_novo')
                     COSFACT(0.8) TEXTURE (PLANE (floor)))
    POLYGON 'down2' (VERTICES (15,10,11,14) MATERIAL ('chao_novo')
                     COSFACT(0.8) TEXTURE (PLANE (floor)))
    POLYGON 'down3' (VERTICES (14,11,2,3) MATERIAL ('chao_novo')
                     COSFACT(0.8) TEXTURE (PLANE (floor)))
  ...
```

4.2.3 A CLASSE SIMPLE

A classe 'simple' é a única classe da aplicação 'SimpleMap'. A modelagem da classe 'simple' aparece na figura 4 com o seu respectivo código no quadro 18.

FIGURA 4 – MODELO DA CLASSE SIMPLE

QUADRO 18 – ESTRUTURA DA CLASSE 'SIMPLE'

```
class Simple
{
private:
  iObjectRegistry* object_reg;
  iEngine* engine;
  iLoader* loader;
  iGraphics3D* g3d;
  iKeyboardDriver* kbd;
  iVirtualClock* vc;
  iSector* room;
  iView* view;
  static bool SimpleEventHandler (iEvent& ev);
  bool HandleEvent (iEvent& ev);
  void SetupFrame ();
  void FinishFrame ();
  bool LoadMap ();
public:
  Simple ();
  ~Simple ();
  bool Initialize (int argc, const char* const argv[]);
  void Start ();
};
```

Os métodos ‘Simple()’ e ‘~Simple()’ são o construtor e destrutor da classe respectivamente, e seu código pode ser observado no quadro 19.

QUADRO 19 – CÓDIGO DO CONSTRUTOR E DESTRUTOR DE ‘SIMPLE’

```
Simple::Simple ()
{
  engine = NULL;
  loader = NULL;
  g3d = NULL;
  kbd = NULL;
  vc = NULL;
  view = NULL;
}

Simple::~Simple ()
{
  if (vc) vc->DecRef ();
  if (engine) engine->DecRef ();
  if (loader) loader->DecRef();
  if (g3d) g3d->DecRef ();
  if (kbd) kbd->DecRef ();
  if (view) view->DecRef ();
  csInitializer::DestroyApplication (object_reg);
}
```

O destrutor da aplicação faz a chamada ao método ‘DestroyApplication()’ da classe csInitializer, que destrói a aplicação, desfazendo todas as inicializações efetuadas com o método ‘CreateEnvironment()’, ou qualquer outra função de inicialização.

O método ‘Initialize()’ de Simple (quadro 20), primeiramente criará tudo o que é necessário para tornar uma aplicação CS funcional, com a chamada à ‘CreateEnvironment()’ da classe csInitializer. Esta função retornará o ponteiro para o registro do objeto onde todos os objetos criados serão registrados. Em seguida é chamado o método ‘RequestPlugins()’, que inicializa alguns *plugins* padrões amplamente utilizados no CS e também faz a leitura do arquivo de configuração (.cfg) padrão. Para inicializar uma função de tratamento de eventos é chamado o método ‘SetupEventHandler()’. Após as inicializações, é chamado o método ‘OpenApplication()’ que envia o comando cscmdOpen à todos os *plugins* carregados.

QUADRO 20 – CÓDIGO DO MÉTODO ‘INITIALIZE()’

```
bool Simple::Initialize (int argc, const char* const argv[])
{
  object_reg = csInitializer::CreateEnvironment (argc, argv);
  if (!object_reg) return false;
  csDebuggingGraph::SetupGraph (object_reg);

  if (!csInitializer::RequestPlugins (object_reg,
        CS_REQUEST_VFS,
        CS_REQUEST_SOFTWARE3D,
        CS_REQUEST_ENGINE,
        CS_REQUEST_FONTSERVER,
        CS_REQUEST_IMAGELOADER,
        CS_REQUEST_LEVELLOADER,
        CS_REQUEST_REPORTER,
        CS_REQUEST_REPORTERLISTENER,
        CS_REQUEST_END))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Can't initialize plugins!");
    return false;
  }

  if (!csInitializer::SetupEventHandler (object_reg, SimpleEventHandler))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Can't initialize event handler!");
    return false;
  }

  // Verifica por help na linha de comando.
  if (csCommandLineHelper::CheckHelp (object_reg))
  {
    csCommandLineHelper::Help (object_reg);
    return false;
  }

  // O relógio Virtual.
  vc = CS_QUERY_REGISTRY (object_reg, iVirtualClock);
  if (!vc)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Can't find the virtual clock!");
    return false;
  }

  // Encontra o ponteiro para o plugin engine.
  engine = CS_QUERY_REGISTRY (object_reg, iEngine);
  if (!engine)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iEngine plugin!");
    return false;
  }
```

```
  loader = CS_QUERY_REGISTRY (object_reg, iLoader);
  if (!loader)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iLoader plugin!");
    return false;
  }

  g3d = CS_QUERY_REGISTRY (object_reg, iGraphics3D);
  if (!g3d)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iGraphics3D plugin!");
    return false;
  }

  kbd = CS_QUERY_REGISTRY (object_reg, iKeyboardDriver);
  if (!kbd)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iKeyboardDriver plugin!");
    return false;
  }

  // Abre o sistema principal. Isto abrirá todos os plug-ins carregados
anteriormente.
  if (!csInitializer::OpenApplication (object_reg))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Error opening system!");
    return false;
  }

  view = new csView (engine, g3d);
  iGraphics2D* g2d = g3d->GetDriver2D ();
  view->SetRectangle (0, 0, g2d->GetWidth (), g2d->GetHeight ());

  if (!LoadMap ()) return false;
  return true;
}
```

No método ‘`Start()`’, é executado um “*loop*” padrão, que exibe a cena inicial e aguarda a digitação de alguma tecla. Quando digita-se a tecla ‘`PAGE UP`’ por exemplo, é executada a rotação, desenhado o ambiente novamente e exibido na tela, aguardando novo pressionamento do teclado. O código do método ‘`Start()`’ é apresentado no quadro 21.

QUADRO 21 – CÓDIGO DO MÉTODO ‘START()’

```
void Simple::Start ()
{
  csDefaultRunLoop (object_reg);
}
```

O método ‘SimpleEventHandler()’ obtém o retorno do método ‘HandleEvent()’. Este último método por sua vez, verifica por eventos de sistema como o pressionamento do teclado e em caso afirmativo, verifica se a tecla pressionada é a tecla ‘ESC’ ou a tecla numérica ‘1’. Caso a tecla ESC tenha sido pressionada, a aplicação é finalizada. Pressionando-se a tecla numérica ‘1’, é chamado primeiramente o método ‘Dump()’ da classe csDebuggingGraph, que desfaz-se dos gráficos contendo o objeto dado. Após isto é apagada a *engine* por completo com o método ‘DeleteAll()’ da classe csEngine. Por último, é chamado novamente o método ‘LoadMap()’, para carregar o arquivo mapa. O código dos métodos ‘SimpleEventHandler()’ e ‘HandleEvent()’ é apresentado do quadro 22.

QUADRO 22 – CÓDIGO DOS MÉTODOS ‘SIMPLEEVENTHANDLER()’ E ‘HANDLEEVENT()’

```
bool Simple::HandleEvent (iEvent& ev)
{
  if (ev.Type == csevBroadcast && ev.Command.Code == cscmdProcess)
  {
    simple->SetupFrame ();
    return true;
  }
  else if (ev.Type == csevBroadcast && ev.Command.Code ==
cscmdFinalProcess)
  {
    simple->FinishFrame ();
    return true;
  }
  else if (ev.Type == csevKeyDown && ev.Key.Code == CSKEY_ESC)
  {
    iEventQueue* q = CS_QUERY_REGISTRY (object_reg, iEventQueue);
    if (q)
    {
      q->GetEventOutlet()->Broadcast (cscmdQuit);
      q->DecRef ();
    }
    return true;
  }
  else if (ev.Type == csevKeyDown && ev.Key.Code == '1')
  {
    csDebuggingGraph::Dump (NULL);
    engine->DeleteAll ();
    csDebuggingGraph::Dump (NULL);
    csDebuggingGraph::Clear (NULL);
    LoadMap ();
```

```
    return true;
  }

  return false;
}

bool Simple::SimpleEventHandler (iEvent& ev)
{
  return simple->HandleEvent (ev);
}
```

O método ‘`SetupFrame()`’ (quadro 23), primeiramente obtém o tempo percorrido do relógio virtual. Então é efetuada a rotação/movimento da câmara de acordo com o que foi digitado via teclado. Para rotacionar a câmara (Seta para Esquerda, Seta para Direita, *Page Up* e *Page Down*) é utilizado o método ‘`GetTransform()`’ da classe `iCamera` e para movimentar a câmara (Seta para Cima e Seta para Baixo) é utilizado o método ‘`Move()`’. Após a movimentação/rotação da câmara, é chamado o método ‘`Draw()`’ da classe `iView` para desenhar o ambiente 3D novamente como visto pela câmara.

QUADRO 23 – CÓDIGO DO MÉTODO ‘`SETUPFRAME()`’

```
void Simple::SetupFrame ()
{
  // primeiramente, obtém o tempo percorrido do relógio virtual.
  CsTicks elapsed_time = vc->GetElapsedTicks ();

  // agora, gira a câmera de acordo com o determinado no teclado (digitado)
  float speed = (elapsed_time / 1000.0) * (0.03 * 20);

  iCamera* c = view->GetCamera();
  if (kbd->GetKeyState (CSKEY_RIGHT))
    c->GetTransform ().RotateThis (CS_VEC_ROT_RIGHT, speed);
  if (kbd->GetKeyState (CSKEY_LEFT))
    c->GetTransform ().RotateThis (CS_VEC_ROT_LEFT, speed);
  if (kbd->GetKeyState (CSKEY_PGUP))
    c->GetTransform ().RotateThis (CS_VEC_TILT_UP, speed);
  if (kbd->GetKeyState (CSKEY_PGDN))
    c->GetTransform ().RotateThis (CS_VEC_TILT_DOWN, speed);
  if (kbd->GetKeyState (CSKEY_UP))
    c->Move (CS_VEC_FORWARD * 4 * speed);
  if (kbd->GetKeyState (CSKEY_DOWN))
    c->Move (CS_VEC_BACKWARD * 4 * speed);

  // Comunica ao driver 3D serão exibidos objetos 3D.
  if (!g3d->BeginDraw (engine->GetBeginDrawFlags () | CSDRAW_3DGRAPHICS))
    return;

  // Comunica à camera para renderizar para dentro do "frame buffer"
(memória temporária
  // que armazena imagens gráficas que não estão sendo no momento
apresentadas na tela.
  view->Draw ();
}
```

O método '`FinishFrame()`', finaliza a estrutura do ambiente efetuando uma troca de página (*page swap*) e imprime a imagem no *backbuffer*, chamando os métodos '`FinishDraw()`' e '`Print()`' da classe `iGraphics3D` respectivamente. O quadro 24 apresenta o código do método '`FinishFrame()`'.

QUADRO 24 – CÓDIGO DO MÉTODO '`FINISHFRAME()`'

```
void Simple::FinishFrame ()
{
  g3d->FinishDraw ();
  g3d->Print (NULL);
}
```

No método '`LoadMap()`' da classe '`Simple`', carrega-se o mapa criado (arquivo "mundo"). O quadro 25 apresenta o código do método '`LoadMap()`'.

QUADRO 25 – CÓDIGO DO MÉTODO '`LOADMAP()`'

```
Bool Simple::LoadMap ()
{
  // Seta o diretório atual VFS para o plano que deseja-se carregar.
  IVFS* VFS = CS_QUERY_REGISTRY (object_reg, iVFS);
  VFS->ChDir ("/lev/partsys");
  VFS->DecRef ();
  // Carrega o arquivo do plano que é chamado de 'world' e encontra-se no
     diretório C:/CS/Data/PartSys.
  if (!loader->LoadMapFile ("world"))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Couldn't load level!");
    return false;
  }
  engine->Prepare ();

  // Encontra a posição de início neste plano.
  CsVector3 pos (0, 0, 0);
  if (engine->GetCameraPositions ()->GetCount () > 0)
  {
    // Existe uma posição de início válida definida neste arquivo de plano,
       no caso "world".
    ICameraPosition* campos = engine->GetCameraPositions ()->Get (0);
    Room = engine->GetSectors ()->FindByName (campos->GetSector ());
    Pos = campos->GetPosition ();
  }
  else
  {
    // Não foi encontrada uma posição de início válida. Então, por
       definição, procura-se um espaço chamado 'room' na posição (0,0,0).
    Room = engine->GetSectors ()->FindByName ("room");
  }
  if (!room)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
```

```
        "crystalspace.application.simple1",
        "Can't find a valid starting position!");
    return false;
  }

  view->GetCamera ()->SetSector (room);
  view->GetCamera ()->GetTransform ().SetOrigin (pos);

  iTextureManager* txtmgr = g3d->GetTextureManager ();
  txtmgr->SetPalette ();
  return true;
}
```

Este código utiliza primeiramente `iVFS::ChDir()` para setar o diretório atual no sistema de arquivos virtuais (*virtual file system*) para '`/lev/partsys/`'. A chamada à `iLoader::LoadMapFile()`, pegará o nome de arquivo informado (no caso, '`world`') e abre-o do diretório VFS corrente. Este arquivo é particionado e é criada a geometria especificada nele.

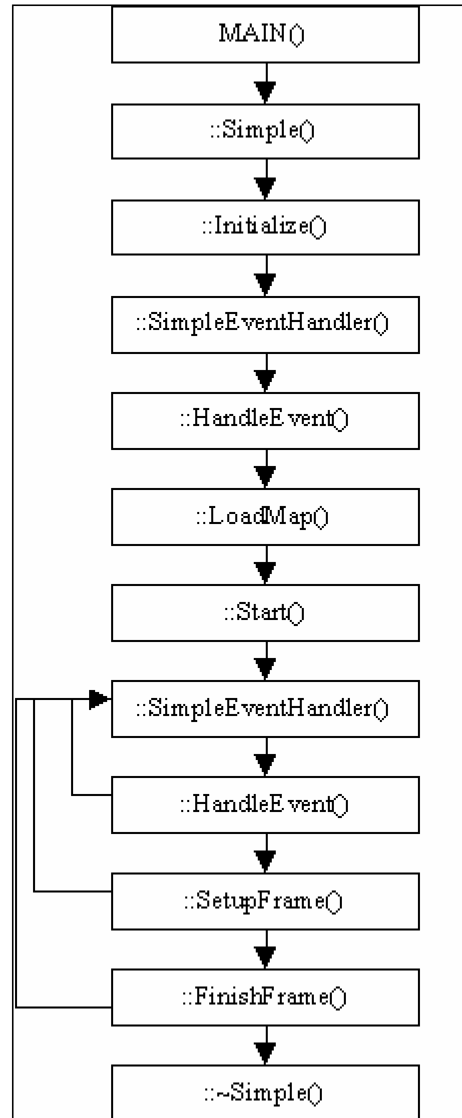
Após isto, chama-se `iEngine::Prepare()` para ter certeza de que todos os mapas de luzes (*lightmaps*) foram corretamente carregados da memória *cache* e se todas as inicializações estão feitas (ex.: texturas registradas, etc).

Como no arquivo '`world`' existe um número de setores e vários objetos, estes são todos carregados em memória. Entretanto isto não é suficiente. Precisa-se inicializar a câmara para algum setor e posição neste ambiente (deve-se lembrar que no Crystal Space uma posição no espaço sempre é definida como um setor em combinação com uma posição). Com o mapa carregado, não se sabe quais setores existem no mapa e onde pode-se inserir a câmara com segurança. Nos arquivos mapa, é possível especificar uma ou mais posições de inicialização.

Utiliza-se `iEngine::GetCameraPositionCount()`, para verificar quantas posições válidas para a câmara existem definidas no mapa. Se isto for 0, então o mapa não define uma posição de início. Neste caso, assume-se que existe um setor chamado '`room`', e inicializa-se a câmara em (0,0,0) neste setor.

A figura 5 apresenta a seqüência de métodos que são executados na classe '`Simple`'.

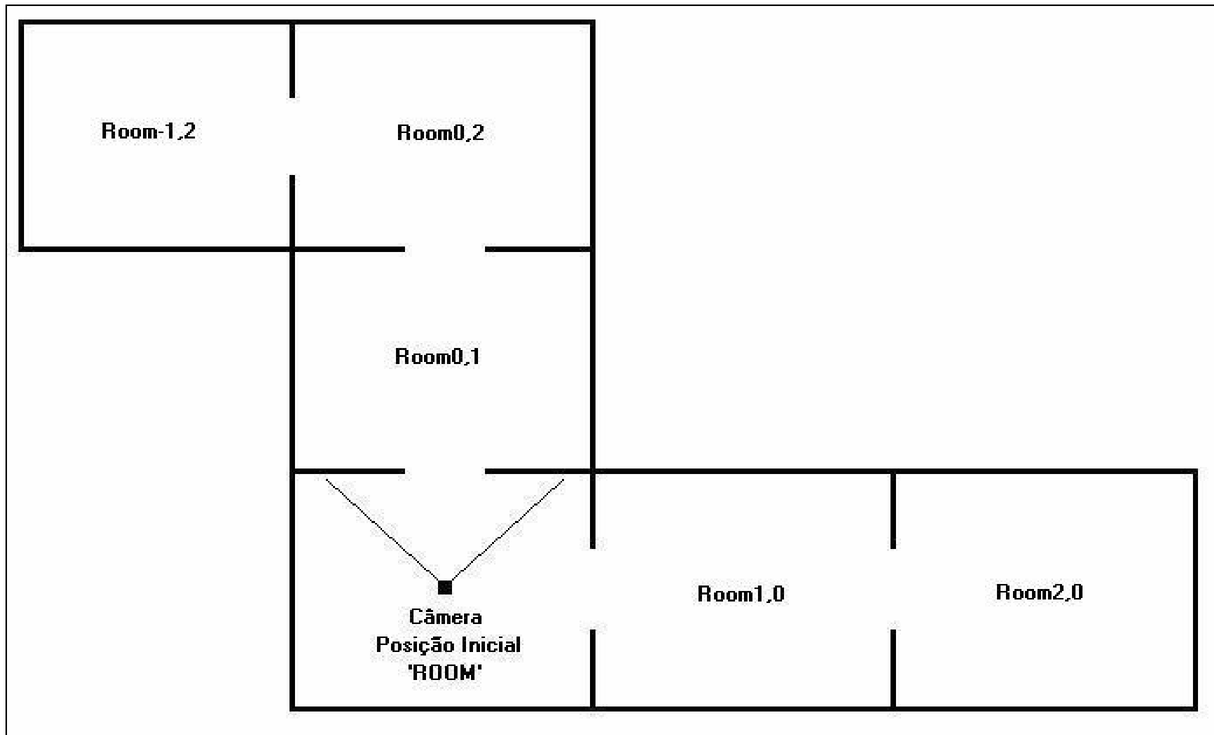
FIGURA 5 – SEQÜÊNCIA DA EXECUÇÃO DOS MÉTODOS DA CLASSE '`SIMPLE`'



4.2.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Nesta seção será apresentada a *interface* do protótipo bem como o seu funcionamento em geral. A figura 6 apresenta a planta baixa do protótipo.

FIGURA 6 – PLANTA BAIXA DO PROTÓTIPO



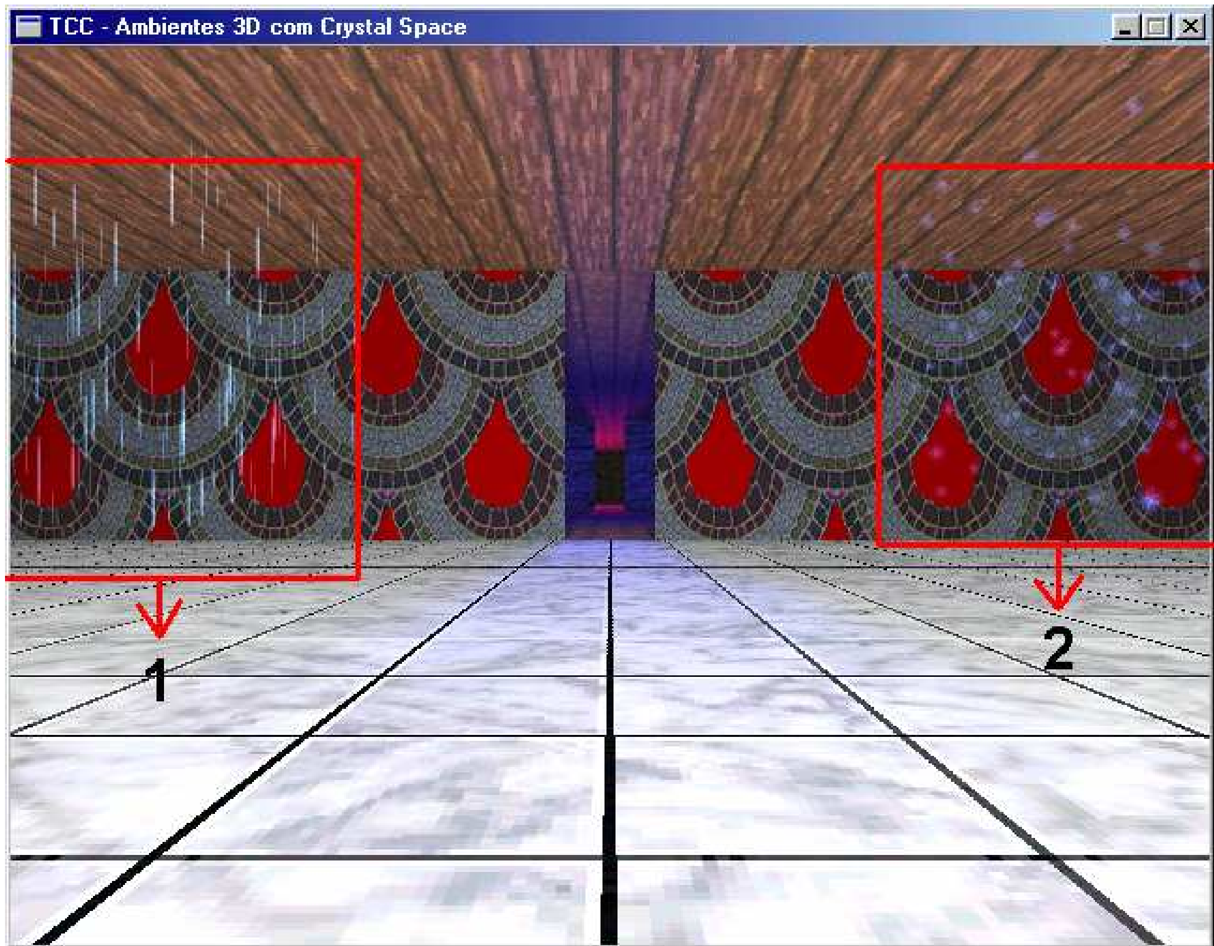
Para executar o protótipo, basta executar o arquivo '`simpmap.exe`' que encontra-se no diretório principal do Crystal Space (C:\CS). Para utilizar o mesmo, basta navegar no ambiente conforme os comandos relacionados na tabela 2.

TABELA 2 – COMANDOS PARA NAVEGAÇÃO NO AMBIENTE

Tecla	Efeito
Seta para Cima	Movimento para frente
Seta para Baixo	Movimento para trás
Seta para Direita	Movimenta para a direita
Seta para Esquerda	Movimenta para a esquerda
Page Up	Rotação para cima
Page Down	Rotação para baixo

A câmara é iniciada no setor '`room`' do arquivo '`world`'. A figura 7 apresenta a cena inicial do protótipo, identificado como "Câmara – Posição Inicial" na planta baixa da figura 6.

FIGURA 7 – CENA INICIAL DO PROTÓTIPO



Alguns dos “efeitos” apresentados no protótipo como o efeito de chuva (1) e neve (2) aparecem na sala inicial ‘`room`’, conforme figura 7. Na figura 8, podem ser observados os efeitos de fumaça azul (3) e fogo (4), encontrados na sala ‘`room-1,2`’ e na figura 9 o efeito de “transparência” (5), localizado na sala ‘`room1,0`’.

FIGURA 8 – FUMAÇA E FOGO

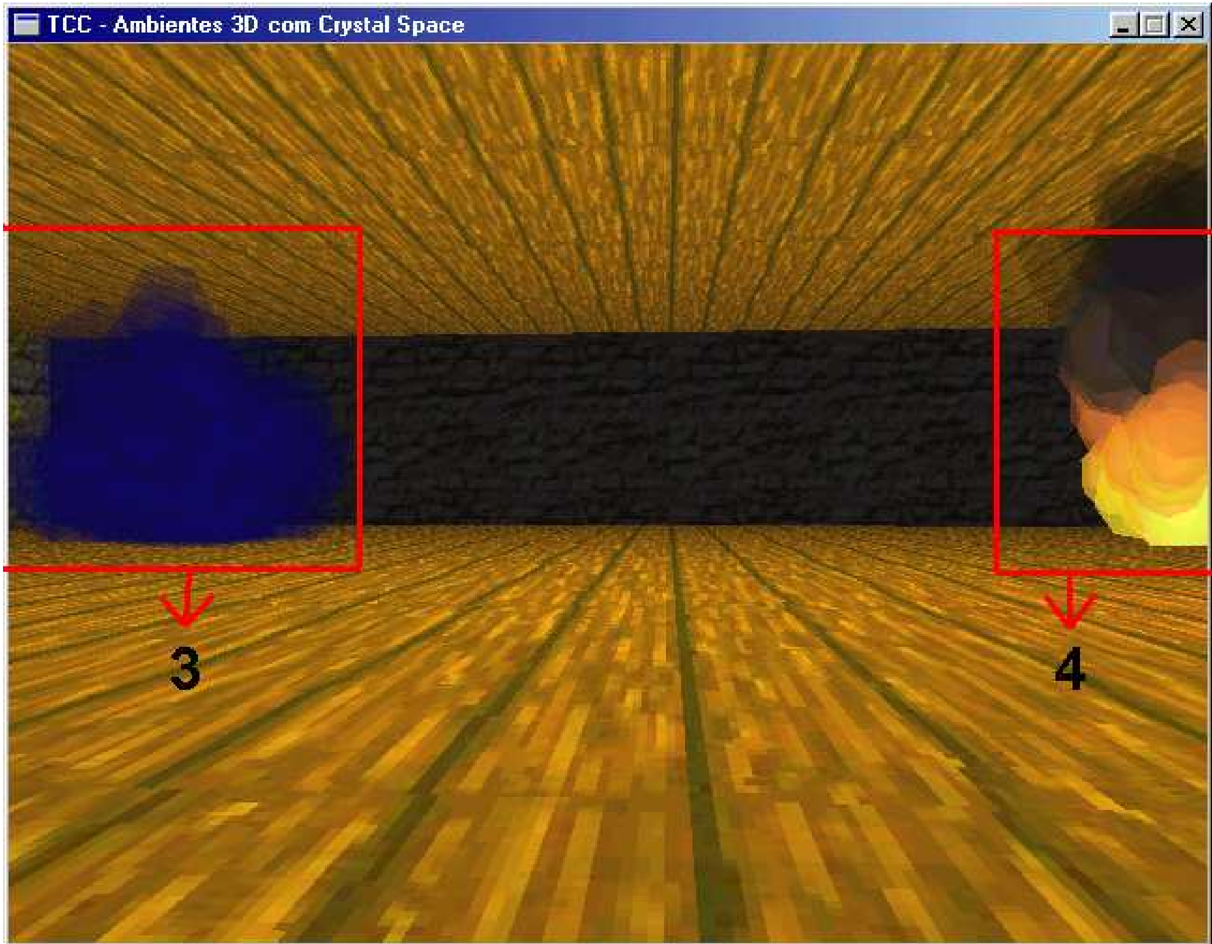
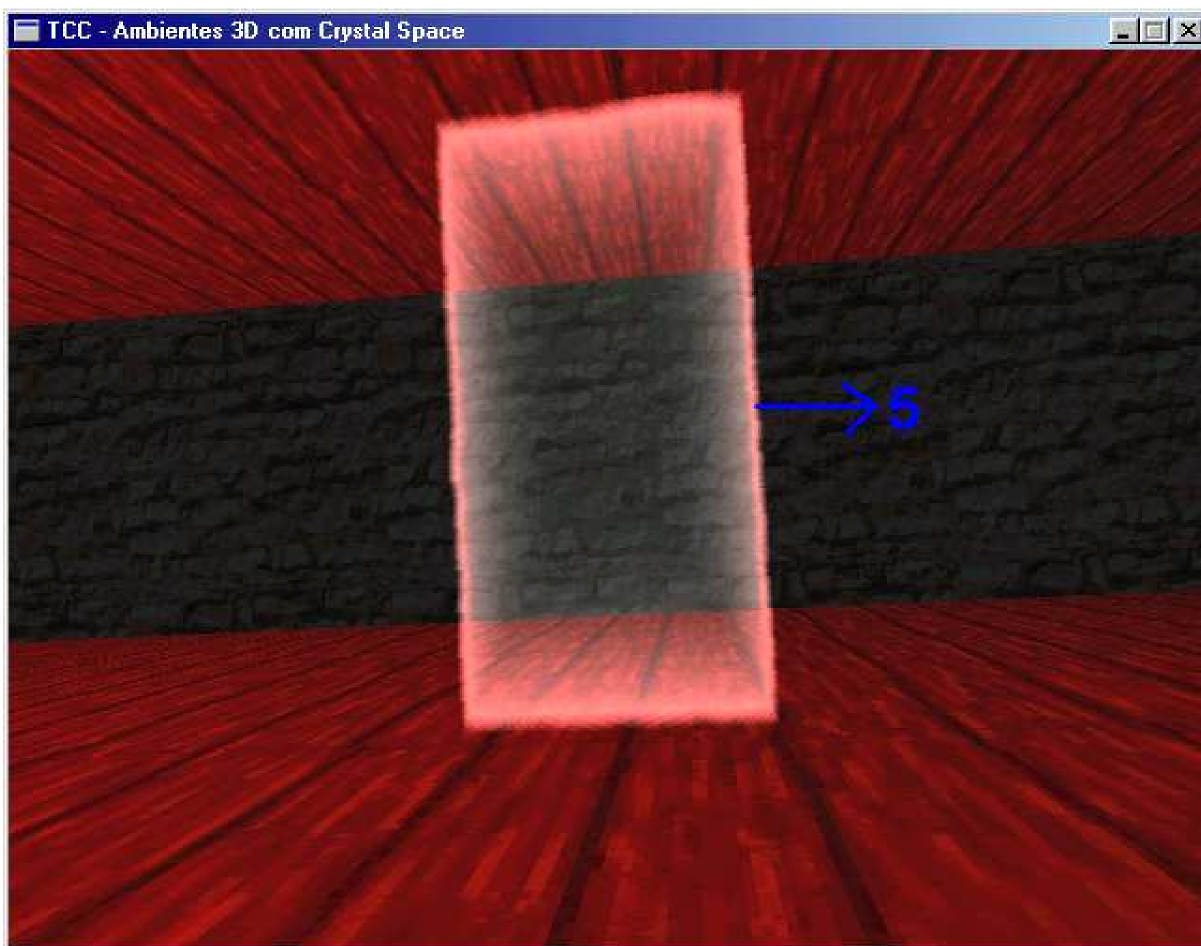


FIGURA 9 - TRANSPARÊNCIA



Não existe detecção de colisão no protótipo, o que não significa que pode-se atravessar as paredes. É possível continuar avançando, mas a visualização disto é difícil. A partir do momento em que a câmara "bate" na parede, ao continuar avançando com a tecla 'Seta para Cima', tem-se a impressão de que não há mais um avanço. Porém, ao teclar a 'Seta para Direita', verifica-se algo como demonstrado na figura 10, onde tem-se a impressão de estar dentro da parede.

FIGURA 10 – SEM DETECÇÃO DE COLISÃO

Também não existe a implementação de “gravidade” no ambiente (embora seja possível fazê-lo no Crystal Space). Sendo assim, a câmara pode “flutuar” livremente por todo o ambiente, limitando-se somente às paredes, ao teto e ao chão.

5 CONCLUSÕES

Durante a elaboração deste trabalho, foram identificados vários pontos cruciais para a construção de um ambiente 3D com a *engine* Crystal Space, desde os requisitos até ao funcionamento da mesma.

Atenção especial deve ser dada ao fato de o CS estar em constante evolução e que não há garantias de seu funcionamento. Não há versões totalmente estáveis. Optou-se por utilizar a versão 094_r002 do CS, pelo fato de ser a primeira versão que foi possível compilar na máquina utilizada.

Um dos pontos mais positivos do Crystal Space é a sua documentação. Foi baseado nesta documentação (redigida por vários colaboradores no mundo todo) encontrada em Crystal Space (2000?b), que foi possível um entendimento da *engine*, do seu funcionamento e todos os detalhes para compilação da mesma. Existe também uma ótima documentação de diversas classes e seus respectivos métodos e tutoriais que demonstram um passo-a-passo do código de algumas aplicações que fazem parte do CS, em uma linguagem muito natural e direta.

A máquina utilizada para a construção do protótipo não possuía *hardware* de aceleração gráfica, item exigido pela documentação da *engine* e que não impossibilitou o seu funcionamento.

A utilização do Microsoft Visual C++ 6.0 (que é recomendada como “preferência” pela documentação existente do Crystal Space), mostrou-se uma opção favorável, pois todo o código da *engine* CS está estruturado dentro de vários projetos do Visual C++ e estes projetos dentro de uma *WorkSpace* (arquivo `.dsw`). Com isto, foi possível a utilização deste código sem a necessidade de qualquer alteração nos mesmos.

Uma das dificuldades encontradas, foi no momento da compilação do CS. Gastou-se precioso tempo até ser possível a compilação de exemplos que acompanham o CS. Deve-se tomar certo cuidado com aplicações que executam em *background*, como o Norton Antivírus por exemplo, que impedia a criação das bibliotecas do CS e acarretavam no “travamento” do sistema operacional. Nas documentações encontradas, não havia nenhuma observação sobre este fato.

Já quanto à utilização da *engine* com o DirectX, pôde-se observar que somente são utilizados alguns arquivos (`.dll`) do mesmo, onde não foi observado o local exato onde é feita a chamada a estes arquivos.

Tendo-se o conhecimento dos principais aspectos referentes à *engine* Crystal Space, foi possível a construção do protótipo alcançando os objetivos desse trabalho, gerando o ambiente tridimensional com "efeitos" e locomoção neste ambiente em visão de primeira pessoa.

5.1 EXTENSÕES

Algumas das possíveis extensões que podem ser feitas a partir deste trabalho estão descritas abaixo:

a) Implementação de detecção de colisão no ambiente existente ('`world`');
b) Geração do mesmo ambiente com as mesmas características para alguma outra plataforma (unix, linux, mac, etc);
c) Implementar outros comandos de teclado para o '`SimpleMap`' (tiro, pular, etc), conforme já existem em outras aplicações do Crystal Space ('`WalkTest`');
d) Tornar este ambiente um ambiente multiusuário;
e) Implementar o efeito de "gravidade" à câmara, não permitindo que a mesma chegue ao teto ou ao solo.

REFERÊNCIAS BIBLIOGRÁFICAS

BELTRÃO, Francisco Rego. **Protótipo de um ambiente 2d na área de entretenimento, utilizando recursos multimídia**. 2000. 66 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CARDOSO, Gustavo. **Os jogos multimedia como meta-sistema de entretenimento**, [S.l.], [2000?]. Disponível em: <http://www.cav.iscte.pt/~gustavo/snm/metajogos.htm>. Acesso em: 02 nov. 2001.

CRYSTAL SPACE. ***Crystal Space*.** [S.l.], [2000?a]. Disponível em: <http://crystal.sourceforge.net>. Acesso em: 02 nov. 2001.

CRYSTAL SPACE. ***Crystal Space*.** [S.l.], [2000?b]. Disponível em: <http://crystal.sourceforge.net/docs/online/manual/cs_12.php#SEC14>. Acesso em: 02 nov. 2001.

EBERLY, David H. ***3d game engine design****: a practical approach to real-time computer graphics*. San Francisco: Morgan Kaufmann, 2001. 560p.

JOFFE, David. ***Game programming with directx*.** [S.l.], 2001. Disponível em: <http://www.scorpioncity.com/djdirectxtut.html>. Acesso em: 08 mar. 2002.

LAMOTHE, André. ***Tricks of the windows game programming gurus***. Indianapolis : Sams, 1999.

LAMOTHE, André. ***DirectX-tasy***, [S.l.], [2001?]. Disponível em: <http://www.xgames3d.com>. Acesso em: 02 nov. 2001.

MICROSOFT CORPORATION. ***DirectX***. Redmond, [2001?a]. Disponível em: <http://www.microsoft.com/directx/>. Acesso em: 02 nov. 2001.

O'NEILL, Rory; MUIR, Eden Greig. ***Web developer.com guide to creating 3D worlds***. New York: Wiley Computer Publishing, 1998.

PATWARDHAN, Bipin. ***An overview of direct3d.*** Juhu, [2001?]. Disponível em: <http://www.gamedev.net/>. Acesso em: 15 fev. 2002.

RAITZ, Luciano. **Estudo e avaliação de alguns métodos de triangularizações de pontos dispersos em uma superfície 3d**. 2001. 96 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

RIO GRÁFICA LTDA. **Microcomputador:** jogos. Rio Gráfica, 1985.

SILVA, Fernanda Andrade Bordallo da. **Protótipo de um ambiente para geração de superfícies 3d com uso de *spline* bézier**. 2000. 66 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SOARES, Geovanni. **Programando com *directdraw* no visual c++**, [S.l.], [2001?]. Disponível em: <http://www.programadoresdejogos.com.br/programando.htm>. Acesso em: 02 nov. 2001.

STRUBE, Alexandre Otto. **Comparação entre as bibliotecas gráficas direct3d e opengl**. 2001. 58 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SULLY, Phil. ***Modeling the world with objects***. Cambridge: Prentice Hall, 1993. 283p.

TREMBLAY, Thierry. ***3d basics.*** [S.l], [1999?]. Disponível em: <http://www.gamedev.net/reference/articles/article673.asp>. Acesso em: 01 abr. 2002.

WATT, Allan; POLICARPO, Fabio. ***3d games****: real-time rendering and software*. 1. Ed. Washington: Wesley, 2000.

ANEXO A: CÓDIGO FONTE DO ARQUIVO

'SIMPLE.CPP'

```
#include "cssysdef.h"
#include "cssys/sysfunc.h"
#include "iutil/vfs.h"
#include "csutil/cscolor.h"
#include "cstool/csview.h"
#include "cstool/initapp.h"
#include "simple.h"
#include "iutil/eventq.h"
#include "iutil/event.h"
#include "iutil/objreg.h"
#include "iutil/csinput.h"
#include "iutil/virtclk.h"
#include "iengine/sector.h"
#include "iengine/engine.h"
#include "iengine/camera.h"
#include "iengine/light.h"
#include "iengine/statlght.h"
#include "iengine/texture.h"
#include "iengine/mesh.h"
#include "iengine/movable.h"
#include "iengine/material.h"
#include "imesh/thing/polygon.h"
#include "imesh/thing/thing.h"
#include "imesh/object.h"
#include "ivideo/graph3d.h"
#include "ivideo/graph2d.h"
#include "ivideo/txtmgr.h"
#include "ivideo/texture.h"
#include "ivideo/material.h"
#include "ivideo/fontserv.h"
#include "igraphic/imageio.h"
#include "imap/parser.h"
#include "ivaria/reporter.h"
#include "ivaria/stdrep.h"
#include "csutil/cmdhelp.h"

CS_IMPLEMENT_APPLICATION

// O ponteiro global para 'simple'
Simple *simple;

Simple::Simple ()
{
  engine = NULL;
  loader = NULL;
  g3d = NULL;
  kbd = NULL;
  vc = NULL;
}

Simple::~Simple ()
```

```
{
  if (vc) vc->DecRef ();
  if (engine) engine->DecRef ();
  if (loader) loader->DecRef();
  if (g3d) g3d->DecRef ();
  if (kbd) kbd->DecRef ();
  csInitializer::DestroyApplication (object_reg);
}

bool Simple::Initialize (int argc, const char* const argv[])
{
  object_reg = csInitializer::CreateEnvironment ();
  if (!object_reg) return false;

  csInitializer::SetupCommandLineParser (object_reg, argc, argv);
  if (!csInitializer::RequestPlugins (object_reg,
        CS_REQUEST_VFS,
        CS_REQUEST_SOFTWARE3D,
        CS_REQUEST_ENGINE,
        CS_REQUEST_FONTSERVER,
        CS_REQUEST_IMAGELOADER,
        CS_REQUEST_LEVELLOADER,
        CS_REQUEST_REPORTER,
        CS_REQUEST_REPORTERLISTENER,
        CS_REQUEST_END))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Can't initialize plugins!");
    return false;
  }

  // Verifica por ajuda na linha de comando(Check for commandline help)
  if (csCommandLineHelper::CheckHelp (object_reg))
  {
    csCommandLineHelper::Help (object_reg);
    return false;
  }

  // O Relógio Virtual.
  vc = CS_QUERY_REGISTRY (object_reg, iVirtualClock);
  if (!vc)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Não foi possível encontrar o Relógio Virtual!");
    return false;
  }

  // Encontrar o ponteiro para o plugin engine
  engine = CS_QUERY_REGISTRY (object_reg, iEngine);
  if (!engine)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Não existe o plugin iEngine!");
    return false;
  }
```

```
  loader = CS_QUERY_REGISTRY (object_reg, iLoader);
  if (!loader)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Não existe o plugin iLoader!");
    return false;
  }

  g3d = CS_QUERY_REGISTRY (object_reg, iGraphics3D);
  if (!g3d)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Não existe o plugin iEngine3D!");
    return false;
  }

  kbd = CS_QUERY_REGISTRY (object_reg, iKeyboardDriver);
  if (!kbd)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Não existe o plugin iKeyboardDriver!");
    return false;
  }

  // Open the main system. This will open all the previously
// loaded plugins.
  if (!csInitializer::OpenApplication (object_reg))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Error opening system!");
    return false;
  }

  csReport (object_reg, CS_REPORTER_SEVERITY_NOTIFY,
        "crystalspace.application.simple",
        "Simple Crystal Space Application version 0.1.");

  return true;
}

void Simple::Start ()
{
  csDefaultRunLoop (object_reg);
}

/*---------------*
 * Main function
 *---------------*/
int main (int argc, char* argv[])
{
  simple = new Simple ();

  if (simple->Initialize (argc, argv))
    simple->Start ();
```

```
  delete simple;
  return 0;
}
```

ANEXO B: CÓDIGO PARA ADIÇÃO DE ITENS À JANELA NO ARQUIVO 'SIMPLE.CPP'

```
bool Simple::Initialize (int argc, const char* const argv[])
{
  ...
  if (!csInitializer::OpenApplication (object_reg))
  {
    ...
  }
  // Setup the texture manager
  iTextureManager* txtmgr = g3d->GetTextureManager ();
  txtmgr->SetVerbose (true);

  // Initialize the texture manager
  txtmgr->ResetPalette ();
  ...
  // First disable the lighting cache. Our app is simple enough
  // not to need this.
  engine->SetLightingCacheMode (0);

  if (!loader->LoadTexture ("stone", "/lib/std/stone4.gif"))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple",
        "Error loading 'stone4' texture!");
    return false;
  }
  iMaterialWrapper* tm =
    engine->GetMaterialList ()->FindByName ("stone");
  room = engine->CreateSector ("room");
  iMeshWrapper* walls =
    engine->CreateSectorWallsMesh (room, "walls");
  iThingState* walls_state =
    SCF_QUERY_INTERFACE (walls->GetMeshObject (), iThingState);
  iPolygon3D* p;
  p = walls_state->CreatePolygon ();
  p->SetMaterial (tm);
  p->CreateVertex (csVector3 (-5, 0, 5));
  p->CreateVertex (csVector3 (5, 0, 5));
  p->CreateVertex (csVector3 (5, 0, -5));
  p->CreateVertex (csVector3 (-5, 0, -5));
  p->SetTextureSpace (p->GetVertex (0), p->GetVertex (1), 3);

  p = walls_state->CreatePolygon ();
  p->SetMaterial (tm);
  p->CreateVertex (csVector3 (-5, 20, -5));
  p->CreateVertex (csVector3 (5, 20, -5));
  p->CreateVertex (csVector3 (5, 20, 5));
  p->CreateVertex (csVector3 (-5, 20, 5));
  p->SetTextureSpace (p->GetVertex (0), p->GetVertex (1), 3);

  p = walls_state->CreatePolygon ();
  p->SetMaterial (tm);
```

```
  p->CreateVertex (csVector3 (-5, 20, 5));
  p->CreateVertex (csVector3 (5, 20, 5));
  p->CreateVertex (csVector3 (5, 0, 5));
  p->CreateVertex (csVector3 (-5, 0, 5));
  p->SetTextureSpace (p->GetVertex (0), p->GetVertex (1), 3);

  p = walls_state->CreatePolygon ();
  p->SetMaterial (tm);
  p->CreateVertex (csVector3 (5, 20, 5));
  p->CreateVertex (csVector3 (5, 20, -5));
  p->CreateVertex (csVector3 (5, 0, -5));
  p->CreateVertex (csVector3 (5, 0, 5));
  p->SetTextureSpace (p->GetVertex (0), p->GetVertex (1), 3);

  p = walls_state->CreatePolygon ();
  p->SetMaterial (tm);
  p->CreateVertex (csVector3 (-5, 20, -5));
  p->CreateVertex (csVector3 (-5, 20, 5));
  p->CreateVertex (csVector3 (-5, 0, 5));
  p->CreateVertex (csVector3 (-5, 0, -5));
  p->SetTextureSpace (p->GetVertex (0), p->GetVertex (1), 3);

  p = walls_state->CreatePolygon ();
  p->SetMaterial (tm);
  p->CreateVertex (csVector3 (5, 20, -5));
  p->CreateVertex (csVector3 (-5, 20, -5));
  p->CreateVertex (csVector3 (-5, 0, -5));
  p->CreateVertex (csVector3 (5, 0, -5));
  p->SetTextureSpace (p->GetVertex (0), p->GetVertex (1), 3);

  walls_state->DecRef ();
  walls->DecRef ();
  iStatLight* light;
  iLightList* ll = room->GetLights ();

  light = engine->CreateLight (NULL, csVector3 (-3, 5, 0), 10,
        csColor (1, 0, 0), false);
  ll->Add (light->QueryLight ());
  light->DecRef ();

  light = engine->CreateLight (NULL, csVector3 (3, 5,  0), 10,
        csColor (0, 0, 1), false);
  ll->Add (light->QueryLight ());
  light->DecRef ();

  light = engine->CreateLight (NULL, csVector3 (0, 5, -3), 10,
        csColor (0, 1, 0), false);
  ll->Add (light->QueryLight ());
  light->DecRef ();

  engine->Prepare ();

  iTextureManager* txtmgr = g3d->GetTextureManager ();
  txtmgr->SetPalette ();

  return true;

}
```

ANEXO C: CÓDIGO FONTE 'SIMPLEMAP.CPP'

```
#include "cssysdef.h"
#include "cssys/sysfunc.h"
#include "iutil/vfs.h"
#include "csutil/cscolor.h"
#include "cstool/csview.h"
#include "cstool/initapp.h"
#include "simpmap.h"
#include "iutil/eventq.h"
#include "iutil/event.h"
#include "iutil/objreg.h"
#include "iutil/csinput.h"
#include "iutil/virtclk.h"
#include "iengine/sector.h"
#include "iengine/engine.h"
#include "iengine/camera.h"
#include "iengine/light.h"
#include "iengine/statlght.h"
#include "iengine/texture.h"
#include "iengine/mesh.h"
#include "iengine/movable.h"
#include "iengine/material.h"
#include "iengine/campos.h"
#include "imesh/thing/polygon.h"
#include "imesh/thing/thing.h"
#include "imesh/object.h"
#include "ivideo/graph3d.h"
#include "ivideo/graph2d.h"
#include "ivideo/txtmgr.h"
#include "ivideo/texture.h"
#include "ivideo/material.h"
#include "ivideo/fontserv.h"
#include "igraphic/imageio.h"
#include "imap/parser.h"
#include "ivaria/reporter.h"
#include "ivaria/stdrep.h"
#include "csutil/cmdhelp.h"
#include "csutil/debug.h"

CS_IMPLEMENT_APPLICATION

//-----------------------------------------------------------------------------

// O ponteiro global para Simple
Simple *simple;

//Construtor de Simple
Simple::Simple ()
{
  engine = NULL;
  loader = NULL;
  g3d = NULL;
  kbd = NULL;
  vc = NULL;
```

```
  view = NULL;
}

Simple::~Simple ()
{
  if (vc) vc->DecRef ();
  if (engine) engine->DecRef ();
  if (loader) loader->DecRef();
  if (g3d) g3d->DecRef ();
  if (kbd) kbd->DecRef ();
  if (view) view->DecRef ();
  csInitializer::DestroyApplication (object_reg);
}

void Simple::SetupFrame ()
{
  // primeiramente, obtém o tempo percorrido do relógio virtual.
  csTicks elapsed_time = vc->GetElapsedTicks ();

  // agora, gira a câmera de acordo com o determinado no teclado (digitado)
  float speed = (elapsed_time / 1000.0) * (0.03 * 20);

  iCamera* c = view->GetCamera();
  if (kbd->GetKeyState (CSKEY_RIGHT))
    c->GetTransform ().RotateThis (CS_VEC_ROT_RIGHT, speed);
  if (kbd->GetKeyState (CSKEY_LEFT))
    c->GetTransform ().RotateThis (CS_VEC_ROT_LEFT, speed);
  if (kbd->GetKeyState (CSKEY_PGUP))
    c->GetTransform ().RotateThis (CS_VEC_TILT_UP, speed);
  if (kbd->GetKeyState (CSKEY_PGDN))
    c->GetTransform ().RotateThis (CS_VEC_TILT_DOWN, speed);
  if (kbd->GetKeyState (CSKEY_UP))
    c->Move (CS_VEC_FORWARD * 4 * speed);
  if (kbd->GetKeyState (CSKEY_DOWN))
    c->Move (CS_VEC_BACKWARD * 4 * speed);

  // Comunica ao driver 3D serão exibidos objetos 3D.
  if (!g3d->BeginDraw (engine->GetBeginDrawFlags () | CSDRAW_3DGRAPHICS))
    return;

  // Comunica à camera para renderizar para dentro do "frame buffer"
(memória temporária
  // que armazena imagens gráficas que não estão sendo no momento
apresentadas na tela.
  view->Draw ();
}

void Simple::FinishFrame ()
{
  g3d->FinishDraw ();
  g3d->Print (NULL);
}

bool Simple::HandleEvent (iEvent& ev)
{
  if (ev.Type == csevBroadcast && ev.Command.Code == cscmdProcess)
  {
    simple->SetupFrame ();
    return true;
```

```
  }
  else if (ev.Type == csevBroadcast && ev.Command.Code ==
cscmdFinalProcess)
  {
    simple->FinishFrame ();
    return true;
  }
  else if (ev.Type == csevKeyDown && ev.Key.Code == CSKEY_ESC)
  {
    iEventQueue* q = CS_QUERY_REGISTRY (object_reg, iEventQueue);
    if (q)
    {
      q->GetEventOutlet()->Broadcast (cscmdQuit);
      q->DecRef ();
    }
    return true;
  }
  else if (ev.Type == csevKeyDown && ev.Key.Code == 'l')
  {
    csDebuggingGraph::Dump (NULL);
    engine->DeleteAll ();
    csDebuggingGraph::Dump (NULL);
    csDebuggingGraph::Clear (NULL);
    LoadMap ();
    return true;
  }

  return false;
}

bool Simple::SimpleEventHandler (iEvent& ev)
{
  return simple->HandleEvent (ev);
}

bool Simple::LoadMap ()
{
  // Seta o diretório atual VFS para o plano que deseja-se carregar.
  iVFS* VFS = CS_QUERY_REGISTRY (object_reg, iVFS);
  VFS->ChDir ("/lev/partsys");
  VFS->DecRef ();
  // Carrega o arquivo do plano que é chamado de 'world' e encontra-se no
diretório C:/CS/Data/PartSys.
  if (!loader->LoadMapFile ("my_world"))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Couldn't load level!");
    return false;
  }
  engine->Prepare ();

  // Encontra a posição de início neste plano.
  csVector3 pos (0, 0, 0);
  if (engine->GetCameraPositions ()->GetCount () > 0)
  {
    // Existe uma posição de início válida definida neste arquivo de plano,
no caso "world".
    iCameraPosition* campos = engine->GetCameraPositions ()->Get (0);
```

```
    room = engine->GetSectors ()->FindByName (campos->GetSector ());
    pos = campos->GetPosition ();
  }
  else
  {
    // Não foi encontrada uma posição de início válida. Então, por
definição, procura-se um espaço
    // chamado 'room' na posição (0,0,0).
    room = engine->GetSectors ()->FindByName ("room");
  }
  if (!room)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simple1",
        "Can't find a valid starting position!");
    return false;
  }

  view->GetCamera ()->SetSector (room);
  view->GetCamera ()->GetTransform ().SetOrigin (pos);

  iTextureManager* txtmgr = g3d->GetTextureManager ();
  txtmgr->SetPalette ();
  return true;
}

bool Simple::Initialize (int argc, const char* const argv[])
{
  object_reg = csInitializer::CreateEnvironment (argc, argv);
  if (!object_reg) return false;
  csDebuggingGraph::SetupGraph (object_reg);

  if (!csInitializer::RequestPlugins (object_reg,
        CS_REQUEST_VFS,
        CS_REQUEST_SOFTWARE3D,
        CS_REQUEST_ENGINE,
        CS_REQUEST_FONTSERVER,
        CS_REQUEST_IMAGELOADER,
        CS_REQUEST_LEVELLOADER,
        CS_REQUEST_REPORTER,
        CS_REQUEST_REPORTERLISTENER,
        CS_REQUEST_END))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Can't initialize plugins!");
    return false;
  }

  if (!csInitializer::SetupEventHandler (object_reg, SimpleEventHandler))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Can't initialize event handler!");
    return false;
  }

  // Verifica por help na linha de comando.
  if (csCommandLineHelper::CheckHelp (object_reg))
```

```
  {
    csCommandLineHelper::Help (object_reg);
    return false;
  }

  // O relógio Virtual.
  vc = CS_QUERY_REGISTRY (object_reg, iVirtualClock);
  if (!vc)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Can't find the virtual clock!");
    return false;
  }

  // Encontra o ponteiro para o plugin engine.
  engine = CS_QUERY_REGISTRY (object_reg, iEngine);
  if (!engine)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iEngine plugin!");
    return false;
  }

  loader = CS_QUERY_REGISTRY (object_reg, iLoader);
  if (!loader)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iLoader plugin!");
    return false;
  }

  g3d = CS_QUERY_REGISTRY (object_reg, iGraphics3D);
  if (!g3d)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iGraphics3D plugin!");
    return false;
  }

  kbd = CS_QUERY_REGISTRY (object_reg, iKeyboardDriver);
  if (!kbd)
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "No iKeyboardDriver plugin!");
    return false;
  }

  // Abre o sistema principal. Isto abrirá todos os plug-ins carregados
anteriormente.
  if (!csInitializer::OpenApplication (object_reg))
  {
    csReport (object_reg, CS_REPORTER_SEVERITY_ERROR,
        "crystalspace.application.simpmap",
        "Error opening system!");
```

```
    return false;
  }

  view = new csView (engine, g3d);
  iGraphics2D* g2d = g3d->GetDriver2D ();
  view->SetRectangle (0, 0, g2d->GetWidth (), g2d->GetHeight ());

  if (!LoadMap ()) return false;
  return true;
}

void Simple::Start ()
{
  csDefaultRunLoop (object_reg);
}

/*---------------------------------------------------------------------*
 * Função Principal
 *---------------------------------------------------------------------*/
int main (int argc, char* argv[])
{
  simple = new Simple ();

  if (simple->Initialize (argc, argv))
    simple->Start ();

  delete simple;
  return 0;
}
```