Sistemas Gráficos Orientados à Aplicação

Tiago Stein D'Agostini

21 de Fevereiro de 2003

Resumo

Contrariando o ditado popular, uma imagem até pode não transmitir tanta informação como mil palavras, mas ela o faz de uma forma muito simples e representativa. Associado a esta idéia, é cada vez maior o número de sistemas computacionais que fazem uso de imagens ou modelos 3D para representar dados ou apresentar ao usuário uma interface convincente e de interpretação natural.

Para que um sistema computacional possa executar armazenamento, manipulação e visualização de informações de um sistema tri-dimensional, são necessários recursos especiais. O conjunto de recursos que permitem tais operações, quando providos por um artefato de software, definem o núcleo de um sistema gráfico, ou como é comumente conhecido, um *graphic engine*.

O núcleo de um sistema gráfico é um artefato de software de alta complexidade, com variabilidade muito grande de propósitos e capacidades entre as implementações. O desenvolvimento de um *graphic engine* é uma tarefa gigantesca que pode muitas vezes envolver a confecção de dezenas de milhares de linhas de código. A reusabilidade de artefatos tão complexos gera sérios problemas de flexibilidade e dimensionamento dos sistemas que os reutilizam. Apenas um *graphic engine* configurável pode ter sucesso na tarefa de garantir a reusabilidade deste tipo de artefato de software.

Este trabalho constitue um estudo e uma proposta de sistema para o desenvolvimento de *graphic engines* orientado às aplicações, utilizando de modernas técnicas de engenharia de software em grande parte baseadas no trabalho de Fröhlich [8] sobre sistemas operacionais orientados à aplicação. O sistema apresentado é um framework metaprogramado, cujo projeto foi baseado na decomposição do domínio em famílias de abstrações independentes de tecnologias, que pode ser utilizado para a construção de *graphic engines* com uma grande gama de possibilidades e recursos.

Palavras Chave: Gráficos, Application Oriented Design, Engenharia de Software, renderização, meta-programação estática

Abstract

Is a popular citation that an image can transmit more than a thousand words. Usually an image can not transmit as much information as a thousand words do, but it does in a much simpler and representative way. Associated to this ideia, each day the number of computer systems that make use of 3D models to represent any kind of data or just to present to the user a convincing interface is increasing.

Providing a system with storage, manipulation and visualization capabilities for 3D images makes room for a special class of resources. The set of resources that makes possible such operations, when provided by a software artifact, defines the core of a graphic system or graphic engine.

The core of a graphic system is a very complex software artifact, with many porpouses and levels of capacity. The development of a graphic engine is a huge task, that usually can involve the writting of many thousands code lines. The reusability of such complex artifacts creates serious problems of flexibility and scaling for the systems that re-use it. Only a configurable graphic engine can succeed in the task of garantee the reusability of this kind of software artifact.

This work is a study and a proposal of a system for graphic engines development in an Application Oriented way, in e very similar way that Fröhlich [8] does with Application Oriented Operating Systems. The presented system is a metaprogramed framework, whose project was based in ideas that are similar to family based design and independent of technology abstractions, so that it can be used to create graphic engines in a vast set of possibilities and resources.

Keywords: Graphics, Application Oriented Design, Software Engineering, rendering, static meta-programming

Conteúdo

1 Introdução **4**
- 1.1 Conceitos 4
- 1.2 Motivação e Objetivos 7

2 Paradigmas de Projeto **10**
- 2.1 O Início 10
- 2.2 O Modelo Dominante 11
- 2.3 Novas Alternativas 11

3 Integração de Componentes **15**
- 3.1 Interface dos componentes 15
  - 3.1.1 Derivação de Classe Abstrata 16
  - 3.1.2 Derivação protegida de Interface 17
- 3.2 *Design Paterns* 18
  - 3.2.1 *Strategy* 19
  - 3.2.2 Adaptadores 19
- 3.3 Meta-Programação Estática 20
  - 3.3.1 Programação genérica 21

4 SPLINE- *Scalable Platform Independent Engine* **23**
- 4.1 Fundamentos e Considerações de Implementação 23
  - 4.1.1 Mediadores de Tecnologia 24
- 4.2 Familias de Abstrações do SPLINE 25
  - 4.2.1 Família Render Context 25
  - 4.2.2 Família Render Device 28
  - 4.2.3 Família SceneGraph 30
  - 4.2.4 Família Transform 32
  - 4.2.5 Família PolygonConstructs 33
  - 4.2.6 Família Evaluators 34
  - 4.2.7 Família Light Source 35
  - 4.2.8 Família Material 36

4.2.9 Família Phisycs Device 37
4.2.10 Outros componentes 38
4.3 Projetos utilizando o SPLINE 38
4.3.1 Cyclops 3D 38
4.3.2 SNOW 40

5 Trabalhos correlatos **41**

6 Trabalhos futuros **43**

7 Conclusão **44**

Capítulo 1

Introdução

Este capítulo tem como objetivo estabelescer o contexto no qual o SPLINE está inserido, e o porquê de sua importância. Conceitos necessários para o entendimento do resto do trabalho são sumariamente apresentados a seguir.

1.1 Conceitos

Dentre todos os sentidos, a visão é aquele ao qual o homem mais fortemente está associado. A visão tambem é o meio principal do cérebro humano obter informações sobre o universo que nos é apresentado. Milhares de anos de evolução culminaram em um sistema altamente complexo, capaz de detectar nuâncias ínfimas e de coletar enormes quantidades de informação. A visão torna-se por tanto a escolha mais óbvia e natural quando o assunto é exibir informações complexas a partir de um sistema computacional. Apresentando imagens com alto grau de fidelidade podemos transmitir a impressão de estarmos em um ambiente ou diante de um objeto que só existe em forma de dados computacionais.

Visando alcançar estes objetivos desenvolveu-se a computação gráfica, uma sub área das ciências da computação que tem crescido em complexidade e capacidade de forma exponencial desde seu surgimento. A inerente complexidade de nosso sistema visual e de nossa capacidade de processarmos informações gráficas todavia causam enorme problemas para a computação gráfica. Para conseguir enganar o cérebro humano, os resultados obtidos através da computação gráfica precisam de um grau de precisão que é extremamente incomum na maioria dos sistemas computacionais.

Para suprir tal grau de complexidade, a computação gráfica buscou na matemática e na física modelos para representar, manipular e gerar visualizações de estruturas e mundos tri-dimensionais. Tais modelos foram mape-

ados em artefatos de software, que podem executar as tarefas necessárias à uma aplicação que utilize de gráficos tri-dimensionais. Os artefatos de software que são resultantes da composição destes modelos são o que chamamos de *graphic core* ou *graphic engine*.

A quantidade de tecnologias desenvolvidas na área da computação gráfica é muito grande e tende a crescer cada vez mais rápido. O enorme apelo comercial das tecnologias de renderização 3D e a disputa na área de gráficos para jogos tem feito com que uma corrida tecnológica se instaurasse entre as principais empresas desenvolvedoras de hadrware e software gráfico. Tornou-se comum que para resolver um mesmo problema, mais de uma linha tecnológica seja desenvolvida e apresentada, muitas vezes com incompatibilidade total entre elas. Diante de tal corrida tecnológica o desenvolvimento de GC tem se baseado na filosofia de *graphic engine* orientado à tecnologia. Isto significa que no desenvolvimento de um GC, o projeto tem se voltado a cobrir todos os aspectos disponíveis de uma tecnologia. Como exemplo podemos citar a indústria de jogos de computador e o surgimento do DirectX 8.0 [19], que disparou uma corrida por sistemas gráficos capazes de usar o que o DX8 oferecia, mesmo que tais recursos fossem inúteis para a maioria das aplicações na época.

Utilizar-se de um *graphic engine* projetado para uma outra aplicação, ou projetado sobre um grupo de tecnologias e não para suprir recursos, pode ter inúmeros efeitos negativos sobre o projeto e implementação de um sistema novo. O principal é a abordagem de projeto, na qual o sistema aplicativo se vê em necessidade de se moldar em função dos recursos providos pelo *graphic engine* e da forma com que este opera. Como resultado, as soluções encontradas nem sempre são as melhores possíveis, mas muitas vezes as que podem ser obtidas a partir do engine disponível. Muitas vezes as aplicações são obrigadas a conviver com efeitos colaterais de se utilizar um núcleo gráfico projetado para cobrir toda uma gama tecnológica muito maior que a necessária, tal como aumento da complexidade do sistema, dificuldade de manutenção e de portabilidade do sistema.

Um *graphic engine* customizável e com ênfase em atender as necessidades das aplicações seria uma solução ideal para a construção de sistemas gráficos. Tal abordagem constituiria em um desenvolvimento de núcleo gráfico orientado à aplicação. A programação modular baseada nas idéias de separação de conceitos e de agrupamento de similaridades (modernamente chamado de *family based design*) foi o início dos fundamentos necessários para se obter o grau necessário de escalabilidade no projeto de sistemas. Com o tempo perecebeou-se que os modelos de reusabilidade baseados em componentes funcionais não são suficientes para trazer o desenvolvimento de sistemas para um nível aceitável de flexibilidade e reusabilidade. A evolução natural do uso

de funções como unidade de re-usabilidade foi o paradigma da orientação a objeto, onde componentes de mais alta granularidade foram tomados como unidades de re-usabilidade e de abstração de elementos do sistema. Os conceitos da Orientação a objeto levaram a um novo nível da capacidade de modelagem e desenvolvimento de sistemas flexíveis e reusáveis, mas ainda não eram capazes de prover isso no nível necessário.

A evolução da Orientação a Objetos levou ao desenvolvimento de técnicas que visavam a construção de componentes de software especialmente projetados com fins de prover reusabilidade e intercambialidade. Seguindo a linha natural deste pensamento chegou-se ao conceito de um framework, onde os papéis deste scomponentes estão previamente estabelescidos e direcionados, facilitando a junção dos componentes em um sistema real.

A computação gráfica, bem como a maioria das áreas da computação extremamente dependentes de desempenho e de hardware, costuma ser extremamente resistente a introdução das modernas técnicas de projeto de sistemas. Até muito recentemente a idéia de construir um *graphic engine* utilizando-se *Object Oriented Design* era considerada extremamente revolucionária e muitas vezes impensável. A elaboração de frameworks ou qualquer outra técnica baseada em artefatos de mais alta granularidade era considerada viável apenas para os periféricos do sistema gráfico, mas não para seu núcleo.

A abordagem tradicional do desenvolvimento de *graphic engines* resultava em artefatos com escalabilidade extremamente limitada e com uma portabilidade entre sistemas muito complicada. Dentre as alternativas mais modernas, foi introduzida a idéia de *graphic engines* com suporte a *scripts*, nos quais componentes do sistema podiam ser selecionados pela simples modificação do *script*. A hierarquia interna do *engine* era montada em tempo de execução a partir da interpretação de *scripts*, proporcionando grande flexibilidade. Exemplos desta tecnologia são os *engines* UnrealWarfare [27] e NebulaDevice [16]. Os *scripts* provêem o sistema com um enorme grau de flexibilidade, que permite com que o engine tenha uma de re-usabilidade muito maior que outras abordagens tradicionais. A abordagem de *script*, entretanto, é usada apenas para permitir a configurabilidade periférica do *engine*. Isto significa que um sistema com uso de *scripts* desenvolvido para um jogo de computador em primeira pessoa, com visualização predominante de ambientes internos, poderá facilmente ser usado para outros sistemas semelhantes. Todavia o nível de configurabilidade deste tipo de abordagem não permite que as característica fundamentais do *graphic engine* sejam alteradas. Em termos simples isso significa que o *graphic engine* continua sendo inapropriado para a confecção de simuladores de vôo, os quais tem características fundamentalmente diferentes das dos jogos em primeira pessoa.

Continua a necessidade de *graphic engines* com configurabilidade a nível básico e a nível estrutural. Tal objetivo parece ser demasiado complicado para um sistema de *scripts*, mas mesmo assim a abordagem a nível de linguagem continuaria ser utilizável em um sistema cujas características fundamentais pudessem ser configuradas através de outras técnicas. Várias destas técnicas são discutidas neste trabalho, bem como uma maneira de utilizá-los efetivamente.

1.2 Motivação e Objetivos

A utilização de recursos de visualização 3D é cada vez mais comum em quase todas áreas da computação interativa, e a complexidade dos sistemas gráficos tem crescido de forma exponencial. A curva da complexidade dos sistemas gráficos resultou em uma situação peculiar em que poucos desenvolvedores tem a capacidade de manejar o desenvolvimento dos sistemas. O recente estouro de tecnologias da computação gráfica faz ser cada vez mais difícil manter o modelo de construção de *graphic engines* baseado em mapeamento dos recursos das tecnologias. Mesmo as melhores equipes de desenvolvedores de sistemas gráficos não foram capazes de completar a nova geração de *engines* entre a penúltima grande onda de tecnologias (baseada em *fragment shading*) e a nova onda que começa a surgir (baseada em *full pipeline programability*). A Insustentabilidade de tal sistema faz necessário o uso de uma abordagem de desenvolvimento independente da tecnologia de suporte.

O exemplo mais crítico desta situação está nas especificações de *pixel shaders*. Quando da introdução da programabilidade de *shaders* no nível de *pixels*, as empresas dominantes no mercado de hadrware gráfico não entraram em acordo sobre quais deveriam ser as especificações dessa tecnologia. Como resultado duas implementações completamente diferentes e incompatíveis entre si foram colocadas no mercado. Este fenômeno prejudicou os desenvolvedores de sistemas que agora precisavam desenvolver e programar para duas especificações completamente diferentes. Sob a perspectiva de que a nova geração de *pixel shaders* pudesse aumentar ainda mais os problemas de incompatibilidade, foram propostas especificações de linguagens de programação gráfica de alto nível, tal como CG (*C for graphics*) por parte da NVIDIA [15] e o OpenGL 2.0 pela 3DLabs [1].

Outro meio de conseguir o desenvolvimento de um *graphic engine* configurável e independente das tecnologias de suporte é com uso de modernas técnicas de engenharia de software para isolar todas as dependências de tecnologia em componentes de software intercambiáveis. Deste modo a dependência de tecnologias e de hardware passa pela mesma configurabilidade

que todo o resto do sistema.

Os *graphic engines* variam muito quanto a sua complexidade e especialmente quanto a capacidade de serem configurados ou reconfigurados para outros usos. Em um extremo nós encontramos os *engines* usados em benchmarks de equipamento de vídeo e de processadores. Este tipo de software é praticamente o único que não apresenta nenhum nível de configurabilidade, nem mesmo a capacidade de configurar que dados serão manuseados e renderizados pelo programa. Nenhuma utilidade além do teste de sistemas ou animações para telas de entrada podem ser encontradas para sistemas tão restritivos. A absoluta maioria dos *graphic engines* pode ser usada para operar dados diferenciados, trazendo um mínimo de utilidade real para este tipo de software. Usualmente características de *render* tais como resolução ou profundidade de cores podem ser configuradas, bem como brilho e contraste entre outros. Todavia a flexibilidade muitas vezes não passa deste ponto, não sendo possível escolher como os dados serão representados internamente, nem como eles serão operados. A impossibilidade de configurar que algorítmos ou estruturas de dados serão usadas impossibilita que o *engine* possa ser usado em um tipo de aplicação diferente mantendo um desempenho aceitável.

Os sistemas gráficos mais configuráveis podem ser controlados em tempo de execução pelo uso de comandos *script*. Alguns *engines* construídos desta forma apresentam um grau de flexibiliadde grande o suficiente para promover a escalabilidade, tal como o núcleo do sistema gráfico usado no software de modelagem 3D Maya da *Alyas-Wavefront*. Nestes casos todas as tarefas que o *graphic engine* executa podem ser configuradas e até mesmo novos recursos podem ser programados e incluídos na forma de *plug-ins* que na verdade são novos *scripts* para serem utilizados. Esta abordagem é bem sucedida a ponto de ter fortes representantes no mundo do CAD e Modelagem 3D bem como no mundo dos jogos de computador. O ponto fraco deste tipo de solução é que o *engine* continua sendo específico para uma aplicação ou tipo de aplicação específico, apenas com a possibilidade de modificações de capacidades, mas sem a possibilidade de configurar a estrutura interna do sistema. Para se construir um *engine* com suporte a *scripts* no mesmo nível que o Maya ou UnrealWarfare é necessário um trabalho dezenas de vezes maior do que para construção de um sistema gráfico convencional. Este trabalho todavia não pode ser aproveitado facilmente para construção de um engine com propósitos radicalmente diferentes. Deste modo, apesar de altamente configurável dentro do âmbito dos sistemas de modelagem 3D, o *graphic engine* do Maya nao pode ser usado de forma eficiente para desenvolver um jogo de computador.

O problema da falta de configurabilidade da estrutura e algorítmos principais do sistema fica mais visível na área dos softwares de uso especialista. To-

memos como exemplo softwares projetados especificamente para visualização de dados científicos em um determinado projeto ou sistema. A excentricidade deste tipo de software usualmente faz necessário o emprego de estruturas de dados e algorítmos específicos para o manuseio eficiente dos dados. Muitas vezes um mesmo projeto pode ter a necessidade de sistemas levemente ou moderadamente diferentes entre si para visualização de dados semelhantes. Neste caso o projeto de um grande *graphic engine* com capacidade de atender todas as necessidades deste pequenos sistemas torna-se inviável em termos de complexibilidade relativa a sua aplicabilidade. Em casos como este a solução tradicional são diversos pequenos engines específicos para as aplicações a que se destinam.

Aqui a configurabilidade através do re-uso de componentes do sistema torna-se imprencindível. Construir dezenas de pequenos *engines* quando a diferença entre eles pode ser restringida a uns poucos componentes intercambiáveis é muito mais fácil e eficiente que outras opções. Este tipo de facilidades só pode ser obtido quando a configurabilidade do sistema se extende ao nível algorítmico-estrutural. O nível de configurabilidade que se busca aqui é exatamente esse.

A principal tarefa na construção de um sistema com tais capacidades é a análise do domínio e identificação de famílias de abstrações independentes de tecnologias. As dependências de tecnologias encontradas no processo de decomposição são representadas por interfaces de famílias de tecnologias. As interfaces de famílias de tecnologias levam ao desenvolvimento de mediadores de tecnologia, que encapsulam funcionalidades de uma determinada tecnologia ou grupo de tecnologias que são usadas em uma ou mais abstrações. O trabalho aqui apresnetado resulta da adpatação de idéias de Application Oriented Programming, em especial derivadas do trabalho de Fröhlich [8], para o mundo da computação gráfica.

A demonstração das idéias deste trabalho sobre um *graphic engine* configurável, escalável e independente de plataforma é feita através da implementação do SPLINE (*Scalable Platform Independent Engine*). A idéia do SPLINE não é ser um simples *graphic engine*, mas sim um framework para a construção dos mesmos. O desenvolvimento do SPLINE também visa possibilitar a adoção do paradigma de programação orientada a aspectos, através do qual características comuns entre membros de várias famílias possam ser aplicadas a ambos sem replicação do esforço de desenvolvimento de código para tais características. Todavia a aplicação de aspectos com uso de ferramentas code weavers só será levada ao ponto de implementação em versão futura do SPLINE, por motivos a serem discutidos mais a frente neste trabalho.

Capítulo 2

Paradigmas de Projeto

Neste capítulo será apresentada uma análise dos principais paradigmas de projeto de sistemas desde os já consagrados, pasando pelos modelos dominates da atualidade e concluíndo com uma breve passagem pelos novos paradigmas. Sempre mantento o foco no problema da reusabilidade e configurabilidade de sistemas.

2.1 O Início

Construir um sistema, qualquer que seja seu tipo, visando obter re-usabilidade e configurabilidade cria a necessidade de técnicas avançadas de análise e de projeto. Variações dos paradigmas de desenho tradicionais surgiram para suprir a necessidade deste tipo de técnicas. Algumas destas técnicas já são usadas de forma corriqueira no desenvolvimento de projetos, enquanto outras mais recentes são pouco usadas fora de meios acadêmicos.

Basicamente a idéia principal por trás da maioria das técnicas modernas de desenho é a separação de conceitos. Separando conceitos, cria-se unidades claras e razoavelmente estanques de reusabilidade. Tal conceito não é novo, tendo sido introduzido pela primeira vez por Dijkstra [4]. Ao redor deste conceito estabelesceu-se o conceito de divisão de domínio em famílias [23], que constituiriam unidades nas quais as similaridades entre seus membros são maiores ou mais relevantes que suas diferenças. Como extensão desta idéia vem o *Incremental System Design*, que introduz a hierarquia de múltiplos níveis ao modelo de famílias. Como exemplo da abordagem em um sistema gráfico, podemos especular sobre a organização de uma família de *RenderDevices*. Todos os componentes de software compartilhando as responsabilidades e funcionalidades de desenho são agrupados nesta família. Todavia nem todos *RenderDevice* são iguais, apesar de suas semelhanças serem mui-

tos grandes, e a família pode se subdividir em sistemas que seguem o modelo natural da luz e aqueles que utilizam de técnicas de projeções e álgebra linear para conseguir os mesmos efeitos. Poderíamos assim fatorar a família em componentes *light based RenderDevice* e*algebric based RenderDevice*. Seguindo o conceito de *Incremental System Design*, poderíamos adicionar um novo nível hierárquico à *light based RenderDevice* denominado *ray-tracing* que apesar de manter a mesma política, implementa mais funcionadidades. Esse ramo poderia ser incrementado por muitos níveis pela inclusão de *Radiosity* e muitos outros tipos mais sofisticados de aproximação do comportamento da luz. Importante ressaltar que *Family Based Design* e *Incremental System Design* são abordagens puramente funcionais, anteriores ao conceito de classes muito usado hoje. Em muitos pontos deste trabalho as referências a famílias de componentes dizem respeito a uma abordagem semelhante a *Incremental System Design*, mas adaptada para aplicação sobre classes como componente básico.

2.2 O Modelo Dominante

O paradigma que atualmente pode ser dito dominante é o *Object Oriented Programming*, no qual objetos são resultados da decomposição do problema. As similaridades são expressas na forma de classes, as quais englobam objetos com responsabilidades semelhantes. Deste modo dois *triangles meshes* podem pertencer a uma mesma classe, mesmo sendo diferentes entre si em cada triângulo ou vértice. As responsabilidades e o comportamento continuam sendo os mesmos. Todavia existe a possibilidade de diferenças serem adicionadas a estas abstrações, resultando em sub-classes. Esta situação pode ser exemplificada pela mesma classe *triangle mesh*, que pode ter adicionado à mesma a capacidade de texturização na forma de uma sub-classe.

Diferentemente de abordagens anteriores a Orientação a Objetos explicita a representação e as relações entre o nível lógico estático e o nível dinâmico. Tal relação pode ser facilmente observada entre as classes e os objetos pertencentes as mesmas. Outro tipo de visão propiciada pela Orientação a Objetos é a do processo de interação entre as entidades tanto no nível estático (colaboração entre as classes) como no nível dinâmico (fluxo das mensagens entre os objetos). A capacidade de representar explicitamente tanto o nível estático como o dinâmico aumenta em muito a expressividade desta técnica.

2.3 Novas Alternativas

Com a evolução da engenharia de software novas abordagens surgiram, dentre elas *Application Oriented Design*. Esta abordagem, que constitui a base sobre o qual este trabalho se desenvolve, se baseia em idéias muito próximas ao modelo de black box framework, livrando-se de limitações do modelo white box framework da Orientação a Objetos pura. Pra fazer uso de paradigmas como este, faz-se necessário o uso de recursos especiais para confecção dos artefatos de software, do mesmo modo que *Object Oriented Programming* faz-se uso de classes. Um exemplo de recurso que pode ser usado neste paradigma é a exploração da configurabilidade na geração do código do sistema. Normalmente isto necessita do uso de ferramentas especiais, embutidas ou não na linguagem de implementação. Os exemplos mais evidentes deste tipo de configurabilidade são representados pela meta programação estática provida pelos *templates* de C++ e pelo sistema de *weaver* de aspectos do AspectC++ [29] e de outras linguagens como AspectJ [13].

O sistema SPLINE que foi desenvolvido no curso da elaboração deste trabalho faz uso extensivo de *templates* e *template functions* para construir um sistema com alta configurabilidade de componentes e código em tempo de compilação. A figura 2.1 mostra como uma classe *Color* pode ser configurada de acordo com o tipo de dado bem como na precisao do mesmo. O uso correto de *templates* pode gerar código com altíssimo desacoplamento entre componentes e cuja compilação gera código compacto e limpo para o sistema resultante. A especialização de templates também permite que determinadas configurações dos componentes tenham implementações radicalmente diferentes sem interfirir com a construção e utilização de casos mais genéricos do mesmo.

Um conjunto de técnicas ainda longe da evidência hoje desfrutada pela Orientação a Objetos, mas que adiciona muito à engenharia de software é *Aspect Oriented Programming*. Estas idéias visam tratar de propriedades não funcionais na decomposição de domínios que também podem ser chamadas *crosscutting concerns*. Introduzida inicialmente por Kiczales [13], já pode ser usada através do emprego de inúmeras ferramentas e tende a crescer muito em importância no projeto de todo tipo de sistemas.

Como o nome indica, *Aspect Oriented Programming* preucupa-se na identificação de aspectos como unidades de re-usabilidade. Os apectos são abstrações de propriedades identificadas na decompisição do domínio mas que não se restringem a uma única abstração ou ramo de família de abstrações. A implementação deste tipo de características através de aspectos evita que exista a replicação da descrição e da implementação de funcionalidades. Isto é conseguido pela descrição e implementação dos aspectos no formato de com-

```
template<typename TYPR_COLOR>class Color{
public:
     Color(){};
     ~Color(){};
     TYPE_COLOR *getData(){return data;}
     void setData(TYPE_COLOR* nd){
       data[0]=nd[0];data[1]=nd[1];
       data[2]=nd[2];data[3]=nd[3];
     }
     TYPE_COLOR oparator[](int par){
        return data[par];
     }
     void setComponent(int32 index,TYPE_COLOR cc)
         data[index]=cc;
     }
protected:
     TYPE_COLOR data[4];
};
```

Figura 2.1: Exemplo de classe parametrizada

ponente que passa por um processo de costura, ou em iglês *weaving*, no código ao qual será aplicado. A aplicação dos aspectos pode ser feita de forma manual, mas tal prática limita em muito a usabilidade desta técnica, pois todo o ganho na eliminação de redundância de implementação é compensado pelo processo de combinação. Para resolver esse problema são desenvolvidas ferramentas chamadas *weavers*, as quais tem como função analisar o código do componente original e o código do aspecto, aplicando o último ao primeiro. A aplicação de aspectos é feita sobre pontos específicos do componente original conhecidos como *join points*. A figura 2.2 mostra a descrição de um aspecto na linguagem AspectC++ [29]. Dependendo da linguagem ou modelo de descrição de aspectos adotados, os *join points* podem ser chamadas de funções ou a própria estrutura do componente.

O uso não de uma, mas de todas estas técnicas e paradigmas de programação e projeto permitem que os sistemas sejam projetados e implementados de forma a obter componentes com alto desacoplamento e capacidade de reusabilidade. Utilizando-se de tais componentes sobre uma estrutura de framework, podemos compor ou alterar *graphic engines* de forma rápida e eficiente. A expressividade deste framework, no que diz respeito a cobrir toda gama de possíveis categorias de *graphic engines*, só é limitada pela construção de novas abstrações e entidades. Para que a escalabilidade seja uma característica real no framework é necessário que as dependênias entre com-

```
aspect SimpleAspect {

advice execution(" % A::%(...)") ||
       execution(" % b(...)") : void before(){
   printf("This printing is an aspect. \n");
   printf("It is executed before methods of class A \n");
   printf("and before any method named b");
}

}
```

Figura 2.2: Exemplo de aspecto descrito em AspectC++

ponentes sejam mínimas ou se possível inexistentes. Ver-se-á a seguir que o SPLINE é uma implementação visando a escalabilidade e total independência de plataforma. Ambas a escalabilidade e a independência de plataforma são obtidas através do emprego de técnicas baseadas no que aqui foi discutido, principalmente a meta-programação estática.

Capítulo 3

Integração de Componentes

Criar componentes não é uma questão de simplesmente separar conceitos e escrever código em unidades separadas, pelo menos não se o objetivo é o de que os componentes possam ser empregados para construir um sistema que funcione. Para tanto é imprecindível que os componentes possam interagir entre si sem que os conceitos por eles representados sejam distorcidos no processo de composição do sistema. Este capítulo objetiva dispor de uma rápida análise de conceitos e técnicas usáveis com esse objetivo.

3.1 Interface dos componentes

De um modo geral a boa prática de programação diz que um componente só poderá ser acessado através da interface definida para o mesmo. A interface de um componente é o conjunto de métodos que este disponibiliza publicamente para outros componentes. Eventualmente o conceito de interface de um componente também pode englobar as operações que podem ser efetuadas sobre o componente. Baseado em que toda a iteração com um componente tem que seguir o que é estabelescido na interface, pode-se dizer que esta é a pedra fundamental da integração de componentes em um sistema concreto. A interface, apesar de fundamental, não é sempre suficiente para garantir que os componentes interajam da forma com que deveriam.

Definir a interface dos componentes de determinadas famílias de abstrações é apenas a primeira parte. É necessário que as interfaces sejam cumpridas, e pra tanto o melhor modo é o uso de recursos providos pela linguagem com este fim. Algumas linguagens como Java implementam diretamente a construção *Interface*, enquanto outras como C++ podem misturar este conceito com o de herança. É verdade que a herança é um método não só de re-uso de código como também é um meio de exportar uma interface,

mas o conceito de interface não é necessariamente ligado à herança. Como a linguagem C++ apresenta as características ideais para o desenvolvimento de sistemas de alto desempenho como os requiridos pela computação gráfica, apresenta-se aqui uma rápida explanação sobre como conseguir contornar este problema. Usando a linguagem C++ existem algumas possibilidades para definir interfaces, talvez outras além das aqui demonstradas existam, mas o objetivo aqui é apenas um esclarescimento breve.

3.1.1 Derivação de Classe Abstrata

Uma maneira de representar e garantir o cumprimento de interfaces em C++ é através da definição da interface como uma classe abstrata. Uma classe puramente abstrata tem todos seus métodos sem implementação como na figura 3.1. Como recurso automático de linguagem o compilador não permitirá a instanciação desta classe. Membros da família definida por esta interface herdariam publicamente a mesma. Novamente por força do mecanismo da linguagem C++, será exigida a implementação dos métodos definidos na interface. A figura 3.1 também mostra uma classe que tem sua interface definida através deste mecanismo de herança pública.

```
class Interface {

public:

  virtual void method1()=0;
  virtual void method2(int i)=0;

};

class Realizacao:public Interface{

public:

  void method1(){/*...*/}
  void method2(){/*...*/}

};
```

Figura 3.1: Exemplo de definição de interface por classe abstrata

O uso deste tipo de definição de interface traz algumas desvantagens. Dentre elas está a natural incorrência do conceito de sub tipo entre a classe mãe e a classe filha. O conceito de que toda herança gera uma relação de

especialização de tipagem nem sempre é desejada por não representar a realidade do domínio que deseja-se descrever. Uma interface em seu conceito básico não é um tipo, portanto não pode existir relação de sub tipo com outras entidades. Outra grande desvantagem do uso de classes abstratas como mecanismo de definição de interface é a necessidade de todos os métodos sofrerem ligação tardia. A ligação tardia permite que o polimorfismo seja explorado o qual por si só é uma ferramenta que pode ser usada na integração dos componentes. O uso de ligação tardia certamente apresenta vantagens em determinadas cituações, mas exige um sistema de chamada de funções mais complexo e associado a uma perda de desemepenho. A perda de desempenho na maioria das vezes é plenamente justificável, como quando o sistema deve realmente apresentar um comportamento polimórfico para facilitar o projeto ou implementação. Por outro lado, os casos onde a ligação dinâmica é realmente necessária são muito mais raros que os casos onde a ligação estática é suficiente, fazendo com que o uso de ligação dinâmica para todas as funções incorra em disperdício de desempenho.

3.1.2 Derivação protegida de Interface

Resolver o problema da representação de interfaces e posterior verificação da adequação à mesma pode ser feito em C++ através de uma abordagem pouco utilizada por usar recursos inexistentes em muitas outras linguagens. A definição de uma classe como interface faz necessário que os métodos da mesma não possuam implementação, caso contrário teríamos uma classe e não uma interface. Uma maneira simples de conseguir isso é não implementar os métodos. A assinatura dos métodos ainda está presente na declaração da classe, mas nenhuma implementação é associada com ela. Para que este tipo de prática não resulte em erros devido a eventual tentativa de chamar um método desta interface, os construtores da classe são declarados como membros protegidos, o que impede qualquer instanciação da mesma. A figura 3.2 mostra uma Interface definida deste modo, bem como uma classe que segue a implementação desta mesma interface. Note-se que a herança utilizada é do tipo protegida o que permite que a interface e a implementação sejam herdadas sem incorrer na relação de sub-tipo. O mecanismo de herança protegida não está presente em muitas linguagens além de C++, o que pode fazer este conceito parecer estranho para desenvolvedores habituados com outras linguagens.

Como maior vantagem desta implementação temos a possibilidade de manter a ligação estática dos métodos. A resolução das ligações em tempo de compilação pode resultar em perceptíveis diferenças de desemepenho dos sistemas. Alem desta vantagem, temos a já citada ausência da relaçãode sub

```
class Interface {

protected:

  Interface(){};

public:

  void method1();
  void method2(int i);

};

class Realizacao: protected Interface{

public:

  Realizacao();
  void method1(){/*...*/};
  void method2(int i){/*...*/}

};
```

Figura 3.2: Exemplo de implementação de interface com herança protegida

tipo entre interface e realizador da interface.

Importante ressaltar que no protótipo apresentado mais a frente neste trabalho, o uso de derivação protegida de interface é um meio de inferir a corretude das realizações das interfaces necessárias. Não é de modo algum necessário que um componente faça uso de qualquer método de herança de interface para ser adaptado ao framework, apenas o uso de tais métodos permite uma garantia do cumprimento desta.

3.2 *Design Paterns*

Repetidamente um projetista de sistemas orietados a Objetos encontra-se com situação de problemas tradicionais e já solucionados muitas vezes. Várias das soluções empregáveis para resolver estes problemas tornaram-se padrões e são listados como *Design Patterns*. Dentre os vários catálogos de *design patterns*, um dos mais aceitos é aquele proposto por Gamma [9]. A utilização destes padrões de forma adequada pode facilitar em muito a construção de componentes com alto desacoplamento, bem como melhorar a compatibilidade dos mesmos dentro de um framework.

Alguns dos padrões descritos por Gamma foram utilizados, muitas vezes com pequenas modificações, no desenvolvimento do SPLINE. É importante ressaltar que a utilização dos padrões no SPLINE foi feita com adaptações a fim de melhor aproveitar a ampla utilização de templates que existe no projeto, portanto as implementações de exemplo de Gamma não correspondem as implementações no SPLINE. Alguns dos padrões merecem uma descrição nesta seção para facilitar o posterior entendimento de todo o projeto SPLINE.

3.2.1 *Strategy*

Este é um *design patttern* que permite que algoritmos sejam encapsulados e agrupados em uma família de modo que mebros desta família sejam intercambiáveis. O uso do padrão *strategy* faz com que seja possível a configurabilidade do sistema a nível de algorítmos, permitindo-os variar independentemente dos clientes que o usam. A implementação de uma solução similar está descrita em Stroustrup [26] com o uso de templates, no caso para a escolha de um algorítmo de ordenação.

O uso do *design pattern Strategy* permite um nível adicional de reusabilidade tanto para os algorítmos como para as entidades que os usam. Como exemplo podemos tomar uma classe de grafo de cena. Não raramente é necessário extrair os componentes de um grafo de cena na forma de uma lista ordenada pela profundidade dos elementos. Alguns tipos de grafos de cena devem ser renderizados *back-to-front*, enquanto outros devem ser renderizados em *front-to-back*. Construindo este componente com um parâmetro de template *Sort* que pode ser completado por uma implementação específica de ordenação, teremos construído um *Strategy*. Ao completarmos o parâmetro de template com um componente *SortUp* conseguimos uma organização em ordem crescente de profundidade enquanto ao usarmos *SortDown* conseguimos uma ordem decrescente. Os compoennte usados para completar o template são independentes deste uso em específico e podem ser re-usados para implementar ordenação em qualquer outro ponto do framework onde isto seja necessário.

Outros usos bastante óbvios de *strategies* são na determinação dos algorítmos de *culling* e detecção de colisão. Através deste método podemos fazer um controle fino sobre a relação eficiência por desempenho destes algorítmos, o que é um problema comum na confecção de engines.

3.2.2 Adaptadores

Adaptadores são utilizados para compatibilizar a interface de componentes para com outra interface. Usualmente são conhecidoscomo *wrappers*. Adap-

tadores interceptam mensagens destinadas à entidade adaptada mas que não possui uma interface adequada para tratar diretamente essas mensagens e as traduzem para termos da interface apresentada por esta entidade. Adaptadores permitem que um componente seja visto por outros como um componente construído para à integração com os mesmos, mesmo que isso não seja uma verdade. A implementação deste *Design Pattern* pode ser feita de modos diferentes, visando adaptar toda uma hierarquia de objetos para a interface de outra hierarquia ou apenas para um único componente. A figura 3.3 retirada diretamente de Gamma [9] demonstra o esquema usual de um adaptador. No SPLINE as adaptações são feitas entre interfaces de famílias de componentes por meio de adaptadores metaprogramados.

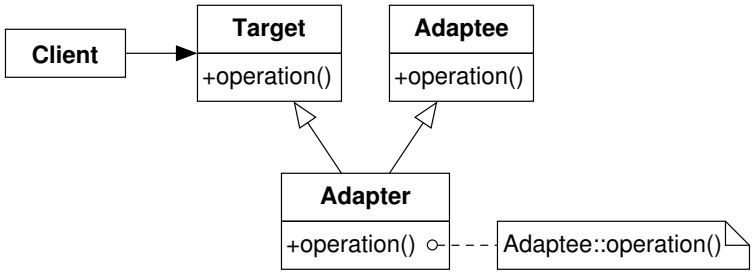

Figura 3.3: Esquema de um adaptador

3.3 Meta-Programação Estática

Metaprogramação é baseada na idéia de programas atuando sobre programas, sejam outros ou eles mesmos. A metaprogramação pode ocorrer sobre o código do programa antes que este seja rodado. Neste caso a metaprogramação é dita estática. Qualquer preprocessamento sobre um programa que altere seu código antes da geraçào do binário final é um metaprograma. O nível de flexibilidade e expressividade dos esquemas de meta programação determinam o quanto estes são poderosos.

A linguagem C++ [26], sobre a qual a implementação deste trabalho foi feita, exibe um mecanismo de templates bastante complexos para suprir essa funcionalidade. O mecanismo de templates de C++ pode ser usado para parametrizar tipos ou valores de constante utilizados em classes ou funções. A figura 3.4 demonstra como um componente, no exemplo responsável pela renderização de buffers de vértices, pode ter seu comportamento resolvido estaticamente. Extendendo a já enorme capacidade que essas características dão ao C++ existe o mecansimo de especialização de templates. Este mecanismo permite que para determinado parâmetros o template tenha uma

implementação totalmente diferente. A figura 3.5 mostra como a especialização de templates pode ser usada para mudar a implementação padrão. No caso o mapeamento de textura com funções tradicionais do OpenGL foi substituída por uma implementação sobre as extensões do OpenGL. O mesmo tipo de template pode ser usado para configurar uma aplicação para um determinado hardware de vídeo sem alterar nada mais que um parâmetro do template.

```
enum{GL_VERTEX_ARRAY,ATI_VERTEX_ARRAY_OBJECT,NV_VERTEX_ARRAY};

template<int n> struct Packsize{enum{RET=0;};}
template<> struct Packsize<GL_C3F_V3F>{enum{RET=6};};
template<> struct Packsize<GL_V3F>{enum{RET=3};};

template<int type, int vertexFormat>
class VertexArray{

protected:
  float *arrayStart;

public:

  void prepareBuffer(int nbrVertex){
     arrayStart= new float(nbrVertex*packsize(vertexFormat));
  }

  void render(int nbrVertex){
    glInterleavedArray(vertexFormat,0,arrayStart);
    glDrawArrays(GL_TRIANGLES,nbrVertex);
  }

  /*...*/

};
```

Figura 3.4: Exemplo de componente metaprogramado

Utilizando-se dos mecanismos de template, muitas questões estruturais que normalmente seriam resolvidas por herança pública podem ser resolvidas de forma estática e livre do acoplamento proveniente da relação de herança. Outra capacidade muito interessante provida pelos templates são as *function templates*, as quais permitem que dentre outras coisas os tipos dos parâmetros sejam configuráveis, não necessitando que a unidade onde a função é descrita tenha acesso a descrição do tipo que porventura será usado como parâmetro. Novamente o uso desta funcionalidade permite diminuir as dependências entre componentes. Este conceito foi amplamente usado nas abstrações de grafo de cena do SPLINE.

```
template<>
class vertexArray<ATI_VERTEX_OBJECT,GL_T2F_N3F_V3F>{
protected:
    float *arrayStart;
    unsigned int idObject, *objectSize;

public:
 void prepareBuffer(int nbrVertex){
   arrayStart= new float(nbrVertex*Packsize(vertexFormnat));
 idObject=glNewObjectBufferATI(objectSize,verts,GL_STATIC_ATI)
 }

 void render(int nbrVertex){glEnableClientState(GL_VERTEXA_ARI
   glEnableClientState(GL_TEXTURE_COORD_ARRAY);
   glEnableClientState(GL_NORMAL_ARRAY);
   glEnableClientState(GL_ELEMENT_ARRAY_ATI);
   glArrayObjectATI(GL_TEXTURE_COORD_ARRAY,2,GL_FLOAT,
       sizeof(vertex_t),idObject,0);
   glArrayObjectATI(GL_NORMAL_ARRAY,3,GL_FLOAT,sizeof(vertex_t
       idObject,sizeof(float)*2);
   glArrayObjectATI(GL_VERTEX_ARRAY,3,GL_FLOAT,sizeof(vertex_t
       idObject,sizeof(flaot)*5);
   glArrayObjectATI(GL_ELEMENT_ARRAY_ATI,1,GL_UNSIGNED_INT,
       0,glElementObject,0);
      glDarwElementArrayATI(GL_TRIANGLES,numTriangles*3);
   glDisableClientState(GL_VERTEX_ARRAY);
   glDisableClientState(GL_NORMAL_ARRAY);
   glDisableClientState(GL_TEXTURE_COORD_ARRAY);
   glDisableClientState(GL_ELEMENT_ARRAY_ATI);
}
/*..*/
};
```

Figura 3.5: Exemplo de template especializado

3.3.1 Programação genérica

É muito comum que implementações de algorítmos e estruturas de dados tenham seu reaproveitamento diminuido ou eliminado devido a tipagem. Programação genérica trata de alcançar altos níveis de re-usabilidade por meio do uso de parâmetros de tipo nas implementações. O exemplo mais conhecido deste tipo de implementação é a *Standard Template Libray* originalmente disponibilizada pela *Silicon Graphics* [25]. O mesmo princípio que faz com que a programação genérica seja tão interessante para a *Standard Template Library*, permite que componentes de diversos níveis de complexidade sejam implementados visando alta reusabilidade e intercambialidade.

Muitos dos *design patterns* podem ser incrementados ou implementados

mais facilmente pelo uso de templates e programação genérica. Isto é especialmente verdadeiro nos *design patterns* que objetivam a intercambiabilidade entre componentes.

Capítulo 4

SPLINE- *Scalable Platform Independent Engine*

Este capítulo trata da implementação protótipo de um framework para construção de sistemas gráficos orientados à aplicação. A denominação SPLINE é um acrônimo para *Scalable Platform Independent Engine*. Após a explicação dos fundamentos e considerações de implementação sobre o SPLINE, segue-se uam análise mais detalhada de cada família implementada. Ao final uma perspectiva de continuação do desenvolvimento do SPLINE e um *road map* do futuro próximo do mesmo.

4.1 Fundamentos e Considerações de Implementação

Ao longo deste trabalho inúmeras técnicas e conceitos foram discutidos sob a ótica do projeto e implementação de sistemas gráficos altamente configuráveis e orientados à aplicação. Pro projeto SPLINE alguns destes conceitos foram selecionados e aplicados, e para entender esta seleção é necessário uma explanação sobre os fndamentos e objetivos do projeto.

Acima de tudo o SPLINE é um framework visando a construção de *graphic engines* configuráveis e voltados à aplicação ao qual se destinam. A opção pela configurabilidade estática foi feita por ser a que permite melhor desempenho e melhor se adequar ao desenvolvimento de pequenos sistemas tais como os de suporte a aplicações científicas, os quais não justificam a adoção de um grande *graphic engine* complexo. Para suportar o mecanismo de configurabilidade estática a absoluta maioria dos componentes foram implementados como templates, muitas vezes apresentando uma interface na forma de *function templates*.

Novamente é importante ressaltar que o termo família aqui usado tem um significado ligeiramente diferente do inicialmente proposto em *Family Based Design*. Aqui o termo família é usado para catalogar grupos de componentes na forma de classes, enquanto em *Family Based Design* a unidade básica é funcional.

As famílias são representadas por uma interface realizada por herança protegida. A interface é inflada para acomodar a união de todas interfaces dos membros da família, tal como no sistema operacional EPOS [8]. A utilização de uma interface inflada aumenta a compatibilidade intra-componentes de uma mesma família.

4.1.1 Mediadores de Tecnologia

Todas as abstrações foram modeladas de forma a excluir aspectos dependentes de tecnologias de suporte, tais como sistema de janela, sistema operacional, hardware gráfico dedicado, API gráfica ou mesmo processamento vetorial de dados. As abstrações que necessitam de recursos provenientes deste tipo de tecnologia fazem uso de componentes que seguem uma interface de família de tecnologia. As interfaces das famílias de tecnologia são implementadas de forma diretamente dependente da tecnologia por meio de mediadores de tecnologia. No proceso de realização final do template da abstração, passa-se um parâmetro determinando que mediador de tecnologia será utilizado pela abstração. Mediadores de tecnologia podem ser repositórios de algorítmos, implementações específicas para determinado hardware ou adaptadores para a interface do sistema operacional. Esta abordagem é em muito semelhante ao *design pattern strategy*, mas é mais ampla por permitir que componentes mais complexos que simples algorítmos sejam adaptados.

A figura 4.1 mostra a esquematização de abstrações e mediadores, enquanto a figura 4.2 mostra um exemplo de mediadores assumindo seus papéis junto a abstrações. No exemplo, temos a presença de um membro da família *RenderDevice* e um membro da família *RenderWindow*. Ambos possuem interfaces que permitem que trabalhem em conjunto como é demonstrado pela linha divisória entre as abstrações. Este é um exemplo onde as abstrações precisam de recursos especiais para realizar suas tarefas, característica representada pela presença de um *slot* para mediadores de tecnologia. O *RenderDevice* necessita dos serviços de uma API para realizar qualquer renderização. Da mesma forma o *RenderWindow* precisa de um mediador de sistema de janela para acessar as funções que permitam que a exibição dos gráficos chegue à janela adequada. Em ambos os casos apenas um mediador cumpridor da interface é necessário para que o sistema seja instanciado, mas existe opcão entre qual mediador cumprirá este papel. No caso das APIs 3D um media-

dor de OpenGL ou de Direct3D poderia ser escolhido, enquanto no sistema de janelas poderia-se usar mediadores para X-Window system ou qualquer outro sistema de janelas.

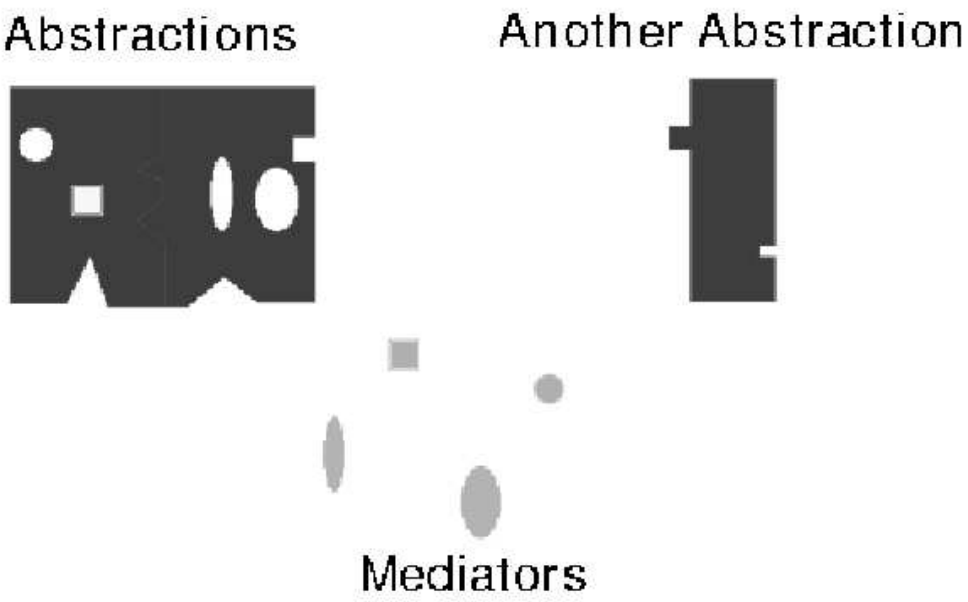

Figura 4.1: Esquematização de mediadores e abstrações

Importante ressaltar que através da troca de poucos mediadores um *graphic engine* implementado inicialmente para linux pode ser portado para Windows com muita facilidade. O encapsulamento de todas as funcionalidades dependentes de uma tecnologia em um único mediador é a melhor forma de facilitar o processo de configuração e reutilização de sistemas .

4.2 Familias de Abstrações do SPLINE

Durante o processo de desenvolvimento do SPLINE o domínio dos *graphic engines* foi fatorado em um conjunto de abstrações, levando em consideração as técnicas e recursos mais usados na computação gráfica. O grande desafio foi estabelecer interfaces para as famílias de abstrações de modo que estas permanecessem independentes e intercambiáveis entre sistemas gráficos radicalmente diferentes. Como já foi discutido, as abstrações foram estabelecidas de forma a manter as dependências de hadrware, API e sistema operacional desacopladas das mesmas.

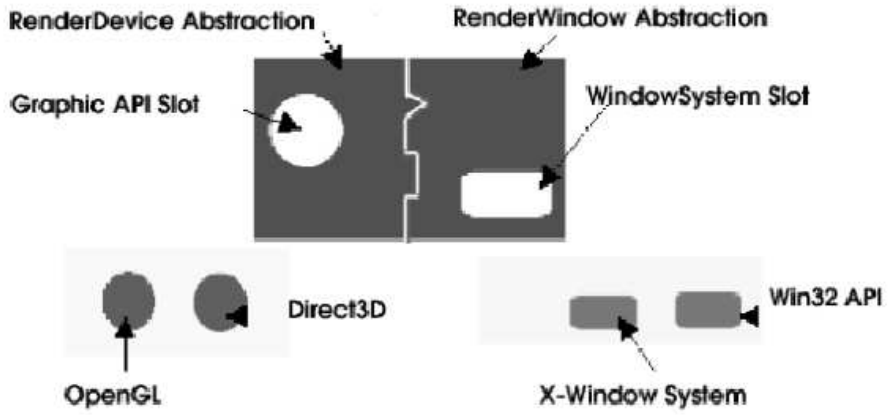

Figura 4.2: Exemplo esquemático de mediadores e abstrações

4.2.1 Família Render Context

Qualquer sistema de computação gráfica tem como objetivo principal a construção e apresentação de imagens. Estas imagens entretanto precisam de um lugar onde serão exibidas ou armazenadas, podendo ser o *frame buffer*, uma textura ou um arquivo entre outras opções. Em outras palavras, é necessário um contexto de renderizaçãopara que o dispositivo de renderizaçào use como alvo. No SPLINE todos elementos relacionados com manuseio e criação de contextos de renderização são gerenciados por membros da família *Rendering Context*. Na figura 4.3 está representada a composição da família em questão como no momento da escrita deste documento.

É necessário ressaltar que esta família possui dois ramos principais, um dos quais representa o contexto de renderização propriamente dito e outro que representa a entidade capaz de criar e disponibilizar contextos de renderização. Para que isto fique mais claro tomemos como exemplo um sistema que executa a renderização sobre uma janela. A janela é uma instanciação de um contexto de renderização, mais precisamente de um *windowDescriptor*. Para criarmos uma janela em um sistema oparacional precisamos antes ter o controle sobre o conjunto de janelas e a capacidade para criá-las. Isto é responsabilidade de um *RenderContextManager*, ou mais precisamente neste caso um *windowManager*. O *windowManager* é uma abstração que depende de recursos de sistema, portanto para ser realizada, necessita de um *windowMediator* que da aceso as funções do sistema de janelas, e de um *windowSystemDescriptor* que guarda quaisquer informações necessárias sobre o sistema de janelas.

No caso de uma renderização direcionada para um arquivo, o *Render Context* seria um *file descriptor* e o seu gerente correspondente seria uma abstração para controlar os arquivos através de mediadores para o sistema operacional (se necessário).

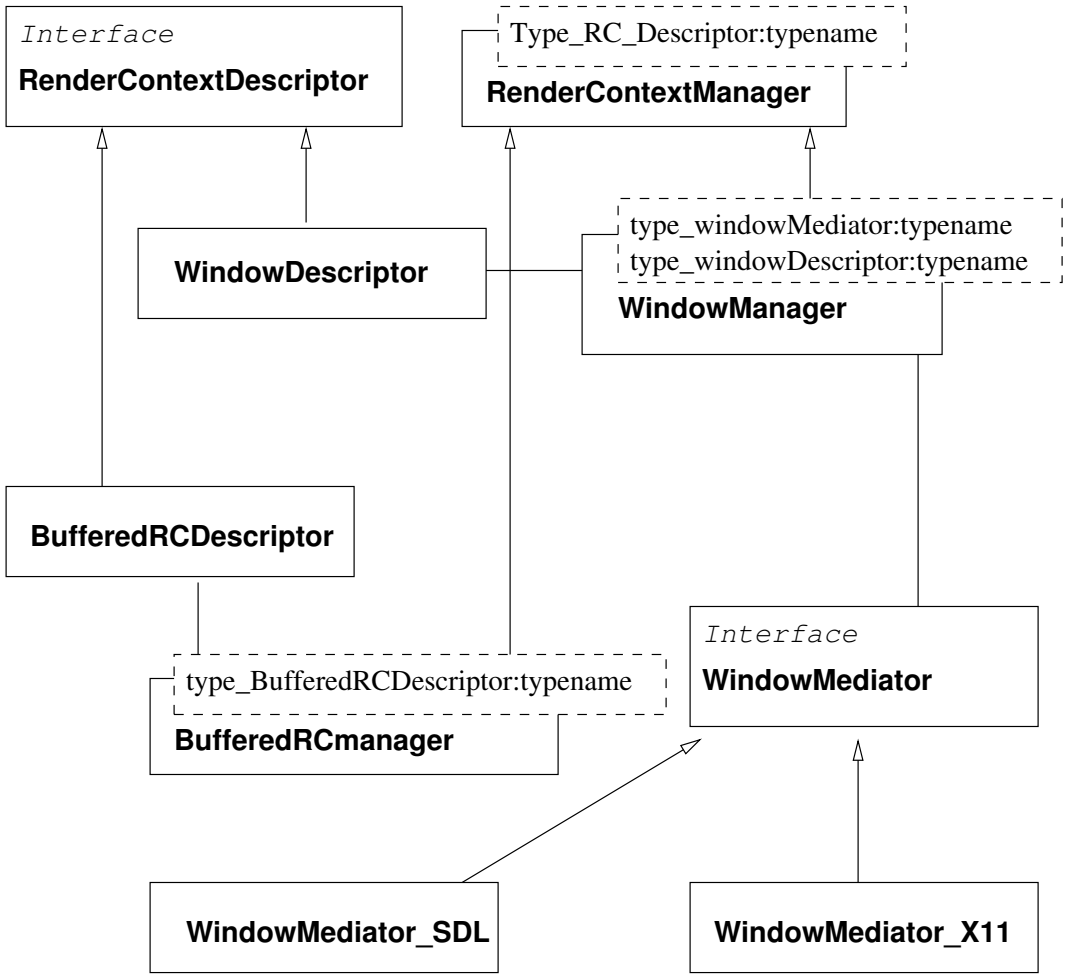

Figura 4.3: Composição da Família RenderContext

A interface desta família é bastante simples, apresentando métodos para acesso a informações tal como profundidade de cores, precisão de depth buffer, presença ou não de stencil buffer e outras informações pertinentes ao formato de pixel.

No estágio atual de implementação o SPLINE possui abstrações para renderização em janelas e em texturas, bem como os managers adequados para estes contextos de renderização. Dentre os mediadores, já foram implementados os mediadores de SDL e GLUT [14].

A família *RenderContext* está entre as famílias mais simples do SPLINE, mas dentre as mais importantes para permitir portabilidade e configurabilidade do sistema. O ponto de conexão de membros desta família com outros membros do framework é junto ao *Render Device* como um parâmetro de template. Deste modo cada *render device* está associado a um *render context*.

4.2.2 Família Render Device

A importancia dos membros dessa famíla é grande ao ponto de não precisar ser discutida em detalhes, bastando deixar claro que todas as funções que geral algum resultado na forma de imagens são responsabilidade dos membros desta família. Estas responsabilidades entretanto são muito grandes, o que faz desta família um exemplo muito particular e complexo.

A família *Render Device* agrupa os membros mais complexos dentre todas as famílias do framework. Cobrir toda a gama de métodos de renderização utilizados na computação gráfica moderna em uma família coerente com uma interface suficientemente simples é um enorme desafio. As duas principais ramificações desta família são os *render device* baseados em vértices e os*render device* baseados em outras estruturas. Uma visão geral da apresentação desta família pode ser vista na figura 4.4.

O primeiro grupo diz respeito ao tipo de *render device* que utilizam-se de hardware gráfico tradicional como as placas aceleradoras com suporte a OpenGL e DirectX (mantendo as APIs como características de mediadores). Como exemplos de uso deste tipo de renderizador podemos citar o modo de edição da maioria dos softwares de modelagem 3D e a maioria dos jogos de computador modernos. O tipo de gráfico construído por esta categoria de sistema também é popularmente conhecido como gráficos poligonais. Mesmo como o termo poligonal se apresentando como o nome mais comumente usado, foi preferido aqui um nome enfocando os vértices, por estes se apresentarem como unidade fundamental de oparação neste tipo de sistema.

O segundo grupo cobre um conjunto muito diverso de técnicas de renderização, dentre as quais a mais famosa e expressiva é o *ray-tracing*. Devido à alta complexidade e variabilidade desta sub-família, a fatoração da mesma deve ser continuada em um futuro não muito distante, para que realizações das mesmas possam ser implementadas. Usualmente este tipo de *render device* é usado para a produção final de imagens de alta qualidade, onde o tempo de produção é medido em minutos e não em milisegundos, tal como filmes e trabalhos finais de modeladores 3D.

Devido ao estágio mais avançado de implementação dos *render device* baseados em vértices, colocaremos um pouco mais de atenção sobre esta

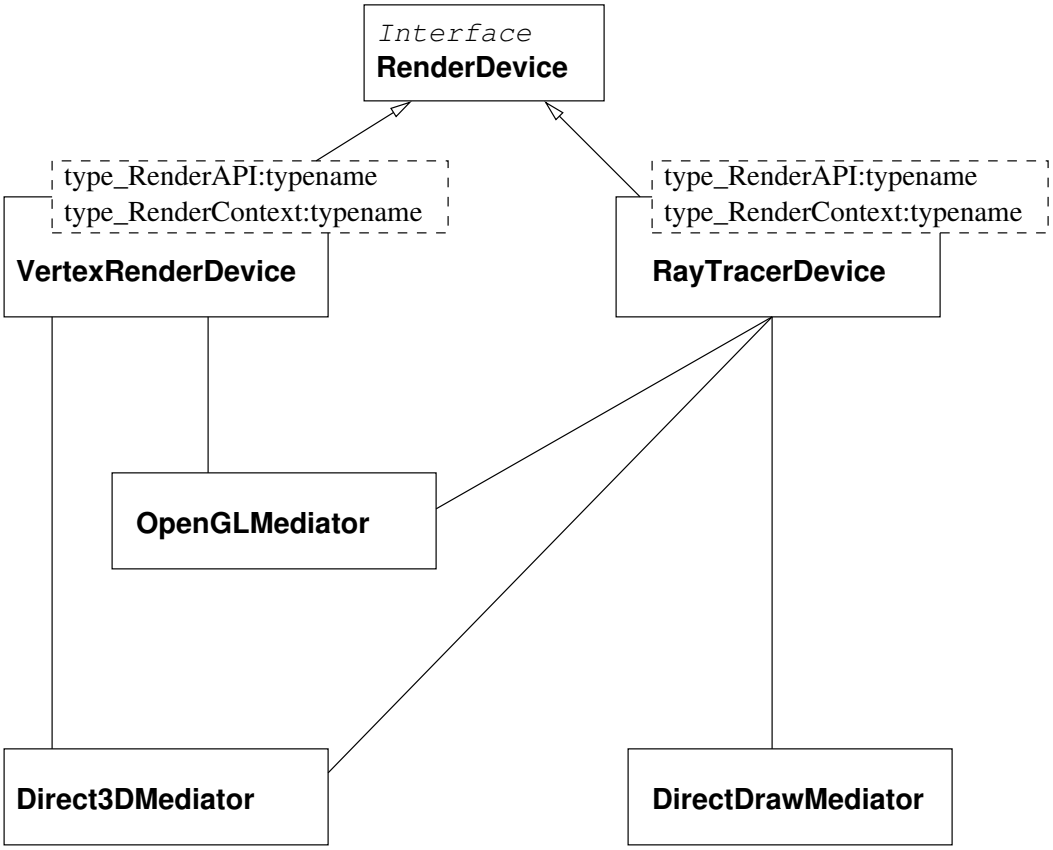

Figura 4.4: Composição da Família RenderContext

sub-divisão da família, especialmente ao estudarmos sua interface.

A interface de um *vertex based render device* é muito extensa, devido a enorme quantidade de funcionalidades que a computação gráfica desenvolveu sobre esta área. Um estudo sobre as implementações deste tipo de sistema demonstrou que as operações implementadas definem-se ao redor das capacidades apresentadas pelo hardware gráfico e APIs usadas para controlá-los. Deste modo a interface dos *vertex based render device* foi construída de forma a cobrir as funcionalidades que as APIs gráficas de baixo nível apresentam. O desenvolvimento da interface teve como base os recursos providos pela API OpenGL. A escolha do OpenGL como definição das capacidades de um renderizador baseado em vertices se deve pelo OpenGL ter se apresentado como um *super-set* de outras APIs da mesma categoria. Deste modo a interface resultante apresenta métodos para realizar qualquer renderização possível com o OpenGL ou qualquer API de capacidade similar.

As chamadas seguindo a interface desta família são direcionadas ao medi-

ador da API escolhido como parâmetro da abstração. O mediador escolhido pode ser ser o OpenGL ou qualquer outro. No caso do mediador de OpenGL o mapeamento é direto para poucas ou uma única chamada de OpenGL, o que facilita a adaptação dos programadores gráficos para o uso do SPLINE.

Uma característica que deve ser exposta é a decisão de, ao contrário da maioria dos sistemas, retirar da família de renderizadores as funcionalidaedes de *buffer swap*, por acreditar que esta funcionalidade é de responsabilidade do contexto de renderização.

4.2.3 Família SceneGraph

Um sistema gráfico além de recursos para gerar resultados, precisa de uma representação adequada das estruturas que serão desenhadas, bem como de uma descrição do fluxo de funcionamento do sistema. As estruturas de dados de alto nível que representam o mundo 3D são normalemnte conhecidas como grafo de cena. Inúmeros modelos de grafos de cena foram desenvolvidos com a evolução da computação gráfica para representar de um modo mais eficiente determinados tipos de estruturas ou ambientes. Um dos exemplos mais bem sucedidos são as *Binary Space Partitioning Trees* ou simplesmente árvore BSP. As BSP são usadas para representar ambientes fechados e de construção artificial, tal como interior de prédios e ruas de cidades, através da subdivisão do espaço por planos. O uso de uma estrutura especializada como essa pode aumentar em muito a eficiência do *graphic engine* se comparado com o uso de uma estrutura menos adequada.

A construção de um *graphic engine* gira em torno do tipo de estrutura ou estruturas de dados usadas para representar o grafo de cena. Os algorítmos de seleção de primitivas, colisão e outros são diretamente dependentes destas estruturas. A importância da correta definição e implementação do grafo de cena é tão grande que costumeiramente os *graphic engines* são classificados de acordo com o tipo de grafo de cena que utilizam.

Paralelamente à organização estrutural do engine representada pelo grafo de cena, existe uma organização dinâmica do funcionamento do sistema. A estrutura dinâmica pode ser muito simples, como um loop que atualiza a posição das estruturas e alimenta a renderização uma única vez, ou pode consistir de um complexo sistema onde *culling* é efetuado, sombras são projetadas em texturas, *light maps*, *bump maps* e outros tipos de sub produtos são calculados e adicionados ao frame em diversas passagens. O OpenPerformer [5] utiliaz-se de um sistema de *pipeline* com três fazes, uma para *host simulation*, uma para culling e uma para a renderização (denominada de *Draw Phase*). Apesar de parecer um esquema bem simples, é extremamente adequado ao papel que o OpenPerformer tem e ao tipo de sistema

em que opera. Por outro lado o Quake III [2] faz uso de diversas passagens para aplicar sombras e outros efeitos à cena final.Nas quatro primeiras passagens o *engine* do Quake III faz a acumulação de *bump map*, seguido de uma passagem par calcular a iluminação difusa, outra para a textura base dos objetos, continunado com duas outra spassagens calculando imuminação especular e emissiva e duas últimas passagens calculando efeitos atmosféricos e *flashes* na imagem. Esta abordagem é bem mais complexa, mas mostra-se mais adequada a situação e objetivos do software a qual é aplicada. Usualmente abordagens derivadas do conceito de pipeline são as mais utilizadas na computação gráfica, mas isso nem sempre pode ser encarado como uma regra.

Para que o SPLINE atingisse seu objetivo de permitir a construção de sistemas configuráveis, era necessário que as abstrações da família *scene graph* permitissem que qualquer tipo de organização estrutural fosse implementada. A primeira tentativa foi a de seguir os conceitos mais tradicionais e separar a organização dinâmica da estrutural. No processo de desenvolvimento e fatoração da família entretanto, foi percebido que esta não era uma abordagem prática pois a flexibilidade ficava muito aquém do necessário. Solucionou-se o problema com uma abordagem unificada tanto da estrutura como da organizaçào dinâmica do *engine*. O grafo de cena do SPLINE permite representar de maneira homogênea objetos poligonais, superfícies parametricas, luzes, transformações, aplicação de texturas, múltiplas passagens e redirecionamento do fluxo de processamento. A figura 4.5 mostra a organização da família *scene graph* que permite isso.

A organização da família *Scene Graph* é tal que seu membros são apenas adaptadores. Os adaptadores são capazes de traduzir os comandos dos *visitors* para os elementos encapsulados, bem como do comando reverso quando do retorno na árvore ou grafo. Alguns dos membros como o *Command Node* permitem que comandos específicos para o renderiador sejam programados em um determinado ponto da árvore. Outros nodos podem apenas registrar passagens pelo ponto específico do grafo com fins de controle do fluxo de execução por outro nodo. Todos os nodos tem como parâmetro de template o tipo que é encapsulado e muitas vezes, dependendo do tipo de nodo, também possuem parâmetros para o *render device* associado ou para o *physics device* associado.

Com a confecção de nodos de tipo diferente pode-se conseguir qualquer tipo de grafo de cena que usualmente seria usado em um sistema gráfico. A grande vantagem do sistema de grafo de cena do SPLINE é a facilidade de integrar tipos de estruturas diversas a grafos de cena previamente construídos, isto porquê o algorítmo de execução do grafo está descrito na prórpia construção do mesmo e não existe nenhum acoplamento fixo que impossibi-

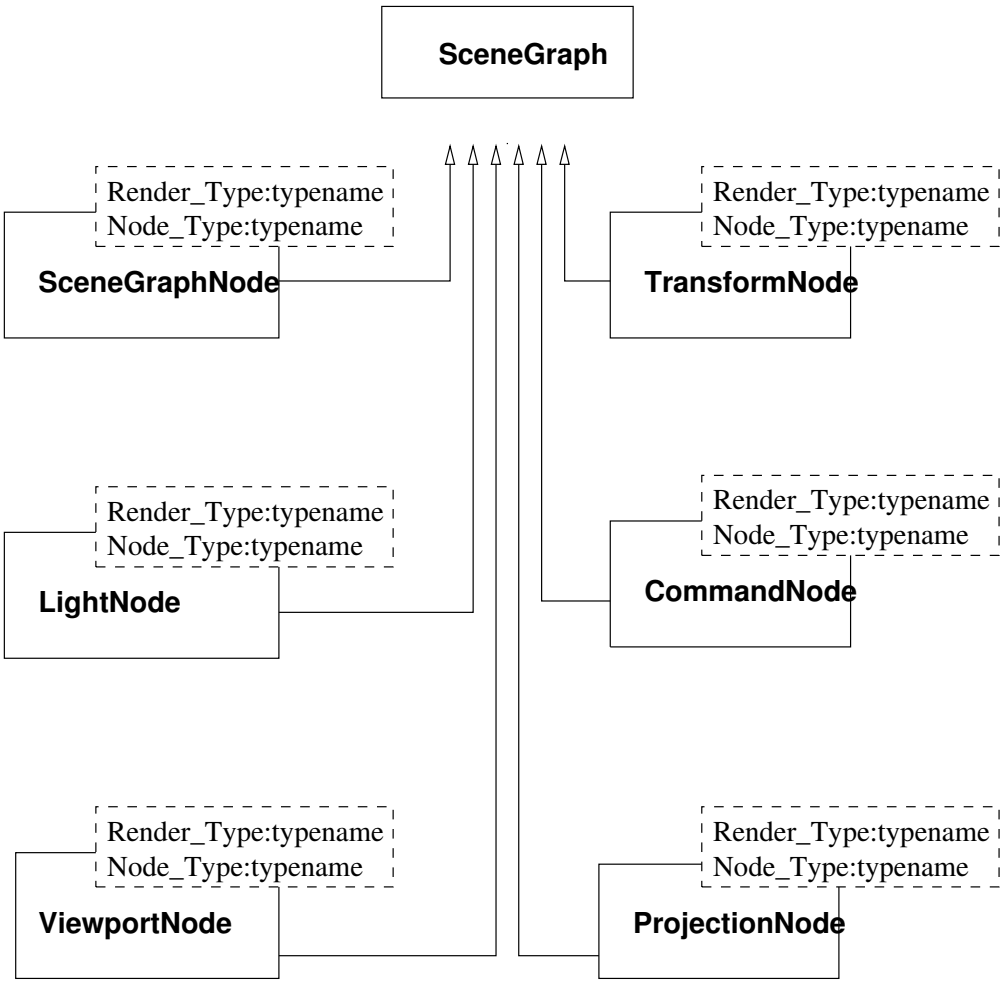

Figura 4.5: Composição da Família SceneGraph

litequalquer tipo de montagem entre os nodos. Alguns algorítmos utilizados por abstrações específicas podem exigir que o grafo de cena seja organizado de acordo com um formato específico e é importante que este formato possa ser conseguido com as abstrações do SPLINE.

4.2.4 Família Transform

Salvo os sistemas gráficos mais simples, visualizar estruturas estáticas está longe de ser útil. Para a maioria dos *graphic engines* existe a necessidade de executar translações e rotações em estruturas ou no ponto de origem da visão. Diversas abstrações matemáticas foram criadas para representar e calcular estas transformações nos sistemas de coordenadas, tais como ângulos

Eulerianos, Matrizes e Quaternions. O SPLINE apresenta um tipo de nodo especial para representar uma transformação no sistema de coordenadas e permitir que esta transformação seja desfeita no processo de retorno pela árvore. O *TransformNode* é uma abstração que encapsula implementações de transformações lineares. Em outras palavras, o *TransformNode* é instanciado com um parâmetro de template descrevendo que componente deve ser usado para armazenar e manipular as transformações. A figura 4.6 demostra a estrutura desta família.

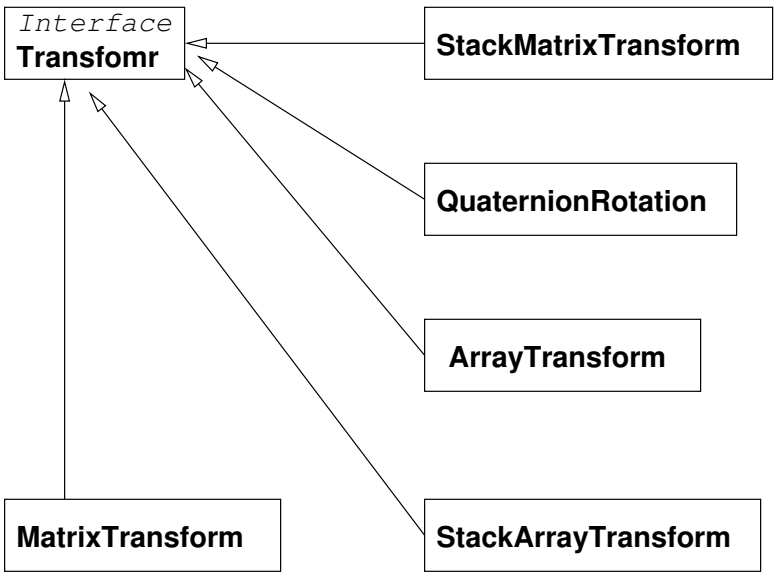

Figura 4.6: Composição da Família Transform

Além das trasnformações normais, as quais são feitas na passagem por um nodo da árvore durante a descida e desfeitas na volta, podem ser usadas *stack transforms* que permitem que o sistema de coordenadas atuais seja salvo antes da aplicação da transformada e posteriormente recuperado. As abstrações representado *stack transforms* devem ser capazes de executar este serviço, seja pelo suporte direto do renderizador ou manualmente. Para isso, usa-se como parâmetro das abstrações um flag adiquirido estaticamente da classe que representa o renderizador associado ao nodo que encapsula a transformação.

Transformações de projeção são tratadas da mesma forma que as transformações sobre os sistemas de coordenadas, incluindo o conceito de pilha de transformações.

4.2.5 Família PolygonConstructs

Grande parte das estruturas e objetos desenhados em um sistema gráfico são compostos por aglomerados de polígonos. Estas estruturas são representadas no SPLINE pelos membros da família *Polygon Construct*. Uma amostra de parte dessa família é apresentada na figura 4.7

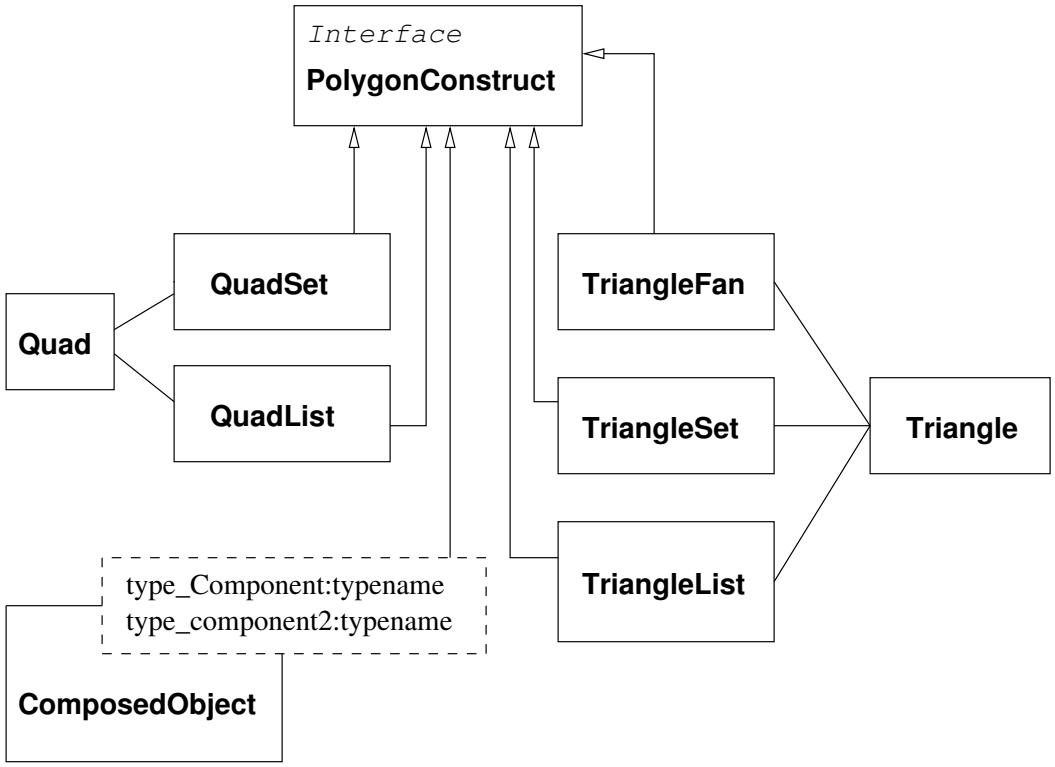

Figura 4.7: Composição da Família Polygon Construct

Dentre as variantes principais encontram-se membros que armazenam os dados em formato de *arrays* ou *buffers* de vértices e fazem requisições explícitas para renderização deste tipo de stream de dados. Outros membros permitem a associação de materiais membros da família *Material*.

Todos membros desta família devem ter a capacidade de responder com informações básicas que podem ser úteis para membros de outras famílias, tal como centro e raio limite de presença. De resto a interface dessa família é limitada ao comandos de renderização e perparação para renderização, os quais são implementados como *function templates* com membros da família *Render Device* como parâmetro. Deste modo um objeto poligonal sabe que comandos de um *render device* precisam ser acionados para que seu conteúdo seja desenhado, e requisita estas operações ao *render device* passado como

parâmetro.

Sistemas complexos podem ser obtidos pelo uso do membro *Composed-Construct* que agrega diversos outros membros de forma a estruturar entidades mais complexas. Um tipo especial de *ComposedConstruct* permite que *level of detail* seja implementado através de rotinas de alteração de estado. Os membros desta família são frequentemente usados em associações com outros membros, espera-se que em breve com a implementação da família *Bones* que representa um tipo de recurso muito usado na computação gráfica moderna.

Esta família pode ainda sofrer um processo grande de complexificação e introdução de novas abstrações com fins de permitir o melhor manejamento das informações representadas. A capacidade de representar e manusear *voxels* é a provável próxima adição a esta família.

4.2.6 Família Evaluators

Além dos objetos poligonais, as estruturas paramétricas tais como sólidos geométricos, curvas e superfícies de Bezier ou splines são muito usadas. Apesar de raras em aplicativos iterativos de alto desempenho como jogos, são muito usadas em modeladores 3D e visualizadores de dados científicos. Em comum estas entidades são definidas por *evaluators*, o que torna a interface desta família bastante simples e homogênea. A figura 4.8 mostra um exemplo de curva de bezier renderizada pelo SPLINE.

A maioria dos comentários feitos sobre a família *Polygon Construct* são válidos para a família Evaluators, tal como a associatividade com materiais e em estruturas de composição. Uma funcionalidade ainda não implementada, mas dentro dos planos atuais é a transformação entre membros das duas famílias.

4.2.7 Família Light Source

As luzes são um elemento muito importante na confecção de ambientes e cenas realistas, pois é com o efeito destas que a profundidade é induzida ao observador. Mais de um modelo de luzes foi proposto no decorrer da história da computação gráfica, todavia apenas um é usado amplamente nos dias de hoje. Uma luz sempre é composta por um componente de cor difusa, um componente de cor especular e um componente de cor ambiente. Uma fonte de luz pode possuir informações posicionais e direcionais, por exemplo no caso de uma *spotlight*.

Usualmente as APIs que suportam sistemas gráficos possuem um limite de luzes que podem estar ativas em um determinado momento. Tratar deste limite não foi considerado como sendo responsabilidade dos membros da

Figura 4.8: Superfície de Bezier

família *Light Source*, mas sim de mebros da família *Render Device* e seus mediadores.

Existem membros mais complexos da família *Light Source*, como por exemplo uma fonte de luz com projeção de sombras. Estas ao serem alcançadas no grafo de cena podem alterar o fluxo de execução ou mesmo o estado de funcioanmento dos *render device* para preparar a projeção da sombra antes da renderização real acontecer. Como a área de *soft shadows projection* é alvo de muitas pesquisas atualmente, espera-se que esta família venha a sofrer grandes modificações e melhorias com o constante aparecimento de novas técnicas.

4.2.8 Família Material

Profundamente associado com o conceito de luzes na computação gráfica encontra-se o conceito de materiais. A idéia por trás deste conceito é a de que materiais com diferentes características respondem de forma diferente à iluminação. As diferenças de características dos materiais determinam que porcentagem dos componentes difuso, especular e ambiente da fonte de

luz serão refletidos. Estas características do matarial associadas as características da luz determinam qual a cor com que visualizamos o objeto.

No seu modo mais simples um material nada mais é que um conjunto de componentes difuso, ambiente, especular, brilho e iluminação própria. Como é esperado, os componentes com correspondentes nas luzes indicam os índices de absorção do componente análogo na fonte de luz. Ocomponente brilho está profundamente associado ao componente especular e indica o tamanho da área de efeito especular. O componente de iluminação própria permite simular o efeito de um objeto que seja uma fonte de luz. Importante salientar que isso é diferente de ser uma fonte de luz para outros objetos, visto que este componente só afeta o objeto ao qual o material se refere.

A texturização, que por muito tempo tem sido a principal técnica para adicionar detalhes e realismo aos objetos sem adicionar grande complexidade geométrica, é uma característica do material. Para cumprir com estas responsabilidades o material tem acesso a dados de texturas e emite comandos específicos para que o *Render Device* aplique estas texturas sobre sua renderização. Múltiplas texturas e filtragem de texturas são tratadas do mesmo modo e de uma forma transparente para quem monta o grafo de cena. Materiais com suporte a texturas procedurais tem acesso a membros de famílias especialmente designadas para gerar este tipo de textura.

Formas muito mais avançadas de materiais são usadas na computação gráfica. Tipos especiais de características de sombreamento, mapas de normais, *vertex shaders* [17] e *pixel shaders* estão dentre os diversos recursos que são responsabilidade dos materiais. Algumas destas características já estão implementadas no SPLINE, enquanto outras ainda precisam ser implementadas.

4.2.9 Família Phisycs Device

Além das funcionalidades de representação e renderização de mundos e objetos 3D, *graphic engines* provêem funcionalidades de simulação da física do mundo representado. Dentre estas responsabilidades está incluso a detecção de colisão, aplicação de forças, mecânica de sistemas de particulas entre outros. A justificativa para a inclusão destas funcionalidades em um artefato de computação gráfica é a de que as estruturas de dados e informações necessárias para a simulação da física estão presentes em vários componentes do *graphic engine*. Um exemplo claro disto é o mecanismo de seleção de geometria, ou *culling* que faz uso das mesmas estruturas de dados e algorítmos que a detecção de colisão. replicar estas estruturas fora do sistema gráfico apenas para separar o controle de física dos elementos gráficos seria muito ineficiente.

As responsabilidades de simulação de física são de membros da família *Physics Device*. Dependendo das implementações dos membros desta família, será necessário que o grafo de cena siga determinadas regras em sua composição. Isto pode ser necessário no caso de um algorítmo de detecção de colisão projetado para operar em um grafo BSP. Uma maneira uniforme de tratar a física é a idéia de que ante qualquer modificação de posição de um objeto ou alterações em uma translação, o nodo que inicia a árvore modificada dispara um evento de notificação para o *Physics Device*. O *Phisic Device* cuidará de computar os efeitos desta modificação, somando as forças que atuam sobre os objetos envolvidos . As modificações podem ser causadas por comportamentos embutidos no elemento, ou serem disparados pelo próprio *Physics Device*.

O *Physics device* mantém registro de todos os nodos cadastrados sob sua gerência. Os nodos que possuirem uma inércia relevante, serão informados no momento adequado de sua nova posição ou orientação. O ajuste na inércia do objeto será efetuado com base nas forças correntemente aplicadas no nodo cadastrado, bem como em forças especiais, tal como atrito e gravidade as quais são calculadas com uso de nodos de ambiente. Nodos de ambiente caracterizam-se por marcarem pontos no grafo de cena em que todos os seus decendentes estão submetidos a alguma situação especial, tal como uma força ou um coeficiente de atrito.

É importante ressaltar que o grafo de cena utilizado para renderização não precisa ser necessariamente o mesmo grafo que é usado para o controle de física. Basta que cada elemento do segundo grafo tenha um correspondente no primeiro, repassando os comandos de modificação. Este tipo de abordagem melhora a modularidade do sistema, permitindo uma melhor divisão entre as estruturas conceitualmente representadas no sistema e os dados utilizados para renderização.

A implementação desta porção do framework é muito complexa e ainda está longe de ser concluída. A interface desta família ainda está crescendo e se adaptando. A escalabilidade propiciada pelo SPLINE permite que várias outras abordagens de controle de de física sejam usadas sem exigir modificações de outros componentes.

4.2.10 Outros componentes

Para manter um alto grau de escalabilidade e configurabilidade, as abstrações do SPLINE são construídas sem acoplamento sintático com outras abstrações. Todavia a grande maioria das abstrações de um sistema de computação gráfica faz uso de conceitos primitivos fundamentais. Os componentes que representam estes conceitos e estruturas são os únicos artefatos sobre

os quais é permitido um forte acoplamento.

Dentre os componentes básicos encontram-se vetores, vértices, matrizes, quaternions, triângulos, quadrados, segmentos de reta, planos e cor. Tododos estes componente apresentam implementações de algorítmos matemáticos clássicos que se mostram fundamentais na computação gráfica. Produtos escalares e cruzados de vetores, multiplicações de matrizes e interpolação de quaternions são apenas alguns dos exemplos destas funções.

Outras famílias de abstrações fazem parte do framework, mas são demasiado simples ou ainda pouco trabalhadas no SPLINE para aparecerem neste documento. Com o crescimento do framework diversas novas fatorações devem surgir em famílias já pensadas e até mesmo em famílias com membros já implementados,e é exatamente esse o objetivo a longo prazo deste projeto.

4.3 Projetos utilizando o SPLINE

Durante o desenvolvimento do protótipo do SPLINE, surgiram duas grandes oportunidades de testar o framework em aplicações reais. Foi decidido que o rumo do desenvolvimento dos componentes de software seria guiado com fins de suprir a necessidade das abstrações e mediadores necessários para completar estas aplicações.

4.3.1 Cyclops 3D

Sob o escopo do projeto Cyclops [3], surgiu a necessidade de ferramentas para visualização de reconstruções de dados médicos. A primeira destas ferramentas foi denominada de Cyclops 3D e visava a visualização e manuseio de reconstruções de tomografias em formato VRML. O desenvolvimento desta aplicação foi um ponto muito importante na decisão do desenvolvimento do SPLINE. Os primeiros protótipos foram desenvolvidos sem o uso de qualquer uma das técnicas discutidas neste trabalho, intencionando a produção de um engine para um uso específico. Após enfrentar um desenvolvimento já prejudicado pela falta de artefatos reusáveis, o projeto entrou em fase de estagnação. A inclusão de novas funcionalidades e alterações no funcioanmento do protótipo faziam necessárias re-estruturações enormes no software e muito dispendio de força de desenvolvimento.

Encarando este problema decidiu-se que um novo protótipo seria reconstruído do zero, sobre um framework que permitisse reutilizaçào de componentes em futuros softwares. A idéia deste framework evoluiu, e quando combinadacom as idéias de Frölich [8], resultou no projeto SPLINE e neste trabalho.

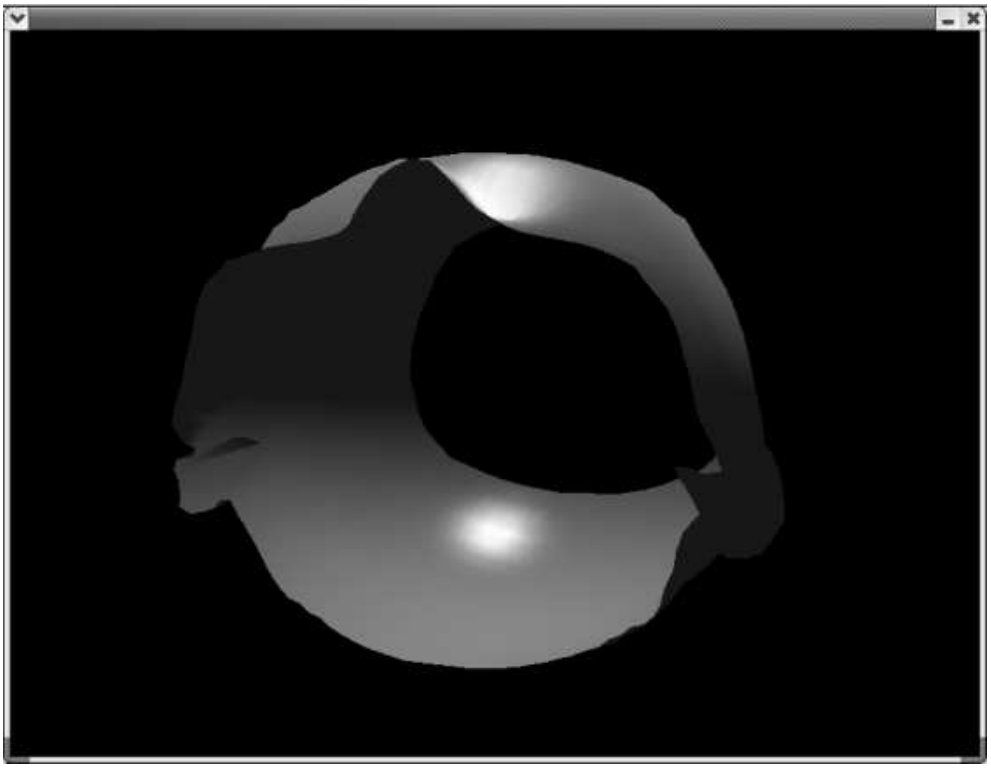

Figura 4.9: Instantâneo da execução do Cyclops 3D

4.3.2 SNOW

Pouco tempo depois do início do desenvolvimento do SPLINE, surgiu a oportunidade de desenvolver projetos de computação gráfica em paralelo no cluster SNOW [24]. A implementação de sistemas capazes de execução em um abiente multiprocessado como um cluster mostrou-se a oportunidade perfeita de comprovar os conceitos de multi plataforma do SPLINE. O aplicativo escolhido foi um sistema de *ray-tracing* em paralelo, que no momento da escrita deste documento ainda está em desenvolvimento.

Com este trabalho não objetiva-se conseguir o software de mais alto desempenho já desenvolvido, mas sim comprovar que a abordagem do SPLINE constitue a forma mais fácil de portar um sistema gráfico de uma plataforma PC para um sistema multi-processado. Como base de comparação será usado o projeto WireGL [11], que faz uso de uma abordagem de distribuição de chamadas de API pelos nodos.

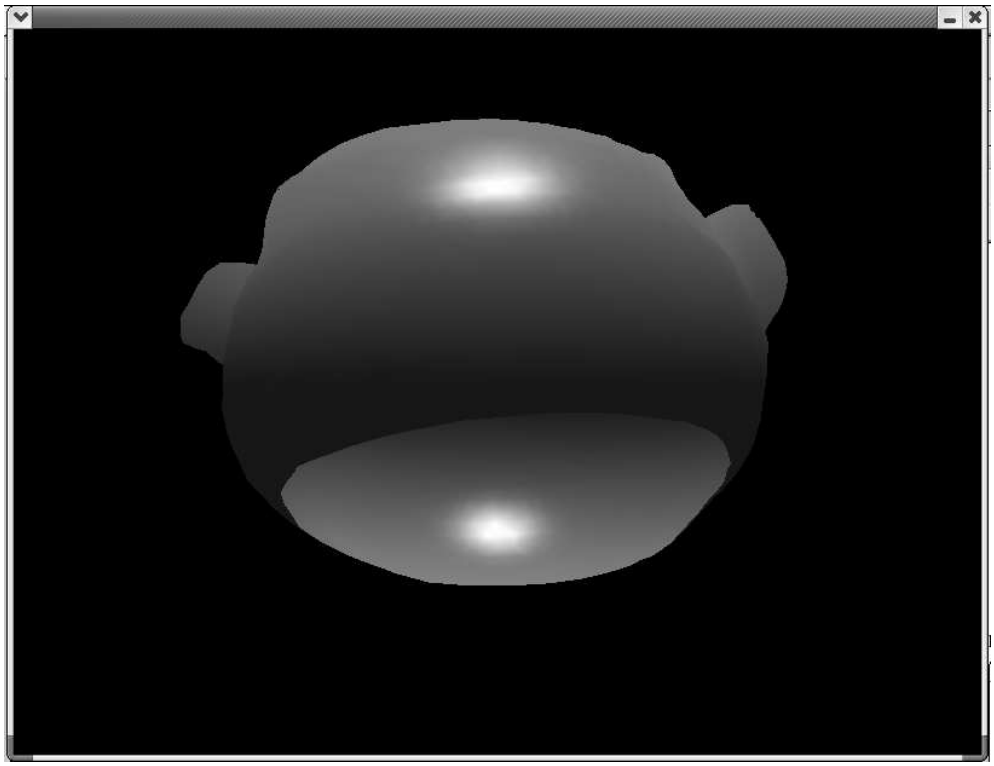

Figura 4.10: Instantâneo da execução do Cyclops 3D

Capítulo 5

Trabalhos correlatos

Para analisar a correlação deste trabalho com outros trabalhos, tanto do mundo científico como comercial, precisamos abordar duas áreas principais. A primeira área é a engenharia de software por trás deste trabalho, principalmente os conceitos de *Application Oriented Design*. A principal obra de inspiração neste campo é o trabalho de Föhlich em *Application Oriented Operating Systems* [8], na qual apresenta-se um aproposta e um protótipo de desenvolvimento de sistemas operacionais orientados à aplicação.

Diversos outros projetos fazem usos de técnicas similares as tratadas neste trabalho, tal como *Aspect Oriented Programming*, e metaprogramação estática para conseguir toda a modularidade e configurabilidade que o framework necesita para gerar systema voltados a uma aplicação específica. A computação gráfica infelizmente não se mostra como uma das áreas com maior entusiasmo sobre a aplicação de técnicas como estas.

A segunda área que precisa ser observada é a da computação gráfica propriamente dita e sua interação com as atuais técnicas de engenharia de software. Não são poucos os trabalhos acadêmicos que visam associar áreas da computação gráfica com engenharia de software. Diversos frameworks para tipos específicos de *graphic engines* são desenvolvidos constantemente. Todavia frameworks e técnicas para construção e configuração de *graphic engines* de qualquer tipo não encontram representantes no meio acadêmico. Tal abordagem só foi aplicada em sistemas comerciais como o OpenPerformer [5] da SGI. Isto se deve provavelmente ao fato de normalmente as abordagesn genéricas na computação serem associadas com baixo desempenho. devido a grande dependênca de desempenho da computação gráfica, os design de sistemas gráficos genéricos foram desfavorecidos em favor de abordagens mais especialistas. Um exemplo claro disto é o *graphic engine Unreal Warfare* [27], extremamente configurável e muito poderoso, todavia de uso bastante restrito a jogos de primeira pessoa.

A idéia de produzir um *graphic engine* genérico o suficiente para ser usado em diversas aplicações produziu diversos experimentos e até produtos comerciais como o Nebula Device [16]. Este é um ótimo exemplo de *graphic engine* que começou como um projeto para um jogo de computador em específicao, mas evoluiu para uma ferramenta altamente configurável. A solução da configurabilidade é feita sobre o modelo de *scripts*, o que é cada vez mais comum no dia de hoje. O SPLINE como foi discutido em vários pontos deste trabalho, encara o problma através de uma solução diferente, menos flexível em tempo de execução, mas permitindo mudanças mais profundas a nível estático.

Alguns dos problemas de independência de plataforma abordados pelo SPLINE, são solucionados de maneiras diversas por outros projetos. Um destaque nesta área é o WireGL [11], que soluciona a renderização distribuída através da interceptaçãos das chamadas de API do OpenGL e redistribuição das mesmas para as estações de renderização. Esta abordagem permite aindependência da aplicação com relação ao ambiente em que é executada, não importando se este é multiprocessado ou não. A desvantagem é exatamente o fato do WireGL só resolver esse único problema, e estar fixado a uma API específica. O SPLINE planeja implementar uma solução mais felxível para este problema, baseado na possibilidade de cad nodo do grafo estar associado a um *render device*, que pode utilizar um mediador local, um mediador distribuído, ou o próprio WireGL.

A computação gráfica é uma área que está sob um vertiginoso desenvolvimento e projetos objetivando melhorar a reusabilidade e escalabilidade surgem a cada momento. Atualmente está em voga a abordagem de configurabilidade por scrips que é bastante diferente das técnicas utilizadas pelo SPLINE. A recente passagem das responsabilidades de *transformation and lighting* para o hardware gráfico permitiu que mais poder de processamento fosse redirecionado para funções como a interpretação de scripts, o que tem permitido o crescimento de tal abordagem. Continua mesmo assim a validade de sistemas como o SPLINE que dispensam a construção de um *engine* extremamente complexo para atender a tarefas mais simples, tal como aplicativos para suporte de outros projetos científicos. Nesta área o SPLINE atualmente mostra-se como uma solução sem concorrência direta, pelo menos fora dos produtos comerciais de alto custo.

Capítulo 6

Trabalhos futuros

O projeto SPLINE caracteriza-se por ser um típico projeto sem fim. Permitir a criação de novos componentes e fácil utilização dos mesmos no framework é foi parte importante neste projeto. Tal característica só será aproveitada se o desenvolvimento do framework for uma tarefa constante. O estágio de implementação das realizações do SPLINE ainda está longe de produzir realizaçòes para as abstrações de todas as famílias, deixando espaço para muito trabalho futuro.

O grande salto futuro no desenvolvimento do SPLINE será adoção de um padrão de descrição e confecção de aspectos, provavelmente AspectC++. Obetiva-se resolver os problemas que dificultam o uso do AspectC++ no SPLINE, bem como em outros projetos em um futuro não muito distante. A disponibilidade de recursos de confeção e *weaving* de aspectos propiciariam ao SPLINE um nível ainda maior de reusabilidade e configurabilidade estática, o colocando em uma categoria realmente única dentre as soluções para o desenvolvimento de *graphic engines*.

No campo de aplicação, o projeto SPLINE ganhou apoio do projeto Cyclops, no qual futuros aplicativos com recursos de visualização de imagens 3D serão desenvolvidos sobre o framework. Dentre os projetos em eminência de realização, destaca-se a reprodução de dados estruturais combinados com dados de atividade do cérebro em uma única reconstrução.

O SPLINE também está disponível para uso por qualquer estudante, que no curso de seus trabalhos acadêmicos necessitar de recursos de visualização de imagens 3D. Espera-se que com o desenvolvimento de novos módulos por parte dos futuros usuários, o SPLINE possa crescer muito tanto em capacidade como em utilidade. Com o tempo e com o corerto desenvolviemnto este protótipo pode chagar ao ponto onde os *graphic engines* com eles montados não devam em nada a qualquer outra alternativa construída sem a capacidade de configurabilidade.

Capítulo 7

Conclusão

A aplicação das modernas técnicas de engenharia de software com fins de possibilitar a reusabilidade e modularidade dos artefatos de software é uma tendência que não pode ser ignorada por nenhuma sub área da computação. Mesmo as áreas que tradicionalmente apresentam resistência a tais técnicas podem e devem fazer uso delas. Este trabalho e em especial o protótipo implementado para sua validação comprovam essa premissa e trazem a possibilidade de um grande desenvolvimento futuro.

O SPLINE não foi elaborado com intuito de produzir um produto capaz de concorrer no mercado com as soluoes já estabelescidas, em especial o Open Performer da Silicon Graphics, o qual compartilha um mesmo nicho de aplicação. Mesmo não tendo estes objetivos, o SPLINE evoluiu para a base de uma possível plataforma competitiva de desenvolvimento de sistemas gráficos. As características de escalabilidade e alta configurabilidade do framework que o SPLINE herdou da engenharia de software utilizada em sua confecção, o colocam em uma posição privilegiada no quesito da facilidade e coesão durante a expansão de suas capacidades.

Bibliografia

[1] 3DLabs. Opengl 2.0 overview. Technical report, 3DLabs, February 2002.

[2] Michael Abrash. Ramblings in realtime.

[3] Tiago Stein D'Agostini. Cyclops 3d. In *Simpósio Catarinense de Processamento Digital de Imagens*. UFSC, November 2001.

[4] Edsger Wybe Dijkstra. Notes on structured programming, 1969.

[5] Programmer's Guide Document. Iris peformer.

[6] Michael Doggett. Programmability features of graphic hardware. Technical report, ATI, 2002.

[7] James D. Foley, Andries van Dam, Steven K. Feiner, and John F.Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley, 1987.

[8] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. PhD thesis, GMD - Forschungszentrum Informationstechnik, 2001.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] Instituto Antônio Houaiss. *Dicionário Houaiss da Língua Portuguesa*. Objetiva, 2001.

[11] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. In *In Proceedings of Supercomputing*, pages 60–60, 2000.

[12] Richard S. Wright Jr. and Michael Sweet. *OpenGL SuperBible*. Waite Group, second edition, December 1999.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[14] Mark J. Kilgard. *The OpenGl Utility Toolkit Programming Interface API.* Silicon Graphics, 1996.

[15] Mark J. Kilgard. Cg in two pages. Technical report, NVIDIA Corporation, January 2003.

[16] Radon Labs. Nebula device. CVS, September 2001.

[17] Chris Maugham and Matthias Wloka. Vertex shaders introduction.

[18] Michael McCool, Zheng Qin, and Tiberiu Popa. Shader metaprogramming.

[19] *DirectX 8.0 SDK.*

[20] Janson L Mitchel. Hardware shading in the ati radeon 9700. *SIGGRAPH*, 2002.

[21] "Tomas Moller"and "Eric Haines". *"Real Time Rendering"*

. A K Peters, 1999.

[22] Jackie Neider and Tom Davis. *OpenGL Programming Guide, 1.1.* Addison-Wesley, January 1997.

[23] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1), March 1976.

[24] Wofgang Schröder-Preikschat, Phillipe Oliver Alexandre Navaux, Antônio Augusto Medeiros Fröhlich, and Sérgio Takeo Kofuji. Snow: A parallel programming environment for clusters of workstations.

[25] Inc. Silicon Graphics. *Standard Template Library.*

[26] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, June 1997.

[27] Tim Sweeney. *UnrealScript Language Reference.* Epic Megagames.

[28] Oxford University. *Oxford Dictionary.* Oxford University Press, 1998.

[29] University of Magdeburg. *Programm Instrumentation for Debugging and Monitoring with AspectC++*, May 2002.

[30] Alex Vlachos. Preparing sushi- how hardware guys write a 3d graphic engine.

[31] Chris Wynn. Opengl vertex programming future generation gpus.