Open Graphics Library (OpenGL®)

por

Filipe Gonçalves Barreto de Oliveira Castilho
Nuno Alexandre Simões Aires da Costa
Departamento de Engenharia Informática
Universidade de Coimbra
3030 Coimbra, Portugal
fgonc@student.dei.uc.pt
nacosta@student.dei.uc.pt

Resumo: *Apresenta-se, de uma forma um pouco técnica, o funcionamento do Open Graphics Library (OpenGL). Para esse efeito, descrevem-se as características gerais da plataforma, seguindo-se uma análise pormenorizada das componentes mais importantes que lhe dão vida. Finaliza-se com um exemplo simples do código de um programa para esta plataforma, seguida de uma comparação entre o OpenGL e o seu oponente directo, o Direct3D. Pretende-se com a abordagem seguida, dar a conhecer melhor o OpenGL e motivar o leitor a aprofundar os seus conhecimentos.*

Palavras-Chave: *Open Graphics Library, OpenGL, 3D Graphics API, Framebuffer, Renderização, Antialiasing, Display List, Direct3D.*

1. Introdução

O grafismo computacional (em particular os gráficos tridimensionais [3D] e gráficos 3D interactivos) está a surgir cada vez mais num grande número de aplicações informáticas, desde os simples programas para construção de gráficos até aos mais complexos e potentes programas de modelação e visualização gráfica para super computadores e estações gráficas. À medida que o interesse pelo processamento gráfico computacional tem vindo a aumentar, também tem vindo a crescer o desejo de produzir aplicações que usufruam dos gráficos 3D e que sejam passíveis de ser executadas em várias arquitecturas com capacidade de processamento gráfico diferente. A introdução de uma norma na área do grafismo computacional facilita esta tarefa, eliminando a necessidade de programar diferentes controladores gráficos para cada arquitectura onde as aplicações finais irão ser executadas.

Na área do grafismo a duas dimensões [2D], várias normas conseguiram ser implementadas. A linguagem *PostScript* é um exemplo. A norma que engloba esta linguagem simplificou o processo de troca electrónica e, até um certo grau de limitação, a manipulação de documentos estáticos que combinavam texto com gráficos 2D. O sistema de janelas X (*X Windows System*) é a norma para os computadores com UNIX. Um programador usa o X para obter uma janela no monitor, na qual se pode escrever texto ou desenhar conteúdos 2D. A adopção do X pelos diversos fabricantes, permite que um programa produza gráficos 2D ou obtenha interacção com o utilizador numa vasta gama de diferentes computadores, necessitando apenas de recompilar o programa. Esta integração até funciona através de uma rede: o programa pode executar uma vez num computador e mostrar a interacção com o utilizador noutro computador situado do outro lado da rede, mesmo que este tenho sido construído por outro fabricante [1].

Foram apresentadas diversas normas para a área dos gráficos a três dimensões, mas a maior parte delas não conseguiu ganhar grande aceitação. Um sistema relativamente bem conhecido é o PHIGS

(*Programmer's Hierarchical Interactive Graphics System*). Baseado no sistema GKS (*Graphics Kernel System*), o PHIGS faz parte da uma norma ANSI (*American National Standards Institute*). O PHIGS (e o seu descendente PHIGS+), fornece meios para manipular e desenhar objectos tridimensionais (3D) através do encapsulamento de descrições e atributos dos objectos para uma *display list* que depois é referenciada quando o objecto é para ser mostrado ou manipulado. Uma vantagem da *display list* é que um objecto complexo precisa apenas de ser descrito uma vez, mesmo quando ele precisa de ser mostrado várias vezes. Isto é especialmente importante se o objecto a ser mostrado precisa de ser transmitido por um canal com baixa largura de banda. Uma desvantagem da *display list* pode surgir quando um objecto está constantemente a ser alterado por interacção com o utilizador, resultando um gasto considerável de esforço para o voltar a especificar. Outra dificuldade do PHIGS e PHIGS+ (e também com o GKS) é a falta de suporte para técnicas de renderização avançada (*advanced rendering*) nomeadamente mapeamento de texturas (*texture mapping*) [1].

O PEX, que é um acrónimo da extensão do PHIGS para o sistema de janelas do UNIX (o X), torna o X capaz de processar e manipular conteúdos 3D. Para além de outras extensões, o PEX torna o PHIGS capaz de mostrar objectos ao mesmo tempo que estes são descritos, ao invés de primeiro ter de completar a *display list*. A esta funcionalidade chama-se "modo de renderização imediata" (*immediate mode rendering*). Uma dificuldade do PEX, é o facto de diferentes fabricantes decidirem suportar diferentes funcionalidades, tornando a portabilidade das aplicações medíocre. O PEX, tal como o PHIGS, também não tem suporte para técnicas de renderização avançada e só está disponível para utilizadores do X.

Em 1992 foi introduzido no mundo da computação o OpenGL, que com o decorrer dos anos acabou por se tornar o ambiente de desenvolvimento de referência para aplicações leves de conteúdos dinâmicos 2D e 3D. O API (A*pplication Programming Interface*) OpenGL tornou-se o utilitário mais usado e com melhor gama de efeitos, para programação de aplicações com gráficos 2D e 3D, podendo ser usado em todo o tipo de computadores pessoais e estações gráficas, proporcionando assim uma grande portabilidade [2, 3].

2. OpenGL

OpenGL ("GL" significa biblioteca gráfica - "*Graphics Library*") é uma interface para o *hardware* gráfico. Esta interface consiste em centenas de funções e procedimentos que permitem ao programador especificar objectos e operadores envolvidos na criação de ambientes gráficos de alta qualidade, em especial imagens coloridas de objectos tridimensionais, podendo ser facilmente integrado num sistema de janelas (por ex. o Microsoft Windows®) [4].

O OpenGL coloca primitivas num *framebuffer* sujeito a vários modos seleccionáveis. Cada primitiva é um ponto, um segmento de linha, um polígono, um pixel ou um bitmap. Cada modo pode ser alterado independentemente, não afectando os restantes. Os modos são conjuntos, primitivas específicas, ou outras operações descritas através do envio de funções ou chamadas a procedimentos.

As primitivas geométricas (pontos, segmentos de recta e polígono) são definidas através de um grupo de um ou mais vértices. Um vértice define um ponto, o fim de uma extremidade ou um canto de um polígono onde duas extremidades se encontram. Os dados (que consistem em coordenadas posicionais, cores, normais e coordenadas das texturas) estão associados a um vector que é processado de forma independente, por ordem e da mesma forma que os outros. A única excepção a esta regra é quando o grupo de vértices está reunido para que a primitiva indicada se ajuste a uma dada região. Neste caso, o vector de dados pode ser alterado e novos vértices criados. A forma como os vértices são agrupados depende da primitiva representada.

O OpenGL permite o controlo directo sobre operações fundamentais de gráficos 2D e 3D. Isto inclui a especificação de alguns parâmetros como a transformação de matrizes, coeficientes de iluminação, métodos de *antialiasing* e operadores para actualização de pixels. Não fornece, no entanto, meios para a descrição e modelação de objectos geométricos complexos. Uma outra alternativa de apresentar esta situação é dizer que o OpenGL oferece mecanismos para descrever a forma como objectos geométricos complexos são renderizados em vez de mecanismos para descrever os próprios objectos.

O modelo de interpretação de comandos do OpenGL é o típico cliente-servidor. Tal significa que um programa (cliente) envia comandos que são interpretados e processados pelo OpenGL (servidor), que pode, ou não, operar no mesmo computador como cliente.

Os efeitos dos comandos de OpenGL no *framebuffer* (Figura 1) são, em última instância, controlados pelo sistema de janelas que aloca os recursos do *framebuffer*. É o sistema de janelas que determina quais as zonas do *framebuffer* às quais, em cada momento, o OpenGL pode aceder e que informa este da forma como o *framebuffer* está estruturado. Da mesma forma, a exibição do conteúdo do *framebuffer* no monitor (inclusive a transformação individual dos valores do *framebuffer* através de técnicas como a *gama correction*) não é endereçada pelo OpenGL. A configuração do *framebuffer* ocorre fora do OpenGL em conjunto com o sistema de janelas. A inicialização de contexto do OpenGL ocorre quando o sistema de janelas aloca uma janela para a renderização do OpenGL. De salientar que o OpenGL não possui, de forma nativa, quaisquer serviços de acesso a dispositivos de *input*, sendo estes fornecidos pelo sistema de janelas, o que torna o OpenGL independente de qualquer sistema de janelas.

Do ponto de vista do implementador de *hardware*, OpenGL é um conjunto de comandos que afectam o funcionamento do *hardware* gráfico. Se o *hardware* consistir apenas num *framebuffer* endereçável, então o OpenGL tem de ser implementado quase por completo no CPU. Tipicamente, o *hardware* gráfico poderá compreender vários graus de aceleração gráfica, desde um subsistema capaz de renderizar linhas bidimensionais e polígonos até processadores de vírgula flutuante sofisticados capazes de transformar e computar dados geométricos. A tarefa do implementador de OpenGL é oferecer uma interface que distribua o trabalho entre o CPU e o *hardware* gráfico. Esta distribuição deve ser feita de forma a optimizar a performance da execução das chamadas de OpenGL.

O OpenGL pode ainda ser visto como uma máquina de estados que controla um conjunto específico de operações gráficas. Este modelo deve oferecer uma especificação que satisfaça as necessidades de programadores e implementadores. No entanto, não oferece necessariamente um modelo para implementação. Esta deve produzir resultados conformes aos produzidos pelos métodos específicos, podendo contudo, existir meios para executar uma computação particular de forma mais eficiente que a especificada.

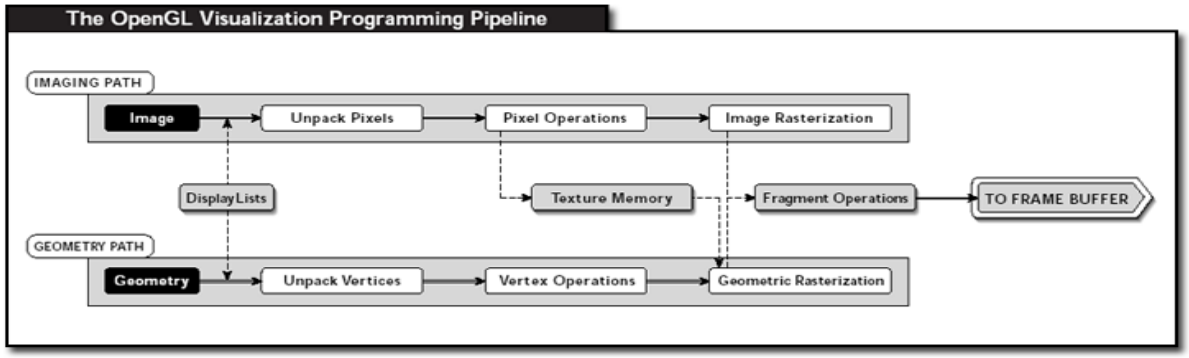

Figura 1 - Visualização do Pipeline do OpenGL [4].

3. Considerações de Design

O design de qualquer API requer um equilíbrio entre vários factores, tais como a simplicidade na realização de operações comuns *versus* generalização, ou muitos comandos com poucos argumentos *versus* poucos comandos com muitos argumentos. Nesta secção pretende-se descrever as considerações peculiares ao design de um API 3D que influenciaram o desenvolvimento do OpenGL.

Desempenho

Uma questão fundamental em gráficos tridimensionais interactivos é a do desempenho. São necessários inúmeros cálculos para renderizar um cenário 3D de modesta complexidade e, numa aplicação interactiva, um cenário é redesenhado várias vezes por segundo. Desta forma, um API desenvolvido para ser utilizado em aplicações 3D interactivas, deve oferecer um acesso eficiente às funcionalidades do *hardware* gráfico que utiliza. No entanto, diferentes subsistemas gráficos oferecem diferentes funcionalidades, pelo que deve ser encontrada uma interface comum.

A interface deve igualmente oferecer meios para a activação ou desactivação das várias funcionalidades de renderização. Este requisito é importante porque algum *hardware* pode não suportar algumas funcionalidades com um desempenho aceitável e mesmo com o suporte por *hardware*, a activação de certas funcionalidades, ou de combinações de funcionalidades, pode afectar significativamente o desempenho. Uma renderização lento pode ser aceitável na produção de uma imagem final de uma cena, mas quando há manipulação de objectos num cenário ou o ajuste de um ponto de vista, é necessário um desempenho razoável para tornar a interactividade positiva. Nestes casos, podem ser desejáveis funcionalidades com fraco desempenho para a imagem final mas são indesejáveis durante a manipulação de um cenário.

Ortogonalidade

Uma vez que é desejável a possibilidade de activar ou desactivar funcionalidades, é também importante que esta não tenha efeitos colaterais sobre outras funcionalidades. Se, por exemplo, é desejável que cada polígono possa ser desenhado com apenas uma cor em vez de uma interpolação de cores sob os seus lados, isto não deve afectar a forma como as texturas ou a iluminação são aplicadas. De forma similar, activar ou desactivar uma funcionalidade não deve conduzir a um estado inconsistente, no qual os resultados da renderização sejam indefinidos. Este tipo de independência de funcionalidades é necessária para permitir ao programador a sua fácil manipulação. Outro benefício da independência de funcionalidades é que estas podem ser combinadas de forma útil, gerando situações diferentes das idealizadas aquando da projecção da interface.

Completo

Um API gráfico 3D a correr num sistema com um subsistema gráfico deve oferecer alguns meios para aceder a grande parte das funcionalidades desse subsistema. Se uma funcionalidade disponível não for oferecida, o programador vê-se obrigado a recorrer a um API diferente para obter as funcionalidades que faltam. Isto pode inviabilizar uma aplicação por causa da interacção entre duas interfaces.

Da mesma forma, se a implementação de um API fornece certas funcionalidades numa plataforma de *hardware*, então estas devem estar presentes em qualquer plataforma, que ofereça o API. Caso esta regra seja quebrada, torna-se difícil usar um API num programa multi-plataforma sem saber quais as funcionalidades suportadas por cada plataforma. Em plataformas sem a aceleração apropriada, algumas funcionalidades podem ter um fraco desempenho mas, pelo menos, uma dada imagem deverá, eventualmente, aparecer.

Interoperabilidade

Muitos ambientes de computação consistem em vários computadores ligados em rede. Num ambiente deste tipo, é extremamente útil ser capaz de enviar comandos gráficos de uma máquina, para serem executados noutra (este é um dos principais factores do sucesso dos sistemas de janelas). Esta interoperabilidade requer que o modelo de execução dos comandos do API seja do tipo cliente-servidor: o cliente endereça comandos e o servidor executa-os. (Este conceito requer simultaneamente que cliente e servidor partilhem a noção, de como os comandos da interface são codificados, para transmissão através da rede.) Claro que cliente e servidor podem estar na mesma máquina.

Uma vez que os comandos da interface podem ser enviados através de uma rede, torna-se impraticável requerer grandes semelhanças entre cliente e servidor. Um cliente pode ter que esperar algum tempo por uma resposta a um pedido devido aos atrasos da rede. Assim pedidos simples ao servidor que não necessitam de reconhecimento podem ser guardados num *buffer* e enviados em grupos e executados pelo servidor, aumentando a eficiência da transmissão.

Extensibilidade

Uma interface de gráficos 3D deve ser extensível, para poder incorporar novas funcionalidades de hardware ou novos algoritmos tornados populares no futuro. Embora garantir que esta meta seja alcançada possa ser difícil antes de decorrido um período de tempo considerável, alguns passos podem ser dados para atingi-la. A ortogonalidade de um API, é um elemento que ajuda a alcançar este objectivo. Outro é simular a forma como a interface pode ser afectada pela introdução de funcionalidades extra.

Aceitação

Pode parecer que o desenvolvimento limpo e consistente de um API de gráficos 3D é um objectivo suficiente, em si mesmo. Contudo, a menos que os programadores decidam usar o API numa variedade de aplicações, o desenvolvimento deste não faz sentido. É portanto extremamente importante considerar os efeitos das decisões ao nível do design na aceitação por parte dos programadores.

4. Pipeline do OpenGL

Vértices e Primitivas

No OpenGL, a maior parte dos objectos geométricos são desenhados, englobando uma série de conjuntos de coordenadas que especificam vértices ou opcionalmente as normas (comprimento de um vector), coordenadas das texturas e cores entre o par de comandos `glBegin/glEnd`. Por exemplo, para especificar um triangulo com os vértices (0, 0, 0) e, (1, 0, 1), podíamos fazer da seguinte maneira:

```
glBegin(GL_POLYGON);
       glVertex3i(0,0,0);
       glVertex3i(0,1,0);
       glVertex3i(1,0,1);
glEnd();
```

Os dez tipos de objectos geométricos que são desenhados desta maneira estão sumariados na Tabela 1. Este grupo particular de objectos é seleccionado porque a geometria destes é especificada por uma simples lista de vértices, uma vez que cada um admite um algoritmo eficiente de renderização, e foi determinado que estes objectos combinados satisfazem as necessidades de quase todas as aplicações gráficas.

Cada vértice pode ser especificado por duas, três ou quatro coordenadas (quatro coordenadas indicam uma localização tridimensional homogénea). Adicionalmente, a norma, as coordenadas da

textura e a cor correntes, podem ser usadas para processar cada vértice. O OpenGL usa normais no cálculo de iluminações; a normal corrente é um vector tridimensional que pode ser definido enviando três coordenas que a especifiquem. As cores podem consistir em vermelho, verde, azul e valores alfa (quando o OpenGL foi inicializado para o modo RGBA), ou através do valor que define a cor. Uma, duas, três ou quatro coordenadas de texturas determinam como a textura de uma imagem é mapeada para uma primitiva.

Objecto	Interpretação dos Vertices
point	cada vértice descreve a localização de um ponto
line strip	séries de segmentos de recta ligados; cada vértice posterior descreve primeiro o ponto final do próximo segmento
line loop	igual ao *line strip* mas o segmento final é adicionado do ultimo vértice para o primeiro vértice
separate line	cada par de vértices descreve um segmento de linha
polygon	*o line loop* é formado por vértices da fronteira de um polígono convexo
triangle strip	cada vértice posterior aos dois primeiros descreve um triangulo dado por esse vértice, mais os dois anteriores
triangle fan	cada vértice posterior aos dois primeiros descreve um triangulo dado por esse vertice, mais o anterior e pelo primeiro vértice
separate triangle	cada trio de vértices descrevem um triangulo
quadrilateral strip	cada par de vértices posterior aos dois primeiros, descreve um quadrilátero, dado por esse par e pelo par anterior
independent quad	cada grupo consecutivo de 4 vértices descreve um quadrilátero

Tabela 1 - Objectos pertencentes ao **glBegin**/**glEnd** [1].

Cada comando que especifique as coordenadas, a normal, as cores, ou a textura de um vértice é disponibilizado em vários formatos para acomodar aos diferentes formatos de aplicações de dados e a números de coordenadas. Os dados podem ser transferidos para estes comandos como listas de argumentos ou como ponteiros para blocos de armazenamento que contenham dados. As variantes são distinguidas por (na linguagem C) mnemónicas de sufixos.

A maior parte dos comandos do OpenGL que não especifiquem vértices ou informação associada não podem aparecer entre o `glBegin` e o `glEnd`. Esta restrição permite às implementações executar de um modo optimizado enquanto são processadas simultaneamente as especificações de primitivas, de modo a garantir um processamento o mais eficiente possível [4].

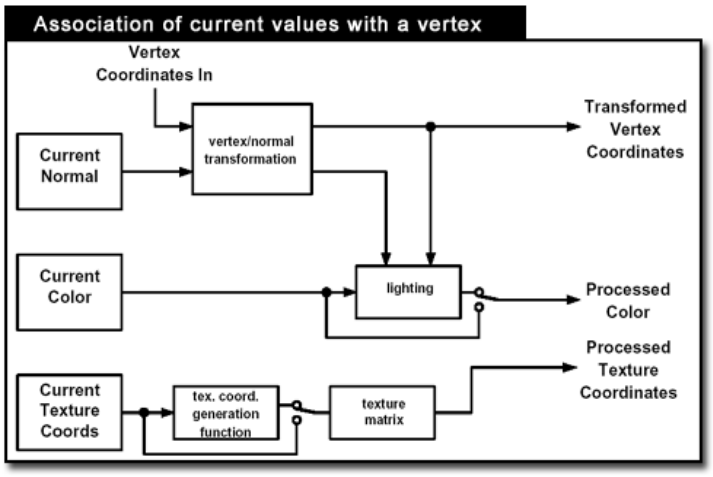

Figura 2 - Associação de valores correntes com um vértice [3].

Quando um vértice é especificado, a sua cor, a normal e as coordenadas da textura são usadas para obter valores que mais tarde serão associados ao vértice (Figura 2). O próprio vértice é transformado

pelo matriz de visualização do modelo (*model-view matrix*), a qual pode representar transformações translacionais e lineares. A cor é obtida através de cálculos da cor pelas iluminações ou, caso as iluminações estejam desactivadas, obtida pela própria cor. As coordenadas das texturas são passadas pela função de *texture coordinate generation*. As coordenadas das texturas resultantes são transformadas pela matriz de texturas (*texture matrix*) (esta matriz pode ser usada para redimensionar ou rodar a textura que é para ser aplicada a uma primitiva).

Existem grupos de comandos que controlam os valores dos parâmetros usados no processamento de vértices. Um desses grupos de comandos manipula transformações de matrizes. Estes comandos são desenhados de modo a formar uma maneira eficiente de gerar e manipular transformações que ocorrem numa hierarquia de cenários 3D. Uma matriz pode ser carregada ou multiplicada através de redimensionamento, rotação, translação ou por uma outra matriz. Outro comando controla qual a matriz afectada por uma manipulação: a matriz de visualização do modelo, a matriz de texturas ou a matriz de projecção (*projection matrix*). A cada um destes três tipos de matrizes está associada uma pilha, na qual as matrizes podem ser inseridas ou removidas (*push* ou *pop*).

Os parâmetros de iluminação são agrupados em três categorias: parâmetros de material, que descrevem as características dos reflexos da superfície iluminada; os parâmetros da fonte de iluminação, que descrevem as propriedades de emissão de cada fonte de luz; e os parâmetros do modelo de luz, que descrevem propriedades globais do modelo de luz. As iluminações são aplicadas com base nos vértices; os resultados das iluminações são interpolados por um segmento de recta ou por um polígono. A forma geral de uma equação de iluminação inclui termos constantes, de difusão e de iluminação focal, cada um deles pode ser atenuado pela distância do vértice da fonte de luz. O programador pode sacrificar o realismo em favor de cálculos para efeitos luminosos mais rápidos, indicando que quem vê a imagem, os pontos de luz ou ambos, estão a uma grande distância do cenário.

Projecção e Corte (*Projection and Clipping*)

Assim que uma primitiva é construída a partir de um grupo de vértices, esta é sujeita a um processo de corte através de superfícies de corte (*clip planes*). As posições destas superfícies (cada implementação em OpenGL tem de conter pelo menos seis) são especificadas usando o comando `glClipPlane`. Cada superfície pode ser activada ou desactivada individualmente.

No caso de um ponto, as superfícies de corte não têm qualquer efeito no ponto ou então simplesmente elimina-o, dependendo se o ponto está dentro ou fora da intersecção dos *half-spaces* determinados pelas superfícies de corte. No caso de se tratar de um segmento de recta ou de um polígono, as superfícies de corte podem, não ter qualquer efeito, eliminar, ou alterar o valor original da primitiva. No último caso, os novos vértices podem ser criados entre os cantos descritos pelos vértices originais; a cor e os valores das coordenadas das texturas para estes novos vértices são determinados interpolando os valores atribuídos relativamente aos vértices originais.

Após o processo das superfícies de corte ter sido aplicado (caso tenha sido necessário), as coordenadas dos vértices das primitivas resultantes são transformadas pela matriz de projecção. Ocorre então o *view frustum clipping*. O *view frustum clipping* é parecido à aplicação das superfícies de corte, mas para superfícies fixas: se as coordenadas depois de uma transformação são dadas por (x, y, z, w), então os seis *half-spaces* definidos por estas superfícies são $-w \leq x$, $x \leq w$, $-w \leq y$, $y \leq w$, $-w \leq z$, $z \leq w$.

Com a operação de *view frustum clipping* terminada, cada grupo de coordenadas de vértices é projectada calculando x/w, y/w e z/w. O resultado (que tem que estar entre [-1, 1]) é multiplicado e compensado pelos parâmetros que controlam o tamanho do *viewport*, onde as primitivas são desenhadas. Os comandos `glViewport` (para x/w e y/w) e `glDepthRange` (para z/w) controlam estes parâmetros.

Rasterização

O processo de rasterização converte uma primitiva *viewport-scaled* numa série de fragmentos. Cada fragmento contém a localização de um pixel no *framebuffer*, a cor, as coordenadas das texturas e a profundidade (Z). Quando um segmento de linha ou um polígono são rasterizados, os dados associados são interpolados através de primitivas de forma a obter o valor para cada fragmento [4].

A rasterização de cada tipo de primitiva é controlada por um correspondente grupo de parâmetros. Uma largura afecta a rasterização de um ponto e a outra afecta a rasterização de um segmento de linha. Adicionalmente, uma *stipple sequence* pode ser especificada para segmentos de recta e, um *stipple pattern* pode ser especificado para polígonos.

O *antialiasing* pode ser activado ou desactivado individualmente para cada primitiva. Quando for activado, um valor de cobertura é computado para cada fragmento descrevendo uma porção desse fragmento que é coberto pela primitiva projectada. Este valor de cobertura é usado depois do processo de texturização ter terminado, para modificar o valor do fragmento alfa (quando em modo RGBA) ou o valor do índice da cor (quando em modo de indexação de cor).

Texturização e Nevoeiro (*Texturing and Fog*)

O OpenGL fornece meios gerais para gerar primitivas mapeadas de texturas. Quando a texturização está activada, cada fragmento das texturas coordena a indexação de textura das imagens, gerando uma *texel*. Esta *texel* pode ter entre um a quatro componentes, de modo que a textura da imagem possa representar, por exemplo, apenas a intensidade (uma componente), cor RGB (três componentes), ou
RGBA (quatro componentes). Quando uma *texel* é obtida, ela modifica a cor do fragmento de acordo com a especificação do ambiente de texturização.

A imagem da textura é especificada usando o comando `glTexImage`, que recebe argumentos semelhantes aos do comando `glDrawpixels` de modo a que o mesmo formato da imagem possa ser usada com o *framebuffer* ou a memória de texturas (todas as placas gráficas têm alocada uma fracção de memória para as texturas). Adicionalmente, o `glTexImage` pode ser usado para especificar *mipmaps*, de modo que a textura possa ser filtrada à medida que é aplicada a uma primitiva. A função responsável pelo filtro é controlada por um número específico de parâmetros usando o comando `glTexParameter`. O ambiente da textura é seleccionado com o `glTexEnv`.

Finalmente, após o processo de texturização, a função do nevoeiro (se estiver activada) é aplicada a cada fragmento. Esta função mistura as cores que recebe com uma cor constante do nevoeiro de acordo com um factor ponderado computado. Este factor é a função da distância (ou uma aproximação à distância) do ponto de vista de quem vê até ao ponto tridimensional que corresponde ao fragmento. Nevoeiro exponencial simula nevoeiro atmosférico e neblina, enquanto o nevoeiro linear pode ser usado para produzir uma perda gradual de qualidade com a distância (*depth-cueing*).

O Framebuffer

O destino dos fragmentos rasterizados é o *framebuffer*, onde os resultados da renderização do OpenGL podem ser mostrados. No OpenGL, o *framebuffer* consiste num tabela rectangular de pixels correspondentes à janela destinada para a renderização OpenGL. Cada pixel é simplesmente um conjunto de alguns números de bits. Bits correspondentes a cada pixel no *framebuffer* são agrupados juntamente num *bitplane*, onde cada *bitplane* contém apenas um bit de cada pixel [4, 5].

Os *bitplanes* são agrupados em vários *buffers* lógicos: de cor, profundidade, *stencil*, e de acumulação. O *buffer* de cor é onde a informação sobre o fragmento da cor é colocado. No *buffer* de profundidade é colocado informação sobre o fragmento de profundidade. No *buffer* de *stencil* existem valores que são sempre actualizados quando o correspondente fragmento chega ao framebuffer. Os valores de *stencil* são úteis em algoritmos multi-passo, onde cada cenário é renderizado várias vezes, até

atingir um efeito como o das operações CGS (união, diferença e intersecção) num dado número de objectos e *capping* de objectos cortados por superfícies de corte.

O *buffer* de acumulação também é útil para algoritmos multi-passo e, pode ser manipulado de modo a calcular a média dos valores armazenados no *buffer* de cores. Isto pode aplicar efeitos como *antialiasing* no écran inteiro (agitando o ponto de vista a cada passagem), profundidade de campo (agitando o ângulo de visão), suavização de movimento (avançando o cenário no tempo). Algoritmos multi-passo são simples de implementar em OpenGL, simplesmente porque um pequeno número de parâmetros têm de ser manipulados antes de cada passagem, e alterar os valores destes parâmetros é eficaz e não traz efeitos secundários aos outros parâmetros que têm de permanecer constantes.

O OpenGL suporta *buffering* duplo e em estéreo (*double-buffering* e *stereo-buffering*), de modo que o *buffer* de cor seja subdividido em quatro *buffer*: os *buffers* de frente, esquerdo e direito e os *buffers* traseiros, esquerdo e direito. Os *buffers* frontais são aqueles que são tipicamente mostrados enquanto os *buffers* traseiros (numa aplicação com *double-buffering*) são usados para compor a próxima frame. Uma aplicação *monoscopic* usaria apenas *buffers* esquerdos. Adicionalmente, podem existir *buffers* auxiliares (que nunca são mostrados) para onde os fragmentos podem ser renderizados. Cada um dos *buffers* pode ser activado ou desactivado individualmente para escrita de fragmentos [4].

Uma cópia particular do OpenGL pode não dispor de *buffers* de profundidade, *stencil*, acumulação ou *buffers* auxiliares. Apenas alguns subconjuntos do *buffer* frontal esquerdo e direito e do *buffer* traseiro esquerdo e direito é que podem estar presentes. Diferentes tipos de *buffers* podem estar disponíveis (dependendo cada um do número de bits) dependendo da plataforma e do sistema de janelas onde o OpenGL está a executar. Todos os sistemas de janelas têm, no entanto, de fornecer pelo menos uma janela com um *buffer* frontal (esquerdo) de cor, de profundidade, *stencil* e de acumulação. Isto garante um mínimo para a configuração que um programador pode assumir que está presente independentemente onde o programa OpenGL se encontra.

Operações por fragmento

Antes ser colocado na localização correspondente do *framebuffer*, um fragmento é sujeito a vários testes e modificações, as quais podem ser individualmente activados, desactivados ou controlados. Os testes e modificações incluem o teste de *stencil*, o teste de profundidade do *buffer* (é tipicamente utilizado para remover superfícies escondidas), e misturas.

O teste de *stencil*, quando activado, compara o valor no *stencil buffer* correspondente ao fragmento com o valor de referência. Se a comparação é bem sucedida, então o valor de *stencil* guardado pode ser alterado por uma função que incrementa, decrementa ou limpa, avançando o fragmento para o teste seguinte. Se o teste falhar, o valor guardado é actualizado por uma função diferente e o fragmento é descartado. De forma similar, o teste de profundidade do *buffer* compara o valor da profundidade do fragmento com o correspondente valor guardado no *buffer* de profundidade. Se a comparação tiver sucesso, o fragmento, é passado para a fase seguinte sendo os valores do *buffer* de profundidade substituídos pelo valor guardado no *buffer* de profundidade (se este *buffer* estiver activo para escritas). Caso a comparação falhe, o fragmento é descartado, não havendo alterações no *buffer* de profundidade.

A mistura (*blending*) une a cor de um fragmento com a cor correspondente já guardada no framebuffer (a mistura ocorre uma vez sempre que um *buffer* de cor é activado para escrita). A função de mistura pode ser especificada por `glBlendFunction`.

Misturar é uma operação que permite alcançar o *antialiasing* para cores RGBA. Se lembrarmos que a computação de cobertura apenas altera o valor alfa do fragmento, vemos que este valor deve ser usado para misturar a cor do fragmento com a cor de fundo já guardada, de forma a obter o efeito de *antialiasing*. Este efeito é também usado para alcançar transparências

5. Integração num Sistema de Janelas

O OpenGL desenha cenários tridimensionais e bidimensionais para um *framebuffer*, mas para ser útil num ambiente heterogéneo, o OpenGL deve ser subordinado a um sistema de janelas que aloca e controla os recursos do *framebuffer*. Na descrição que se segue mostra-se como o OpenGL é integrado no *X Window System*, mas a integração noutro sistema de janelas (por exemplo, Microsoft Windows NT®) é semelhante.

O X oferece uma interface procedimental e um protocolo de rede para criação e manipulação das janelas do *framebuffer* e desenho de certos objectos bidimensionais nestas janelas. O OpenGL é integrado no X através de uma extensão formal do X, designada por GLX. GLX consiste em dezenas de chamadas (com a correspondente codificação para rede) que oferece uma integração compacta e geral do OpenGL em X. Tal como noutras extensões de X (dois exemplos são o *Display PostScript* e o PEX), há um protocolo específico de rede para os comandos de *rendering* do OpenGL, encapsulados no X fluxo de bytes.

OpenGL requer uma região de um *framebuffer* na qual as primitivas possam ser *rendered*. Em X, esta região designa-se *drawable*. Uma janela, do tipo *drawable*, tem associada um visual que descreve a configuração do *framebuffer* da janela. Em GLX, o visual é uma estrutura que inclui informação acerca dos *buffers* de OpenGL que não estão presentes no X (profundidade, acumulação, etc.).

O X oferece ainda um segundo tipo de *drawable*, o *pixmap*, que não é mais do que um *framebuffer* fora do monitor. GLX oferece um *GLX pixmap* que corresponde a um *X pixmap*, mas com *buffers* adicionais. O *GLX pixmap* fornece meios, às aplicações OpenGL, para a renderização fora do monitor num *buffer* criado por software.

Para utilizar um *drawable* de OpenGL, o programador tem que criar um contexto tendo como alvo esse *drawable*. Quando o contexto é criado, é inicializada uma cópia da renderização do OpenGL com a informação visual do *drawable*. Esta renderização é conceptualmente (se não verdadeiramente) parte do servidor de X, de forma que, uma vez criado, um cliente X pode ligar-se ao contexto do OpenGL e enviar comandos OpenGL (Figura 3). Múltiplos contextos podem ser criados para os mesmos ou diferentes *drawable*'s. Qualquer *drawable* de OpenGL pode também ser normalmente usado no desenho de X. As chamadas são fornecidas para sincronizar imagens entre o OpenGL e o X. É da responsabilidade do cliente efectuar esta sincronização caso seja necessária.

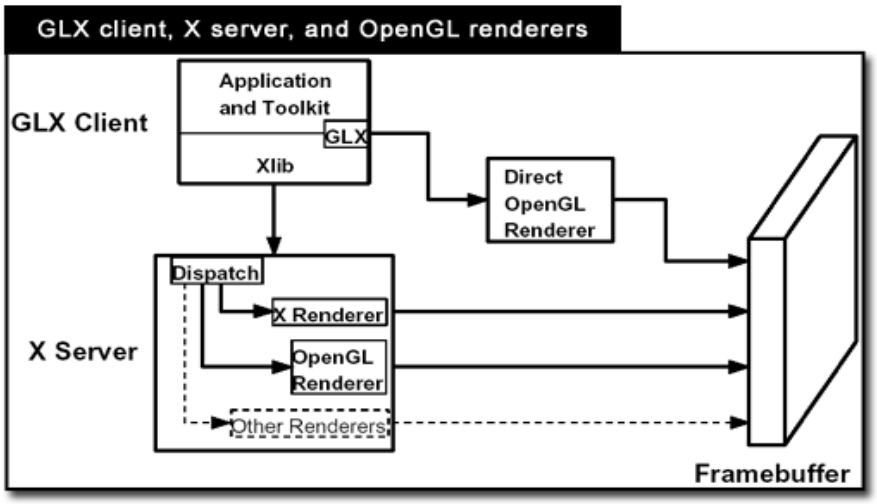

Figura 3 - Renderização do OpenGL, Cliente GLX e Servidor X [1].

Um cliente de GLX que se encontra a correr num computador de cujo subsistema gráfico faz parte, pode evitar passar *tokens* para o servidor de X. Este direct *rendering* pode resultar num aumento do desempenho gráfico uma vez que, o *overhead* da codificação, descodificação e envio do *token* é eliminado. A renderização directa é suportada embora não seja um requisito do GLX (um cliente pode determinar se o servidor deve ou não fornecer a renderização directa). A renderização directa é possível, uma vez que a sequência entre os comandos de X e de OpenGL não necessita de ser mantida, excepto quando os comandos são explicitamente sincronizados.

6. Exemplo: Três Tipos de Texto

Para ilustrar a flexibilidade do OpenGL no desempenho de diferentes tipos de tarefas de rendering, mostramos três métodos para a visualização de texto. Os três métodos são: a utilização de bitmaps, a utilização de segmentos de recta para gerar texto de contorno e a utilização de texturas para gerar texto suavizado (*antialiased*) [6].

O primeiro método define a fonte como uma série de *display lists*, cada uma contendo um único *bitmap*:

```
for i = start + 'a' to start + 'z'
{
	glBeginList(i);
		glBitmap( . . . );
	glEndList();
}
```

`glBitmap` especifica simultaneamente um ponteiro para a codificação do *bitmap* e o offset que indica a posição relativa deste em relação aos *bitmaps* anterior e seguinte. Em GLX, o efeito de se definir um dado número de *display lists*, desta forma pode também ser alcançado através da função `glXUseXFont`. Esta função gera um número de *display lists*, contendo cada uma um *bitmap* (e os *offset*'s associados) de um caractere da fonte X especificada. Em ambos os casos, a *string* "Bitmapped Text" cuja origem é a projecção da posição em 3D produzida por:

```
glRasterPos3i(x, y, z);
glListBase(start);
glCallLists("Bitmapped Text", 14, GL_BYTE);
```

Ver figura 4. `glListBase` define uma *display list* de forma que as funções `glCallLists` chamadas posteriormente, referem os caracteres definidos. O segundo argumento desta função indica o comprimento da *string* e o terceiro refere que a *string* é um *array* de 8-bit bytes (podem também ser usados inteiros de 16 e 32 bits, para aceder a fontes com mais de 256 caracteres).

O segundo método, é similar ao primeiro embora use segmentos de recta para desenhar cada caractere. Cada *display list* contém uma série de segmentos de recta:

```
glTranslate(ox, oy, 0);
glBegin(GL_LINES);
	glVertex(…);
		…
glEnd();
glTranslate(dx-ox, dy-oy, 0);
```

A função `glTranslate` inicial actualiza a matriz de transformação para posicionar o caractere relativamente á origem. O `glTranslate` final actualiza a posição inicial do caractere de forma a preparar o carácter seguinte. A string é visualizada com este método da mesma forma que no exemplo anterior, mas como os segmentos de recta têm posições tridimensionais, o texto pode ser orientado e posicionado em três dimensões (Figura 4b). Geralmente, as *display lists* podem conter polígonos e segmentos de recta, podendo estes ser *antialiased*.

Finalmente, surge como uma abordagem diferente a criação de uma imagem de textura contendo um *array* de caracteres. Os diferentes alcances das coordenadas da textura, correspondem aos vários caracteres na imagem. Cada carácter, pode ser desenhado em qualquer tamanho e em qualquer orientação tridimensional, desenhando um rectângulo, com as coordenadas de textura apropriadas nos seus vértices:

```
glTranslate(ox, oy, 0);
glBegin(GL_QUADS)
       glTextCoord(…);
       glVertex(…);
              …
glEnd();
glTranslate(dx-ox, dy-oy, 0);
```

Se cada grupo de comandos para cada caractere for incluído numa *display list*, e os comandos para descrever a imagem de textura em si, forem incluídos noutra *display list* chamada TEX, então a string "Texture Mapped Text!" pode ser exibido por:

```
glCallList(TEX);
glCallLists("Bitmapped Text", 22, GL_BYTE);
```

Uma vantagem deste método, é que, usando simplesmente o filtro de textura apropriado, os caracteres resultantes são suavizados (Figura 4c).

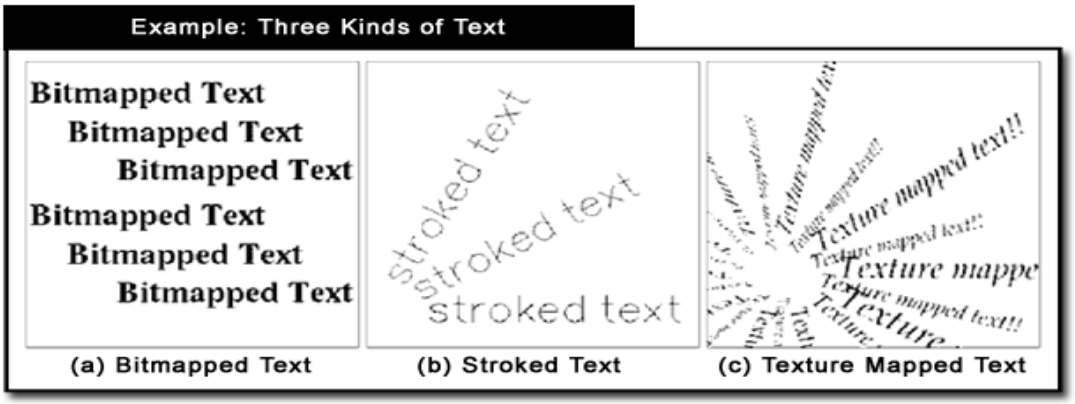

Figura 4 - Exemplo: Três Tipos de Texto [1, 6].

7. Silicon Graphics Inc. OpenGL *versus* Microsoft Direct3D

Ambos, o OpenGL e o Direct3D, mostram um tradicional *pipeline* para a renderização gráfica. A arquitectura do *pipeline* tem estado presente desde muito cedo na computação gráfica e tem sido melhorada e aumentada ao longo dos anos, com o aumento das capacidades do hardware, contudo o paradigma fundamental não se alterou.

Uma comparação entre estes dois API's, não é algo fácil e objectivo. Tentamos aqui mostrar, principalmente as diferenças para que seja possível ao leitor escolher.

Um dos aspectos muito citados na comparação destas duas interfaces é, como não podia deixar de ser, o desempenho. Este não é, no entanto, um aspecto importante. O desempenho de ambos os API's é praticamente idêntico para programas bem escritos, independentemente da interface. A performance apenas pode ser medida em função da máquina. No caso de programação de jogos, se um jogo está bem escrito, por programadores com um conhecimento profundo do API, o desempenho fica por conta dos fabricantes das placas gráficas e dos *drivers*. O que determina a performance, é a forma como API, *driver*, sistema operativo e hardware interagem [7].

O mesmo se aplica quando falamos de qualidade. O factor mais importante na qualidade é a capacidade do artista. O bom artista pode fazer um mau jogo parecer fantástico, mas o inverso também se aplica.

A capacidade de renderização por software é outra das falsas questões. A renderização por software adiciona funcionalidades que o hardware não implementa. Como se pode imaginar, estas funcionalidades têm de ser escritas pelo programador, o que adiciona um enorme *overhead* ao programa. Como a renderização corre juntamente com o programa, o desempenho é muito penalizado. Adicionando o facto de que, escrever um bom software de renderização, é uma tarefa extremamente difícil. Embora haja algumas vantagens na renderização por software, elas são muito poucas [8].

A batalha num ambiente Windows entre Direct3D e OpenGL é grande. Se pretender escrever programas para plataformas que não o Windows, então a escolha é simples, uma vez que o Direct3D não corre nessas plataformas. Se pretende escrever programas para Windows, então a decisão é difícil. Os API's são semelhantes por isso, se conhecer um, facilmente aprender o outro, já que os conceitos fundamentais são os mesmos. Se não tem uma preferência, então procure uma moeda. Não interessa o tipo ou valor, basta que tenha dois lados diferentes. Escolha um para ser "cara" e o outro "coroa". Atire a moeda ao ar e se for "cara" escolha o OpenGL, se for "coroa" opte por Direct3D. No entanto, seja qual for o API que escolha mantenha-se fiel. A programação gráfica não é fácil de aprender. Não importa questionar-se se seria melhor o outro, porque no fim, verdadeiramente, não interessa.

Para ajudar numa eventual escolha, baseada em métodos menos aleatórios segue-se a tabela 2, que não pretende ser exaustiva, das características de cada um dos API's:

Funcionalidade	OpenGL 1.2 Core	Direct3D 7	Direct3D 8
System Mechanics			
Operating System Support	Windows (9x, NT, 2000), MacOS, BeOS, *nix, others	Windows (9x, 2000, CE)	Windows (9x, 2000)
API Definition Control	OpenGL ARB	Microsoft	Microsoft
API Specification	OpenGL Specification	SDK/DDK Documentation and DDK Reference	SDK Documentation
API Mechanism	includes and libraries	COM	COM
Software Emulation of Unaccelerated Features	Sim	Não	Não
Extension Mechanism	Sim	Não	Sim
Source Implementation Available	Sim	Sim	Não
Modeling			
Fixed-Function Vertex Blending	Não	Sim	Sim
Programmable Vertex Blending	Não	Não	Sim
Parametric Curve Primitives	Sim	Não	Sim
Parametric Surface Primitives	Sim	Não	Sim
Hierarchical Display Lists	Sim	Não	Não
Rendering			
Two-sided Lighting	Sim	Não	Não
Point Size Rendering Attributes	Sim	Não	Sim
Line Width Rendering Attributes	Sim	Não	Não
Programmable Pixel Shading	Não	Não	Sim
Triadic Texture Blending Operations	Não	Não	Sim
Cube Environment Mapping	Não	Sim	Sim
Volume Textures	Sim	Não	Sim
Multitexture Cascade	Não	Sim	Sim
Texture Temporary Result Register	Não	Não	Sim
Mirror Texture Addressing	Não	Sim	Sim
Texture "Wrapping"	Não	Sim	Sim
Range-Based Fog	Não	Sim	Sim
Bump Mapping	Não	Sim	Sim
Modulate 2X Texture Blend	Não	Sim	Sim
Modulate 4X Texture Blend	Não	Sim	Sim
Add Signed Texture Blend	Não	Sim	Sim
Frame Buffer			
Hardware Independent Z Buffer Access	Sim	Não	Não
Full-Screen Antialiasing	Sim	Sim	Sim
Motion Blur	Sim	Não	Sim
Depth of Field	Sim	Não	Sim
Accumulation Buffers	Sim	Não	Não
Miscellaneous			
Picking Support	Sim	Não	Não
Multiple Monitor Support	Não	Sim	Sim
Stereo Rendering	Sim	Sim	Não

Tabela 2 – Diferenças entre OpenGL e Direct3D [9].

8. Conclusão

OpenGL é um API para gráficos 3D orientado para a utilização em aplicações interactivas. Desenvolvida para oferecer máximo acesso às capacidades do hardware gráfico, é uma interface procedimental extremamente flexível que permite o controlo directo sobre operações fundamentais. O OpenGL não limita a descrição de objectos 3D, a um método particular ou à forma como estes devem ser visualizados. Oferece antes, os meios básicos para o renderizar desses objectos, independentemente da forma como são descritos. Esta característica torna o OpenGL independente da plataforma.

Impondo uma estrutura mínima na renderização 3D, este API torna-se numa excelente base para o desenvolvimento de bibliotecas destinadas à manipulação de objectos geométricos estruturados, independentes da estrutura particular de cada um. Desta forma, uma biblioteca que usa OpenGL herda a independência relativamente à plataforma, tornando-a útil a um grande número de programadores.

Objectivos de elevada performance, funcionalidades ortogonais, interoperabilidade, possibilidade de implementação em vários sistemas e extensibilidade, conduziram o desenvolvimento do OpenGL. O resultado directo foi um API generalista e com grande facilidade de utilização em diferentes aplicações.

É provável, que o trabalho futuro em OpenGL, se centre em melhorar implementações e estender a interface, para lidar com as novas técnicas e funcionalidades oferecidas pelo hardware gráfico. A forma cuidada e sustentada como o desenvolvimento deste API tem sido feito, resultará provavelmente num API de gráficos 3D útil e interessante por muitos anos.

Agradecimentos

Os autores agradecem aos pais e irmãos que muito ajudaram e a todos os colegas que os incentivaram e ajudaram na pesquisa e elaboração do artigo.

Referências

[1] Mark Segal e Kurt Akeley. The Design of the OpenGL® Graphics Interface, 1994.
[2] Silicon Graphics Inc.. OpenGL® Datasheet, 1998.
[3] Silicon Graphics Inc: http://www.sgi.com/
[4] Mark Segal e Kurt Akeley. The OpenGL® Graphics System: A Specification (Version 1.4), 2002.
[5] nVIDIA: 3D Glossary - http://www.nvidia.com/glossary.asp
[6] OpenGL - High Performance 2D-3D Graphics: http://www.opengl.org/
[7] GameDev.net: http://www.gamedev.net/
[8] Promit Roy. Direct3D vs. OpenGL: Which API to Use, When, Where and Why, 2002.
[9] Direct3D vs. OpenGL: A Comparison: http://www.xmission.com/~legalize/d3d-vs-opengl.html