IA725 – Computação Gráfica I

Hardware Gráfico

Harlen Batagelo

FEEC/DCA

Visão geral

- Cronologia da evolução do hardware gráfico
  - Sistemas compatíveis com APIs GL
  - Comparação de desempenho, qualidade visual, custo e recursos
- Arquitetura do hardware gráfico programável
  - Integração com o fluxo de visualização tradicional
  - Arquitetura do hardware gráfico programável
    - *Processador de vértices*
    - *Processador de fragmentos*
  - Introdução à programação de *shaders*
  - Estado-da-arte e novas gerações

Em 25 anos...

1983: SGI Iris 1000

30 mil vértices/segundo
40 milhões pixels/segundo

2008: NVIDIA GeForce 9800 GX2

5 bilhões vértices/segundo
76.8 bilhões pixels/segundo

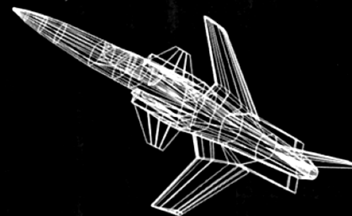
Parâmetros de comparação

- Desempenho de processamento, qualidade da imagem e recursos implementados
- Desempenho:
  - Vértices processados por segundo (*vertex rate*)
  - Fragmentos de *pixels* ou *texels* processados por segundo (*fill rate*)
- Qualidade da imagem:
  - Resolução, *bits* por *pixel*, *multisampling*, filtragem de texturas
- Conjunto de recursos:
  - Transformação geométrica, iluminação, texturização, remoção de superfícies escondidas, *culling*, *tesselation* e programação de *shaders*

Evolução do hardware gráfico

- Gerações de hardware gráfico não são definidas necessariamente pelo desempenho ou qualidade da imagem
  - GFX 5200 é mais lenta que uma G4 Ti 4200
  - Qualidade de imagem é um referencial subjetivo
- Divisão em 4 gerações segundo recursos implementados:
  - 1) Transformação geométrica (1983--1987)
  - 2) Triângulos sombreados e iluminação (1987--1992)
  - 3) Texturização (1992--2000)
  - 4) Programação de *shaders* (2000--hoje)

1983-1987: 1ª geração



- Transformação geométrica
- SGI IRIS (*Integrated Raster Imaging System*) 1000, 2000 e 3000
- Processamento de vértices
  - Transformação geométrica, recorte e projeção
- Processamento de fragmentos
  - Rasterização de linhas com interpolação de cores
  - Rasterização de triângulos com cores sólidas
- Sem *frame buffer*

IRIS 1000

1987-1992: 2ª geração

- Triângulos sombreados e iluminação
- SGI GTX
- Processamento de vértices
  - Avaliação da equação de iluminação
- Processamento de fragmentos
  - Interpolação de cores (difusa e especular) e valores de profundidade
- Características do *frame buffer*
  - *Z-buffer* e *alpha blending*

4D 210 GTX

1992-2000: 3ª geração

- Texturização
- SGI RealityEngine, de US$1.000.000,00
- Processamento de vértices
  - Transformação de coordenadas de texturas
- Processamento de fragmentos
  - Amostragem de texturas
  - Texturas 3D
  - *Fog*
- Características do *frame buffer*
  - *Buffer* estêncil, *Antialiasing*

Onyx 1

2000-Hoje: 4ª geração

- Arquitetura programável
- NVIDIA GeForce 3
- Processamento de vértices
  - Programação do fluxo de vértices
- Processamento de fragmentos
  - Programação do fluxo de fragmentos
- Características do *frame buffer*
  - *Buffers* de ponto flutuante
  - Renderização em texturas
  - Renderização simultânea em múltiplos *buffers*

GeForce FX 5200

Evolução do hardware gráfico

- Diferentes rumos de evolução entre sistemas SGI e PCs.
- SGIs:

- PCs:

Evolução do hardware gráfico

- Tempo médio de duplicação do desempenho:
  - 18 meses para CPUs (*Lei de Moore*)
    - Aumento de ~60x em 10 anos
  - Até 6 meses para GPUs
    - Aumento de ~1000x em 10 anos
- Redução no custo e dimensões do hardware:
  - 1992: SGI RealityEngine, US$1.000.000,00, do tamanho de um pequena geladeira
  - 2001: NVIDIA GeForce, US$100,00, cabe num *slot* de PC
- Além do desempenho, novos recursos (como já vimos)
- Desempenho + recursos = qualidade da imagem

Evolução do hardware gráfico

- Comparação da qualidade de imagem de uma NVIDIA Riva 128 (1997) com uma GeForce FX (2003)

NVIDIA Riva 128

x

NVIDIA GeForce FX

Moto Racer

3D Mark '03

Qualidade da imagem (Riva 128)

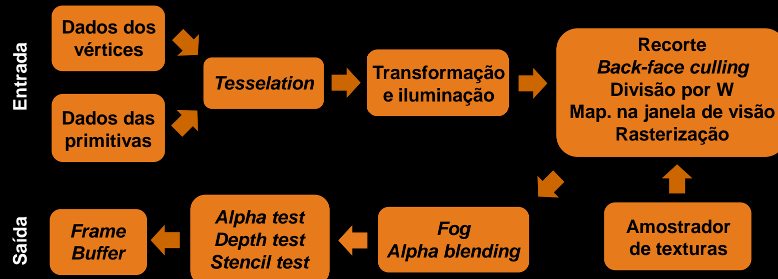
Moto Racer (Electronic Arts, Delphine Software, Inc., 1997)

Qualidade da imagem (GeForce FX)

3D Mark 2003, Mother Nature (Futuremark, Inc., 2003)

Arquitetura do hardware gráfico

- Diagrama de fluxo de visualização do hardware gráfico não-programável (1995-2000):

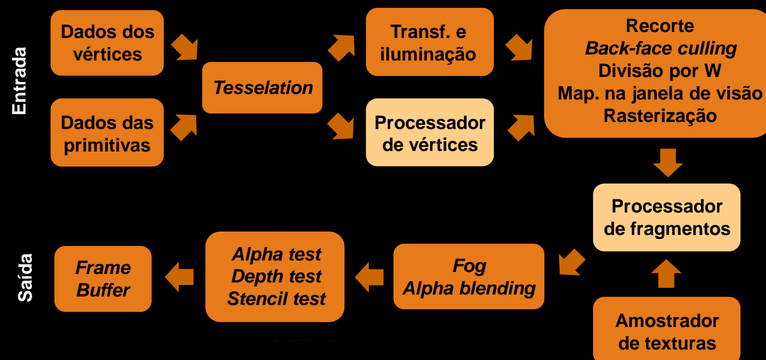


Arquitetura do hardware gráfico

- Principal mudança com o advento das GPUs (*Graphics Processing Units*): Inclusão de processadores programáveis de vértices e fragmentos
- Os programas são chamados de *shaders* – termo proveniente do RenderMan, da Pixar
  - *Vertex/pixel shaders*, de acordo com o processador utilizado
- A programação de shaders substitui apenas uma parte das tarefas do fluxo de visualização
  - Processador de vértices substitui a etapa de transformação e iluminação
  - Processador de fragmentos adiciona funcionalidades à etapa de rasterização e amostragem de texturas

Arquitetura do hardware gráfico

- Diagrama de fluxo de visualização do hardware gráfico atual:



Arquitetura do hardware gráfico

- *Vertex shader*:
  - Programa executado para cada vértice
- Entrada:
  - Um vértice (normalmente no espaço do objeto)
  - Pode conter informações tais como: normal, cor, coordenadas de textura, tamanho do ponto, etc
- Saída:
  - Vértice em espaço homogêneo de recorte

Arquitetura do hardware gráfico

- O que um *vertex shader* pode fazer:
  - Transformação de vértices (mudança de base, rotação, translação, etc)
  - Normalização de vetores
  - Cálculo de iluminação por vértice
  - Geração e modificação de coordenadas de textura
- O que um *vertex shader* não pode fazer:
  - Montagem de primitivas
  - *Frustum culling, back-face culling*
  - Divisão perspectiva
  - Mapeamento na janela de visão

Arquitetura do hardware gráfico

- Diagrama do ambiente de execução de um *vertex shader*:

Arquitetura do hardware gráfico

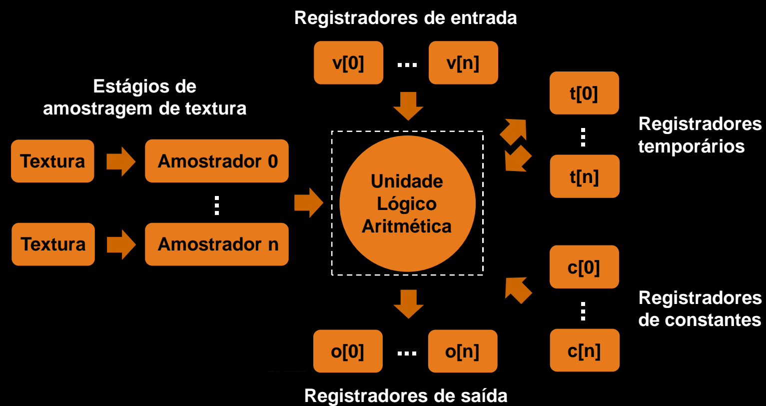
- *Fragment shader*:
  - Programa executado para cada fragmento
- Entrada:
  - Um fragmento interpolado segundo os modelos *flat* ou *smooth*
  - Inclui as informações (também interpoladas) da saída do *vertex shader*
- Saída:
  - Cor (RGBA)
  - Profundidade

Arquitetura do hardware gráfico

- O que um *fragment shader* pode fazer:
  - Amostragem e combinação de texturas
  - *Bump mapping*, *environment mapping*, cálculo de *fog* e uma infinidade de outras operações envolvendo cores
- O que um *fragment shader* não pode fazer:
  - Mudar o modelo de interpolação
  - Realizar teste alfa, teste de profundidade, teste do buffer estêncil
  - Habilitar ou desabilitar o modo de *dithering*

Arquitetura do hardware gráfico

- Diagrama do ambiente de execução de um *fragment shader*:

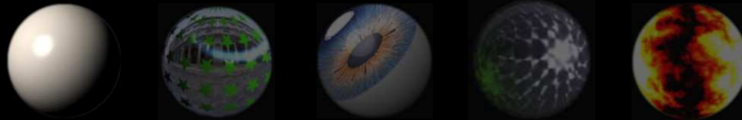


Arquitetura do hardware gráfico

- Exemplos de alguns *shaders* aplicados sobre esferas:

Arquitetura do hardware gráfico

- Exemplos de alguns *shaders* aplicados sobre esferas:



Phong shading

Avaliação da equação de iluminação de Phong (Phong, 1973):
Iluminação = luz ambiente + luz difusa + luz especular
$I = I_a k_a + I_i k_d \cos\theta + I_i k_s \cos^n \Phi$

Arquitetura do hardware gráfico

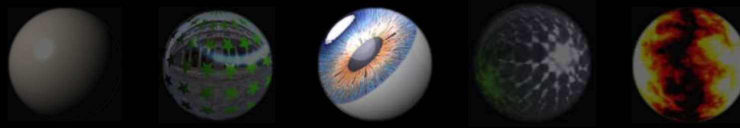
- Exemplos de alguns *shaders* aplicados sobre esferas:

Environment mapping com multitextura

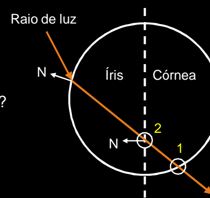
Amostragem de textura com *cube mapping*
Combinação com uma textura planar (estrelas verdes)

Arquitetura do hardware gráfico

- Exemplos de alguns *shaders* aplicados sobre esferas:



Olho renderizado com traçado de raios

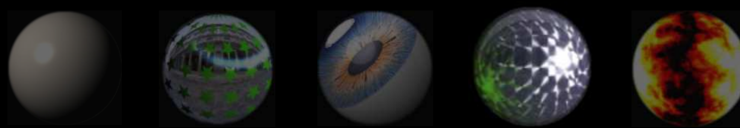


1) Interseção raio/esfera:
O raio saiu atrás da córnea?

2) Interseção raio/plano:
Obtém cor da íris

Arquitetura do hardware gráfico

- Exemplos de alguns *shaders* aplicados sobre esferas:



Metal escovado

Reflexão anisotrópica controlada por uma textura
Modelo de iluminação de Phong

Arquitetura do hardware gráfico

- Exemplos de alguns *shaders* aplicados sobre esferas:

Simulação de fogo

Ruído de Perlin em tempo real com número variável de oitavas
(Ken Perlin, 1982)
http://mrl.nyu.edu/~perlin/

Arquitetura do hardware gráfico

- Exemplo de um *vertex shader* que simula o fluxo de função fixa para uma fonte de luz direcional e uma textura:

```
m4x4 oPos, v0, c0   ; Transforma posição do vértice por WVP (em c0 até c3)
m4x4 r1, v1, c4     ; Transforma a normal ao sistema de ref. do mundo
                    ; c4 até c7 contém a inversa transposta da matriz W

dp3 r1.w, v1, v1    ; Normaliza vetor normal (v1)
rsq r1.w, r1.w      ; r1.w = 1/sqrt(r1.w)
mul r1, r1, r1,w

mov r2.xyz, c8      ; Normaliza direção da luz (c8)
dp3 r2.w, c8, c8
rsq r2.w, r2.w
mul r2, r2, r2.w
```

Arquitetura do hardware gráfico

- Continuação:

```
; Calcula a iluminação
; Cor = ambiente + difusa * max(0, dot(normal, direção da luz))
dp3 r3.x, r1, r2
max r3.x, r3.x, c11.x
mad oD0, c9, r3,x, c10     ; oD0 = c9*r3.x + c10

; Passa adiante as coordenadas de textura, sem modificação
mov oT0, v2
```


Linguagens de alto nível

- Todas as propostas são parecidas com C
- *Stanford real-time shading language*
  - Modelo unificado, parecido com o RenderMan
- NVIDIA Cg
  - Multiplataforma
- Microsoft HLSL
  - Semanticamente e sintaticamente equivalente ao Cg
  - Descrição de "efeitos" (conjuntos de *shaders*)
  - Depurador integrado com o VS .NET

Linguagens de alto nível

- GLSL (*OpenGL Shading Language*)
  - Integrado com o OpenGL 2.0
  - Simula em software os recursos não disponíveis em hardware
- BrookGPU (http://graphics.stanford.edu/projects/brookgpu/)
  - Utilização da GPU em programação de propósito geral
  - Modelo de programação orientada a objetos
  - Dois objetos principais:
    - *Stream*: fluxo de informações
    - *Kernel*: função aplicada a cada informação
  - Independência da API

Exemplo usando Cg

- Conversão do *vertex shader* mostrado anteriormente:

```
// Declara entrada do vertex shader
struct IN {
   float4 pos        : POSITION;
   float3 normal    : NORMAL;
   float2 texcoord : TEXCOORD0;
};

// Declara saída do vertex shader
struct OUT {
   float4 pos        : POSITION;
   float4 color       : COLOR0;
   float2 texcoord : TEXCOORD0;
};
```

Exemplo usando Cg

```
OUT main( IN input,
          uniform float4x4 worldViewProj,
          uniform float4x4 invTransWorld,
          uniform float3 lightDir,
          uniform float4 diffuseColor,
          uniform float4 ambientColor )
{
   OUT output;

   // Transforma posição do vértice em espaço homogêneo de recorte
   output.pos = mul(worldViewProj, input.pos);

   // Transforma normal do sistema de ref. local para sistema do mundo
   float3 worldNormal = mul((float3x3)invTransWorld, input.normal);

   // Normaliza normal e direção da luz
   worldNormal = normalize(worldNormal);
   float3 worldLightDir = normalize(lightDir);
```


Exemplo usando Cg

```
   // Calcula iluminação
   float diffuse = max(0, dot(worldNormal, worldLightDir));
   output.color = ambientColor + diffuse * diffuseColor;

   // Passa adiante as coordenadas de textura
   output.texcoord = input.texcoord;

   return output;
}
```

Vídeo

- Implementação do trabalho de Paul Debevec sobre iluminação HDR (*High Dynamic Range*)
  - *Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography* (SIGGRAPH 98, Julho/1998.)
- Iluminação da cena definida por uma textura de ambiente
- Integração com objetos de diferentes materiais

http://developer.amd.com/Downloads/ATI-9700-DebevecRNL-Demo-v1.1.exe
http://developer.amd.com/media/gpu_videos/ATI-9700-rnl-Movie-v1.0.mpg

Estado-da-arte e próximas gerações

- Arquitetura unificada
  - Sob demanda, processador pode atuar como processador de vértices ou de fragmentos
- Processador de primitivas
  - Geração e destruição de primitivas na GPU
  - Acesso e modificação de informações de conectividade da geometria
  - Característica já suportada, com limitações, em *hardware* gráfico atual (ex.: NVIDIA GeForce 8 e posteriores)
- Capacidade de manter informações sobre o relacionamento dos objetos numa cena 3D (grafo de cena)
- Implementação completa da especificação RenderMan na GPU