Introdução ao Hardware Gráfico

Marcelo Walter
UFPE

Muitos slides "roubados" do colega Christian Hofsetz :-)

Atualização maio/2008

Qual o principal impacto da introdução das GPUs?

2

Qual o principal impacto da introdução das GPUs?

- SLI
- LMA
- HyperZ
- GART
- GTS

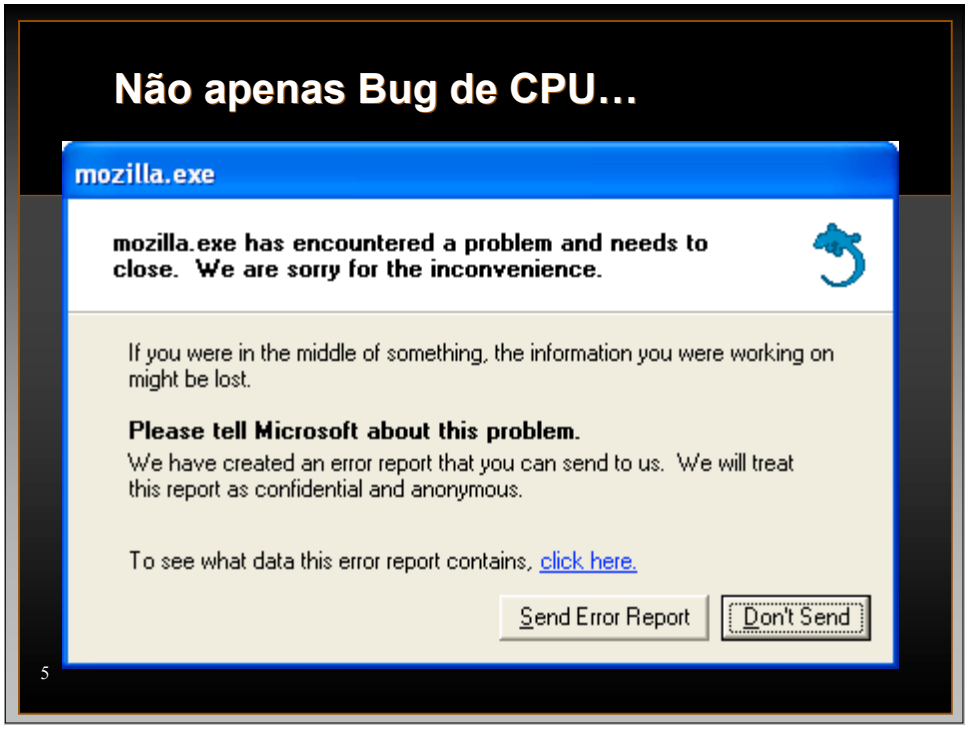
Mais Siglas!! :-)

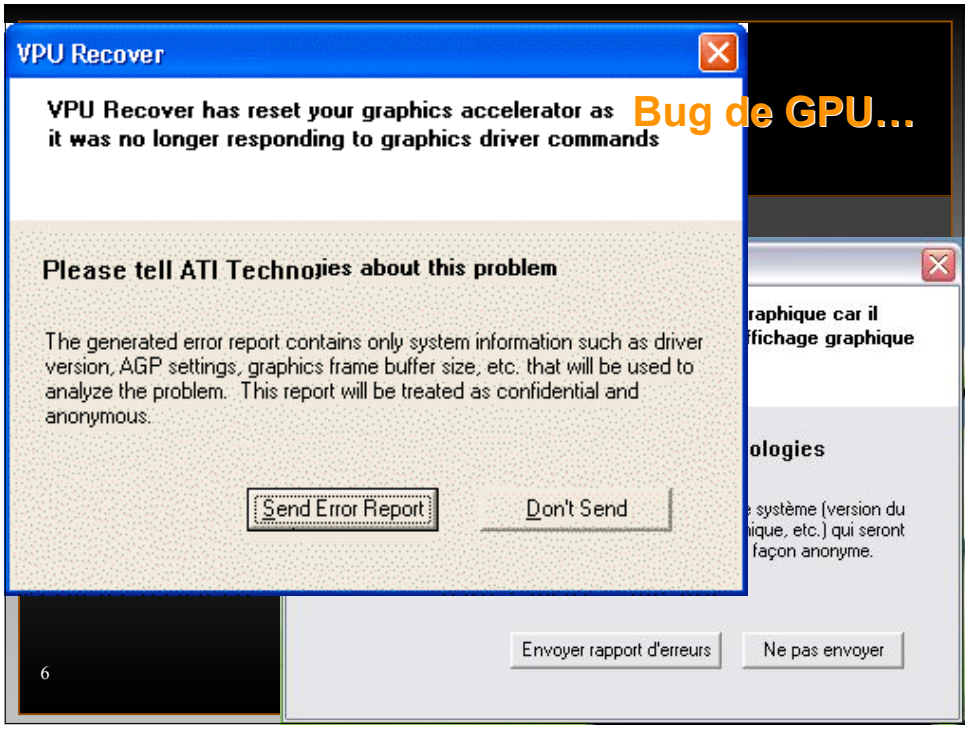
3

Qual o principal impacto da introdução das GPUs?

- SLI - Scalable Link Interface
- LMA - Lightspeed Memory Architecture
- HyperZ - Hierarchical Z-buffer
- GART - Graphics Address Remapping Table
- GTS - Giga Texel Shader

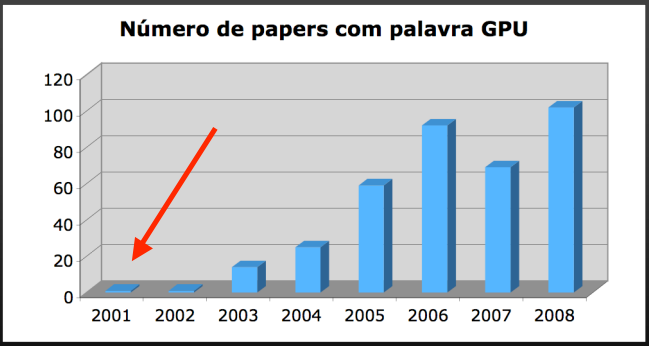
4


Não apenas Bug de CPU…
mozilla.exe
mozilla.exe has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost.
Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.
To see what data this error report contains, click here.
Send Error Report
Don't Send
5


VPU Recover
VPU Recover has reset your graphics accelerator as it was no longer responding to graphics driver commands
Bug de GPU…
Please tell ATI Technologies about this problem
The generated error report contains only system information such as driver version, AGP settings, graphics frame buffer size, etc. that will be used to analyze the problem. This report will be treated as confidential and anonymous.
Send Error Report
Don't Send
Envoyer rapport d'erreurs
Ne pas envoyer
6

Qual o principal impacto da introdução das GPUs?

Amostragem da base bibliográfica da ACM/Siggraph



Pergunta importante número 1:
O que aconteceu em 2001?

7

GeForce3

- **Nvidia lança a primeira placa programável!**

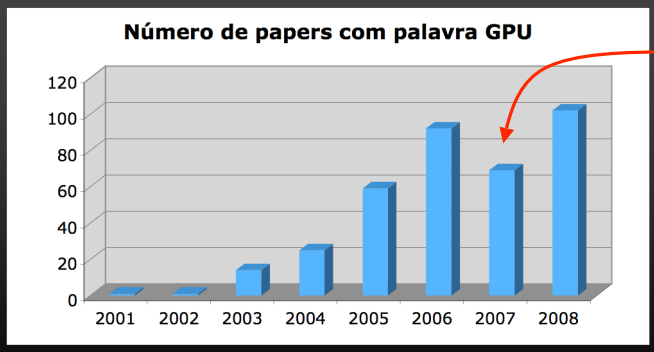
A User-Programmable Vertex Engine
Erik Lindholm, Mark J. Kilgard, Henry Moreton
Proceedings of ACM SIGGRAPH 2001, August 2001, pp. 149-158.

In this paper we describe the design, programming interface, and implementation of a very efficient user-programmable vertex engine. The vertex engine of NVIDIA's GeForce3 GPU evolved from a highly tuned fixed-function pipeline requiring considerable knowledge to program. Programs operate only on a stream of independent vertices traversing the pipe…

8

Qual o principal impacto da introdução das GPUs?

Amostragem da base bibliográfica da ACM/Siggraph



Pergunta importante número 2:

O que aconteceu em 2007?

37.5% a menos de papers!

9

Sumário

- Introdução
- História: 5 gerações

10

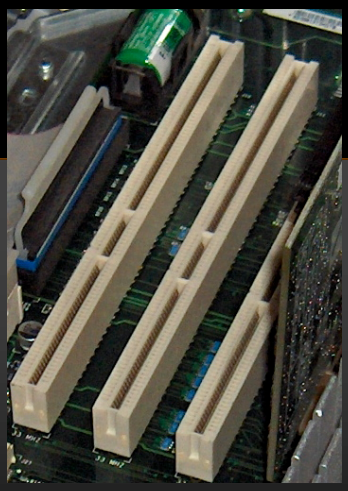
Algumas definições

- Comunicação Periféricos/CPU
- Memórias

11

PCI

- ***Peripheral Component Interconnect***
  - Versão estável 2.0 lançada em 1993 pela Intel
- **Comunicação entre CPU e dispositivos**



64 bits PCI Slots

12

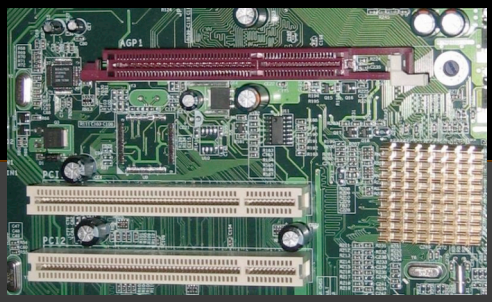
AGP

- ***Accelerated Graphics Port***
  - Lançada em 1997 pela Intel
- ***Point to point protocol***
  - Contato direto com a CPU e com a memória RAM (*PCI devices* compartilham bus)
  - Acesso direto à memória principal, sem necessidade de copiar para graphics card (*Graphics Address Remapping Table* - GART)

1a sigla!

13

AGP



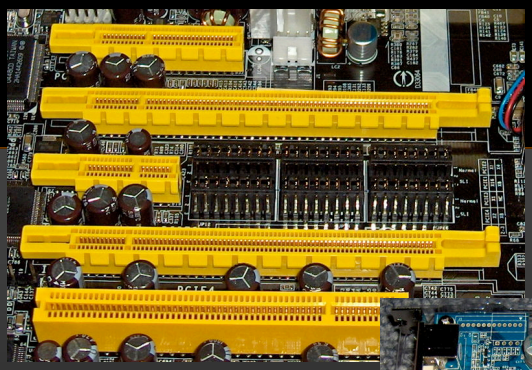
- **AGP 8x**
- **Canal de 32bits com taxa de 2Gb/s**
- **Variantes: AGP Express, 64bit AGP, …**
- **As melhores placas hoje que ainda têm AGP são ATi X1950XT e nVidia 7950GT**

14

PCI Express

- **Lançada em 2004 pela Intel**
- **Permite comunicação em duas vias: recebe e transmite independentemente**
- **PCI Express 1x = 250Mb/s/direção**
  - Placas gráficas começam em 16x! Dobro da AGP
- **É a base da interface *SLI - Scalable Link Interface* e *Crossfire* da ATI**
  - Permite mais de uma GPU trabalhando em conjunto
  - Mais sobre isto logo em seguida

15

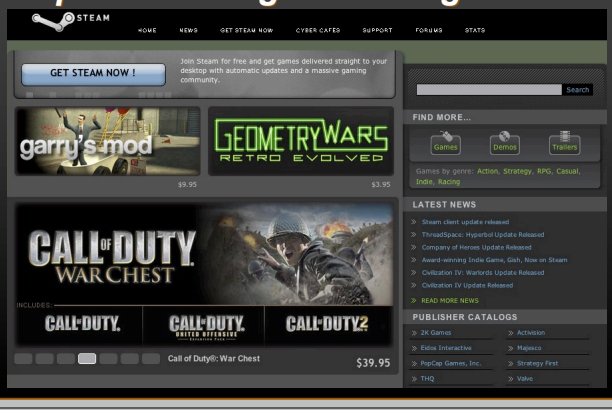


PCI Express Slots

16

Quem está ganhando?

- ***A survey by Valve Corporation in November 2006 found approximately a 50:50 split between AGP and PCI Express amongst Steam gamers***



17

E as memórias?

- **DDR = Double Data Rate Synchronous DRAM**
  - É uma SDRAM que consegue transferir dados duas vezes por ciclo de clock (*the rising and falling edges of the cycle*)
- **DDR2, DDR3 = *lower voltage, faster clocks***

18

GDDR-1, 2, 3 G is for Graphics...

Modules	Type	Voltage VDD,VDDQ	Frequency Range
128M bit	GDDR-1	2.5V, 2.5V	183-500 (800)MHz*
256M bit	GDDR-1	2.5V, 2.5V	183-500 (800)MHz*
256M bit	GDDR-2	2.5V, 1.8V	400-500MHz
256M bit	GDDR-3	1.8V, 1.8V	500-800MHz

Padrão na série 8000 e acima da Nvidia

19

GDDR4

- ATI Radeon HD 4670 XT e outras
- Clock 1Ghz ~ 2.0 GBits/second

20

Lá fora é uma selva

- **Qual é mais rápida?**
  - NVidia GeForce FX 6600 GT AGP ou NVidia GeForce FX 6600 PCI Express ?
  - ATI Radeon 9600 Pro ou NVidia GeForce FX 6600 AGP?
  - NVidia GeForce FX 6600 GT AGP ou ATI Radeon 9800 Pro?

21

	GeForce 6600 GT	GeForce 6600	GeForce 6800 GT	ATI Radeon X700 Pro
Manufacturing process	0.11 micron	0.11 micron	0.13 micron	0.11 micron
Chip technology	256-bit	256-bit	256-bit	256-bit
Transistor count	143 million	143 million	220 million	120 million
Core clock speed.	500MHz	300MHz	350MHz	425MHz
Memory clock speed.	500MHz (DDR=1000MHz)	350MHz (DDR=700MHz)	500MHz (DDR=1000MHz)	430MHz (DDR=860MHz)
Memory Bus width	128-bit GDDR3	128-bit GDDR	256-bit GDDR	128-bit GDDR3
Memory bandwidth.	16.0GB/sec	11.2GB/sec	32.0GB/sec	13.8GB/sec
Rendering Pixel pipelines.	8	8	16	8
Texture units per pipeline.	1	1	1	1
Pixel fill rate.	2.0Gpixels/sec	1.2Gpixels/sec	5.6Gpixels/sec	1.7Gpixels/sec
Vertex shader units.	3	3	6	6
Colour bits per channel	10	10	10	10
DirectX version	9.0c	9.0c	9.0	9.0
Max displays/Ramdacs	2/2	2/2	2/2	2/2

Muitas possibilidades de combinações…

22

Introdução

Graphics hardware – Porquê?

- Em média 1000x faster!
- Outra razão: em média 1000x faster!
- Simples de paralelizar e ”*pipelinizar*”

24

Arquiteturas de Hardware Gráfico (HG)

- **A evolução do HG começou no final do pipeline**
  - Rasterizador foi a primeira parte a ser ”hardwarizada” (maior ganho de performance)
  - Depois o estágio de geometria
- **Duas maneiras principais de obter melhor desempenho:**
  - Pipelining
  - Paralelização
  - Combinações destas duas

25

Rapidamente sobre pipelining

- **Na GeForce3: 600-800 estágios de pipeline!**
  - 57 million transistors
  - Pentium IV: 20 stages, 42 million transistors
- **Idealmente: n estágios → n vezes mais *throughput***
  - Mas aumenta a latência (tempo para passar nos estágios)!
  - Entretanto os altos clocks amenizam o problema

26

Rapidamente sobre pipelining

- **HG é mais simples para pipeline porque:**
  - Pixels são (na maioria das vezes) independentes entre si
  - Poucos desvios e muita functionalidade fixa
  - Simples de prever o padrão do acesso de memória (*do prefecthing!)*

27

Paralelismo

- **Computa n resultados em paralelo e combina os resultados**
- **GeForce GTX 275: 240 stream processors em paralelo!!**
- **Muitos dados (vértices e pixels) sendo processados simultaneamente**
- **Nem sempre simples, mas vértices e pixels são independentes entre si, então mais simples para o HG**

28

Largura de banda e memória

- **Operações sobre pixels**
  - TR - leitura textura
  - RZ - leitura z-buffer
  - WZ - escrita z-buffer
  - RC - leitura color
  - WC - escrita cor
- **Assumindo 2 texturas por pixel**
  - TR costs 3*8 bytes (trilinear MIP-mapping)
  - Todos os outros custam 4 bytes
- **Um pixel "normal" custa:**
  - RZ+WZ+WC+2*TR = **60 bytes per pixel**

29

Largura de banda e memória

- **Com 60 fps e 1280x1024: 4.5 Gb/s**
- **Mas um pixel não é processado apenas uma vez**
- **Supondo overdraw = 4 temos 18 Gb/s!**
- **Assumindo DDRAM a 300 MHz, 256 bits por acesso: 9.6 Gb/s**
- **18>9.6!!!!**
- **Outros fatores pioram ainda mais este valor. Nunca se consegue utilizar a largura de banda a 100% e podemos querer mais texturas e anti-aliasing por exemplo**

30

Técnicas para redução do uso da largura de banda

- *Texture caching with prefetching*
- Compressão de Texturas
- Z-compression
  - Enviar valor de z comprimido sem perda (menos bandwidth)
- Teste de oclusão de Z (HyperZ)

31

Hyper Z

- Lançado em 2000 pela ATI
- Gargalo: Em média RZ e WZ ocupam 50% da banda, quando não 100%
- Em média 60% dos pixels passam no z-buffer
- Idéia: não desenhar os pixels que não passam no z-buffer
- Nvidia equivalente: *LMA - Lightspeed Memory Architecture*

32

Z-occlusion testing and Z-compression

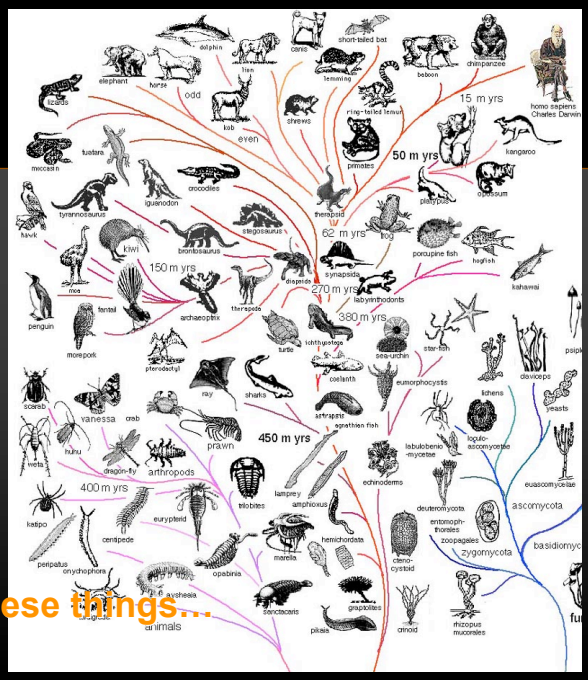
- Divide tela em *tiles* de 8x8 pixels
- Mantém status do tile na memória da GPU
- Armazena z_{max} per tile
- Rasteriza um *tile* de cada vez
- Testa se z_{min} do triângulo está atrás do tile
  - Sim, este tile não precisa mais ser processado
  - Economiza operações para todo o tile
- Quake por exemplo: 19% pixels falharam z-teste. Destes 51% foram detectados pelo HyperZ

33

História

Parte II

Como as coisas evoluíram?



Well, not these things…

35

No Início era o Caos…

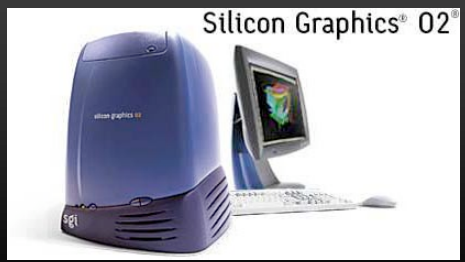
- **Hardware Gráfico Pré-GPU**
  - The 70's
    - Evans & Sutherland: Clipping & Proj. Perspectiva
  - The 80's
    - VGA - *Video Graphics Array*
    - SGI workstations desenhavam linhas e polígonos em tempo real
  - The 90's
    - SGI Reality Engine: Mapeamento de textura em tempo real (1994)

36

Silicon Graphics

- Jim Clark (professor em Stanford) cria a Silicon Graphics
- 1984 - IRIS 1400, usando aceleradores gráficos VLSI (GeometryEngines) e um frame buffer
  - Primeiro sistema interativo para CAD e visualização científica
- No início da década de 90, iniciativas de engenheiros da SGI para criar mapeamento de texturas usando hardware eram descartadas por "não ter mercado"

37



38

Primeira Geração 1994-1998

- **NVidia TNT2, ATI Rage, 3dfx Voodoo3**
- **Características:**
  - Rasterização de triângulos pré-transformados
  - Mapeamento de 1 ou 2 texturas (dual texture)
  - DirectX 6
  - Liberam CPU p/ pixels individuais
  - AGP 4x (TNT2)
- **Limitações:**
  - Sem transformação 3D
  - Poucas operações matemáticas para combinar texturas

39

Primeira Geração 1994-1998

- **Chip da NVidia avançado para época, mas não trabalha com polígonos! (1995)**

Virtua Fighter
(SEGA Corporation)

40

Primeira Geração 1994-1998

- 3dfx Voodoo (1996) é a única placa que apenas acelera 3D (ou seja, precisava de outra placa como VGA)
- Jogos passam de 8-bits 15fps para 16-bit 30fps, usando textura filtrada e z-buffer!!

41

Interlúdio: SIGGRAPH 1996

- Microsoft tenta criar padrão de "tiles" (projeto Talisman)
- J. Torborg and J.T. Kajiya. *Talisman: Commodity Realtime 3D Graphics for the PC*. SIGGRAPH '96
- Suporte de Samsung, Fujitsu e Cirrus Logic
- Mas quem ditou o futuro foi a Voodoo e similares

Chicken Crossing
Animação feita para Demonstrar as habilidades de Talisman

42

Primeira Geração 1994-1998

- **3dfx domina o mercado em 1997**
  - Firma acordo com Sega para o Dreamcast, mas Sega acaba desistindo
- **ATI compete com 3dfx com a Rage II, mas o grande ganho de performance vem com Rage Pro**
  - Mas drivers eram ruins
  - Foi praticamente ignorada pelo modismo da voodoo

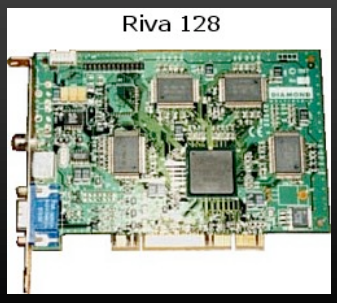
3D Rage Pro

43

Primeira Geração 1994-1998

- **NVidia NV3 (Riva 128) é melhor que a voodoo também**

Riva 128



44

Primeira Geração 1994-1998

- **Voodoo 3 é lançada em 1999, mas perde espaço para NVidia**

45

Segunda Geração (1999-2000)

- **NVidia GeForce 256, GeForce 2, ATI's Radeon 7500, S3's Savage3D**
- **Características:**
  - Transformações de vértices (OpenGL e DirectX 7) e iluminação na GPU!
  - Anteriormente, isto só era possível em workstations
  - Mapeamento de texturas cúbicas e operações matemáticas com sinais
  - Mais configurável, mas não realmente programável
  - Memória DDR (GeForce2)

46

Segunda Geração (1999-2000)

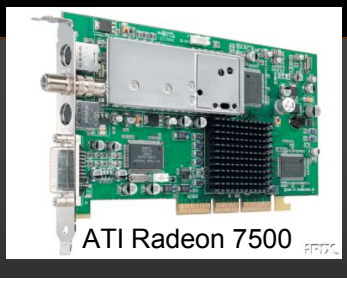
- GeForce 256: Hardware “TnL” (transformation and lighting)
- Mas sucesso deve-se ao “fill rate”, que era 2x superior ao da concorrência
- Chip anterior (TNT 2) já era rápido mas tinha sido igualado pela Voodoo3 e superado pela Matrox G300

GeForce 256

47

Segunda Geração (1999-2000)

- Única competição para a GeForce 256 é a ATI Radeon, lançada em 2000
- Mas NVidia lança GeForce2 GTS 48 horas depois… “Giga Texel Shader”
- NVidia é selecionada em março de 2000 para fazer o chip da **XBOX**



ATI Radeon 7500

GeForce2 GTS

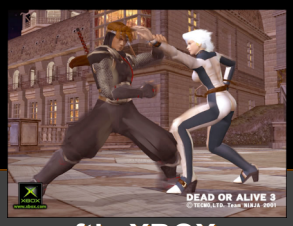
48

Segunda Geração (1999-2000)

- ATI é escolhida para fazer o chip da Nintendo Gamecube (na verdade comprou a companhia que já estava fazendo o chip)
- Apesar do domínio da NVidia, a Voodoo5 ganhou o prêmio de "Best Hardware Product" do ano 2000

49

Terceira Geração (2001)

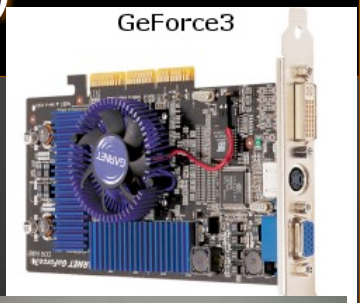


- NVidia's GeForce 3, GeForce 4 Ti, Microsoft's XBOX, ATI's Radeon 8500.
- Características:
  - Finalmente vertex/fragment programming capabilities!
    - Fragmentos *not truly programable*
    - Especificam uma seqüência de instruções para processamento de vértices
    - Entretanto, não são poderosos o suficiente (poucas instruções)
  - DirectX 8 e diversas extensões proprietárias do OpenGL
  - 3D Textures e Shadow Maps

50

Terceira Geração (2001)

- NVidia GeForce3 (fevereiro): 30% mais rápida que a GeForce2; DirectX 8.0
- ATI Radeon 8500 (agosto); DirectX 8.1 10% a 20% mais rápido que a GeForce3



ATI Radeon 8500

51

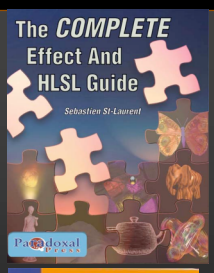
Quarta Geração (2002-03)

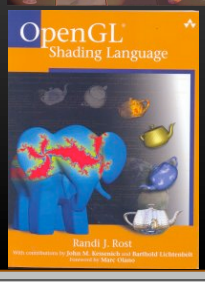
- NVidia GeForce FX, Radeon 9700, 9800, X800.
- Características:
  - Número de instruções por programa (vértice/fragmento) aumenta drasticamente
  - *Truly programable fragments*
  - **Introdução das linguagens de alto nível para placas gráficas: Cg (C for Graphics), GLSL, HLSL**
  - DirectX 9.x, OpenGL 2.0.

52

Como programar a GPU?

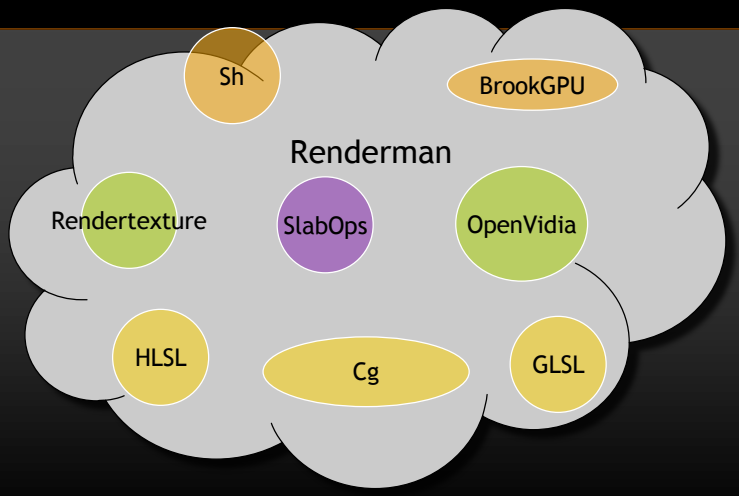
- Assembler
- Linguagens específicas
- Cg
  - NVidia
- HLSL
  - Microsoft
- GLSL
  - OpenGL





53

Language Zoo



54

CG em duas linhas…

- Neutra
- Primeira opção, portanto ainda bastante utilizada
- Difícil de usar para aplicações não-gráficas

55

Cg in Two Pages

Mark J. Kilgard
NVIDIA Corporation
Austin, Texas
January 16, 2003

Cg Toolkit

1. Cg by Example

Cg is a language for programming GPUs. Cg programs look a lot like C programs. Here is a Cg vertex program:

```
void simpleTransform(float4 objectPosition : POSITION,
                     float4 color          : COLOR,
                     float4 decalCoord     : TEXCOORD0,
                     float4 lightMapCoord  : TEXCOORD1,
                 out float4 clipPosition   : POSITION,
                 out float4 oColor         : COLOR,
                 out float4 oDecalCoord    : TEXCOORD0,
                 out float4 oLightMapCoord : TEXCOORD1,
                 uniform float brightness,
                 uniform float4x4 modelViewProjection)
{
  clipPosition = mul(modelViewProjection, objectPosition);
  oColor = brightness * color;
  oDecalCoord = decalCoord;
  oLightMapCoord = lightMapCoord;
}
```

1.1 Vertex Program Explanation

The program transforms an object-space position for a vertex by a 4x4 matrix containing the concatenation of the modeling, viewing, and projection transforms. The resulting vector is output as the clip-space position of the vertex. The per-vertex color is scaled by a floating-point parameter prior to output. Also, two texture coordinate sets are passed through unperturbed.

Cg supports scalar data types such as `float` but also has first-class support for vector data types. `float4` represents a vector of four floats. `float4x4` represents a matrix. `mul` is a standard library routine that performs matrix by vector multiplication. Cg provides function overloading like C++; `mul` is an overloaded function so it can be used to multiply all combinations of vectors and matrices.

Cg provides the same operators as C. Unlike C however, Cg operators accept and return vectors as well as scalars. For example, the scalar, `brightness`, scales the vector, `color`, as you would expect.

In Cg, declaring a parameter with the `uniform` modifier indicates that its value is initialized by an external source that will not vary over a given batch of vertices. In this respect, the `uniform` modifier in Cg is different from the `uniform` modifier in RenderMan but used in similar contexts. In practice, the external source is some OpenGL or Direct3D state that your application takes care to load appropriately. For example, your application must supply the `modelViewProjection` matrix and the `brightness` scalar. The Cg runtime library provides an API for loading your application state into the appropriate API state required by the compiled program.

The `POSITION`, `COLOR`, `TEXCOORD0`, and `TEXCOORD1` identifiers following the `objectPosition`, `color`, `decalCoord`, and `lightMapCoord` parameters are input semantics. They indicate how their parameters are initialized by per-vertex varying data. In OpenGL, `glVertex` commands feed the `POSITION` input semantic; `glColor` commands feed the `COLOR` semantic; `glMultiTexCoord` commands feed the `TEXCOORDn` semantics.

The `out` modifier indicates that `clipPosition`, `oColor`, `oDecalCoord`, and `oLightMapCoord` parameters are output by the program. The semantics that follow these parameters are therefore output semantics. The respective semantics indicate the program outputs a transformed clip-space position and a scaled color. Also, two sets of texture coordinates are passed through. The resulting vertex is feed to primitive assembly to eventually generate a primitive for rasterization.

Compiling the program requires the program source code, the name of the entry function to compile (`simpleTransform`), and a profile name (`vs_1_1`).

The Cg compiler can then compile the above Cg program into the following DirectX 8 vertex shader:

```
vs.1.1
mov oT0, v7
mov oT1, v8
dp4 oPos.x, c1, v0
dp4 oPos.y, c2, v0
dp4 oPos.z, c3, v0
dp4 oPos.w, c4, v0
mul oD0, c0.x, v6
```

The profile indicates for what API execution environment the program should be compiled. This same program can be compiled for the DirectX 9 vertex shader profile (`vs_2_0`), the multi-vendor OpenGL vertex program extension (`arbvp1`), or NVIDIA-proprietary OpenGL extensions (`vp20` & `vp30`).

The process of compiling Cg programs can take place during the initialization of your application using Cg. The Cg runtime contains the Cg compiler as well as API-dependent routines that greatly simplify the process of configuring your compiled program for use with either OpenGL or Direct3D.

1.2 Fragment Program Explanation

In addition to writing programs to process vertices, you can write programs to process fragments. Here is a Cg fragment program:

```
float4 brightLightMapDecal(float4 color       : COLOR,
                           float4 decalCoord  : TEXCOORD0,
                           float4 lightMapCoord : TEXCOORD1,
                       uniform sampler2D decal,
                       uniform sampler2D lightMap) : COLOR
{
  float4 d = tex2Dproj(decal, decalCoord);
  float4 lm = tex2Dproj(lightMap, lightMapCoord);
  return 2.0 * color * d * lm;
}
```

The input parameters correspond to the interpolated color and two texture coordinate sets as designated by their input semantics.

The `sampler2D` type corresponds to a 2D texture unit. The Cg standard library routine `tex2Dproj` performs a projective 2D texture lookup. The two `tex2Dproj` calls sample a decal and light map texture and assign the result to the local variables, `d` and `lm`, respectively.

Exemplos de Funções - Cg

Table 1 Mathematical Functions

Mathematical Functions	
Function	**Description**
`abs(x)`	Absolute value of `x`
`acos(x)`	Arccosine of `x` in range $[-\pi/2,\pi/2]$, `x` in [-1,1]
`all(x)`	Returns `true` if every component of `x` is not equal to 0. Returns `false` otherwise.
`any(x)`	Returns `true` if any component of `x` is not equal to 0. Returns `false` otherwise.
`asin(x)`	Arcsine of `x` in range $[0,\pi]$; `x` should be in [-1,1].
`atan(x)`	Arctangent of `x` in range $[-\pi/2,\pi/2]$
`atan2(y, x)`	Arctangent of `y/x` in range $[-\pi,\pi]$
`ceil(x)`	Smallest integer not less than `x`
`clamp(x, a, b)`	`x` clamped to the range [`a`,`b`] as follows: • Returns `a` if `x` is less than `a`. • Returns `b` if `x` is greater than `b`. • Returns `x` otherwise.
`cos(x)`	Cosine of `x`
`cosh(x)`	Hyperbolic cosine of `x`
`cross(a, b)`	Cross product of vectors `a` and `b`; `a` and `b` must be 3-component vectors.
`degress(x)`	Radian-to-degree conversion
`determinant(M)`	Determinant of matrix `M`
`dot(a, b)`	Dot product of vectors `a` and `b`
`exp(x)`	Exponential function e^x
`exp2(x)`	Exponential function 2^x
`floor(x)`	Largest integer not greater than `x`

Table 2 Geometric Functions

Geometric Functions	
Function	**Description**
`distance(pt1, pt2)`	Euclidean distance between points `pt1` and `pt2`
`faceforward(N, I, Ng)`	`N` if `dot(Ng, I)` < 0; otherwise, `-N`.
`length(v)`	Euclidean length of a vector
`normalize(v)`	Returns a vector of length 1 that points in the same direction as vector `v`.
`reflect(i, n)`	Computes reflection vector from entering ray direction `i` and surface normal `n`. Only valid for 3-component vectors.
`refract(i, n, eta)`	Given entering ray direction `i`, surface normal `n`, and relative index of refraction `eta`, computes refraction vector. If the angle between `i` and `n` is too large for a given `eta`, returns (0,0,0). Only valid for 3-component vectors.

56

Exemplo de vertex program simples em Cg

Pre-defined names
Binding semantics

Vetor 4 floats

```
void main( float4 iPosition: POSITION,
           float4 iColour: COLOR,
           out float4 oPosition: POSITION,
           out float4 oColour: COLOR,
           uniform float4x4 modelviewProjection)
{
    oPosition = mul (modelviewProjection, iPosition);
    oColour = iColour;
}
```

57

O que faz este programa?

Outro exemplo

```
// Define inputs from application.
struct appin
{
  float4 Position     : POSITION;
  float4 Normal       : NORMAL;
};

// Define outputs from vertex shader.
struct vertout
{
  float4 HPosition    : POSITION;
  float4 Color        : COLOR;
};

vertout main(appin IN,
             uniform float4x4 ModelViewProj,
             uniform float4x4 ModelViewIT,
             uniform float4 LightVec)
{
  vertout OUT;

  // Transform vertex position into homogenous clip-space.
  OUT.HPosition = mul(ModelViewProj, IN.Position);

  // Transform normal from model-space to view-space.
  float3 normalVec = normalize(mul(ModelViewIT,
                                   IN.Normal).xyz);

  // Store normalized light vector.
  float3 lightVec = normalize(LightVec.xyz);

  // Calculate half angle vector.
  float3 eyeVec = float3(0.0, 0.0, 1.0);
  float3 halfVec = normalize(lightVec + eyeVec);

  // Calculate diffuse component.
  float diffuse = dot(normalVec, lightVec);

  // Calculate specular component.
  float specular = dot(normalVec, halfVec);

  // Use the lit function to compute lighting vector from
  // diffuse and specular values.
  float4 lighting = lit(diffuse, specular, 32);

  // Blue diffuse material
  float3 diffuseMaterial = float3(0.0, 0.0, 1.0);

  // White specular material
  float3 specularMaterial = float3(1.0, 1.0, 1.0);

  // Combine diffuse and specular contributions and
  // output final vertex color.
  OUT.Color.rgb = lighting.y * diffuseMaterial +
                  lighting.z * specularMaterial;
  OUT.Color.a = 1.0;

  return OUT;
}
```

O que faz esta chamada?

58

`lit(ndotl, ndoth, m)`	Computes lighting coefficients for ambient, diffuse, and specular light contributions. Returns a 4-vector as follows: • The `x` component of the result vector contains the ambient coefficient, which is always 1.0. • The `y` component contains the diffuse coefficient which is zero if `(n ● l)` < 0; otherwise `(n ● l)`. • The `z` component contains the specular coefficient which is zero if either `(n ● l)` < 0 or `(n ● h)` < 0; `(n ● h)`^{m} otherwise. • The `w` component is 1.0. There is no vectorized version of this function

Poder de expressão: com uma única chamada calcula Phong!

59

Exemplo de fragment program simples em Cg

```
void main(     float4 iColour: COLOR,
               out float4 oColour: COLOR)
{
       oColour = iColour;
}
```

O que faz este programa?

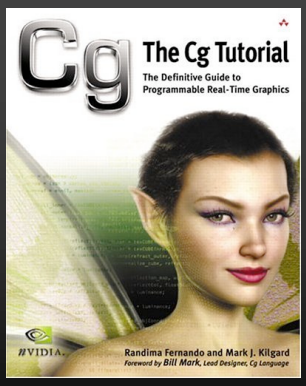
60

Profiles em Cg

- Grande variabilidade entre as capacidades das GPUs
- Cg introduz o conceito de ***language profiles***
- Um profile em Cg define um subconjunto do repertório total de possibilidades que é suportado numa plataforma específica

61

Futuro de Cg?

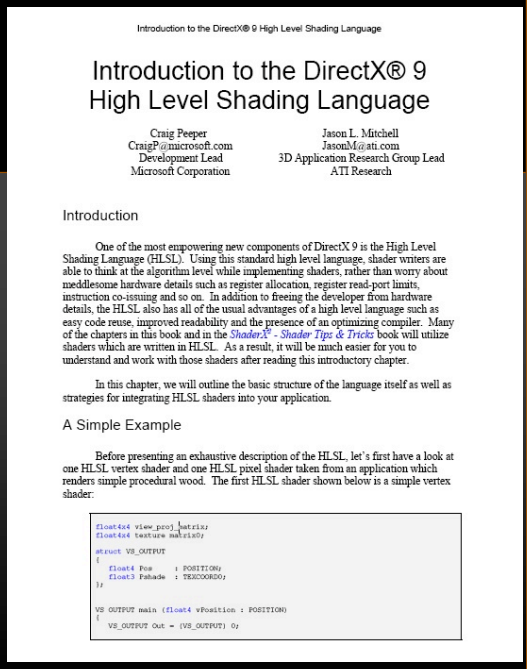


- Versão 2.2 lançada em Abril 2009
- Nvidia parece interessada em continuar o suporte

62

HLSL

- Microsoft
- Similar a Cg
- Parceria com ATI
- XNA e programação no XBOX



Introduction to the DirectX® 9 High Level Shading Language

Introduction to the DirectX® 9
High Level Shading Language

Craig Peeper
CraigP@microsoft.com
Development Lead
Microsoft Corporation

Jason L. Mitchell
JasonM@ati.com
3D Application Research Group Lead
ATI Research

Introduction

One of the most empowering new components of DirectX 9 is the High Level Shading Language (HLSL). Using this standard high level language, shader writers are able to think at the algorithm level while implementing shaders, rather than worry about meddlesome hardware details such as register allocation, register read-port limits, instruction co-issuing and so on. In addition to freeing the developer from hardware details, the HLSL also has all of the usual advantages of a high level language such as easy code reuse, improved readability and the presence of an optimizing compiler. Many of the chapters in this book and in the *ShaderX² - Shader Tips & Tricks* book will utilize shaders which are written in HLSL. As a result, it will be much easier for you to understand and work with those shaders after reading this introductory chapter.

In this chapter, we will outline the basic structure of the language itself as well as strategies for integrating HLSL shaders into your application.

A Simple Example

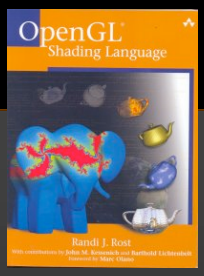
Before presenting an exhaustive description of the HLSL, let's first have a look at one HLSL vertex shader and one HLSL pixel shader taken from an application which renders simple procedural wood. The first HLSL shader shown below is a simple vertex shader:

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;

struct VS_OUTPUT
{
    float4 Pos      : POSITION;
    float3 Pshade   : TEXCOORD0;
};

VS_OUTPUT main (float4 vPosition : POSITION)
{
    VS_OUTPUT Out = (VS_OUTPUT) 0;
```

63

GLSL

- OpenGL 2.0
- 3DLabs
- Futuro no cell processor? (Playstation)



GLSL	Cg
	Context = cgCreateContext();
F=glCreateShaderObject(GL_FRAGMENT_SHADER_ARB);	FragmentProgram = cgCreateProgramFromFile(Context, CG_SOURCE, "vertexShader.cg", VertexProfile, NULL, NULL);
glShaderSource(f, 1, &ff,NULL);	cgGLLoadProgram(FragmentProgram);
glCompileShader(f);	
p = glCreateProgramObject();	
glAttachObject(p,f);	
glLinkProgram(p);	
glUseProgramObject(p);	cgGLBindProgram(FragmentProgram);

64

Escrevendo Shaders

- Todas as 3 linguagens fornecem um mecanismo de “runtime” para carregar os shaders, linkar as variáveis da CPU com GPU, habilitar e desabilitar shaders
- Cg e HLSL compilam o código shader para assembler (“source-to-source”).
- GLSL depende do fabricante da placa que fornece um compilador direto para o código de máquina da GPU, sem passo intermediário

65

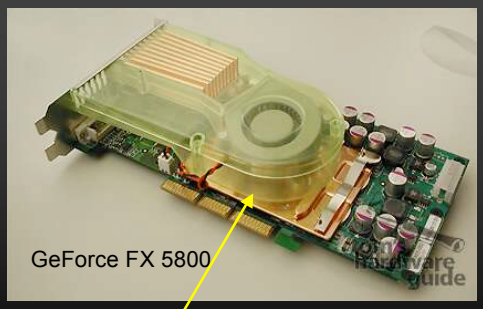
Compilando

- Código Cg pode ser compilado para diferentes GPUS (directx, nvidia, arbfp)
- HLSL compila diretamente para directx
- GLSL compila nativo

66

Quarta Geração (2002-03)

- ATI supera Nvidia
- NVidia GeForce FX lançada em 2003 vem com uma série de problemas devido à nova tecnologia adotada (125 milhões de transístores a 0.13 µm).



Olhem o tamanho do ventilador…

67

Quarta Geração (2002-03)

- ATI cria a VPU (o nome GPU foi criado pela NVidia anos antes…e registrado)
- Interface de memória de 256 bits, 8 rendering pipelines, 110 milhões de transistores (0.15 µm), 325Mhz.

68

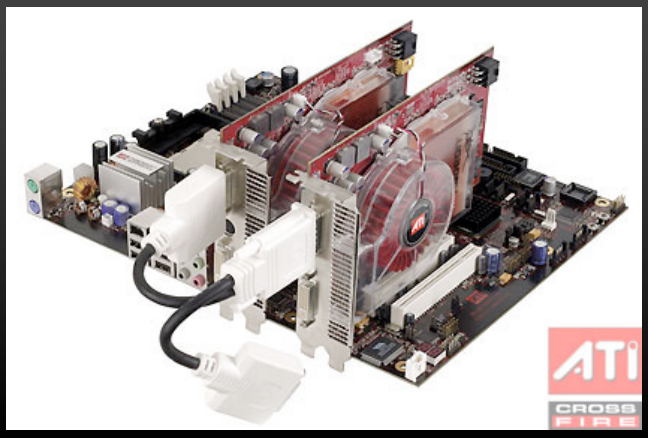
Quinta Geração (2004-05)

- Nvidia 6800
  - 16 pipes fragment shader
- Nvidia 7800
  - 24 pipes
- ATI X1800
  - Programas maiores
  - Desvios nos programas
  - Acesso à memória nos Vertex Shaders

69

2004-2005 MULTI-GPU

- **ATI Crossfire**

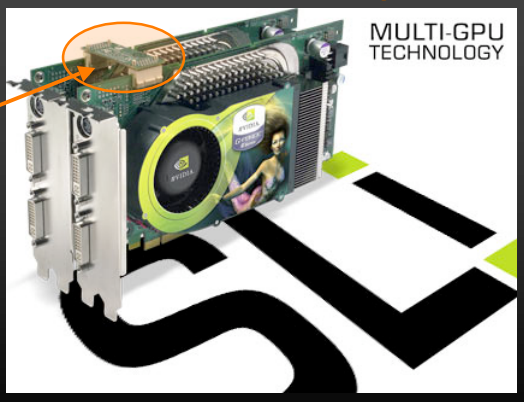


70

2004-2005 MULTI-GPU

- SLIs (Scalable Link Interface):

Ponte SLI



NVidia 6800 Ultra SLI

71

Scalable Link Interface

- Permitir que 2 ou mais GPUs compartilhem a carga de trabalho para renderização de uma cena
- Configuração mestre-escravo
- Comunicação via PCIe x16
- Ambas placas recebem a cena, mas metade é enviada para o escravo através da ponte SLI
- Após renderização o escravo re-envia sua parte para o mestre que monta a cena no FB

72

2005-2006 Videogame Wars

- **XBOX 360 (ATI)**
  - 500 Mhz
  - 512MB GDDR3 @ 700Mhz
  - CPU 3.2 Ghz 2.0 TFLOPS
- **Playstation 3 (NVidia)**
  - 550 Mhz
  - 256MB GDDR3 @ 700Mhz
  - CPU 3.2Ghz 1.0 TFLOPS

73

Resumo

- **Primórdios: sem aceleração 3D**
- **Primeira Geração: Projeção 3D – 2D (Voodoo)**
- **Segunda Geração: Lighting (GeForce 256)**
- **Terceira Geração: Programação vertex/fragment (GeForce 3)**
- **Quarta Geração: Programas muito maiores, linguagens de alto nível (Cg & GLSL) (ATI Radeon 9700)**
- **Quinta Geração: multi-GPUs**

74

Resumo

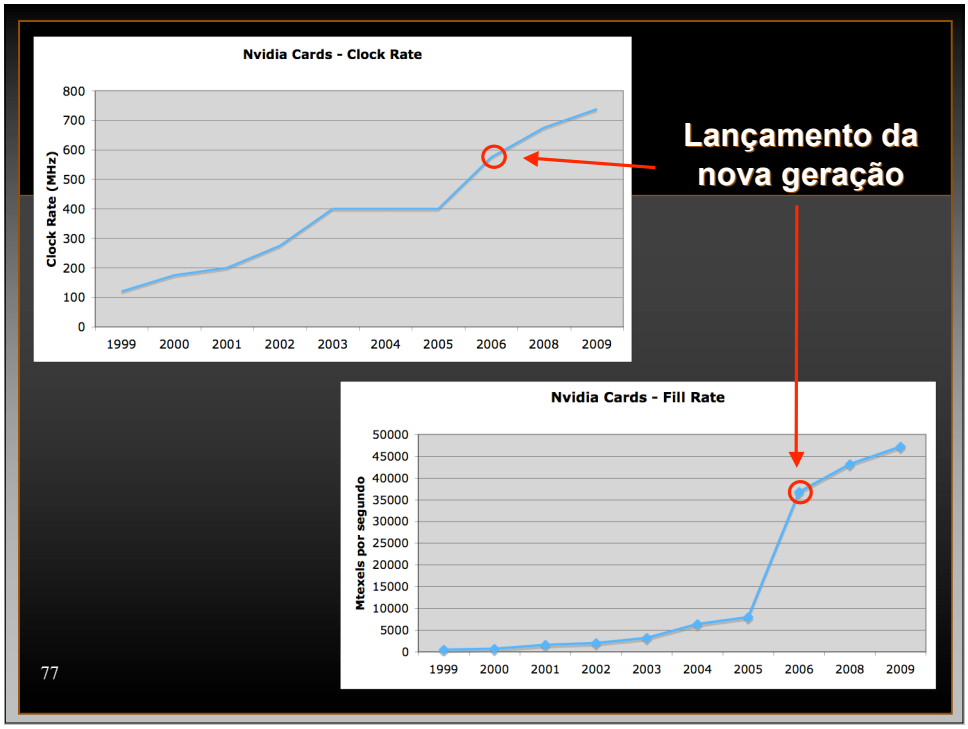
Primórdios	Aceleração 2D	Silicon Graphics Trident, Diamond, etc.
1ª Geração	Projeção	3dfx Voodoo
2ª Geração	Lighting	NVidia GeForce 256
3ª Geração	Programação Básica	NVidia GeForce 3
4ª Geração	Programação Avançada	ATI Radeon 9700

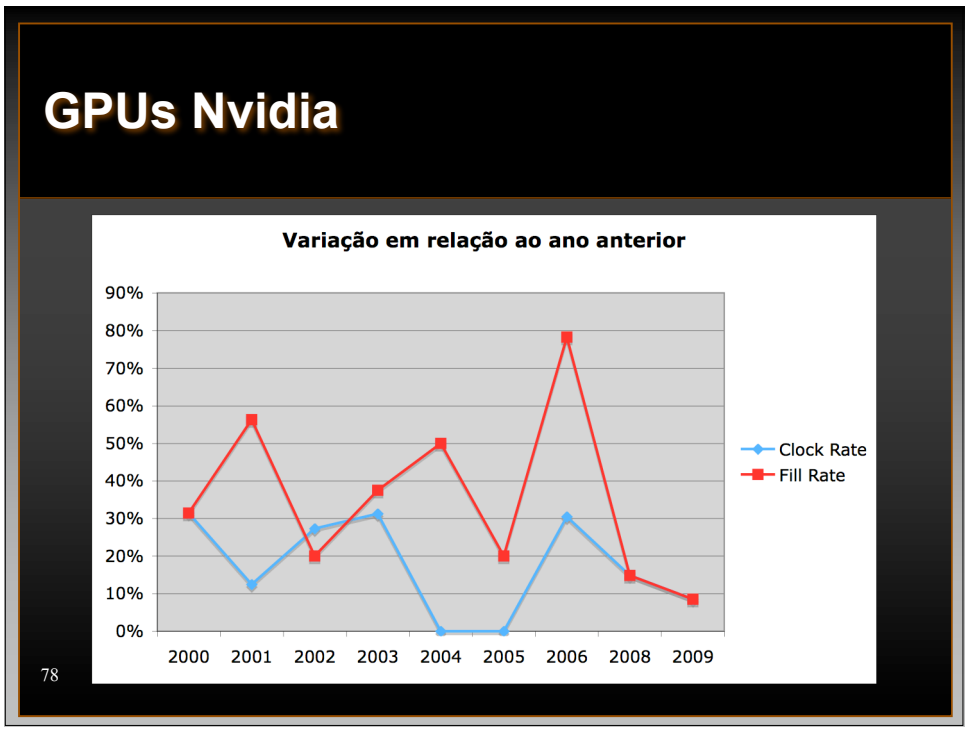
75

Breve Histórico GPUs Nvidia

Ano	Modelo	Processo	#Trans	Mpixels/s	pipes	
1998	RivaZX	0.25	5M	250		
1999	TNT2	0.22	9M	480		
2000	GeForce2 GTS	0.18	25M	800		
2001	GeForce3	0.15	57M	800		**
2002	GeForce4 Ti	0.15	63M	1000	4	
2003	GeForce 5900	0.13	125M	1600	4/8	
2005	GeForce 6800	0.13	222M	3900	16	
2006	GeForce 7800	0.09	302M	6400	24	$599

76


Nvidia Cards - Clock Rate
Clock Rate (MHz)
800
700
600
500
400
300
200
100
0
1999 2000 2001 2002 2003 2004 2005 2006 2008 2009
Lançamento da nova geração
Nvidia Cards - Fill Rate
Mtexels por segundo
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
1999 2000 2001 2002 2003 2004 2005 2006 2008 2009
77


GPUs Nvidia
Variação em relação ao ano anterior
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
2000 2001 2002 2003 2004 2005 2006 2008 2009
Clock Rate
Fill Rate
78