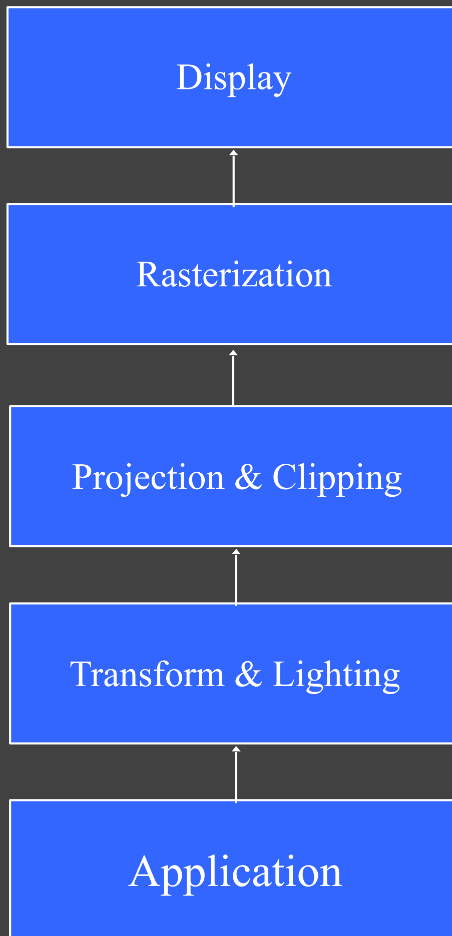
GPUs

CS 4620 Lecture 25

© Kavita Bala, Computer Science, Cornell University

Brief History



The dark ages (early-mid 1990's), when there were only frame buffers for normal PC's.

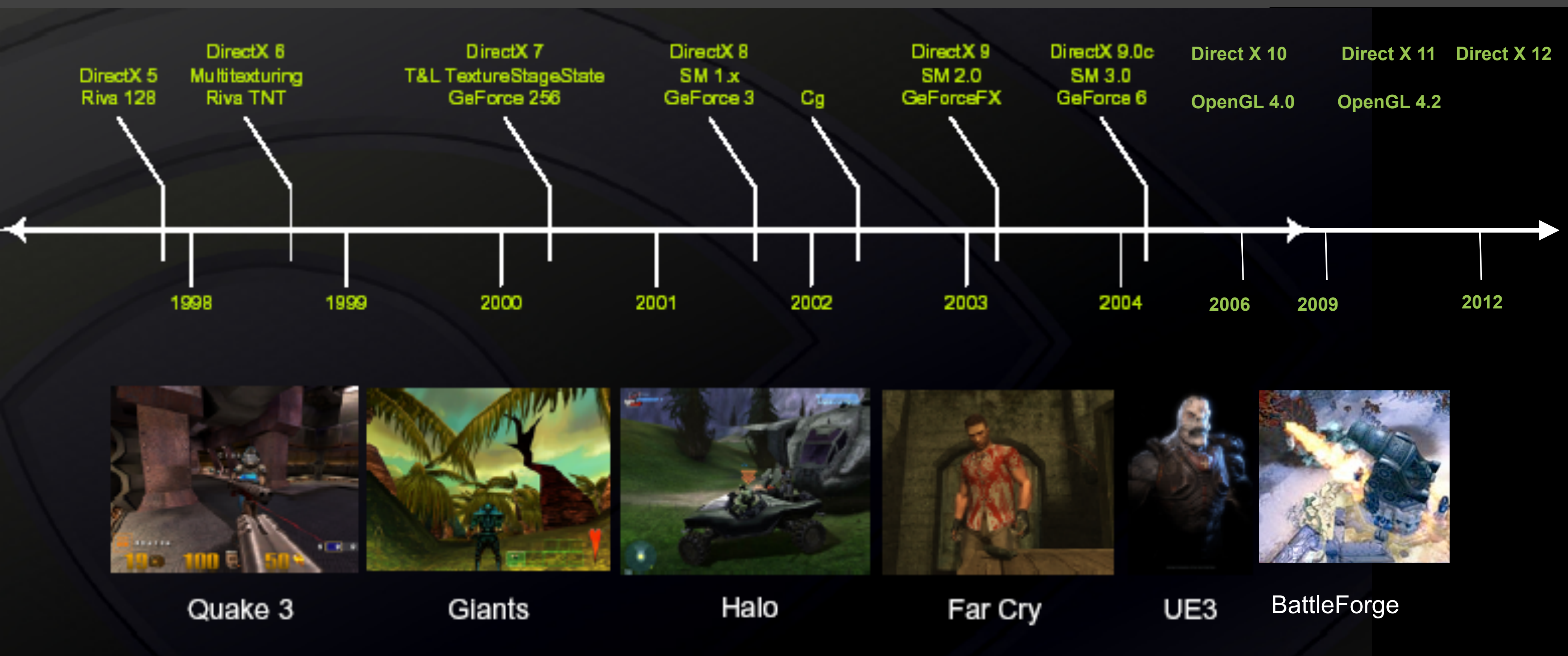
Some accelerators were no more than a simple chip that sped up linear interpolation along a single span, so increasing fill rate.

This is where pipelines start for PC commodity graphics, prior to Fall of 1999.

This part of the pipeline reaches the consumer level with the introduction of the NVIDIA GeForce256.

Hardware today has moved traditional application processing into the graphics accelerator.

© Kavita Bala, Computer Science, Cornell University



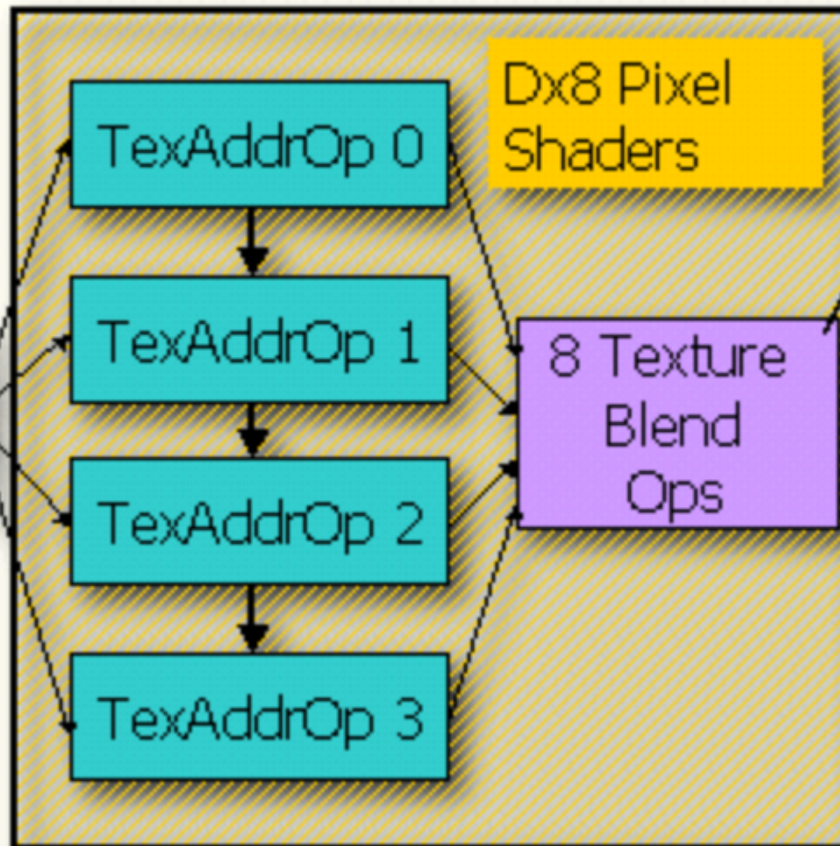
© Kavita Bala, Computer Science, Cornell University

Era of GPUs

Nvidia's GeForce 256 was the first graphics chip to actually be called a GPU, based on the addition of a hardware-based transformation and lighting engine (T&L).

This engine allowed the graphics chip to undertake the heavily floating-point intensive calculations of transforming the 3D objects and scenes – and their associated lighting – into the 2D representation of the rendered image. Previously, this computation was undertaken by the CPU, which could easily bottleneck with the workload, and tended to limit available detail.

Nvidia Grass Demo (GeForce 256)



© Kavita Bala, Computer Science, Cornell University

Concepts

© Kavita Bala, Computer Science, Cornell University

Multi-Pass Rendering

- Limits to what hardware can do in 1 pass
  - In fact might depend on results of previous pass
- So multi-pass rendering
  - Each pass does some part of shading
  - Outputs a “fragment”: rgb, alpha, z
  - Add or blend with previous pass

© Kavita Bala, Computer Science, Cornell University

Multi-pass rendering

Diefenbach 1997

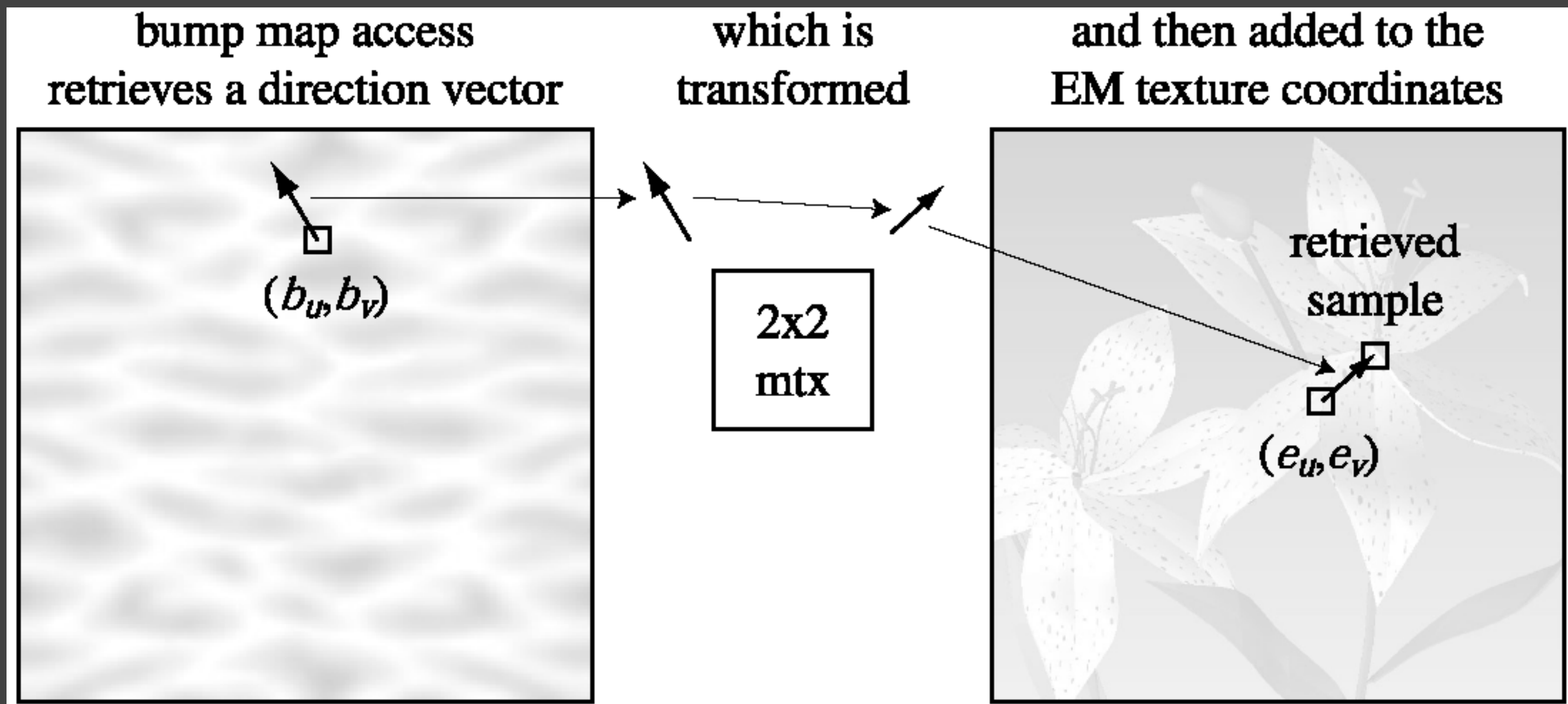
© Kavita Bala, Computer Science, Cornell University

Dependent Texture Reads

- Introduced in 1999
- Number of passes proportional to the longest "chain" of operations you need
- Dependent texture reads helps
  - Can read a texture
  - Transform it
  - And then read another texture based on transformed value!
  - Much more efficient

© Kavita Bala, Computer Science, Cornell University

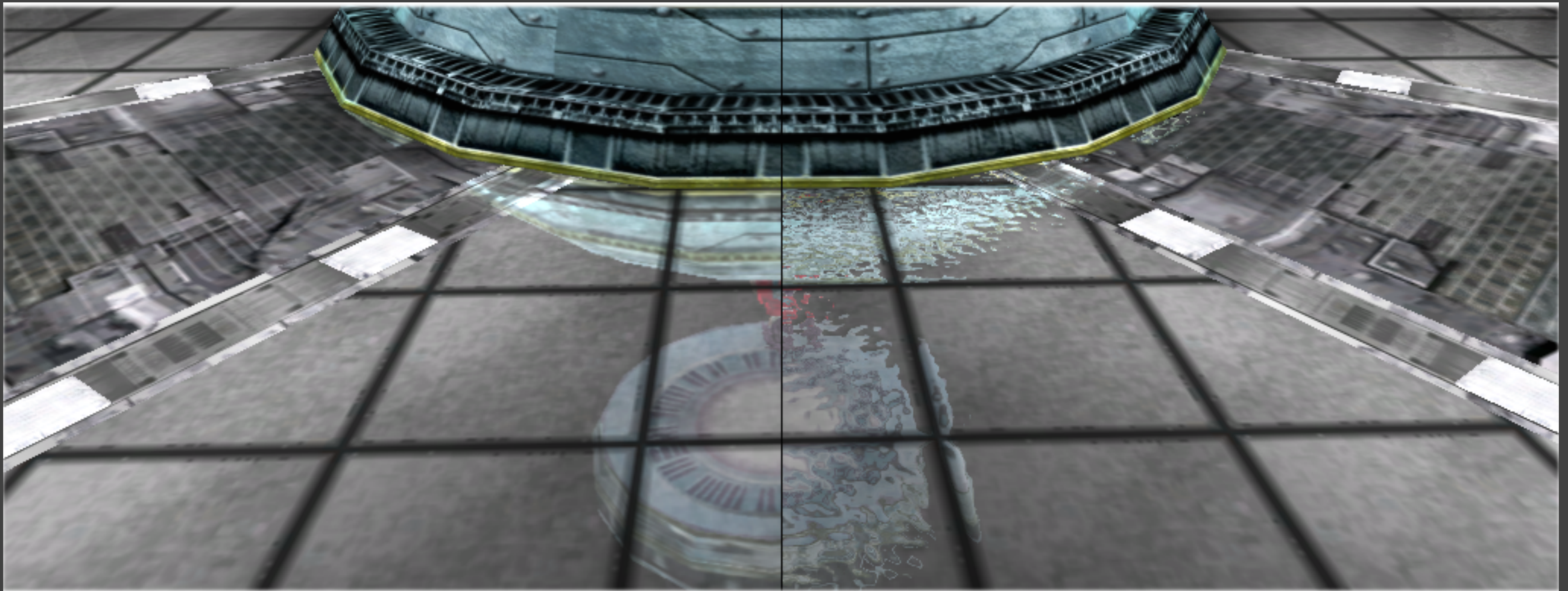
Dependent Texture Reads



© Kavita Bala, Computer Science, Cornell University

Reflections and Normal Maps

Environment Map Bump Mapping (EMBM)



© Kavita Bala, Computer Science, Cornell University

Rendering: forward shading

This is what we have done so far in 4620:

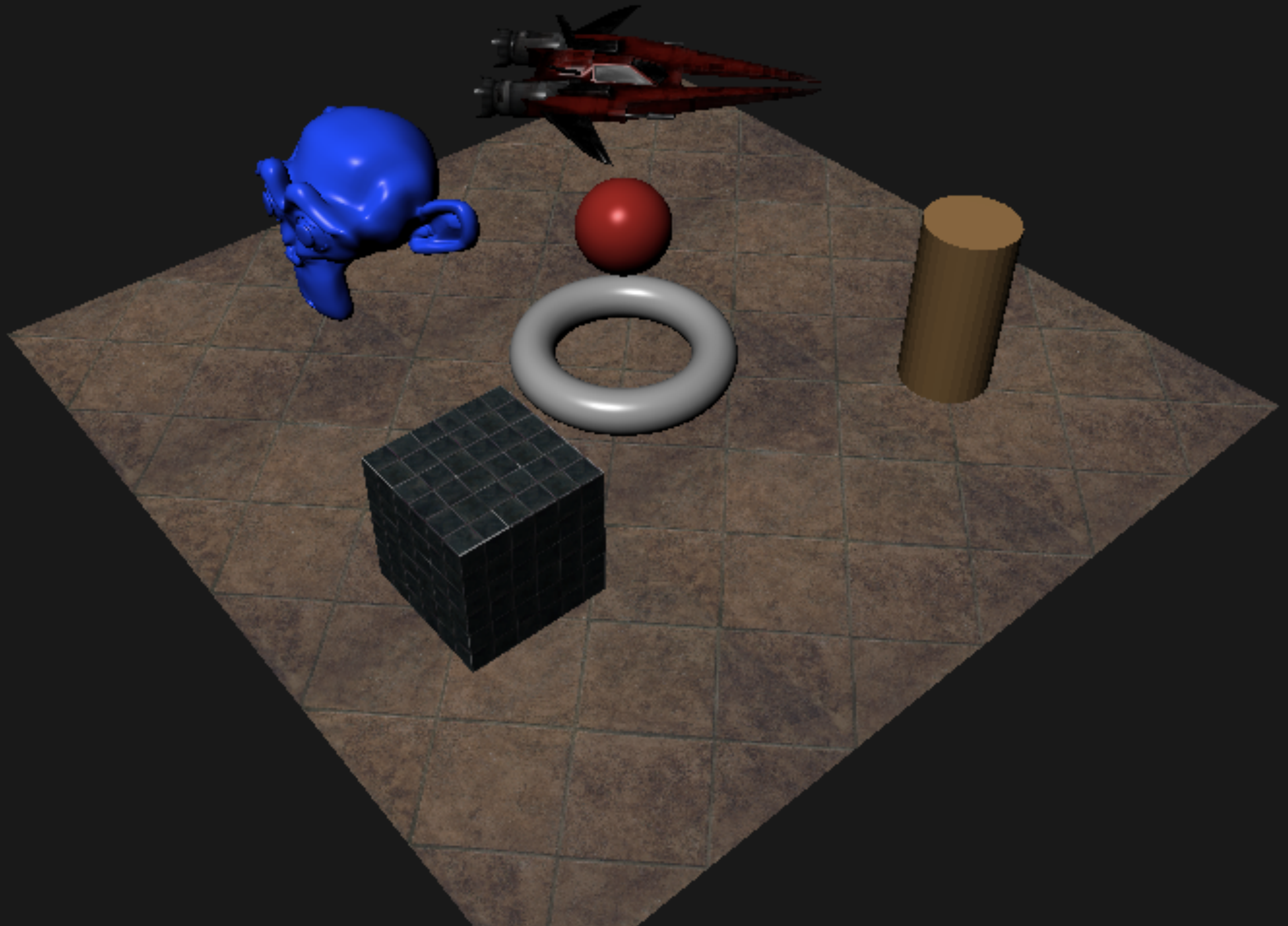
```
for each object in the scene
   for each triangle in this object
      for each fragment f in this triangle

       gl_FragColor = shade(f)
         if (depth of f < depthbuffer[x, y])
            framebuffer[x, y] = gl_FragColor
            depthbuffer[x, y] = depth of f
         end if

       end for
    end for
    end for
```

© Kavita Bala, Computer Science, Cornell University

Output: the shaded scene



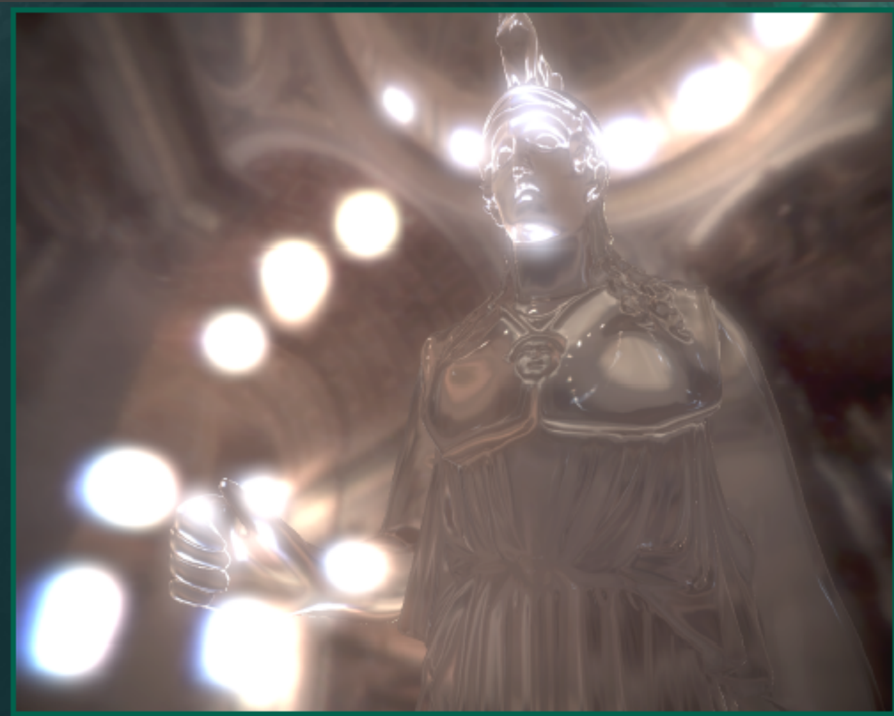
Drawbacks of Forward Shading

- If shade (f) is very expensive
  - e.g., many lights, shadow maps, complex shaders
  - overdraw by closer geometry wastes work on each fragment

© Kavita Bala, Computer Science, Cornell University

Drawbacks of Forward Shading

- Many other complex effects: Image processing effects
  - tonemapping, screen-space ambient occlusion, bloom, toon shaders etc., are very expensive



Overdraw: Real Example

(Battlefield 3)

© Kavita Bala, Computer Science, Cornell University

Deferred Shading Step 1

Code structure is nearly the same as forward shading, with one key difference:

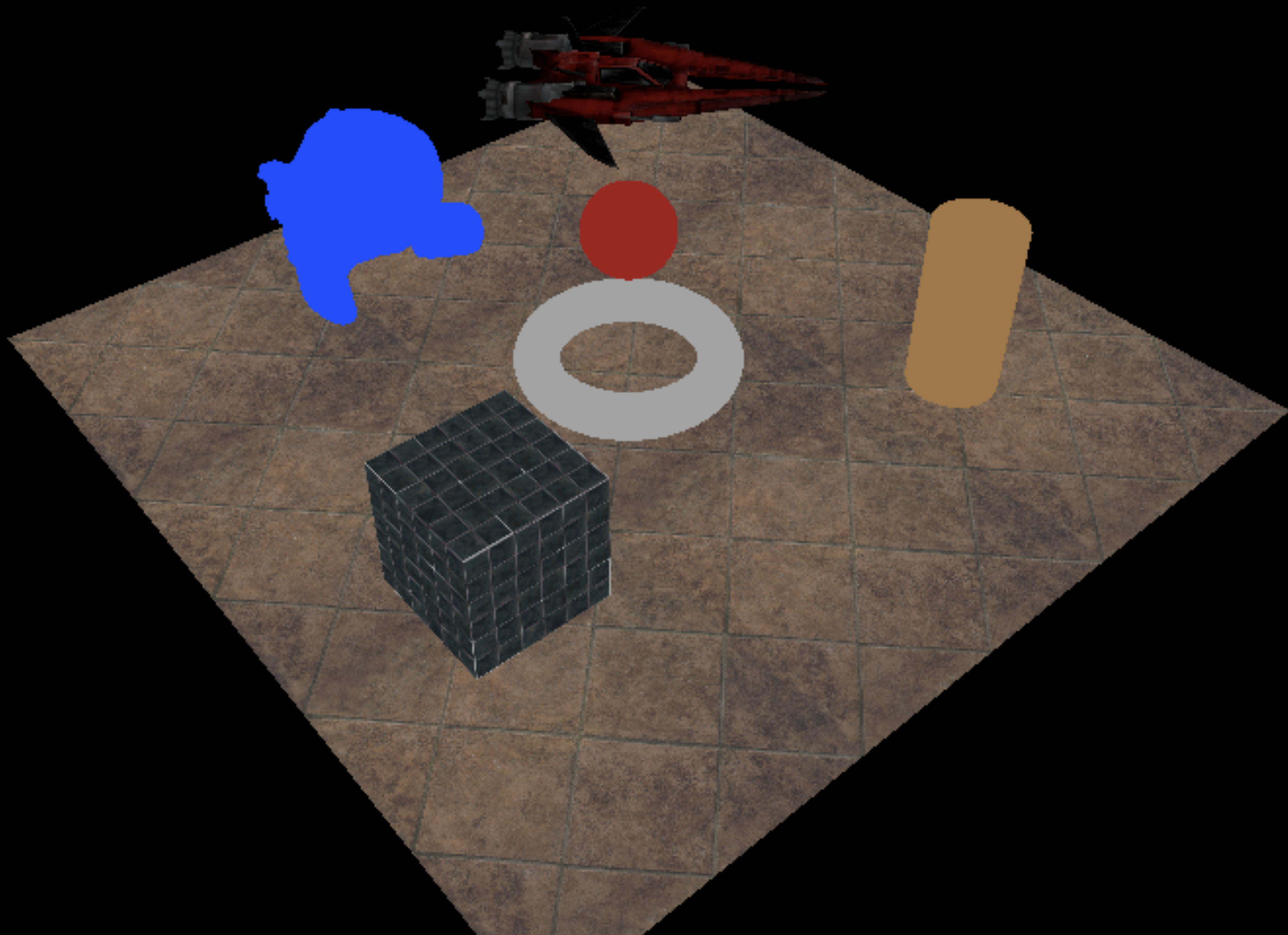
```
for each object in the scene
     for each triangle in this object
        for each fragment f in this triangle

        gl_FragColor = material properties of f
           if (depth of f < depthbuffer[x, y])
           gbuffer[x, y] = gl_FragColor
              depthbuffer[x, y] = depth of f
           end if

        end for
     end for
  end for
```

© Kavita Bala, Computer Science, Cornell University

Output: just the materials

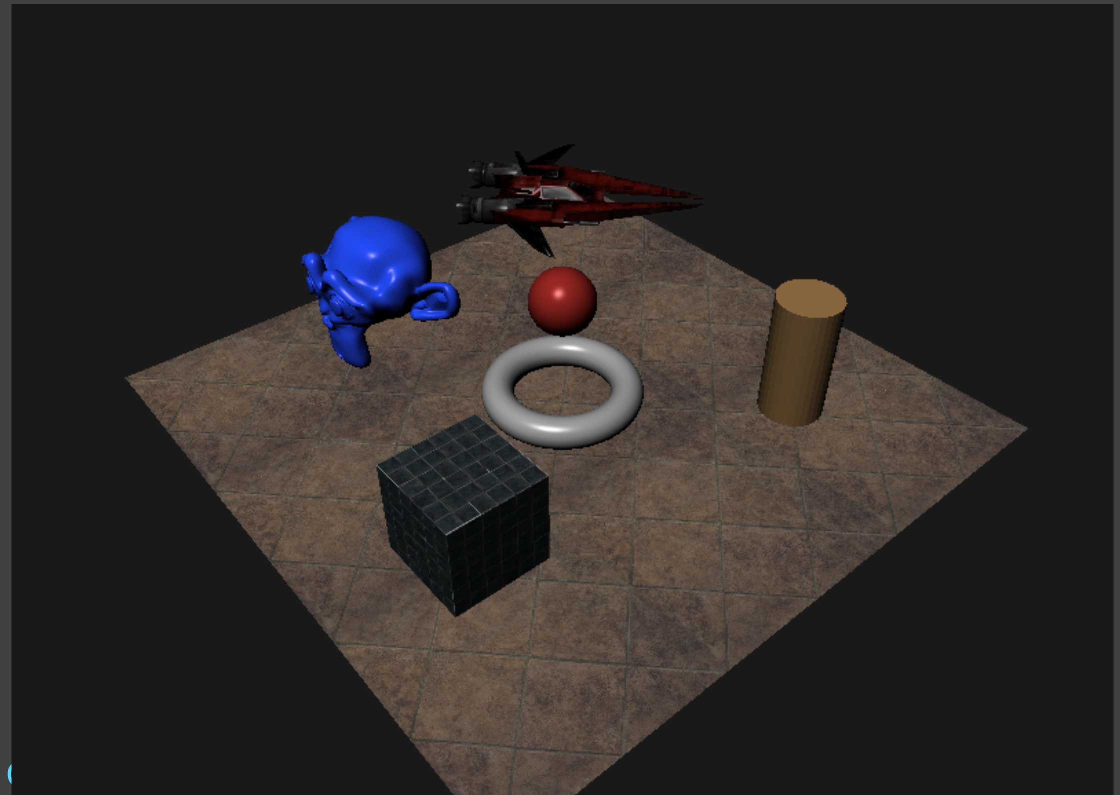


Deferred Shading Step 2

```
for each fragment f in the gbuffer
      framebuffer[x, y] = shade (f)
   end for
```

Key improvement: shade (f) only executed for visible fragments.

Output is the same →



© Kavita Bala, (

The ubershader

- Shader which computes lighting based on g-buffer: has code for all material/lighting models in a single huge shader.

```
shade (f) {
   result = 0;
   if (f is Lambertian) {
      for each light
         result += (n . l) * diffuse;
      end for
   } else if (f is Blinn-Phong) {
      ...
   } else if (f is ...) {
      ...
   }
   return result;
}
```

© Kavita Bala, Computer Science, Cornell University

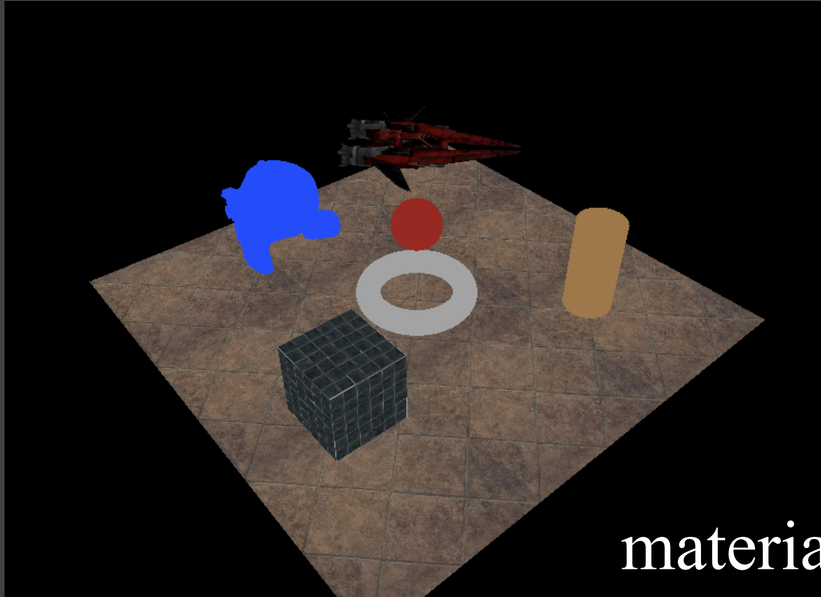
Ubershader inputs

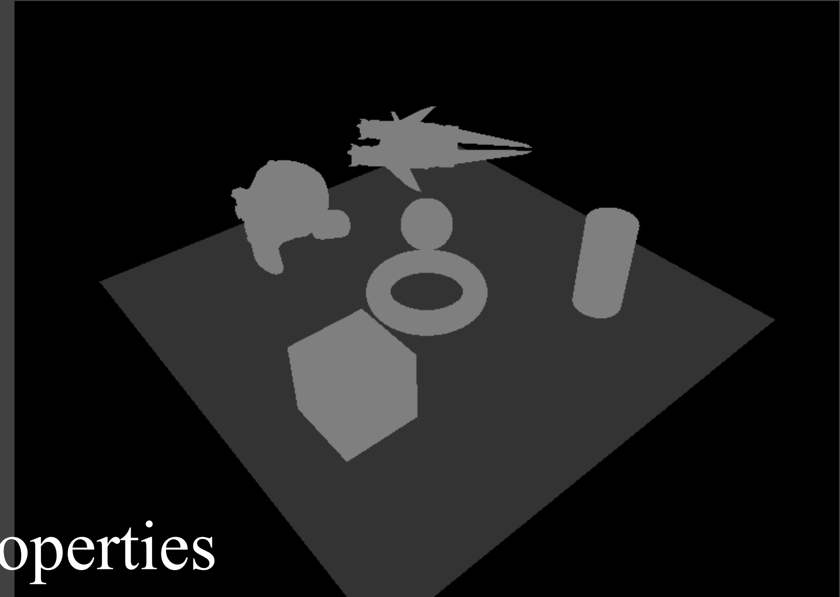
- Need access to all parameters of the material for the current fragment:
  - Blinn-Phong: kd, ks, n
  - Cook-Torrance: kd, ks, alpha
  - etc.
- Also need fragment position and surface normal
- Solution: write all that out from the material shaders:

```
{outputs} = {f.material, f.position, f.normal}
if (depth of f < depthbuffer[x, y])
   gbuffer[x, y] = {outputs}
   depthbuffer[x, y] = depth of f
end if
```

© Kavita Bala, Computer Science, Cornell University

G-buffer: multiple textures

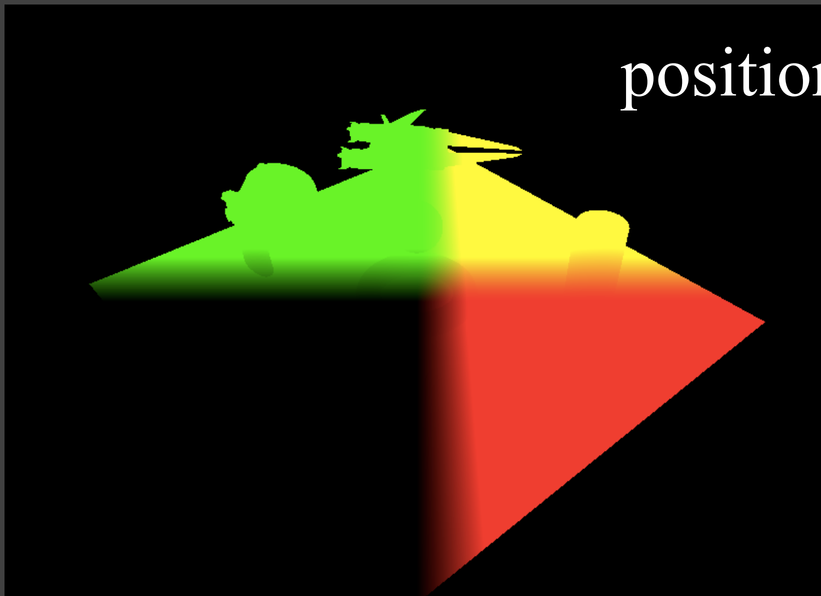


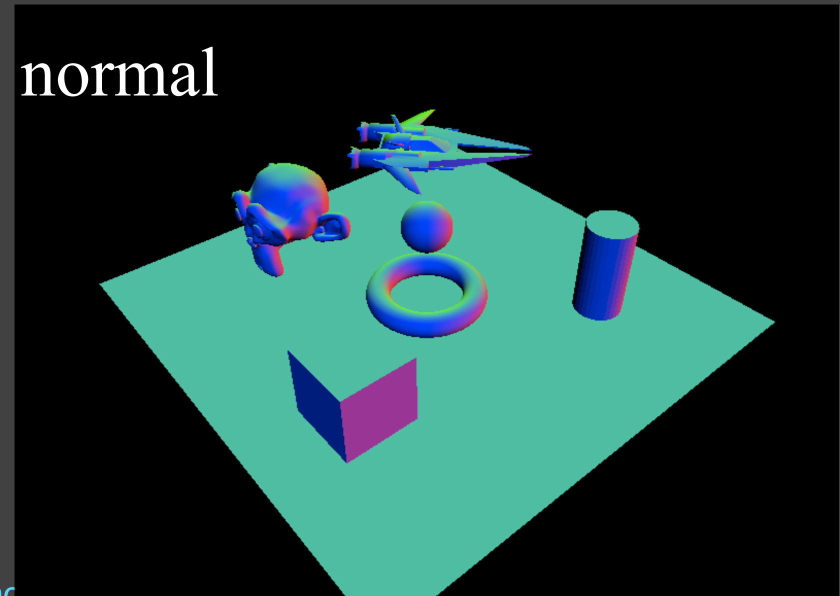


material properties

position

normal



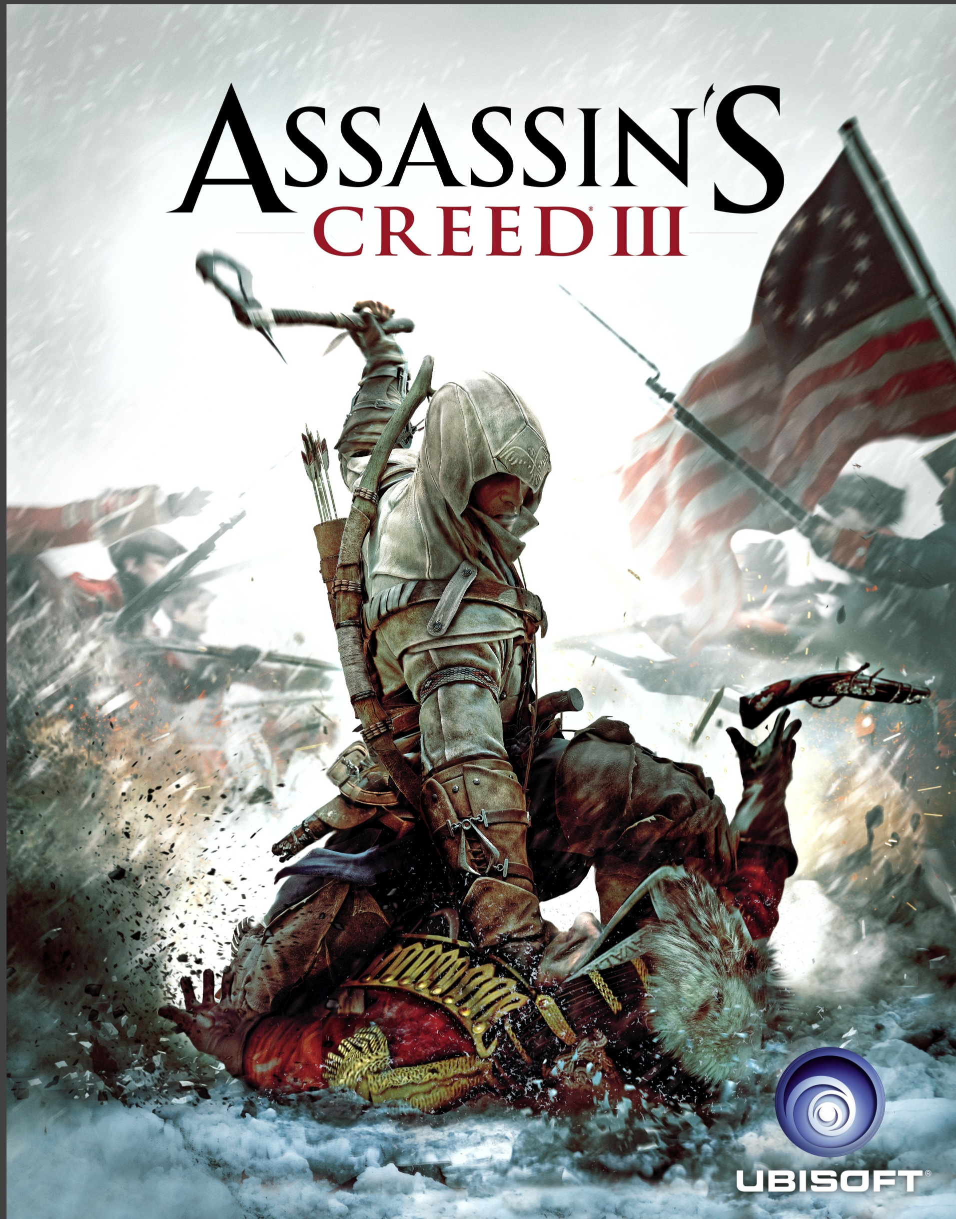


Power of Deferred Shading

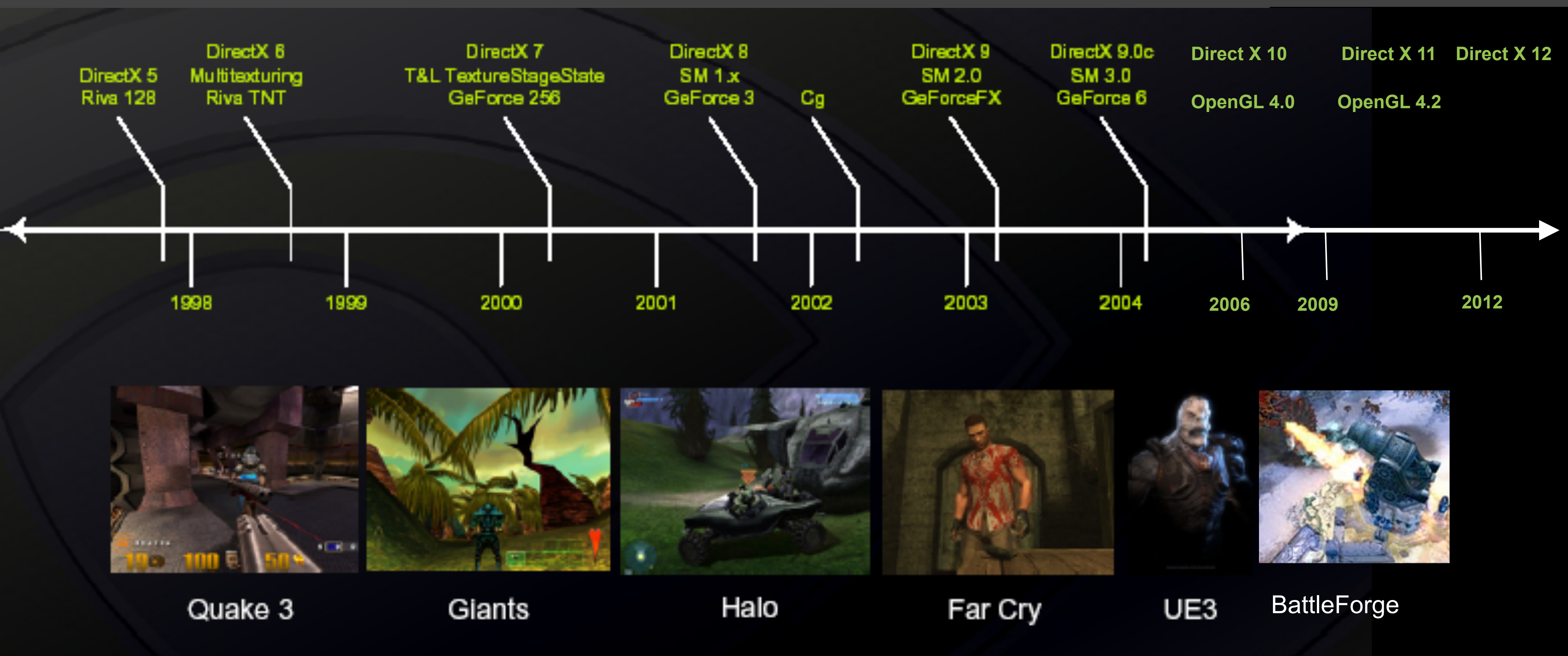
- Can do any image processing between step 1 and step 2!
  - Recall: step 1 = fill g-buffer, step 2 = light/shade
  - Could add a step 1.5 to filter the g-buffer

- Examples:
  - bloom
  - screen-space ambient occlusion
  - high-dynamic range rendering (adaptive exposure control and tone-mapping)
  - silhouette detection/toon rendering
  - "mood effects" (tinted colors, blurry or warped vision)

© Kavita Bala, Computer Science, Cornell University



© Kavita Bala, Computer Science, Cornell University



© Kavita Bala, Computer Science, Cornell University

DirectX 6/OpenGL 1.2: Quake 3

DIFFUSE Lighting

Multitexture Lightmaps

© Kavita Bala, Computer Science, Cornell University

Why multi-pass?

- Scalable

Quake III Engine

1. Passes 1-4: accumulate bump map
2. Pass 5: Diffuse lighting
3. Pass 6: Base texture
4. Pass 7: Specular lighting
5. Pass 8: Emissive lighting
6. Pass 9: Volumetric lighting

© Kavita Bala, Computer Science, Cornell University

OpenGL 1.3/DirectX 7 (2001)

- Improved multi-texturing
  - DOT3 (per-pixel lighting)
  - Dependent texture reads
    - EMBM: Environment Map Bump Mapping
  - Cube maps & projected textures (for shadows)

- Support for HW Transform & Lighting
  - Directional, point, and spot lights
  - Vertex tweening & skinning: better animation
  - Texture coordinate transformation & generation
- Example Hardware
  - GeForce 256, ATI Radeon, Intel Extreme Graphics 2

© Kavita Bala, Computer Science, Cornell University

OpenGL 1.4/DirectX 8,SM 1.x / (2002)

- Programmable vertex shaders
  - Up to 128 floating-point instructions

- Programmable pixel shaders
  - Up to 16 fixed-point vector instructions and 4 textures
  - 3D texture support
  - 1 level of dependent texturing
  - Advanced Render-to-Texture support
- Example Hardware
  - GeForce 3, ATI Radeon 8500, XGI Volari V3, Matrox Parhelia

© Kavita Bala, Computer Science, Cornell University

SM 1.x-era Game: Halo

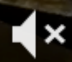
- Vertex shaders used to add Fresnel reflection to ice
- Pixel shaders used to add glow to sun
- Render-to-texture used to distort pistol scope
- Dependent texturing used to animate & light water

Open GL 1.5/DirectX 9, SM 2.0(2003)

- Floating point pixel processing
  - 16/32-bit floating point shaders, render targets & textures
  - Up to 64 vector instructions and 16 textures
- Arbitrary dependent texturing
- Longer vertex processing – 256 instructions
- Multiple Render Targets – up to 16 outputs per pixel
- Example Hardware
  - GeForce FX 5900, ATI Radeon 9700, S3 DeltaChrome

© Kavita Bala, Computer Science, Cornell University

2003


0:06 / 2:07

Matrox G400 Tech Demo EMBM (720p)

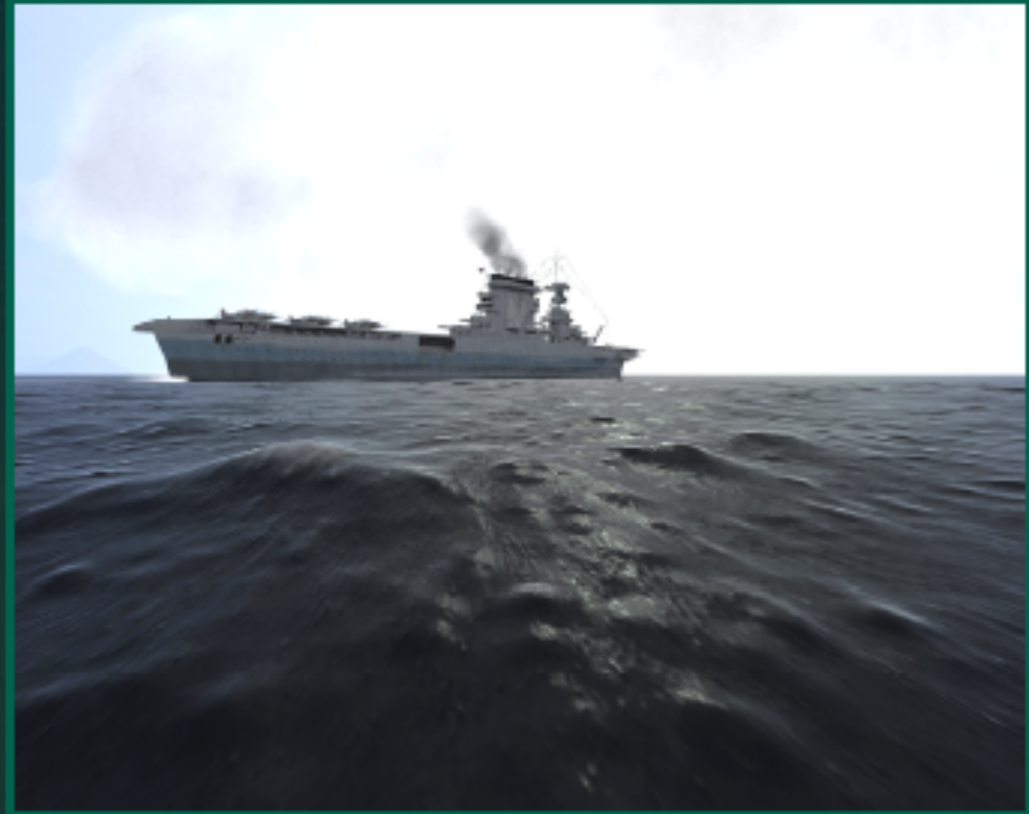
OpenGL 2.0 /DirectX 9.0c, SM 3.0(2004)

- Unified shader programming model
  – Pixel & vertex shader flow control
  – Infinite length vertex & pixel shaders
  – Vertex shader texture lookups
- Floating-point filtering & blending
- Geometry instancing
- Example Hardware
  – GeForce 6800, GeForce 7800 GTX

© Kavita Bala, Computer Science, Cornell University

Vertex Textures

Without Vertex Textures



With Vertex Textures

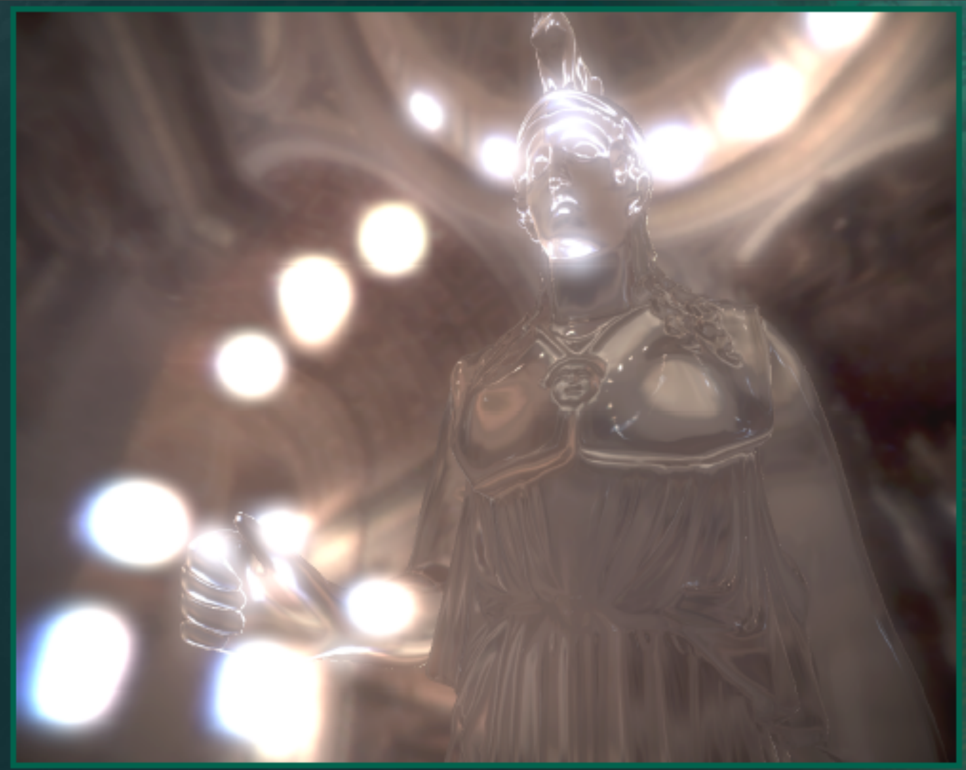
Images used with permission from *Pacific Fighters*. © 2004 Developed by 1C:Maddox Games.
All rights reserved. © 2004 Ubi Soft Entertainment.

© Kavita Bala, Computer Science, Cornell University

SM 3.0-era Game: Unreal Engine 3

- 16-bit blending for high dynamic range lighting
- 16-bit filtering accelerates glow and exposure FX
- Long shaders & flow control for virtual displacement mapping, soft shadows, iridescence, fog, etc.

© Kavita Bala, Computer Science, Cornell University



© Kavita Bala, Computer Science, Cornell University

G70 (Based on NV40): 2005

GeForce 7800

Parallelism

Host / FW / VTF

Cull / Clip / Setup

Z-Cull

Shader Instruction Dispatch

L2 Tex

Fragment Crossbar

Memory Partition

Memory Partition

Memory Partition

Memory Partition

DRAM(s)

DRAM(s)

DRAM(s)

DRAM(s)

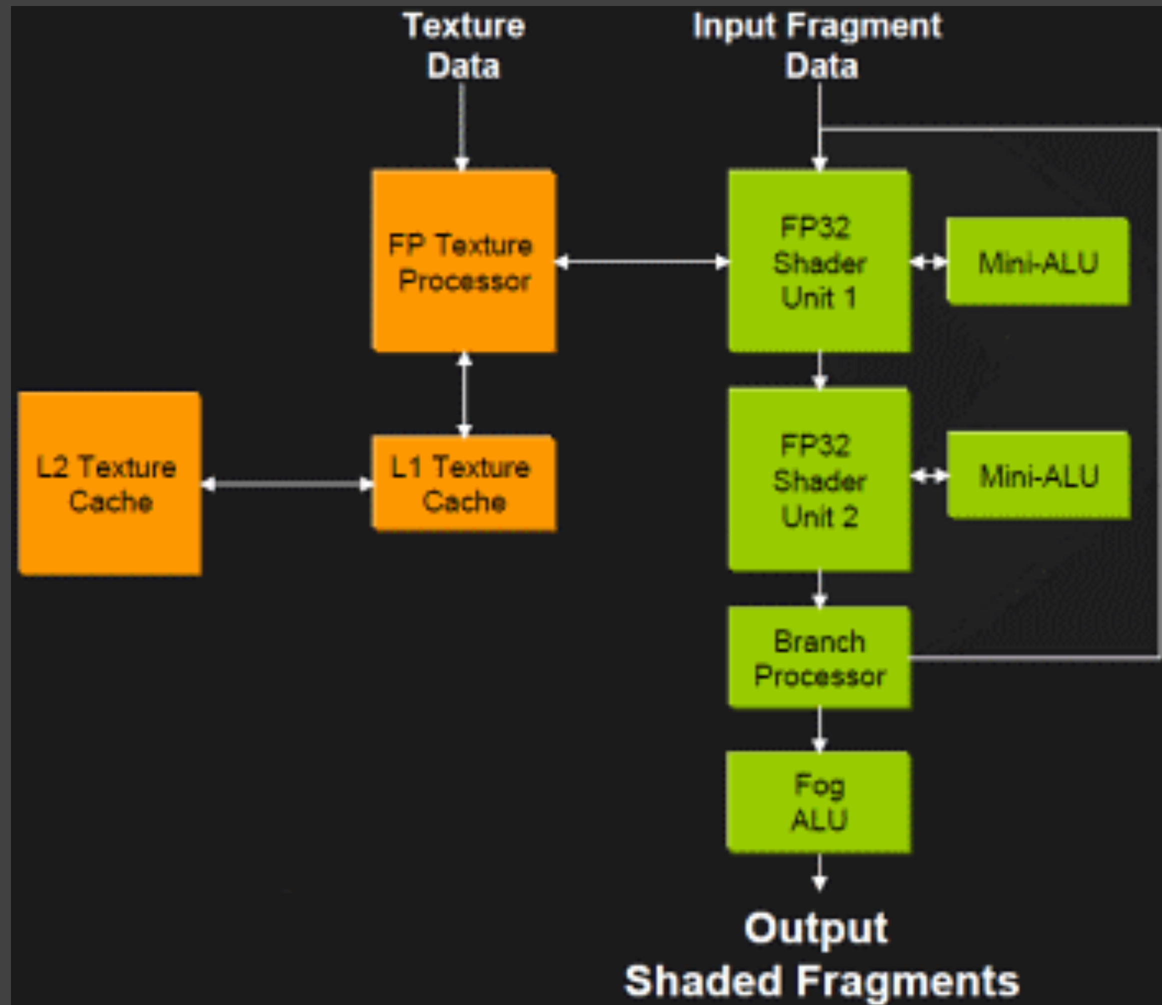
Vertex Shaders (G stage)

- 8 parallel

© Kavita Bala, Computer Science, Cornell University

Fragment Shaders (FG)

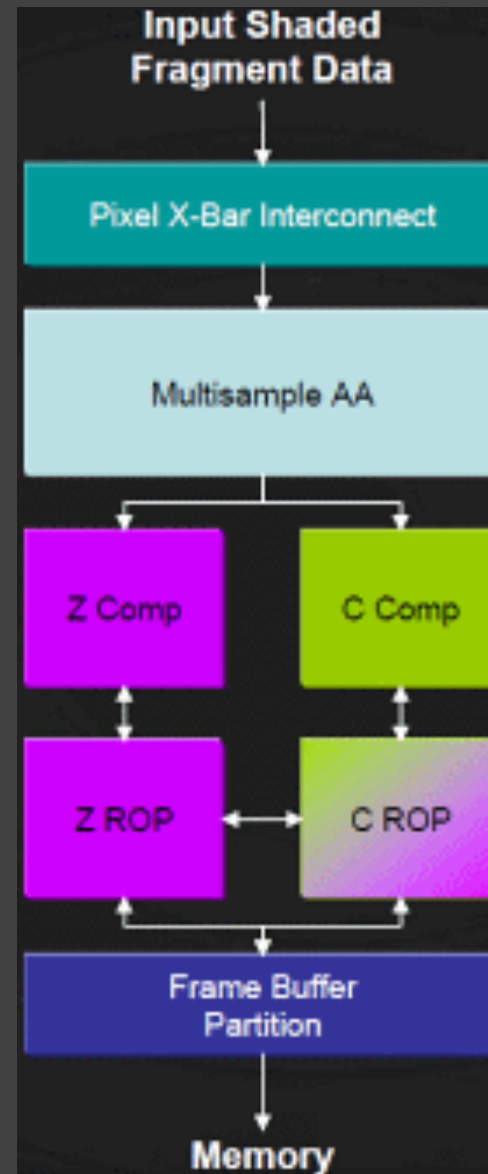
- 24 parallel



© Kavita Bala, Computer Science, Cornell University

Raster Operators (FM)

- 16 ROPs



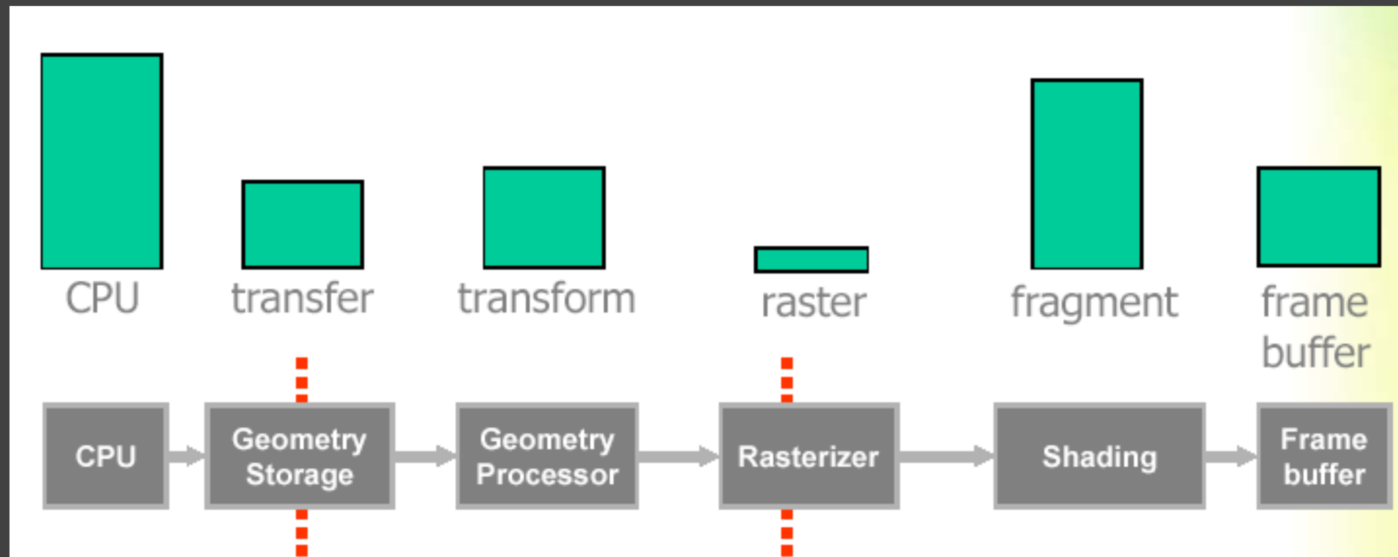
© Kavita Bala, Computer Science, Cornell University

2006 onwards

- Push for unified shader architecture?
  - Why?

© Kavita Bala, Computer Science, Cornell University

Performance



CPU bound   Vertex Bound   Pixel Bound

© Kavita Bala, Computer Science, Cornell University

Bottleneck

- Reduce workload of each stage
- If performance does not change
  - This is not problem
- Else..

© Kavita Bala, Computer Science, Cornell University

Vertex performance

- Reduce triangles
- Reduce vertex shaders

© Kavita Bala, Computer Science, Cornell University

Pixel shader performance

- Pixel shader
  - Does resolution change performance?
- Reasons
  - Memory bandwidth
  - Shader performance
  - Texture filtering

© Kavita Bala, Computer Science, Cornell University

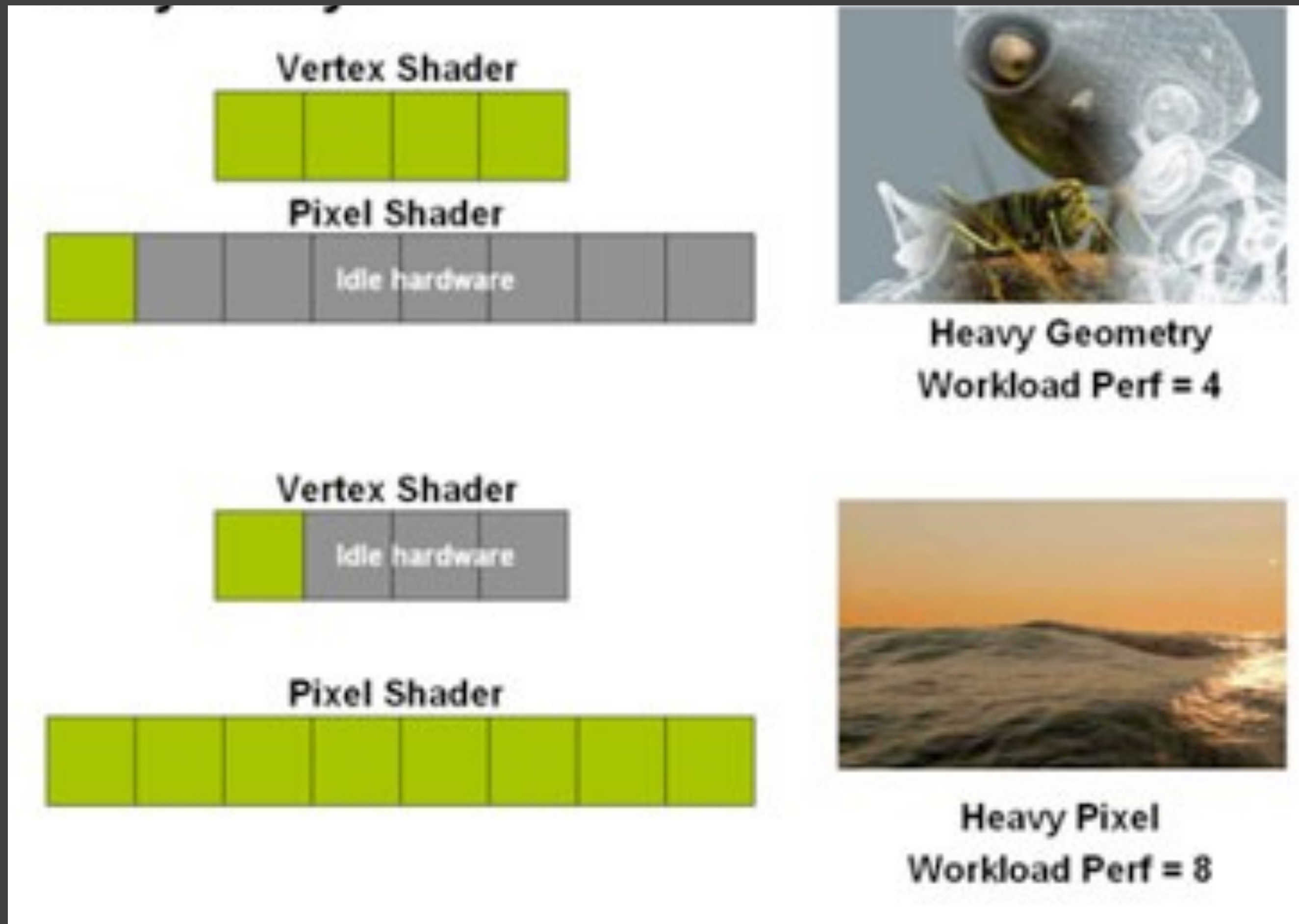
Why Unified Shader Architecture?

- Load balancing
  - Guesswork before
  - Unified lets GPU do it right

© Kavita Bala, Computer Science, Cornell University

NV80 onwards

- Unified shader architecture



© Kavita Bala, Computer Science, Cornell University

Why Unified architectures?

Unified Shader

Heavy Geometry
Workload Perf =12

Unified Shader

Heavy Pixel
Workload Perf = 12

© Kavita Bala, Computer Science, Cornell University

G80 (2006)

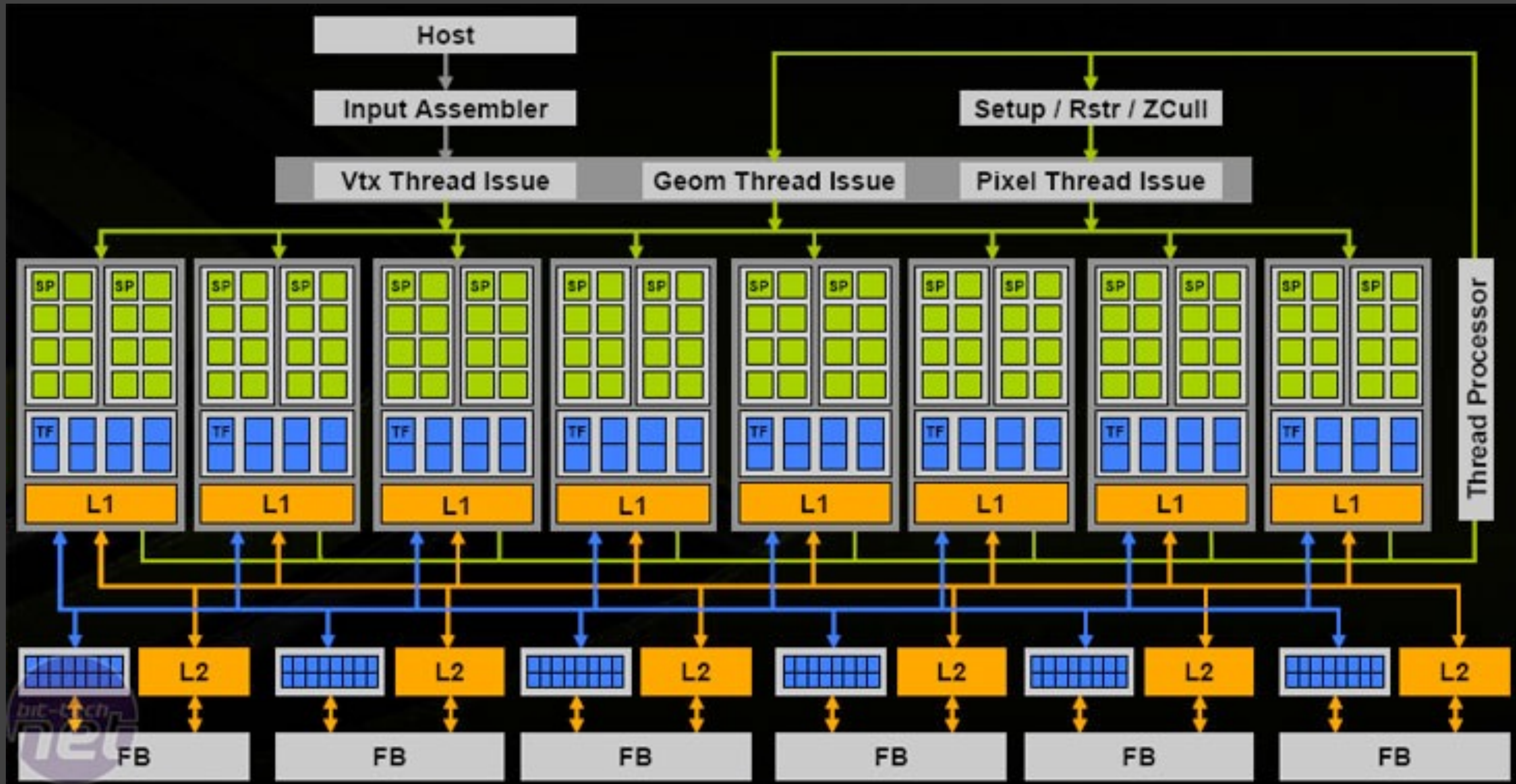
- Fundamental change in architecture

- Full Direct3D 10 Support
- DirectX10 Shader Model 4.0 Support
  - Vertex Shader 4.0, Geometry Shader 4.0, Pixel Shader 4.0
  - Internal 128-bit Floating Point (FP32) Precision
- Unlimited shader lengths
- Up to 128 textures per pass

- Support for FP32 texture formats with filtering
- Non-Power of two texture support
- 8 Multiple Render Targets

© Kavita Bala, Computer Science, Cornell University

G80



© Kavita Bala, Computer Science, Cornell University

GTX TITAN GPU Engine Specs:

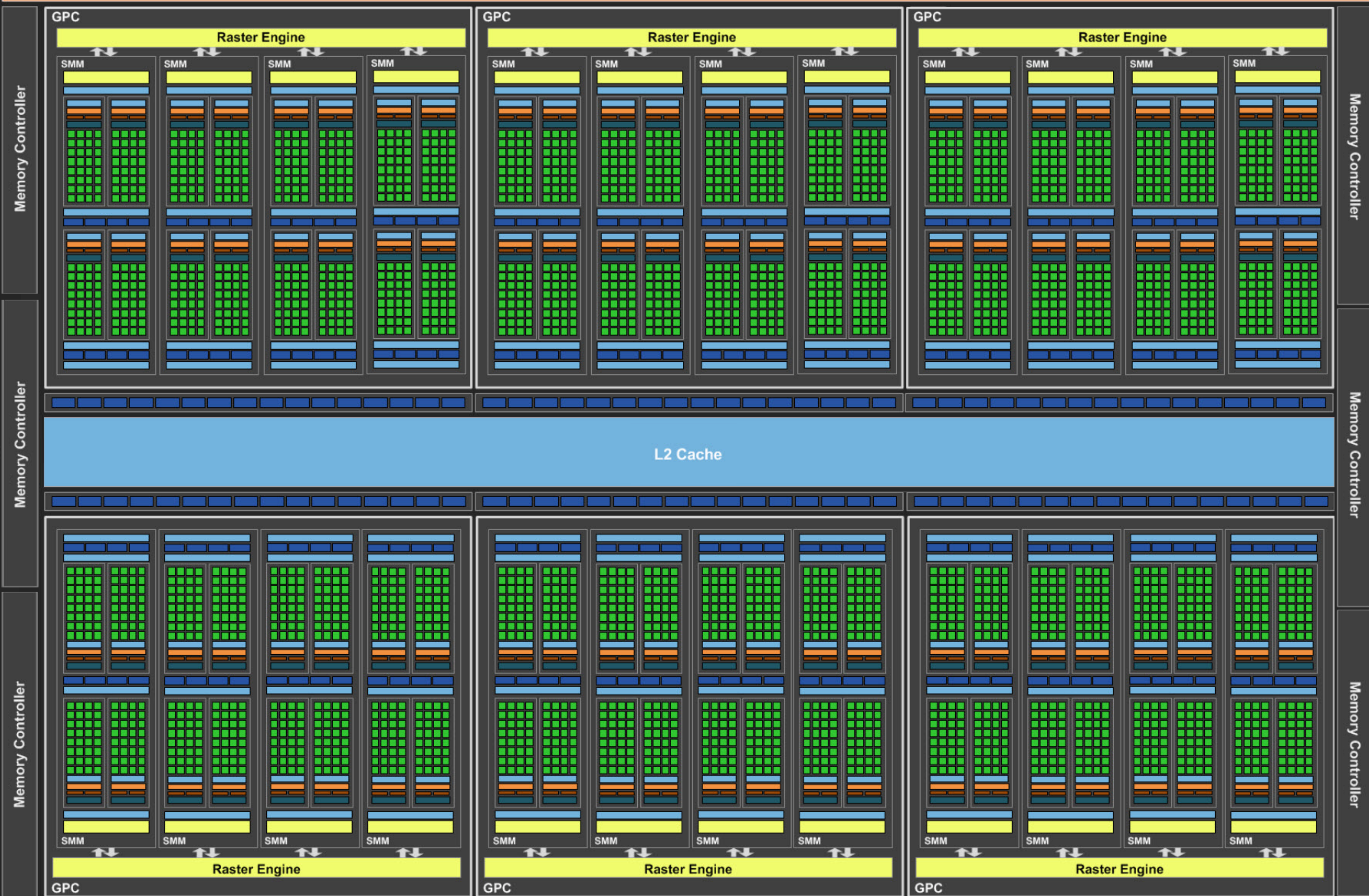
CUDA Cores	2688
Base Clock (MHz)	837
Boost Clock (MHz)	876
Texture Fill Rate (billion/sec)	187.5

GTX TITAN Memory Specs:

Memory Clock	6.0 Gbps
Standard Memory Config	6144 MB
Memory Interface	GDDR5
Memory Interface Width	384-bit GDDR5
Memory Bandwidth (GB/sec)	288.4

GTX TITAN Support:

Important Technologies	GPU Boost 2.0, PhysX, TXAA, NVIDIA G-SYNC-ready, SHIELD-ready
Other Supported Technologies	3D Vision, CUDA, Adaptive VSync, FXAA, NVIDIA Surround, SLI-ready
OpenGL	4.4
Bus Support	PCI Express 3.0
Certified for Windows 7, Windows 8, Windows Vista, or Windows XP	Yes
3D Vision Ready	Yes
Microsoft DirectX	12 API
Blu Ray 3D	Yes
3D Gaming	Yes
3D Vision Live (Photos and Videos)	Yes



© Kavita Bala, Computer Science, Cornell University

Architectural Trends

- More general purpose
- More shaders: vertex, pixel, geometry, tesselation

- Longer shaders
  - Length of shaders: 16, 128, … unbounded
- More bits
  - More texturing: more, bigger, and greater precision
  - Better floating point
  - Better HDR support

- More SIMD cores
  - More parallelism

© Kavita Bala, Computer Science, Cornell University

Hardware

- 1999-2007: Frontier times

- 2015: Heterogeneous Parallel Computing
  - New frontier
  - CPUs (multicore)
  - GPUs (SIMD or MIMD clusters)

- Programming these is going to be hard

© Kavita Bala, Computer Science, Cornell University