EE382V: Principles in Computer Architecture

Parallelism and Locality

Fall 2008

# Lecture 10 – The Graphics Processing Unit

Mattan Erez

The University of Texas at Austin

# Outline

- What is a GPU?
- Why should we care about GPUs?
- 3D graphics pipeline
- Programmable graphics pipeline

- Most slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - From The University of Illinois ECE 498AI class
- Some slides courtesy Massimiliano Fatica (NVIDIA)

# A GPU Renders 3D Scenes

- A ***Graphics Processing Unit (GPU)*** accelerates rendering of 3D scenes
  - Input: description of scene
  - Output: colored pixels to be displayed on a screen
- Input:
  - Geometry (triangles), colors, lights, effects, textures
- Output:

# State of the Art in 1985

- First movie from Pixar – Luxo Jr.
- 2 – 3 hours per frame on a Cray-1 supercomputer

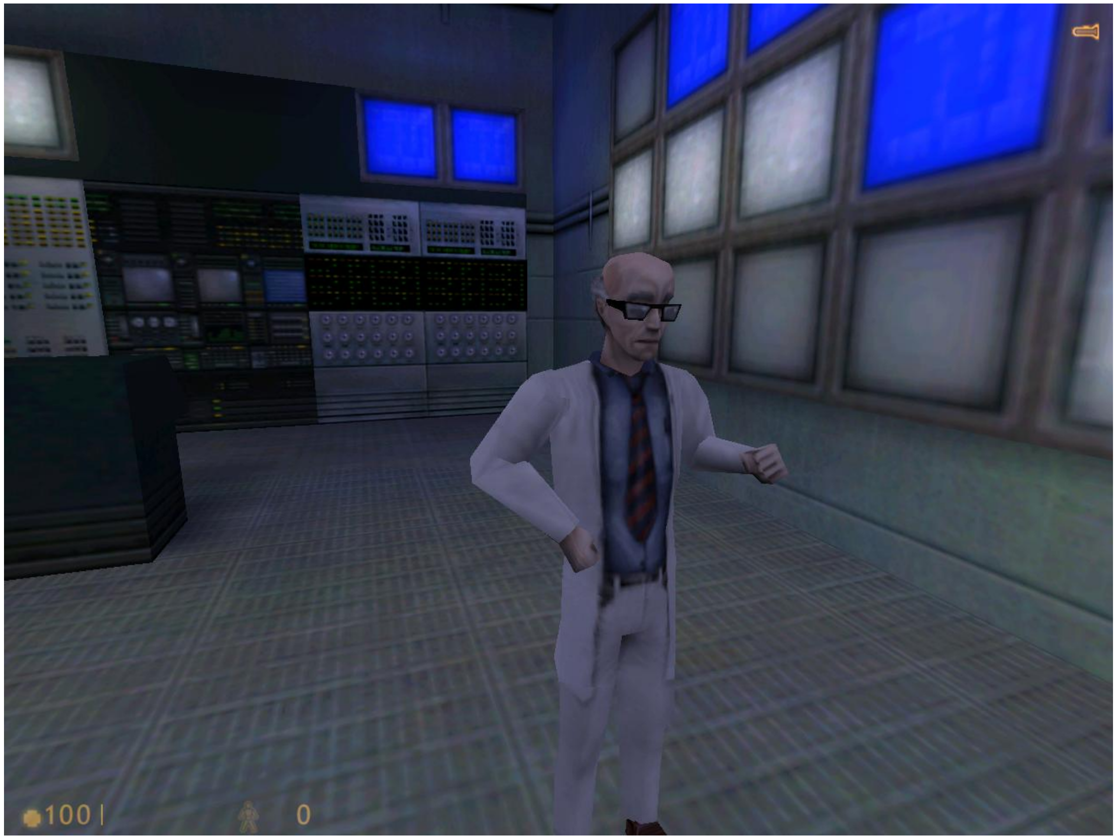
- Today: 1/30$^{th}$ of a second on a PC
  - Over 300,000x faster

- Still not even close to where we need to be… but look how far we've come!

# GPU Scene Complexity Defined by Standard Interfaces (DirectX and OpenGL)

- DirectX and OpenGL define the interface between applications and the GPU
- **Geometry** describes the objects and layout
  - Triangles (vertices) describe all objects
    - Can have millions of triangles per scene
  - Can modify triangle surfaces
    - Bumps, ripples, …
  - Lights are part of the scene geometry
- **Pixel Shaders** describe how to add color
  - Colors of triangle vertices
  - Textures (patterns)
  - How to determine color of pixels within a triangle
  - …

# GPUs in 1997 – DirectX 5



# GPUs in 1998 – DirectX 6

# GPUs in 2000 – DirectX 7

# GPUs in 2001 – DirectX 8

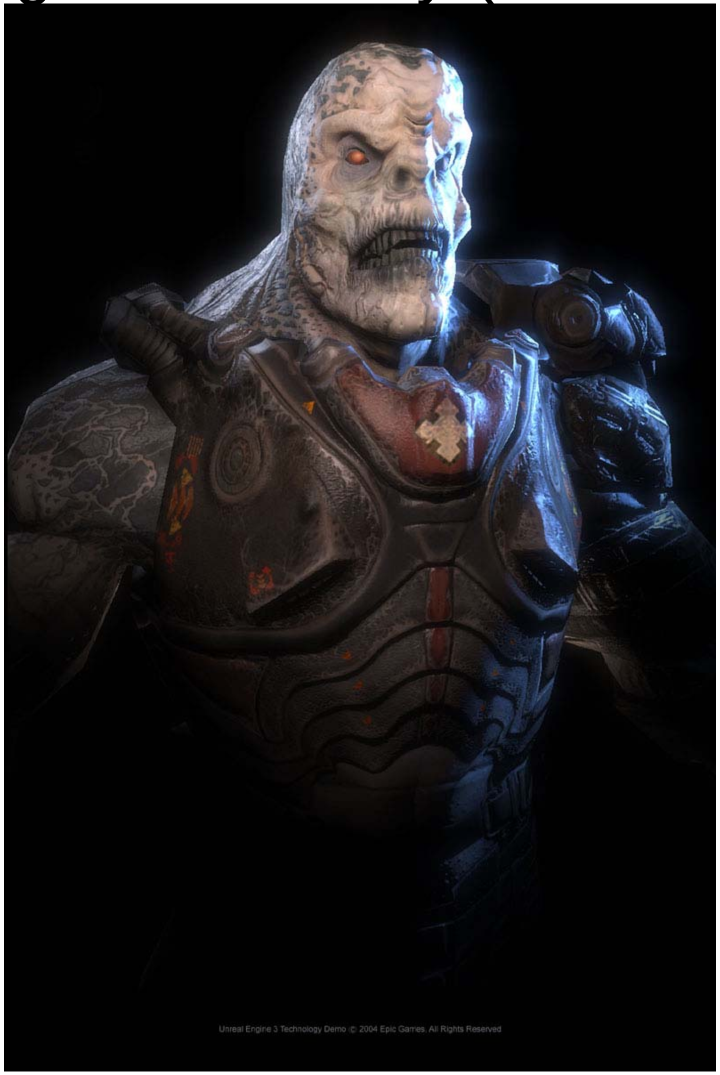
- First programmable graphics (Shader Model 1)

# GPUs in 2003 – DirectX 9

- More programmability (Shader Model 2)

# GPUs in 2004 – DirectX 9.0c

- Yet more programmability (Shader Model 3)



# GPUs in 2007 – DirectX 10

- Full programs in pipeline (Shader Model 4)


DirectX 9

DirectX 10

# Outline

- What is a GPU?
- Why should we care about GPUs?
- 3D graphics pipeline
- Programmable GPUs

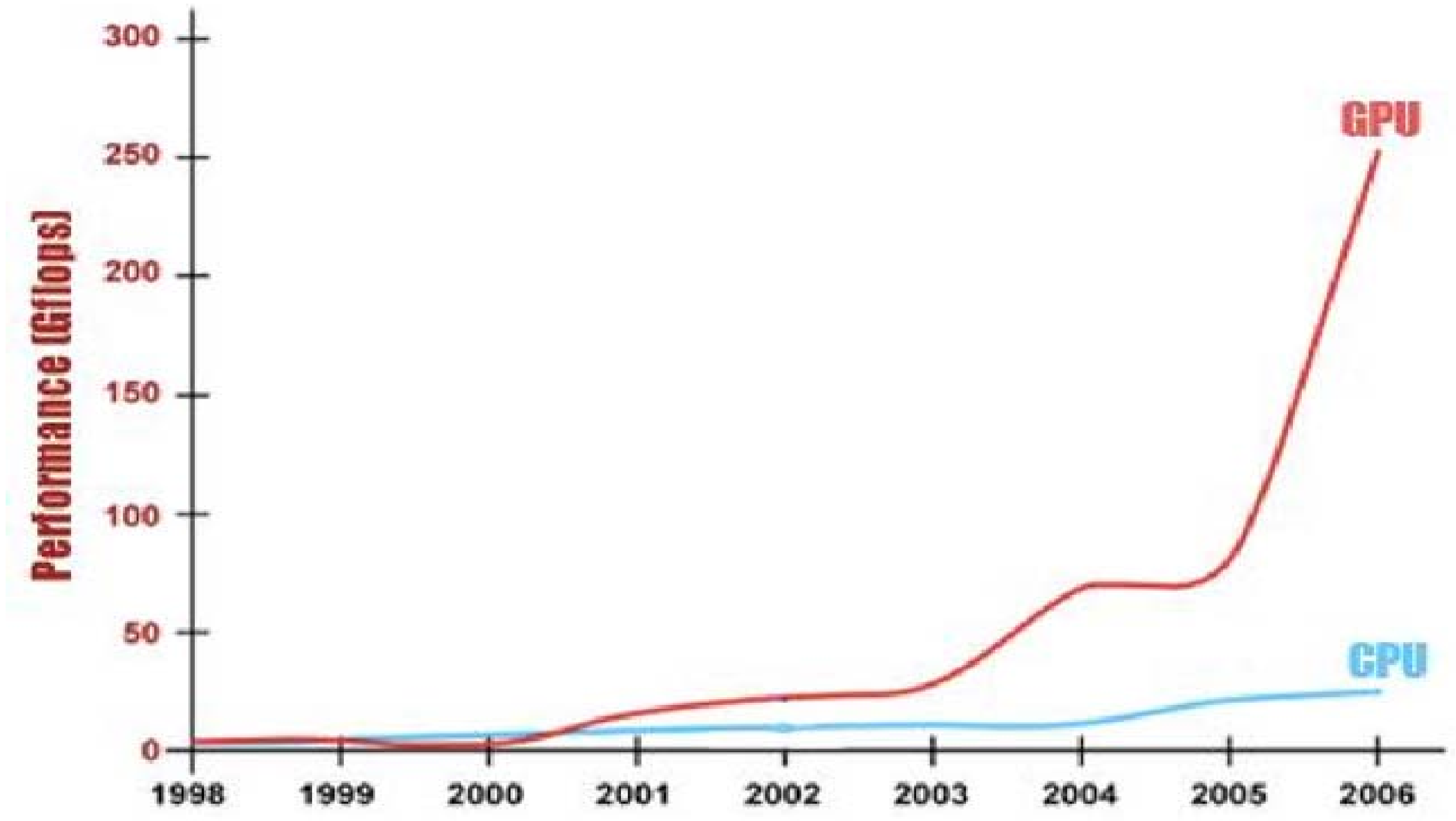
- Many slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - From The University of Illinois ECE 498AI class
- Other slides courtesy Massimiliano Fatics (NVIDIA)

# Complexity and Quality are Orders of Magnitude Better





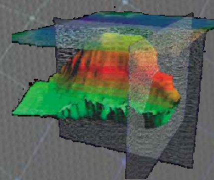
# GPU Performance is Increasing Much Faster than CPUs

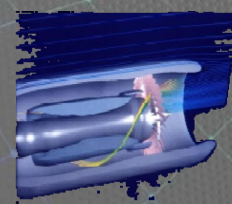


# The GPU is Now a Fully Programmable General Purpose Processor

- Programmability needed by graphics – can be exploited for GP computation

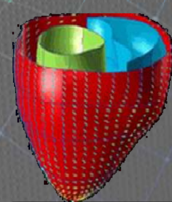
**Computational Geoscience**

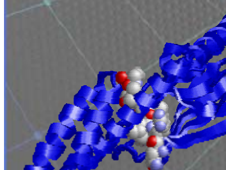




**Computational Modeling**

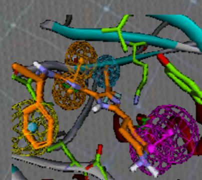
**Computational Medicine**

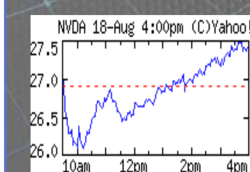




**Computational Biology**

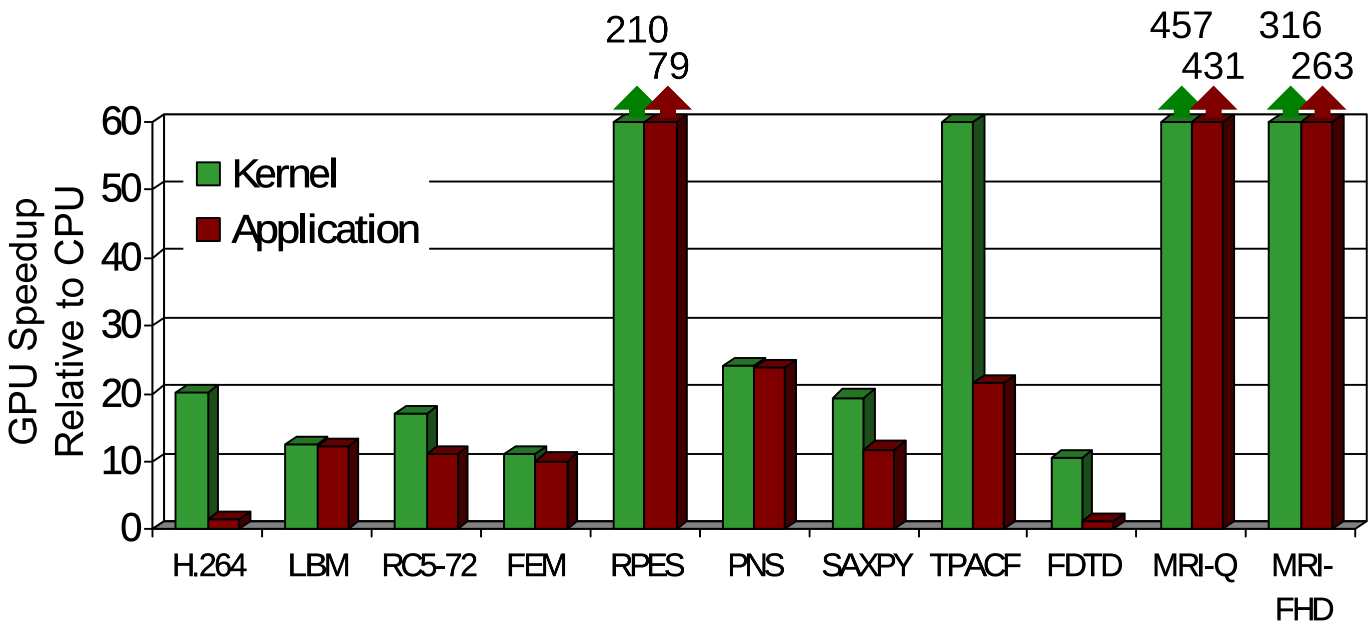
**Computational Chemistry**






**Computational Finance**

# Speedup of Applications




- GeForce 8800 GTX vs. 2.2GHz Opteron 248
  - 10x speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
  - 25x to 400x speedup if the function's data requirements and control flow suit the GPU and the application is optimized
- Keep in mind that the speedup also reflects how suitable the CPU is for executing the kernel

# GPU and CPU Architectures are Starting to Converge

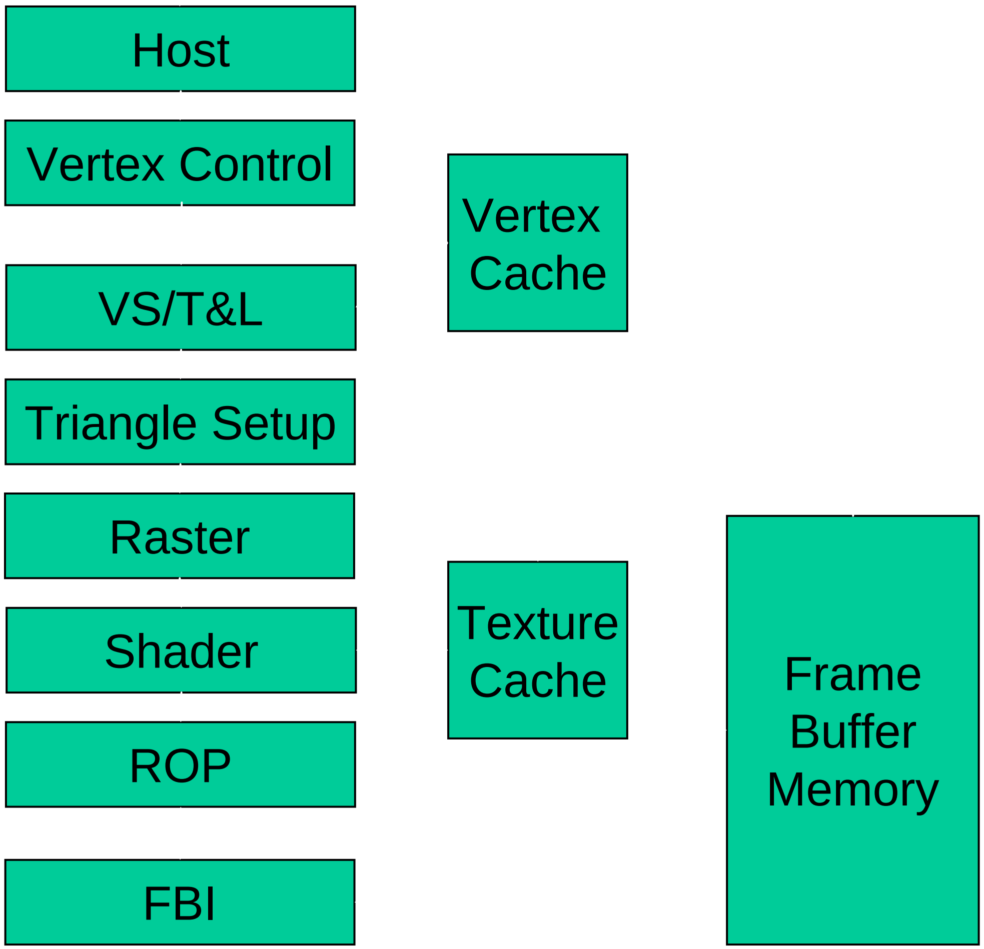
| | CPUs | GPUs |
|---|---|---|
| 1997 | no explicit parallelism | not programmable |
| 2000 | explicit short vectors | emerging programmability (2001 – 2002), “infinite” DP |
| 2003 | explicit short vectors<br>explicit threading (~2) | fully programmable<br>explicit “infinite” DP<br>no scatter |
| 2006 | explicit short vectors<br>explicit threading (~4) | explicit vectors<br>explicit threading (~16) |
| 2009? | explicit vectors<br>explicit threading (>16) | explicit vectors<br>explicit threading (>16) |

# Outline

- What is a GPU?
- Why should we care about GPUs?
- 3D graphics pipeline
- Programmable GPUs

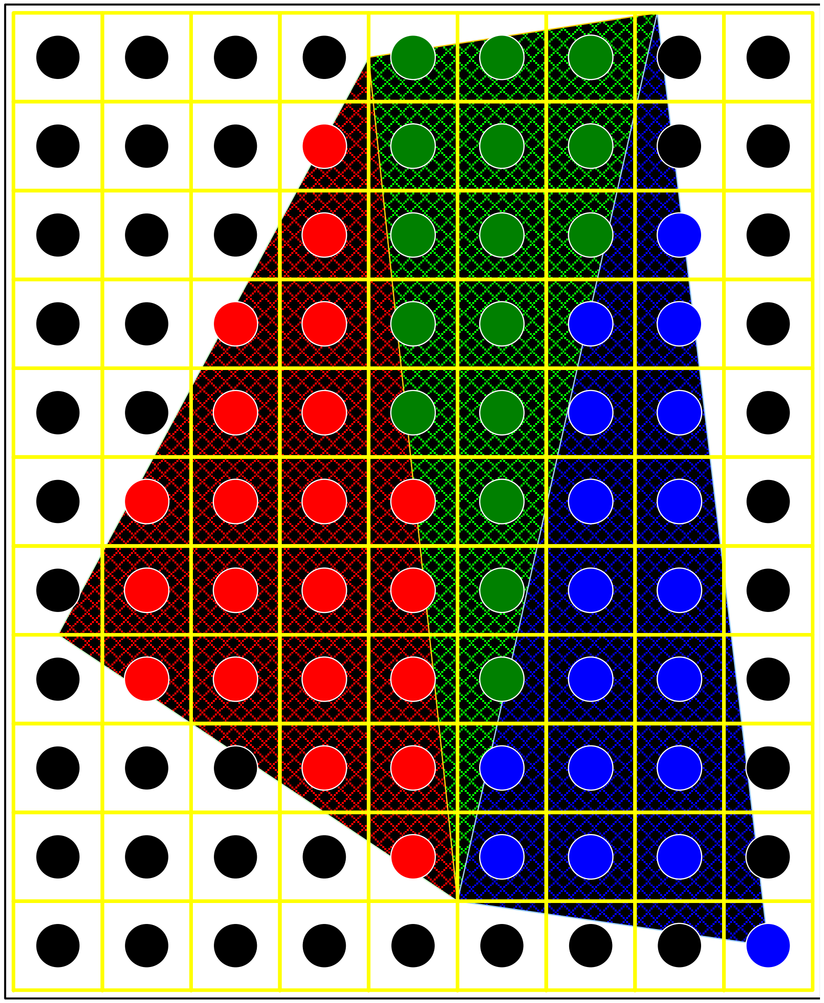
- Many slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - From The University of Illinois ECE 498AI class
- Other slides courtesy Massimiliano Fatics (NVIDIA)

# The NVIDIA GeForce Graphics Pipeline

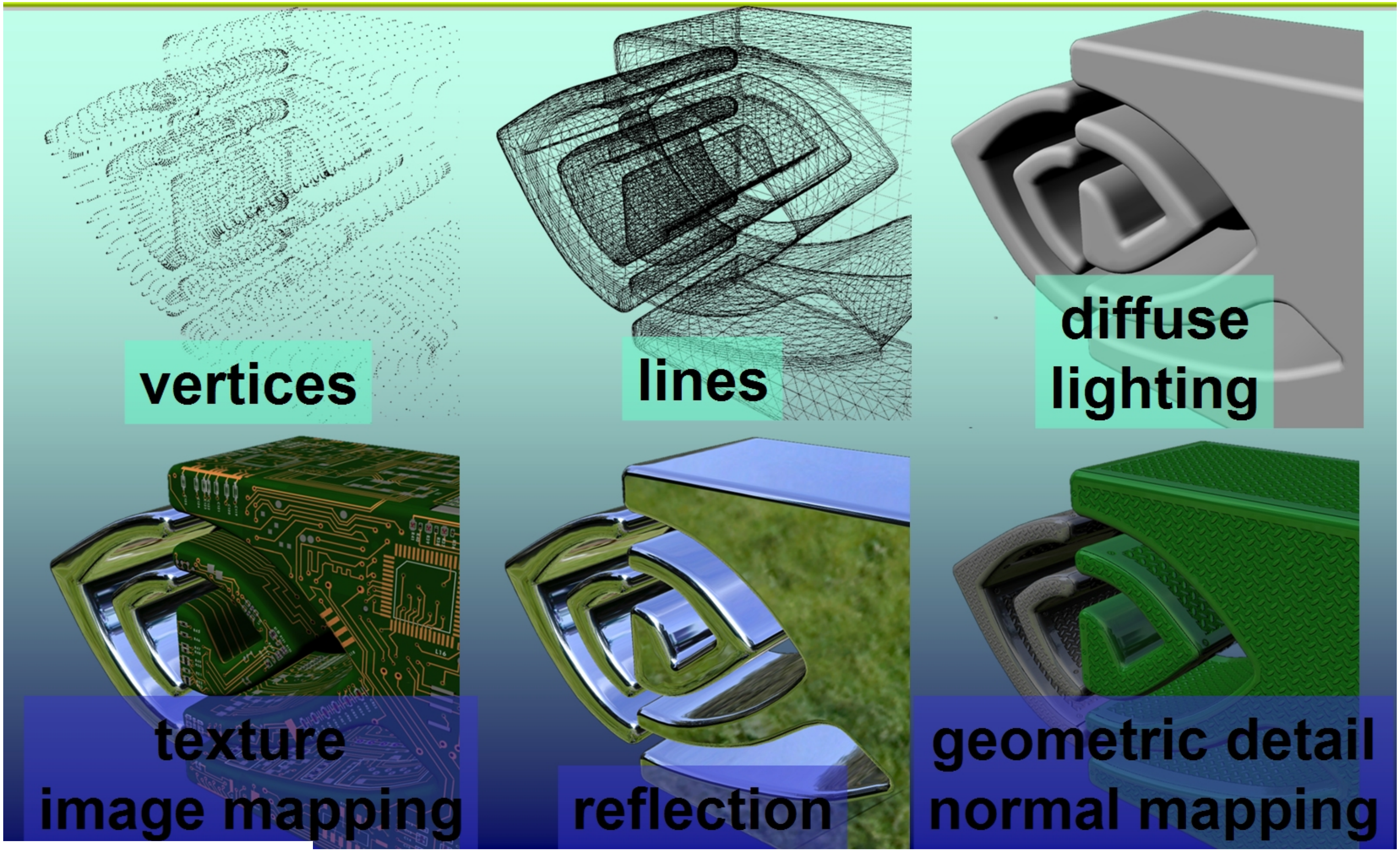


# Color Framebuffer ("Display")

- 2D array of R,G,B color *pixel* values
- 8 bits (256 levels) per color component
- Three 8-bit components can represent 16 million different colors, including 256 shades of gray
- 4th component: *alpha*; used for blending
- Typical high end: 2048x1536 pixels



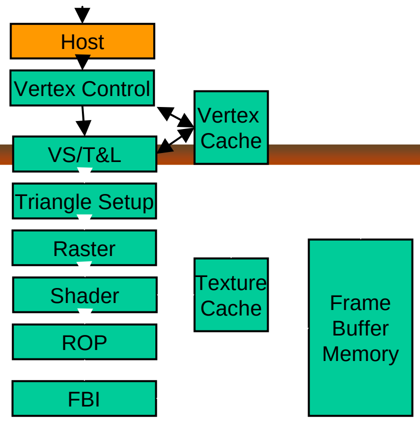
# Describing an Object



# Feeding the GPU

- GPU accepts a sequence of commands and data
  - Vertex positions, colors, and other shader parameters
  - Texture map images
  - Commands like “draw triangles with the following vertices until you get a command to stop drawing triangles”.
- Application pushes data using Direct3D or OpenGL
- GPU can pull commands and data from system memory or from its local memory

# Host Interface

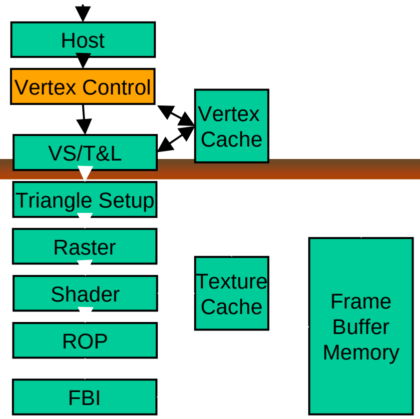



- Bus Interface
- DMA Engines
- Class Interfaces
  - This enables our Unified Driver Architecture
- How the CPU communicates to our GPU
- How our GPU communicates back to the CPU
- How we move data back and forth to the CPU

# Transform Vertex Positions

- Why transform vertices?
  - Rotate, translate and scale each object to place it correctly among the other objects that make up the scene *model*.
  - Rotate, translate, and scale the entire scene to correctly place it relative to the camera's position, view direction, and field of *view*.
- How?
  - Multiply every floating point vertex position by a combined 4x4 model-view matrix to get a 4-D [x y z w] *eye-space* position

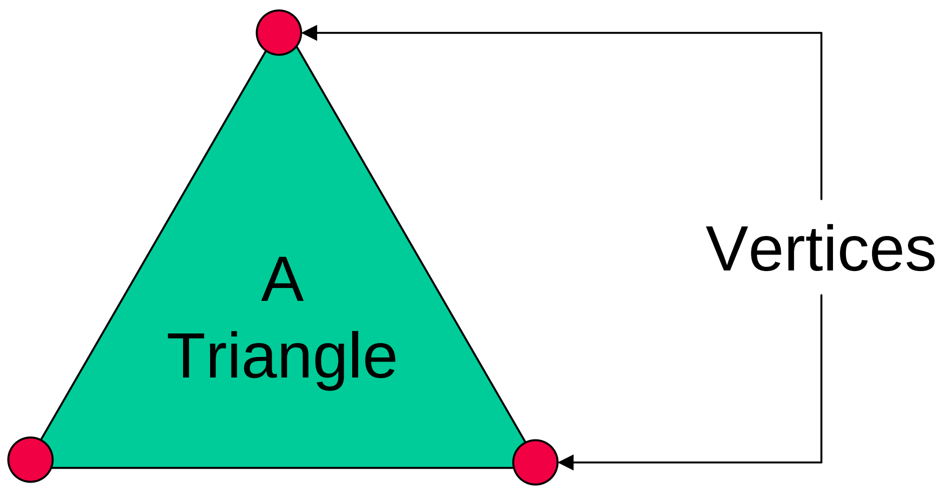
# Vertex Control



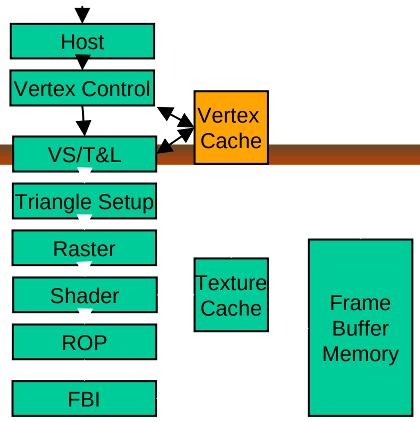

- Receives parameterized vertex data
- Inputs data to vertex cache
- Formats vertices for processing
- Data can come to our GPU in a variety of formats
- Vertex control organizes vertex data into a consistent, hardware understandable format

# What's a Vertex?

- The defining "corners" of a primitive
- For GeForce that means a triangle



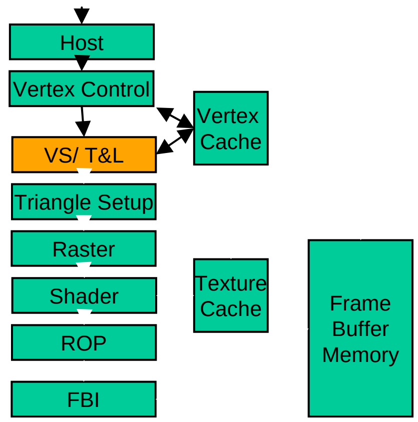
# Vertex Cache




- Temporary store for vertices, used to gain higher efficiency
- Re-using vertices between primitives saves AGP/PCI-E bus bandwidth
- Re-using vertices between primitives saves GPU computational resources
- A vertex cache attempts to exploit “commonality” between triangles to generate vertex reuse
- Unfortunately, many applications do not use efficient triangular ordering

# Geometry/Vertex Processing

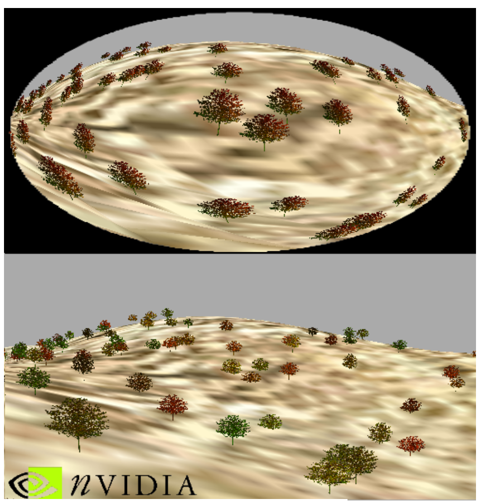
- Transform & Lighting
  - Fixed set of transformations and effects



# Vertex Processing Examples

- Deformation
- Warping
- Procedural Animation

- Range-based Fog
- Elevation-based Fog

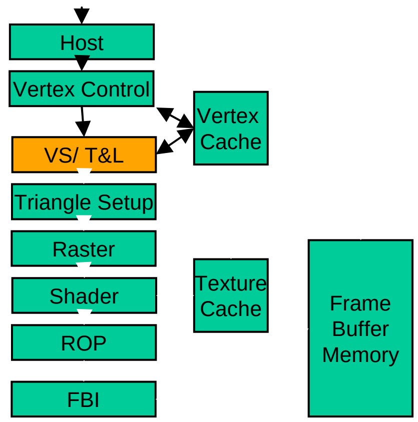


- Lens Effects

- Animation
  - Morphing
  - Interpolation

# Geometry/Vertex Processing

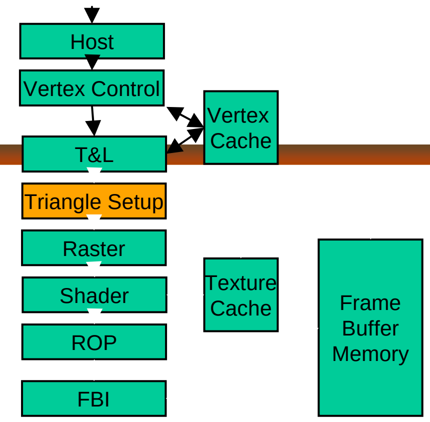
- Transform & Lighting
  - Fixed set of transformations and effects
- Today: "Vertex Shading"
  - Programmable programs run on a per vertex basis
  - One vertex in → One vertex out: **DP "stream" processing**
  - "Flow-through" programming architecture



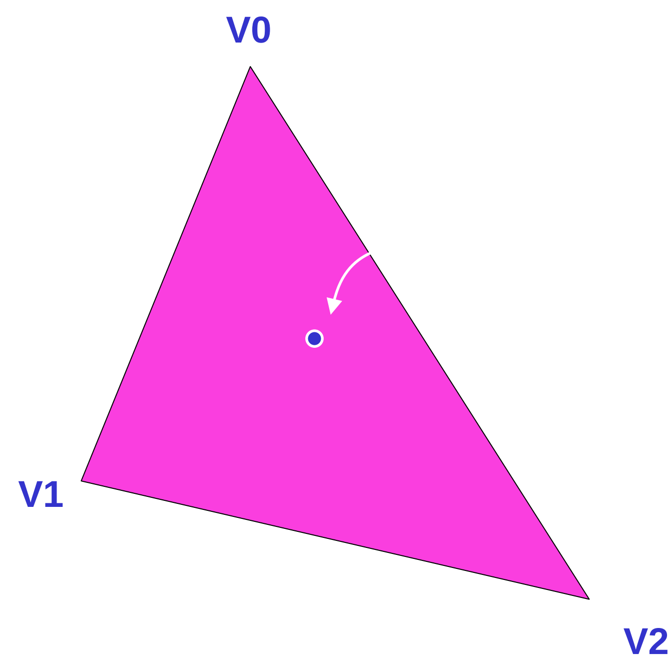
# Vertex Lighting

- Vertex lighting generates a color value at each vertex.
- Simplest GPU "lighting": application calculates and delivers an (R,G,B) triplet for every vertex.
- A more typical GPU lighting equation models the physics of light transport. We sum contributions of:
  - Ambient – uniform light from all directions
  - Emissive – light given off by the object itself
  - Specular – glossy, mirror-like reflections
  - Diffuse – dull, matte-finish reflections

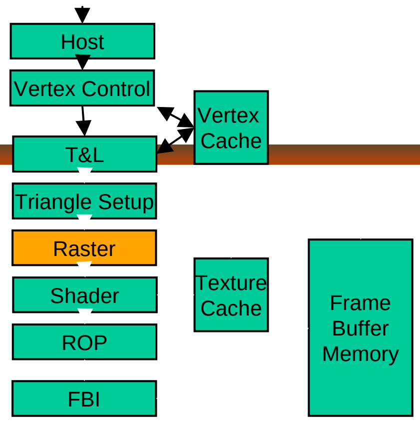
# Triangle Setup




- Each vertex of each polygon contains parameters used by Triangle Setup – typically 4 or more
- In Setup, this vertex data is used to create a map relating pixel coordinates with the variables that will ultimately determine their color



# Rasterization

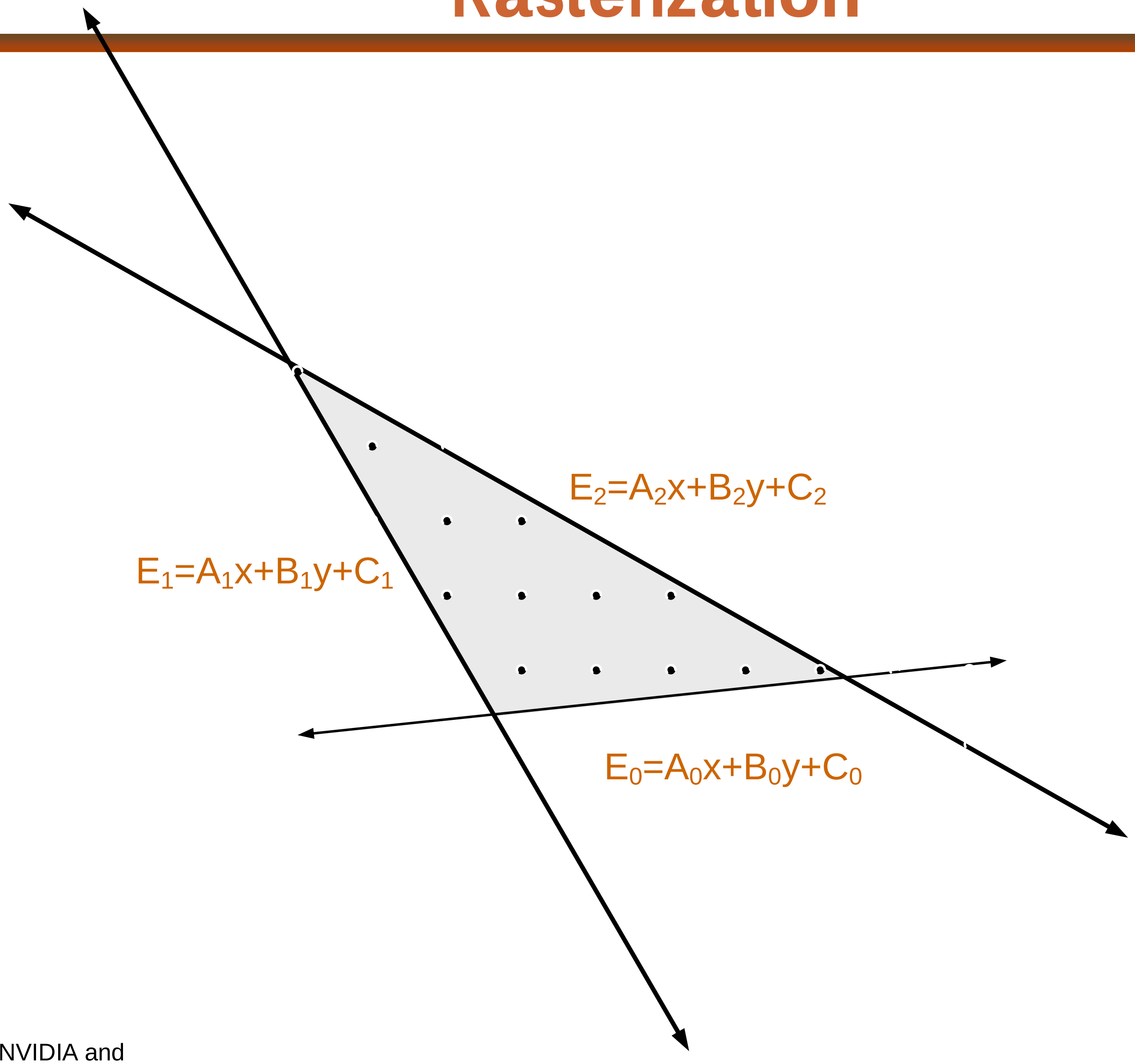



- Rasterization is the process of determining which pixels are contained in each triangle
- For each of these pixels, the rasterizer creates the necessary information for pixel shading
- It includes information like
  - Position
  - Color
  - Texture coordinates for each pixel
  - Pattern for rasterization (which helps fill texture cache ahead of time)
- In GeForce, it also includes Z-Occlusion

# Rasterization

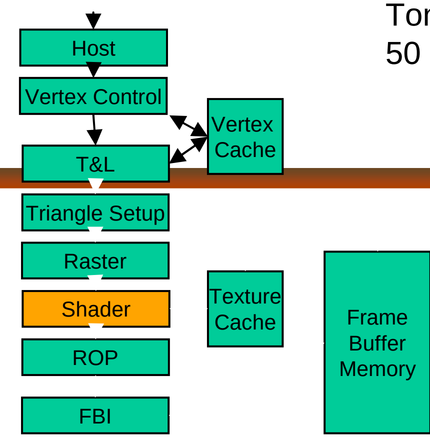
- Given a triangle, identify every pixel that belongs to that triangle
- Point Sampling
  - A pixel belongs to a triangle if and only if the center of the pixel is located in the interior of the triangle
  - Evaluate 3 edge equations of the form E=Ax+By+C, where E=0 is exactly on the line, and positive E is towards the interior of the triangle.

# Rasterization



# Shading

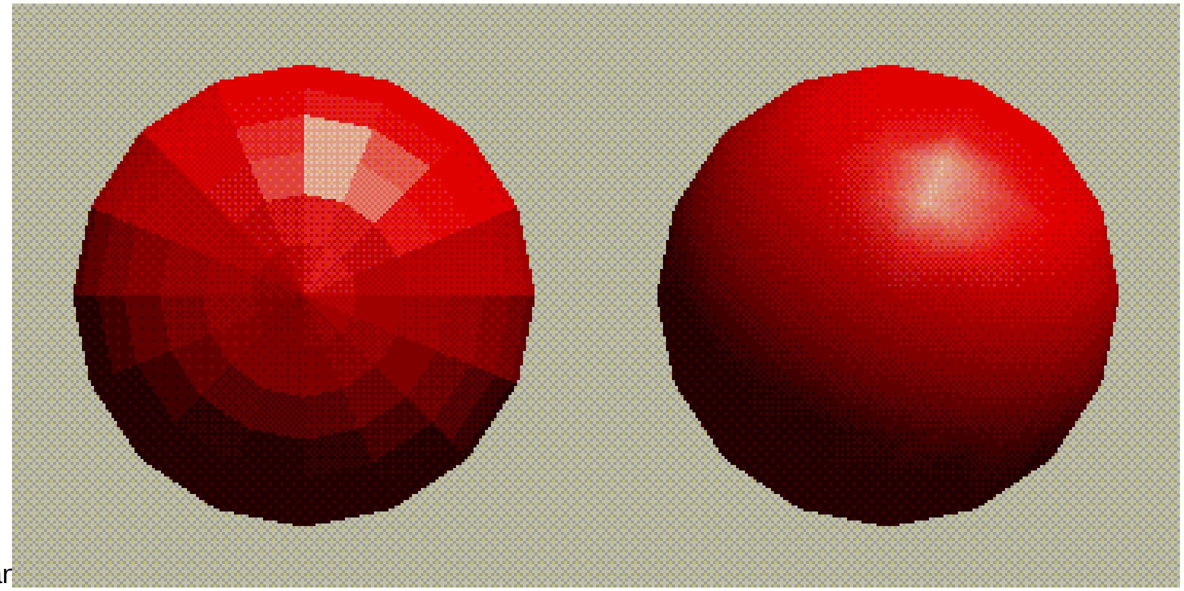
Tony
50




- Shading is assigning color values to pixels
- Color values can be determined by:
  - Interpolated shading (ex. Gouraud or Phong)
  - Texture mapping
  - Per pixel lighting mathematics
  - Reflections
  - Complex pixel shader programs
- Shading includes Texture Mapping
- A color value can now be procedurally generated...

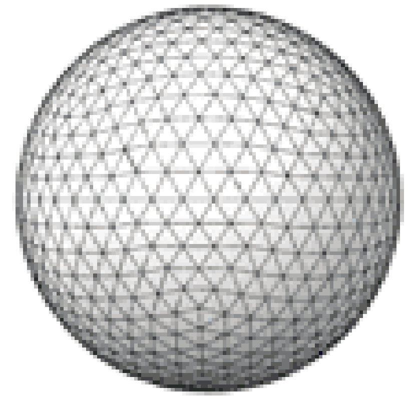
# Gouraud Interpolation

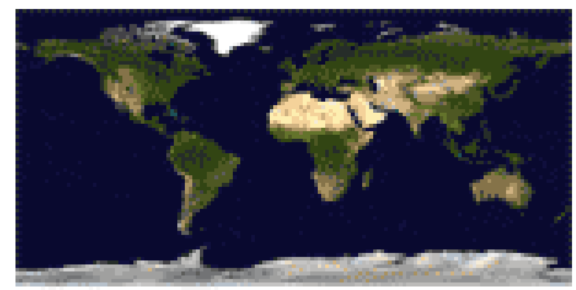
- Also called "smooth shading"
- Linearly vary color values across the triangle interior.
- More realistic than flat shading because the facets in the model are less obvious.

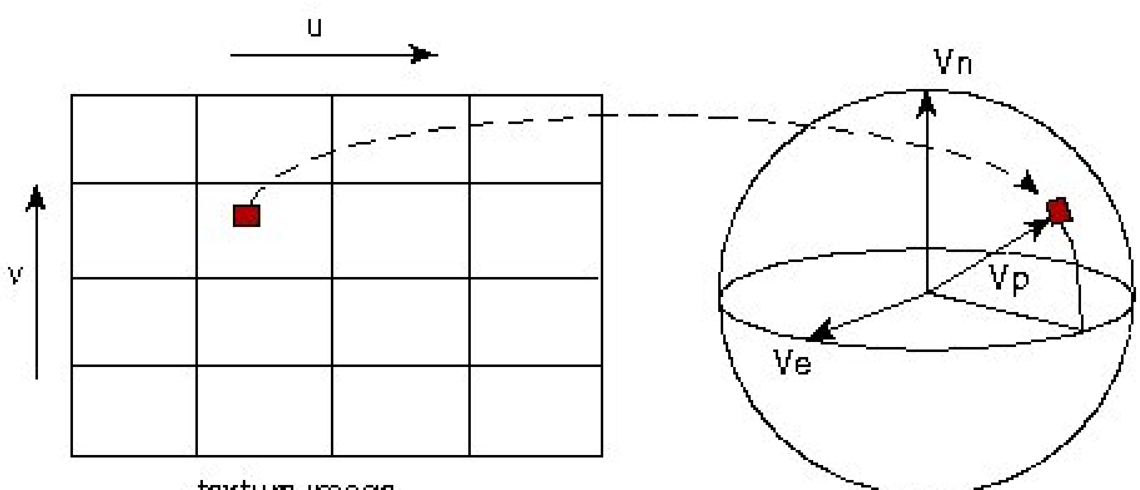


# Texture Mapping

- Associate points in an image to points in a geometric object


Sphere with no texture


Texture image



texture image


Sphere with texture

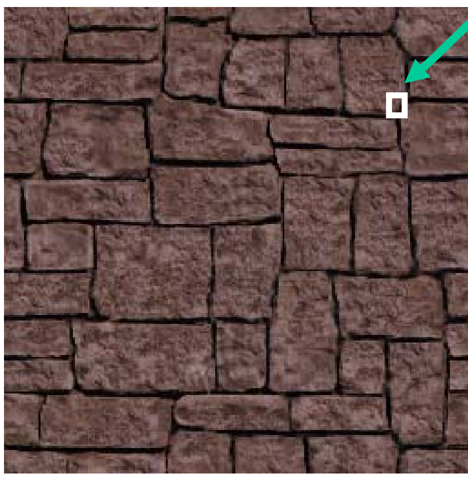
# Mip Mapping

**Mip Mapping** is a technique to manage pixel level of detail (LOD).
Scaled versions of the original texture are generated and stored. These smaller stored textures are used for the texture samples as objects appear smaller with greater distance.

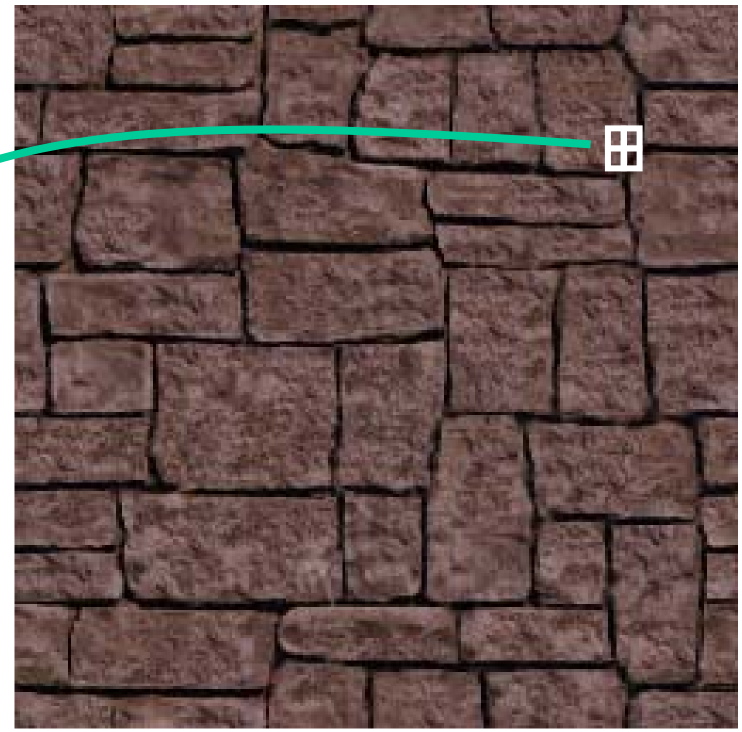
1024x1024 512x512 256x256 128x128 64x64

# Bilinear Filtering

Individual texel colors are interpolated from the four nearest texels of the closest stored mip map.



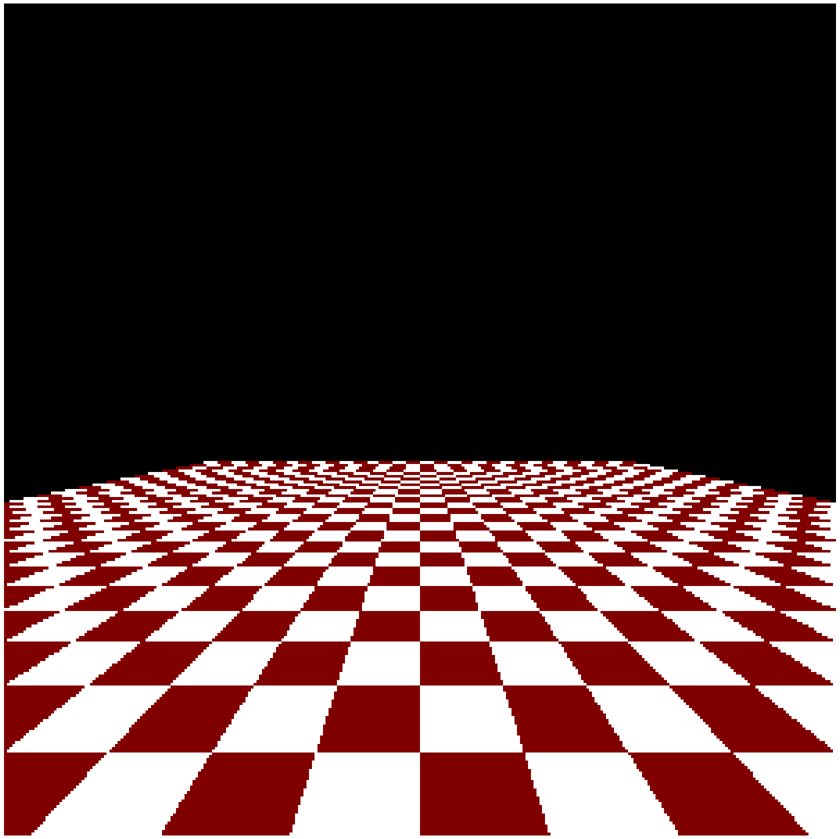
*Random Sized Texture Needed in a Given Frame of an Applicaiton*



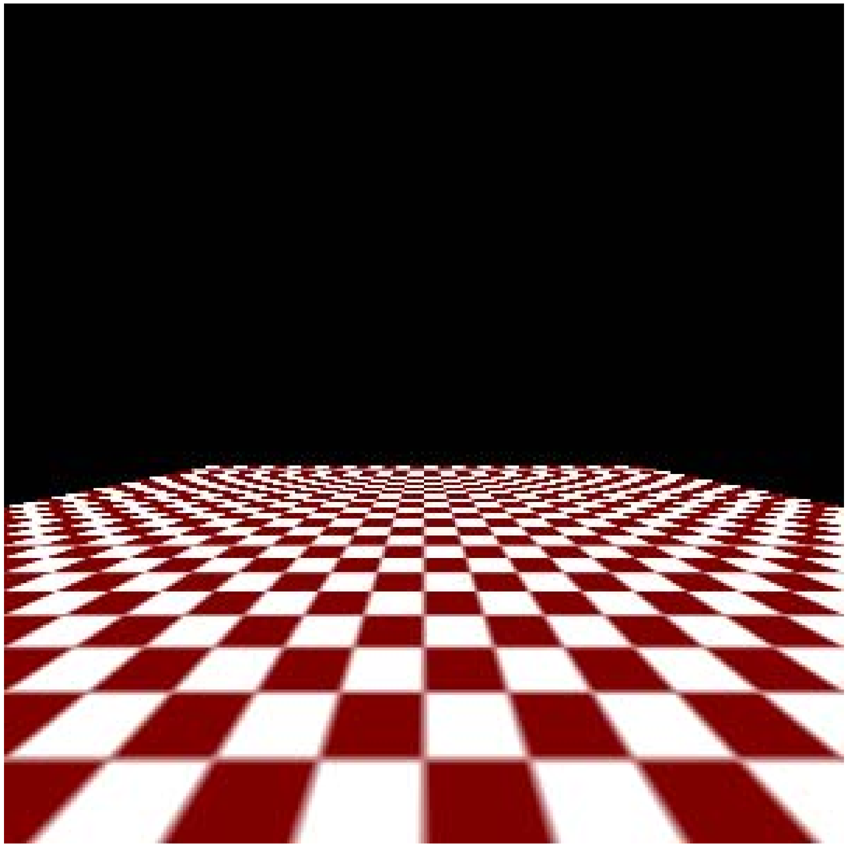
*Stored Mip Map Texture*

# Texture Filtering - Good

Nearest

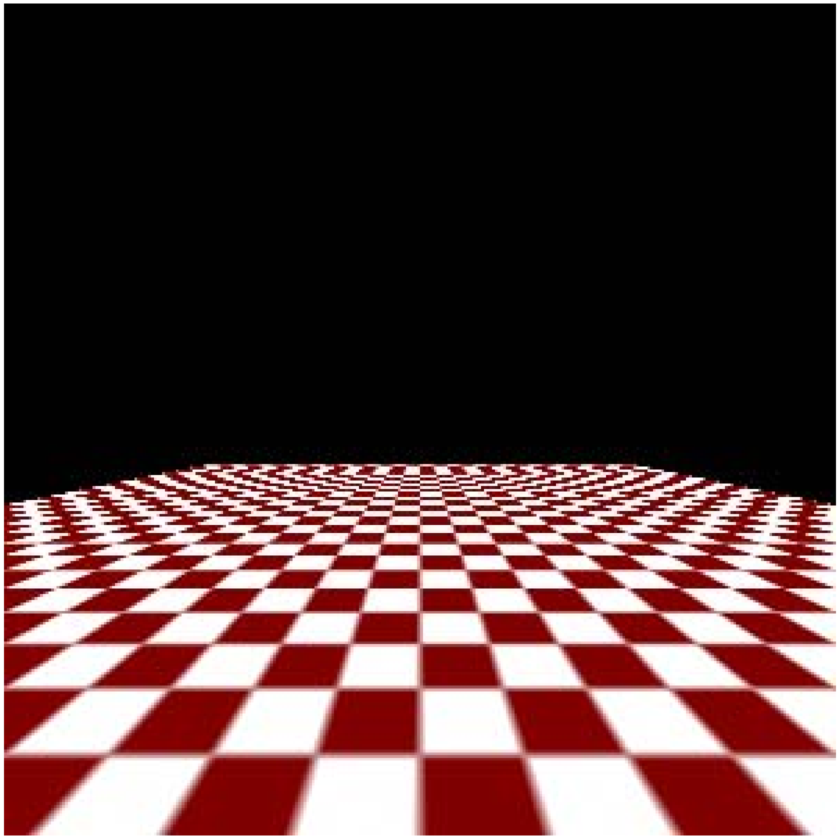


Bilinear

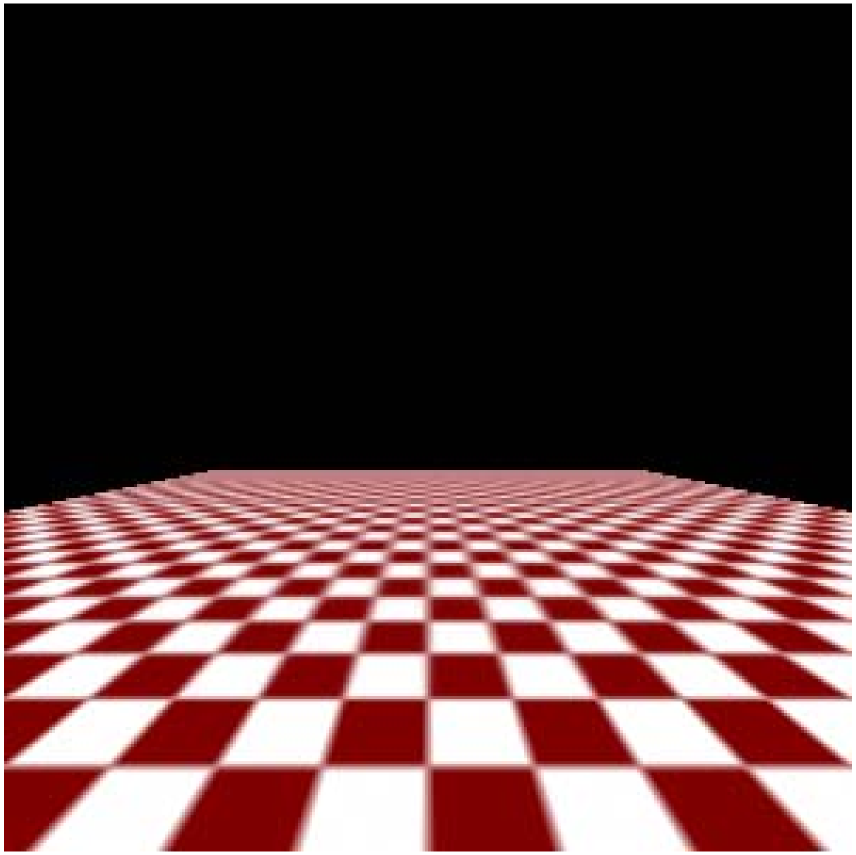


# Texture Filtering - Better

Bilinear

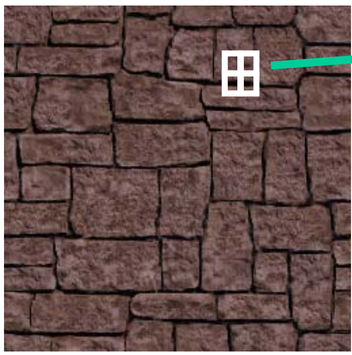


Trilinear

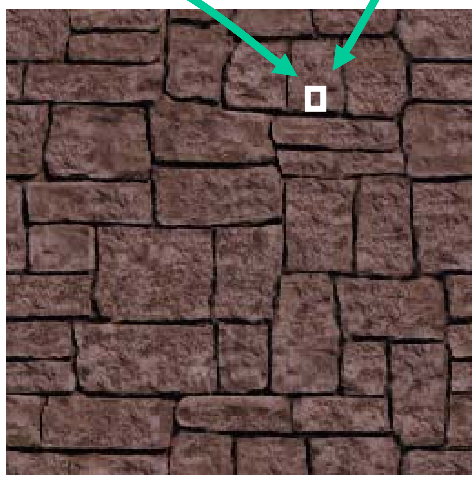


# Trilinear Filtering

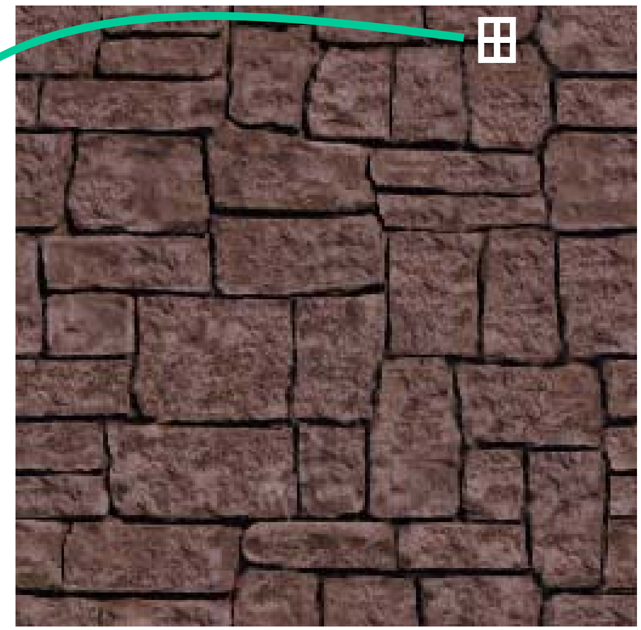
Individual texel colors are interpoloated from bilinear interpolations of nearest adjacent mip maps.



*Stored Mip Map Texture*

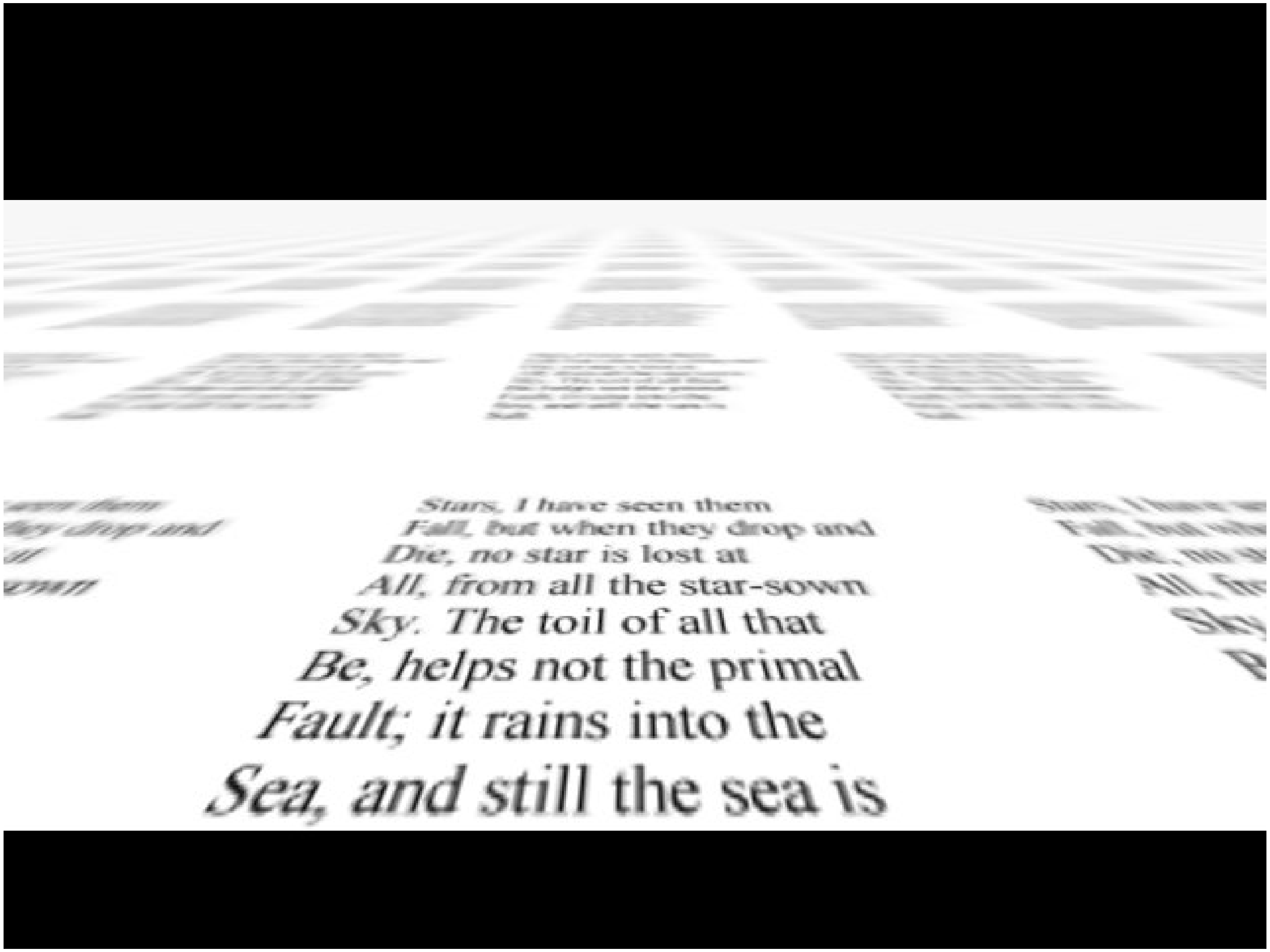


*Random Sized Texture Needed in a Given Frame of an Application*

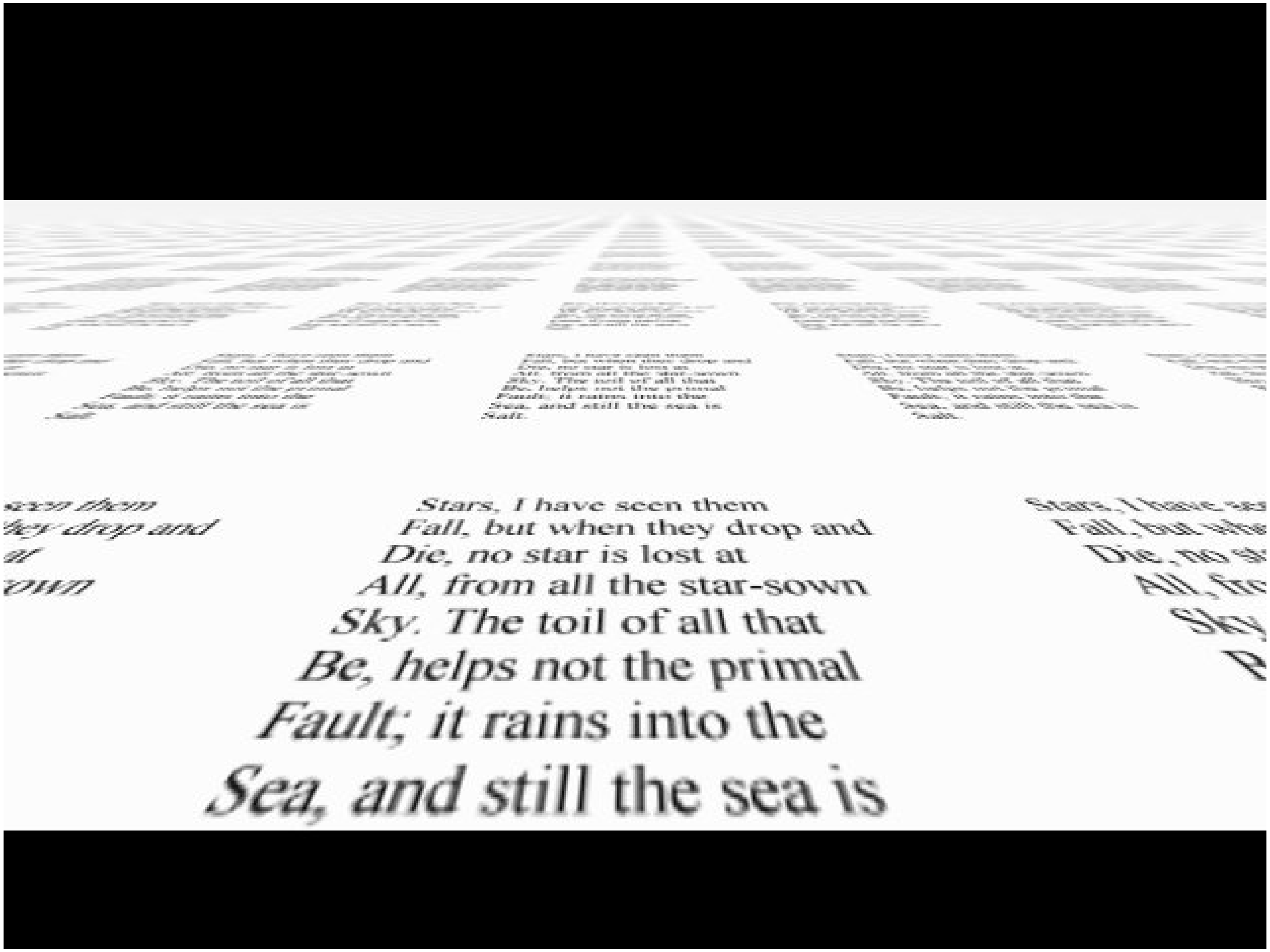


*Stored Mip Map Texture*

# Trilinear Filtering



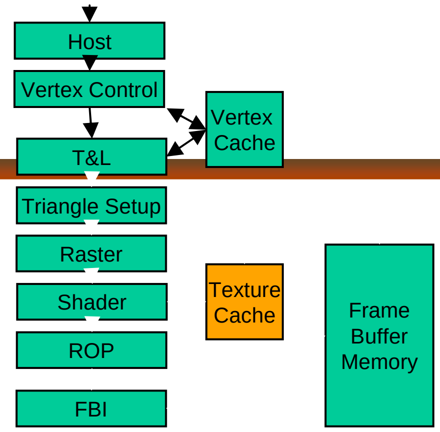
# Anisotropic Filtering



# Filtering techniques

- Point sampling:
  - pixel values are calculated by choosing one texture pixel (texel) color
- Bilinear filtering:
  - interpolating colors from 4 neighboring texels. This gives a smoothing (if somewhat blurry) effect and makes the scene look more natural and prevents abrupt transitions between neighboring texels.
- Trilinear filtering:
  - interpolating bilinearly filtered samples from two mip-maps. Trilinear mip-mapping prevents moving objects from displaying a distracting “sparkle” caused by abrupt transitions between mipmaps.
- Anisotropic filtering:
  - interpolating and filtering multiple samples from one or more mip-maps to better approximate very distorted textures. Gives a sharper effect when severe perspective correction is used. Trilinear mipmapping blurs textures more.

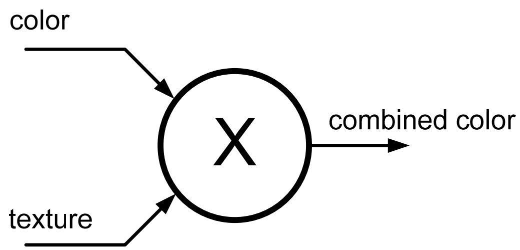
# Texture Cache




- Stores temporally local texel values to reduce bandwidth requirements
- Due to nature of texture filtering high degrees of efficiency are possible
- Efficient texture caches can achieve 75% or better hit rates
- Reduces texture (memory) bandwidth by a factor of four for bilinear filtering

# Pixel Shading

- 1999 (DirectX 7)
  - Application could select from a few simple combinations of texture and interpolated color
    - Add
    - Decal
    - Modulate




- Next (DirectX 9)
  - Write a general program that executes for every pixel with a nearly unlimited number of interpolated inputs, texture lookups and math operations
  - Can afford to perform sophisticated lighting calculations at every pixel

```
# clock 3
rcp r0.a, r0.a                # reciprocal in shader 0
mul r0.rg r0, r0.a            # div instruction in shader 0
mul r0.a, r0.a, r1.a          # dual issue in shader 0
texld r2, r0, s1              # texture fetch
mad r2.rgb, r0.a, r2, c5      # mad in shader 1
abs r0.a, r0.a                # abs in shader 1
log r0.a, r0.a                # log in shader 1

# clock 4
rcp r0.a, t1.a                # reciprocal in shader 0
mul r0.rg, t1, r0.a           # div instruction in shader 0
mul r0.a, r0.a, c2.g          # dual issue in shader 0
texld r1, r0, s3              # tex fetch
mad r1.rgb, r1, c4, -r2       # mad in shader 1
exp r0.a, r0.a                # dual issue in shader 1

# clock 5
texld r0, r1.bar, s2          # texture coordinates swizzle
mad r0.rgb, r0, v0, r1        # color calculation in shader 1
mul r0.a, r1, v0              # dual issue in shader 1

# clock 6
mul r1.rgb, r0.a, c5.a        # mul in shader 0
mad r0.rgb, r1, r0.a, r0      # mad in shader 1
mov r0.a, c3.a                # move in shader 1
mov oC0, r0                   # move in shader 1
```

# GeForce FX Fragment/Pixel Program Examples