I3D

Programming Graphics Hardware

Randy Fernando, Cyril Zeller

NVIDIA®

Overview of the Tutorial

10:45	**Introduction to the Hardware Graphics Pipeline** Cyril Zeller
12:00	Lunch
14:00	**High-Level Shading Languages** Randy Fernando
15:15	break
15:45	**GPU Applications** Cyril Zeller / Randy Fernando
17:00	End

Programming Graphics Hardware

Introduction to the Hardware Graphics Pipeline

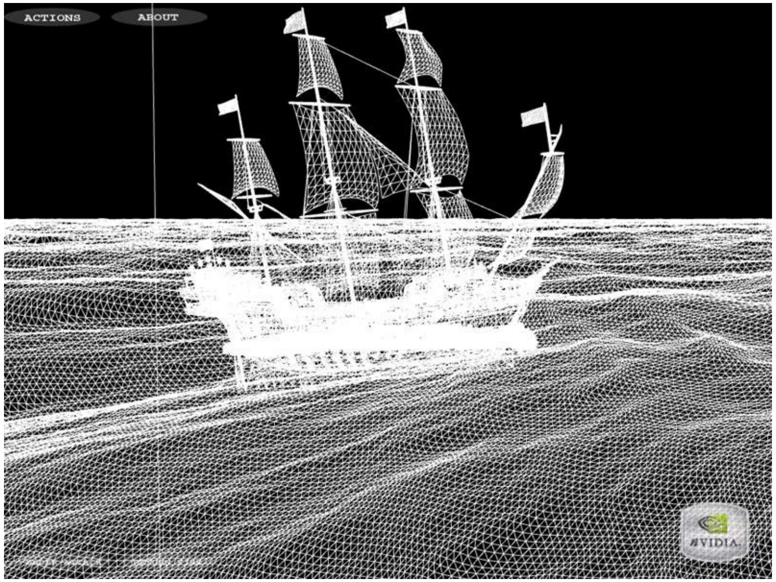
Cyril Zeller

Overview

- Concepts:
  - Real-time rendering
  - Hardware graphics pipeline
- Evolution of the PC hardware graphics pipeline:
  - 1995: Texture mapping and z-buffer
  - 1998: Multitexturing
  - 1999: Transform and lighting
  - 2001: Programmable vertex shader
  - 2002: Programmable pixel shader
  - 2004: Shader model 3.0 and 64-bit color support
- PC graphics software architecture
- Optimization

Real-Time Rendering

- Graphics hardware enables real-time rendering
- Real-time means display rate at more than 10 images per second



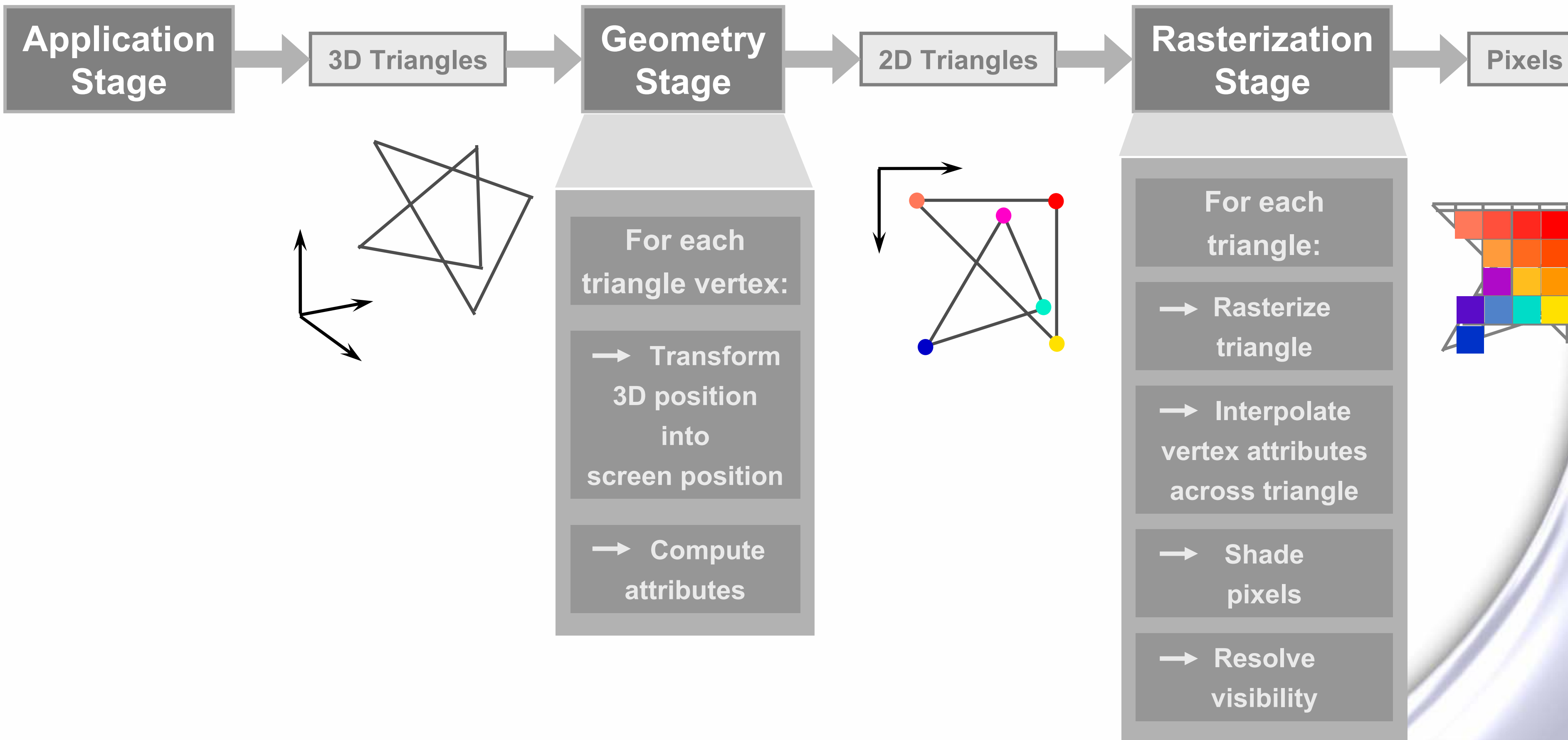
3D Scene =

Collection of
3D primitives (triangles, lines, points)

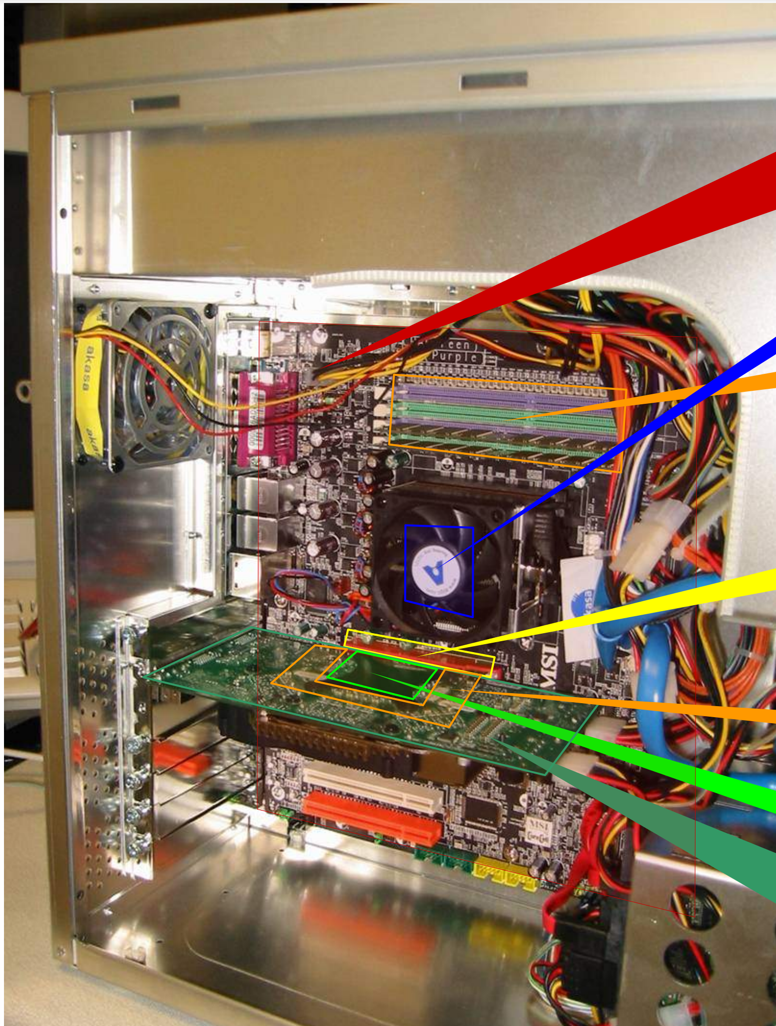
Image =

Array of pixels

Hardware Graphics Pipeline



PC Architecture



Motherboard

Central Processor Unit (CPU)

System Memory

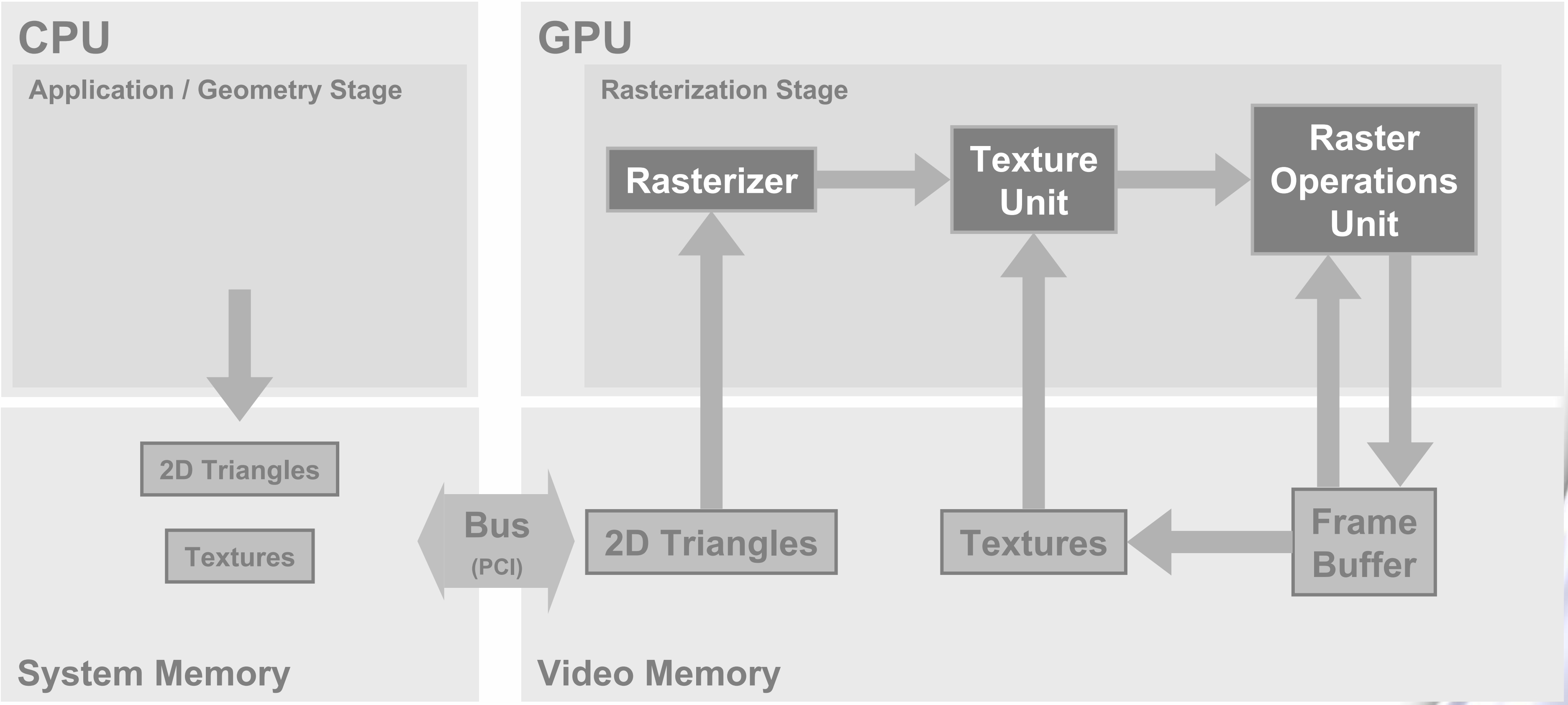
Bus Port (PCI, AGP, PCIe)

Video Memory

Graphics Processor Unit (GPU)

Video Board

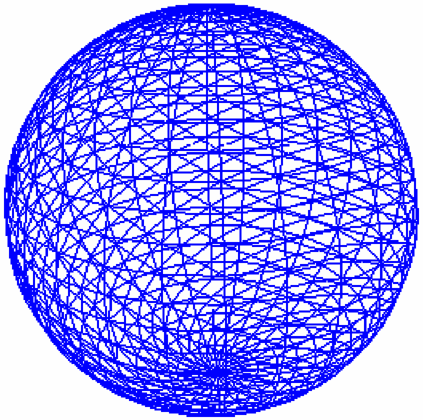
1995: Texture Mapping and Z-Buffer



- **PCI: Peripheral Component Interconnect**
- **3dfx's Voodoo**

Texture Mapping

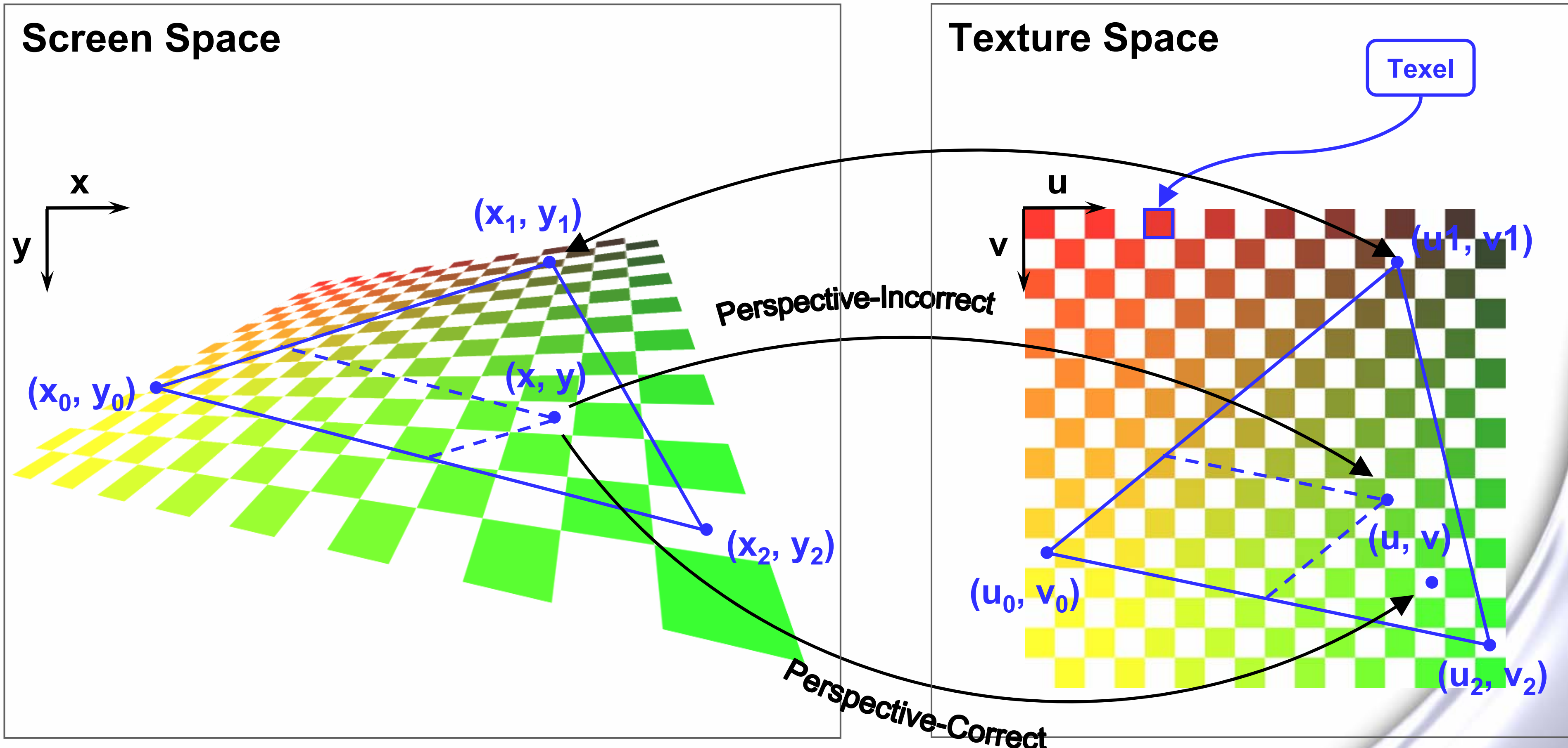
Triangle Mesh **textured with** **Base Texture**



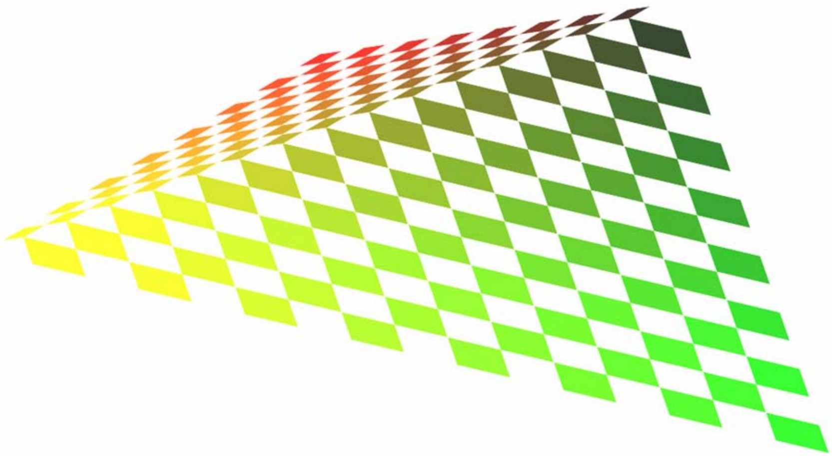
+

=

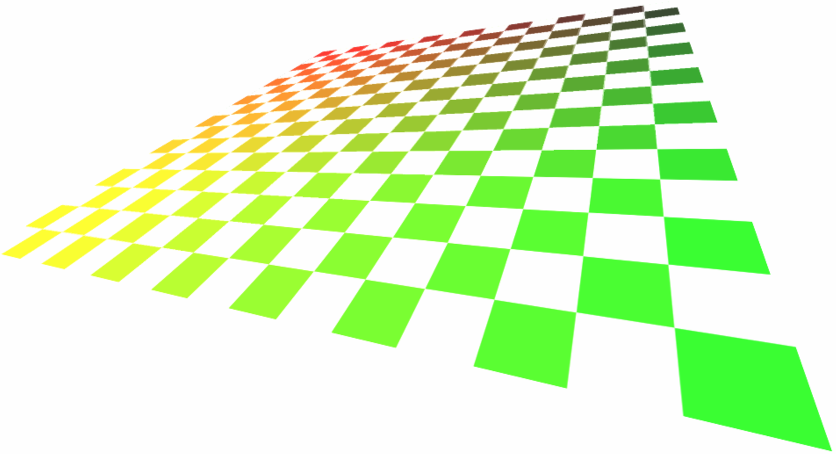
Texture Mapping: Texture Coordinates Interpolation



Texture Mapping: Perspective-Correct Interpolation

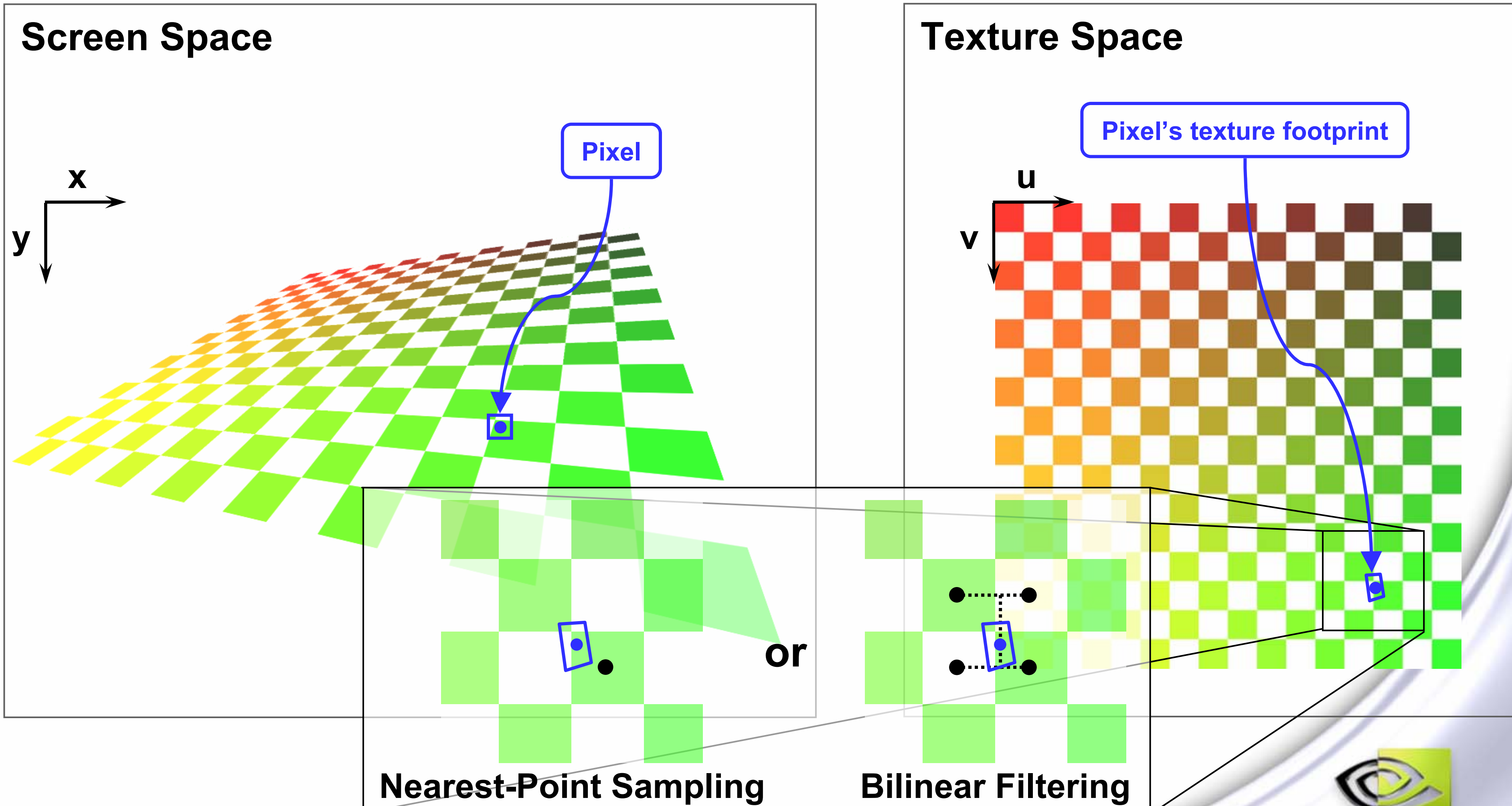


Perspective-Incorrect

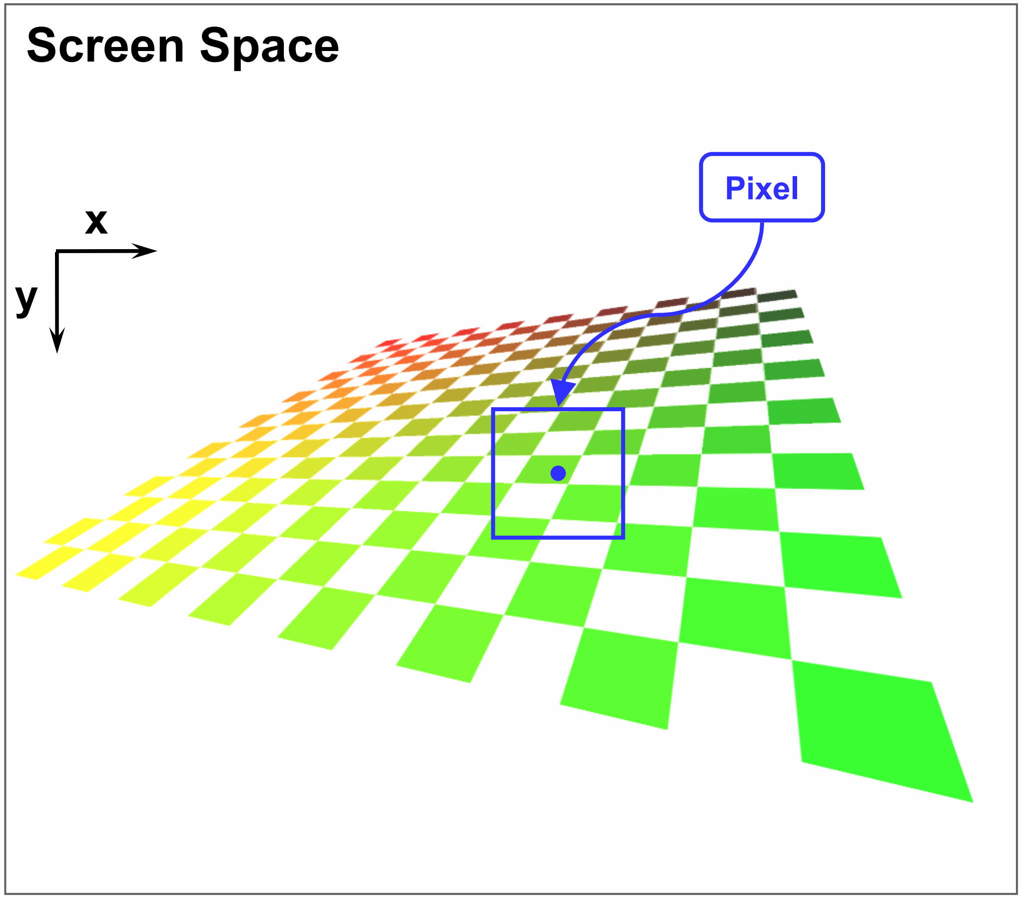


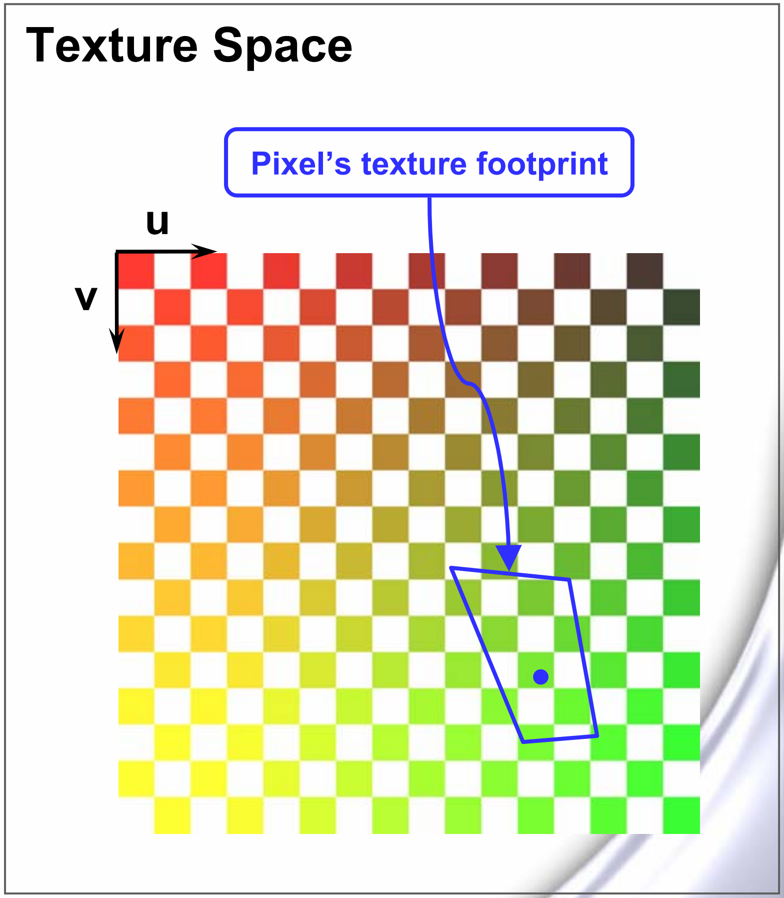
Perspective-Correct

Texture Mapping: Magnification

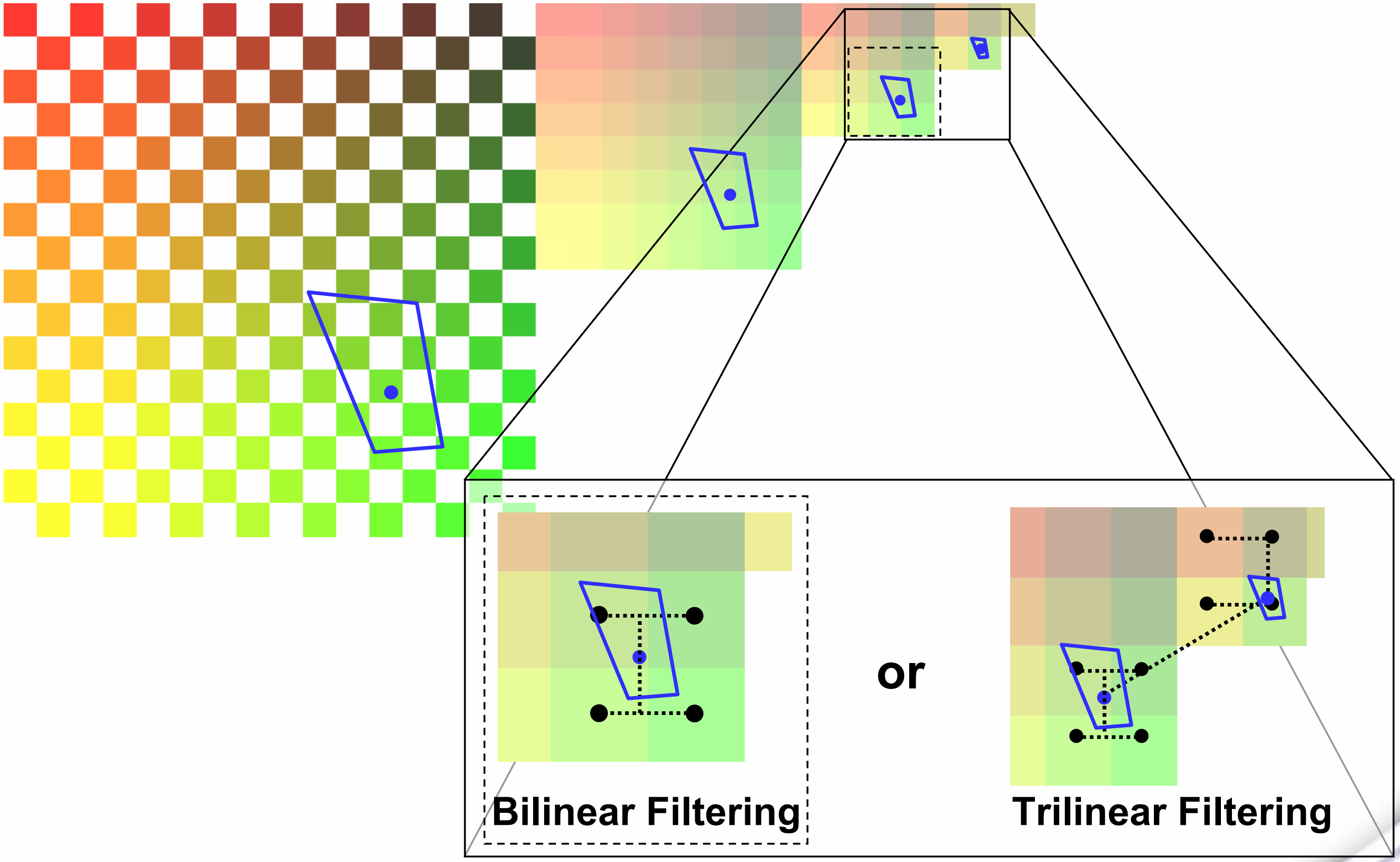


Texture Mapping: Minification

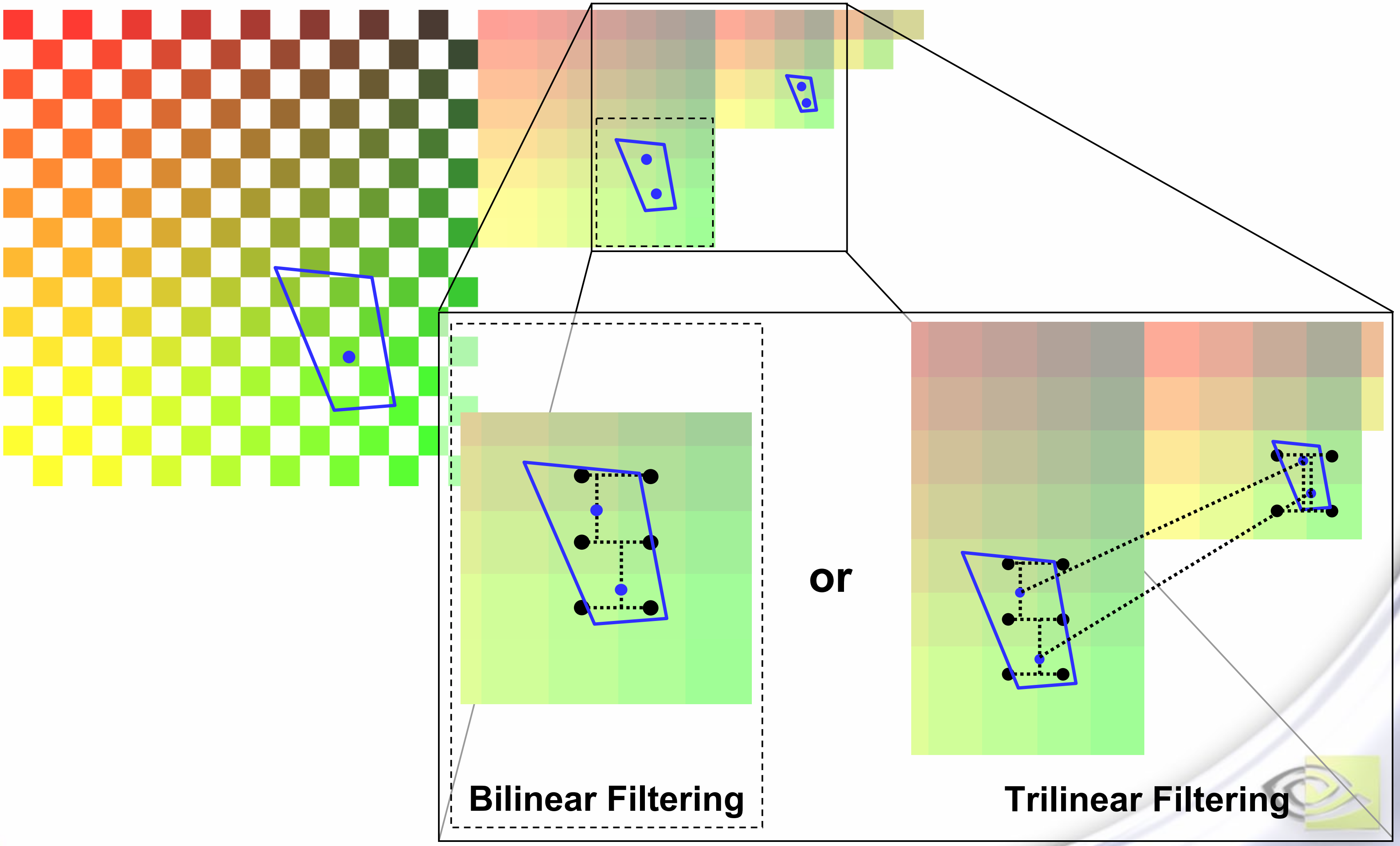




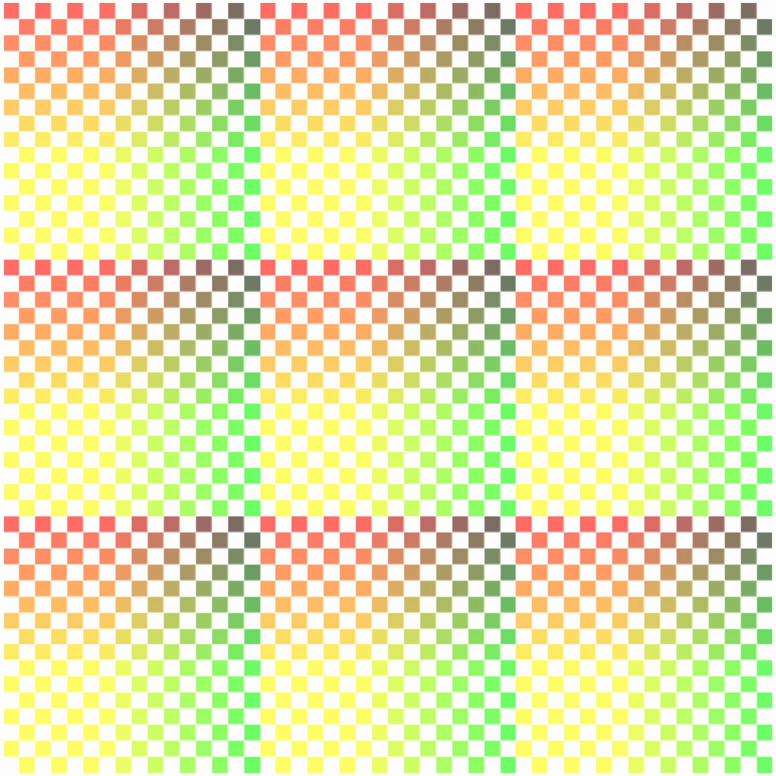
Texture Mapping: Mipmapping



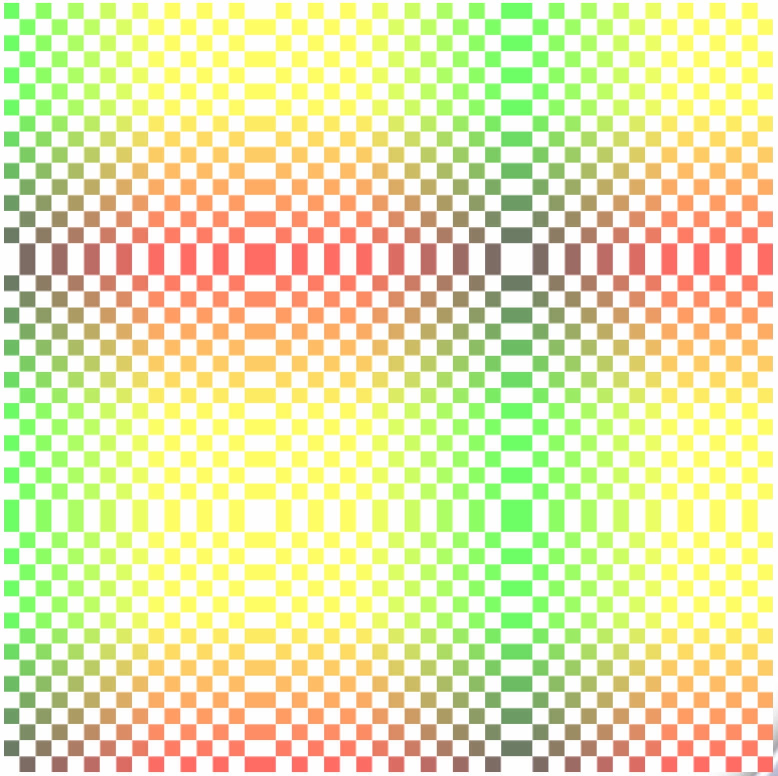
Texture Mapping: Anisotropic Filtering



Texture Mapping: Addressing Modes

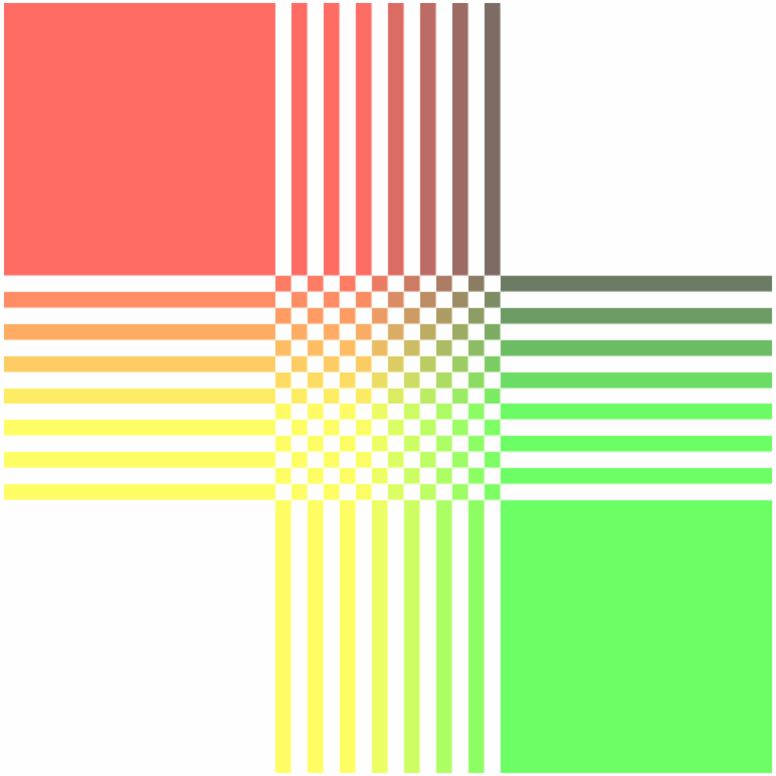


Wrap

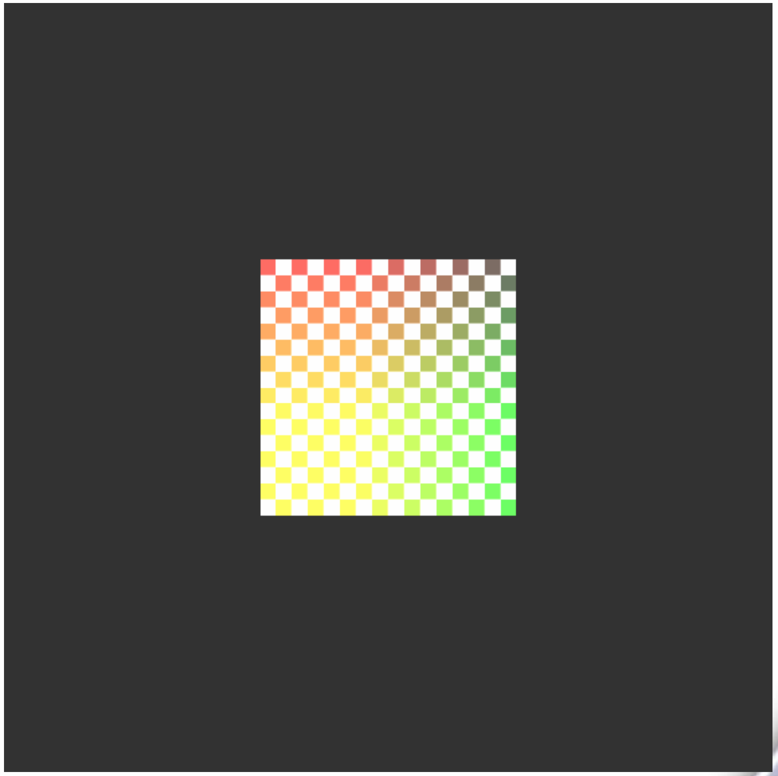


Mirror

Texture Mapping: Addressing Modes

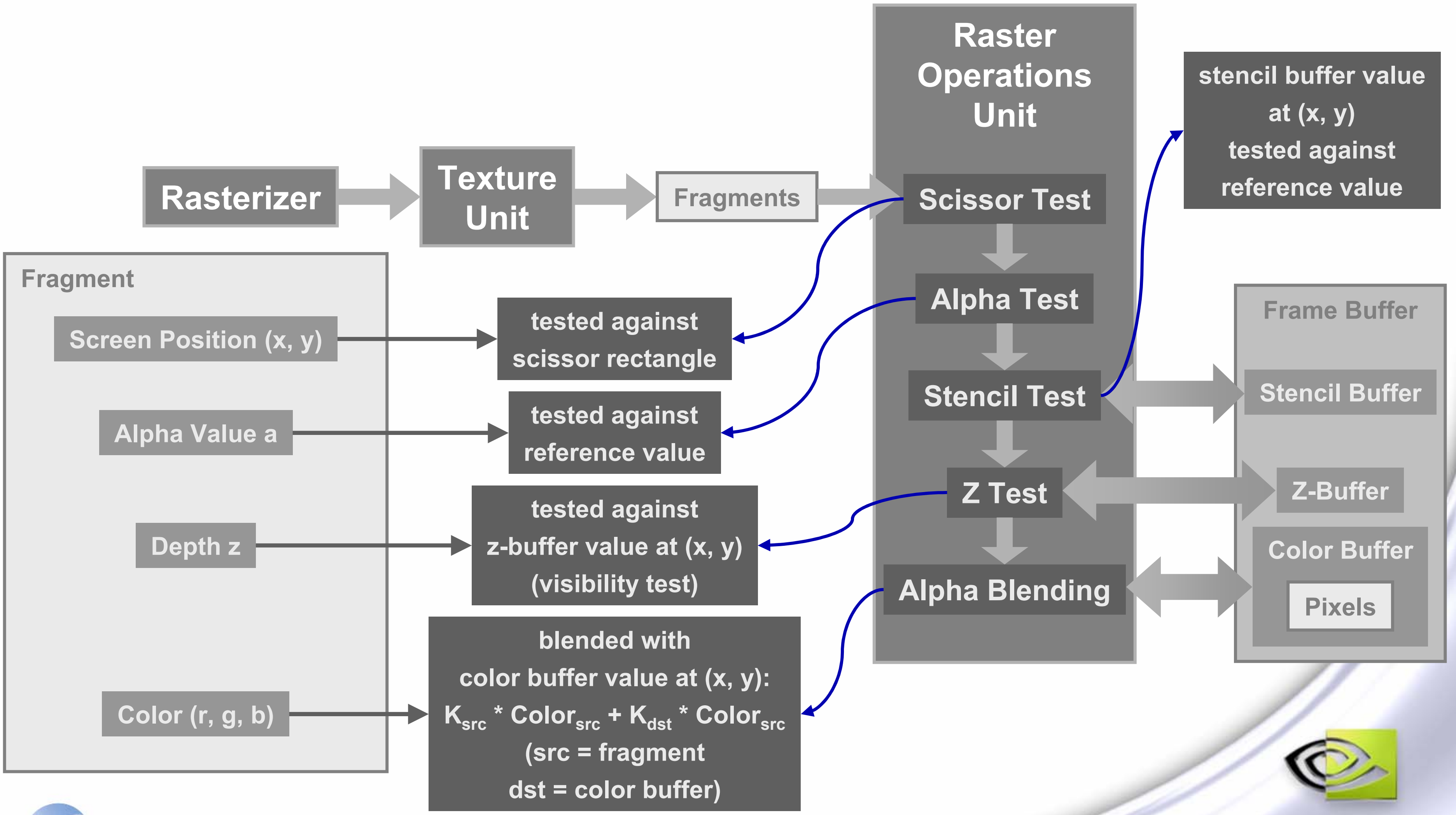


Clamp

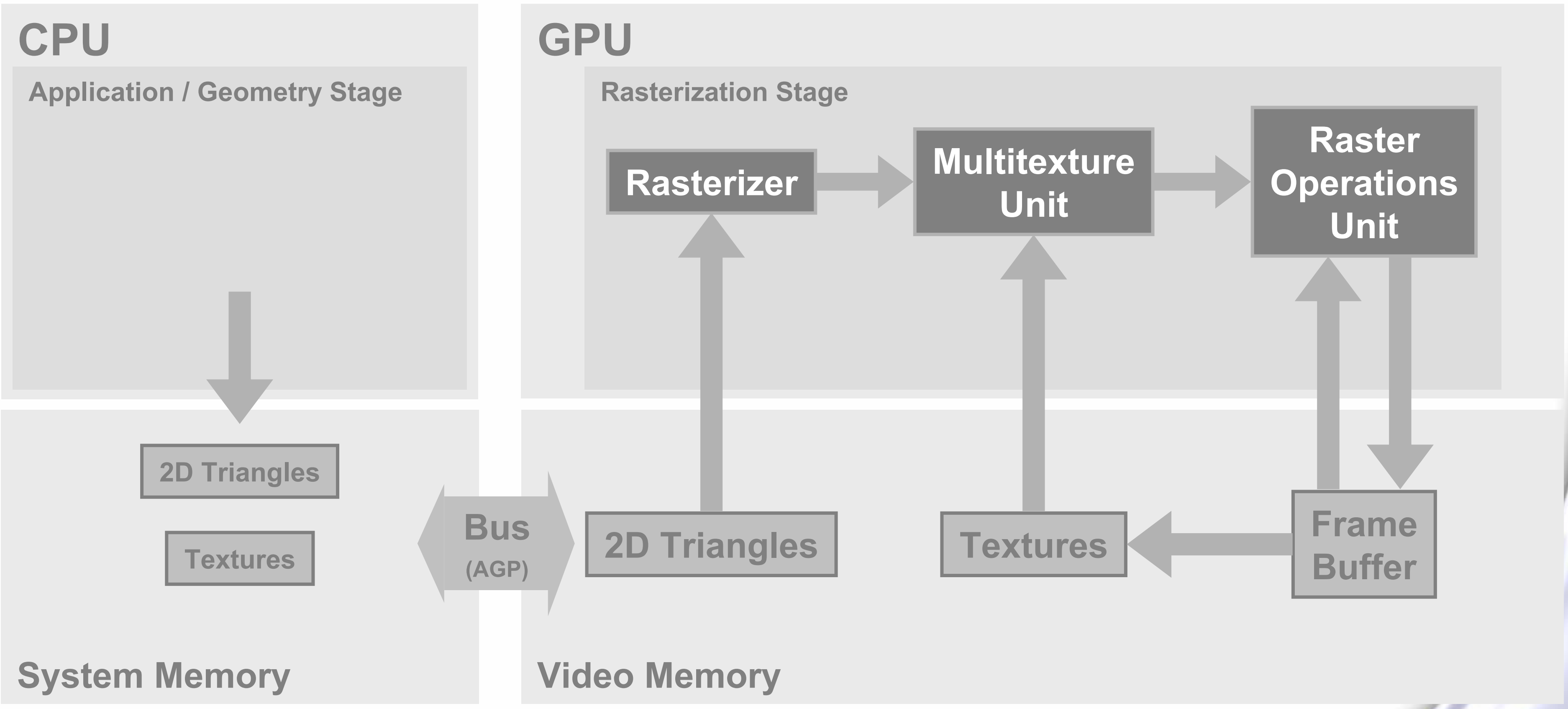


Border

Raster Operations Unit (ROP)



1998: Multitexturing



- **AGP: Accelerated Graphics Port**
- **NVIDIA's TNT, ATI's Rage**

AGP

- PCI uses a parallel connection
- AGP uses a serial connection

→ Fewer pins, simpler protocol → Cheaper, more scalable

- PCI uses a shared-bus protocol
- AGP uses a point-to-point protocol

→ Bandwidth is not shared among devices

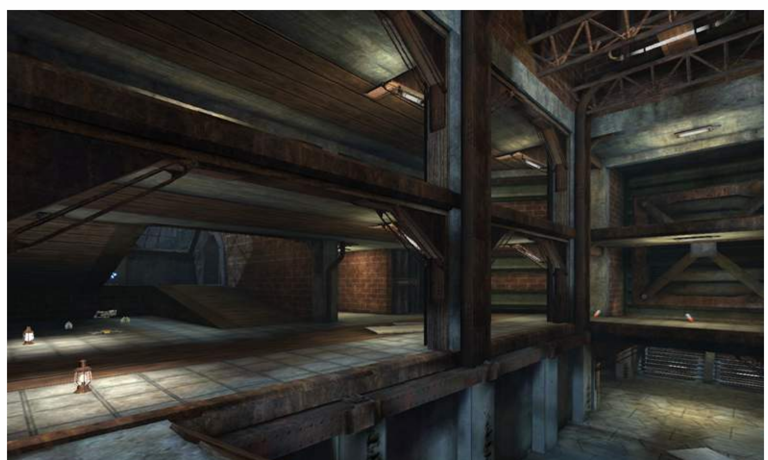
- AGP uses a dedicated system memory called AGP memory or non-local video memory
  - The GPU can lookup textures that resides in AGP memory
  - Its size is called the AGP aperture
- Bandwidth: AGP = 2 x PCI (AGP2x = 2 x AGP, etc.)

Multitexturing

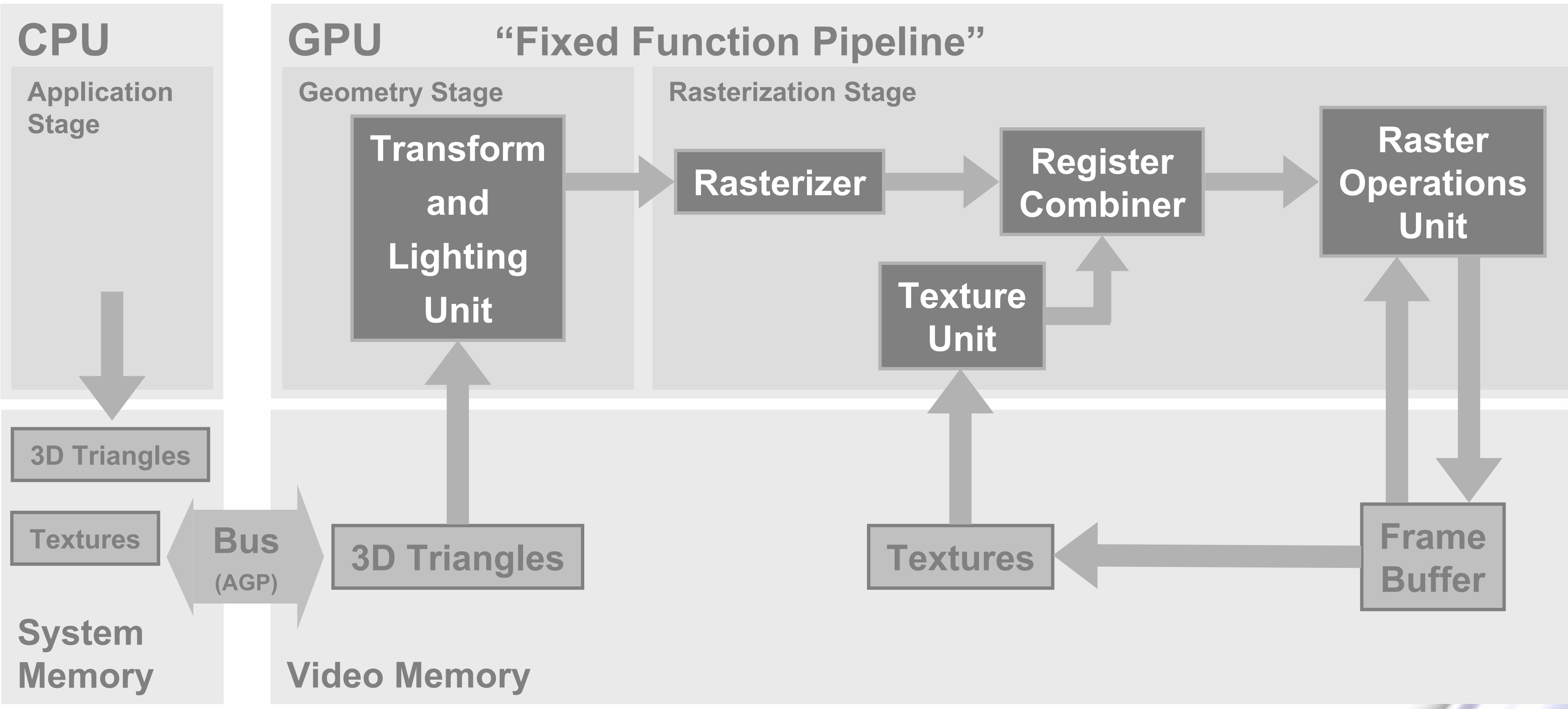
Base Texture **modulated by** **Light Map**

X

=

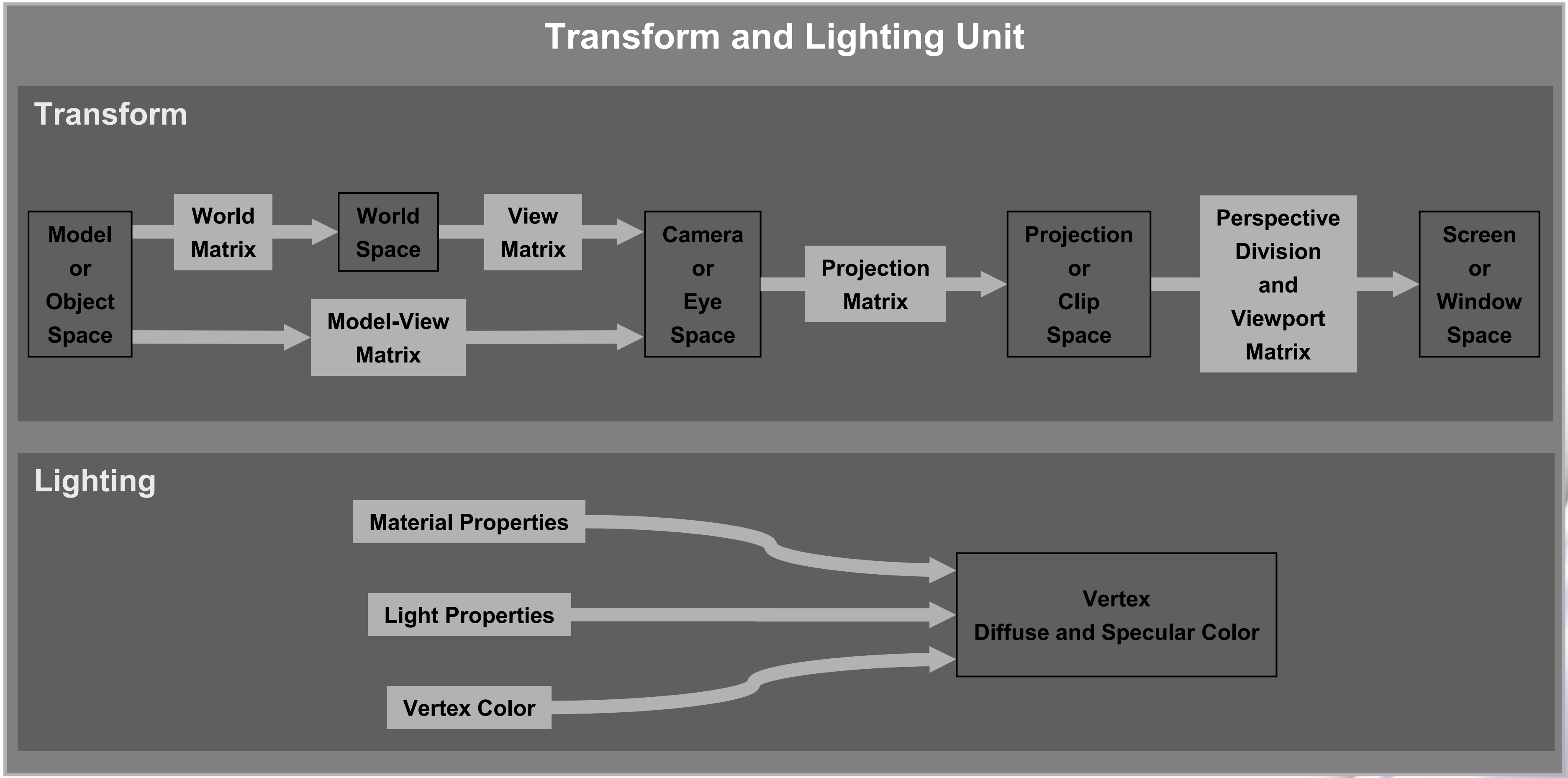


1999: Transform and Lighting



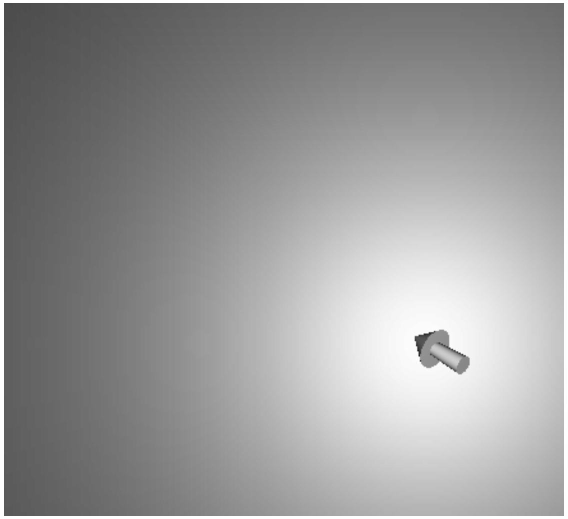
- Register Combiner: Offers many more texture/color combinations
- NVIDIA's GeForce 256 and GeForce2, ATI's Radeon 7500, S3's Savage3D

Transform and Lighting Unit (TnL)

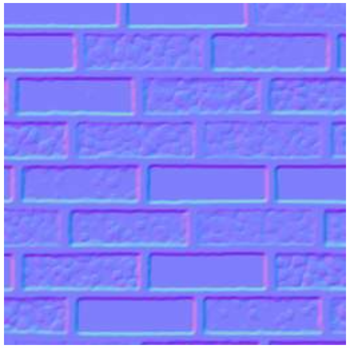


Bump Mapping

- Bump mapping is about fetching the normal from a texture (called a normal map) instead of using the interpolated normal to compute lighting at a given pixel



+



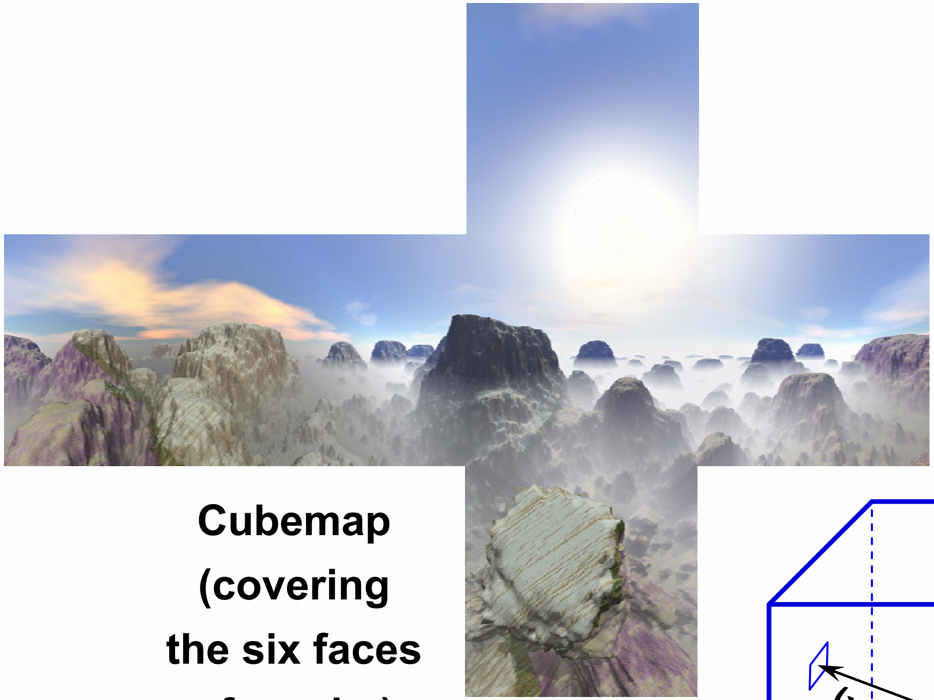
Normal Map

=

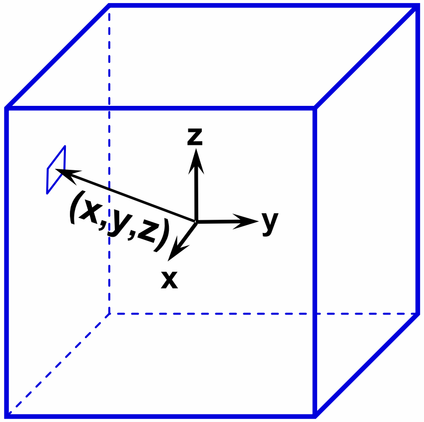
Diffuse light without bump

Diffuse light with bumps

Cube Texture Mapping



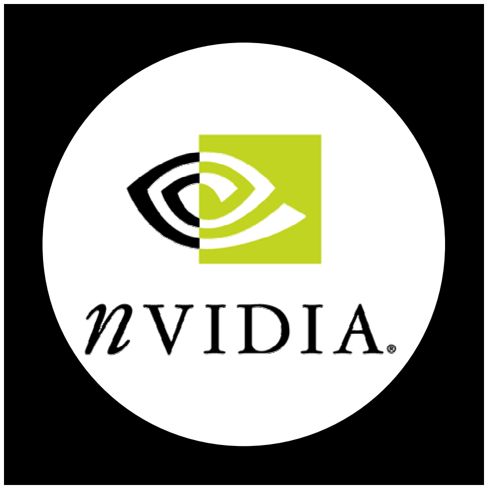
Cubemap (covering the six faces of a cube)



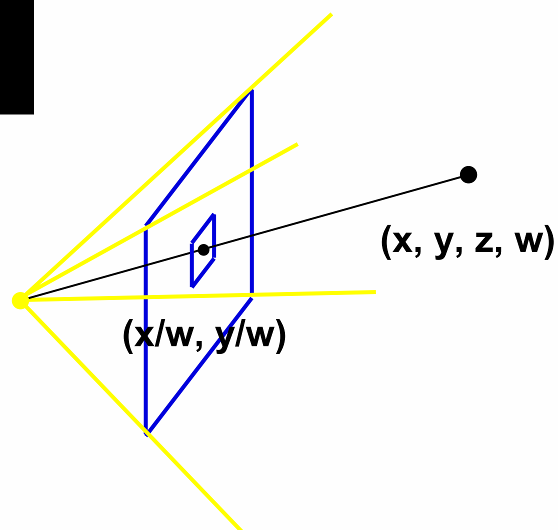
Cubemap lookup (with direction (x, y, z))

Environment Mapping (the reflection vector is used to lookup the cubemap)

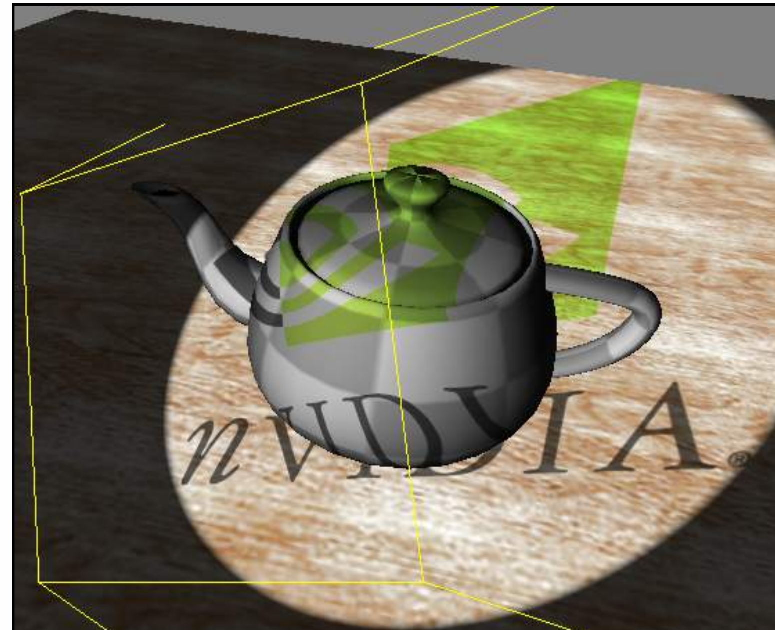
Projective Texture Mapping



Projected Texture

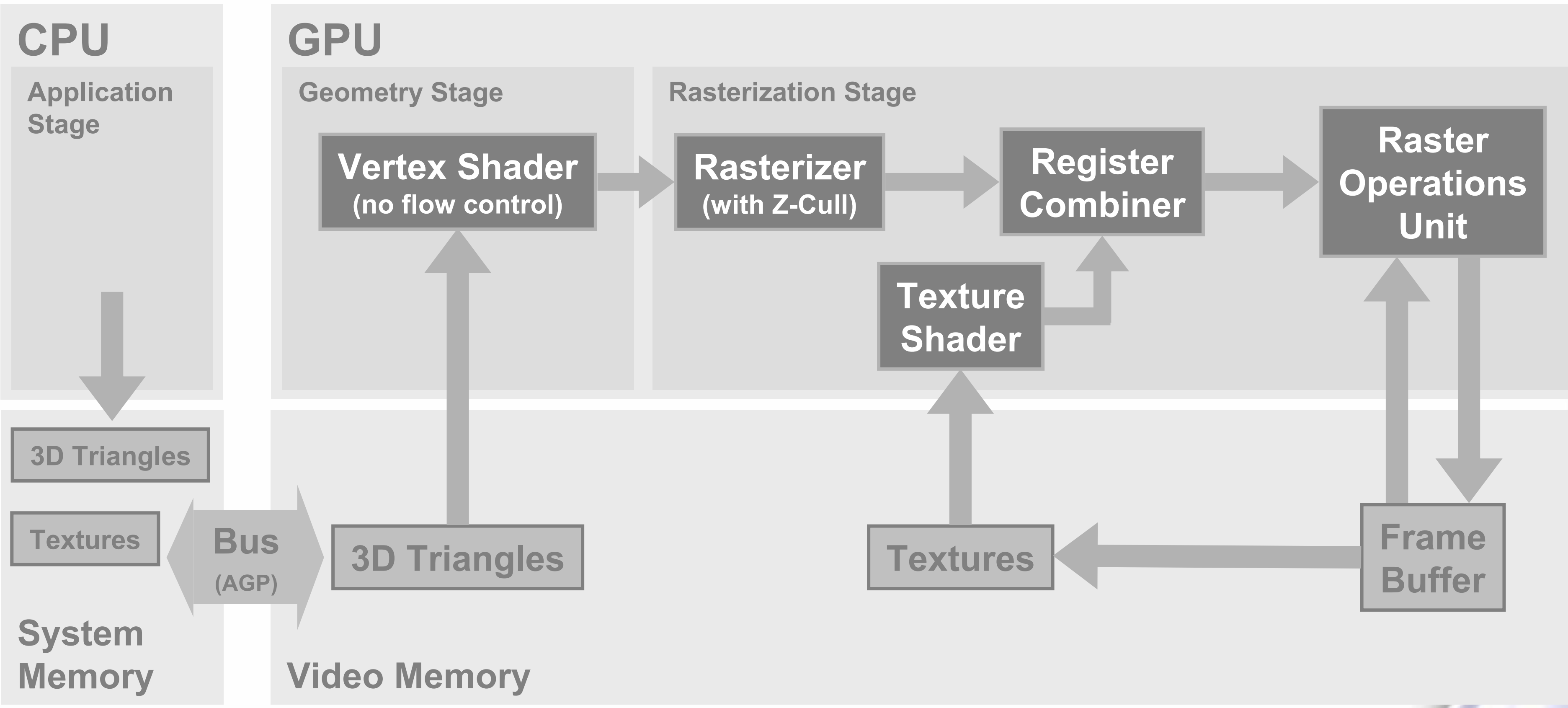


Projective Texture lookup



Texture Projection

2001: Programmable Vertex Shader



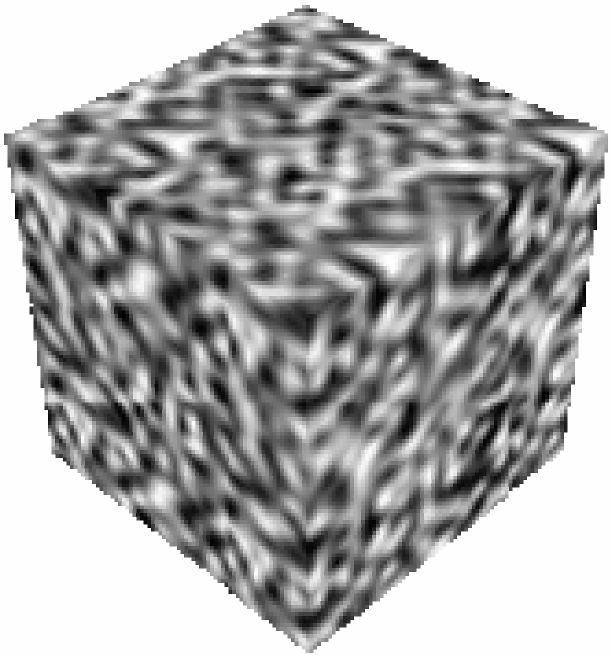
- **Z-Cull**: Predicts which fragments will fail the Z test and discards them
- **Texture Shader**: Offers more texture addressing and operations
- **NVIDIA's GeForce3 and GeForce4 Ti, ATI's Radeon 8500**

Vertex Shader

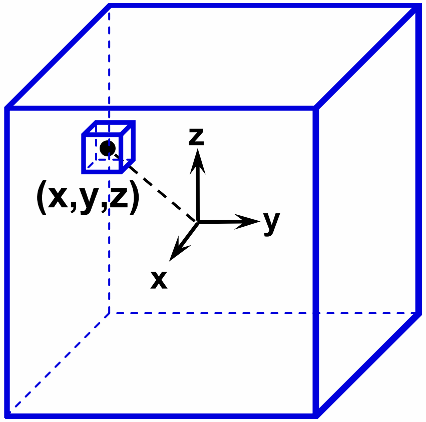
- **A programmable processor for per-vertex computation**

```
void VertexShader(
          // Input per vertex
       in float4 positionInModelSpace,
       in float2 textureCoordinates,
       in float3 normal,

          // Input per batch of triangles
  uniform float4x4 modelToProjection,
  uniform float3 lightDirection,

          // Output per vertex
      out float4 positionInProjectionSpace,
      out float2 textureCoordinatesOutput,
      out float3 color
)
{
  // Vertex transformation
  positionInProjectionSpace = mul(modelToProjection, positionInModelSpace);

  // Texture coordinates copy
  textureCoordinatesOutput = textureCoordinates;

  // Vertex color computation
  color = dot(lightDirection, normal);
}
```

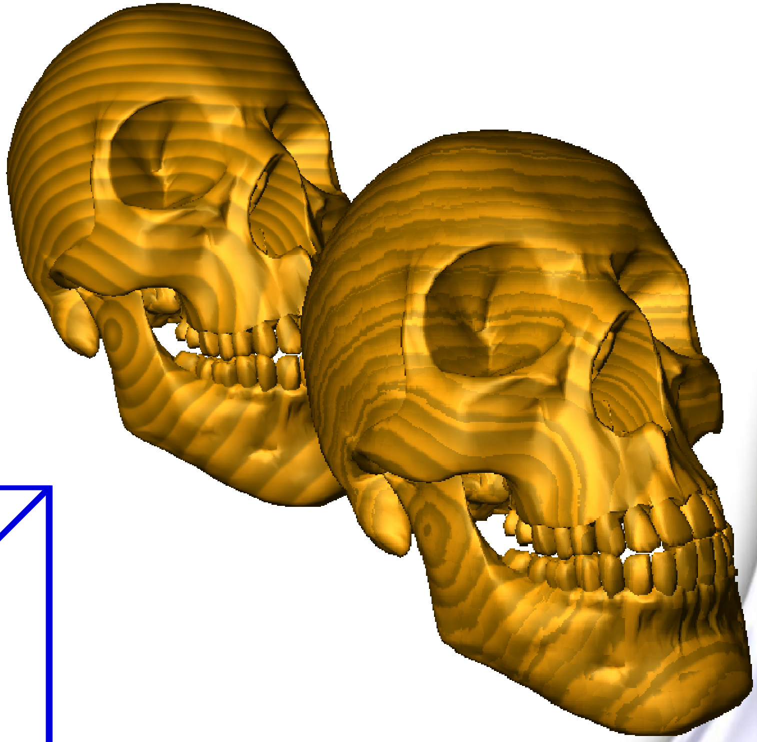
Volume Texture Mapping



Volume Texture
(3D Noise)



Volume Texture lookup
(with position (x, y, z))

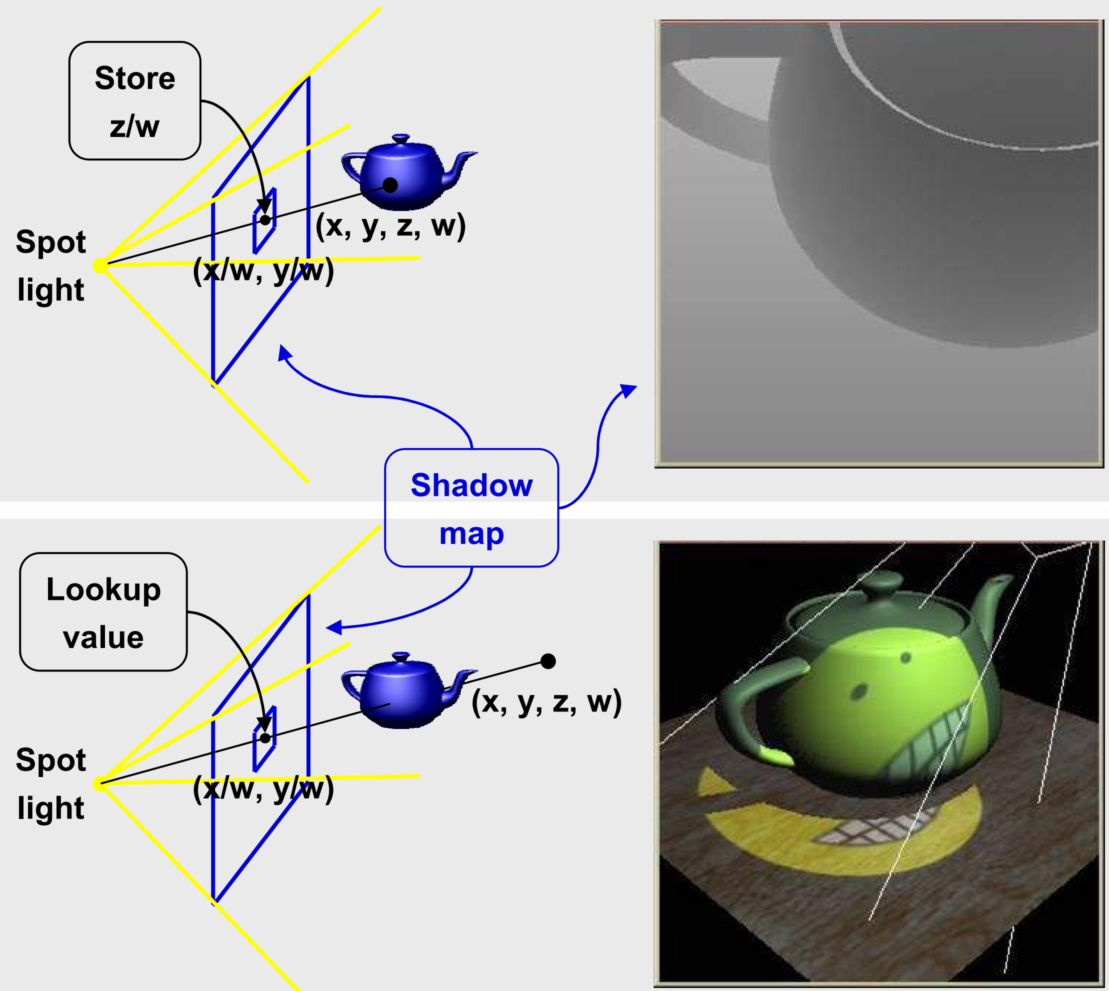


Noise Perturbation

Hardware Shadow Mapping

Shadow Map Computation

The shadow map contains
the depth z/w of the 3D points visible
from the light's point of view



Shadow Rendering

A 3D point (x, y, z, w) is in shadow if:
z/w < value of shadow map at (x/w, y/w)
A hardware shadow map lookup
returns the value of this comparison
between 0 and 1

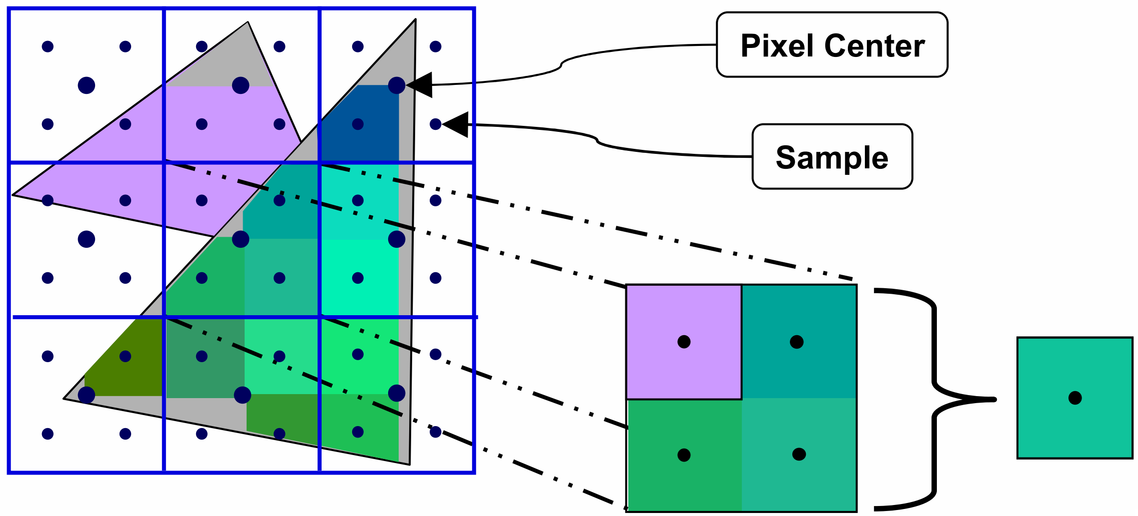
Antialiasing: Definition

- Aliasing: Undesirable visual artifacts due to insufficient sampling of:
  - Primitives (triangles, lines, etc.) → jagged edges
  - Textures or shaders → pixelation, moiré patterns

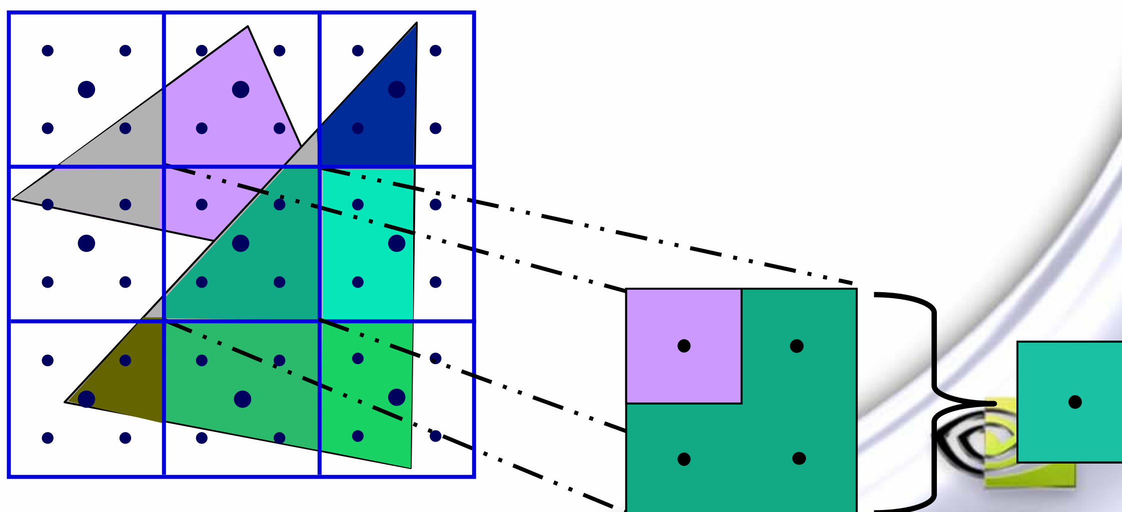
  Those artifacts are even more noticeable on animated images
- Antialiasing: Method to reduce aliasing
  - Texture antialiasing is largely handled by proper mipmapping and anisotropic filtering
  - Shader antialiasing can be tricky (especially with conditionals)

Antialiasing: Supersampling and Multisampling

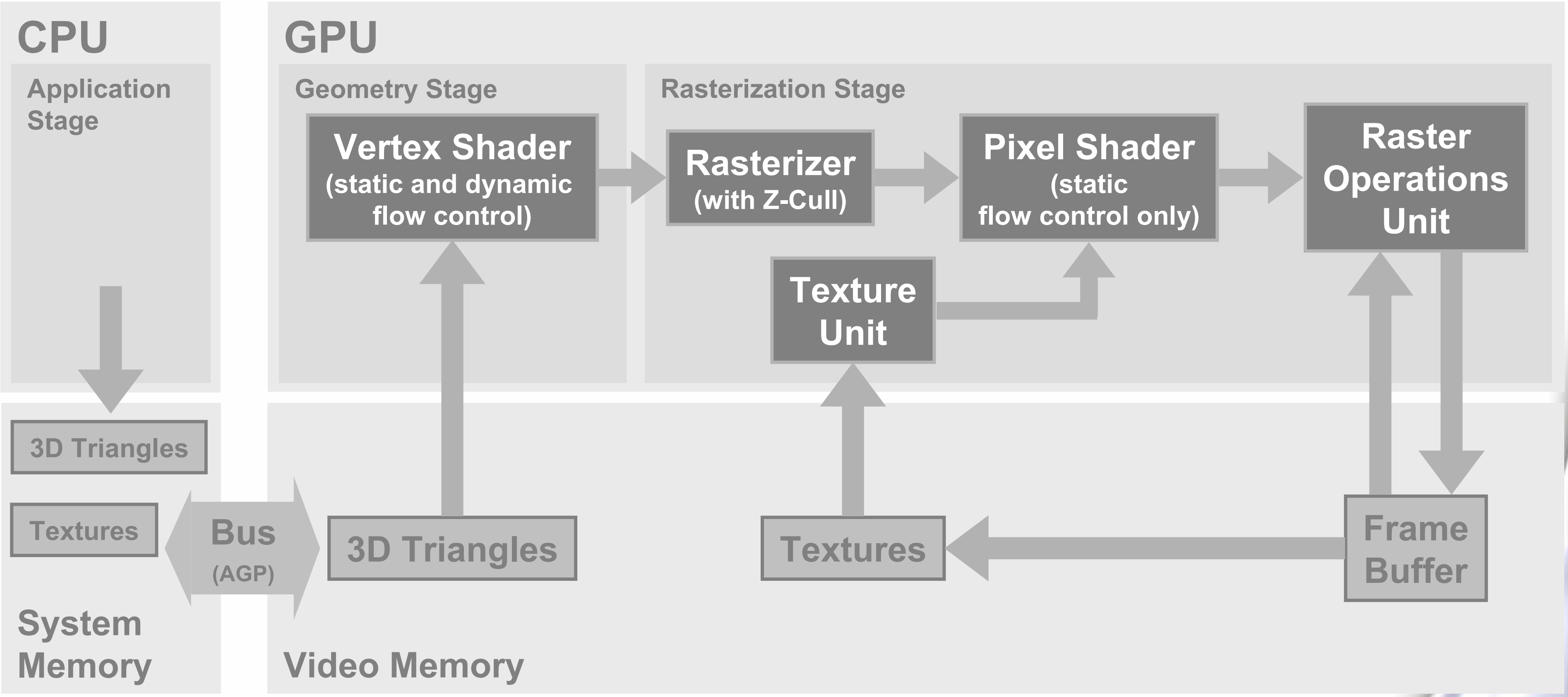
- **Supersampling**:
  Compute color and Z at higher resolution and display averaged color to smooth out the visual artifacts



- **Multisampling**:
  Same thing except only Z is computed at higher resolution
  - As a result, multisampling performs antialiasing on primitive edges only



2002: Programmable Pixel Shader



- **MRT**: Multiple Render Target
- NVIDIA's GeForce FX, ATI's Radeon 9600 to 9800 and X600 to X800

Pixel Shader

- **A programmable processor for per-pixel computation**

```
void PixelShader(
          // Input per pixel
       in float2 textureCoordinates,
       in float3 normal,

          // Input per batch of triangles
  uniform sampler2D baseTexture,
  uniform float3 lightDirection,

          // Output per pixel
      out float3 color
)
{
  // Texture lookup
  float3 baseColor = tex2D(baseTexture, textureCoordinates);

  // Light computation
  float light = dot(lightDirection, normal);

  // Pixel color computation
  color = baseColor * light;
}
```

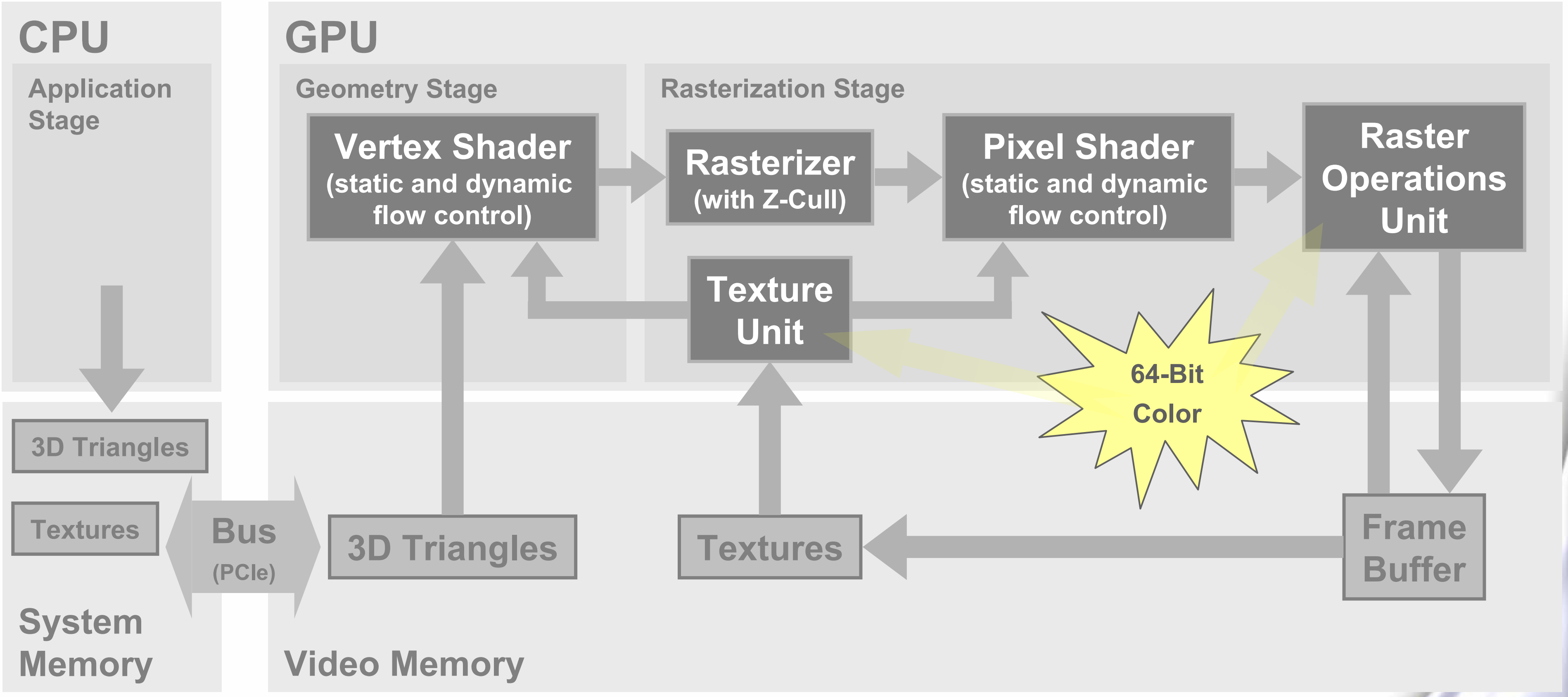
Shader: Static vs. Dynamic Flow Control

Static Flow Control
(condition varies
per batch of triangles)

Dynamic Flow Control
(condition varies
per vertex or pixel)

```
void Shader(
    ...
          // Input per vertex or per pixel
       in float3 normal,

          // Input per batch of triangles
  uniform float3 lightDirection,
  uniform bool computeLight,

    ...
)
{
  ...
  if (computeLight) {
    ...
    if (dot(lightDirection, normal)) {
      ...
    }
    ...
  }
  ...
}
```

2004: Shader Model 3.0 and 64-Bit Color Support



- **PCIe: Peripheral Component Interconnect Express**
- **NVIDIA's GeForce 6 Series (6800, 6600 and 6200)**

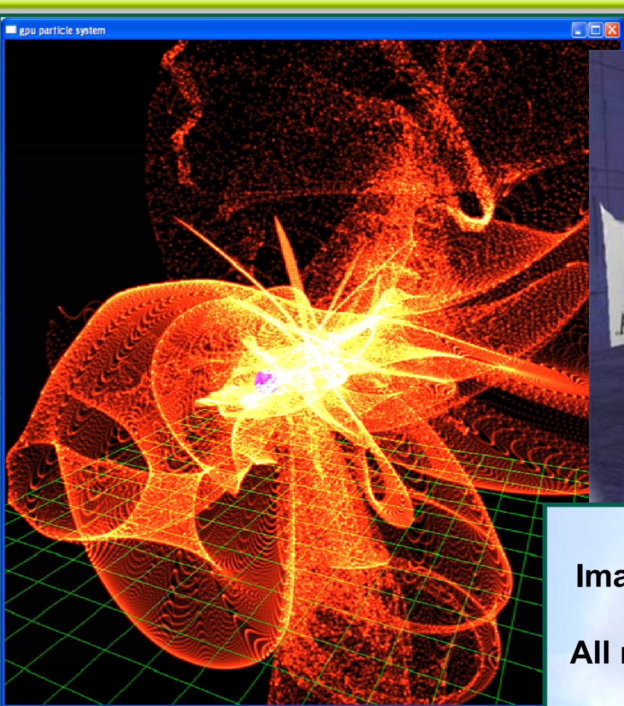
PCIe

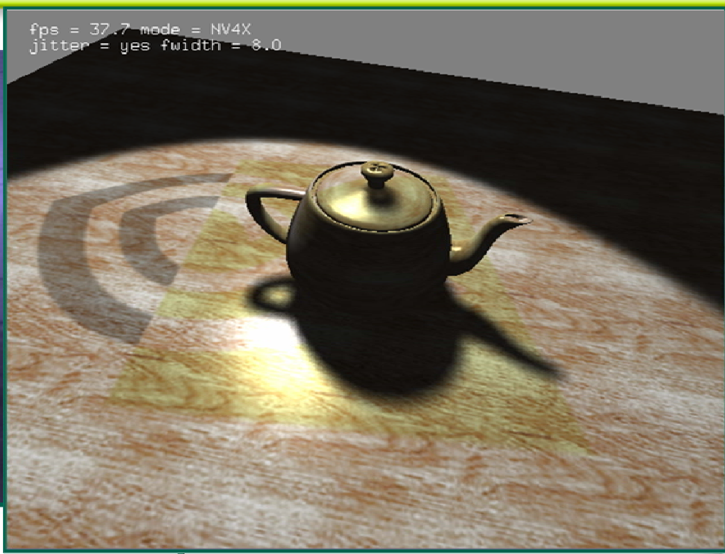
- Like AGP:
  - Uses a serial connection → Cheap, scalable
  - Uses a point-to-point protocol → No shared bandwidth
- Unlike AGP:
  - General-purpose (not only for graphics)
  - Dual-channels: Bandwidth is available in both directions
- Bandwidth: PCIe = 2 x AGP8x

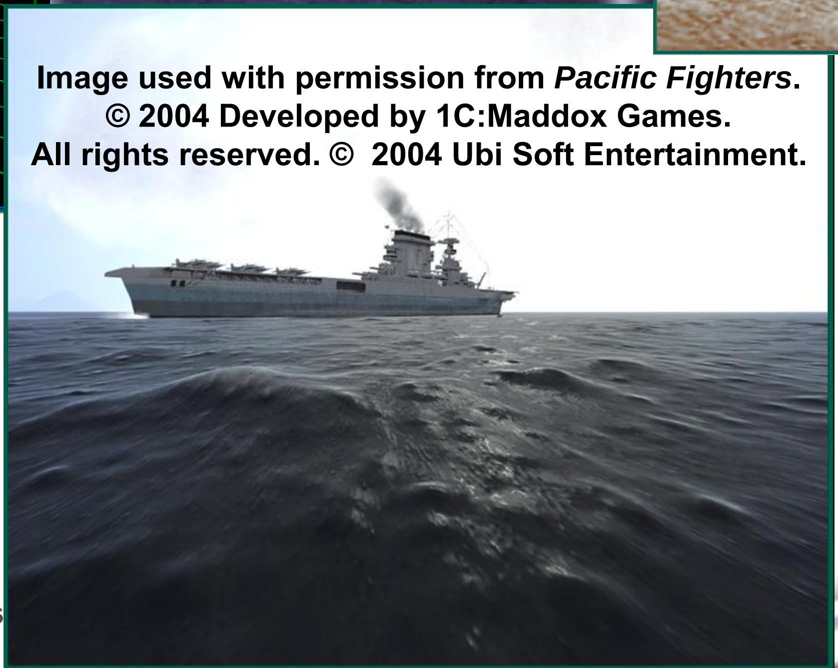
Shader Model 3.0

- Shader Model 3.0 means:
  - Longer shaders → More complex shading
  - Pixel shader:
    - Dynamic flow control → Better performance
    - Derivative instructions → Shader antialiasing
    - Support for 32-bit floating-point precision → Fewer artifacts
    - Face register → Faster two-sided lighting
  - Vertex shader:
    - Texture access (Vertex Texture Fetch)
      → Simulation on GPU, displacement mapping
    - Vertex buffer frequency → Efficient geometry instancing

Shader Model 3.0 Unleashed







64-Bit Color Support

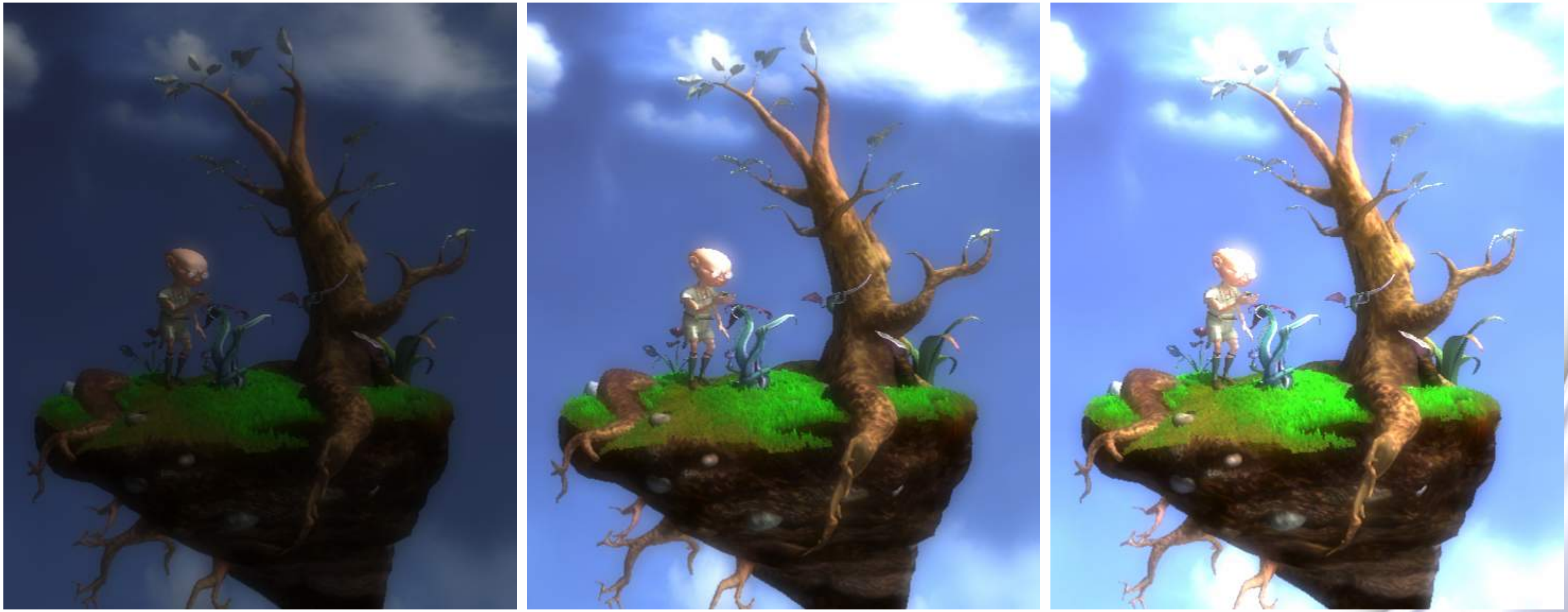
- **64-bit color** means one 16-bit floating-point value per channel (R, G, B, A)
- **Alpha blending** works with 64-bit color buffer (as opposed to 32-bit fixed-point color buffer only)
- **Texture filtering** works with 64-bit textures (as opposed to 32-bit fixed-point textures only)
- Applications:
  - High-precision image compositing
  - High dynamic range imagery

High Dynamic Range Imagery

- The dynamic range of a scene is the ratio of the highest to the lowest luminance
- Real-life scenes can have high dynamic ranges of several millions
- Display and print devices have a low dynamic range of around 100
- Tone mapping is the process of displaying high dynamic range images on those low dynamic range devices
- High dynamic range images use floating-point colors
- OpenEXR is a high dynamic range image format that is compatible with NVIDIA's 64-bit color format

Real-Time Tone Mapping

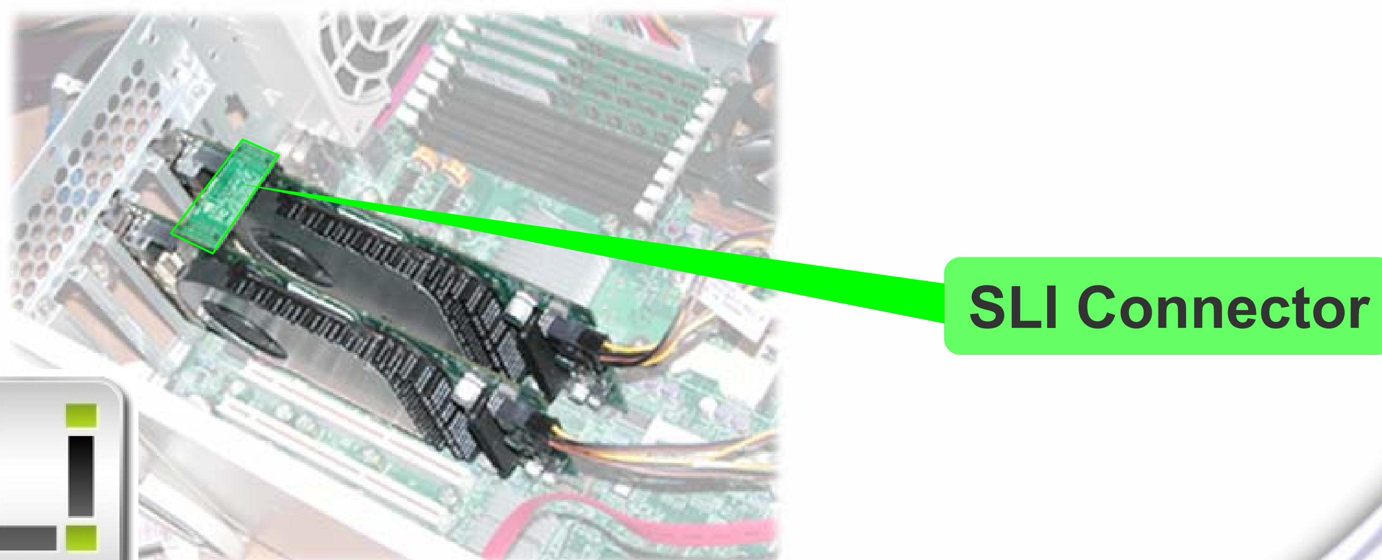
- **The image is entirely computed in 64-bit color and tone-mapped for display**



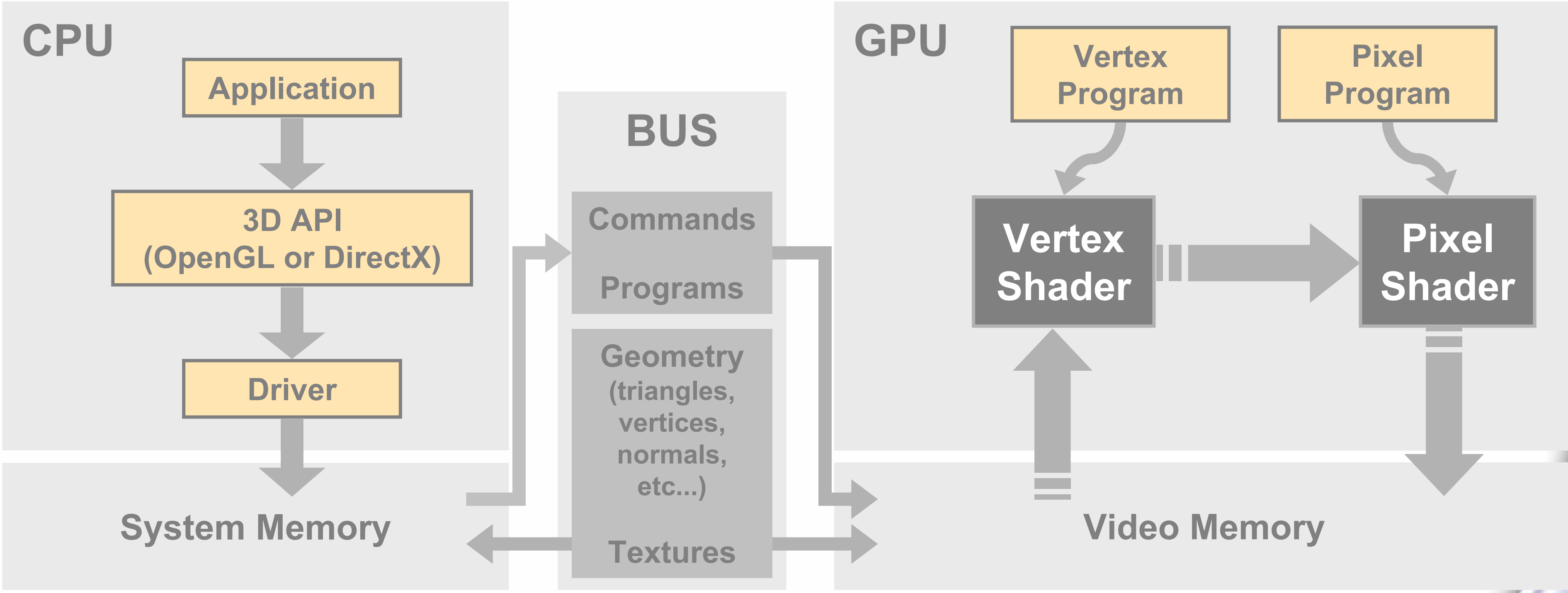
From low to high exposure image of the same scene

2005: Multi-GPU

- NVIDIA's Scalable Link Interface multi-GPU technology takes advantage of the increased bandwidth of the PCI Express to automatically accelerates applications through a combination of intelligent hardware and software solutions



PC Graphics Software Architecture



- The application, 3D API and driver are written in C or C++
- The vertex and pixel programs are written in a high-level shading language (Cg, DirectX HLSL, OpenGL Shading Language)

Basic Structure of a Graphics Application

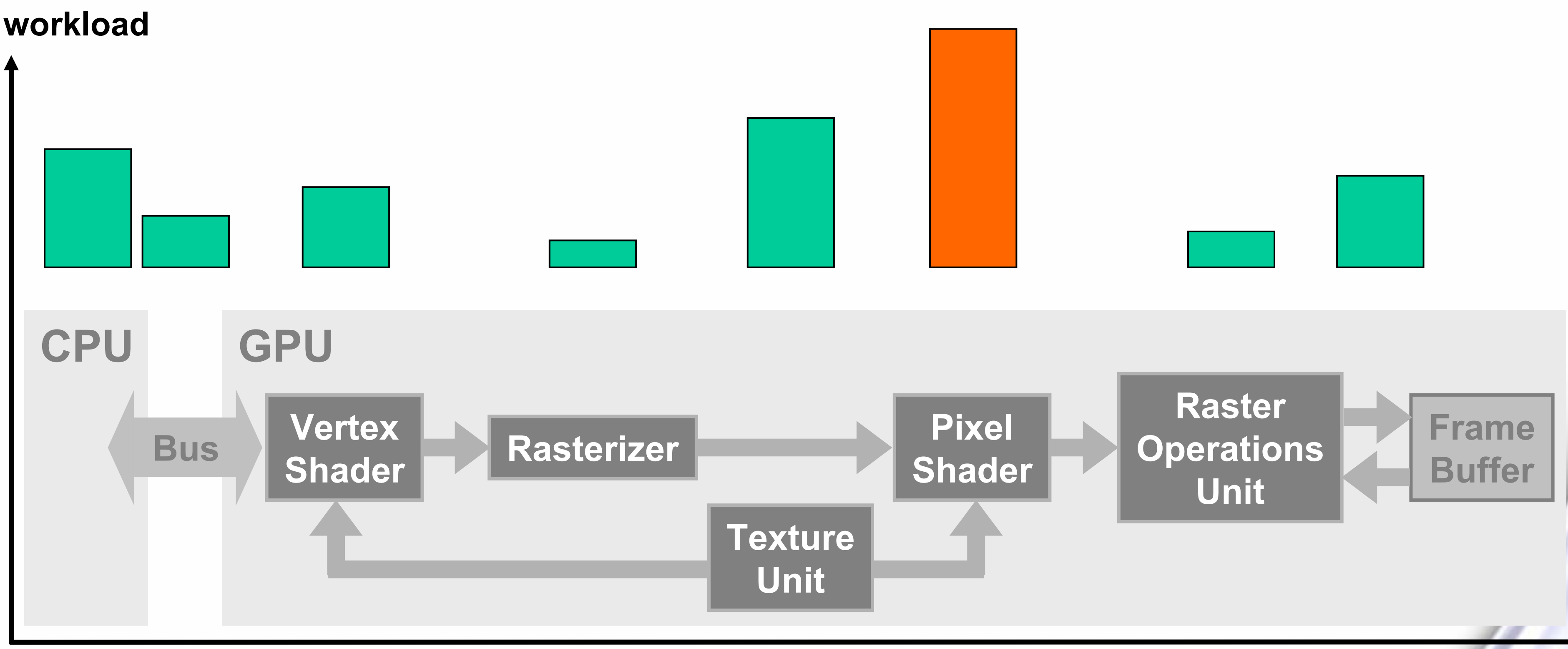
Initialization

- Initialize API
- Check hardware capabilities
- Create resources: render targets, shaders, textures, geometry

Rendering loop

- For every frame:
  - Draw to back buffer:
    - Clear frame buffer
    - For each draw call:
      - Set index and vertex buffers
      - Set vertex and pixel shaders and their parameters
      - Set texture sampling parameters
      - Set render states
      - Set render target
      - Issue draw command
  - Swap back buffer and front buffer

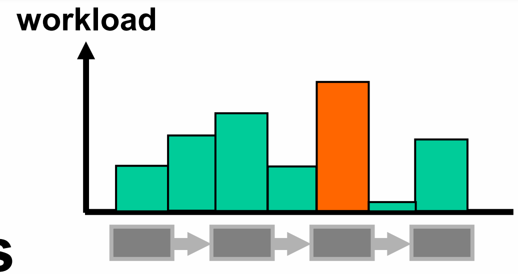
Optimization: Bottleneck



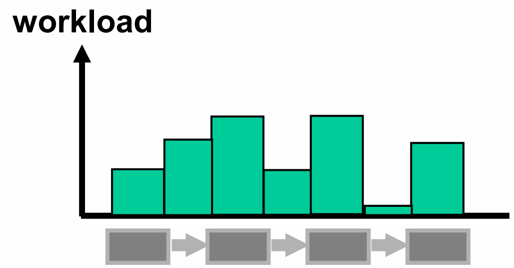
- A multi-processor pipeline architecture means that the overall throughput is determined by the bottleneck
- The bottleneck varies from one draw call to another

Optimization: Working on the Bottleneck

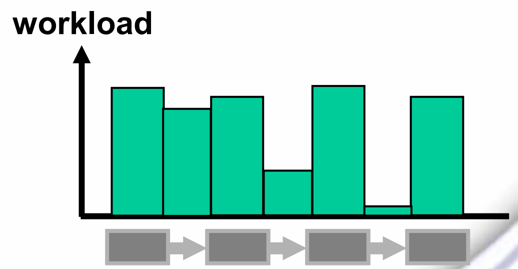
1. Find bottleneck by selectively:
   - modifying workload of stages
   - under-clocking various domains (CPU, bus, GPU)



2. Decrease workload of bottleneck:

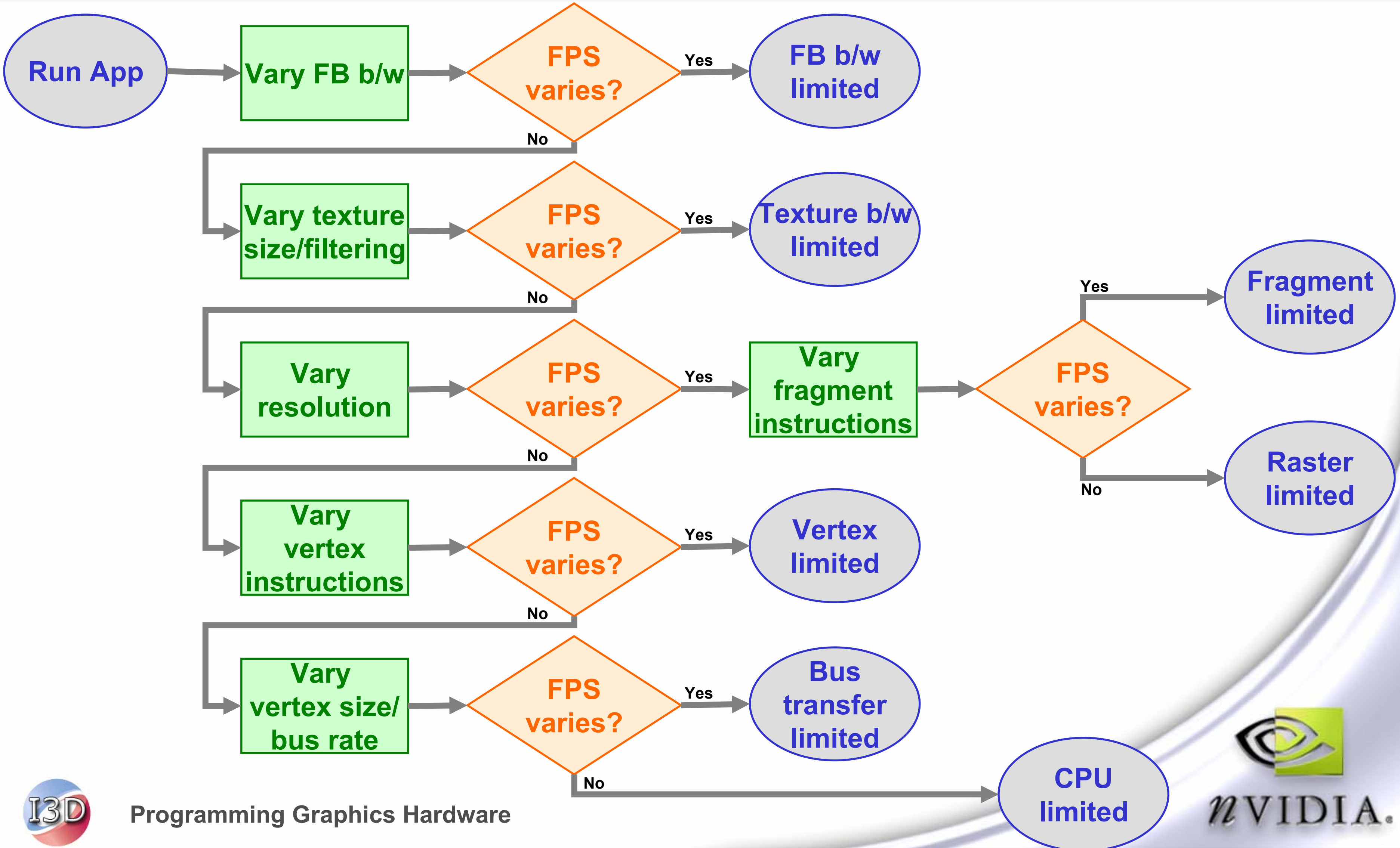


3. Or increase workload of non-bottleneck stages:

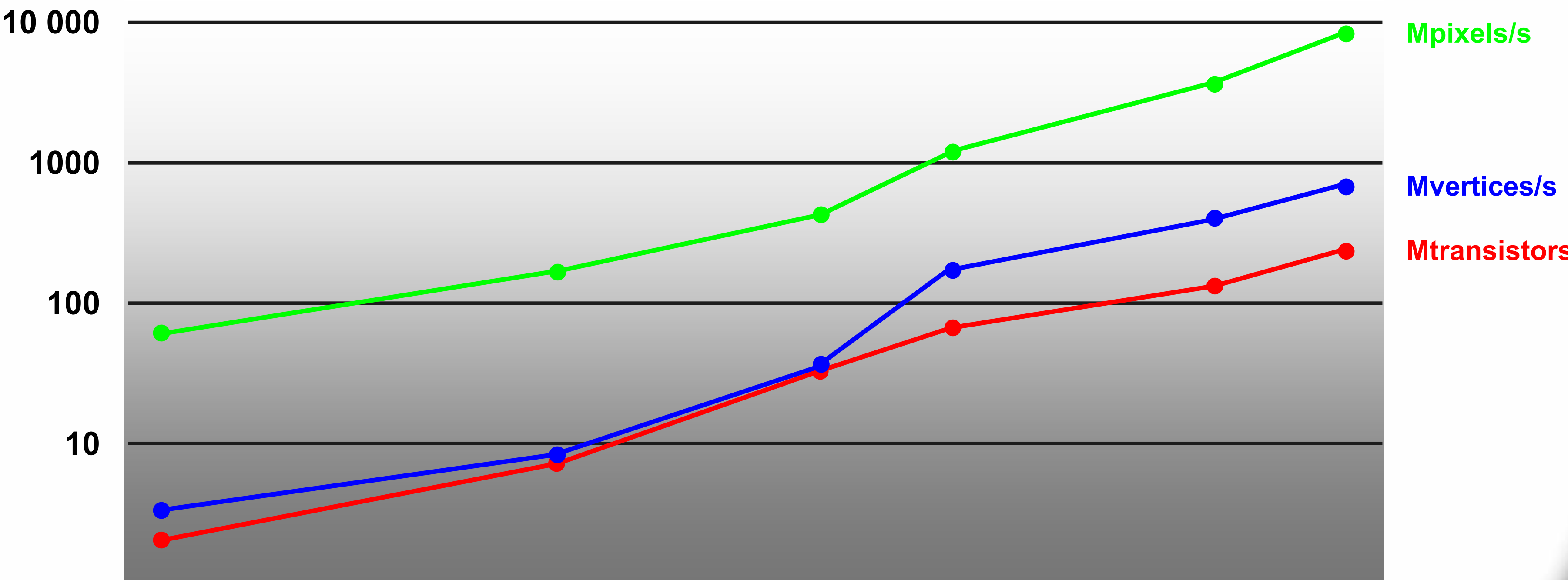


4. Go back to 1.

Optimization: Finding the Bottleneck



Evolution of Performance



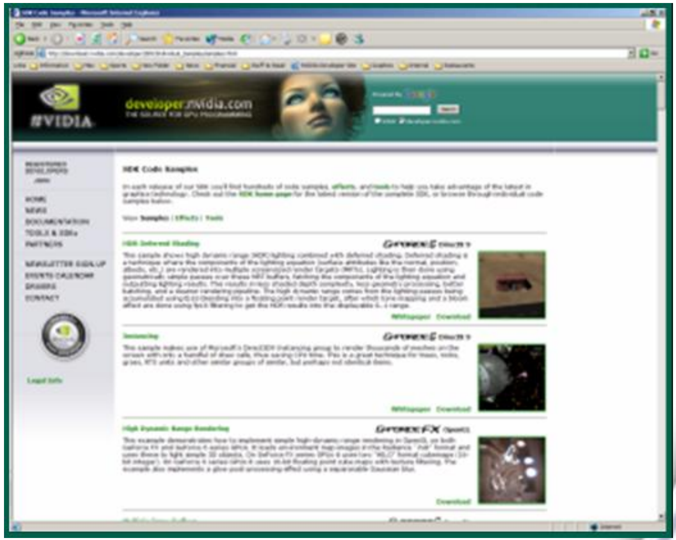
	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004
Bus	PCI (133 MB/s)		AGP (266 MB/s)	AGP2x (533 MB/s)	AGP4x (1.06 GB/s)			AGP8x (2.1 GB/s)		PCIe (4 GB/s)
Video memory	4 MB			32 MB	64 MB	128 MB			256 MB	512 MB
API	DirectX 1 OpenGL 1.1	DirectX 2	DirectX 3	DirectX 5 OpenGL 1.2	DirectX 6	DirectX 7	DirectX 8 OpenGL 1.3	OpenGL 1.4	DirectX 9 OpenGL 1.5	

The Future

- Unified general programming model at primitive, vertex and pixel levels
- Scary amounts of:
  - Floating point horsepower
  - Video memory
  - Bandwidth between system and video memory
- Lower chip costs and power requirements to make 3D graphics hardware ubiquitous:
  - Automotive (gaming, navigation, heads-up displays)
  - Home (remotes, media center, automation)
  - Mobile (PDAs, cell phones)

References

- Tons of resources at http://developer.nvidia.com:
  - Code samples
  - Programming guides
  - Recent conference presentations



- A good website and book on real-time rendering: http://www.realtimerendering.com

Questions

- Support e-mail:
  - devrelfeedback@nvidia.com [Technical Questions]
  - sdkfeedback@nvidia.com [Tools Questions]