The OpenGL Rendering Pipeline

CSE 781 Winter 2010

Han-Wei Shen

Brief History of OpenGL

- Originated from a proprietary API called Iris GL from Silicon Graphics, Inc.
- Provide access to graphics hardware capabilities at the lowest possible level that still provides hardware independence
- The evolution is controlled by OpenGL Architecture Review Board, or ARB.
- OpenGL 1.0 API finalized in 1992, first implementation in 1993
- In 2006, OpenGL ARB became a workgroup of the Khronos Group
- 10 revisions since 1992

OpenGL Evolution

- 1.1 (1997): vertex arrays and texture objects
- 1.2 (1998): 3D textures
- 1.3 (2001): cubemap textures, compressed textures, multitextures
- 1.4 (2002): mipmap generation, shadow map textures, etc
- 1.5 (2003): vertex buffer object, shadow comparison functions, occlusion queries, non-power-of-2 textures

OpenGL Evolution

- 2.0 (2004): vertex and fragment shading (GLSL 1.1), multiple render targets, etc
- 2.1 (2006): GLSL 1.2, pixel buffer objects, etc
- 3.0 (2008): GLSL 1.3, deprecation model, etc
- 3.1 (2009): GLSL 1.4, texture buffer objects, move much of deprecated functions to ARB compatible extension
- 3.2 (2009)

OpenGL Extensions

- New features/functions are marked with prefix
- Supported only by one vendor
  - NV_float_buffer (by nvidia)
- Supported by multiple vendors
  - EXT_framebuffer_object
- Reviewed by ARB
  - ARB_depth_texture
- Promoted to standard OpenGL API

Deprecation Model, Contexts, and Profiles

- Redundant and In-efficient functions are deprecated – to be removed in the future
  - glBegin(), glEnd()
- OpenGL Contexts – data structures where OpenGL stores the state information used for rendering
  - Textures, buffer objects, etc
- Profile – A subset of OpenGL functionality specific to an application domain
  - Gaming, computer-aided design, embedded programs

The Rendering Pipeline

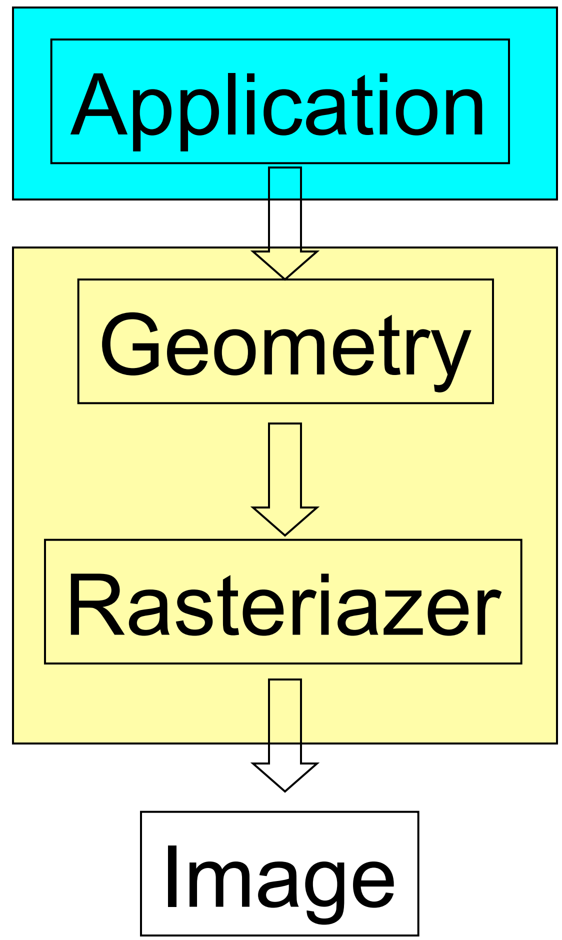
- The process to generate two-dimensional images from given virtual cameras and 3D objects
- The pipeline stages implement various core graphics rendering algorithms
- Why should you know the pipeline?
  - Understand various graphics algorithms
  - Program low level graphics systems
  - Necessary for programming GPUs
  - Help analyze the performance bottleneck

The Rendering Pipeline

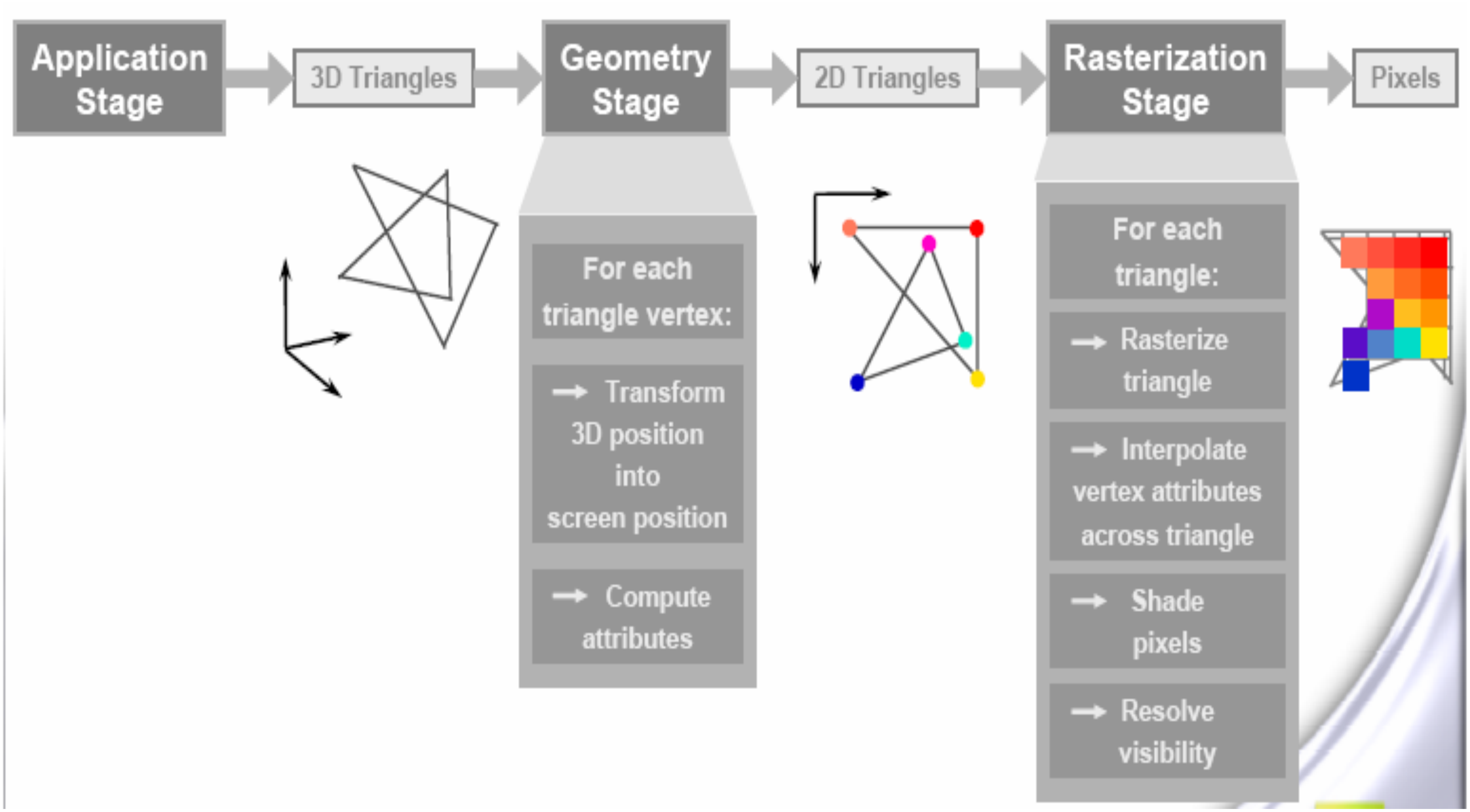
- The basic construction – three conceptual stages
- Each stage is a pipeline and runs in parallel
- Graphics performance is determined by the slowest stage
- Modern graphics systems:

software:

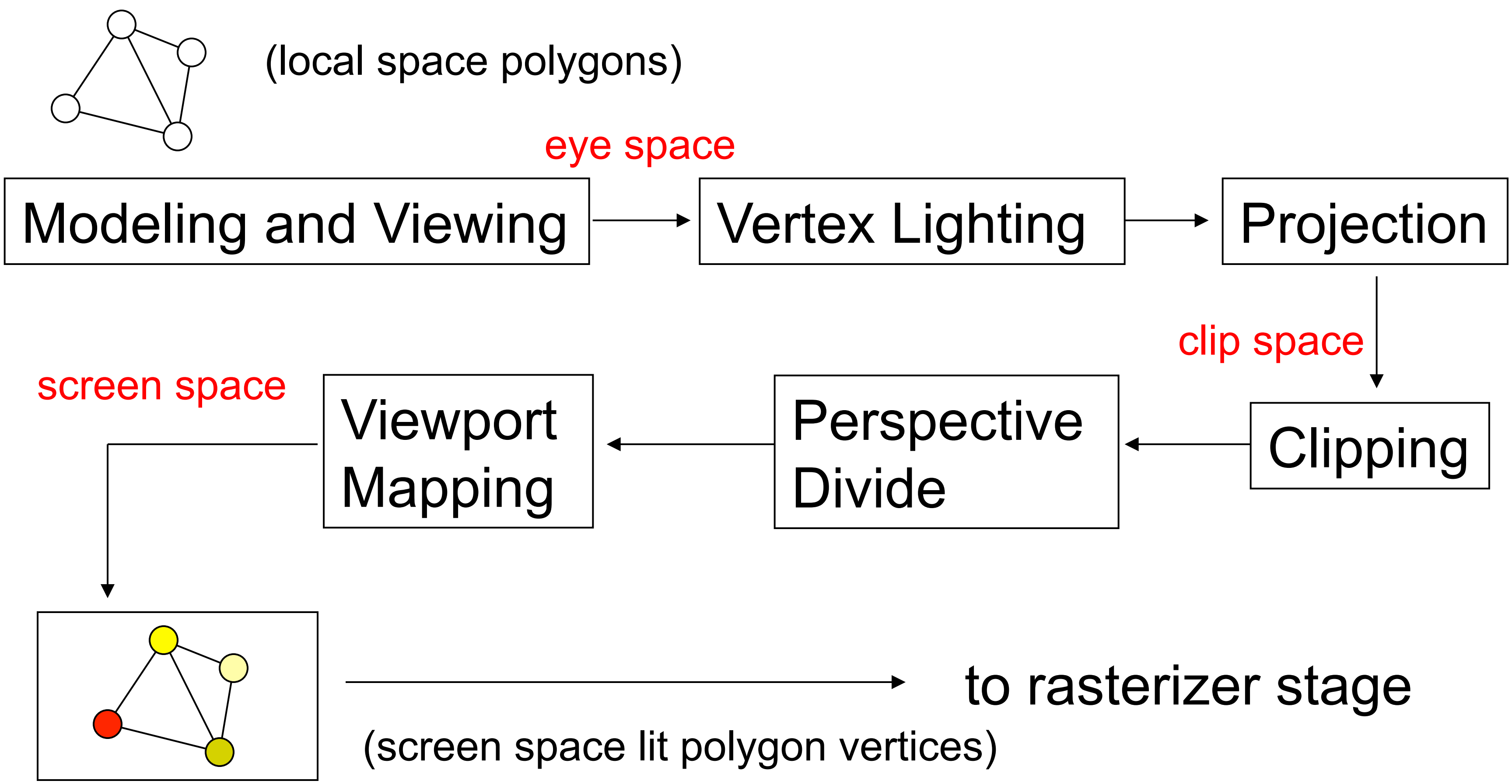
hardware:



The Rendering Pipeline

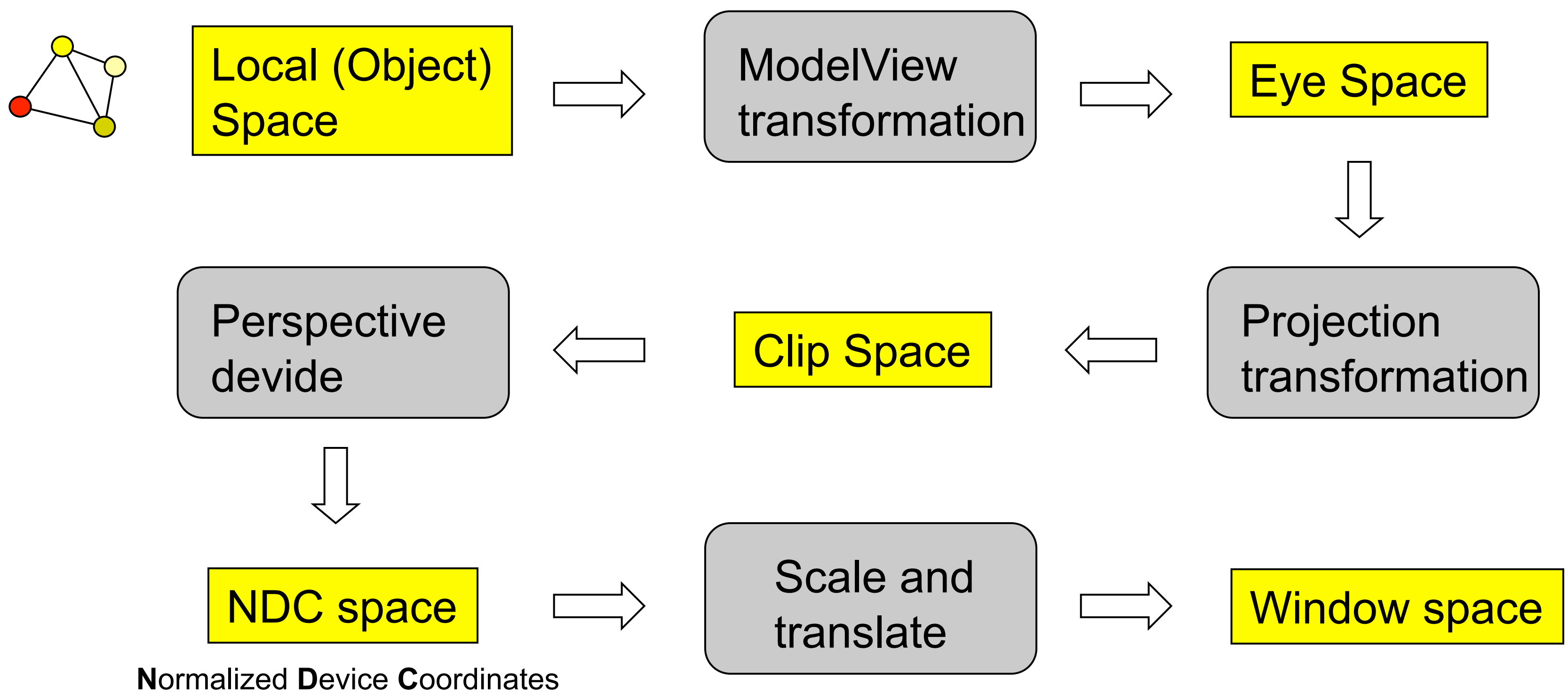


The Geometry Stage



Transformation Pipeline

- Another view of the graphics pipeline



Different Spaces

- Local space
  - A space where you define the vertex coordinates, normals, etc. This is before any transformations are taking place
  - These coordinates/normals are multiplied by the OpenGL modelview (VM) matrix into the eye space
  - Modelview matrix: Viewing transformation matrix (V) multiplied by modeling transformation matrix (M), i.e., GL_MODELVIEW = V * M
  - OpenGL matrix stack is used to allow different modelview matrices for different objects

Different Spaces (cont'd)

- Eye space
  - Where per vertex lighting calculation is occurred
  - Camera is at (0,0,0) and view's up direction is by default (0,1,0)
  - Light position is stored in this space after being multiplied by the OpenGL modelview matrix
  - Vertex normals are consumed by the pipeline in this space by the lighting equation

Different Spaces (cont'd)

- Clip Space
  - After projection and before perspective divide
  - Clipping against view frustum done in this space
    - -W <= X <= W; -W <=Y <=W; -W <=Z <=W;
  - New vertices are generated as a result of clipping
  - The view frustum after transformation is a parallelepiped regardless of orthographic or perspective projection
- Perspective Divide
  - Transform clip space into NDC space
  - Divide (x,y,z,w) by w where w = z/-d (d=1 in OpenGL so w = -z)
  - Result in foreshortening effect

Different Spaces (cont'd)

- Window Space
  - Map the NDC coordinates into the window
    - X and Y are integers, relative to the lower left corner of the window
    - Z are scaled and biased to [0,1]
    - Rasterization is performed in this space
  - The geometry processing ends in this space

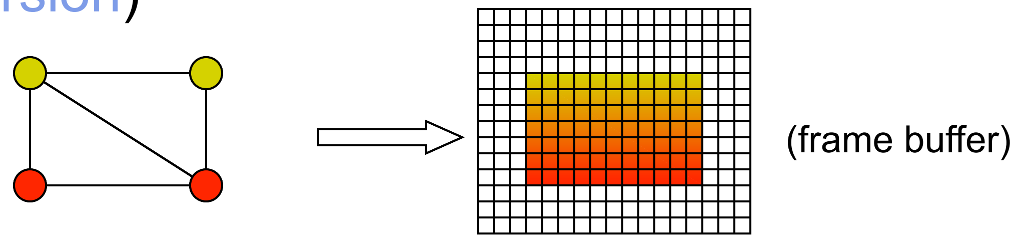
The Geometry Stage

- Transform coordinates and normal
  - Model->world
  - World->eye
- Normalize the normal vectors
- Compute vertex lighting
- Generate (if necessary) and transform texture coordinates
- Transform to clip space (by projection)
- Assemble vertices into primitives
- Clip against viewing frustum
- Divide by w (perspective divide if applies)
- Viewport transformation
- Back face culling

Introduce vertex dependences ☹

The Rasterizer Stage

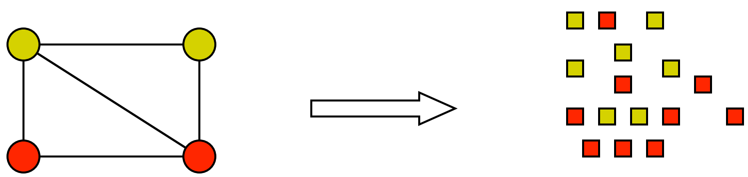
- Per-pixel operation: assign colors to the pixels in the frame buffer (a.k.a scan conversion)



- Main steps:
  - Setup
  - Sampling (convert a primitive to fragments)
  - Texture lookup and Interpolation (lighting, texturing, z values, etc)
  - Color combinations (illumination and texture colors)
  - Fogging
  - Other pixel tests (scissor, alpha, stencil tests etc)
  - Visibility (depth test)
  - Blending/compositing/Logic op

The Rasterization Stage

- Convert each primitive into fragments (not pixels)
- Fragment: transient data structures
  - position (x,y); depth; color; texture coordinates; etc



- Fragments from the rasterized polygons are then selected (z buffer comparison for instance) to form the frame buffer pixels

The Rasterization Stage

- Two main operations
  - Fragment selection: generate one fragment for each pixel that is intersected by the primitive
  - Fragment assignment: sample the primitive properties (colors, depths, etc) for each fragment - nearest neighbor continuity, linear interpolation, etc

Polygon Scan Conversion

- The goal is to compute the scanline-primitive intersections
- OpenGL Spec does not specify any particular algorithm to use
- Brute Force: try to intersect each scanline with all edges as we go from ymin to ymax
- We can do better
  - Find ymin and ymax for each edge and only test the edge with scanlines in between
  - For each edge, only calculate the intersection with the ymin; calculate dx/dy; calculate the new intersection as y=y+1, x +dx/dy
  - Change x=x+dx/dy to integer arithmetic (such as using Bresenham's algorithm)

Rasterization steps

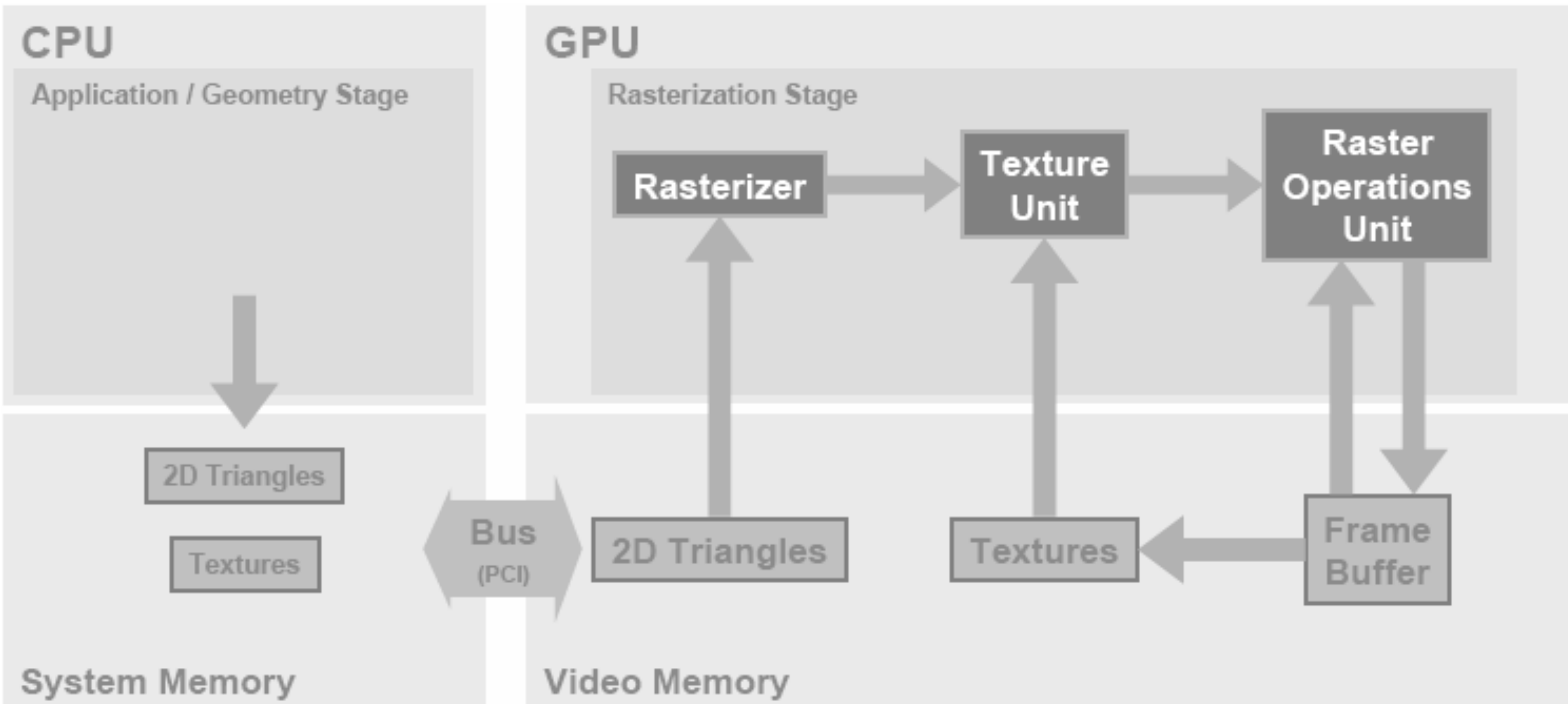
- Texture interpolation
- Color interpolation
- Fog (blend the fog color with the fragment color based on the depth value)
- Scissor test (test against a rectangular region)
- Alpha test (compare with alpha, keep or drop it)
- Stencil test(mask the fragment depending on the content of the stencil buffer)
- Depth test (z buffer algorithm)
- Alpha blending
- Dithering (make the color look better for low res display mode)

Overview of PC Graphics Hardware

Evolution of the PC hardware graphics pipeline:

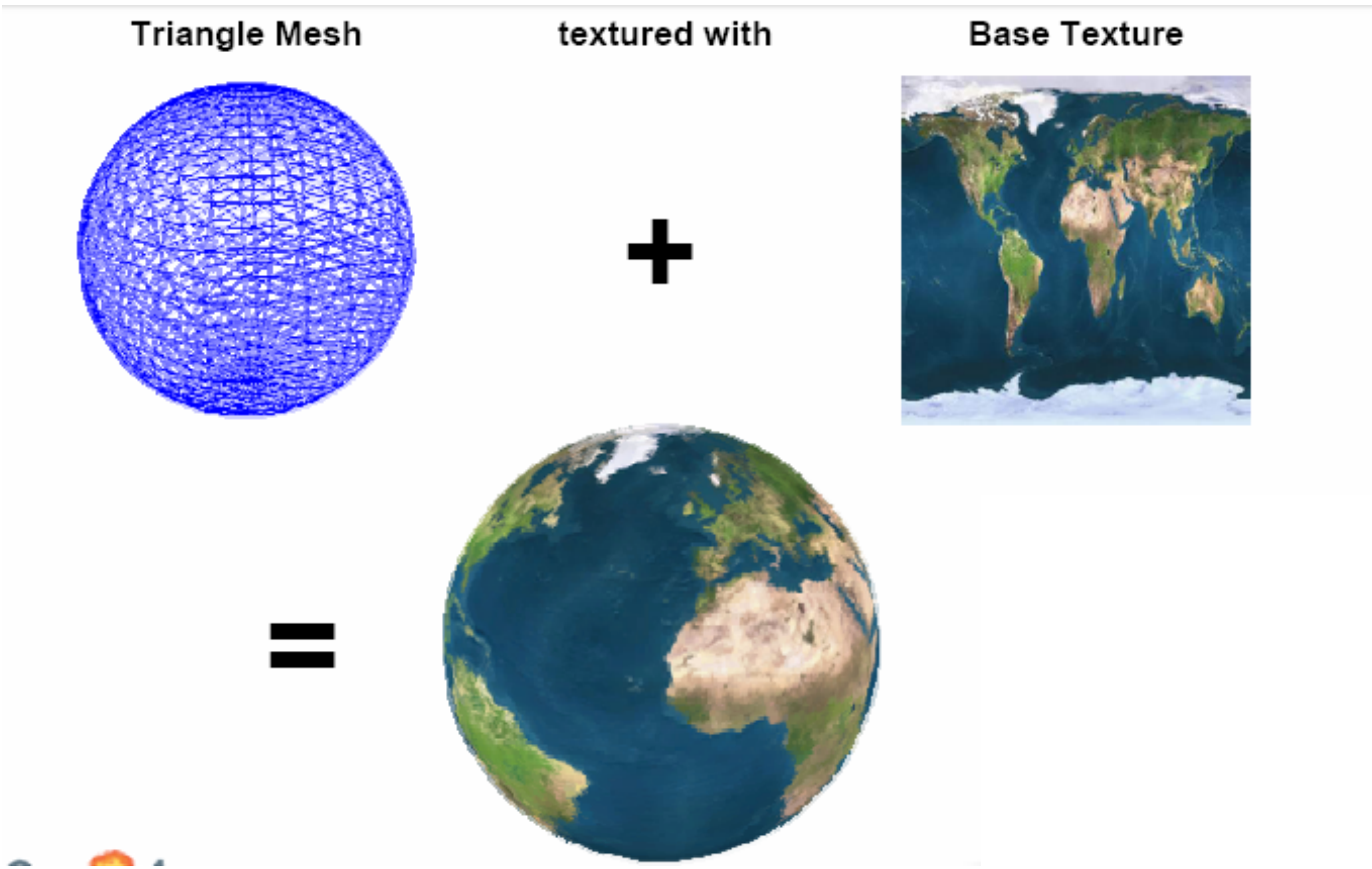
- 1995-1998: Texture mapping and z-buffer
- 1998: Multitexturing
- 1999-2000: Transform and lighting
- 2001: Programmable vertex shader
- 2002-2003: Programmable pixel shader
- 2004: Shader model 3.0 and 64-bit color support

1995-1998: texture mapping and z buffer

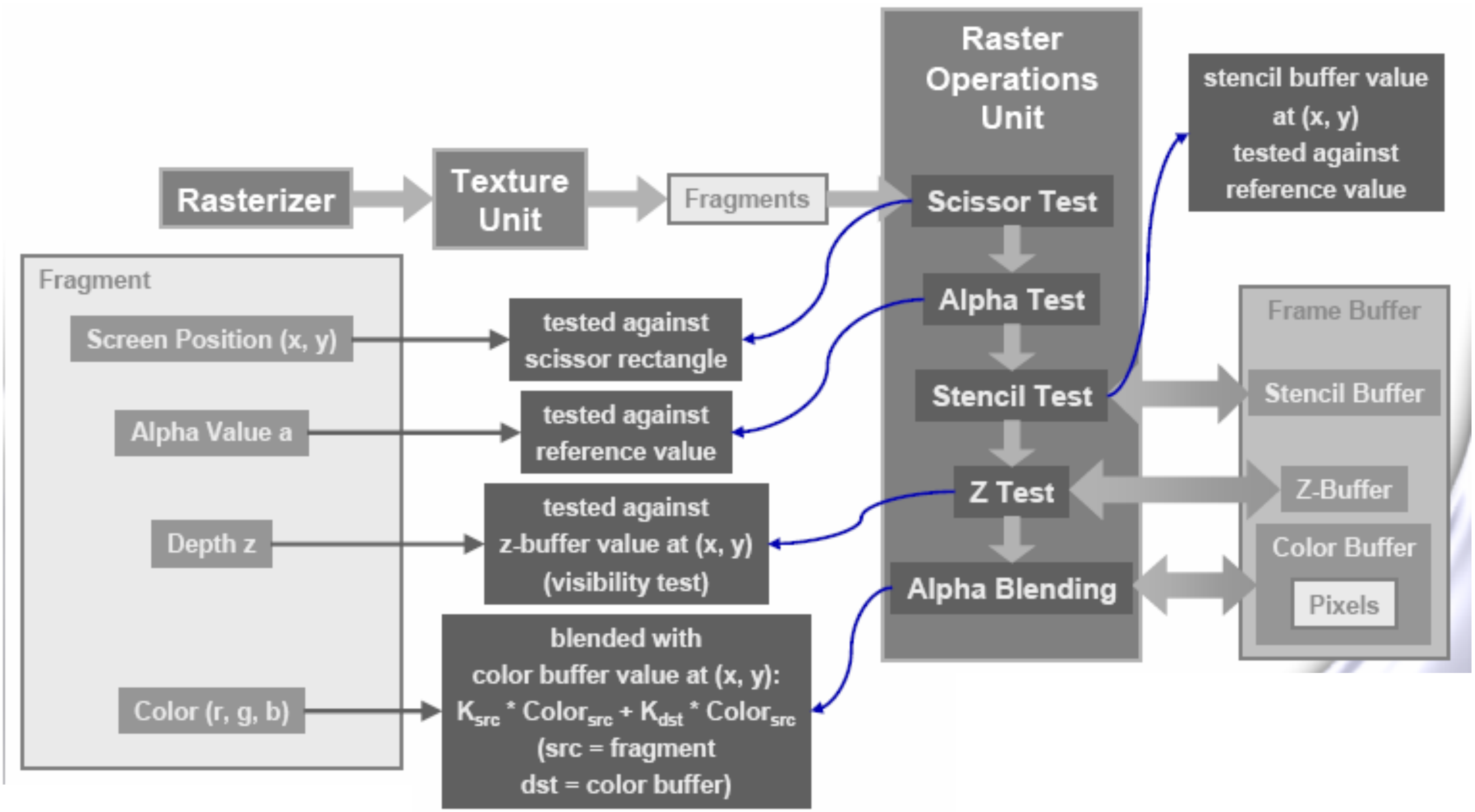


- PCI: Peripheral Component Interconnect
- 3dfx's Voodoo

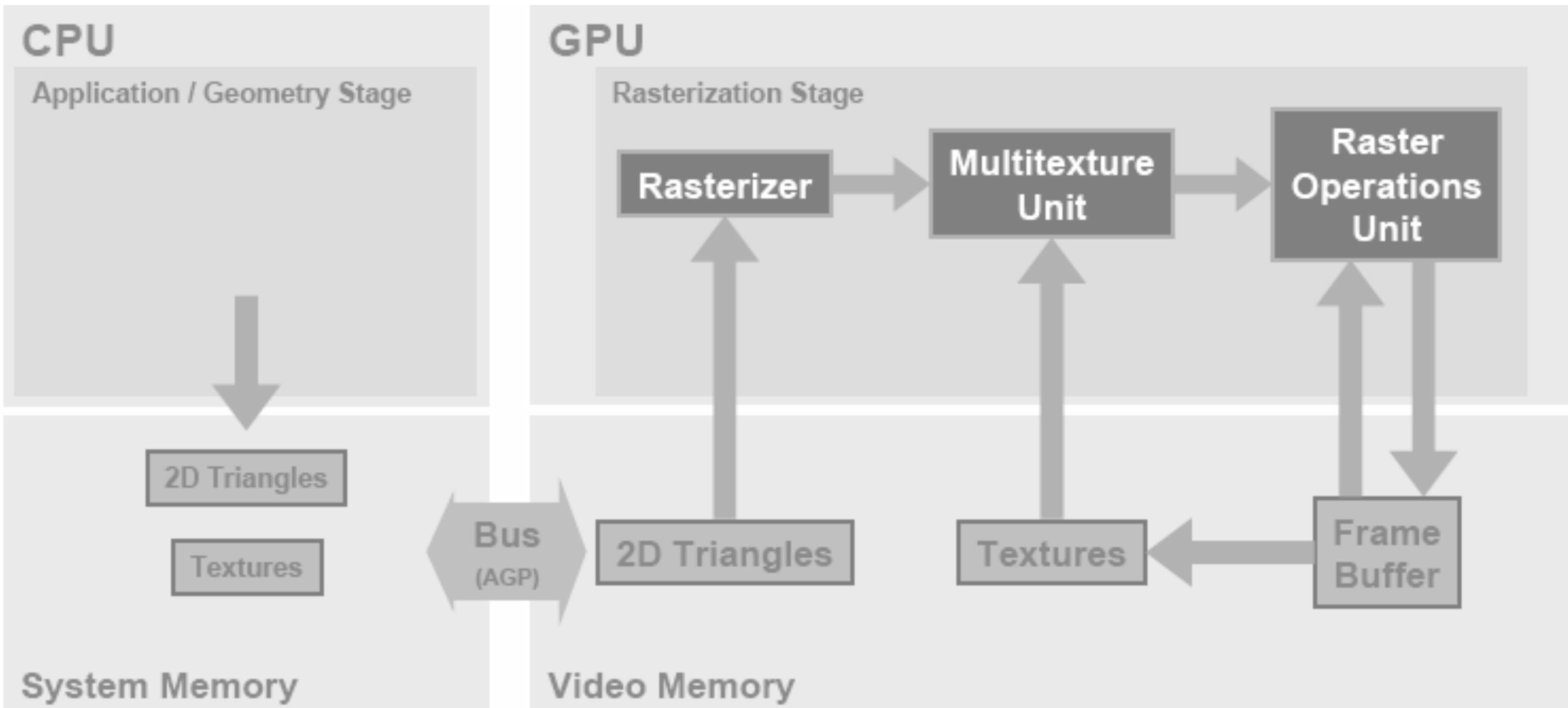
Texture Mapping



Raster Operations Unit

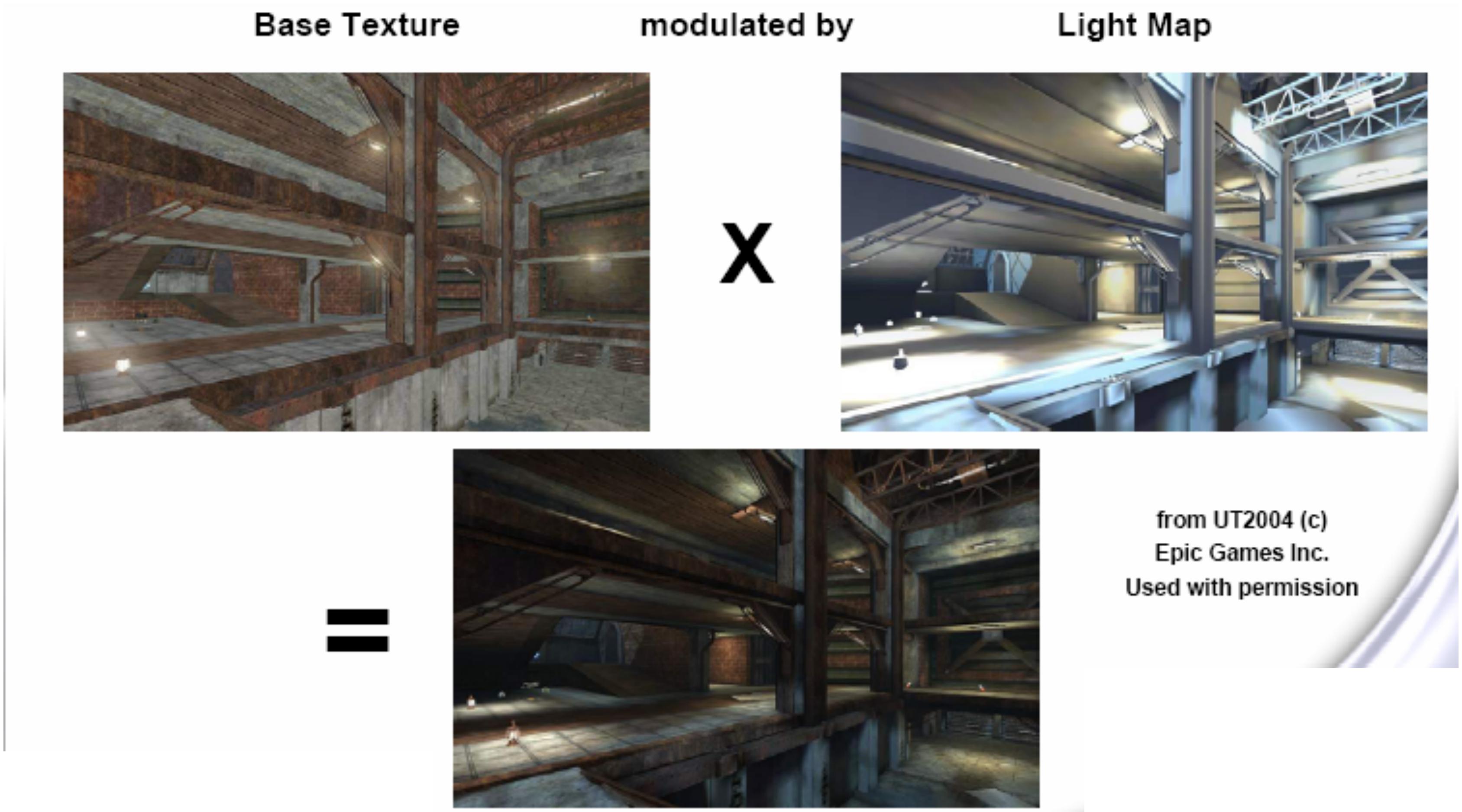


1998: multitexturing

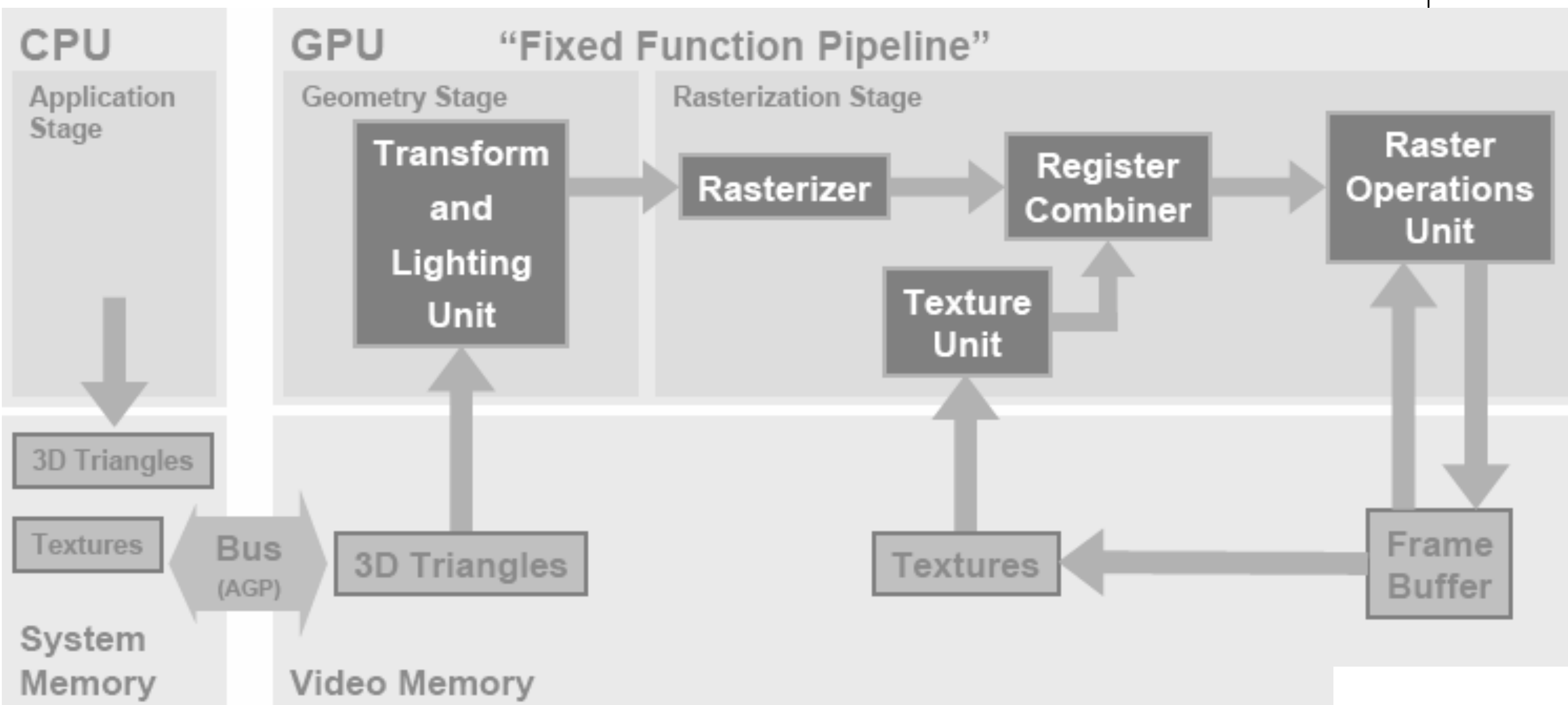


- AGP: Accelerated Graphics Port
- NVIDIA's TNT, ATI's Rage

Multitexturing

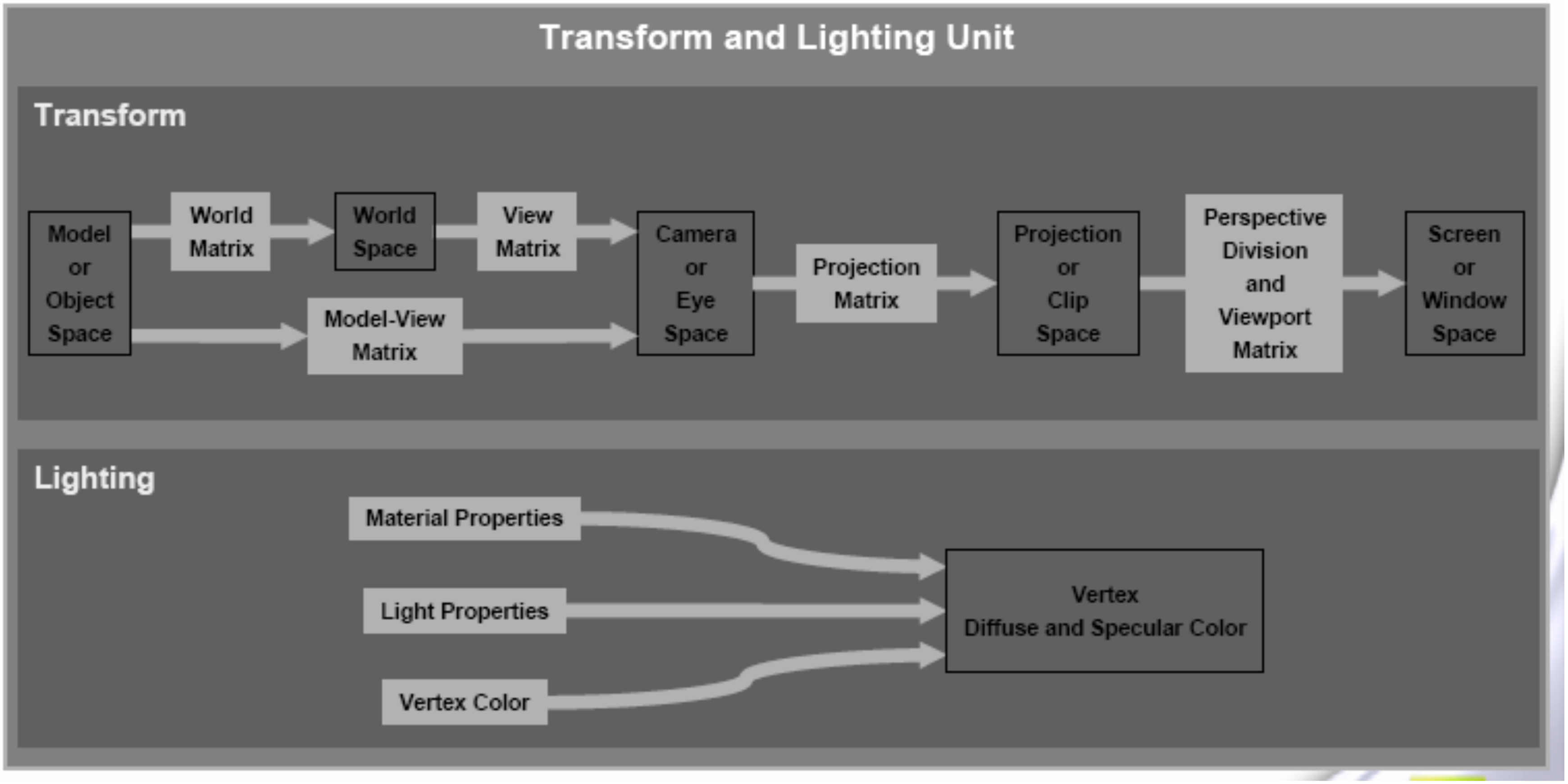


1999-2000: transform and lighting



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

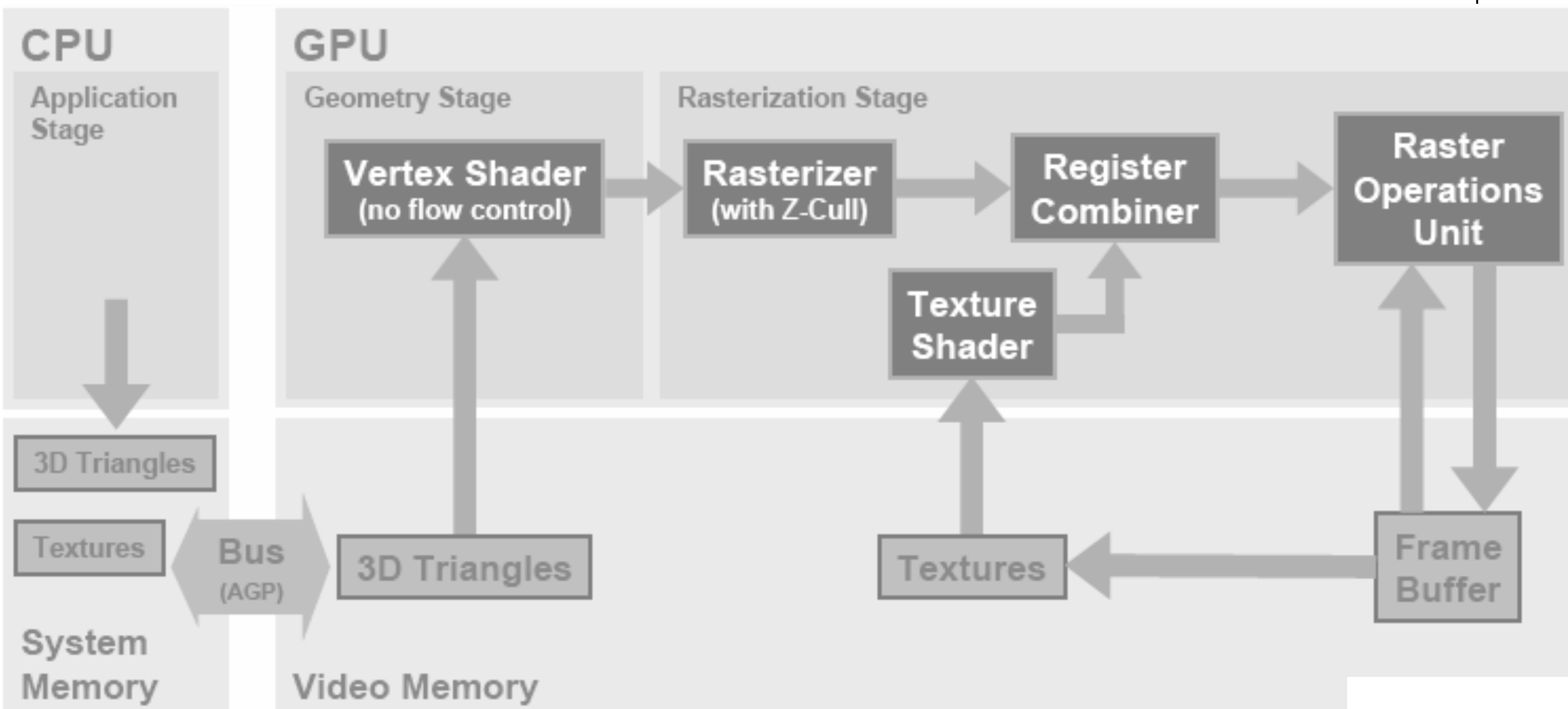
Transform and Lighting (TnL) unit



Programmable GPUs

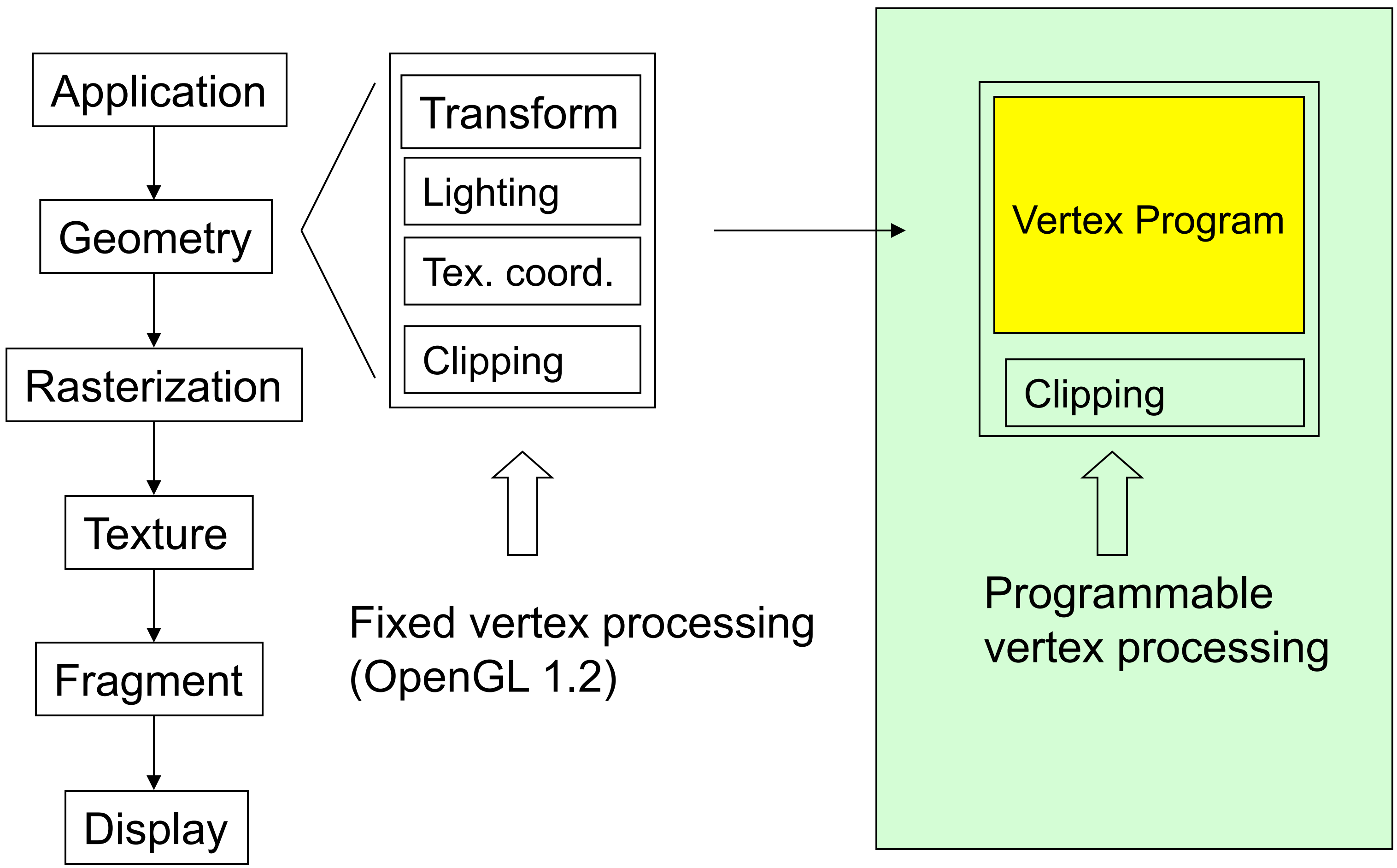
- So far we only discuss fixed graphics pipeline
  - Fixed T&L algorithms
  - Fixed Fragment processing steps
- New GPU trends – programmable vertex, geometry, and fragment processing

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

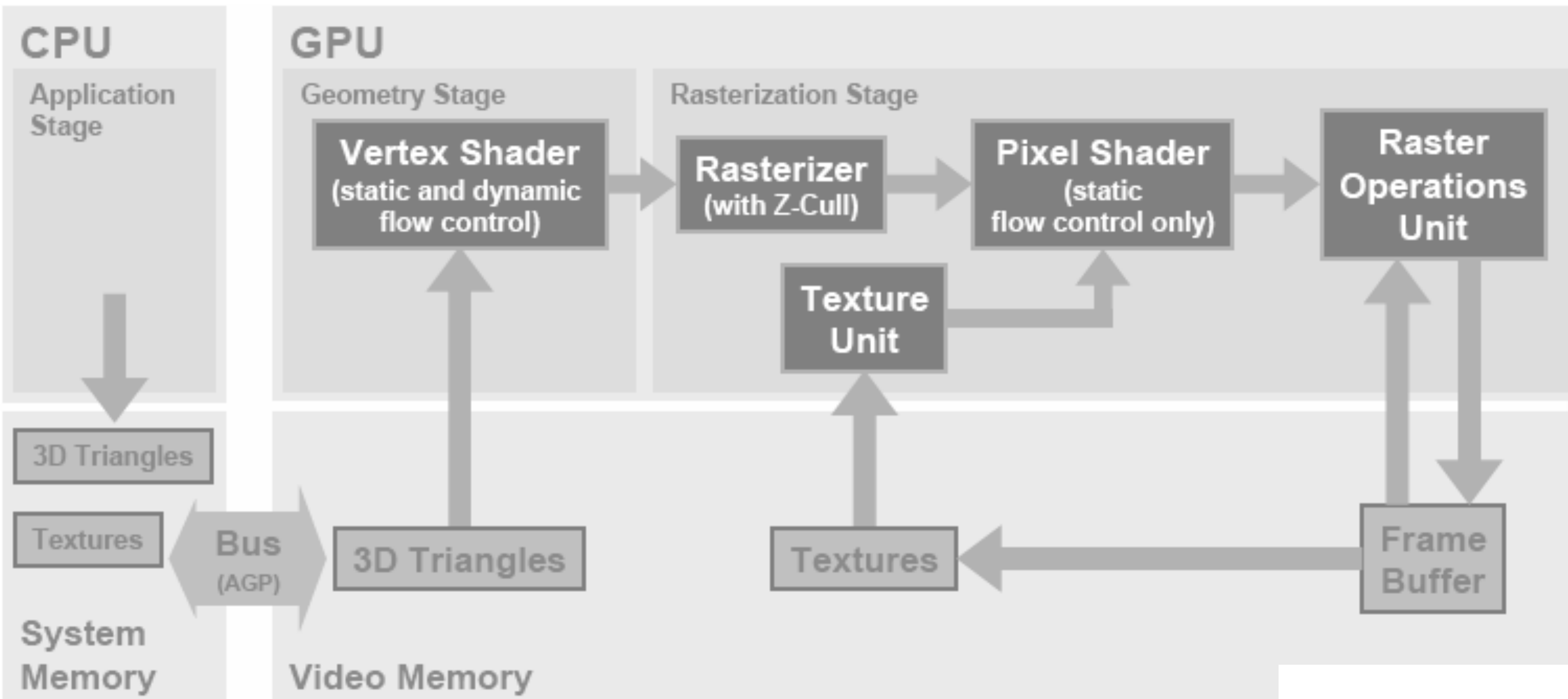
Vertex Program



Vertex Program

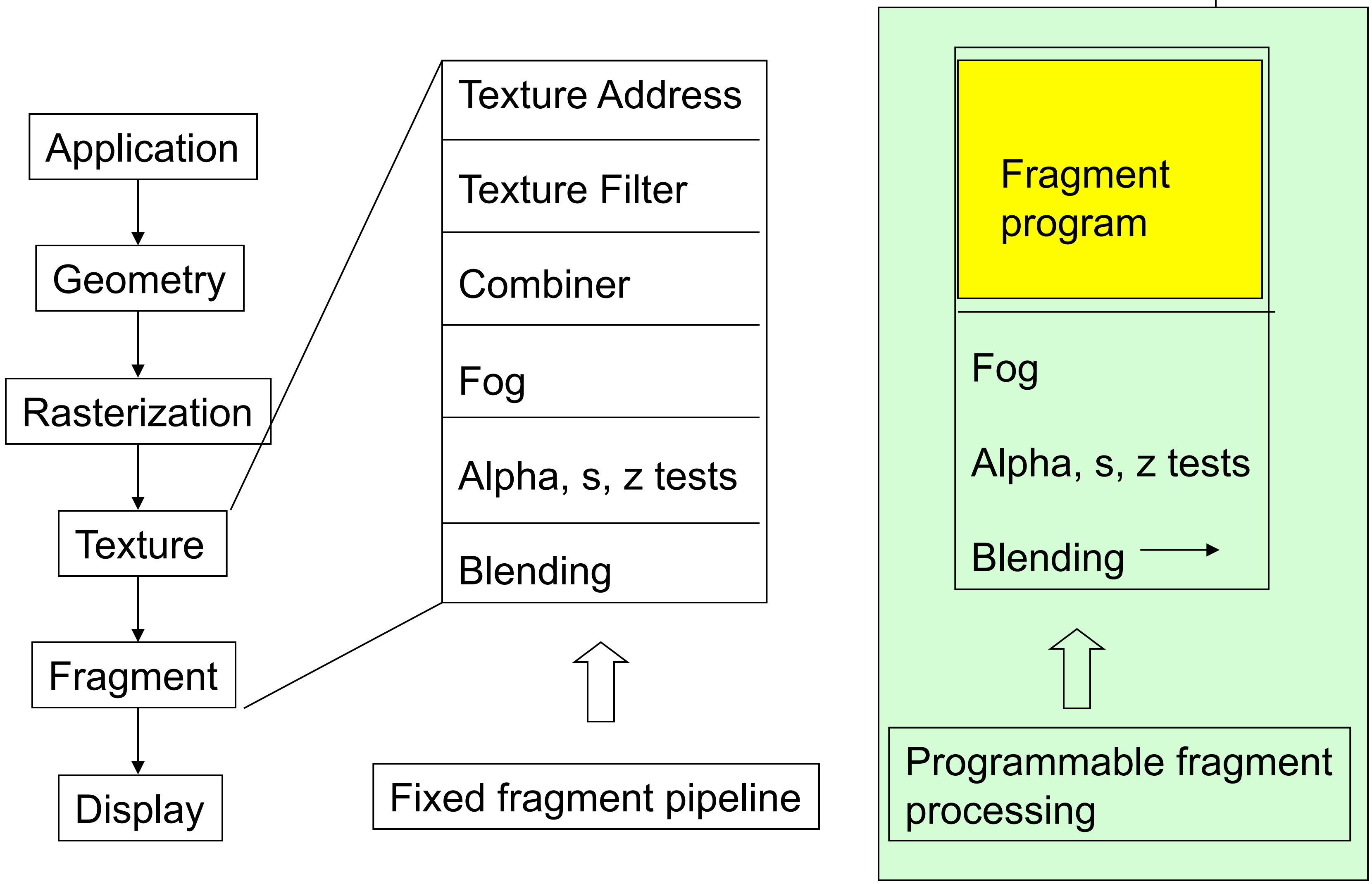
- Used to be only assembly language interface to T&L unit (2002)
  - GPU instruction set to perform all vertex math
  - Reads an untransformed, unlit vertex
  - Creates a transformed vertex
  - Optionally creates
    - Lights a vertex
    - Creates texture coordinates
    - Creates fog coordinates
    - Creates point sizes
- High level programming language APIs are available (GLSL, Cg, HLSL, etc)

2002-2003: programmable pixel shader

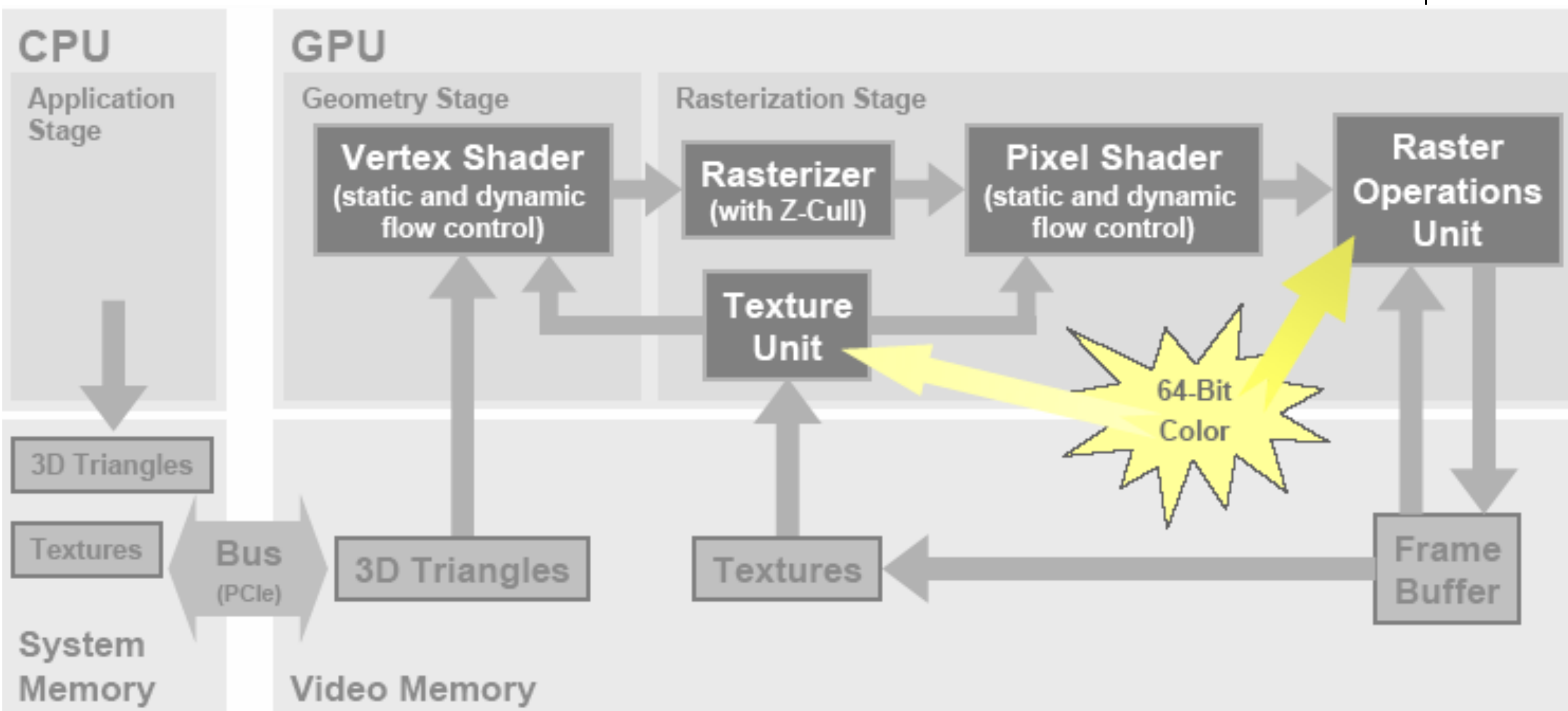


- MRT: Multiple Render Target
- NVIDIA's GeForce FX, ATI's Radeon 9600 to 9800 and X600 to X800

Fragment Programs



2004: shader model 3.0 and 64-bit colors

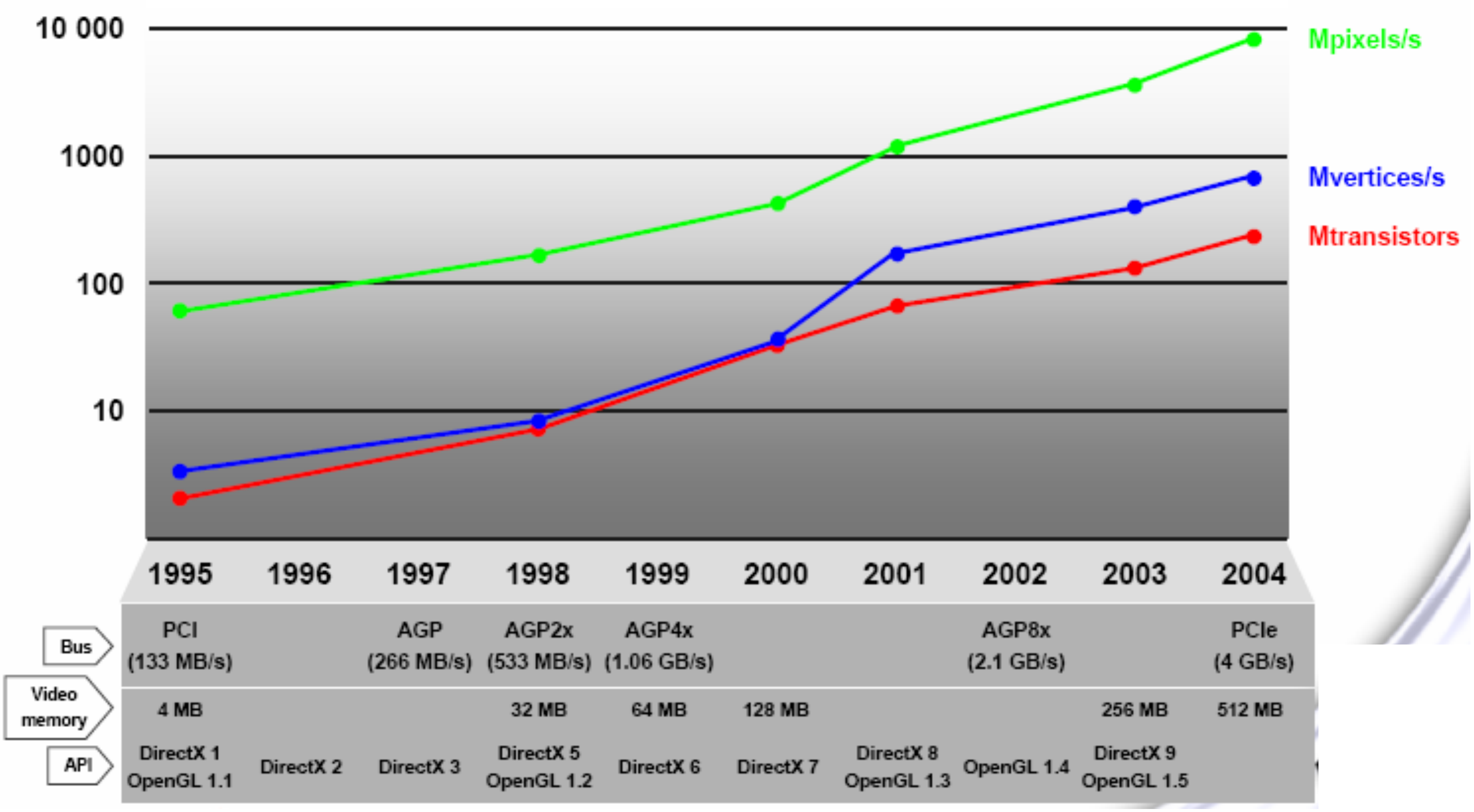


- PCIe: Peripheral Component Interconnect Express
- NVIDIA's GeForce 6 Series (6800 and 6600)

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 direction
- Bandwidth: PCIe = 2 x AGP8x

Evolution of Performance



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)