Understanding the graphics pipeline

Lecture 2

Original Slides by: Suresh Venkatasubramanian
Updates by Joseph Kider

Lecture Outline

- A historical perspective on the graphics pipeline
  - Dimensions of innovation.
  - Where we are today
  - Fixed-function vs programmable pipelines
- A closer look at the fixed function pipeline
  - Walk thru the sequence of operations
  - Reinterpret these as stream operations
- We can program the fixed-function pipeline !
  - Some examples
- What constitutes data and memory, and how access affects program design.

The evolution of the pipeline

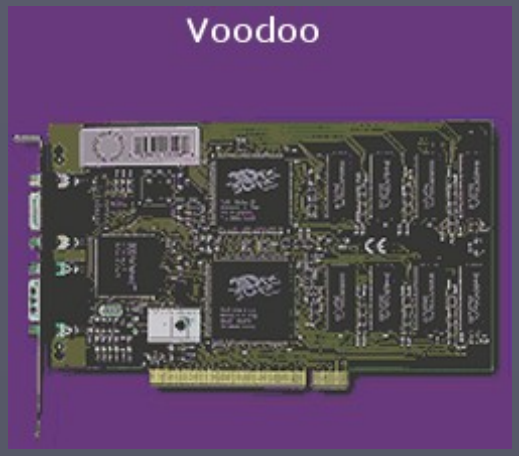
Elements of the graphics pipeline:

1. A scene description: vertices, triangles, colors, lighting
2. Transformations that map the scene to a camera viewpoint
3. “Effects”: texturing, shadow mapping, lighting calculations
4. Rasterizing: converting geometry into pixels
5. Pixel processing: depth tests, stencil tests, and other per-pixel operations.

Parameters controlling design of the pipeline:

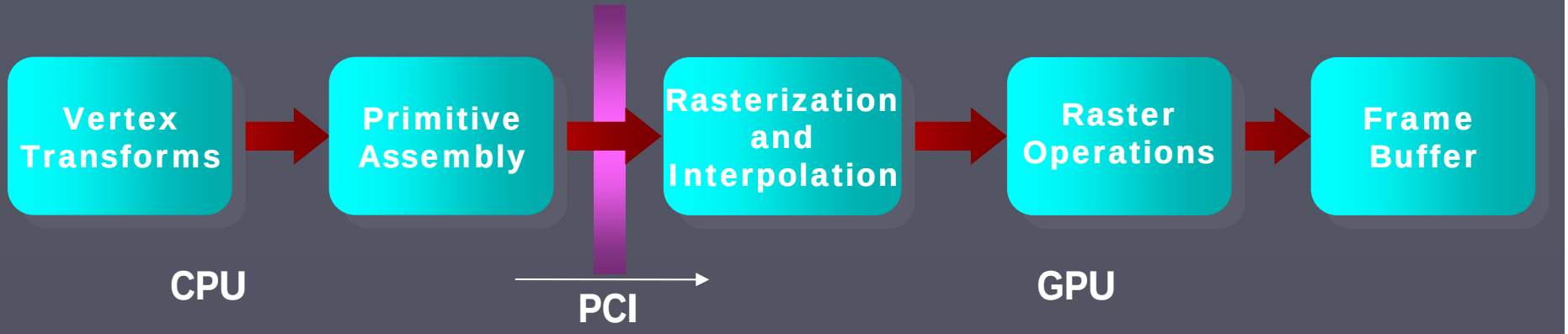
1. Where is the boundary between CPU and GPU ?
2. What transfer method is used ?
3. What resources are provided at each step ?
4. What units can access which GPU memory elements ?

Generation I: 3dfx Voodoo (1996)



http://accelenation.com/?ac.id.123.2

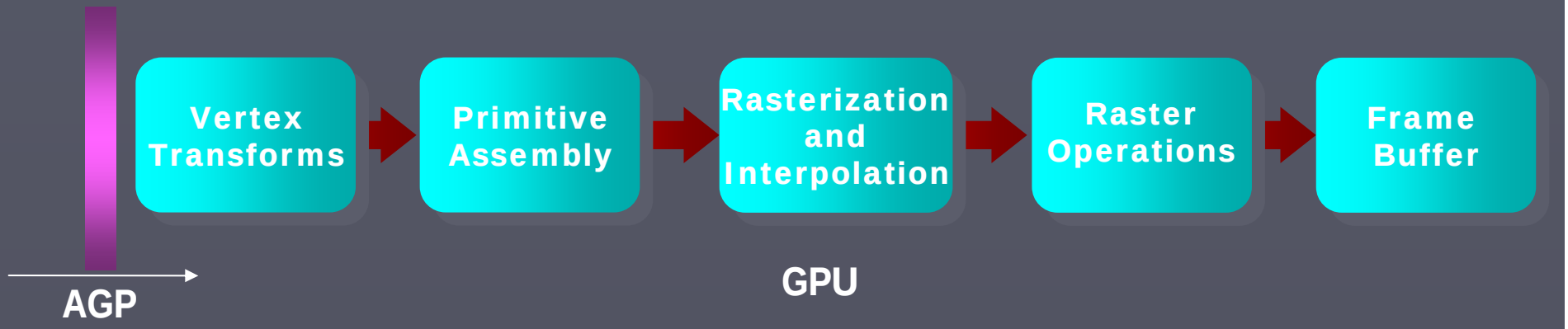
- One of the first true 3D game cards
- Worked by supplementing standard 2D video card.
- Did not do vertex transformations: these were done in the CPU
- Did do texture mapping, z-buffering.



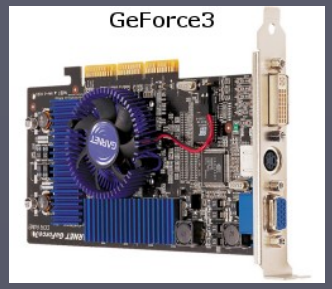
Generation II: GeForce/Radeon 7500 (1998)

http://accelenation.com/?ac.id.123.5

- Main innovation: shifting the transformation and lighting calculations to the GPU
- Allowed multi-texturing: giving bump maps, light maps, and others..
- Faster AGP bus instead of PCI

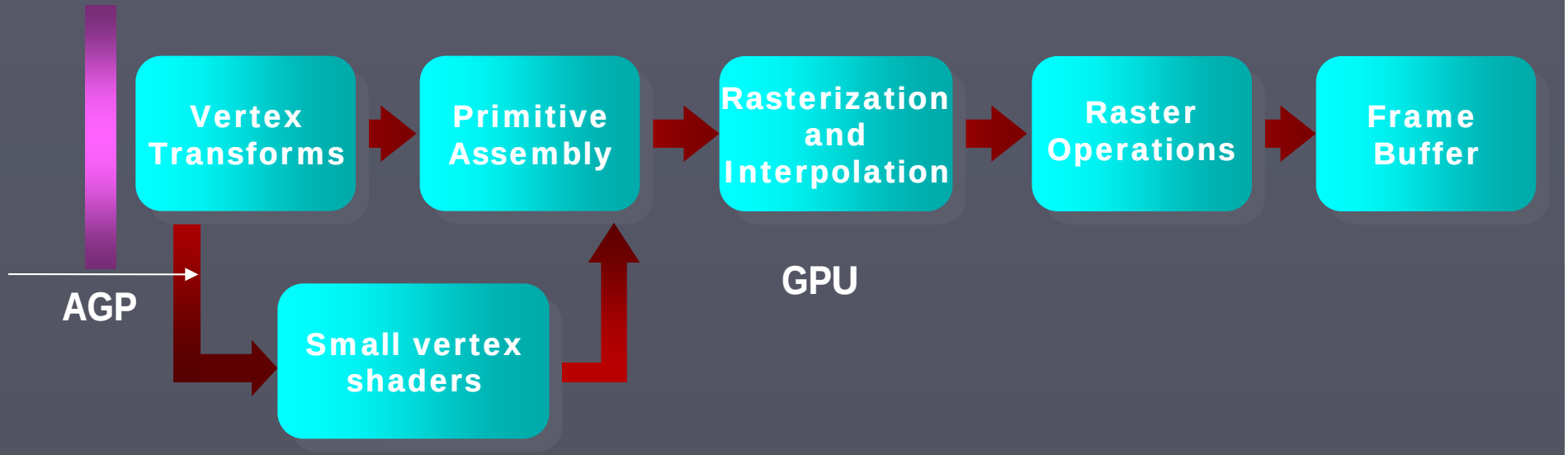


Generation III: GeForce3/Radeon 8500(2001)



http://accelenation.com/?ac.id.123.7

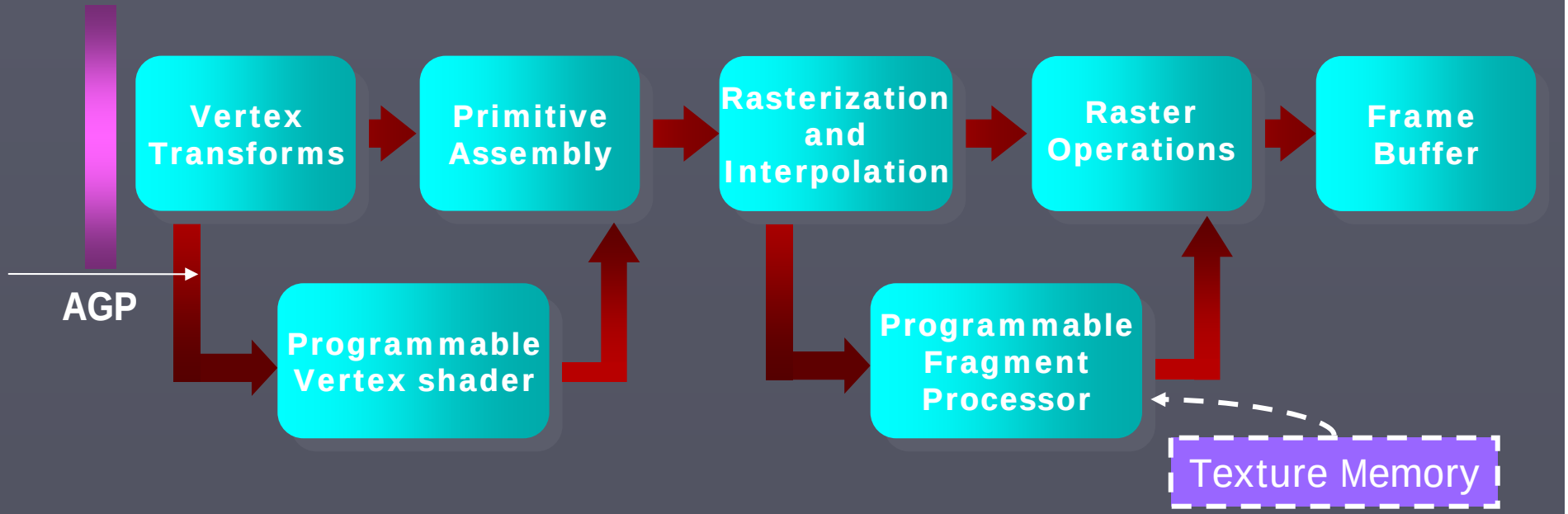
- For the first time, allowed limited amount of programmability in the vertex pipeline
- Also allowed volume texturing and multi-sampling (for antialiasing)



Generation IV: Radeon 9700/GeForce FX (2002)

http://accelenation.com/?ac.id.123.8

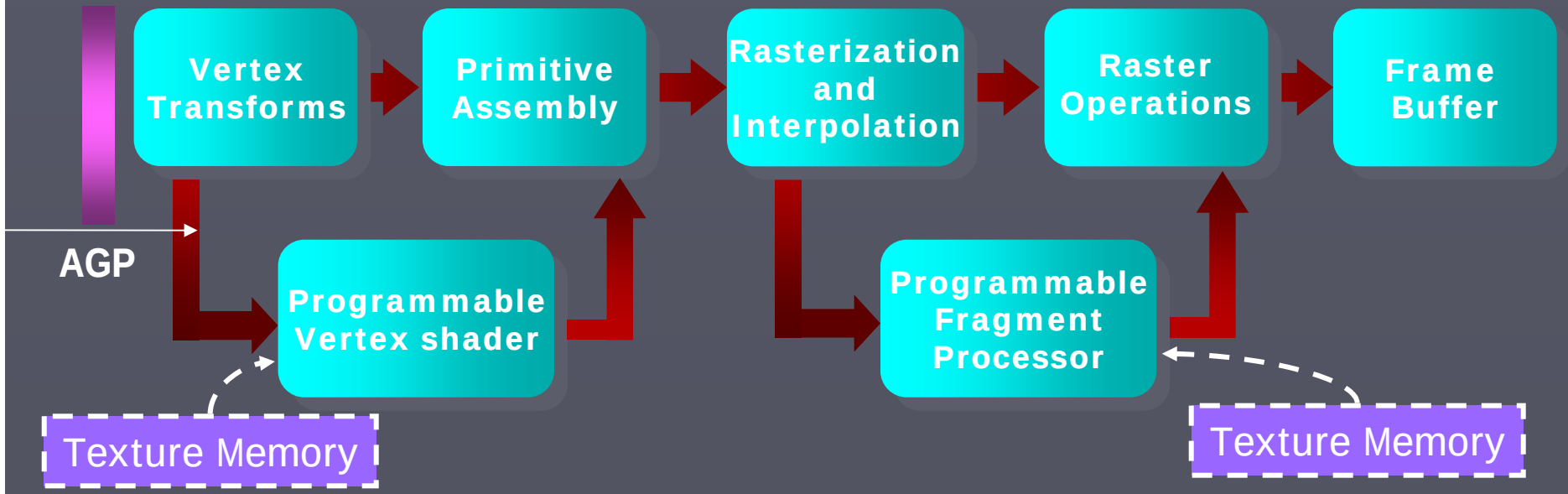
- This generation is the first generation of fully-programmable graphics cards
- Different versions have different resource limits on fragment/vertex programs



Generation IV.V: GeForce6/X800 (2004)

Not exactly a quantum leap, but…

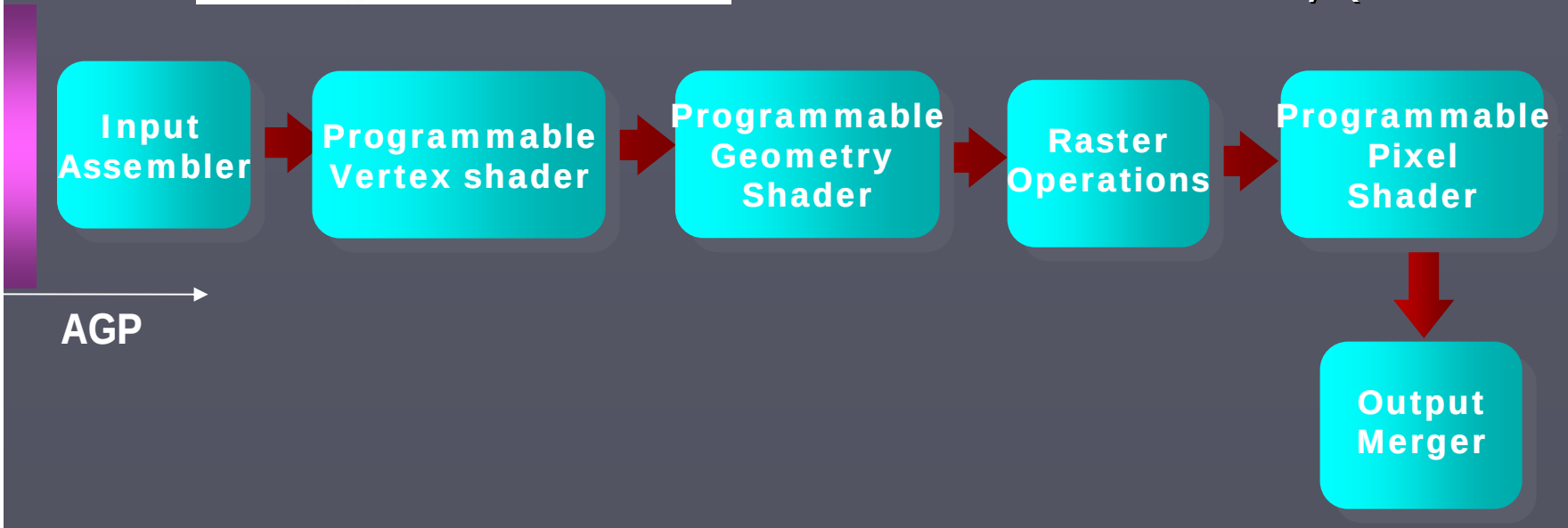
- Simultaneous rendering to multiple buffers
- True conditionals and loops
- Higher precision throughput in the pipeline (64 bits end-to-end, compared to 32 bits earlier.)
- PCIe bus
- More memory/program length/texture accesses
- Texture access by vertex shader

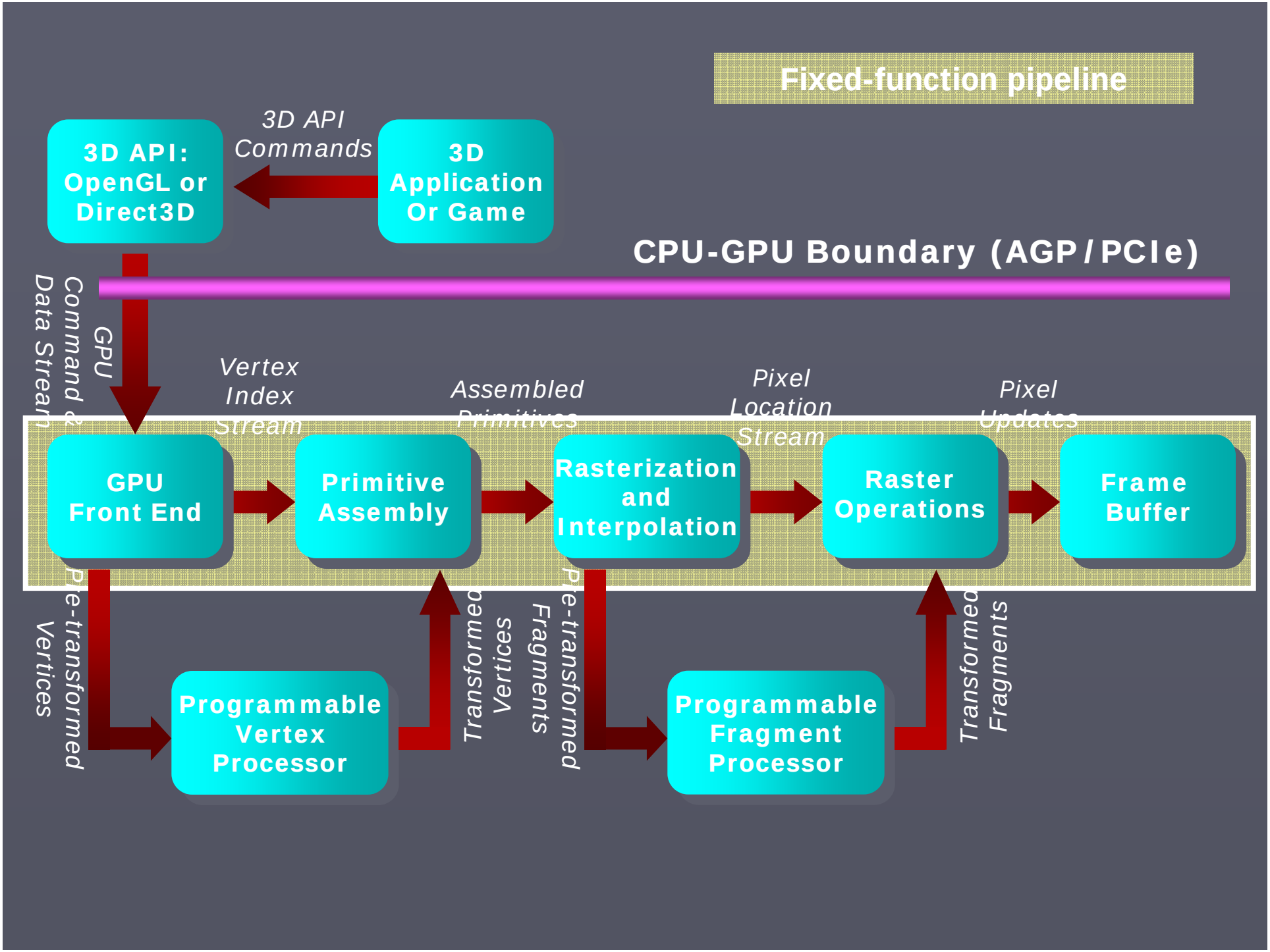


Generation V: GeForce8800/HD2900 (2006)

Complete quantum leap

- Ground-up rewrite of GPU
- Support for DirectX 10, and all it implies (more on this later)
- Geometry Shader
- Support for General GPU programming
- Shared Memory (NVIDIA only)




Fixed-function pipeline
3D API: OpenGL or Direct3D
3D API Commands
3D Application Or Game
CPU-GPU Boundary (AGP/PCIe)
GPU Command & Data Stream
Vertex Index Stream
Assembled Primitives
Pixel Location Stream
Pixel Updates
GPU Front End
Primitive Assembly
Rasterization and Interpolation
Raster Operations
Frame Buffer
Pre-transformed Vertices
Programmable Vertex Processor
Transformed Vertices
Pre-transformed Fragments
Programmable Fragment Processor
Transformed Fragments

A closer look at the fixed-function pipeline

Pipeline Input

Vertex

- (x, y, z)

(r, g, b,a)

(Nx,Ny,Nz)

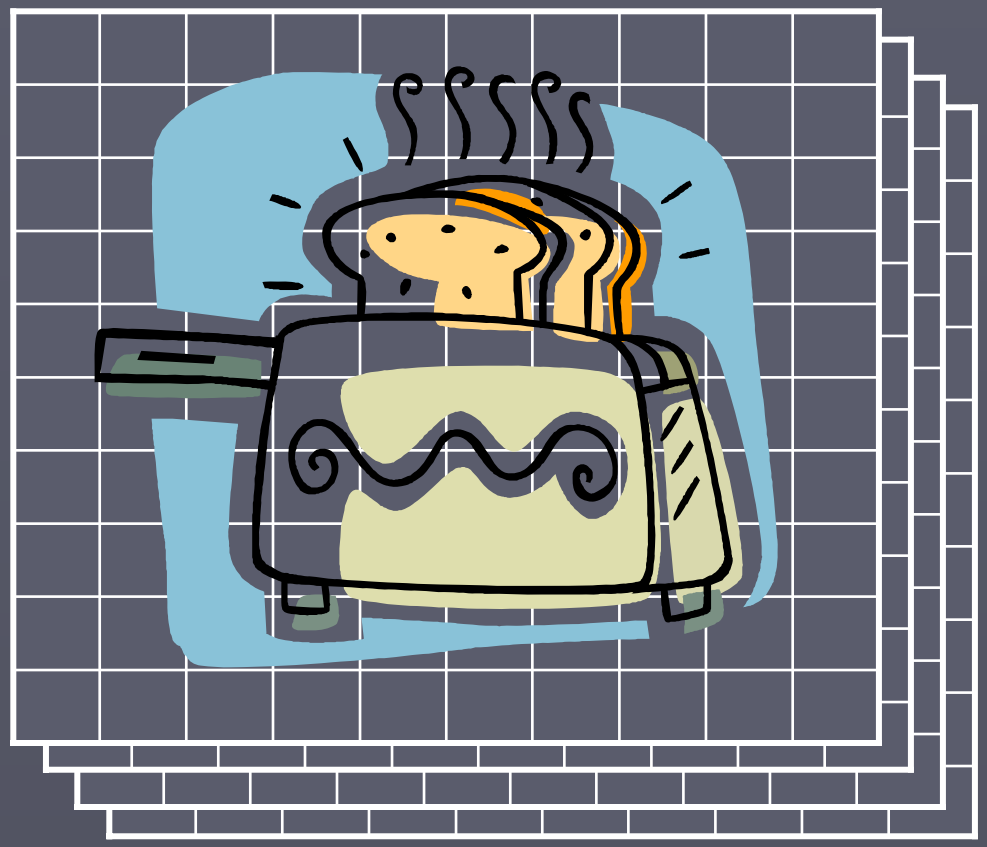
(tx, ty,[tz])

(tx, ty)

(tx, ty)

Material properties*

Image F(x,y) = (r,g,b,a)



ModelView Transformation

- Vertices mapped from object space to world space
- M = model transformation (scene)
- V = view transformation (camera)

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ W' \end{pmatrix} = M * V * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Each matrix transform is applied to each vertex in the input stream. Think of this as a kernel operator.

Lighting

Lighting information is combined with normals and other parameters at each vertex in order to create new colors.

Color(v) = *emissive* + *ambient* + *diffuse* + *specular*

Each term in the right hand side is a function of the vertex color, position, normal and material properties.

Clipping/Projection/Viewport(3D)

- More matrix transformations that operate on a vertex to transform it into the viewport space.
- Note that a vertex may be eliminated from the input stream (if it is clipped).
- The viewport is two-dimensional: however, vertex z-value is retained for depth testing.

Clip test is first example of a conditional in the pipeline.

However, it is not a fully general conditional. Why ?

Rasterizing+Interpolation

- All primitives are now converted to fragments.
- Data type change ! Vertices to fragments

Fragment attributes:

(r,g,b,a)

(x,y,z,w)

(tx,ty), …

Texture coordinates are interpolated from texture coordinates of vertices.

This gives us a linear interpolation operator for free. VERY USEFUL !

Per-fragment operations

- The rasterizer produces a stream of fragments.
- Each fragment undergoes a series of tests with increasing complexity.

Test 1: Scissor

If (fragment lies in fixed rectangle)
let it pass **else** discard it

Test 2: Alpha

If(fragment.a >= <constant>)
let it pass **else** discard it.

Scissor test is analogous to clipping operation in fragment space instead of vertex space.

Alpha test is a slightly more general conditional. Why ?

Per-fragment operations

- Stencil test: S(x, y) is stencil buffer value for fragment with coordinates (x,y)
- If f(S(x,y)), let pixel pass else kill it. Update S(x, y) conditionally depending on f(S(x,y)) and g(D(x,y)).
- Depth test: D(x, y) is depth buffer value.
- If g(D(x,y)) let pixel pass else kill it. Update D(x,y) conditionally.

Per-fragment operations

- Stencil and depth tests are more general conditionals. Why ?
- These are the only tests that can change the state of internal storage (stencil buffer, depth buffer).
- One of the update operations for the stencil buffer is a “count” operation. Remember this!
- Unfortunately, stencil and depth buffers have lower precision (8, 24 bits resp.)

Post-processing

- Blending: pixels are accumulated into final framebuffer storage

  new-val = old-val *op* pixel-value

  If *op* is +, we can sum all the (say) red components of pixels that pass all tests.

Problem: In generation<= IV, blending can only be done in 8-bit channels (the channels sent to the video card); precision is limited.

We could use accumulation buffers, but they are very slow.

Quick Review: Buffers

- Color Buffers
  - Front-left
  - Front-right
  - Back-left
  - Back-right
- Depth Buffer (z-buffer)
- Stencil Buffer
- Accumulation Buffer

Quick Review: Tests

- Scissor Test

  If(fragment exists inside rectangle)

  keep

  Else

  delete

- Alpha Test – Compare fragment's alpha value against reference value
- Stencil Test – Compare fragment against stencil map
- Depth Test – Compare a fragment's depth to the depth value already present in the depth buffer
  - Never
  - Always
  - Less
  - Less-Equal
  - Greater-Equal
  - Greater
  - Not-Equal

Readback = Feedback

What is the output of a "computation" ?

1. Display on screen.
2. Render to buffer and retrieve values (readback)

Readbacks are VERY slow !

PCI and AGP buses are asymmetric: DMA enables fast transfer TO graphics card. Reverse transfer has traditionally not been required, and is much slower. PCIe is symmetric but still very slow compared to GPU speeds.

This motivates idea of "pass" being an atomic "unit cost" operation.

What options do we have ?

1. Render to off-screen buffers like accumulation buffer
2. Copy from framebuffer to texture memory ?
3. Render directly to a texture ?

Time for a puzzle...

An Example: Voronoi Diagrams.

Definition

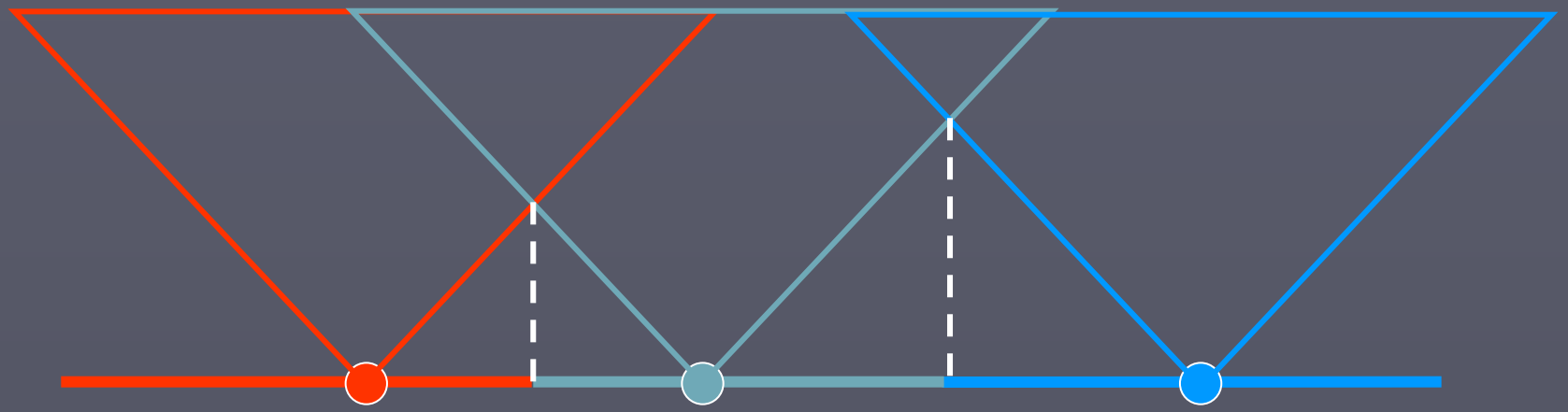
- You are given n sites (p_1, p_2, p_3, ... p_n) in the plane (think of each site as having a color)
- For any point p in the plane, it is closest to some site p_j. Color p with color i.
- Compute this colored map on the plane. In other words,

  Compute the nearest-neighbour diagram of the sites.

Example

So how do we do this on the graphics card?
Note, this does not use any programmable features of the card

Hint: Think in one dimension higher



The lower envelope of “cones” centered at the points is the Voronoi diagram of this set of points.

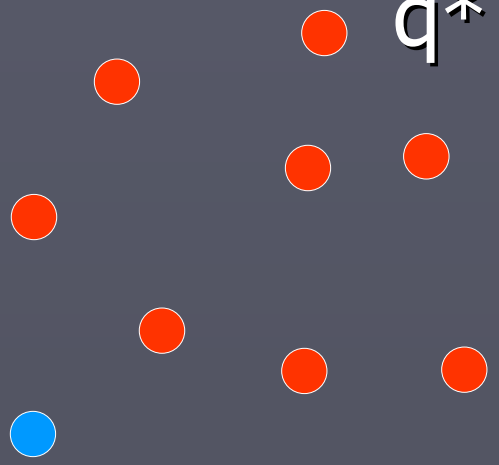
The Procedure

- In order to compute the lower envelope, we need to determine, at each pixel, the fragment having the smallest depth value.
- This can be done with a simple depth test.
  - Allow a fragment to pass only if it is smaller than the current depth buffer value, and update the buffer accordingly.
- The fragment that survives has the correct color.

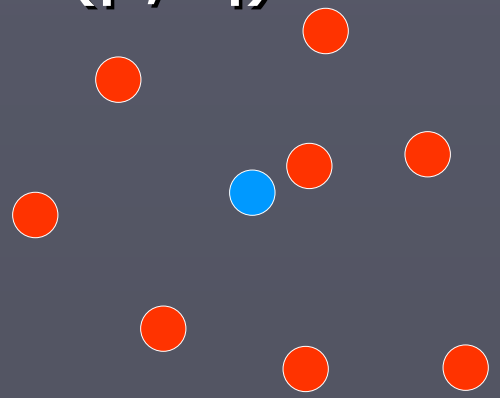
Let's make this more complicated

- The 1-median of a set of sites is a point q* that minimizes the sum of distances from all sites to itself.

$$q^* = \arg\min \Sigma\, d(p, q)$$



WRONG !



RIGHT !

A First Step

Can we compute, for each pixel q, the value

$$F(q) = \Sigma\, d(p, q)$$

We can use the cone trick from before, and instead of computing the minimum depth value, compute the sum of all depth values using blending.

What's the catch ?

We can’t blend depth values !

- Using texture interpolation helps here.
- Instead of drawing a single cone, we draw a shaded cone, with an appropriately constructed texture map.
- Then, fragment having depth z has color component 1.0 * z.
- Now we can blend the colors.
- OpenGL has an aggregation operator that will return the overall min

Warning: we are ignoring issues of precision.

Now we apply a streaming perspective…

Two kinds of data

- Stream data (data associated with vertices and fragments)
  - Color/position/texture coordinates.
  - Functionally similar to member variables in a C++ object.
  - Can be used for limited message passing: I modify an object state and send it to you.
- "Persistent" data (associated with buffers).
  - Depth, stencil, textures.
- Can be modifed by multiple fragments in a single pass.
- Functionally similar to a global array BUT each fragment only gets one location to change.
- Can be used to communicate across passes.

Who has access ?

- Memory "connectivity" in the graphics use of a GPU is tricky.
- In a traditional C program, all global variables can be written by all routines.
- In the fixed-function pipeline, certain data is private.
  - A fragment cannot change a depth or stencil value of a location different from its own.
  - The framebuffer can be copied to a texture; a depth buffer cannot be copied in this way, and neither can a stencil buffer.
  - Only a stencil buffer can count (efficiently)
- In the fixed-function pipeline, depth and stencil buffers can be used in a multi-pass computation only via readbacks.
- A texture cannot be written directly.
- In programmable GPUs, the memory connectivity becomes more open, but there are still constraints.

Understanding access constraints and memory "connectivity" is a key step in programming the GPU.

How does this relate to stream programs ?

- The most important question to ask when programming the GPU is:

  What can I do in one pass ?

- Limitations on memory connectivity mean that a step in a computation may often have to be deferred to a new pass.
- For example, when computing the second smallest element, we could not store the current minimum in read/write memory.
- Thus, the “communication” of this value has to happen across a pass.

Graphics pipeline