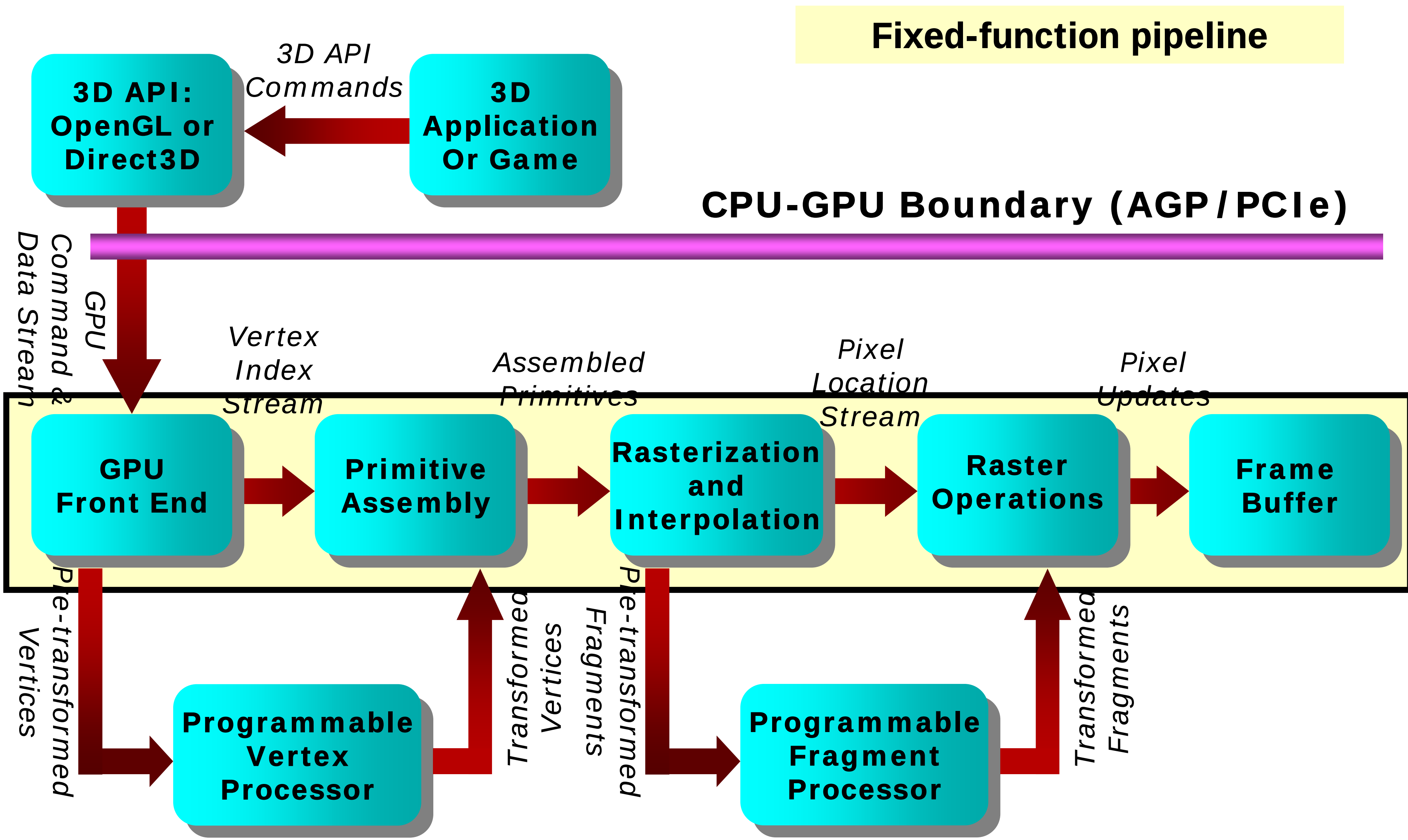
GPU Memory Model

Adapted from:

Aaron Lefohn
University of California, Davis
With updates from slides by
Suresh Venkatasubramanian,
University of Pennsylvania
Updates performed by Gary J. Katz,
University of Pennsylvania

Review

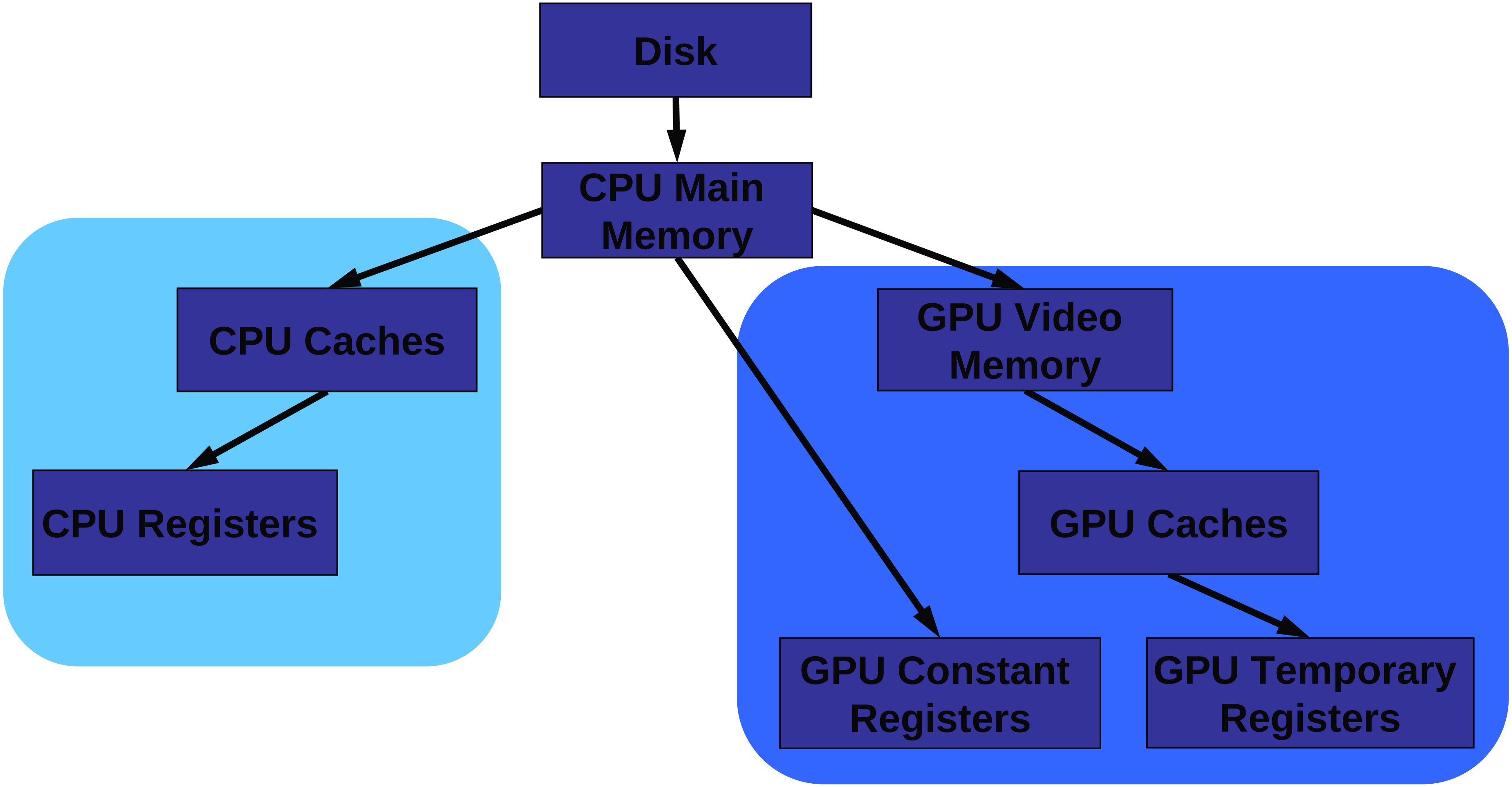

Fixed-function pipeline
3D API Commands
3D API: OpenGL or Direct3D
3D Application Or Game
CPU-GPU Boundary (AGP/PCIe)
GPU Command & Data Stream
Vertex Index Stream
Assembled Primitives
Pixel Location Stream
Pixel Updates
GPU Front End
Primitive Assembly
Rasterization and Interpolation
Raster Operations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

Overview

- GPU Memory Model
- GPU Data Structure Basics
- Introduction to Framebuffer Objects
- Fragment Pipeline
- Vertex Pipeline

Memory Hierarchy

- **CPU and GPU Memory Hierarchy**



CPU Memory Model

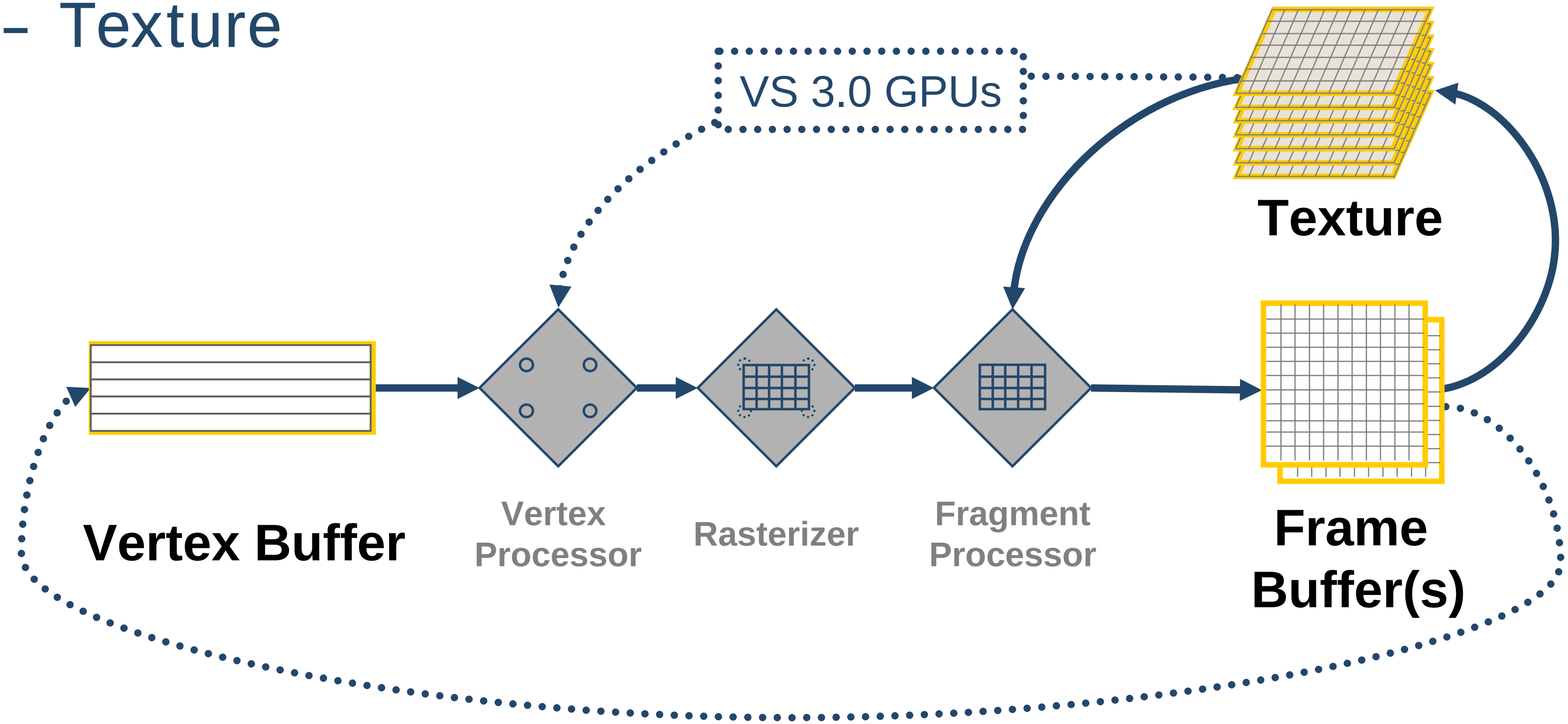
- **At any program point**
  - Allocate/free local or global memory
  - Random memory access
    - Registers
      - Read/write
    - Local memory
      - Read/write to stack
    - Global memory
      - Read/write to heap
    - Disk
      - Read/write to disk

GPU Memory Model

- **Much more restricted memory access**
  - Allocate/free memory only before computation
  - Limited memory access during computation (kernel)
    - Registers
      - Read/write
    - Local memory
      - Does not exist
    - Global memory
      - Read-only during computation
      - Write-only at end of computation (pre-computed address)
    - Disk access
      - Does not exist

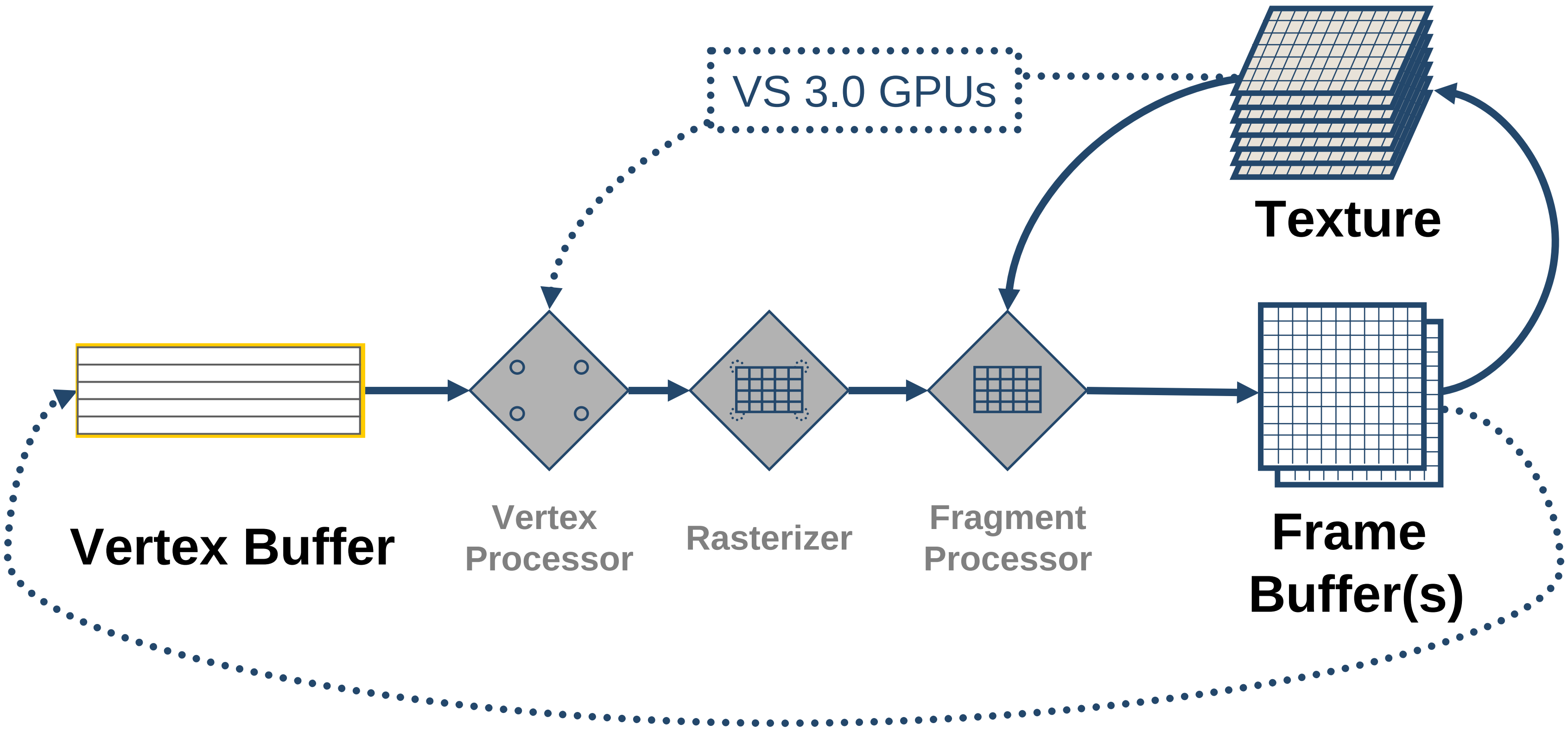
GPU Memory Model

- **Where is GPU Data Stored?**
  - Vertex buffer
  - Frame buffer
  - Texture



Vertex Buffers

- GPU memory for vertex data
- Vertex data required to initiate render pass



Vertex Buffers

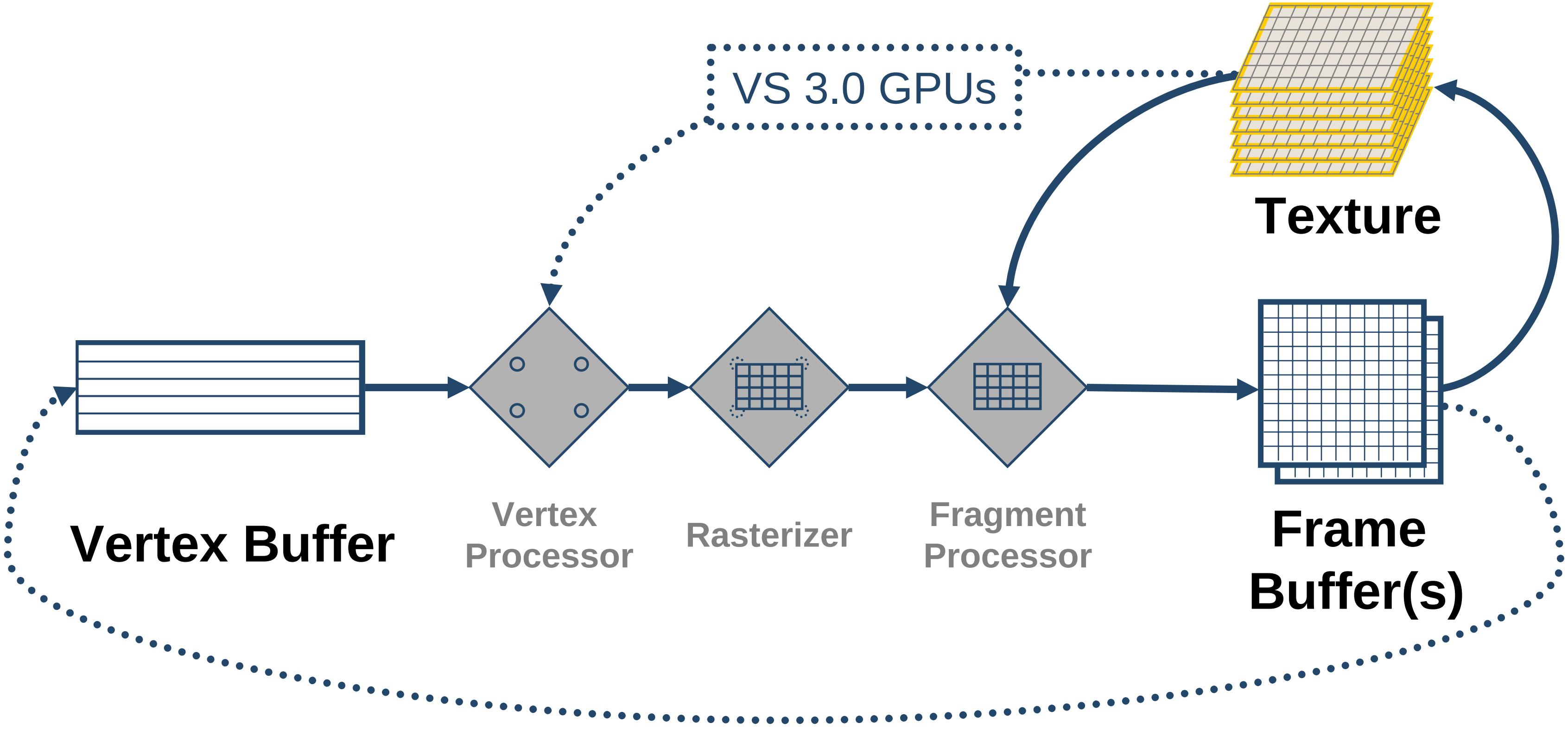
- **Supported Operations**
  - CPU interface
    - Allocate
    - Free
    - Copy CPU → GPU
    - Copy GPU → GPU (Render-to-vertex-array)
    - Bind for read-only vertex stream access
  - GPU interface
    - Stream read (vertex program only)

Vertex Buffers

- **Limitations**
  - CPU
    - No copy GPU → CPU
    - No bind for read-only random access
  - GPU
    - No random-access reads
    - No access from fragment programs

Textures

- Random-access GPU memory



Textures

- **Supported Operations**
  - CPU interface
    - Allocate
    - Free
    - Copy CPU → GPU
    - Copy GPU → CPU
    - Copy GPU → GPU (Render-to-texture)
    - Bind for read-only random access (vertex or fragment)
  - GPU interface
    - Random read

Textures

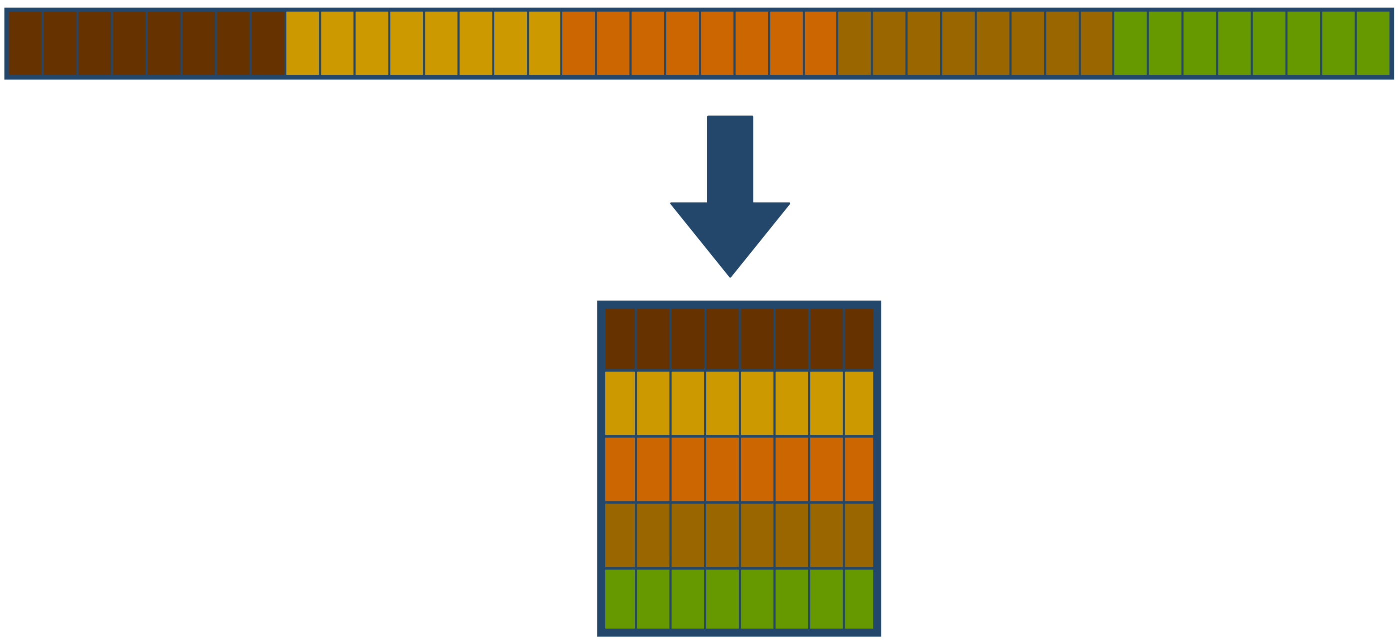
- **Limitations**
  - No bind for vertex stream access

Next …

- **GPU Data Structure Basics**

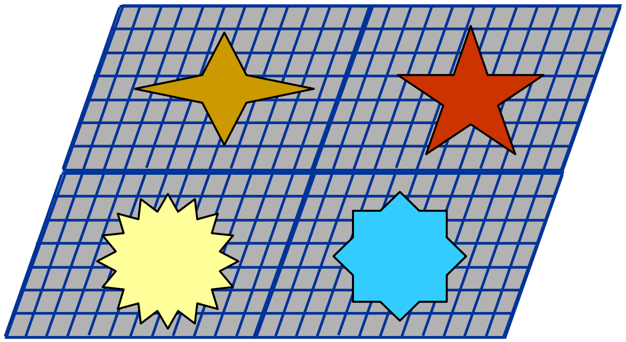
GPU Arrays

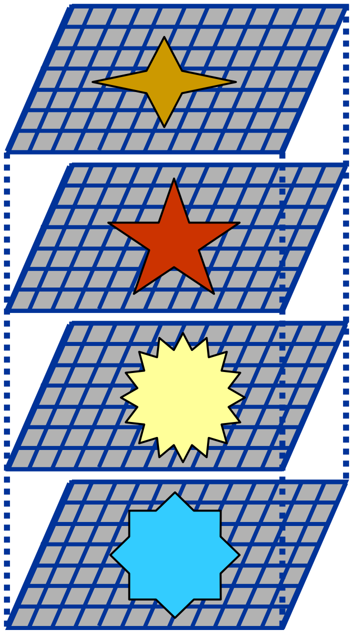
- **Large 1D Arrays**
  - Current GPUs limit 1D array sizes
  - Pack into 2D memory
  - 1D-to-2D address translation



GPU Arrays

- **3D Arrays**
  - Problem
    - GPUs do not have 3D frame buffers
  - Solutions
    1. Stack of 2D slices
    2. Multiple slices per 2D buffer





GPU Arrays

- **Problems With 3D Arrays**
  - Cannot read stack of 2D slices as 3D texture
  - Must know which slices are needed in advance
- **Solutions**
  - Flat 3D textures
  - Need packing functions for coordinate transfer

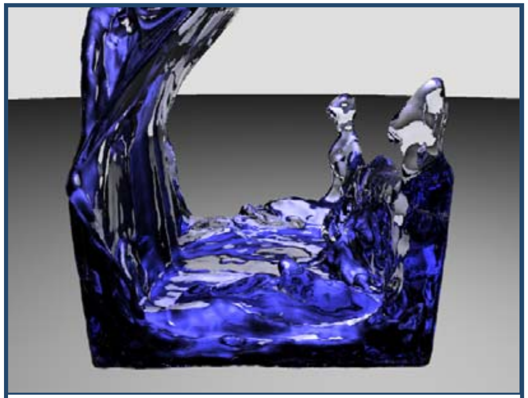
GPU Arrays

- **Higher Dimensional Arrays**
  - Pack into 2D buffers
  - N-D to 2D address translation
  - Same problems as 3D arrays if data does not fit in a single 2D texture

Sparse/Adaptive Data Structures

- **Why?**
  - Reduce memory pressure
  - Reduce computational workload

- **Examples**
  - Sparse matrices
    - Krueger et al., Siggraph 2003
    - Bolz et al., Siggraph 2003



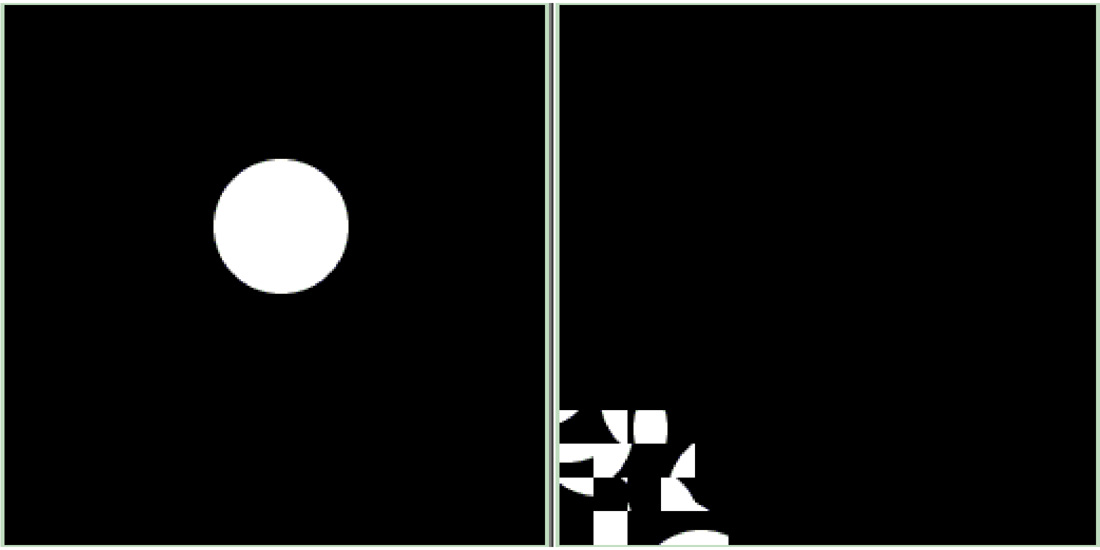
Premoze et al.
Eurographics 2003

  - Deformable implicit surfaces (sparse volumes/PDEs)
    - Lefohn et al., IEEE Visualization 2003 / TVCG 2004

  - Adaptive radiosity solution (Coombe et al.)

Sparse/Adaptive Data Structures

- **Basic Idea**
  - Pack "active" data elements into GPU memory

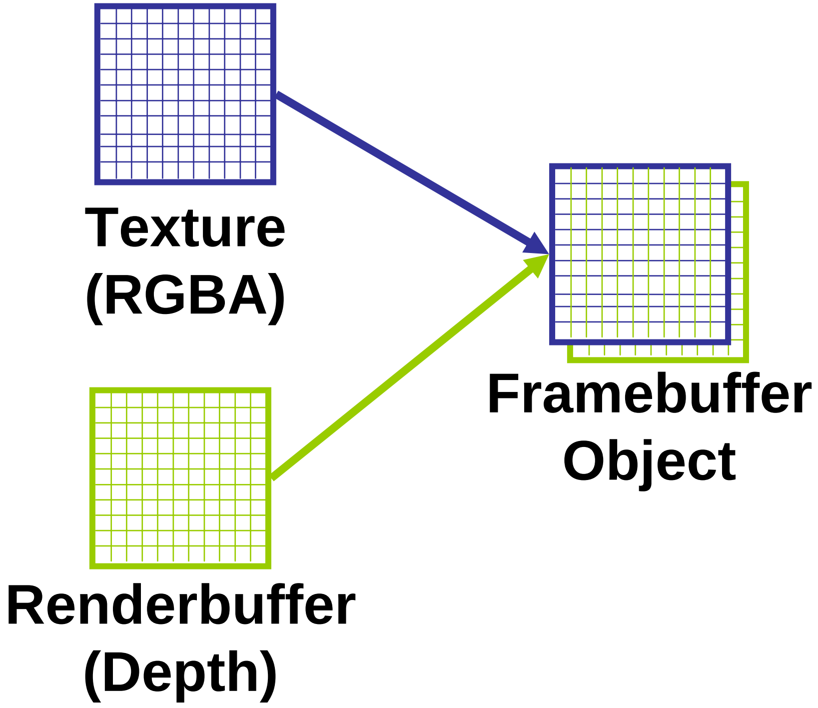


GPU Data Structures

- **Conclusions**
  - Fundamental GPU memory primitive is a fixed-size 2D array
  - GPGPU needs more general memory model
  - Building and modifying complex GPU-based data structures is an open research topic…

Framebuffer Objects

- **What is an FBO?**
  - A struct that holds pointers to memory objects
  - Each bound memory object can be a framebuffer rendering surface
  - Platform-independent



Framebuffer Objects

- **Which memory can be bound to an FBO?**
  - Textures
  - Renderbuffers
    - Depth, stencil, color
    - Traditional write-only framebuffer

The fragment pipeline

Input: Fragment

Attributes

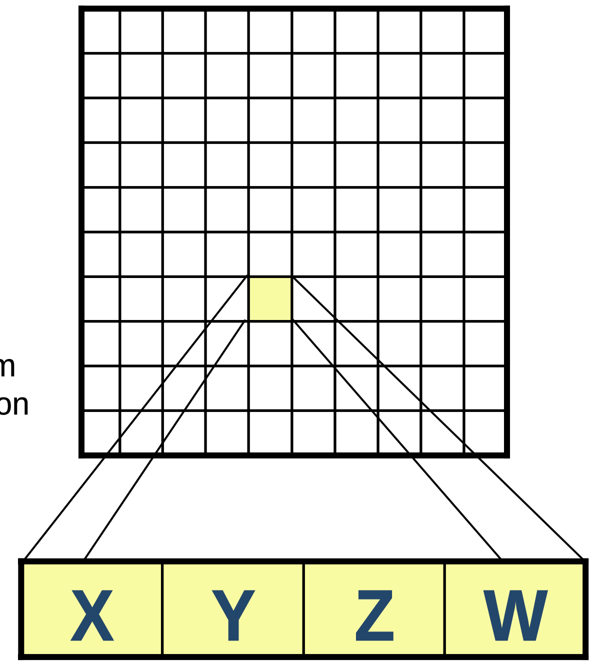
Color	R	G	B	A
Position	X	Y	Z	W
Texture coordinates	X	Y	[Z]	-
Texture coordinates	X	Y	[Z]	-
…				

Interpolated from vertex information

32 bits = float

16 bits = half

Input: Texture Image



- Each element of texture is 4D vector
- Textures can be “square” or rectangular (power-of-two or not)

The fragment pipeline

Input: Uniform parameters

- Can be passed to a fragment program like normal parameters
- set in advance before the fragment program executes

Example:

A counter that tracks which pass the algorithm is in.

Input: Constant parameters

- Fixed inside program
- E.g. float4 v = (1.0, 1.0, 1.0, 1.0)

Examples:

3.14159..

Size of compute window

The fragment pipeline

Math ops: USE THEM !

- cos(x)/log2(x)/pow(x,y)
- dot(a,b)
- mul(v, M)
- sqrt(x)
- cross(u, v)

Using built-in ops is more efficient than writing your own

Swizzling/masking: an easy way to move data around.

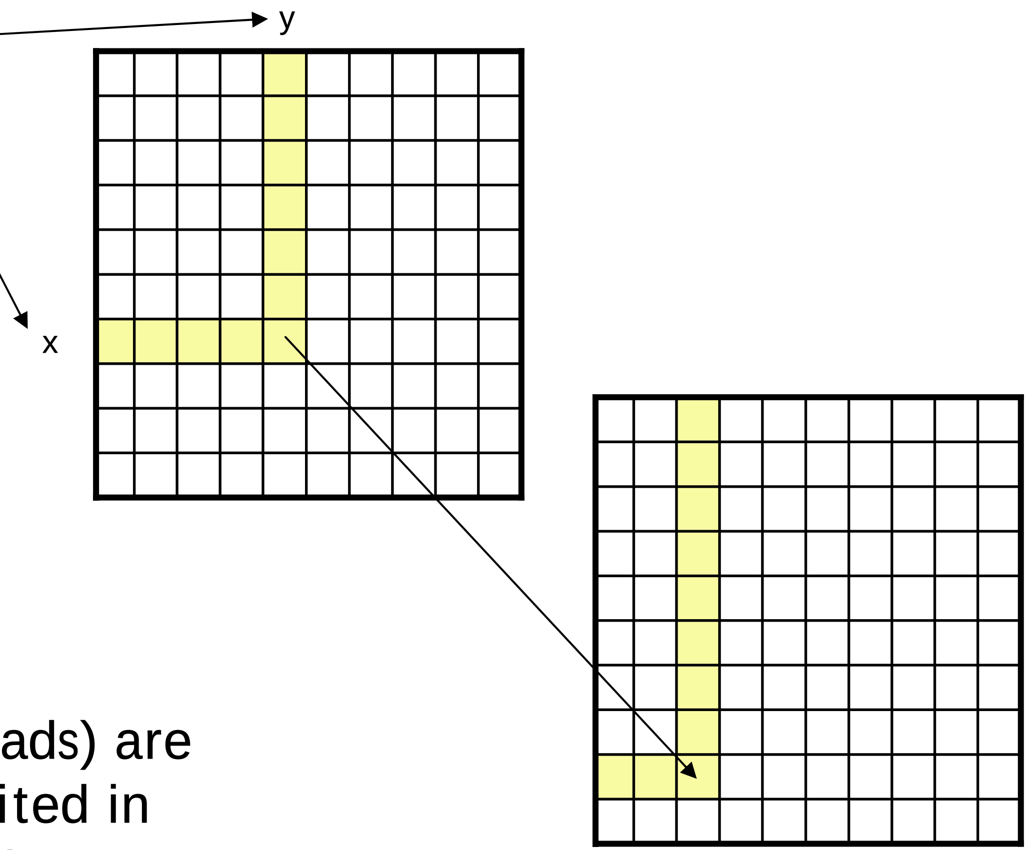
```
v1 = (4,-2,5,3); // Initialize
v2 = v1.yx;          // v2 = (-2,4)
s = v1.w;            // s = 3
v3 = s.rrr;          // v3 = (3,3,3)
```

Write masking:

```
v4 = (1,5,3,2);
v4.ar = v2;          // v4=(4,5,4,-2)
```

The fragment pipeline

`float4 v = tex2D(IMG, float2(x,y))`



Texture access is like an array lookup.

The value in v can be used

to perform another lookup!

This is called a dependent read

Texture reads (and dependent reads) are expensive resources, and are limited in different GPUs. Use them wisely !

The fragment pipeline

Control flow:

- **(<test>)?a:b operator.**
- **if-then-else conditional**
  - **[nv3x]** Both branches are executed, and the condition code is used to decide which value is used to write the output register.
  - **[nv40]** True conditionals
- **for-loops and do-while**
  - **[nv3x]** limited to what can be unrolled (i.e no variable loop limits)
  - **[nv40]** True looping.

The fragment pipeline

Fragment programs use call-by-result

```
out float4 result : COLOR
// Do computation
result = <final answer>
```

Notes:

- **Only output color can be modified**
- **Textures cannot be written**
- **Setting different values in different channels of result can be useful for debugging**

The Vertex Pipeline

Input: vertices

- position, color, texture coords.

Input: uniform and constant parameters.

- Matrices can be passed to a vertex program.
- Lighting/material parameters can also be passed.

The Vertex Pipeline

Operations:

- **Math/swizzle ops**
- **Matrix operators**
- **Flow control (as before)**

[nv3x] **No access to textures.**

Output:

- **Modified vertices (position, color)**
- **Vertex data transmitted to primitive assembly.**

Vertex programs are useful

- We can replace the entire geometry transformation portion of the fixed-function pipeline.
- Vertex programs used to change vertex coordinates (move objects around)
- There are many fewer vertices than fragments: shifting operations to vertex programs improves overall pipeline performance.
- Much of shader processing happens at vertex level.
- We have access to original scene geometry.

Vertex programs are not useful

- **Fragment programs allow us to exploit full parallelism of GPU pipeline ("a processor at every pixel").**

Rule of thumb:
If computation requires intensive calculation,
it should probably be in the fragment processor.

If it requires more geometric/graphic computing,
it should be in the vertex processor.

Conclusions

- **GPU Memory Model Evolving**
  - Writable GPU memory forms loop-back in an otherwise feed-forward pipeline
  - Memory model will continue to evolve as GPUs become more general data-parallel processors
- **Data Structures**
  - Basic memory primitive is limited-size, 2D texture
  - Use address translation to fit all array dimensions into 2D