CSE 690: GPGPU

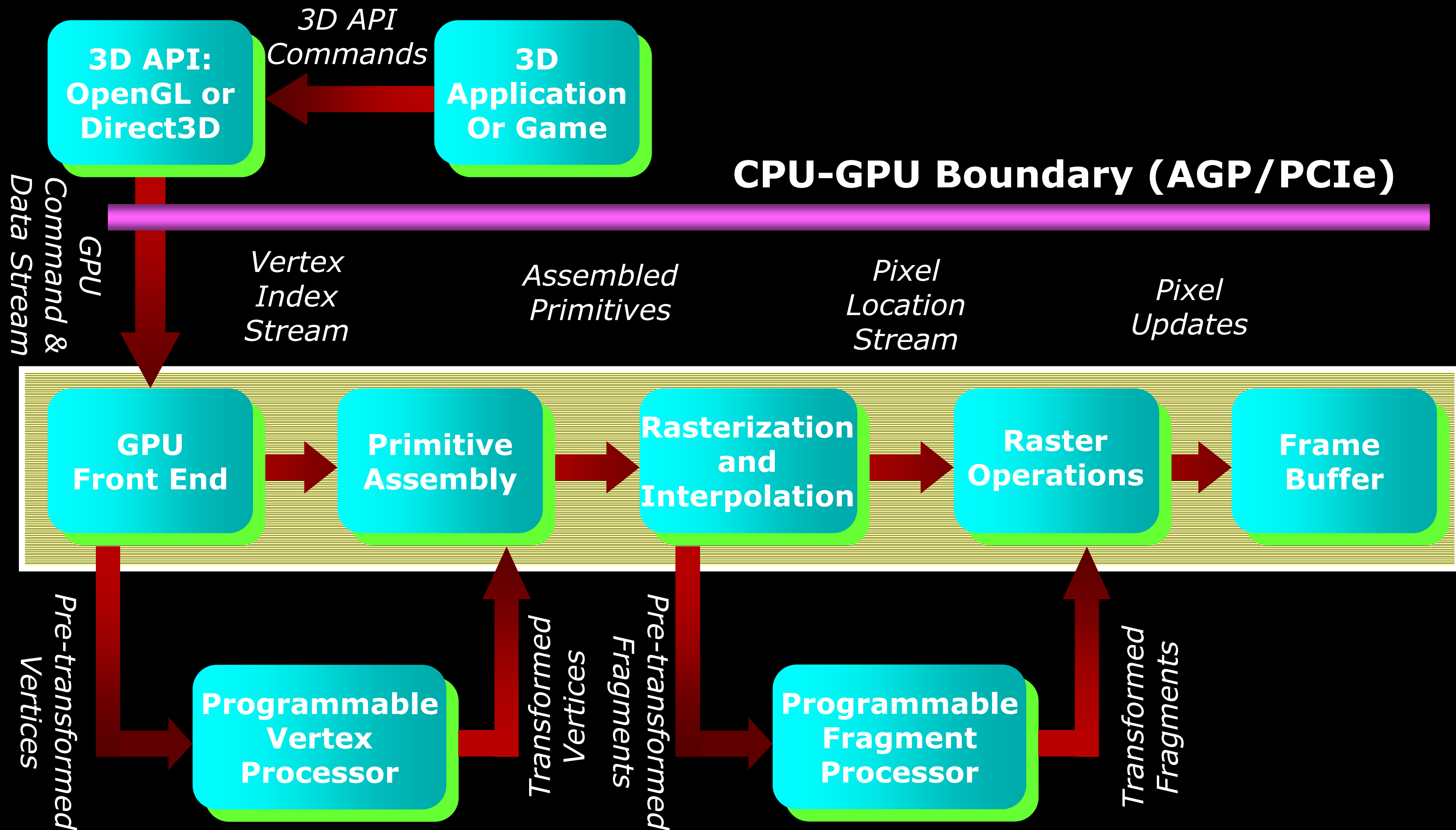
Lecture 5: The Graphics Pipeline

Klaus Mueller

Computer Science, Stony Brook University

adapted from: Suresh Venkatasubramanian, U Penn

The Fixed Function Pipeline



Pipeline Input

Vertex

(x, y, z)

(r, g, b,a)

(Nx,Ny,Nz)

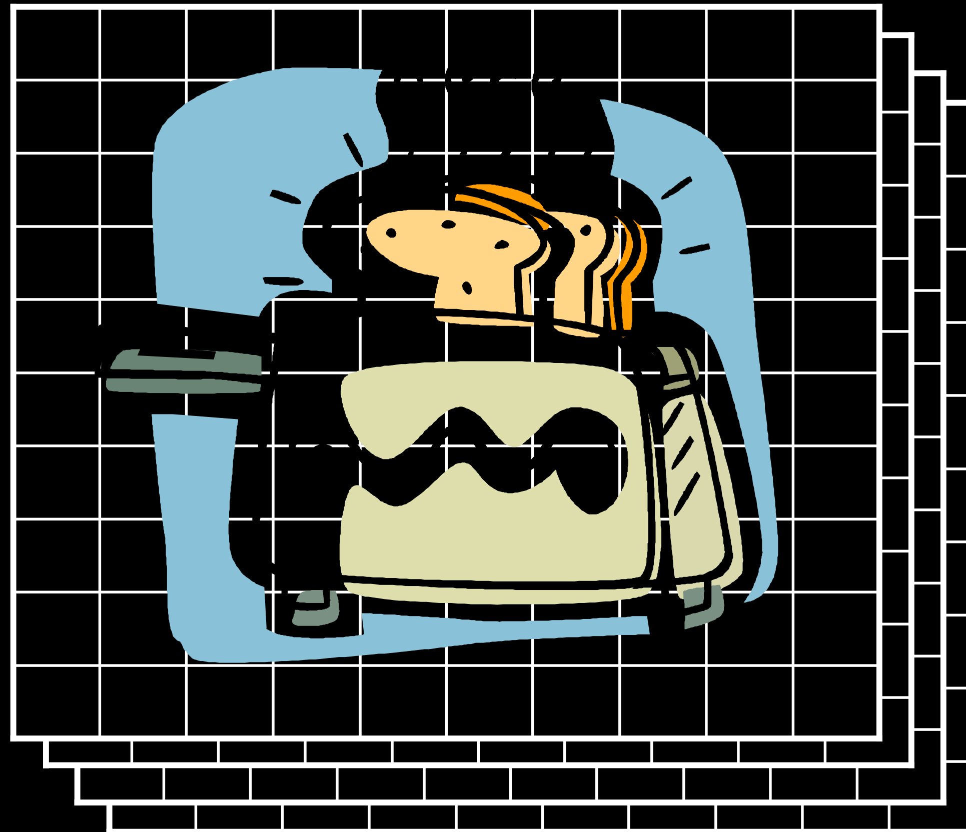
(tx, ty,[tz])

(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.

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 !

F(x, y) = (lo * x + range, lo’ * y + range’)

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 very useful. It is a slightly more general conditional.

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.
- These are the only tests that can change the state of internal storage (stencil buffer, depth buffer). This is very important.
- 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.

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 has changed this.)

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 ?

Stay tuned…

An Example: Voronoi Diagrams.

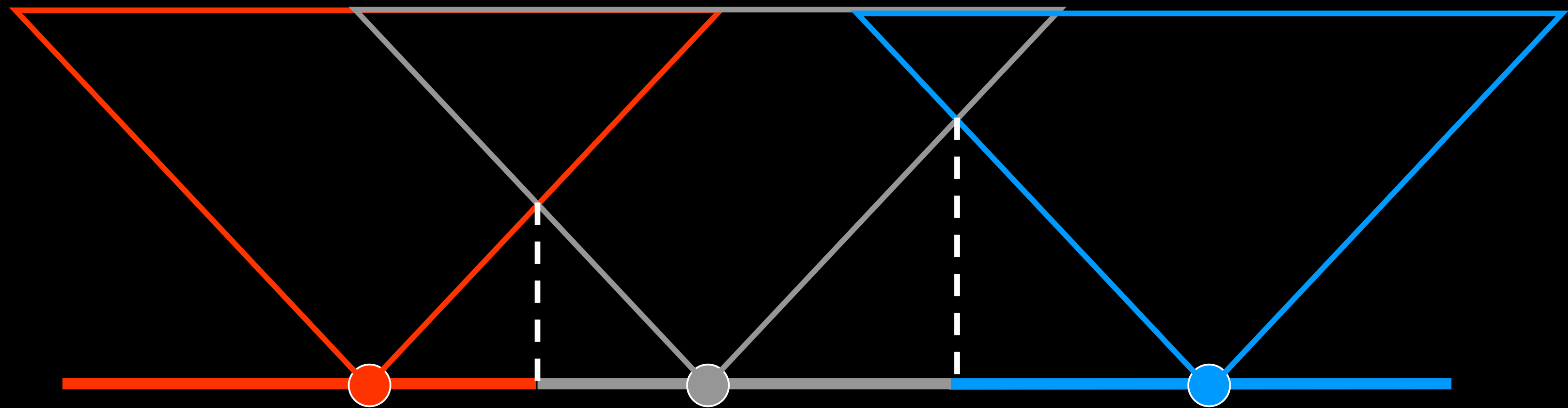
Definition

- 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

Hint: Think in one dimension higher



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

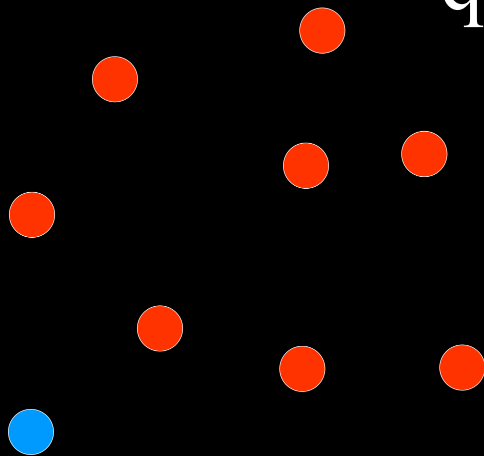
The Procedure

- 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.

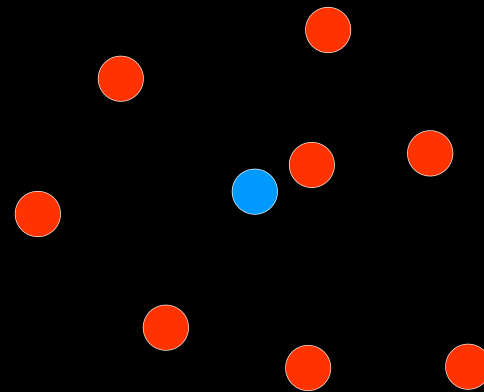
Let's make this more complicated

- 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.
  - This is how hardware shadow mapping can be done (using the alpha-channel)
- "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 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.

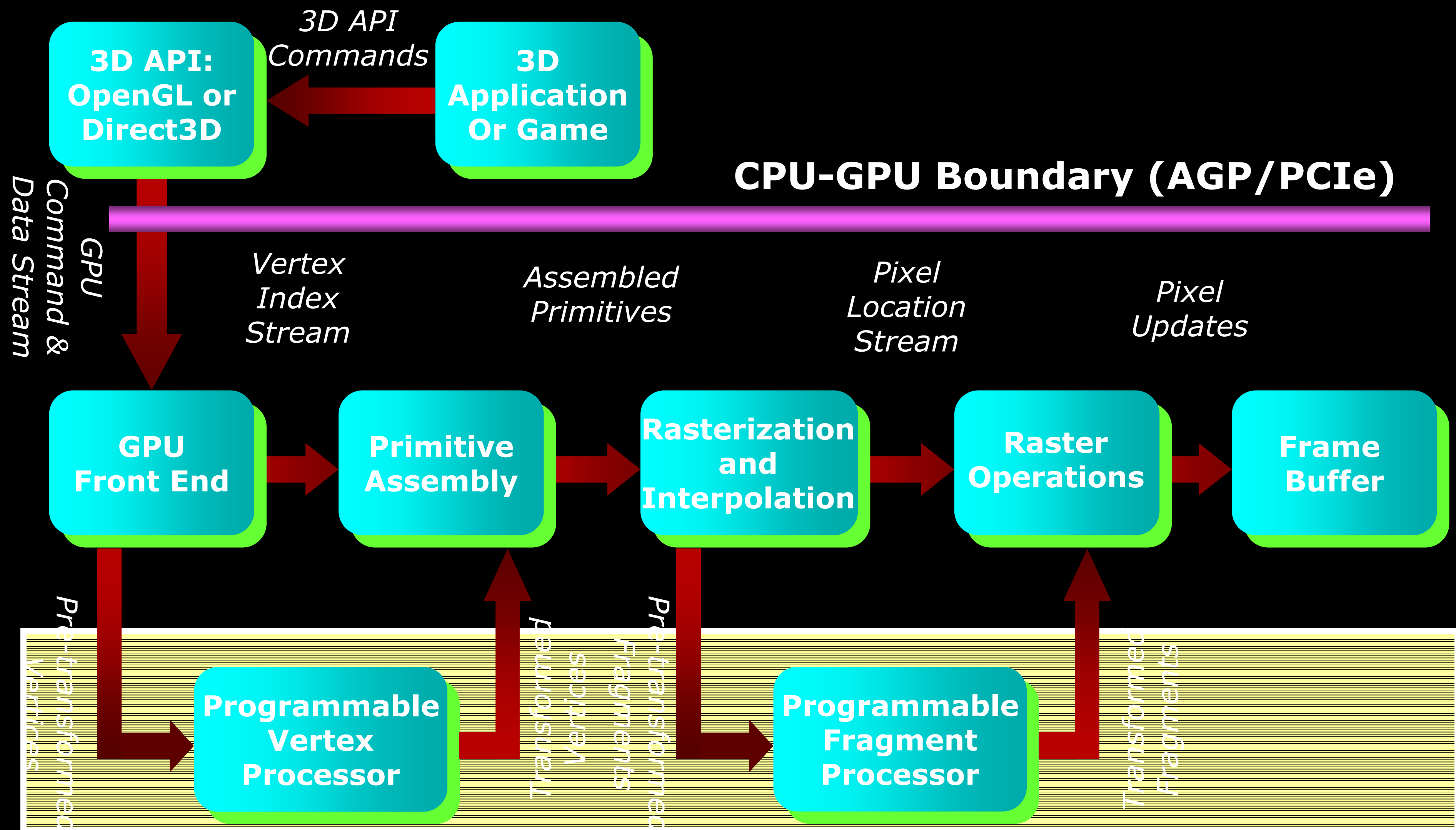
How does this relate to stream programs?

- 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.

The Programmable Pipeline



The fragment pipeline

Input: Fragment

Attributes

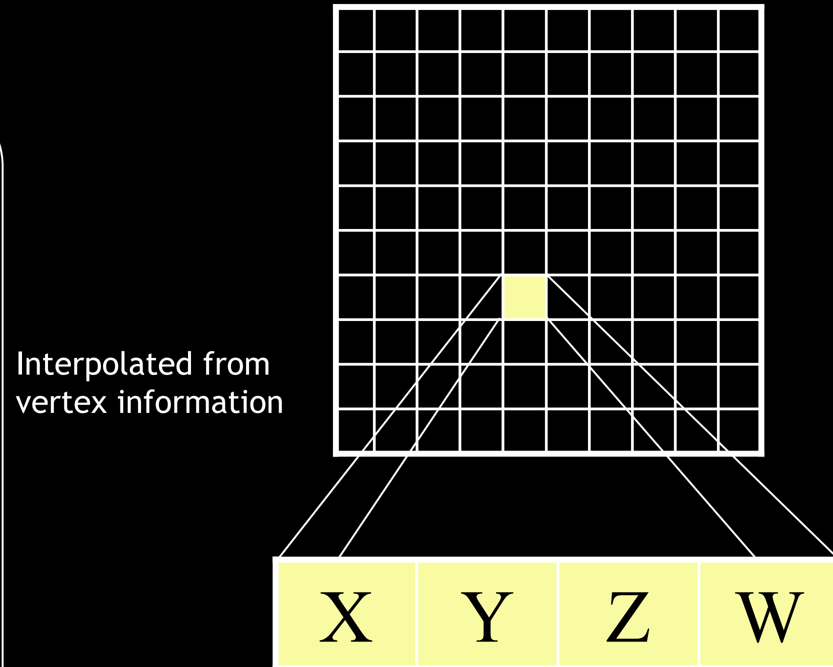
Color	R	G	B	A
Position	X	Y	Z	W
Texture coordinates	X	Y	[Z]	-
Texture coordinates	X	Y	[Z]	-
…				

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)/rsqrt(x)
- cross(u, v)

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

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

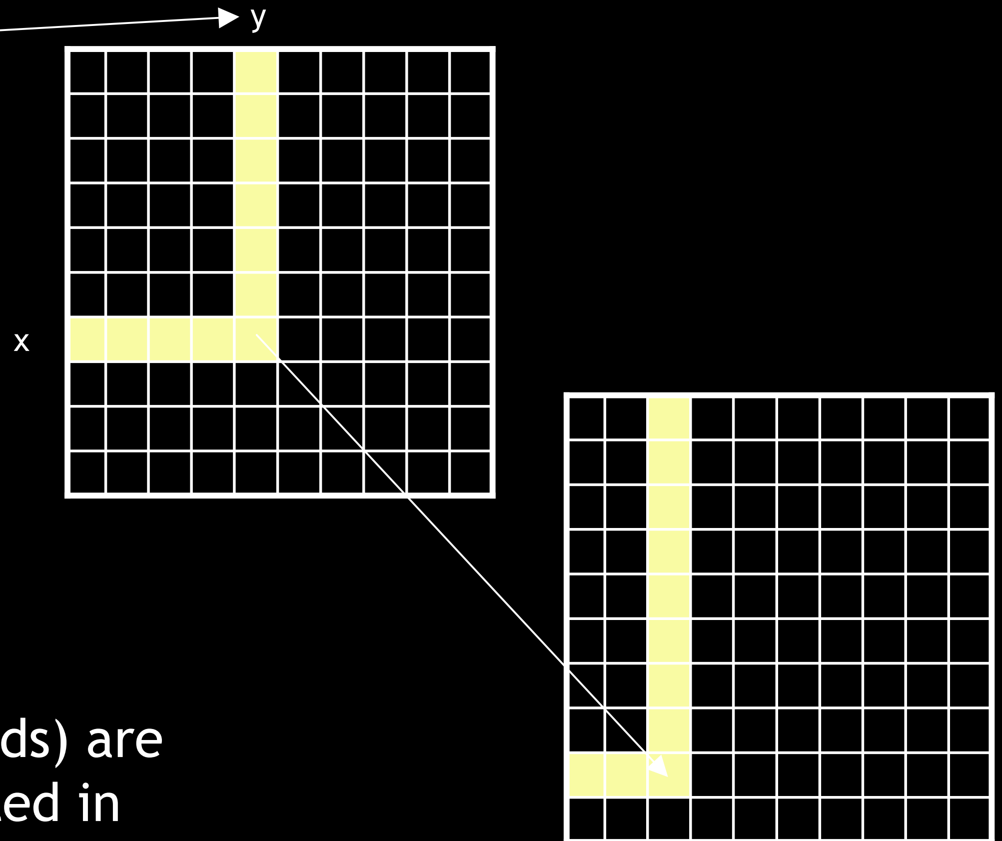
```
v1 = (4,-2,5,3); // Initialize
v2 = v1.yx;      // v2 = (-2,4)
s = v1.w;        // s = 3
v3 = s.rrr;      // v3 = (3,3,3)
```

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.

WARNING: Even though nv40 has true flow control, performance will suffer if there is no coherence (more on this later)

The fragment pipeline

Fragment programs use call-by-result

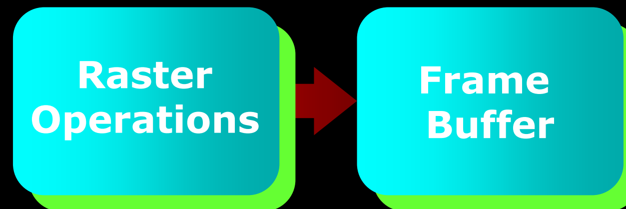
```
out float4 result : COLOR
// Do computation
result = <final answer>
```

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 fragment pipeline

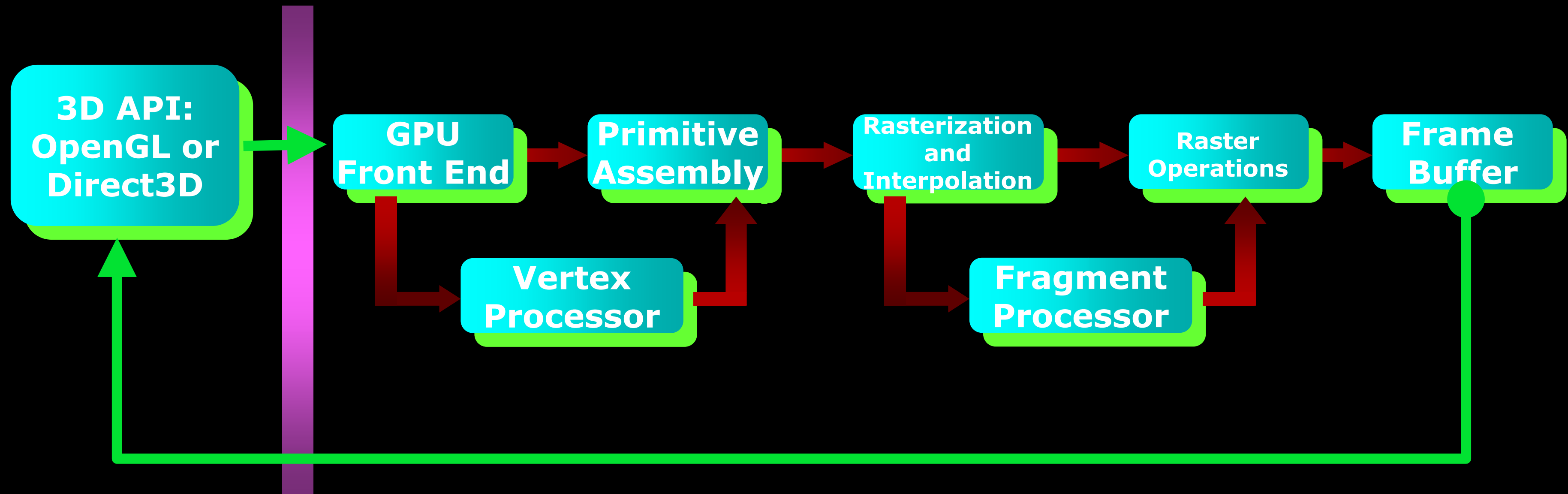
What comes after fragment programs ?



- Depth/stencil happen after frag. program
- Blending and aggregation happen as usual
- Early z-culling: fragments that would have failed depth test are killed before executing fragment program.

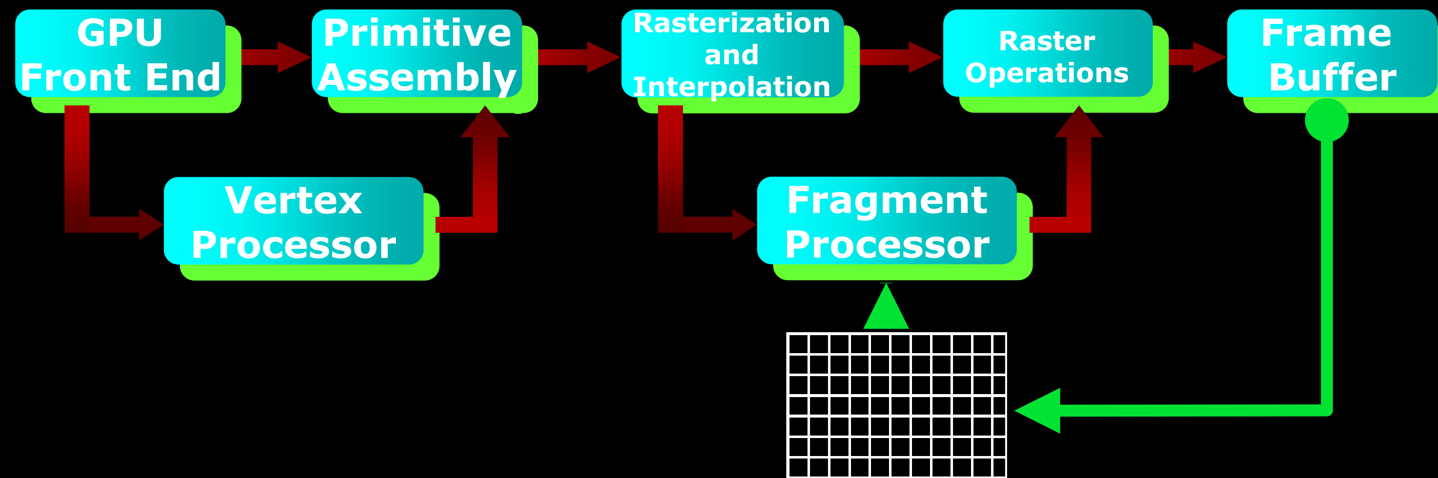
Optimization point: avoid work in the fragment program if possible.

Getting data back I: Readbacks



- Readbacks transfer data from the frame buffer to the CPU.
- ☺ They are very general (any buffer can be transferred)
- ☺ Partial buffers can be transferred
- ☹ They are slow: reverse data transfer across PCI/AGP bus is very slow (PCIe is a lot better)
- ☹ Data mismatch: readbacks return image data, but the CPU expects vertex data (or has to load image into texture)

Getting data back II: Copy-to-texture



- Copy-to-texture transfers data from frame buffer to texture.

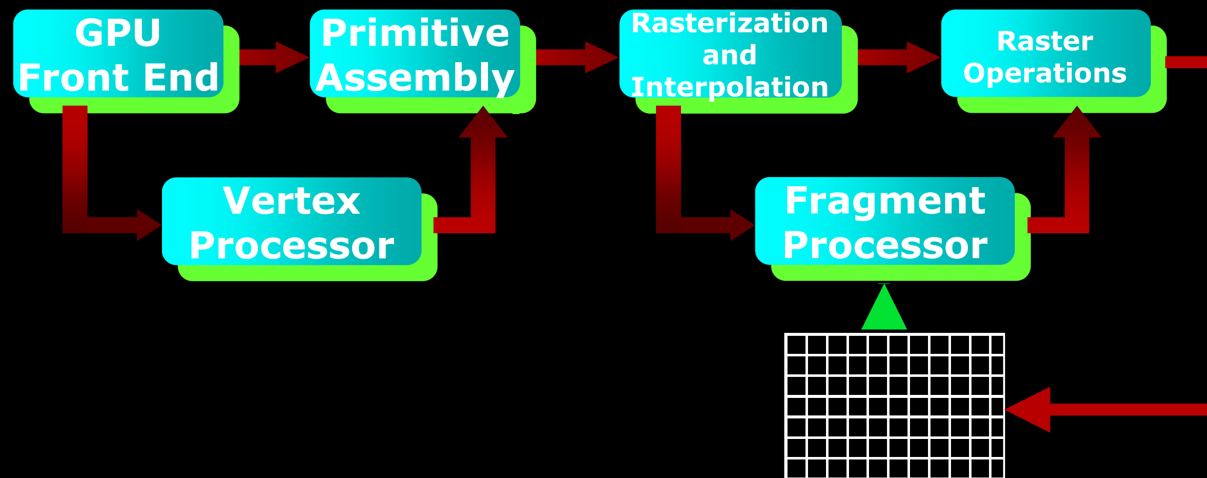
☺ Transfer does not cross GPU-CPU boundary.

☺ Partial buffers can be transferred

☹ Not very flexible: depth and stencil buffers cannot be transferred in this way, and copy to texture is still somewhat slow.

☹ Loss of precision in the copy.

Getting data back III: Render-to-texture



- Render-to-texture renders directly into a texture.
- ☺ Transfer does not cross GPU-CPU boundary.
- ☺ Fastest way to transfer data to fragment processor
- ☹ Only works with depth and color buffers (not stencil).

Render-to-texture is the best method for reading data back after a computation.

Using Render-to-texture

- Using the render-texture extension is tricky.
- You have to set up a pbuffer context, bind an appropriate texture to it, and then render to this context.
- Then you have to change context and read the bound texture.
- You cannot write to a texture and read it simultaneously
- Mark Harris (NVIDIA) has written a RenderTexture class that wraps all of this.
- The tutorial will have more details on this.
- RenderTextures are your friend !

Back to computing second smallest element

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").
- Vertex programs can't read input ! **[nv3x]**

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.

When might a VP need access to textures?

- n-body simulation:
  - We have a force field in a texture
  - Each vertex moves according to this force field.

    $\Delta v = a \, \Delta t$

    $\Delta s = v \, \Delta t$
  - In each pass, all vertex coordinates are updated.
  - New locations create new force field.
- How do we update vertex coordinates ?

Sending data back to vertex program

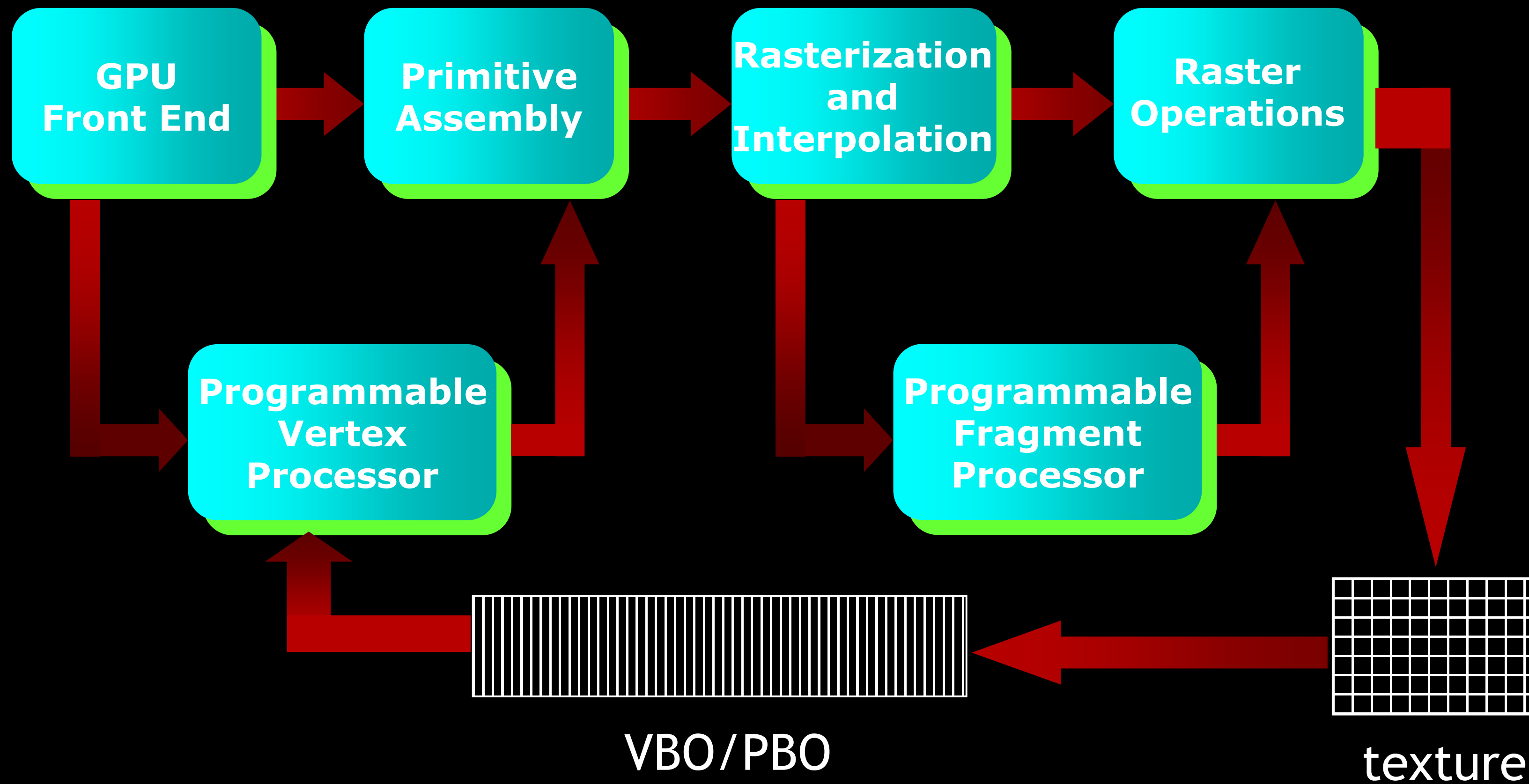
Solution:

- [Pass 1] Render all vertices to be stored in a texture.
- [Pass 2] Compute force field in fragment program
- [Pass 3] Update texture containing vertex coordinates in a fragment program using the force field.
- [Pass 4] Retrieve vertex data from texture. **How?**

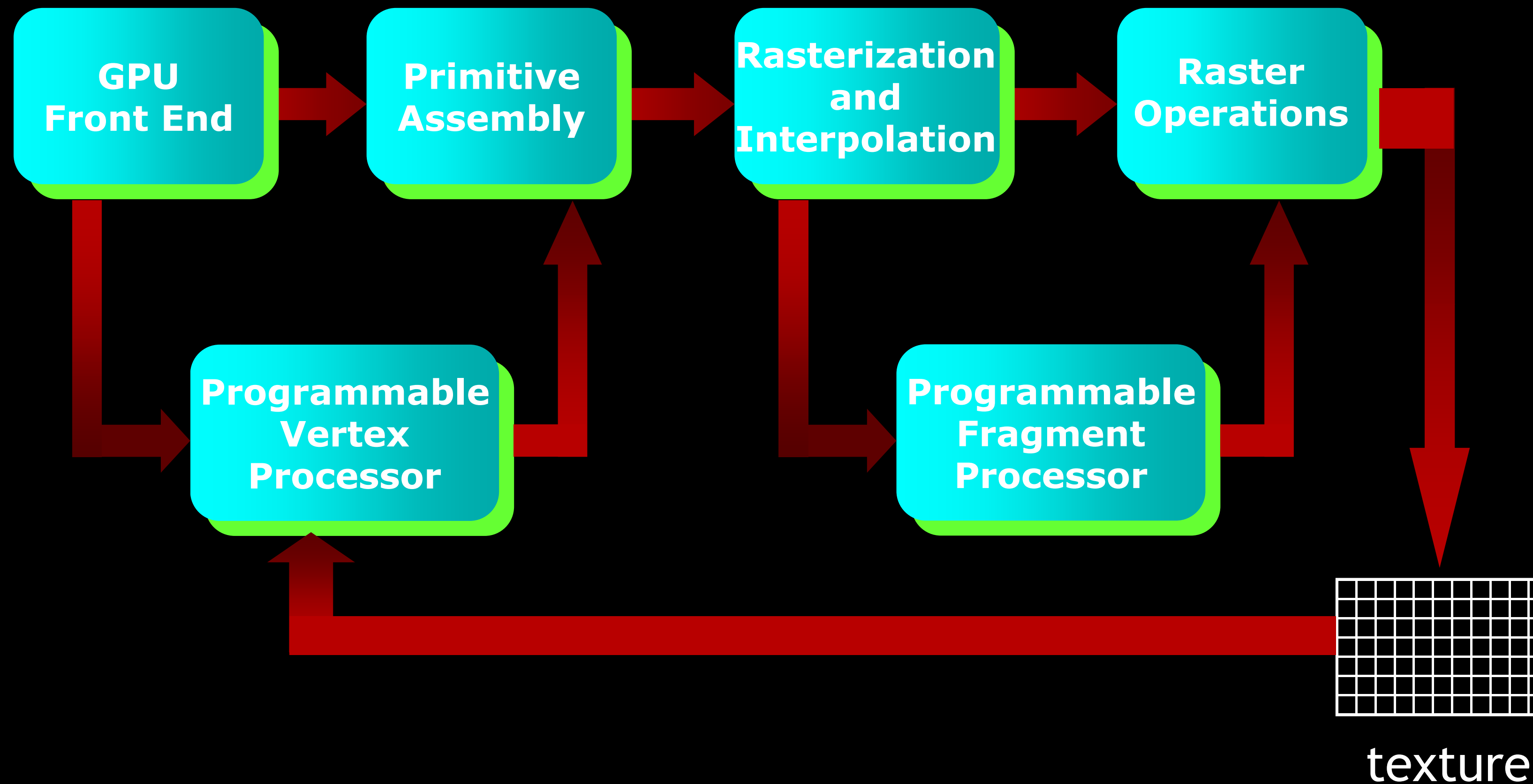
Vertex/Pixel Buffer Objects

- V/P buffer objects are ways to transfer data between framebuffer/vertex arrays and GPU memory.
- Conceptually, V/PBO are like CPU memory, but on the GPU.
- Can use glReadPixels to read to PBO
- Can create vertex array from VBO

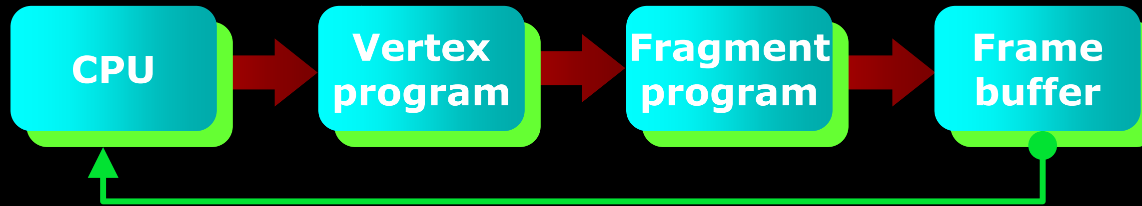
Solution



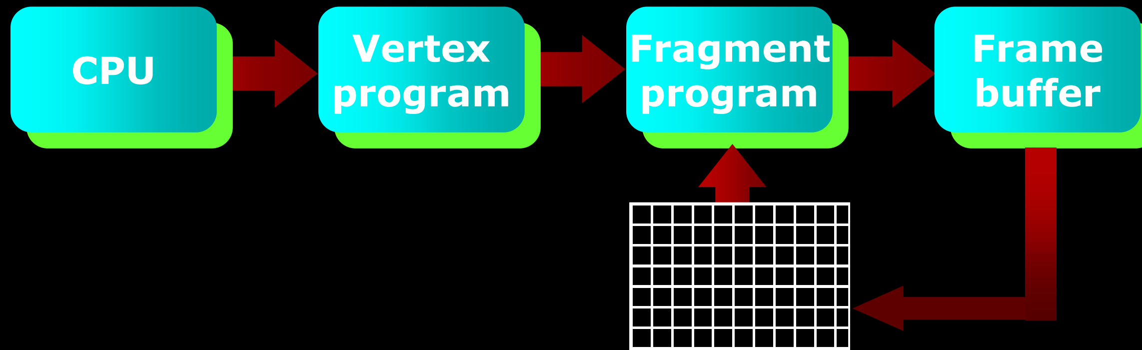
NV40: Vertex programs can read textures



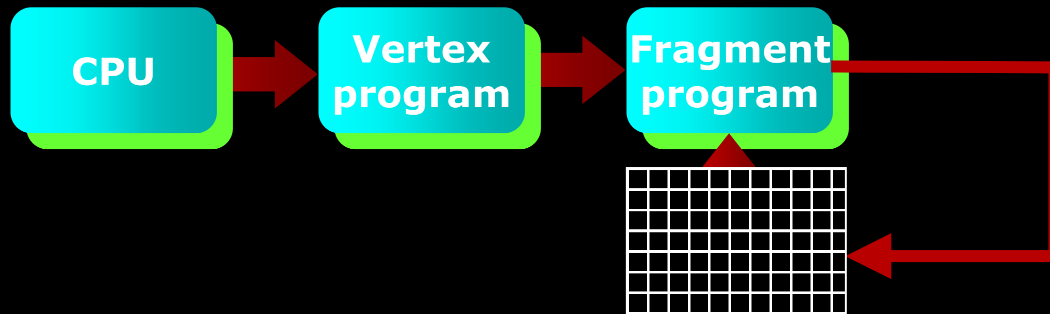
Summary of memory flow



Readback

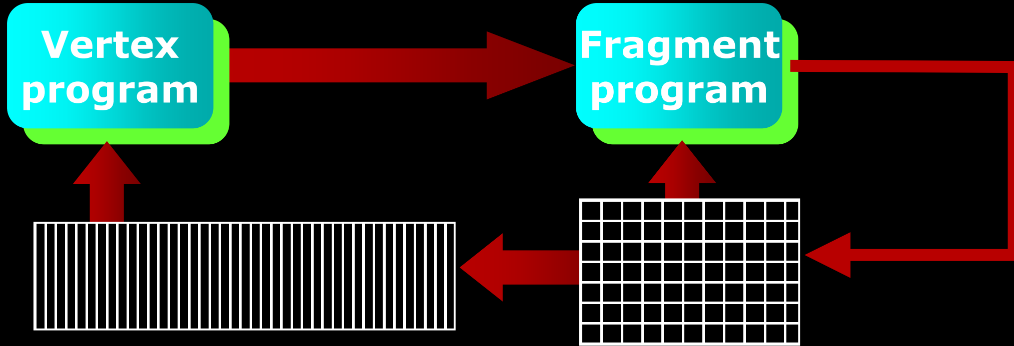


Copy-to-Texture

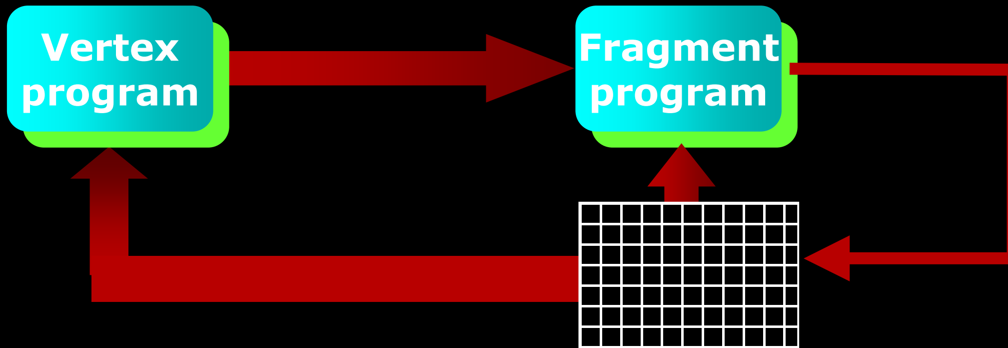


Render-to-Texture

Summary of memory flow



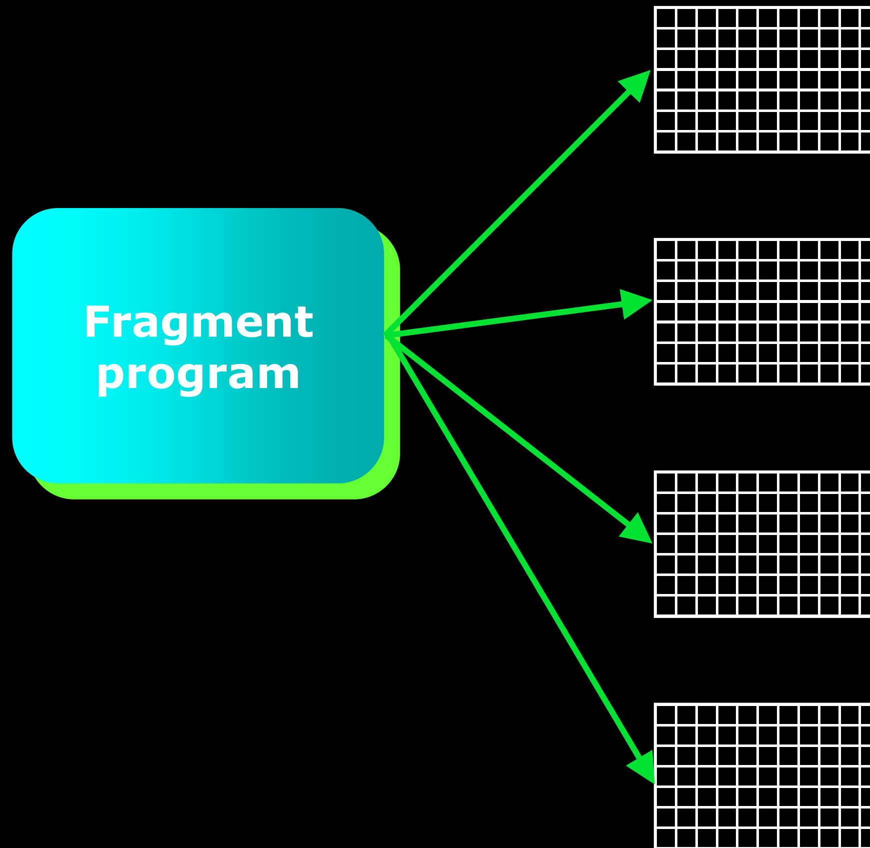
VBO/PBO transfer



nv40 texture ref

in vertex program

Multiple Render to Texture (MRT) [nv40]



MRT allows us to compress multiple passes into a single one.

This does not fundamentally change the model though, since read/write access is still not allowed.