Lecture 20:

Drawing Fast: The Graphics Pipeline

Explaining the Pipeline

Review: The “abstractions” of 3D Drawing

Triangles as “the” primitive

each triangle is independent

Vertices are independent (except they make triangles)

information at vertices, not triangles

Triangles cover pixels

pixels are independent (each with a coordinate)

local lighting/shading (texture lookups, light references)

Z-Buffer sorts it all out (order independence)

What happens when we draw

We draw **triangles**

They start out as positions (local coordinates)
They end up as colored pixels

Today…. The triangle’s Journey

Why do you care how the hardware works?

Because it is cool!

Why do you care how the hardware works?

The limitations of the hardware explain why we do things.

Why prefer texture hacks to more principled lighting?

It will help us understand how to make things fast.

Next week, we will program the hardware directly.

It will not make sense unless without understanding

The steps of 3D graphics

Model objects (make triangles)

Transform (find point positions)

Shade (lighting– per tri / vertex)

Transform (projection)

Rasterize (figure out pixels)

Shade (per-pixel coloring) Materials

Write pixels (with Z-Buffer test)

1. Put a 3D primitive in the World
   Modeling (model transformations)
2. Figure out what color it should be
   Shading
3. Position relative to the Eye
   Viewing / Camera Transformation
4. Get rid of stuff behind you/offscreen
   Clipping
5. Figure out where it goes on screen
   Projection (sometimes called Viewing)
6. Figure out if something else blocks it
   Visibility / Occlusion
7. Draw the 2D primitive
   Rasterization (convert to Pixels)

Practical Caveats

Colors might be determined later

Often we combine transformations

Z-Buffer reverses these two steps

A Pipeline

Triangles are independent
Vertices are independent
Pixels (within triangles) are independent

(caveats about sharing for efficiency)

Don’t need to finish 1 before start 2
(might want to preserve finishing order)

Is it really a pipeline?

In theory, yes – good way to think about

Historically – it is how the hardware works

In practice with modern hardware?

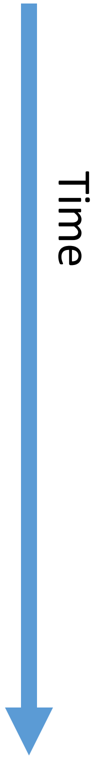
who knows

1 2 3 4

1

Pipelining in conventional processors

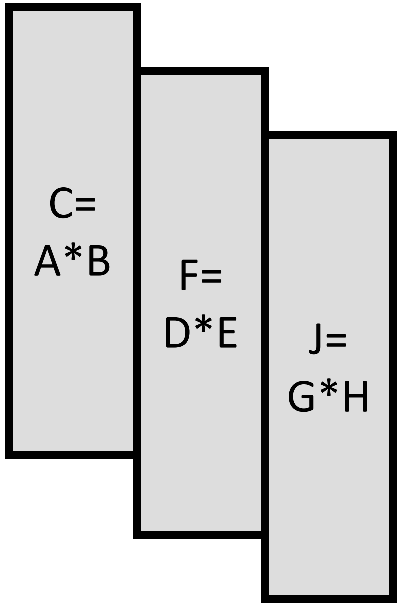
Start step 2 before step 1 completes



1 C = A * B

2 F = D * E

J = G * H



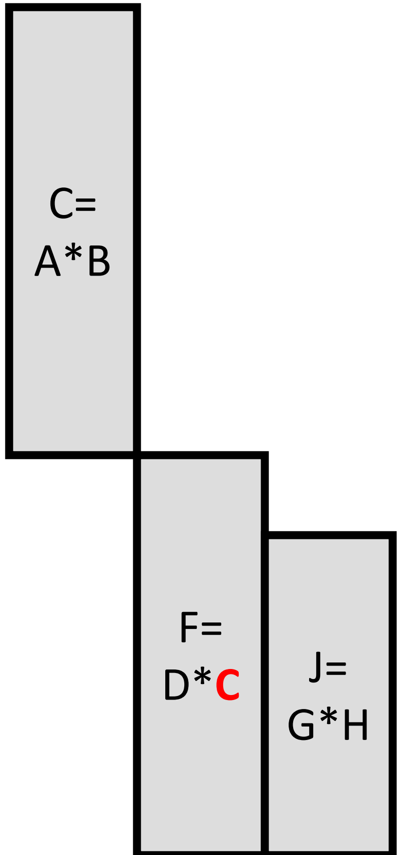
Unless step 2 depends on step 1

Pipe Stall

C = A * B

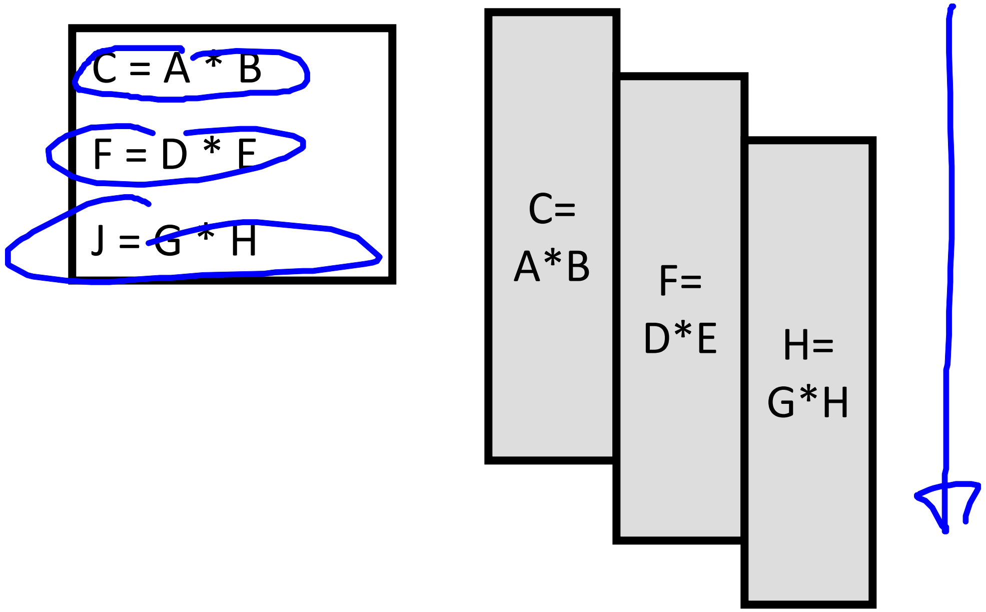
F = D * C

J = G * H



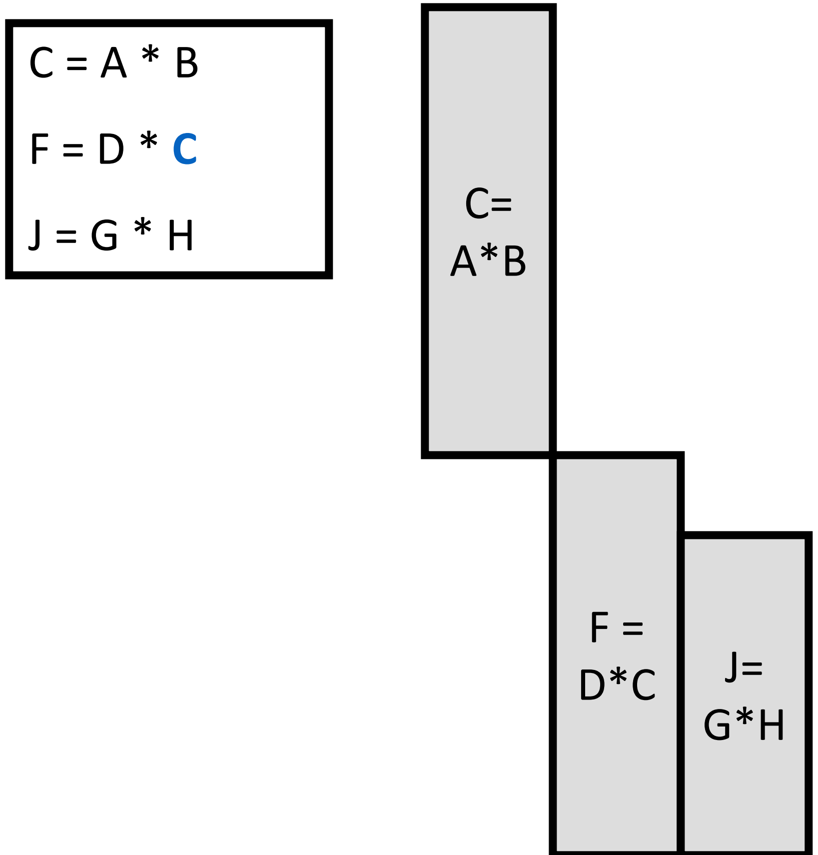
Triangles are independent!
No stalls! (no complexity of handling stalls)

Start step 2 before step 1 completes

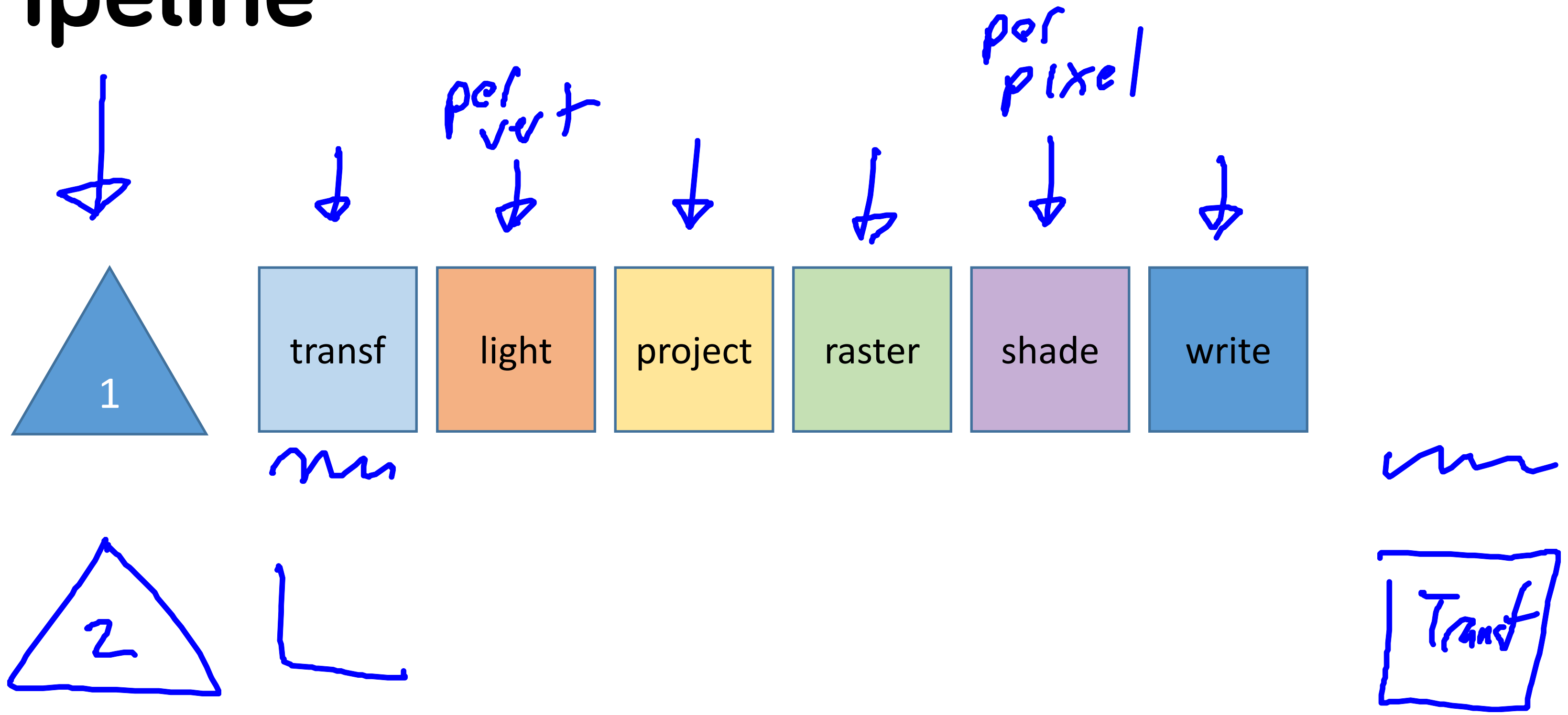


Unless step 2 depends on step 1

Pipe Stall

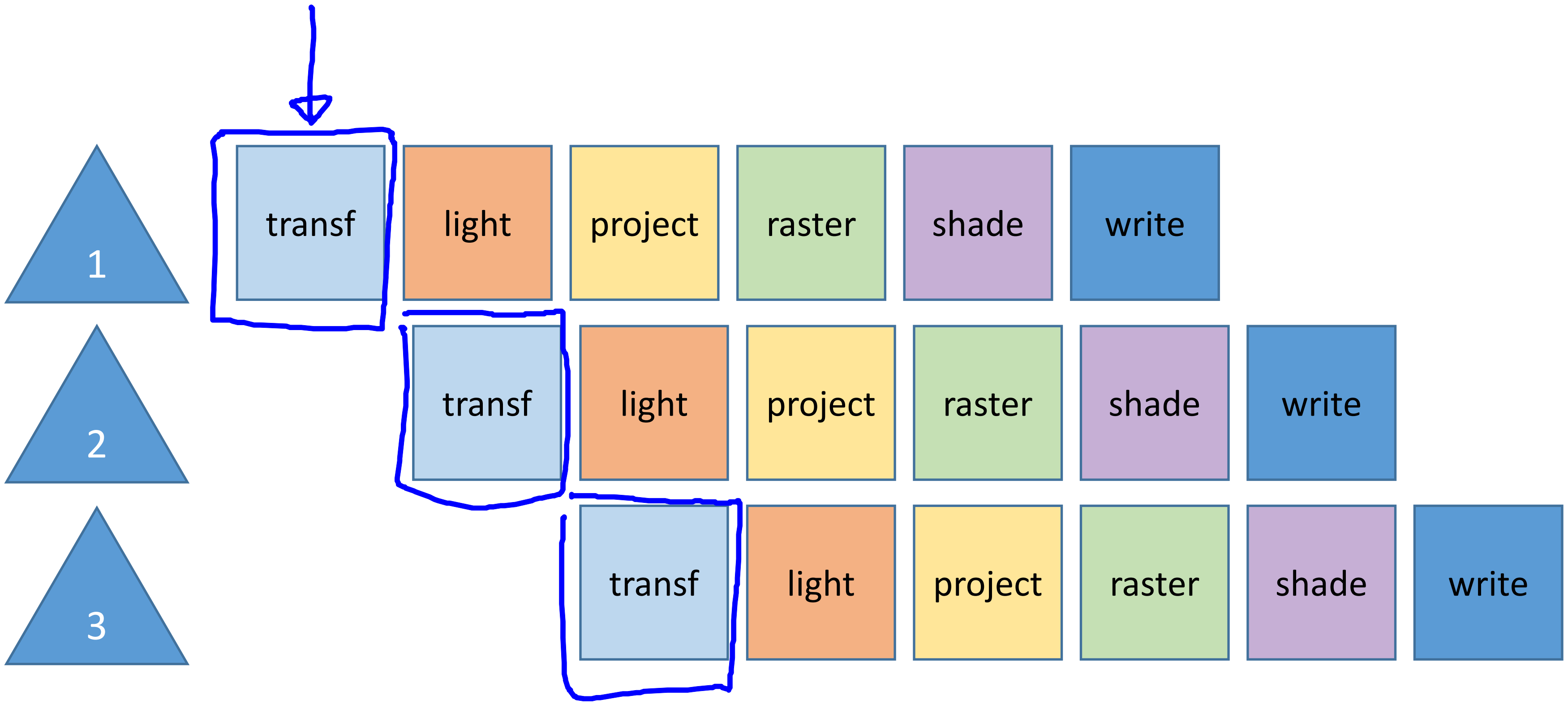


A Pipeline

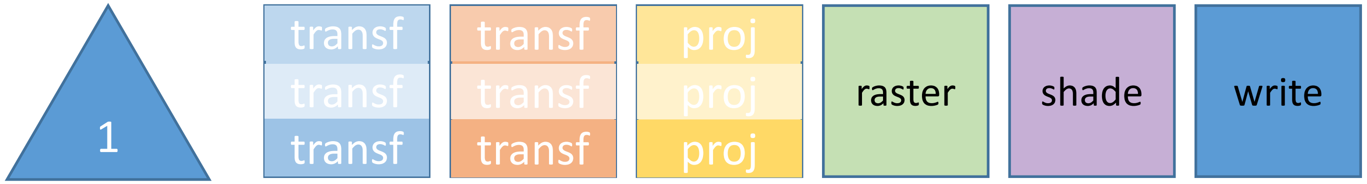


per vert
per pixel
1
transf
light
project
raster
shade
write
2
Transf

A Pipeline


1
2
3
transf
light
project
raster
shade
write
transf
light
project
raster
shade
write
transf
light
project
raster
shade
write

Vertices are independent
Parallelize!



Parallelization

Vertex operations

split triangles / re-assemble

compute **per-vertex** not per-triangle

Pixel (fragment) operations

lots of potential parallelism

less predictable

Use queues and caches

Why do we care?

This is why the hardware can be fast

It requires a specific model
Hardware implements this model

The programming interface is designed for this model.
You need to understand it.

Some History...

Custom Hardware (pre-1980)

rare, each different

Workstation Hardware (early 80s-early 90s)

increasing features, common methods

Consumer Graphics Hardware (mid 90s-)

cheap, eventually feature complete

Programmable Graphics Hardware (2002-)

Graphics Workstations 1982-199X

Implemented graphics in hardware

Providing a common abstraction set

Fixed function –

it was built into the hardware

Silicon Graphics (SGI)

Stanford Research Project 1980

Spun-off to SGI (company) 1982

The Geometry Engine

- 4x4 matrix multiply chip
- approximate division

Raster engine (Z-buffer)

1988: The Personal Iris

The 4D-2X0 series

4 processors (240)

Different graphics

1988 – GT/GTX

1990 - VGX

Why do you care?

It's the first time the abstractions were right

later stuff adds to it

It's where the programming model is from

it was IrisGL before OpenGL (before WebGL)

It's the pipeline at its essence

we'll add to it, not take away

The Abstractions

Points / Lines / Triangles
Vertices in 4D
Color in 4D (RGBA = transparency)
Per-Vertex transform (4x4 + divide by w)
Per-Vertex lighting
Color interpolation
Fill Triangle
Z-test (and other tests)
Double buffer (and other buffers)

What’s left to add?

All of this was in hardware in the 80s

1990 – texture

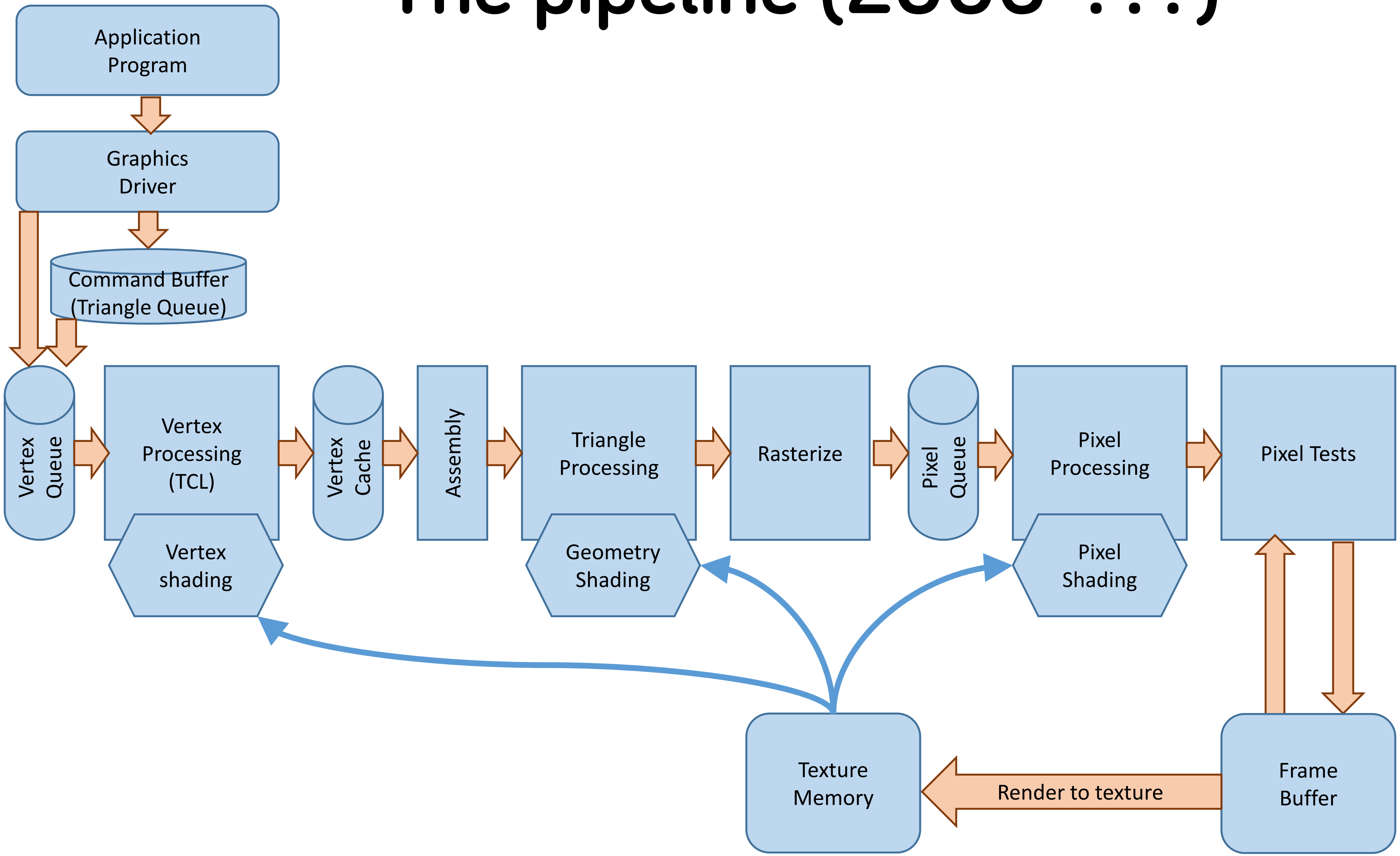
1992 – multi-texture (don’t really need)

1998 (2000) – programmable shading

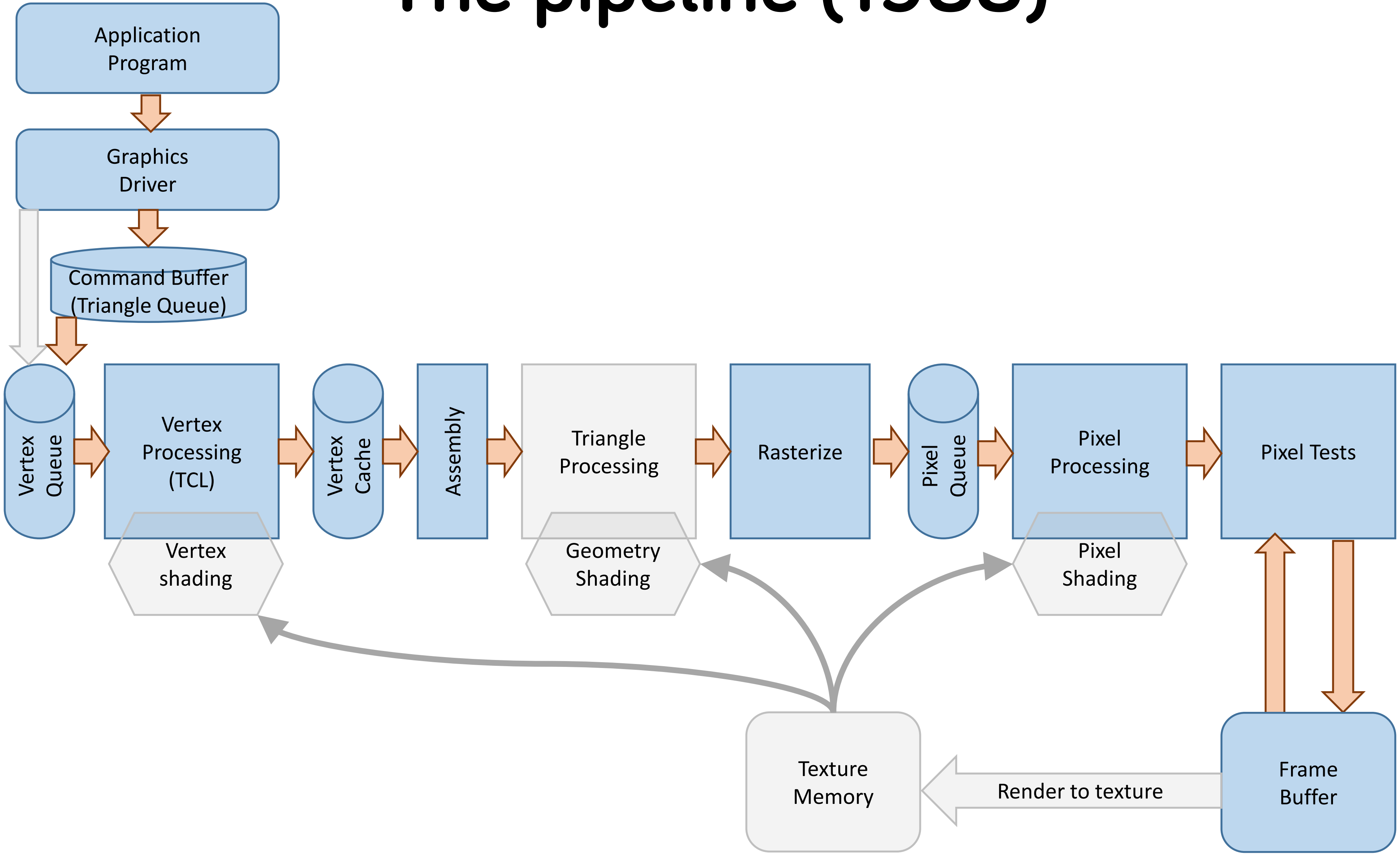
2002 – programmable pipelines

2005 – more programmability

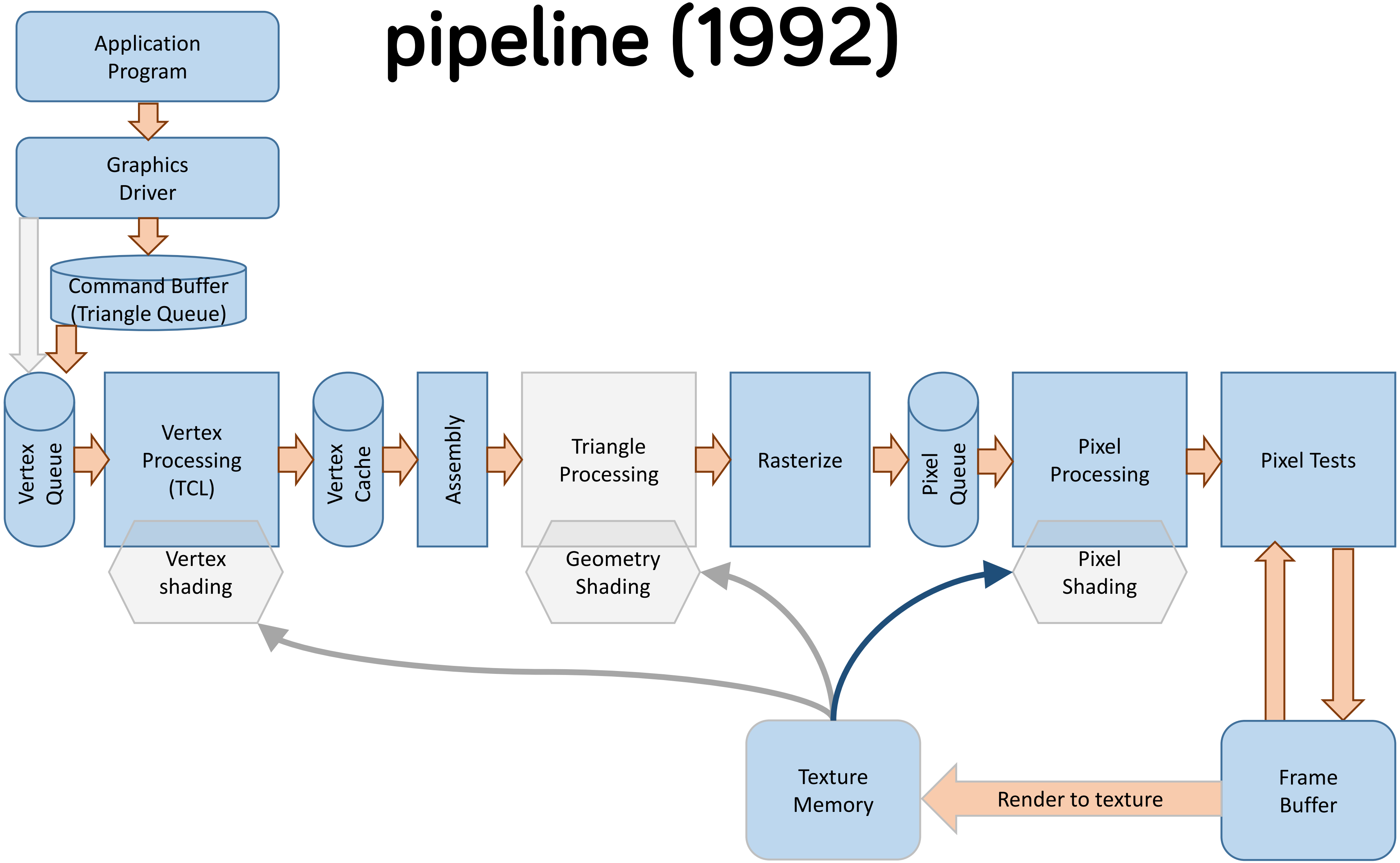
The pipeline (2006-???)



The pipeline (1988)



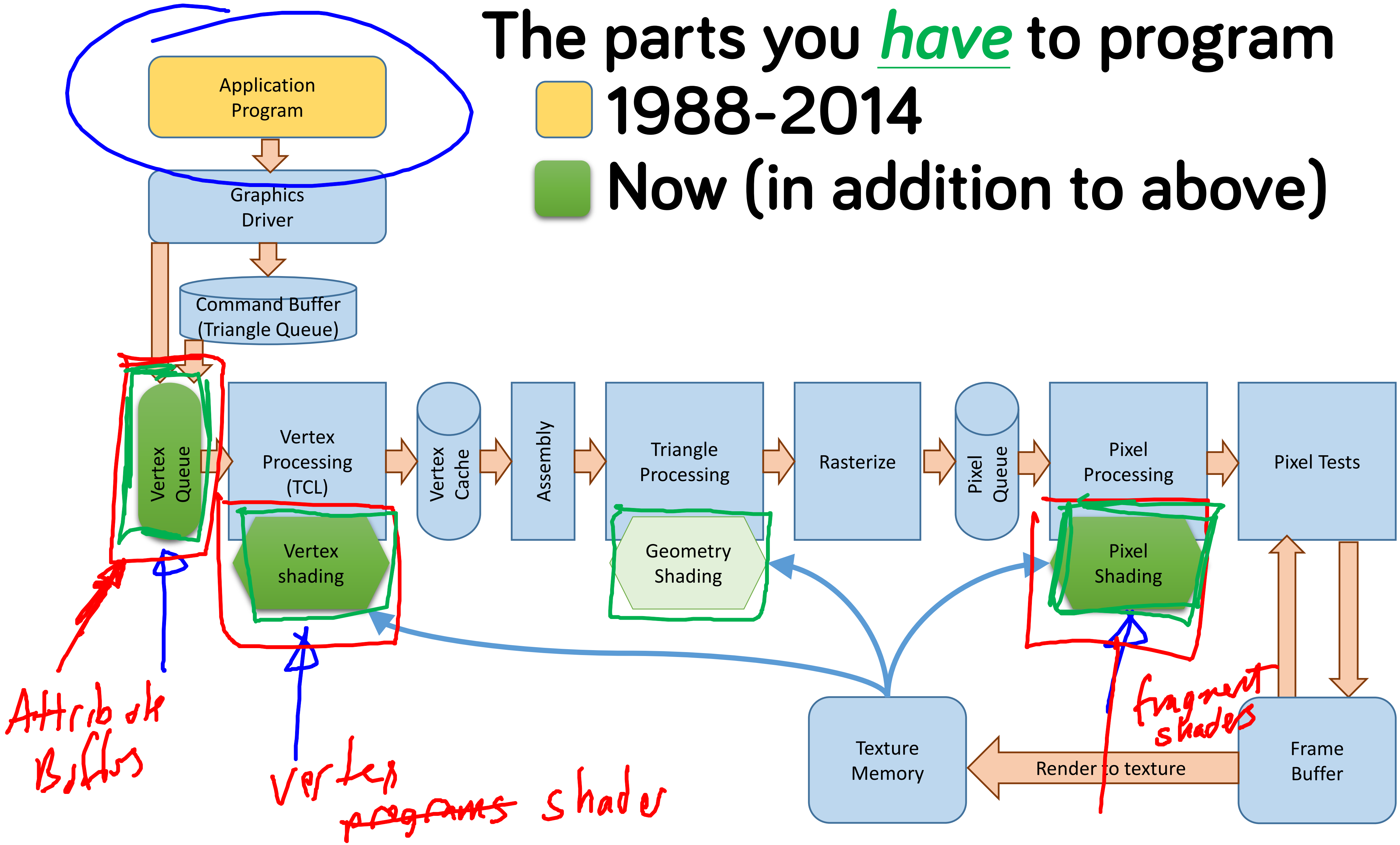
The full fixed-function pipeline (1992)



The parts you *have* to program

1988-2014

Now (in addition to above)



A Triangle's Journey

Things to observe
as we travel through the pipeline…

What does each stage do?

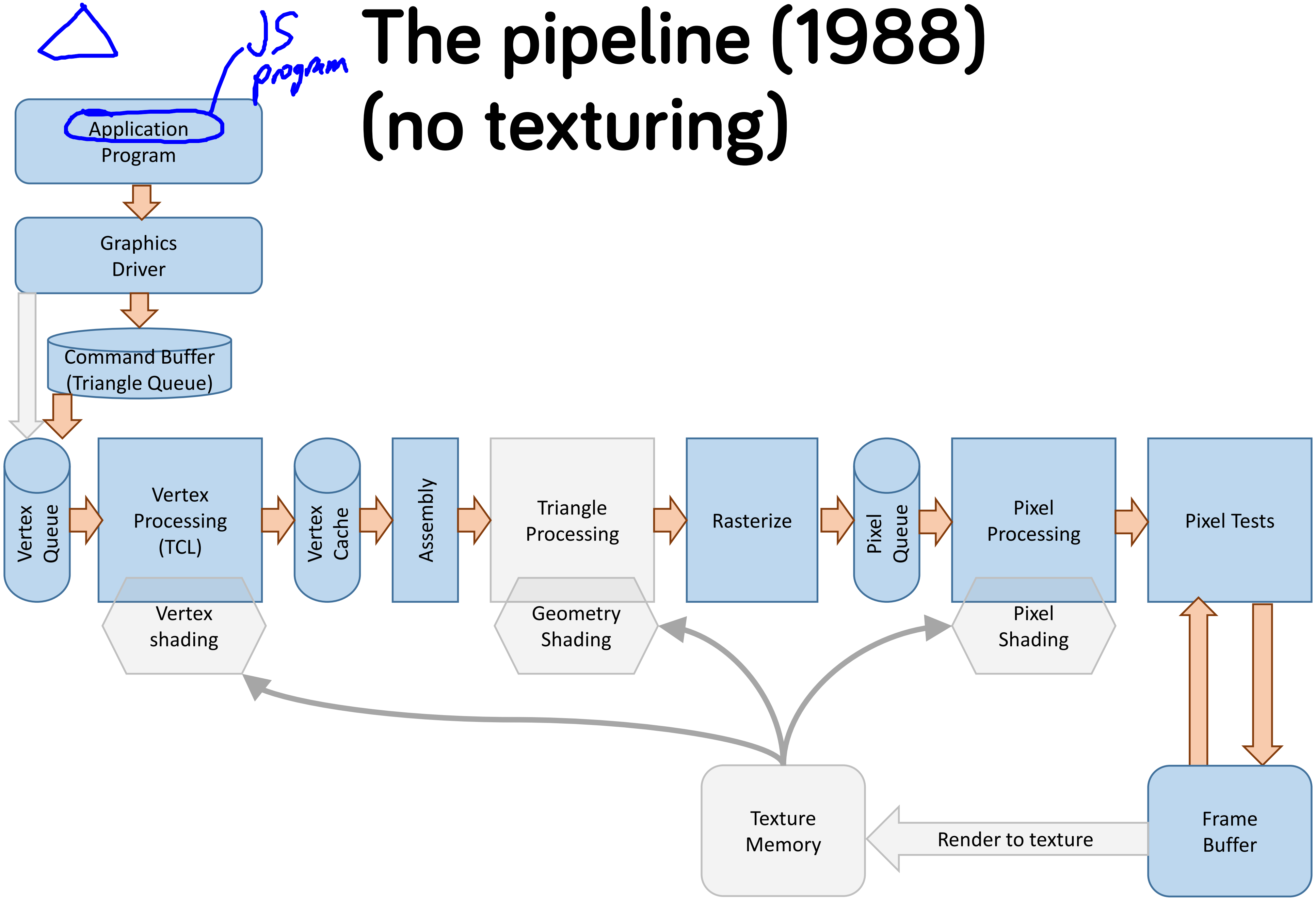
What are its inputs and output?

  important for programmability

Why would it be a bottleneck?

  and what could we do to avoid it

The pipeline (1988) (no texturing)



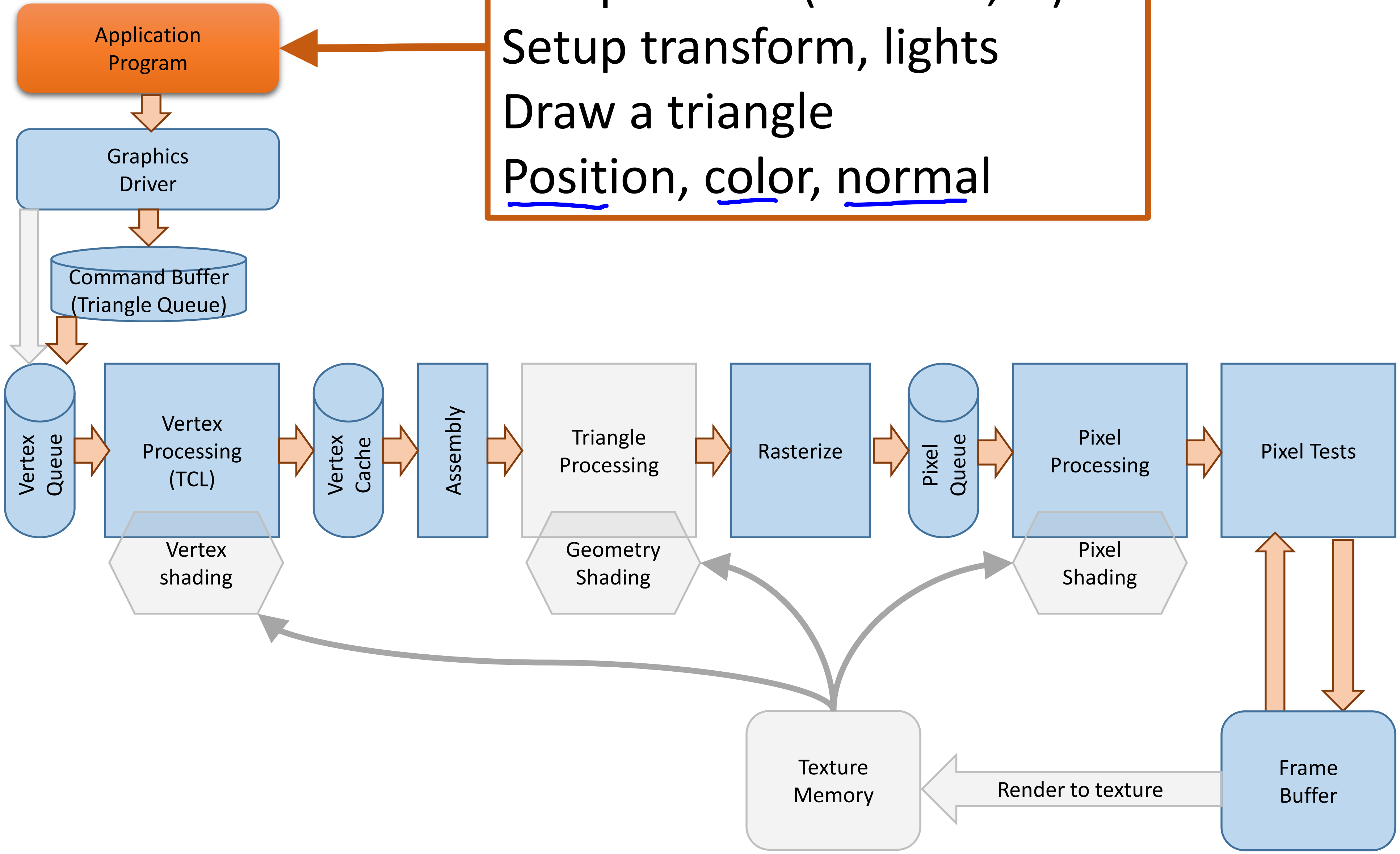
Start here

Setup modes (window, …)

Setup transform, lights

Draw a triangle

Position, color, normal



Drawing a triangle

Modes per triangle (group)

which window, how to fill, use z-buffer, ...

Data per-vertex

position

normal

color

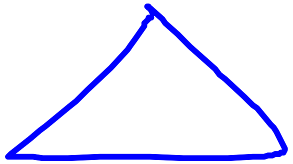
other things (texture coords)

Per Vertex?

Modes per triangle

which window, how to fill, use z-buffer, …

Data per-vertex



position

normal ← allow us to make non-flat

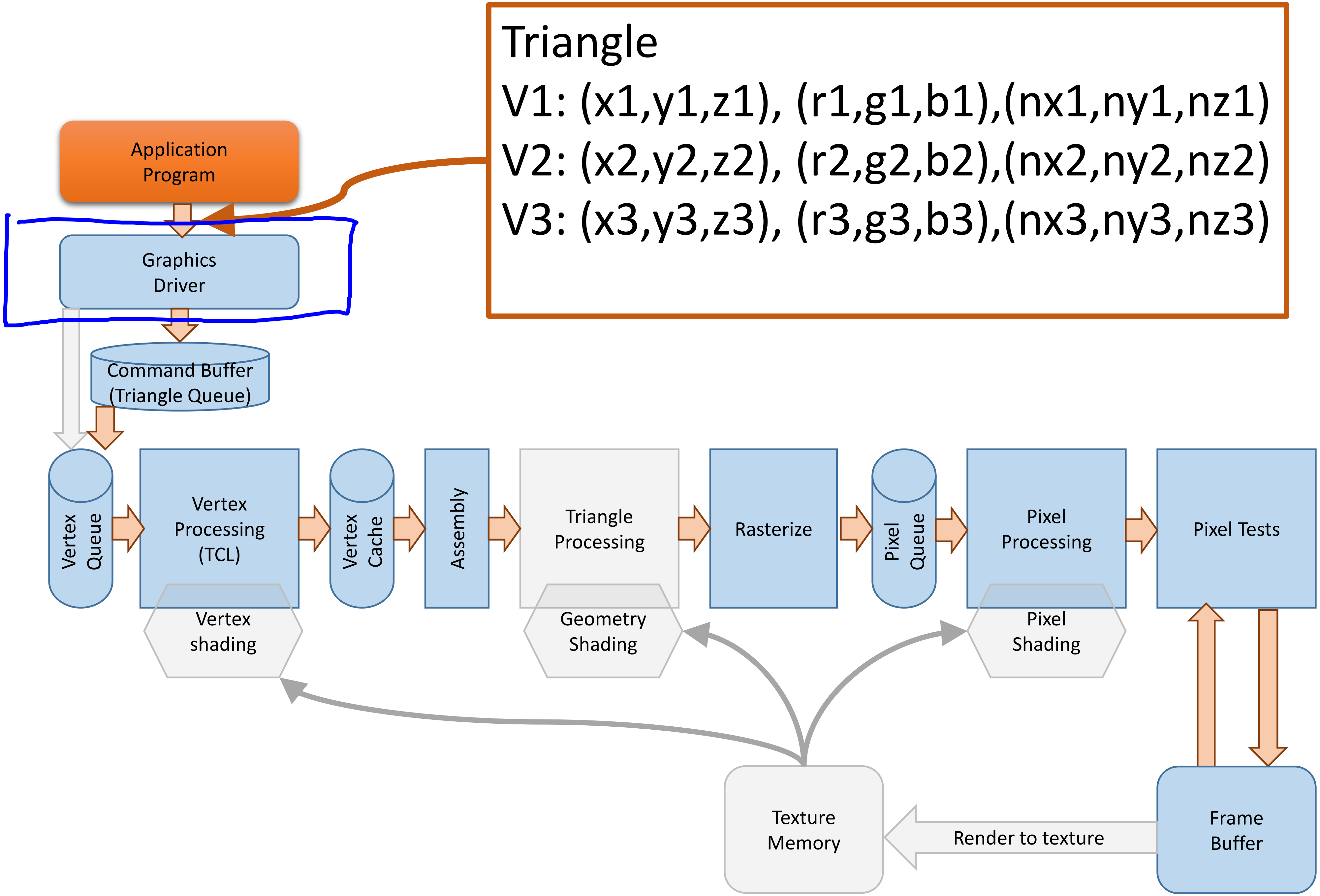
color ← allows us to interpolate

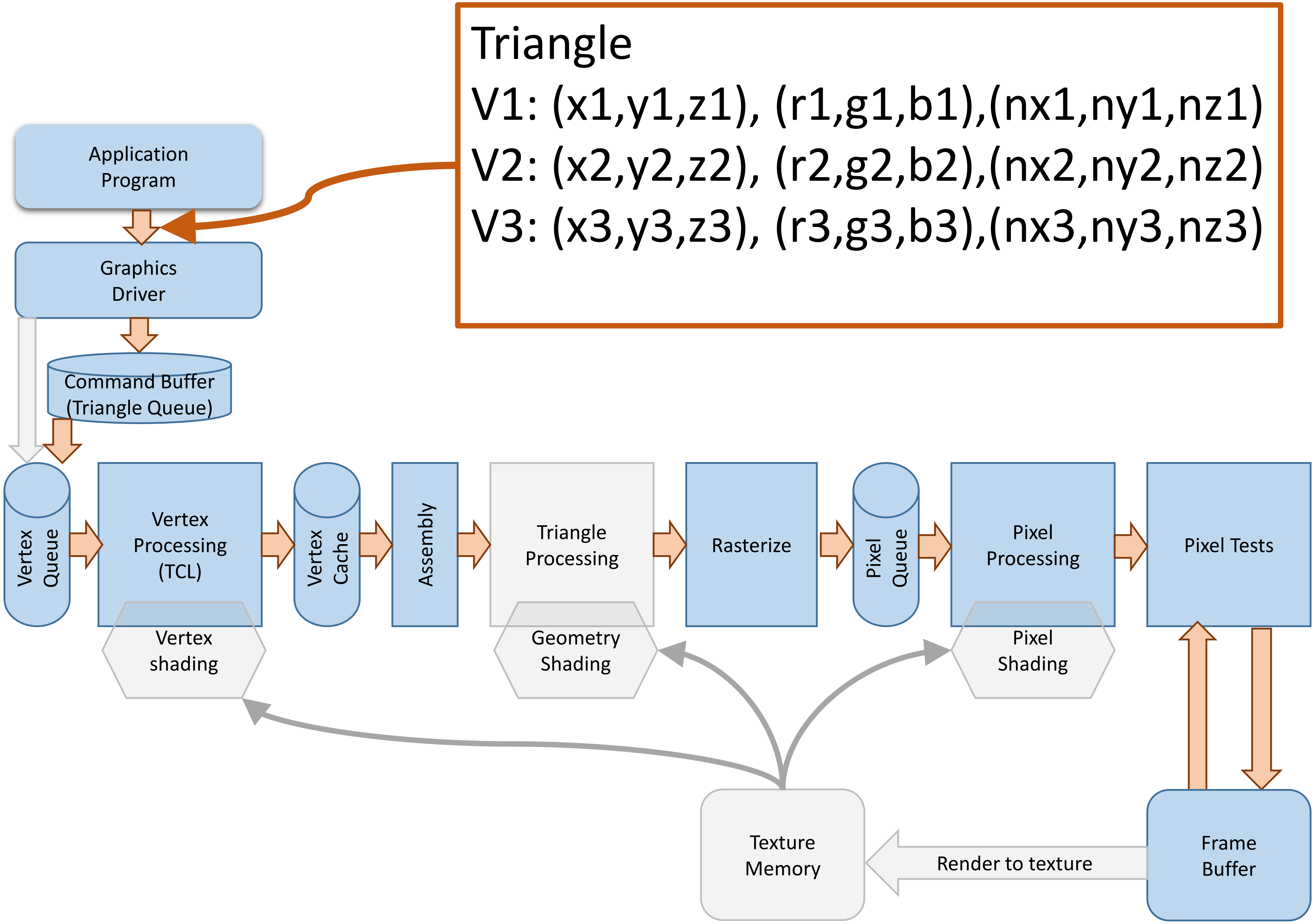
other things (texture coords)

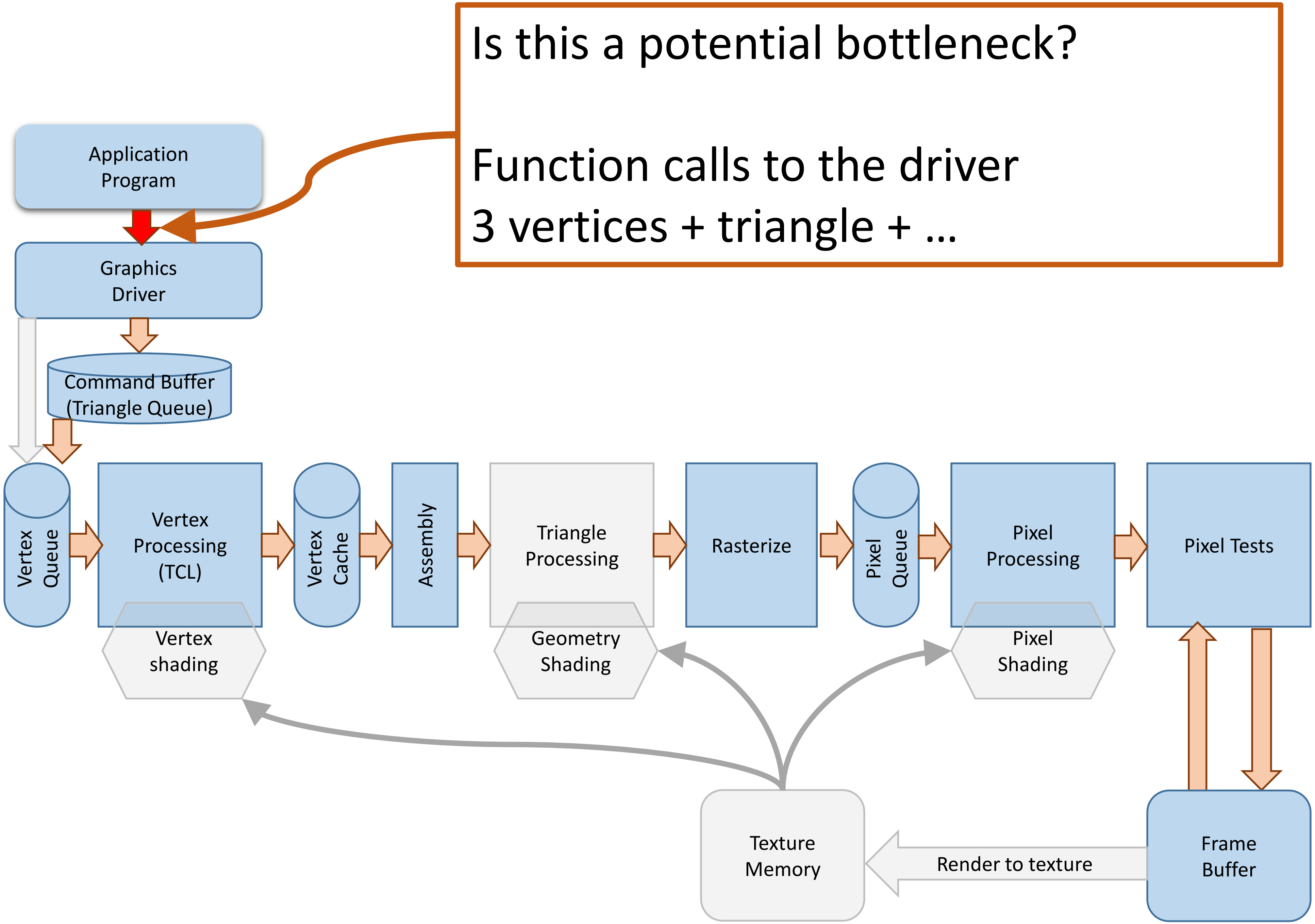
Per-Vertex not Per-Triangle

Allows sharing vertices between triangles

Or make all the vertices the same
(color, normal, …) to get truly flat


Triangle
V1: (x1,y1,z1), (r1,g1,b1),(nx1,ny1,nz1)
V2: (x2,y2,z2), (r2,g2,b2),(nx2,ny2,nz2)
V3: (x3,y3,z3), (r3,g3,b3),(nx3,ny3,nz3)
Application Program
Graphics Driver
Command Buffer (Triangle Queue)
Vertex Queue
Vertex Processing (TCL)
Vertex shading
Vertex Cache
Assembly
Triangle Processing
Geometry Shading
Rasterize
Pixel Queue
Pixel Processing
Pixel Shading
Pixel Tests
Texture Memory
Render to texture
Frame Buffer


Triangle
V1: (x1,y1,z1), (r1,g1,b1),(nx1,ny1,nz1)
V2: (x2,y2,z2), (r2,g2,b2),(nx2,ny2,nz2)
V3: (x3,y3,z3), (r3,g3,b3),(nx3,ny3,nz3)
Application Program
Graphics Driver
Command Buffer (Triangle Queue)
Vertex Queue
Vertex Processing (TCL)
Vertex shading
Vertex Cache
Assembly
Triangle Processing
Geometry Shading
Rasterize
Pixel Queue
Pixel Processing
Pixel Shading
Pixel Tests
Texture Memory
Render to texture
Frame Buffer



Is this a potential bottleneck?
Function calls to the driver
3 vertices + triangle + …
Application Program
Graphics Driver
Command Buffer (Triangle Queue)
Vertex Queue
Vertex Processing (TCL)
Vertex shading
Vertex Cache
Assembly
Triangle Processing
Geometry Shading
Rasterize
Pixel Queue
Pixel Processing
Pixel Shading
Pixel Tests
Texture Memory
Render to texture
Frame Buffer

Old style OpenGL

```
begin(TRIANGLE);
c3f(r1,g1,b1);
n3f(nx1,ny1,nz1);
v3f(x1,y1,z1);
c3f(r2,g2,b2);
n3f(nx2,ny2,nz2);
v3f(x2,y2,z2);
c3f(r3,g3,b3);
n3f(nx3,ny3,nz3);
v3f(x3,y3,z3);
end(TRIANGLE);
```

11 function calls
35 arguments pushed

Old days:
This is a lot less than the number of pixels!

Nowadays:
Just the memory access swamps the process

Attribute buffers!

Block transfers of data

Data for lots of triangles moved as a block

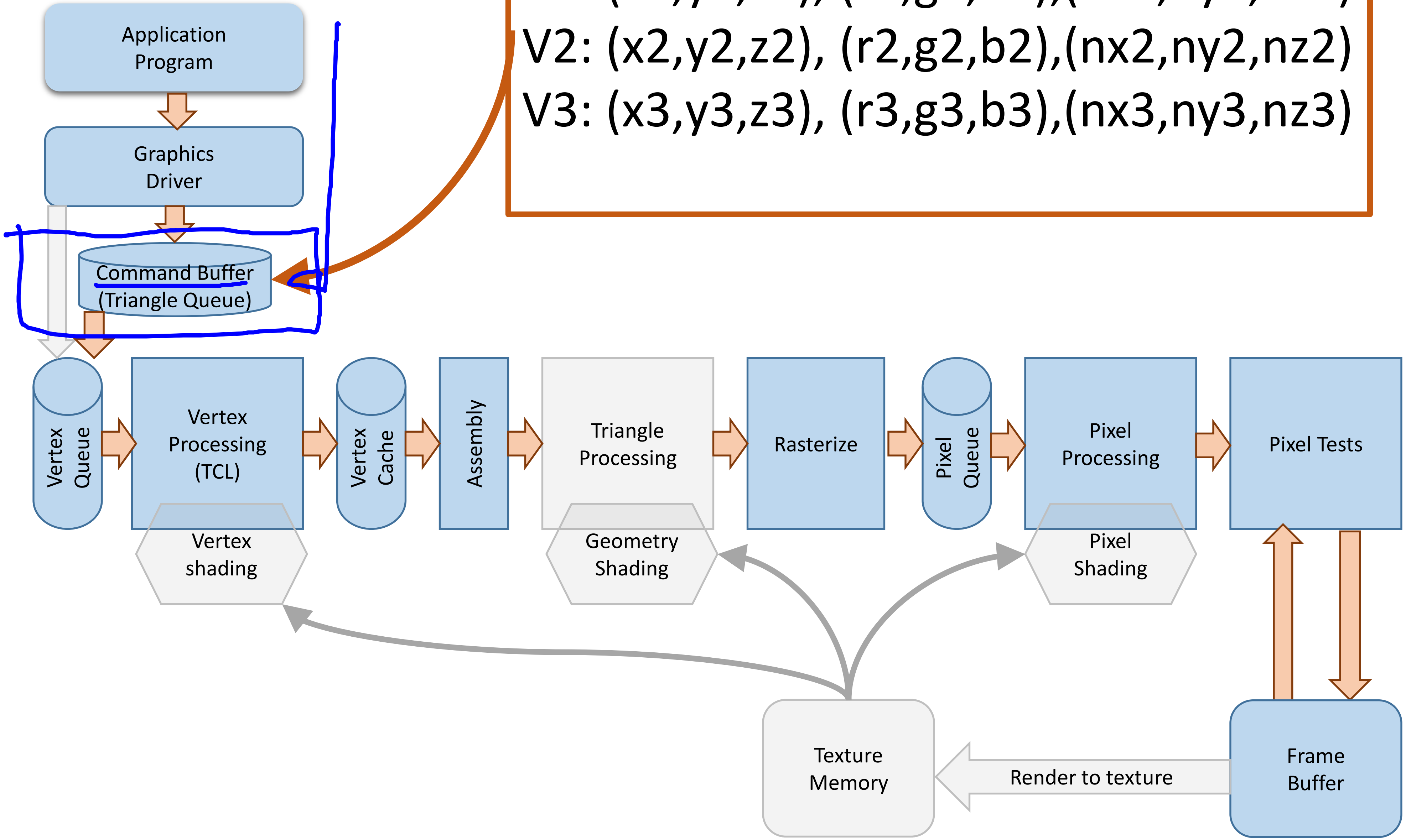
Try to draw groups of triangles

Triangle

V1: (x1,y1,z1), (r1,g1,b1),(nx1,ny1,nz1)

V2: (x2,y2,z2), (r2,g2,b2),(nx2,ny2,nz2)

V3: (x3,y3,z3), (r3,g3,b3),(nx3,ny3,nz3)

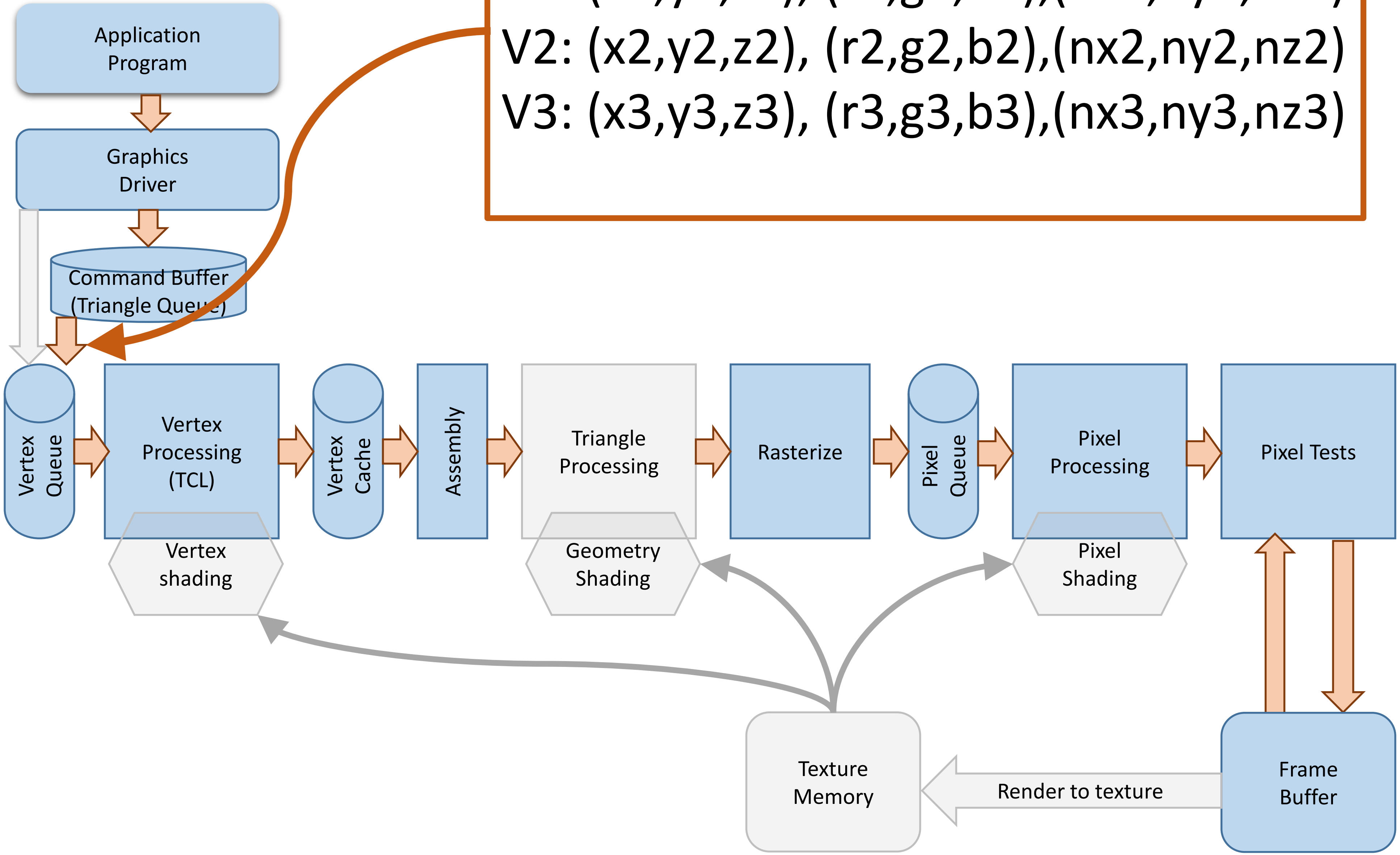


Split up triangles into **vertices**

V1: (x1,y1,z1), (r1,g1,b1),(nx1,ny1,nz1)

V2: (x2,y2,z2), (r2,g2,b2),(nx2,ny2,nz2)

V3: (x3,y3,z3), (r3,g3,b3),(nx3,ny3,nz3)

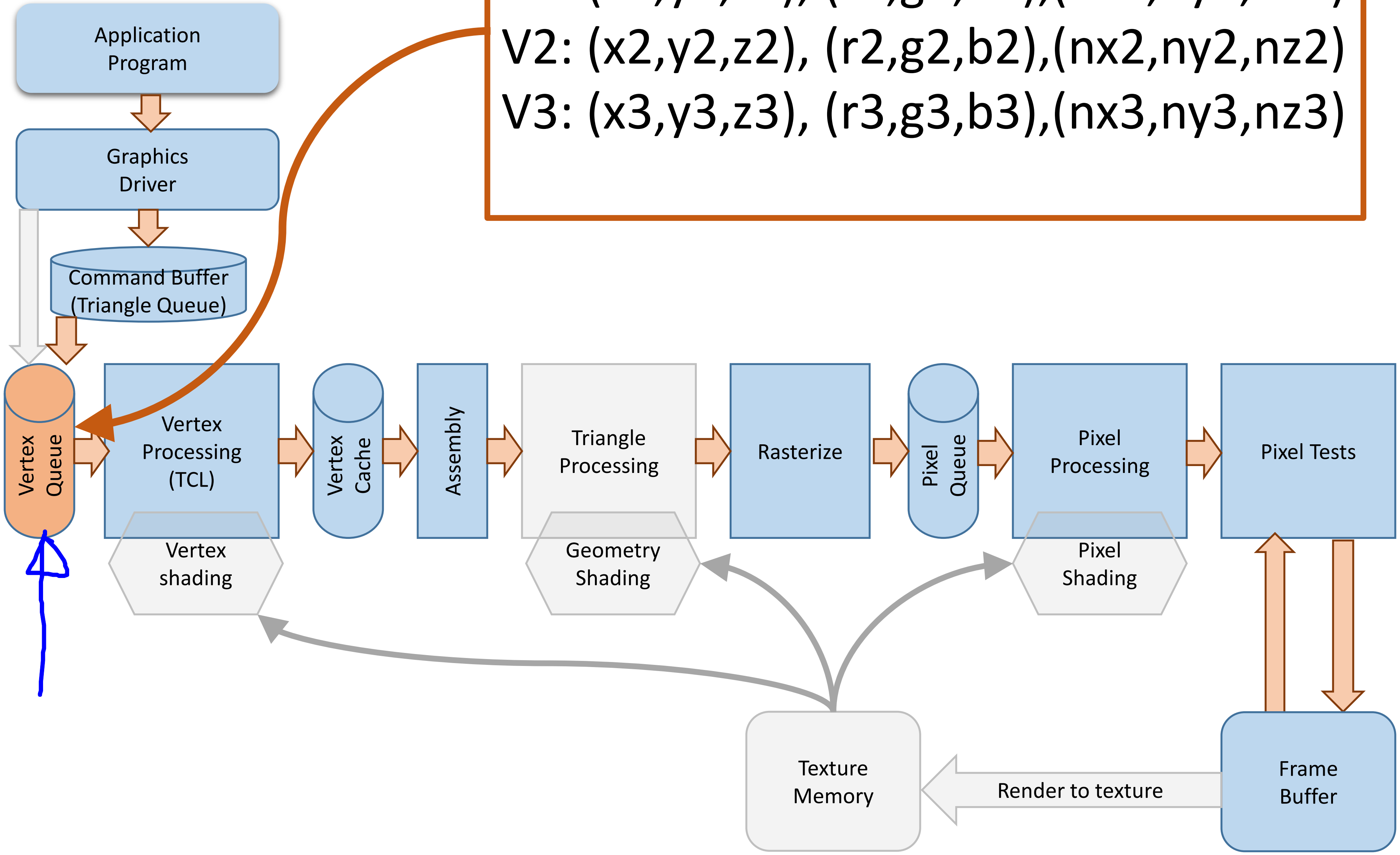


Buffer / Queue the **vertices**

V1: (x1,y1,z1), (r1,g1,b1),(nx1,ny1,nz1)

V2: (x2,y2,z2), (r2,g2,b2),(nx2,ny2,nz2)

V3: (x3,y3,z3), (r3,g3,b3),(nx3,ny3,nz3)



Buffering Vertices

Old Days:

- Vertex processing expensive
- Try to maximize re-use
- Process once an use for many triangles

Nowadays

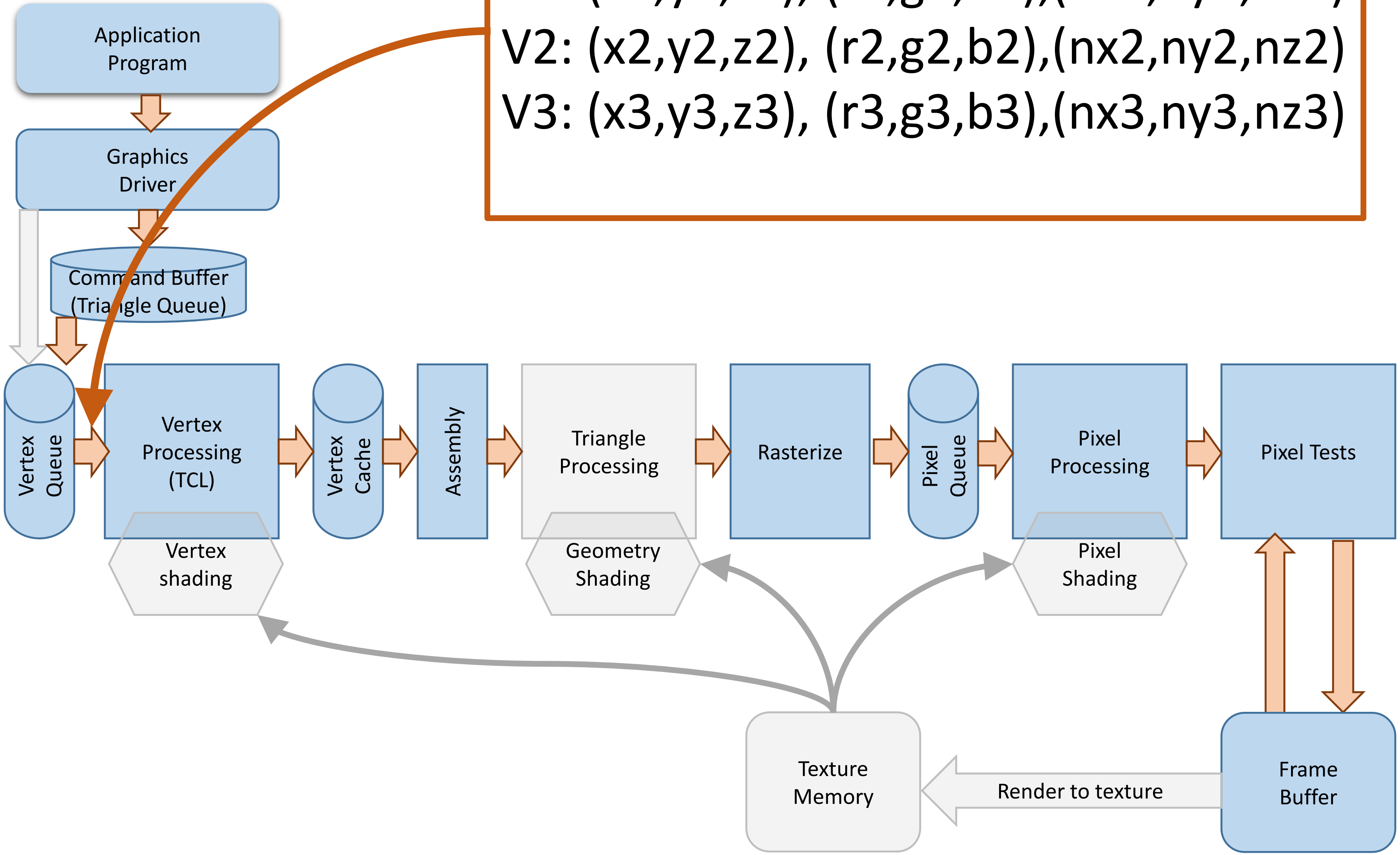
- Getting vertex to hardware is expensive
- Process vertices in parallel

Buffer / Queue the **vertices**

V1: (x1,y1,z1), (r1,g1,b1),(nx1,ny1,nz1)

V2: (x2,y2,z2), (r2,g2,b2),(nx2,ny2,nz2)

V3: (x3,y3,z3), (r3,g3,b3),(nx3,ny3,nz3)



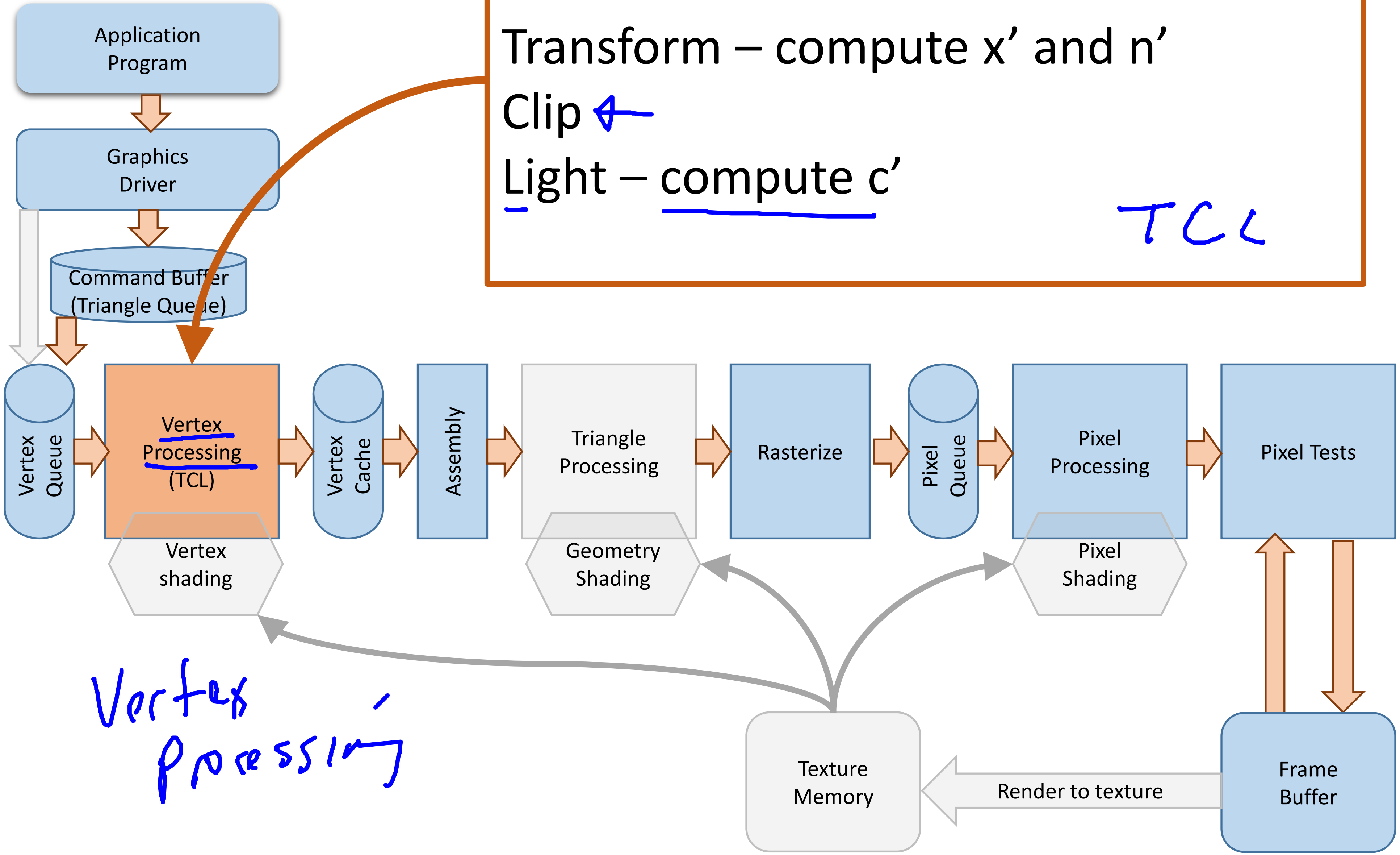
Process each vertex independently

Transform – compute x’ and n’

Clip

Light – compute c’

TCL



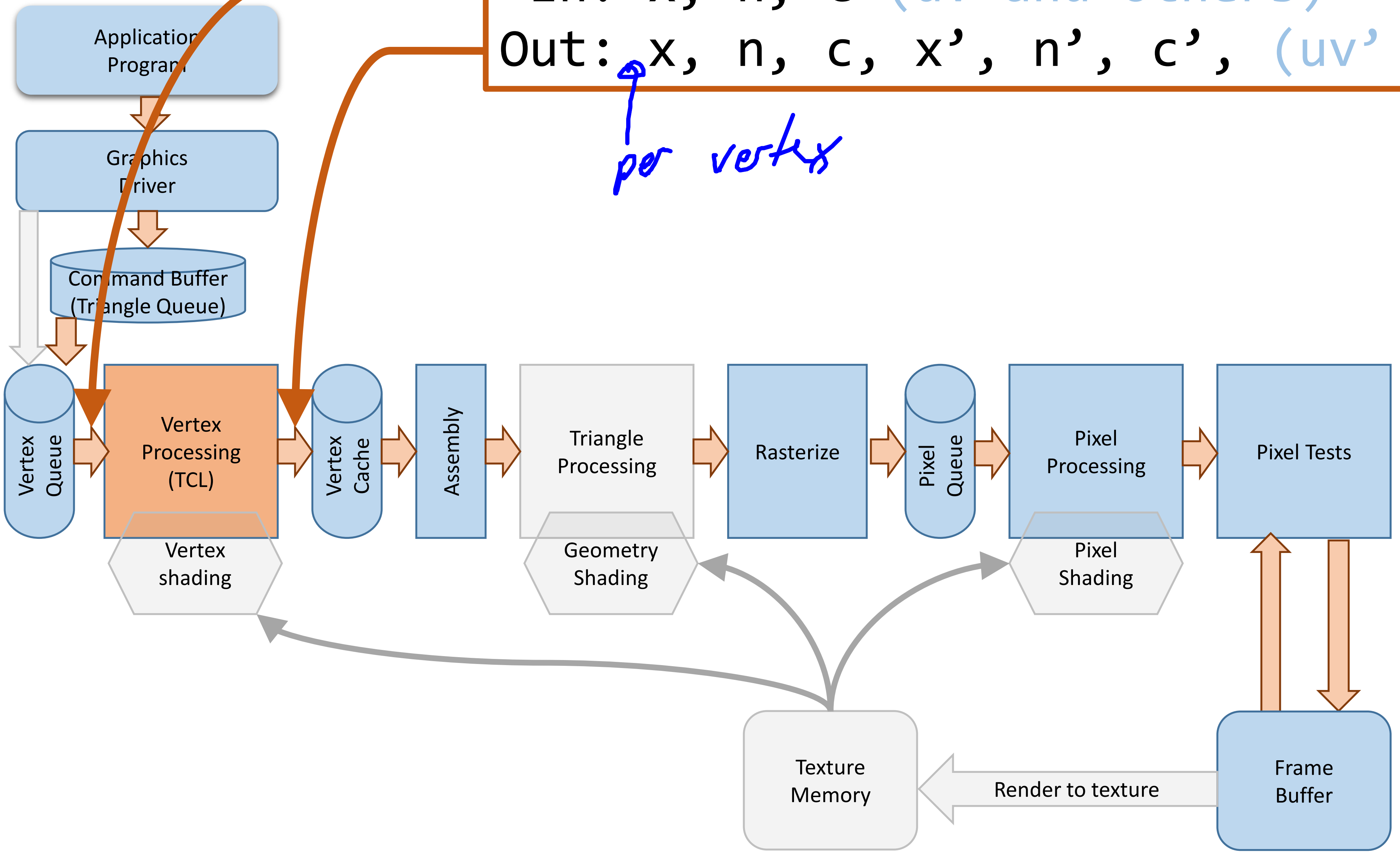
Vertex processing

Vertex in → Vertex out

In: x, n, c (uv and others)

Out: x, n, c, x’, n’, c’, (uv’ …)

per vertex



Vertex Processing

Just adds information to vertices

Computes transformation
	screen space positons, normals
Computes "lighting"
	new colors

(in the old days, clipping done here hence TCL)
(in the old days, lighting done per-vertex)

Vertex Processing: Each vertex is independent

Inputs are:

vertex information for **this vertex**

any "global" information

current transform, lighting, …

MODES

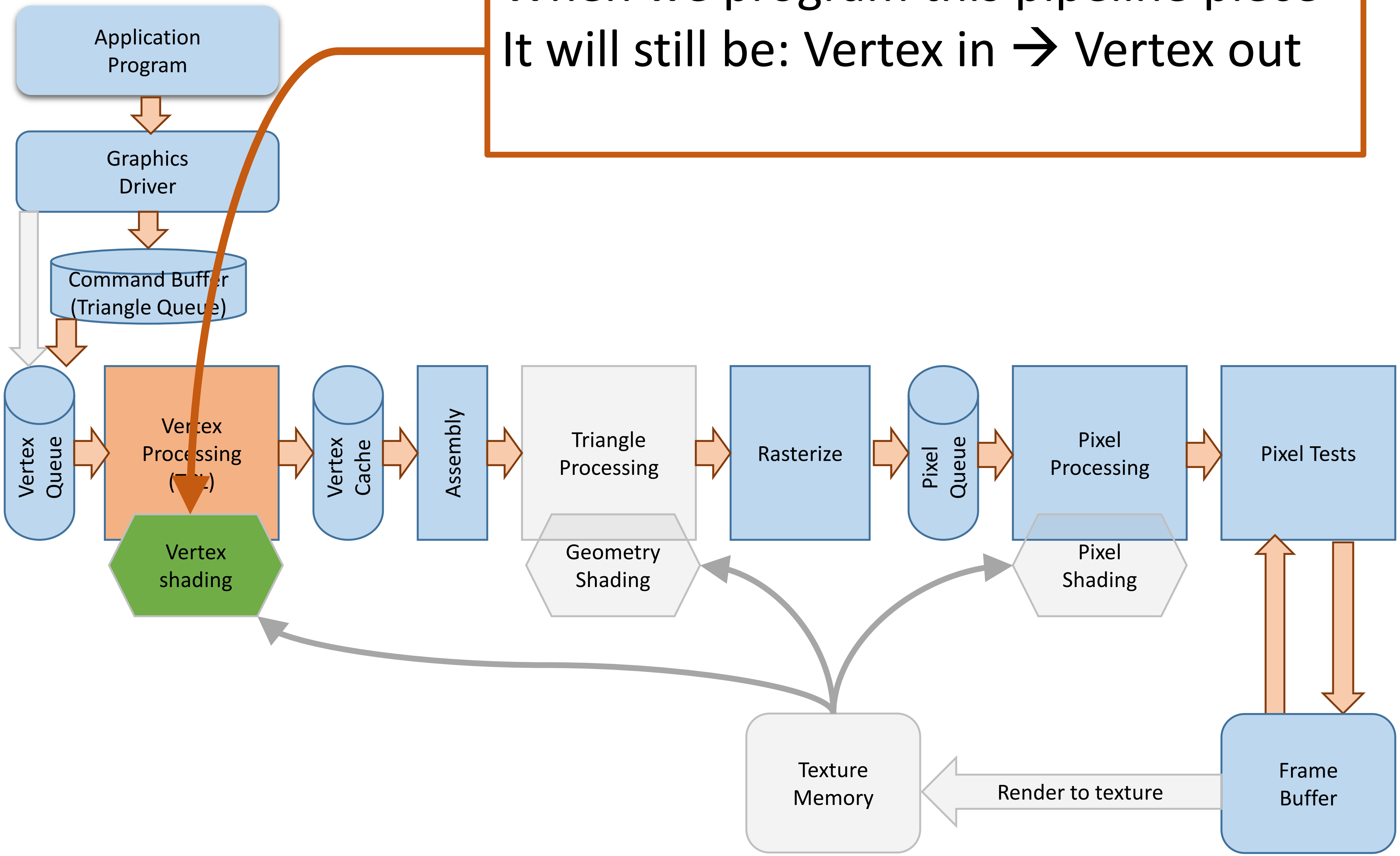
Outputs are:

vertex information for **this vertex**

Looking ahead…

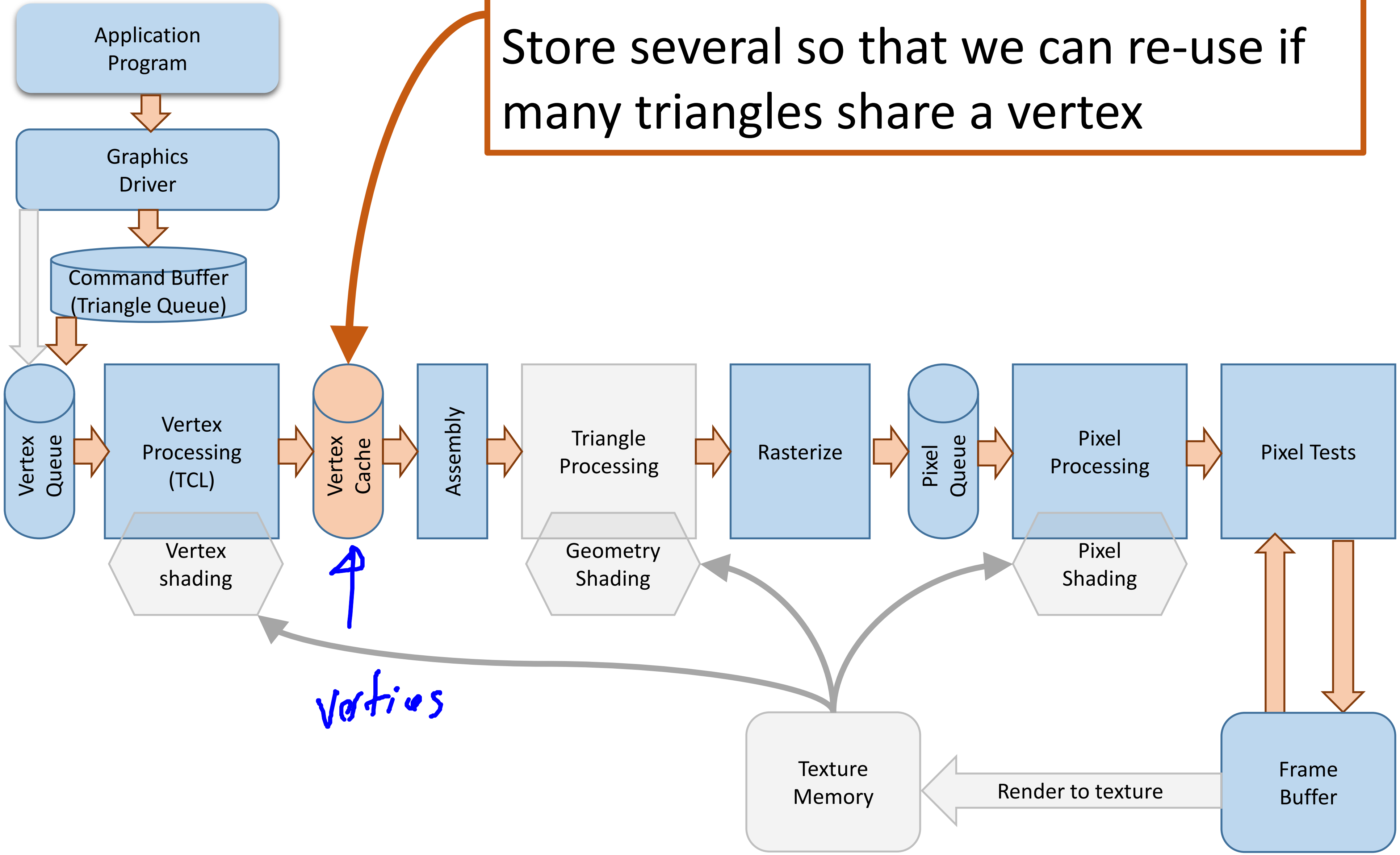
When we program this pipeline piece

It will still be: Vertex in → Vertex out



Store processed vertices in a **cache**

Store several so that we can re-use if many triangles share a vertex



Vertex Caching

Old days:

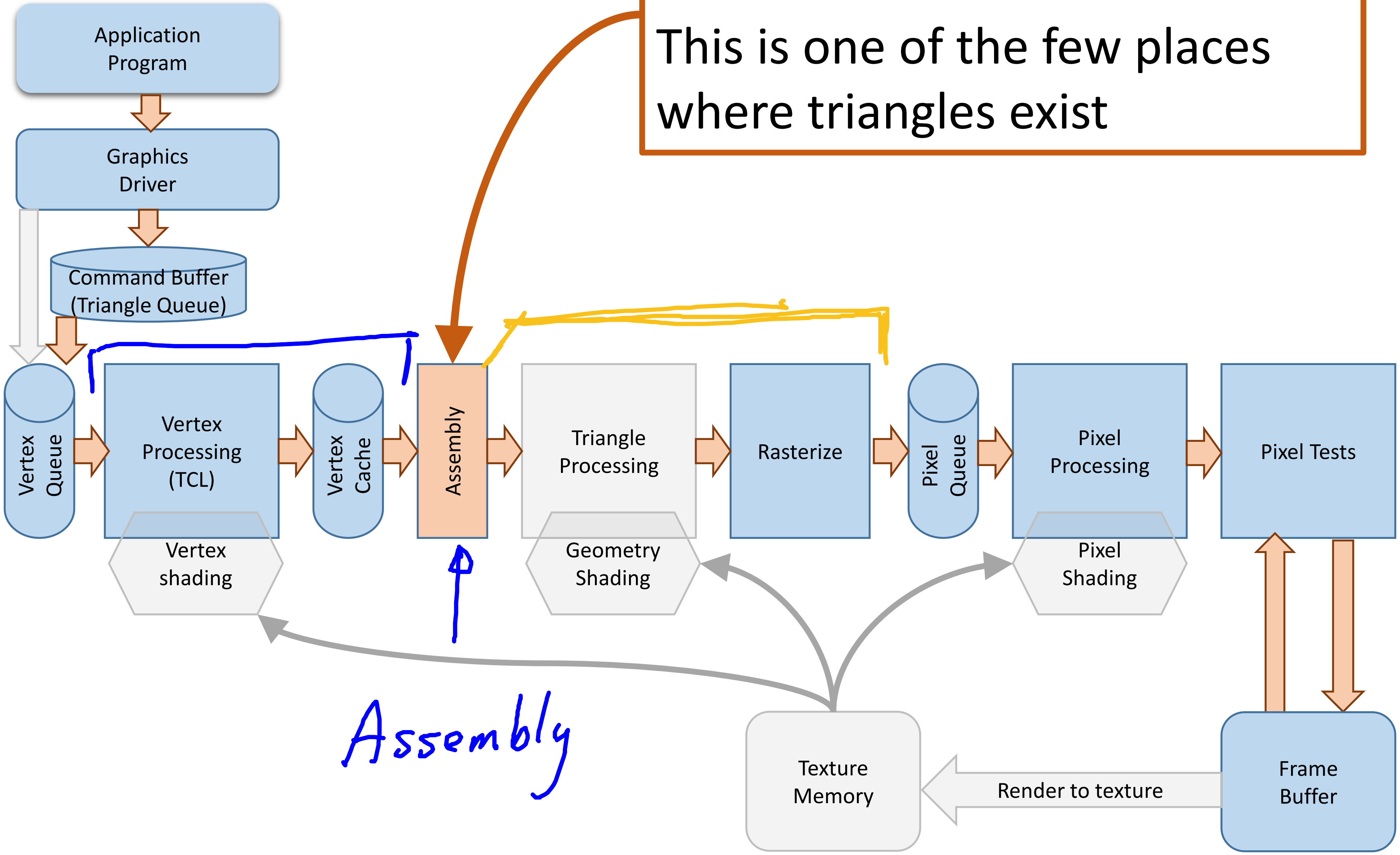
Big deal, important for performance

Now:

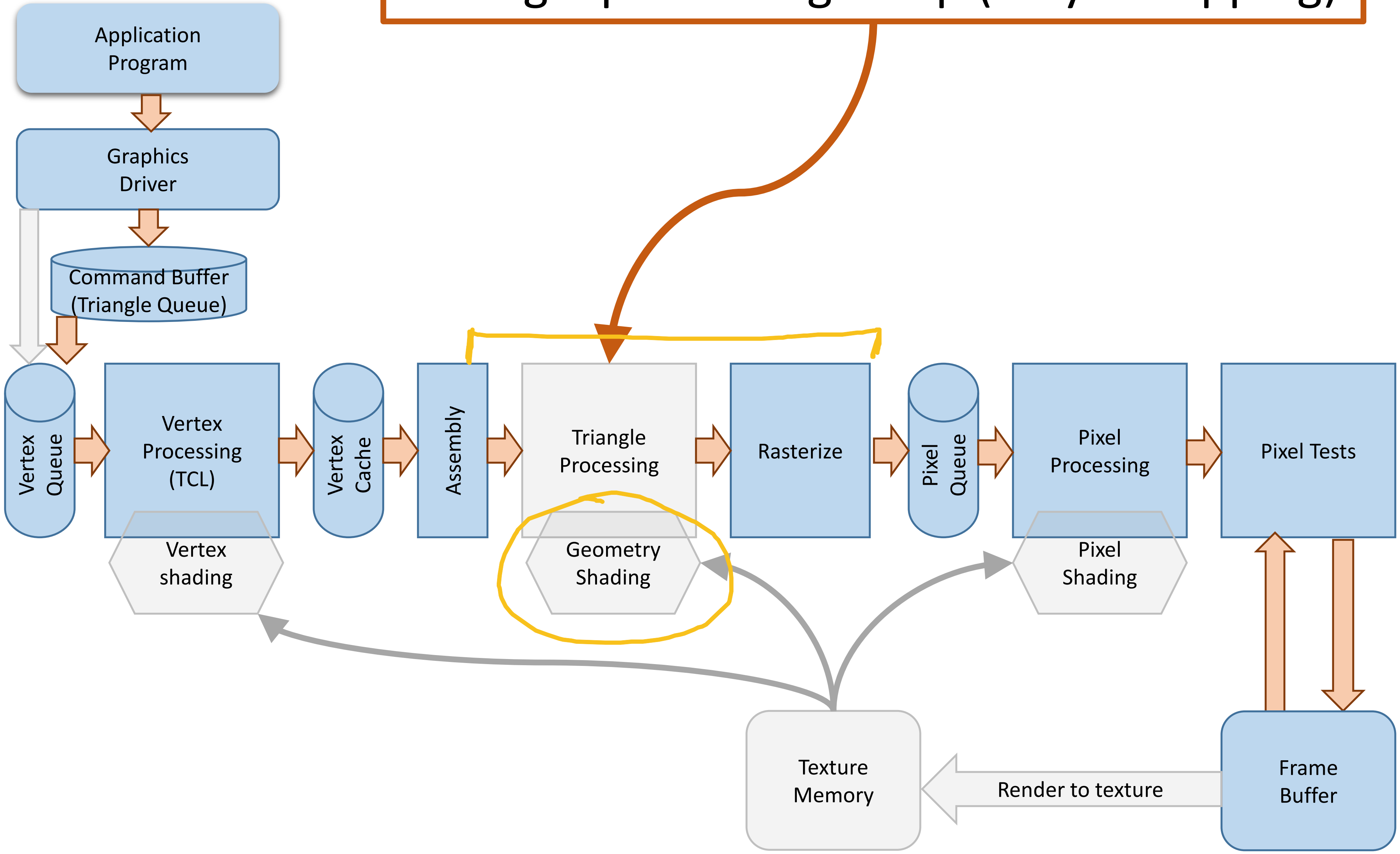
Not even sure that it's always done

Put triangles back together

This is one of the few places where triangles exist



In the fixed-function pipeline, there is no "triangle processing" step (maybe clipping)

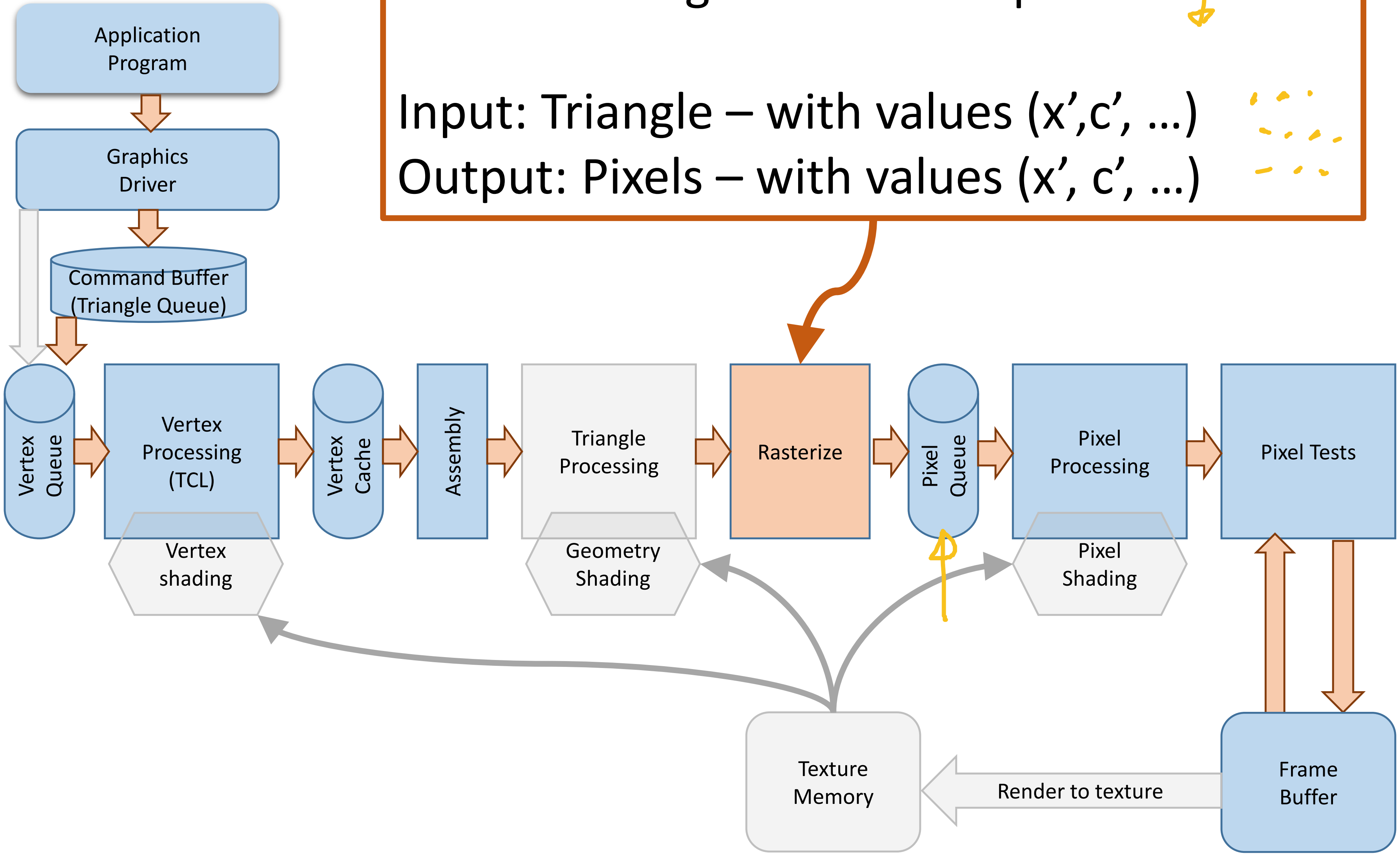


Rasterizer:

Convert triangles to a list of pixels

Input: Triangle – with values (x’,c’, …)

Output: Pixels – with values (x’, c’, …)



Pixels or Fragments

I am using the terms interchangably

(actually, today I am using pixel)

Technically...

Pixel = a dot on the screen

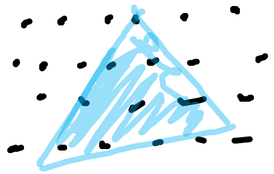
Fragment = a dot on a triangle

might not become a pixel (fails z-test)

might only be part of a pixel

Rasterization

Figure out which pixels a primitive "covers"



Turns primitives into pixels

Interpolate vertex values to interior pixels

Barycentric Coordinates

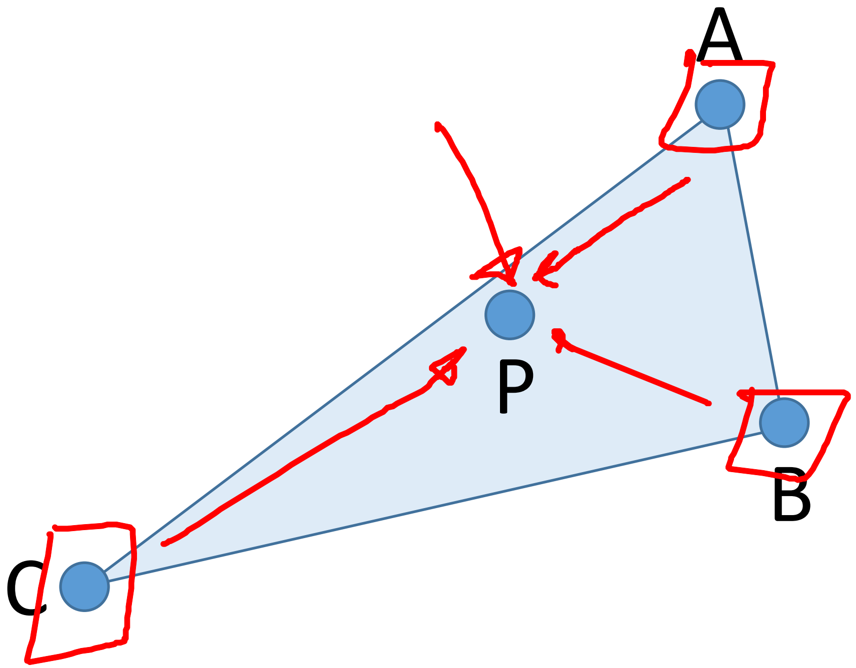
Any point in the plane is a convex combination of the vertices of the triangle

$P = \alpha A + \beta B + \gamma C$

$\alpha + \beta + \gamma = 1$

Inside triangle

$0 <= \alpha, \beta, \gamma <= 1$



Where do pixel values come from?

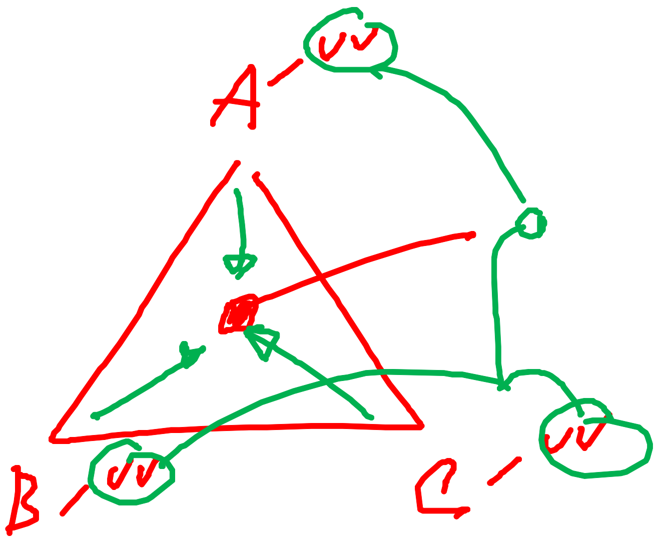
Each vertex has values

Each pixel comes from 3 vertices

Pixels interpolate their vertices' values

Barycentric interpolation

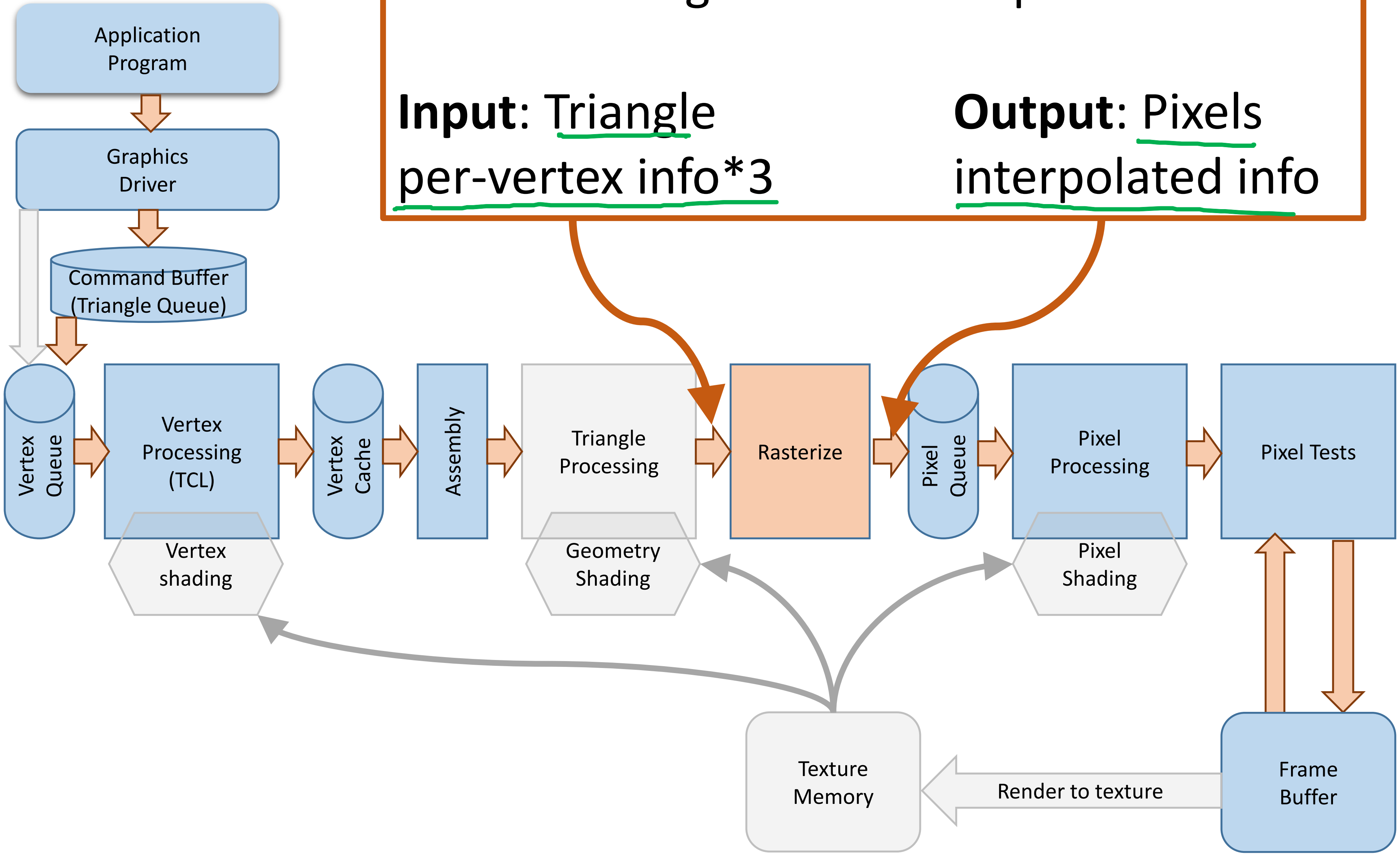
All values (in a pixel) are interpolated



Rasterizer:
Convert triangles to a list of pixels

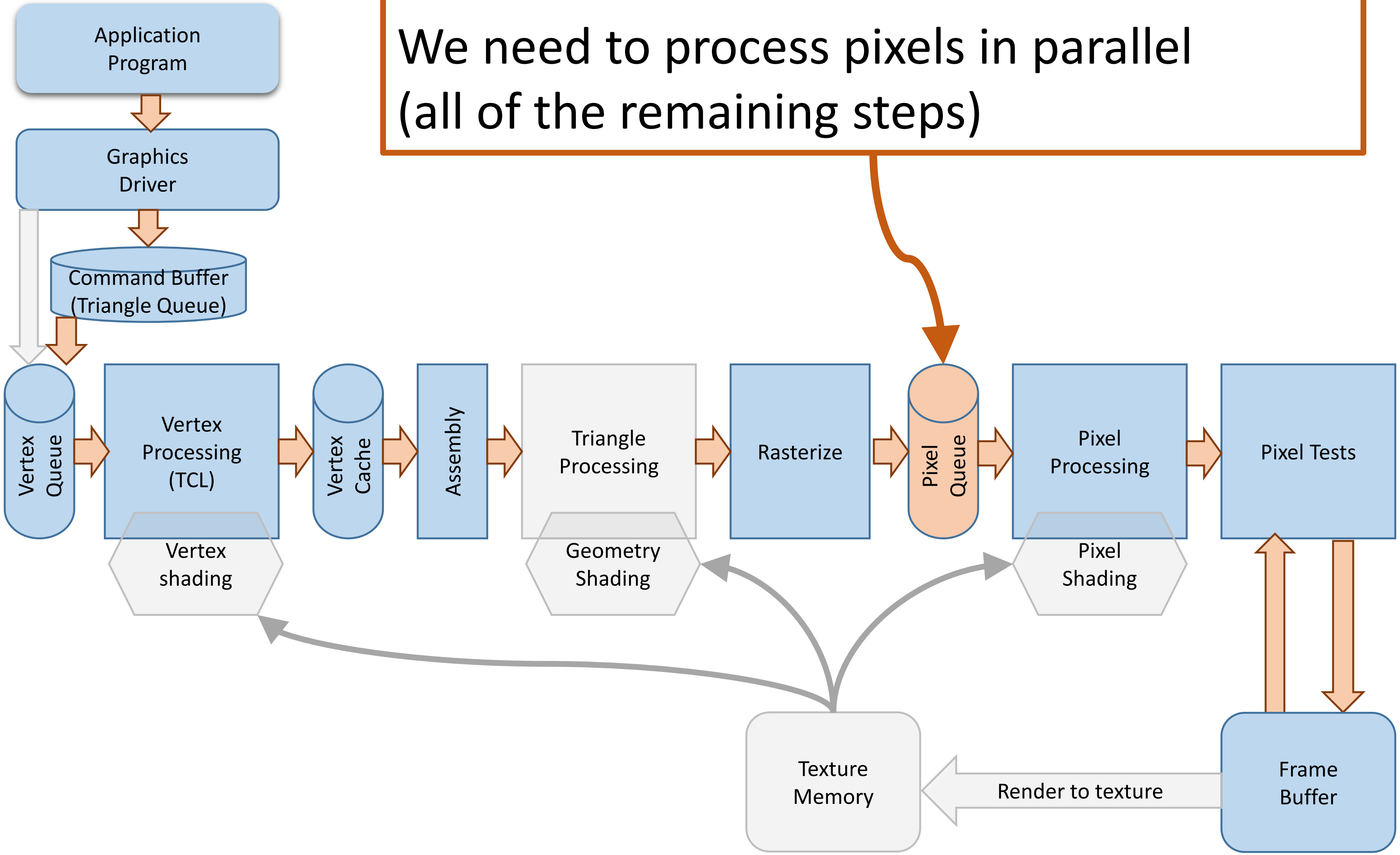
Input: Triangle
per-vertex info*3

Output: Pixels
interpolated info



Each triangle can make lots of pixels

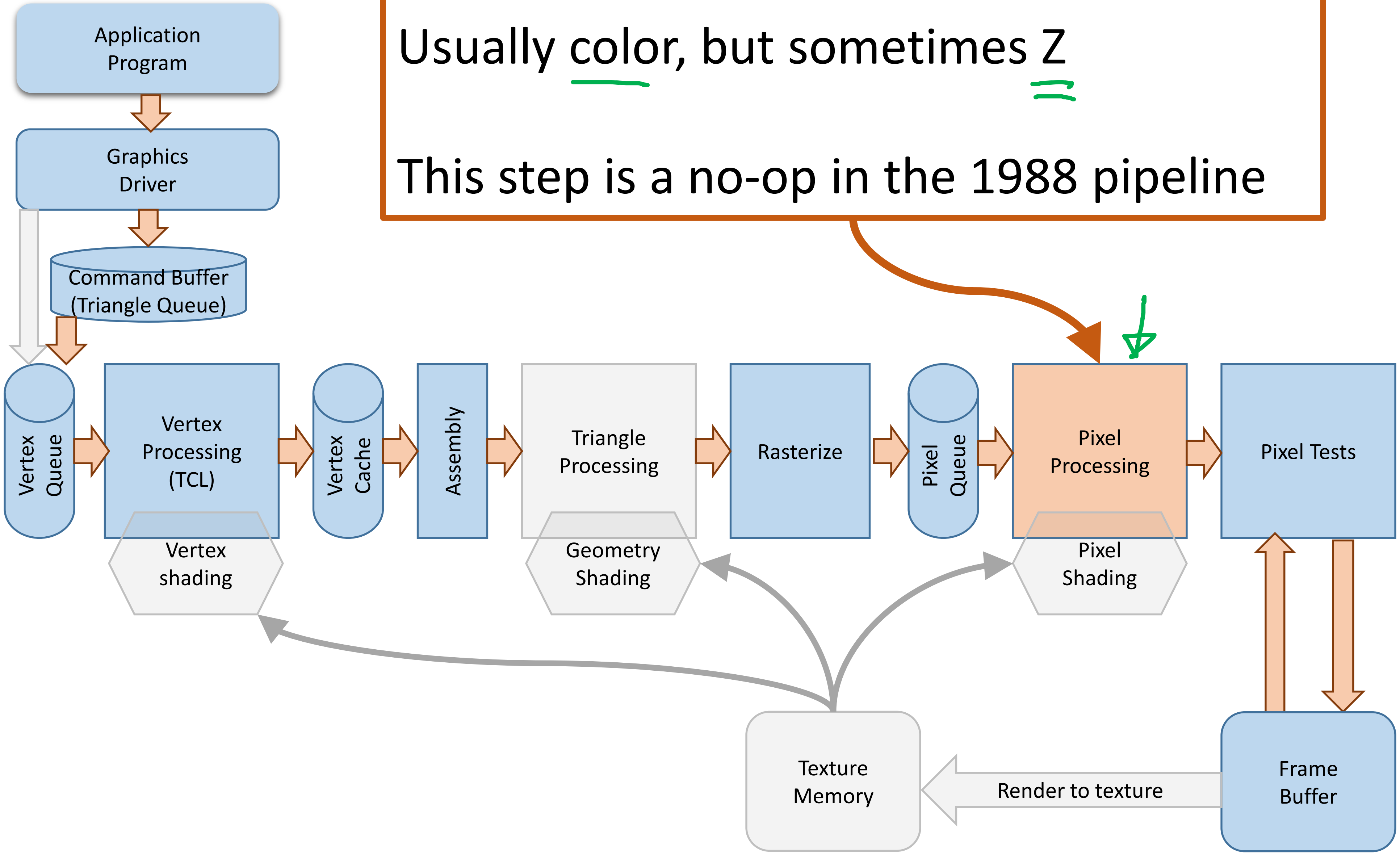
We need to process pixels in parallel
(all of the remaining steps)



Process each pixel to get its final values

Usually color, but sometimes Z

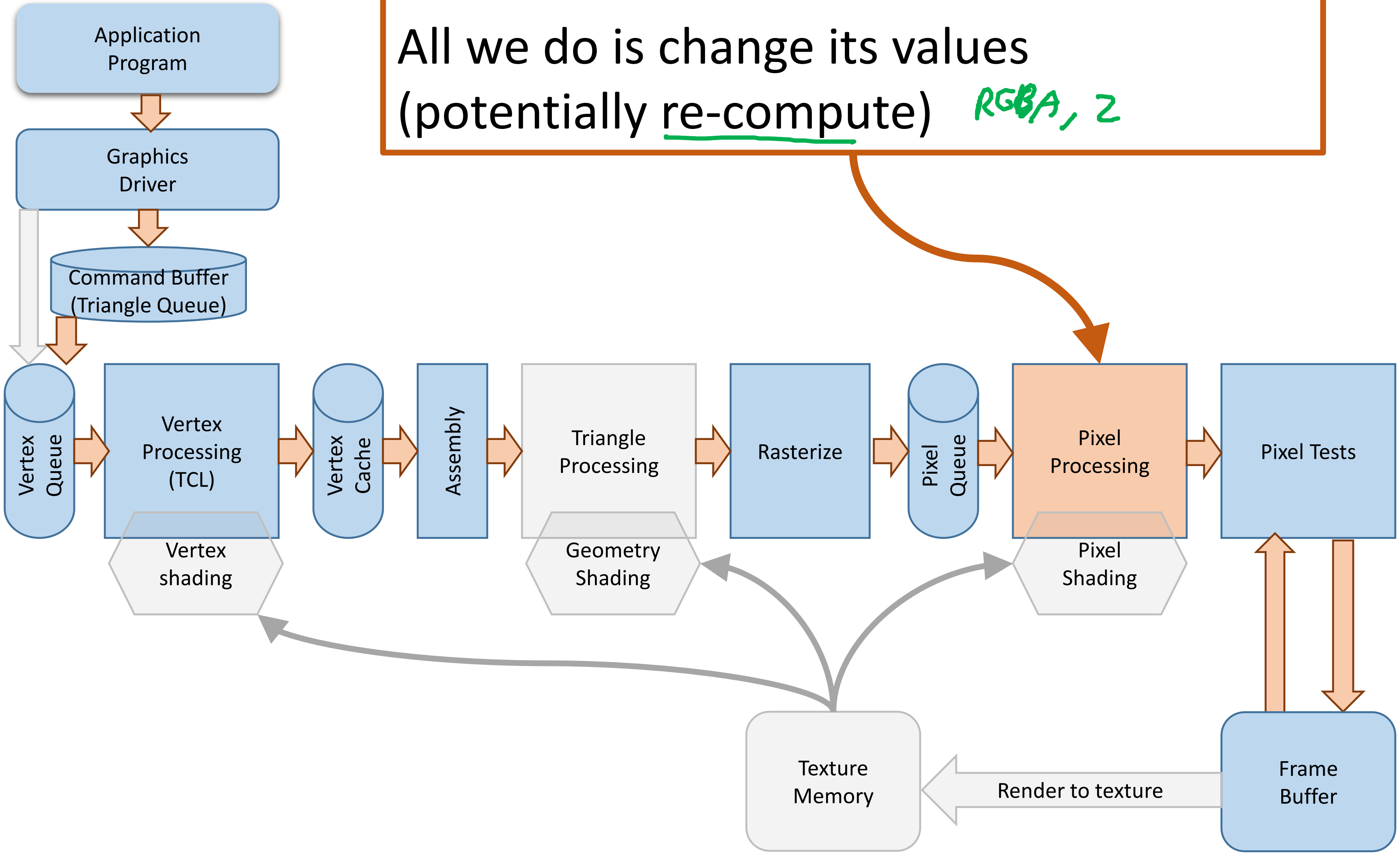
This step is a no-op in the 1988 pipeline



Pixel in → Pixel out, each independent

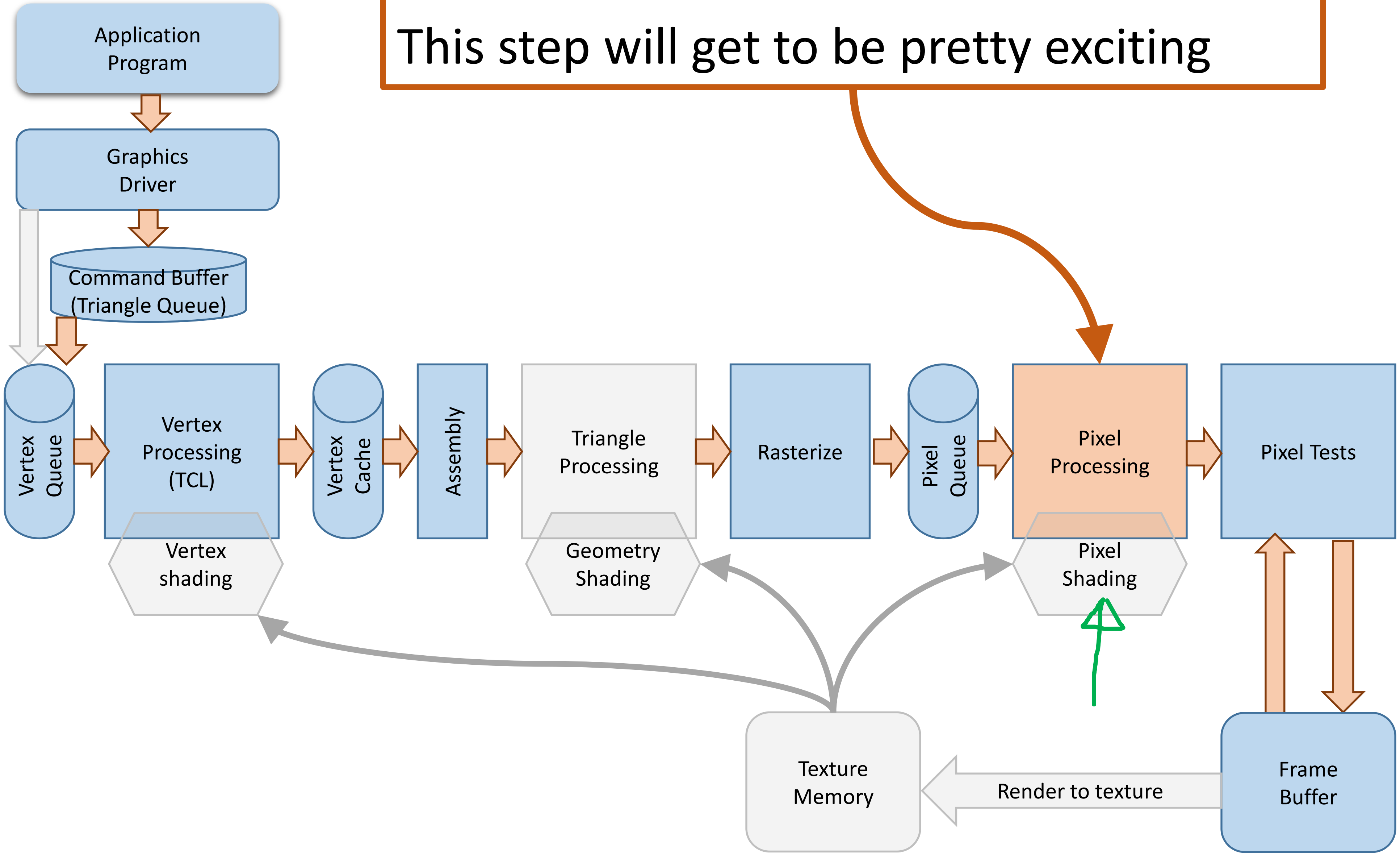
All we do is change its values

(potentially re-compute) RGBA, Z



Coming attractions…

This step will get to be pretty exciting



Pixel Processing Ground Rules

Pixels are independent

Pixel in → Pixel out

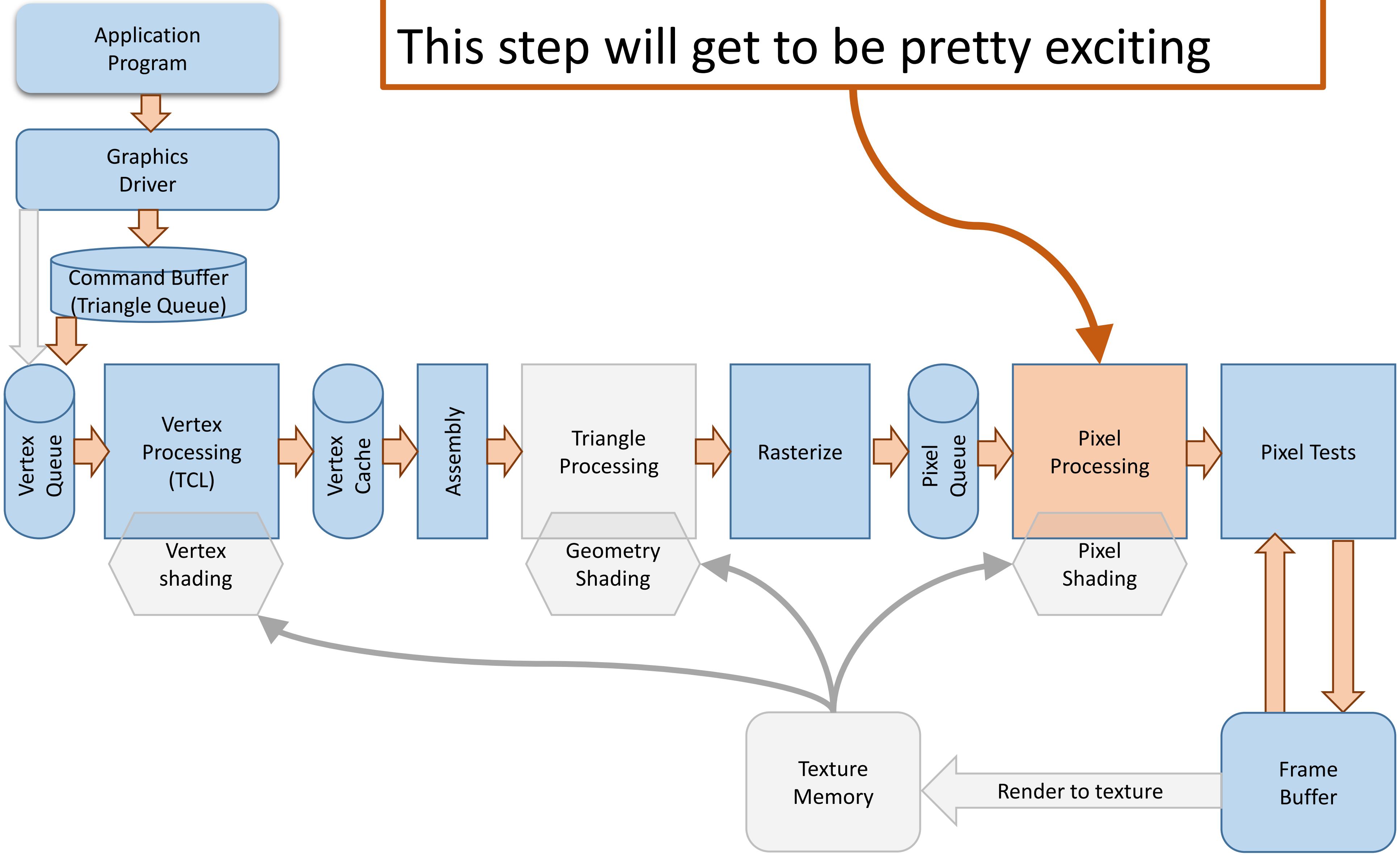
Changing its position (x,y) makes it a different pixel (so you can't)

Can change other values color, z

Or "reject"

Coming attractions…

This step will get to be pretty exciting



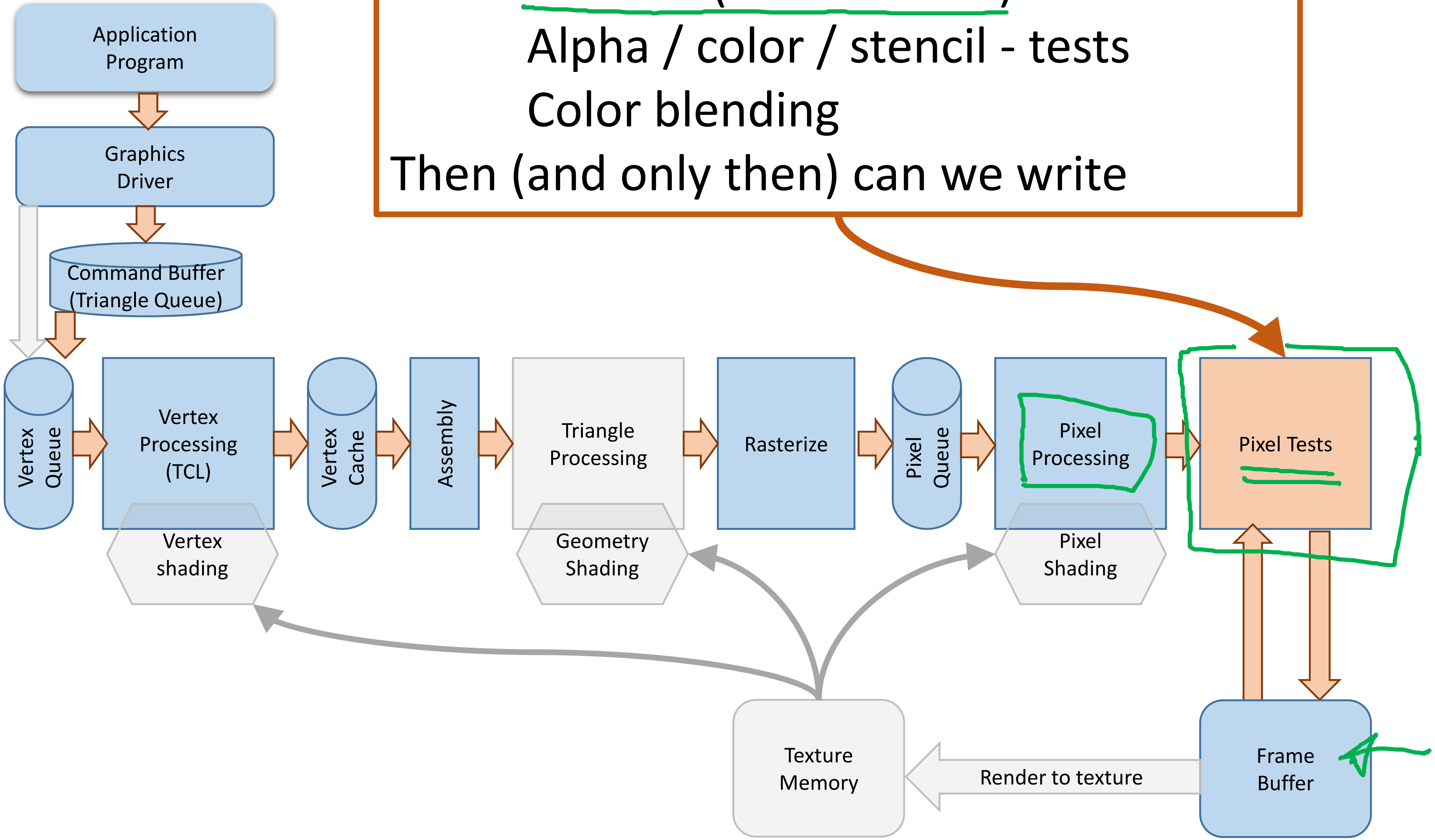
Consider the pixel and it's destination

Z-buffer (is it in front?)

Alpha / color / stencil - tests

Color blending

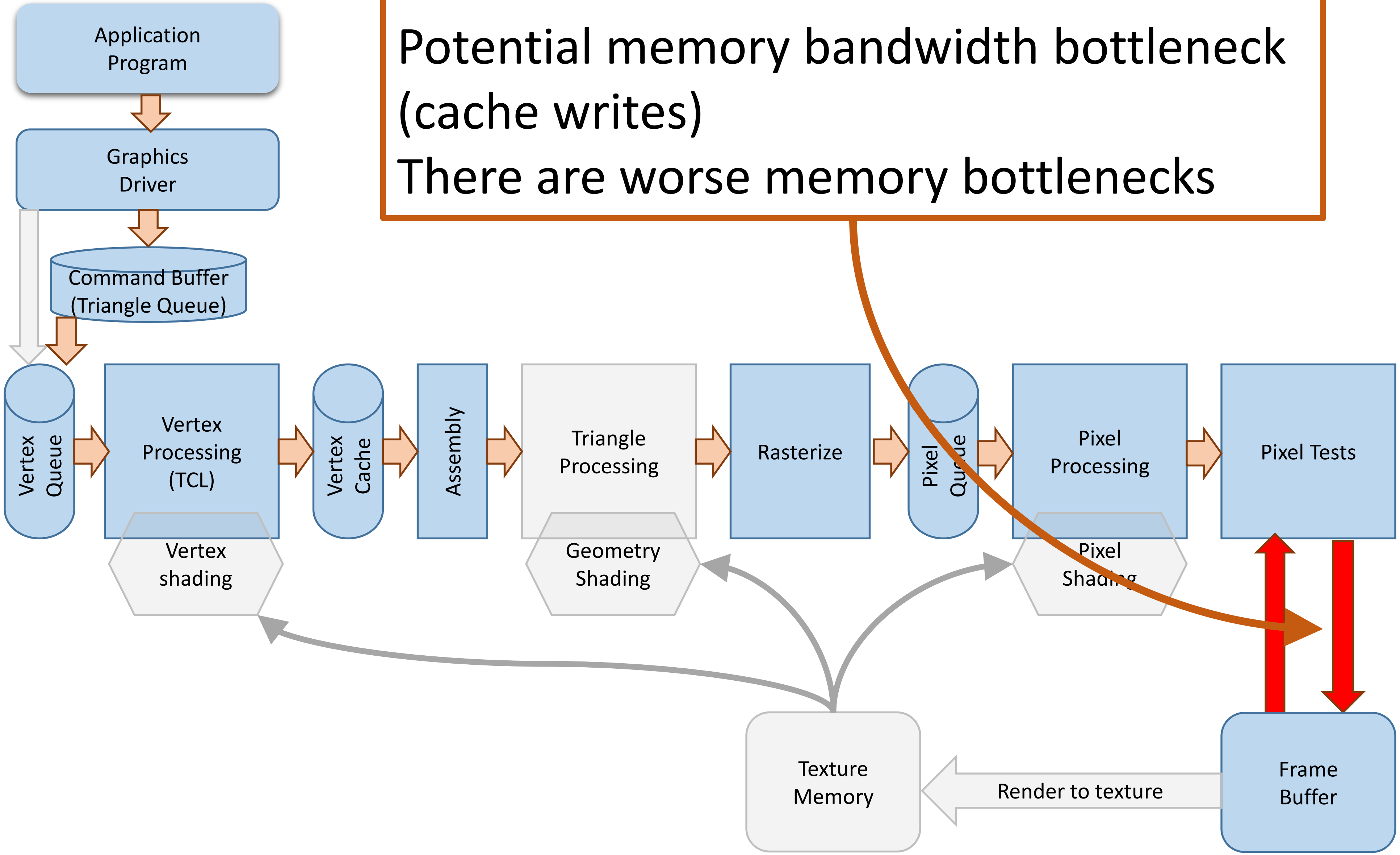
Then (and only then) can we write



Each pixel requires a read/write cycle

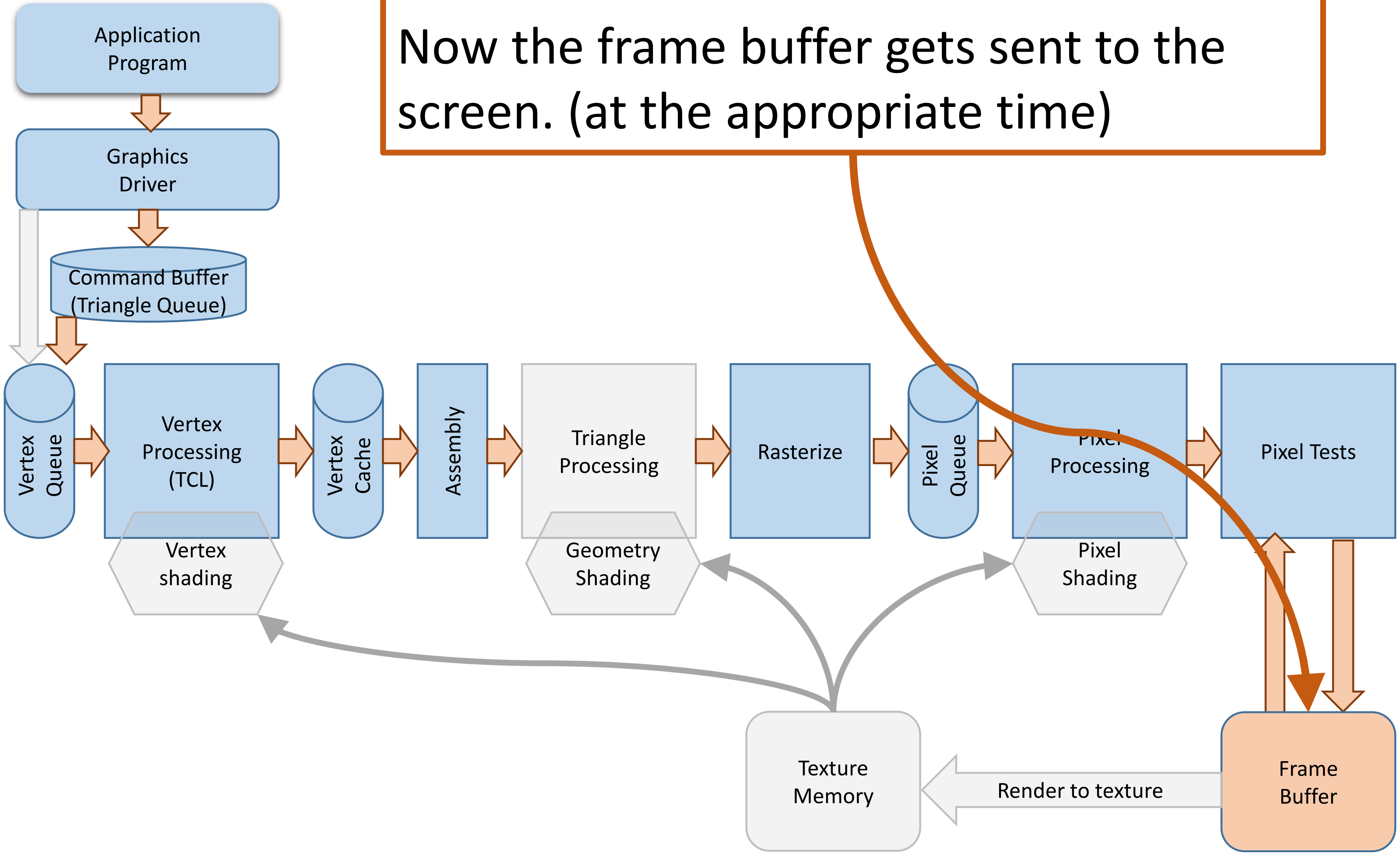
Potential memory bandwidth bottleneck (cache writes)

There are worse memory bottlenecks



We’ve made it!

Now the frame buffer gets sent to the screen. (at the appropriate time)



What if we didn't make it...

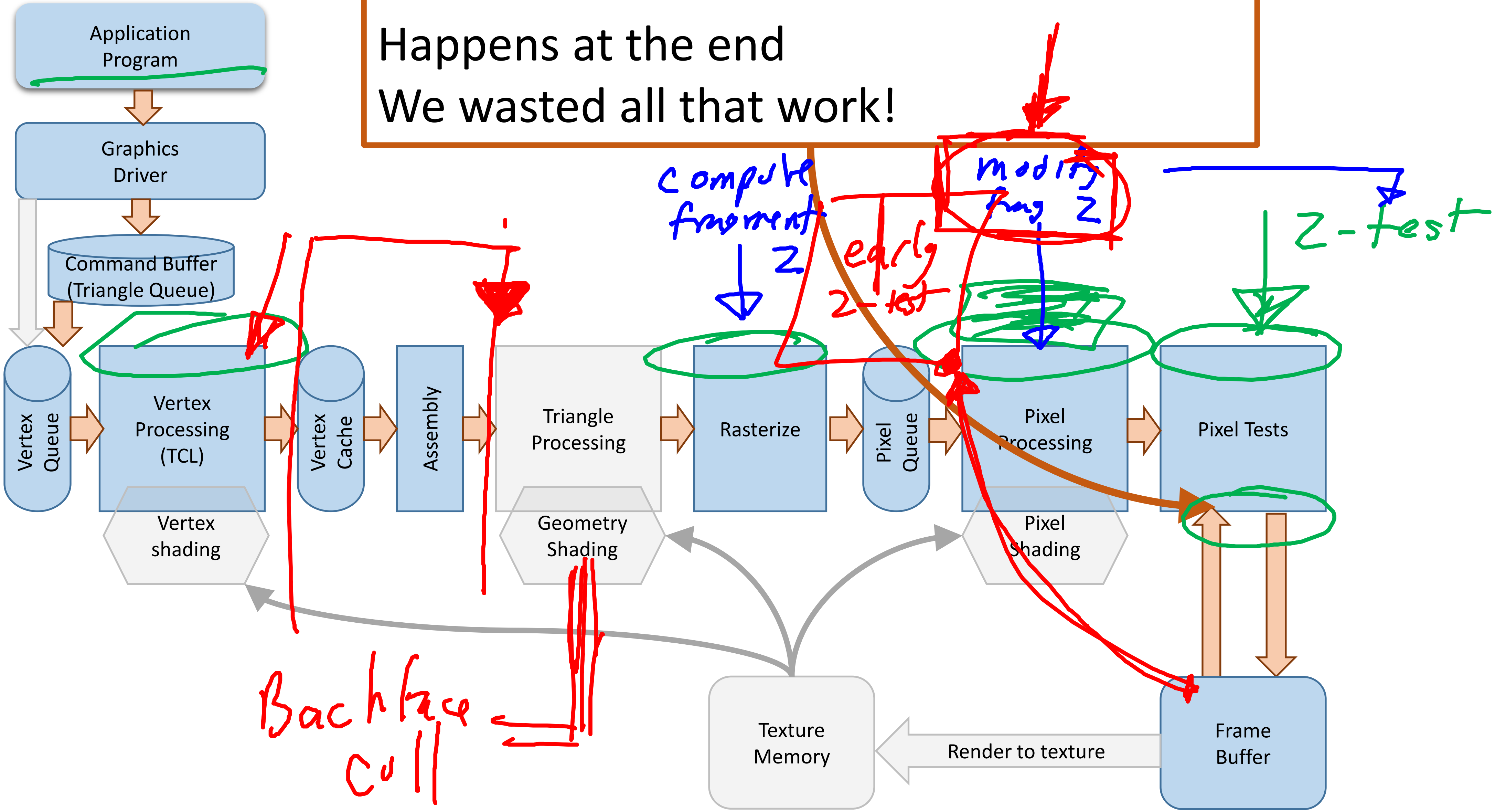
Suppose the triangle's pixels are occluded

Removed by the z-buffer

Normal Z-Test

Happens at the end

We wasted all that work!



Summary: A Triangle's Journey

Graphics happens as a series of steps (Pipeline)

Tr Raster Pixel Pixel Test

Transfer information about vertices from application

Process vertices

Assemble / Rasterize Triangles

Process fragments/pixels

Test/write pixel results

What's next? Program these steps!

If you understand what the boxes do...

vert proc

You can program them! (Shaders)

pixel proc

fragment

Efficient transfer of data from application

Program the Vertex Processor

Program the Fragment Processor