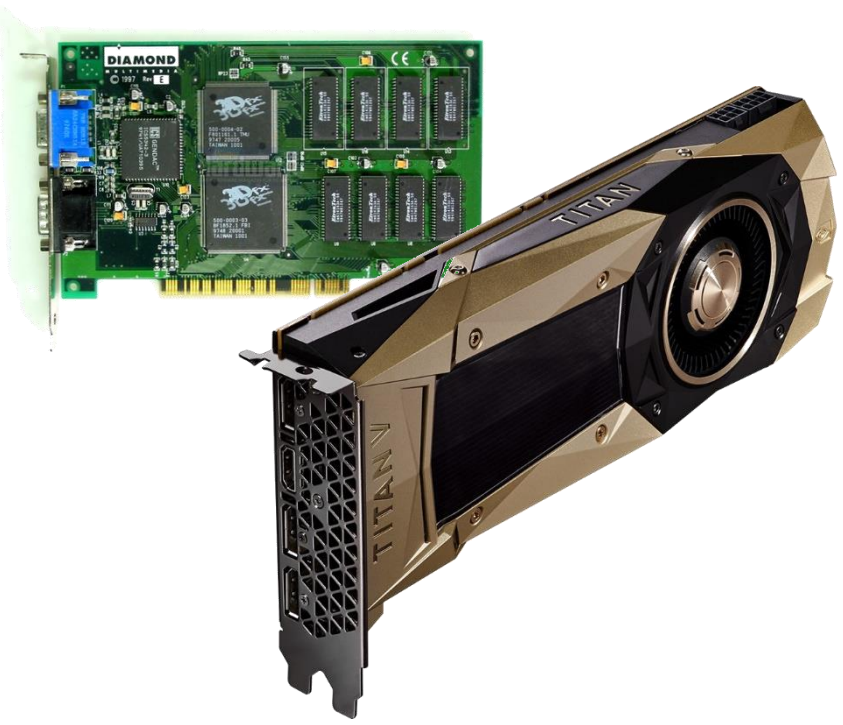
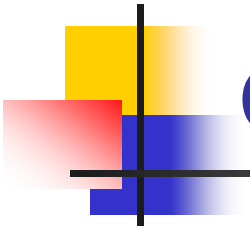
Graphics hardware

Robert G. Belleman
Universiteit van Amsterdam

FCG CH17 - Using Graphics Hardware

Met materiaal van:
- Kurt Akeley, Stanford University
- Tamara Munzner, University of British Columbia
- Jorik Blaas, TU Delft





Overzicht

- Graphics performance
- Real-time/interactive graphics
- Programmeerbare graphics hardware
- General Purpose computing on GPUs: GPGPU

History of graphics hardware

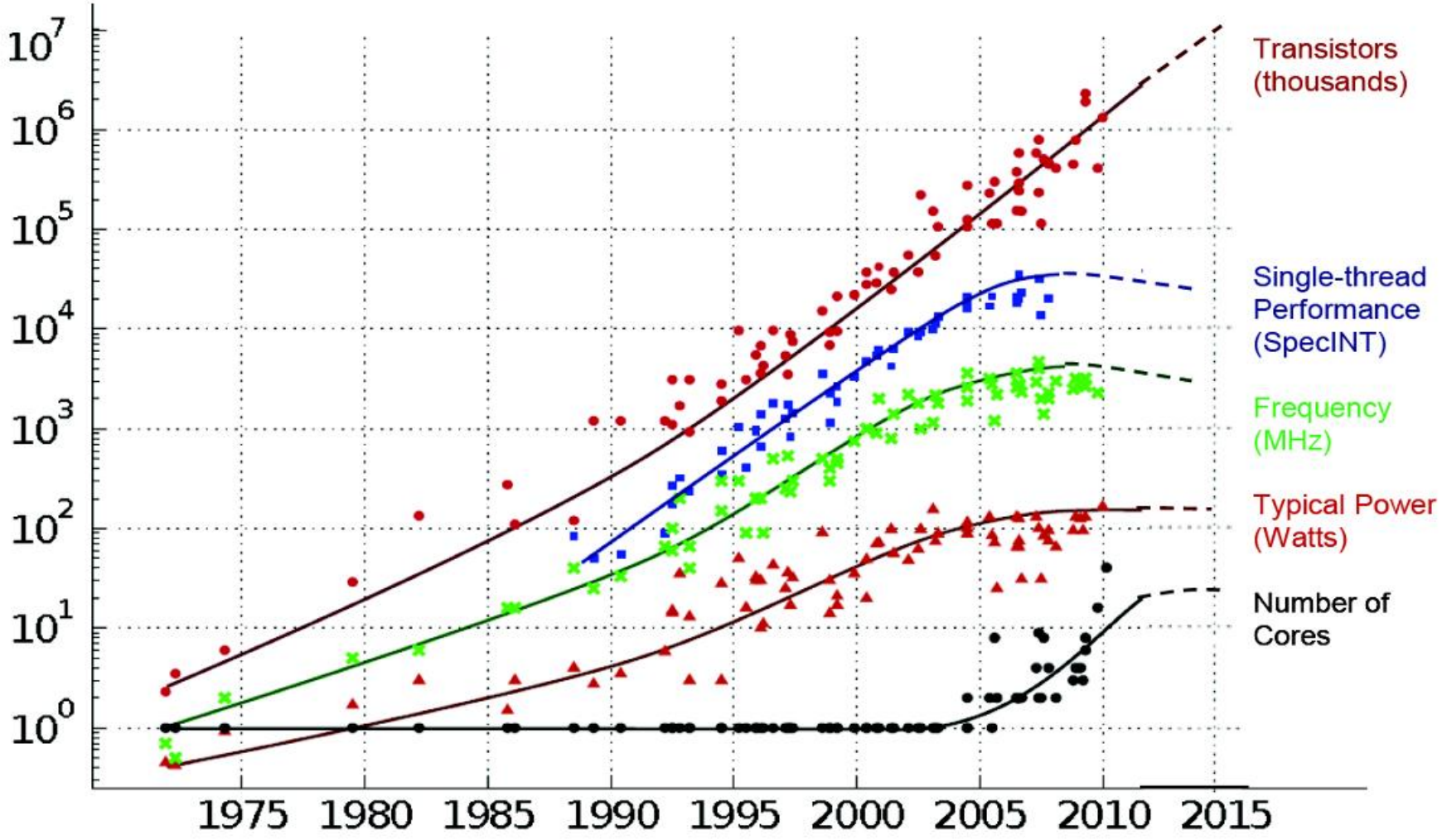
- 1950: MIT Whirlwind (CRT)
- 1955: Sage, Radar with CRT and light pen
- 1958: Willy Higinbotham “Tennis for Two”
- 1960: MIT „Spacewar“ on DEC PDP-1
- 1963: Ivan Sutherland‘s „Sketchpad“ (CAD)
- 1969: ACM Siggraph founded
- 1968: Tektronix storage tube ($5-10.000)
- 1968: Evans&Sutherland (flight simulators) founded
- 1968: Douglas Engelbart: computer mouse (“mother of all demos”)
- 1970: Xerox GUI (inspiration to Jobs & Wozniak for the Apple Macintosh)
- 1971: Gouraud shading
- 1974: Z-buffer
- 1975: Phong shading model
- 1976: First animations rendered
- 1979: Eurographics founded
- 1980: Whitted: Ray tracing
- 1981: Apollo Workstation, IBM PC
- 1982: Silicon Graphics (SGI) founded
- 1984: X Window System, Apple Macintosh
- 1984: First Silicon Graphics Workstations (IRIS GL, precursor to OpenGL)
- Until mid/end of 1990s: Dominance of SGI in the high end
  - HW: RealityEngine, InfiniteReality, RealityMonster, ...
  - SW: OpenGL, OpenInventor, Performer, Digital Media Libs, ...
- End of 1990s:
  - Low- to mid range taken over by “PCs” (Nvidia, ATI, ...)
    - HW: Fast development cycles, Graphics-on-a-chip, ...
  - SW: Direct 3D & OpenGL, computer games
- 1995: First feature film “Toy Story”
- Today:
  - Programmable graphics hardware, Cg, CUDA
  - Realtime Ray Tracing

NVIDIA performance trends

Year	Product	Tri rate	incr	Tex rate	incr
1998	Riva ZX	3m	-	100m	-
1999	Riva TNT2	9m	3.0	350m	3.5
2000	GeForce2 GTS	25m	2.8	664m	1.9
2001	GeForce3	30m	1.2	800m	1.2
2002	GeForce Ti 4600	60m	2.0	1200m	1.5
2003	GeForce FX	167m	2.8	2000m	1.7
2004	GeForce 6800 Ultra	170m	1.0	6800m	2.7
2005	GeForce 7800 GTX	215m	1.2	6800m	1.0
2006	GeForce 7900 GTX	260m	1.3	15600m	2.3
2007	GeForce 8800 Ultra	306m	1.2	39200m	2.5
			1.7		1.9

Yearly Growth is well above 1.5 (Moore's Law)

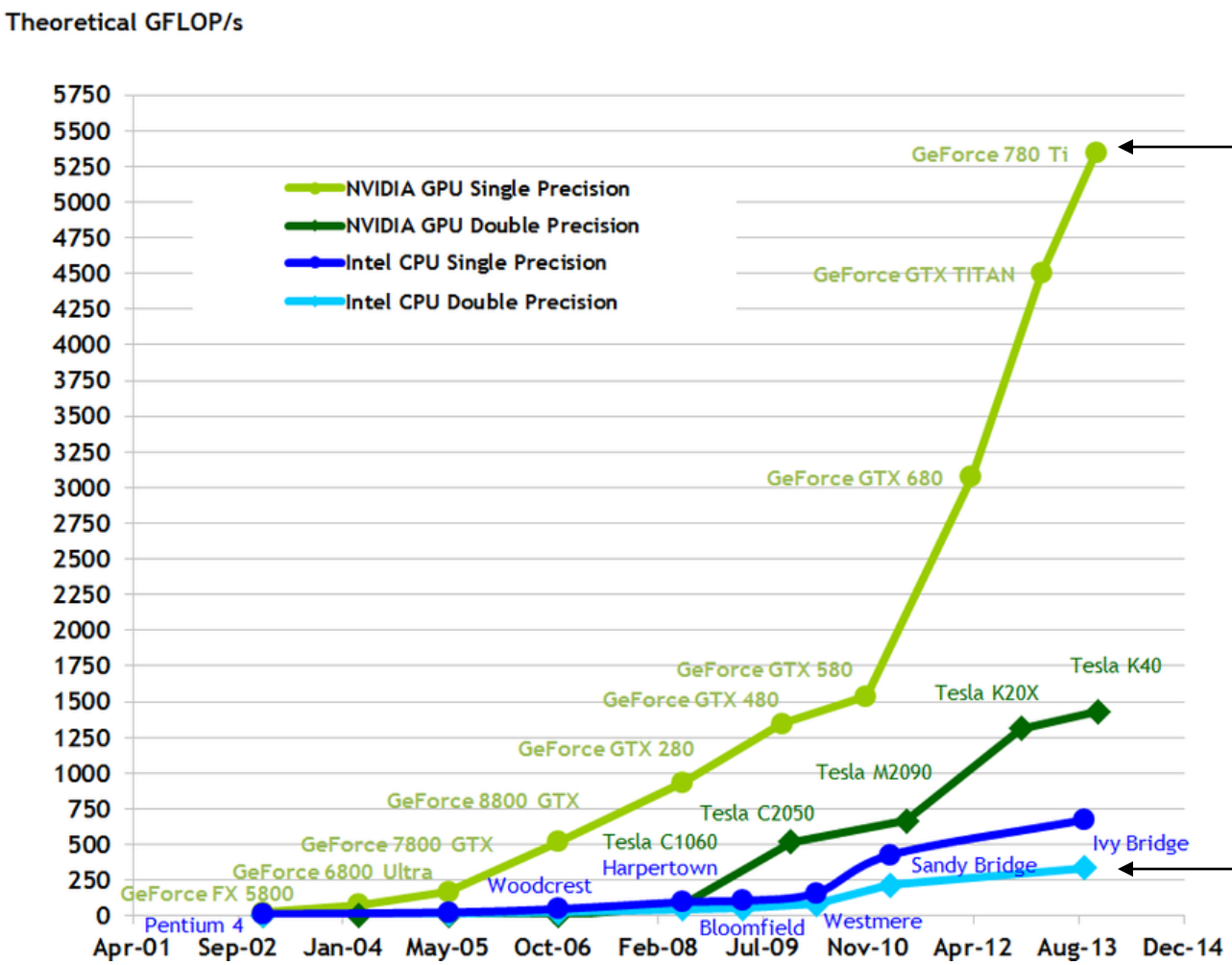
35 YEARS OF MICROPROCESSOR TREND DATA



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Performance vergelijking CPU-GPU

Floating-point peak performance



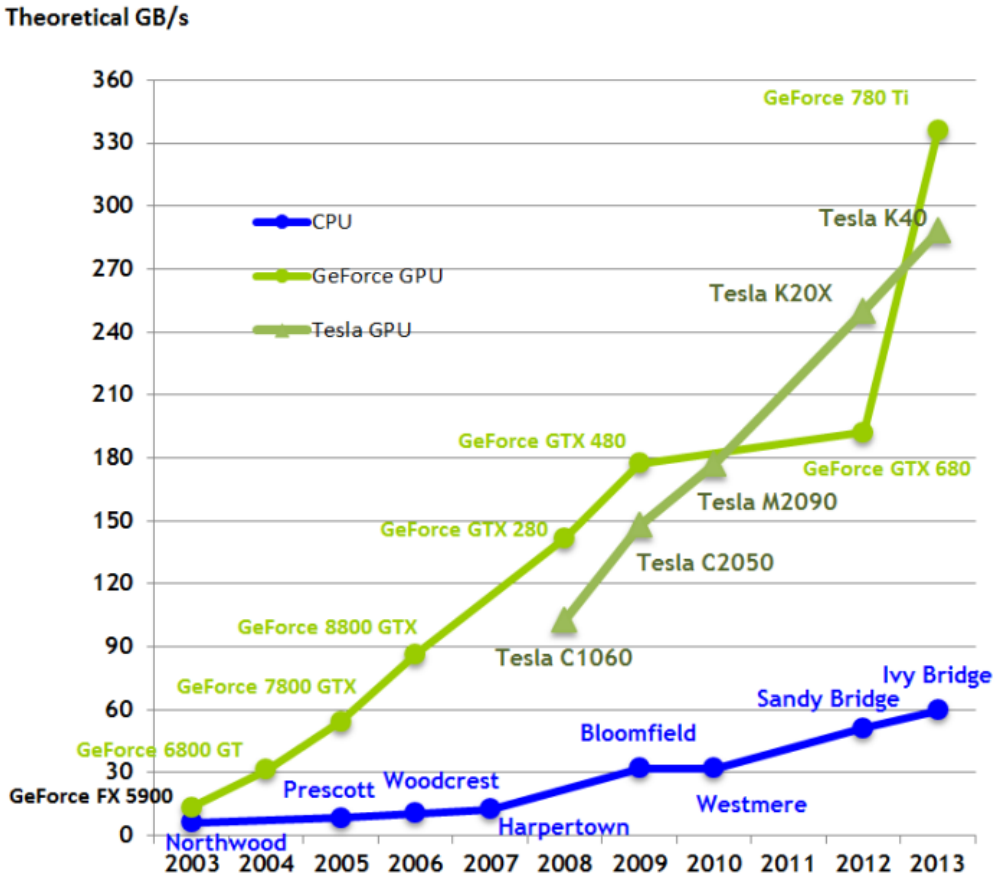
GPU: verdubbeling elke 9 maanden

CPU: verdubbeling elke 18 maanden (Moore's law)

Source: NVIDIA Corp. "CUDA C Programming Guide"

Performance vergelijking CPU-GPU

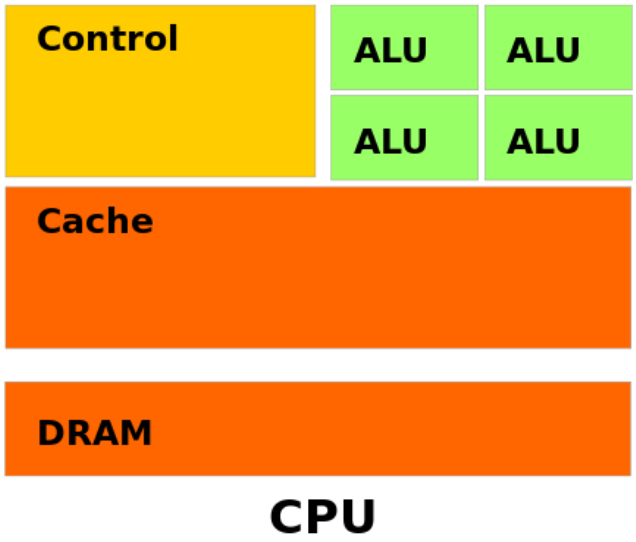
Memory bandwidth

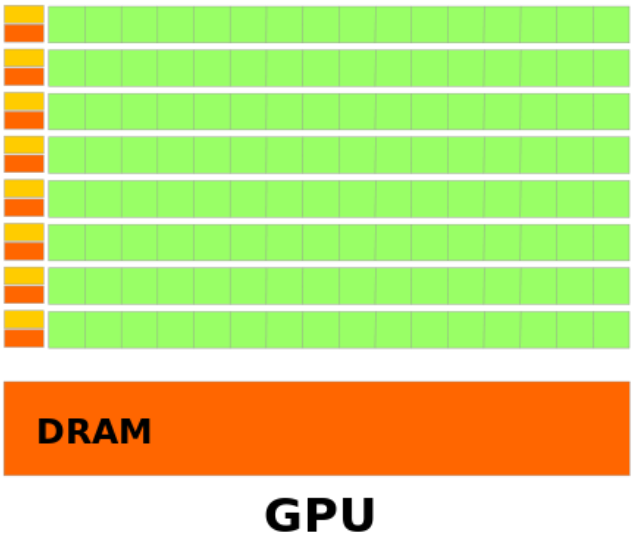


Source: NVIDIA Corp. “CUDA C Programming”

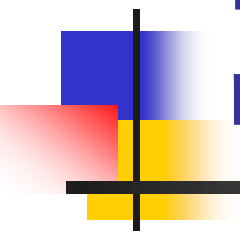
Comparison GPU vs. CPU

- Modern GPUs contain more transistors than CPUs
  - In GPUs most of these are used for calculations
  - In CPUs a significant number is used for flow control, memory/cache management

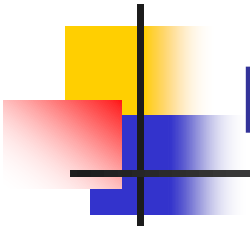




Hardware support towards real-time/interactive graphics



- Optimizations:
  1. Graphics in the PC architecture
  2. Graphics state machine
  3. Line/triangle/quad strips, triangle fan
  4. Display lists
  5. Vertex arrays
  6. Vertex Buffer Objects (VBO)
  7. Data and task parallelism
- Programmable Graphics hardware



Real-time/interactive graphics

- "Real-time": optimize frame rate and refresh rate
  - Increase throughput: the number of frames per second (fps)
    - >20 fps for perception of (jerky) motion
    - >60 fps for smooth motion
    - >50 Hz refresh rate to reduce eye strain due to flickering
- "Interactive": optimize response time
  - Decrease latency: the time lapse between cause and effect
    - Goal in typical user interfaces: < 100 ms

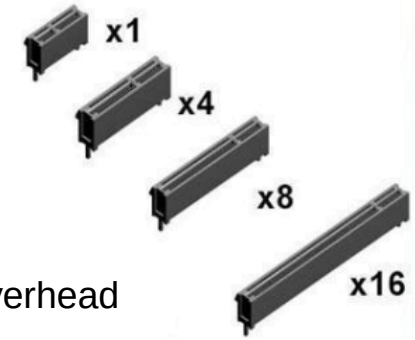
The gaming industry has been the driving force in increasing frame rate and decreasing latency of PC graphics hardware.

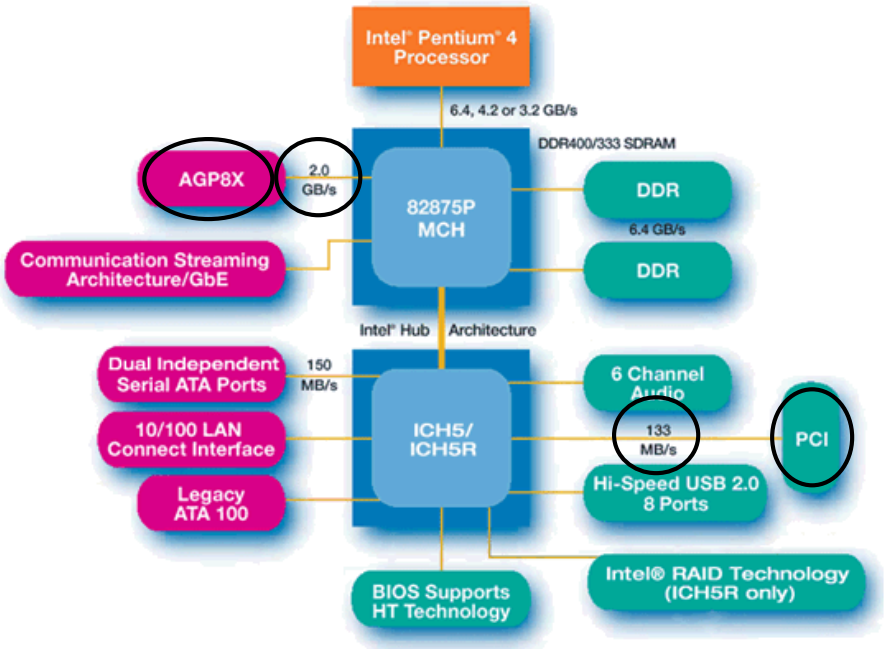
1. Graphics in the PC architecture

- PCI
  - 133 MB/s
- AGP
  - 1x: 266 MB/s
  - 2x: 533 MB/s
  - 4x: 1066 MB/s
  - 8x: 2133 MB/s

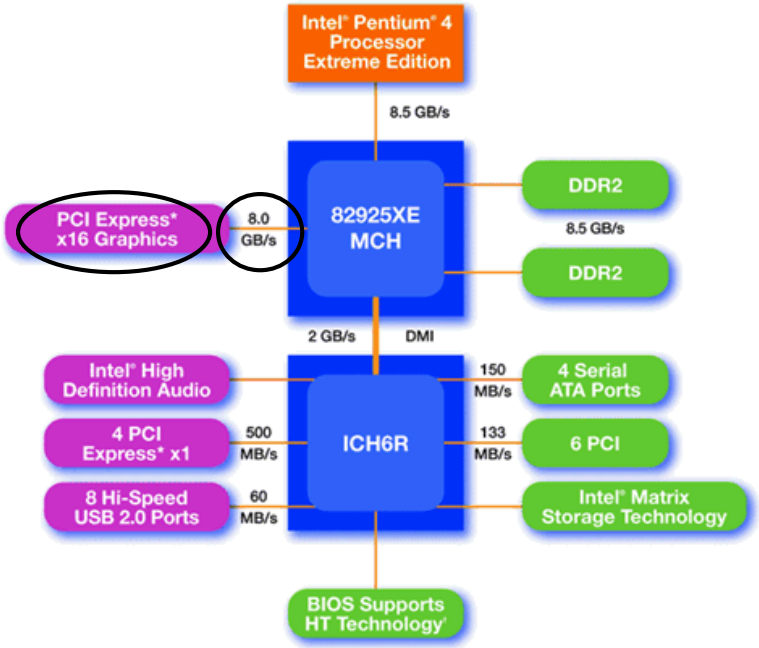
Asymetric!
(CPU to GPU is fastest)

- PCI Express (PCIe)
  per lane (each direction):
  - v1.x: 250 MB/s
  - v2.x: 500 MB/s
  - v3.0: 1.0 GB/s
  - v4.0: 2.0 GB/s
- 16 lane v3.0 ~ 8GB/s
  - Without protocol overhead
  - Realistic: 4 - 6GB/s

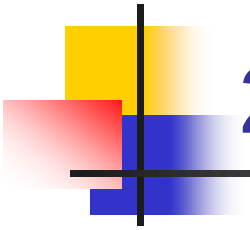




old



new(ish)



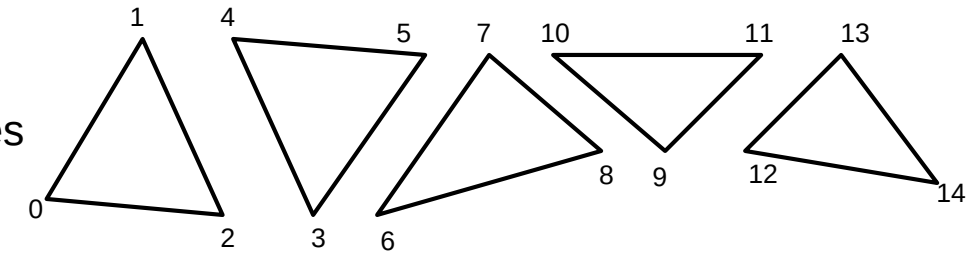
2. Graphics state machine

- Optimization method: “graphics state” defines default characteristics
  - No need to specify everything for every individual element; graphics state is used to “fill in the blanks”
- Included in graphics state:
  - Draw mode (points, lines, polygons, shading mode, ...)
  - Per vertex attributes (color, normal, texture coordinate)
  - Primitive type (points, lines, triangles, triangle strip, ...)
- Problem: frequent state changes slow down rendering
  - Therefore it’s a good idea to cluster similar geometry to minimize state changes as much as possible
    - Use data structures, e.g. scene graphs
  - Unfortunately this is often not possible

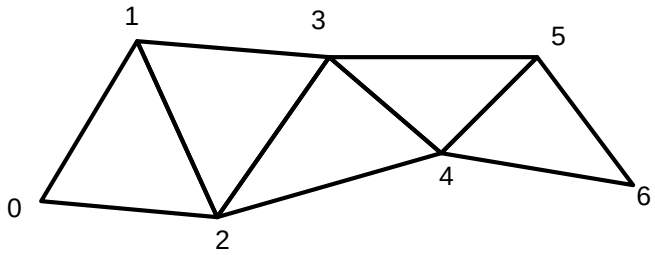
3. Strips and fans

- Optimization method: minimize storage overhead

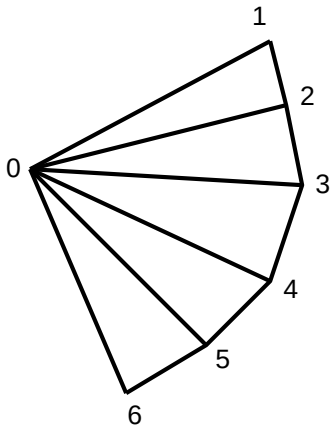
- Triangle list
  - N points define N/3 triangles



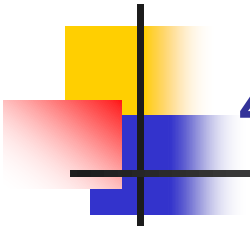
- Triangle strip
  - N points define N-2 triangles



- Triangle fan
  - N points define N-2 triangles



Similar for lines and quads



4. Display lists

- Optimization method: draw commands are compiled and stored in a compact format on the graphics card
  - Note: not *all* commands can be compiled into display lists!
- Only useful for static geometry
  - Any change requires that the display list is recompiled
- OpenGL example:

```
GLuint id = glGenLists(1); // generate display list
glNewList(id, GL_COMPILE); // begin display list
glBegin(GL_POINTS);
  glVertex3i(0, 1, 0);
  glVertex3i(2, 3, 0);
  glVertex3i(4, 5, 0);
  ...
glEnd();
glEndList();               // end display list
...
glCallList(id);            // call display list
```

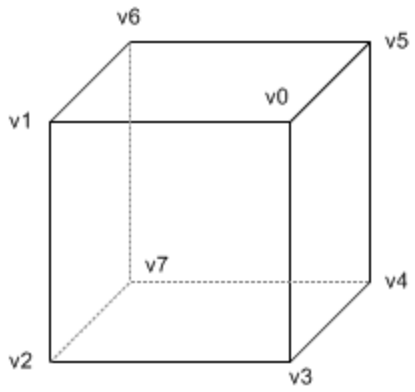
5. Vertex arrays

- Optimization method: aggregate geometry into continuous, coherent memory areas
- Application defines vertex data sequentially in separate arrays:
  - `void VertexPointer ( int size, enum type, sizei stride, void *pointer );`
  - `void NormalPointer ( enum type, sizei stride, void *pointer );`
  - `void ColorPointer ( int size, enum type, sizei stride, void *pointer );`
  - `void TexCoordPointer ( int size, enum type, sizei stride, void *pointer );`
  - `void EdgeFlagPointer ( sizei stride, void *pointer );`
  - `void IndexPointer ( enum type, sizei stride, void *pointer );`
- Note that these arrays are stored *in the application* (“client side”)
  - Elements are accessed when needed, so the contents of the array may be changed
- Each array can be separately enabled/disabled:
  - `void EnableClientState ( enum array );`
  - `void DisableClientState ( enum array );`
  - *array*: `VERTEX_ARRAY`, `NORMAL_ARRAY`, `COLOR_ARRAY`, `TEXTURE_COORD_ARRAY`, `EDGE_FLAG_ARRAY` or `INDEX_ARRAY`
- Draw geometry using one the following:
  - `void DrawArrays ( enum mode, int first, sizei count );`
  - `void DrawElements ( enum mode, sizei count, enum type, void *indices );`

5. Vertex arrays

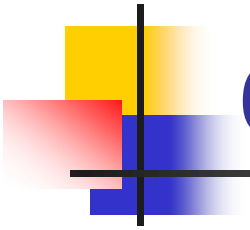
```
// draw a cube with 6 quads
glBegin(GL_QUADS);
  glVertex3fv(v0); // front face
  glVertex3fv(v1);
  glVertex3fv(v2);
  glVertex3fv(v3);

  glVertex3fv(v0); // right face
  glVertex3fv(v3);
  glVertex3fv(v4);
  glVertex3fv(v5);

  glVertex3fv(v0); // up face
  glVertex3fv(v5);
  glVertex3fv(v6);
  glVertex3fv(v1);

  ... // draw other 3 faces

glEnd();
```

:this



becomes:

```
// define vertex coords (8 x 3 = 24 GLfloats)
GLfloat vertices[] = {...};
...
// activate and specify pointer to vertex array
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// draw a cube
glDrawArrays(GL_QUADS, 0, 24);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
```



6. Vertex Buffer Objects (VBO)

- Optimization method: aggregate geometry into continuous, coherent memory areas *on the graphics hardware*
- Very similar to vertex arrays
- VBOs hold geometry and state *on the graphics hardware*
  - Significant reduction in rendering time
- Provide mapping from application memory to graphics memory
  - Allows fast updates when geometry changes

6. Vertex Buffer Objects (VBO)

```
// Generate vertex buffer, bind and load data
glGenBuffers(1, &vbo_vertices);
glBindBuffer(GL_ARRAY_BUFFER, vbo_vertices);
glBufferData(GL_ARRAY_BUFFER,
             nvertices * 3 * sizeof(float),
             vertices, GL_STATIC_DRAW);

// Same for texture coordinates:
glGenBuffers(1, &vbo_texcoords);
glBindBuffer(GL_ARRAY_BUFFER, vbo_texcoords);
glBufferData(GL_ARRAY_BUFFER,
             nvertices * 3 * sizeof(float),
             texcoords, GL_STATIC_DRAW);

// Data is now in graphics card, so we can
// free our copy
delete [] vertices; vertices = NULL;
delete [] texcoords; texcoords = NULL;

...

// Drawing works the same as with vertex arrays
glDrawArrays(...);

...

// Map vertex buffer to client's memory
glBindBuffer(GL_ARRAY_BUFFER, vbo_vertices);
p = glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);
```

Frequency:

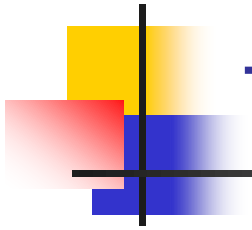
- STREAM — data contents will be modified once and used at most a few times
- STATIC — data contents will be modified once and used many times
- DYNAMIC — data contents will be modified repeatedly and used many times

Nature:

- DRAW — data contents are modified by the application, and used as the source for GL drawing and image specification commands
- READ — data contents are modified by reading data from the GL, and used to return that data when queried by the application
- COPY — data contents are modified by reading data from the GL, and used as the source for GL drawing and image specification commands

- GL_READ_ONLY
- GL_WRITE_ONLY
- GL_READ_WRITE

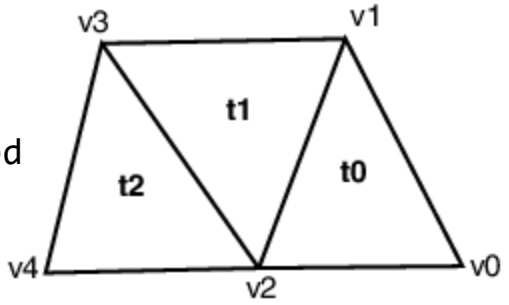
7. Data and task parallelism



- Observation: graphics is “embarrassingly parallel”
  - We need to dive into the graphics pipeline again to see why this is true

OpenGL vertex pipeline: shaded-tristrip code

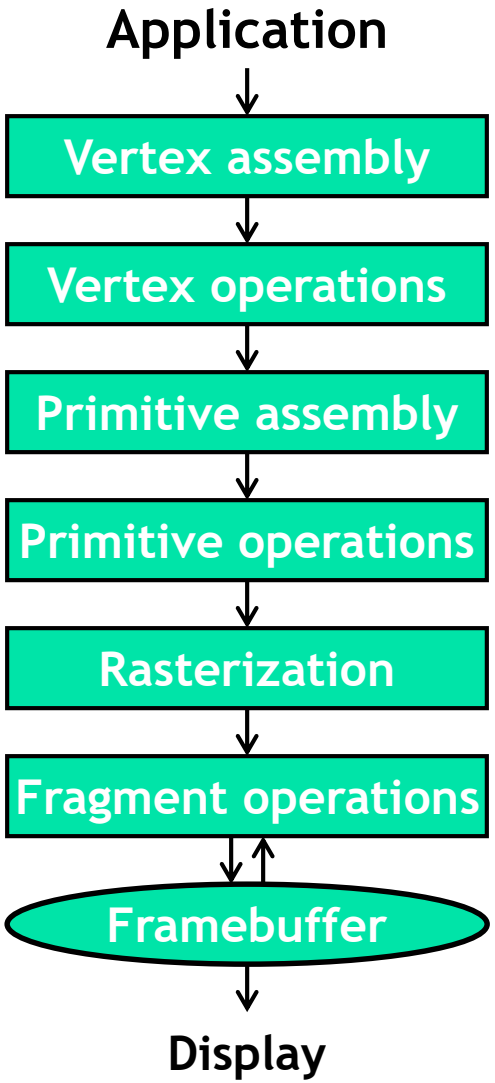
```
glClearColor(1, 1, 1, 1);            // opaque white background
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
glOrtho(0, 100, 0, 100, -1, 1);      // left/right/top/bot + near/far
glBegin(GL_TRIANGLE_STRIP);
  glColor3f(0, 0.5, 0);              // colour dark green
  glVertex2i(11, 31);                // v0
  glVertex2i(37, 71);                // v1
  glColor3f(0.5, 0, 0);              // colour dark red
  glVertex2i(91, 38);                // v2
  glVertex2i(65, 71);                // v3
  glVertex2i(45, 29);                // v4
glEnd();
glFlush();
```



OpenGL vertex pipeline

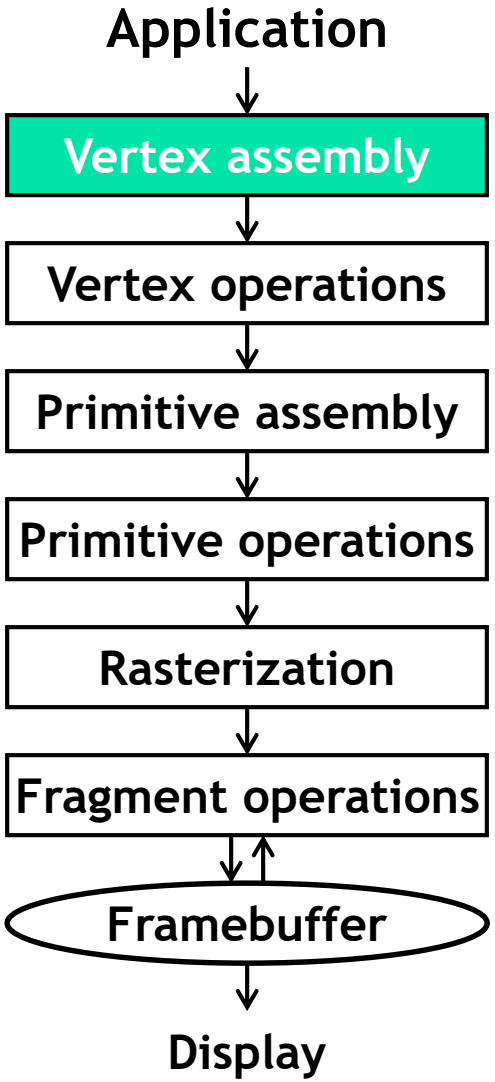
In what follows:

- emphasis is on data types
- diagram ignores
  - pixel pipeline
  - texture memory
  - display lists
  - …
- Display is not part of OpenGL



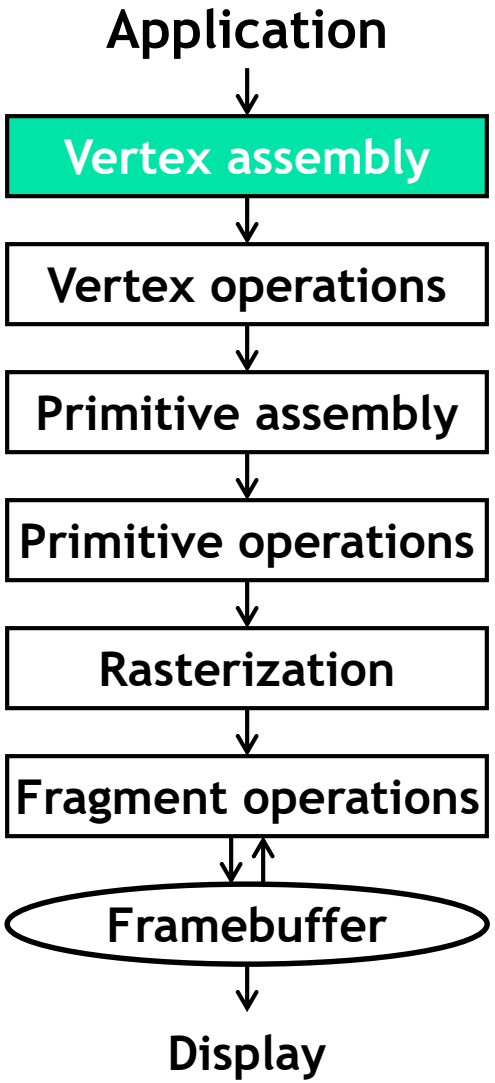
Vertex assembly (data types)

```
struct {
   float x,y,z,w;
   float r,g,b,a;
} vertex;
```



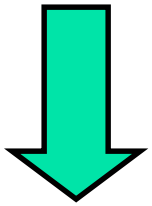
Vertex assembly (OpenGL)

- Vertex assembly
  - Force input to “canonical” format
    - Convert to internal representation
      - E.g., x, y to float
    - Initialize unspecified values
      - E.g., z = 0, w=1
    - Insert current modal state
      - E.g., color to 0,0.5,0,1
  - Or create using evaluators
- Error detection
  - INVALID_ENUM
  - INVALID_VALUE
  - INVALID_OPERATION
    - Especially between Begin and End

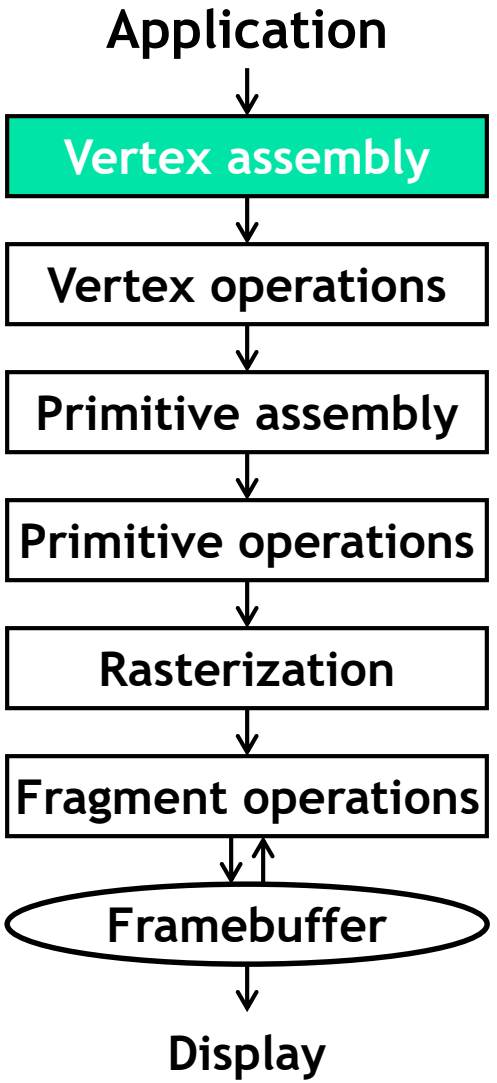


Vertex assembly (in our case)

```
glColor3f(0, 0.5, 0);
glVertex2i(11, 31);
glVertex2i(37, 71);
glColor3f(0.5, 0, 0); // no effect
```

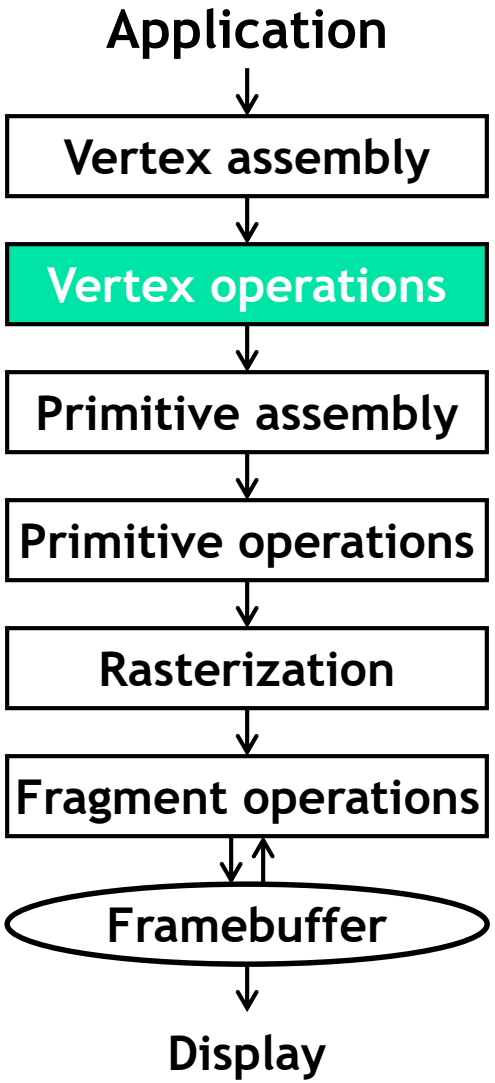


```
struct {
  float x,y,z,w; // 11, 31, 0, 1
  float r,g,b,a; // 0, 0.5, 0, 1
} vertex;

struct {
  float x,y,z,w; // 37, 71, 0, 1
  float r,g,b,a; // 0, 0.5, 0, 1
} vertex;
```



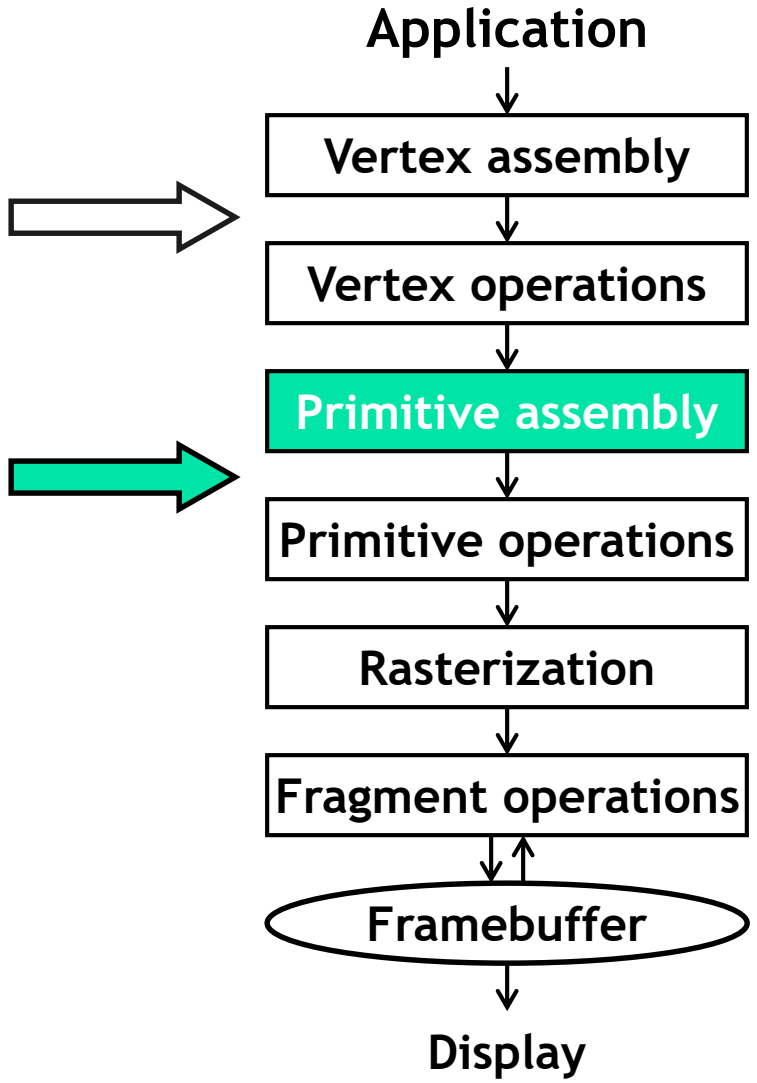
Vertex operations

- OpenGL
  - Transform coordinates
    - 4x4 matrix arithmetic
  - Compute (vertex) lighting
  - Compute texture coordinates
  - …
- In our case:
  - Scale coordinates to fit 100x100 window
  - No lighting, no texture coordinates



Primitive assembly (data types)

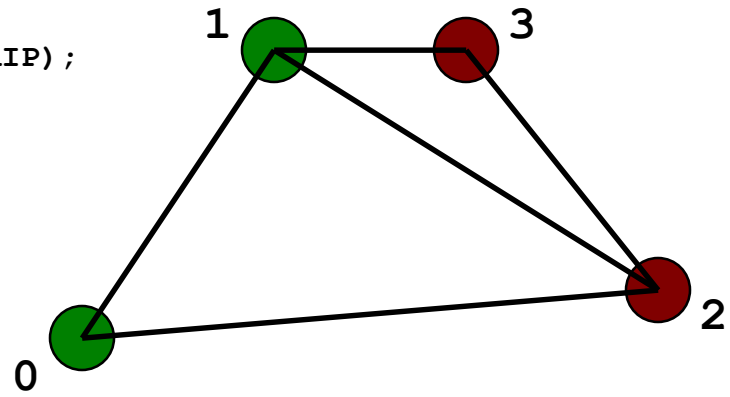
```
struct {
  float x,y,z,w;
  float r,g,b,a;
} vertex;

struct {
  vertex v0,v1,v2;
} triangle;

    or

struct {
  vertex v0,v1;
} line;

    or

struct {
  vertex v0;
} point;
```

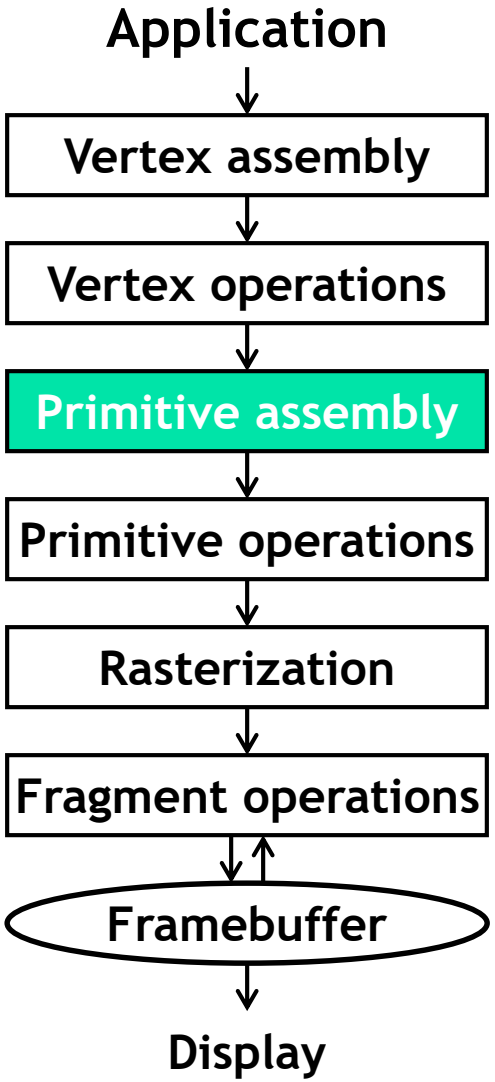


Primitive assembly

- OpenGL
  - Group vertices into primitives:
    - points,
    - lines, or
    - triangles
  - Decompose polygons to triangles
  - Duplicate vertices in strips or fans
- In our case:
  - Create two triangles from a strip:

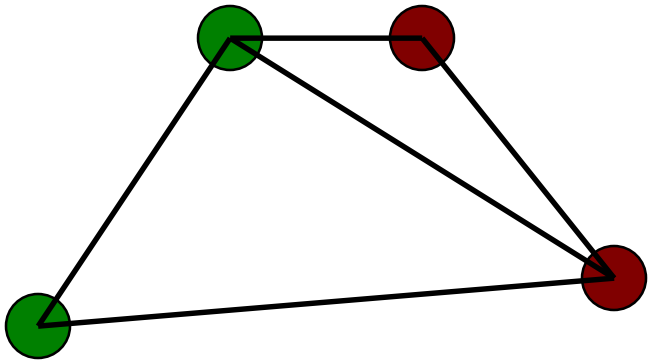
```
glBegin(GL_TRIANGLE_STRIP);
glColor(green);
glVertex2i(…); // 0
glVertex2i(…); // 1
glColor(red);
glVertex2i(…); // 2
glVertex2i(…); // 3
glEnd();
```

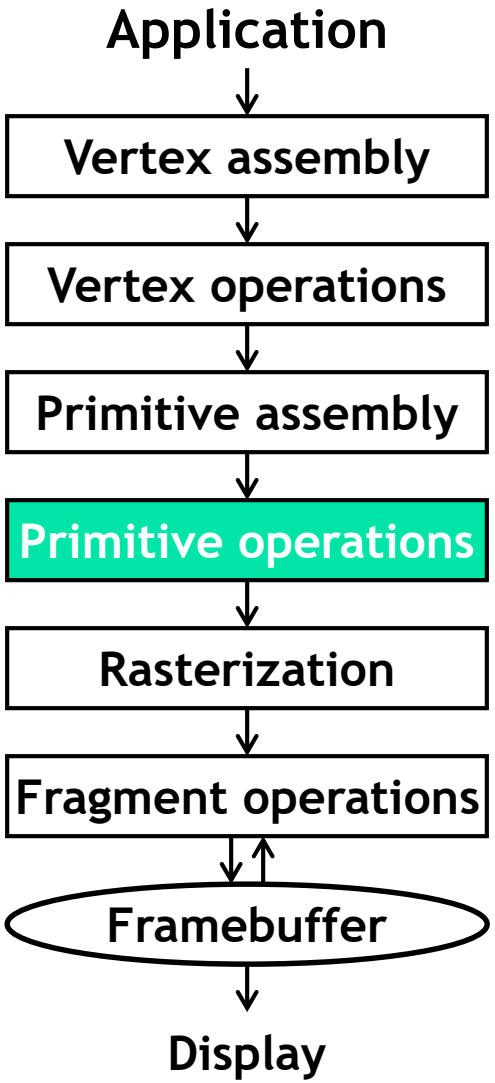




Primitive operations

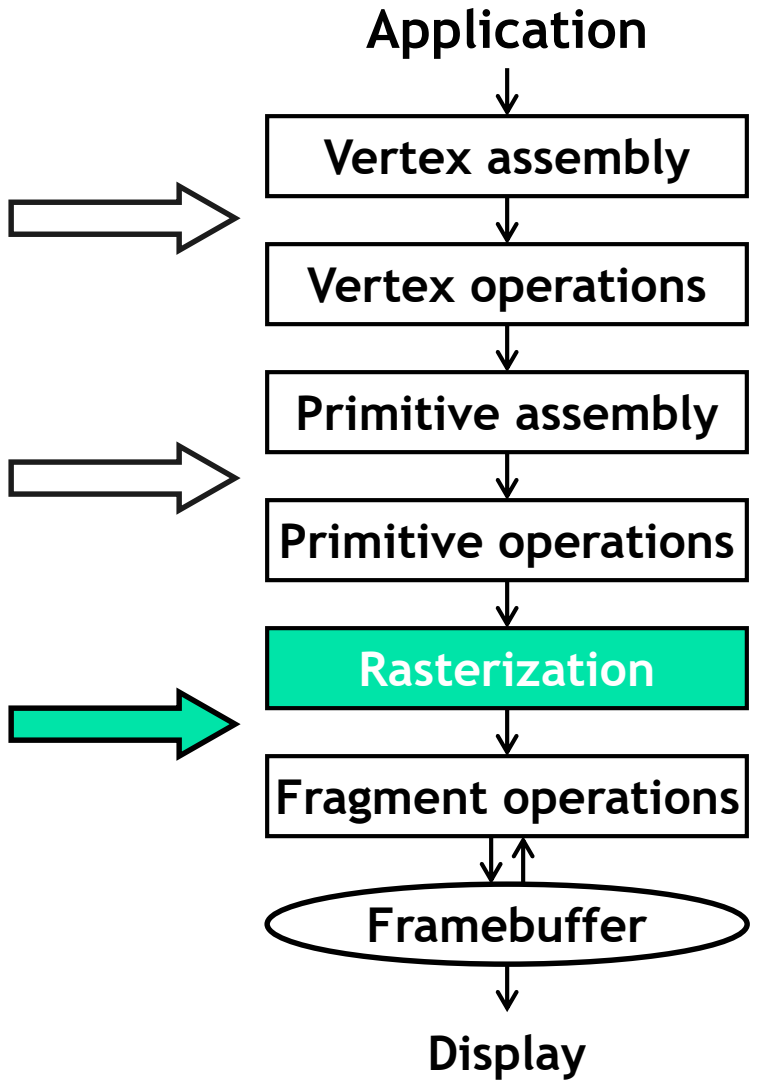
- OpenGL
  - Clip to the window boundaries
    - Actually to the frustum surfaces
  - Perform back-face / front-face ops
    - Culling
    - Color assignment for 2-side lighting
- In our case
  - Nothing happens





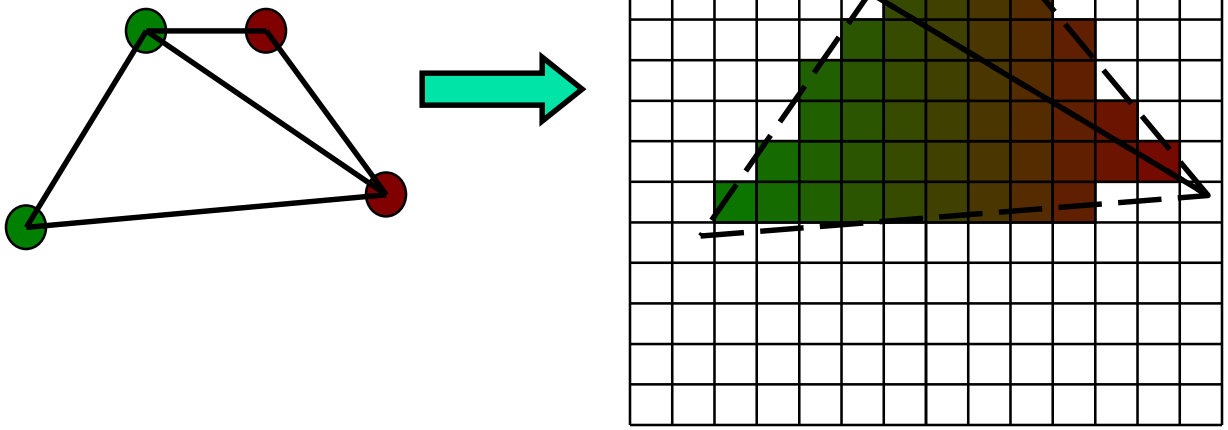
Rasterization (data types)

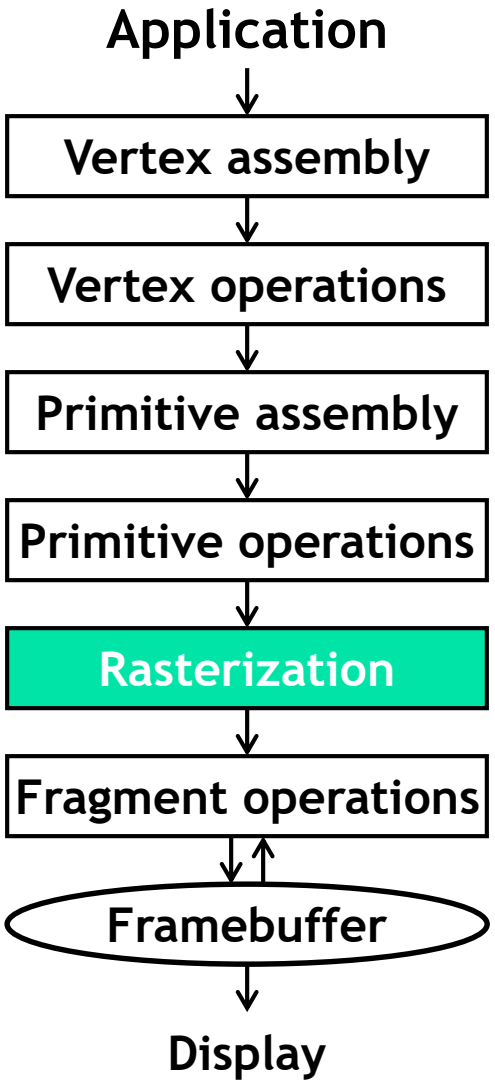
```
struct {
  float x,y,z,w;
  float r,g,b,a;
} vertex;

struct {
  vertex v0,v1,v2
} triangle;

struct {
  short int x,y;
  float depth;
  float r,g,b,a;
} fragment;
```



Rasterization

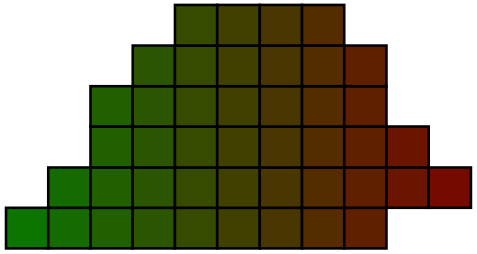
- OpenGL
  - Determine which pixels are included in the primitive
    - Generate a fragment for each such pixel
  - Assign attributes (e.g., color) to each fragment
- In our case:

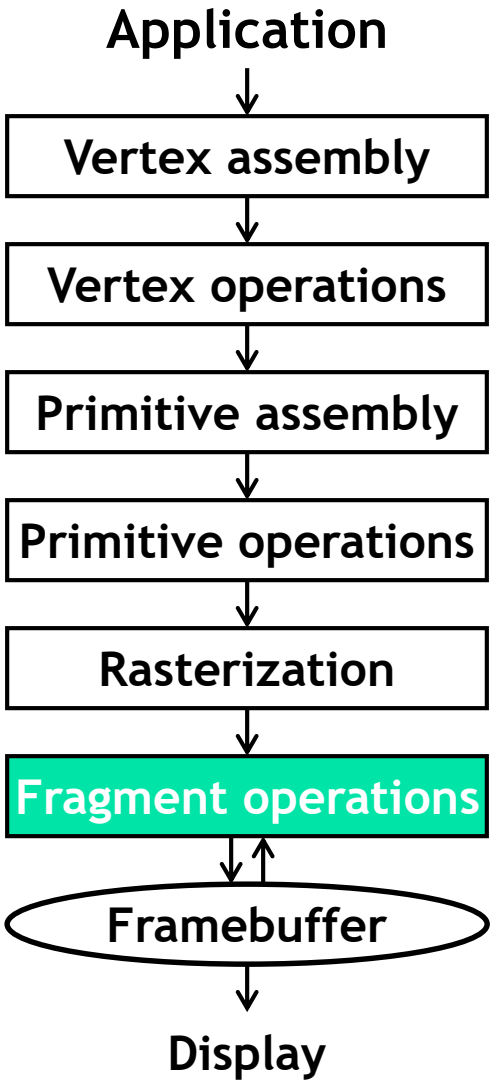




Fragment operations

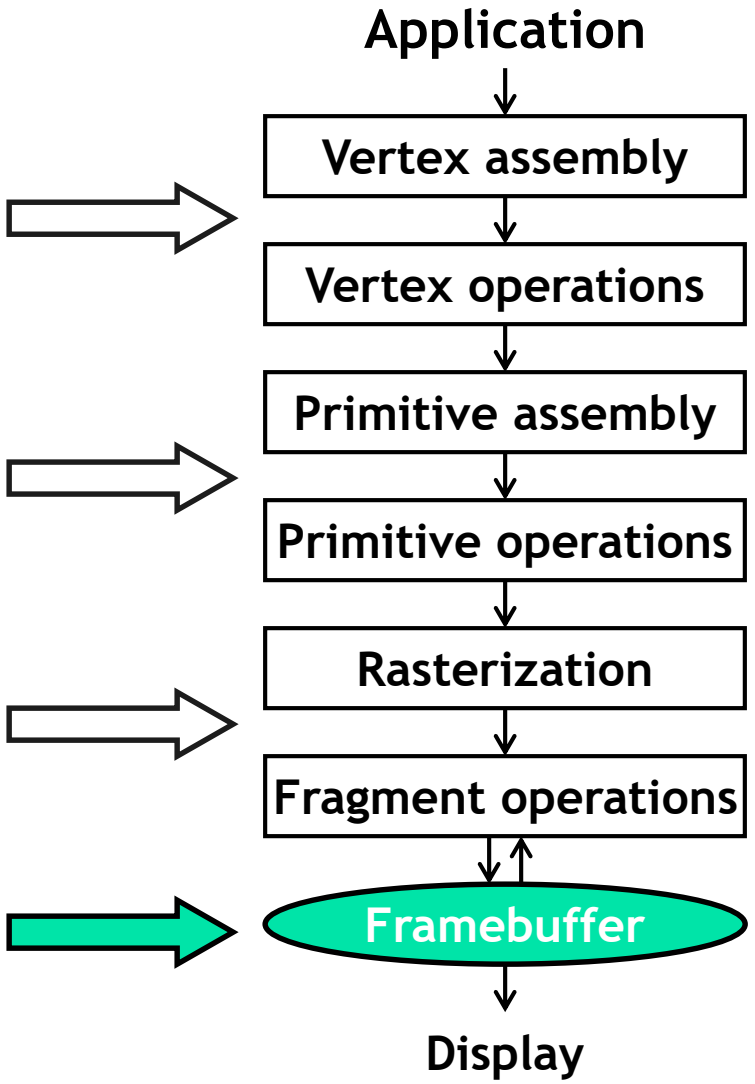
- OpenGL
  - Texture mapping
  - Fragment lighting (OpenGL 2.0)
  - Fog
  - Scissor test
  - Alpha test
- In our case, nothing happens:





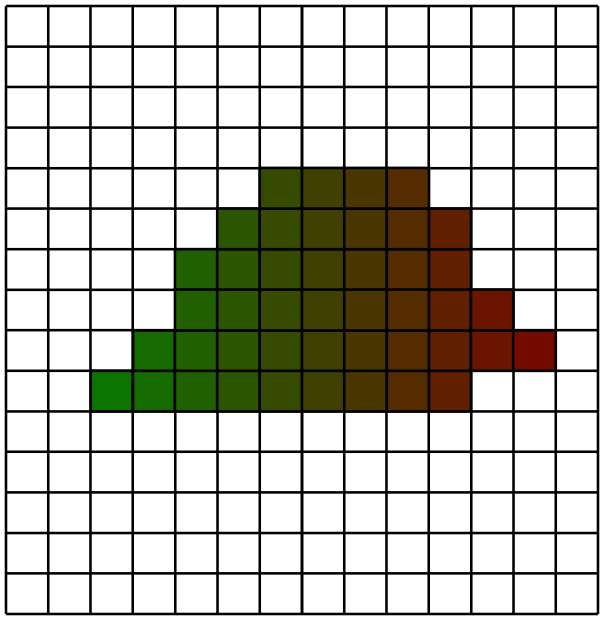
Framebuffer (2-D array of pixels)

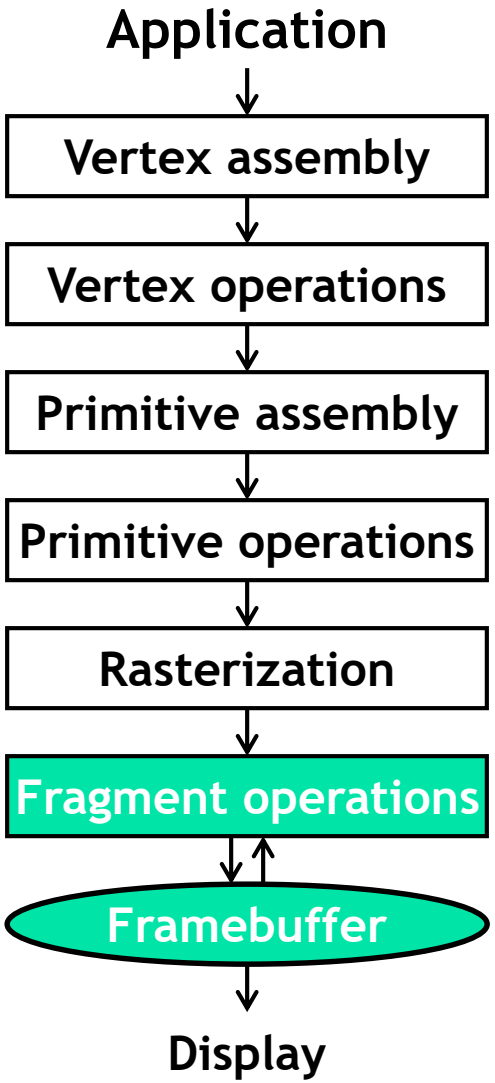
```
struct {
  float x,y,z,w;
  float r,g,b,a;
} vertex;

struct {
  vertex v0,v1,v2
} triangle;

struct {
  short int x,y;
  float depth;
  float r,g,b,a;
} fragment;

struct {
  int depth;
  byte r,g,b,a;
} pixel;
```



Fragment ←→ framebuffer operations

- OpenGL
  - Color blending
  - Depth testing (aka z-buffering)
  - Conversion to pixels
- In our case, conversion to pixels:





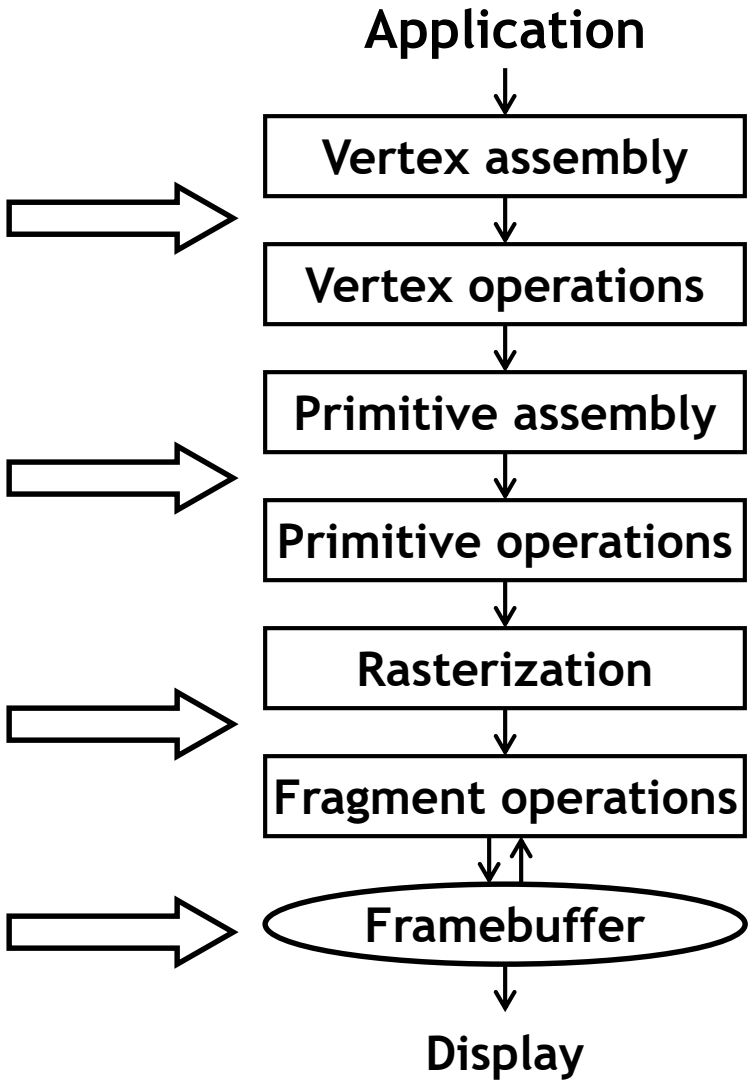
Key idea: images are *built* in the framebuffer, not just *placed* there!

Graphics is “embarrassingly parallel”

Many separate tasks

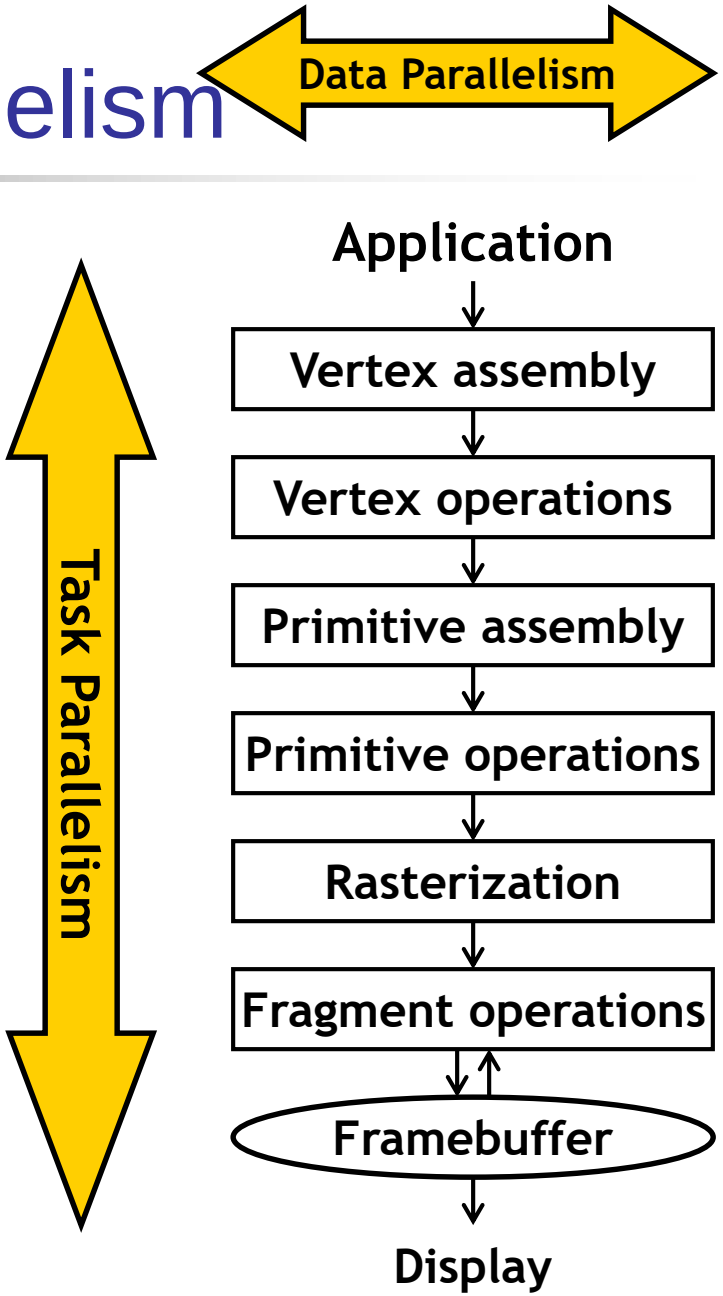
No “horizontal” dependencies, few “vertical” (in-order execution)

```
struct {
  float x,y,z,w;
  float r,g,b,a;
} vertex;

struct {
  vertex v0,v1,v2
} triangle;

struct {
  short int x,y;
  float depth;
  float r,g,b,a;
} fragment;

struct {
  int depth;
  byte r,g,b,a;
} pixel;
```

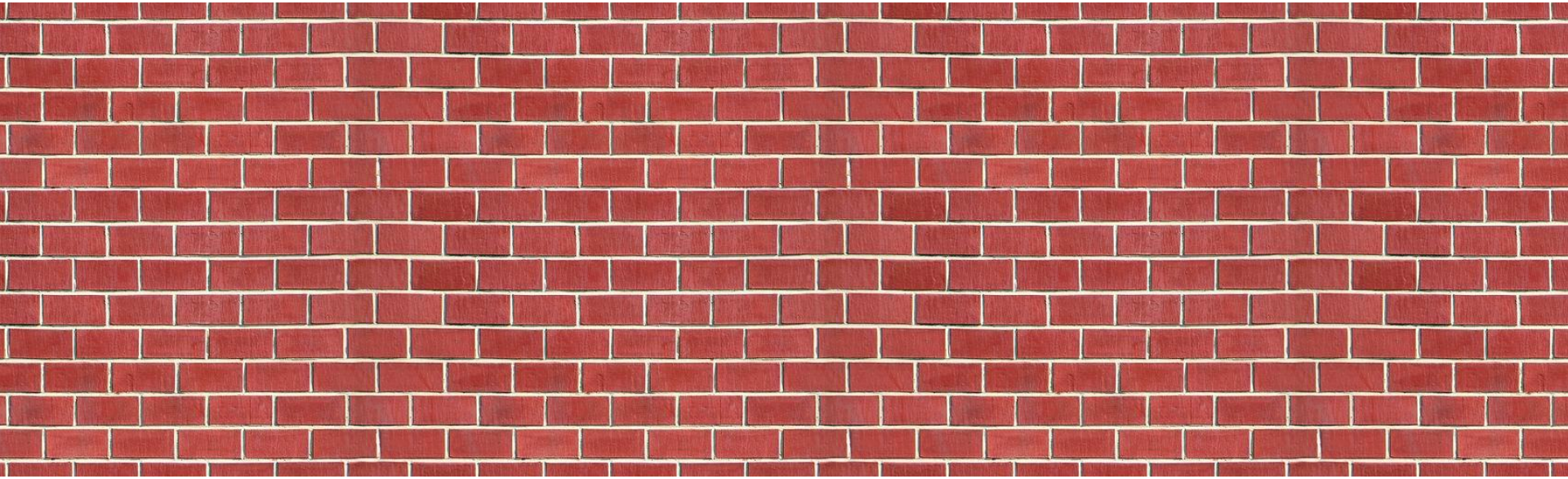


Data and task parallelism

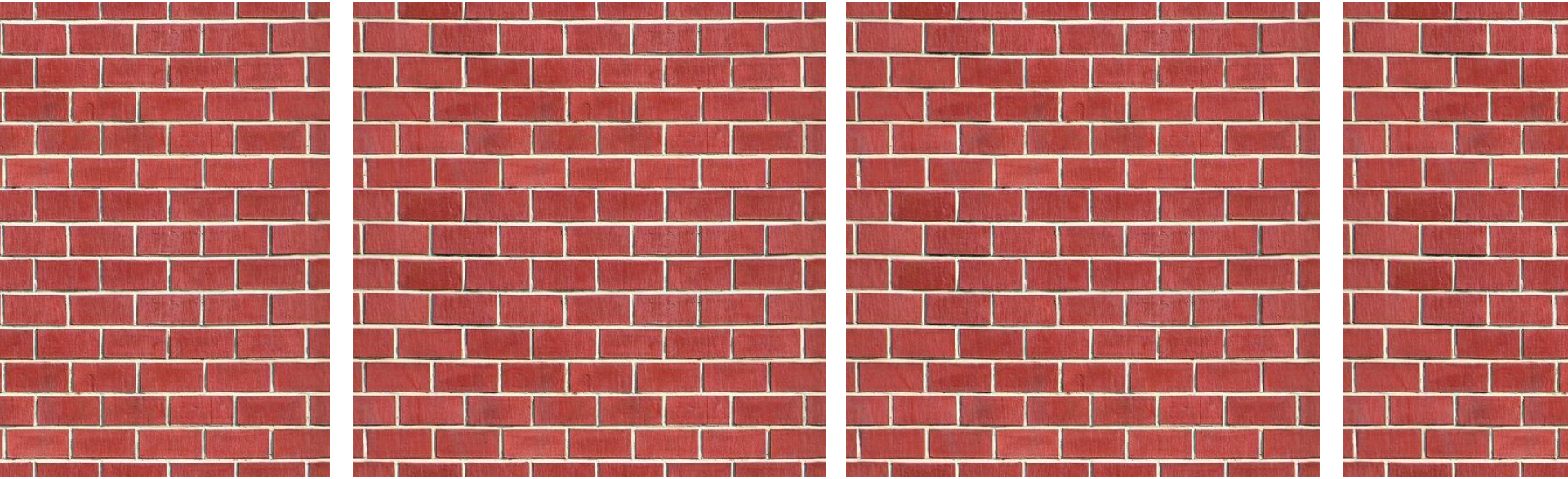
- Data parallelism
  - Simultaneously doing the same thing to similar data
  - E.g., transforming vertexes
  - Some variance in “same thing” is possible
- Task parallelism
  - Simultaneously doing different things
  - E.g., the tasks (stages) of the vertex pipeline



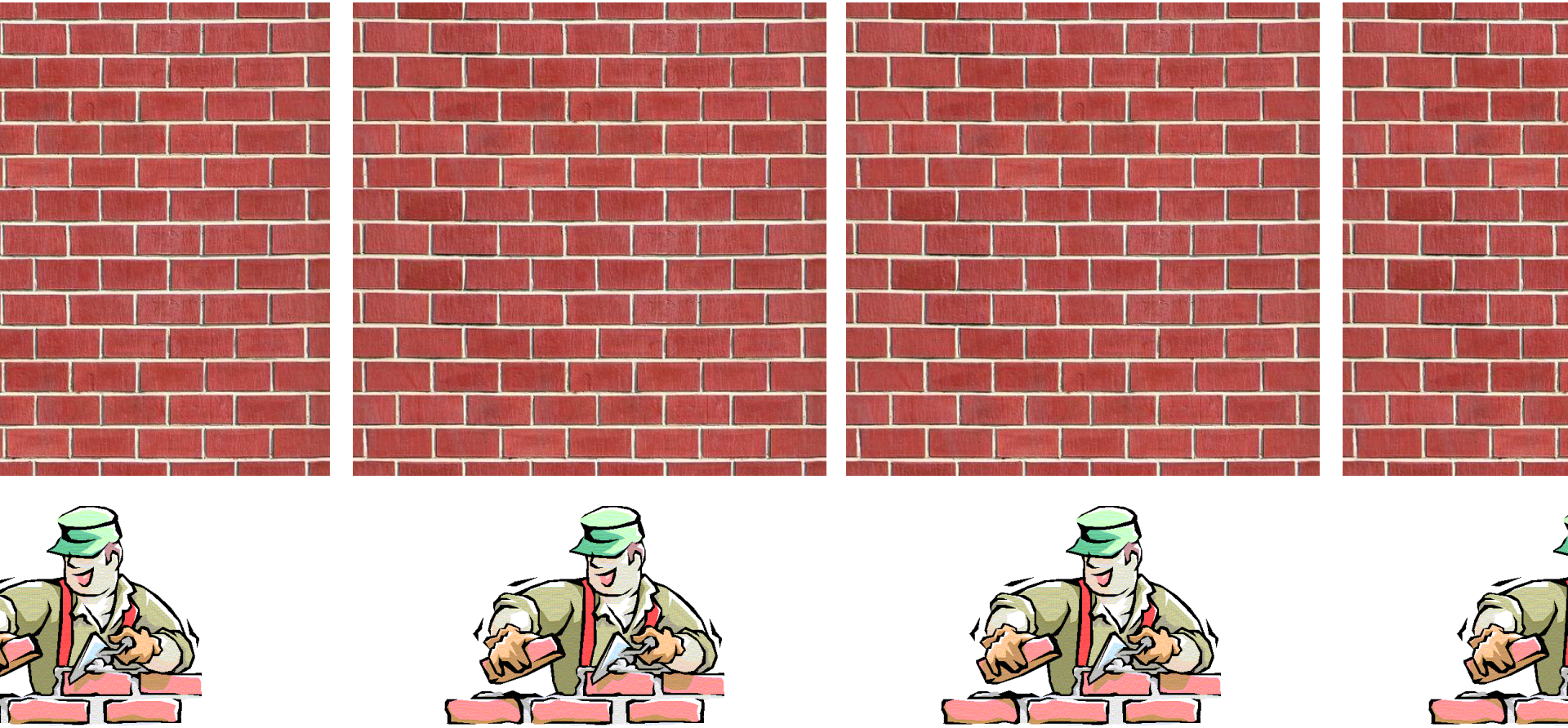
Data and task parallelism



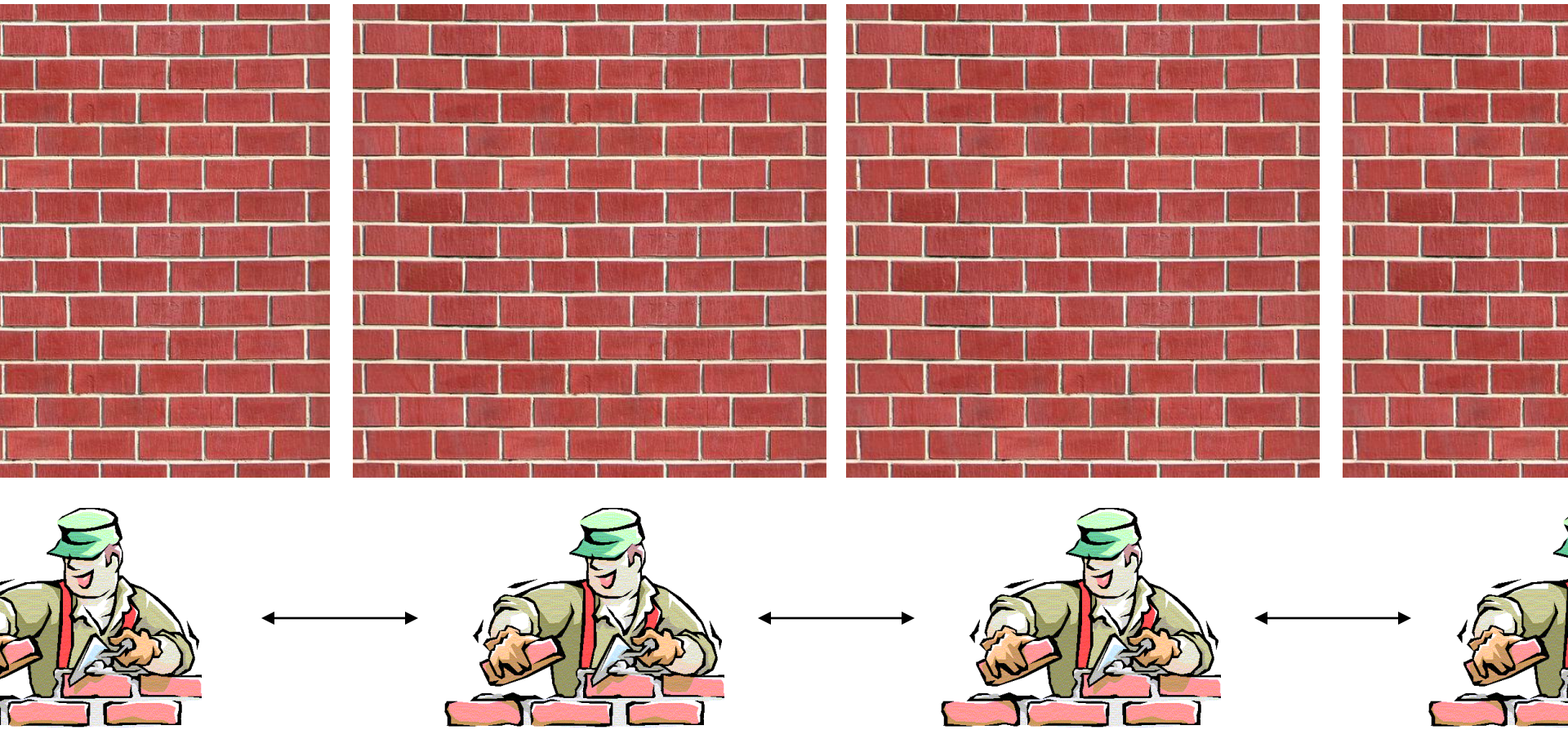
Data and task parallelism



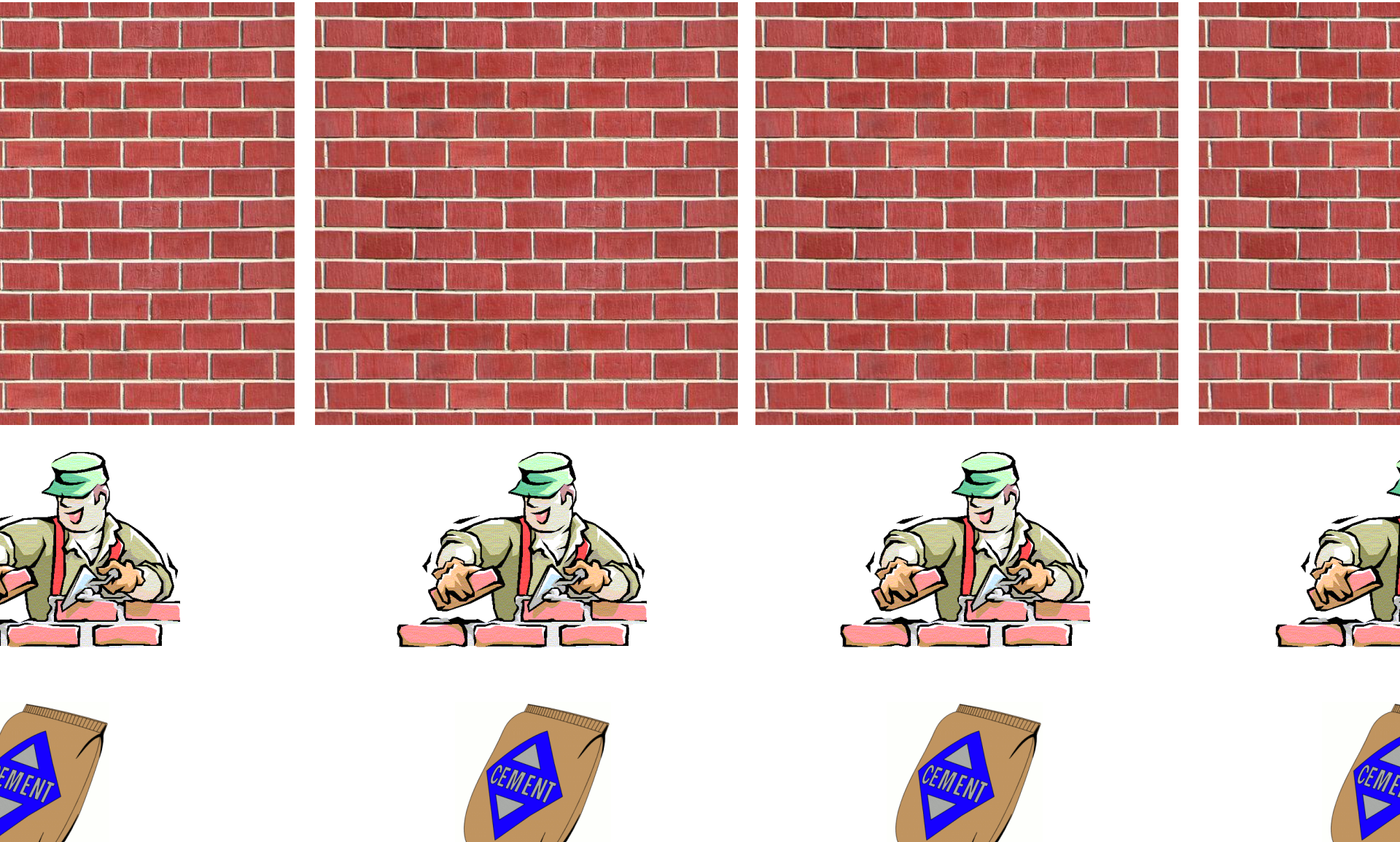
Data and task parallelism



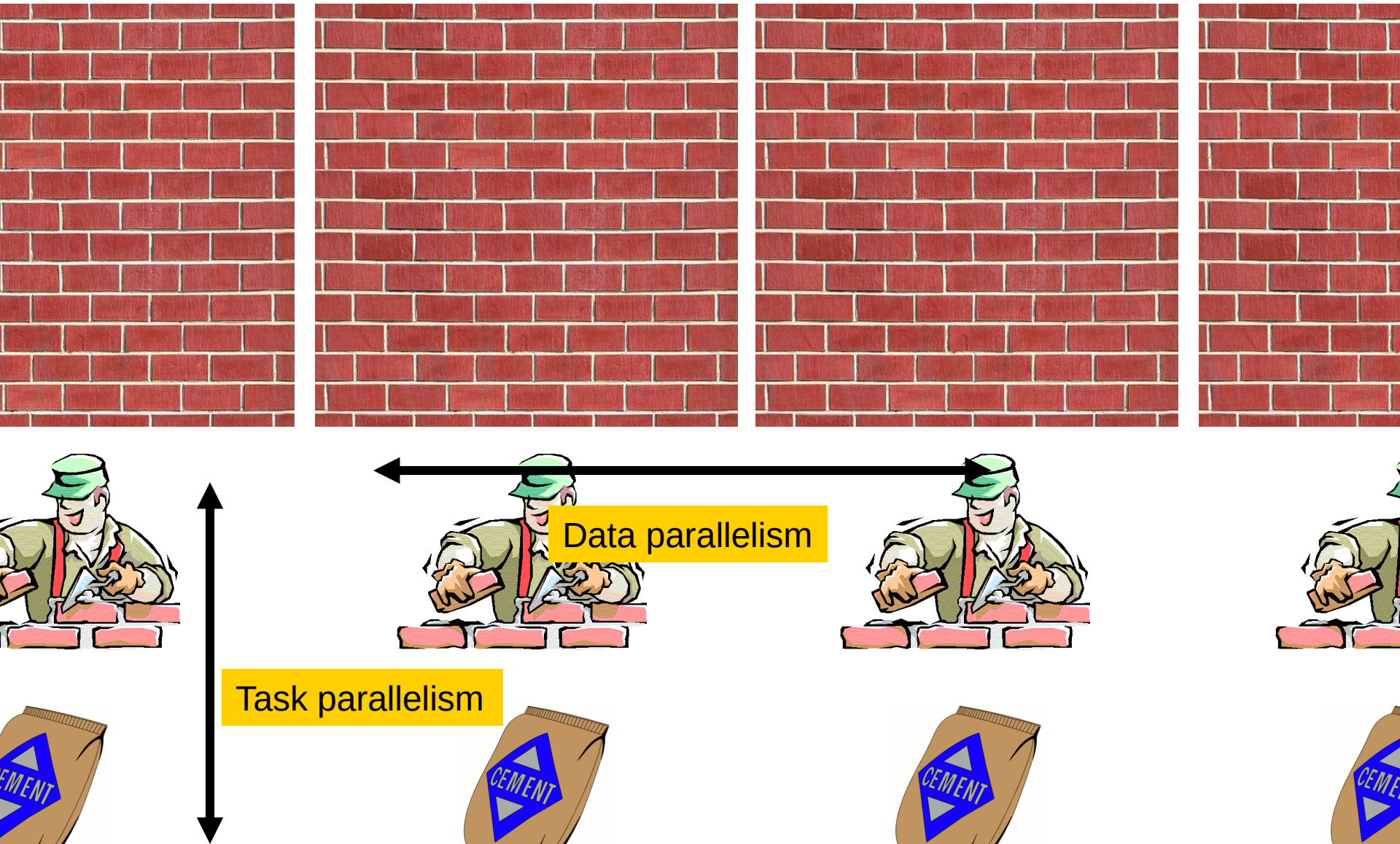
Data and task parallelism



Data and task parallelism

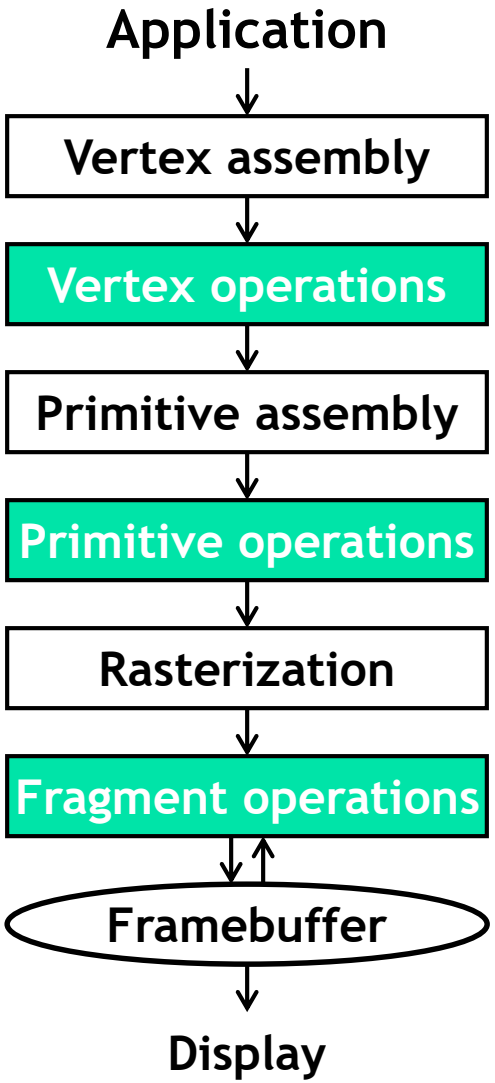


Data and task parallelism

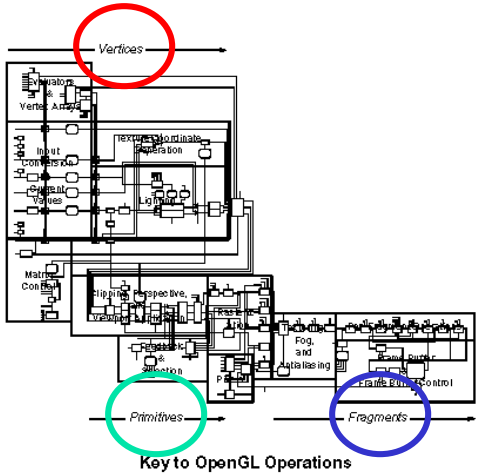


OpenGL mechanism

- Programmer specifies operation modes:
  - Vertex operations
    - Transformation
    - Lighting
  - Primitive operations
    - Back-face culling
  - Fragment operations
    - Shading (texture and lighting)
    - Framebuffer (blending and depth testing)
- Extreme differentiation in functionality: great example is the diagram at the back of the old (out of print) “blue book” (on next page)



The OpenGL® Machine

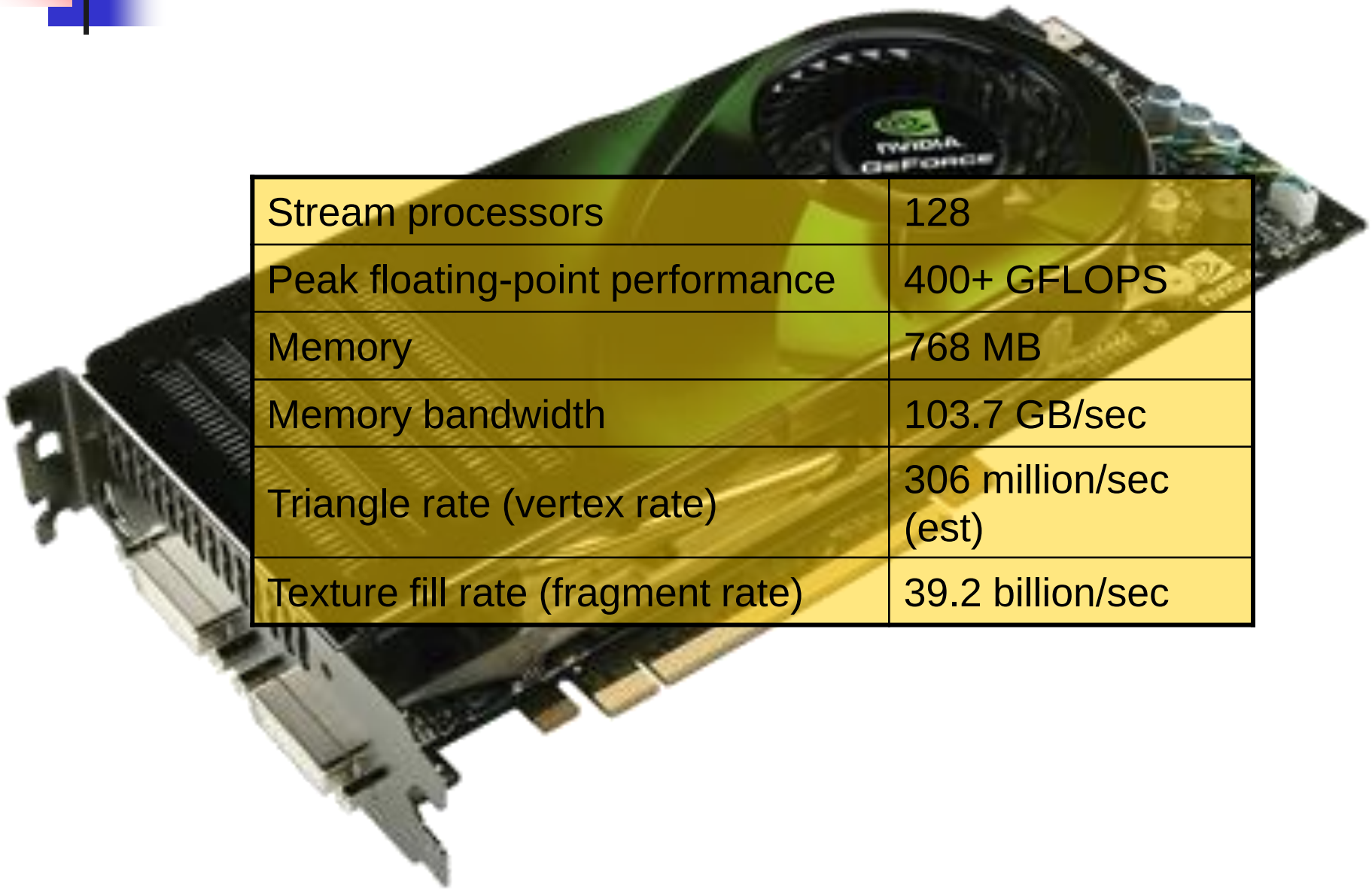


Key to OpenGL Operations

Notes:

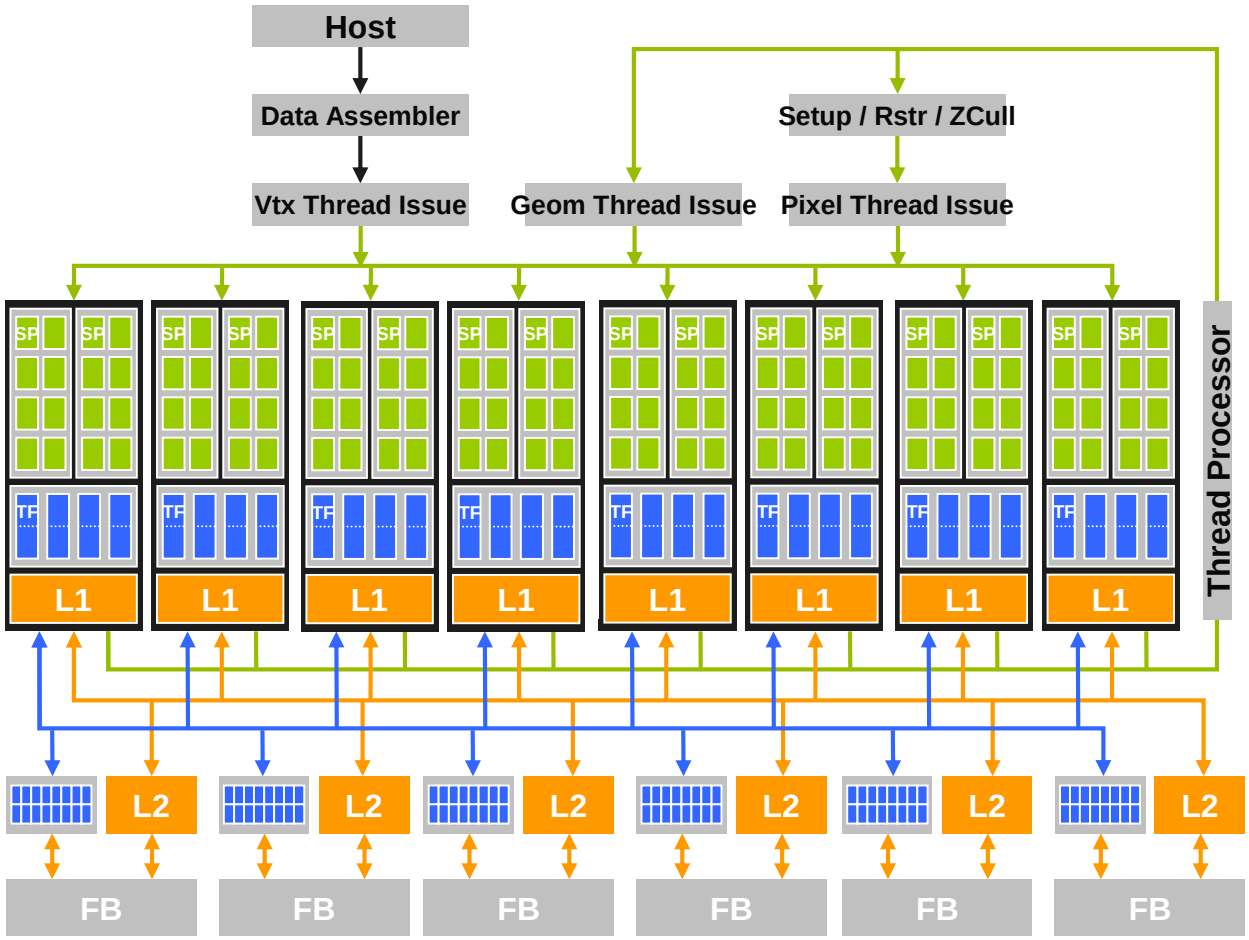
1. Commands (and constants) are shown without the gl (or GL_) prefix.
2. The following commands do not appear in this diagram: glAccum, glClearAccum, glFlush, display list commands, texture object commands, commands for obtaining OpenGL state (glGet commands and glIsEnabled), and glPushAttrib and glPopAttrib. Utility library routines are not shown.
3. After their execution, glDrawArrays and glDrawElements leave affected current values indeterminate.
4. This diagram is schematic; it may not directly correspond to any actual OpenGL implementation.

NVIDIA 8800 Ultra (G80 family, 2007)

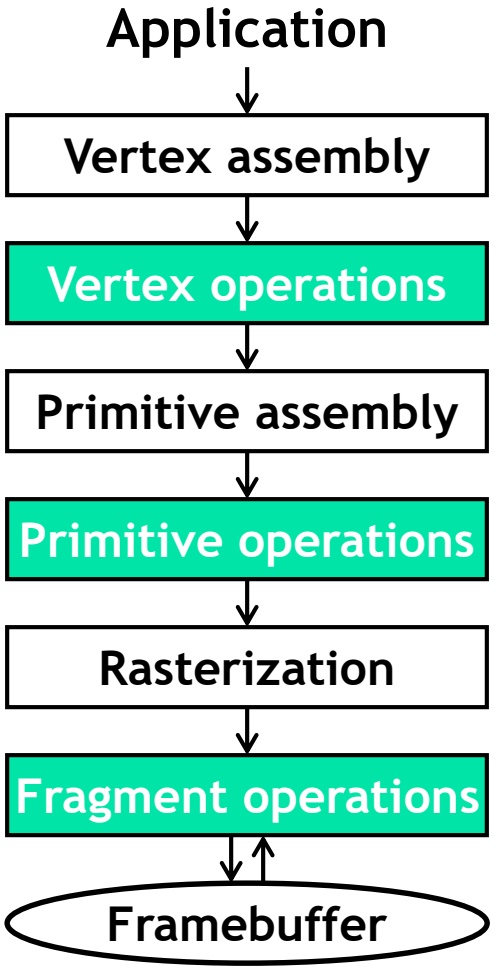


Stream processors	128
Peak floating-point performance	400+ GFLOPS
Memory	768 MB
Memory bandwidth	103.7 GB/sec
Triangle rate (vertex rate)	306 million/sec (est)
Texture fill rate (fragment rate)	39.2 billion/sec

Implementation

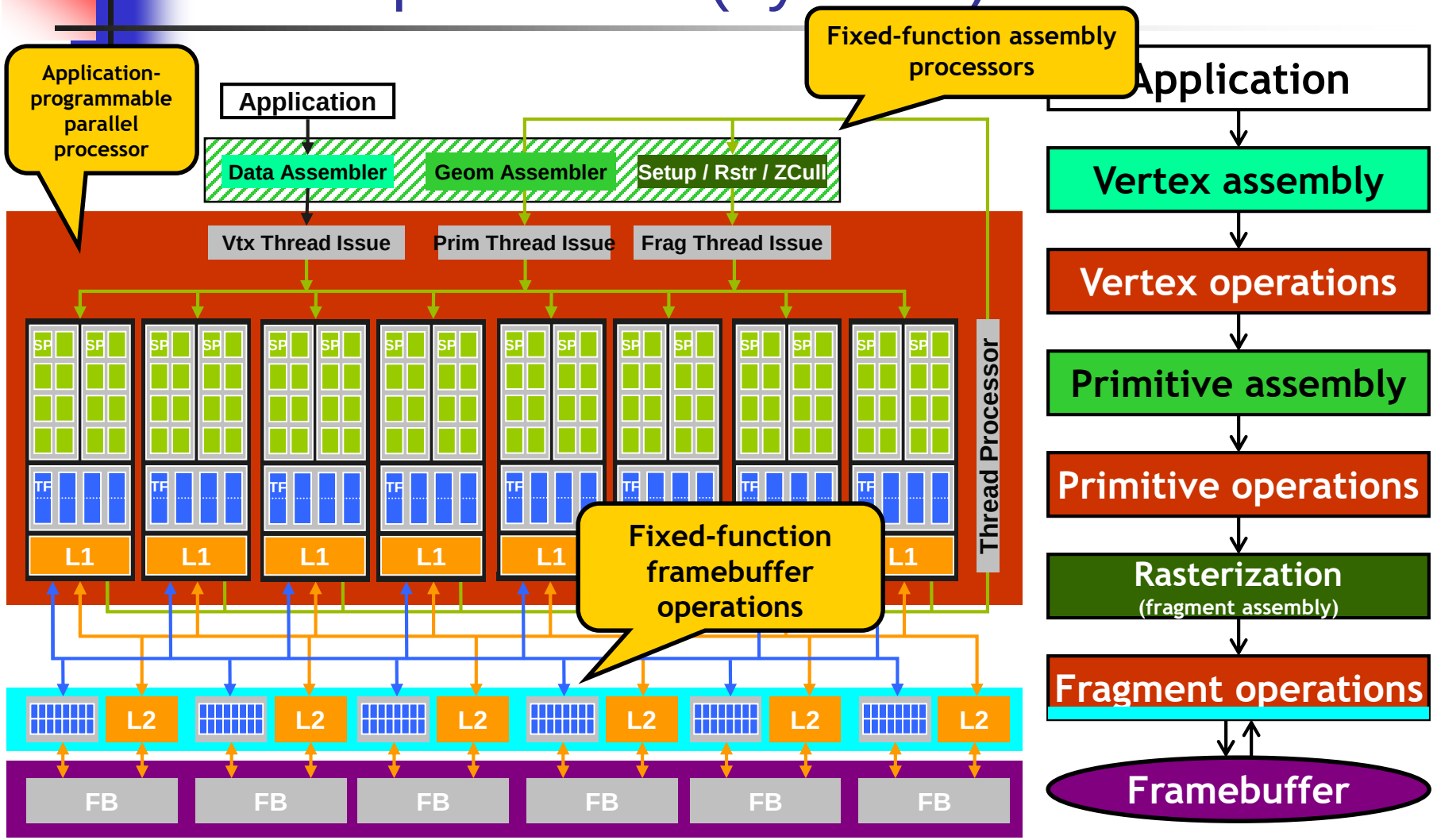


NVIDIA GeForce 8800



OpenGL Pipeline

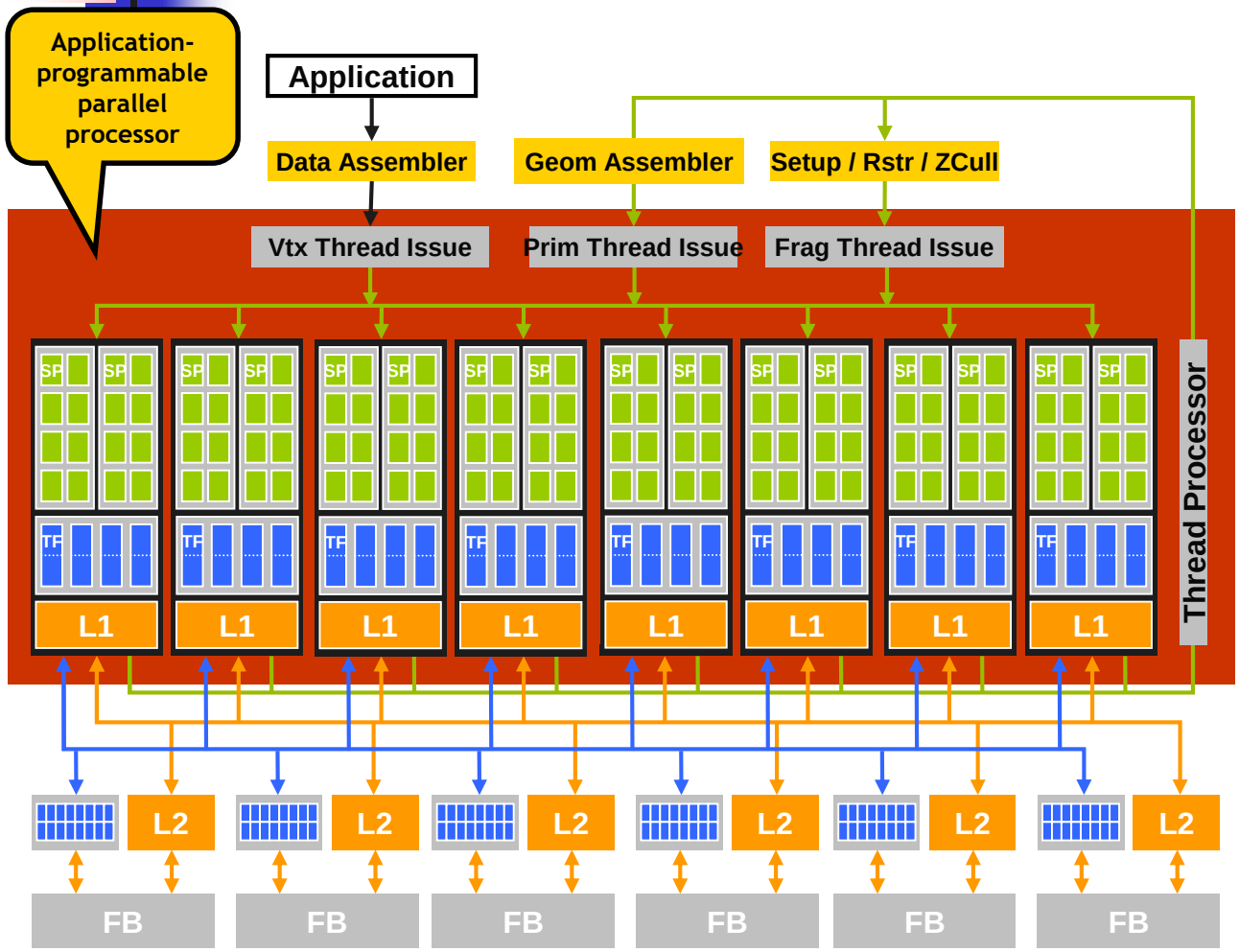
Correspondence (by color)

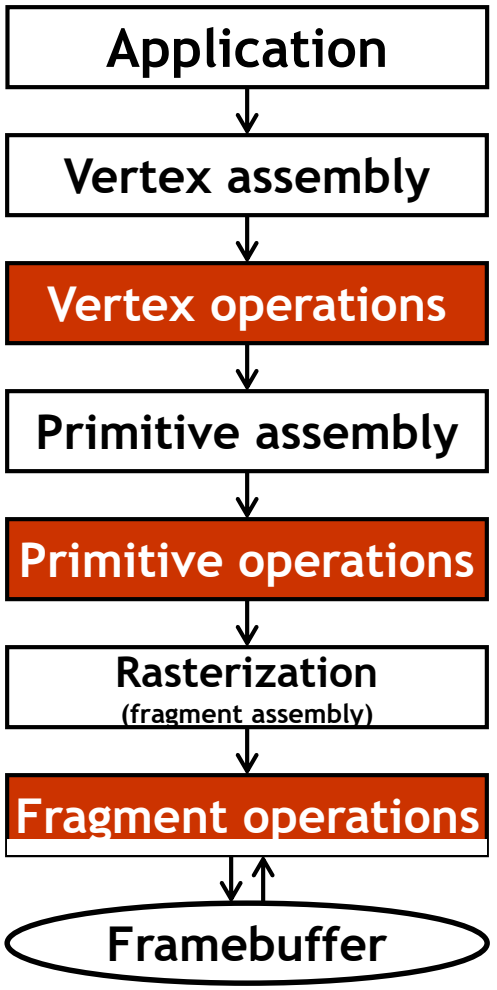


NVIDIA GeForce 8800

OpenGL Pipeline

Unified pipeline architecture





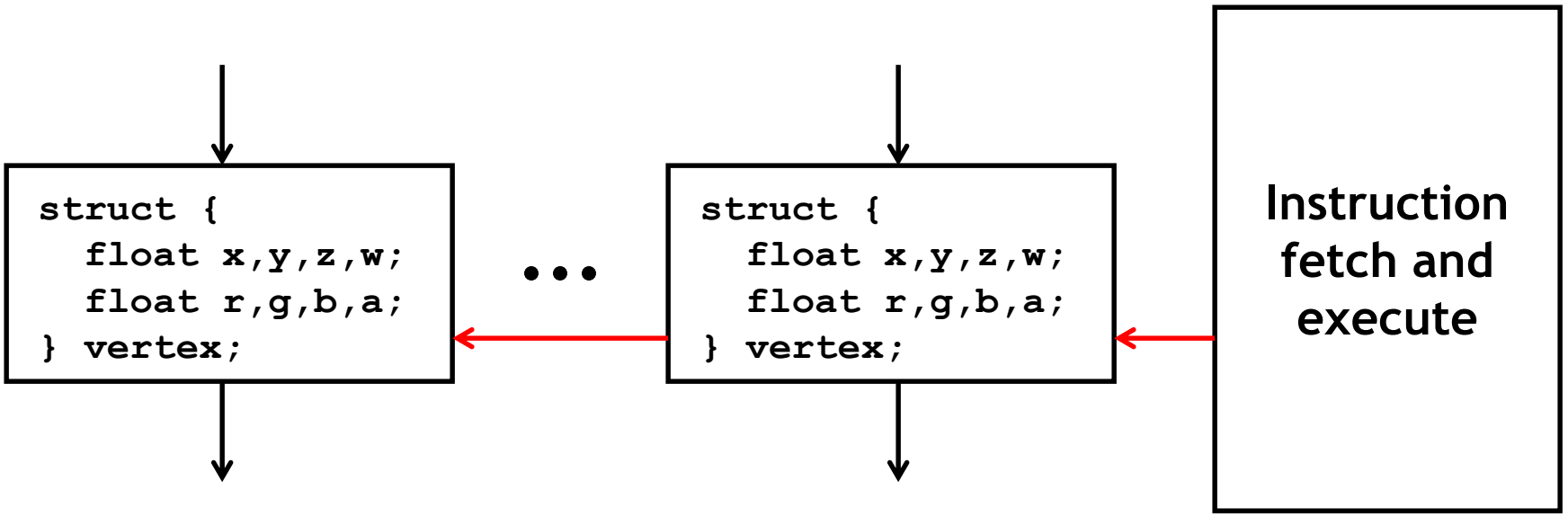
NVIDIA GeForce 8800

OpenGL Pipeline

Computational coherence

Data parallelism is computationally coherent

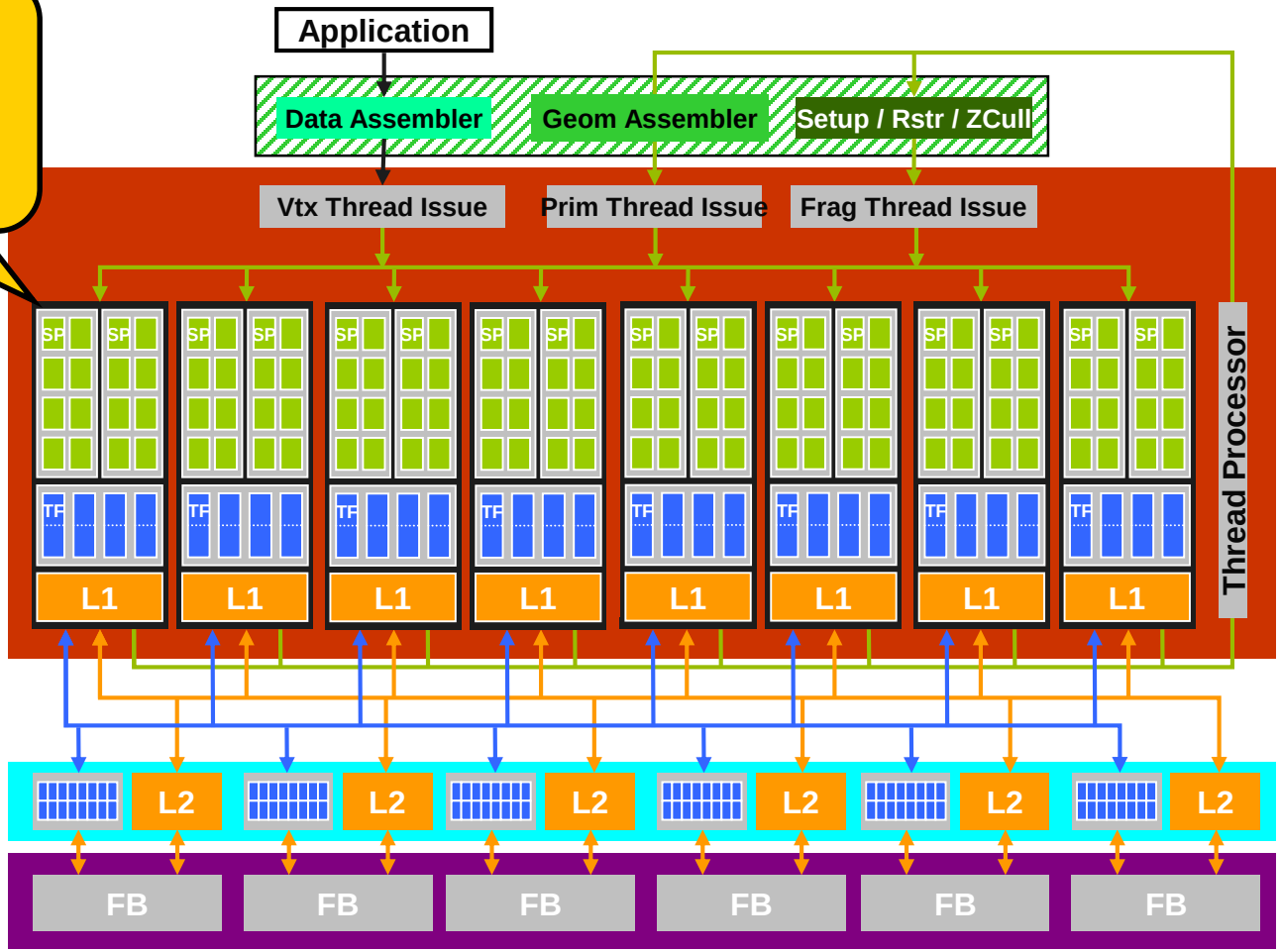
- Simultaneously doing the *same thing* to similar data
- Can share a single instruction sequencer with multiple data paths:



SIMD – Single Instruction Multiple Data

SIMD processing

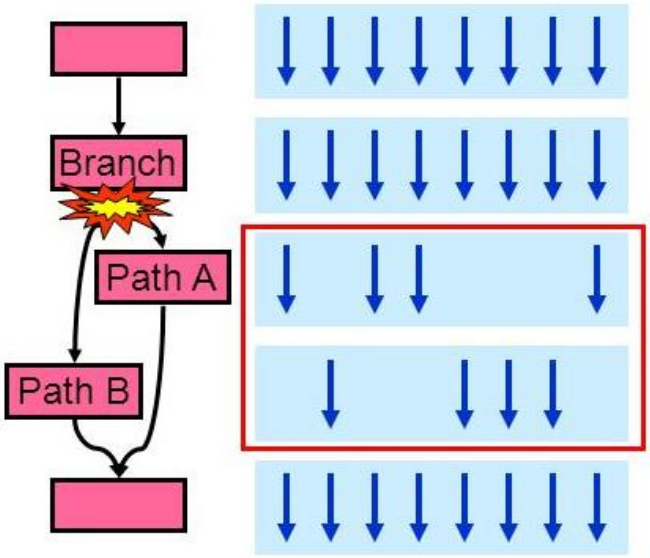
One of eight 16-wide SIMD processors



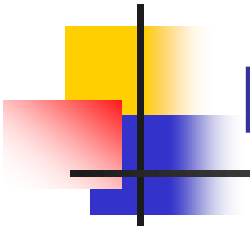
NVIDIA GeForce 8800

SIMD conditional control flow

- The “shader” abstraction operates on each data element independently
- But SIMD implementation shares a single execution unit across multiple data elements
- If data elements in the same SIMD unit branch differently, the execution unit must follow both paths (sequentially)
- The solution is *predication*:
  - Both paths are executed
  - Data paths are enabled only during their selected path
  - Can be nested
  - Performance is obviously lost!
- SIMD width is a compromise:
  - Too wide → too much performance loss due to predication
  - Too narrow → inefficient hardware implementation



Branch divergence



Latency issues

Overall rendering latency

- Typically measured in frames
- Of concern to application programmers
- Short on modern GPUs

Memory access latency

- Typically measured in clock cycles (and reaching thousands of those)
- Of direct concern to GPU architects and implementers
  - But useful for application programmers to understand too!

Multi-threading

Another kind of processor virtualization

- Unified GPUs share *a single execution engine* among multiple pipeline (task) stages
  - Equivalent to CPU multi-tasking
- Multi-threading shares a single execution engine among multiple data-parallel work elements
  - Similar to CPU hyper-threading

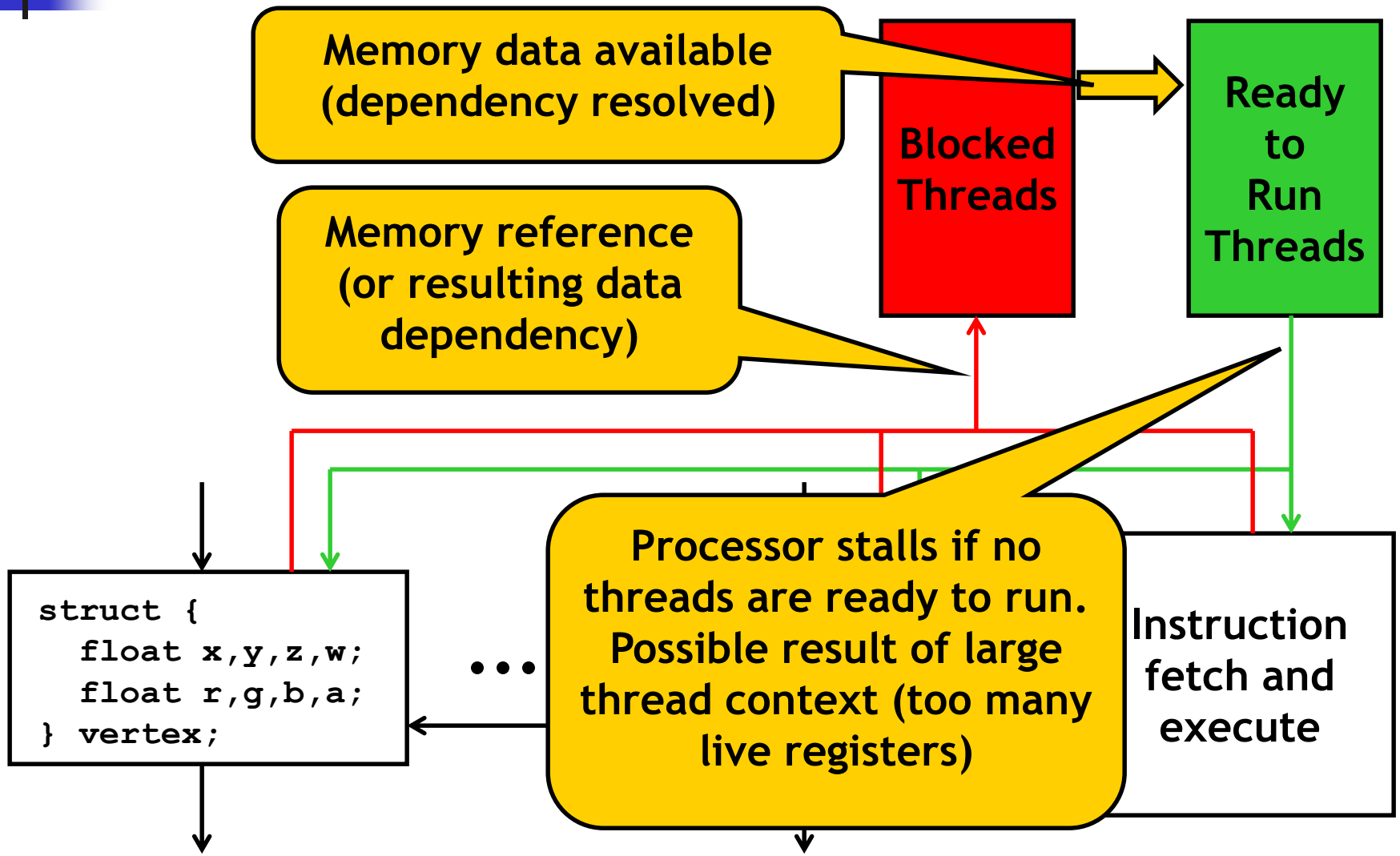
The 8800 Ultra multi-threading mechanism is used to support both multi-tasking and data-parallel multi-threading.

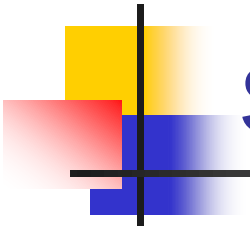
A thread is a data structure:

```
struct {
  int pc;         // program counter
  float reg[n]; // live register state
  enum ctxt;     // context information
  …
} thread;
```

More live registers mean more memory usage

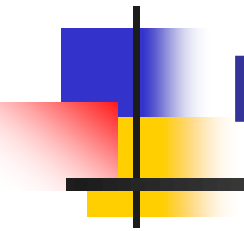
Multi-threading hides latency





Summary

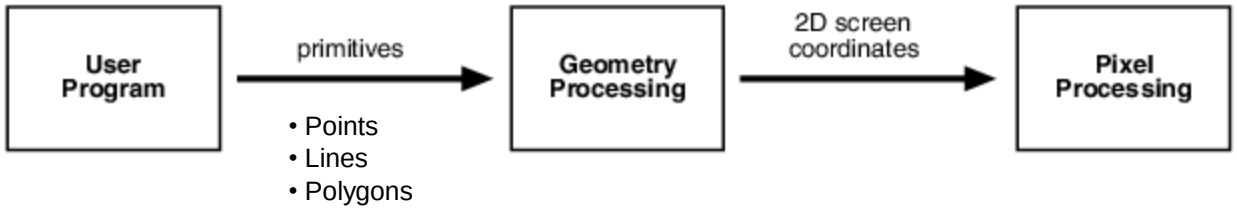
- Modern GPUs exploit data and task parallelism in the vertex pipeline through parallelism in the hardware
  - **Data** processed by the vertex pipeline
  - **Tasks** in the vertex pipeline
- Parallelism is achieved through many, grouped processing units
  - SIMD: Single Instruction, Multiple Data
  - The tasks in the vertex pipeline are split up in threads that execute on processing units
  - The processing units are no longer fixed function; they are unified
- Multithreading is used to hide latency



Programmable graphics hardware

In the beginning…

- Programmable graphics hardware has existed already for 40 years
  - Silicon Graphics Inc. (SGI)
  - PixelFlow
- Fixed function graphics hardware
  - Transformation and Lighting (T&L)
  - Rasterization
- Geometric primitives processed per-vertex
- Pixels coloured per-pixel

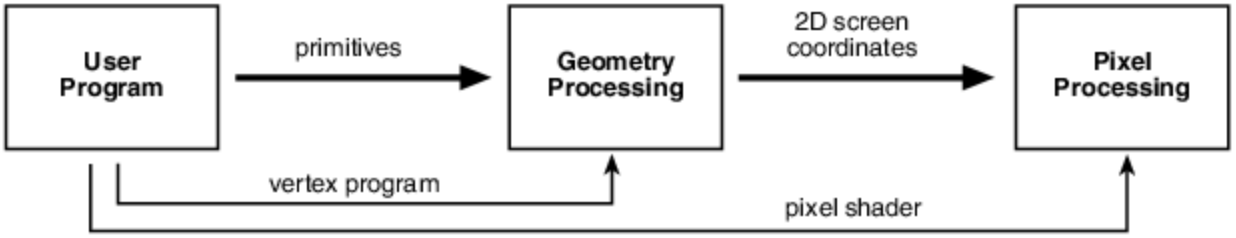


Programmable hardware

- Vertex shader/program
  - Operates on a vertex
- Geometry shader/program
  - Adds or deletes elements from geometry
- Fragment shader/program
  - Operates on a pixel (fragment)

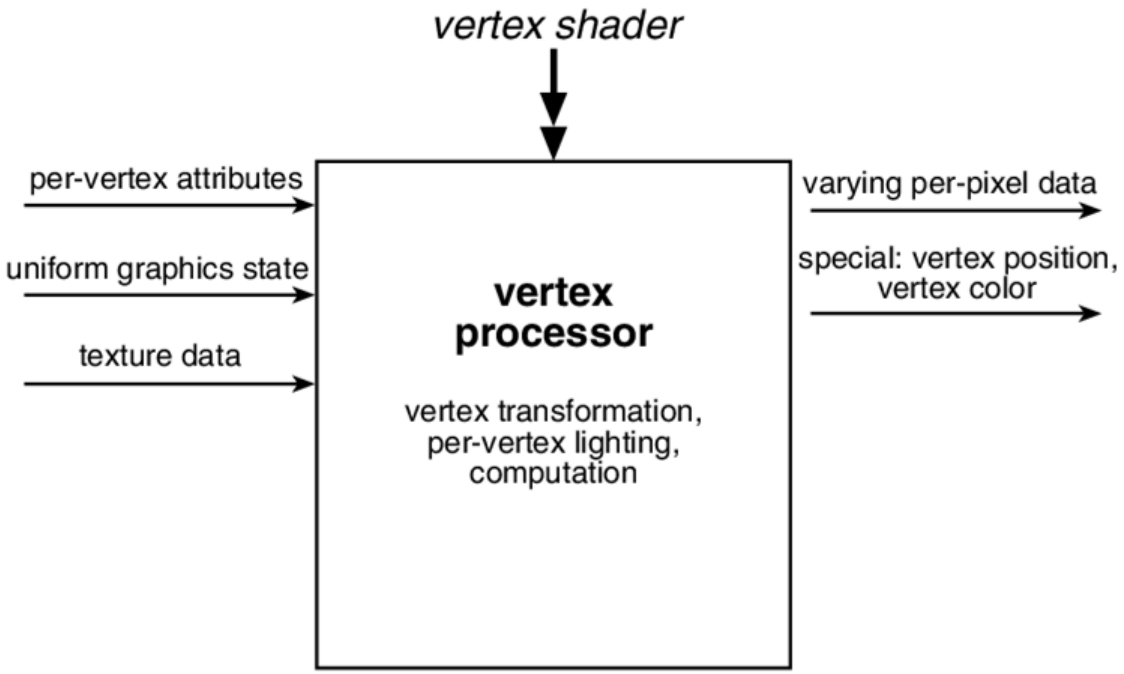
In this order

Shaders are part of the graphics state



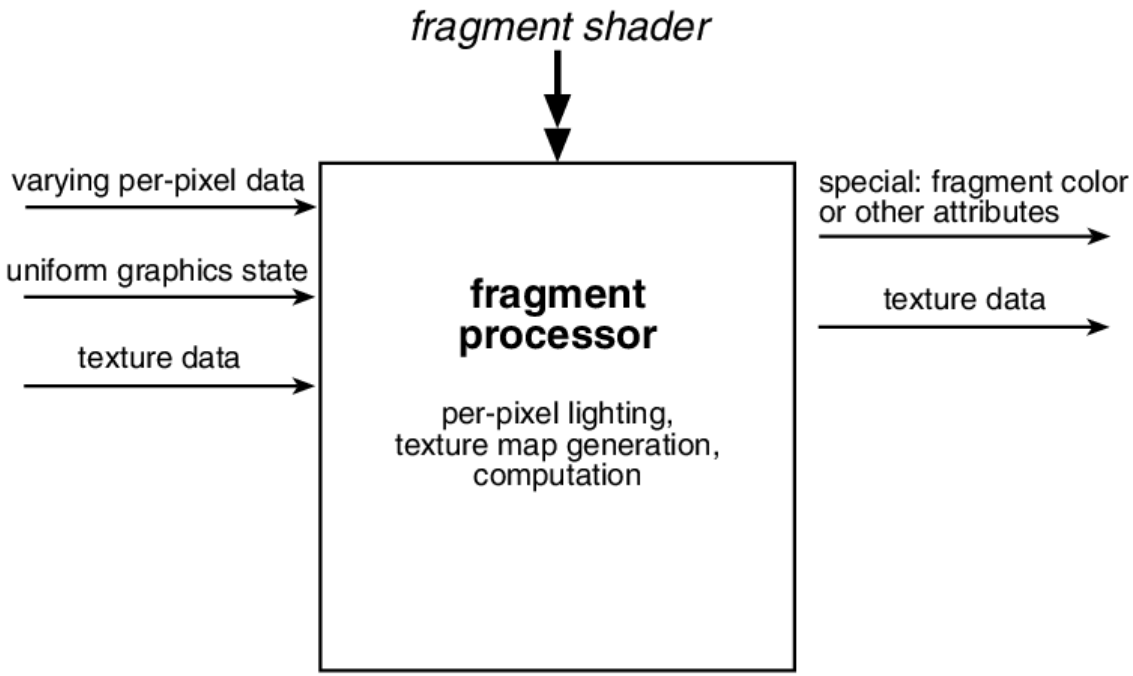
Vertex shader

- Vertex and normal transformations
  - Out: vertices transformed into screen coordinates
    - i.e. Multiplied by modelview and projection matrix
- Texture coordinate generation
- Per-vertex lighting calculations

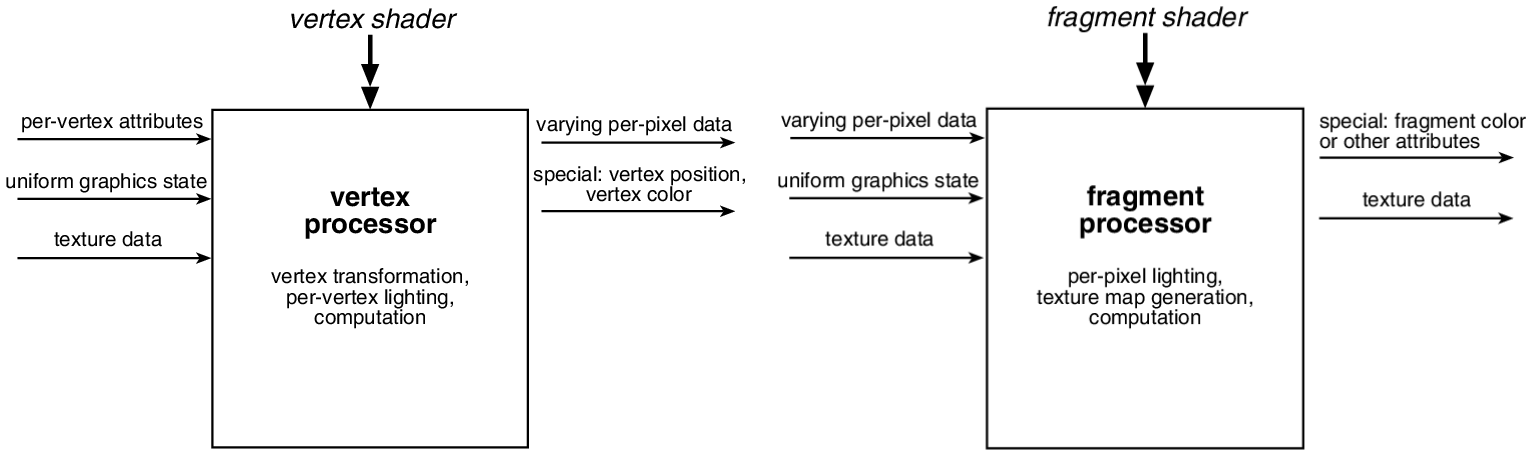


Fragment shader

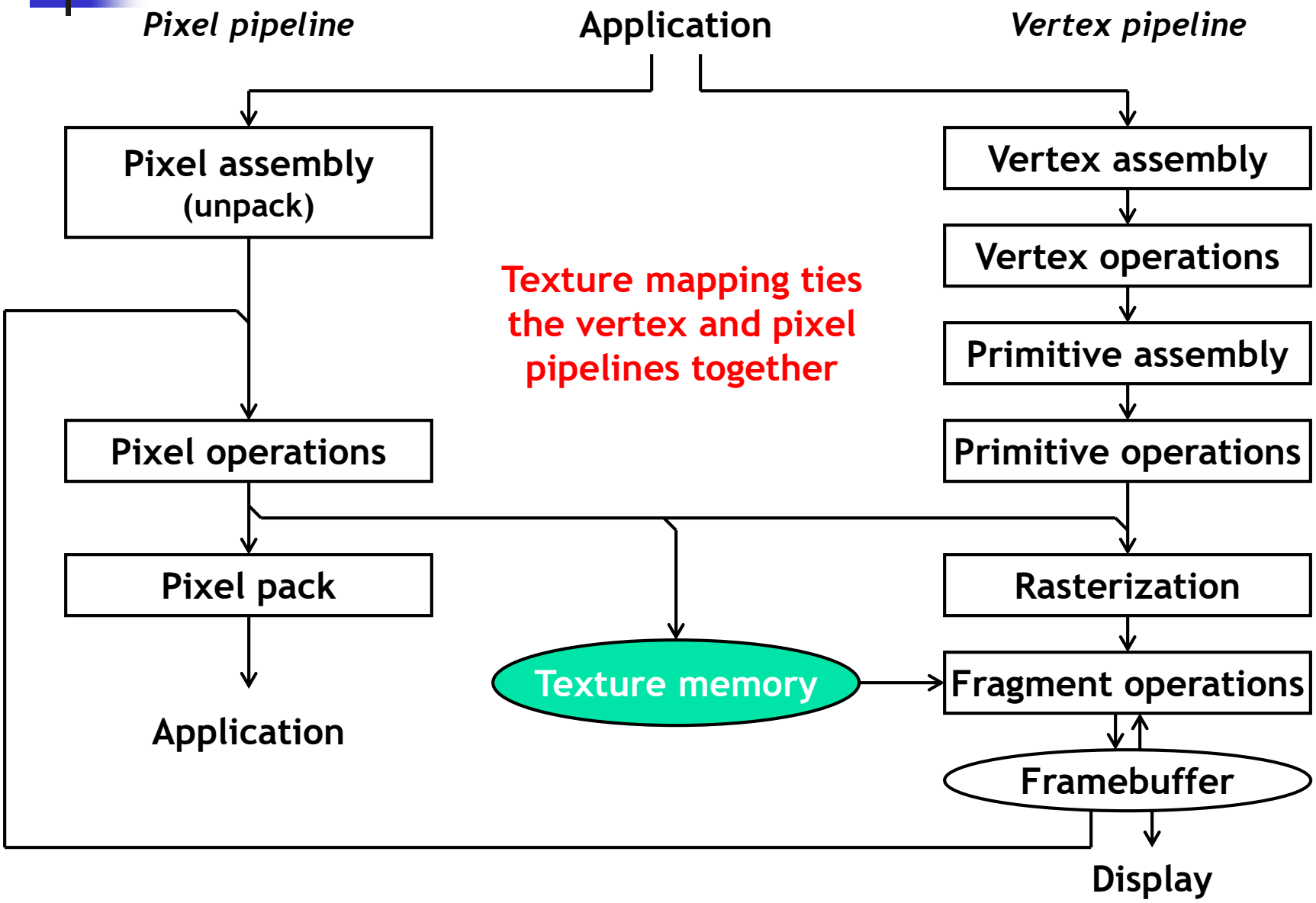
- Pixel shading
  - Out: vertex colour
- Texture application



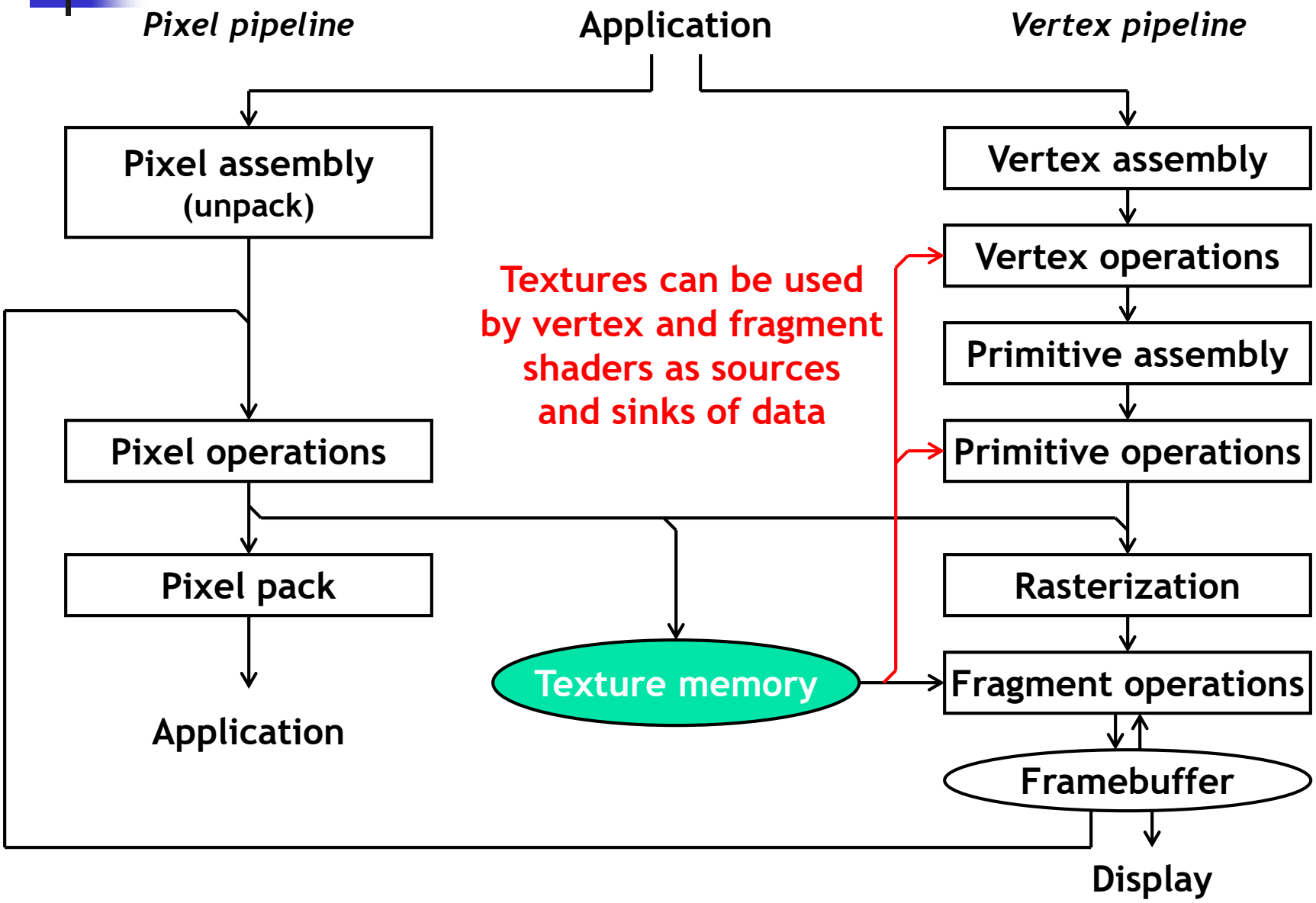
Data passed from vertex to fragment shader



Textures


Pixel pipeline
Application
Vertex pipeline
Pixel assembly (unpack)
Vertex assembly
Vertex operations
Primitive assembly
Primitive operations
Texture mapping ties the vertex and pixel pipelines together
Pixel operations
Pixel pack
Rasterization
Texture memory
Fragment operations
Framebuffer
Application
Display

Textures


Pixel pipeline
Application
Vertex pipeline
Pixel assembly (unpack)
Vertex assembly
Vertex operations
Primitive assembly
Primitive operations
Rasterization
Fragment operations
Framebuffer
Display
Pixel operations
Pixel pack
Application
Texture memory
Textures can be used by vertex and fragment shaders as sources and sinks of data

High-level shading languages

- OpenGL Shading Language: GLSL
  - OpenGL >= 2.0
- Microsoft's High Level Shading Language (HLSL)
  - DirectX
- NVIDIA's Cg
  - OpenGL and DirectX

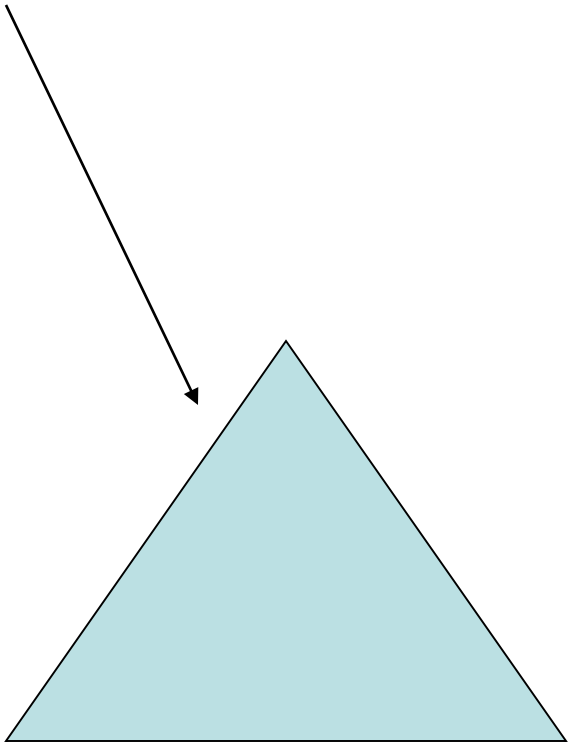
Matrix multiplication in shader assembly:

```
DP4 p[0].x, M[0], v[0];
DP4 p[0].y, M[1], v[0];
DP4 p[0].z, M[2], v[0];
DP4 p[0].w, M[3], v[0];
```

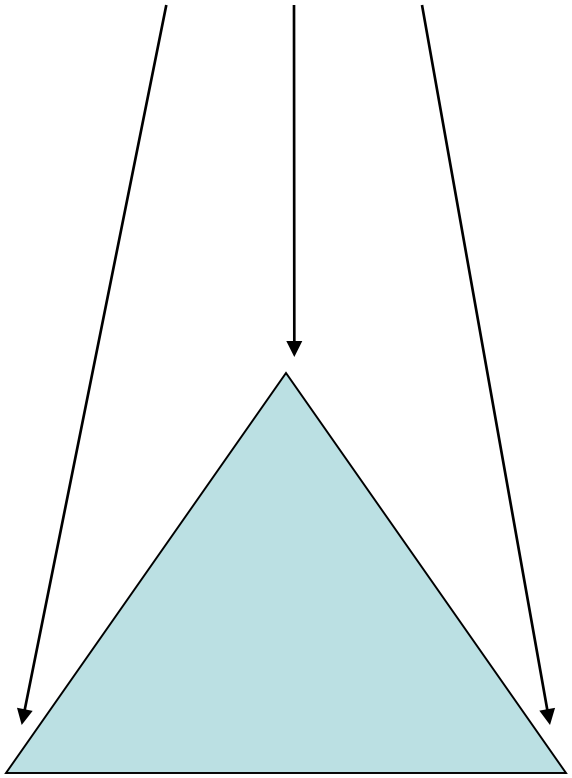
Matrix multiplication in GLSL:

```
p = M * v;
```

How many times are the fragment and vertex shaders executed on this scene?

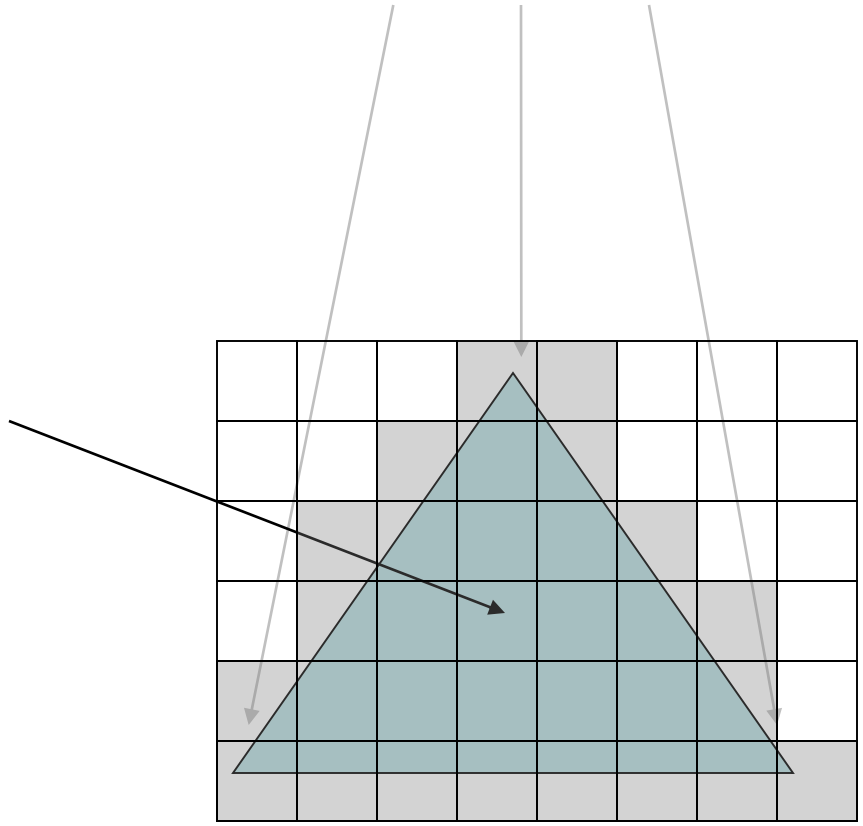


Vertex shader runs once per vertex



Vertex shader runs once per vertex

Fragment shader
runs once per pixel



Vertex Shader

```
in vec4 gl_Vertex;

void main(void) {

  gl_Position = gl_Vertex;

}
```

Output variables

Note:

Ignores modelview and projection matrices

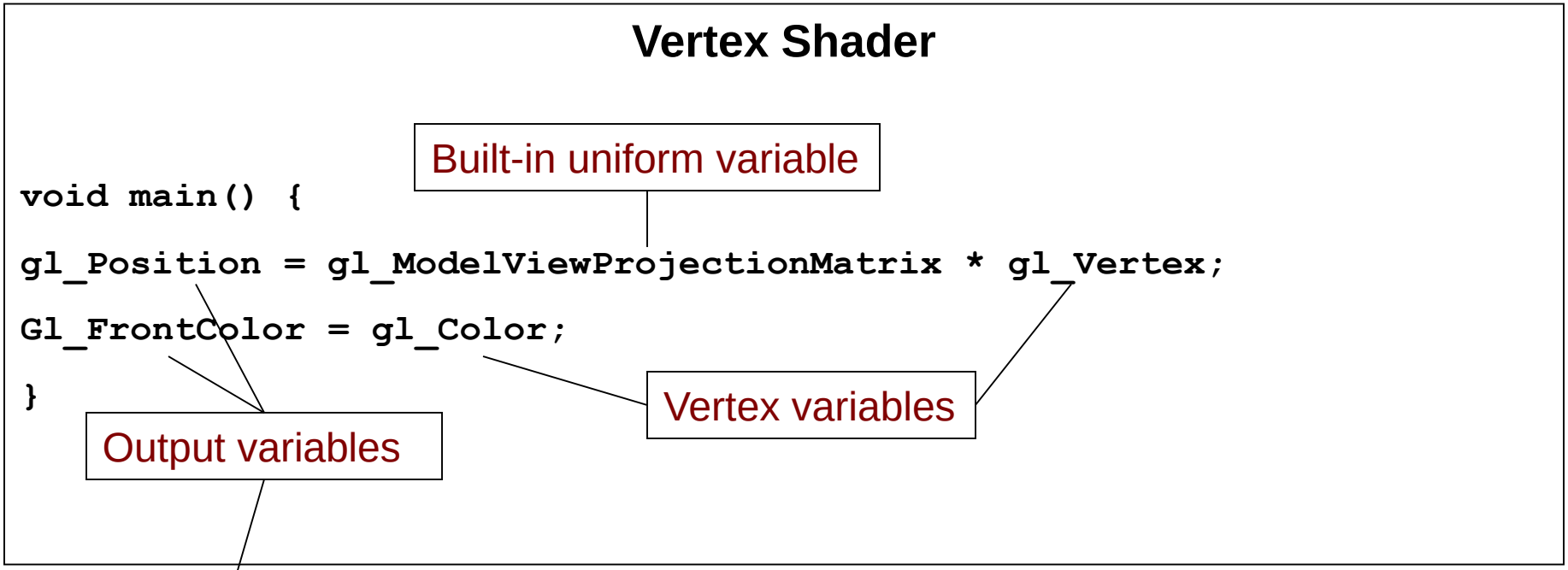
Fragment Shader

```
out vec4 gl_FragColor;

void main(void) {

  gl_FragColor = vec4(1, 0, 0, 1);

}
```

Note:

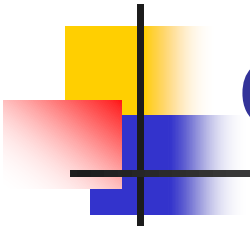
Ignores vertex color attributes

Vertex Shader



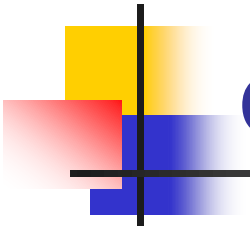
Fragment Shader

```
void main() {
gl_FragColor = vec4(1, 0, 0, 1);
}
```



GLSL Types

- Float types:
  - `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, `mat4`
- Bool types:
  - `bool`, `bvec2`, `bvec3`, `bvec4`
- Int types:
  - `int`, `ivec2`, `ivec3`, `ivec4`
- Texture types:
  - `sampler1D`, `sampler2D`, `sampler3D`



Characteristics of variables in shaders

- **`const`**
  - Constants
- **`attribute`**:
  - Data that changes on a per-vertex basis
    - Often tied to the graphics state
    - E.g. normal vector, texture coordinate
- **`uniform`**:
  - Data that does not change during shader execution
    - Set by the application, often through graphics state
    - E.g. modelview and projection matrix, location of light sources
- **`varying`**:
  - Data that is passed from vertex to fragment shader
    - Written by vertex shader on a per-vertex basis
    - Read by fragment shader as interpolated value across the face of a primitive between neighbouring vertices

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
gl_Position = gl_Vertex;
fragmentGrayness = vertexGrayness
}
```

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
const vec4 red = vec4(1, 0, 0, 1);
const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                    gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
gl_Position = gl_Vertex;
fragmentGrayness = vertexGrayness
}
```

Set once per vertex, like color, normal, or texcoord.

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
const vec4 red = vec4(1, 0, 0, 1);
const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                    gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
gl_Position = gl_Vertex;
fragmentGrayness = vertexGrayness
}
```

Set once per polygon, outside glBegin/glEnd

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
const vec4 red = vec4(1, 0, 0, 1);
const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                    gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
gl_Position = gl_Vertex;
fragmentGrayness = vertexGrayness
}
```

Matching Declarations

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
const vec4 red = vec4(1, 0, 0, 1);
const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                    gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;

varying float fragmentGrayness;

void main() {

gl_Position = gl_Vertex;

fragmentGrayness = vertexGrayness

}
```

Set at each vertex in vertex shader

Interpolated in hardware

Fragment Shader

```
uniform float brightness;

varying float fragmentGrayness;

void main() {

const vec4 red = vec4(1, 0, 0, 1);

const vec4 gray = vec4(0.5, 0.5, 0.5, 1);

gl_FragColor = brightness * (red * (1 - fragmentGrayness) +

                                    gray * fragmentGrayness);

}
```

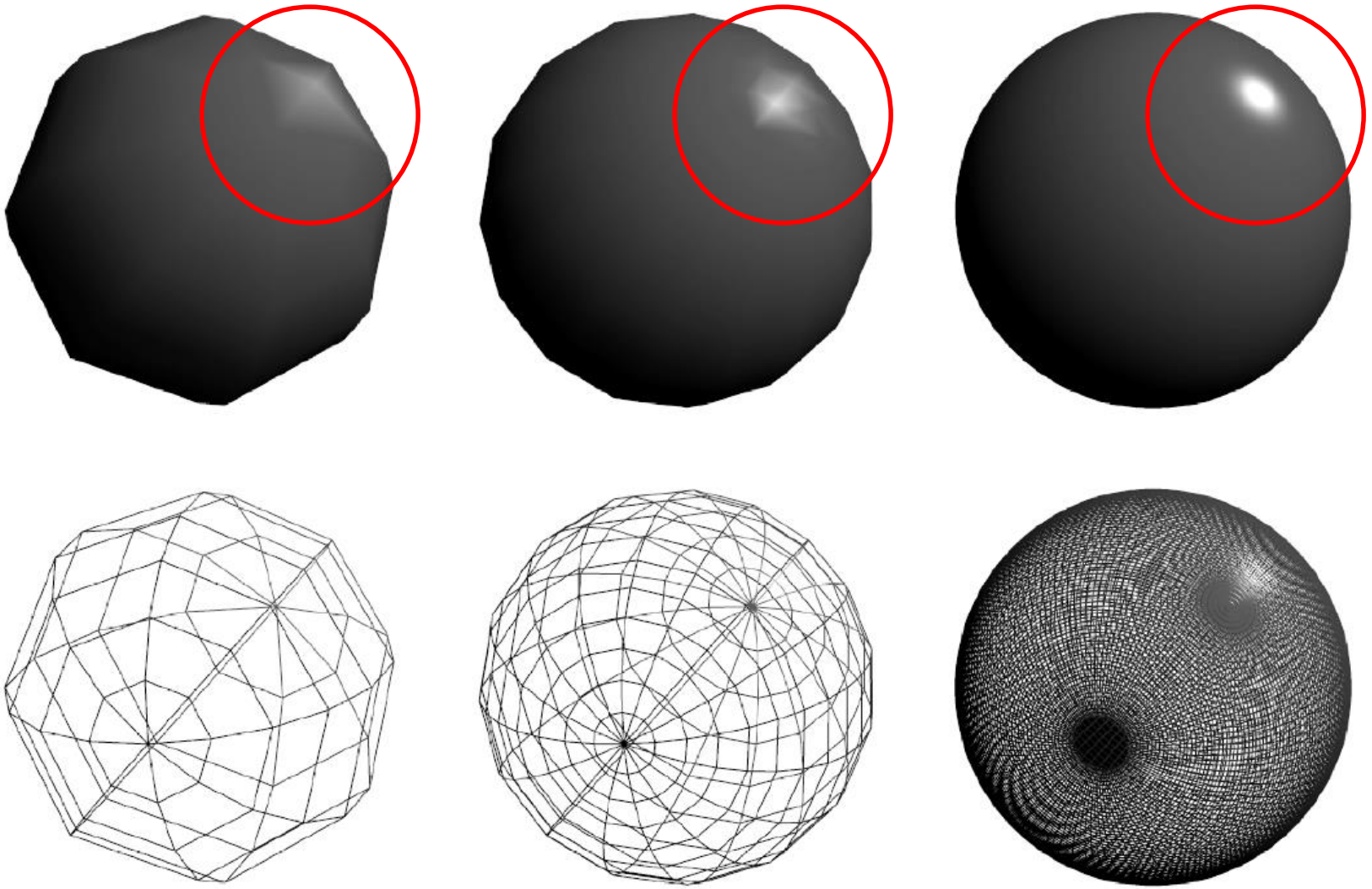
Available in fragment shader

Per-vertex Phong shading

Vertex Shader

```
void main(void)
{
   vec4 v = gl_ModelViewMatrix * gl_Vertex;
   vec3 n = normalize(gl_NormalMatrix * gl_Normal);
   vec3 l = normalize(gl_LightSource[0].position - v);
   vec3 h = normalize(l - normalize(v));

   float p = 16;
   vec4 cr = gl_FrontMaterial.diffuse;
   vec4 cl = gl_LightSource[0].diffuse;
   vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

   vec4 color;
   if (dot(h,n) > 0)
     color = cr * (ca + cl * max(0,dot(n,l))) +
             cl * pow(dot(h,n), p);
   else
     color = cr * (ca + cl * max(0,dot(n,l)));

   gl_FrontColor = color;
   gl_Position = ftransform();
}
```

Per-vertex Phong shading



Per-pixel Phong shading

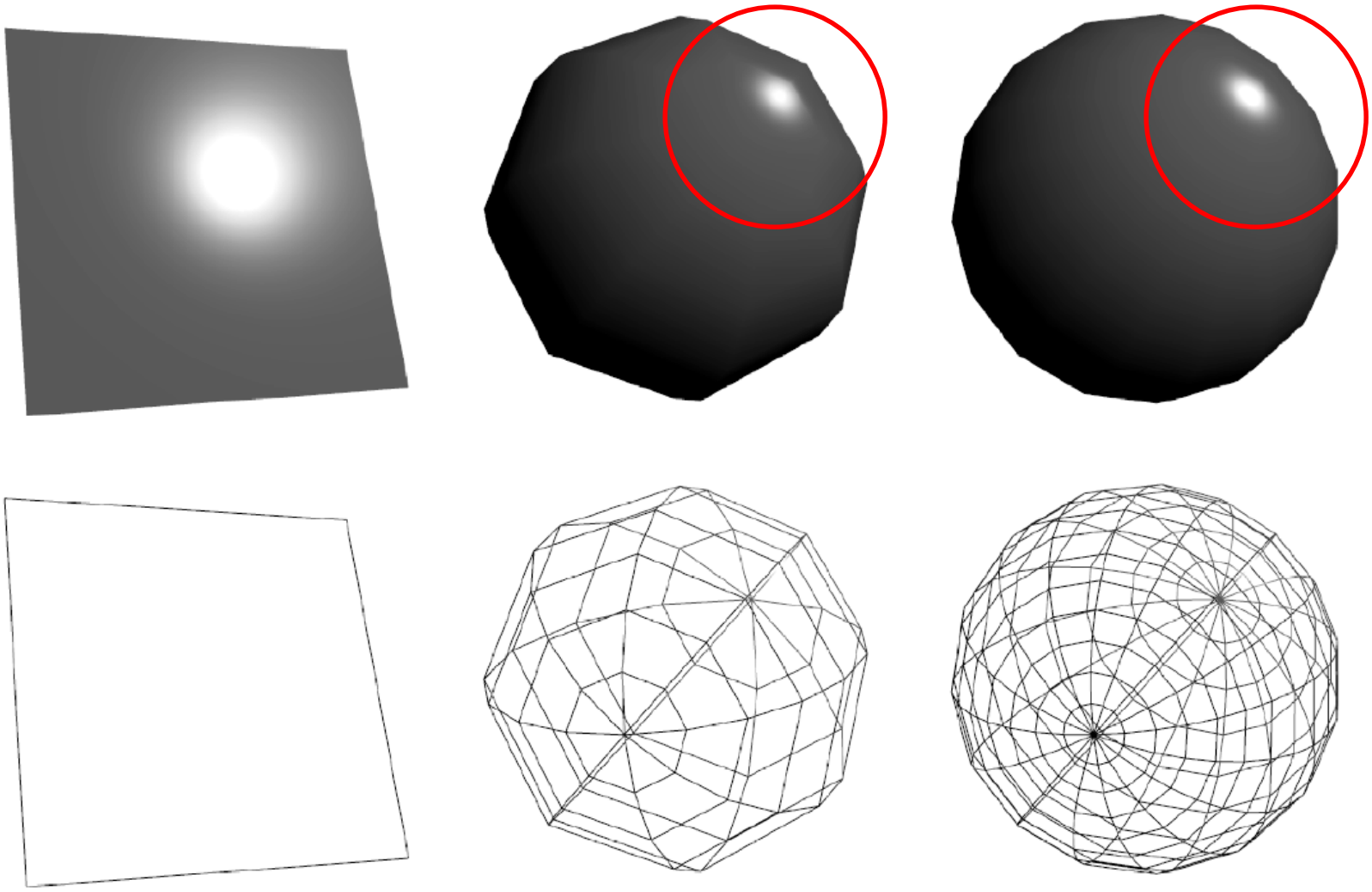
Vertex Shader

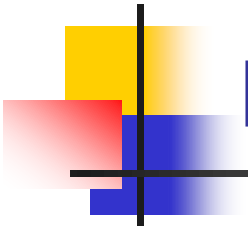
```
varying vec4 v;
varying vec3 n;
void main(void) {
   v = gl_ModelViewMatrix * gl_Vertex;
   n = normalize(gl_NormalMatrix * gl_Normal);
   gl_Position = ftransform();
}
```

Fragment Shader

```
varying vec4 v;
varying vec3 n;
void main(void) {
   vec3 l = normalize(gl_LightSource[0].position - v);
   vec3 h = normalize(l - normalize(v));

   float p = 16;
   vec4 cr = gl_FrontMaterial.diffuse;
   vec4 cl = gl_LightSource[0].diffuse;
   vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

   vec4 color;
   if (dot(h,n) > 0)
     color = cr * (ca + cl * max(0,dot(n,l))) + cl * pow(dot(h,n), p);
   else
     color = cr * (ca + cl * max(0,dot(n,l)));

   gl_FragColor = color;
}
```

Per-pixel Phong shading

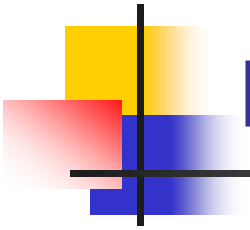




References

Creating your own shaders:

- Get the extension functions
  - use glew http://glew.sourceforge.net/
- Load the shaders as strings
- Compile them
- Link them into a ‘program’
- Enable the program
- Draw something



References

- A great tutorial and reference:
  - http://www.lighthouse3d.com/opengl/glsl/
- The spec for GLSL, includes all the available builtin functions and state:
  - http://www.opengl.org/documentation/glsl/
- OpenGL ‘orange book’
- http://nehe.gamedev.net/
- https://www.shadertoy.com/