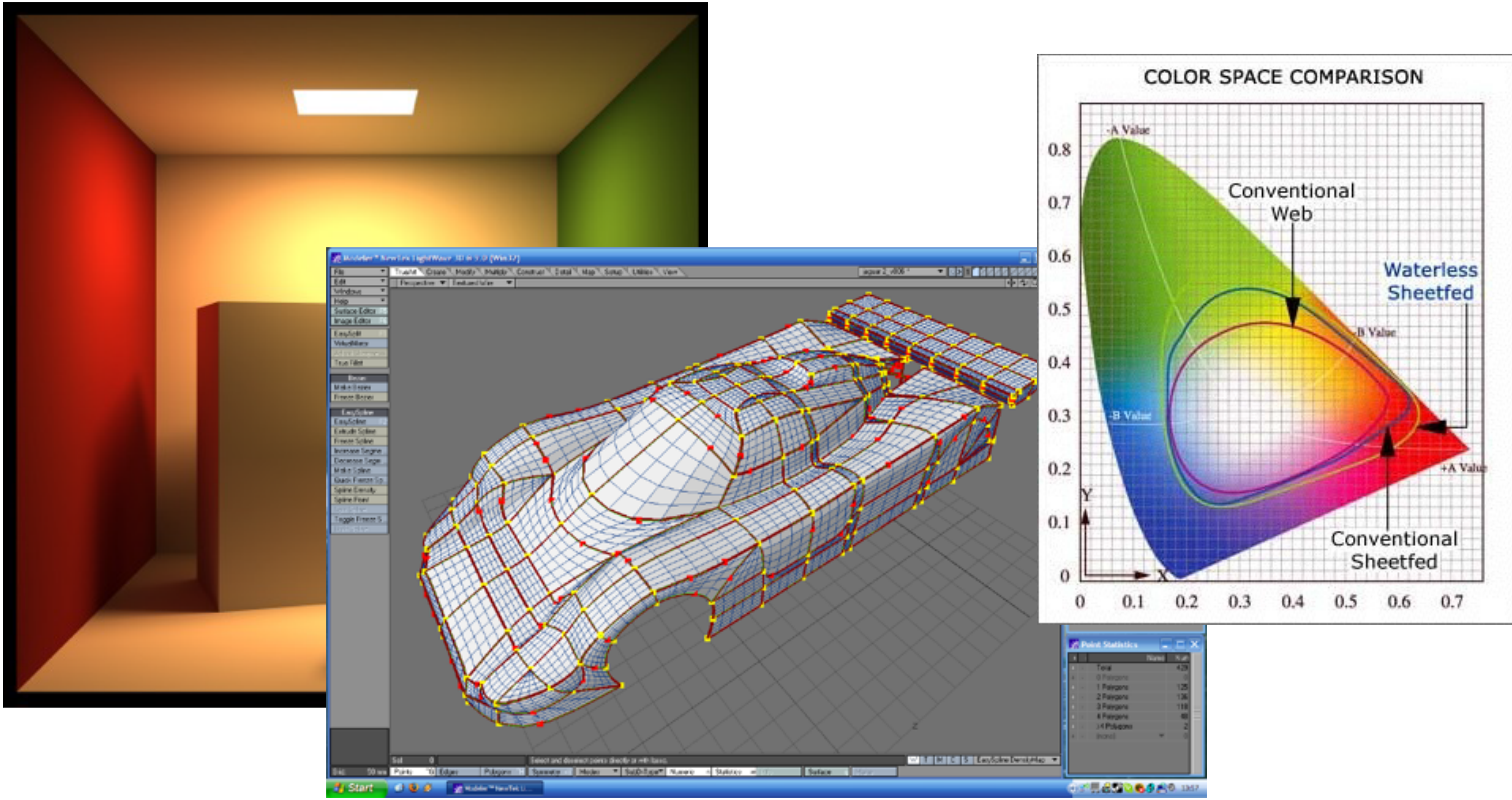


Lecture 18 : Shaders and GLSL

Tuesday November 9th 2021

Logistics

- HW#4 due tomorrow (Nov 10th)
- HW#5 coming shortly afterwards, will be due before Thanksgiving week.
  - Topic : Drawing in 3D
- HW#6 will be released to you before Thanksgiving, but due no earlier than (late) in the week after the holiday.
- Midterm grading: For sure by Saturday, hoping for earlier if all goes well.

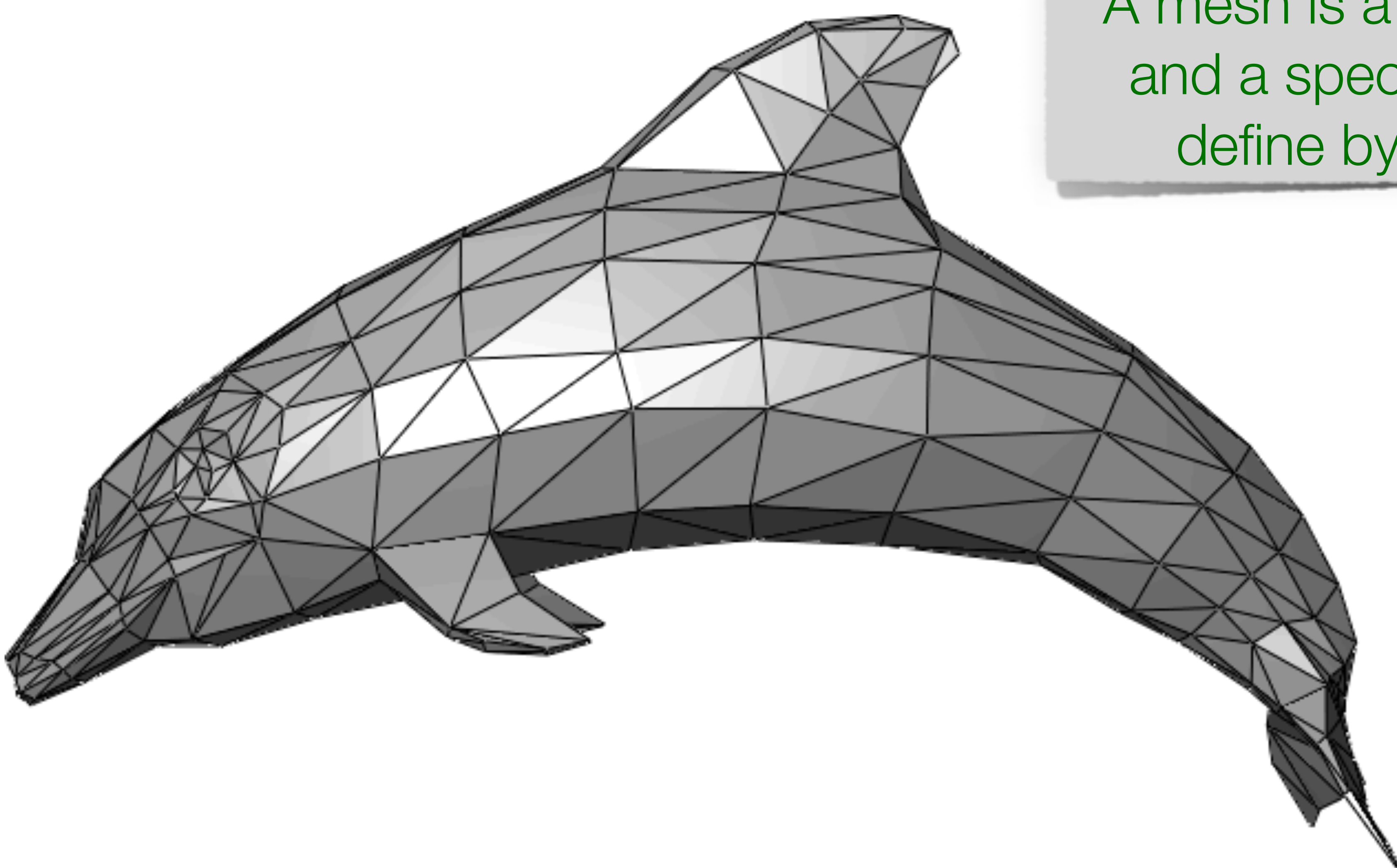
Today's lecture

- Triangles as drawing primitives, meshes (in brief)
- We will discuss the OpenGL Shading Language, for vertex and fragment shaders
- Demos of shaders

Shading, rasterization, the graphics pipeline and GLSL

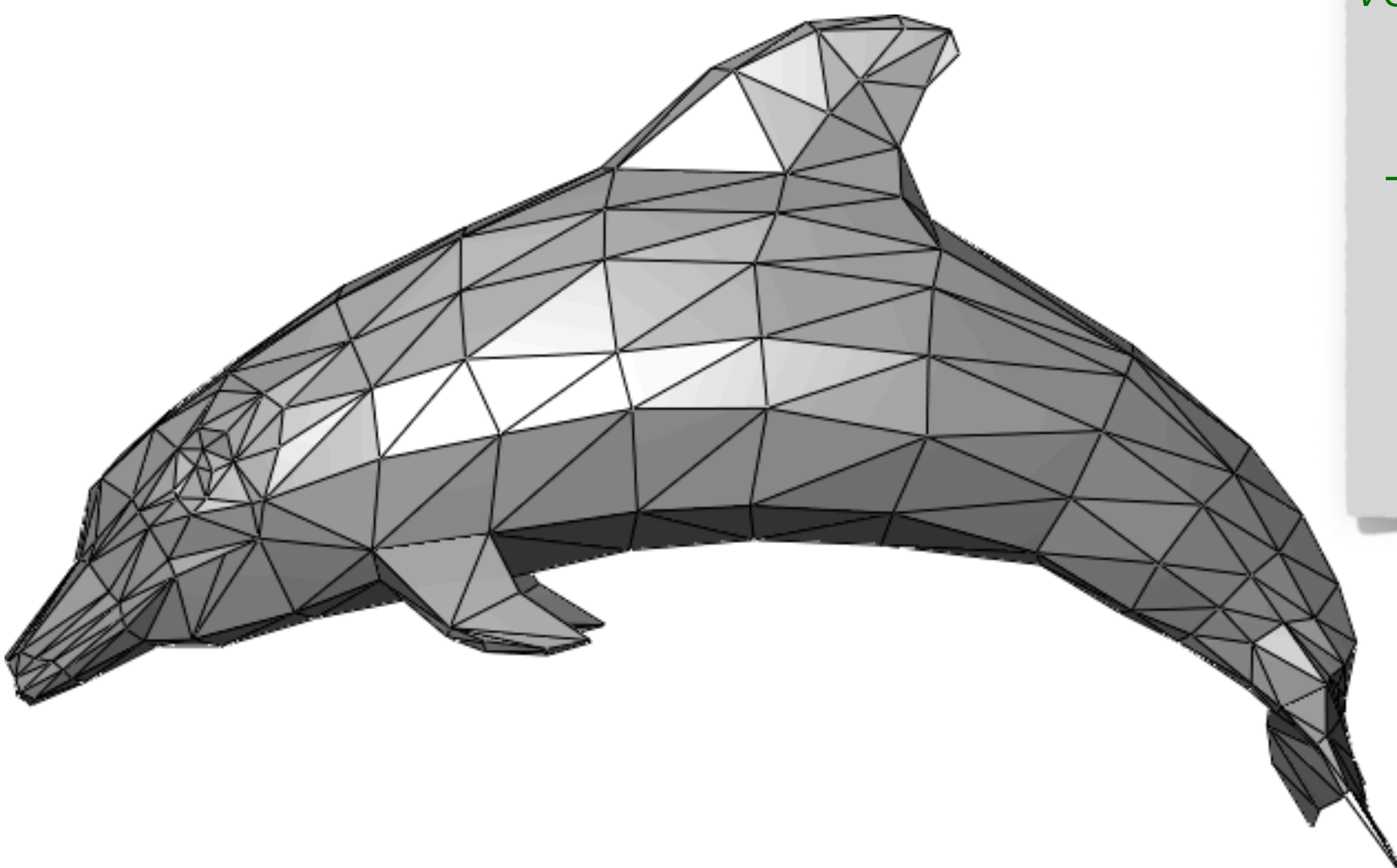
The pursuit of interactive graphics (and the design choices that prevailed among GPUs) have emphasized a different modeling paradigm: *triangle (or polygon) meshes*

A mesh is a collection of points (vertices), and a specification of polygons that we define by connecting such vertices.



Shading, rasterization, the graphics pipeline and GLSL

Motivation for triangle meshes:
They are expressive enough to describe intricate 3D objects, and they use a simple enough building block (triangles …) that can be made very fast!



Vertices carry *geometric information*, i.e. (x,y,z) location, and sometimes additional *vertex properties* (color, normal, texture, etc)

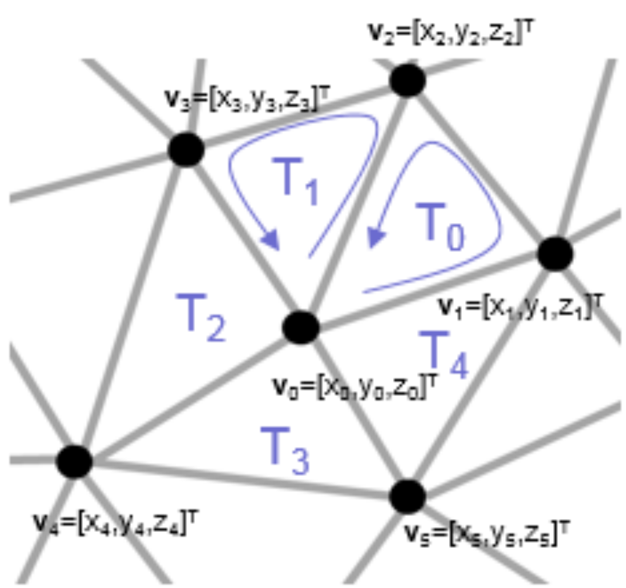
Triangles typically just identify the vertices they connect (specifying just “topology”) rather than reiterating their properties

Shading, rasterization, the graphics pipeline and GLSL

Motivation for triangle meshes:
They are expressive enough to describe intricate 3D objects, and they use a simple enough building block (triangles …) that can be made very fast!

Vertices carry *geometric information*, i.e. (x,y,z) location, and sometimes additional *vertex properties* (color, normal, texture, etc)

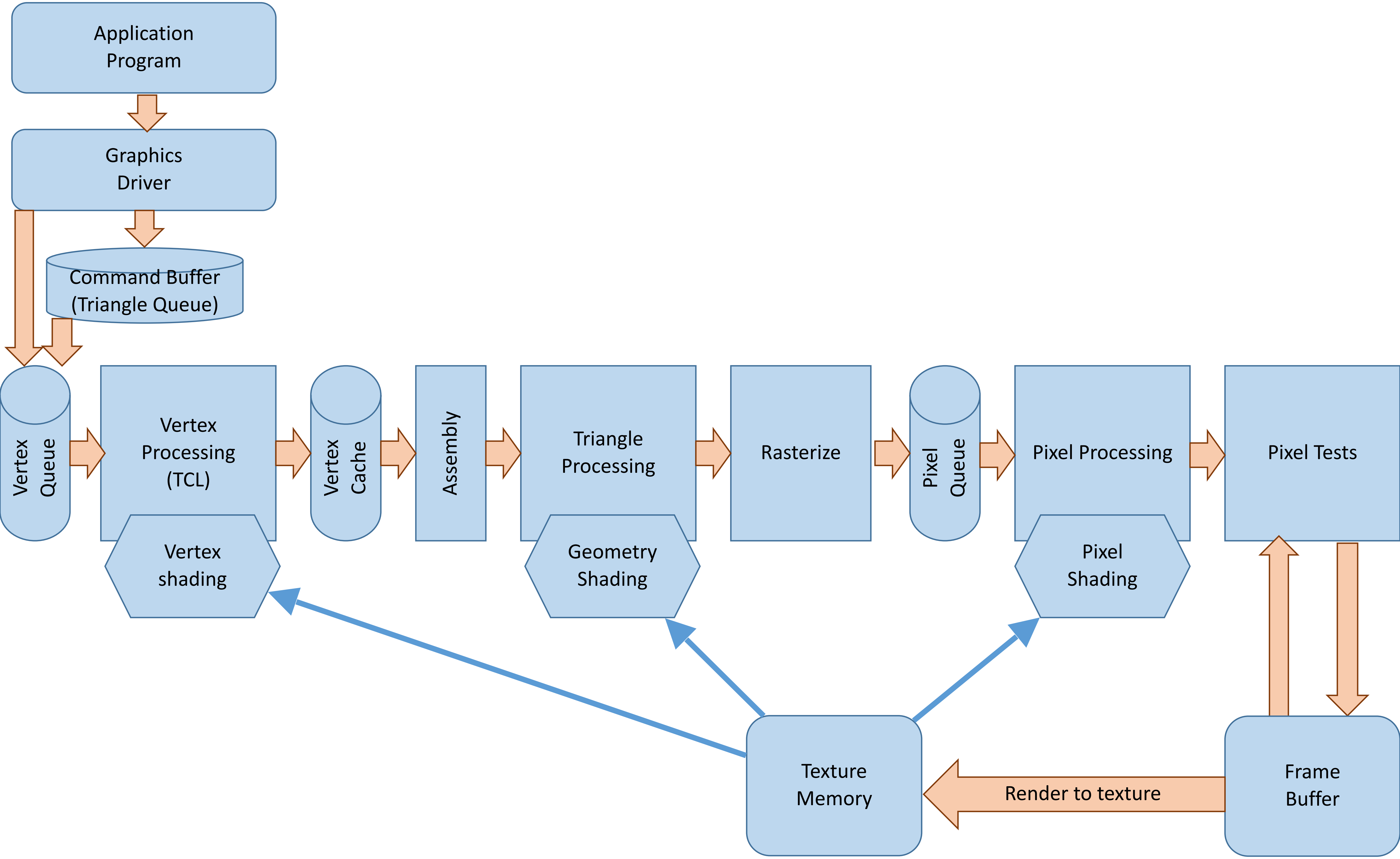
Triangles typically just identify the vertices they connect (specifying just “topology”) rather than reiterating their properties



	Vertex list
v_0	x_0,y_0,z_0
v_1	x_1,y_1,z_1
v_2	x_2,y_2,z_2
v_3	x_3,y_3,z_3
v_4	x_4,y_4,z_4
v_5	x_5,y_5,z_5
	...

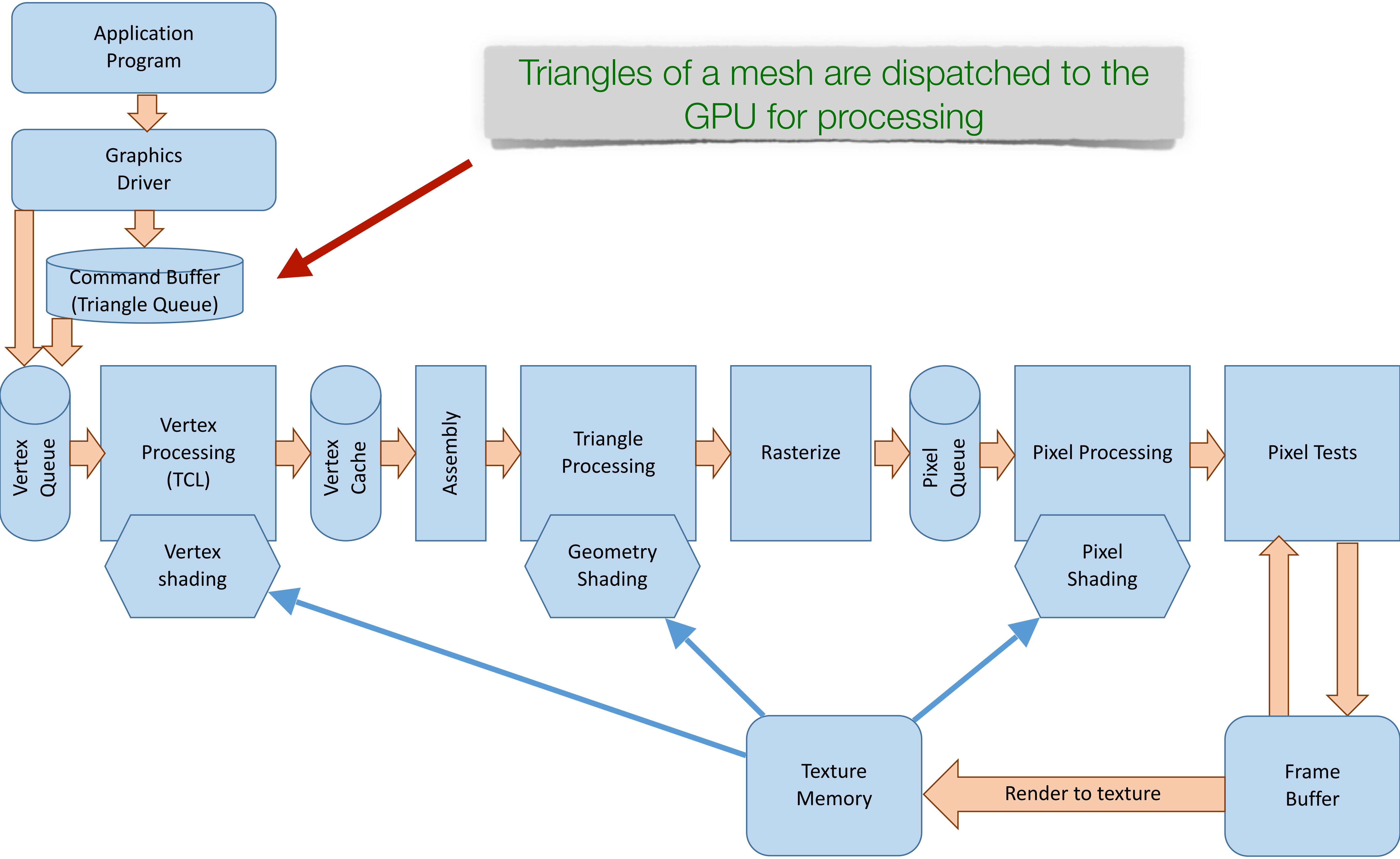
	Triangle list
T_0	0, 1, 2
T_1	0, 2, 3
T_2	4, 0, 3
T_3	0, 4, 5
T_4	0, 5, 1
	...

The (GPU) graphics pipeline



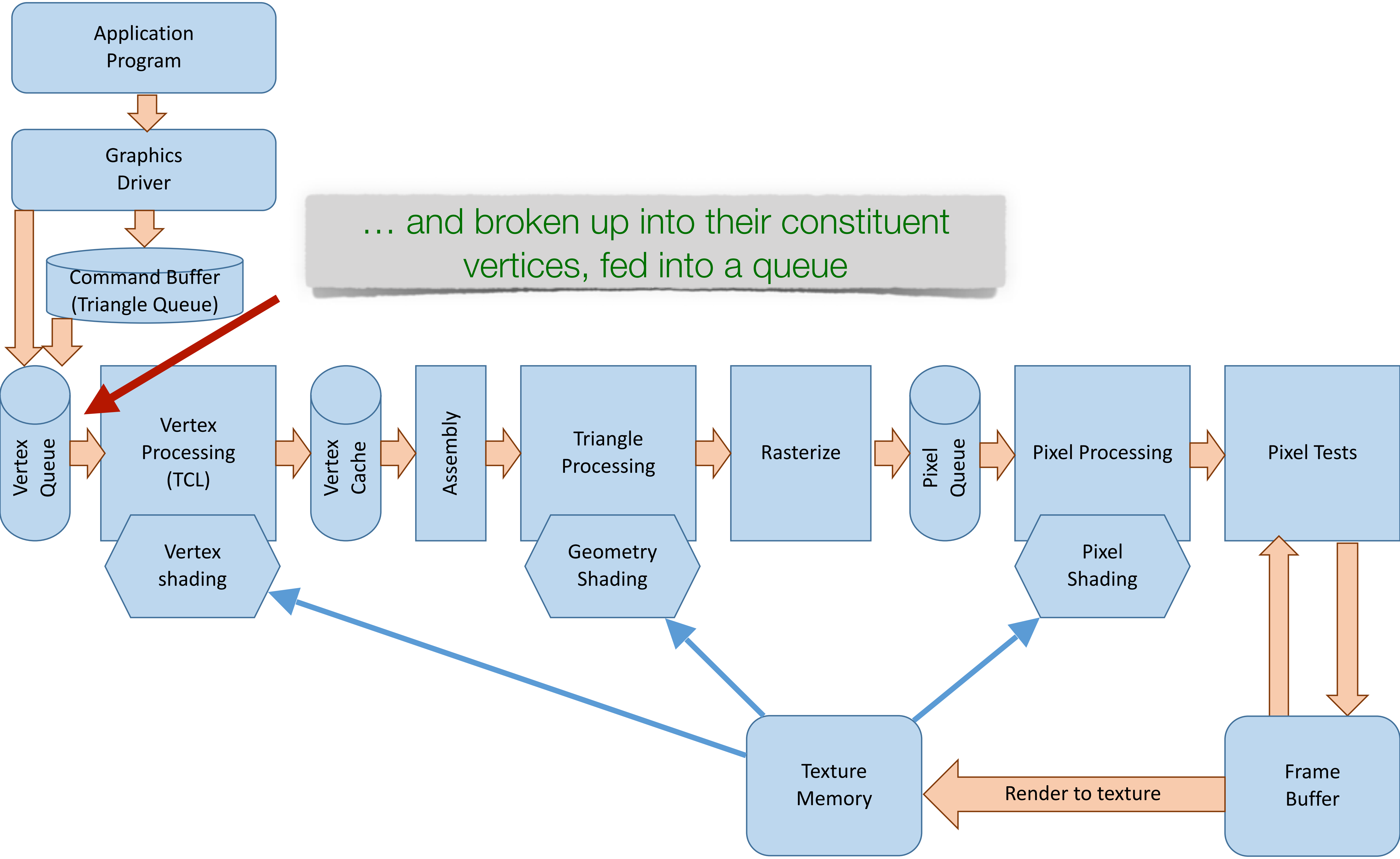
The (GPU) graphics pipeline

Triangles of a mesh are dispatched to the GPU for processing



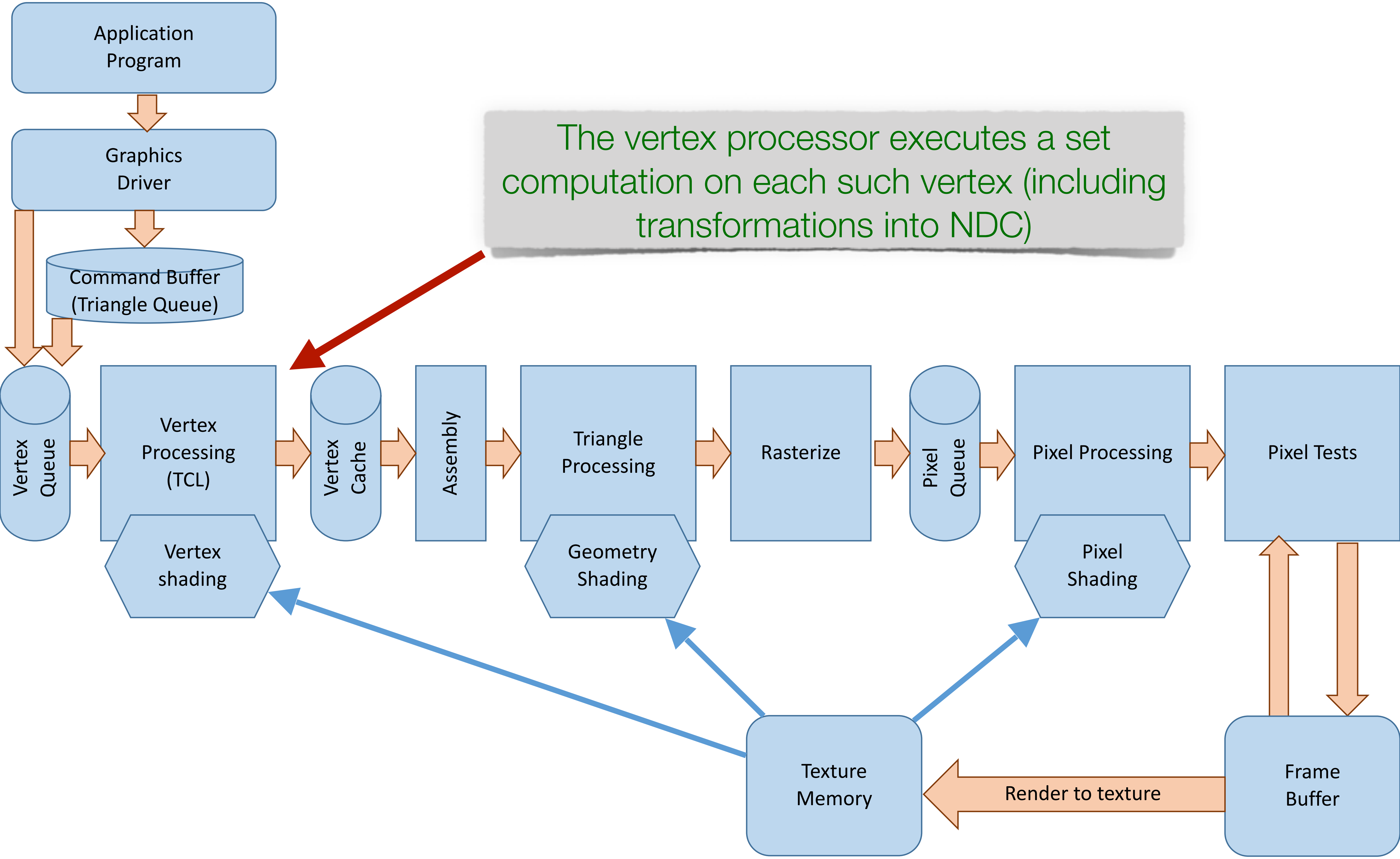
The (GPU) graphics pipeline

… and broken up into their constituent vertices, fed into a queue



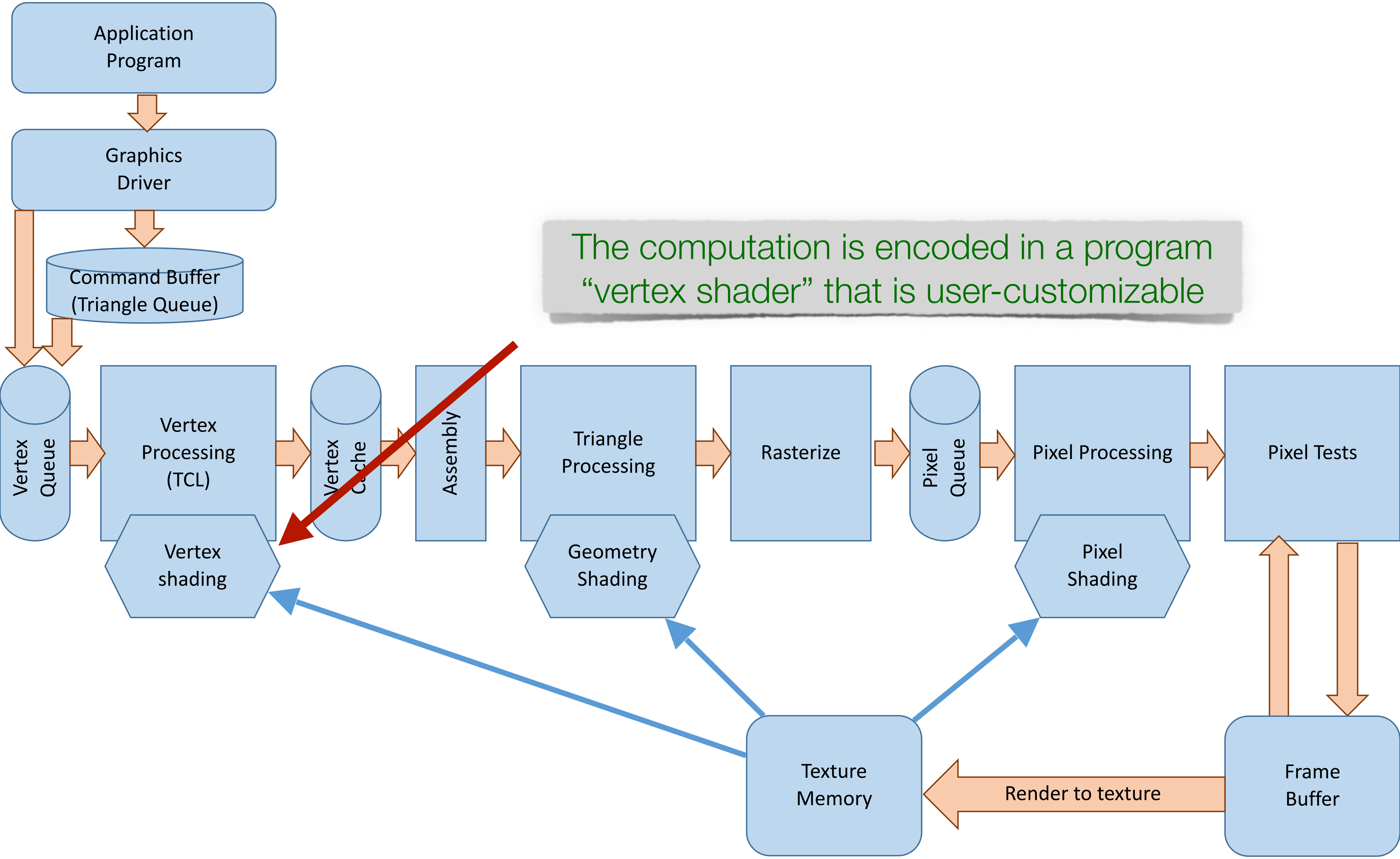
The (GPU) graphics pipeline

The vertex processor executes a set computation on each such vertex (including transformations into NDC)



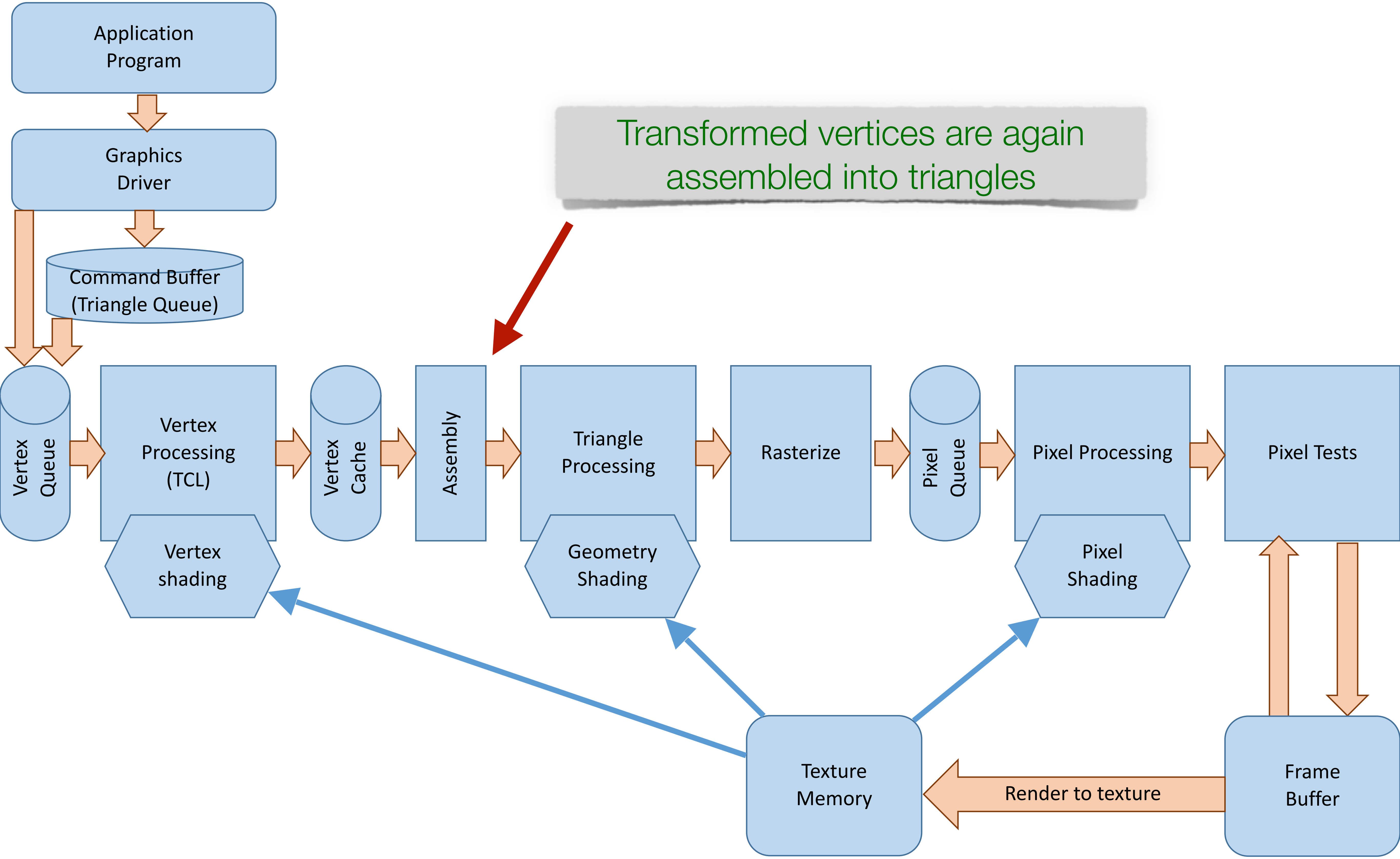
The (GPU) graphics pipeline

The computation is encoded in a program
"vertex shader" that is user-customizable



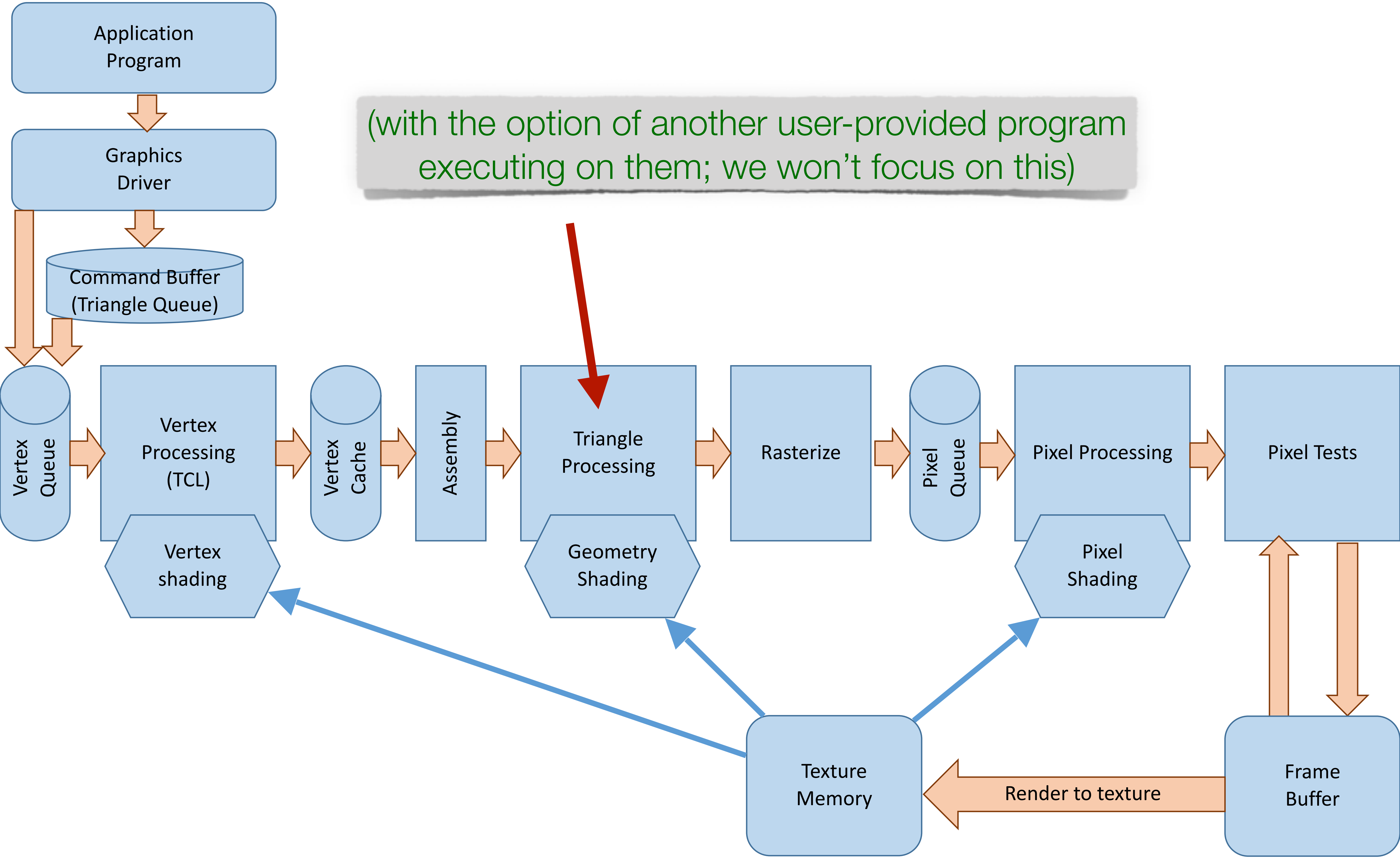
The (GPU) graphics pipeline

Transformed vertices are again assembled into triangles



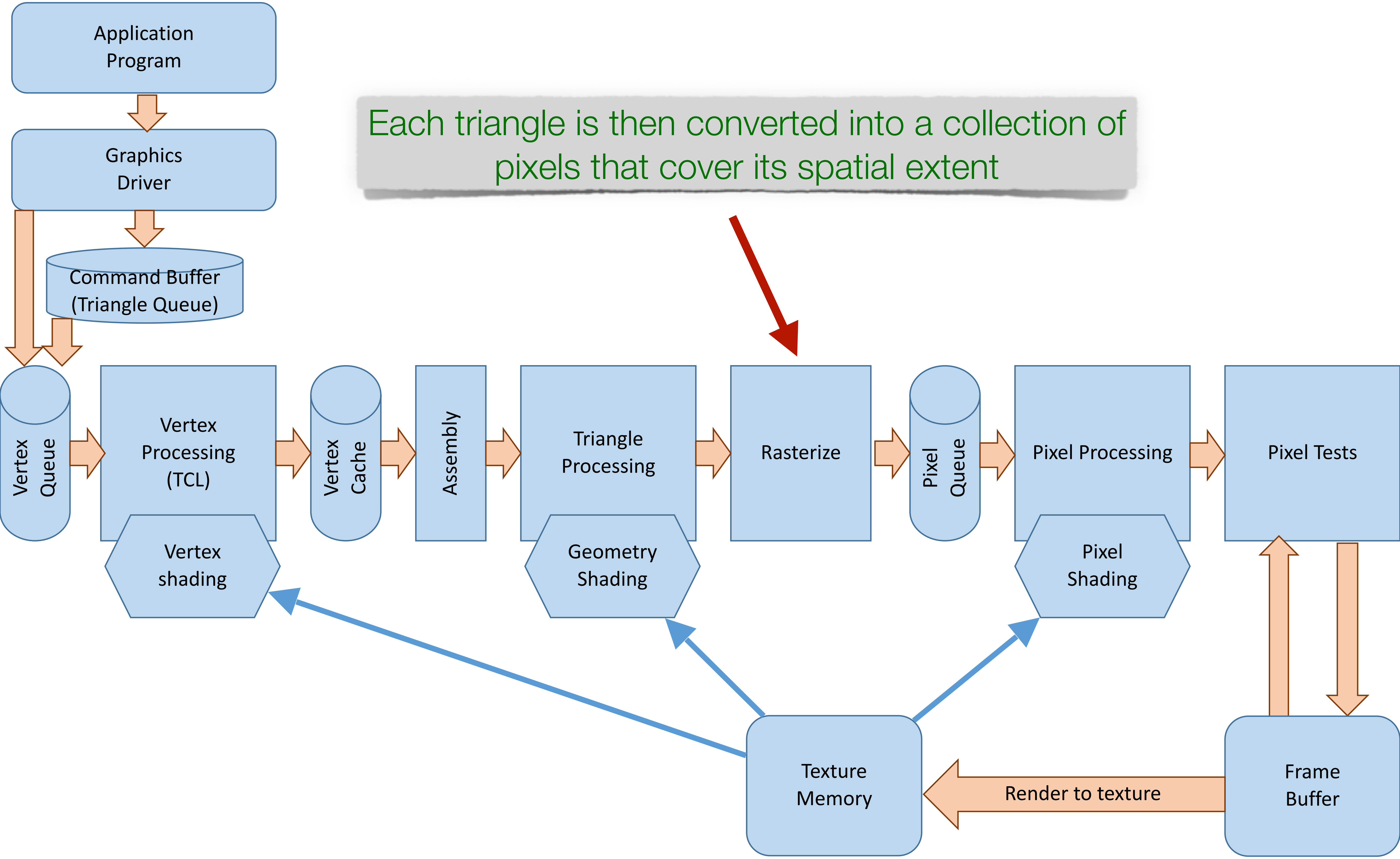
The (GPU) graphics pipeline

(with the option of another user-provided program executing on them; we won't focus on this)



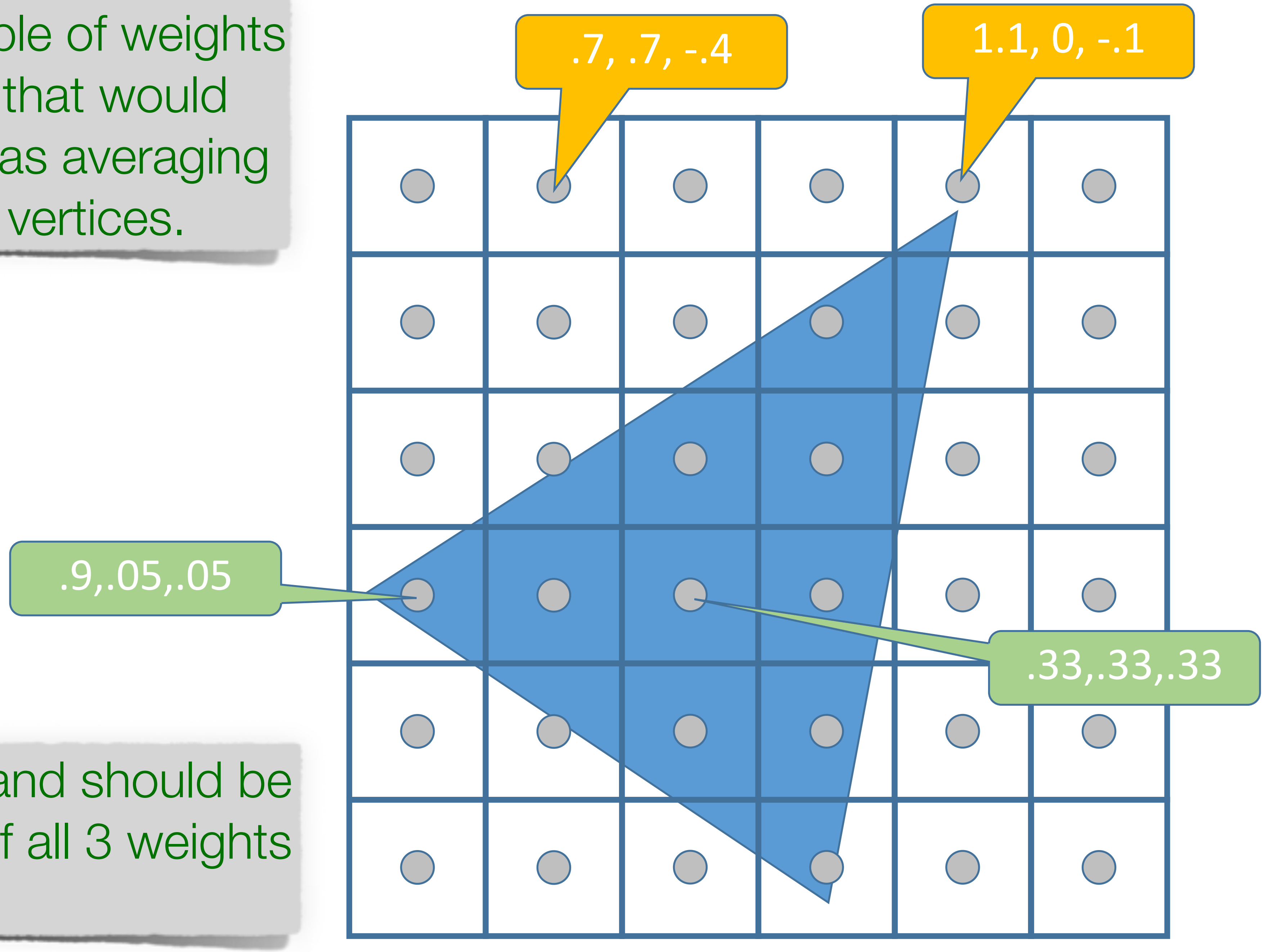
The (GPU) graphics pipeline

Each triangle is then converted into a collection of pixels that cover its spatial extent



Rasterization (in hardware)

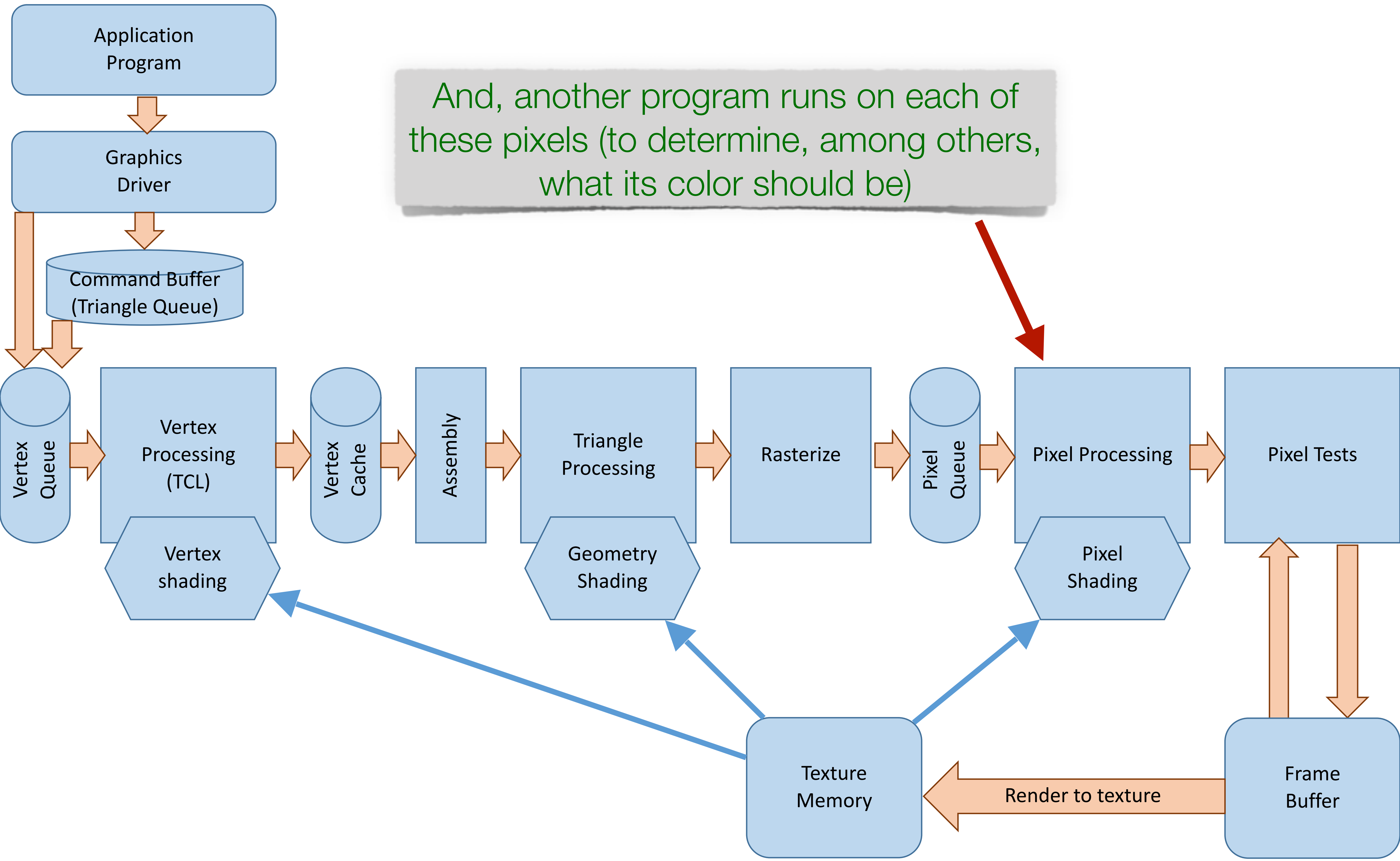
For every pixel, compute a triple of weights ("barycentric coordinates") that would reconstruct the point if used as averaging weights from the triangle vertices.



A pixel is *inside the triangle* (and should be "rasterized" into a fragment) if all 3 weights are positive.

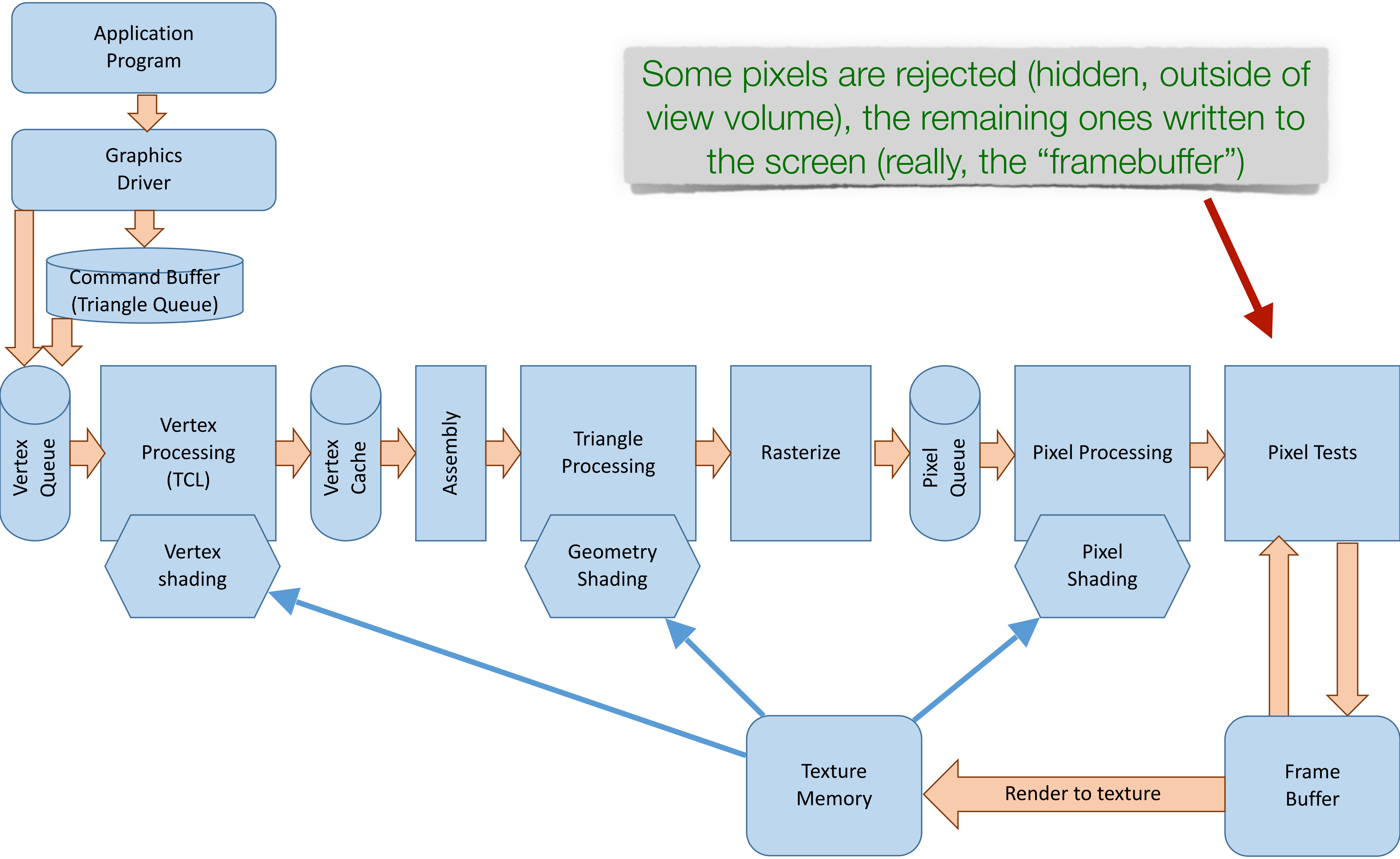
The (GPU) graphics pipeline

And, another program runs on each of these pixels (to determine, among others, what its color should be)

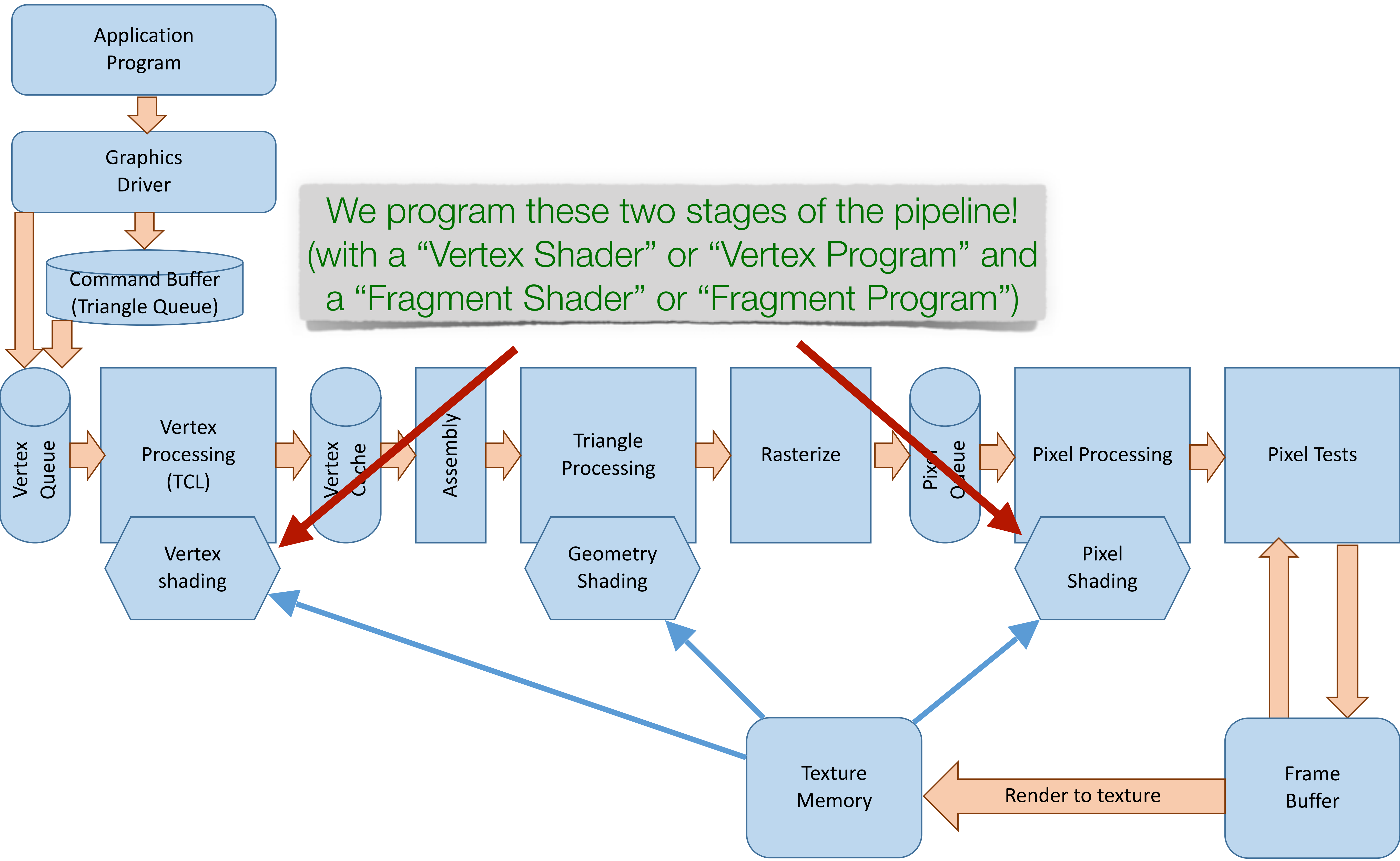


The (GPU) graphics pipeline

Some pixels are rejected (hidden, outside of view volume), the remaining ones written to the screen (really, the "framebuffer")

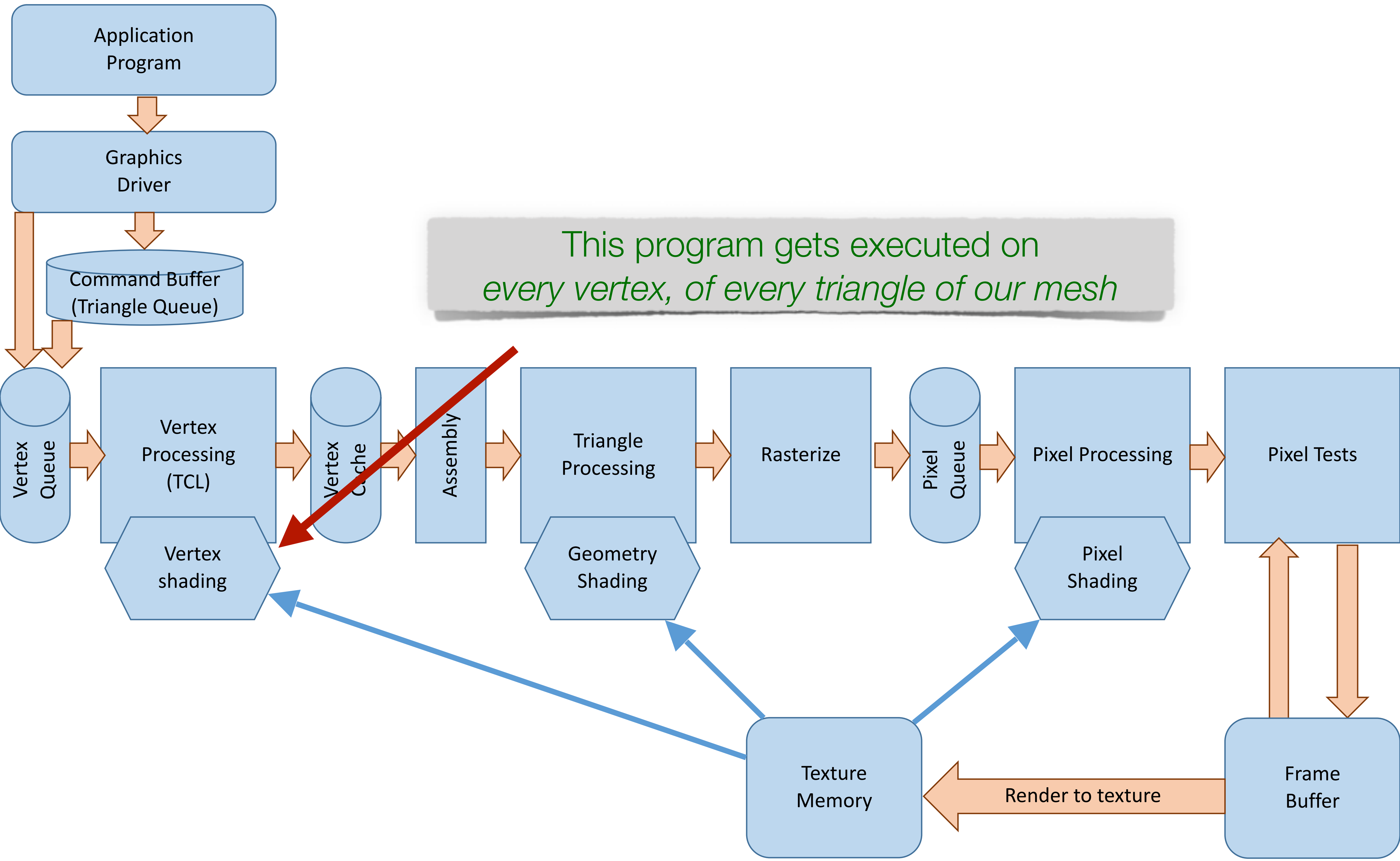


The (GPU) graphics pipeline



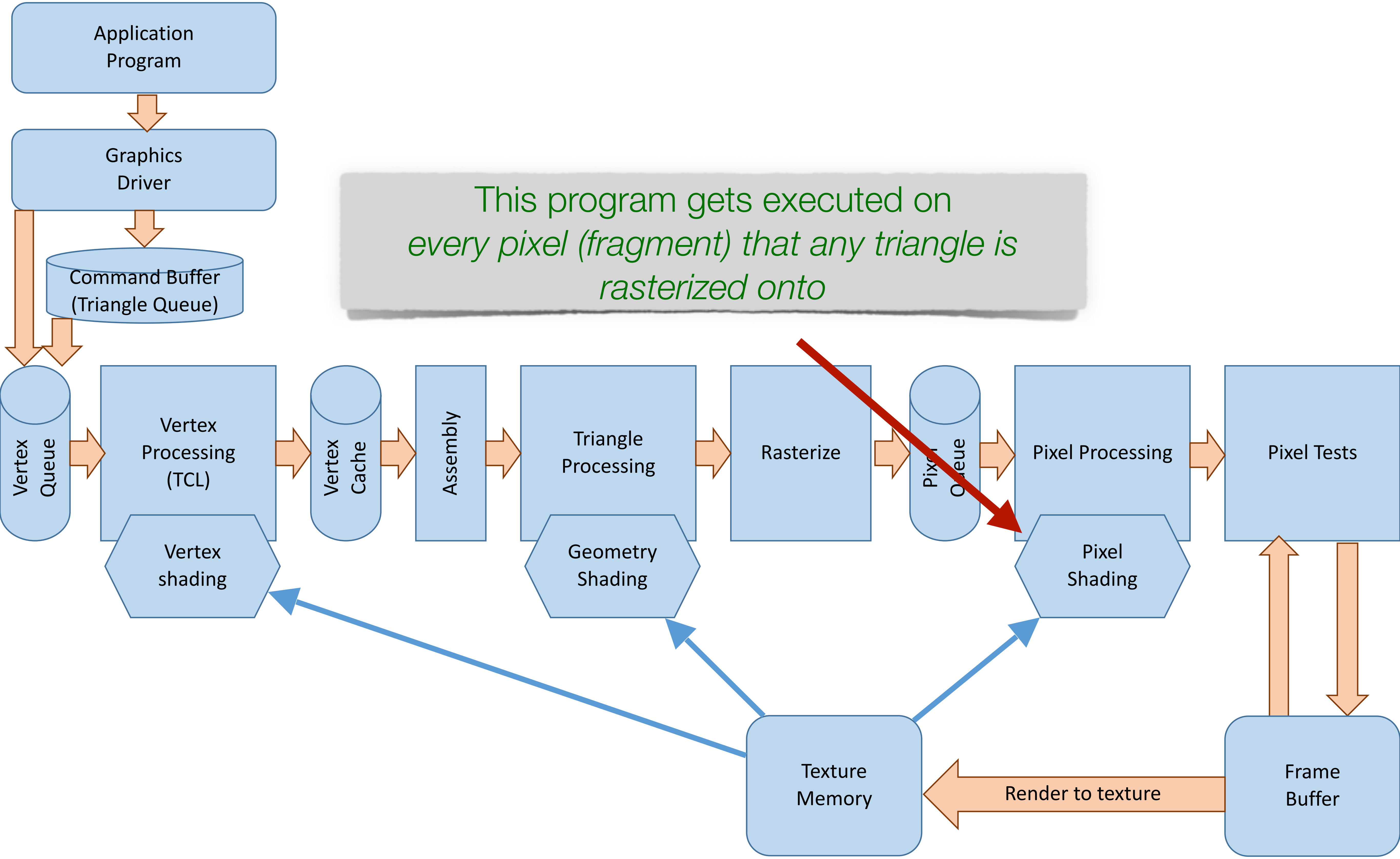
The (GPU) graphics pipeline

This program gets executed on
every vertex, of every triangle of our mesh



The (GPU) graphics pipeline

This program gets executed on
every pixel (fragment) that any triangle is rasterized onto



GLSL - The OpenGL Shading Language

- A special language for shader programs
- Used both for vertex and fragment shaders
- The compiler is built directly into the graphics driver

- We will show demonstrations using shdr.bkcore.com

GLSL Basics

- The main() function is the entry point for both vertex and fragment shaders
- You can have other, user-defined functions, too!
- main() takes no arguments, does not return anything; shaders interface with the outside world via *variables*

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

GLSL Basics

- The main() function is the entry point for both vertex and fragment shaders
- You can have other, user-defined functions, too!
- main() takes no arguments, does not return anything; shaders interface with the outside world via *variables*

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

The GLSL Type System

- Strongly and Strictly Typed!
- Note that floats/ints are different
- Explicitly cast all conversions
- Scalar data types
  - void, int, float, bool …
- Matrix/Vector data types
  - vec2, vec3, vec4
  - mat3, mat4
  - (less common) ivec2, bvec3, …

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

Vector operations

- We can easily access subsets of vectors, or make a vector by concatenating vector components

```
vec3 v;
float a = v.x; // access a component

vec2 b = v.xy; // any subset
vec2 c = v.yz;

vec3 d = v.zyx; // any order (swizzle)
vec3 e = v.xxx; // even repeats
```

- Vector components indexed as “x/y/z/w” or “r/g/b/a”
- More complex operations via explicit cast

```
vec2 a,b;
vec3 c;

vec3 f = vec3(1.0,2.0,3.0);
vec3 g = vec3(1,2,3); // rare times an integer works
vec3 h = vec3(a,1);
vec3 j = vec3(1,a);

vec4 k = vec4(a,b);
vec4 l = vec4(a,c.xy);
```

Linear algebra

- Many linear algebra operations are built-in, using intuitive syntax

```
vec4 x;
vec3 p;
mat4 m;

vec4 y = M * x;
vec4 z = M * vec4(p,1);

float a = dot(x, vec4(p,0));
```

- For more built-in operations and functions, check out the GLSL Reference Card (e.g. Version 4.5)
  - e.g. dot(), normalize(), min(), max(), length() …

Type qualifiers

- Very important to understanding how shaders work!
- Four main types: const/uniform/attribute/varying
- (we'll discuss "varying" last ...!)

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

Type qualifiers

goo.gl/gMWgIO

(fragment shader)

```
precision highp float;
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;
varying vec3 rawX;

const vec3  lightV1    = vec3(0.0,1.0,0.0); // stationary light
const float lightI     = 1.0;               // only for diffuse component
const float ambientC   = 0.15;
const float diffuseC   = 0.7;
const float specularC1 = 1.0;               // For stationary light
const float specularE1 = 64.0;
const float specularE2 = 16.0;
const vec3  lightCol   = vec3(1.0,1.0,1.0);
const vec3  objectCol  = vec3(1.0,0.6,0.0); // yellow-ish orange

vec2 blinnPhongDir(vec3 lightDir, float lightInt, float Ka, float Kd, float Ks, float shininess)
{
  vec3 s = normalize(lightDir);
  vec3 v = normalize(-fPosition);
  vec3 n = normalize(fNormal);
  vec3 h = normalize(v+s);
  float diffuse = Ka + Kd * lightInt * max(0.0, dot(n, s));
  float spec =  Ks * pow(max(0.0, dot(n,h)), shininess);
  return vec2(diffuse, spec);
}

void main()
{
  float angle      = 25.0*time;
  vec3 lightV2     = vec3(sin(angle),-0.5,cos(angle));
  float specularC2 = 0.7;  // For moving light -- make this zero to keep only stationary light

  vec3 ColorS1 = blinnPhongDir(lightV1,0.0   ,0.0,     0.0,     specularC1,specularE1).y*lightCol;
  vec3 ColorS2 = blinnPhongDir(lightV2,0.0   ,0.0,     0.0,     specularC2,specularE2).y*lightCol;
  vec3 ColorAD = blinnPhongDir(lightV1,lightI,ambientC,diffuseC,0.0      ,1.0       ).x*objectCol;
  gl_FragColor = vec4(ColorAD+ColorS1+ColorS2,1.0);

  // Stripe-discard effect -- comment out to keep solid model
  if(sin(50.0*rawX.x)>0.5) discard;
}
```

Type qualifiers

goo.gl/gMWgIO

Const variables are compile-time constants (they can appear both in v/s and f/s)

(fragment shader)

```
precision highp float;
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;
varying vec3 rawX;

const vec3  lightV1    = vec3(0.0,1.0,0.0); // stationary light
const float lightI     = 1.0;               // only for diffuse component
const float ambientC   = 0.15;
const float diffuseC   = 0.7;
const float specularC1 = 1.0;               // For stationary light
const float specularE1 = 64.0;
const float specularE2 = 16.0;
const vec3  lightCol   = vec3(1.0,1.0,1.0);
const vec3  objectCol  = vec3(1.0,0.6,0.0); // yellow-ish orange

vec2 blinnPhongDir(vec3 lightDir, float lightInt, float Ka, float Kd, float Ks, float shininess)
{
  vec3 s = normalize(lightDir);
  vec3 v = normalize(-fPosition);
  vec3 n = normalize(fNormal);
  vec3 h = normalize(v+s);
  float diffuse = Ka + Kd * lightInt * max(0.0, dot(n, s));
  float spec =  Ks * pow(max(0.0, dot(n,h)), shininess);
  return vec2(diffuse, spec);
}

void main()
{
  float angle       = 25.0*time;
  vec3 lightV2      = vec3(sin(angle),-0.5,cos(angle));
  float specularC2 = 0.7;  // For moving light -- make this zero to keep only stationary light

  vec3 ColorS1 = blinnPhongDir(lightV1,0.0   ,0.0,     0.0,     specularC1,specularE1).y*lightCol;
  vec3 ColorS2 = blinnPhongDir(lightV2,0.0   ,0.0,     0.0,     specularC2,specularE2).y*lightCol;
  vec3 ColorAD = blinnPhongDir(lightV1,lightI,ambientC,diffuseC,0.0      ,1.0       ).x*objectCol;
  gl_FragColor = vec4(ColorAD+ColorS1+ColorS2,1.0);

  // Stripe-discard effect -- comment out to keep solid model
  if(sin(50.0*rawX.x)>0.5) discard;
}
```

Type qualifiers

goo.gl/gMWgIO

Attributes are vertex properties that are supplied by the host program, and made available to the vertex shader.

The host application determines them; shdr.bkcore.com provides a few (position, normal)

(vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

uniform float time;

const float pi=3.14159;
varying vec3 modelX;
varying vec3 modelN;
varying vec3 rawX;

vec2 Rotate2D(vec2 vec_in, float angle)
{
  vec2 vec_out;
  vec_out.x=cos(angle)*vec_in.x-sin(angle)*vec_in.y;
  vec_out.y=sin(angle)*vec_in.x+cos(angle)*vec_in.y;
  return vec_out;
}

void main()
{
  modelX=position;
  rawX=position;
  modelN=normal;

  // Comment these lines out to stop twisting
  modelX.xz = Rotate2D(modelX.xz,0.5*pi*modelX.y*sin(10.0*time));
  modelN.xz = Rotate2D(modelN.xz,0.5*pi*modelX.y*sin(10.0*time));
  fNormal = normalize(normalMatrix * modelN);
  vec4 pos = modelViewMatrix * vec4(modelX, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

Type qualifiers

goo.gl/gMWgIO

Uniform's are variables that have the same (constant/uniform) value for the entire scene (contrast with attributes). They are available to both v/s and f/s.
(the host program sets those)

shdr.bkcore.com provides a few …

(be mindful of their semantics! Check the video of the lecture!)

(vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

uniform float time;

const float pi=3.14159;
varying vec3 modelX;
varying vec3 modelN;
varying vec3 rawX;

vec2 Rotate2D(vec2 vec_in, float angle)
{
  vec2 vec_out;
  vec_out.x=cos(angle)*vec_in.x-sin(angle)*vec_in.y;
  vec_out.y=sin(angle)*vec_in.x+cos(angle)*vec_in.y;
  return vec_out;
}

void main()
{
  modelX=position;
  rawX=position;
  modelN=normal;

  // Comment these lines out to stop twisting
  modelX.xz = Rotate2D(modelX.xz,0.5*pi*modelX.y*sin(10.0*time));
  modelN.xz = Rotate2D(modelN.xz,0.5*pi*modelX.y*sin(10.0*time));
  fNormal = normalize(normalMatrix * modelN);
  vec4 pos = modelViewMatrix * vec4(modelX, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

GLSL Basics

goo.gl/gMWgIO

Varying's are the only way we communicate information from the vertex shader to the fragment shader!

They are *invisible* to the host program (just used within shaders)

Assigned on a per-vertex basis by the vertex shader, interpolated on a per-fragment basis by the fragment shader)

Interpolation using barycentric weights (from rasterization)

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

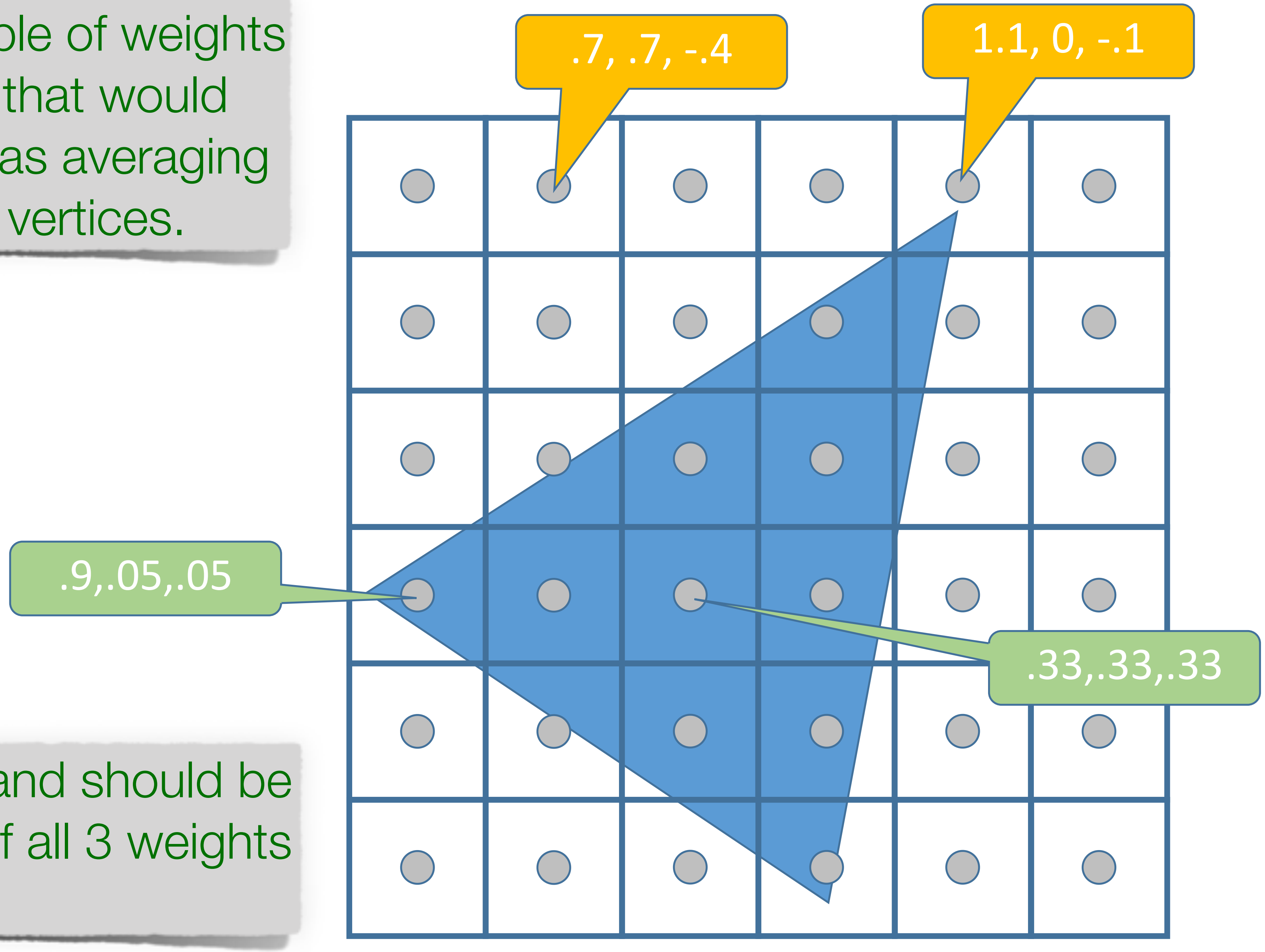
(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

Rasterization (in hardware)

For every pixel, compute a triple of weights ("barycentric coordinates") that would reconstruct the point if used as averaging weights from the triangle vertices.



A pixel is *inside the triangle* (and should be "rasterized" into a fragment) if all 3 weights are positive.

GLSL Basics

goo.gl/gMWgIO

Varying's are the only way we communicate information from the vertex shader to the fragment shader!

They are *invisible* to the host program (just used within shaders)

Assigned on a per-vertex basis by the vertex shader, interpolated on a per-fragment basis by the fragment shader)

Interpolation using barycentric weights (from rasterization)

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

GLSL Basics

goo.gl/gMWgIO

Responsibilities of the vertex shader:

- Prepare any <u>varying</u> variables that the fragment shader might need.

- Set the special variable **gl_Position** to the Normalized Device Coordinates of the vertex being processed.

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

GLSL Basics

goo.gl/gMWglO

… or *discard* drawing the fragment altogether

(sample vertex shader)

```
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  fNormal = normalize(normalMatrix * normal);
  vec4 pos = modelViewMatrix * vec4(position, 1.0);
  fPosition = pos.xyz;
  gl_Position = projectionMatrix * pos;
}
```

(sample fragment shader)

```
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
  gl_FragColor = vec4(fNormal, 1.0);
}
```

goo.gl/gMWgIO

… or *discard* drawing the fragment altogether

(fragment shader)

```
precision highp float;
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;
varying vec3 rawX;

const vec3  lightV1    = vec3(0.0,1.0,0.0); // stationary light
const float lightI     = 1.0;               // only for diffuse component
const float ambientC   = 0.15;
const float diffuseC   = 0.7;
const float specularC1 = 1.0;               // For stationary light
const float specularE1 = 64.0;
const float specularE2 = 16.0;
const vec3  lightCol   = vec3(1.0,1.0,1.0);
const vec3  objectCol  = vec3(1.0,0.6,0.0); // yellow-ish orange

vec2 blinnPhongDir(vec3 lightDir, float lightInt, float Ka, float Kd, float Ks, float shininess)
{
  vec3 s = normalize(lightDir);
  vec3 v = normalize(-fPosition);
  vec3 n = normalize(fNormal);
  vec3 h = normalize(v+s);
  float diffuse = Ka + Kd * lightInt * max(0.0, dot(n, s));
  float spec =  Ks * pow(max(0.0, dot(n,h)), shininess);
  return vec2(diffuse, spec);
}

void main()
{
  float angle      = 25.0*time;
  vec3 lightV2     = vec3(sin(angle),-0.5,cos(angle));
  float specularC2 = 0.7;  // For moving light -- make this zero to keep only stationary light

  vec3 ColorS1 = blinnPhongDir(lightV1,0.0   ,0.0,     0.0,     specularC1,specularE1).y*lightCol;
  vec3 ColorS2 = blinnPhongDir(lightV2,0.0   ,0.0,     0.0,     specularC2,specularE2).y*lightCol;
  vec3 ColorAD = blinnPhongDir(lightV1,lightI,ambientC,diffuseC,0.0      ,1.0       ).x*objectCol;
  gl_FragColor = vec4(ColorAD+ColorS1+ColorS2,1.0);

  // Stripe-discard effect -- comment out to keep solid model
  if(sin(50.0*rawX.x)>0.5) discard;
}
```

Odds and ends

- Control structures

```
uniform vec2 resolution;
void main()
{
  vec3 color;
  // gl_FragCoord is in pixels - so convert...
  float ndcx = (gl_FragCoord.x / resolution.x) - 1.0;
  if (ndcx > 0.0) {
    color = vec3(1,1,0);
  } else {
   color = vec3(1,0,1);
  }
  gl_FragColor = vec4(color, 1.0);
}
```

- "Step" interpolation

```
color = mix(vec3(1,1,0), vec3(1,0,1),
step(0.0, ncdx) );

color = mix(vec3(1,1,0), vec3(1,0,1),
smoothstep(-0.1, 0.1, ncdx) );
```

Odds and ends

goo.gl/gMWglO

User-defined functions

(fragment shader)

```
precision highp float;
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;
varying vec3 rawX;

const vec3  lightV1    = vec3(0.0,1.0,0.0); // stationary light
const float lightI     = 1.0;               // only for diffuse component
const float ambientC   = 0.15;
const float diffuseC   = 0.7;
const float specularC1 = 1.0;               // For stationary light
const float specularE1 = 64.0;
const float specularE2 = 16.0;
const vec3  lightCol   = vec3(1.0,1.0,1.0);
const vec3  objectCol  = vec3(1.0,0.6,0.0); // yellow-ish orange

vec2 blinnPhongDir(vec3 lightDir, float lightInt, float Ka, float Kd, float Ks, float shininess)
{
  vec3 s = normalize(lightDir);
  vec3 v = normalize(-fPosition);
  vec3 n = normalize(fNormal);
  vec3 h = normalize(v+s);
  float diffuse = Ka + Kd * lightInt * max(0.0, dot(n, s));
  float spec =  Ks * pow(max(0.0, dot(n,h)), shininess);
  return vec2(diffuse, spec);
}

void main()
{
  float angle      = 25.0*time;
  vec3 lightV2     = vec3(sin(angle),-0.5,cos(angle));
  float specularC2 = 0.7;  // For moving light -- make this zero to keep only stationary light

  vec3 ColorS1 = blinnPhongDir(lightV1,0.0   ,0.0,     0.0,     specularC1,specularE1).y*lightCol;
  vec3 ColorS2 = blinnPhongDir(lightV2,0.0   ,0.0,     0.0,     specularC2,specularE2).y*lightCol;
  vec3 ColorAD = blinnPhongDir(lightV1,lightI,ambientC,diffuseC,0.0      ,1.0       ).x*objectCol;
  gl_FragColor = vec4(ColorAD+ColorS1+ColorS2,1.0);

  // Stripe-discard effect -- comment out to keep solid model
  if(sin(50.0*rawX.x)>0.5) discard;
}
```

Examples

- Diffuse — *goo.gl/A81r7a*
- Ambient/Diffuse/Specular — *goo.gl/ooE6NL*
- A more complex one — *goo.gl/gMWgIO*