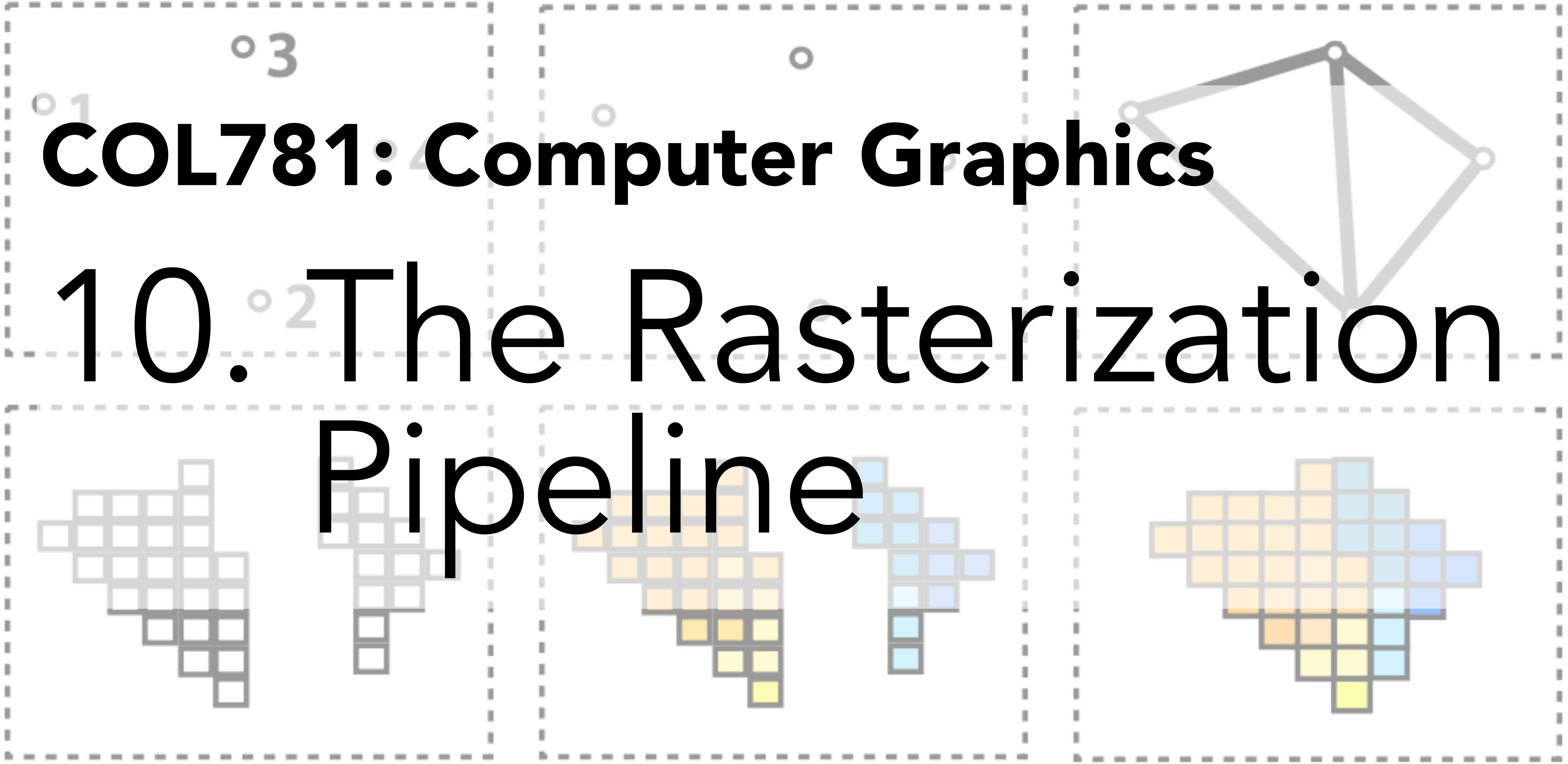
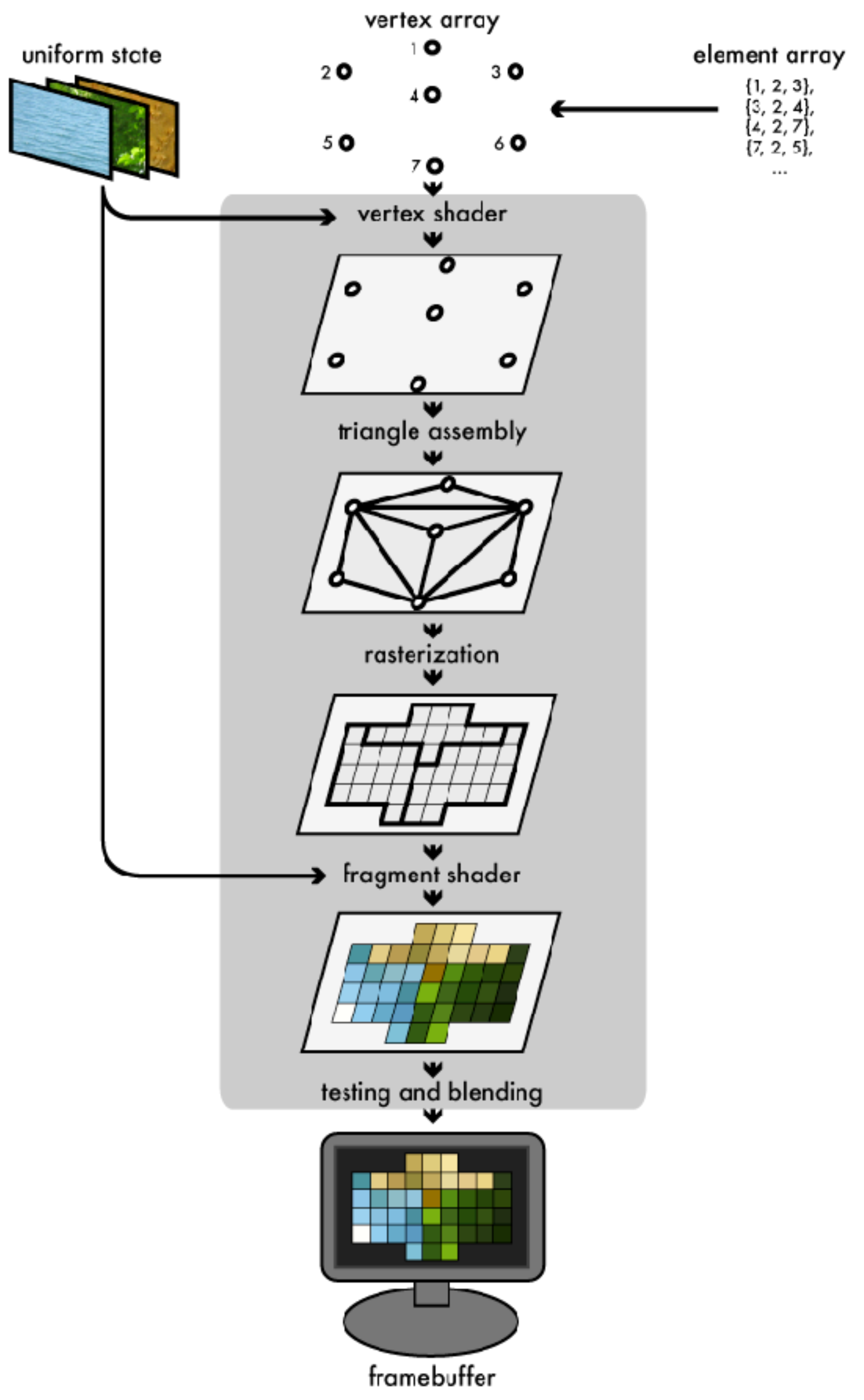


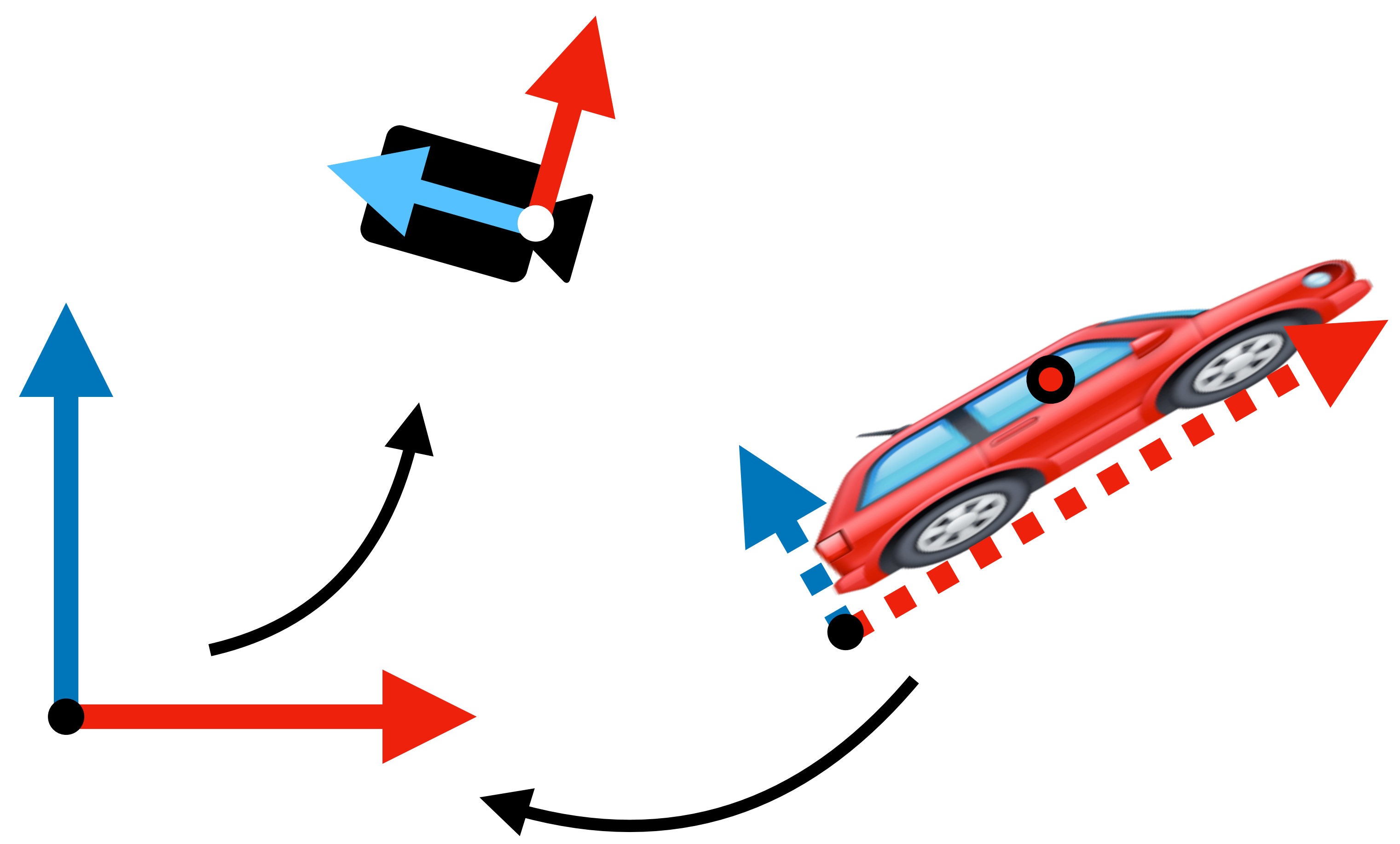
COL781: Computer Graphics

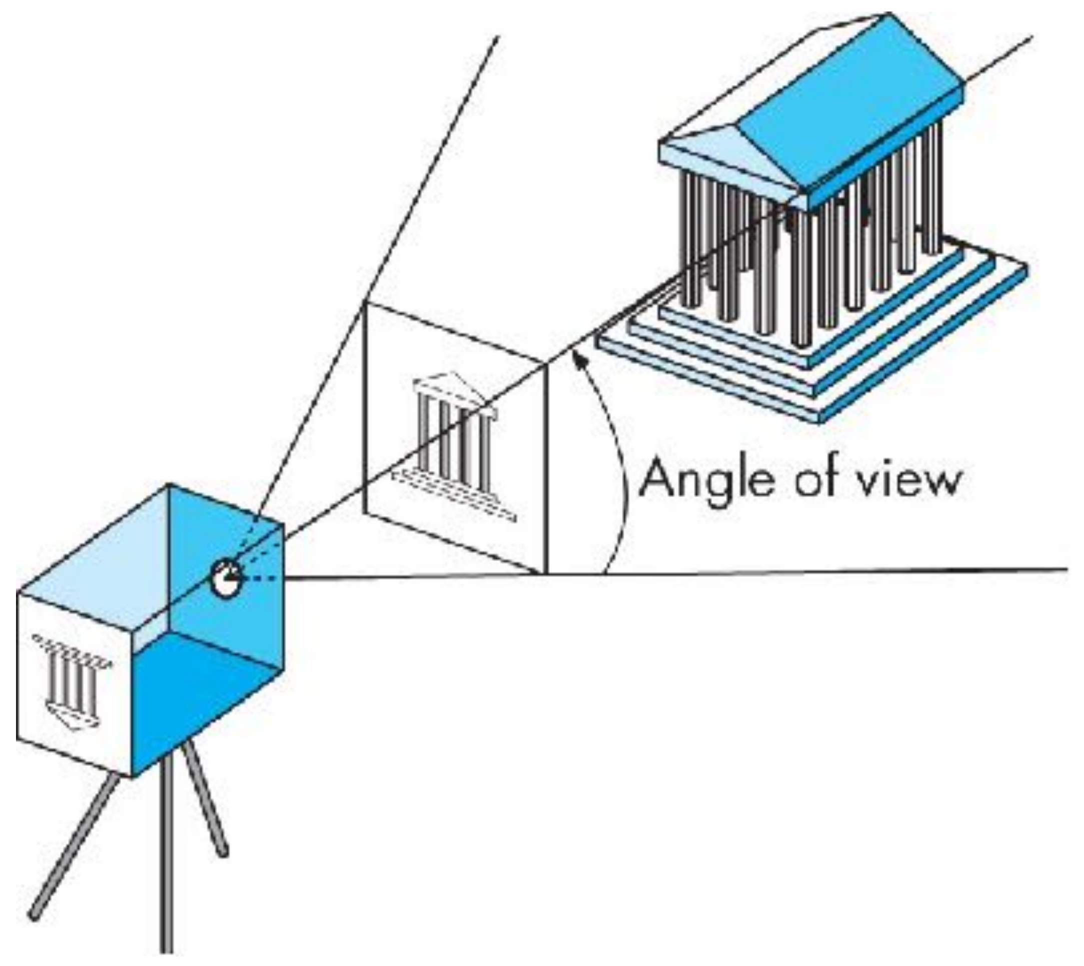
10. The Rasterization Pipeline

Today: Putting it all together

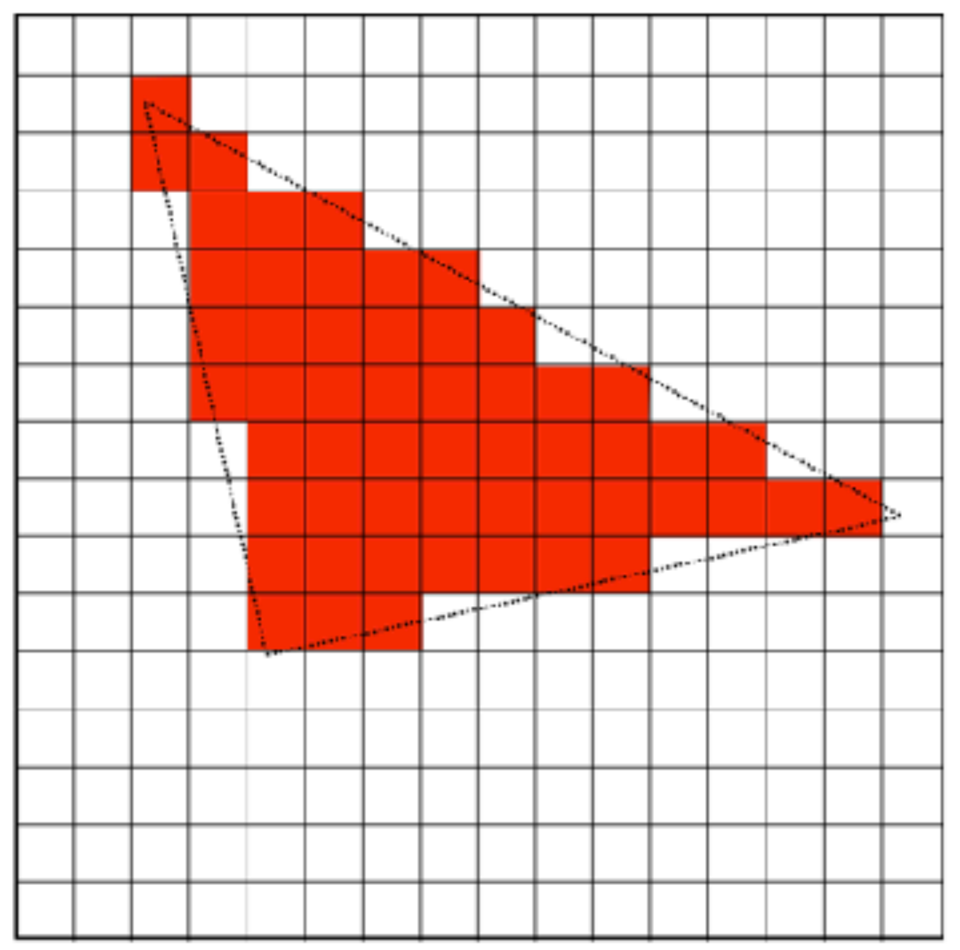


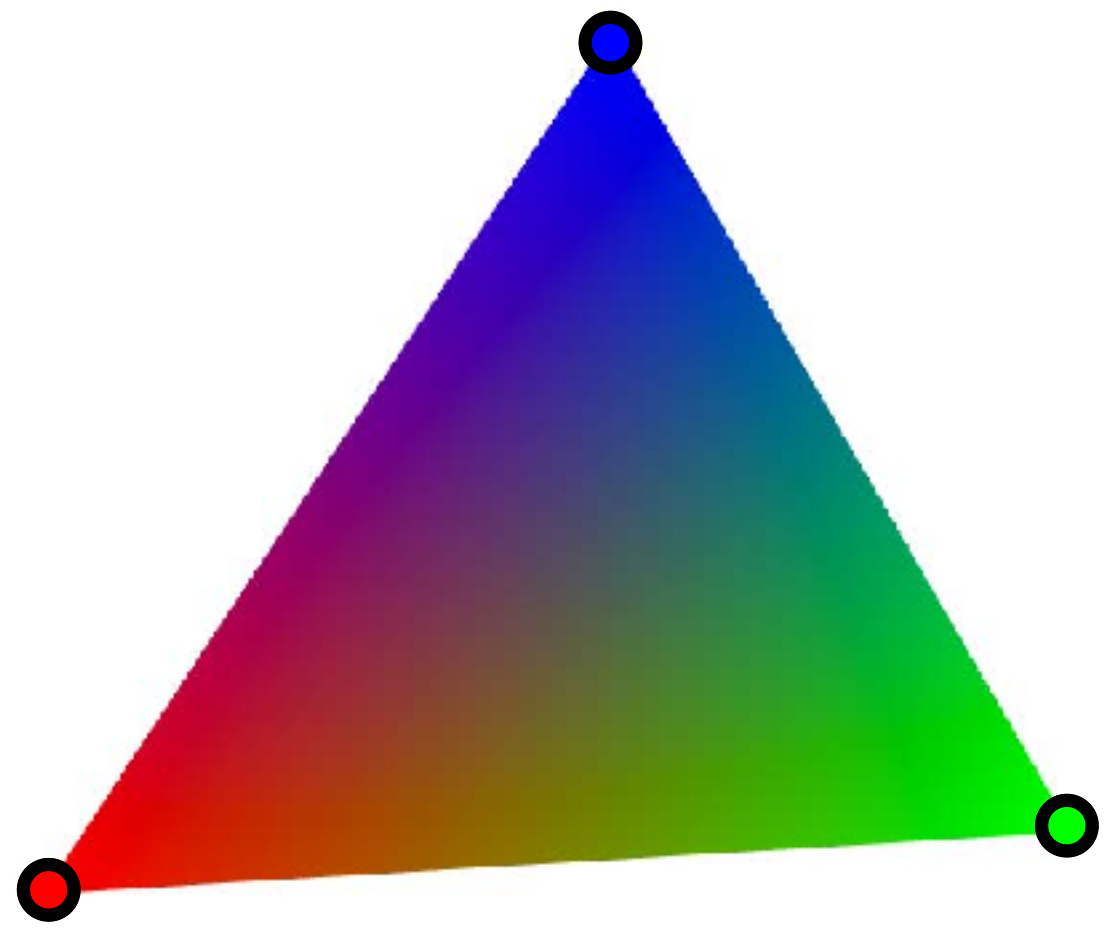
Joe Groff, duriansoftware.com


Transformations



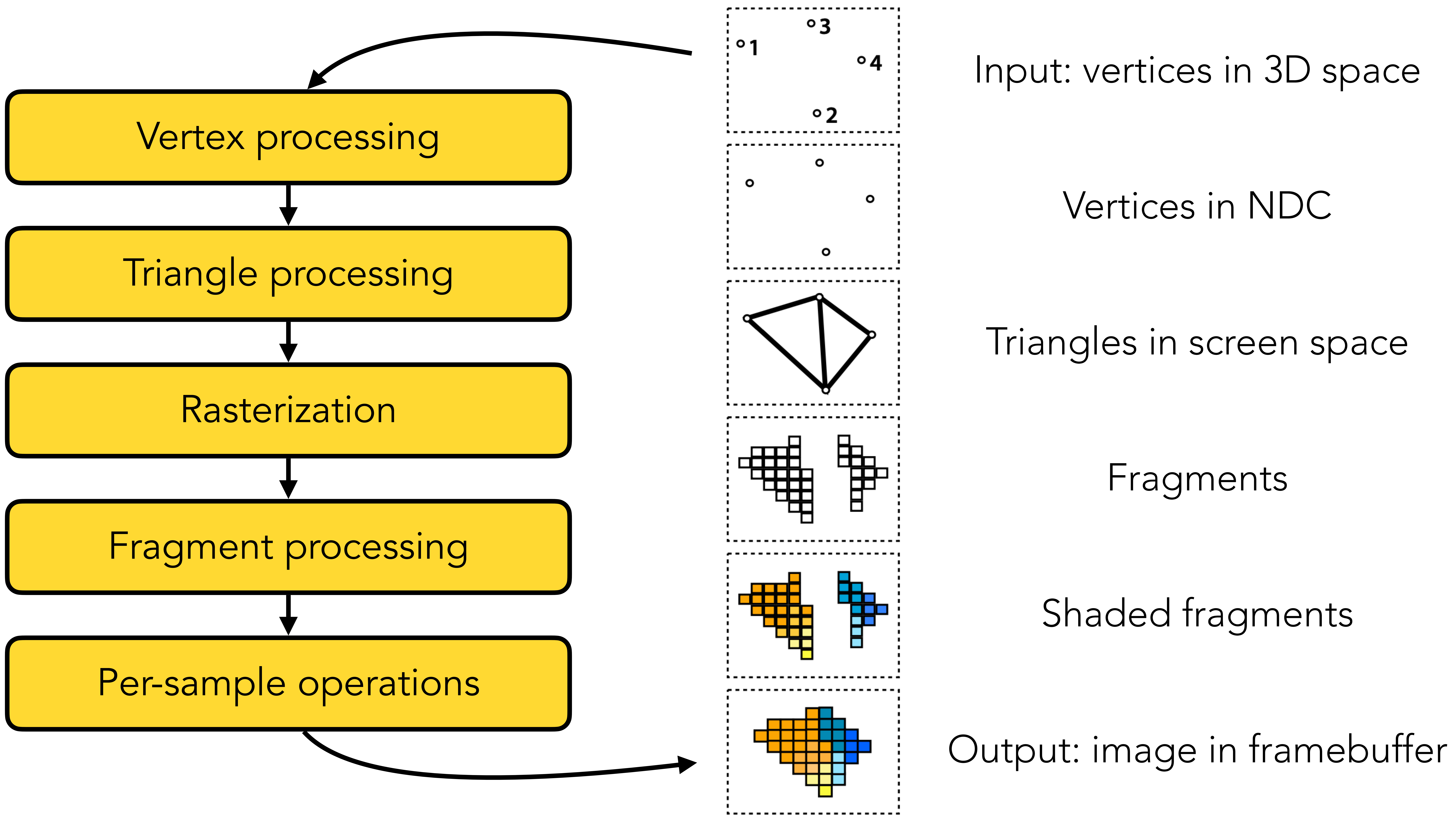
Projection


Rasterization

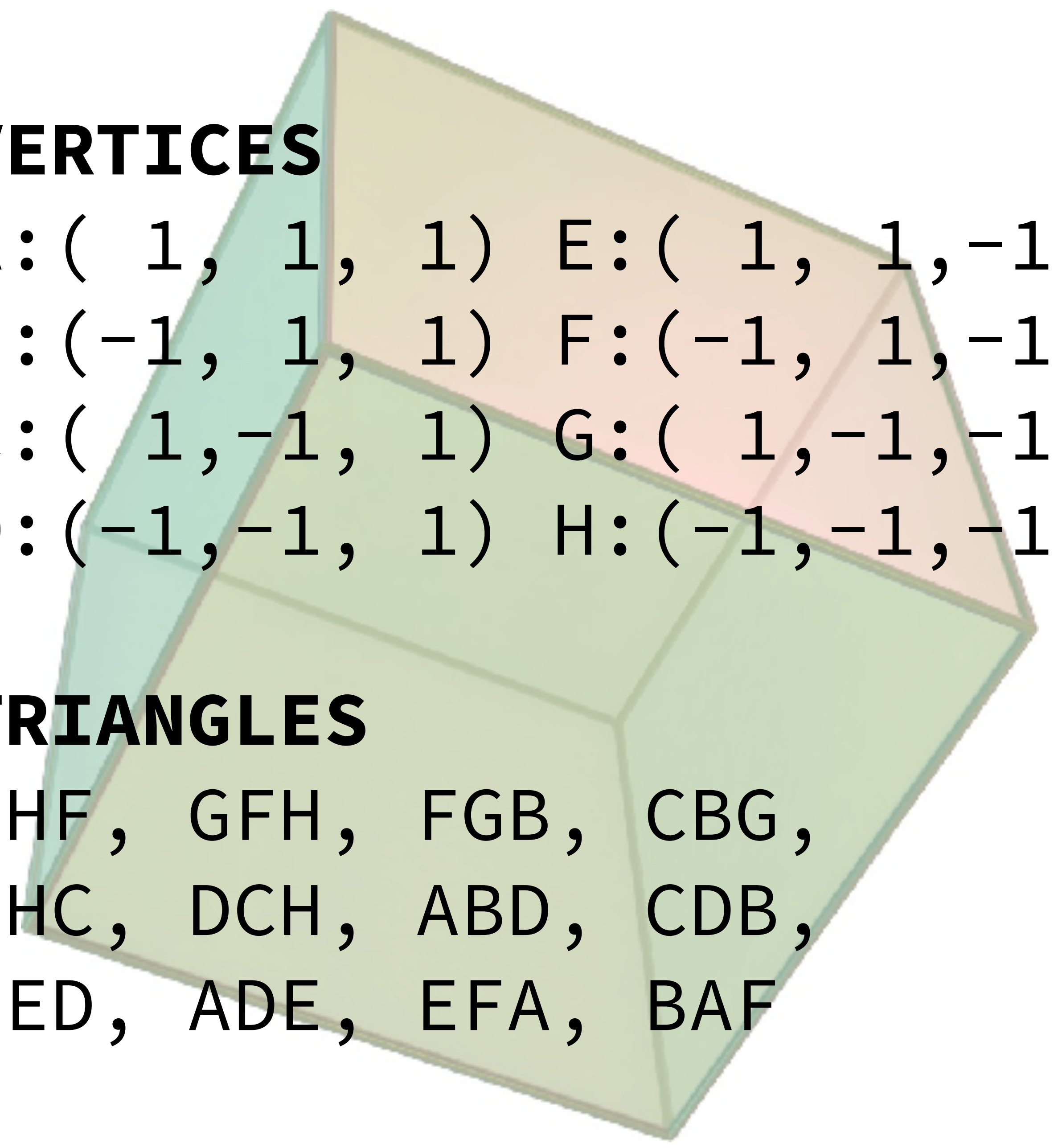

Interpolation

Texture mapping

Visibility


Vertex processing
Triangle processing
Rasterization
Fragment processing
Per-sample operations
1
2
3
4
Input: vertices in 3D space
Vertices in NDC
Triangles in screen space
Fragments
Shaded fragments
Output: image in framebuffer

Inputs to the pipeline



```
VERTICES
A:( 1, 1, 1) E:( 1, 1,-1)
B:(-1, 1, 1) F:(-1, 1,-1)
C:( 1,-1, 1) G:( 1,-1,-1)
D:(-1,-1, 1) H:(-1,-1,-1)

TRIANGLES
EHF, GFH, FGB, CBG,
GHC, DCH, ABD, CDB,
HED, ADE, EFA, BAF
```

For each object, we have two streams:

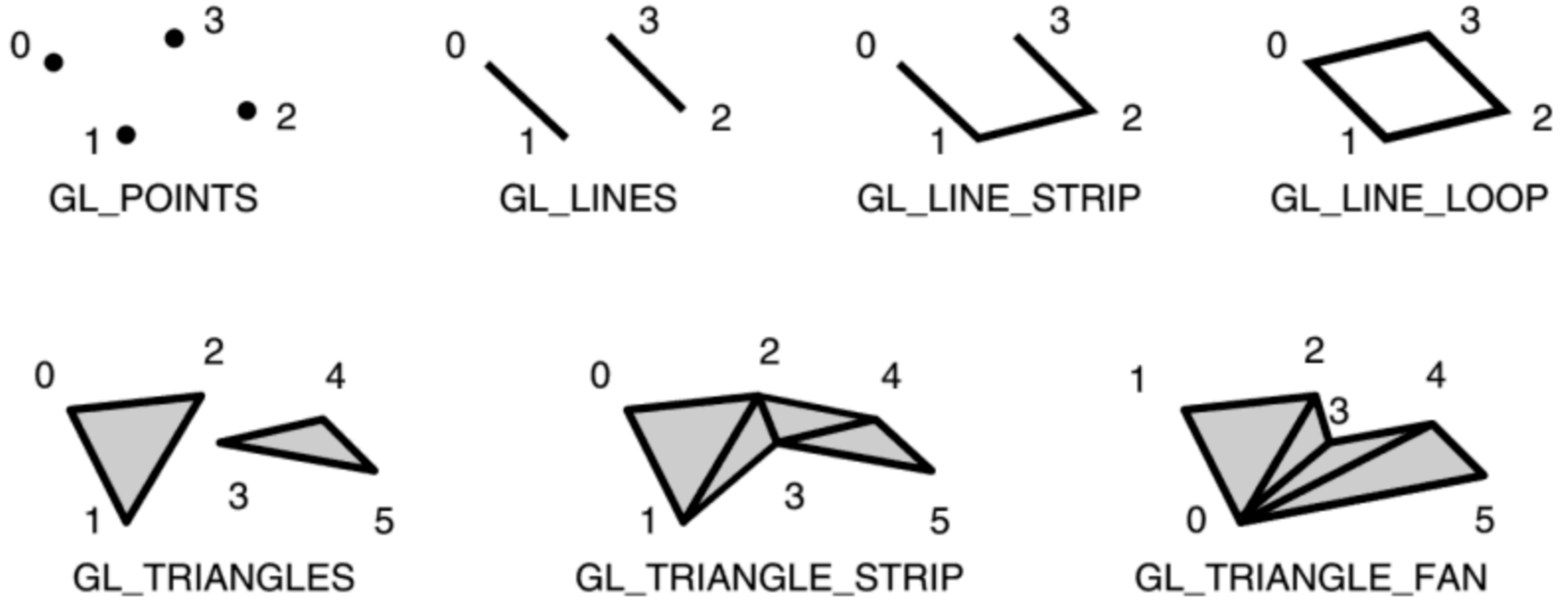
- Vertices with various **attributes** (position, colour, texture coordinates, etc.)
- Indices of triangles (or other primitives)

Why? Each vertex is shared between many primitives (on average ~6 triangles!)

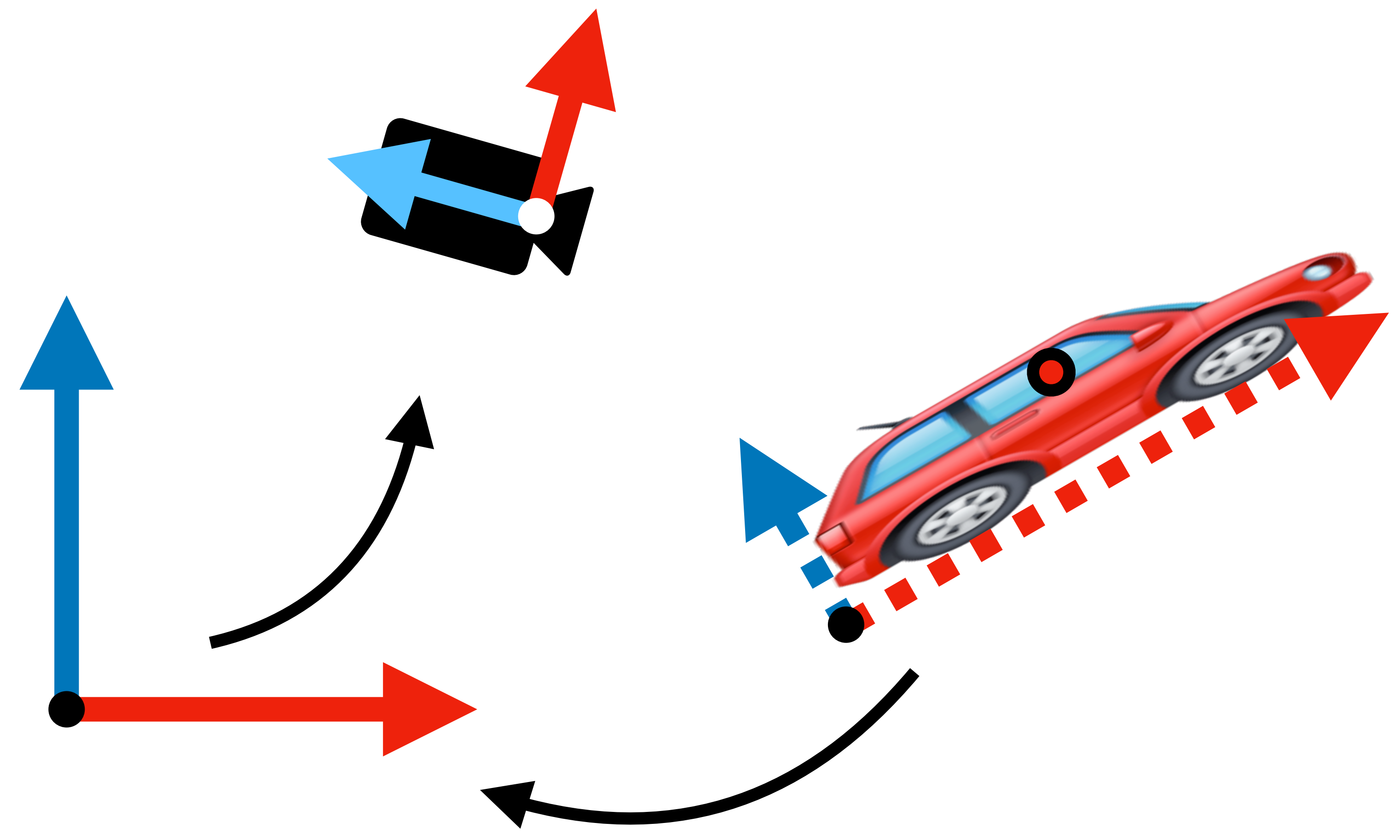
We also have **uniform** data, common to all vertices/triangles of an object:

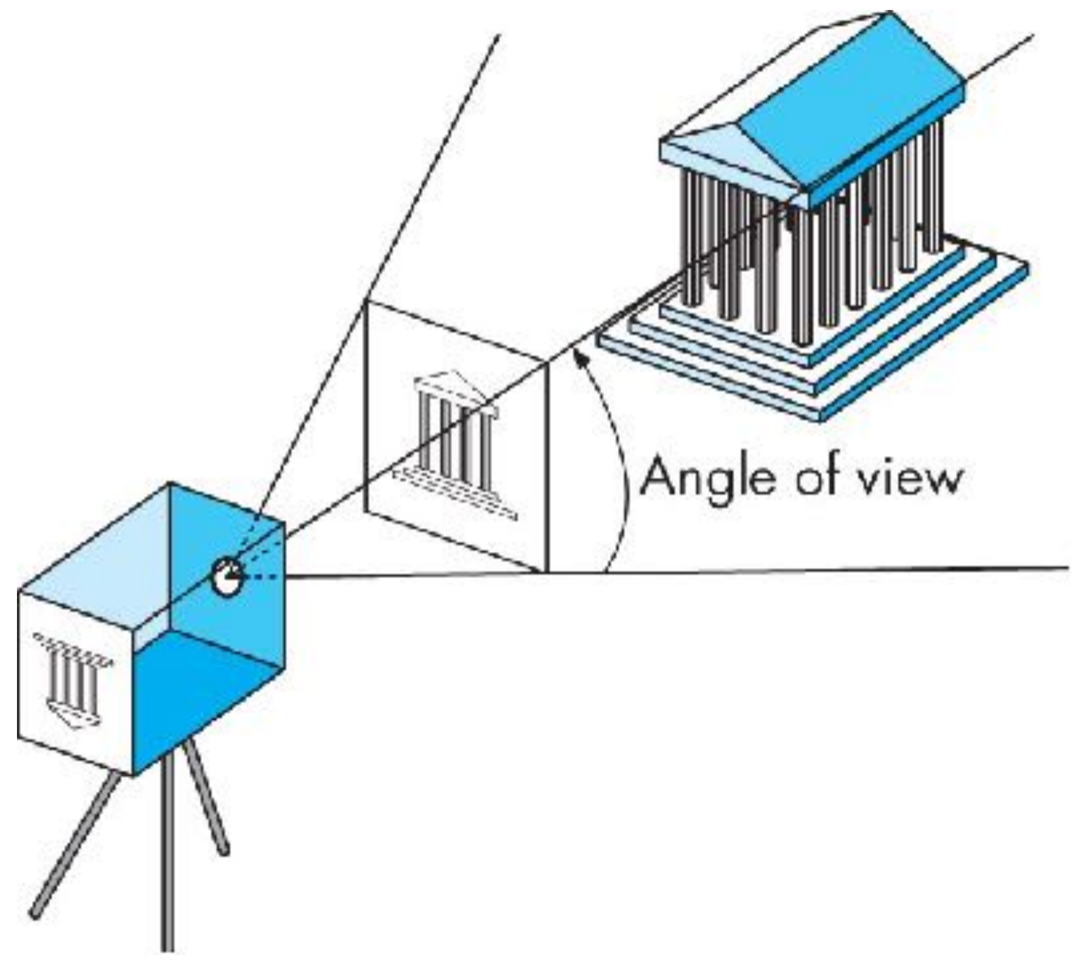
- Transformation matrices, texture images, etc.

Primitives


0
1
2
3
GL_POINTS
0
1
2
3
GL_LINES
0
1
2
3
GL_LINE_STRIP
0
1
2
3
GL_LINE_LOOP
0
1
2
3
4
5
GL_TRIANGLES
0
1
2
3
4
5
GL_TRIANGLE_STRIP
0
1
2
3
4
5
GL_TRIANGLE_FAN

Vertex processing





Every vertex is subject to the same operations:

- Modelling transformation: object space → world space
- Viewing transformation: world space → camera space
- Projection transformation: camera space → normalized device coordinates

This stage is programmable, done by programmer-specified **vertex shader**

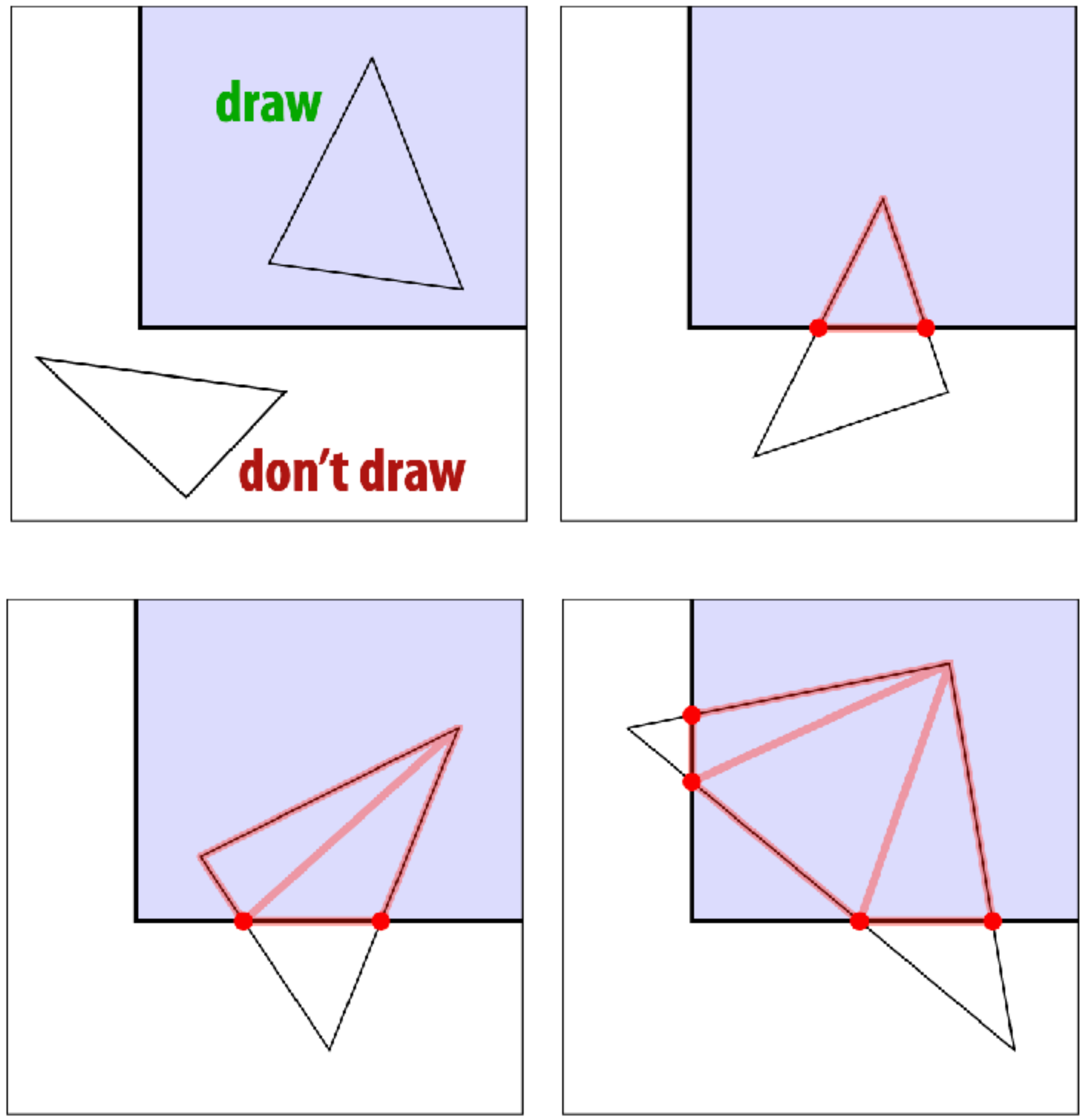
Output: transformed position in NDC (before division)

Triangle processing

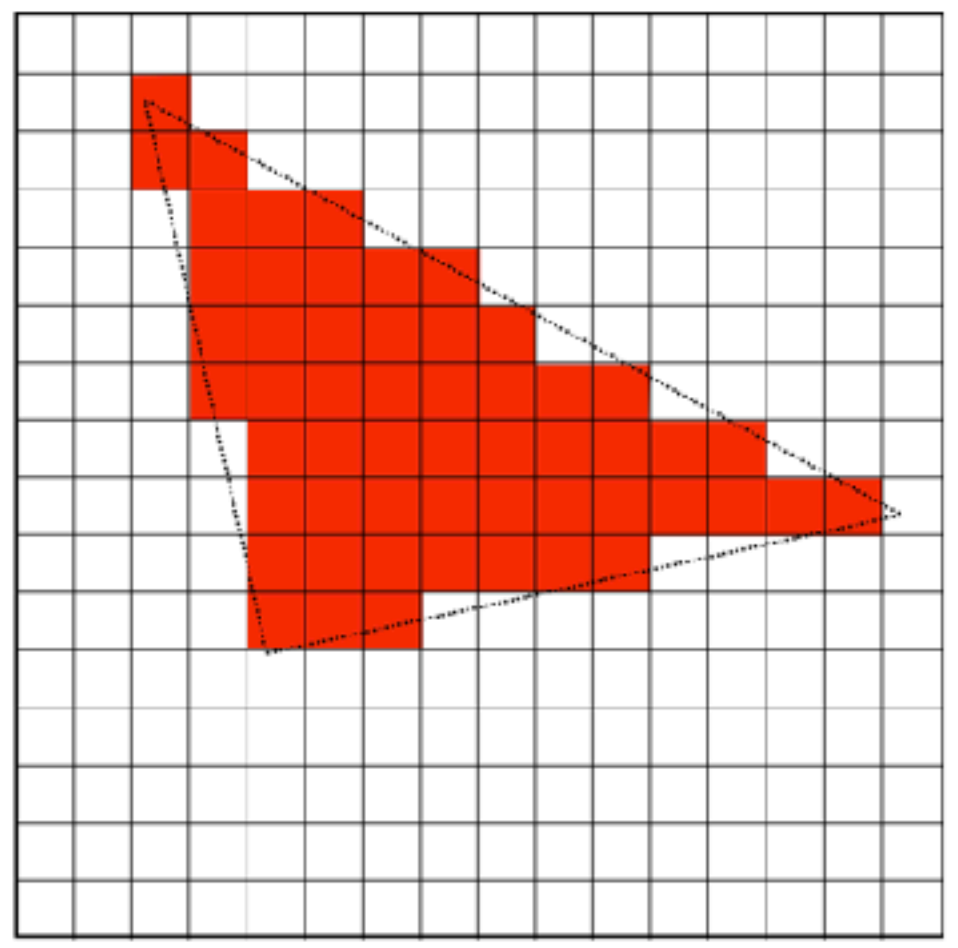
For each triangle (*i*, *j*, *k*):

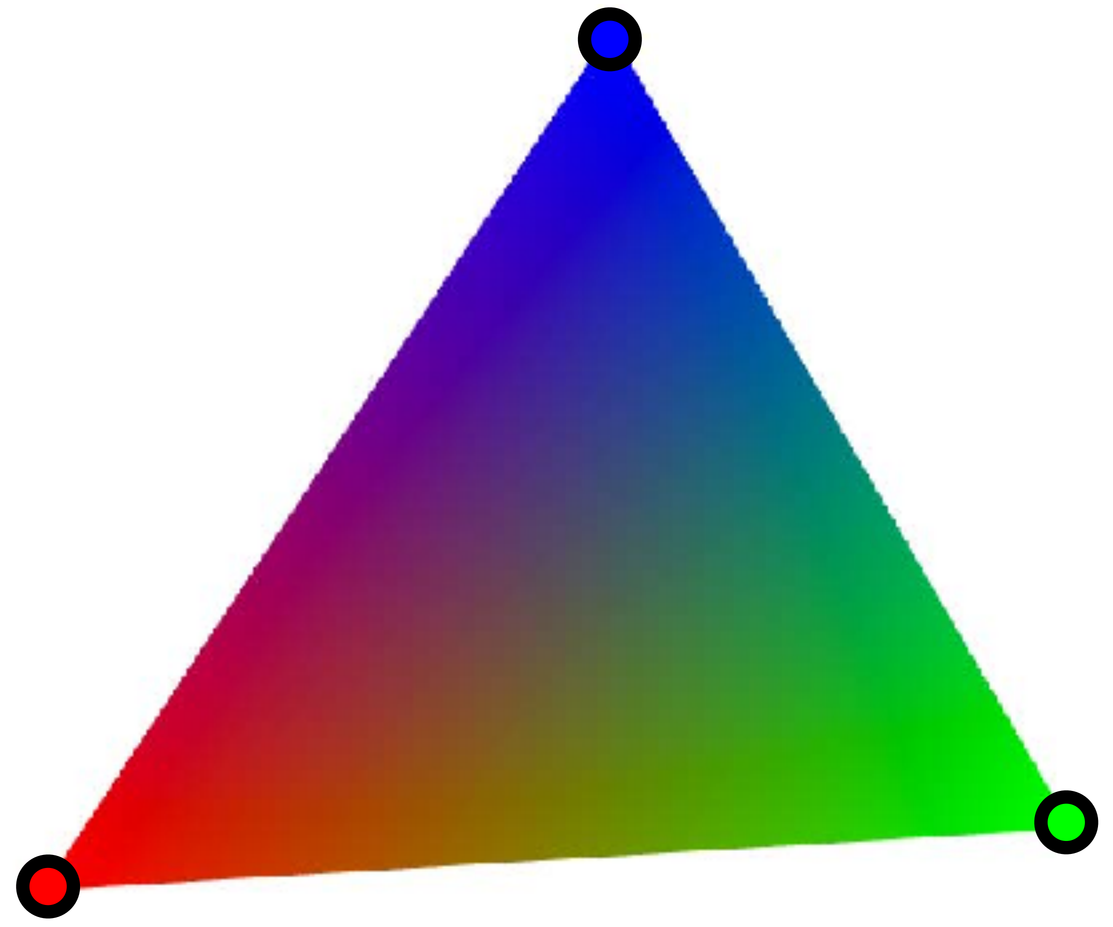
- Get transformed positions $\mathbf{p}_i$, $\mathbf{p}_j$, $\mathbf{p}_k$ of corresponding vertices
- Clip against the canonical view volume $[-1, 1]^3$
- Divide by *w* and transform to pixel coordinates

Output: clipped triangle(s) in screen space



Rasterization





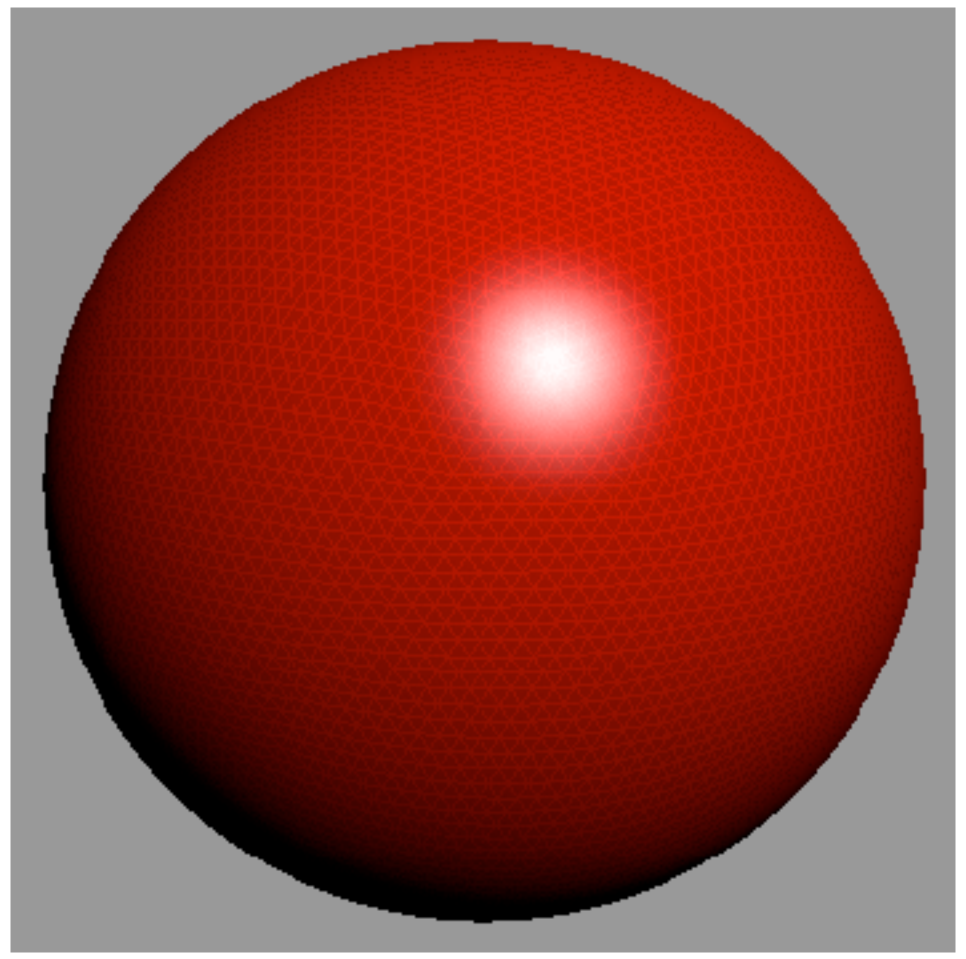
For each triangle, we will produce
a set of sample points that it covers.

But also: interpolate the vertex
attributes (colour, texture coordinates, etc.) to each covered sample.

We will need these in the next stage!

Output: stream of **fragments**, i.e. sample-sized pieces of triangle with interpolated attributes

Fragment processing



We may want to do some computation to decide the colour of a fragment, e.g.

- Texture lookup
- Lighting computation (next class)

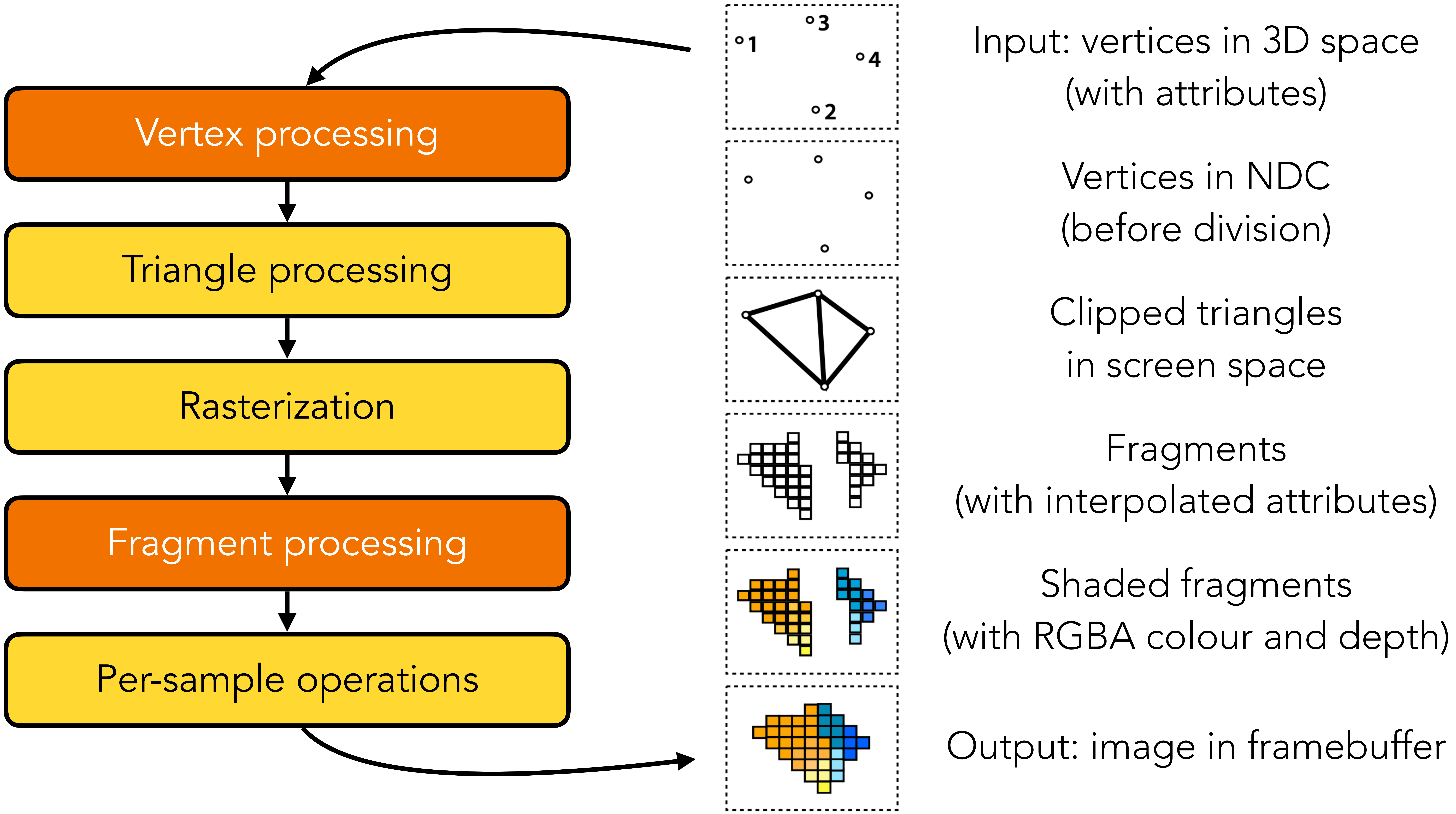
This stage is also programmable: **fragment shader**

Output: fragment colour as a 4-tuple: red, green, blue, alpha (opacity)

Per-sample operations

- Test each sample's depth vs. z-buffer
- Blend with existing colour in framebuffer using alpha

Once all this is done for all objects in the scene, the framebuffer contains the final rendered image.


Vertex processing
Triangle processing
Rasterization
Fragment processing
Per-sample operations
1
3
4
2
Input: vertices in 3D space
(with attributes)
Vertices in NDC
(before division)
Clipped triangles
in screen space
Fragments
(with interpolated attributes)
Shaded fragments
(with RGBA colour and depth)
Output: image in framebuffer

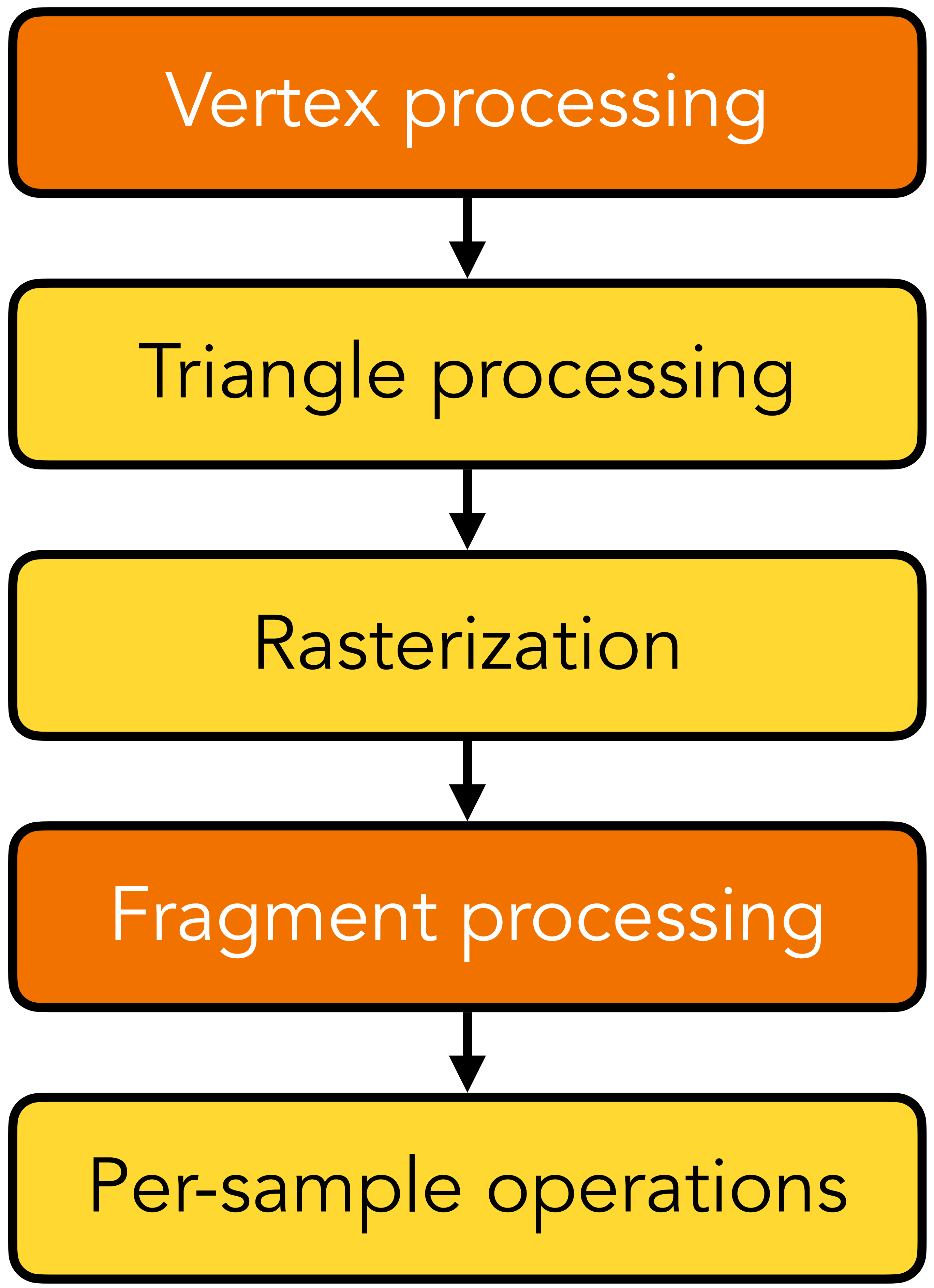
Programmer's view

Initialization:

- Compile vertex and fragment shaders
- Send uniform variables (transformation matrices, texture images, etc.) to GPU
- For each object: send vertex attributes, triangle indices

Per frame, for each object:

- Update uniform variables
- Request draw



Assignment 1

Initialization:

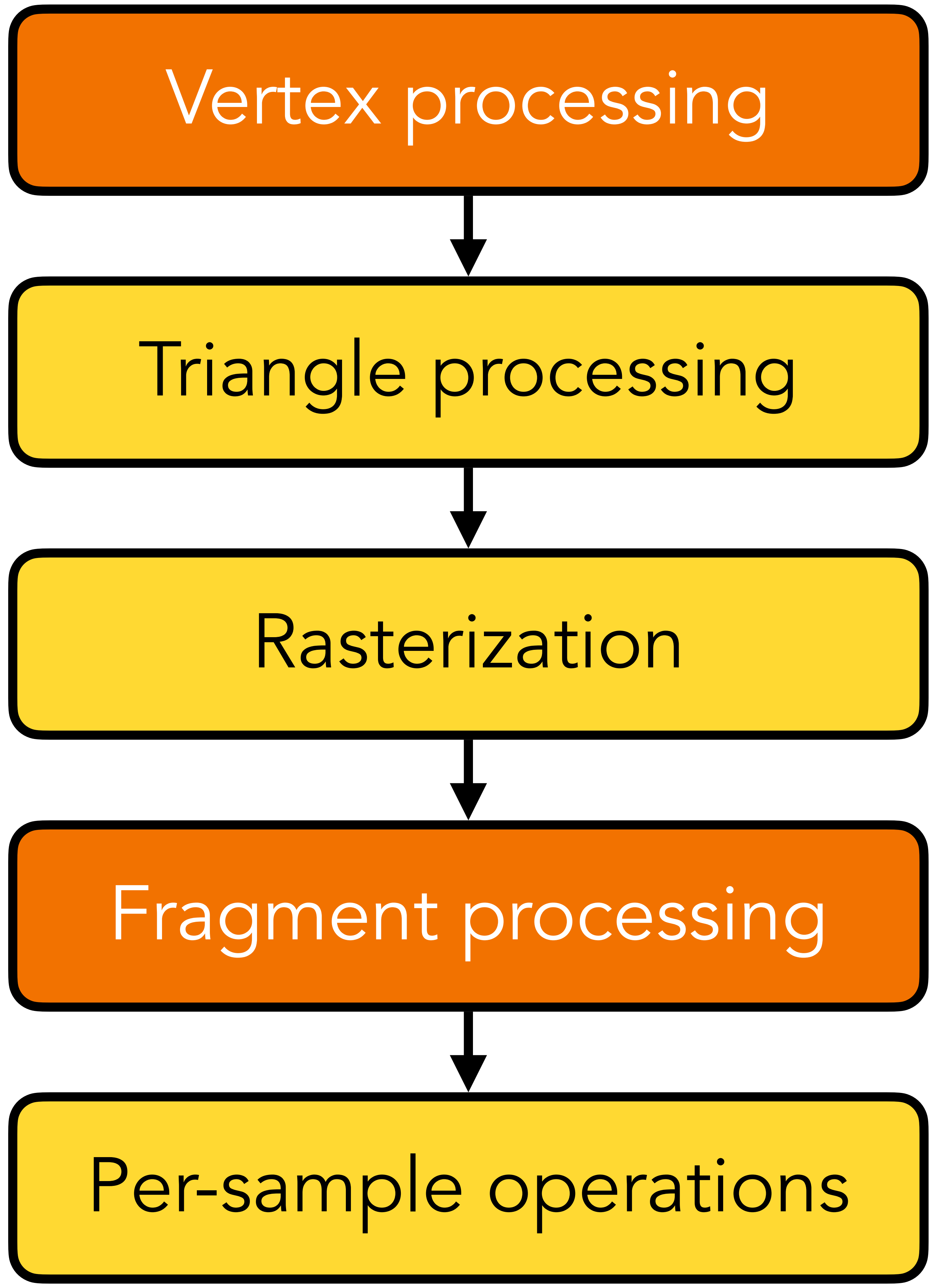
- Compile vertex and fragment shaders
- Send uniform variables (transformation matrices, texture images, etc.) to GPU
- For each object: send vertex attributes, triangle indices

Per frame, for each object:

- Update uniform variables
- Request draw

You will implement most of this…

…not this!
(Well, you will do a little of this too)



Assignment 1

Implement a software rasterization library that plays the role of the GPU, so that the example programs we provide can run

- Call programmer-defined vertex and fragment shaders
- Implement screen-space transformation, rasterization, barycentric interpolation
- Add support for supersampling, z-buffering
- **Optional:** TBA
- **For 1-person groups:** TBA

Shaders

~~The vertex shader applies modelling, viewing, projection transformations to compute the NDC position~~

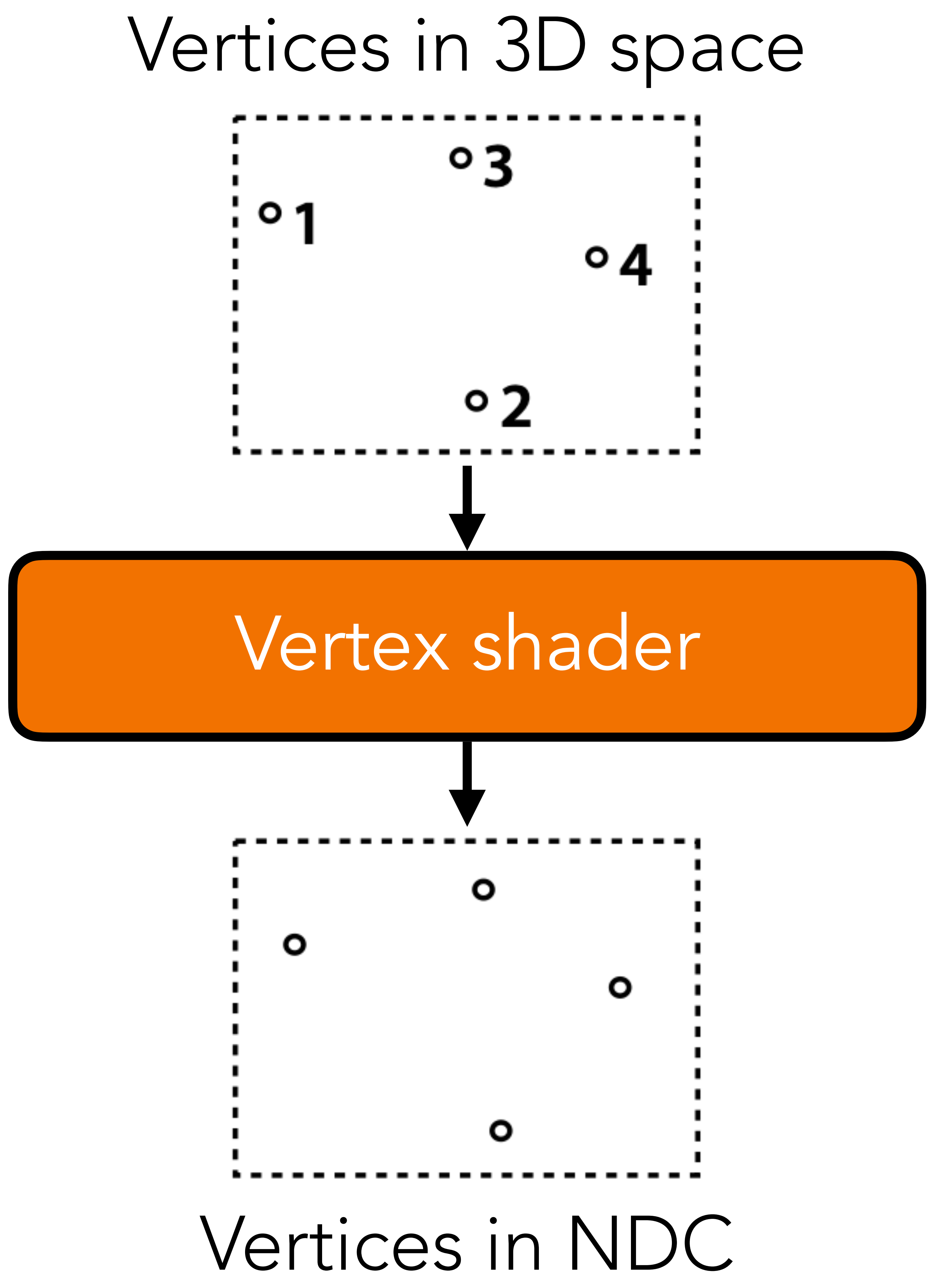
The vertex shader is an arbitrary function that can do whatever you want to compute the NDC position!

Runs on each vertex independently

- Can't pass information to other vertices
- Can't have side-effects (e.g. no writing to global memory, no print statements)

Inputs: attributes of current vertex, uniform variables

Outputs: vertex position in NDC, other attributes to interpolate to fragments



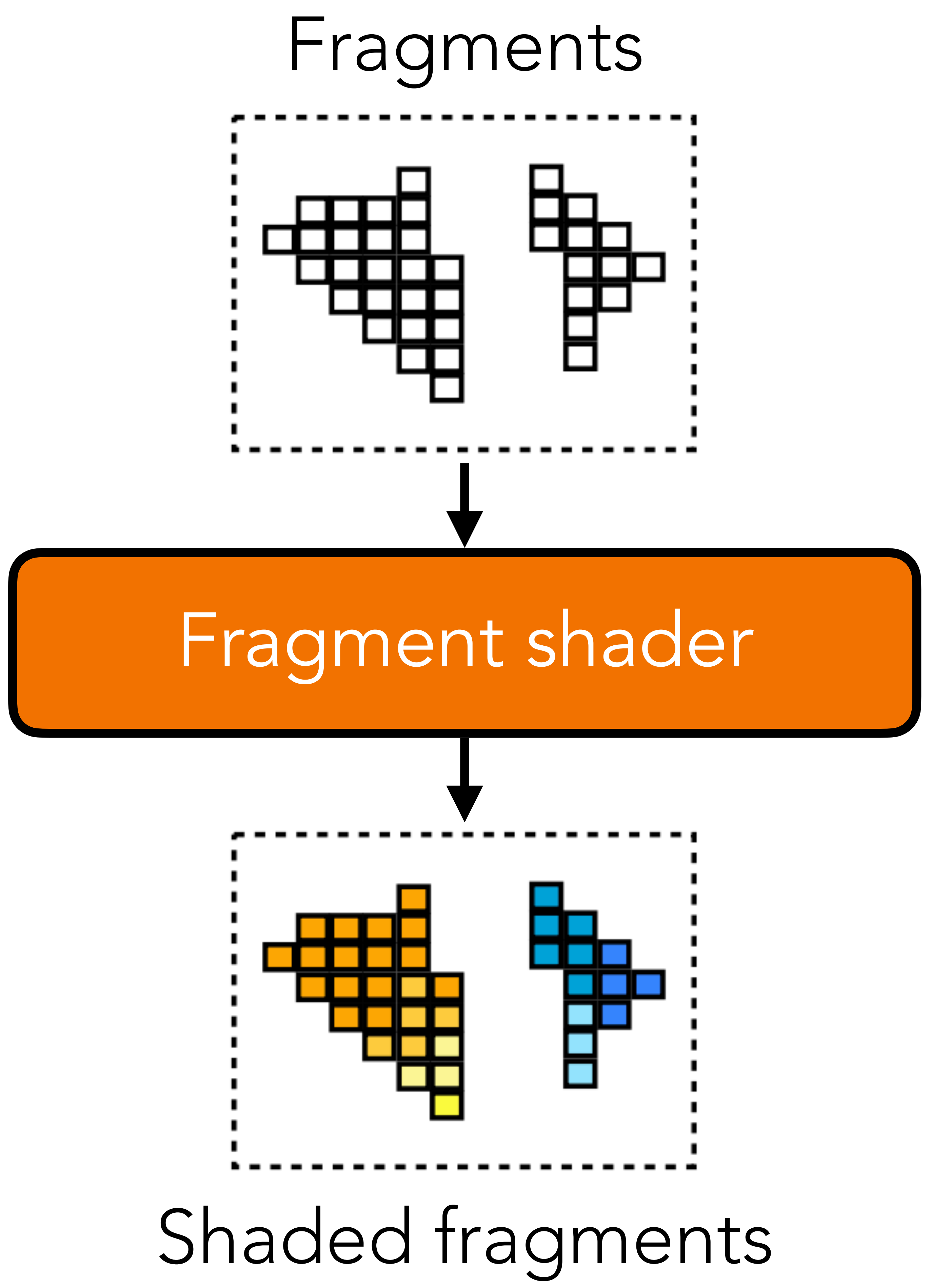
The fragment shader is another arbitrary function.

It can do anything (e.g. texture lookup, lighting computation, etc.) to compute the fragment colour.

Again, runs on each fragment independently

Inputs: attributes interpolated from vertex shader output, uniform variables

Outputs: fragment colour (RGBA), optional: modified fragment depth

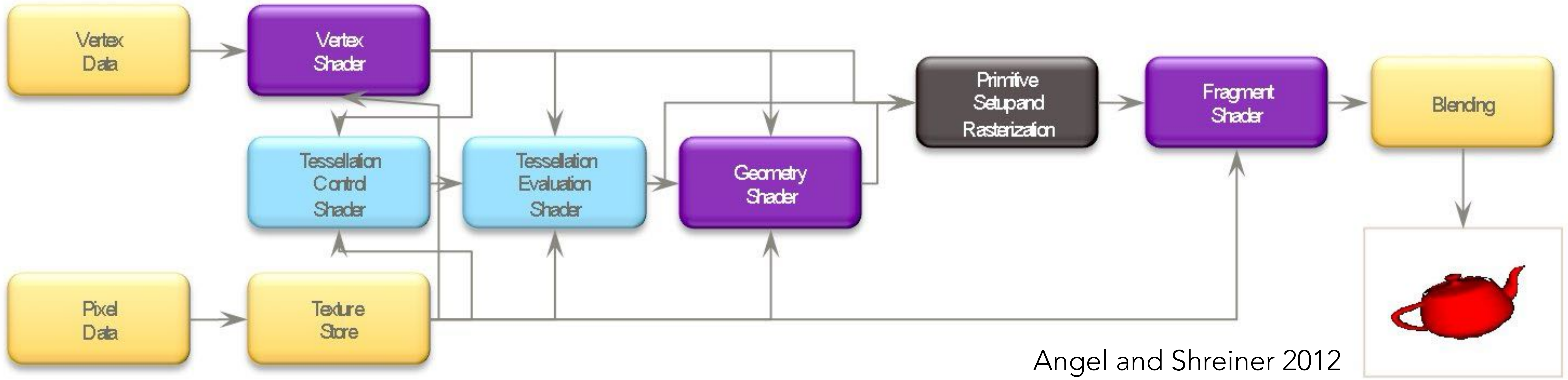


Fragment shader can change fragment depth but not fragment position! (Why?)

Other programmable stages

Modern GPUs have a few more stages we won't cover:

- Tessellation shaders: subdivide primitives into smaller pieces
- Geometry shader: create new geometry from given primitives



Vertex Data
Vertex Shader
Tessellation Control Shader
Tessellation Evaluation Shader
Geometry Shader
Primitive Setup and Rasterization
Fragment Shader
Blending
Pixel Data
Texture Store

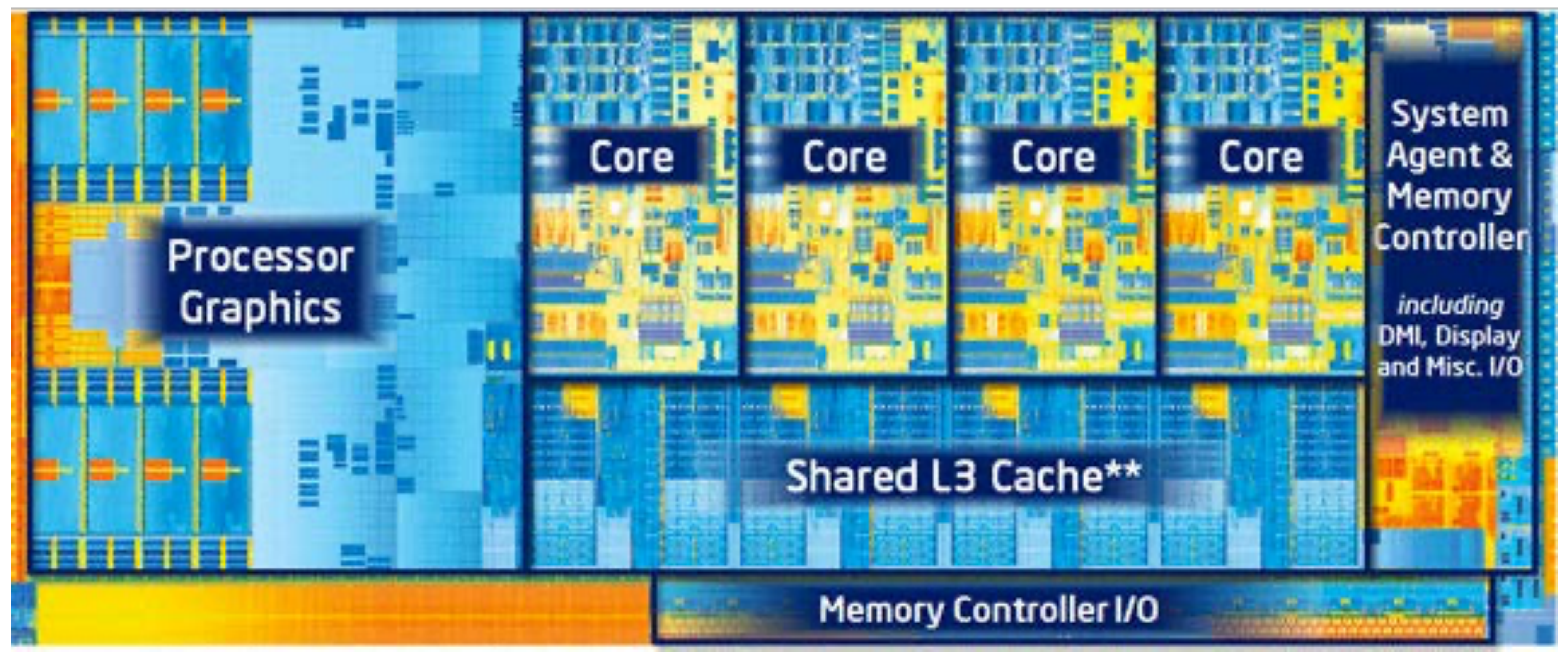
Angel and Shreiner 2012

GPUs

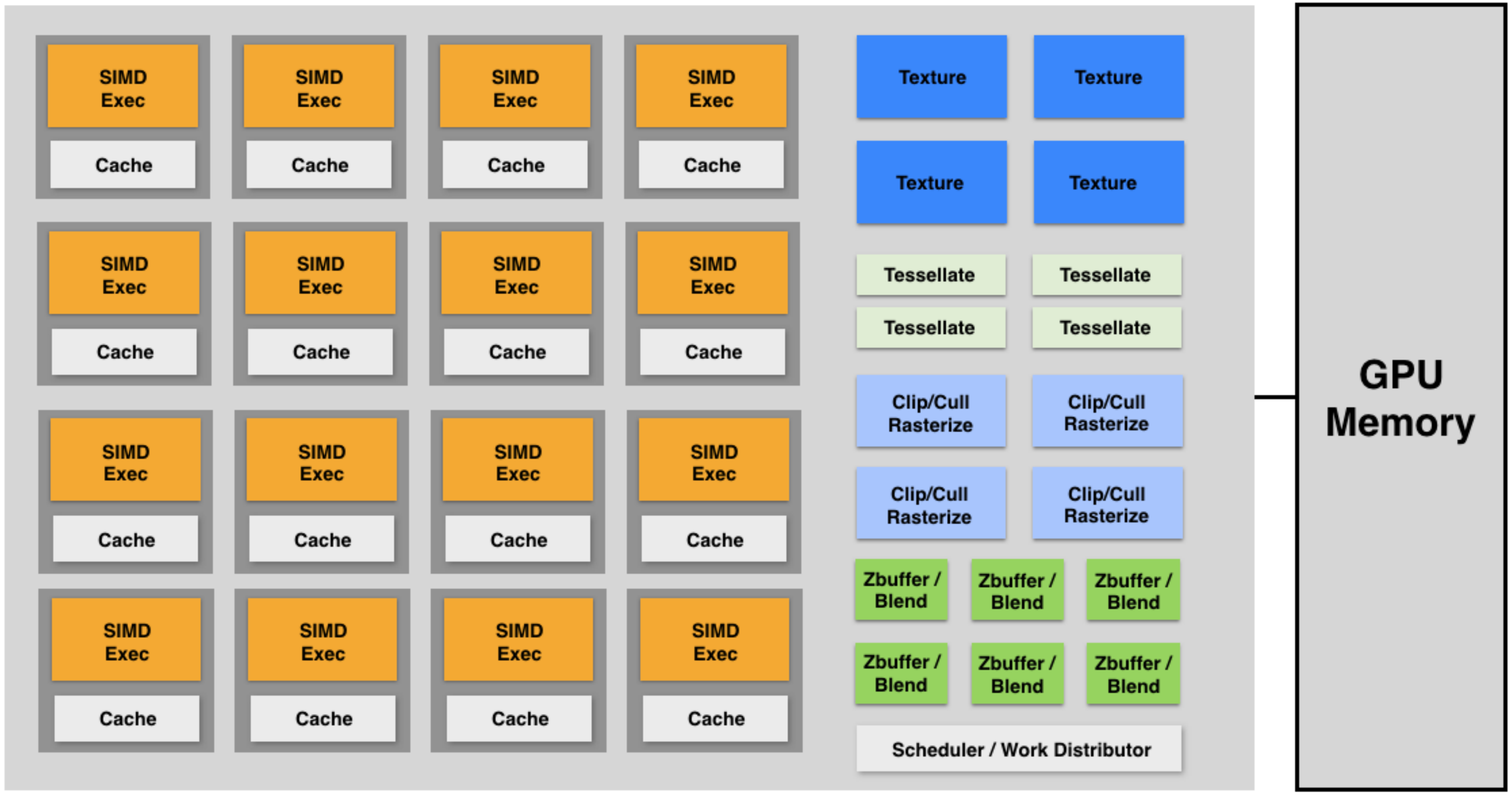
Modern graphics processing units (GPUs) provide a highly parallelized implementation of the rasterization pipeline

- Many SIMD cores for running vertex and fragment shaders in parallel
- Lots of fixed-function hardware for non-programmable stages (clipping, rasterization, texture sampling, z-buffering, etc.)

Discrete GPU card



Integrated GPU (part of CPU die)


SIMD Exec
Cache
Texture
Tessellate
Clip/Cull Rasterize
Zbuffer / Blend
Scheduler / Work Distributor
GPU Memory

Homework problems

1. Figure out a way to draw a pixel-perfect disk with radius *r* centered at a point (*x*,*y*) in 2D.

   Hint: Send a single quad (4 vertices and 2 triangles), and do the point-in-disk test inside the fragment shader.

2. Figure out a way to draw a pixel-perfect disk with radius *r* in pixels centered at the projected location of a 3D point (*x*,*y*,*z*).

   Hint: Set all 4 vertices' position to (*x*,*y*,*z*), but use an extra attribute to move them to the right corners in the vertex shader.

3. Assignment 1 will be out tonight! Keep an eye out on the course webpage.