Interactive Computer Graphics: Lecture 4

The Graphics Pipeline: OpenGL and GLSL

Thanks to Markus Steinberger and Dieter Schmalstieg

The Graphics Pipeline: High-level view

- Declarative (What, not How)
  - For example virtual camera with scene description, e.g. scene graphs
  - Every object may know about every other object
  - Renderman, Inventor, OpenSceneGraph,...

- Imperiative (How, not What)
  - Emit a sequence of drawing commands
  - For example: draw a point (vertex) at position (x,y,z)
  - Objects can be drawn independant from each other
  - OpenGL, PostScript, etc.

The Graphics Pipeline

Modelling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

Application

drawing commands

Input:
- geometric model
- illumination model
- camera model
- viewport

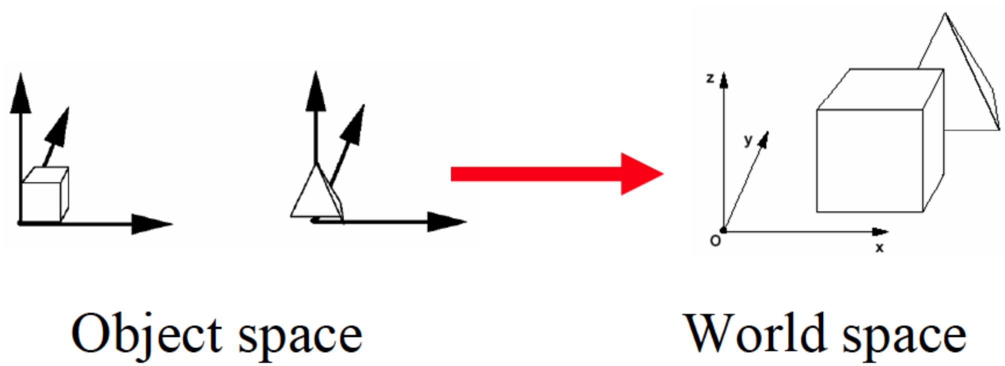
imperative pipeline!

Output: 2D image for framebuffer display

The Graphics Pipeline

- Modelling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

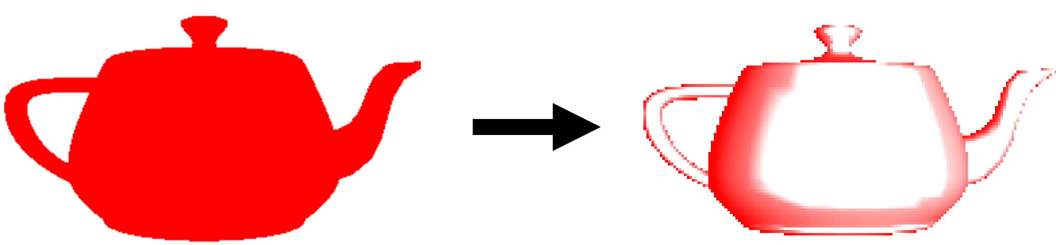
* 3D models are defined in their own coordinate system
* Modeling transformations orient the models within a common coordinate frame (world coordinates)



The Graphics Pipeline

- Modelling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Vertices are lit (shaded) according to material properties, surface properties and light sources
- Uses a local lighting model



The Graphics Pipeline

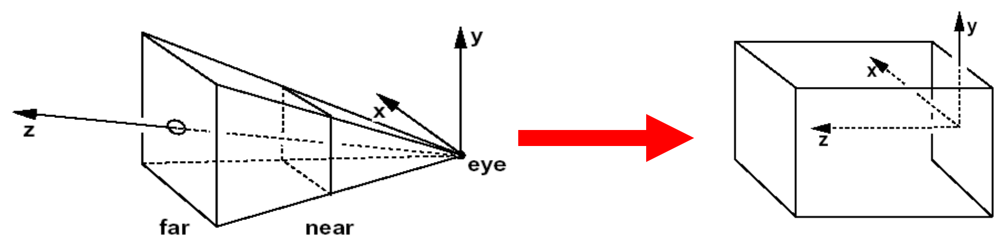
- Modelling Transformations
- Illumination (Shading)
- **Viewing Transformation (Perspective / Orthographic)**
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Maps world space to eye (camera) space (matrix evaluation)
- Viewing position is transformed to origin and viewing direction is oriented along some axis (typically z)

The Graphics Pipeline

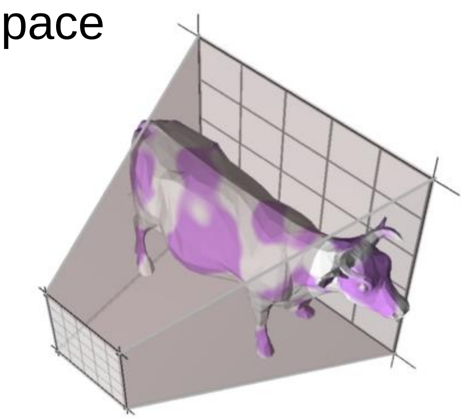
- Modelling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- **Clipping**
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Portions of the scene outside the viewing volume (view frustum) are removed (clipped)
- Transform to Normalized Device Coordinates



Eye space

NDC



The Graphics Pipeline

Modelling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

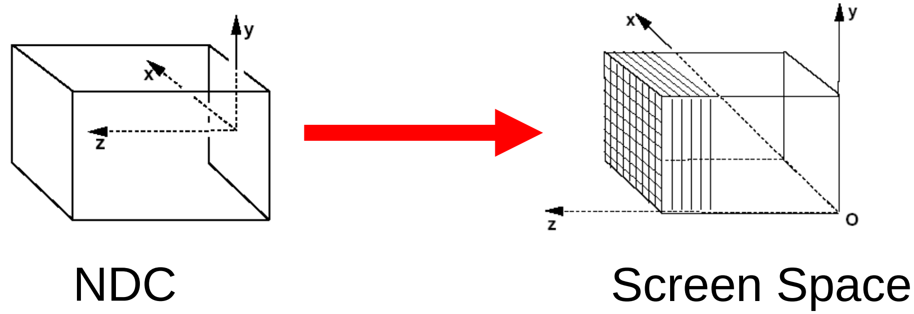
Clipping

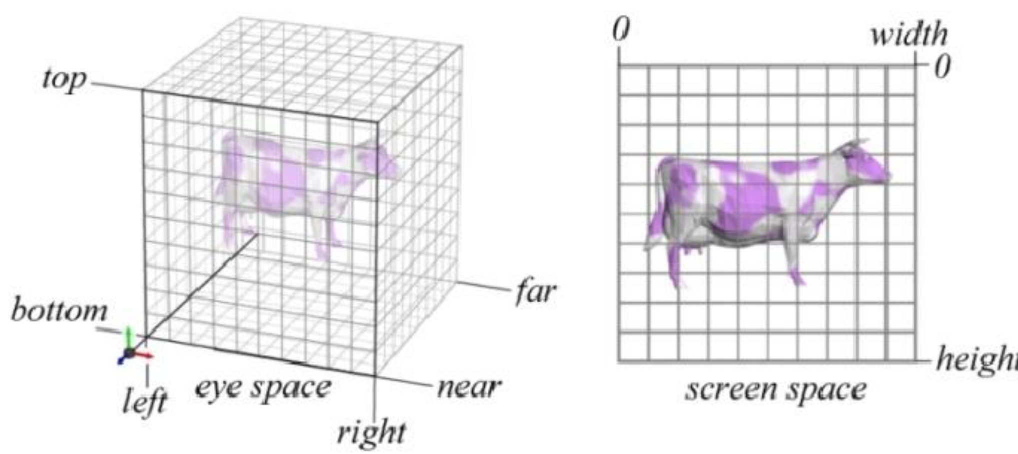
Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

- The objects are projected to the 2D imaging plane (screen space)





The Graphics Pipeline

Modelling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

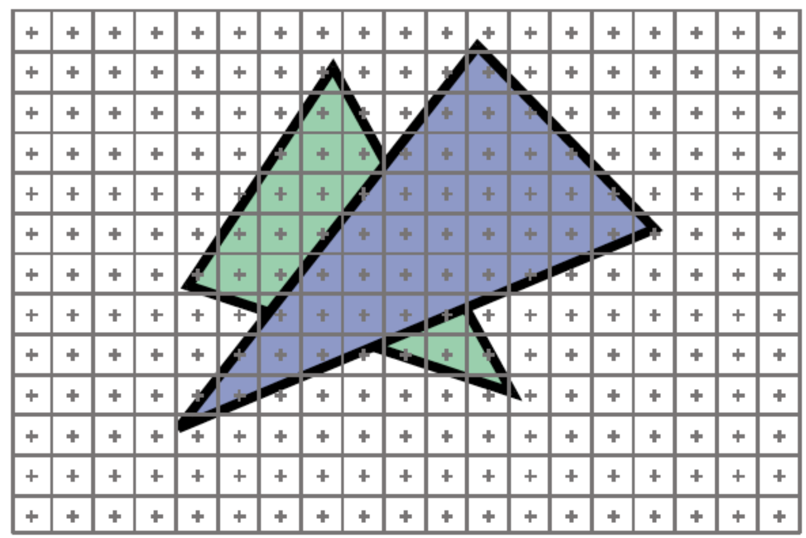
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

- Rasterizes objects into pixels
- Interpolate values inside objects (color, depth, etc.)



The Graphics Pipeline

Modelling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

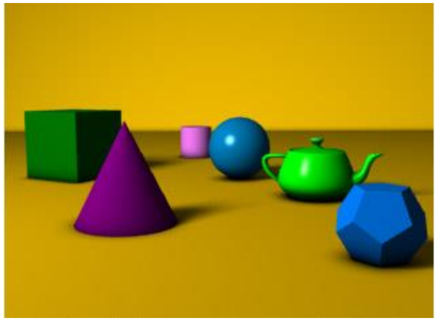
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

- Handles occlusions and transparency blending
- Determines which objects are closest and therefore visible
- Depth buffer



What do we want to do?

- Computer-generated imagery (CGI) in **real-time**
- Very computationally demanding:
  - full HD at 60hz:

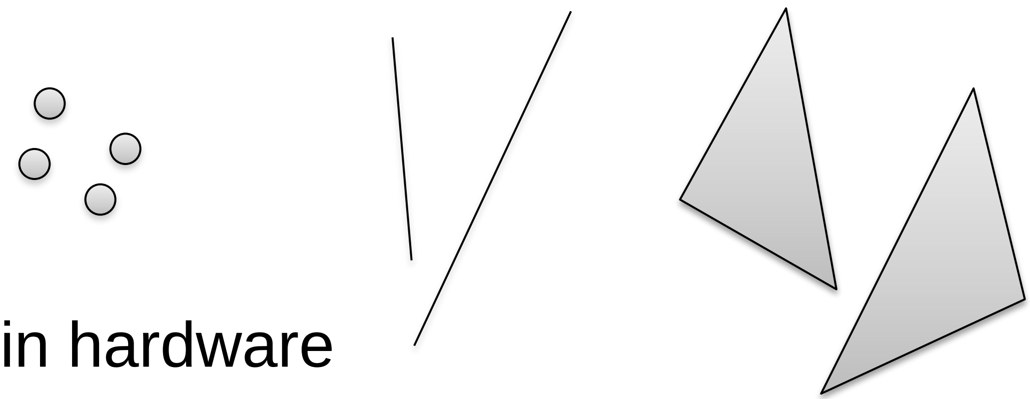
    1920 x 1080 x 60hz = 124 Mpx/s
  - and that's just the output data

→ use specialized hardware for immediate mode (real-time) graphics

Solution

Most of real-time graphics is based on

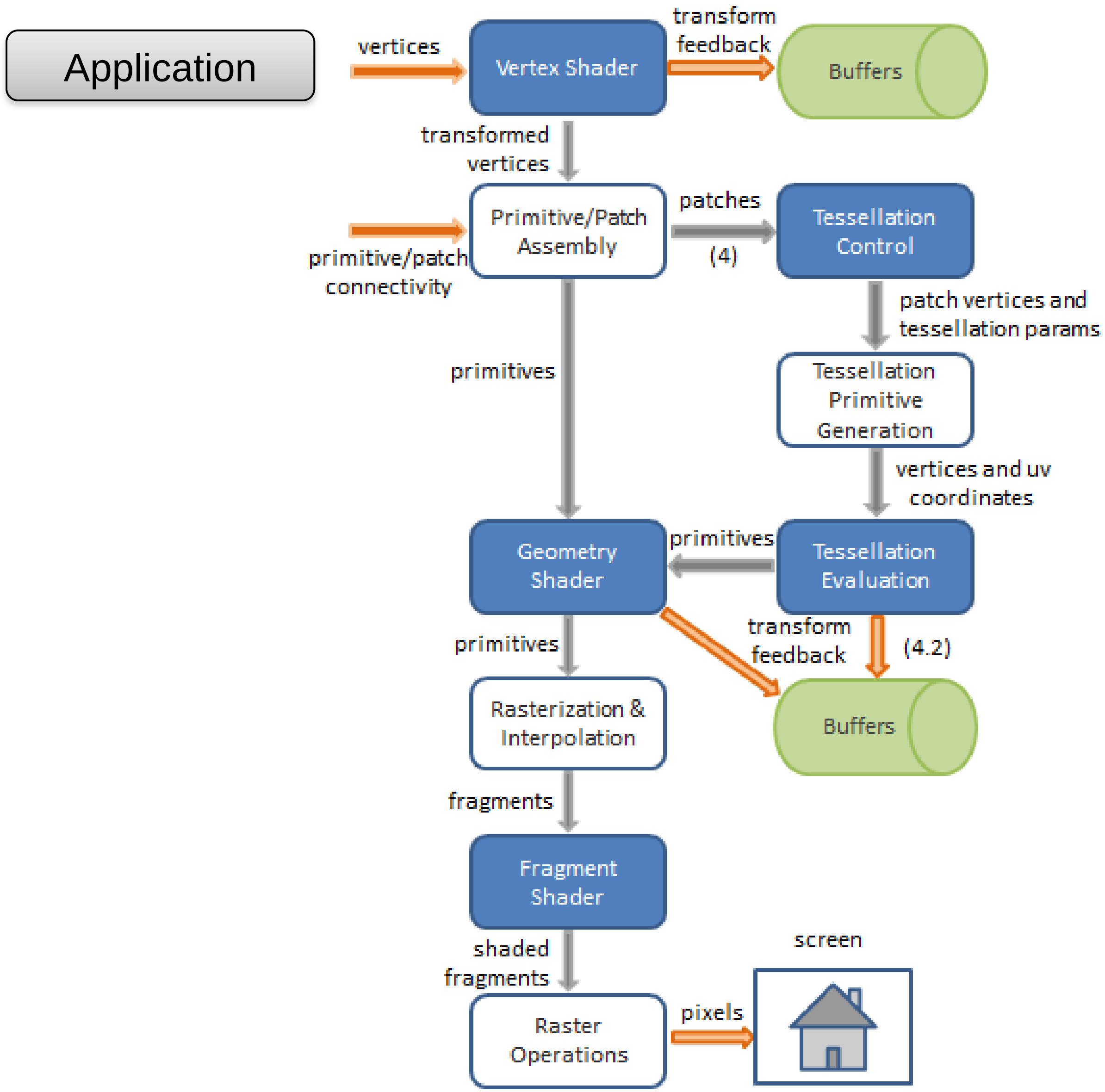
- rasterization of graphic *primitives*
  - points
  - lines
  - triangles
  - ...



- Implemented in hardware
  - *graphics processing unit* (GPU)
  - controlled through an API such as OpenGL
  - certain parts of graphics pipeline are programmable, e.g. using GLSL

    ➔ shaders

The Graphics Pipeline: OpenGL 3.2 and later



Programmable

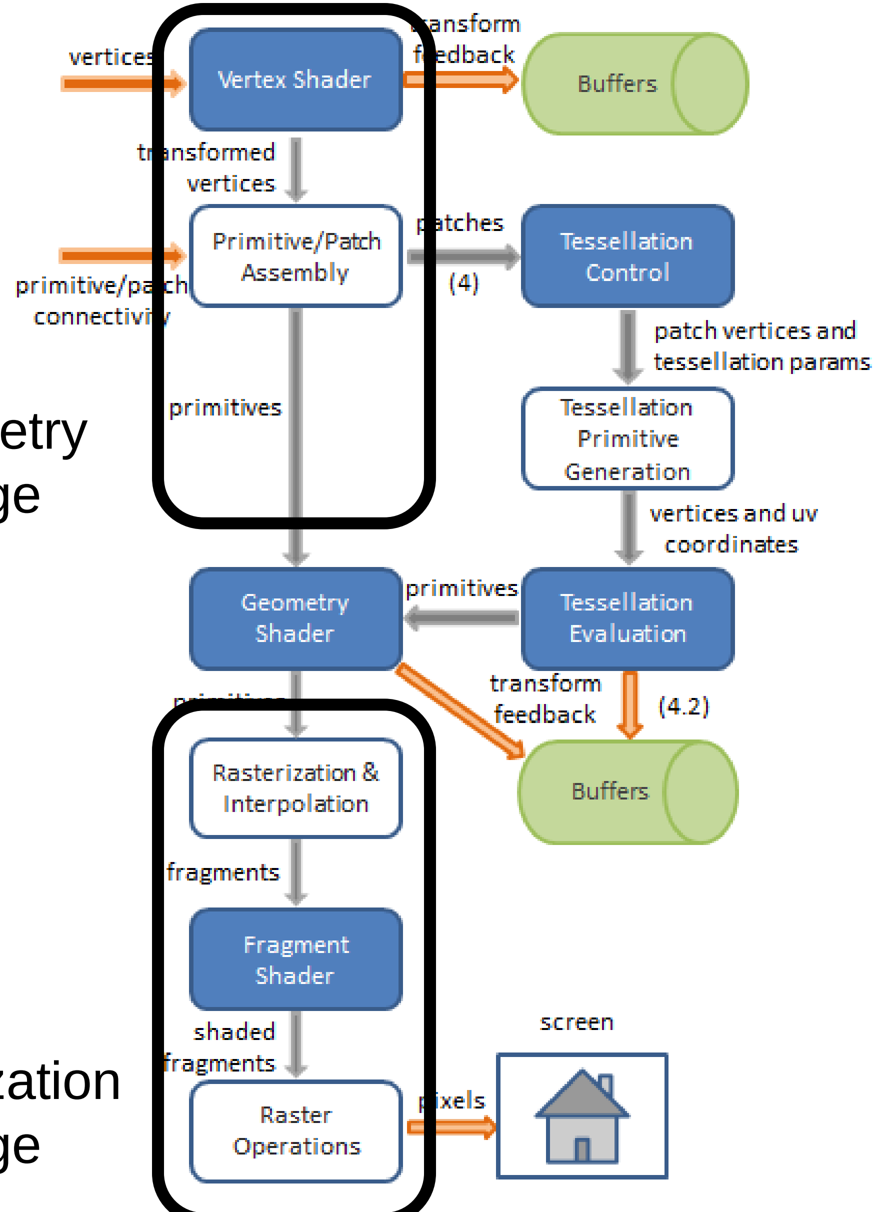
Fixed function

Source: www.lighthouse3d.com

The Graphics Pipeline: OpenGL 3.2 and later

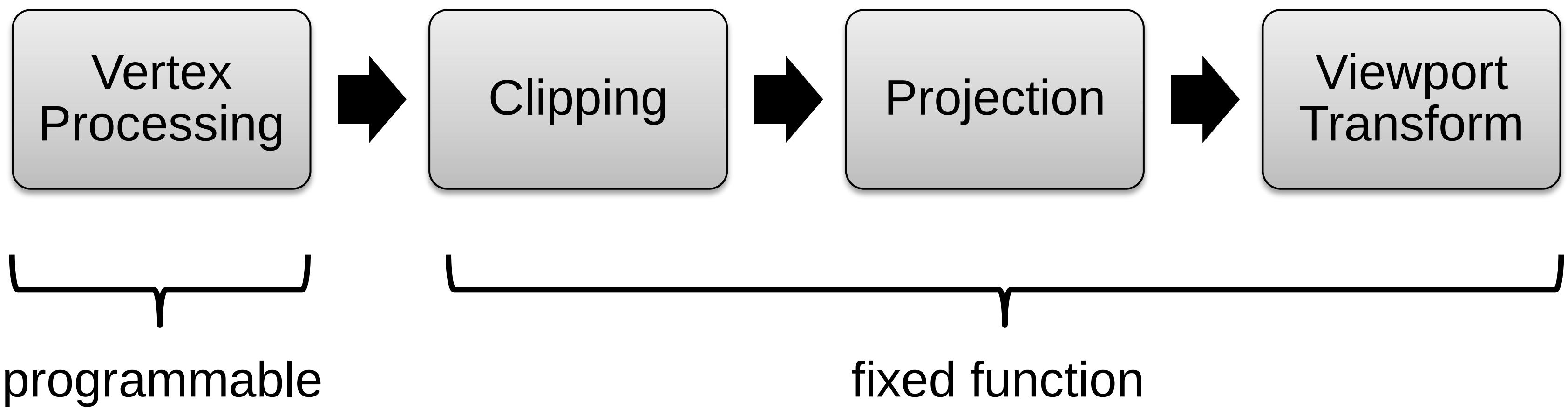
geometry stage

rasterization stage



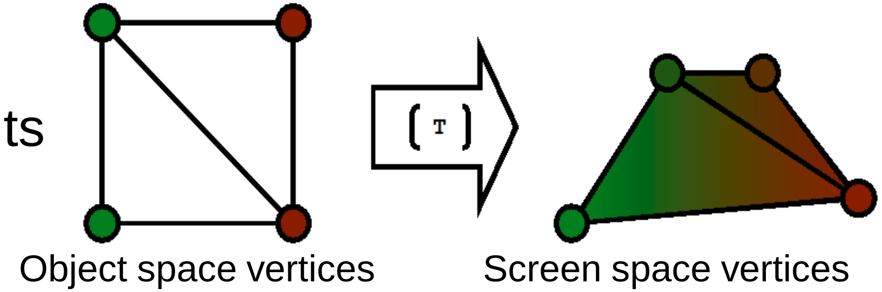
Source:
www.lighthouse3d.com

Geometry Stage



Geometry Stage: Vertex Processing

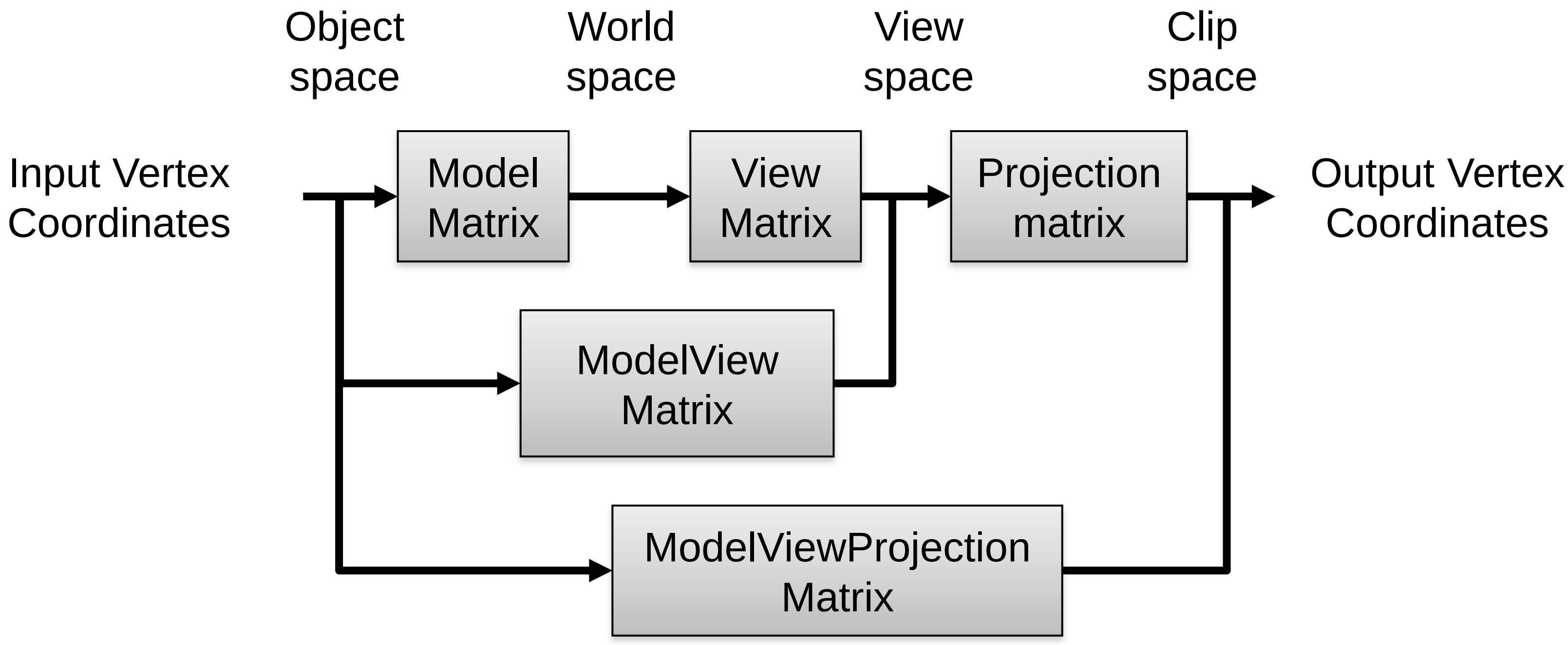
- The input vertex stream
  - composed of arbitrary vertex attributes (position, color, …).
- is transformed into stream of vertices mapped onto the screen
  - composed of their clip space coordinates and additional user-defined attributes (color, texture coordinates, …).
  - clip space: homogeneous coordinates
- by the ***vertex shader***
  - GPU program that implements this mapping.



- Historically, “Shaders” were small programs performing lighting calculations, hence the name.

Geometry Stage: Vertex Post-Processing

- Uses a common transformation model in rasterization-based 3D graphics:

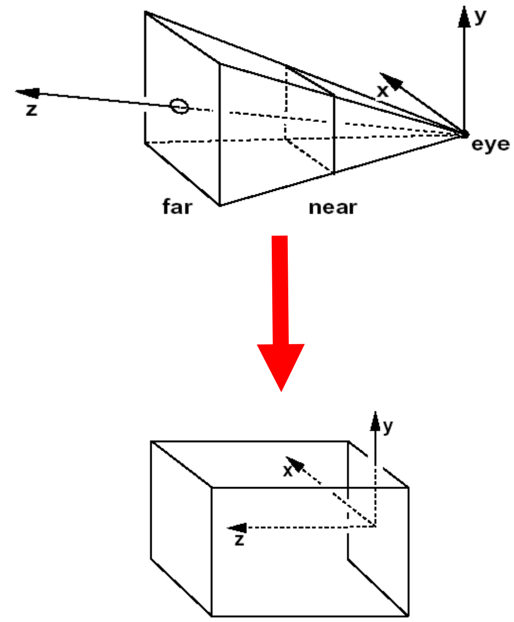


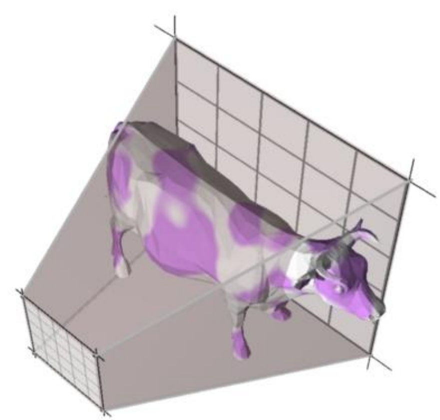
Geometry Stage: Vertex Post-Processing

- Clipping
  - Primitives not entirely in view are clipped to avoid projection errors
- Projection
  - Projects clip space coordinates to the image plane

  → Primitives in normalized device coordinates
- Viewport Transform:
  - Maps resolution-independent normalized device coordinates to a rectangular window in the frame buffer, the viewport.

  → Primitives in window (pixel) coordinates

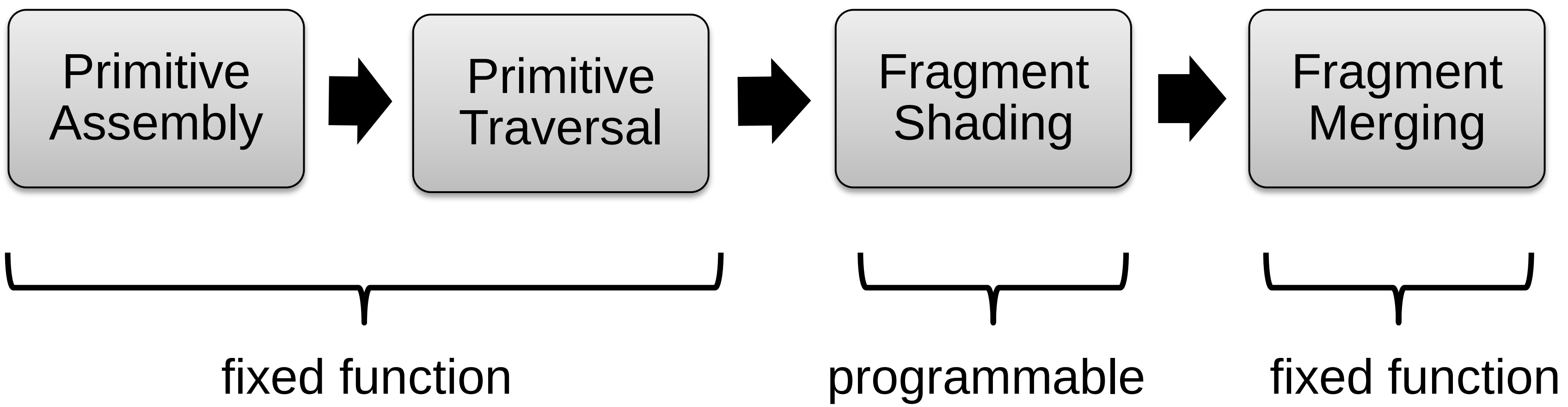




Geometry Shader

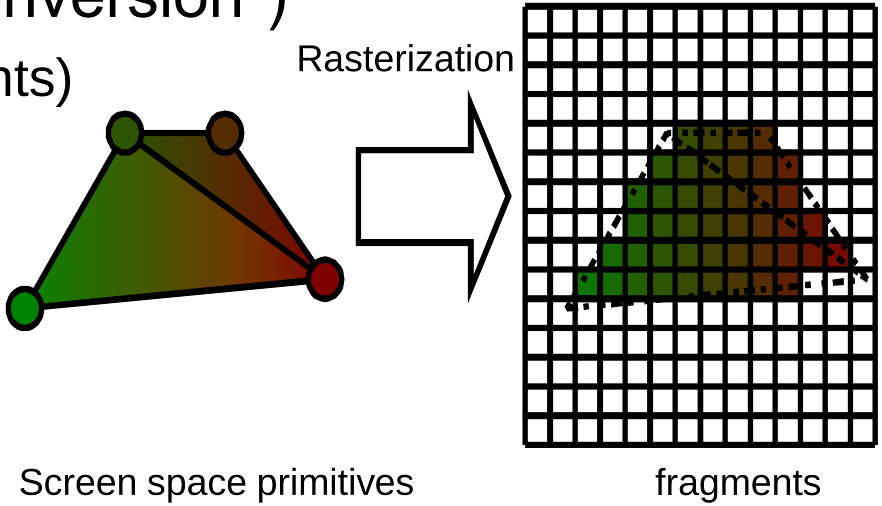
- Optional stage between vertex and fragment shader
- In contrast to the vertex shader, the geometry shader has full knowledge of the primitive it is working on
  - For each input primitive, the geometry shader has access to all the vertices that make up the primitive, including adjacency information.
- Can generate primitives dynamically
  - Procedural geometry, e.g. growing plants

Rasterization Stage

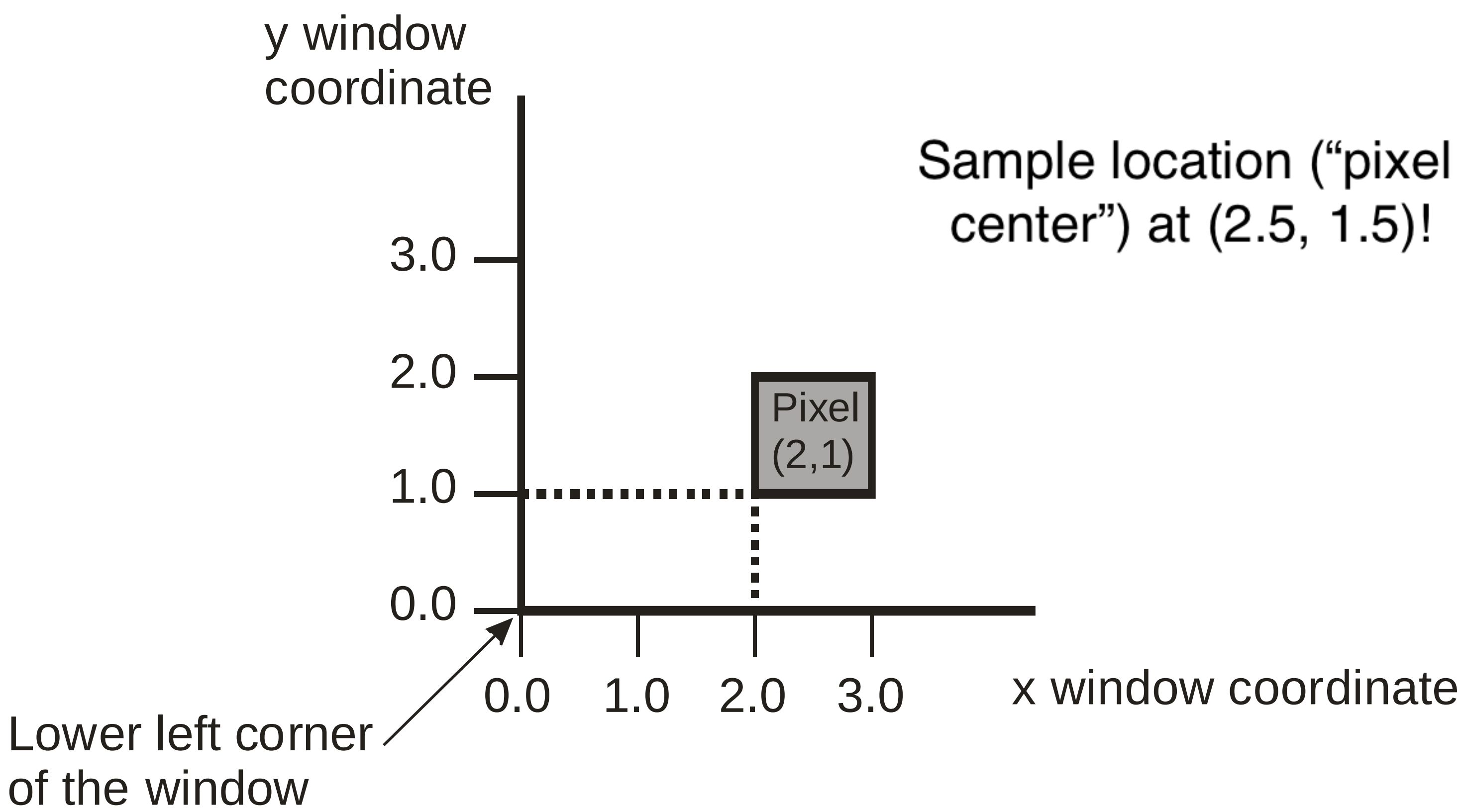


Rasterization Stage

- Primitive assembly
  - Backface culling
  - Setup primitive for traversal
- Primitive traversal (“scan conversion”)
  - Sampling (triangle → fragments)
  - Interpolation of vertex attributes (depth, color, …)
- Fragment shading
  - Compute fragment colors
- Fragment merging
  - Compute pixel colors from fragments
  - Depth test, blending, …

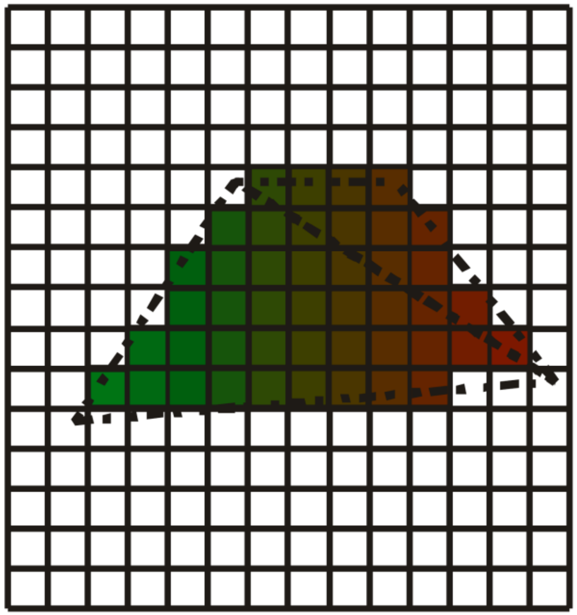


Rasterization – Coordinates



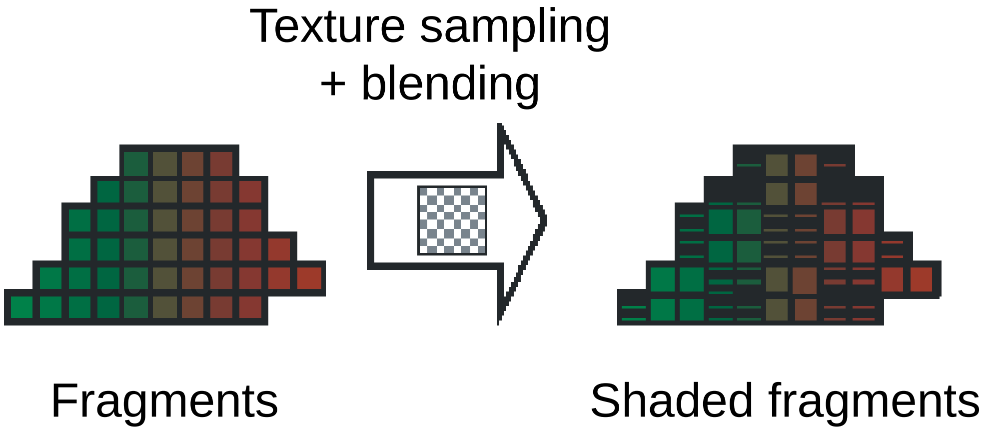
Rasterization – Rules

- Different rules for each primitive type
  - "fill convention"
- Non-ambiguous!
  - artifacts…
- Polygons:
  - Pixel center contained in polygon
  - Pixels on edge: only one is rasterized



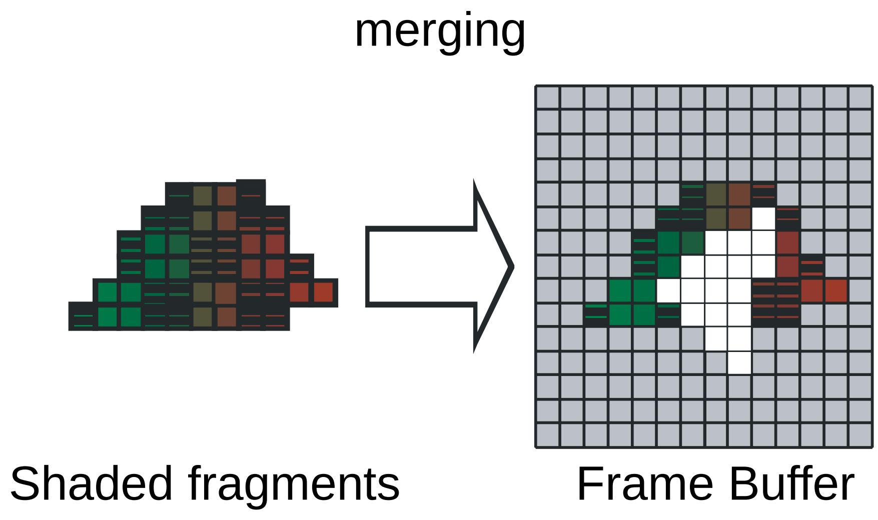
Fragment Shading

- "Fragment":
  - Sample produced during rasterization
  - Multiple fragments are *merged* into pixels.
- Given the interpolated vertex attributes,
  - output by the Vertex Shader
- the *Fragment Shader* computes color values for each fragment.
  - Apply textures
  - Lighting calculations
  - …

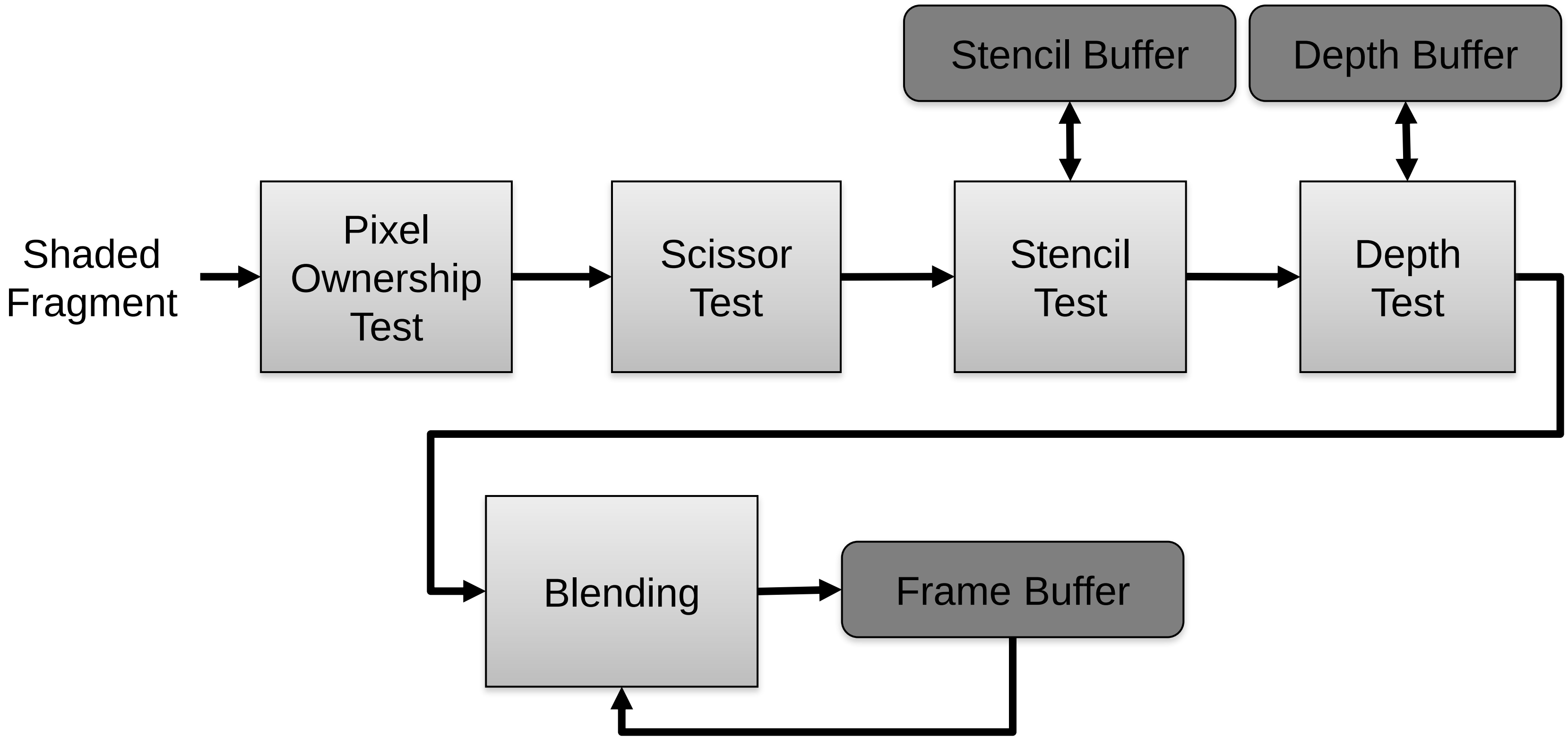


Fragment Merging

- Multiple primitives can cover the same pixel.
- Their Fragments need to be composed to form the final pixel values.
  - Blending
  - Resolve Visibility
    - Depth buffering

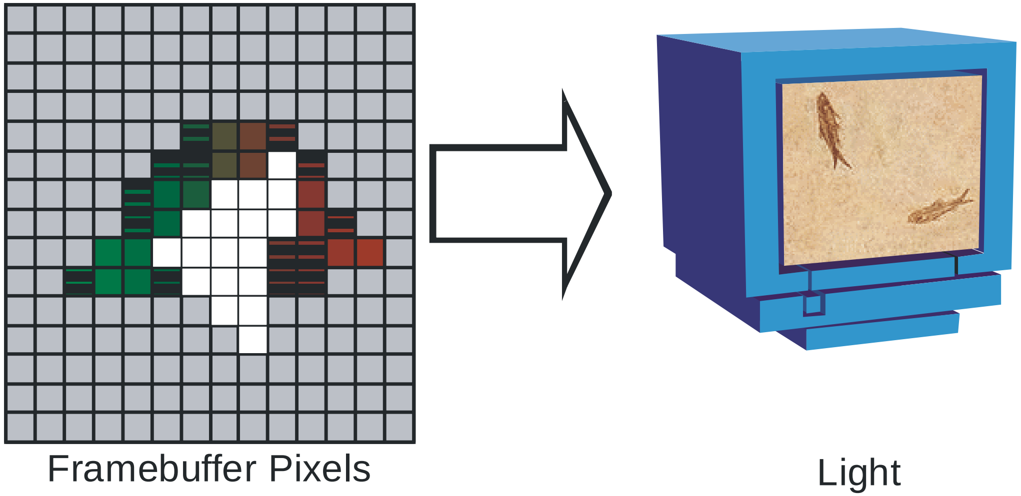


Fragment Merging



Display Stage

- Gamma correction
- Historically: Digital to Analog conversion
- Today: Digital scan-out, HDMI encryption, etc.



Display Format

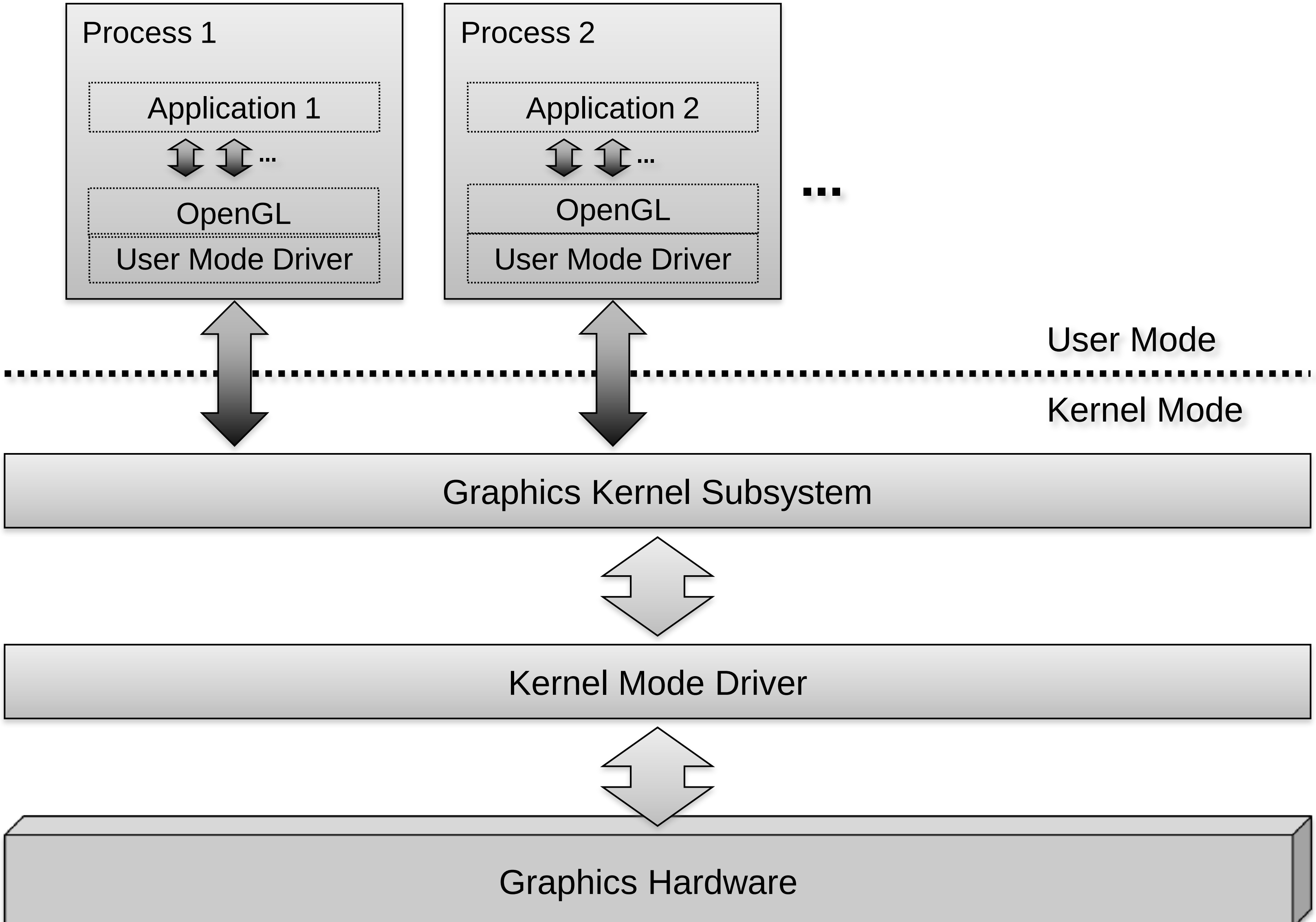
- Frame buffer pixel format:
  RGBA vs. index (obsolete)
- Bits: 16, 32, 64, 128 bit floating point, …
- Double buffer vs. single buffer
- Quad-buffered stereo
- Overlays (extra bitplanes)
- Auxilliary buffers: alpha, stencil

Functionality vs. Frequency

- Geometry processing = per-vertex
  - Transformation and Lighting (T&L)
  - Historically floating point, complex operations
  - Millions of vertices per second
  - Today: Vertex Shader
- Fragment processing = per-fragment
  - Blending, texture combination
  - Historically fixed point and limited operations
  - Billions of fragments (“Gigapixel” per second)
  - Today: Fragment Shader

Architectural Overview

- Graphics Hardware is a shared resource
- User Mode Driver (UMD)
  - Prepares Command Buffers for the hardware
- Graphics Kernel Subsystem
  - Schedules access to the hardware
- Kernel Mode Driver (KMD)
  - Submits Command Buffers to the hardware



Process 1
Application 1
...
OpenGL
User Mode Driver
Process 2
Application 2
...
OpenGL
User Mode Driver
...
User Mode
Kernel Mode
Graphics Kernel Subsystem
Kernel Mode Driver
Graphics Hardware

What is OpenGL?

- a low-level graphics API specification
  - not a library!
    - The interface is platform independent,
    - but the implementation is platform dependent.
  - Defines
    - an abstract rendering device.
    - a set of functions to operate the device.
  - “immediate mode” API
    - drawing commands
    - no concept of permanent objects

What is OpenGL?

- Platform provides OpenGL *implementation*.
  - Part of the graphics driver, or
  - runtime library built on top of the driver

- Initialization through platform specific API
  - WGL (Windows)
  - GLX (Unix/Linux)
  - EGL (mobile devices)
  - ...

- State machine for high efficiency!

Basic Concepts

- Context
- Resources
- Object Model
  - Objects
  - Object Names
  - Bind Points (Targets)

Context

- Represents an instance of OpenGL
- A process can have multiple contexts
  - These can share resources
- A context can be *current* for a given thread
  - one to one mapping
    - only one current context per thread
    - context only current in one thread at the same time
  - OpenGL operations work on the current context

Resources

- Act as
  - sources of input
  - sinks for output
- Examples:
  - buffers
  - images
  - state objects
  - …

Resources

- Buffer objects
  - linear chunks of memory
- Texture images
  - 1D, 2D, or 3D arrays of *texels*
  - Can be used as input for *texture sampling*

Object Model

- OpenGL is object oriented
  - but in its own, strange way
- Object instances are identified by a *name*
  - basically just an unsigned integer handle
- Commands work on *targets*
  - Each target has an object currently *bound* to the target
    - That’s the one commands will work with
- Object oriented, you said?
  - target ⇔ type
  - commands ⇔ methods

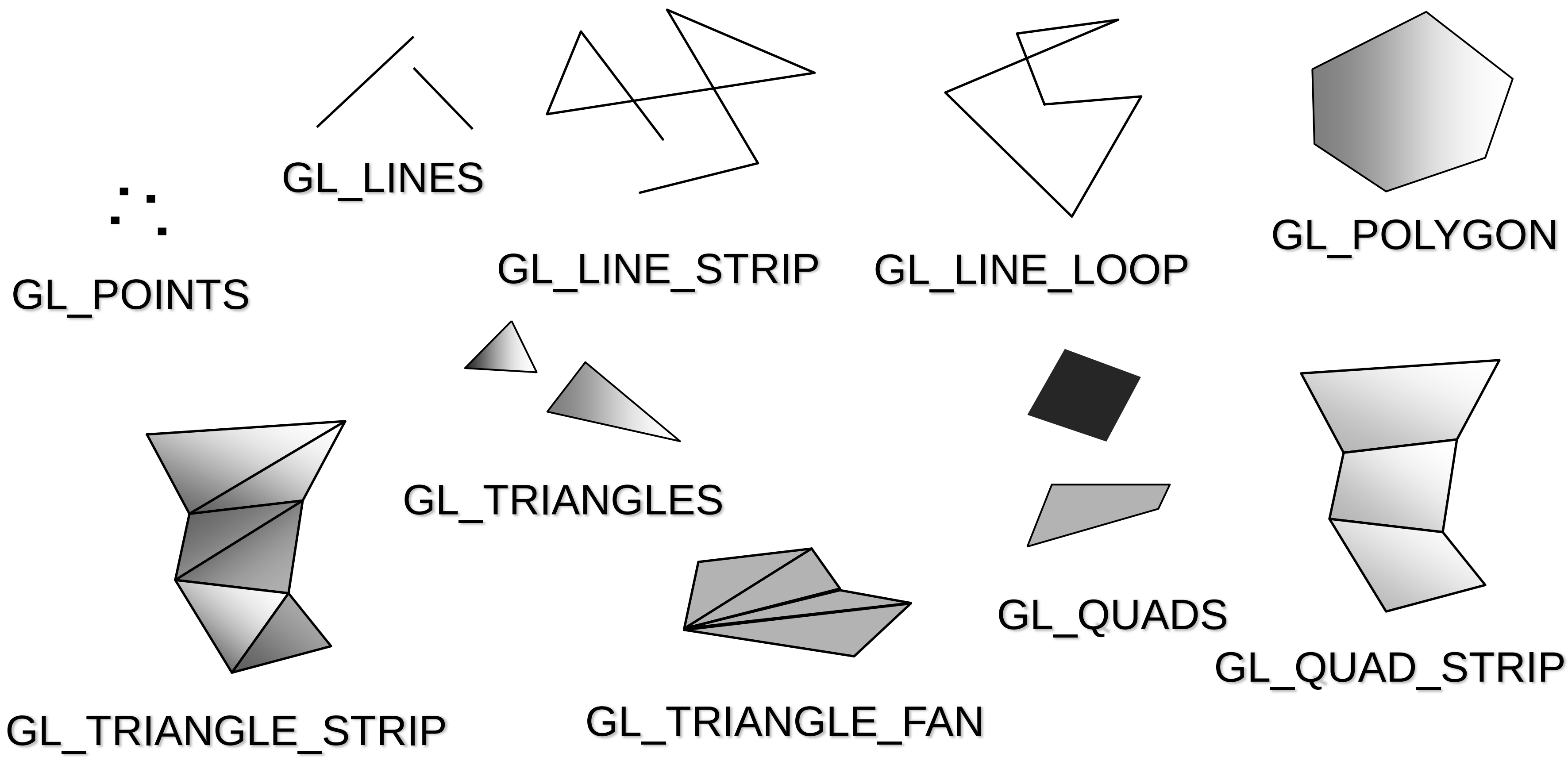
Object Model

- By binding a name to a target
  - the object it identifies becomes current for that target
    - "latched state"
    - might change soon (`EXT_direct_state_access`)
  - An object is created when a name is first bound.
- Notable exceptions: Shader Objects, Program Objects
  - Some commands work directly on object names.

Example: Buffer Object

```
GLuint my_buffer;

// request an unused buffer object name
glGenBuffers(1, &my_buffer);

// bind name as GL_ARRAY_BUFFER
// bound for the first time ⇒ creates
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);

// put some data into my_buffer
glBufferStorage(GL_ARRAY_BUFFER, …);

// “unbind” buffer
glBindBuffer(GL_ARRAY_BUFFER, 0);

// probably do something else…
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
// use my_buffer…

glDrawArrays(GL_TRIANGLES, 0, 33);
// draw content example (type, startIdx, numer of elements)

// delete buffer object, free resources, release buffer object name
glDeleteBuffers(1, &my_buffer);
```

Primitive types



Draw Call

- After pipeline is configured:
  - issue *draw call* to actually draw something

primitive type

e. g.:

```
glBegin(GL_TRIANGLE_STRIP);
glColor3f(0.0, 1.0, 0.0);
glVertex3f(1.0, 0.0, 0.0);
…
glEnd();
```

Color “state”

vertex index

Shaders

- Shader Objects
  - parts of a pipeline (Vertex Shader, Fragment Shader, etc.)
  - compiled during runtime from GLSL code
    - OpenGL Shading Language
    - C-like syntax
- Program Object
  - a whole pipeline
  - Shader objects linked together during runtime

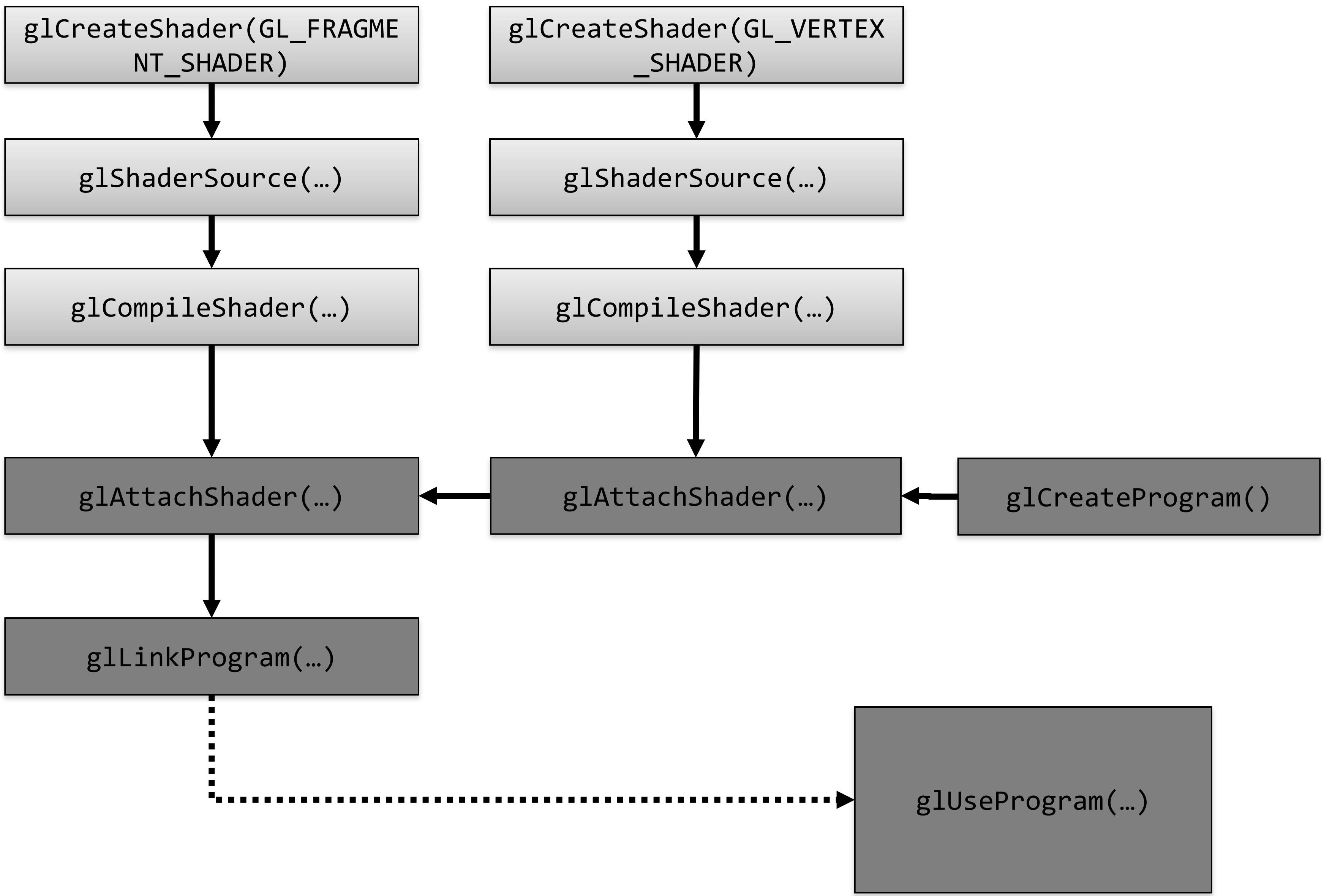
Anatomy of a GLSL Shader

```
#version 330

uniform vec4 some_uniform;

layout(location = 0) in vec3 some_input;
layout(location = 1) in vec4 another_input;

out vec4 some_output;

void main()
{

}
```

Set by application (configuration values, e.g. ModelViewProjection Matrix)

Optional flexible register configuration between shaders

Output definition for next shader stage


glCreateShader(GL_FRAGMENT_SHADER)
glShaderSource(…)
glCompileShader(…)
glCreateShader(GL_VERTEX_SHADER)
glShaderSource(…)
glCompileShader(…)
glAttachShader(…)
glAttachShader(…)
glCreateProgram()
glLinkProgram(…)
glUseProgram(…)

Vertex Shader

- Processes each vertex
- Input: vertex attributes
- Output: vertex attributes
  - mandatory: gl_Position

Rasterizer

- Fixed-function
- Rasterizes primitives
- Input: primitives
  - vertex attributes
- Output: fragments
  - interpolated vertex attributes

Fragment Shader

- Processes each fragment
- Input: interpolated vertex attributes
- Output: fragment color

Example: Vertex Shader

```
#version 150 compatibility
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec4 vertex_color;

out vertexData
{
  vec3 pos;
  vec3 normal;
  vec4 color;
}vertex;

void main()
{
  vertex.pos = vec3(gl_ModelViewMatrix * gl_Vertex);
  vertex.normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex.color = vec4(1.0,0.0,0.0,1.0);
}
```

Example: Fragment Shader

```
#version 150 compatibility
in vec4 color;

in fragmentData
{
  vec3 pos;
  vec3 normal;
  vec4 color;
}frag;

void main()
{
   vec4 outcol = frag.color;
    gl_FragColor = outcol;
}
```