Interactive Computer Graphics: Lecture 4

Graphics Pipeline and APIs

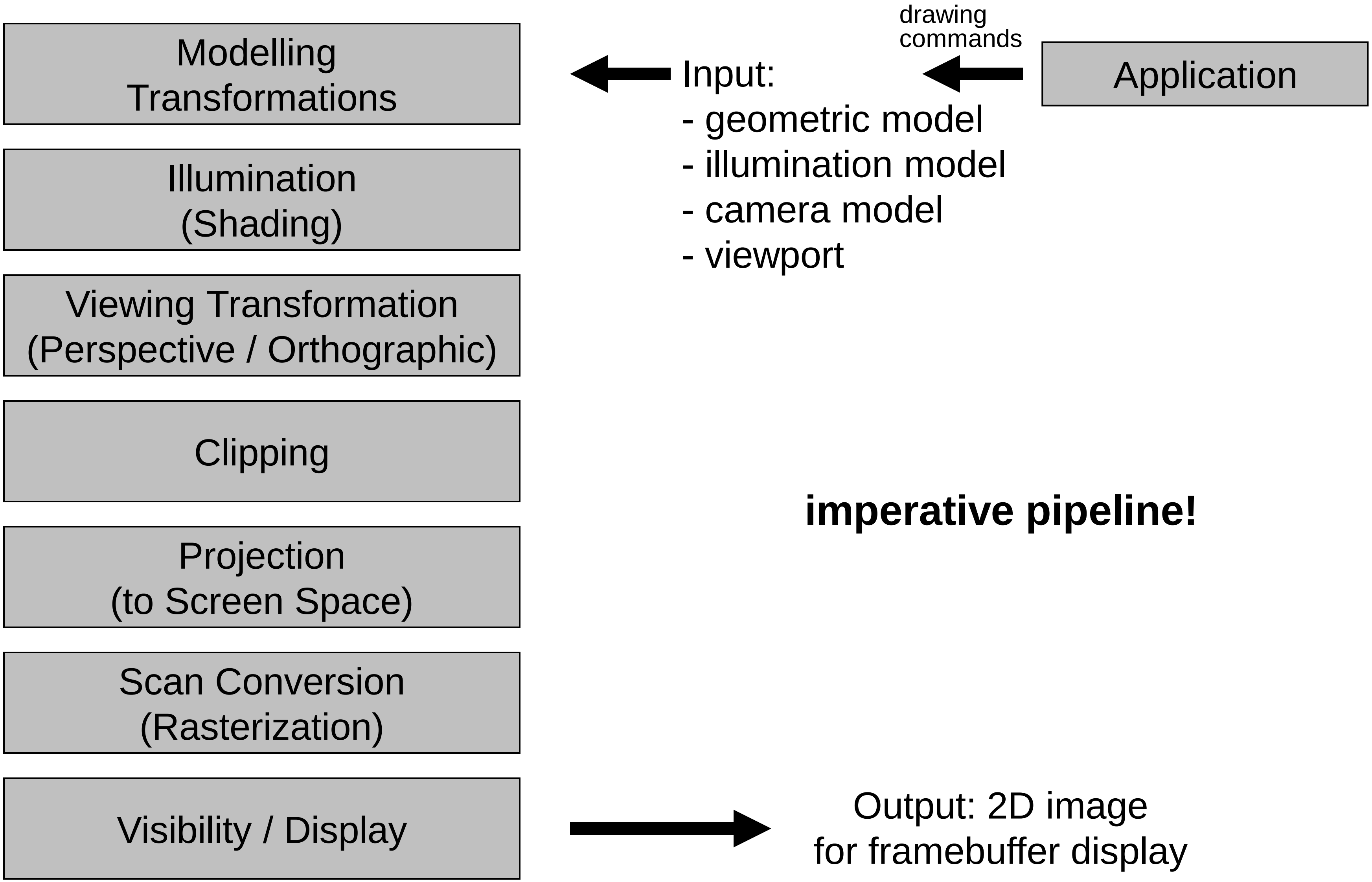
Some slides adopted from Markus Steinberger and Dieter Schmalstieg

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.

- You can always build a declarative pipeline on top of imperative model

The Graphics Pipeline

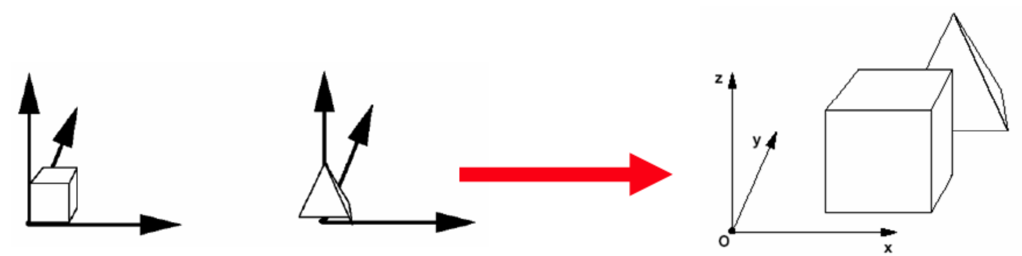


imperative pipeline!

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)



Object space World space

The Graphics Pipeline

Modelling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

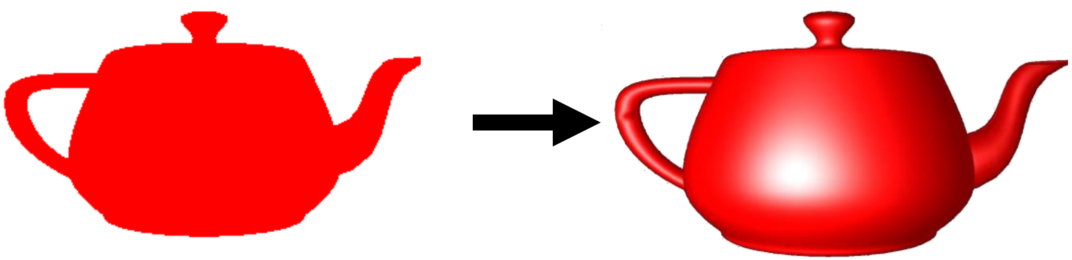
Clipping

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

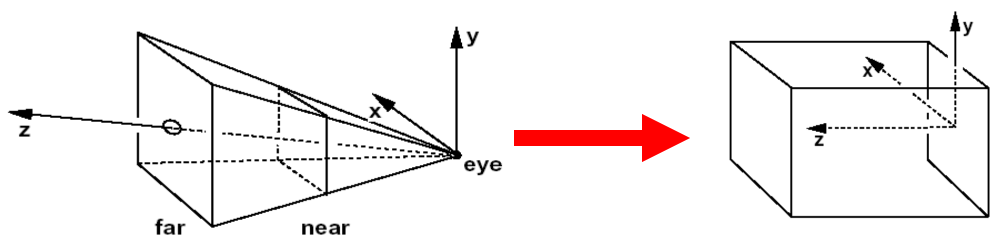
Modelling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display

- 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)

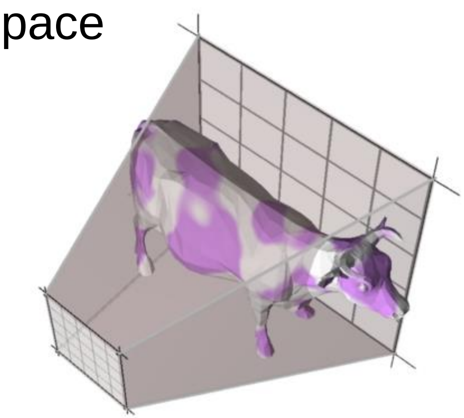
The Graphics Pipeline

- 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)

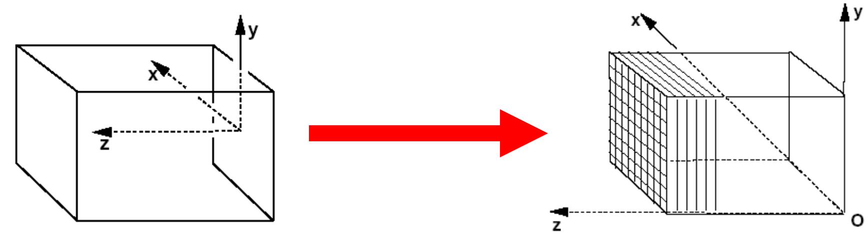
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

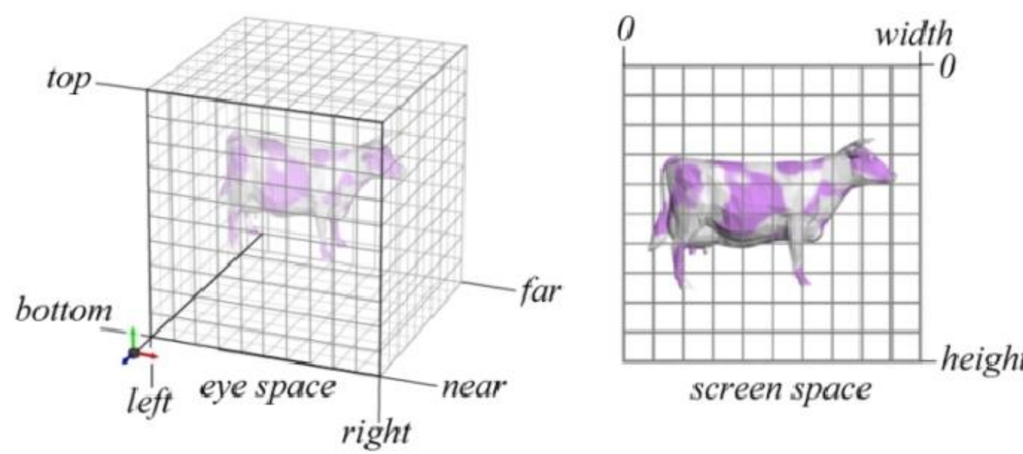
Visibility / Display

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



NDC

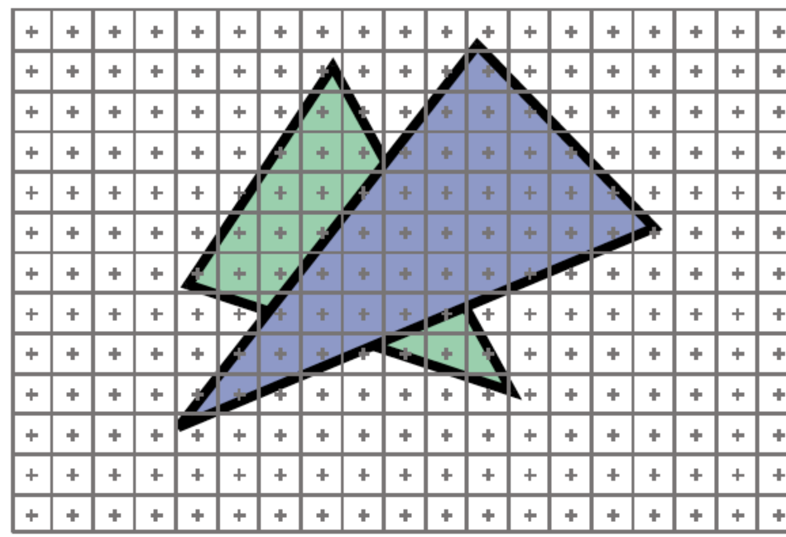
Screen Space



The Graphics Pipeline

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

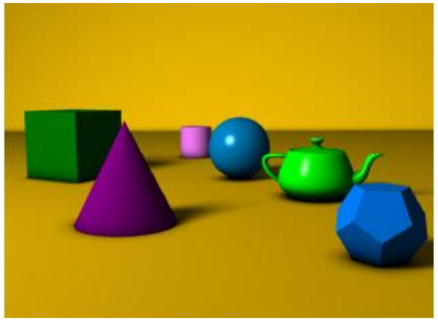
- Rasterizes objects into pixels
- Interpolate values inside objects (color, depth, etc.)



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:

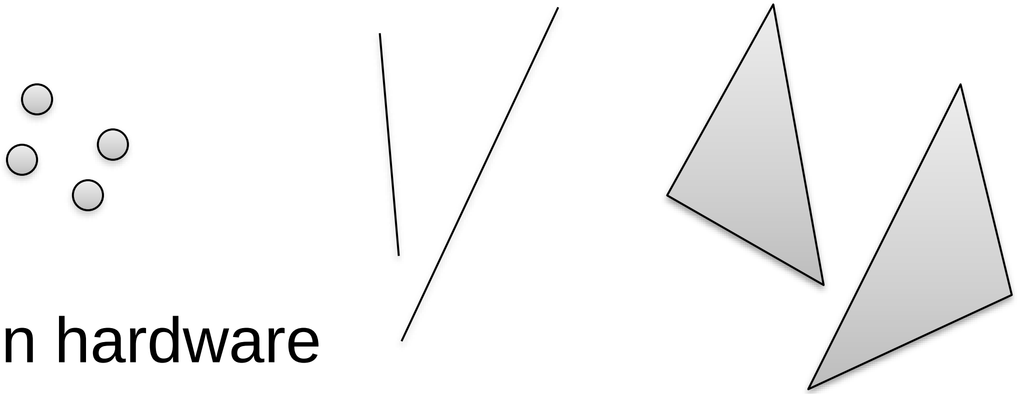
    1920 x 1080 x 60hz = 124 Mpx/s
  - and that's just the output data

→ 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
  - ...



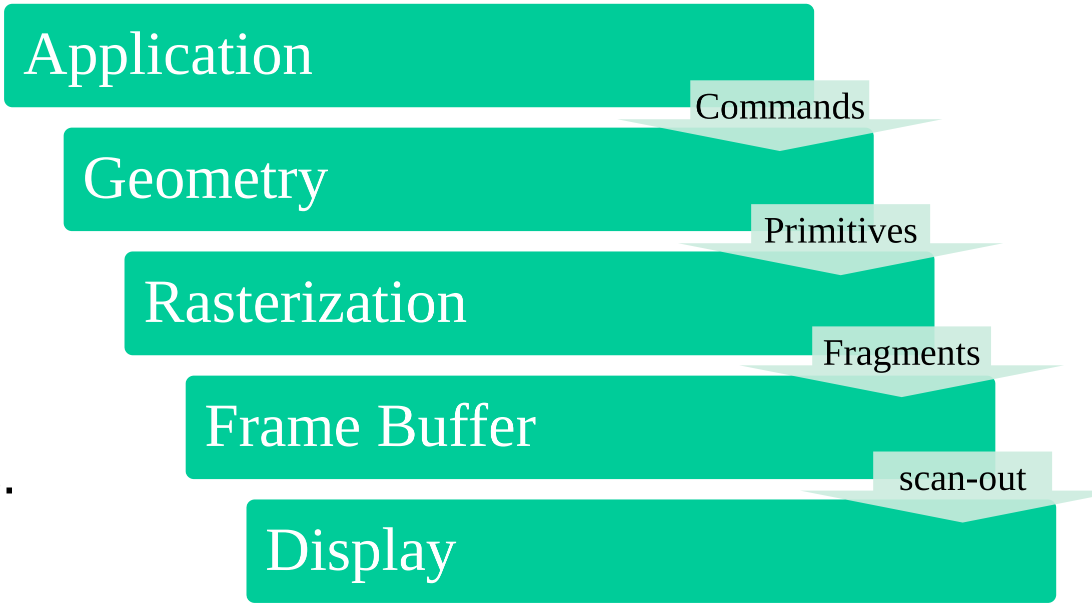
- Implemented in hardware
  - *graphics processing unit* (GPU)
  - controlled through an API such as OpenGL
  - certain parts of graphics pipeline are programmable, e.g. using GLSL

    ➔ shaders

The Graphics Pipeline different view

- High-level view:

- “Vertex”
  - a point in space defining geometry
- “Fragment”:
  - Sample produced during rasterization
  - Multiple fragments are *merged* into pixels.



Application Stage

- Generate database
  - Usually only once
  - Load from disk
  - Build acceleration structures (hierarchy, …)
- Simulation
- Input event handlers
- Modify data structures
- Database traversal
- Utility functions

Application Stage

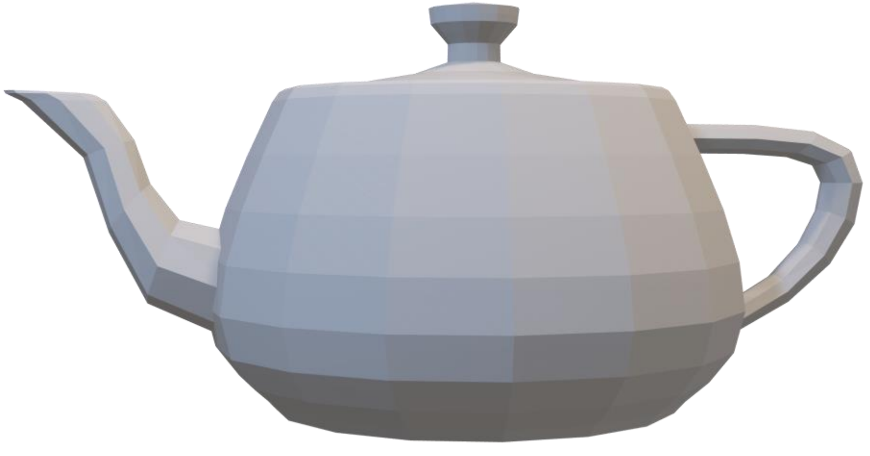
- Generate render area in OS
- Generate database
  - Usually only once
  - Load from disk
  - Build acceleration structures (hierarchy, …)
- Simulation
- Input event handlers
- Modify data structures
- Database traversal
- Utility functions

Application Stage

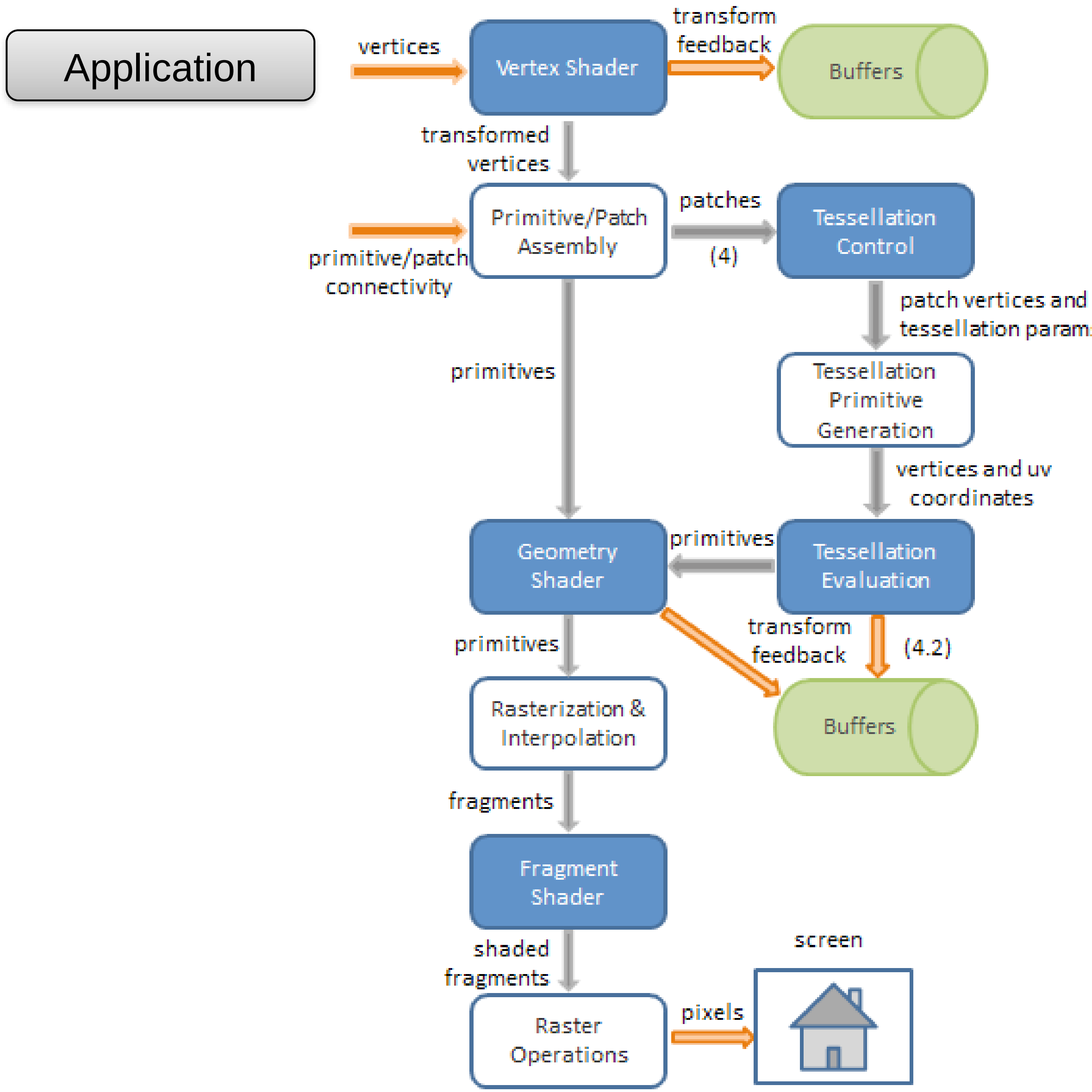
```
 solid TEATEST
facet normal  0.986544E+00  0.100166E+00  0.129220E+00
 outer loop
  vertex  0.167500E+02  0.505000E+02  0.000000E+00
  vertex  0.164599E+02  0.505000E+02  0.221480E+01
  vertex  0.166819E+02  0.483135E+02  0.221480E+01
 endloop
endfacet
facet normal  0.986495E+00  0.100374E+00  0.129434E+00
 outer loop
  vertex  0.166819E+02  0.483134E+02  0.221470E+01
  vertex  0.169653E+02  0.483840E+02  0.000000E+00
  vertex  0.167500E+02  0.505000E+02  0.000000E+00
 endloop
Endfacet
….
```

Application Stage

```
 solid TEATEST
facet normal  0.986544E+00  0.100166E+00  0.129220E+00
 outer loop
  vertex  0.167500E+02  0.505000E+02  0.000000E+00
  vertex  0.164599E+02  0.505000E+02  0.221480E+01
  vertex  0.166819E+02  0.483135E+02  0.221480E+01
 endloop
endfacet
facet normal  0.986495E+00  0.100374E+00  0.129434E+00
 outer loop
  vertex  0.166819E+02  0.483134E+02  0.221470E+01
  vertex  0.169653E+02  0.483840E+02  0.000000E+00
  vertex  0.167500E+02  0.505000E+02  0.000000E+00
 endloop
Endfacet
….
```



The Graphics Pipeline: OpenGL 3.2 and later



Programmable

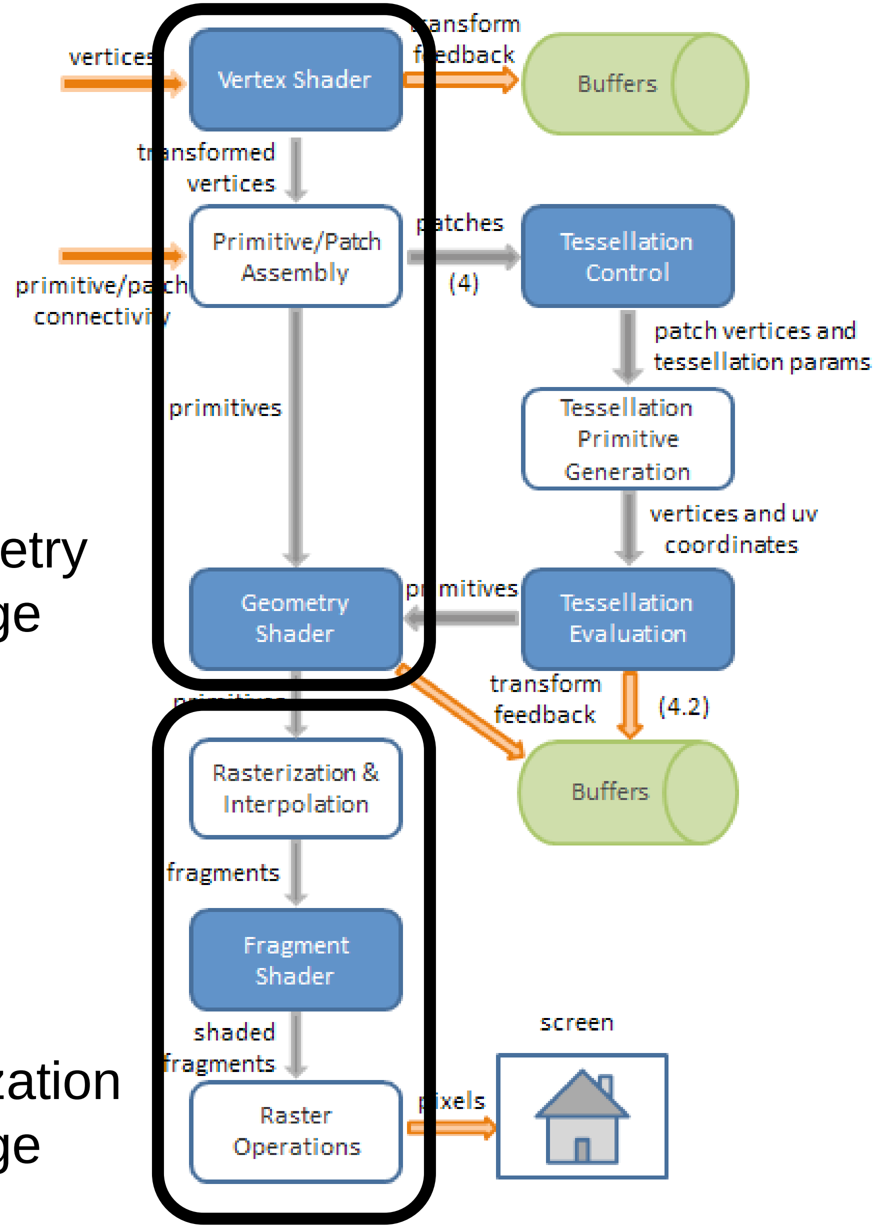
Fixed function

Source: www.lighthouse3d.com

The Graphics Pipeline: OpenGL 3.2 and later

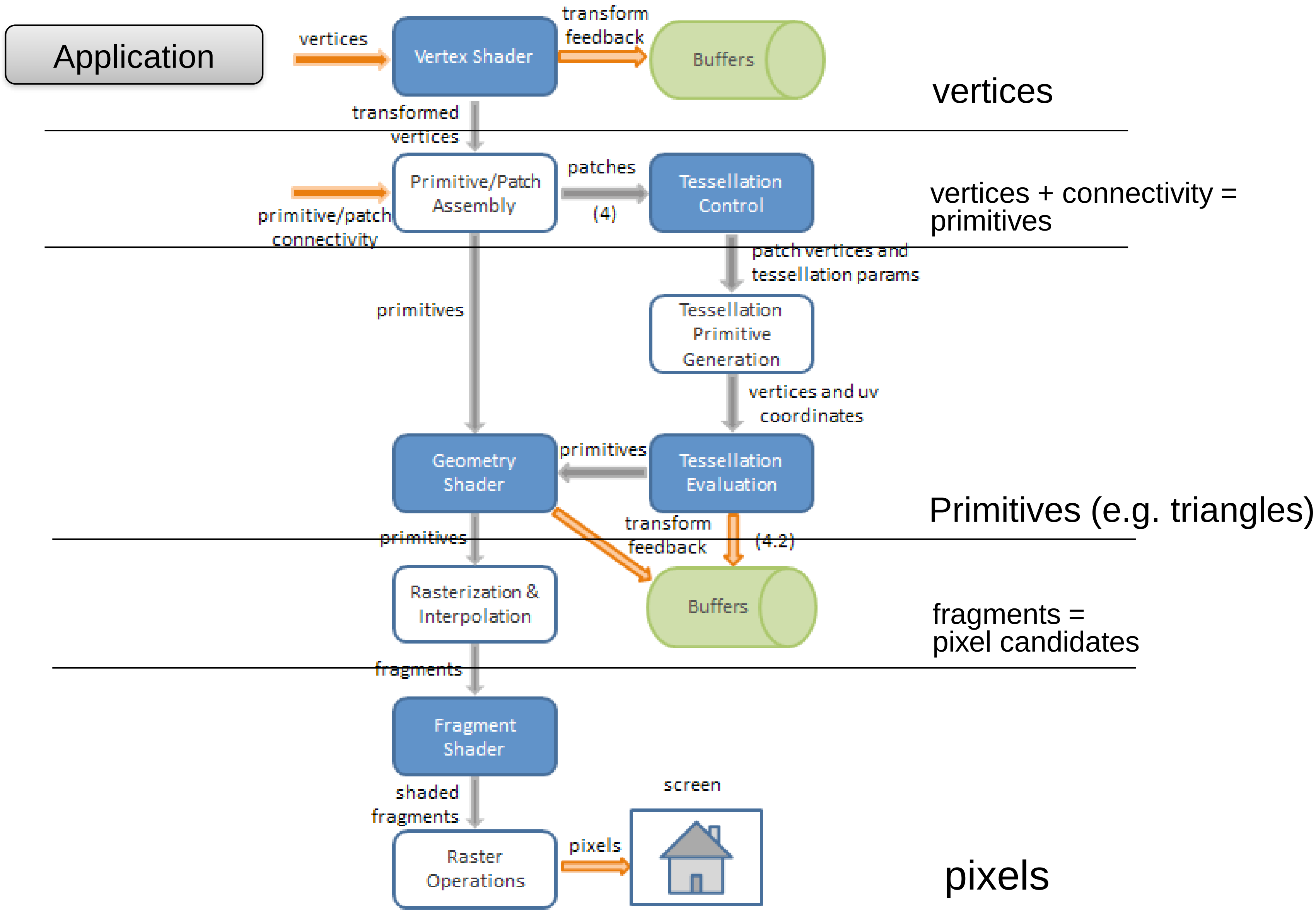
geometry
stage

rasterization
stage

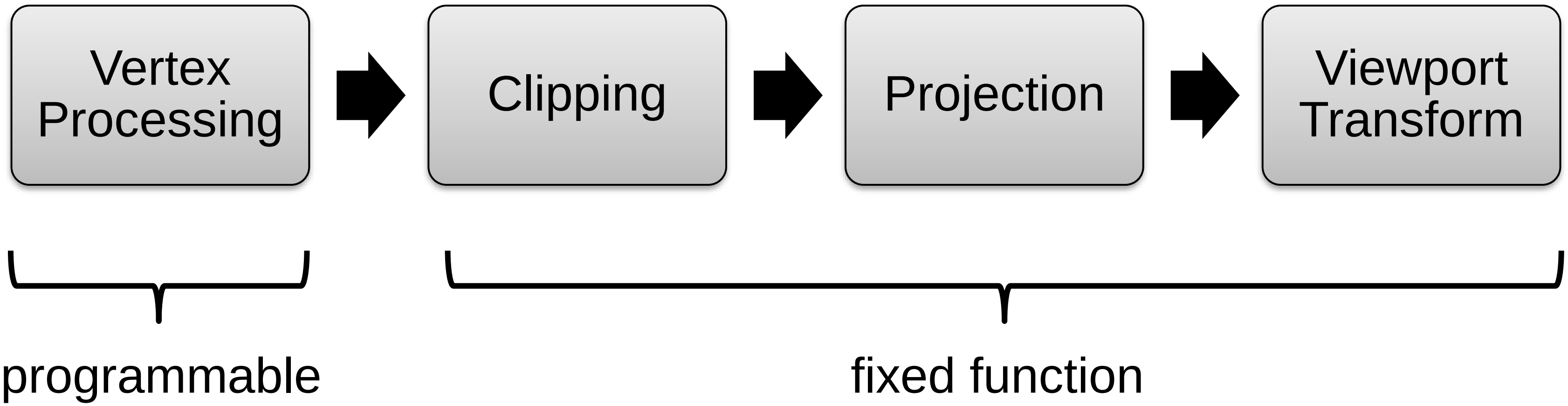


Source:
www.lighthouse3d.com

The Graphics Pipeline: OpenGL 3.2 and later

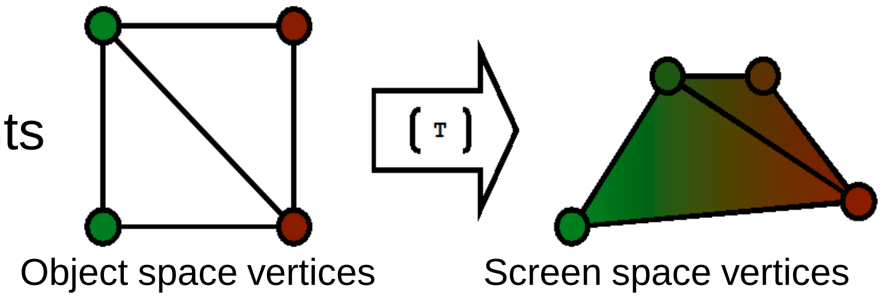


Geometry Stage



Geometry Stage: Vertex Processing

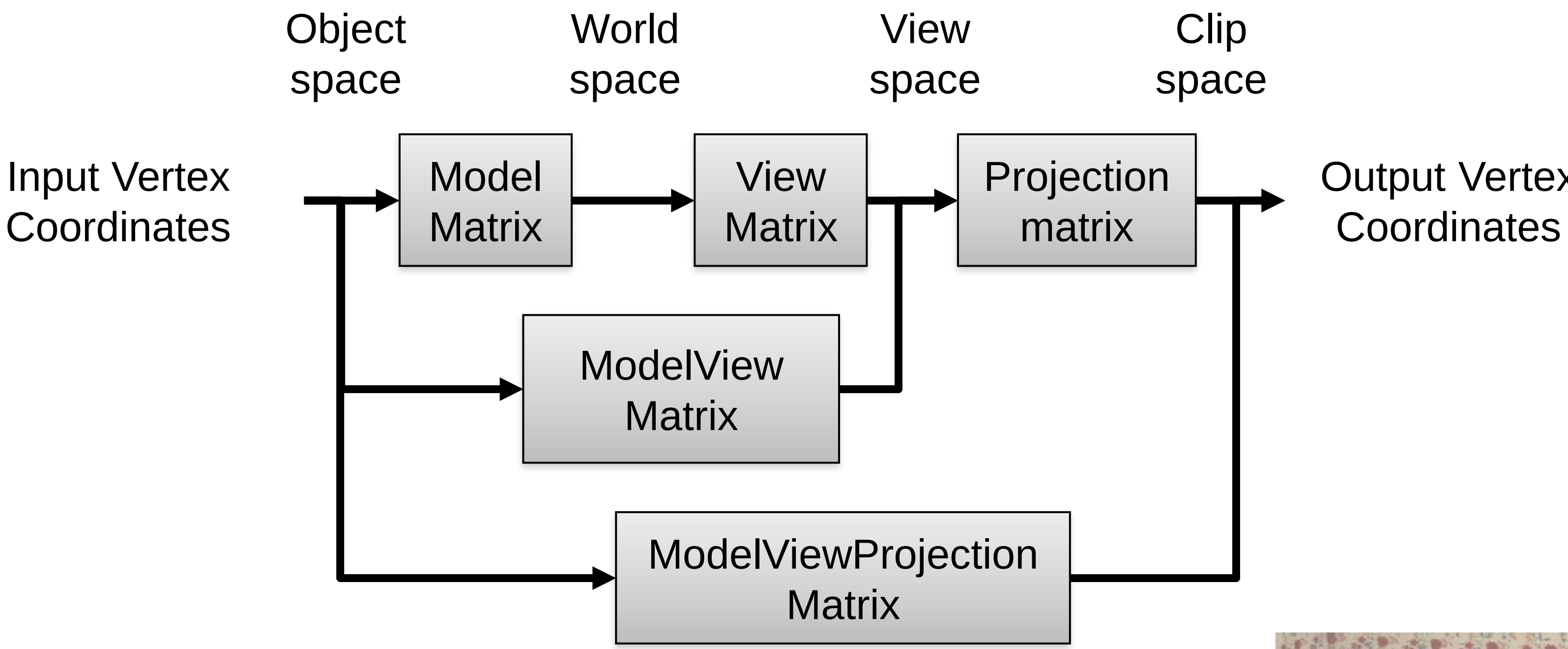
- The input vertex stream
  - composed of arbitrary vertex attributes (position, color, …).
- is transformed into stream of vertices mapped onto the screen
  - composed of their clip space coordinates and additional user-defined attributes (color, texture coordinates, …).
  - clip space: homogeneous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



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

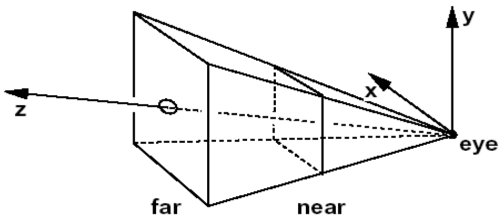
Geometry Stage: Vertex Post-Processing

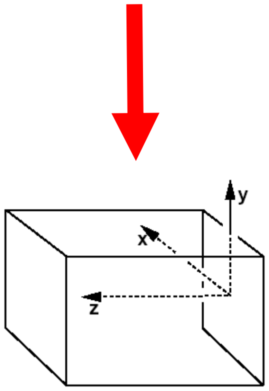
- Uses a common transformation model in rasterization-based 3D graphics:

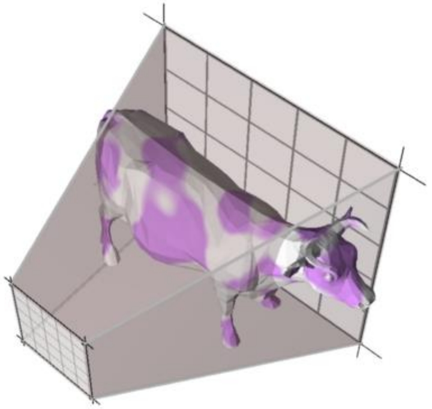


Geometry Stage: Vertex Post-Processing

- 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



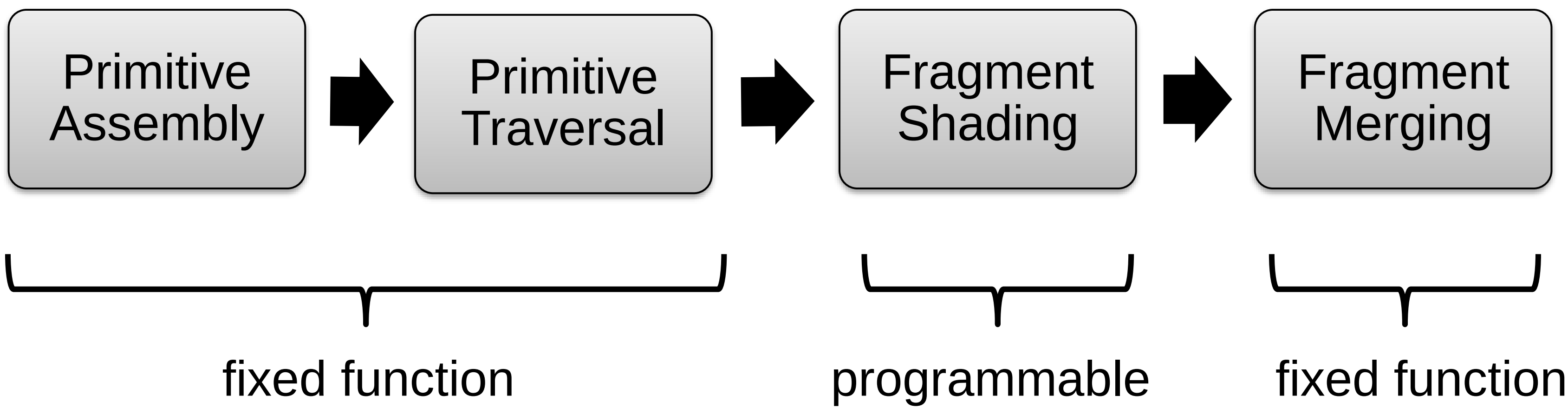




Geometry Shader

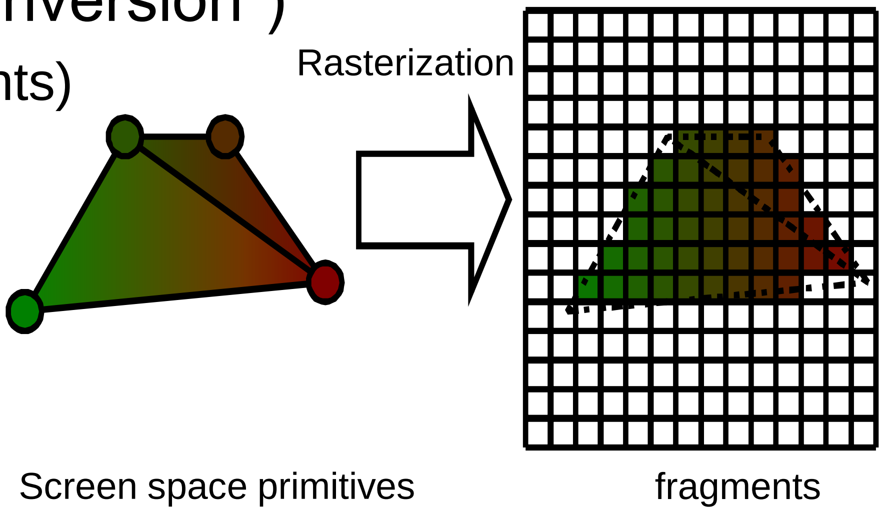
- Optional stage between vertex and fragment shader
- In contrast to the vertex shader, the geometry shader has full knowledge of the primitive it is working on
  - For each input primitive, the geometry shader has access to all the vertices that make up the primitive, including adjacency information.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

Rasterization Stage

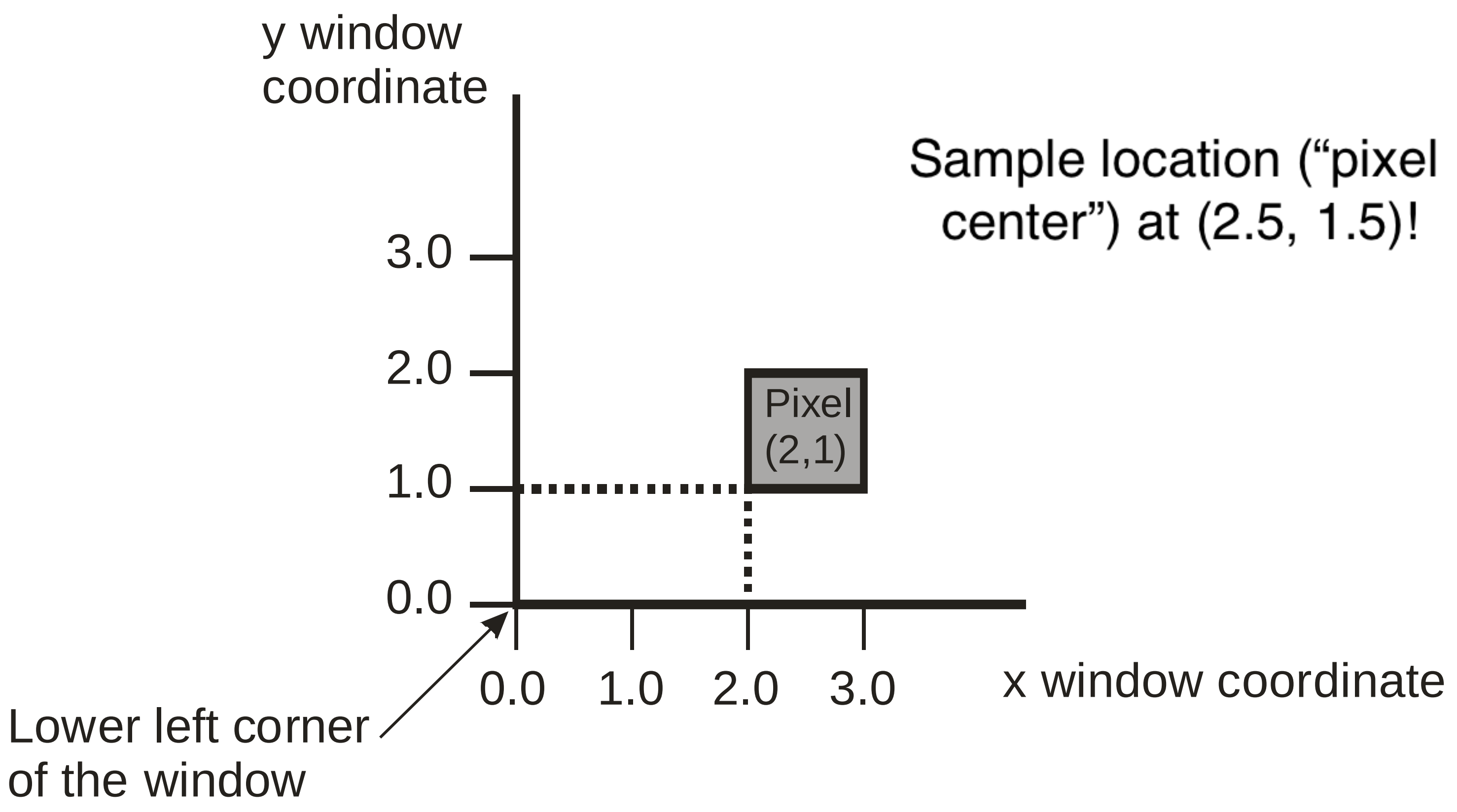


Rasterization Stage

- 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, …

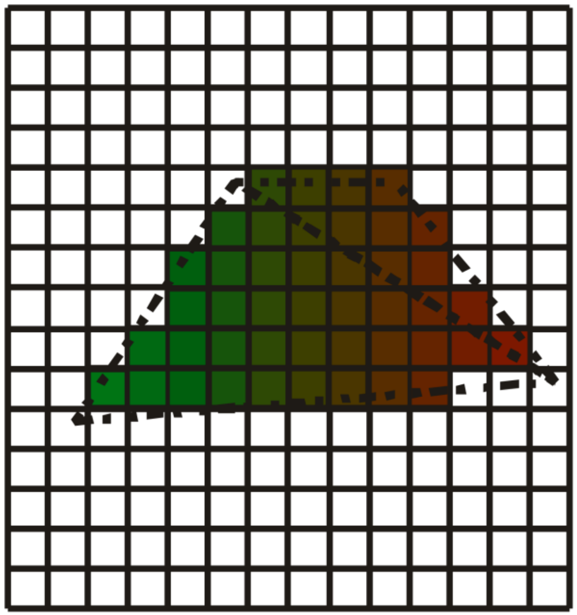


Rasterization – Coordinates



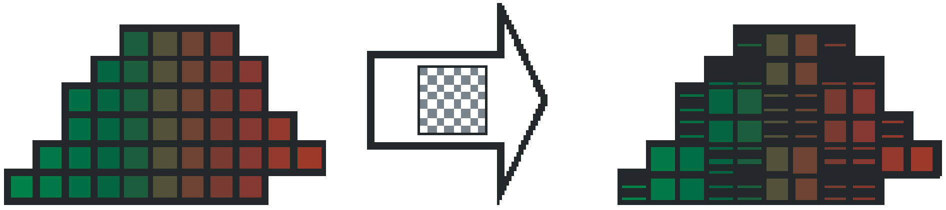
Rasterization – Rules

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



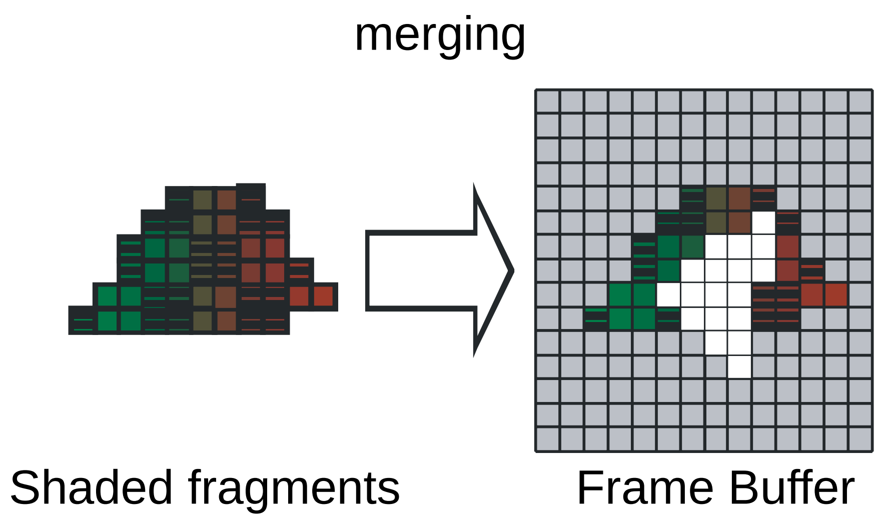
Fragment Shading

- "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
  - …

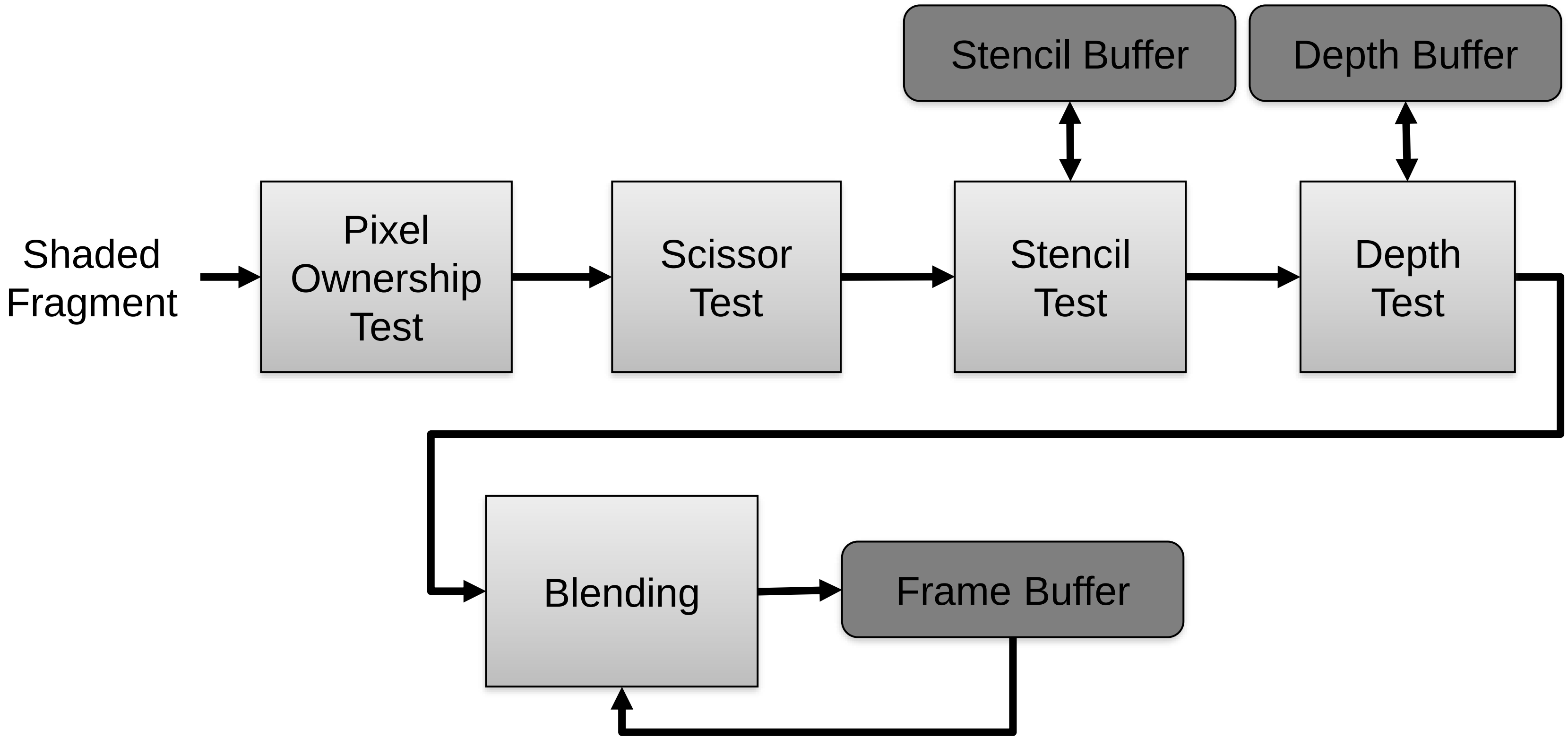


Fragment Merging

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

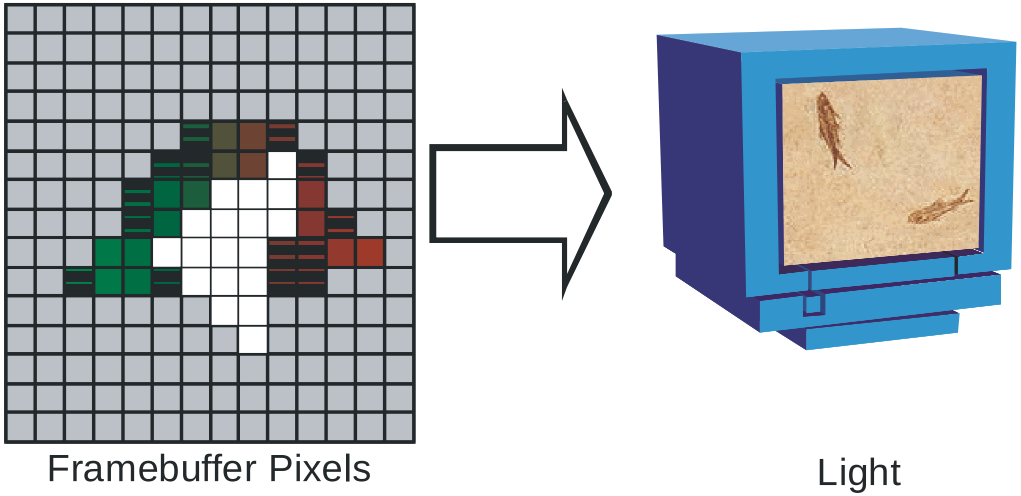


Fragment Merging



Display Stage

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



Display Format

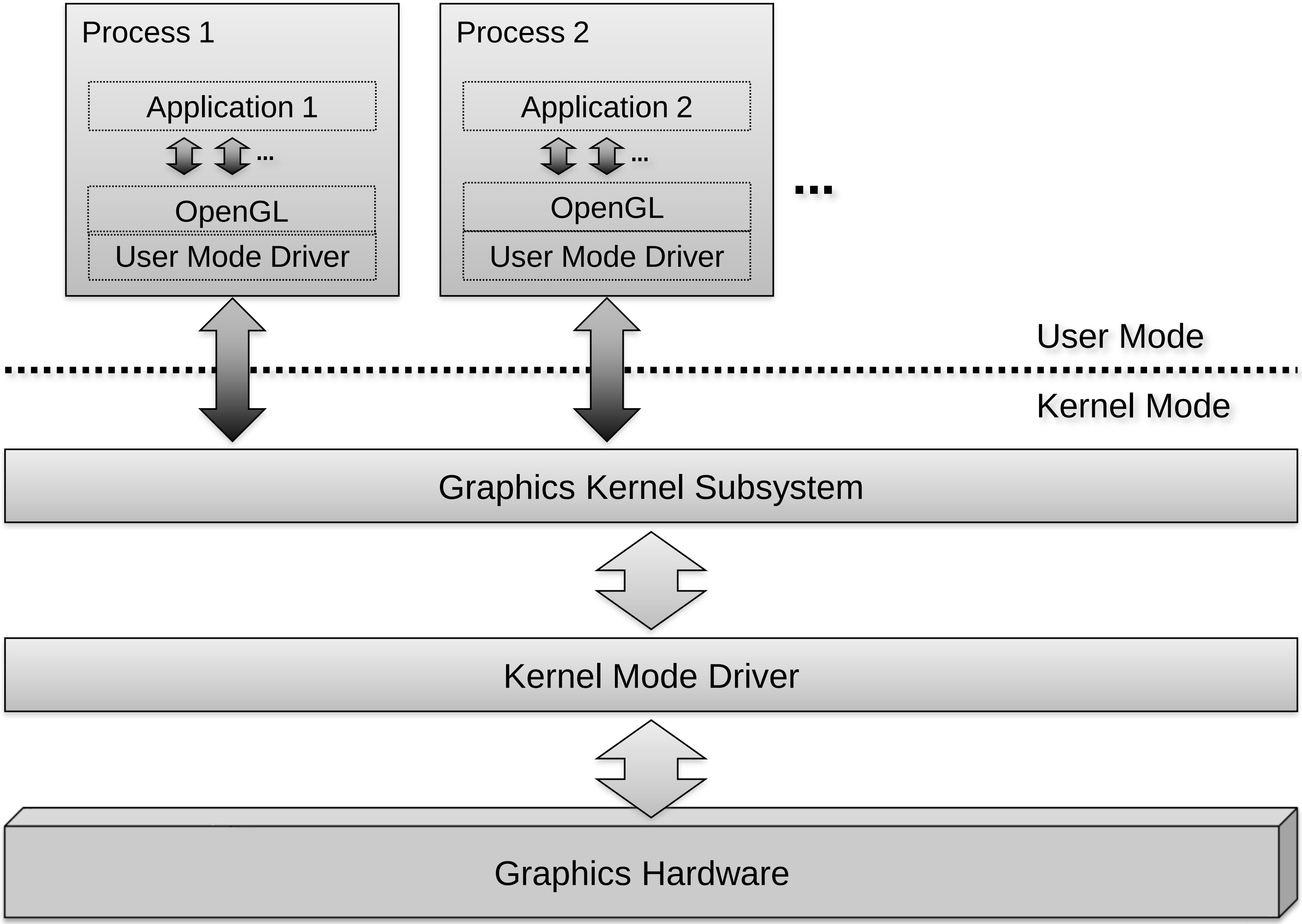
- Frame buffer pixel format:
  RGBA vs. index (obsolete)
- Bits: 16, 32, 64, 128 bit floating point, …
- Double buffer vs. single buffer
- Quad-buffered stereo
- Overlays (extra bitplanes)
- Auxilliary buffers: alpha, stencil

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

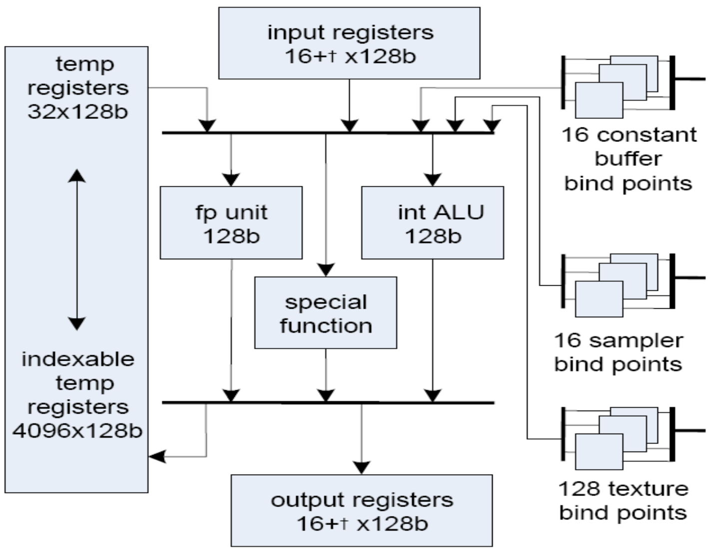
- Graphics Hardware is a shared resource
- User Mode Driver (UMD)
  - Prepares Command Buffers for the hardware
- Graphics Kernel Subsystem
  - Schedules access to the hardware
- Kernel Mode Driver (KMD)
  - Submits Command Buffers to the hardware



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

Unified Shader Model

- Since shader Model 4.0
- Unified Arithmetic and Logic Unit (ALU)
- Same instruction set and capabilities for all Shader types
- Dynamic load balancing geometry/fragment
- Floating point or integer everywhere
- IEEE-754 compliant
- Geometry Shader can write to memory
  - „Stream Output“
  - Enables multi-pass for geometry



Graphics APIs

Low-level 3D API

- OpenGL
  - Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.
- OpenGL ES
  - OpenGL for Embedded Systems is a subset of OpenGL
- DirectX, Direct3D
  - a graphics API for Microsoft Windows

Graphics APIs cont.

- Vulcan
  - OpenGL successor
  - targets high-performance realtime 3D graphics applications across all platforms
  - offers higher performance and lower CPU usage than older APIs.
- Mantle
  - low level graphics API by AMD. AMD will move to Vulcan
- Metal
  - low-level, low-overhead hardware-accelerated graphics and compute API by Apple (since IOS 8)

Graphics APIs cont.

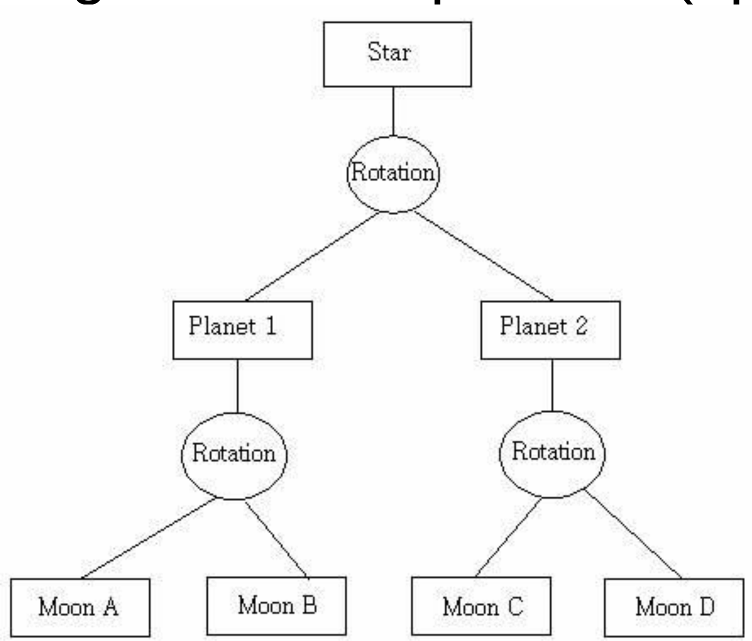
- RenderMan
  - Interface Specification by Pixar Animation Studios
  - open API
  - describe three-dimensional scenes and turn them into digital photorealistic images.
  - It includes the RenderMan Shading Language.
- WebGL
  - JavaScript API for rendering interactive 3D computer graphics and 2D graphics within any compatible web browser without the use of plug-ins.

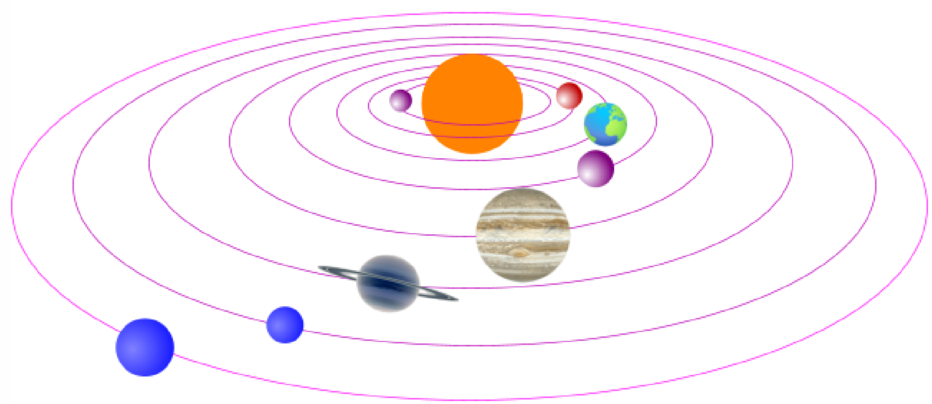
Graphics APIs cont.

High-level 3D API – declarative models

a lot! Java, SceneGraphs, performer, Irrlicht, mobile SDKs

e.g. SceneGraph APIs (openSG, openInventor, etc.)





http://archive.gamedev.net/archive/reference/programming/features/scenegraph/index.html