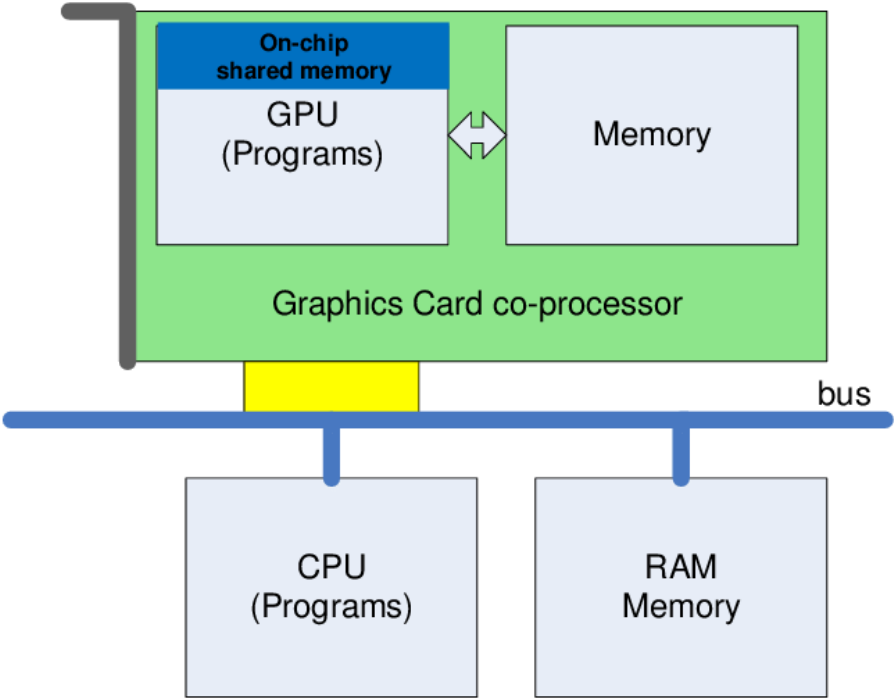
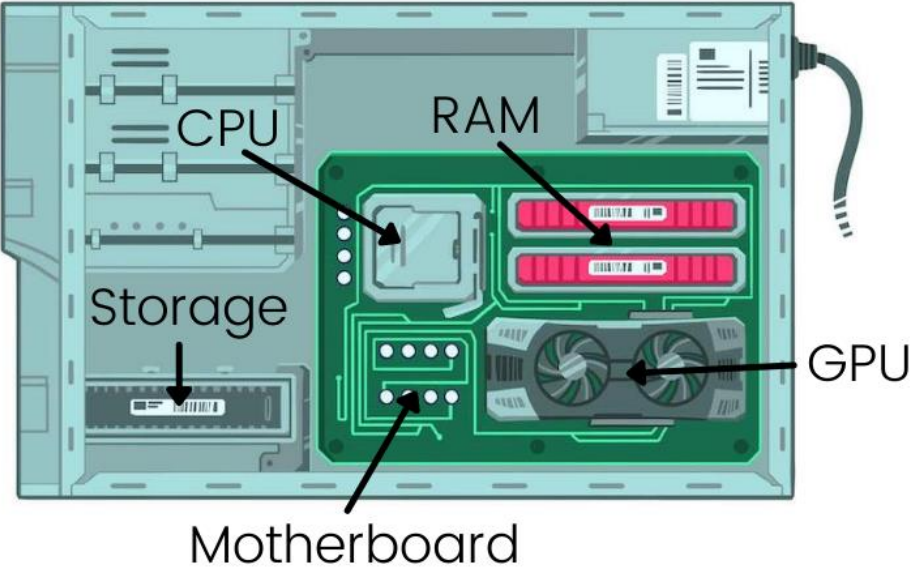


CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

2.1.1 Graphics Pipeline

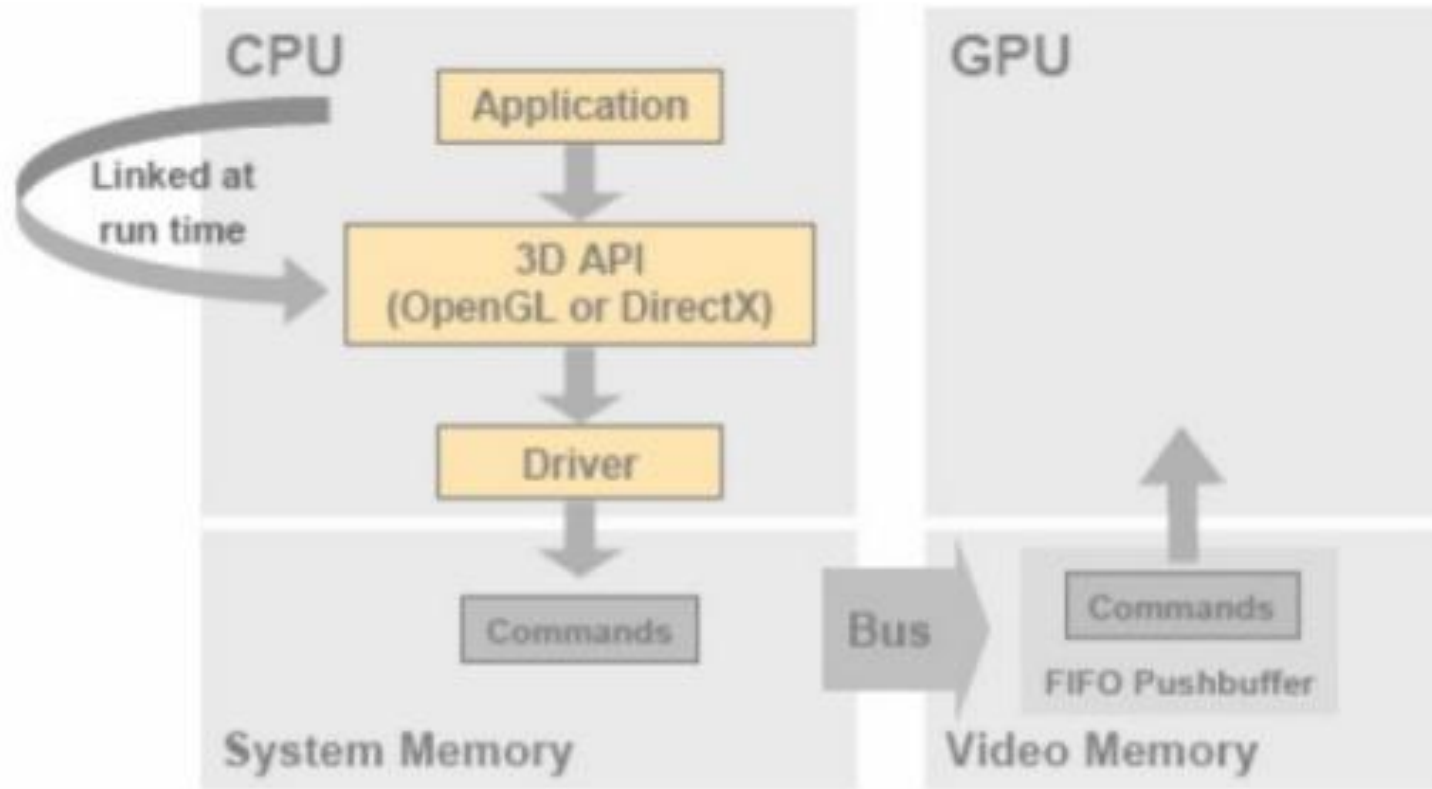
Graphics Pipeline Hardware

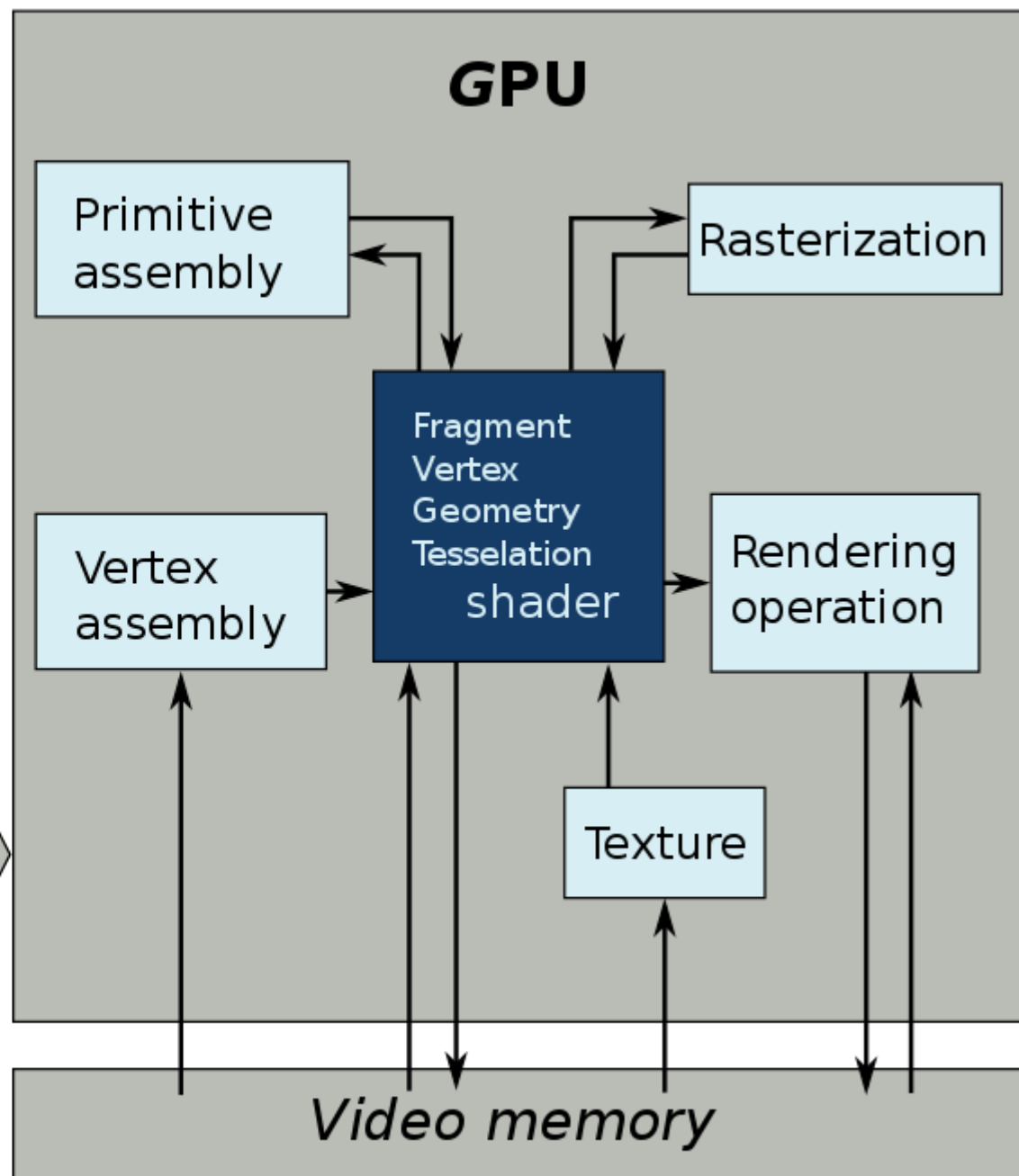
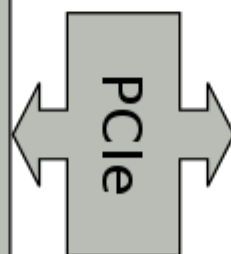
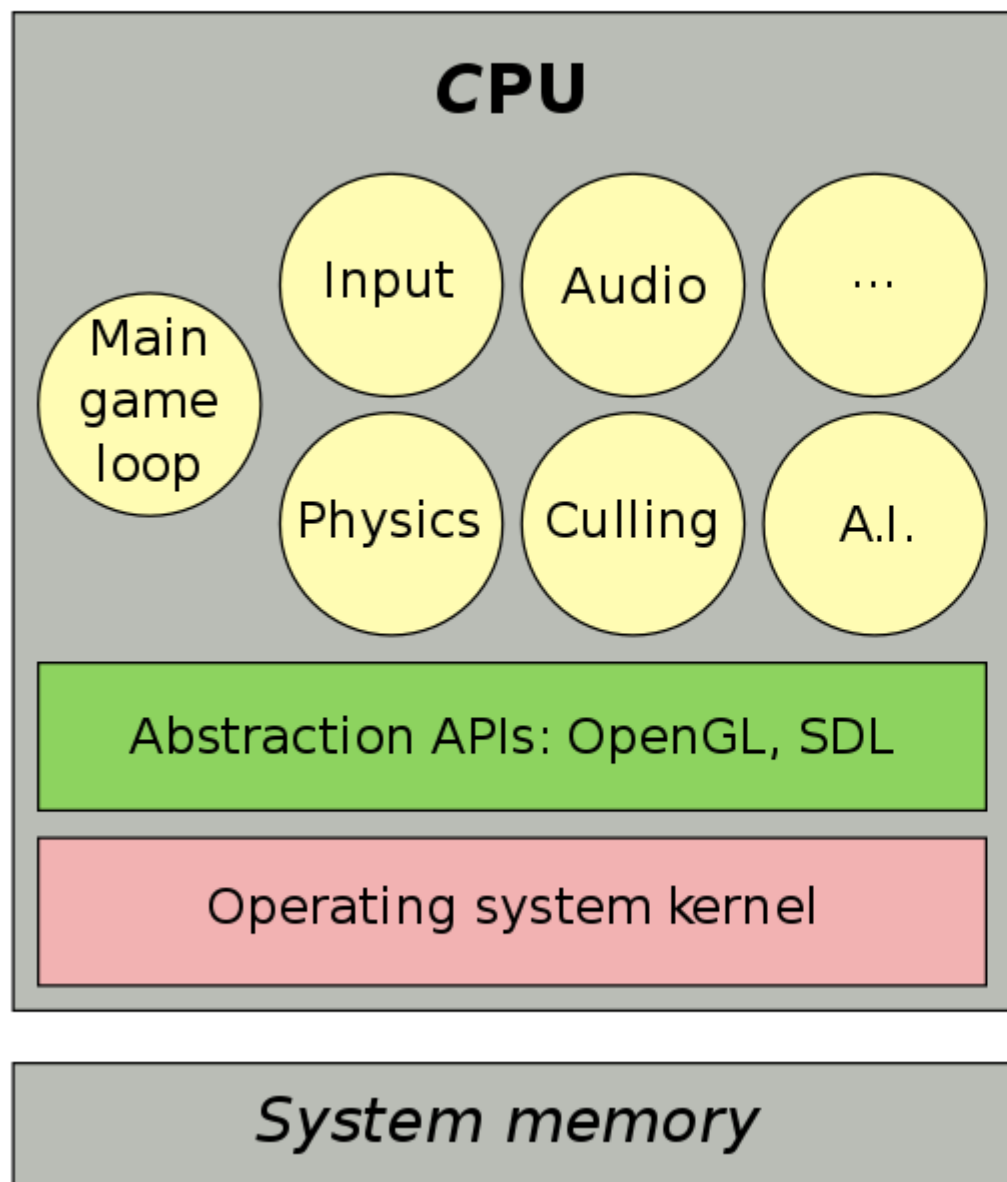




Programming Graphics Applications

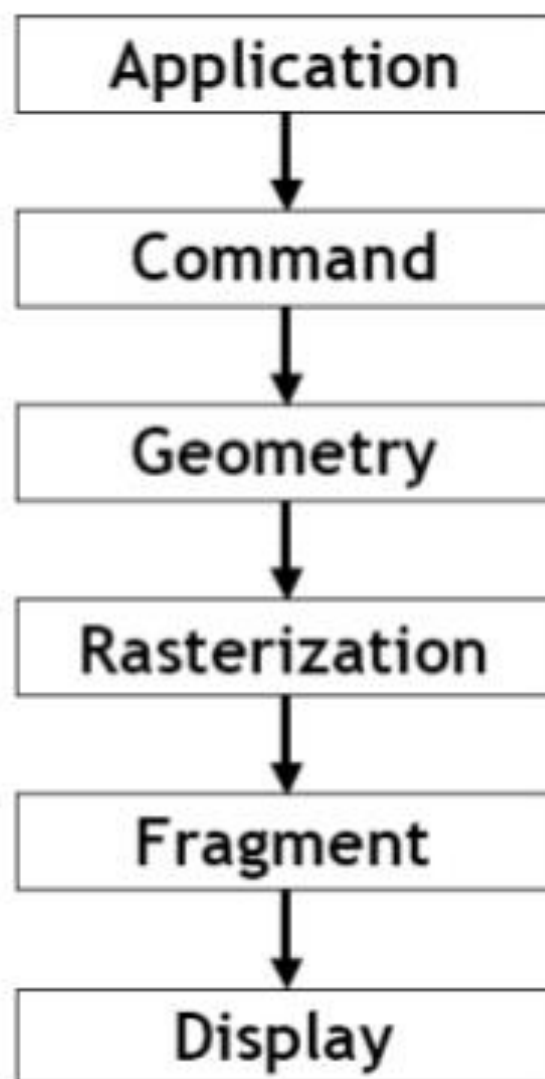
- The application is programmed to the 3D graphics API and links with the driver at runtime





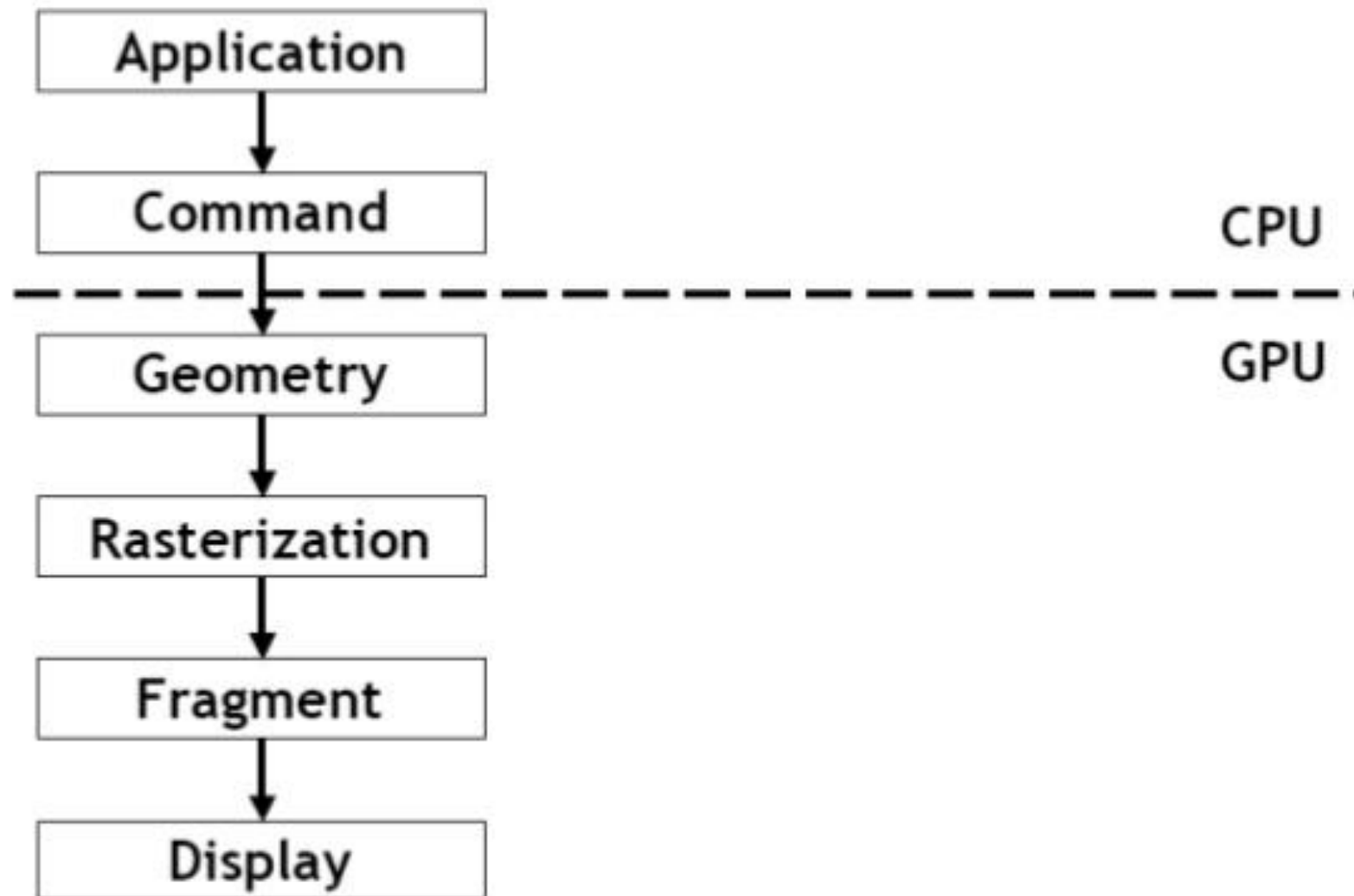


The Graphics Pipeline



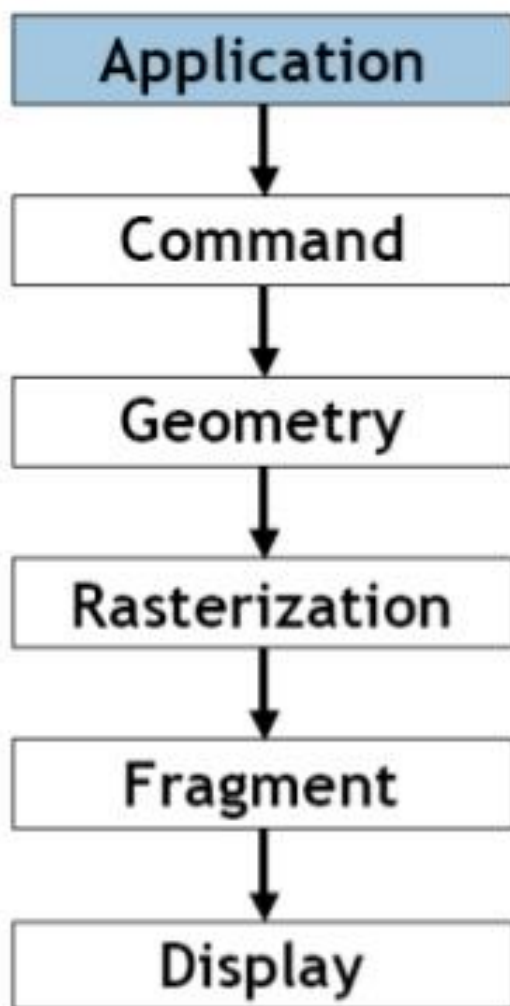


The Graphics Pipeline





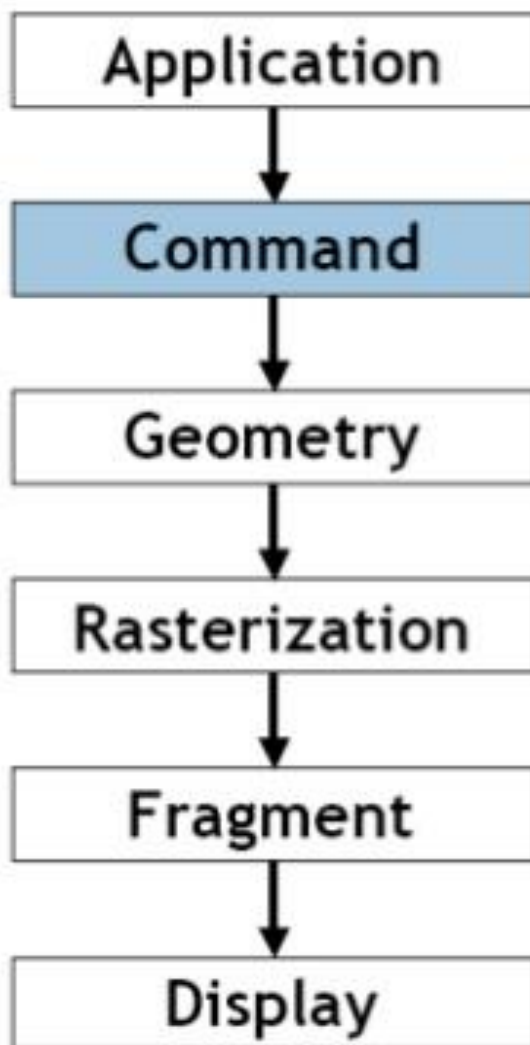
The Graphics Pipeline



- ☐ SolidWorks:
- ☐ define software behavior
- ☐ user input events
- ☐ Software logic
- ☐ CAD operations



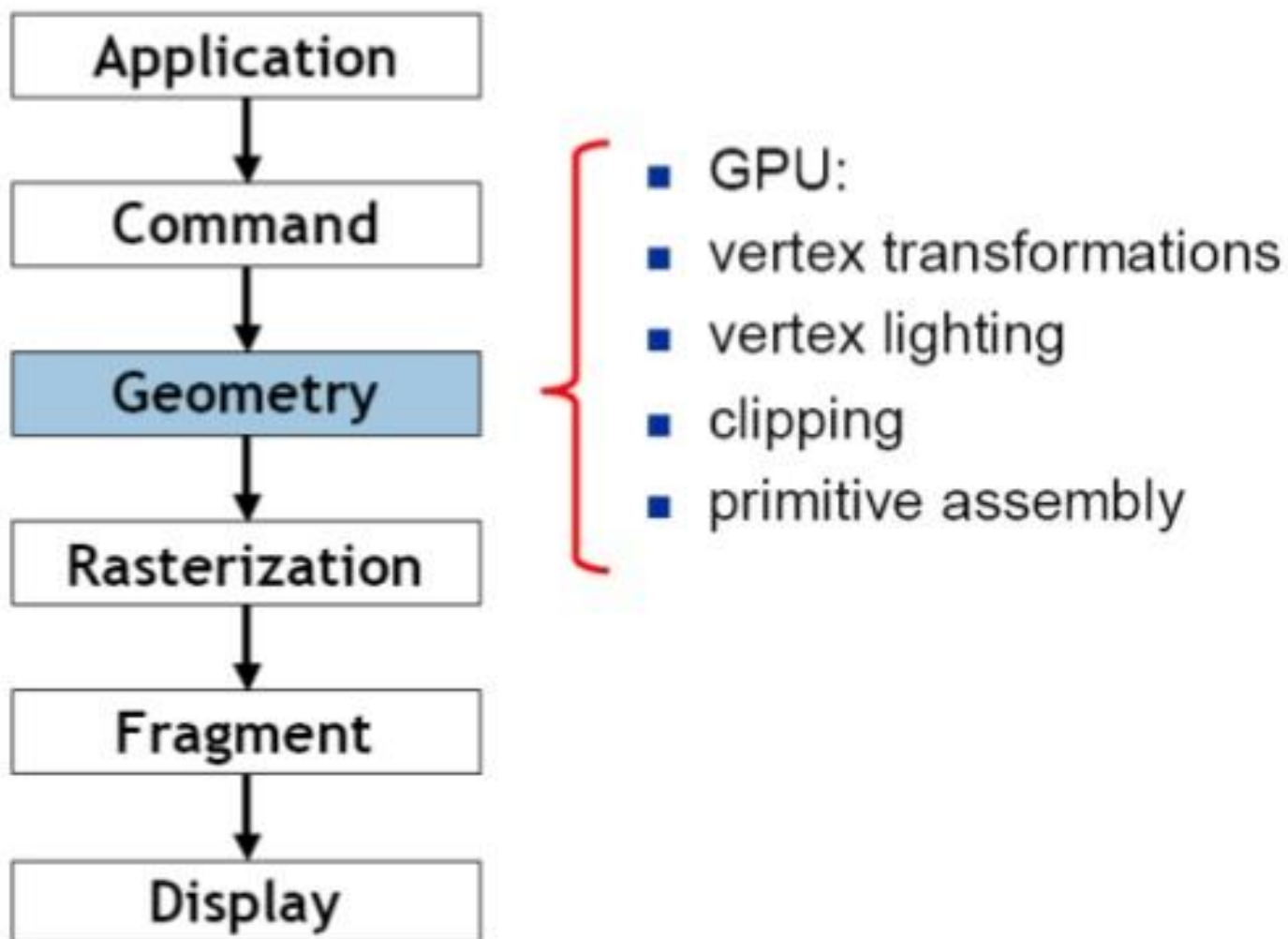
The Graphics Pipeline



- ☐ SolidWorks:
- ☐ send OpenGL commands
- ☐ OpenGL driver:
- ☐ process GL command stream
- ☐ talk to GPU

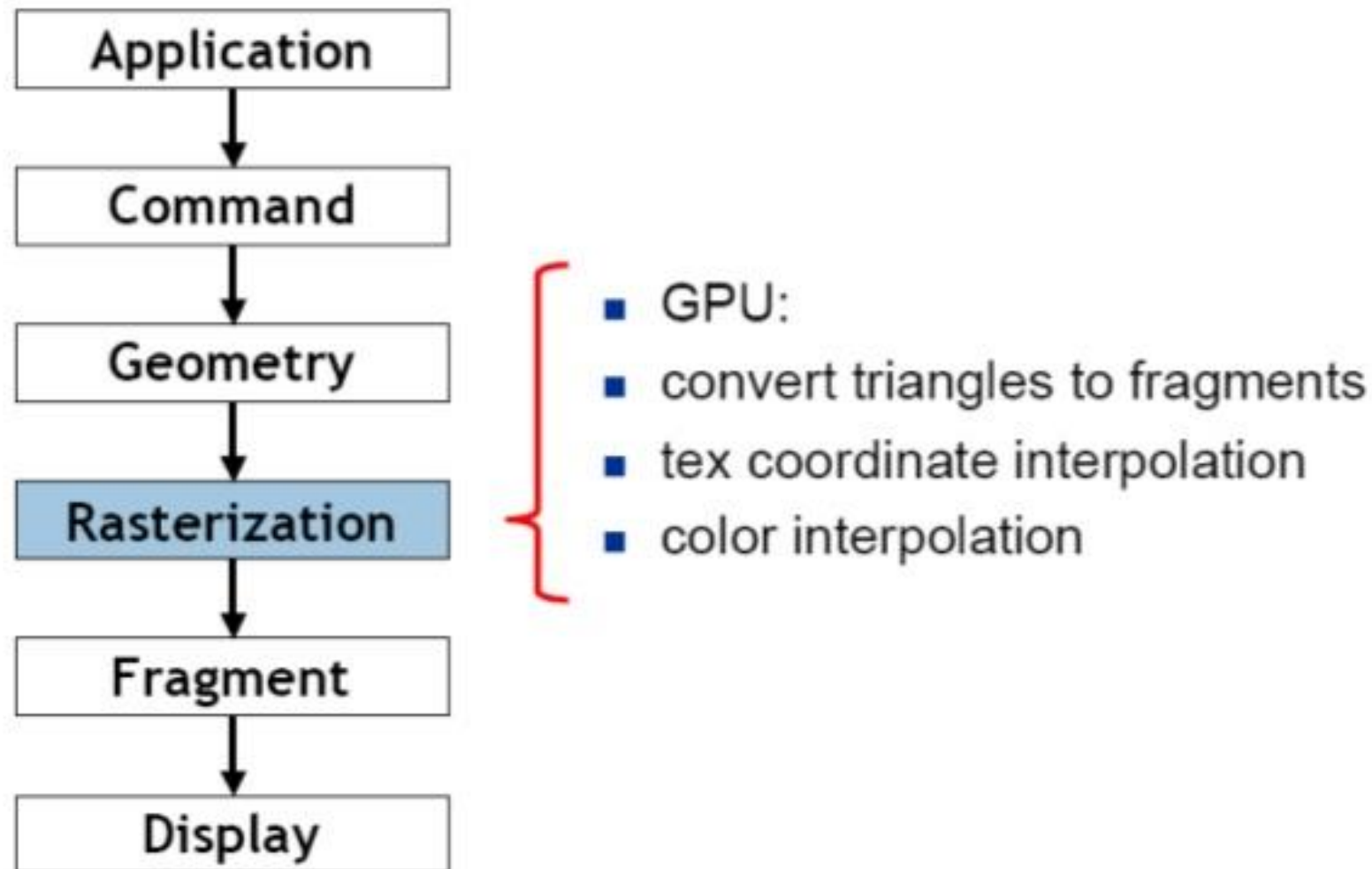


The Graphics Pipeline



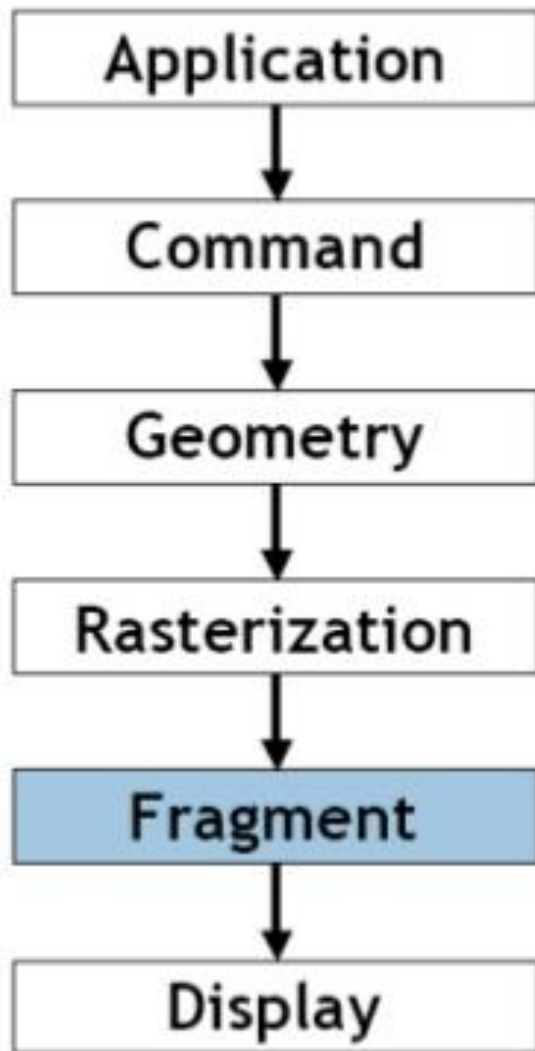


The Graphics Pipeline





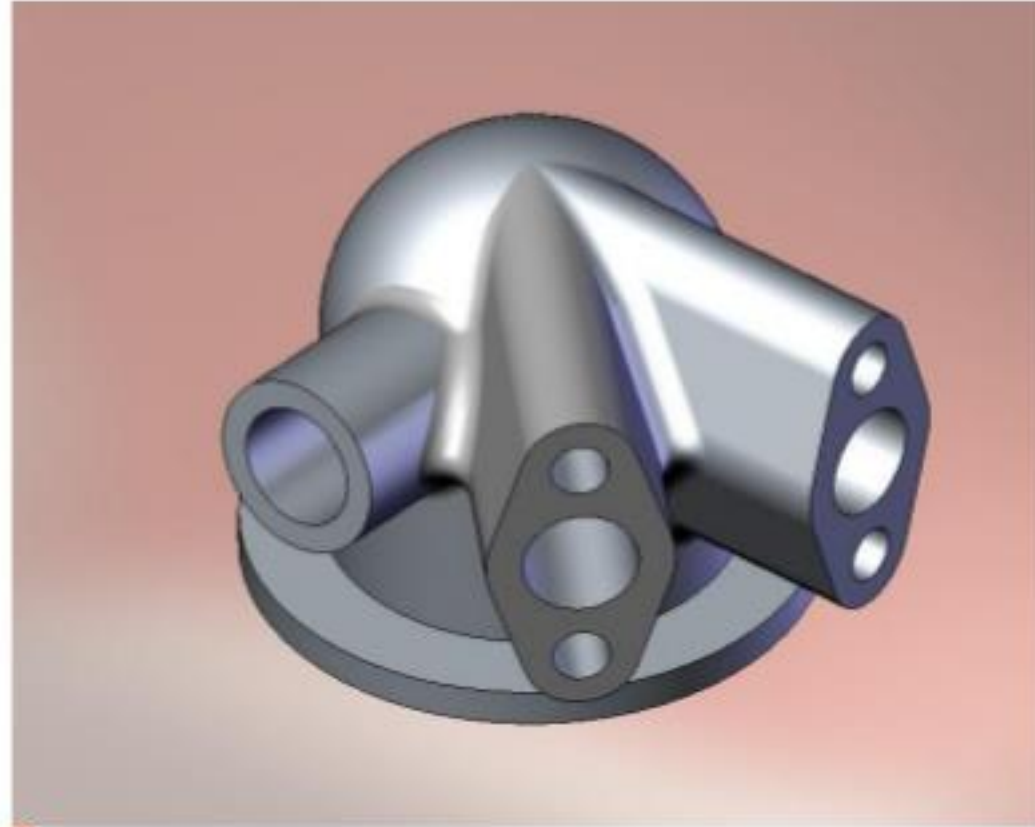
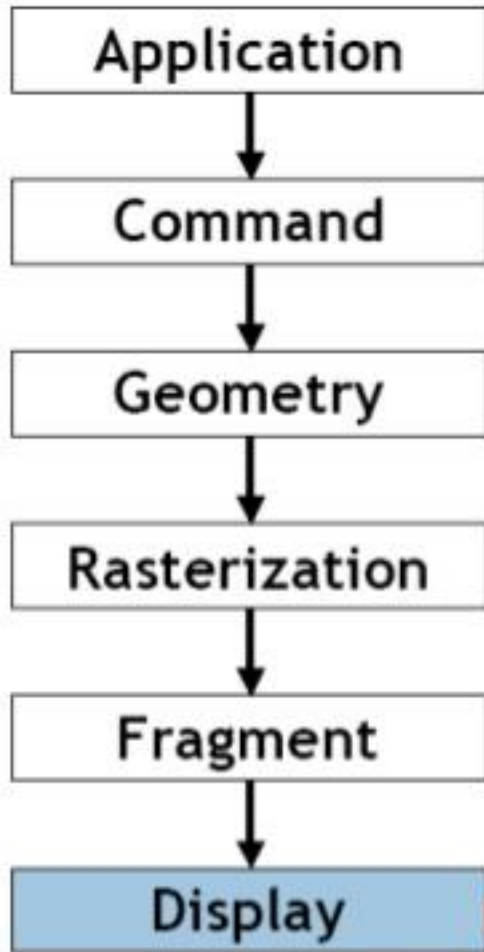
The Graphics Pipeline



- GPU:
- texturing
- depth test
- color blending



The Graphics Pipeline



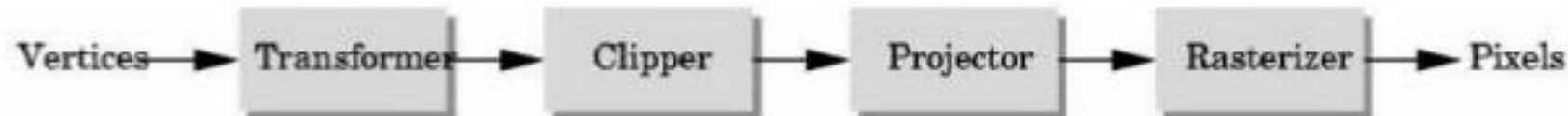
2.1.2 Programming a Pipeline



Programming a Pipeline



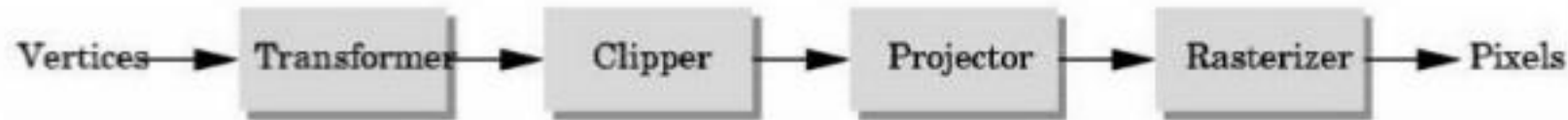
- Specify the operation of each box
- Replace or accumulate
- State and lack of modularity
- Immediate mode graphics
 - On-line (OpenGL)
- Modeling-rendering pipeline
 - Off-line (Pixar's Renderman)



- Vertices in **world coordinates**
- `void glVertex3f(GLfloat x, GLfloat y, GLfloat z)`
 - Vertex (x, y, z) sent down the pipeline
 - Function call returns
- Use *GLtype* for portability and consistency
 - `glVertex{234}{sfid}[v](TYPE coords)`



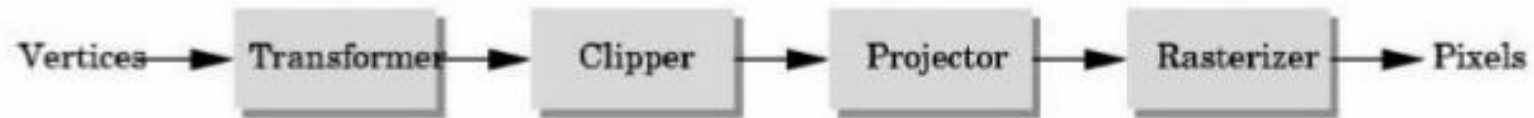
Transformer



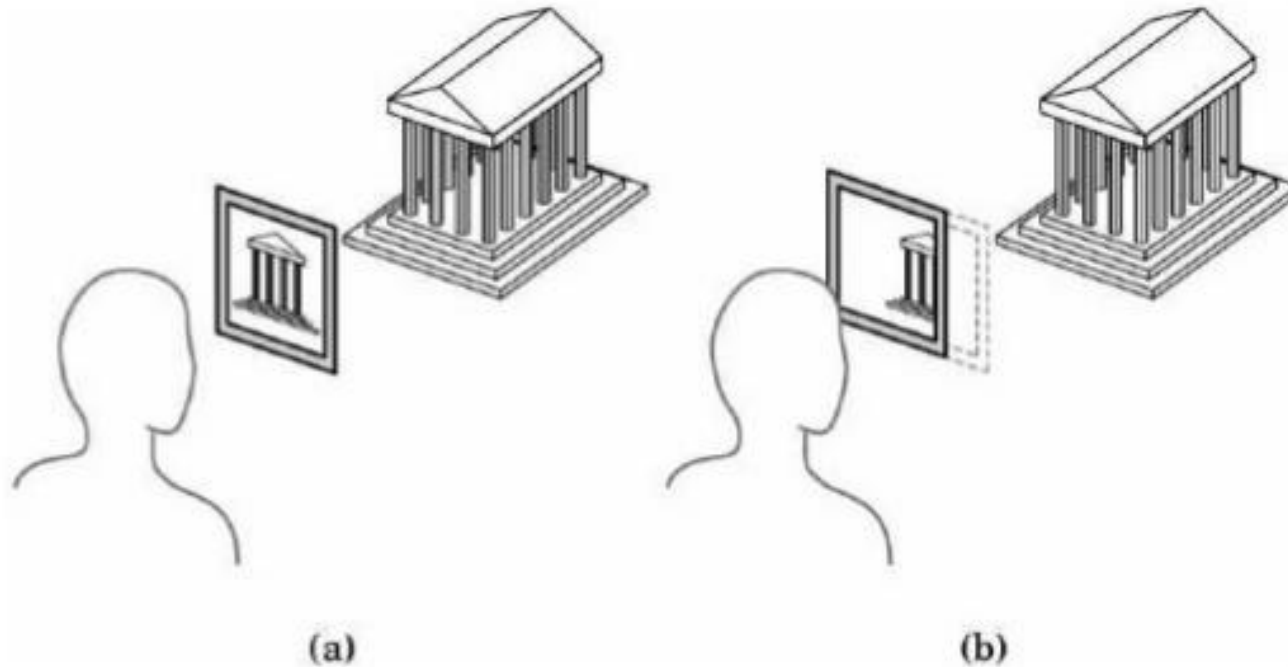
- Transformer in **world coordinates**
- Must be set **before** object is drawn!
 - `glRotatef(45.0, 0.0, 0.0, -1.0);`
 - `glVertex2f(1.0, 0.0);`



Clipper

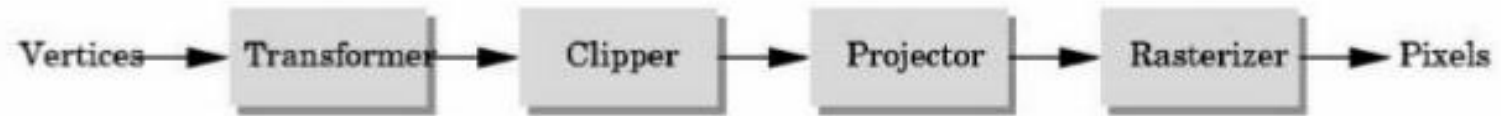


- Mostly automatic from viewport

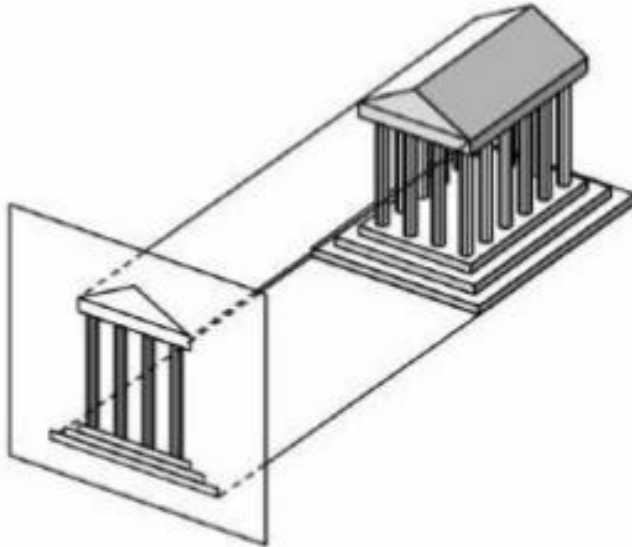




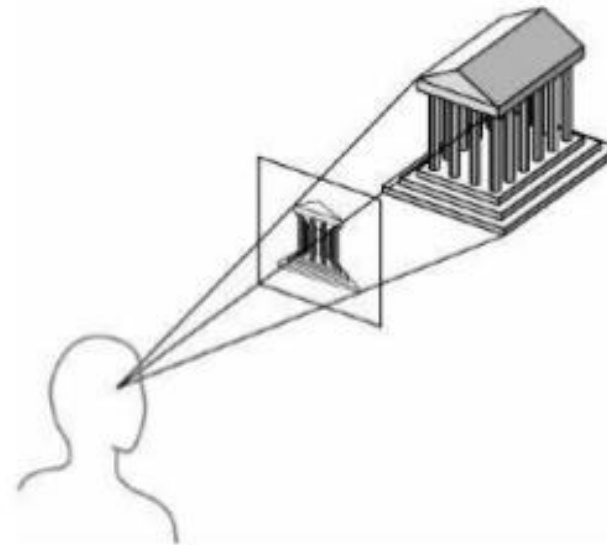
Projector



Orthographic

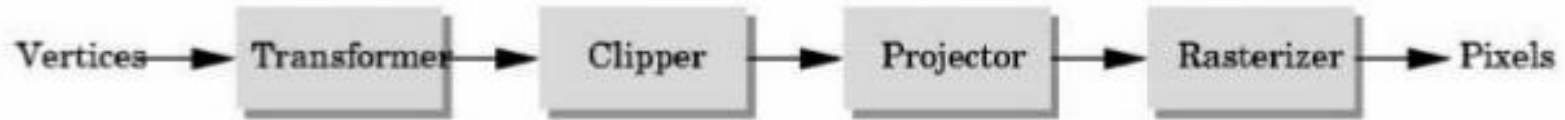


Perspective

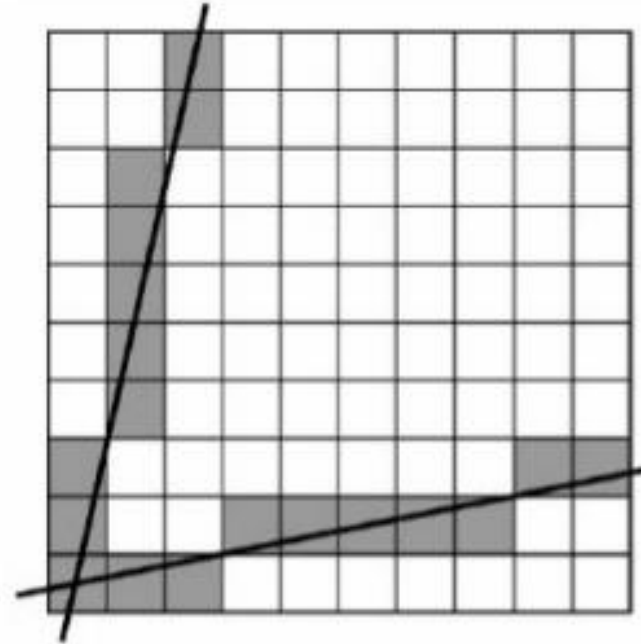




Rasterizer



- Interesting algorithms
- To window coordinates



2.1.2 Programming 2D with OpenGL

Why Learn About OpenGL?

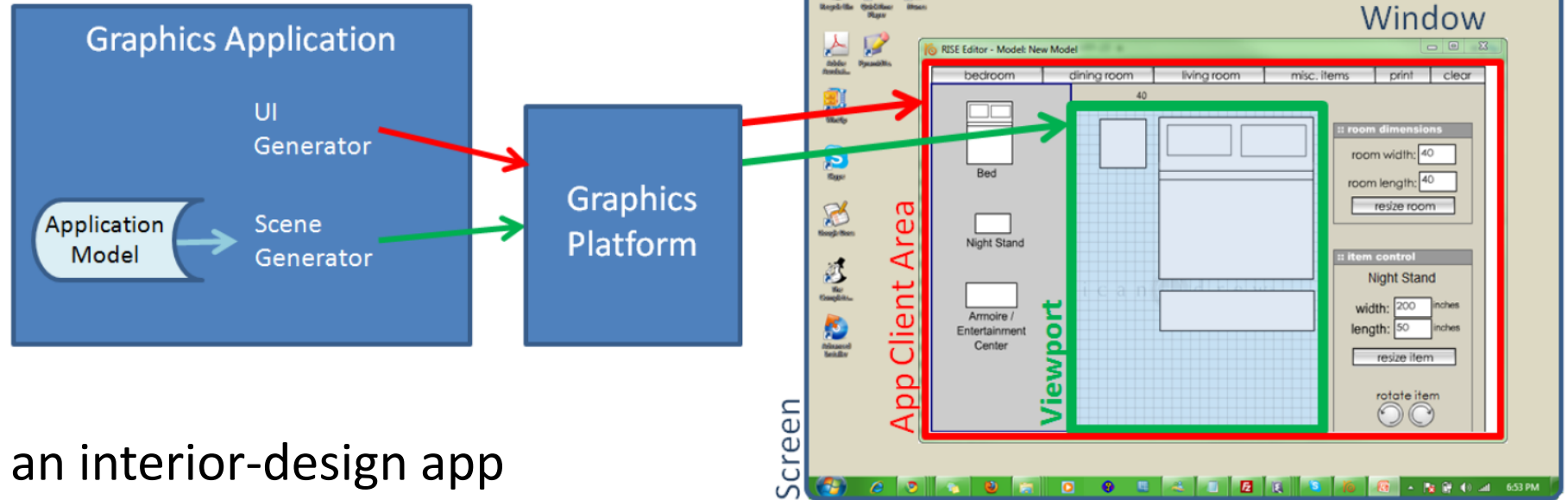
- A well-known industry standard for real-time 2D and 3D computer graphics
- Available on most platforms
 - Desktop operating systems, mobile devices (OpenGL ES* , e.g., Android), browsers (WebGL)
- Older (OpenGL 1.0) API provided features for rapid prototyping; newer API (OpenGL 2.0 and newer) provides more flexibility and control
 - Many old features exist in new API as “deprecated” functionality, supported only for backwards-compatibility with legacy apps
 - We will use the new API exclusively

* ES is for “Embedded Systems”

Why Learn 2D first?

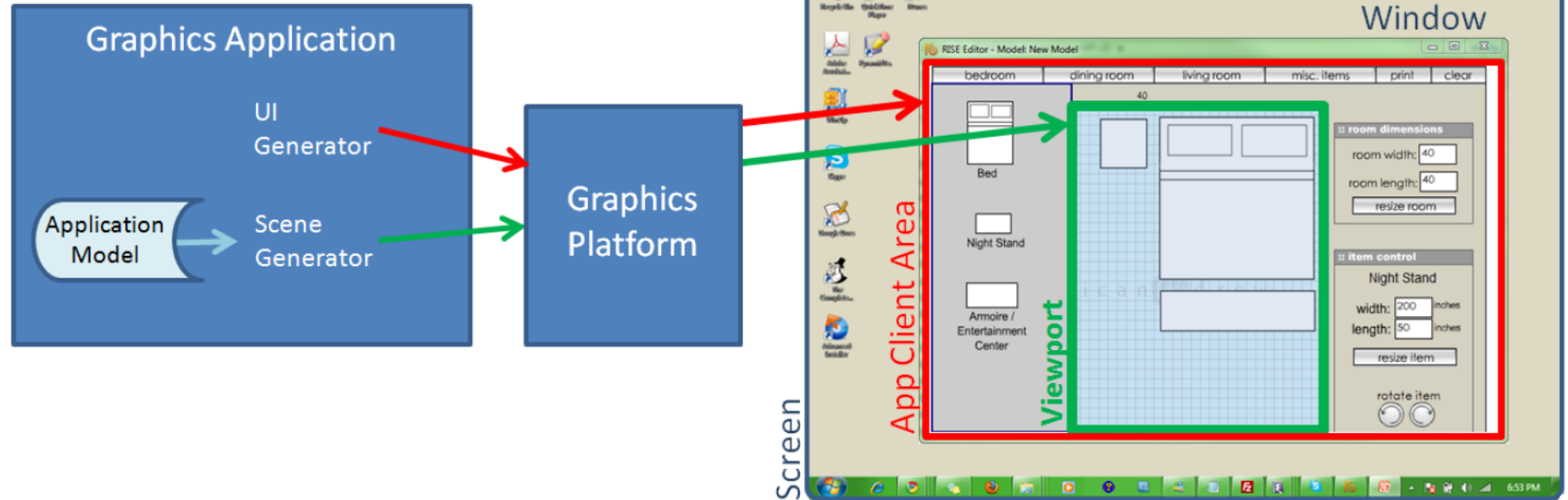
- A good stepping stone towards 3D – many issues much easier to understand in 2D
 - no need to simulate lights, cameras, the physics of light interacting with objects, etc.
 - intro to modeling vs. rendering and other notions
 - get used to rapid prototyping in OpenGL, both of designs and concepts
 - 2D is still really important and the most common use of computer graphics, e.g. in UI/UX, documents, browsers
-

Graphics Platforms (1/4)



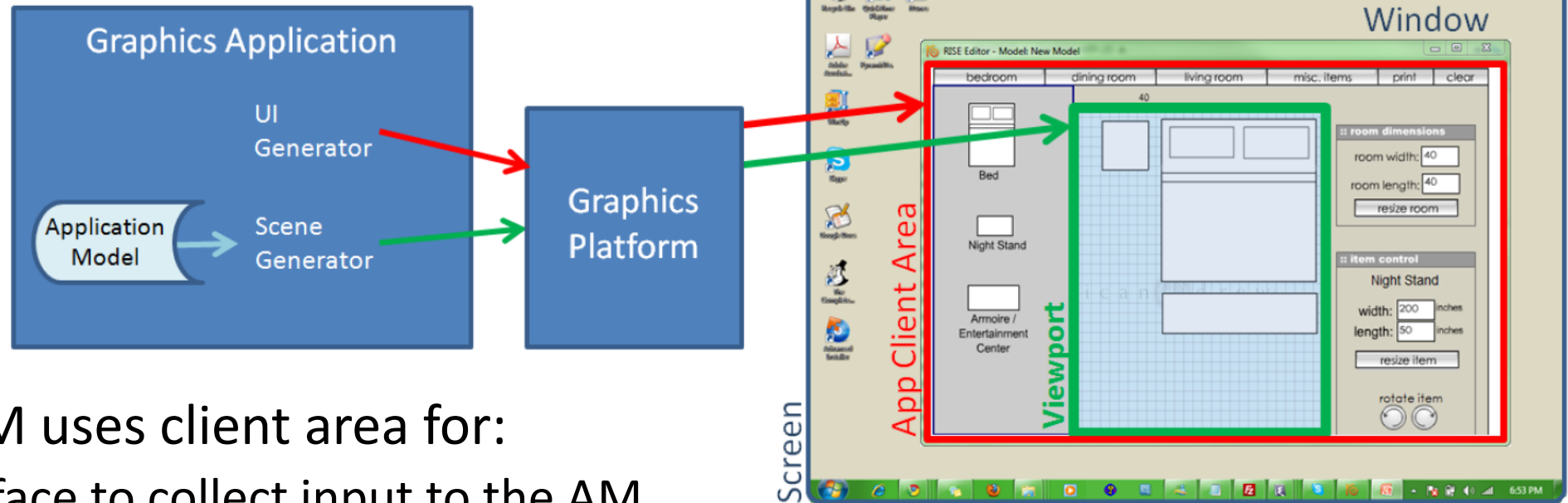
- We're writing an interior-design app
- **Application Model (AM)** is the data being represented by a rendered image
 - manipulated by user interaction with the application
 - typically a hierarchical model, with components built from lower-level components
 - In our application, AM contains positions & size of each bed, dresser, and table

Graphics Platforms (2/4)



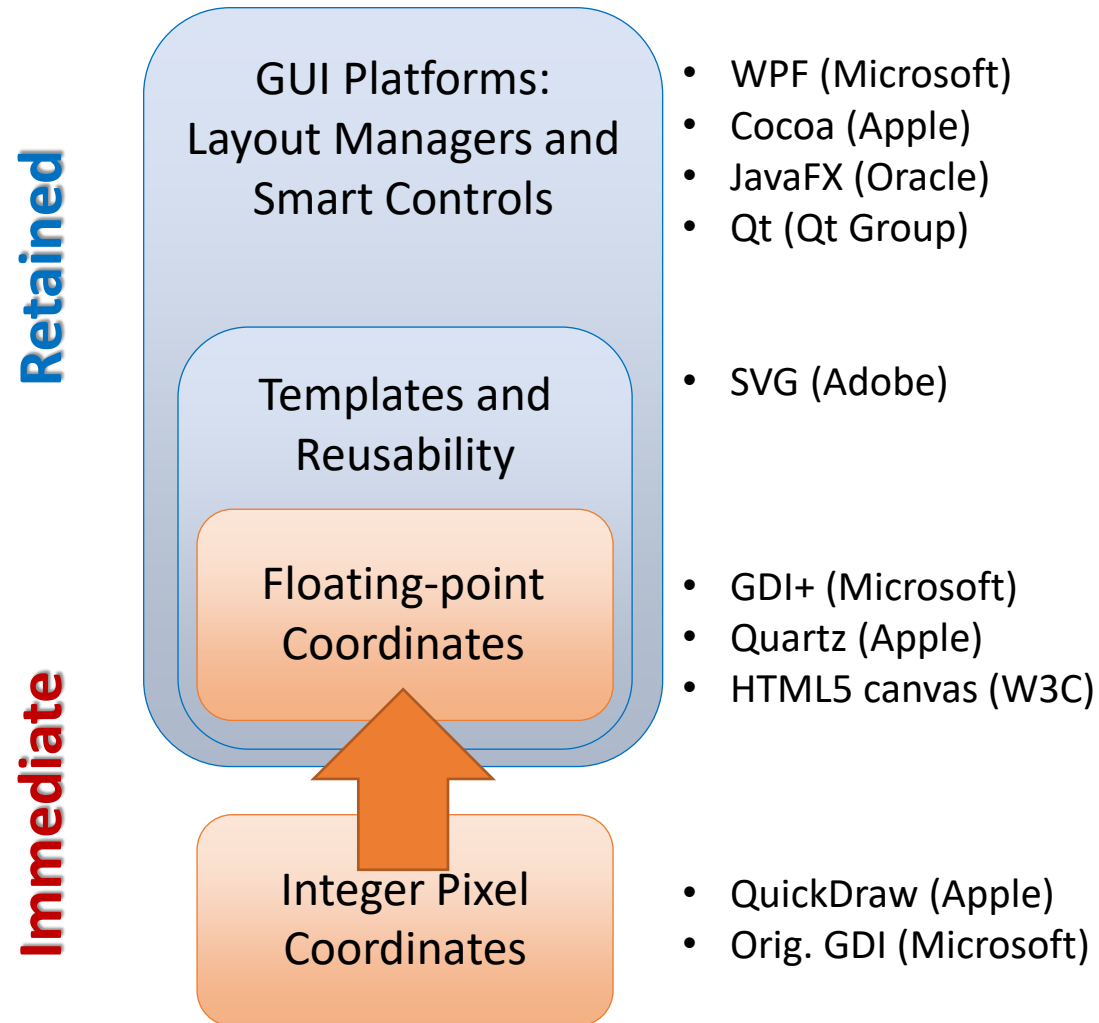
- Graphics Platform runs in conjunction with **window manager**
 - Determines what section of the screen is allocated to the application
 - Handles “chrome” (title bar, resize handles); **client area** is controlled by application

Graphics Platforms (3/4)



- Typically, AM uses client area for:
 - user interface to collect input to the AM
 - display some representation of AM in the **viewport**
 - This is usually called the **scene**, in the context of both 2D and 3D applications
 - Scene is rendered by the **scene generator**, which is typically separate from the **UI generator**, which renders rest of UI

Modern Graphics Platforms (2/2)



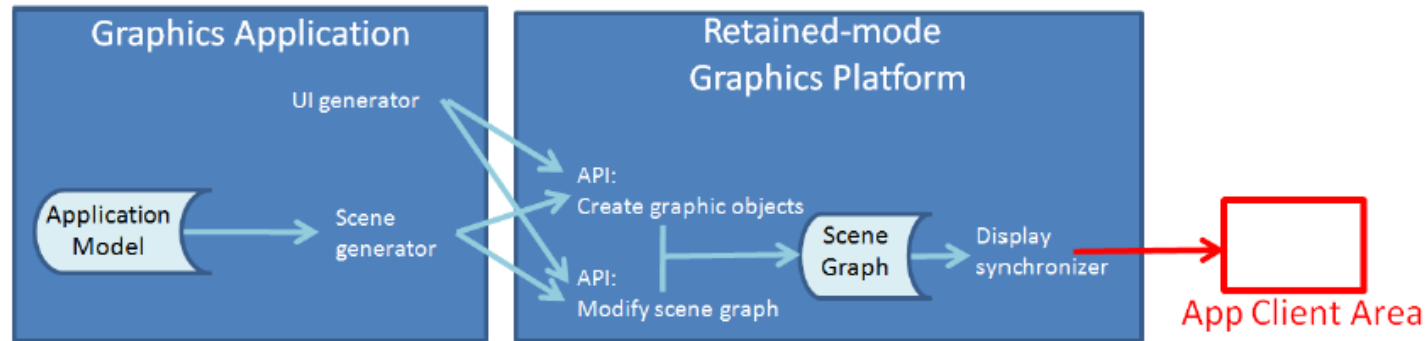
Immediate Mode Vs Retained Mode

Immediate Mode (OpenGL, Vulkan, Microsoft's DirectX, Apple's Metal)

- Driven from **Application model**: as always stores both geometric information and non-geometric information in Application Database
 - Graphics platform keeps no record of primitives that compose scene
 - Vulkan was originally the next version of OpenGL (code name OpenGL Next) but was eventually released as its own API. Both are maintained by the Khronos Group
-

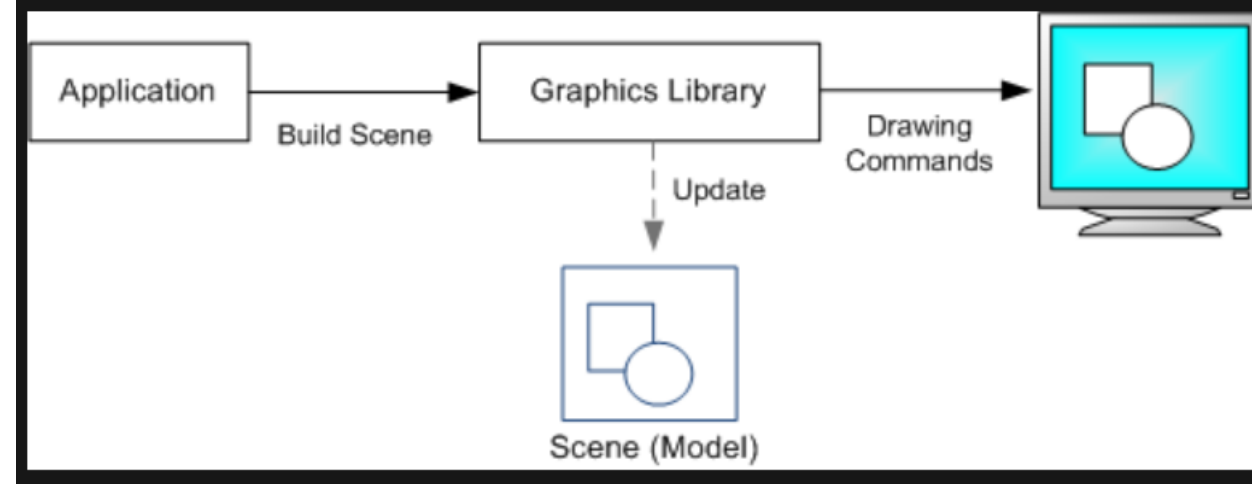
Immediate Mode Vs Retained Mode

Retained Mode (WPF, SVG, most game engines)

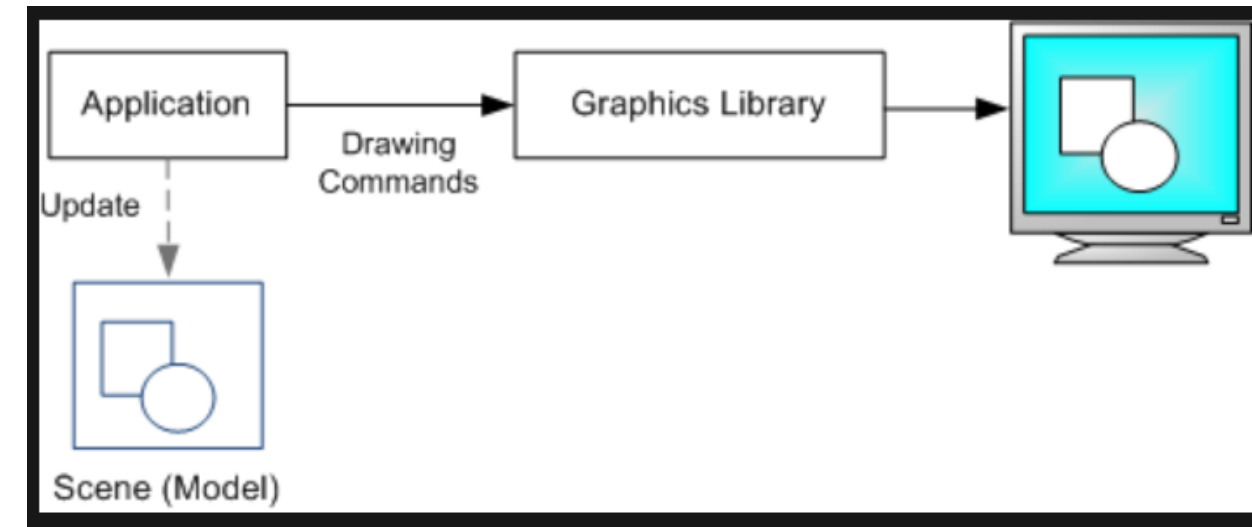


- **Application model** in app and **Display model** in graphics platform
 - **Display model** contains information that defines geometry to be viewed
 - is a geometric subset of **Application model** (typically a **scene graph**)
 - Graphics platform keeps record of primitives that compose scene
 - Simple drawing application does not need **Application model** (e.g., clock example)
 - **No right answer on which to use** – context-dependent tradeoffs
 - convenience vs. overhead of the library having to maintain and synch this data structure
-

A retained-mode API is declarative. The application constructs a scene from graphics primitives, such as shapes and lines. The graphics library **stores a model of the scene in memory**. To draw a frame, the graphics library transforms the scene into a set of drawing commands. **Between frames, the graphics library keeps the scene in memory**. To change what is rendered, the application issues a command to update the scene—for example, to add or remove a shape. The library is then responsible for redrawing the scene.



An immediate-mode API is procedural. Each time a new frame is drawn, the application directly issues the drawing commands. **The graphics library does not store a scene model between frames**. Instead, the application keeps track of the scene.

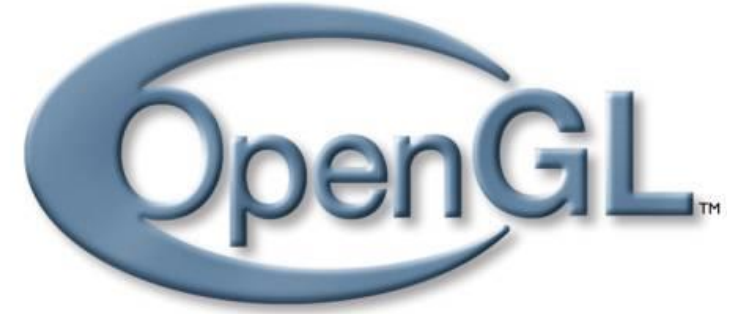


Retained-mode APIs can be simpler to use, because the API does more of the work for you, such as initialization, state maintenance, and cleanup. On the other hand, they are often less flexible, because the API imposes its own scene model. Also, a retained-mode API can have higher memory requirements, because it needs to provide a general-purpose scene model. With an immediate-mode API, you can implement targeted optimizations.

2.1.3 OpenGL Vertex Pipeline

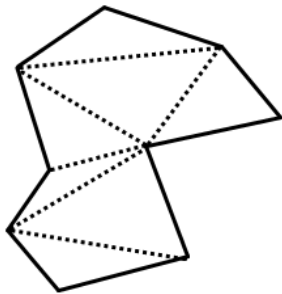
OpenGL (1/3)

- Immediate-mode graphics API
 - No display model, application must direct OpenGL to draw primitives
- Implemented in C, also works in C++
 - Bindings available for many other programming languages
- Various types of input to shaders
 - **Attributes** are the properties of a single vertex
 - Position, normal vector are examples of these
 - **Uniforms** are properties with a single value for multiple vertices (or an entire object)
 - Scale factor, rotation are examples of these
 - OpenGL has many built in types including vectors and matrices

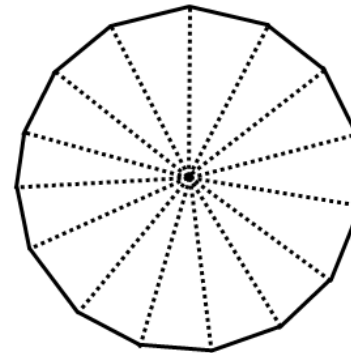


Representing Shapes

- 3D shapes are usually represented as a collection of vertices that make up triangles or quads
 - OpenGL uses triangles
 - Other methods include 3D voxels, polynomial spline curves and surfaces, etc.
- We can use triangles to build arbitrary polygons, and approximate smooth shapes.



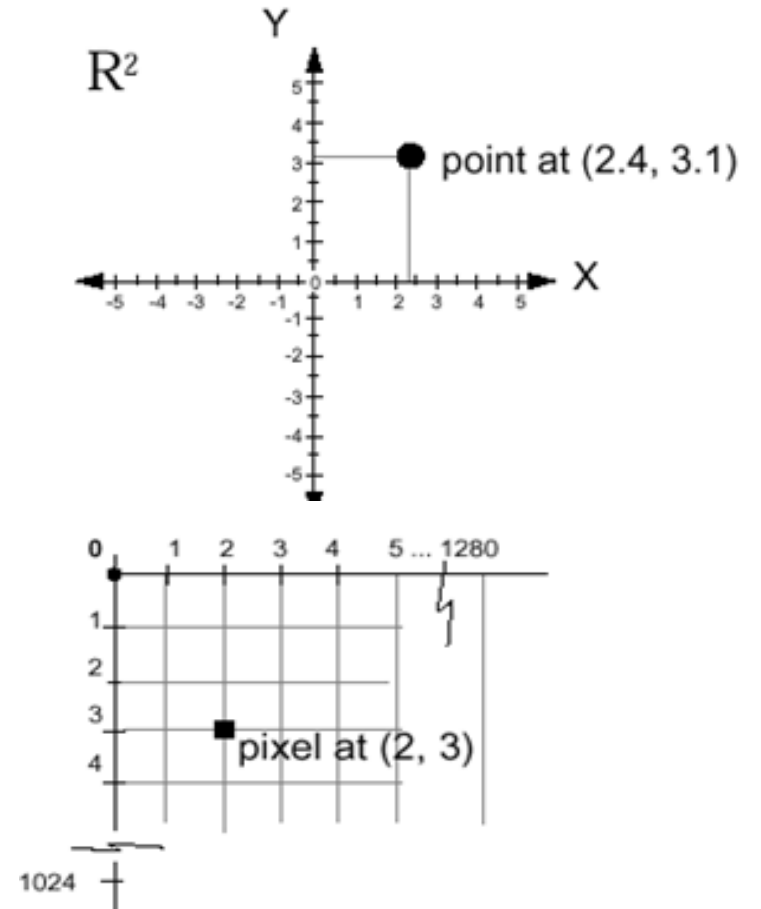
A complex polygon made of
triangle primitives



An approximate circle made of
triangle primitives

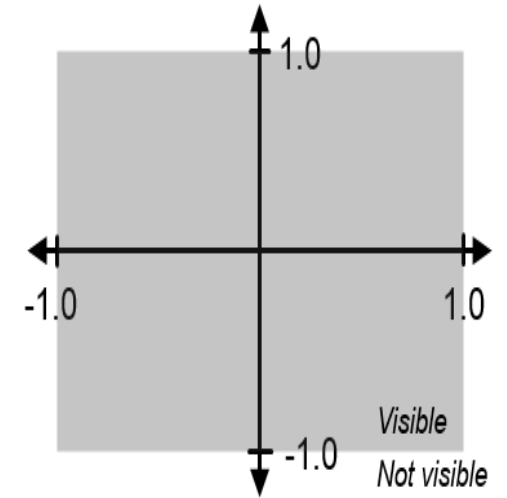
Coordinate Systems (1/3)

- Cartesian coordinates in math, engineering
 - typically modeled as floating point
 - typically X increasing right, Y increasing up
- Display (physical) coordinates
 - integer only
 - typically X increasing right, Y increasing down
 - 1 unit = 1 pixel
- But we want to be insulated from physical display (pixel) coordinates
 - OpenGL is the intermediary



Coordinate Systems (2/3)

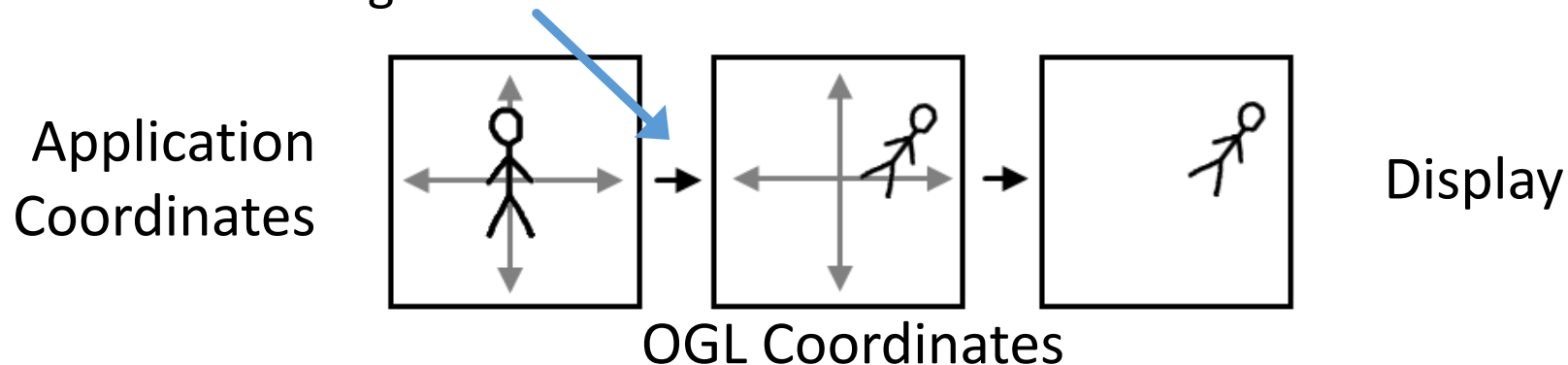
- OpenGL Coordinates map from app to pixels in the window
 - Choose a convention
 - We will use a traditional Cartesian system
 - Units are based on the size of the window or screen
 - Visible area stretches to fill canvas – our OGL glue code provides a fixed-size square canvas
 - Stretches all content inside, so a square will stretch to a rectangle as window is expanded
 - Units correlate to **percentage** of window size, don't correspond to physical units or pixels
- Define coordinate system using the **projection matrix**. Supply it to shader as a *uniform* variable (the term **projection matrix** will become clear)
 - We use standard $[-1, 1]$ coordinate system for 2D



```
glm::mat4 projectionMat; // Our projection matrix is a 4x4 matrix
projectionMat = glm::ortho(-1, // X coordinate of left edge
                           1,  // X coordinate of right edge
                           -1,  // Y coordinate of bottom edge
                           1,    // Y coordinate of top edge
                           1,    // Z coordinate of the "near" plane
                           -1); // Z coordinate of the "far" plane
```

Coordinate Systems (3/3)

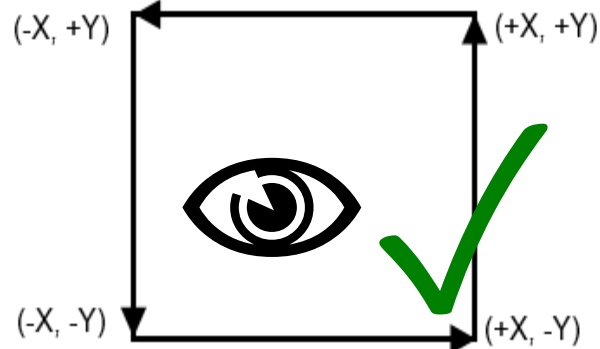
- Two choices on how to think
 1. Draw everything in OpenGL coordinate system
 - This can get very inconvenient!
 2. Choose your own abstract coordinate system natural for your app (in nanometers, lightyears,...), then specify all app's primitives to OpenGL using your coordinates.
 - Must also specify a **transformation** to map the application coordinates to OpenGL coordinates
- “**Transformation**” usually means a change - scale, rotate, and/or translate – or a combination of changes



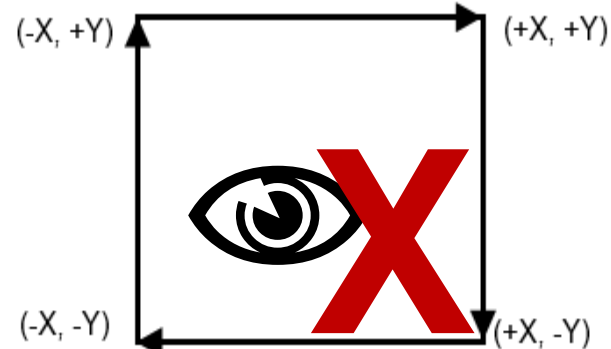
Winding Order (a little intrusion of 3D into our 2D world)

- Order is important: vertices must be specified in **counter-clockwise** order relative to the viewer. Otherwise nothing shows up!
 - Winding order determines whether a triangle's **normal vector** is facing in front or behind it. Triangles facing the wrong way will be invisible!
 - Counter-clockwise winding consistent with the “right-hand rule”

```
GLfloat vertexData[] = {  
    -.7, -.7,  
    .7, -.7,  
    .7, .7,  
    -.7, .7, };
```



```
GLfloat vertexData[] = {  
    -.7, -.7,  
    -.7, .7,  
    .7, .7,  
    .7, -.7, };
```



OpenGL State

- OpenGL is a state machine
- OpenGL functions are of two types
 - Primitive generating
 - Can cause output if primitive is visible
 - How vertices are processed and appearance of primitive are controlled by the state
 - State changing
 - Transformation functions
 - Attribute functions

Lack of Object Orientation

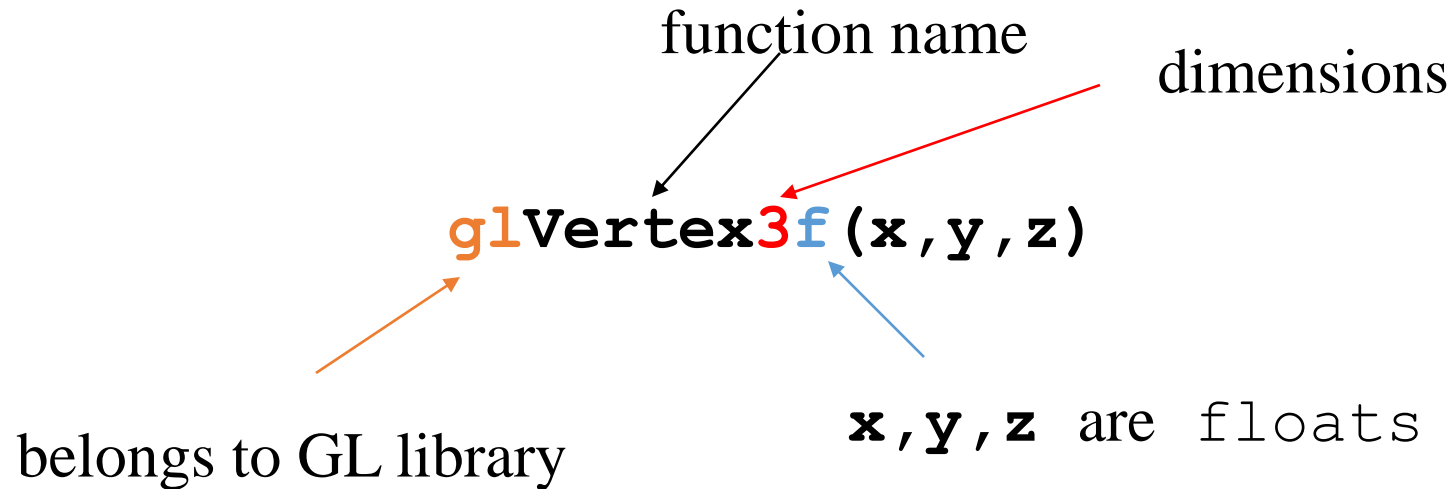
- OpenGL is not object oriented so that there are multiple functions for a given logical function
 - `glVertex3f`
 - `glVertex2i`
 - `glVertex3dv`
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

OpenGL function format

function name dimensions

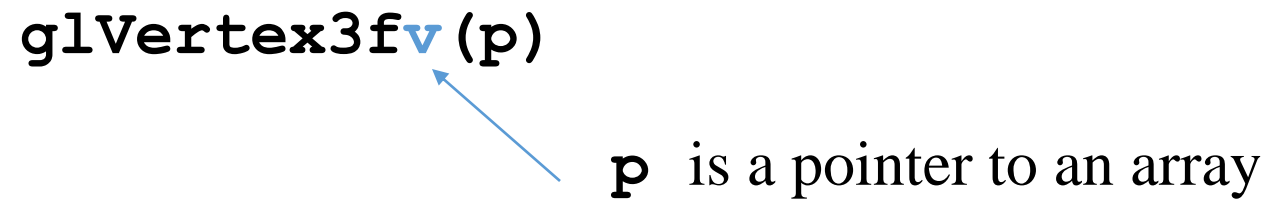
glVertex3f(x, y, z)

belongs to GL library **x, y, z** are floats

A diagram showing the function signature glVertex3f(x, y, z). The 'gl' is orange, 'Vertex' is black, '3' is red, and 'f' is blue. An orange arrow points from 'belongs to GL library' to 'gl'. A black arrow points from 'function name' to 'Vertex'. A red arrow points from 'dimensions' to '3'. A blue arrow points from 'x, y, z are floats' to 'f'.

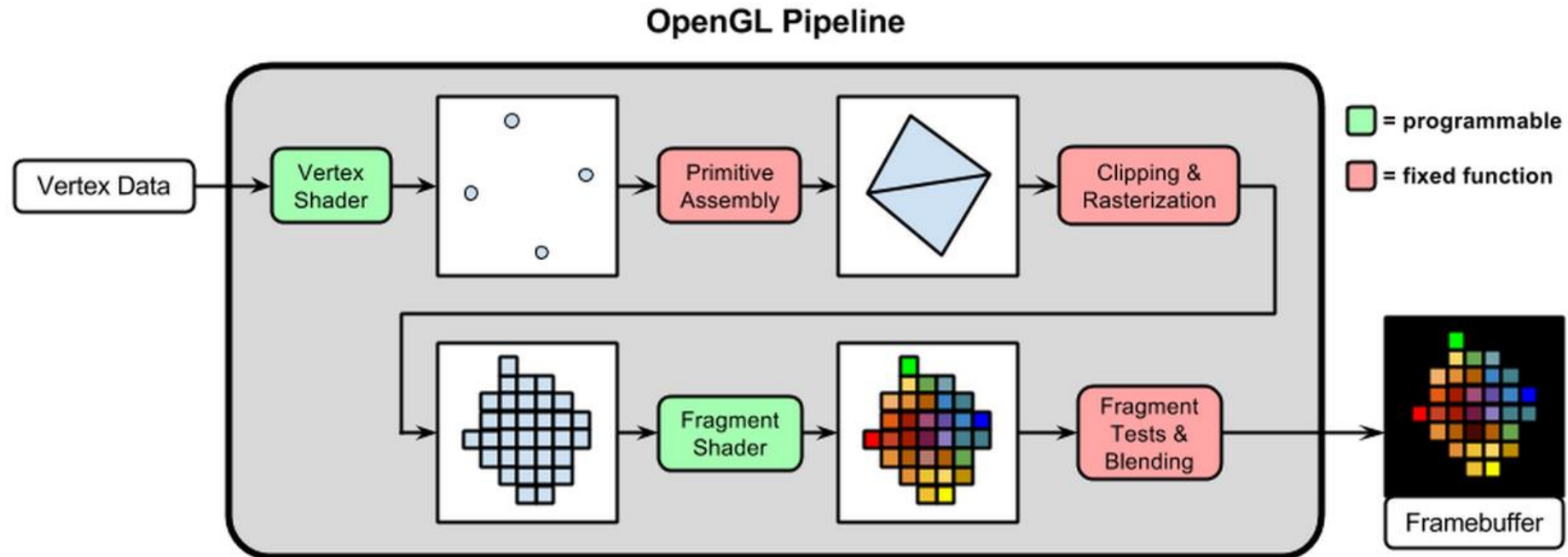
glVertex3fv(p)

p is a pointer to an array

A diagram showing the function signature glVertex3fv(p). The 'glVertex3f' is black and 'v' is blue. A blue arrow points from 'p is a pointer to an array' to 'v'.

OpenGL Basics

- An intro to OpenGL data structures to represent vertex geometry
 - Generate 2D graphics and learn the modern OpenGL pipeline

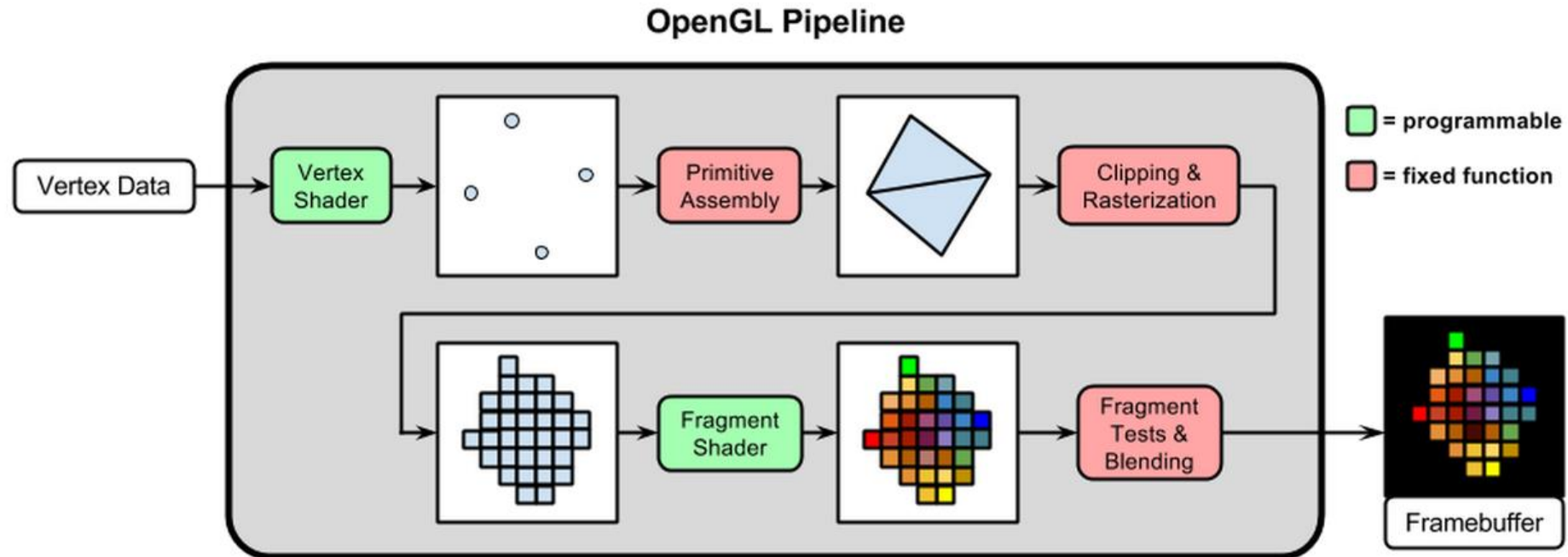


- **Fragment:** a pixel-independent sample within a triangle with all its associated attributes, e.g., color, depth, texture coordinates, ... the fragment shader provides a many-to-one mapping between fragments and pixels (e.g. for supersampling, see Image Processing 2)

2.1.2 OpenGL Pipeline

OpenGL Vertex Pipeline

- An intro to OpenGL data structures to represent vertex geometry held this week
 - Generate 2D graphics and learn the modern OpenGL pipeline



- Fragment: a pixel-independent sample within a triangle with all its associated attributes, e.g., color, depth, texture coordinates, ... the fragment shader provides a many-to-one mapping between fragments and pixels (e.g. for supersampling, see Image Processing 2)

OpenGL Vertex Pipeline

The OpenGL specification specifies exactly what the result/output of each function should be and how it should perform. It is then up to the developers implementing this specification to come up with a solution of how this function should operate.



The modern approach requires the developer to truly understand OpenGL and graphics programming and while it is a bit difficult, it allows for much more flexibility, more efficiency and most importantly: a much better understanding of graphics programming.

OpenGL Vertex Pipeline

Extensions

A great feature of OpenGL is its support of extensions. Whenever a graphics company comes up with a new technique or a new large optimization for rendering this is often found in an extension implemented in the drivers.

The developer has to query whether any of these extensions are available before using them (or use an OpenGL extension library)

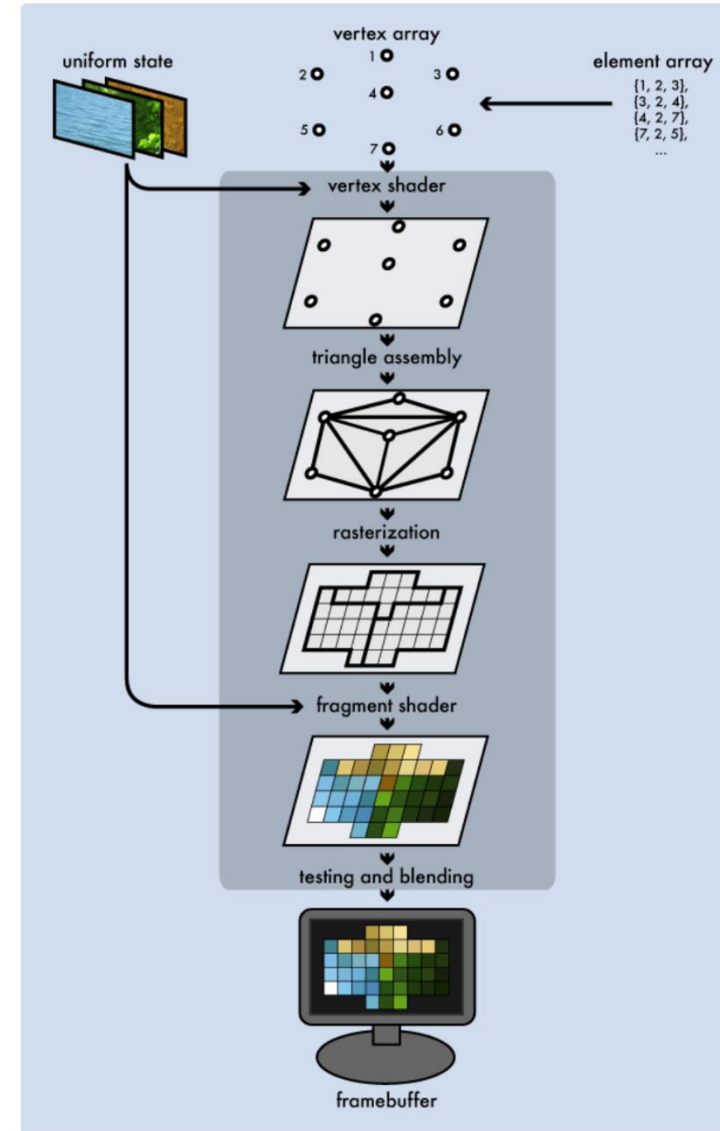
```
if(GL_ARB_extension_name)
{
    // Do cool new and modern stuff supported by hardware
}
else
{
    // Extension not supported: do it the old way
}
```



OpenGL Vertex Pipeline

State machine

OpenGL is by itself a large state machine: a collection of variables that define how OpenGL should currently operate. The state of OpenGL is commonly referred to as the OpenGL context.



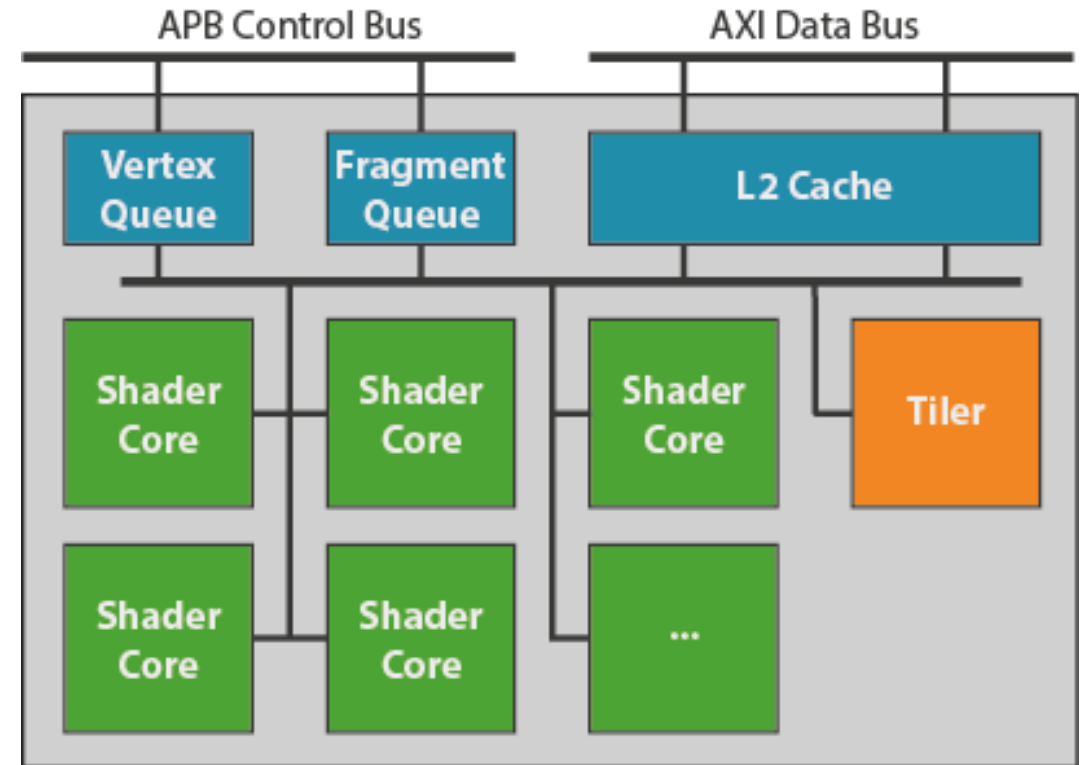
OpenGL Vertex Pipeline

Shaders

Graphics cards of today have thousands of small processing cores to quickly process your data within the graphics pipeline.

The processing cores run small programs on the GPU for each step of the pipeline. These small programs are called **shaders**.

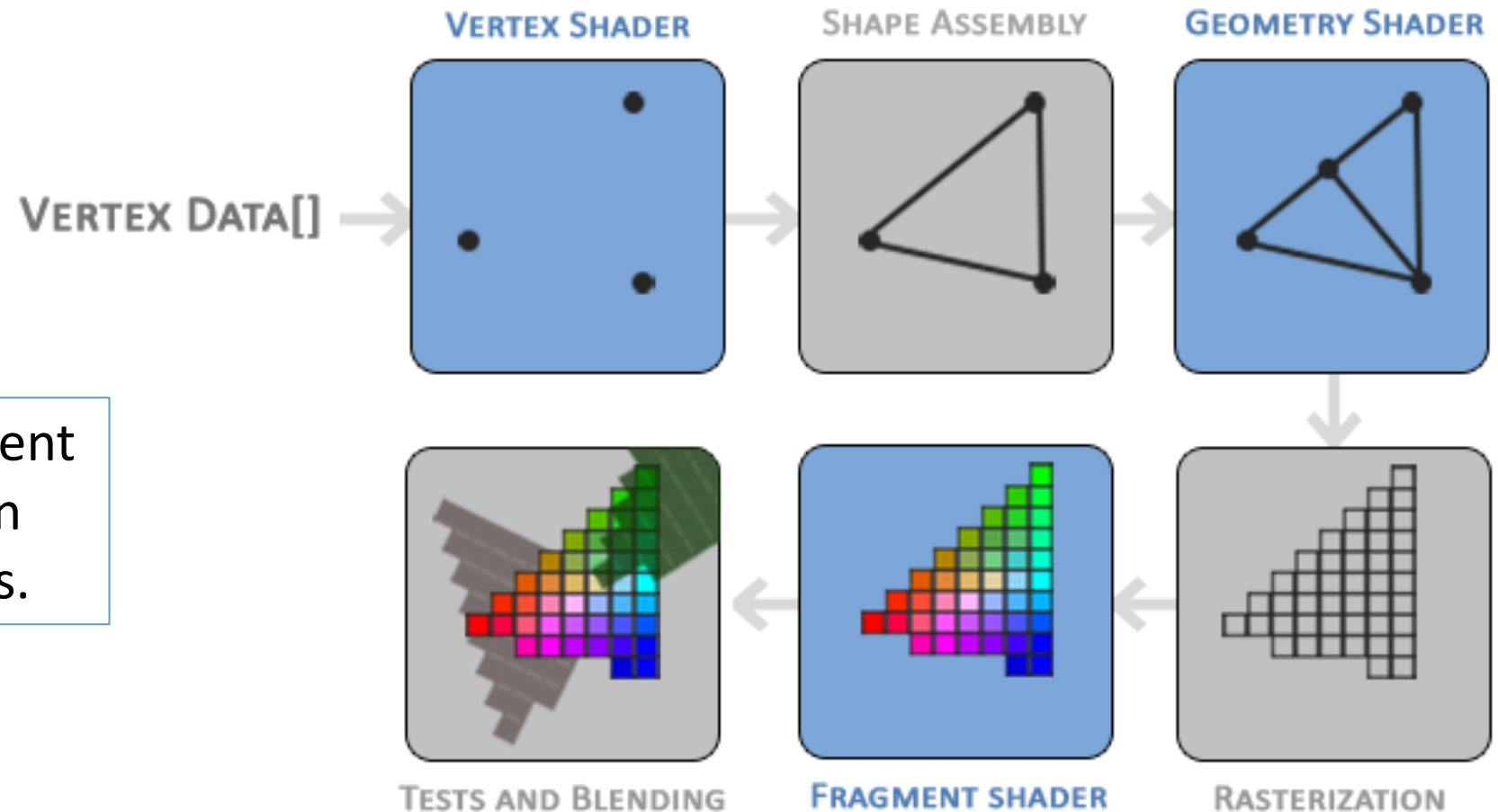
Mali GPU Block Model



Shaders are written in the [OpenGL Shading Language \(GLSL\)](#)

OpenGL Vertex Pipeline

Pipeline



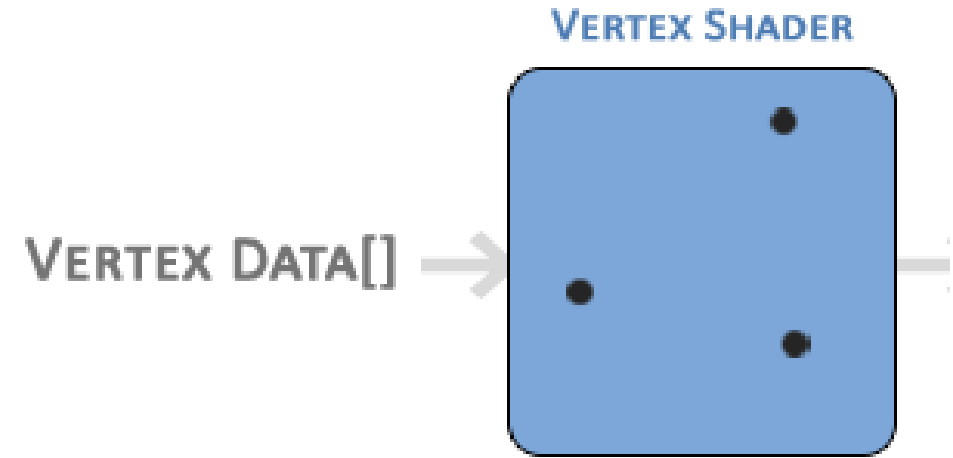
the blue sections represent sections where we can inject our own shaders.

In OpenGL everything is in 3D space, but the screen or window is a 2D array of pixels so a large part of OpenGL's work is about transforming all 3D coordinates to 2D pixels that fit on your screen. The process of transforming 3D coordinates to 2D pixels is managed by the graphics pipeline of OpenGL.

OpenGL Vertex Pipeline

Vertex Data

As input to the graphics pipeline we pass in a list of three **3D coordinates** that should form a triangle in an **array** here called **Vertex Data**; this vertex data is a collection of vertices. A vertex is a collection of data per 3D coordinate.

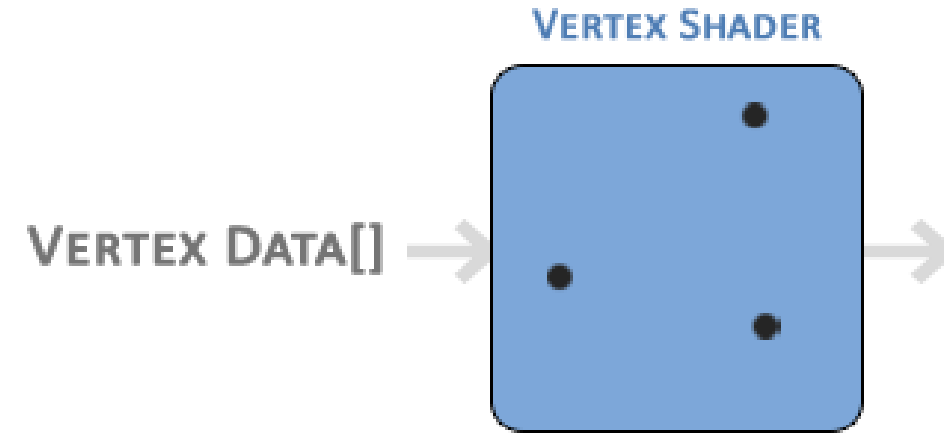


OpenGL Vertex Pipeline

Vertex Shader

The first part of the pipeline is the vertex shader that takes as input a single vertex.

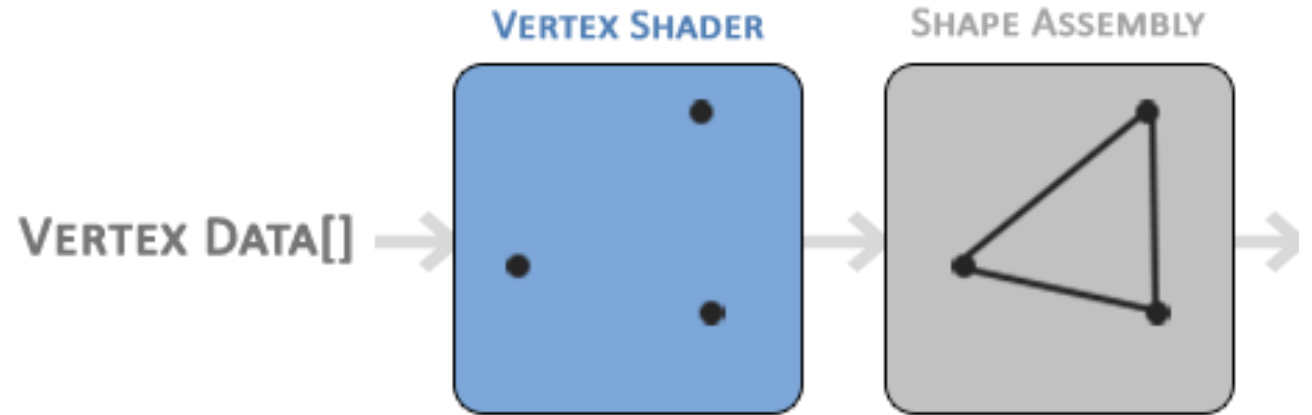
The main purpose of the **vertex shader** is to transform 3D coordinates into different 3D coordinates (more on that later) and the vertex shader allows us to do some basic processing on the vertex attributes



OpenGL Vertex Pipeline

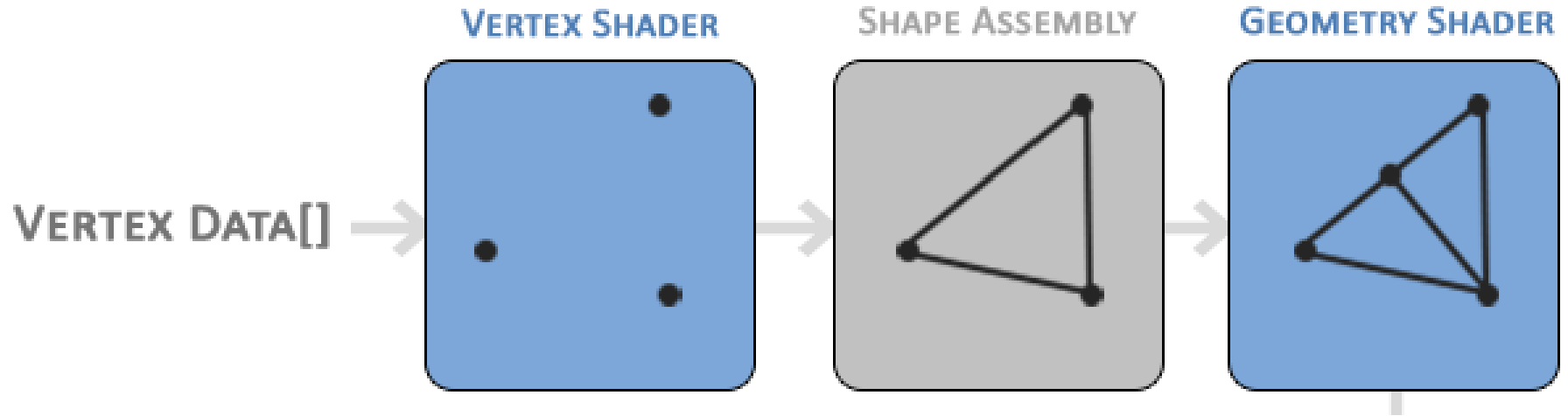
Shape Assembly

The **primitive assembly** stage takes as input all the vertices (or vertex if GL_POINTS is chosen) from the vertex shader that form a primitive and assembles all the point(s) in the primitive shape given; in this case a triangle.



OpenGL Vertex Pipeline

Geometry Shader

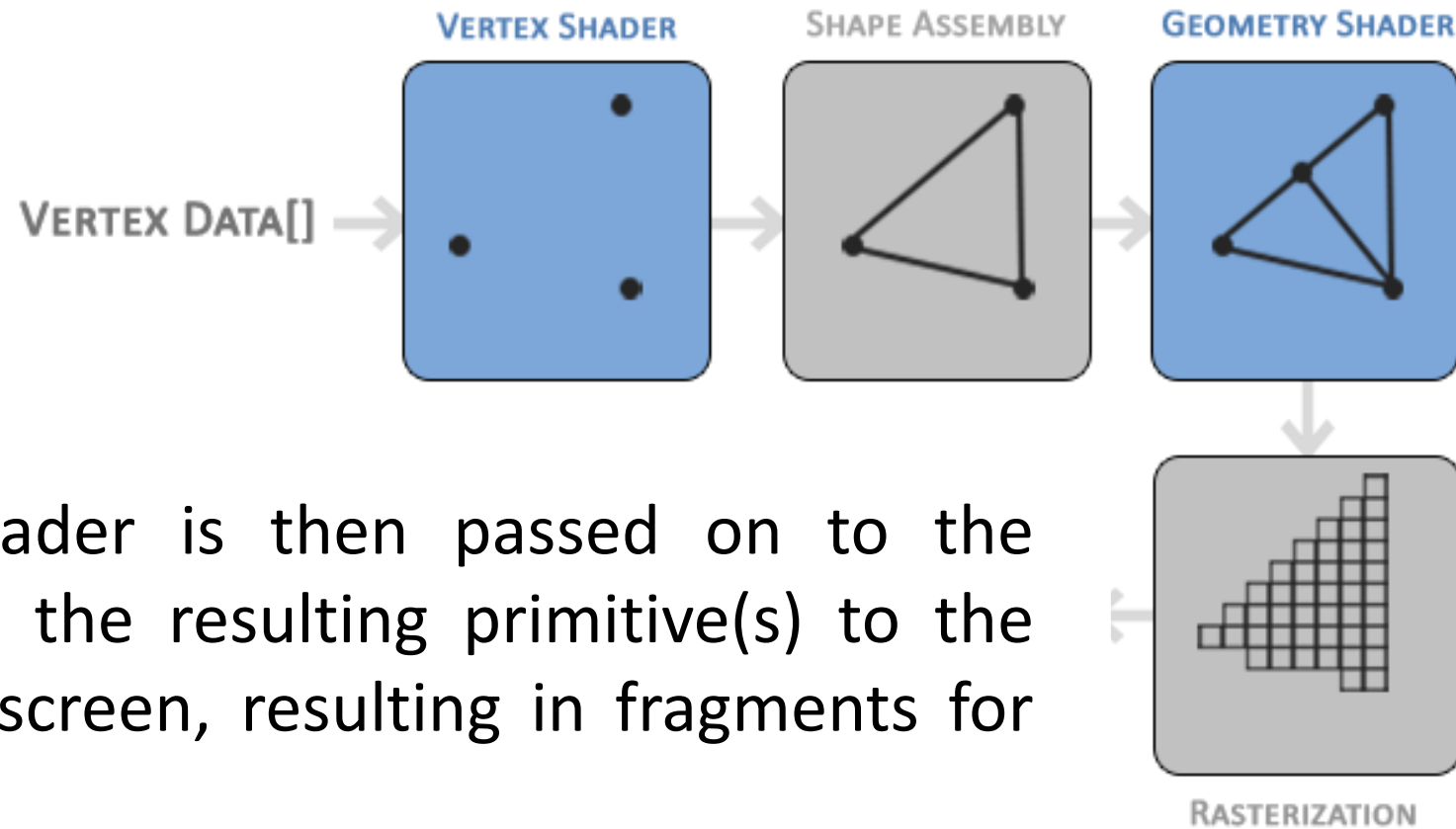


The output of the primitive assembly stage is passed to the geometry shader. **The geometry shader** takes as input a collection of vertices that form a primitive and has the ability to generate other shapes by emitting new vertices to form new (or other) primitive(s).

In this example case, it generates a second triangle out of the given shape.

OpenGL Vertex Pipeline

Rasterization



The output of the geometry shader is then passed on to the **rasterization stage** where it maps the resulting primitive(s) to the corresponding pixels on the final screen, resulting in fragments for the fragment shader to use.

Before the fragment shaders run, **clipping** is performed. Clipping discards all fragments that are outside your view, increasing performance.

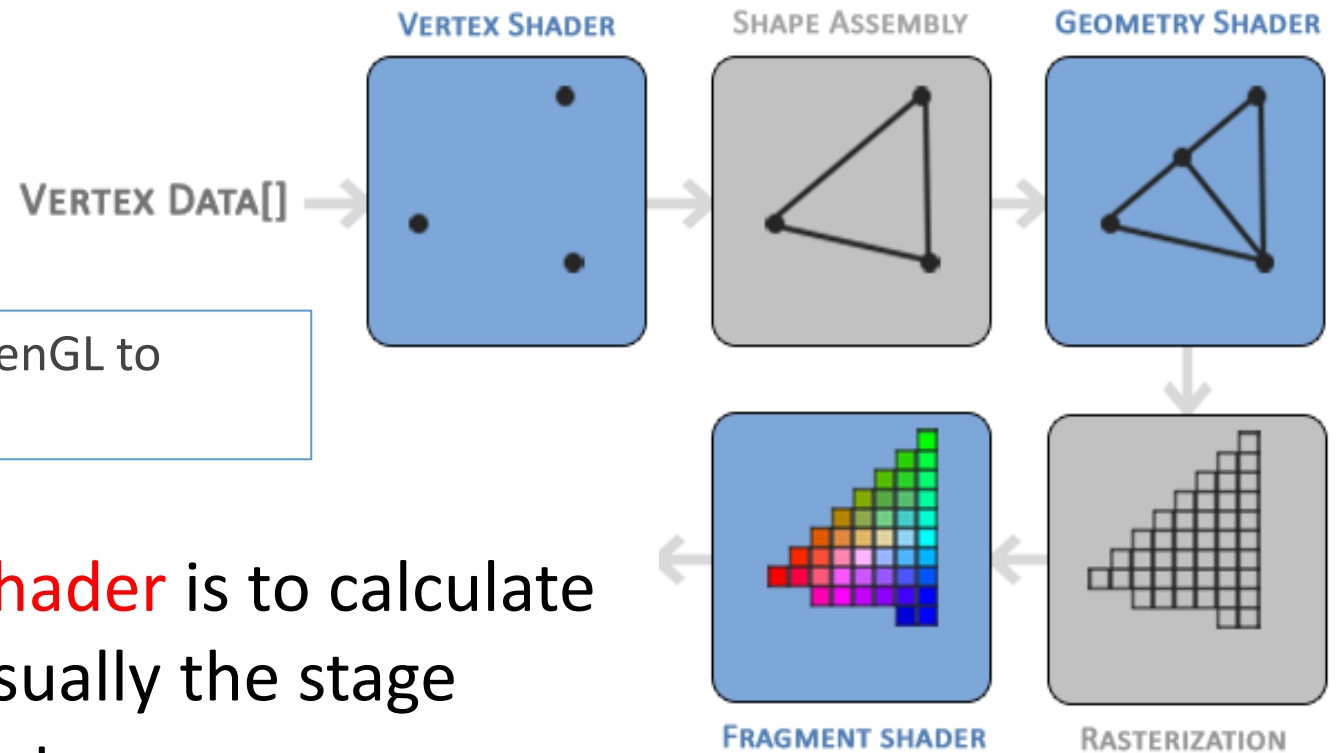
OpenGL Vertex Pipeline

Fragment Shader

A **fragment** in OpenGL is all the data required for OpenGL to render a single pixel.

The main purpose of the **fragment shader** is to calculate the final color of a pixel and this is usually the stage where all the advanced OpenGL effects occur.

Usually the fragment shader contains data about the 3D scene that it can use to calculate the final pixel color (like lights, shadows, color of the light and so on).

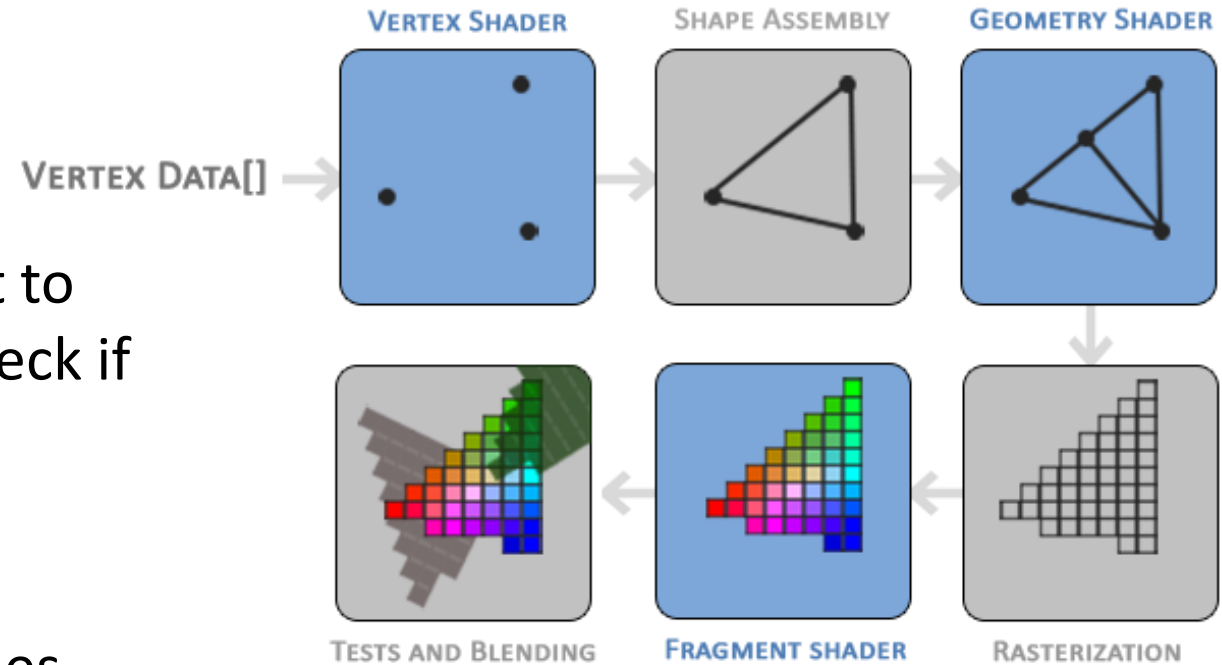


OpenGL Vertex Pipeline

Test and Blending

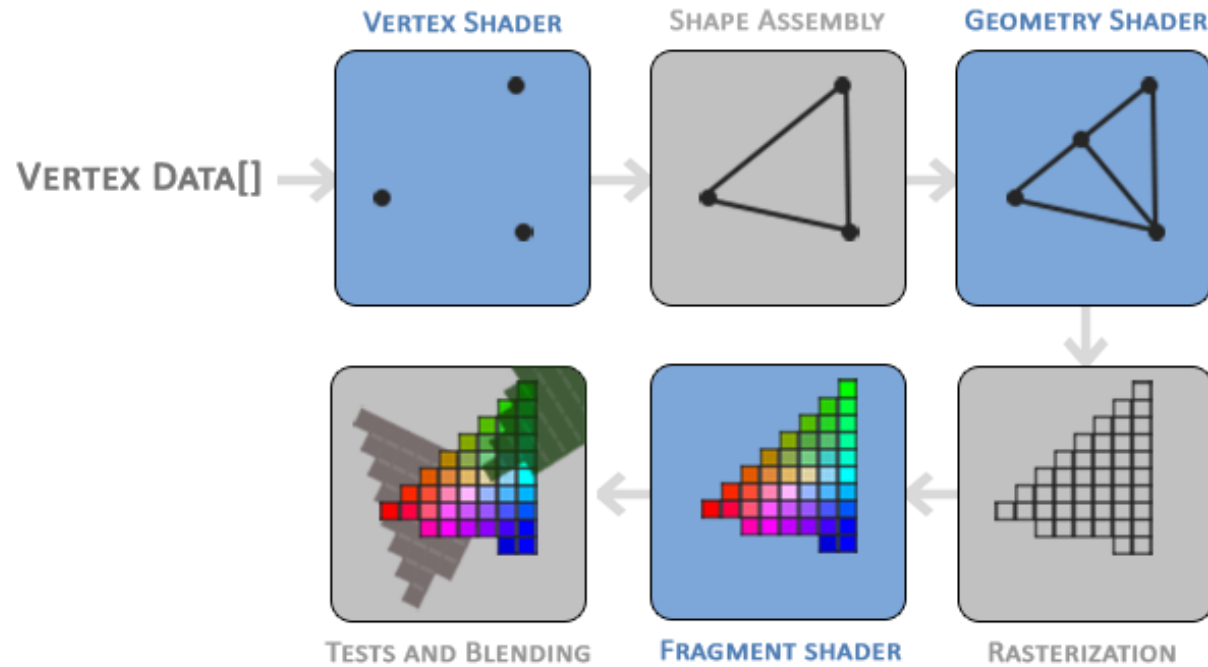
The alpha **test and blending** stage checks the corresponding depth (and stencil) value (we'll get to those later) of the fragment and uses those to check if the resulting fragment is in front or behind other objects and should be discarded accordingly.

The stage also checks for alpha values (alpha values define the opacity of an object) and blends the objects accordingly.



So even if a pixel output color is calculated in the fragment shader, the final pixel color could still be something entirely different when rendering multiple triangles.

OpenGL Vertex Pipeline



https://chamilo.grenoble-inp.fr/courses/ENSIMAG4MMG3D6/document/resources/opengl_slides/index.html#/