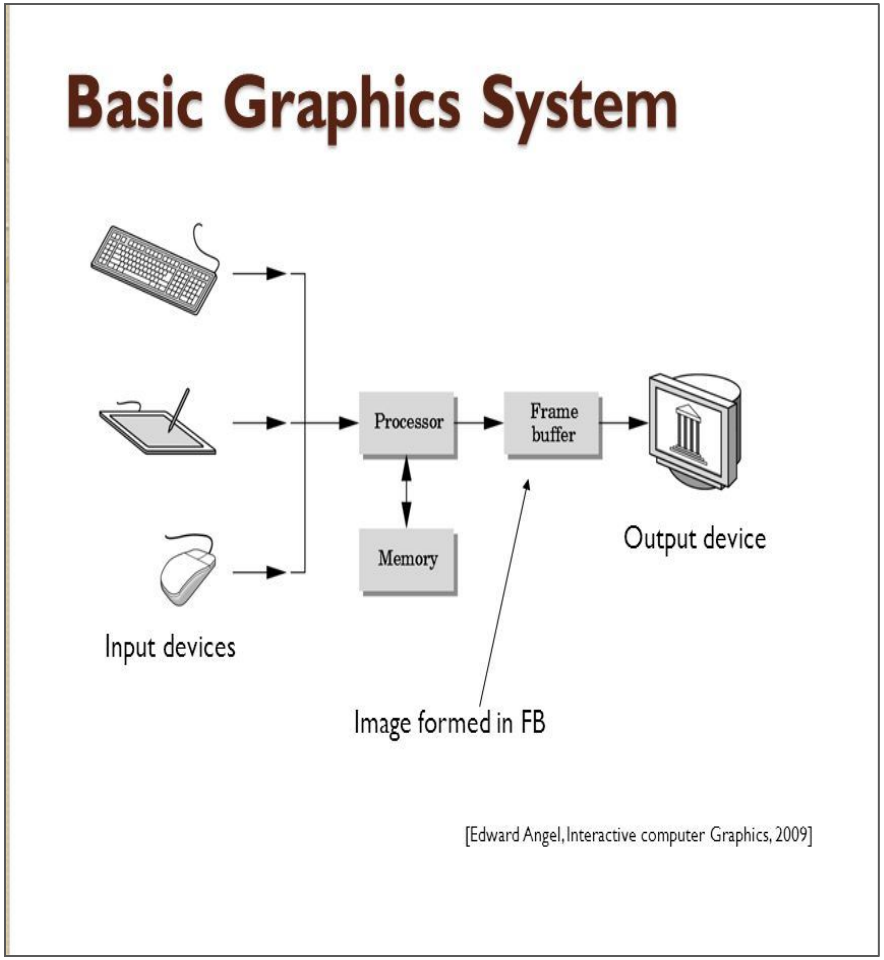
Graphics Pipeline Architecture & OGL/WebGL

CSE606: Computer Graphics

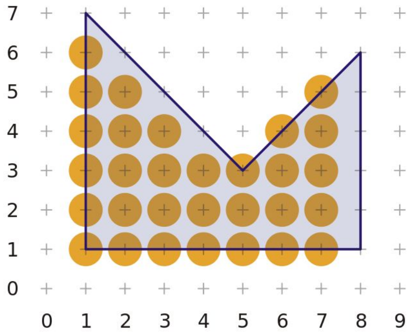
Jaya Sreevalsan Nair, IIIT Bangalore

January 13, 2025

Processor

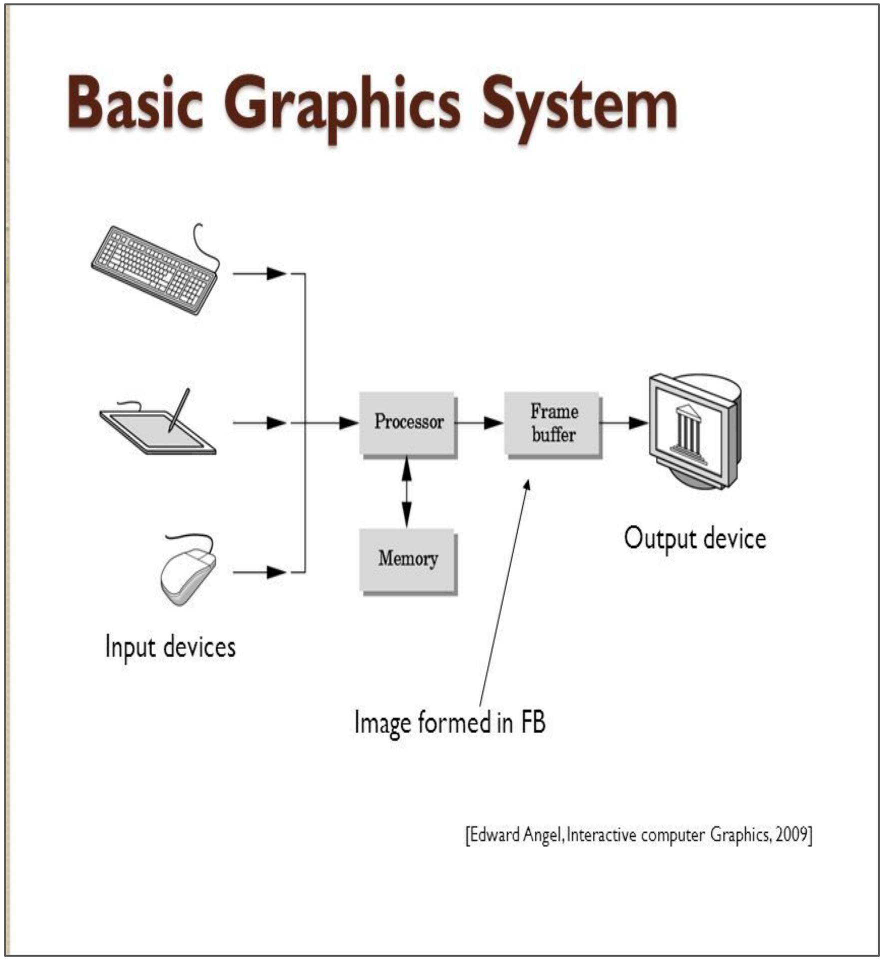


- Graphical processing is primarily **Rasterization** or **Scan Conversion**,
  - which is conversion of geometric entities to pixel information.
  - includes specifying location of entity on the pixels and color of pixels.



Scan Conversion of a Polygon.
(Image courtesy: Wikimedia Commons.)

Processor



- Earlier, CPU was used for normal compute/processing and graphical processing.
- Today, special-purpose Graphics processing units (GPU) uses hardware accelerator to fill up FB.
    - GPU can be part of motherboard or graphics card.
    - Hence FB may be included in the graphics card as well.

Graphics Programming

Ingredients

- Objects
  - geometry, color/material
- Scene
  - composition with objects
- Lighting
  - instances, positions, properties
- Projection Plane (for Image Generation)
  - position, properties

Graphics Programming

Implementation

- Conversion of 3D objects to 2D image
- Color assignment to each pixel
  - Information from object properties and location
  - Information from light properties and location
  - Interaction between objects and light

Graphics Programming

System Requirements

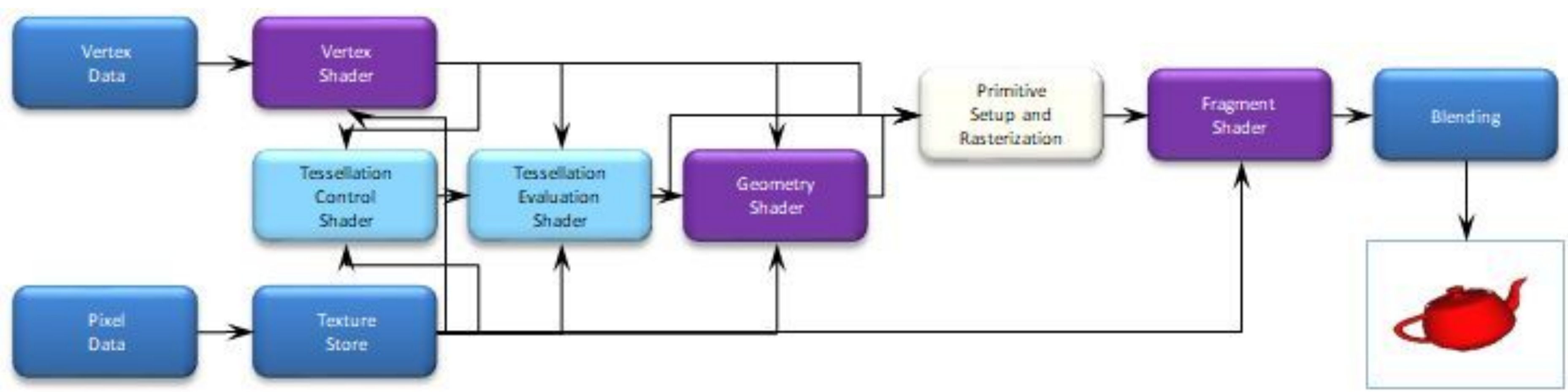
- Desired amount of user interaction
  - Real-time computations for interactive applications
  - No user interaction ⇒ offline rendering
- Desired effects of realism
  - Photo-realism vs functional realism
- Desired frame rate
  - Speed of generating images, refreshing framebuffer

Graphics Pipeline Architecture

OpenGL now (4.x)

[From Angel and Shreiner, and S606-lecture_2023-01-09.pdf]

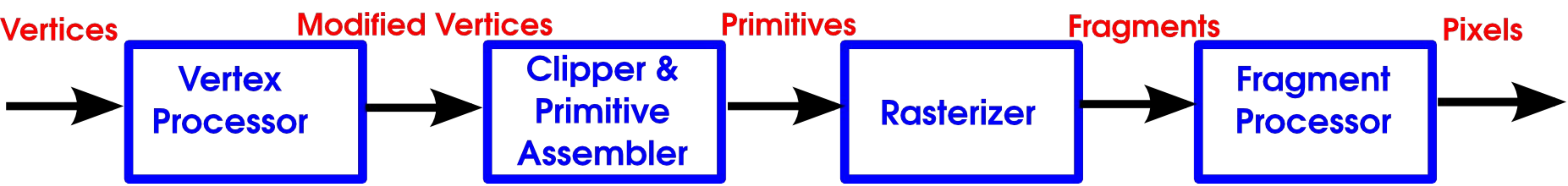
- Architecture of OpenGL has evolved to enable exploiting GPU and providing flexibility for the applications
- Rendering done using GPU rather than CPU
- GPU controlled through programs called shaders, which control different aspects of the rendering process
- Application's job is to send data to GPU



Graphics Architecture – governing OpenGL

Use pipeline architecture using special VLSI chips.

- Multiple processes in sequence overlap, thus, increasing throughput, and masking latency.
- Latency: (Significant) delay between start and stop of a process.
- Throughput: Result of the entire master process - is significantly high compared to a single process at a time.



Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

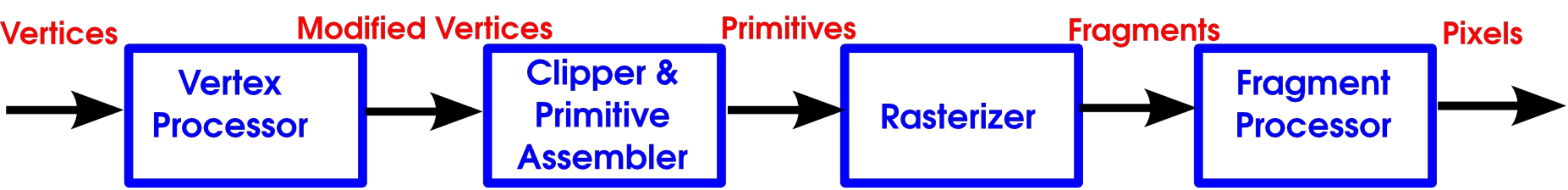
Graphics Architecture – governing OpenGL

Use pipeline architecture using special VLSI chips.

- Multiple processes in sequence overlap, thus, increasing throughput, and masking latency.
- Latency: (Significant) delay between start and stop of a process.
- Throughput: Result of the entire master process - is significantly high compared to a single process at a time.

Fragment:

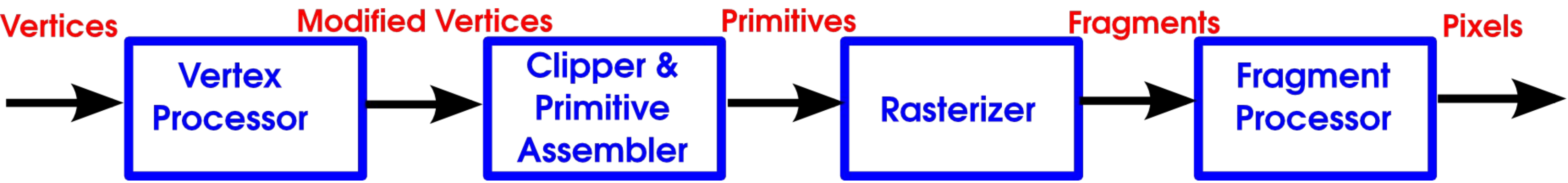
- A potential pixel, carries information on:
  - Location & color: used for updating the corresponding pixel in FB;
  - Depth: to determine the order of rendering of fragments at a given pixel location.



Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

Processes

Vertex Processing: Coordinate transformations on vertices; compute a color for a vertex.

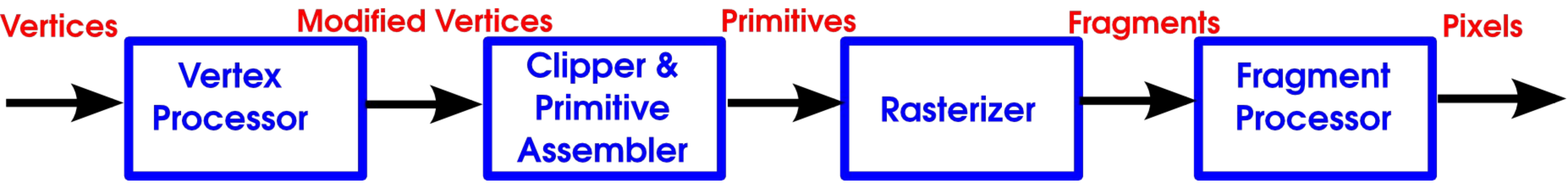


Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

Processes

Vertex Processing: Coordinate transformations on vertices; compute a color for a vertex.

Clipping & Primitive Assembly: Assembling sets of vertices as primitives; retaining primitives within clipping volume (in the field of view).



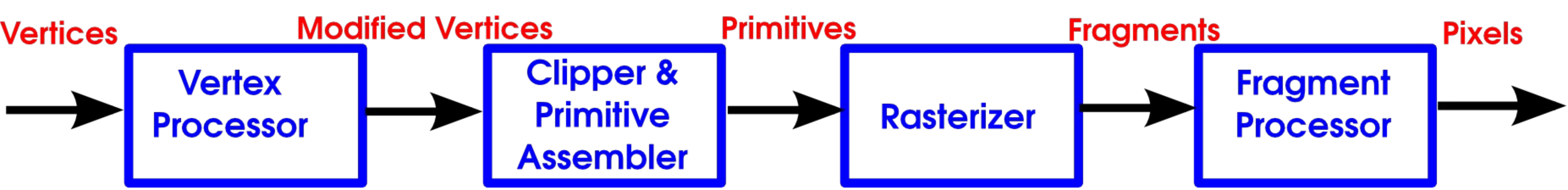
Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

Processes

Vertex Processing: Coordinate transformations on vertices; compute a color for a vertex.

Rasterization: Using scan-conversion/ rasterization to convert primitives to fragments.

Clipping & Primitive Assembly: Assembling sets of vertices as primitives; retaining primitives within clipping volume (in the field of view).



Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

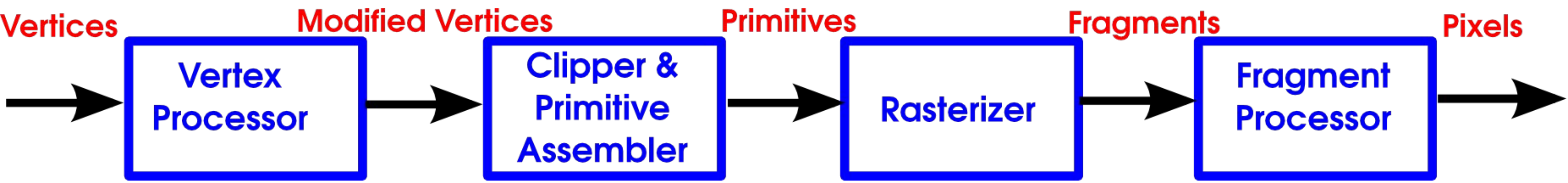
Processes

Vertex Processing: Coordinate transformations on vertices; compute a color for a vertex.

Clipping & Primitive Assembly: Assembling sets of vertices as primitives; retaining primitives within clipping volume (in the field of view).

Rasterization: Using scan-conversion/ rasterization to convert primitives to fragments.

Fragment Processing: Fragments are used to update the pixels in FB.



Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

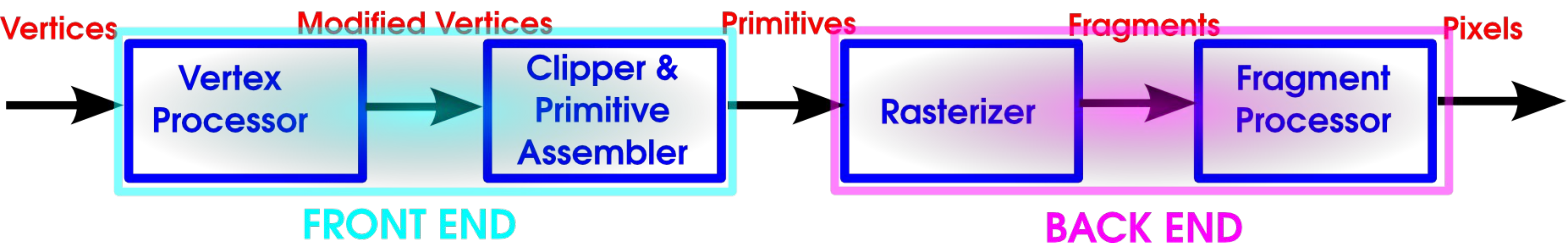
Processes

Vertex Processing: Coordinate transformations on vertices; compute a color for a vertex.

Clipping & Primitive Assembly: Assembling sets of vertices as primitives; retaining primitives within clipping volume (in the field of view).

Rasterization: Using scan-conversion/ rasterization to convert primitives to fragments.

Fragment Processing: Fragments are used to update the pixels in FB.



Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

Processes

Front end: Geometric Processing

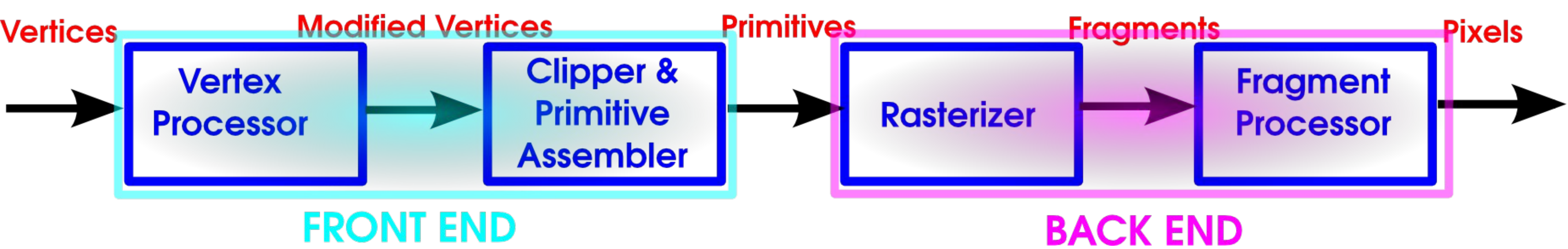
Vertex Processing: Coordinate transformations on vertices; compute a color for a vertex.

Clipping & Primitive Assembly: Assembling sets of vertices as primitives; retaining primitives within clipping volume (in the field of view).

Back end: FB Processing

Rasterization: Using scan-conversion/ rasterization to convert primitives to fragments.

Fragment Processing: Fragments are used to update the pixels in FB.



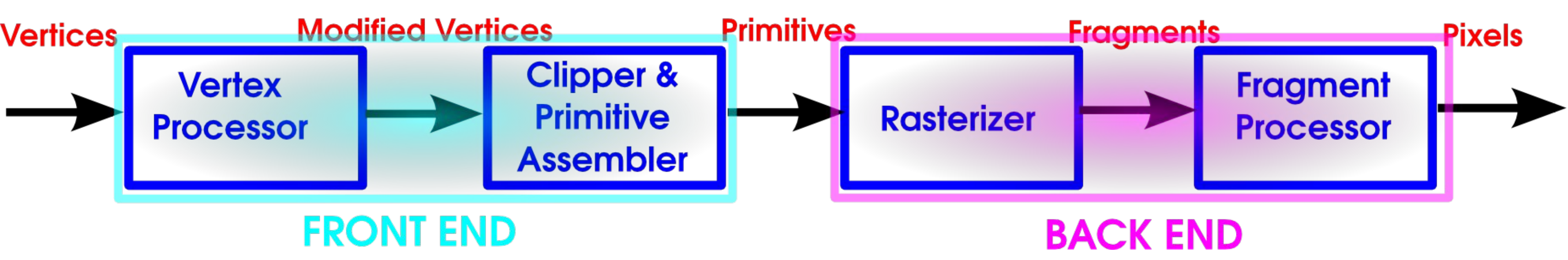
Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

Performance Characteristics

Overall performance of a graphics system characterized by:

- how fast geometric entities move in the front end;
- by how many pixels/second is FB refreshed or altered in the back end.

Now, commodity graphics card can contain the entire pipeline in a single chip, within the GPU.



Geometric Pipeline.
Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL.

Programmable Pipeline

Commodity graphic cards have pipelines built into graphics processing units (GPU).

Traditional pipelines had **fixed functionality**.

Now, vertex processor, geometry processor, and fragment processor are programmable by application program. [We also have tessellation shaders.]

Programmable Pipeline

Commodity graphic cards have pipelines built into graphics processing units (GPU).

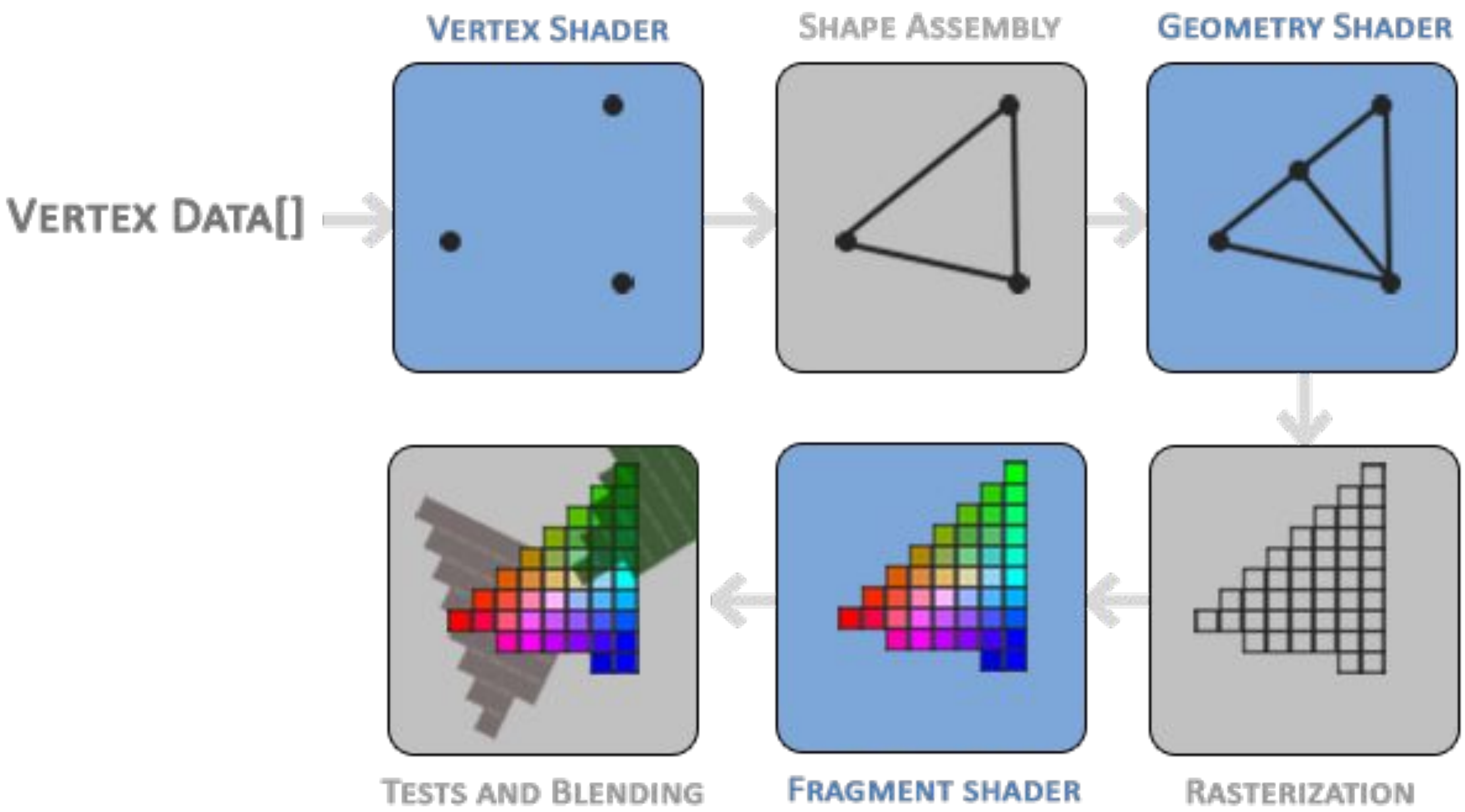
Traditional pipelines had **fixed functionality**.

Now, vertex processor, geometry processor, and fragment processor are programmable by application program. [We also have tessellation shaders.]

Several real-time techniques are doable at interactive frame rates.

- **Vertex processor (vertex shader)** can alter vertices programmatically - to achieve various light-material models or new transformations.
- **Fragment processor (fragment shader)** programs allow use of textures in new ways.

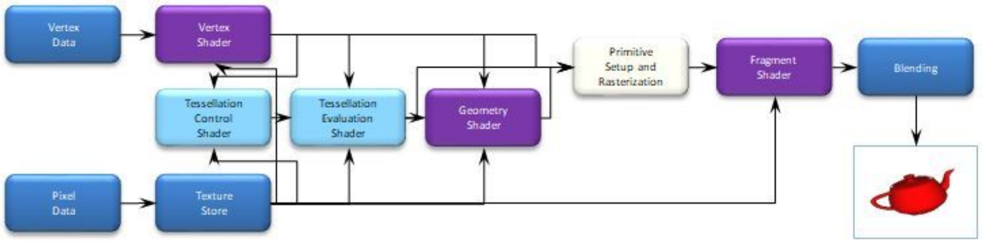
Objects in the Pipeline



https://learnopengl.com/Getting-started/Hello-Triangle

OpenGL

OpenGL Geometry+Pixel Pipeline
(OpenGL 4.0 Logical Diagram)



- OpenGL is a specification, and not a library.
- Application programmer's interface (API): Set of functions in graphics library that interfaces between an application program and a graphics system.
  - Three-dimensional Graphics API: OpenGL R , Direct3D, Open Scene Graph
  - Has functions to specify objects, viewer, light sources, material properties.
- Software drivers: interpret API output for the specific hardware.

Simplified Complete Pipeline

OpenGL supports two types of primitives: Geometric & Image/Raster.

- Image primitives (e.g. text, bitmap) cannot be manipulated in coordinate space as geometric ones (e.g. points, lines).
- Parallel processing of the different primitive types.

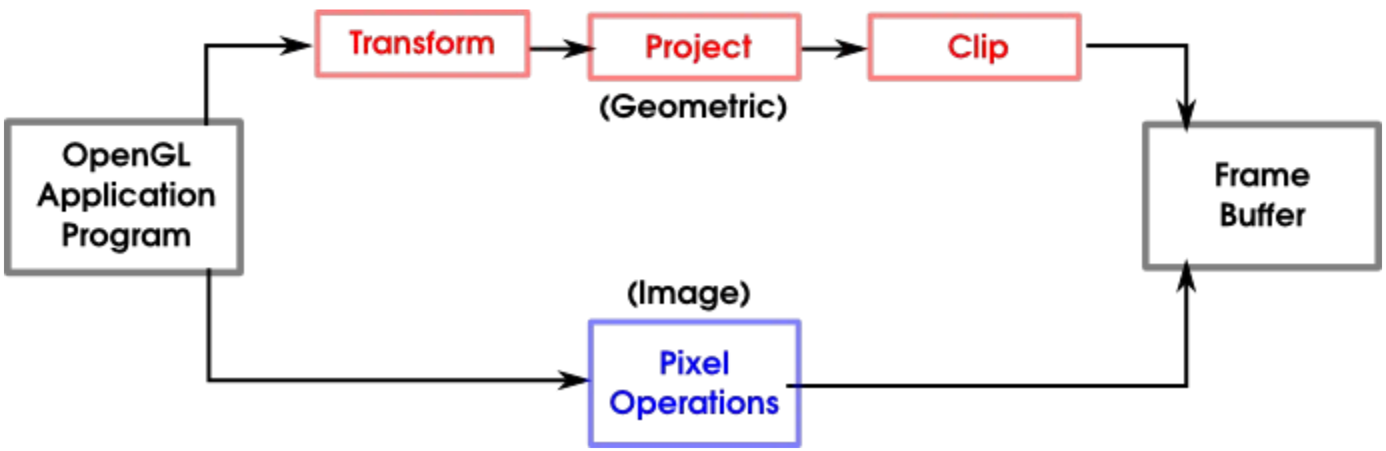
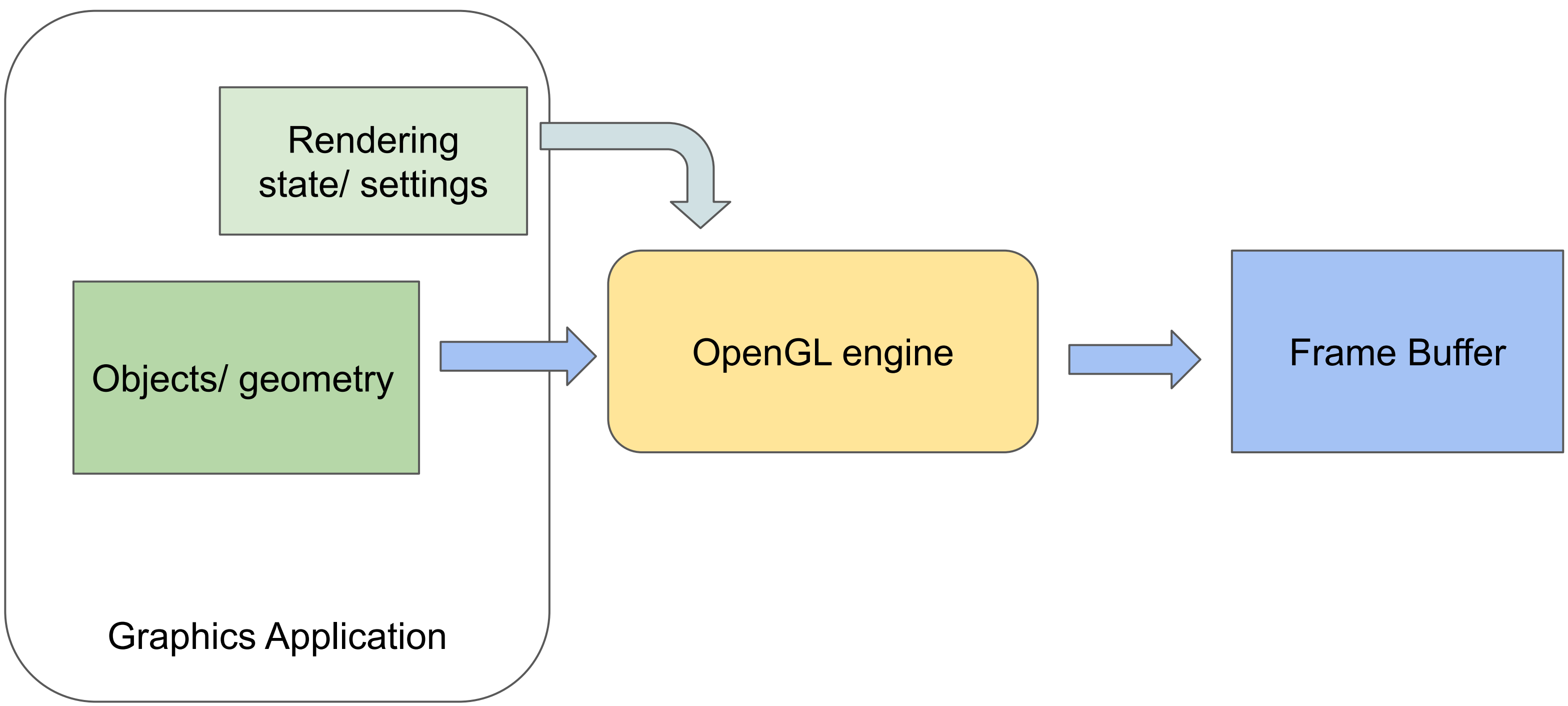


Image adapted from Interactive Computer Graphics: A Top-Down Approach Using OpenGL®.

Overview of OpenGL and WebGL (Programming)

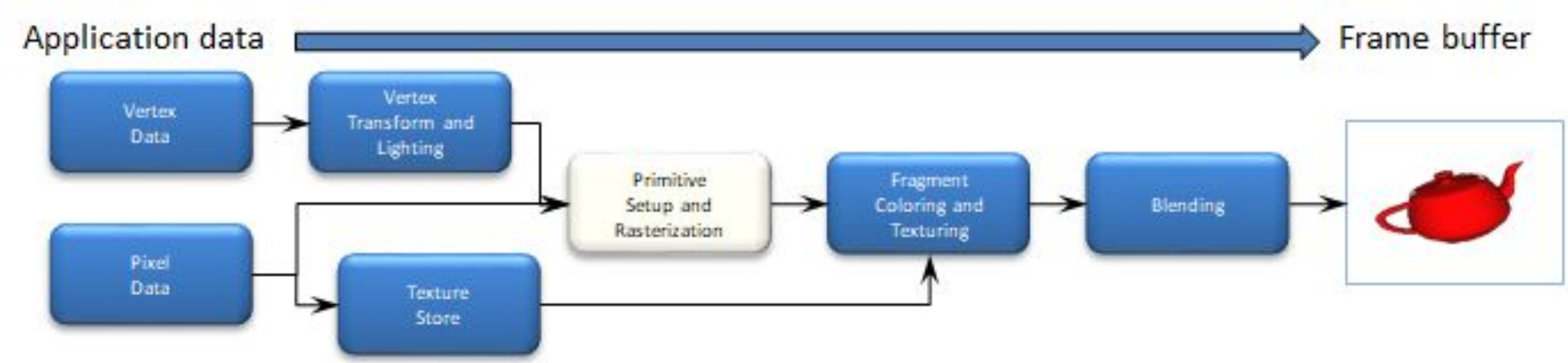
Overview of OpenGL

- Computer graphics rendering API
- One of the “most widely deployed” 3D graphics API
- Support for multiple languages and platforms
- Provides hardware and OS agnostic access to rendering utilities



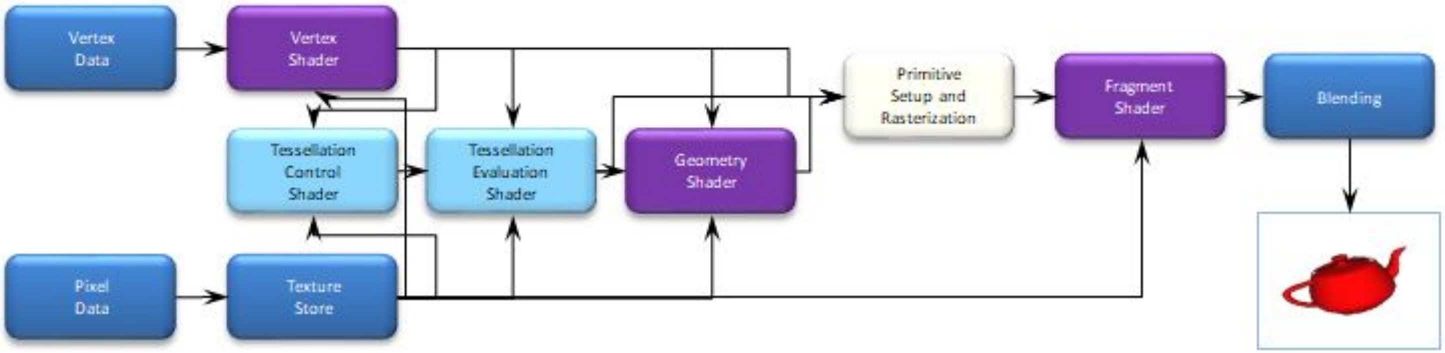
Overview of OpenGL

- Graphics processing organized as a pipeline of operations
- Data flows through the pipeline
- Input variables and state and certain operations are managed by the application
- GPU's provide hardware acceleration and enable "real time" performance
- OpenGL 1.0 introduced in 1992. Now at OpenGL 4.6



OpenGL now (4.x)

- Architecture of OpenGL has evolved to enable exploiting GPU and providing flexibility for the applications
- Rendering done using GPU rather than CPU
- GPU controlled through programs called **shaders,** which control different aspects of the rendering process
- Application's job is to send data to GPU



OpenGL ES

OpenGL for Embedded Systems

OpenGL ES 2.0 - slightly simplified version of OpenGL 3.1

Supports functionality for most common graphics applications

Runs on desktops, mobiles and other devices - default in Android, iOS

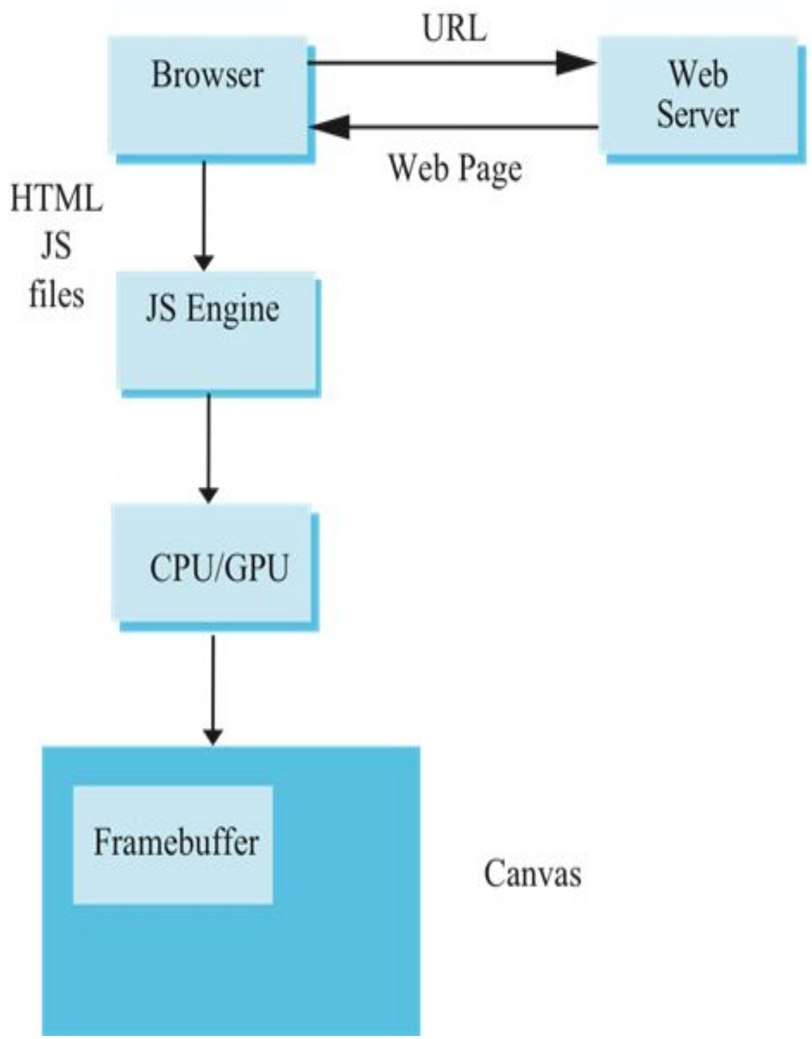
WebGL

WebGL: a Javascript API for OpenGL ES2.0

Supported on most browsers - no special software installation needed

Two parts to a WebGL program:

1. Javascript code that runs within a HTML *canvas* (like a web page)
    a. The main application logic, models, state, interactions,
2. Shaders implemented in GLSL (C-like)
    a. The core rendering related operations



WebGL Program Structure

1. Describe application page (HTML)
   a. Get a WebGL context in the js code

2. Define shaders (GLSL) - added as scripts to html/js files

3. Compute/read models and other data (JS)

4. Send data to GPU; Set state (JS)

5. Render data (JS)

To be discussed in detail in tutorials

```
<canvas id="canvas"></canvas>
--------
// Get A WebGL context
var canvas =
 document.querySelector("#canvas");
var gl =
 canvas.getContext("webgl");
```

WebGL Program Structure

1. Describe application page (HTML)
   a. Get a WebGL context in the js code
2. Define shaders (GLSL) - added as scripts to html/js files
3. Compute/read models and other data (JS)
4. Send data to GPU; Set state (JS)
5. Render data (JS)

To be discussed in detail in tutorials

```
<!-- vertex shader -->
<script  id="vertex-shader-2d"
type="x-shader/x-vertex">

attribute vec2 a_position;
uniform vec2 u_offset;

void main() {
  gl_Position =
        a_position + u_offset;
}

</script>
```

WebGL Program Structure

1. Describe application page (HTML)
    a. Get a WebGL context in the js code

2. Define shaders (GLSL) - added as scripts to html/js files

3. Compute/read models and other data (JS)

4. Send data to GPU; Set state (JS)

5. Render data (JS)

To be discussed in detail in tutorials

```
<script id="fragment-shader-2d"
type="notjs">
precision mediump float;

 void main() {
      gl_FragColor =
            vec4(1,0.25,0.5,0.75);
 }

</script>
```

WebGL Program Structure

1. Describe application page (HTML)
   a. Get a WebGL context in the js code

2. Define shaders (GLSL) - added as scripts to html/js files

3. Compute/read models and other data (JS)

4. Send data to GPU; Set state (JS)

5. Render data (JS)

To be discussed in detail in tutorials

WebGL Program Structure

1. Describe application page (HTML)
   a. Get a WebGL context in the js code

2. Define shaders (GLSL) - added as scripts to html/js files

3. Compute/read models and other data (JS)

4. Send data to GPU; Set state (JS)

5. Render data (JS)

To be discussed in detail in tutorials

```
// set up viewports, common
attributes etc
...

// Fill the buffer with the values
that define a rectangle.
  var x1 = x;
  var x2 = x + width;
  var y1 = y;
  var y2 = y + height;
  gl.bufferData(
      gl.ARRAY_BUFFER,
      new Float32Array([ x1, y1,
                  x2, y1, x1, y2,
       x1, y2,  x2, y1,  x2, y2 ]),
      gl.STATIC_DRAW);
}
```

WebGL Program Structure

1. Describe application page (HTML)
   a. Get a WebGL context in the js code

2. Define shaders (GLSL) - added as scripts to html/js files

3. Compute/read models and other data (JS)

4. Send data to GPU; Set state (JS)

5. Render data (JS)

To be discussed in detail in tutorials

```
…

// set the color
gl.uniform4fv(colorLocation,
              color);


…
```

WebGL Program Structure

1. Describe application page (HTML)
   a. Get a WebGL context in the js code
2. Define shaders (GLSL) - added as scripts to html/js files
3. Compute/read models and other data (JS)
4. Send data to GPU; Set state (JS)
5. Render data (JS)

To be discussed in detail in tutorials

```
…

// set the color
gl.uniform4fv(colorLocation,
              color);

…

// Draw the rectangle.
var primitiveType =
             gl.TRIANGLES;
var offset = 0;
var count = 6;
gl.drawArrays(primitiveType,
                 offset, count);
```

WebGL Examples

Examples from webglfundamentals.org

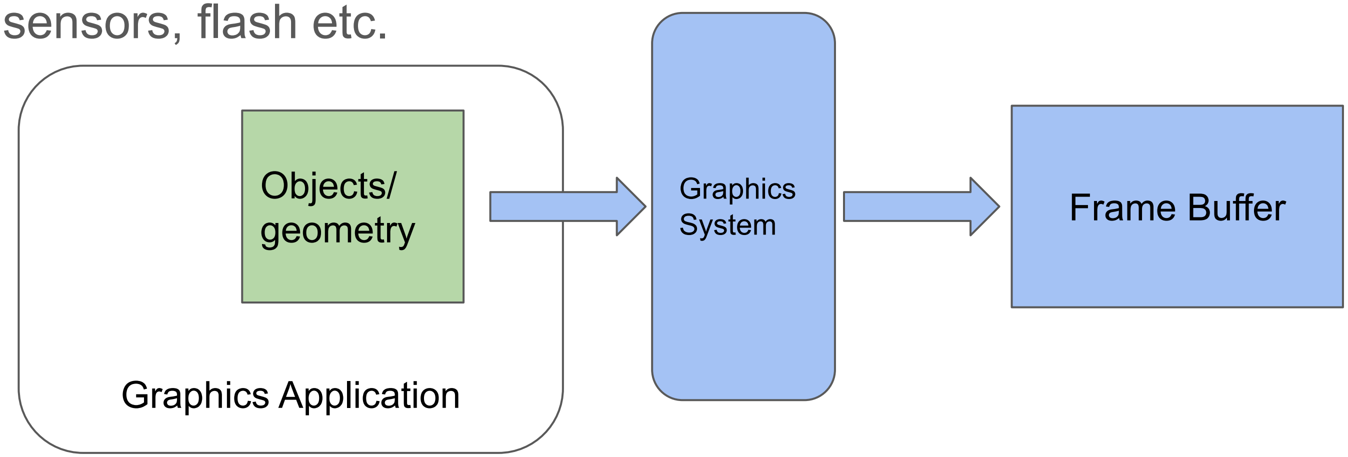
Contains programs to illustrate most basic features of WebGL (with complete code!)

Topics Covered Today

- Graphics pipeline architecture
  - 4 stages – vertex processor, clipper and primitive assembler, rasterizer, fragment processor
  - Fixed functionality vs programmable pipelines
- Overview of OpenGL libraries
- (Additional notes beyond this slide) Overview of graphics processing

Graphics Processing - Overview

- The application program defines a “scene” - collection of objects (models) that need to be rendered, along with lighting and other information that influences how objects are rendered
- Application generates geometry information for the models
- At a high level, the graphics system converts this information into an array of coloured pixels in the frame buffer. Each pixel can be assigned only one colour
- Analogous to a camera generating a 2D image of the 3D environment - using a lens, sensors, flash etc.



Realistic Rendering

What is the difference in these two paintings?

Francesco Granacci/ Michaelangelo. ~1500 AD

Papyrus Art – Egypt
~ 3000 BC

Challenges in Rendering Scenes

Realism

- Match visual perception

vietcema.com

Challenges in Rendering Scenes

Realism

- Match visual perception
- Photorealism
    - Examples: POVRay Hall of Fame

Bonsais by Jaime Vives Piqueres: POVRay

Challenges in Rendering Scenes

Realism

- Match visual perception
- Photorealism
    - Examples: POVRay Hall of Fame
- Physics-based (especially for dynamic scenes)

Performance

- Real-time
    - 30-60 frames per second. Each frame ~ 2M pixels (for a 1080p screen)
- Interactive
    - Low latency

Factors Impacting Rendering

- The scene: model objects and their relative positions and orientations
- Physical properties of the objects (related to how they are visually perceived)
- Lighting environment: types of light sources, numbers, characteristics
- Camera and view settings
- Temporal variations and Interactions between objects
- Model size: number of objects and their modeling complexity
- Screen size: pixels rendered
- Frame rate
- Desired effects and quality: e.g shadows, caustics, fog, water, …
- Expected quality - or performance/realism tradeoff
- Graphics hardware and software