Quaternions and Rasterizers/Shaders

CSE606: Computer Graphics
Jaya Sreevalsan Nair, IIIT Bangalore
February 05, 2025

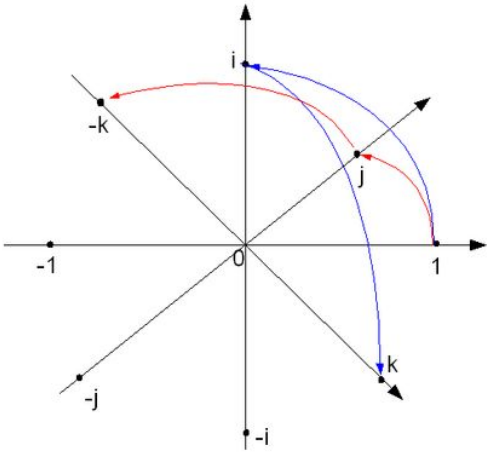
Introduction to Quaternions

Introduction to Quaternions

Quaternions: Extensions to complex numbers; lesser intuitive method for describing & manipulating rotations.

Highly recommended videos:

- https://www.youtube.com/watch?v=d4EgbgTm0Bg&t=280s
  - [What are quaternions, and how do you visualize them? A story of four dimensions. (2018) by 3Blue1Brown]
- https://www.youtube.com/watch?v=zjMuIxRvygQ
  - [Quaternions and 3d rotation, explained interactively. (2018) by 3Blue1Brown]



Graphical representation of quaternion units product as 90°-rotation in 4D-space

ij = k
ji = -k
ij = -ji

Quaternions

Using complex numbers to describe rotations, using polar representations:

$$e^{i\theta} = \cos\theta + i\sin\theta$$
$$c = a + ib = \sqrt{a^2 + b^2}\, e^{i\theta}$$

To describe, we need axis of rotation **q** and angle of rotation (q_0); where quaternion (q_0,**q**) has:

$$\mathbf{q} = \begin{pmatrix} q_1 & q_2 & q_3 \end{pmatrix}$$
$$\mathbf{q} = q_1.\mathbf{i} + q_2.\mathbf{j} + q_3.\mathbf{k}$$

Base rules:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$
$$\mathbf{ij} = -\mathbf{ji} = \mathbf{k}$$
$$\mathbf{jk} = -\mathbf{kj} = \mathbf{i}$$
$$\mathbf{ki} = -\mathbf{ik} = \mathbf{j}$$

Properties

Quaternion algebra is denoted as $\mathbb{H}$ in the honor of Sir William Rowan Hamilton, who invented it.

For $a = (q_0, \mathbf{q})$ and $b = (p_0, \mathbf{p})$,

$$a + b = (q_0 + p_0, \mathbf{q} + \mathbf{p})$$

$$ab = (q_0 p_0 - \mathbf{q} \cdot \mathbf{p}, q_0 \mathbf{p} + p_0 \mathbf{q} + \mathbf{q} \times \mathbf{p})$$

$$|a| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \sqrt{q_0^2 + \mathbf{q} \cdot \mathbf{q}}$$

Multiplicative identity of quaternion $= (1, \mathbf{0})$

Multiplicative inverse,

$$a^{-1} = \frac{1}{|a|^2} \cdot (q_0, -\mathbf{q}) = \frac{1}{|a|^2} \cdot \bar{a}.$$

where $\bar{a}$ is the conjugate of a.

Nice property of conjugation:

$$p\bar{q} = \bar{q} \cdot \bar{p}$$

Rotations Using Quaternions

If $q = (q_0, \mathbf{q})$ is such that $|q| = 1$, then q is a unit quaternion. For a unit quaternion $q^{-1} = \bar{q}$.

A pure unit quaternion is a unit quaternion $u = (0, \mathbf{u})$, which gives $u^{-1} = -u$.

For a unit quaternion q, $q_0^2 + |\mathbf{q}| = 1$. This implies that there exists a real number θ and a pure unit quaternion $u = (0, \mathbf{u})$, such that $q = (cos\theta, \mathbf{u}.sin\theta)$. Since $u^2 = -1$, the power series expansion of e^t gives: $e^{u\theta} = \sum_{n=0}^{\infty} \frac{(u\theta)^n}{n!} = cos\theta + u.sin\theta$.
Thus $q = e^{u\theta}$.

We denote the space of pure quaternions as $\mathbb{E}^3$.

Rotations Using Quaternions

Proposition: If q is a unit quaternion, then there exists a pure unit quaternion u and a real scalar θ such that $q = e^{u\theta}$. The transformation $C : \mathbb{E}^3 \to \mathbb{E}^3$ defined by $C(y) = qy\bar{q} = e^{u\theta} y e^{-u\theta}$, for $y \in \mathbb{H}$, is a rotation in the plane orthogonal to $\mathbf{u}$ through an angle 2θ.

Reference for proof: Weiner, Joel L., and George R. Wilkens. "Quaternions and Rotations in $\mathbb{E}^4$." American Mathematical Monthly (2005): 69-76.

Another interpretation of point/vector rotation using quaternions:
For a point $\mathbf{p} = \begin{pmatrix} x & y & z \end{pmatrix}$ and unit vector $\mathbf{v}$, consider

- $p = (0, \mathbf{p})$, and $r = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}.\mathbf{v} \right)$.
- $r^{-1} = \left(\cos\frac{\theta}{2}, -\sin\frac{\theta}{2}.\mathbf{v} \right)$.
- $p' = rpr^{-1} = (0, \mathbf{p}')$, where $\mathbf{p}' = \cos^2\frac{\theta}{2}.\mathbf{p} + \sin^2\frac{\theta}{2}(\mathbf{p}.\mathbf{v})\mathbf{v} + 2\sin\frac{\theta}{2}\cos\frac{\theta}{2}(\mathbf{v} \times \mathbf{p}) - \sin\frac{\theta}{2}(\mathbf{v} \times \mathbf{p}) \times \mathbf{v}$.

$\mathbf{p}'$ is the point obtained by rotating $\mathbf{p}$ by angle θ about an axis $\mathbf{v}$.

Recall: Concatenation of Transformations

Rotation about a direction vector **u** by an angle θ.

1. Move the fixed point of (object) frame to origin of reference frame.
2. Use unit vector û from origin, and use rotations about x-axis and y-axis to align the vector along z-axis.
3. Rotate by θ about z-axis.
4. Rotate back about y- and x-axes.
5. Move back the fixed point.

$$\mathbf{M} = \mathbf{T}(p_0)\mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathbf{T}(-p_0)$$

$$\hat{\mathbf{u}} = [\,\alpha_x \quad \alpha_y \quad \alpha_z\,]^T,$$

$$d = \sqrt{\alpha_y^2 + \alpha_z^2} \text{ and } \sqrt{d^2 + \alpha_x^2} = 1,$$

$$\theta_x = \cos^{-1}\left(\frac{\alpha_z}{d}\right) \text{ and } \theta_y = \cos^{-1}(d).$$

Benefits of Quaternions

Quaternion product from r and p: An alternative to matrix transformations using multiplication.

- Lesser number of operations.
- Faster implementations as, quaternions have been built into both hardware and software implementations.
- Smoother sequences of rotations for animations, as opposed to rotation matrices – especially smoother interpolations to resolve *gimbal lock.*

Euler's Rotation Theorem, Euler's Angles

Euler's Theorem on rotation states that any two independent orthonormal n-dimensional coordinate frames can be related by a sequence of no more than n rotations about basis vectors such that consecutive rotations are about distinct basis vectors.

Another version of the theorem: Any 3-dimensional rotation which is about an arbitrary axis can be described using three angles. These three angles are called Euler angles. A = BCD, as given in the image:

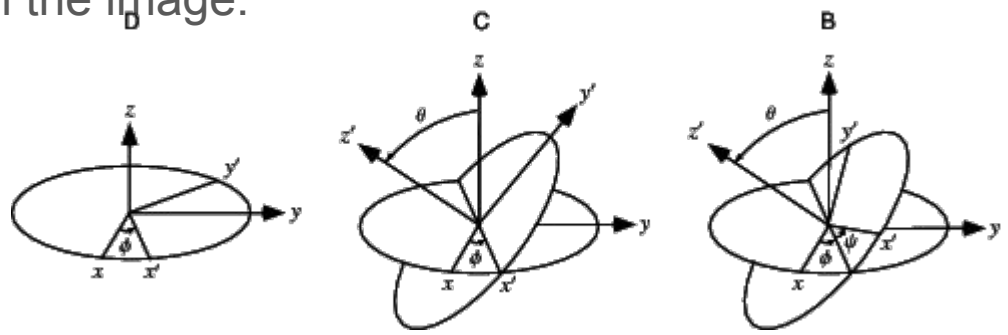
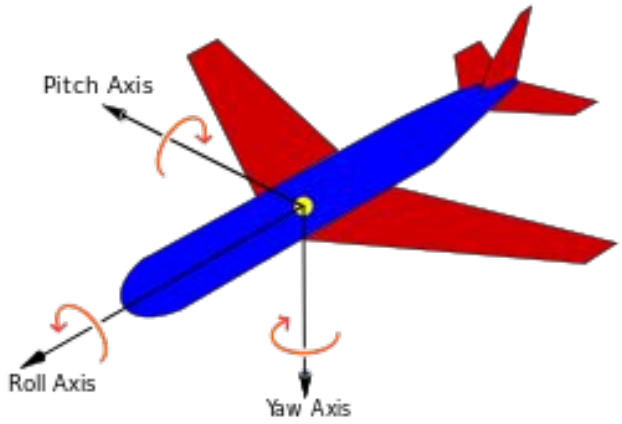


Image courtesy: http://mathworld.wolfram.com/EulerAngles.html

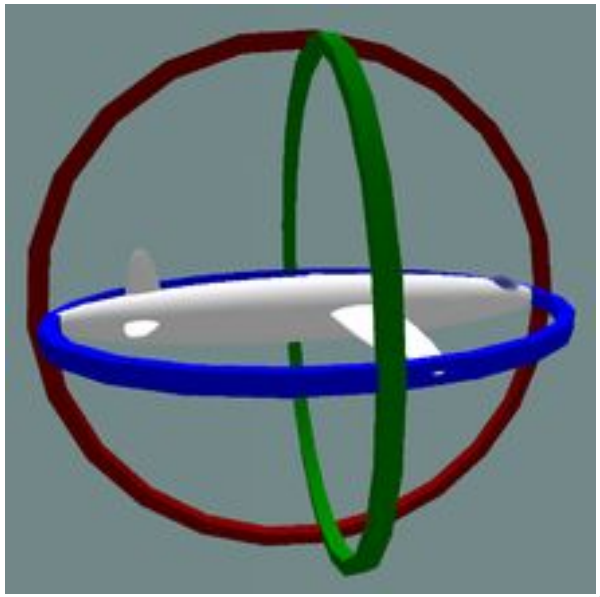
Gimbal Lock

(Top) Euler angles (3 degrees of freedom); no gimbal lock;
(Bottom) gimbal lock.
Image courtesy: Wikipedia



Gimbal lock occurs when, for an orientation, there is a loss of one or more degrees of freedom.

Recommended video:
https://www.youtube.com/watch?v=Mm8tzzfy1Uw
[Gimbal Lock (2015), by Fabio Garcia Rojas].



Gimbal Lock and Quaternions

Euler angles cannot resolve gimbal lock since it is a 3-dimensional (3D) construct of the 3D space; where quaternions is a 4D one.

Interpolation using Euler angles does not help in the case of gimbal lock since the angles are being linearly interpolated from the initial (source) key orientation to the final (destination) one.

In quaternions the interpolations are done based on the curve (geodesic) on the 4D hypersphere of the unit quaternion, connecting the source and destination orientations. At each interpolated quaternion value, one finds the corresponding rotation matrix.

Alan Watt and Mark Watt, “Advanced Animation and Rendering Techniques: Theory and Practice

Rasterizers

Section Agenda

Overview

- Mesh surfaces - recap
- Graphics pipeline
  - Vertex processing
  - Rasterization
  - Fragment processing
- Shaders
  - Shader language: GLSL
  - Data flow and Buffers
  - Vertex shader
  - Fragment shader

Mesh Surfaces

Model such objects directly as a 2D mesh:

Surface defined by set of vertices, and connectivity between vertices that define triangles or quadrilaterals

For the purposes of rendering, we can assume all models are defined as mesh surfaces.

Even when defined as geometric primitives, the application converts them to mesh representations prior to rendering

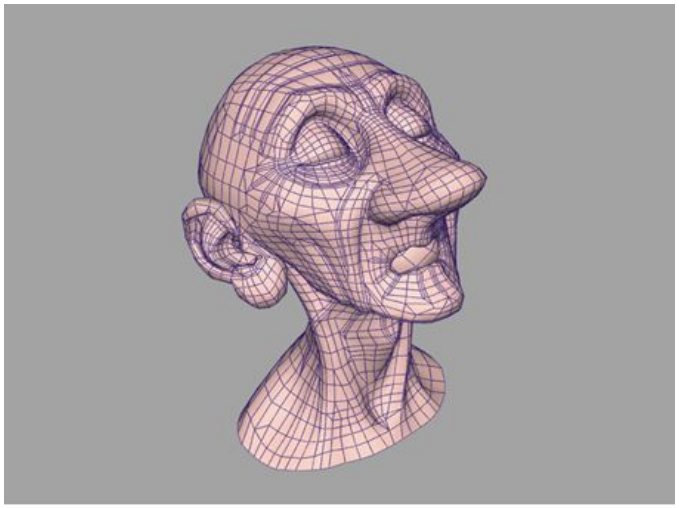


Image from: *Subdivision Surfaces in Character Animation*, Tony DeRose, Michael Kass, Tien Truong; Pixar Animation Studios

Mesh Surfaces

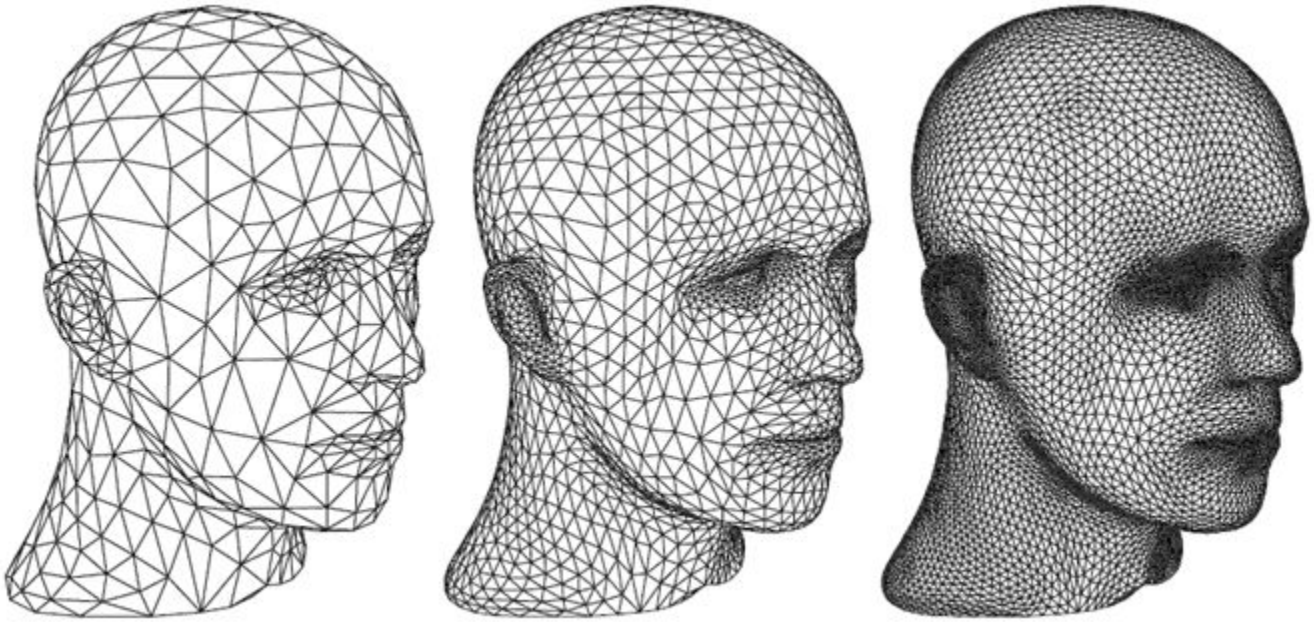
Mesh representation - vertices and connectivity - provides information on the shape of the object.

- Coordinates of vertices
- Triples of vertices defining triangular mesh elements

Additional computations needed to compute properties such as surface normal and curvature

1. Normal at a vertex can be computed as a weighted average of the normals of the mesh elements (triangles) at that vertex
2. Use refinement techniques to compute better approximations of the mesh (e.g. subdivision schemes) and use these to compute normals etc

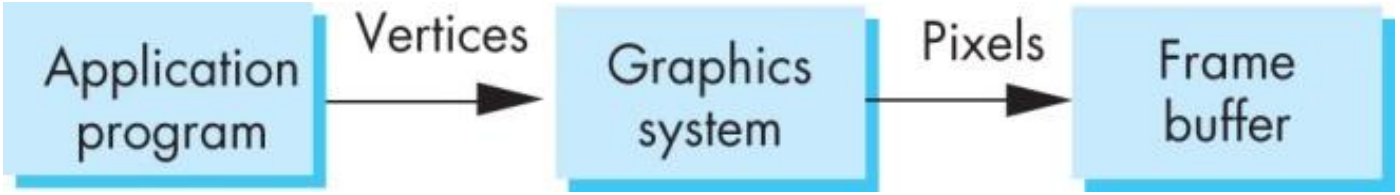
Refinement of Surface Meshes



The second and third images are produced by subdividing each triangle of the previous image into 4 triangles using a specific rule (Loop subdivision). SIGGRAPH 2000 Course Notes on Subdivision for Modeling and animation, D. Zorin and P. Shroeder

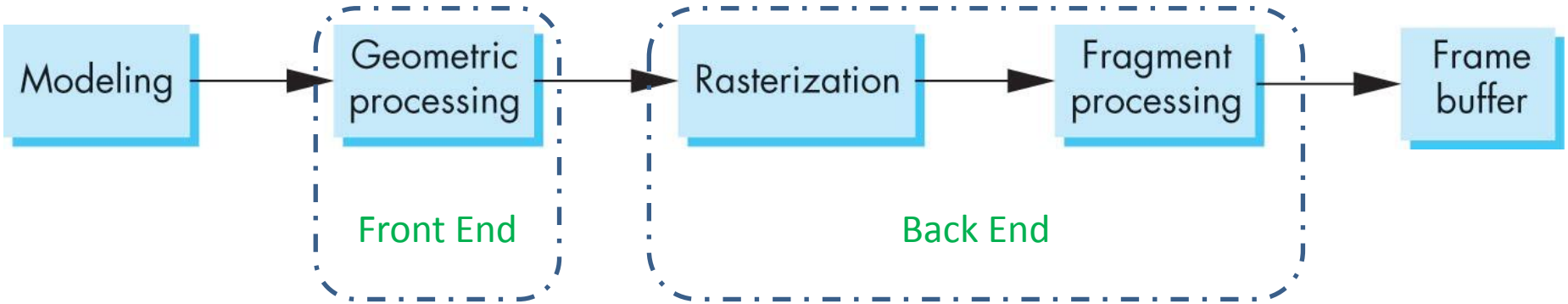
Graphics Processing: Overview

- The application program generates vertices (with connectivity and state information)
- At a high level, the graphics system converts this information into an array of coloured pixels in the frame buffer
- Tasks within the graphics system include:
  - Transformations, clipping,, projection, shading, hidden-surface removal, rasterization
  - Process each primitive and light source, and assign colour to every pixel



Implementation Strategies

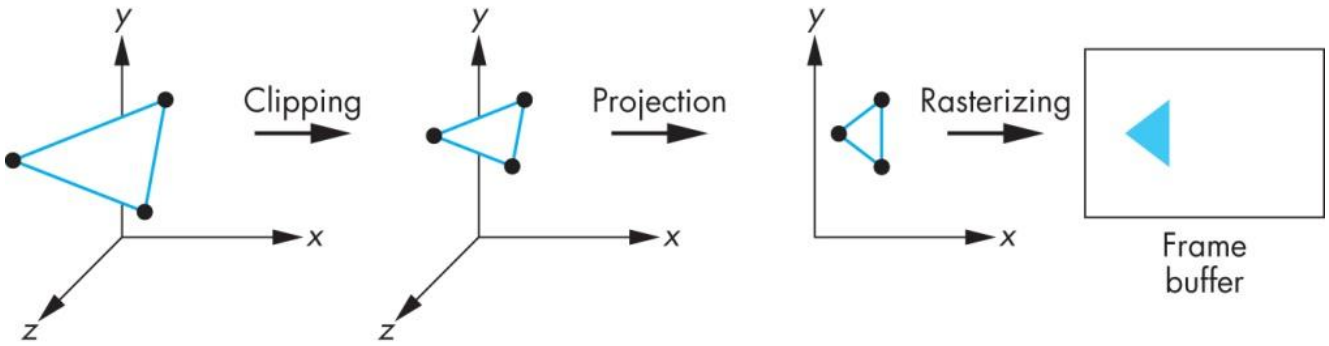
- **Object oriented**:
  - Iterate through each object in the system
  - Process each through a sequence of modules that perform successive stages of the graphics process
- **Image oriented:**
  - Process each pixel to determine its colour
  - Determine the geometric primitive that affects this pixel
  - Ray tracing as an example



Graphics Pipeline Stages

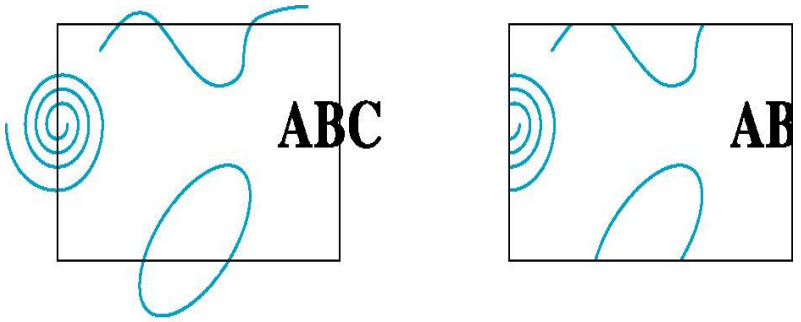
For each triangle (polygon) of the model (defined in the application):

- Transform each vertex to view volume
- Clip and compute primitives (triangles)
- Project to screen coordinates
- Rasterize - map to locations in frame buffer
- compute color for each pixel



Clipping

- Clipper decides which parts of a primitive (line, polygon etc.) are potentially visible and should be accepted (for further processing), or are not visible and can be rejected (culled).
- Examples: 2D against clipping window, 3D against clipping volume
- Easy for line segments and polygons. Efficient algorithms such as Cohen-Sutherland, Liang-Barsky and variants provide for efficient hardware implementation
- For curves and text, convert to lines and polygons first
- In OpenGL, clipping done before rasterization



Rasterization

Moving from normalized device coordinates to window coordinates. Focus on 2-d coordinates in screen space

Rasterization (scan conversion):

- Determine which pixels are inside primitive specified by a set of vertices
- Produces a set of fragments (potential contributors to pixel color)
- Fragments have a location (pixel location) and other attributes such as color and texture coordinates that are determined by interpolating values at vertices

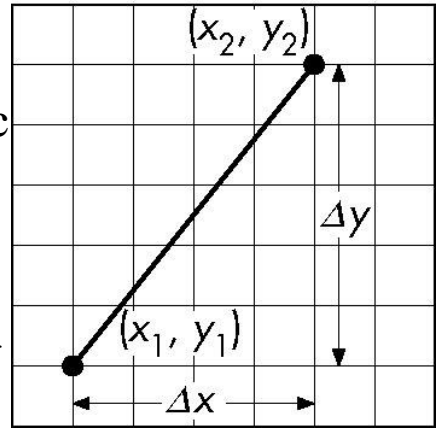
Pixel colors determined later using color, transparency, texture, and other fragment input properties, as well as interaction with other fragments that map to the same pixel location

Scan Conversion: Line Segments

- Line segment in window coordinates with integer values for endpoints
- Need to compute (x,y) values of fragments that correspond to this line
- Simple approach (Digital Differential Analyzer):
  - For each successive value of x, compute y by adding m and rounding to integer (Note: x,y are integers)

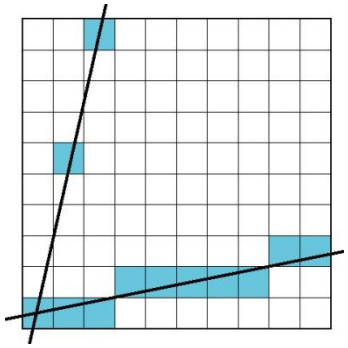
$y = mx + c$

$m = \frac{\Delta y}{\Delta x}$

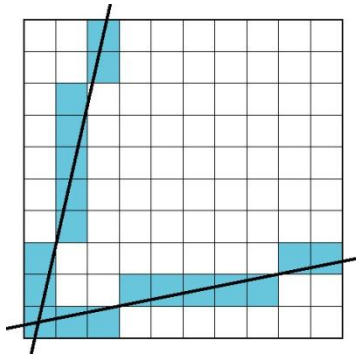


Scan Conversion: Line Segments

- Does not work well if slope > 1
  - Can swap role of x and y in that case



- Can we avoid floating computations per pixel?
  - Enable simpler/cheaper hardware
  - Algorithms like Bresenham's optimize computations



Generating Fragments

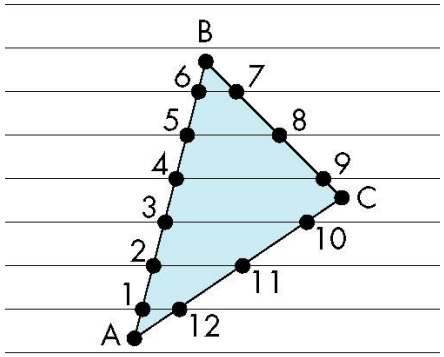
A fragment is a potential pixel

Computed at end of pipeline

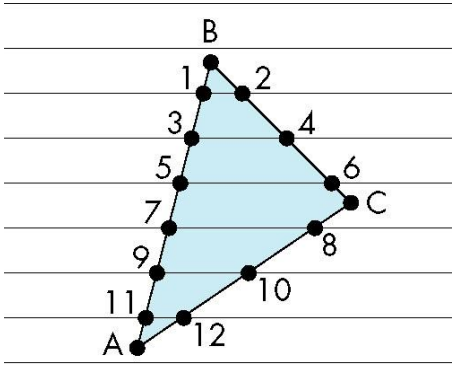
- Convex Polygons only
  - Non-convex polygons assumed to have been tessellated
- Vertex properties available (e.g. from vertex shader)
- March across scan lines interpolating properties
  - Incremental work small
- Combine with z-buffer algorithm

Scan Line Filling

1. Compute intersection of line segments with scan lines
2. Sort by scan line
3. Fill in each scan - compute pixels between intersection points per scan



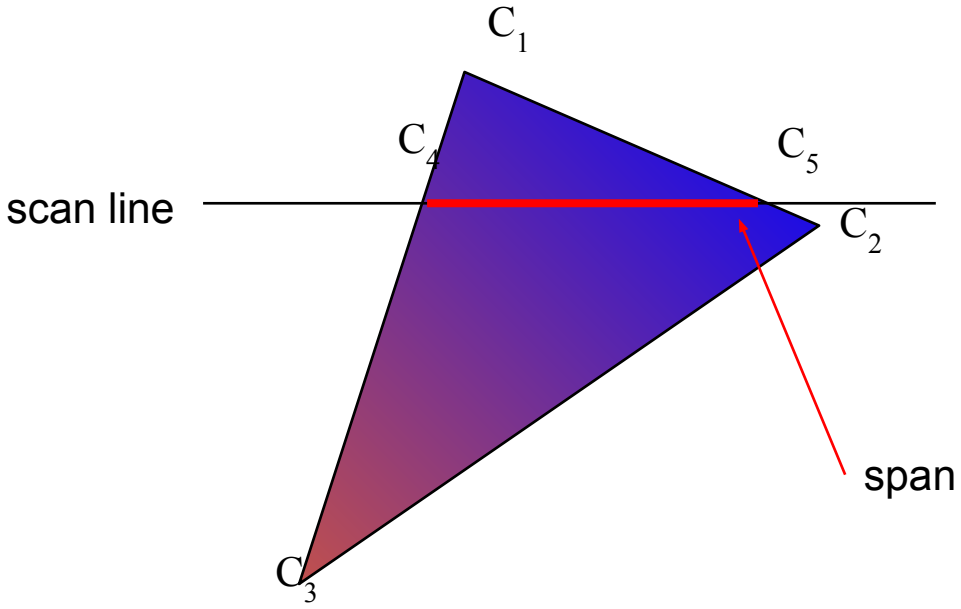
vertex order generated
by vertex list



desired order

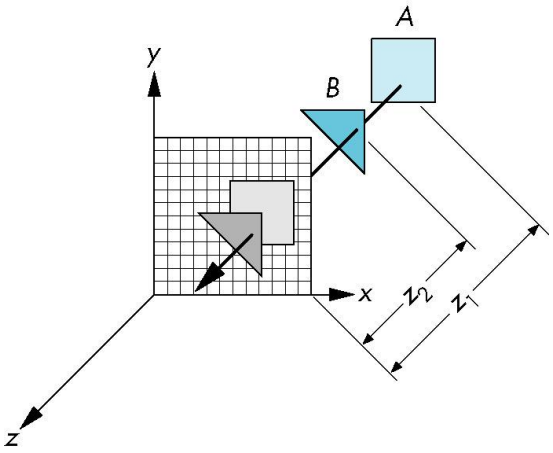
Interpolating properties

- C_1 C_2 C_3 specified as output of vertex shader
- C_4 determined by interpolating between C_1 and C_2 (e.g. Bresenham's)
- C_5 determined by interpolating between C_2 and C_3
- Interpolate between C_4 and C_5 along span
- Position and properties through linear interpolation



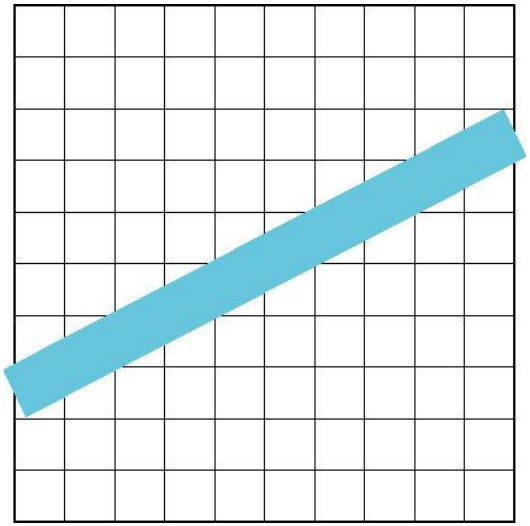
Integrating Z-buffer

- Z or depth buffer stores the depth of the closest (to the camera) fragment at each pixel found so far
- As we render each fragment, compare its depth to that stored in the z buffer for that pixel
- If less, place shade of pixel in color buffer and update z buffer
- Interpolation of z-values can also be computed along with those of (x,y)
- Thus integrate z-buffer computations into normal fragment processing – combines shading and hidden-surface removal

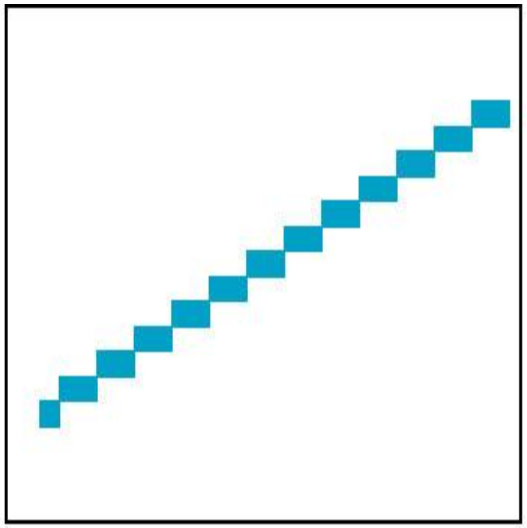


Aliasing

- Ideal rasterized line should be 1 pixel wide
- Choosing best y for each x (or vice versa) produces aliased raster lines (multiple lines map to the same set of pixels)



Desired



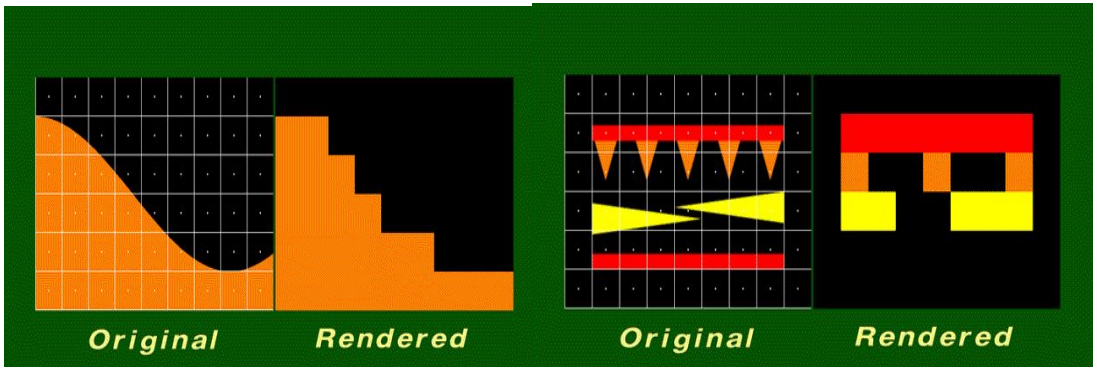
Rendered

Aliasing - artefacts and loss of detail

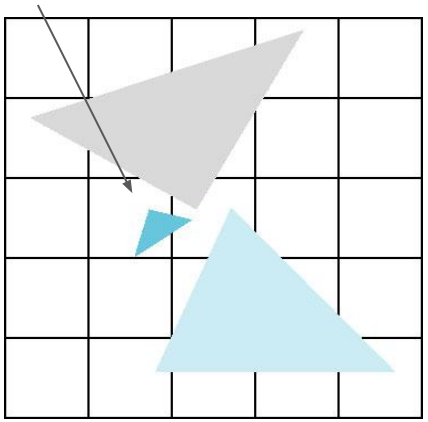
Depending on sampling technique used, can produce many undesirable effects.

Anti-aliasing techniques can help mitigate such rendering effects

- broadly involves additional sampling and appropriate filtering based on neighbor pixels



All three polygons should contribute to color



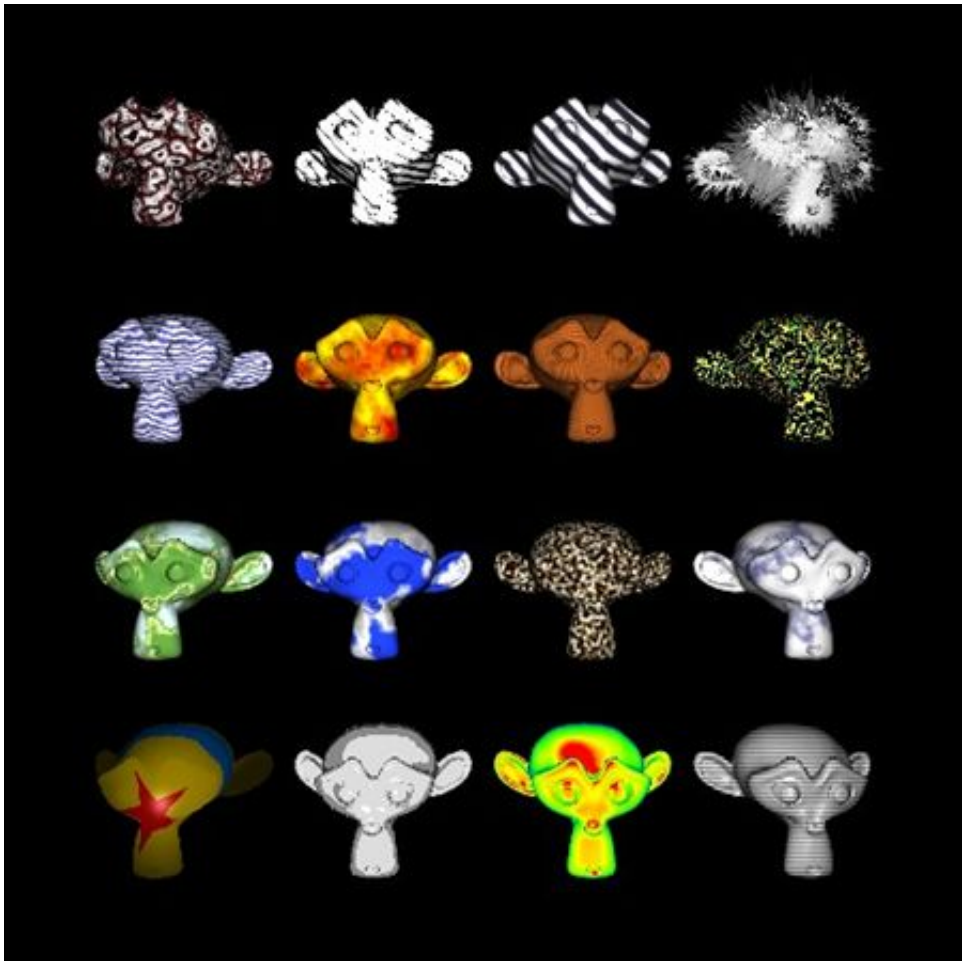
Tomas Akenine-Möller, "*Real Time Rendering*"

Shader Programming

Flexibility in Rendering

How does one produce different rendering effects for the same model. Fixed geometry but different colors/patterns/lighting effects across the elements of the model.

Need a flexible mechanism that provides low-level, granular (pixel level) control to the program

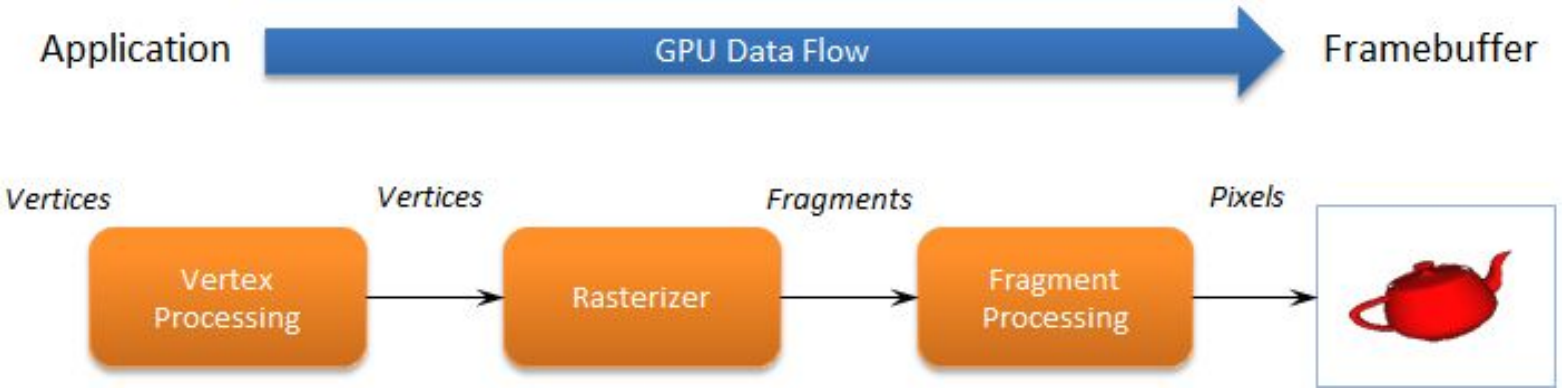


Programmable Graphics Pipeline

Simplified view of graphics pipeline:

- Transform vertices from model to view space (clipping coordinates)
- Clip and rasterize
- Compute contribution of fragments to pixels of the framebuffer

How do we allow the application (running on the CPU) to extend or modify these steps (which are computed on the GPU)?

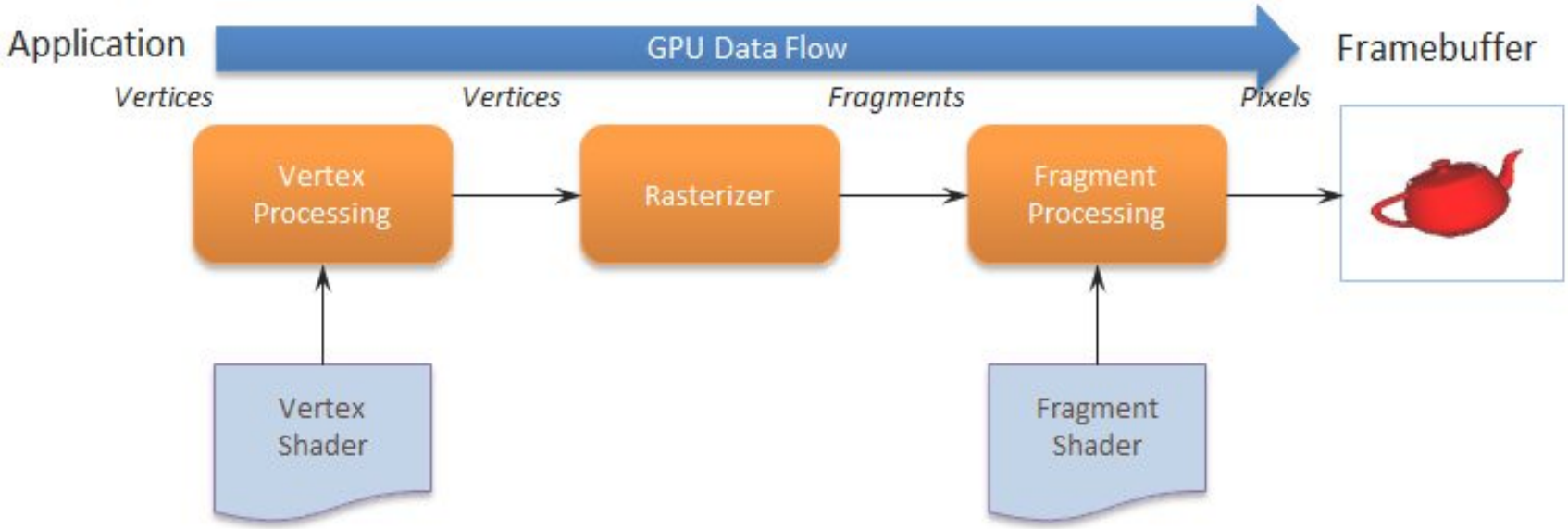


From Graphics “Engine” to “Framework”

Implement a program or “shader” that is to be invoked by the graphics system on each data block - for key stages of the pipeline

In essence, a graphics processing **framework**

- Defines the flow of data across stages (the “plumbing”)
- Application defines the behaviour at each shader
  - Specify the output data of the shader

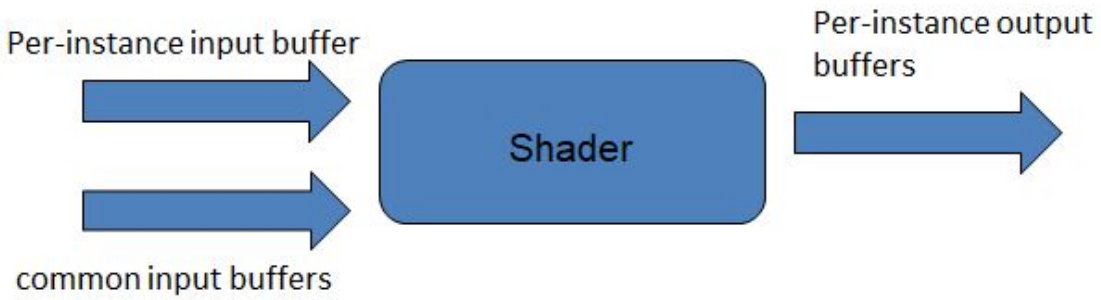


Structure of Shaders

An instance of the shader program runs on each processing element of the GPU. Multiple instances running in parallel.

Shader accesses data from shared buffers (SIMD). Two types of buffers:

- Per-instance input or output
- Common input to all instances



Structure of Shaders: Buffers

Input buffers - read-only access:

Per-shader instance, set up by application or previous shader/stage in pipeline

- e.g. vertex coords, normals, color, etc.

Common to all shaders of this stage, typically set up by application

- e.g. model view transformation, texture map

Output buffer:

- Accessible by application, or input to subsequent shaders. e.g. transformed vertex coordinates, pixel RBGA

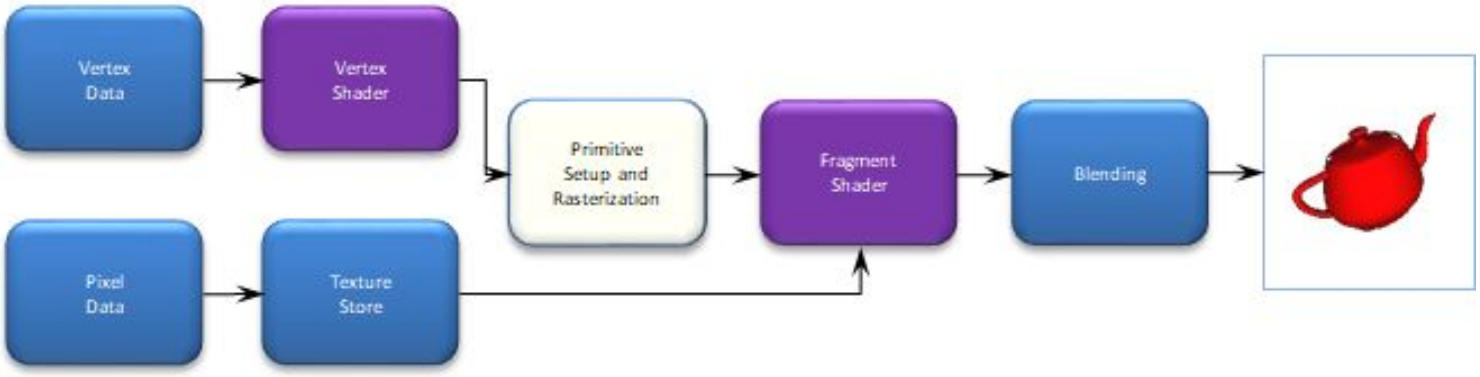
Each instance of the Shader program processes input data (buffers) and generates output data (buffers)

Graphics Library Shader Language (GLSL)

C-like, for implementation of shader programs

Convenient vector and matrix representations: vec2, vec3, vec4, mat2, mat3, mat4, etc. Supports operations on vectors and matrices, e.g. vector*matrix, matrix*matrix. Also has trigonometric and other useful math functions

Different types of buffers: **attribute (**per-instance input), **uniform (**common input), **varying** (per instance output and input)

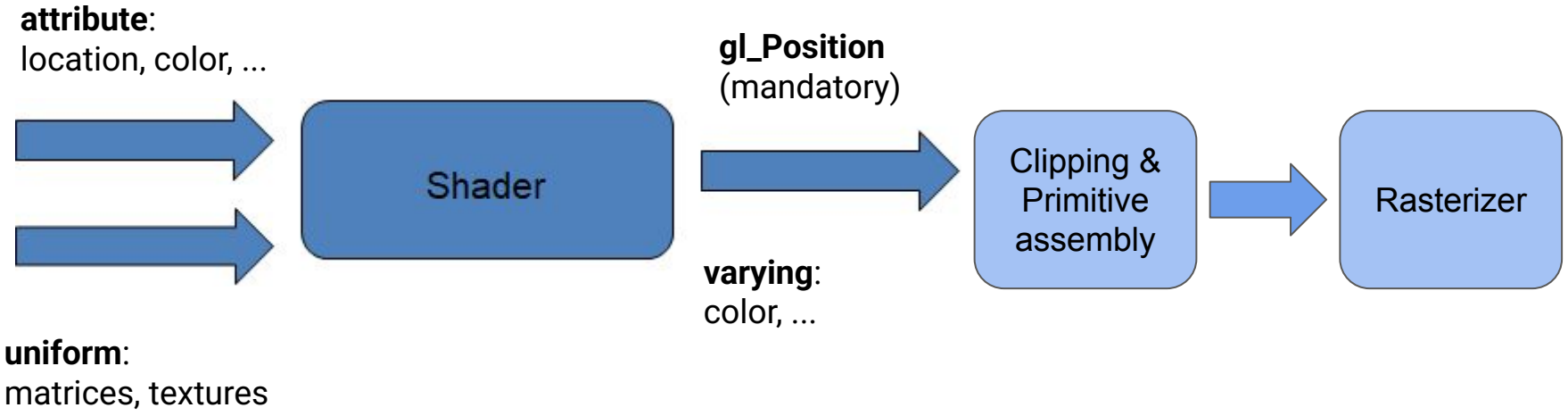


Vertex Shader

Invoked once for each vertex to be processed.

Instance specific data - world coordinate location and other input data - accessed from **attribute** buffers. Common data from **uniform** buffers

Should output **gl_Position** - location in clipping coordinates. Optionally, other values as **varying** buffers - to be fed to next process step/shader

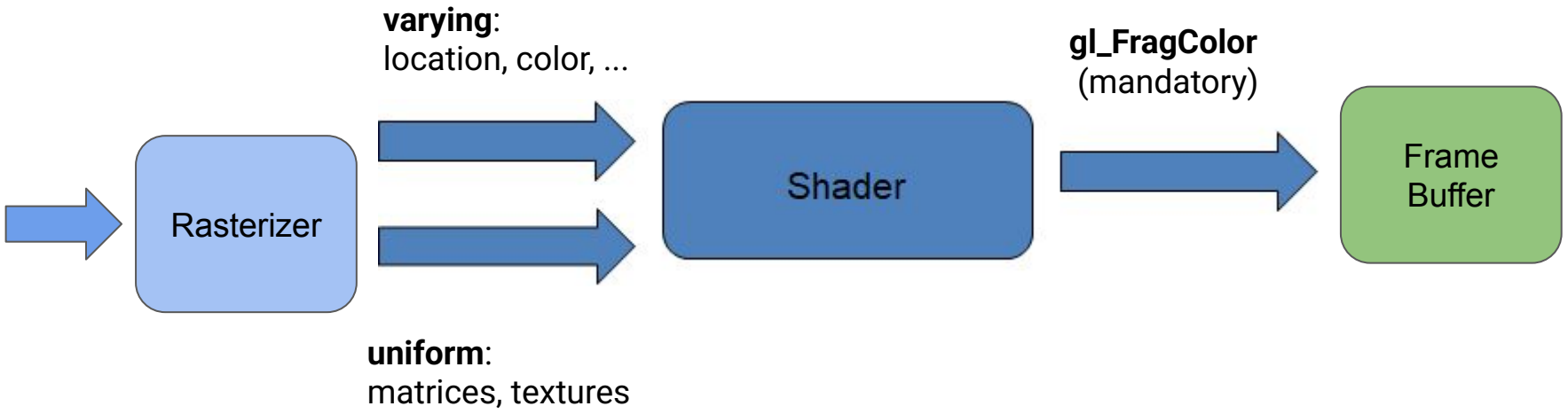


Fragment Shader

Invoked once for each fragment (a potential pixel). Fragments generated by Rasterizer, which also interpolates **location** and **varying** buffers from those of the corner vertices of the (triangle) primitive

Inputs accessed through **varying** or **uniform** buffers

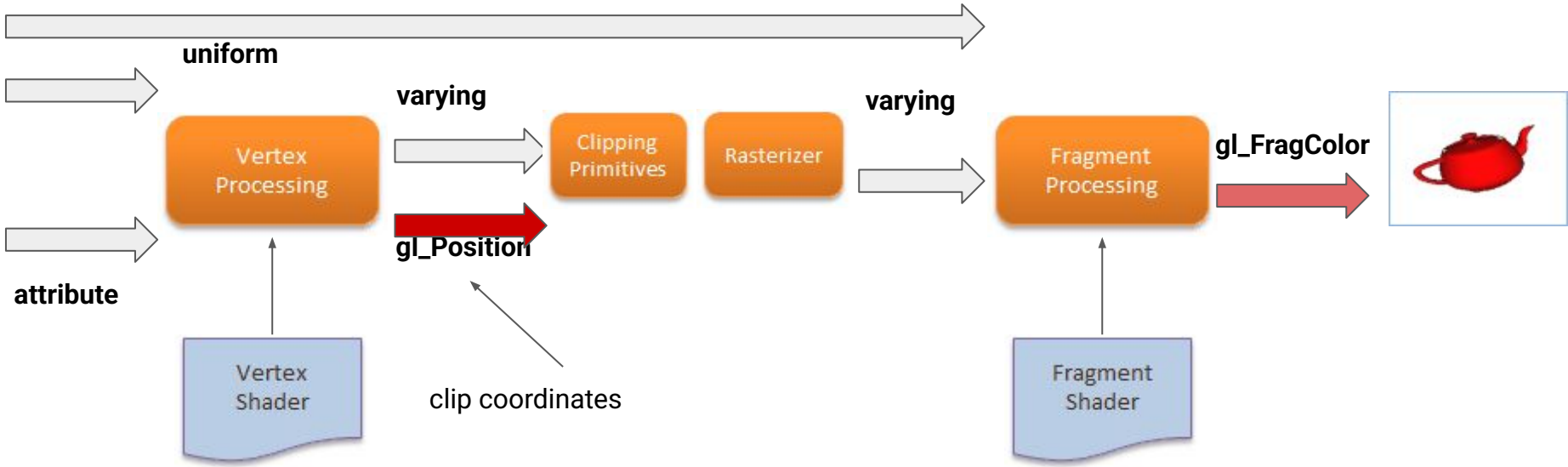
Should compute the color of the fragment: **gl_FragColor**.



Shaders in the Pipeline

Data flows through stages of the pipeline, with appropriate shader instances getting invoked at each stage - in parallel.

OpenGL 4.x has introduced many other shaders such as Geometry, Tessellation shaders. Now also ray-tracing shaders



Structure of OpenGL/WebGL Programs

Define Shaders in GLSL
(vertex and fragment)

Set up viewport, etc

Initialize:

for each object:

Set up data buffers

populate with data
(coordinates, color, matrices etc)

Render:

for each object

Set up attributes/uniform:
(vertex and other data)

Attach shaders*

Draw buffer/elements
(triggers shaders through the pipeline processing - runs on the GPU)

** if common for all objects, can be done at initialize*

Sample Shaders

```
// vertex shader
attribute vec4 a_location;
attribute vec4 color;
varying vec4 vcolor;
main() {
    gl_Position = a_location;
    vcolor = color;
}

// fragment shader
varying vec4 vcolor;
main() { // vertex shader
    gl_FragColor = vcolor;
}
```

For the fragment shader, **varying** buffer *vcolor* is computed as the linear interpolation of the *vcolor* of the vertices of the triangle.

If **attribute** *color* has a constant color for all vertices of a triangle, then the output will be triangle of uniform color, else, will vary uniformly over the triangle.

Sample Shaders

```
// vertex shader
attribute vec4 a_location;
attribute vec4 color;
uniform mat4 final_matrix;
varying vec4 vcolor;
// other output such as normal

main() {
    gl_Position =
      final_matrix * a_location;
    vcolor =
    <compute using normal, etc>;
}
```

```
// fragment shader
varying vec4 vcolor;
// other input such as normal

main() { // vertex shader
    gl_FragColor =
     <compute based on inputs>;
}
```

Summary

- Quaternions
  - Introduction (2 videos)
  - Equivalence to rotation about arbitrary axis
  - Euler angles
  - Gimbal lock (1 video)
- Rasterizers
- Shaders