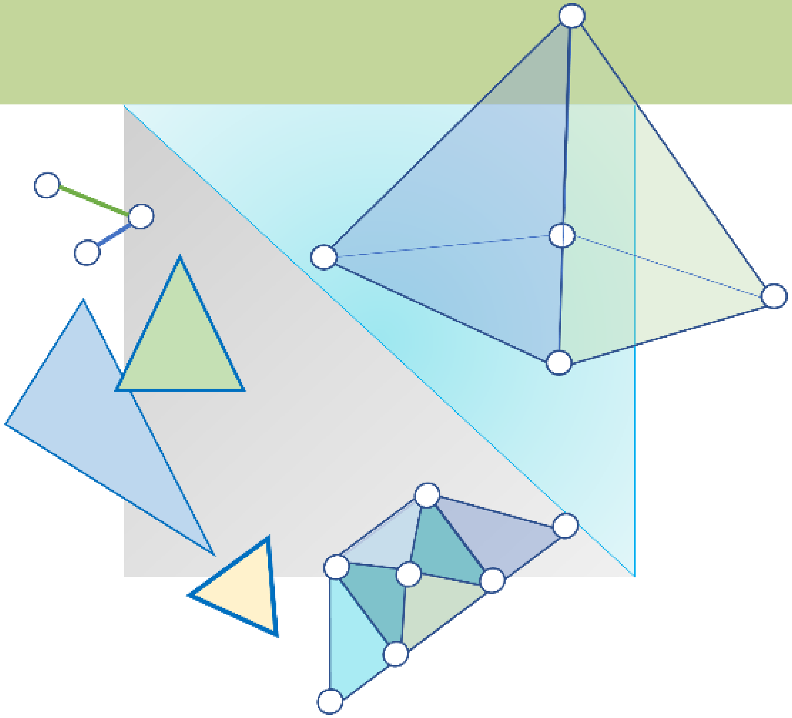
CITS3003 Graphics & Animation

Lecture 3:
Pipeline Architecture

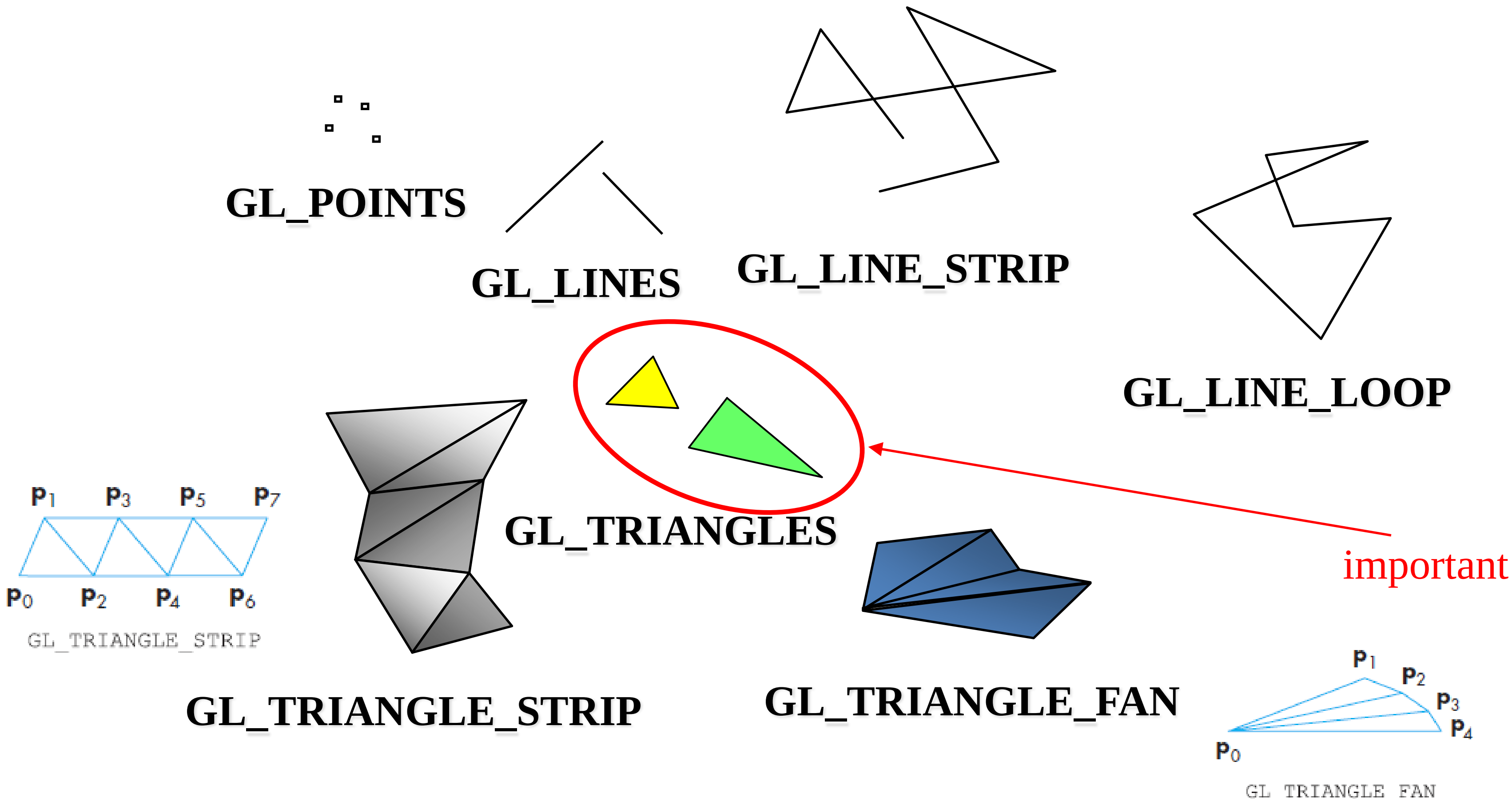


Content

- Expanding on primitives
- Vertex attributes
- OpenGL pipeline architecture
- Understand immediate mode graphics vs retained mode graphics

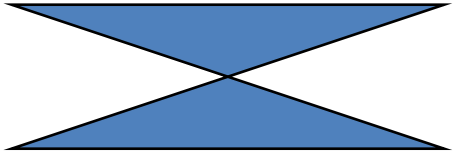
OpenGL Primitives

Recall from a previous lecture…



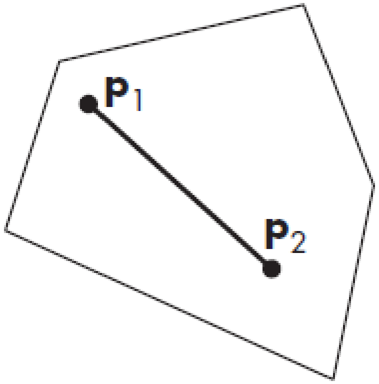
Polygon Issues

- Graphics systems like triangles because triangles are:
  - **Simple**: edges cannot cross
  - **Convex**: All points on a line segment between two points in a polygon are also in that polygon
  - **Flat**: all vertices are in the same plane



Non-simple polygon

nonconvex polygon

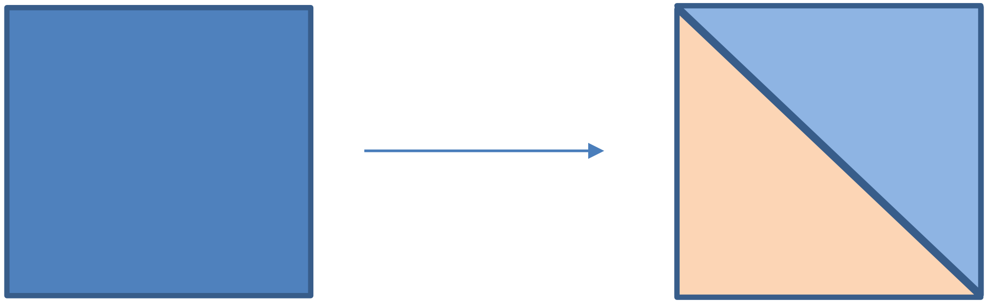


convexity

Polygon Issues (cont.)

- If other polygons are used, they are tessellated into triangles (a.k.a *triangulation*)
- OpenGL contains a tessellator.

Tessellation (tiling) of a flat surface is the process of covering it with one or more geometric shapes (the tiles): Wikipedia

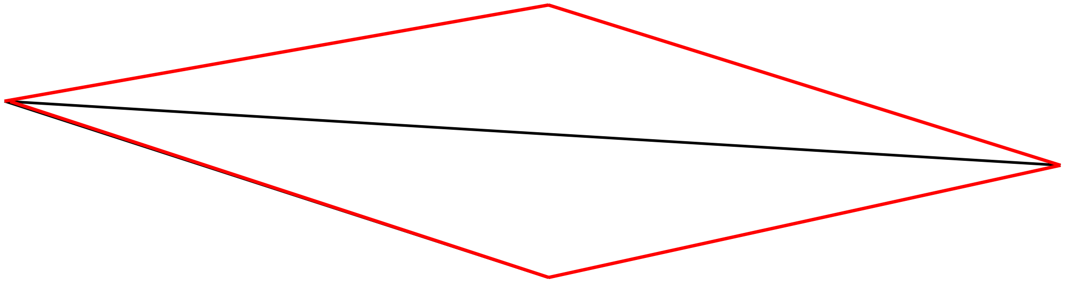


Polygon Testing

- **Polygon testing** refers to testing a polygon for its simplicity and convexity
- Conceptually it is a simple procedure, however, it is time consuming
- Earlier versions of OpenGL assumed both and left the polygon testing to the application
- OpenGL renders triangles
  - Need algorithm to triangulate an arbitrary polygon

Triangulation
Good and Bad Triangles

- Long thin triangles render badly



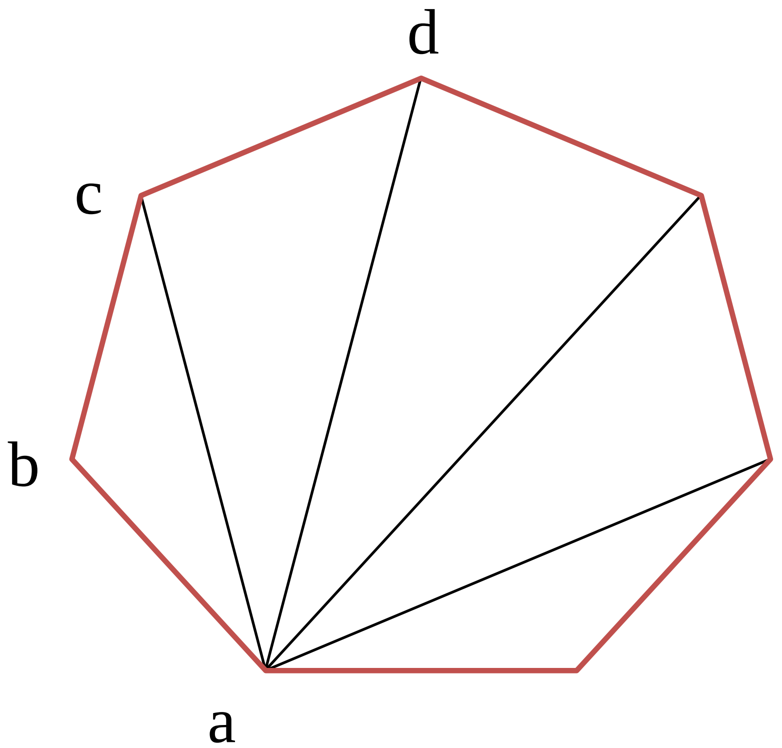
- Equilateral triangles render well
- To get good triangles for rendering

➔ Maximize the minimum interior angle

- **Delaunay triangulation** (very expensive) can be used for unstructured points

Recursive Triangulation of Convex Polygon

- If the polygon is convex, then we can recursively triangulate it to form triangles:
  1. Start with abc to form the 1st triangle, then
  2. Remove b (the resultant polygon has one fewer vertex
  3. (Recursion) Go to Step 1 to form the 2nd triangle



- Does not guarantee all triangles are good.
- Convexity helps in easy triangulation

Attributes of Primitives

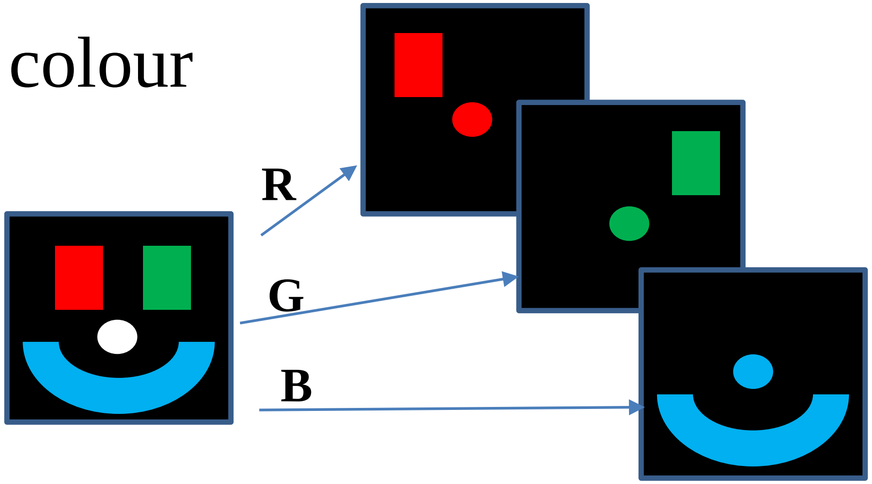
Recall from a previous lecture…

- Attributes are properties associated with the primitives that give them their different appearances, e.g.
  - Color (for points, lines, polygons)
  - Size and width (for points, lines)
  - Stipple pattern (for lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges
    - Display vertices

RGB Colour

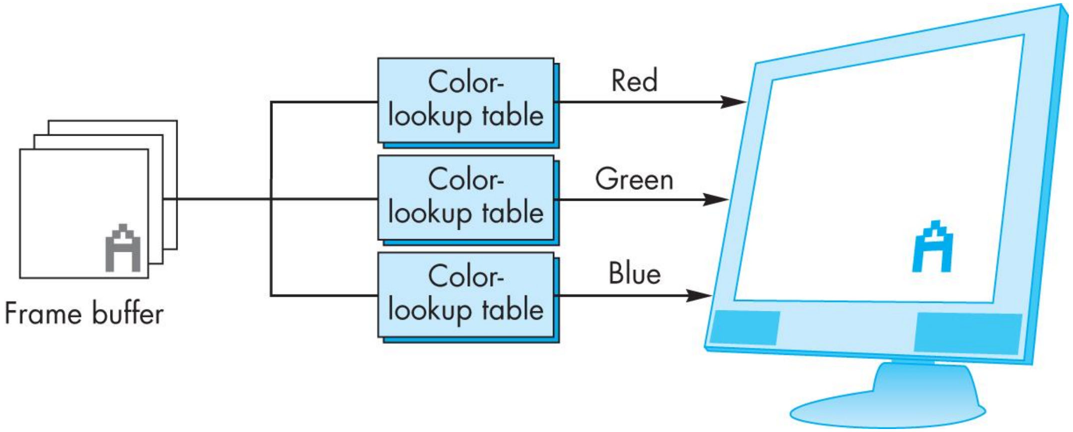
- Each colour component is stored separately in the frame buffer
- Occupies 8 bits per component in the buffer
- Colour values range
    - from 0 to 255 using unsigned integers, or
    - from 0.0 (none) to 1.0 (all) using floats
- Use vec3 or vec4 to represent colour

```
vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
```



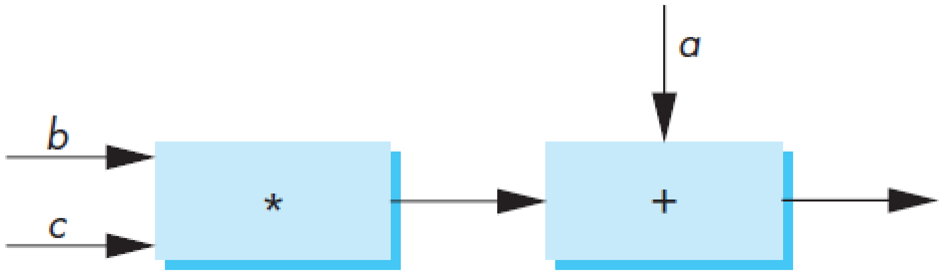
Indexed Colour

- Colours are indices into tables of RGB values
- Requires less memory
  - not as important now
    - Memory inexpensive
    - Need more colors for shading



Pipeline Architectures

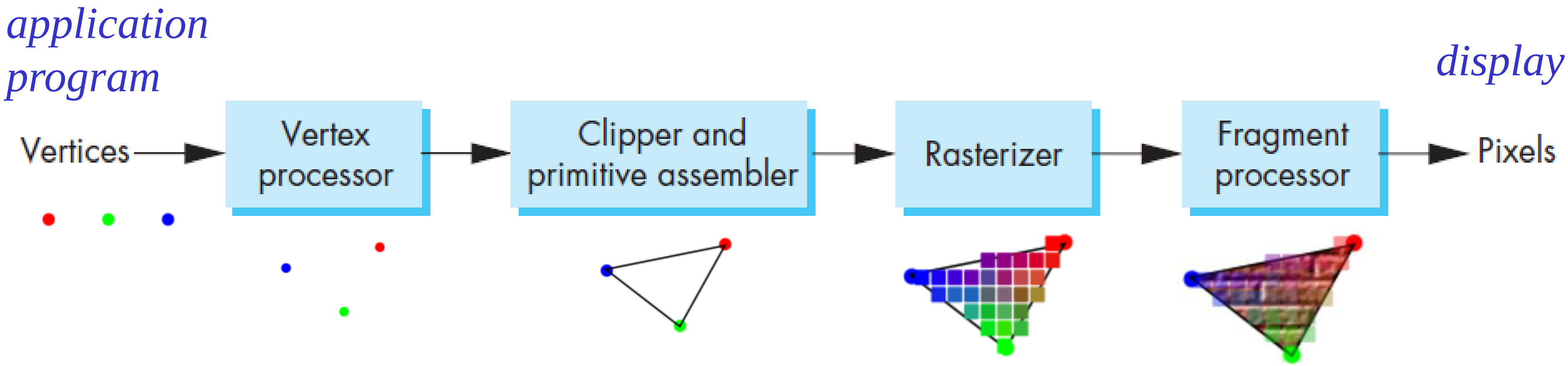
- Pipeline architectures are very common and can be found in many application domains. E.g., an arithmetic pipeline:



- When two sets of a, b, and c values are passed to the system, the multiplier can carry out the 2nd multiplication without waiting for the adder to finish → the calculation time is shortened!

The Graphics Pipeline

- The Graphics Pipeline adopts:

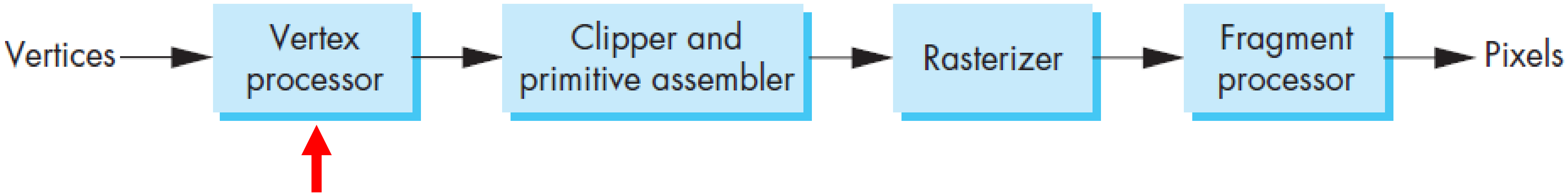


- Objects passed to the pipeline are processed one at a time in the order they are generated by the application program
- All steps can be implemented in hardware on the graphics card

Image credits: Author: Romain Vergne link

Vertex Processing

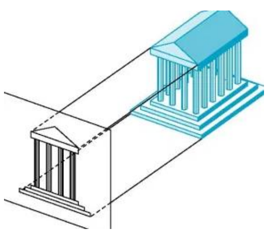
- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates
- Every change of vertex coordinates is the result of a matrix transformation being applied to the vertices
- Vertex processor can also compute vertex colors

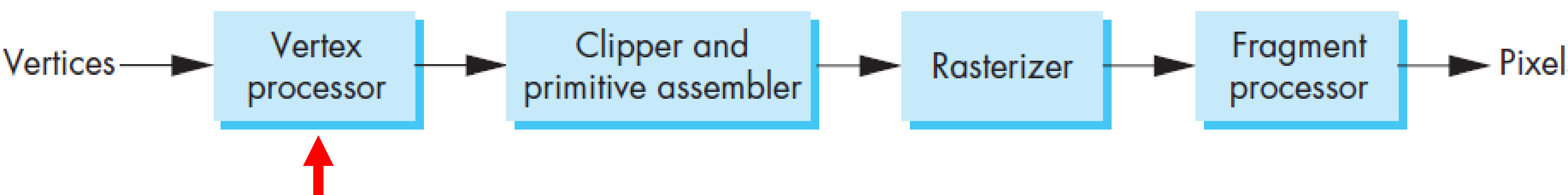


Projection

- *Projection* is the process that combines the 3D viewer with the 3D objects to produce the 2D image

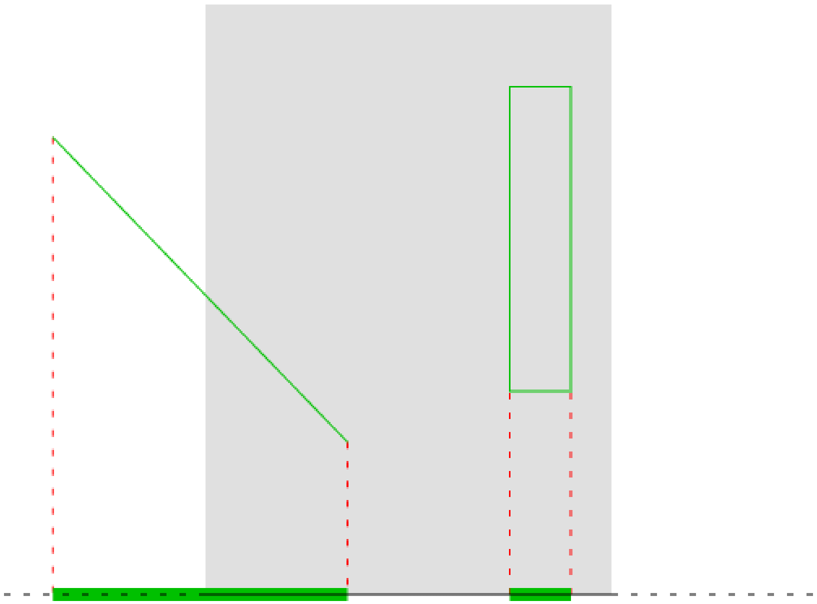
  - **Perspective projections:** all projected rays meet at the center of projection
  - **Parallel projection:** projected rays are parallel; centre of projection is at infinity. (specify the direction of projection instead of the centre of projection)



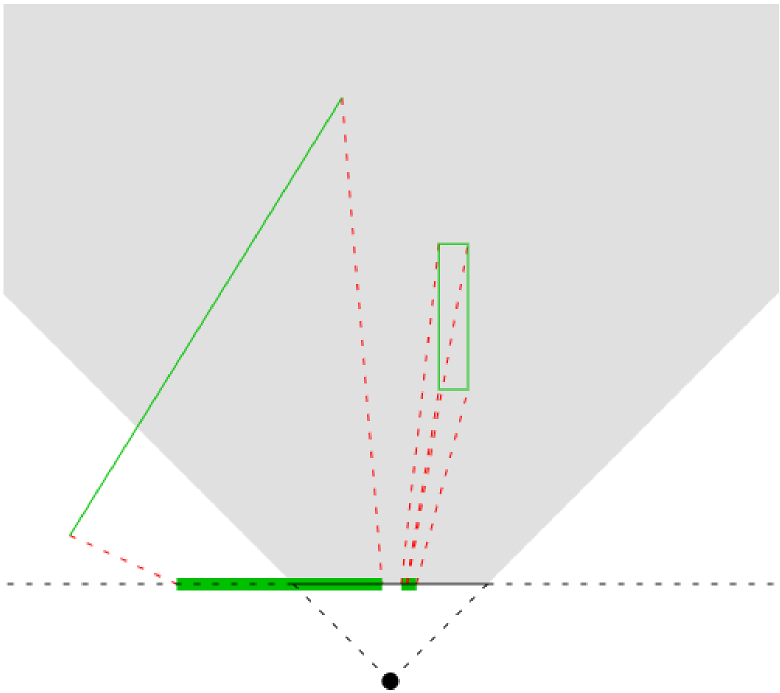


Projection

- Example



2D to 1D Orthographic/Parallel Projection



2D to 1D Perspective Projection

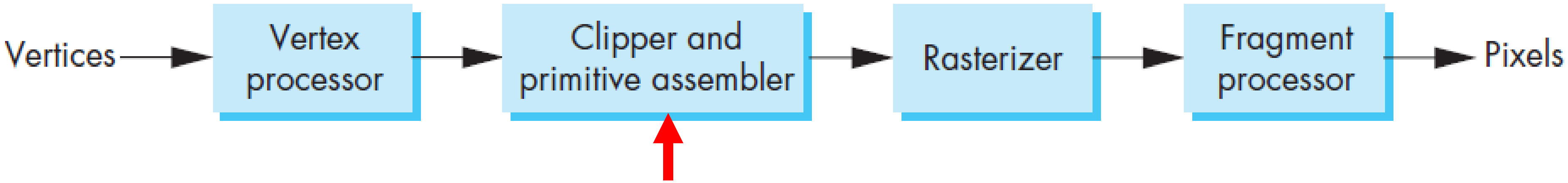
The gray box represents the part of the world that is visible to the projection; parts of the scene outside of this region are not seen

Credits: [link]()

Primitive Assembly

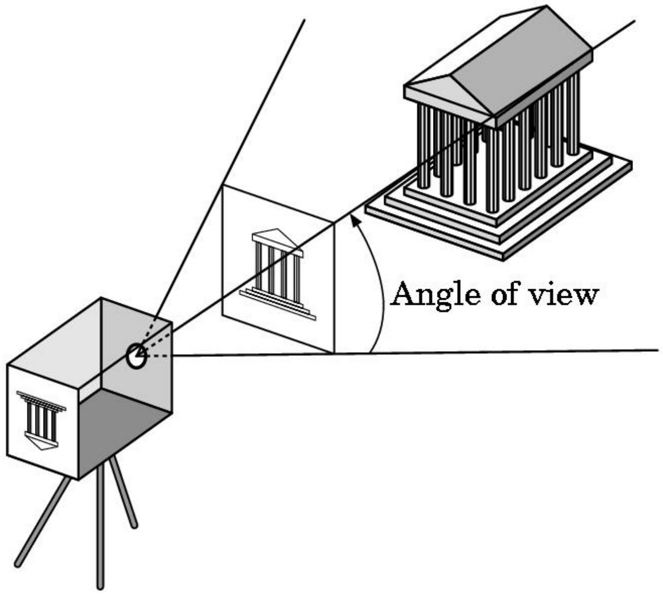
- Vertices must be collected into geometric objects before clipping and rasterization can take place.
  - Line segments
  - Polygons
  - Curves and surfaces

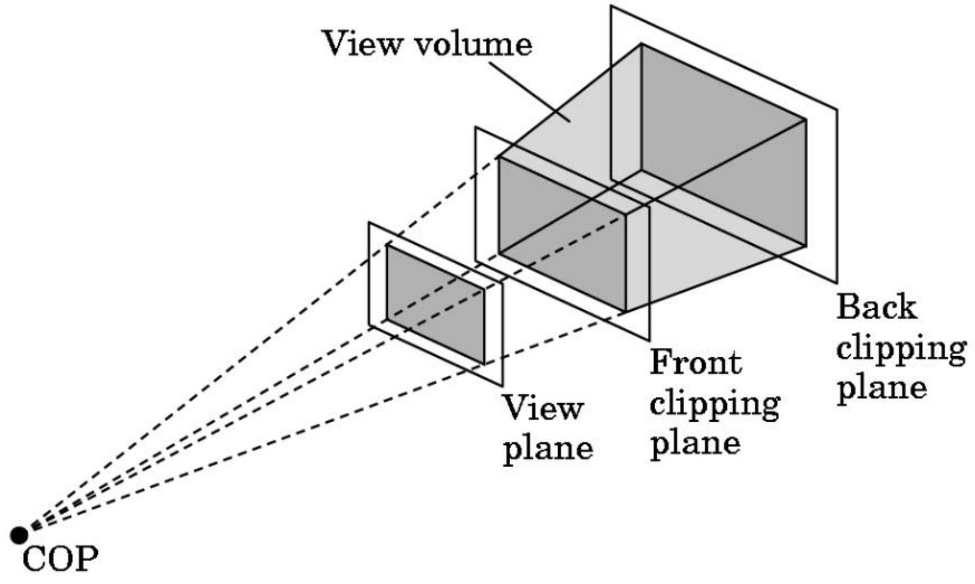
  are formed by the grouping of vertices in this step of the pipeline.



Clipping

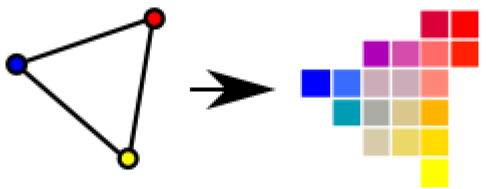
- Just as a real camera cannot "see" the whole world, the virtual camera can only see part of the world or object space
  - Objects that are not within this volume are said to be *clipped* out of the scene





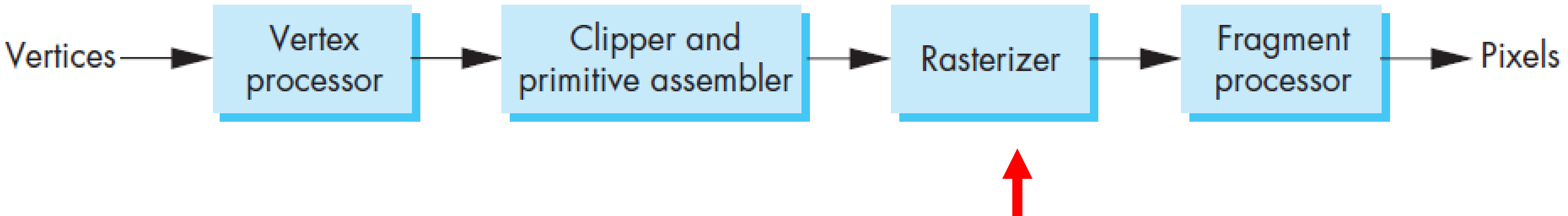
Rasterization

- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
- Fragments are "**potential pixels**". They
  - have a location in the frame buffer
  - have colour, depth, and alpha attributes



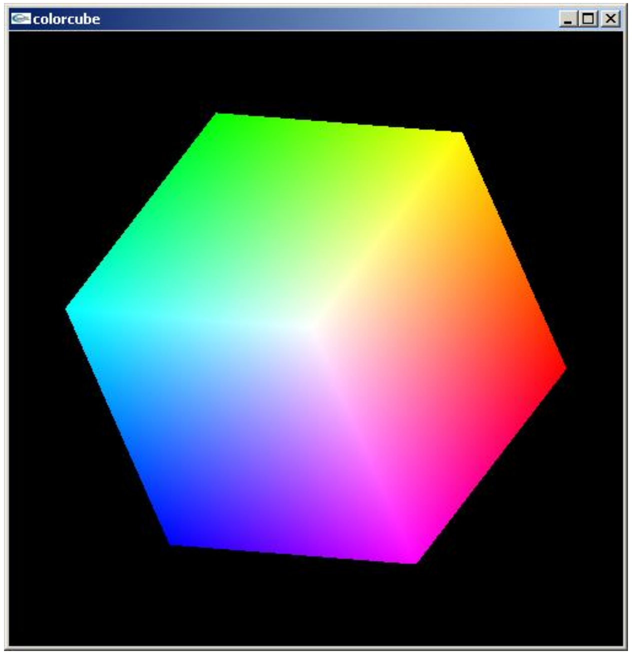
link

- Vertex attributes (colour, transparency) are interpolated over the objects by the rasterizer



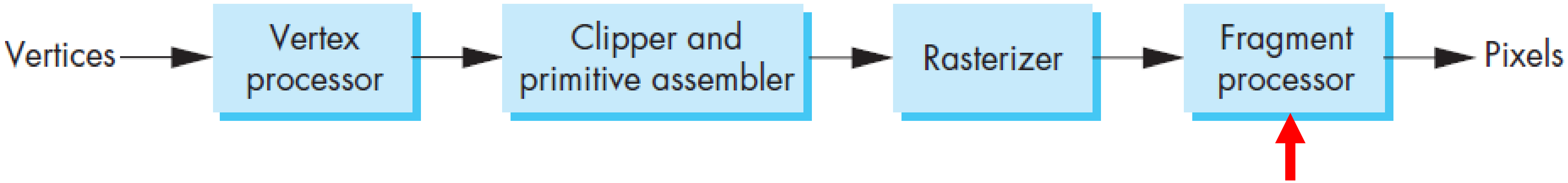
Smooth Color

- We can tell the *rasterizer* in the pipeline how to interpolate the vertex colours across the vertices
- Default is ***smooth shading***
    - OpenGL interpolates vertex colors across visible polygon
- Alternative is ***flat shading***
    - Color of the first vertex determines the fill color
- Shading is handled in the fragment shader



Fragment Processing

- Fragments are processed to determine the colour of the corresponding pixel in the frame buffer
- The colour of a fragment can be determined by texture mapping or by interpolation of vertex colours
- Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal



Graphics Modes

- Immediate Mode API
  - Immediate-mode APIs are normally procedural
  - Each time a new frame is drawn, the application issues the drawing commands.
  - The library does not store a scene model between the frames

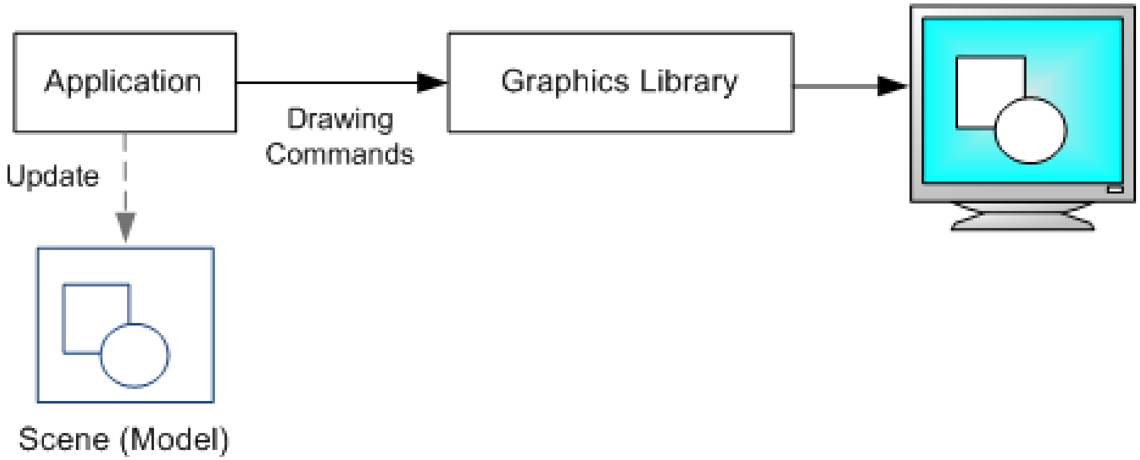


Image taken from docs.microsoft.com

Graphics Modes

- Retained Mode API
  - A retained-mode API is declarative
  - The application constructs a scene, and the library stores a model of the scene in the memory.
  - The application issues commands to update the scene (e.g., add or remove shapes)
  - The library redraws

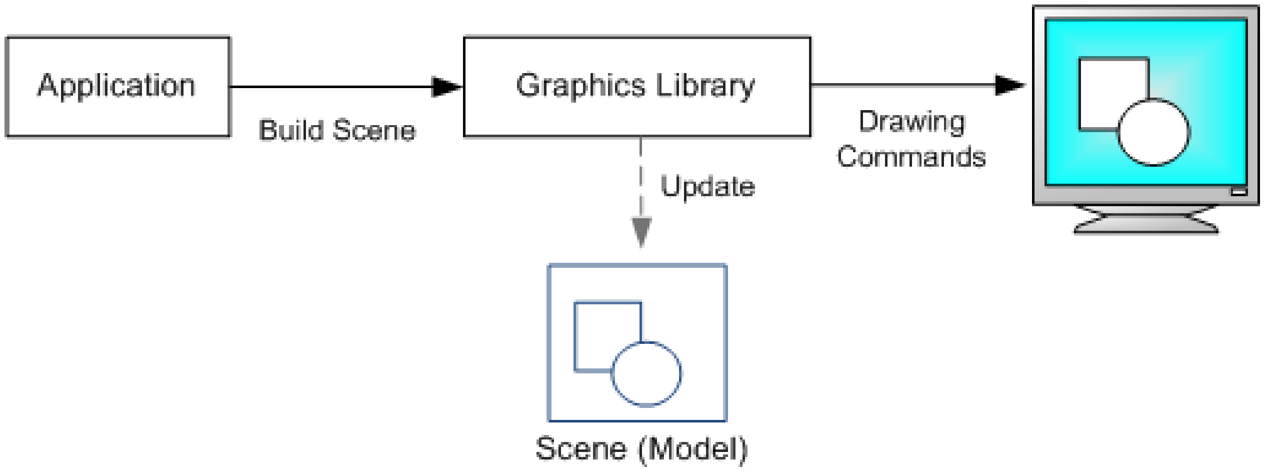


Image taken from docs.microsoft.com

Immediate Mode with OpenGL

- Older versions of OpenGL adopted **immediate mode graphics**, where
  - Each time a vertex is specified in application, its location is sent immediately to the GPU
  - Old style programming, uses **glVertex**

Immediate Mode with OpenGL

- Advantage:
  - No memory is required to store the geometric data (memory efficient)
- Disadvantages:
  - As the vertices are not stored, if they need to be displayed again, the entire vertex creation and the display process must be repeated.
  - Creates bottleneck between CPU and GPU
- Immediate mode graphics has been removed from OpenGL 3.1

Retained Mode Graphics with OpenGL

- Put all vertex and attribute data into an array, send and store that on the GPU
- Update when required
- Retained mode graphics is adopted in OpenGL 3.1 onward.

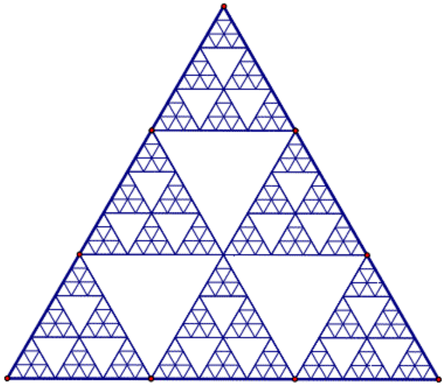
Comparison of the two modes

- Immediate mode graphics

```
main()
{
 initialize_the_system();
 p = find_initial_point();
 for (some_no_of_points) {
  q = generate_a_point(p);
  display(q);
  p = q;
 }
 cleanup();
}
```

- Retained mode graphics

```
main()
{
 initialize_the_system();
 p = find_initial_point();
 for (some_no_of_points) {
  q = generate_a_point(p);
  store_the_point(q);
  p = q;
 }
 display_all_points();
 cleanup();
}
```



2D Sierpinski triangle

Pseudo code for the 2D Sierpinski triangle program for the 2 modes

Further Reading

“Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL” by Edward Angel and Dave Shreiner, 6th Ed, 2012

- Sec. 1.7.2 – 1.7.7 Pipeline Architectures … Fragment Processing
- Sec. 2.1 The Sierpinski Gasket; immediate mode graphics vs retained mode graphics
- Sec 2.4 – 2.4.4 Primitives and Attributes … Triangulation