The OpenGL Rendering Pipeline

The Rendering Pipeline

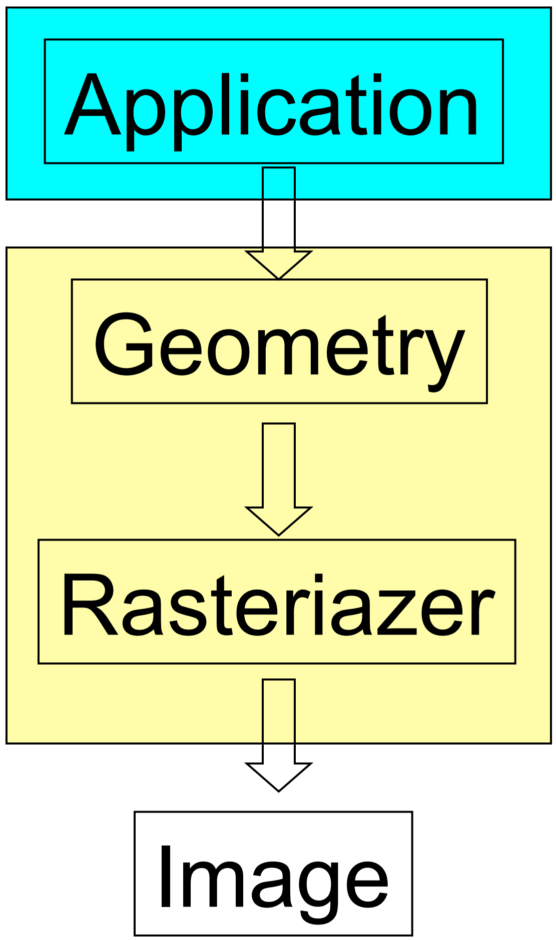
- The process to generate two-dimensional images from given virtual cameras and 3D objects
- The pipeline stages implement various core graphics rendering algorithms
- Why should you know the pipeline?
  - Understand various graphics algorithms
  - Program low level graphics systems
  - Necessary for programming GPUs
  - Help analyze the performance bottleneck

The Rendering Pipeline

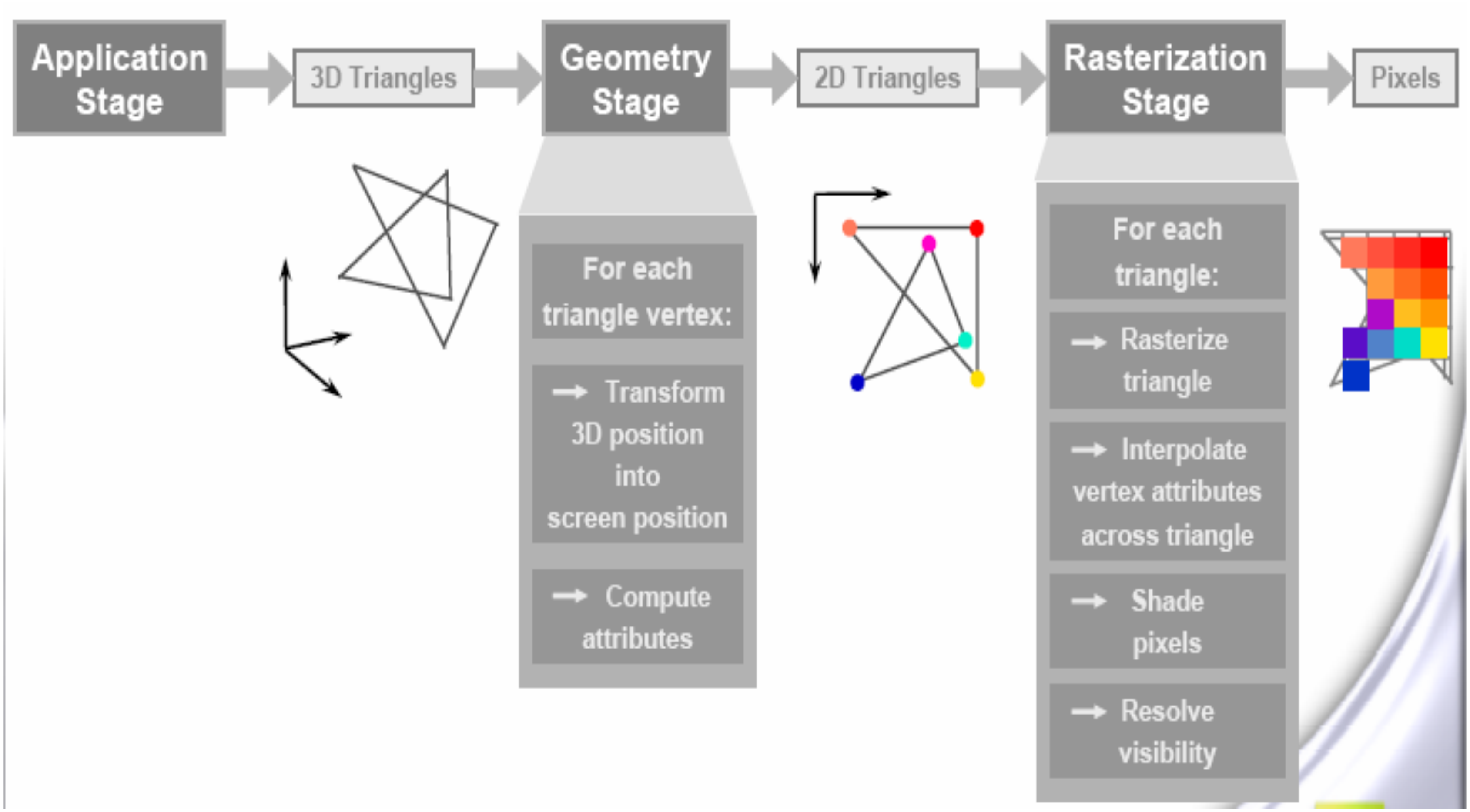
- The basic construction – three conceptual stages
- Each stage is a pipeline and runs in parallel
- Graphics performance is determined by the slowest stage
- Modern graphics systems:

  software:

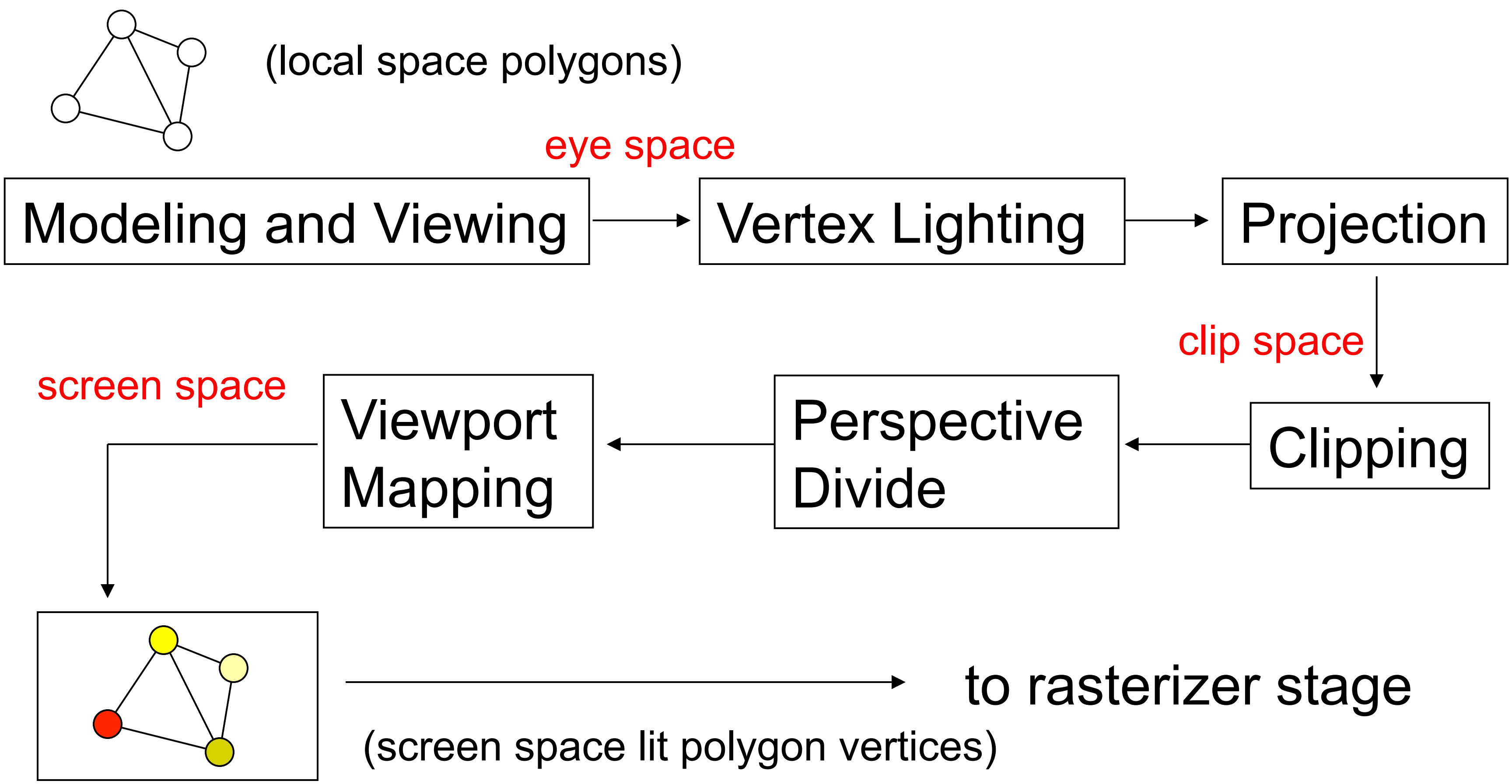
  hardware:



The Rendering Pipeline

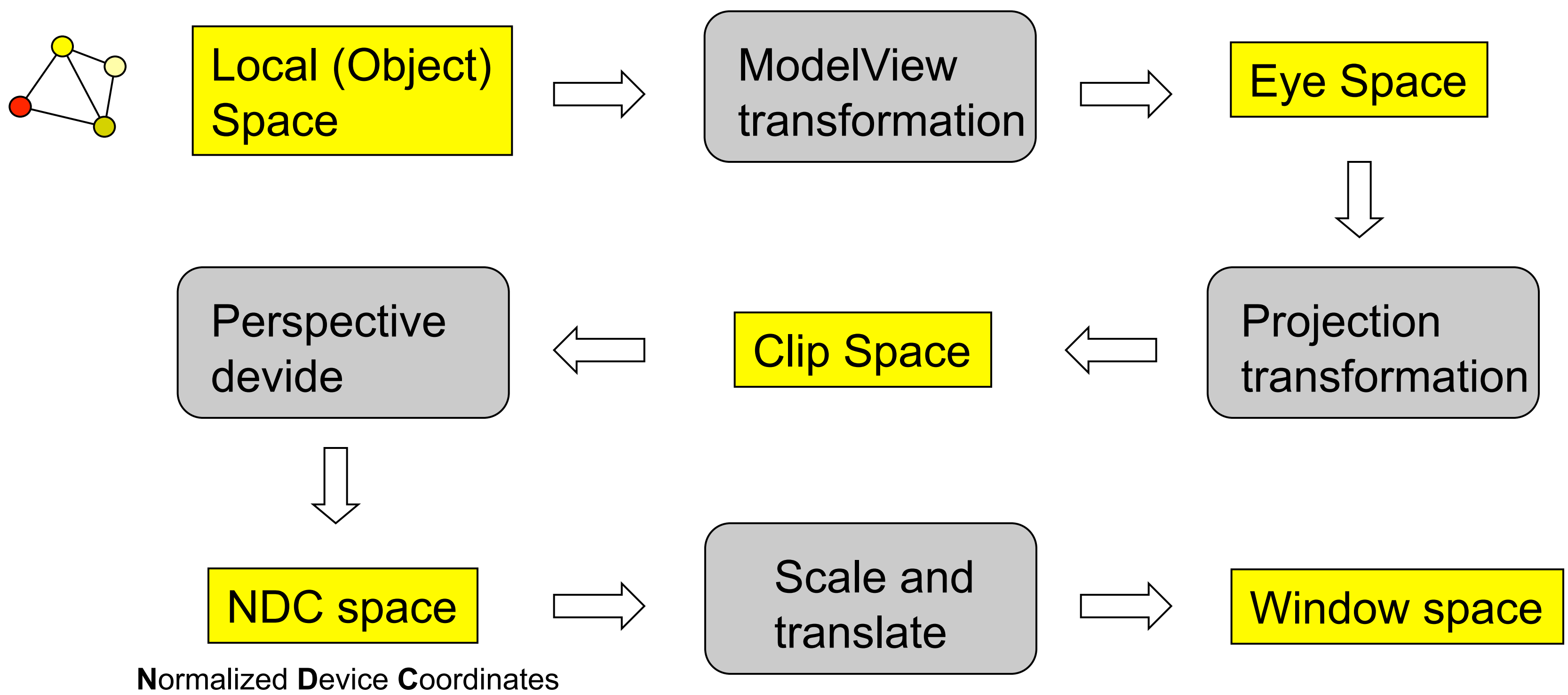


The Geometry Stage



Transformation Pipeline

- Another view of the graphics pipeline



Different Spaces

- Local space
  - A space where you define the vertex coordinates, normals, etc. This is before any transformations are taking place
  - These coordinates/normals are multiplied by the OpenGL modelview (VM) matrix into the eye space
  - Modelview matrix: Viewing transformation matrix (V) multiplied by modeling transformation matrix (M), i.e., GL_MODELVIEW = V * M
  - OpenGL matrix stack is used to allow different modelview matrices for different objects

Different Spaces (cont’d)

- Eye space
  - Where per vertex lighting calculation is occurred
  - Camera is at (0,0,0) and view’s up direction is by default (0,1,0)
  - Light position is stored in this space after being multiplied by the OpenGL modelview matrix
  - Vertex normals are consumed by the pipeline in this space by the lighting equation

Different Spaces (cont'd)

- Clip Space
  - After projection and before perspective divide
  - Clipping against view frustum done in this space
    - $-W <= X <= W$; $-W <= Y <= W$; $-W <= Z <= W$;
  - New vertices are generated as a result of clipping
  - The view frustum after transformation is a parallelepiped regardless of orthographic or perspective projection
- Perspective Divide
  - Transform clip space into NDC space
  - Divide (x,y,z,w) by w where $w = z/-d$ ($d=1$ in OpenGL so $w = -z$)
  - Result in foreshortening effect

Different Spaces (cont'd)

- Window Space
  - Map the NDC coordinates into the window
    - X and Y are integers, relative to the lower left corner of the window
    - Z are scaled and biased to [0,1]
    - Rasterization is performed in this space
  - The geometry processing ends in this space

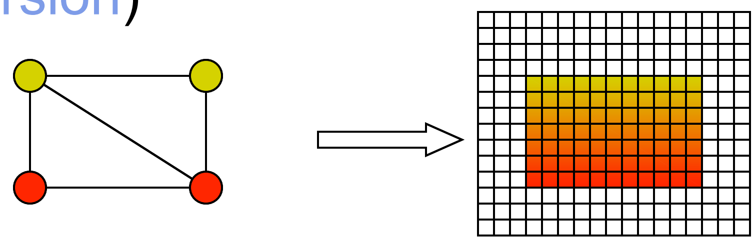
The Geometry Stage

- Transform coordinates and normal
  - Model->world
  - World->eye
- Normalize the normal vectors
- Compute vertex lighting
- Generate (if necessary) and transform texture coordinates
- Transform to clip space (by projection)
- Assemble vertices into primitives
- Clip against viewing frustum
- Divide by w (perspective divide if applies)
- Viewport transformation
- Back face culling

Introduce vertex dependences ☹

The Rasterizer Stage

- Per-pixel operation: assign colors to the pixels in the frame buffer (a.k.a scan conversion)

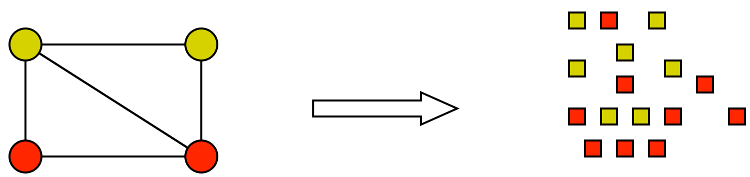


(frame buffer)

- Main steps:
  - Setup
  - Sampling (convert a primitive to fragments)
  - Texture lookup and Interpolation (lighting, texturing, z values, etc)
  - Color combinations (illumination and texture colors)
  - Fogging
  - Other pixel tests (scissor, alpha, stencil tests etc)
  - Visibility (depth test)
  - Blending/compositing/Logic op

The Rasterization Stage

- Convert each primitive into fragments (not pixels)
- Fragment: transient data structures
  - position (x,y); depth; color; texture coordinates; etc



- Fragments from the rasterized polygons are then selected (z buffer comparison for instance) to form the frame buffer pixels

The Rasterization Stage

- Two main operations
  - Fragment selection: generate one fragment for each pixel that is intersected by the primitive
  - Fragment assignment: sample the primitive properties (colors, depths, etc) for each fragment - nearest neighbor continuity, linear interpolation, etc

Polygon Scan Conversion

- The goal is to compute the scanline-primitive intersections
- OpenGL Spec does not specify any particular algorithm to use
- Brute Force: try to intersect each scanline with all edges as we go from ymin to ymax
- We can do better
  - Find ymin and ymax for each edge and only test the edge with scanlines in between
  - For each edge, only calculate the intersection with the ymin; calculate dx/dy; calculate the new intersection as y=y+1, x +dx/dy
  - Change x=x+dx/dy to integer arithmetic (such as using Bresenham's algorithm)

Rasterization steps

- Texture interpolation
- Color interpolation
- Fog (blend the fog color with the fragment color based on the depth value)
- Scissor test (test against a rectangular region)
- Alpha test (compare with alpha, keep or drop it)
- Stencil test(mask the fragment depending on the content of the stencil buffer)
- Depth test (z buffer algorithm)
- Alpha blending
- Dithering (make the color look better for low res display mode)