Graphics Pipeline in 2D

CS 417 Lecture 10

Object vs. image order

- Object order
  - process objects in isolation, one at a time
  - update image after each
  - most often used when efficiency is important
- Image order
  - process pixels in isolation, one at a time
  - potentially access entire scene for each one
  - most often used when realism is important
- Talking about object order today

Object-order graphics

- For fast and/or hardware accelerated graphics, the field has converged on a fairly standard sequence of operations
  - this sequence of operations is the *graphics pipeline*
- Today I will discuss things in 2D
  - leaves off some stages at the beginning that are concerned with getting 3D objects onto the 2D screen

Graphics hardware

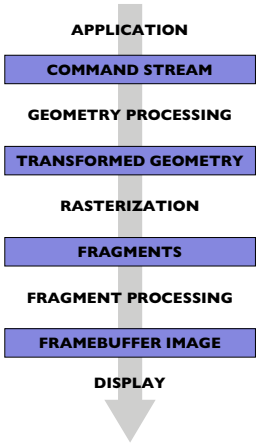
- Present in essentially all PCs sold today
- GPU on same order of magnitude size as CPU
  - considerably more computing power due to parallelism
- Graphics system has large chunk of local RAM
- Today we are concerned with general architecture
  - will cover harware in more detail in a later lecture

Immediate vs. retained mode

- Immediate mode: primitives drawn as they are specified
  - advantages: simple, flexible
- Retained mode: graphics system stores scene graph
  - advantages: store data local to graphics hardware
- Most used today: immediate mode
  - but provisions for storing data in the GPU's RAM as needed

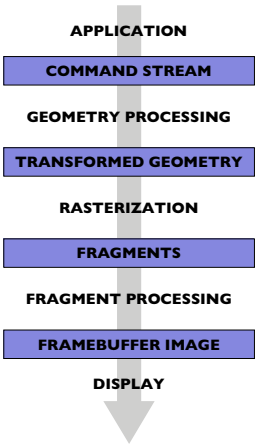
Graphics pipeline

- The sequence of operations needed to get primitives onto the screen
- Fairly uniform across different graphics hardware today
- OpenGL is our interface to the pipeline



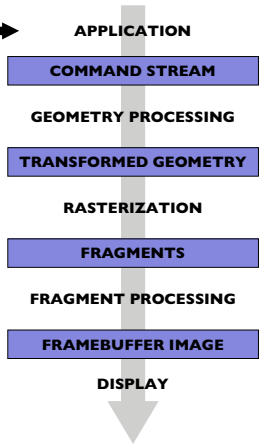
Graphics pipeline

- "Pipeline"
  - same sense as in CPUs
  - all stages work in parallel on a long stream of work
  - analogous to assembly line

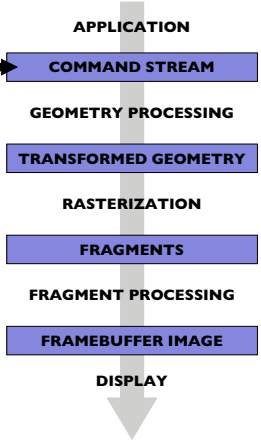


You are here

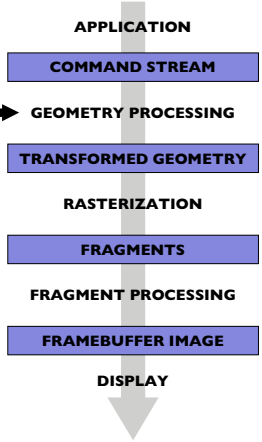
- Application issues commands
  - OpenGL calls
  - "set color to red"
  - "transform by T"
  - "draw this triangle"
  - …



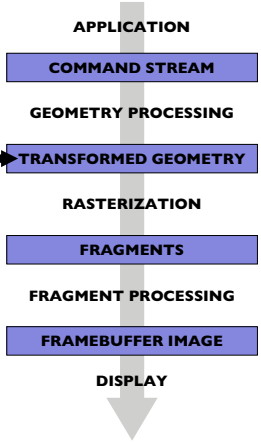
- Command stream defines what is to be done
  - geometry specified by vertices
  - can be forwarded to many types of systems, from local software renderers to large parallel machines



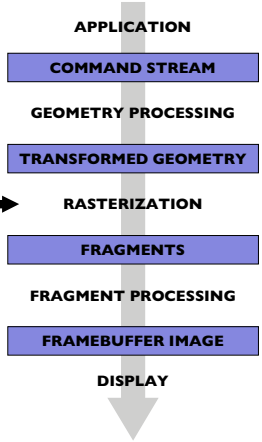
- Geometry processing works on geometric primitives
  - mainly geometric transformations
  - clipping to viewport



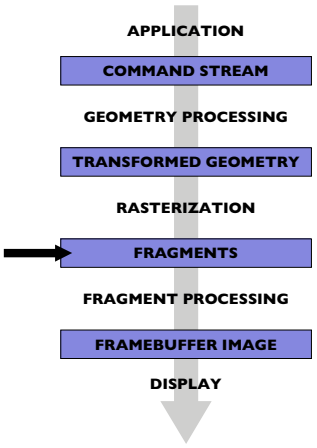
- Transformed geometry
  - represented in pixel coordinates
  - ready to go into framebuffer



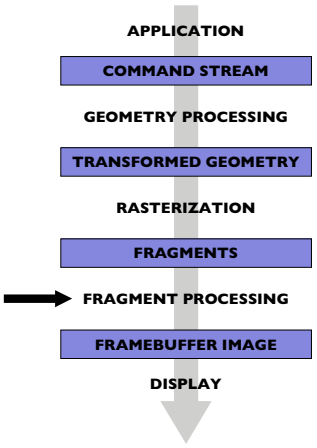
- Rasterization
  - discussed earlier
  - transform geometry into *fragments*, or colors at pixels
  - generally many fragments per primitive



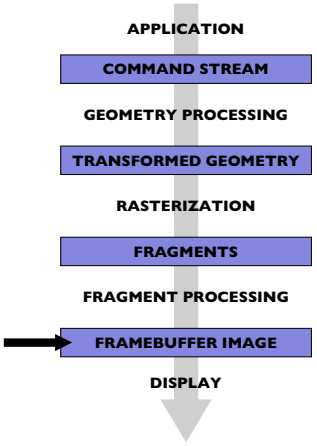
- Fragment stream
  - ready to write straight into framebuffer
  - order matters



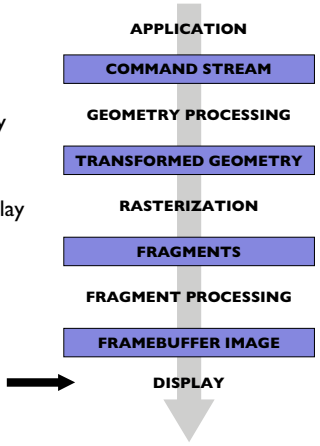
- Fragment processing
  - various things go on to change fragment colors
  - fragments get composited into framebuffer



- Framebuffer
  - a chunk of RAM that holds the image that belongs on the screen
  - double buffering: front buffer is displayed while back buffer is updated



- Display output
  - framebuffer is continuously read out
  - used to generate video or otherwise refresh the display



Announcements

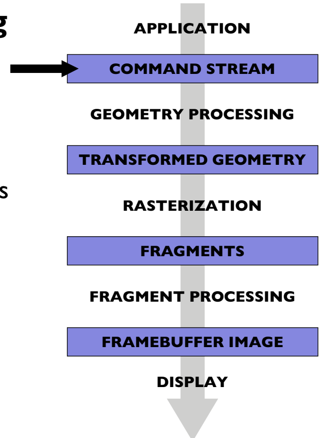
- Small clarification to hw2 to be posted soon
- Project 1 is due right now
  - however, because the handin system was late we'll accept handins until **6pm** tonight
  - hand in using the web site or if that does not work mail your .jar to spf@graphics.cornell.edu
- Grading for Project 1
  - we'll turn on the web system for signing up by tomorrow
  - most likely will be Thursday and Friday this week

Pipeline specifics (OpenGL)

- A few more details about this particular case
- Representative of most modern systems

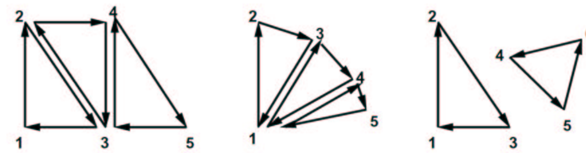
Geometry encoding

- Vertices
  - associated data
  - e.g., color (always RGBA)
- Assembled into primitives
  - points
  - lines
  - triangles
  - polygons (convex)
- Raster images
  - bypass geometry stages



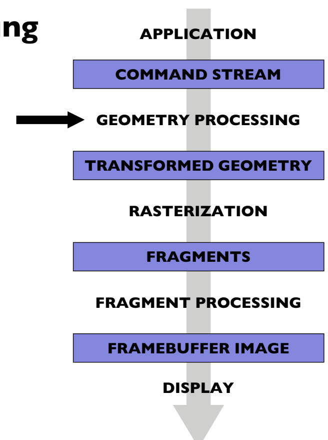
Efficient representation of triangles

- Bandwidth is scarce in the command stream
- Triangle strips: replace oldest vertex
- Triangle fans: replace second vertex
- In both cases 3n vertices is reduced to (n + 2)



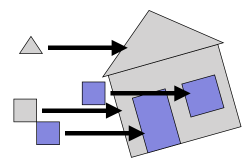
Geometry processing

- Transformations
- Clipping



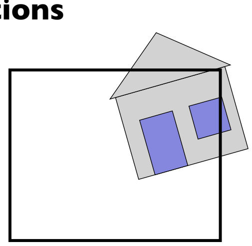
Geometric transformations

- Modeling
  - place objects in scene
- Viewing
  - transform scene to canonical coordinates for clipping
- Viewport
  - transform final geometry to pixel coordinates



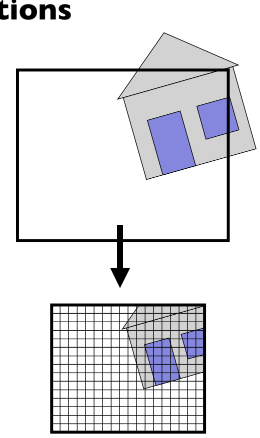
Geometric transformations

- Modeling
  - place objects in scene
- Viewing
  - transform scene to canonical coordinates for clipping
- Viewport
  - transform final geometry to pixel coordinates



Geometric transformations

- Modeling
  - place objects in scene
- Viewing
  - transform scene to canonical coordinates for clipping
- Viewport
  - transform final geometry to pixel coordinates



Clipping geometry

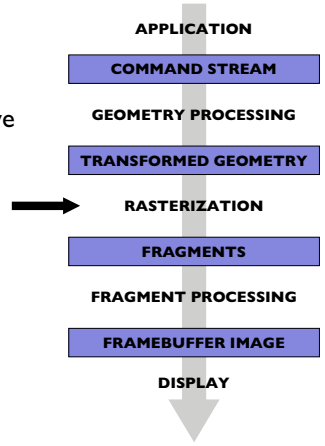
- Want to avoid rasterizing primitives that extend outside viewport
- Intersect primitives with viewport
  - points: discard (cull) if outside viewport
  - lines: intersection is another line segment
  - triangles: intersection can be a polygon (but convex)
  - polygons: intersection could be messy
    - so don't do this; use triangles instead

Specifying transformations

- Current transformation is a state variable
- Reset it by sending `LoadIdentity`
- Compose transformations onto it with `MultMatrix`
  - various convenience routines also
- Save and restore it using `PushMatrix`, `PopMatrix`
  - used to implement hierarchy traversal

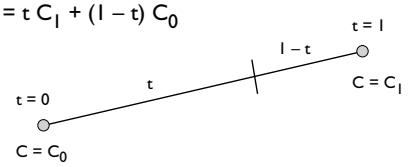
Rasterization

- Efficiently list pixels intersecting each primitive
- Raster data comes in at this step
  - e.g. `WritePixels`



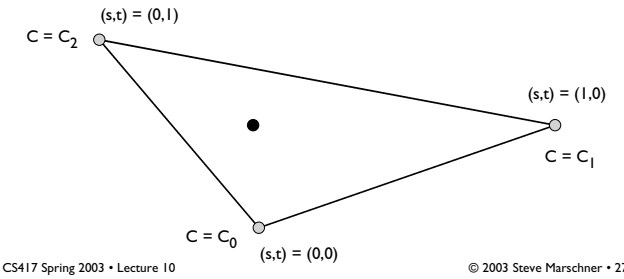
Rasterization: interpolation

- Data (e.g. color) associated with vertices needs to be associated with fragments
  - if all vertices have the same color, it's easy
  - if colors are different, we want a smoothly varying result
- Linear interpolation
  - along a line: want linear ramp as a function of length
  - $C(t) = t\,C_1 + (1 - t)\,C_0$



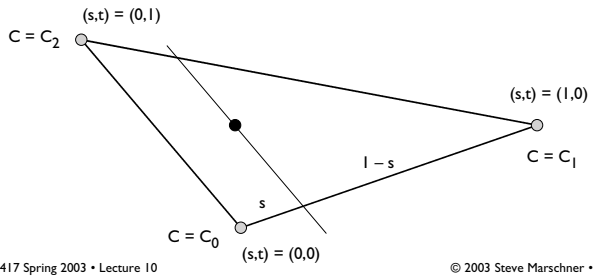
Rasterization: interpolation

- Linear interpolation
  - across a triangle: unique affine function matching at vertices
  - $C(s, t) = s\,C_1 + t\,C_2 + (1 - s - t)\,C_0$



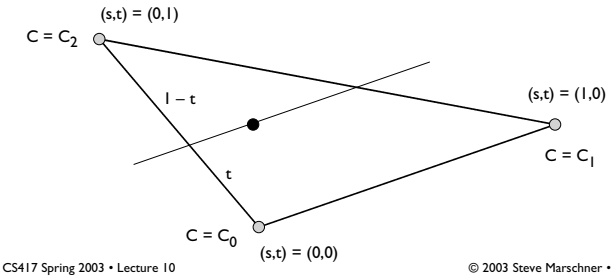
Rasterization: interpolation

- Linear interpolation
  - across a triangle: unique affine function matching at vertices
  - $C(s, t) = s\,C_1 + t\,C_2 + (1 - s - t)\,C_0$



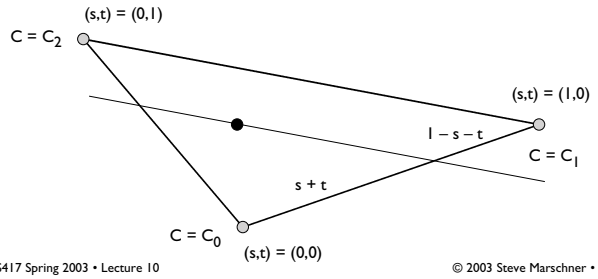
Rasterization: interpolation

- Linear interpolation
  - across a triangle: unique affine function matching at vertices
  - $C(s, t) = s\,C_1 + t\,C_2 + (1 - s - t)\,C_0$



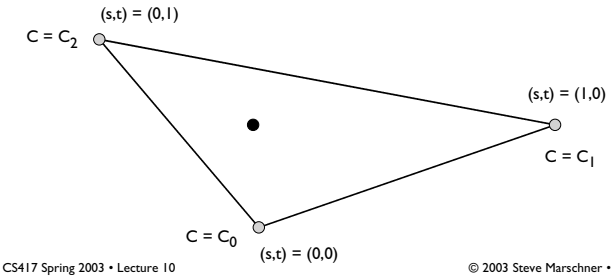
Rasterization: interpolation

- Linear interpolation
  - across a triangle: unique affine function matching at vertices
  - $C(s, t) = s\,C_1 + t\,C_2 + (1 - s - t)\,C_0$



Rasterization: interpolation

- Linear interpolation
  - across a triangle: unique affine function matching at vertices
  - $C(s, t) = s\,C_1 + t\,C_2 + (1 - s - t)\,C_0$

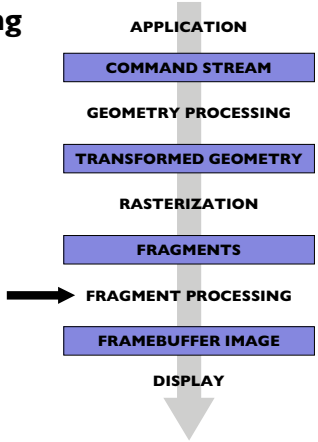


Rasterization: interpolation

- Interpolation is a fundamental function of rasterizer
  - many more uses will emerge in 3D
  - even rasterization itself can be framed as interpolation
    - interpolate s, t, and $r = 1 - s - t$
    - generate fragments when all > 1

Fragment processing

- Culling tests
  - e.g. alpha test (paintbrush example)
  - e.g. scissors test
- Blending
  - compositing operations in framebuffer (PaintFmwk example)



Framebuffer operations

- A few commands skip directly to the framebuffer
  - `Clear` is the most common example
- Reverse path also exists
  - `ReadPixels` to get the result of what you drew
  - But often very slow…