CMSC427
Rendering pipeline

Credit: some slides from Dr. Zwicker

The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates

modeling

- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

vertex in mesh

$$\mathrm{p}' = \overset{3}{\mathbf{D}}\overset{2}{\mathbf{P}}\mathbf{C}^{-1}\overset{1}{\mathbf{M}}\mathbf{p}$$

Object space

World space

Camera space

Canonic view volume

Image space

viewport

The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

$$\mathbf{p}' = \mathbf{DPC}^{-1}\mathbf{Mp}$$

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

Pixel coordinates $\begin{matrix} x'/w' \\ y'/w' \end{matrix}$

OpenGL details

- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{P}$, viewport matrix $\mathbf{D}$

$$\mathbf{p}' = \mathbf{D}\underbrace{\mathbf{P}}_{\text{Projection matrix}}\overbrace{\mathbf{C}^{-1}\mathbf{M}}^{\text{Model-view matrix}}\mathbf{p}$$

- OpenGL rendering pipeline performs these matrix multiplications in vertex shader program
- User just specifies the model-view and projection matrices
- See Java code jrtr.GLRenderContext.draw and default vertex shader in file default.vert

OpenGL details

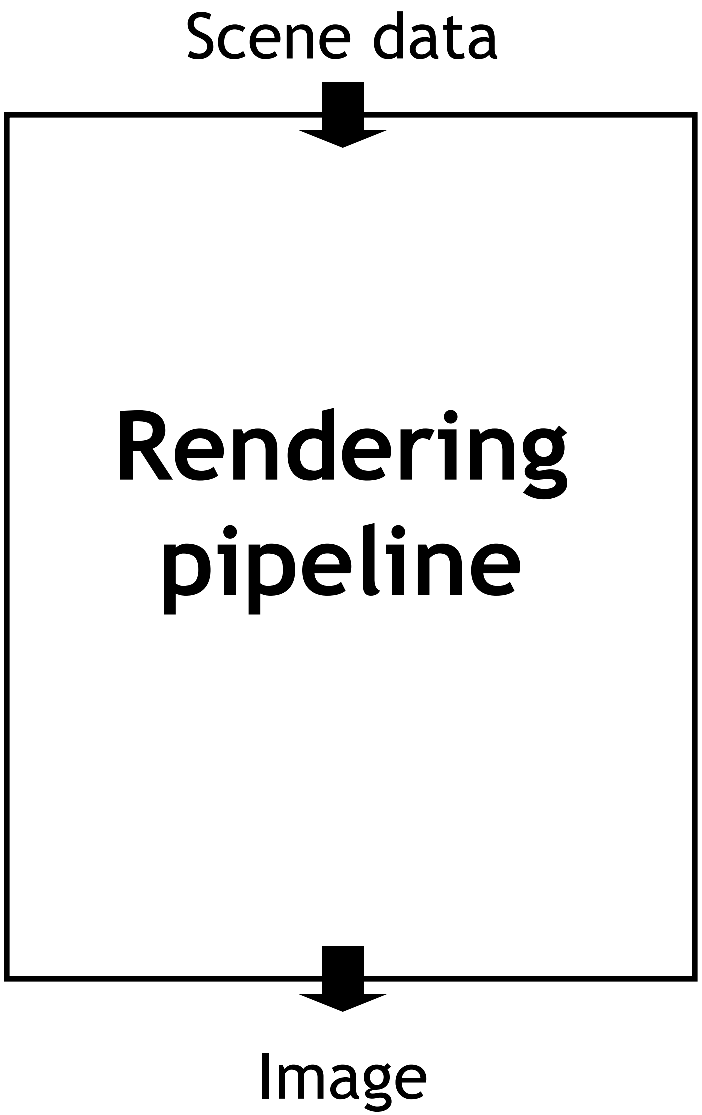
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{P}$, viewport matrix $\mathbf{D}$

$$\mathbf{p}' = \mathbf{D}\underbrace{\mathbf{P}}_{\text{Projection matrix}}\overbrace{\mathbf{C}^{-1}\mathbf{M}}^{\text{Model-view matrix}}\mathbf{p}$$

- Exception: viewport matrix, **D**
  - Specified implicitly via glViewport()
  - No direct access, not used in shader program

Rendering pipeline

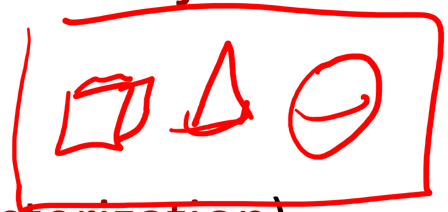
http://en.wikipedia.org/wiki/Graphics_pipeline

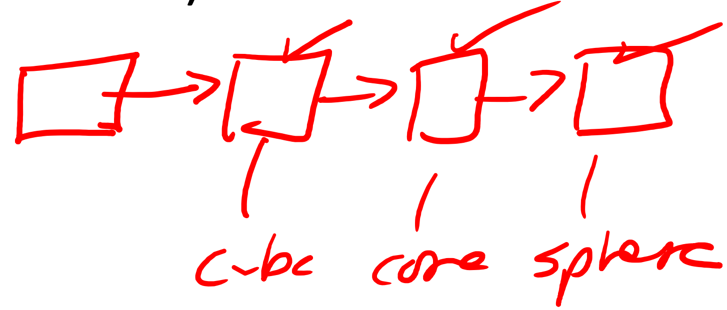


- Hardware & software that draws 3D scenes on the screen
- Most operations performed by specialized hardware (graphics processing unit, GPU, http://en.wikipedia.org/wiki/Graphics_processing_unit)
- Access to hardware through low-level 3D API (DirectX, OpenGL)
  - jogl is a Java binding to OpenGL, used in our projects http://jogamp.org/jogl/www/
- All scene data flows through the pipeline at least once for each frame (i.e., image)

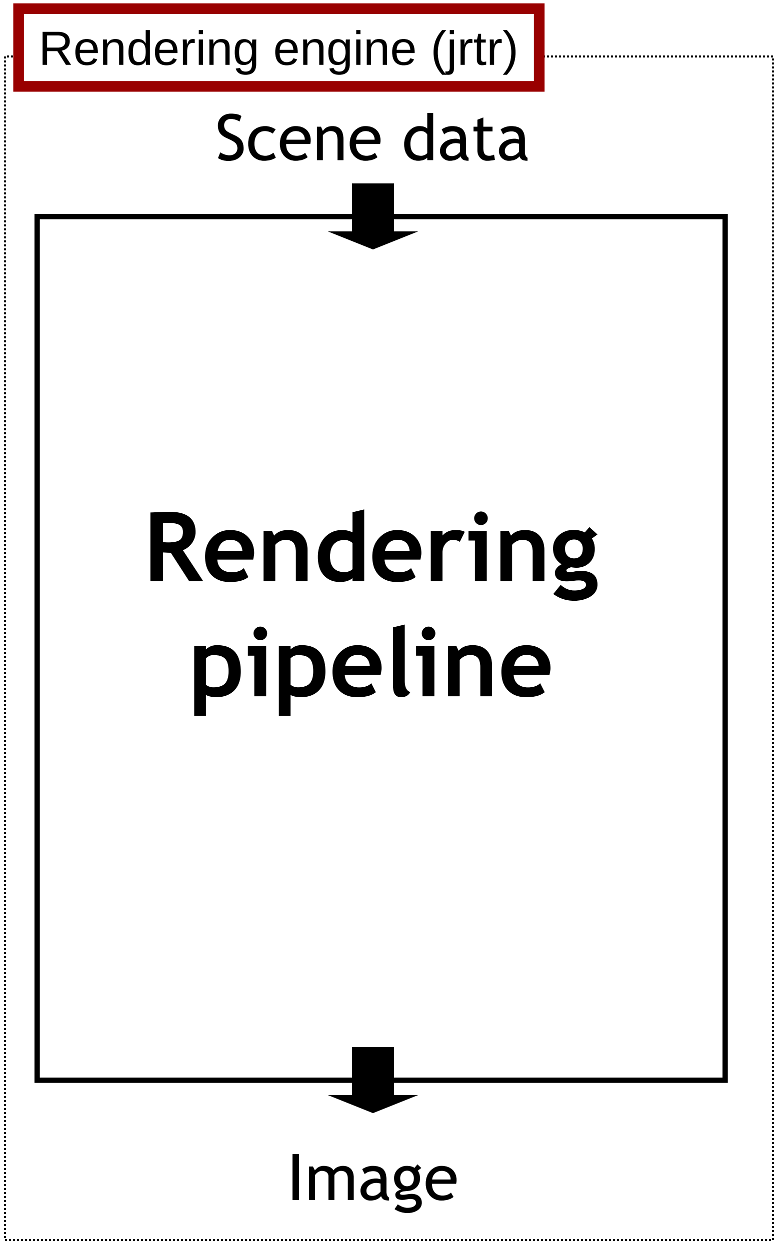
Rendering pipeline

- Rendering pipeline implements object order algorithm
  - Loop over all objects
  - Draw triangles one by one (rasterization)
- Alternatives?
- Advantages, disadvantages?





Rendering engine



- Additional software layer ("middle-ware") encapsulating low-level API (OpenGL, DirectX, ...)
- Additional functionality (file I/O, scene management, ...)
- Layered software architecture common in industry
  - Game engines http://en.wikipedia.org/wiki/Game_engine

Rendering pipeline stages (simplified)

Scene data

↓

Vertex processing, modeling and viewing transformation

↓

Projection

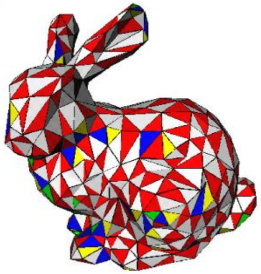
↓

Rasterization, fragment processing, visibility

↓

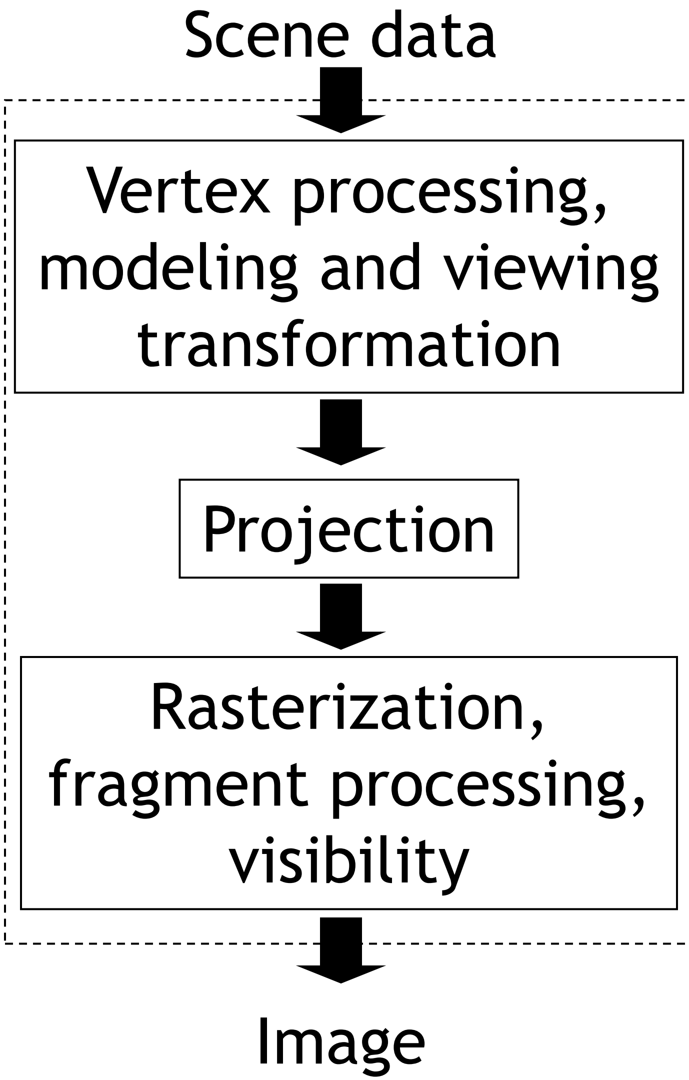
Image

- Geometry – mesh
  - Vertices and how they are connected
  - Triangles, lines, point sprites, triangle strips
  - Attributes such as color



- Specified in object coordinates
- Processed by the rendering pipeline one-by-one

Rendering pipeline stages (simplified)



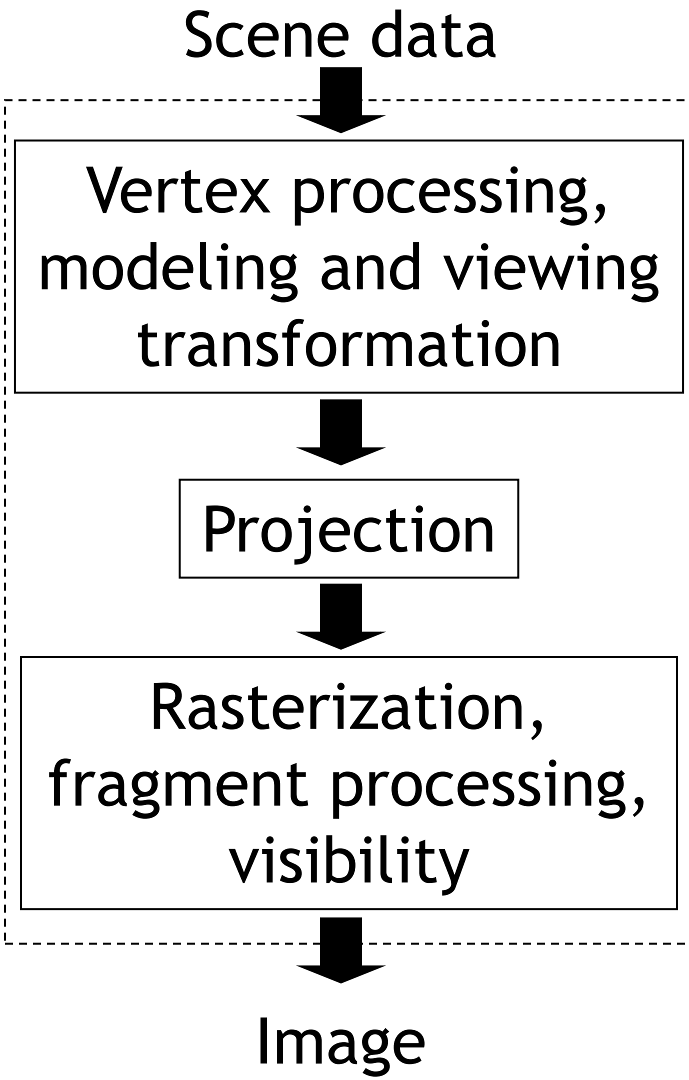
on GPU

- Transform object to camera coordinates

$$\mathbf{p}_{camera} = \underbrace{\mathbf{C}^{-1}\mathbf{M}}_{\text{MODELVIEW matrix}}\mathbf{p}_{object}$$

- Additional processing on per-vertex basis
  - Shading, i.e., computing per-vertex colors
  - Deformation, animation
  - Etc.

Rendering pipeline stages (simplified)



- Project 3D vertices to 2D image positions
- ~~This lecture~~

Rendering pipeline stages (simplified)

Scene data

↓

Vertex processing, modeling and viewing transformation

↓

Projection

↓

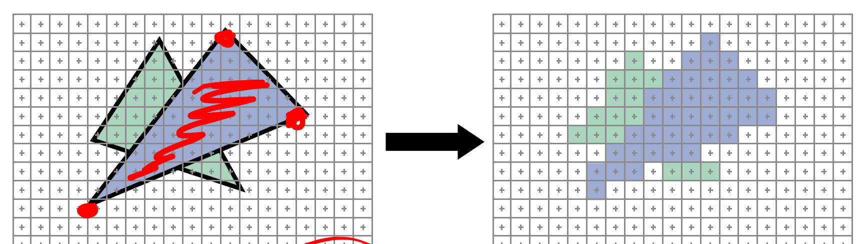
Rasterization, fragment processing, visibility

↓

Image

- Draw primitives pixel by pixel on 2D image (triangles, lines, point sprites, etc.)
- Compute per fragment (i.e., pixel) color
- Determine what is visible
- Next lecture

triangle, line



Rasterization

Rendering pipeline stages (simplified)

Scene data ↓ geometry shader

Vertex processing, modeling and viewing transformation ← vertex shader

↓

Projection

↓

Rasterization, fragment processing, visibility ← fragment shader

↓

Image ➡

- Grid (2D array) of RGB pixel colors