CSE 167:
Introduction to Computer Graphics
Lecture #5: Rasterization

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2015

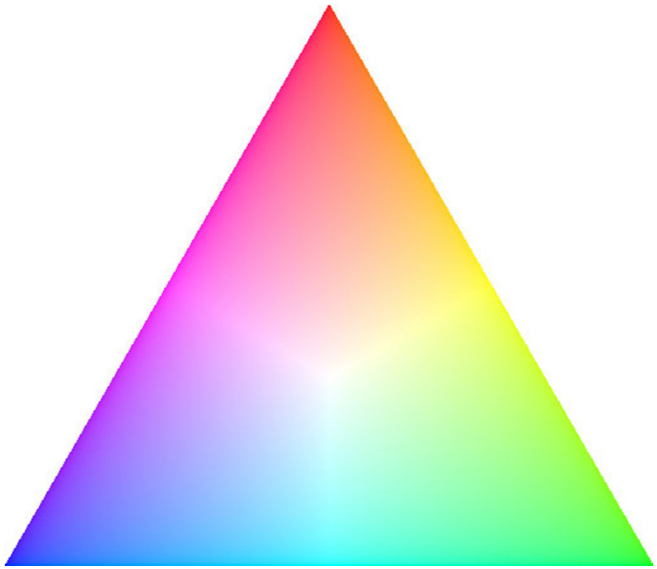
Announcements

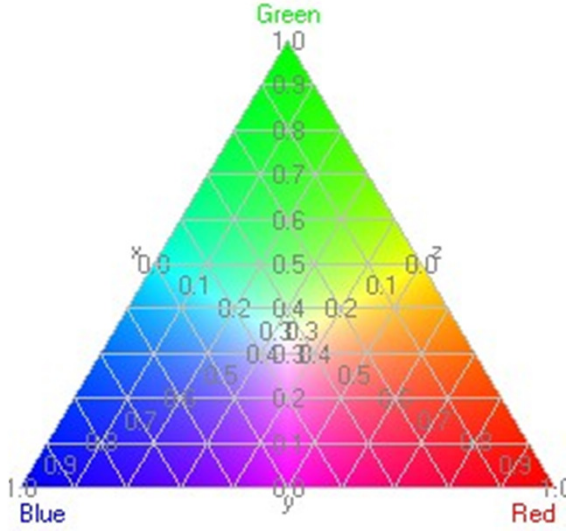
- Project 2 due tomorrow at 2pm
  - Grading window is 2-3:30pm
  - Upload source code to Ted by 2pm
- Project 3 discussion next Monday at 3pm

Lecture Overview

- Barycentric Coordinates
- Rendering Pipeline
- Rasterization
- Visibility

Color Interpolation





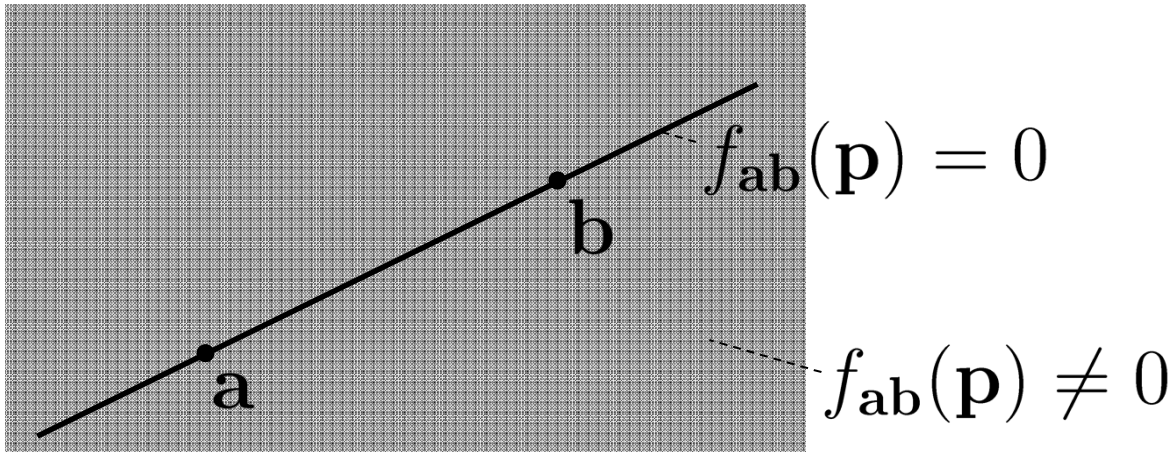
Source: efg's computer lab

- What if a triangle's vertex colors are different?
- Need to interpolate across triangle
  - How to calculate interpolation weights?

UCSD

Implicit 2D Lines

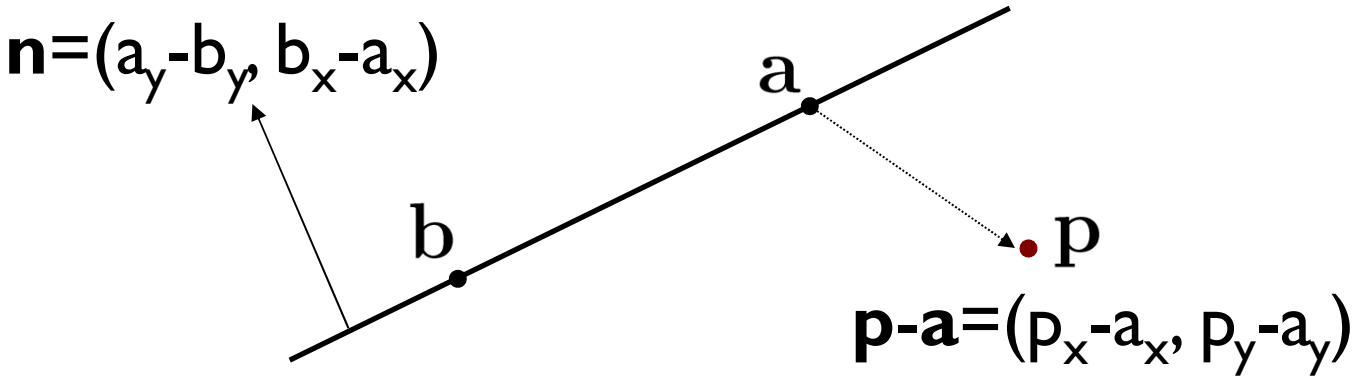
- Given two 2D points **a**, **b**
- Define function $f_{\mathbf{ab}}(\mathbf{p})$ such that $f_{\mathbf{ab}}(\mathbf{p}) = 0$ if **p** lies on the line defined by **a**, **b**



UCSD

Implicit 2D Lines

- Point **p** lies on the line, if **p-a** is perpendicular to the normal **n** of the line

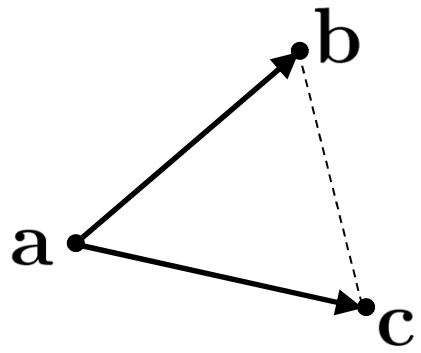


- Use dot product to determine on which side of the line **p** lies. If f(**p**)>0, **p** is on same side as normal, if f(**p**)<0 **p** is on opposite side. If dot product is 0, **p** lies on the line.

$$f_{\mathbf{ab}}(\mathbf{p}) = (a_y - b_y, b_x - a_x) \cdot (p_x - a_x, p_y - a_y)$$

Barycentric Coordinates

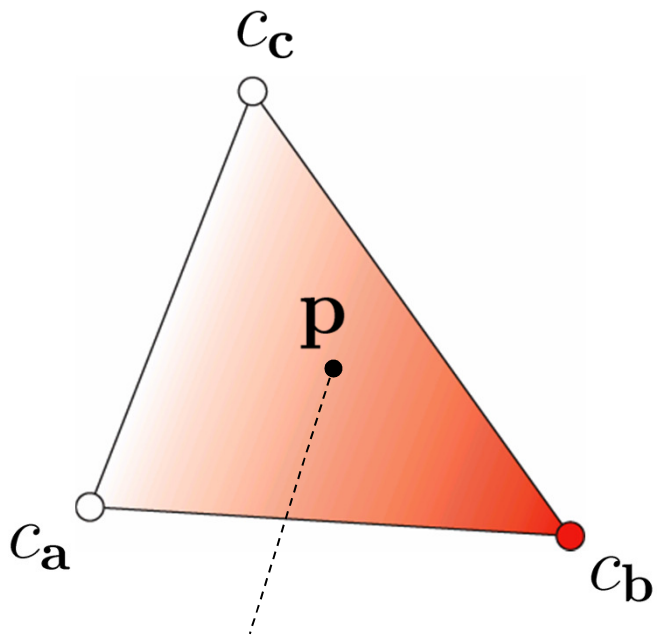
- Coordinates for 2D plane defined by triangle vertices **a, b, c**
- Any point **p** in the plane defined by **a, b, c** is
  $\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$
- Solved for a, b, c:
  $\mathbf{p} = (1 - \beta - \gamma)\,\mathbf{a} + \beta\,\mathbf{b} + \gamma\,\mathbf{c}$
- We define $\alpha = 1 - \beta - \gamma$
  ➔ $\mathbf{p} = \alpha\,\mathbf{a} + \beta\,\mathbf{b} + \gamma\,\mathbf{c}$
- α, β, γ are called **barycentric** coordinates
- If we imagine masses equal to α, β, γ in the locations of the vertices of the triangle, the center of mass (the Barycenter) is then **p**. This is the origin of the term "barycentric" (introduced 1827 by Möbius)



UCSD

Barycentric Interpolation

- Interpolate values across triangles, e.g., colors



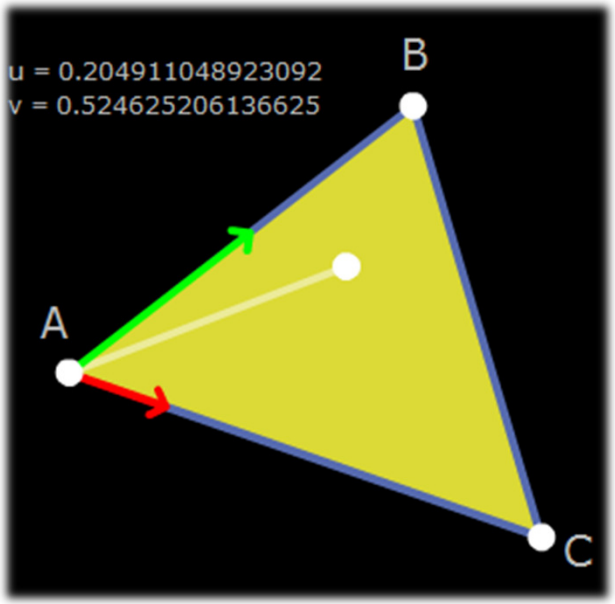
- Done by linear interpolation on triangle:

$$c(\mathbf{p}) = \alpha(\mathbf{p})c_\mathbf{a} + \beta(\mathbf{p})c_\mathbf{b} + \gamma(\mathbf{p})c_\mathbf{c}$$

- Works well at common edges of neighboring triangles

Barycentric Coordinates

- Demo:
  - http://adrianboeing.blogspot.com/2010/01/barycentric-coordinates.html

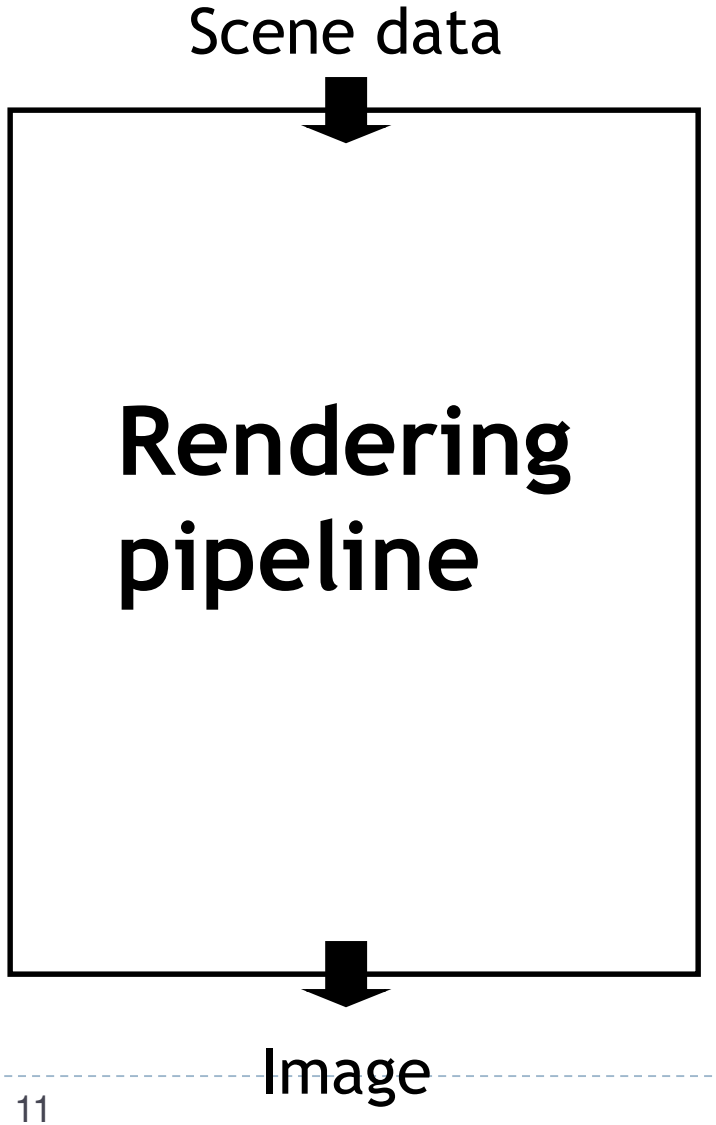


UCSD

Lecture Overview

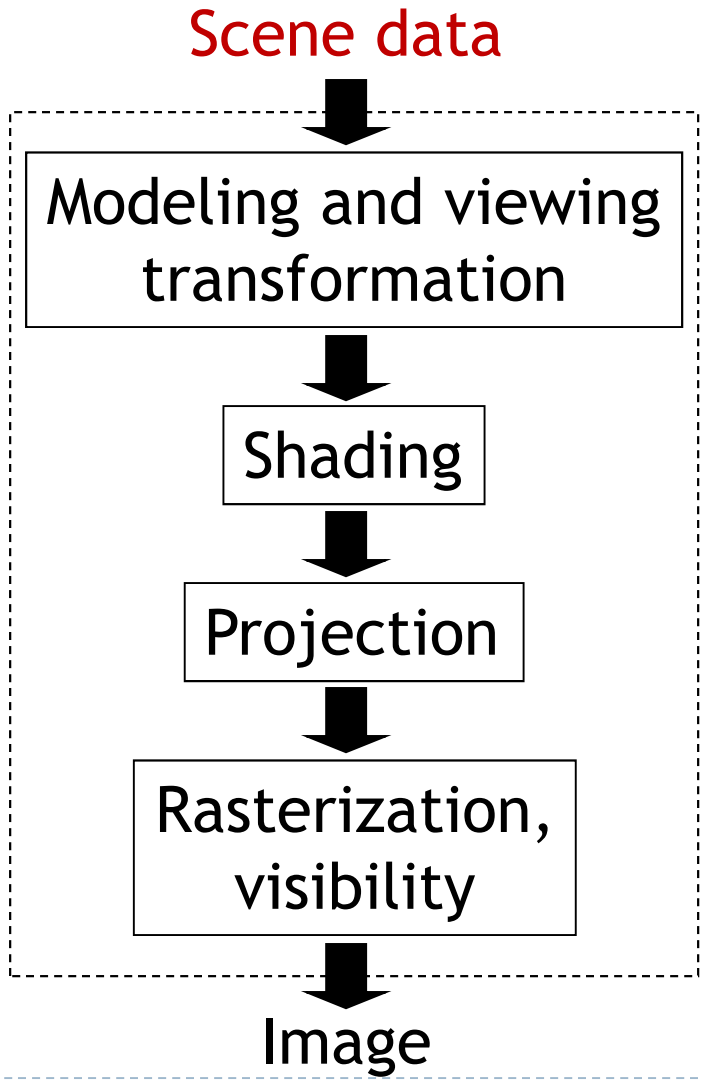
- Barycentric Coordinates
- Rendering Pipeline
- Rasterization
- Visibility

Rendering Pipeline

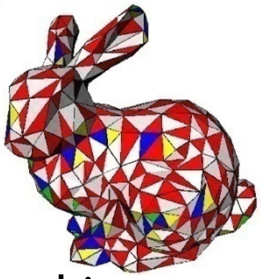


- Hardware and software which draws 3D scenes on the screen
- Consists of several stages
  - Simplified version here
- Most operations performed by specialized hardware (GPU)
- Access to hardware through low-level 3D API (OpenGL, DirectX)
- All scene data flows through the pipeline at least once for each frame

Rendering Pipeline

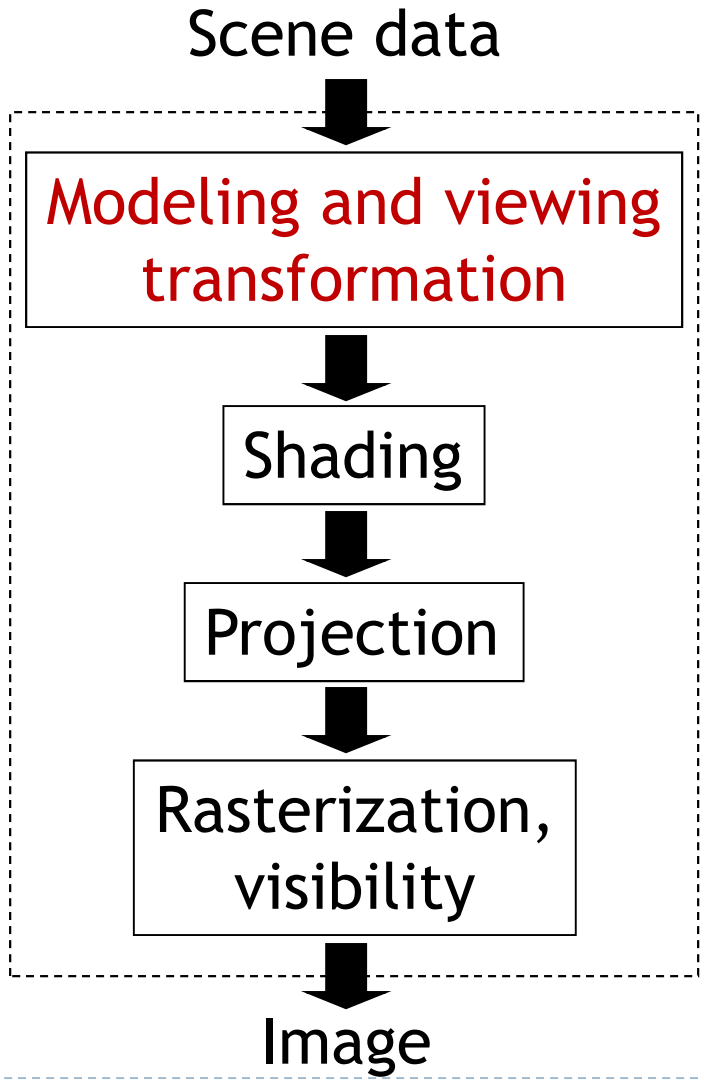


- Textures, lights, etc.
- Geometry
  - Vertices and how they are connected
  - Triangles, lines, points, triangle strips
  - Attributes such as color



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

Rendering Pipeline



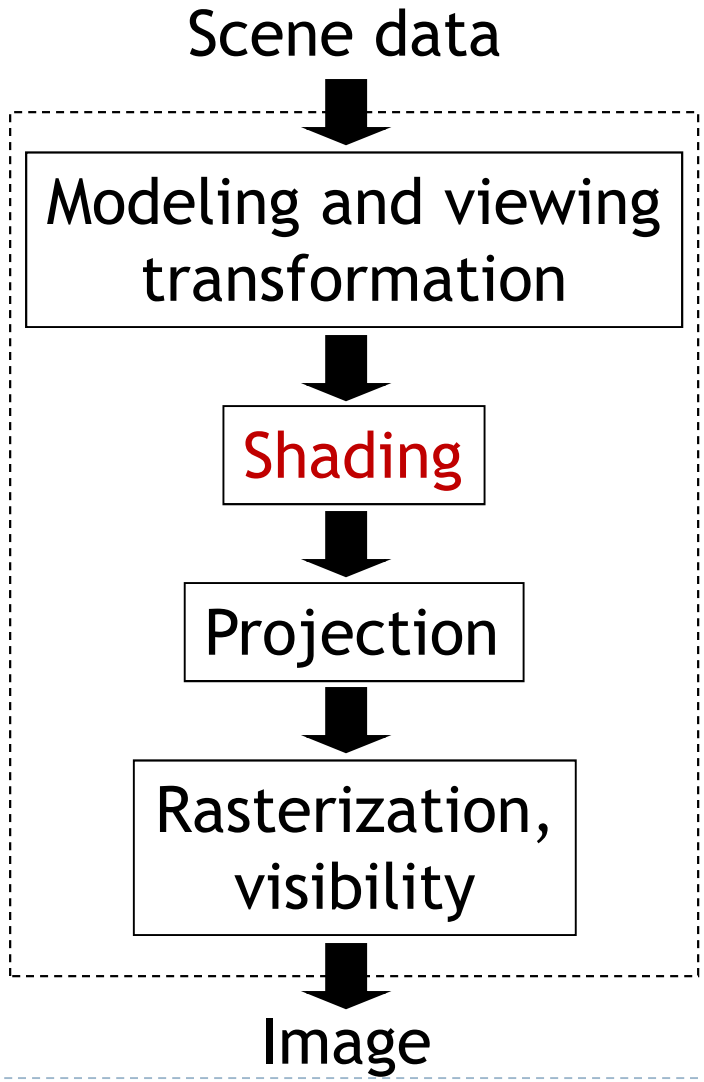
- Transform object to camera coordinates
- Specified by GL_MODELVIEW matrix in OpenGL
- User computes GL_MODELVIEW matrix as discussed

$$\mathbf{p}_{camera} = \mathbf{C}^{-1}\mathbf{M}\mathbf{p}_{object}$$

MODELVIEW matrix

UCSD

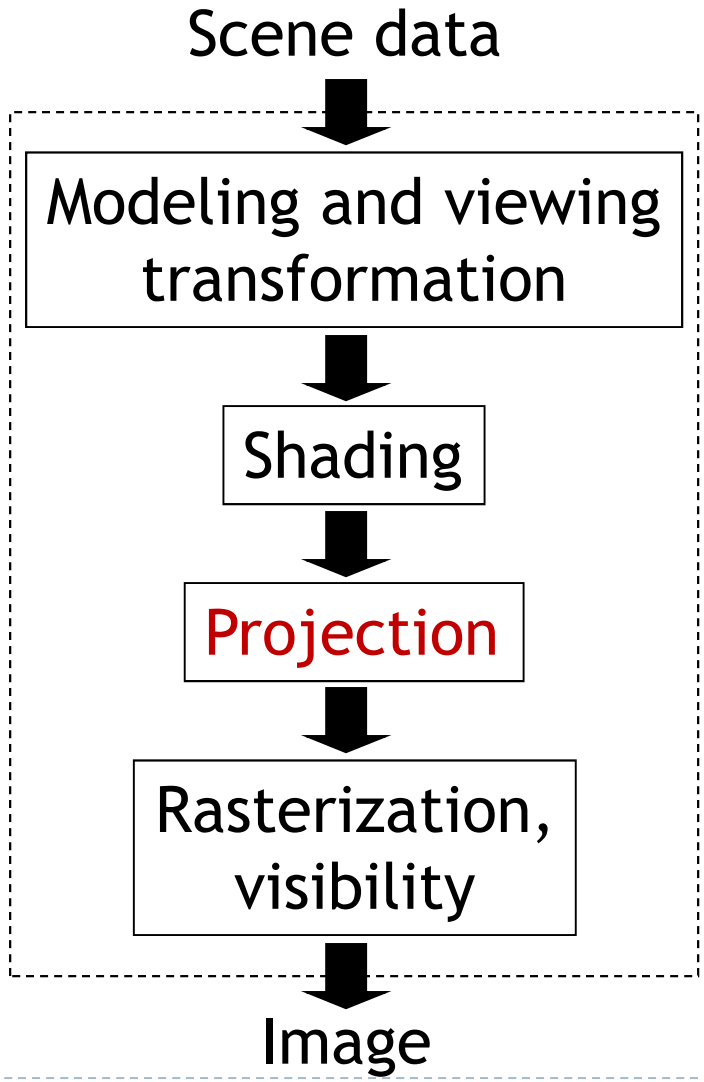
Rendering Pipeline



- Look up light sources
- Compute color for each vertex

UCSD

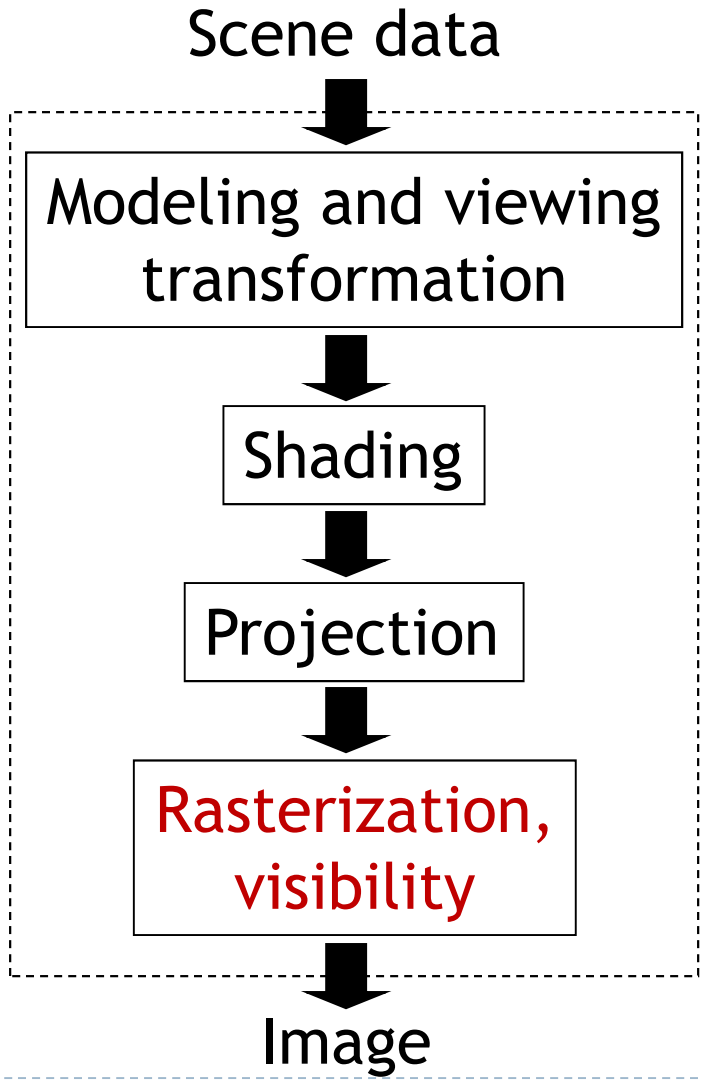
Rendering Pipeline



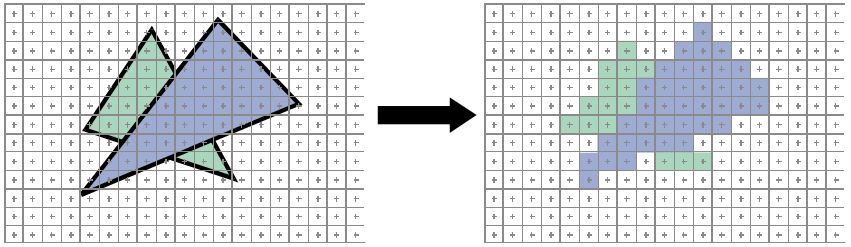
- Project 3D vertices to 2D image positions
- GL_PROJECTION matrix

UCSD

Rendering Pipeline

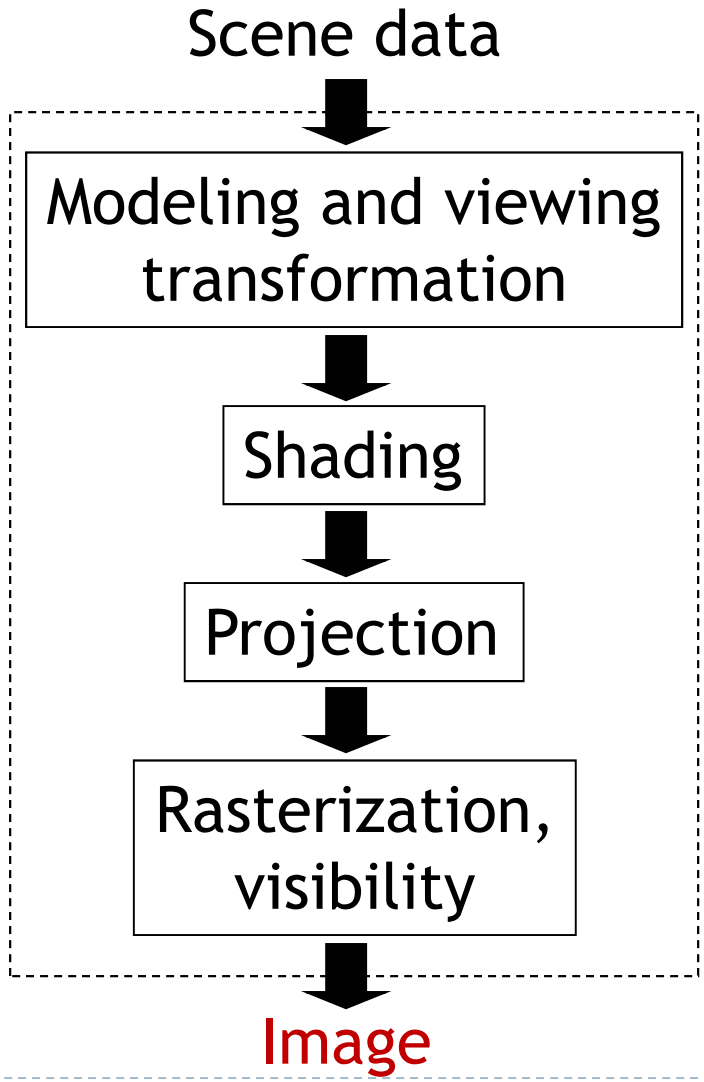


- Draw primitives (triangles, lines, etc.)
- Determine what is visible



UCSD

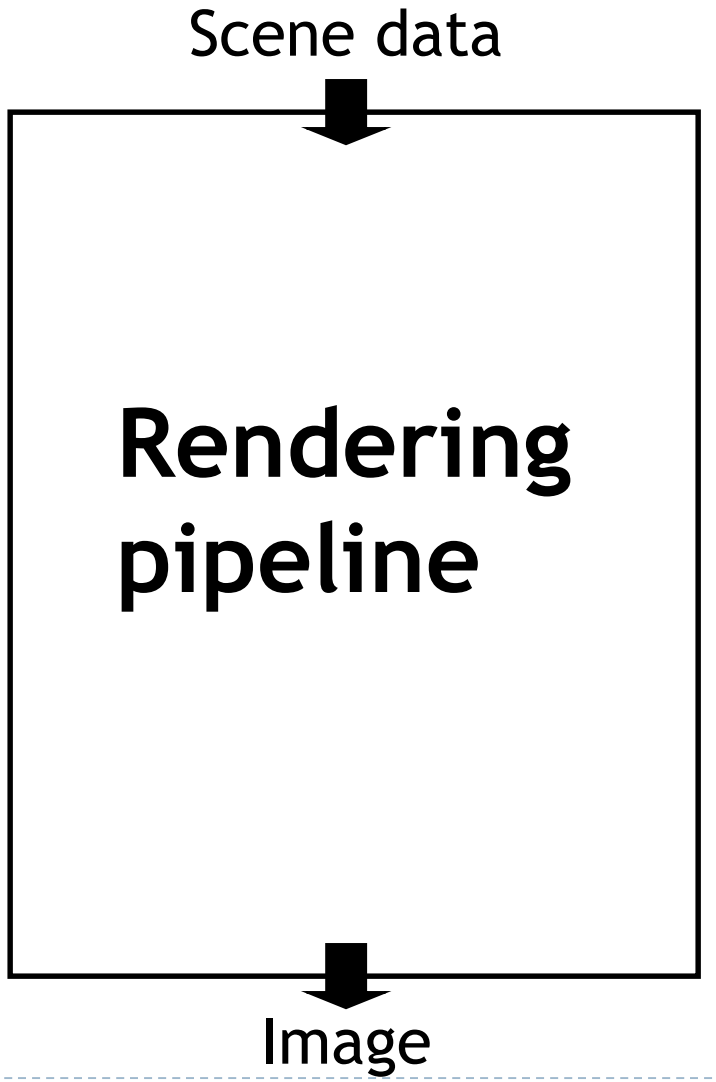
Rendering Pipeline



- Pixel colors

UCSD

Rendering Engine



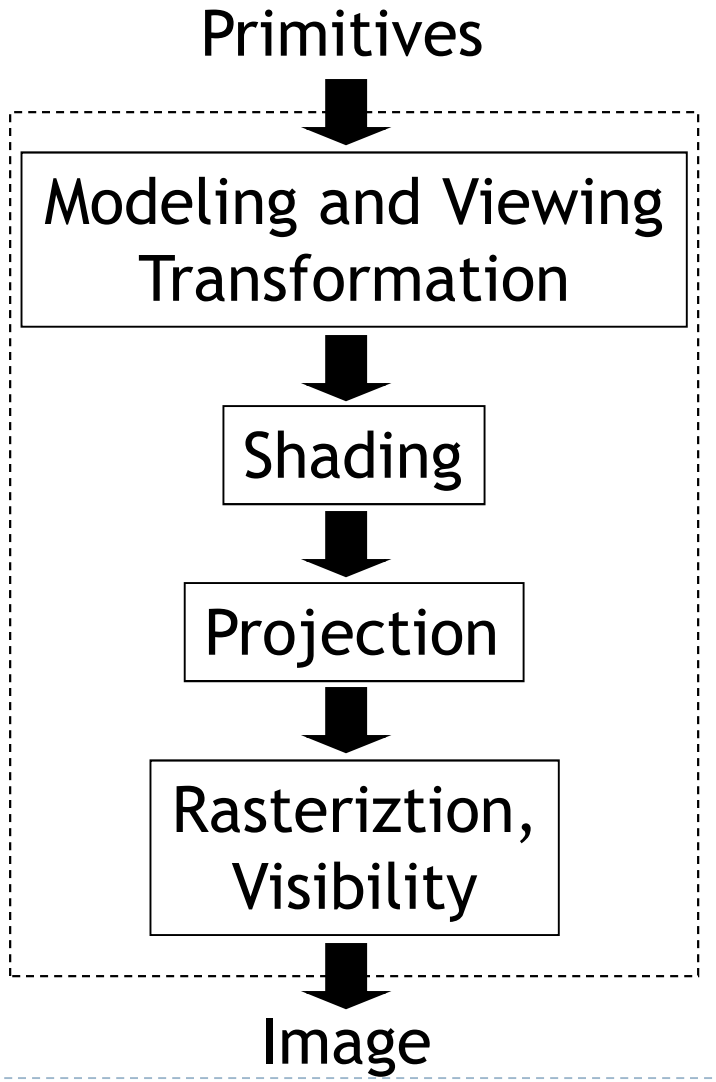
Rendering Engine:

- Additional software layer encapsulating low-level API
- Higher level functionality than OpenGL
- Platform independent
- Layered software architecture common in industry
  - Game engines
  - Graphics middleware

Lecture Overview

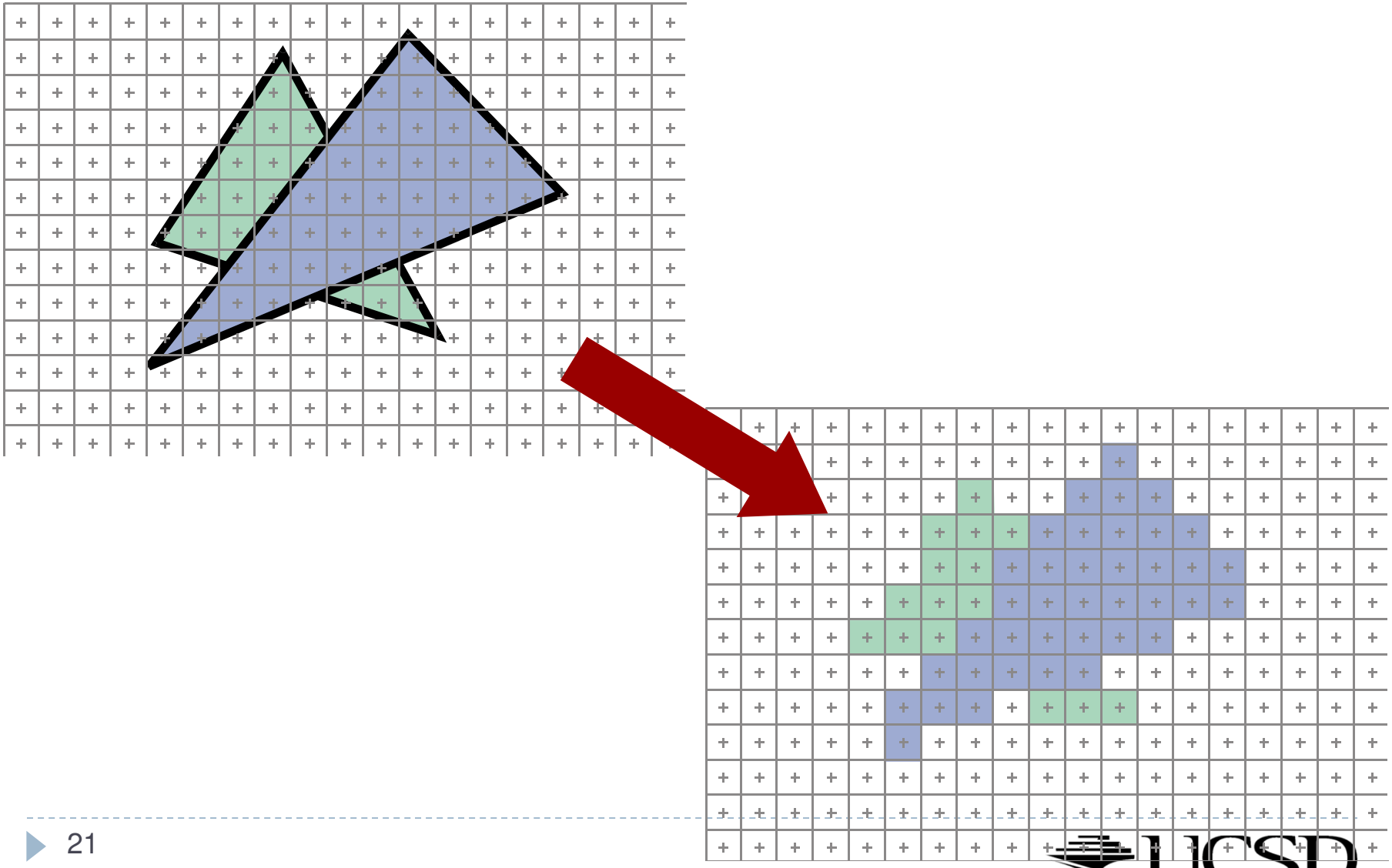
- Barycentric Coordinates
- Rendering Pipeline
- Rasterization
- Visibility

Rendering Pipeline



- Scan conversion and rasterization are synonyms
- One of the main operations performed by GPU
- Draw triangles, lines, points (squares)
- Focus on triangles in this lecture

Rasterization



Rasterization

- Given vertices in pixel coordinates

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

World space

Camera space

Clip space

Image space

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

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

Rasterization

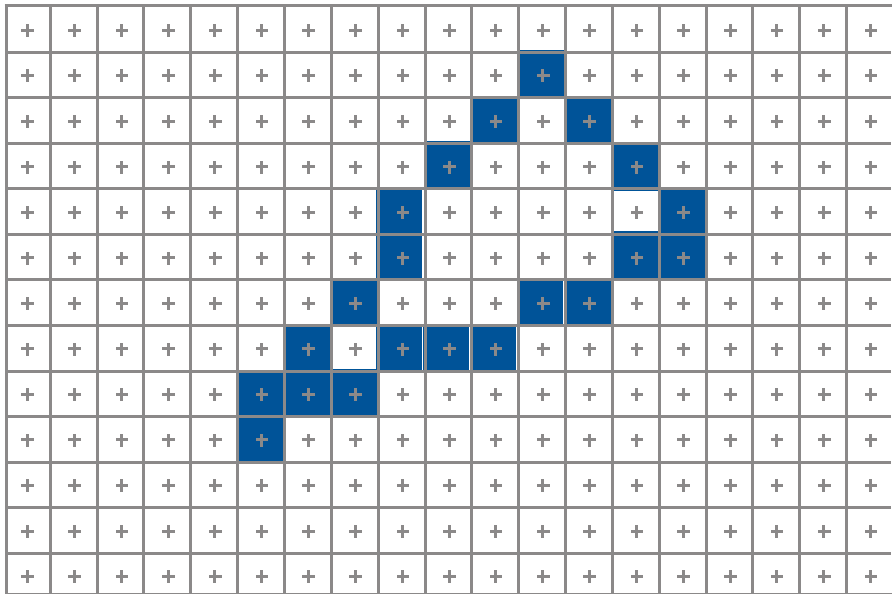
- How many pixels can a modern graphics processor draw per second?

Rasterization

- How many pixels can a modern graphics processor draw per second?
- NVidia GeForce GTX 980
  - 144 billion pixels per second
  - Multiple of what the fastest CPU could do

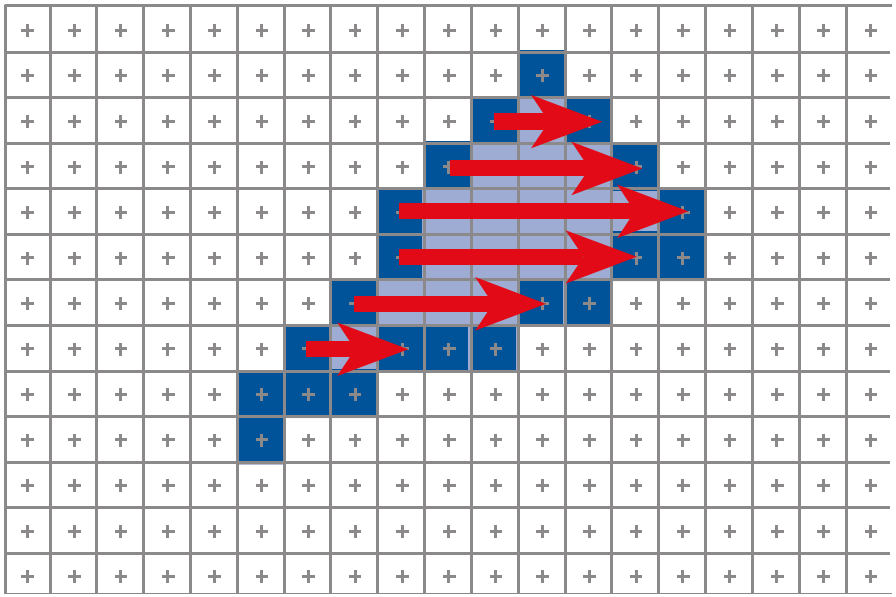
Rasterization

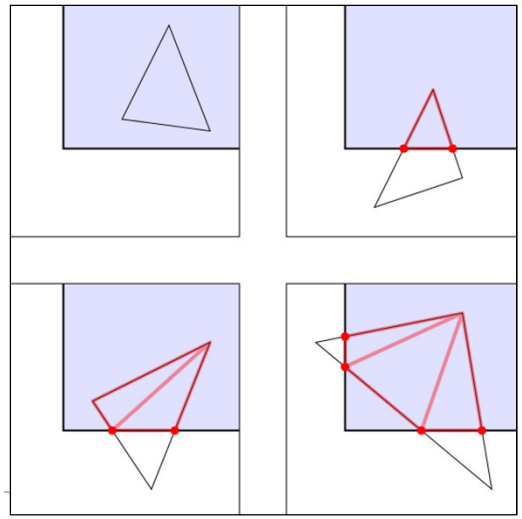
- Many different algorithms
- Old style
  - Rasterize edges first



Rasterization

- Many different algorithms
- Example:
  - Rasterize edges first
  - Fill the spans (scan lines)
- Disadvantage:
  - Requires clipping





Source: http://www.arcsynthesis.org

Rasterization

- Given vertices in pixel coordinates

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

World space

Camera space

Clip space

Image space

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

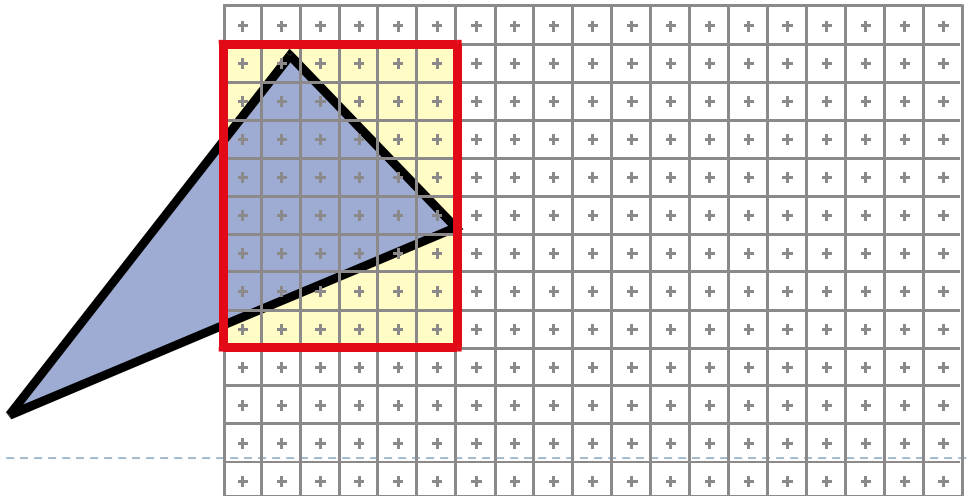
Pixel coordinates $\begin{matrix} x'/w' \\ y'/w' \end{matrix}$

Rasterization

- Simple algorithm

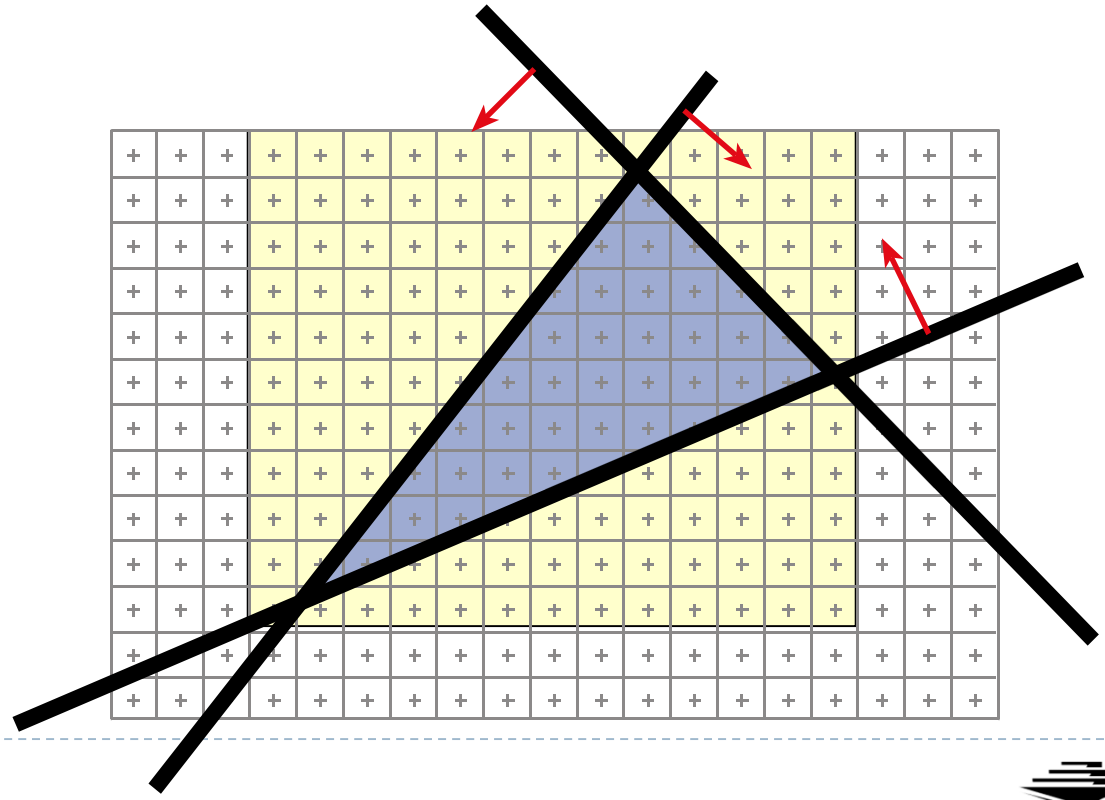
```
compute bbox
clip bbox to screen limits
for all pixels [x,y] in bbox
   compute barycentric coordinates alpha, beta, gamma
   if 0<alpha,beta,gamma<1 //pixel in triangle
            image[x,y]=triangleColor
```

- Bounding box clipping trivial



Rasterization

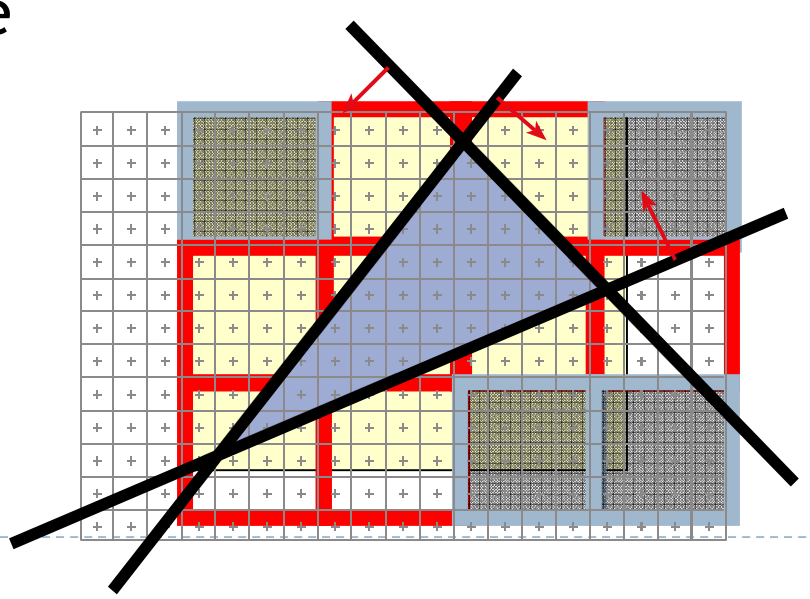
- So far, we compute barycentric coordinates of many useless pixels
- How can this be improved?



Rasterization

Hierarchy

- If block of pixels is outside triangle, no need to test individual pixels
- Can have several levels, usually two-level
- Find right granularity and size of blocks for optimal performance



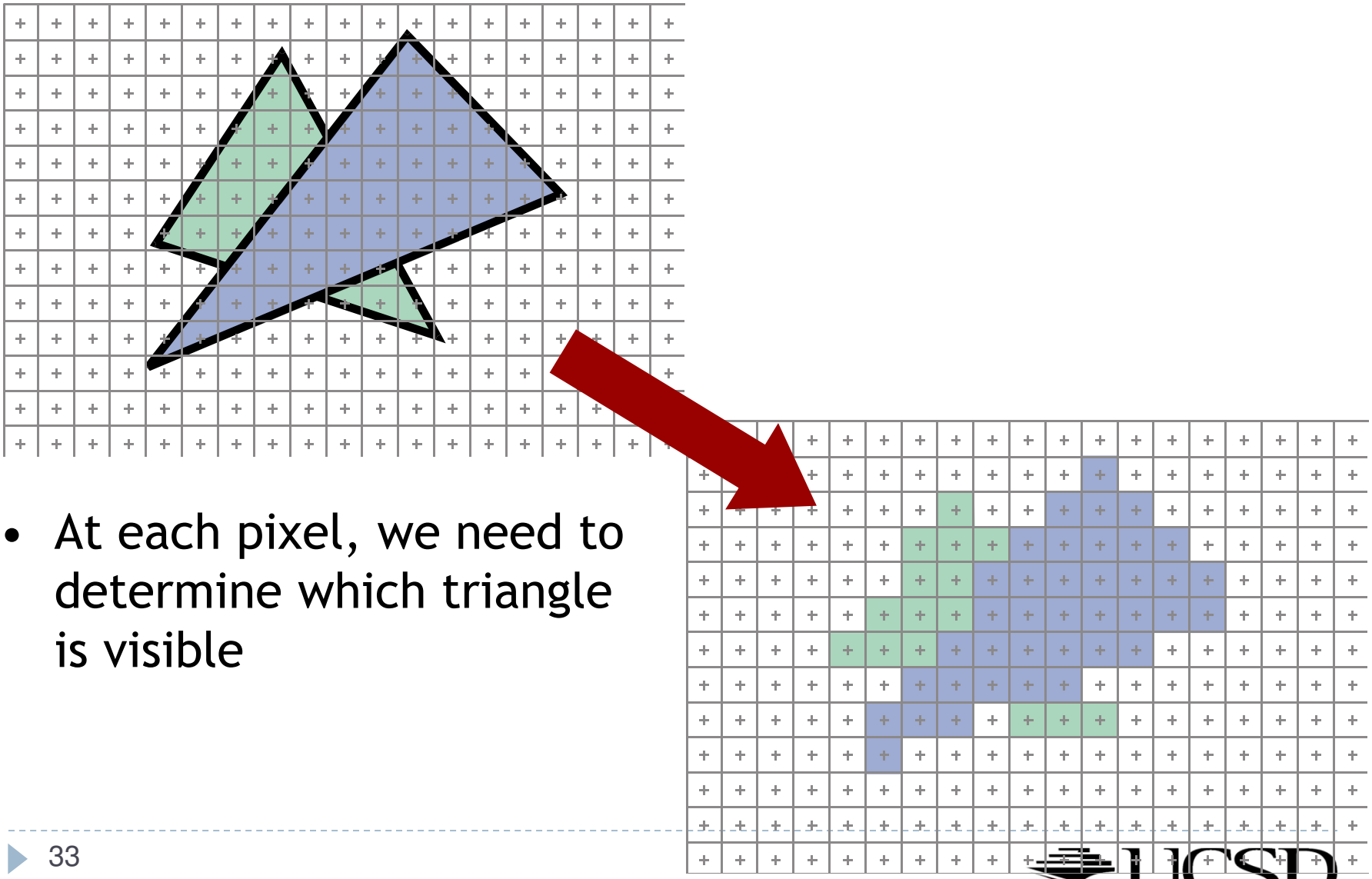
2D Triangle-Rectangle Intersection

- If one of the following tests returns true, the triangle intersects the rectangle:
  - Test if any of the triangle's vertices are inside the rectangle (e.g., by comparing the x/y coordinates to the min/max x/y coordinates of the rectangle)
  - Test if one of the quad's vertices is inside the triangle (e.g., using barycentric coordinates)
  - Intersect all edges of the triangle with all edges of the rectangle

Lecture Overview

- Barycentric Coordinates
- Rendering Pipeline
- Rasterization
- Visibility

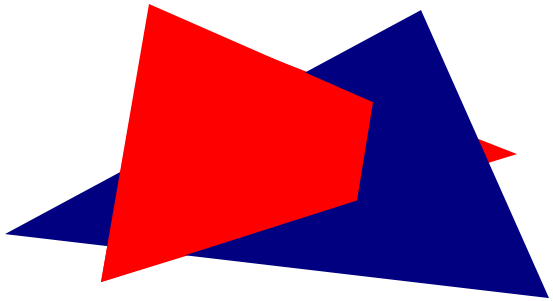
Visibility



- At each pixel, we need to determine which triangle is visible

Painter's Algorithm

- Paint from back to front
- Every new pixel always paints over previous pixel in frame buffer
- Need to sort geometry according to depth
- May need to split triangles if they intersect



- Outdated algorithm, created when memory was expensive

Z-Buffering

- Store z-value for each pixel
- Depth test
  - During rasterization, compare stored value to new value
  - Update pixel only if new value is smaller

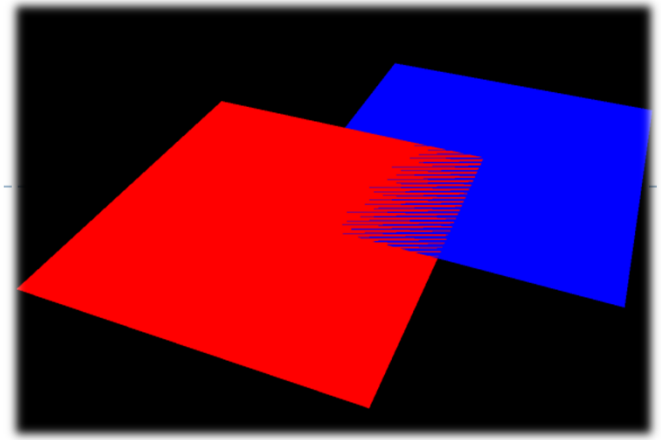
```
setpixel(int x, int y, color c, float z)
if(z<zbuffer(x,y)) then
  zbuffer(x,y) = z
  color(x,y) = c
```

- z-buffer is dedicated memory reserved for GPU (graphics memory)
- Depth test is performed by GPU

Z-Buffering in OpenGL

- In your application:
  - Ask for a depth buffer when you create your window.
  - Place a call to glEnable (GL_DEPTH_TEST) in your program's initialization routine.
  - Ensure that your *zNear* and *zFar* clipping planes are set correctly (in glOrtho, glFrustum or gluPerspective) and in a way that provides adequate depth buffer precision.
  - Pass GL_DEPTH_BUFFER_BIT as a parameter to glClear.
- Note that the z buffer is non-linear: it uses smaller depth bins in the foreground, larger ones further from the camera.

Z-Buffer Fighting



- Problem: polygons which are close together don't get rendered correctly. Errors change with camera perspective → flicker
- Cause: differently colored fragments from different polygons are being rasterized to same pixel and depth → not clear which is in front of which
- Solutions:
  - move surfaces farther apart, so that fragments rasterize into different depth bins
  - bring near and far planes closer together
  - use a higher precision depth buffer. Note that GLUT often defaults to 16 bit even if your graphics card supports 24 bit or 32 bit depth buffers

Translucent Geometry

- Need to depth sort translucent geometry and render with Painter's Algorithm (back to front)
- Problem: incorrect blending with cyclically overlapping geometry

- Solutions:
  - Storage of multiple depth and color values per pixel (not practical in real-time graphics)
  - Or back to front rendering of translucent geometry, after rendering opaque geometry
    - Does not always work correctly: programmer has to weight rendering correctness against computational effort