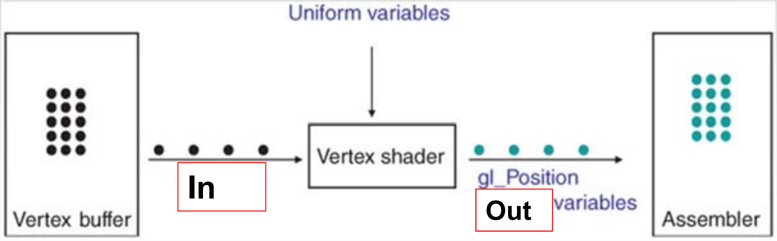
CPSC 436D
Video Game Programming

Rendering

vertex shader

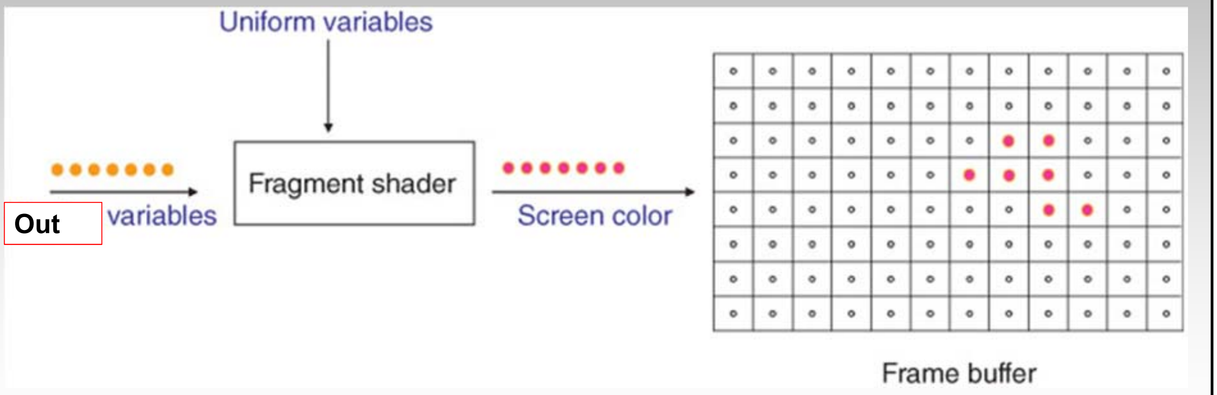
Vertices and attributes → Vertex Shader →

- VS is run for each vertex SEPARATELY
- By default doesn't know connectivity
- Input: vertex coordinates in Object Coordinate System
- It's main goal is to set **gl_Position**



Object coordinates -> WORLD coordinates -> VIEW coordinates

fragment SHADER



concepts

uniform **C/C++→ Vertex Shader → Fragment Shader**

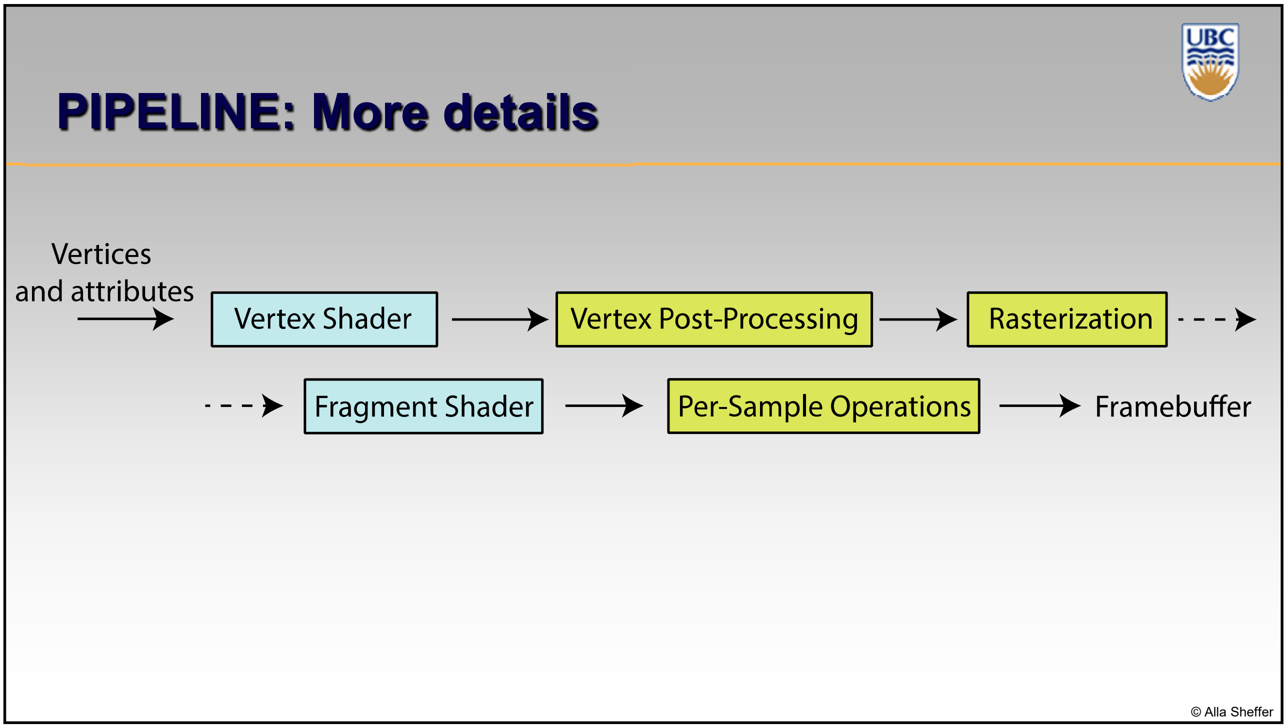
- same for all vertices

Out (varying) **Vertex Shader → Fragment Shader**

- computed per vertex, automatically interpolated for fragments

In (attribute) **C/C++ → Vertex Shader**

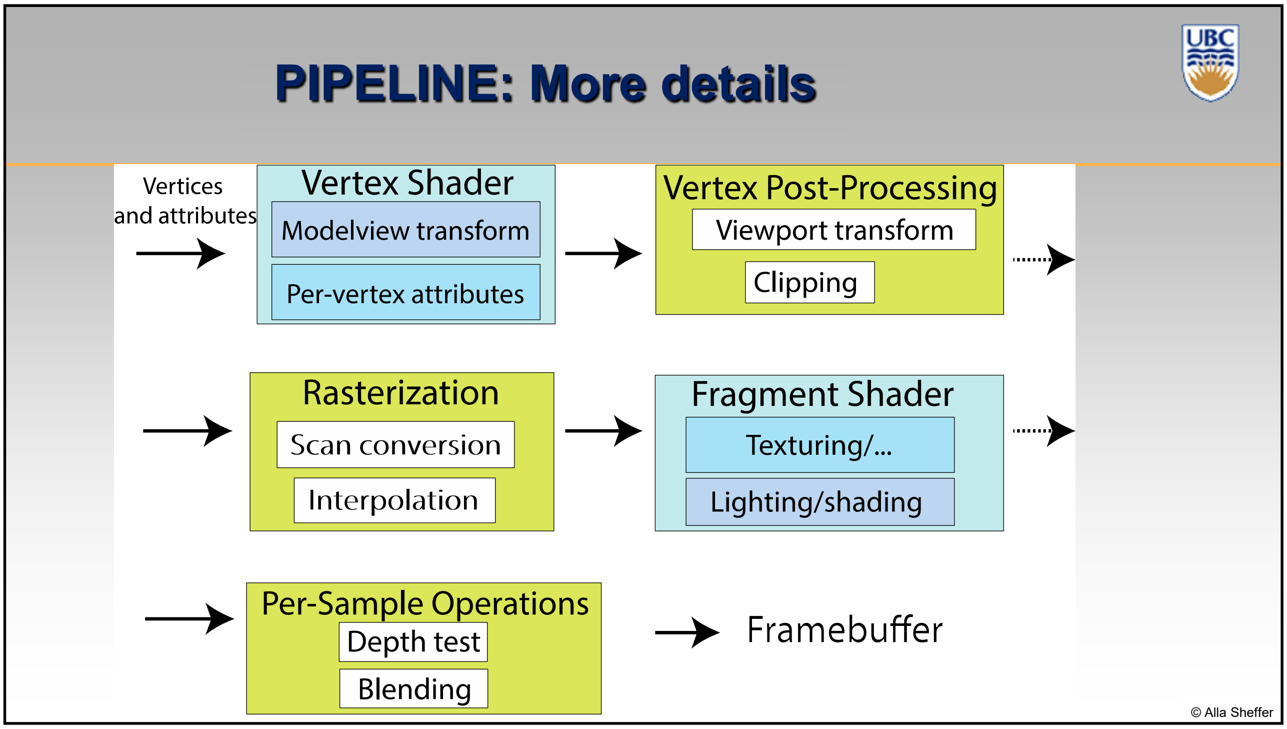
- some values per vertex
- available only in Vertex Shader



PIPELINE: More details

Vertices and attributes → Vertex Shader → Vertex Post-Processing → Rasterization - - →

- - → Fragment Shader → Per-Sample Operations → Framebuffer



PIPELINE: More details

Vertices and attributes →

Vertex Shader
- Modelview transform
- Per-vertex attributes

→

Vertex Post-Processing
- Viewport transform
- Clipping

→

Rasterization
- Scan conversion
- Interpolation

→

Fragment Shader
- Texturing/...
- Lighting/shading

→

Per-Sample Operations
- Depth test
- Blending

→ Framebuffer

Shapes - Curves/Surfaces

Mathematical representations:

- Explicit functions: y = f(x)
  - *Rarely useful*
- Parametric functions
- Implicit functions

Shapes: Parametric Functions

Curves:

- 2D: x and y are functions of a parameter value t
- 3D: x, y, and z are functions of a parameter value t

$$C(t) := \begin{pmatrix} P_y^0 \\ P_x^0 \end{pmatrix} t + \begin{pmatrix} P_y^1 \\ P_x^1 \end{pmatrix} (1-t)$$

$$C(t) := \begin{pmatrix} \cos t \\ \sin t \end{pmatrix}$$

Shapes: Implicit

Curve (2D) or Surface (3D) defined by zero set (roots) of function

- E.g:

$$S(x,y): x^2 + y^2 - 1 = 0$$

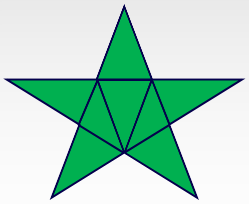
$$S(x,y,z): x^2 + y^2 + z^2 - 1 = 0$$

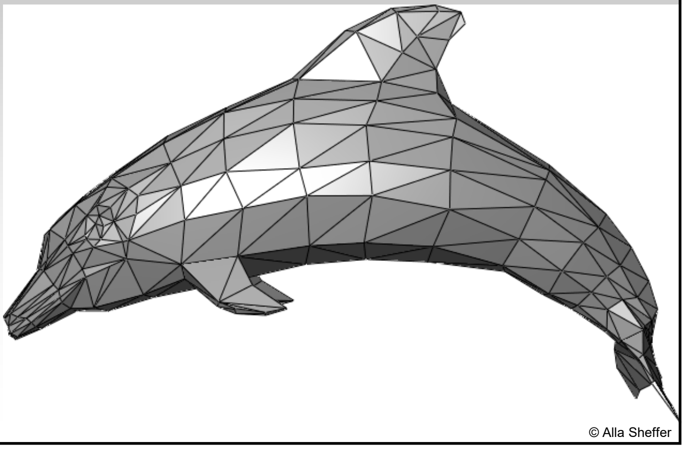
Shapes: Triangle Meshes

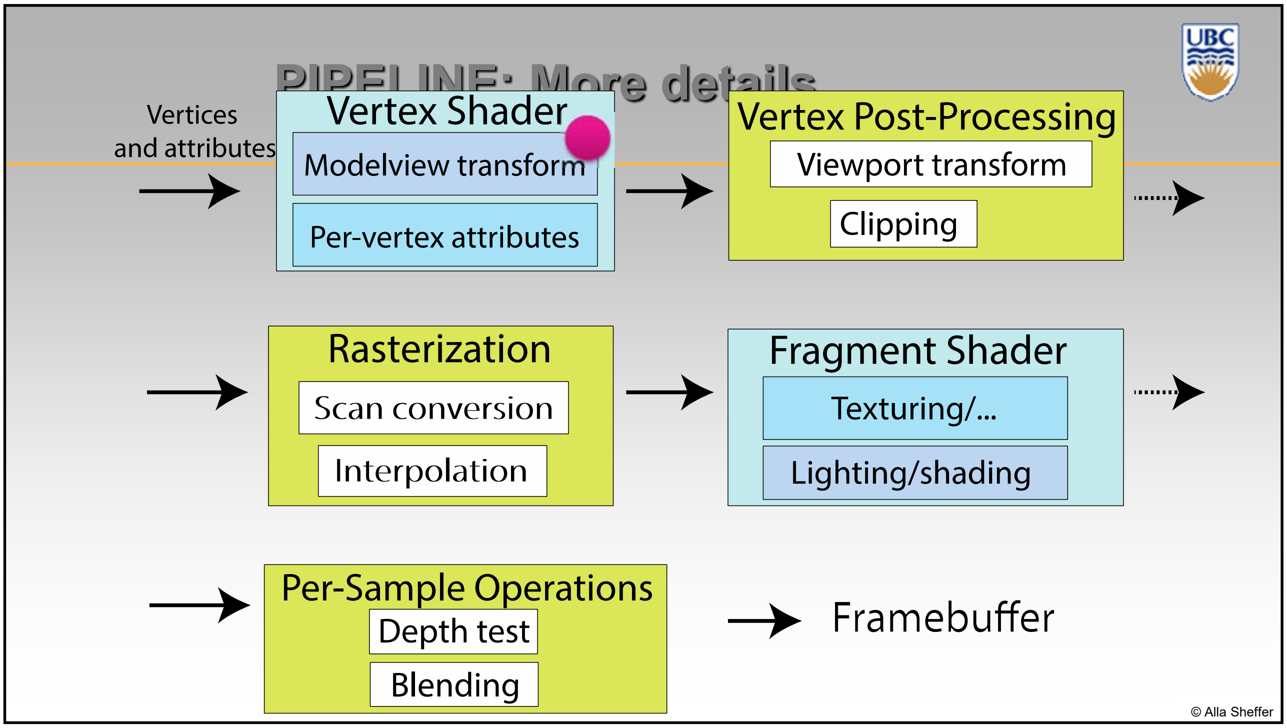
Triangle = 3 vertices

Mesh = {vertices, triangles}

Examples







PIPELINE: More details

Vertices and attributes →

Vertex Shader
- Modelview transform
- Per-vertex attributes

→

Vertex Post-Processing
- Viewport transform
- Clipping

→

Rasterization
- Scan conversion
- Interpolation

→

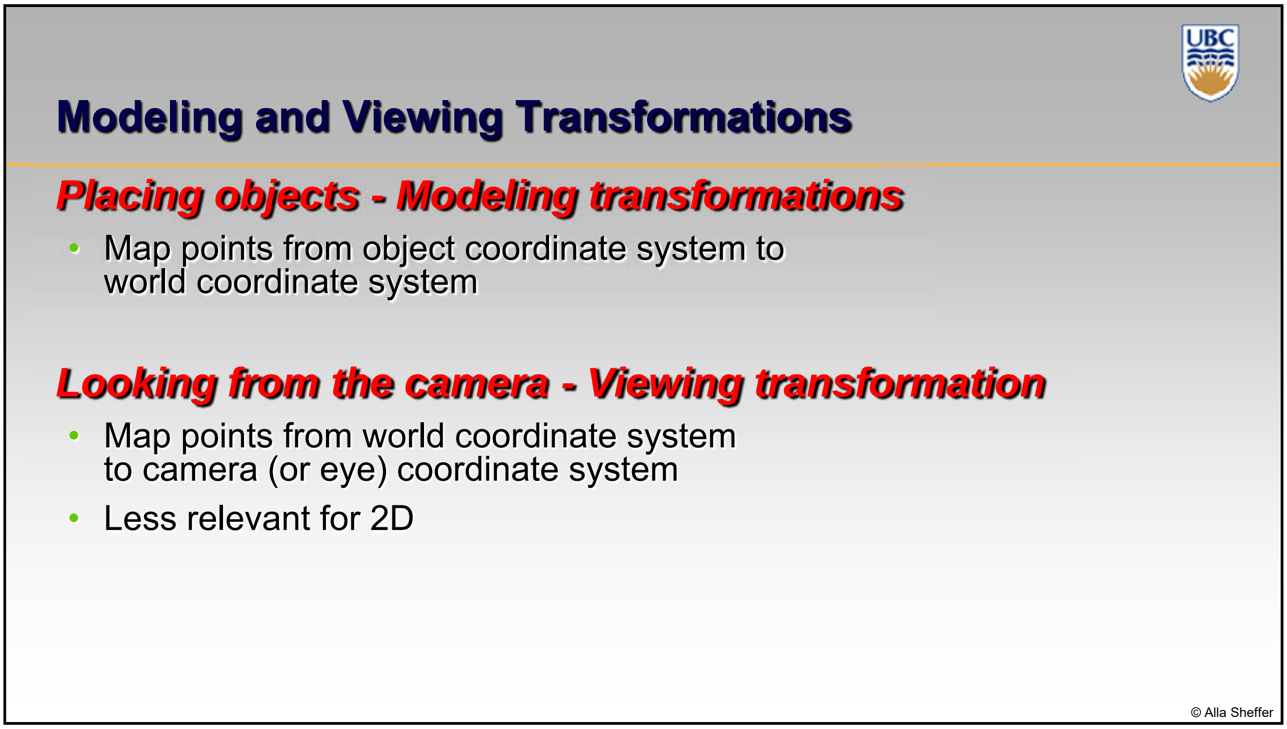
Fragment Shader
- Texturing/...
- Lighting/shading

→

Per-Sample Operations
- Depth test
- Blending

→ Framebuffer

© Alla Sheffer



Modeling and Viewing Transformations

Placing objects - Modeling transformations

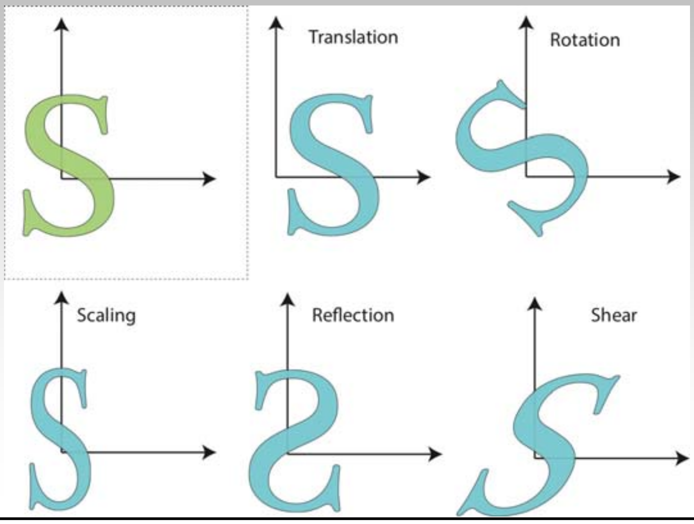
- Map points from object coordinate system to world coordinate system

Looking from the camera - Viewing transformation

- Map points from world coordinate system to camera (or eye) coordinate system
- Less relevant for 2D

© Alla Sheffer

Modeling Transformations



© Alla Sheffer

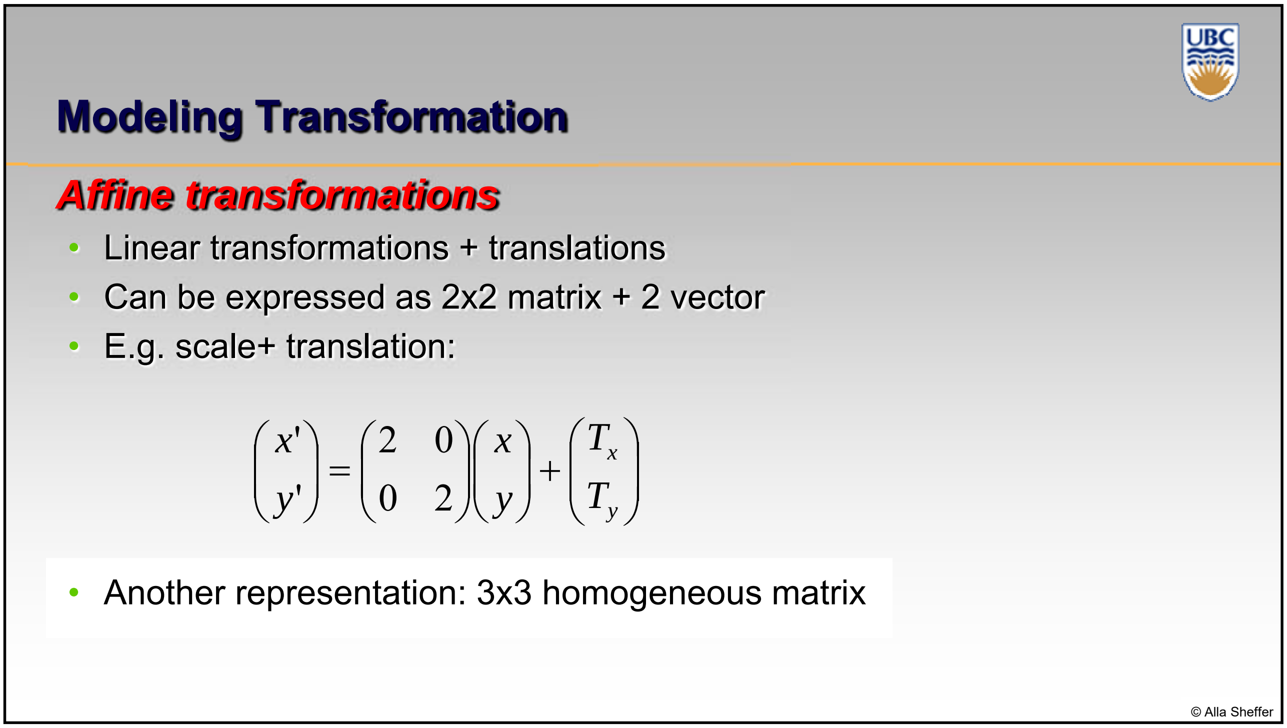
Modeling Transformation

Linear transformations

- Rotations, scaling, shearing
- Can be expressed as 2x2 matrix (2D)
- E.g.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

© Alla Sheffer



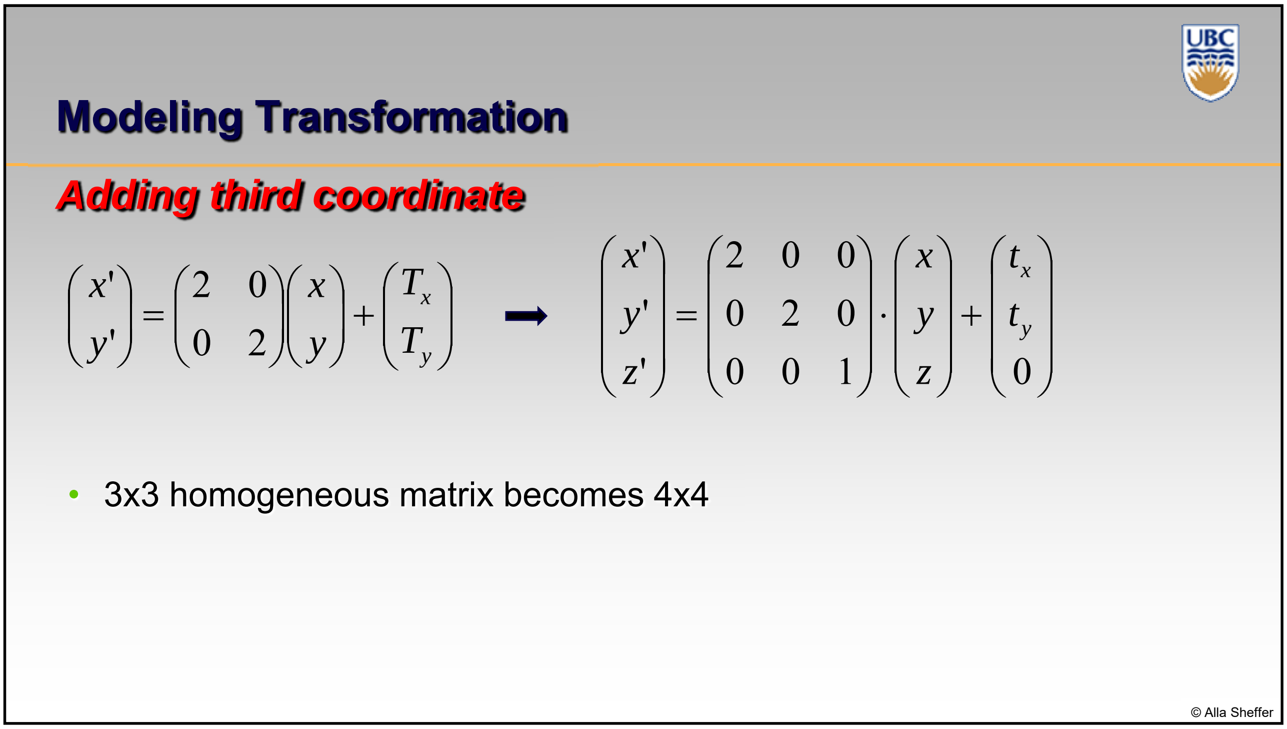
Modeling Transformation

Affine transformations

- Linear transformations + translations
- Can be expressed as 2x2 matrix + 2 vector
- E.g. scale+ translation:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} T_x \\ T_y \end{pmatrix}$$

- Another representation: 3x3 homogeneous matrix

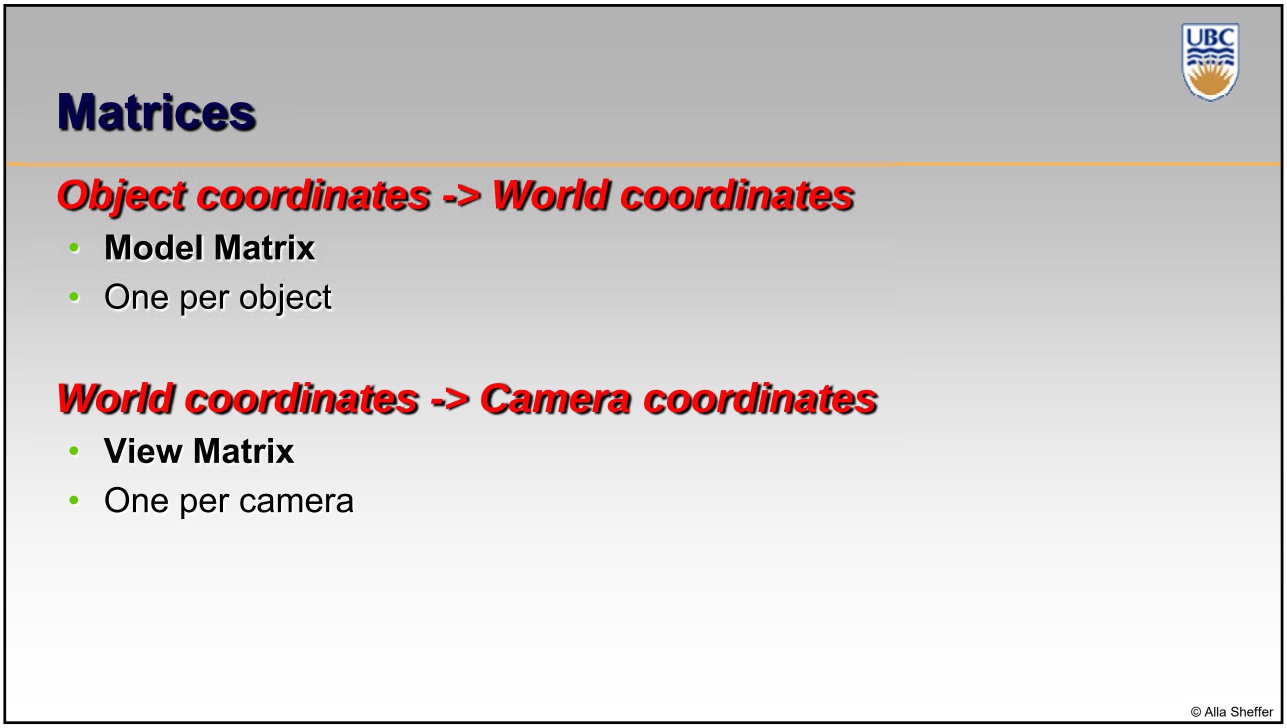


Modeling Transformation

Adding third coordinate

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} T_x \\ T_y \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ 0 \end{pmatrix}$$

- 3x3 homogeneous matrix becomes 4x4



Matrices

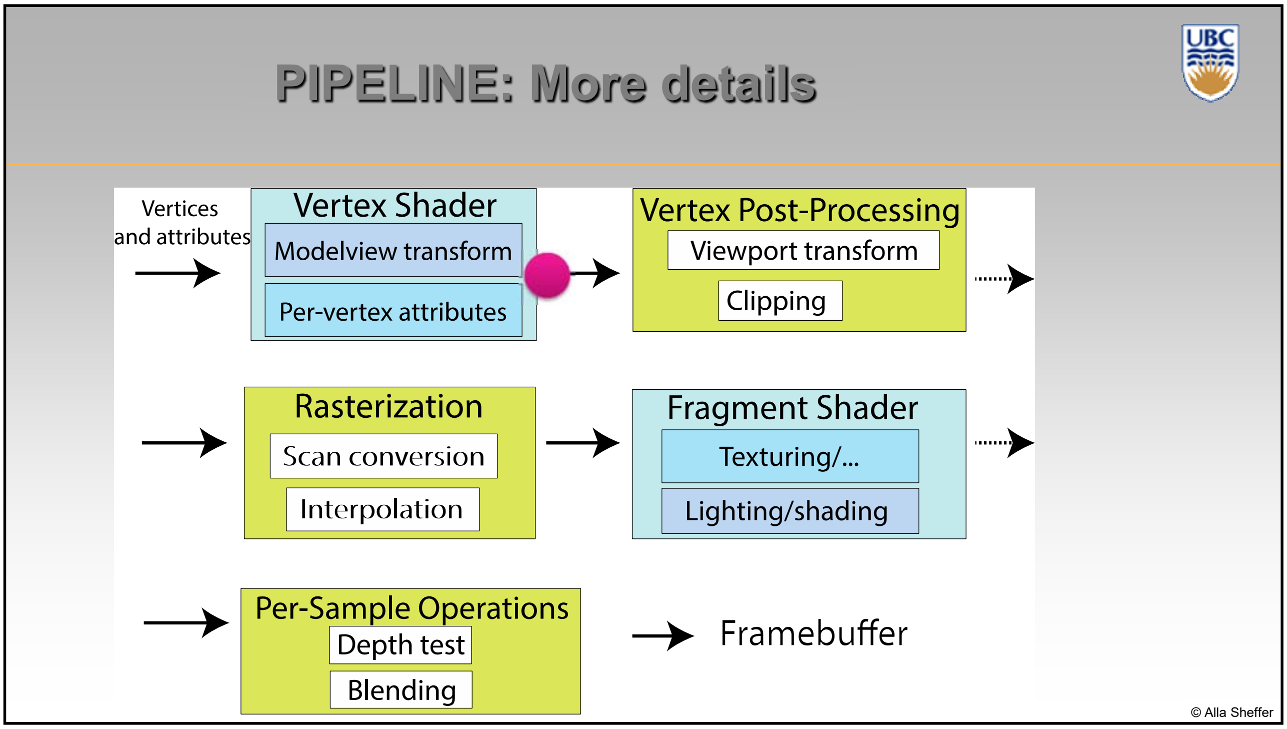
Object coordinates -> World coordinates

- **Model Matrix**
- One per object

World coordinates -> Camera coordinates

- **View Matrix**
- One per camera

© Alla Sheffer



PIPELINE: More details

Vertices and attributes →

Vertex Shader
- Modelview transform
- Per-vertex attributes

→ Vertex Post-Processing
- Viewport transform
- Clipping

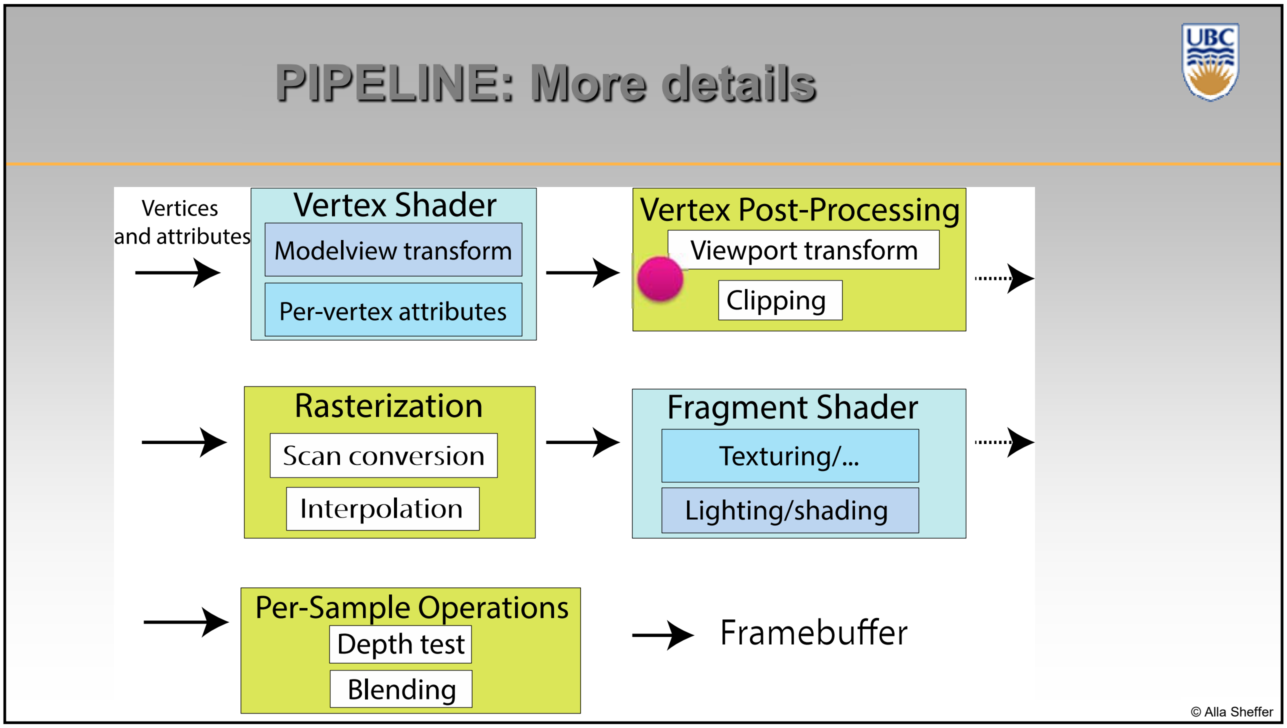
→ Rasterization
- Scan conversion
- Interpolation

→ Fragment Shader
- Texturing/...
- Lighting/shading

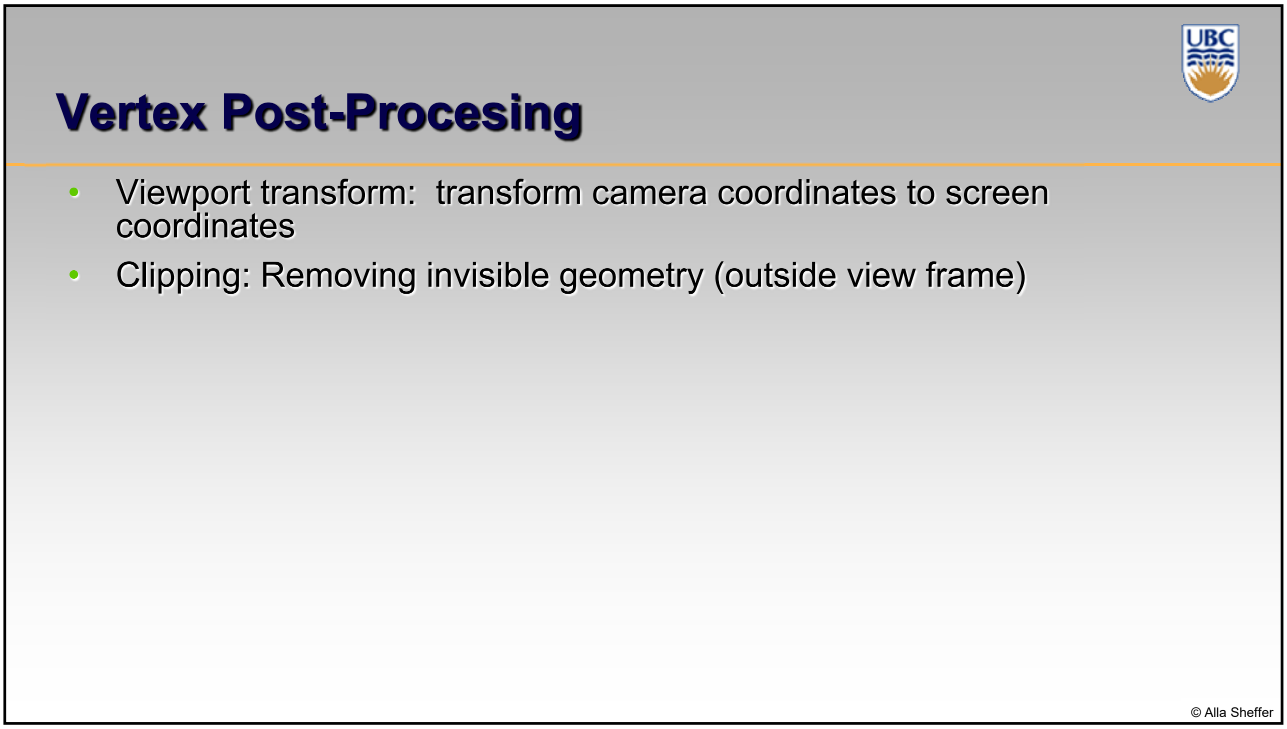
→ Per-Sample Operations
- Depth test
- Blending

→ Framebuffer

© Alla Sheffer

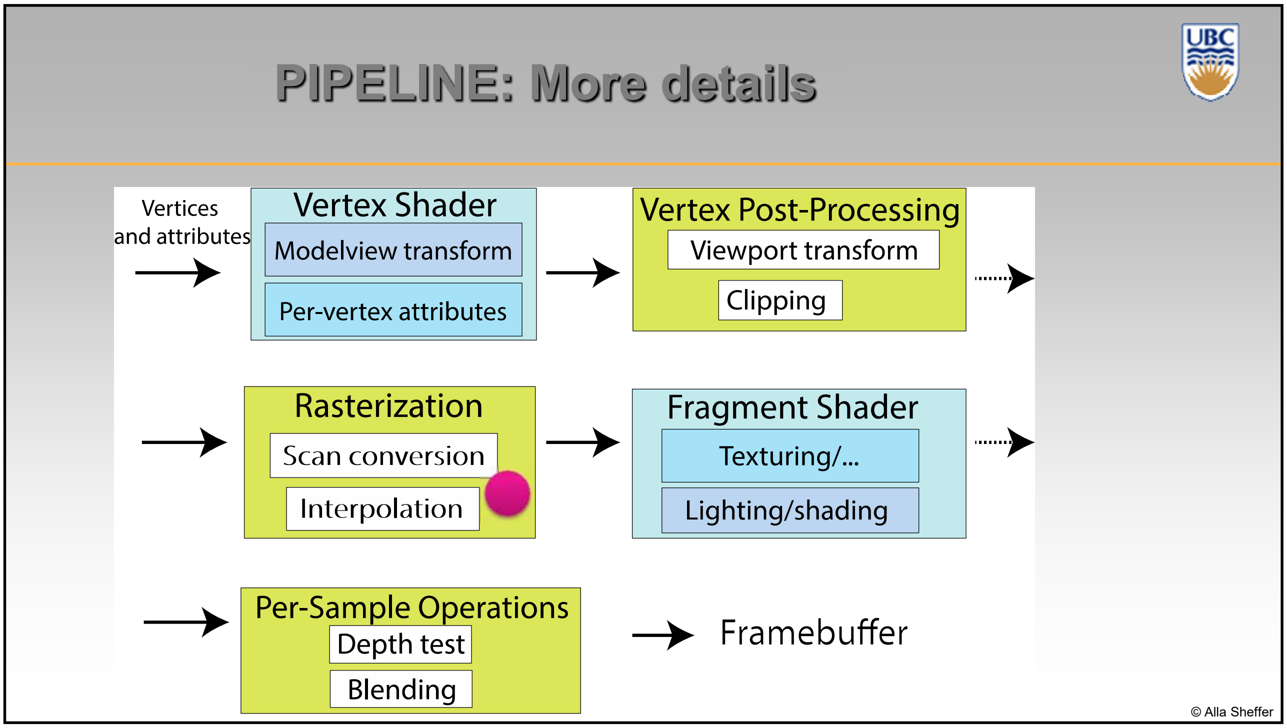


PIPELINE: More details



Vertex Post-Procesing

- Viewport transform: transform camera coordinates to screen coordinates
- Clipping: Removing invisible geometry (outside view frame)



PIPELINE: More details

Vertices and attributes →

Vertex Shader
- Modelview transform
- Per-vertex attributes

→

Vertex Post-Processing
- Viewport transform
- Clipping

→

Rasterization
- Scan conversion
- Interpolation

→

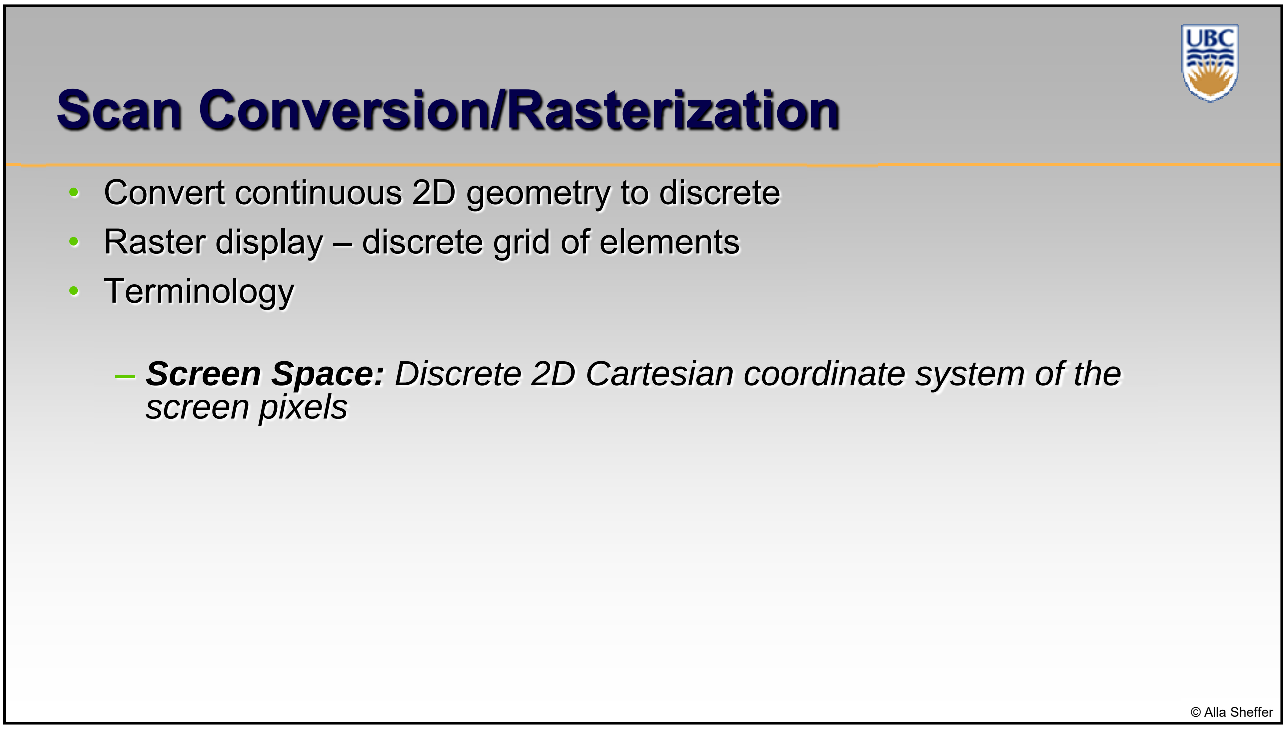
Fragment Shader
- Texturing/...
- Lighting/shading

→

Per-Sample Operations
- Depth test
- Blending

→ Framebuffer

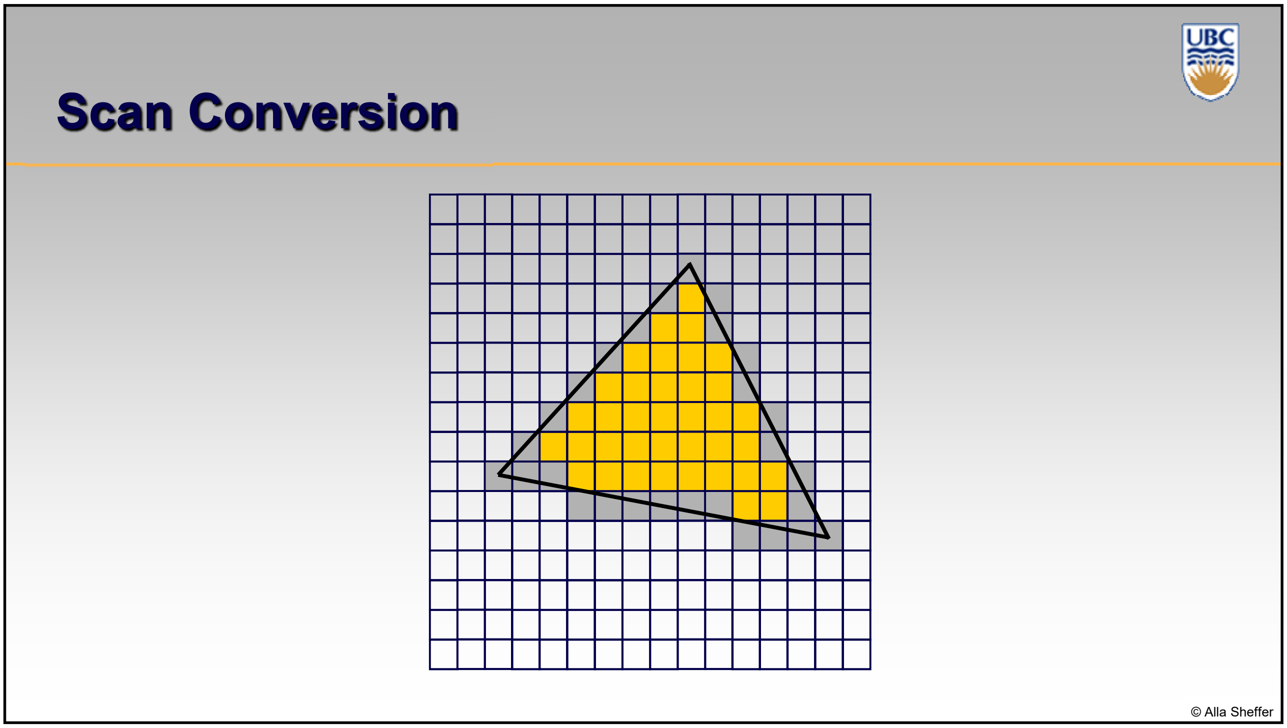
© Alla Sheffer



Scan Conversion/Rasterization

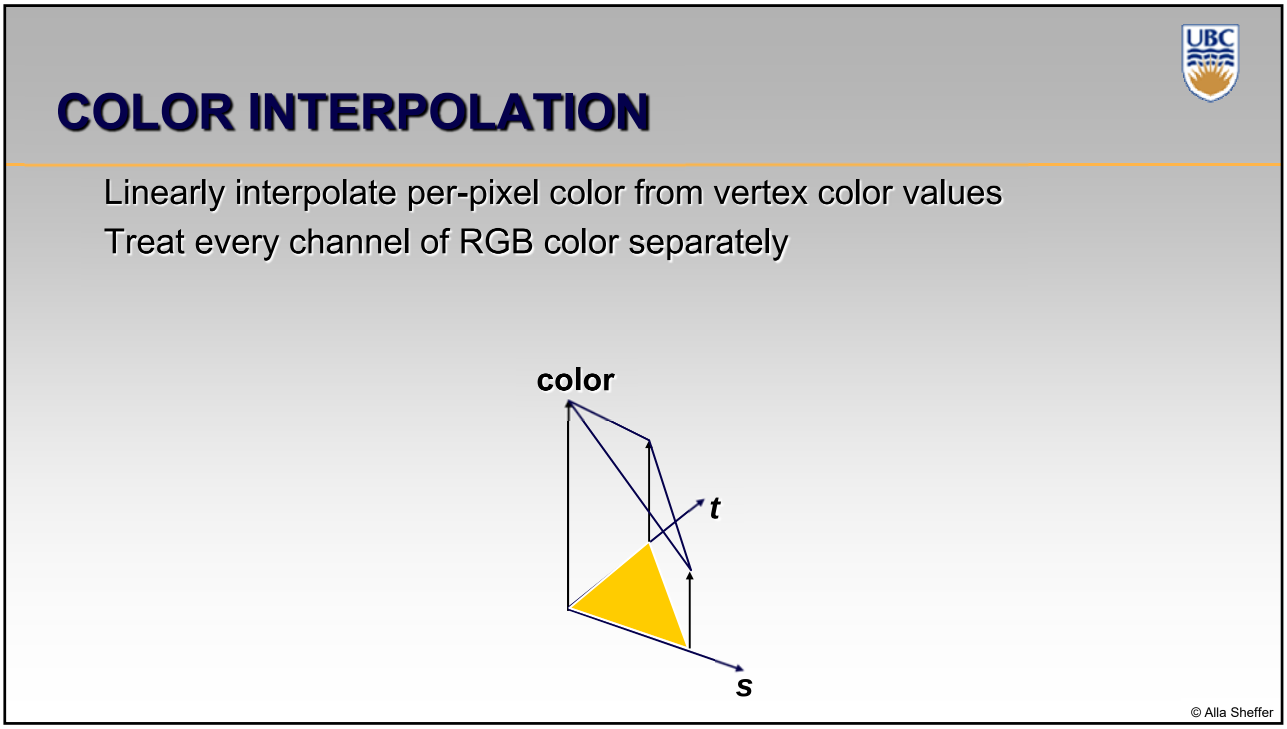
- Convert continuous 2D geometry to discrete
- Raster display – discrete grid of elements
- Terminology
  - ***Screen Space:*** *Discrete 2D Cartesian coordinate system of the screen pixels*

© Alla Sheffer



Scan Conversion

© Alla Sheffer

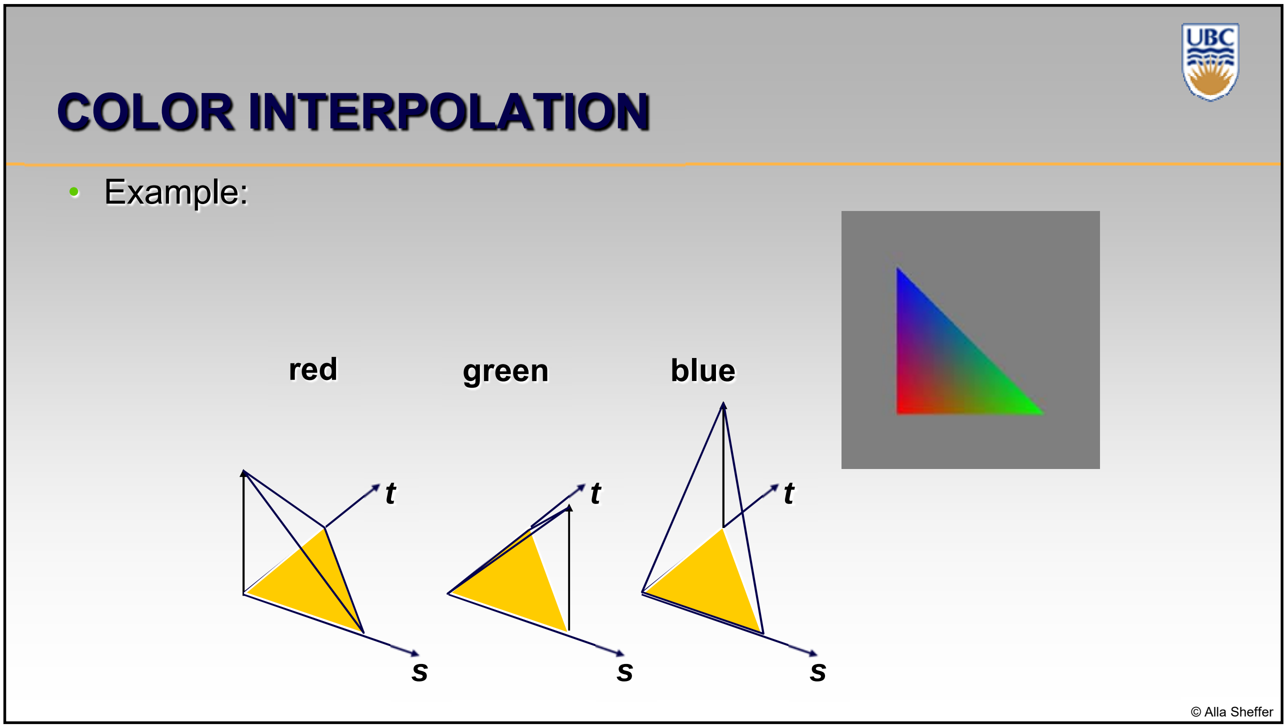


COLOR INTERPOLATION

Linearly interpolate per-pixel color from vertex color values

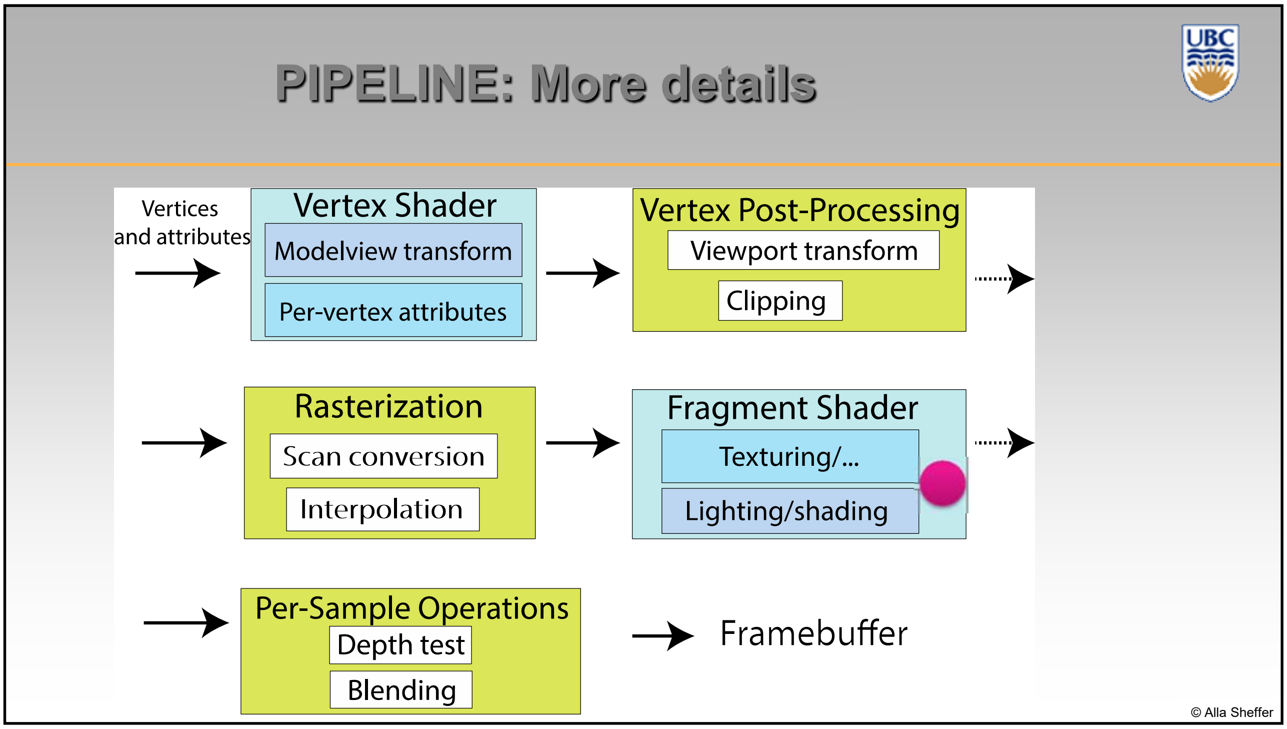
Treat every channel of RGB color separately

© Alla Sheffer



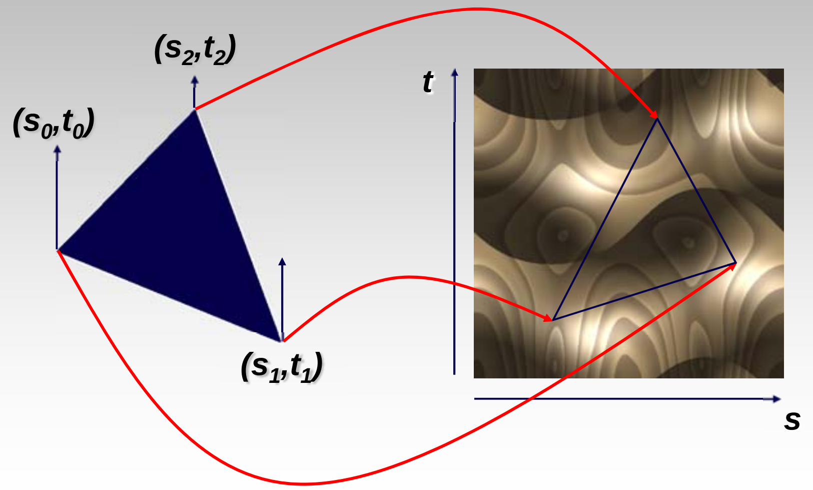
COLOR INTERPOLATION

- Example:

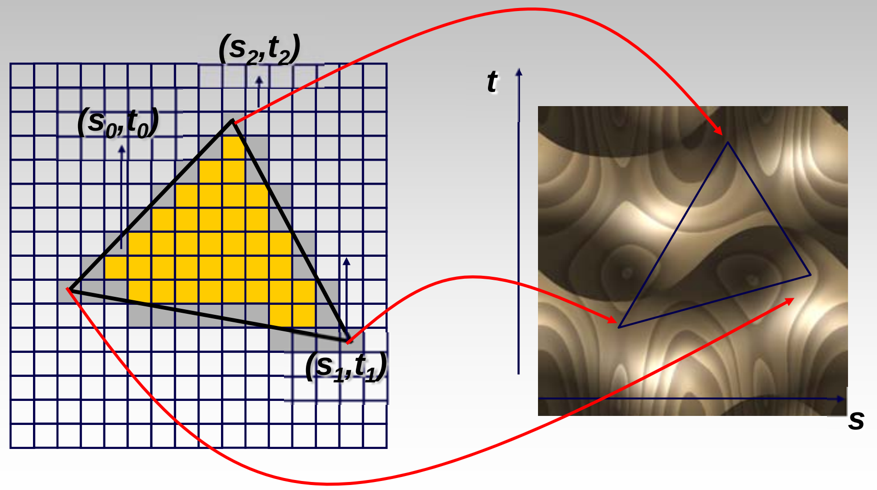


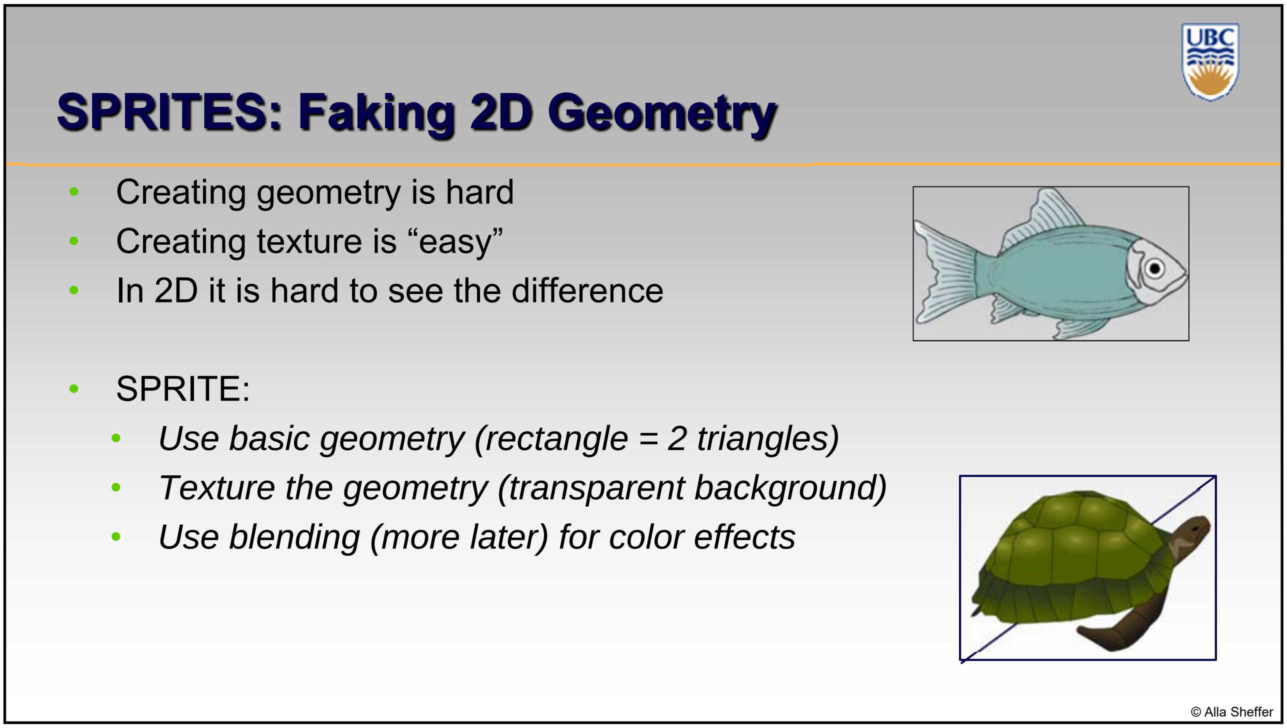
PIPELINE: More details

Texturing



Texturing

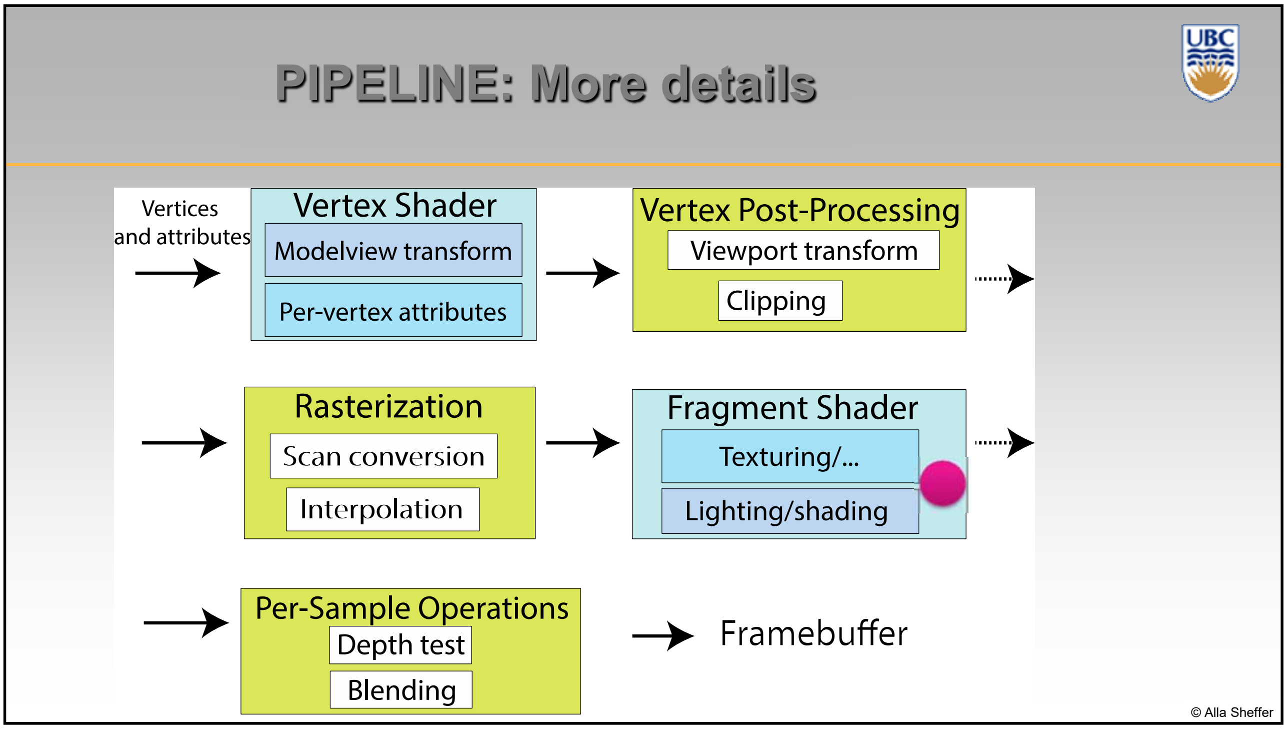




SPRITES: Faking 2D Geometry

- Creating geometry is hard
- Creating texture is “easy”
- In 2D it is hard to see the difference

- SPRITE:
  - *Use basic geometry (rectangle = 2 triangles)*
  - *Texture the geometry (transparent background)*
  - *Use blending (more later) for color effects*



PIPELINE: More details

Vertices and attributes →

Vertex Shader
- Modelview transform
- Per-vertex attributes

→

Vertex Post-Processing
- Viewport transform
- Clipping

→

Rasterization
- Scan conversion
- Interpolation

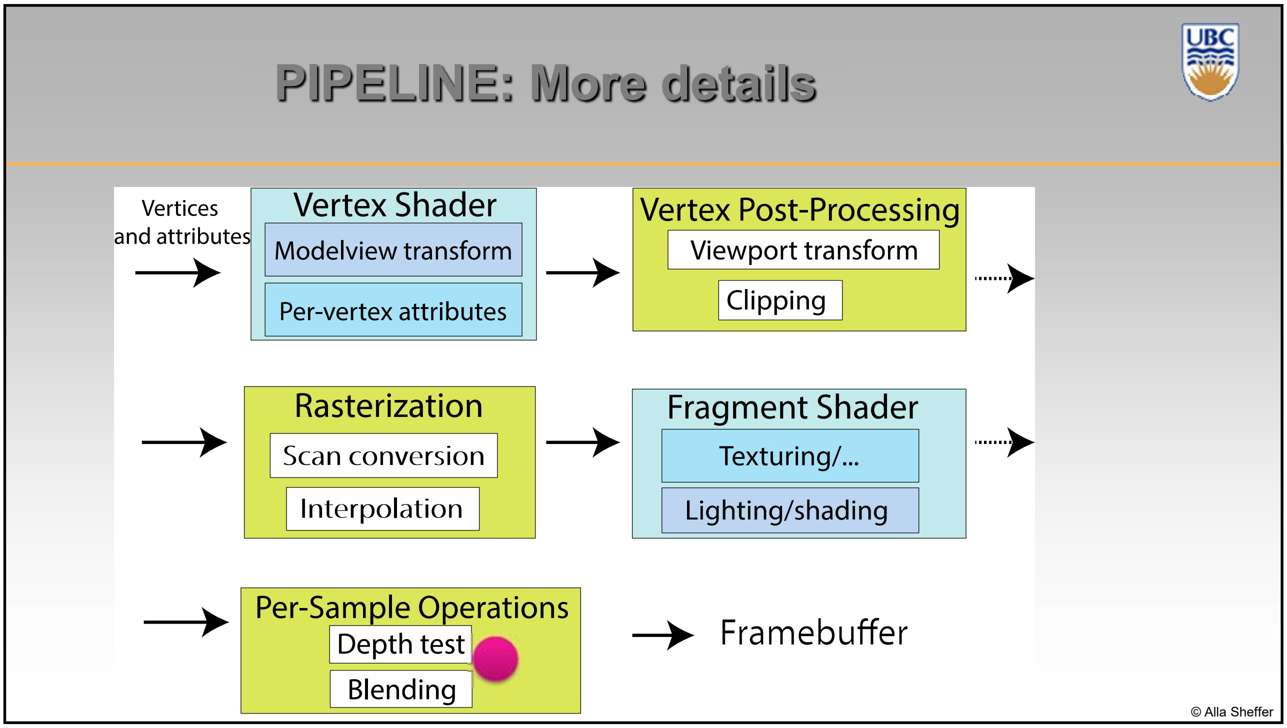
→

Fragment Shader
- Texturing/...
- Lighting/shading

→

Per-Sample Operations
- Depth test
- Blending

→ Framebuffer



PIPELINE: More details

Vertices and attributes →

Vertex Shader
- Modelview transform
- Per-vertex attributes

→

Vertex Post-Processing
- Viewport transform
- Clipping

→

Rasterization
- Scan conversion
- Interpolation

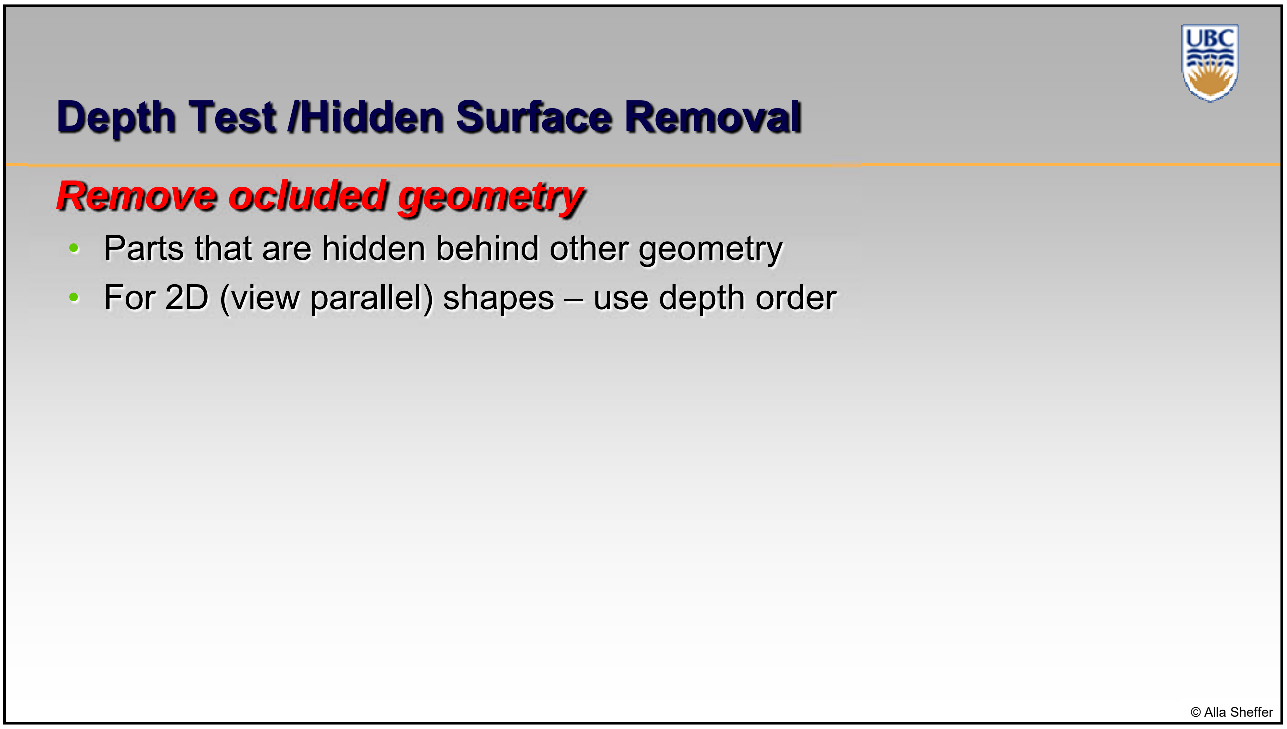
→

Fragment Shader
- Texturing/...
- Lighting/shading

→

Per-Sample Operations
- Depth test
- Blending

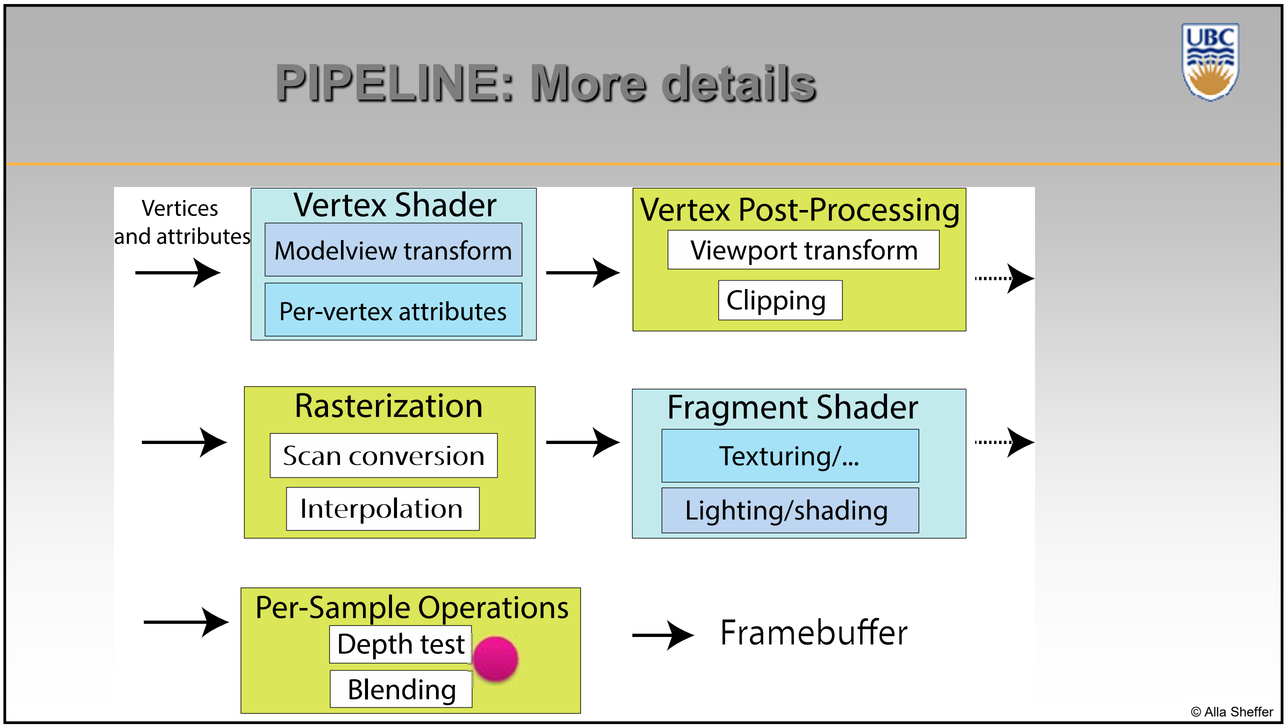
→ Framebuffer



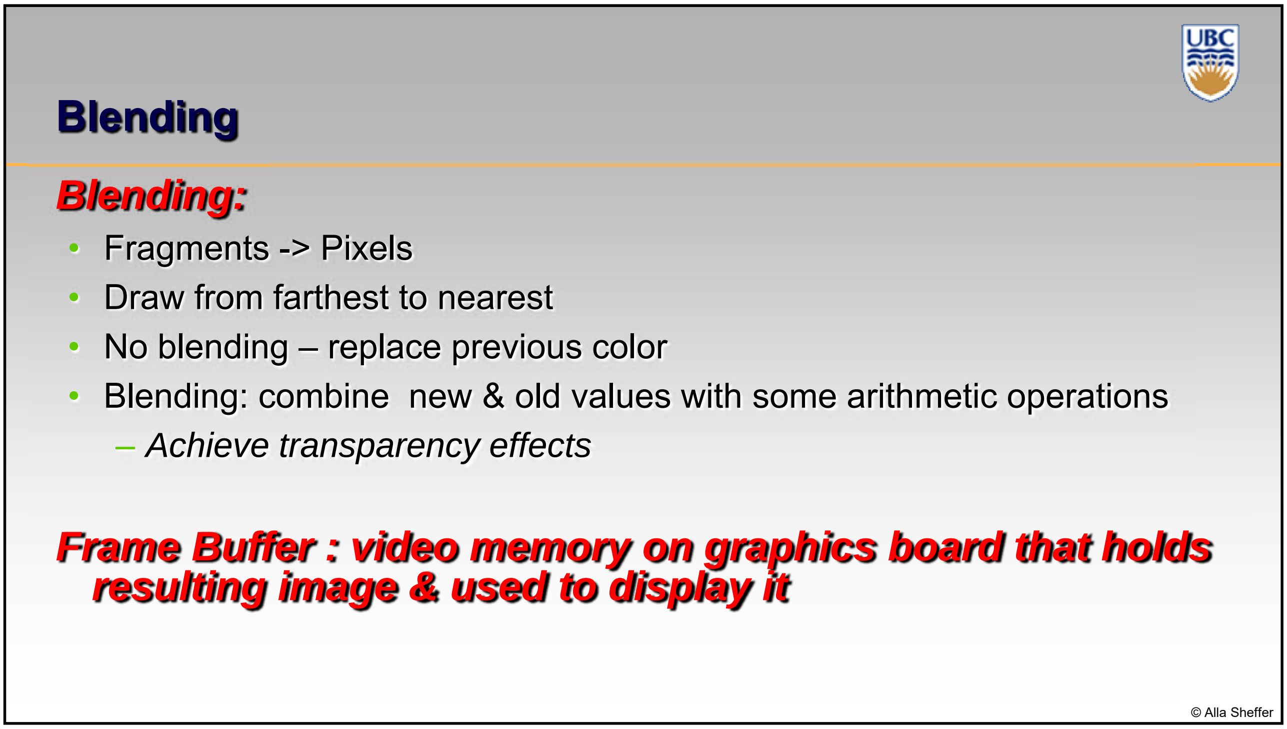
Depth Test /Hidden Surface Removal

Remove ocluded geometry

- Parts that are hidden behind other geometry
- For 2D (view parallel) shapes – use depth order



PIPELINE: More details



Blending

Blending:

- Fragments -> Pixels
- Draw from farthest to nearest
- No blending – replace previous color
- Blending: combine new & old values with some arithmetic operations
  - *Achieve transparency effects*

Frame Buffer : video memory on graphics board that holds resulting image & used to display it