Computer Graphics
Rasterization

Matthias Teschner

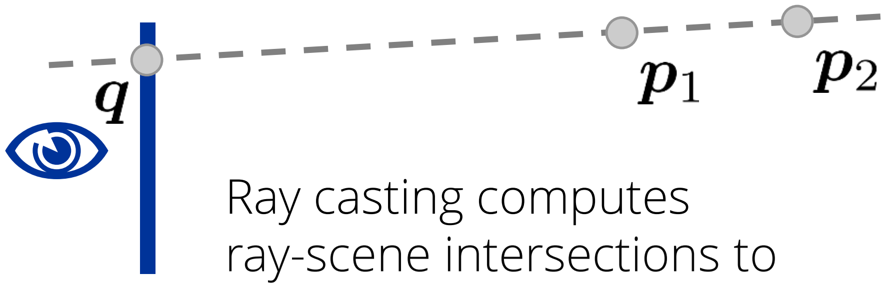
UNI
FREIBURG

Outline

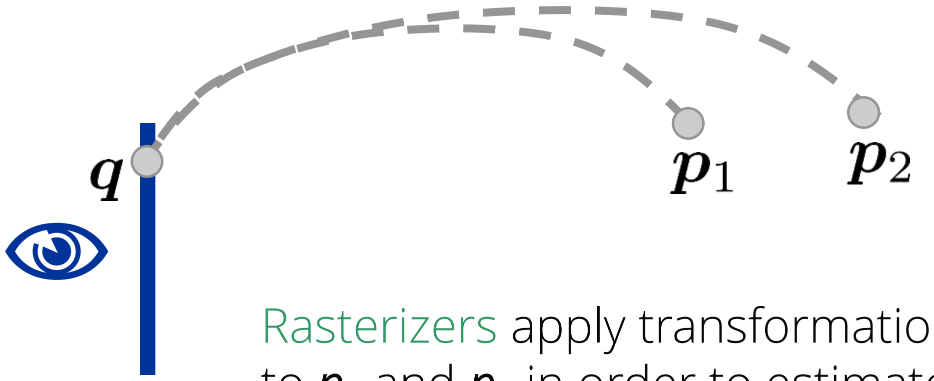
- Context
- Rasterization-based rendering
- Vertex processing
- Rasterization
- Fragment processing
- Framebuffer update

What is visible at the sensor?

– Visibility can be resolved by ray casting or by rasterization



Ray casting computes ray-scene intersections to estimate q from p_1 and p_2.

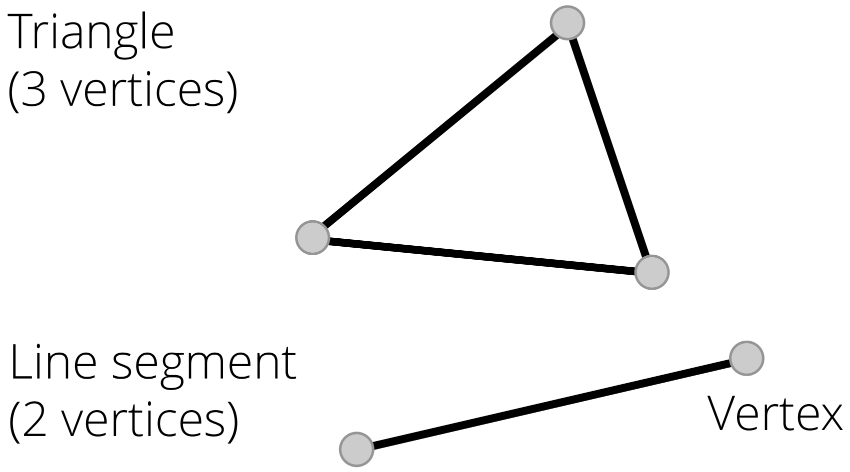


Rasterizers apply transformations to p_1 and p_2 in order to estimate q.

– If more than one scene point p_i is mapped to the same sensor position q, the scene point closest to the viewer is selected

Rasterization

– Computation of pixel positions in an image plane that represent a projected primitive



Primitives represented by vertices

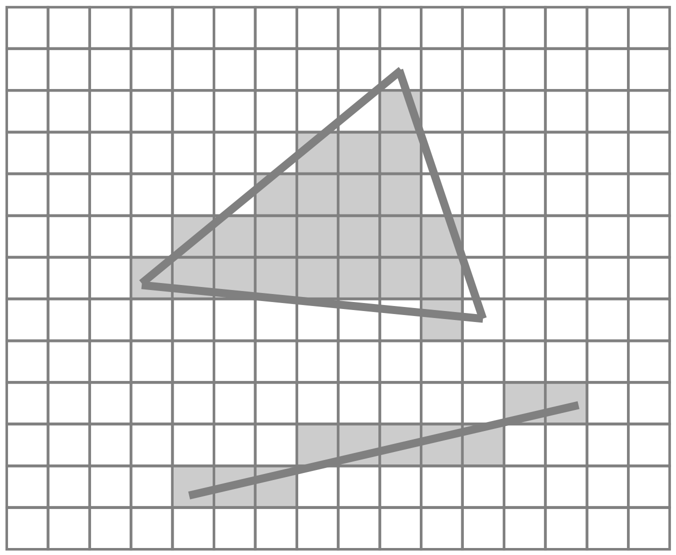
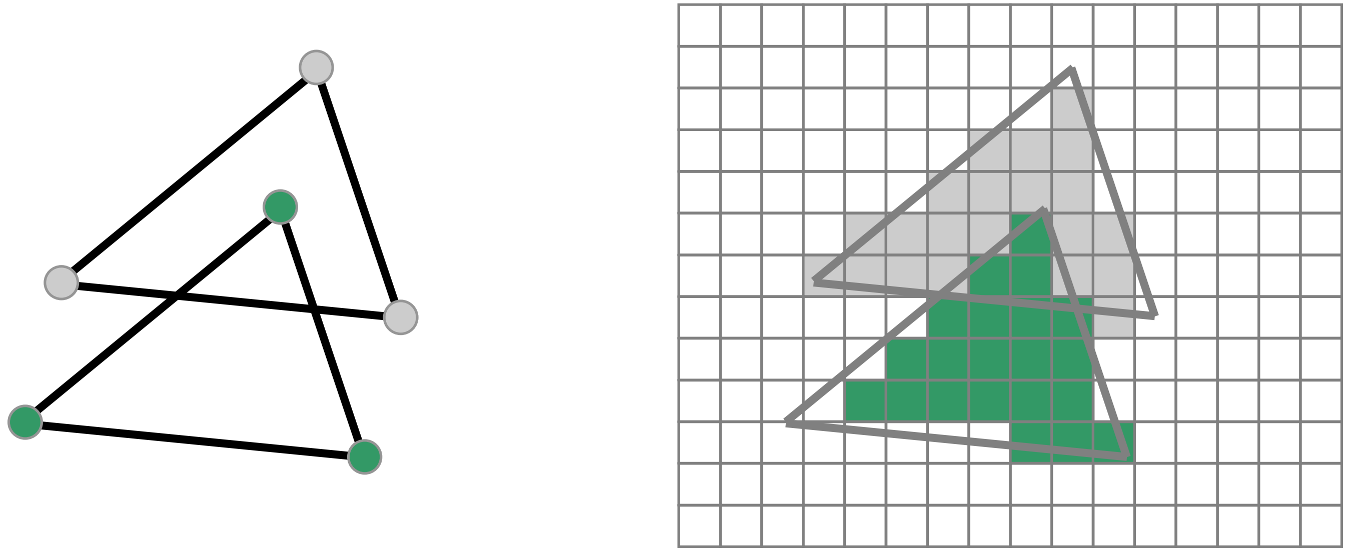


Image plane / 2D array of pixels

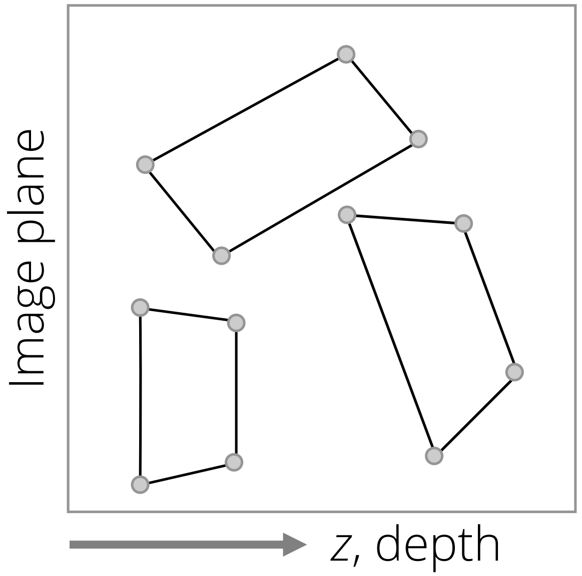
Rasterization and Visibility

– After rasterization, visibility can be efficiently resolved per pixel position
  – Distances of primitives to the viewer, i.e. depth values, can be compared per pixel position

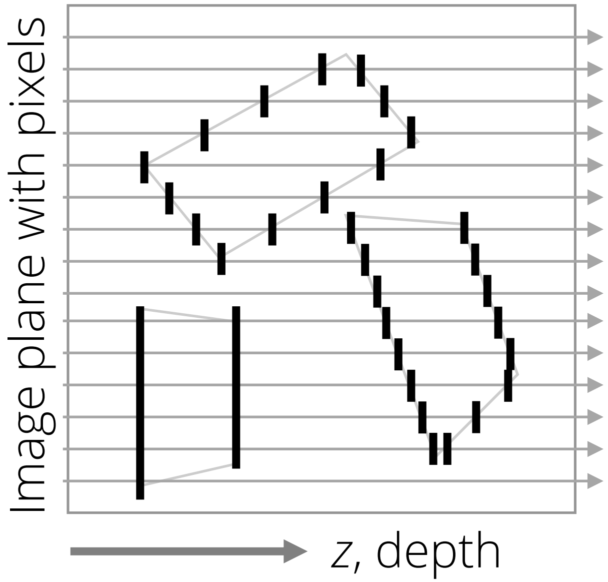


Rasterization and Canonical View Volume

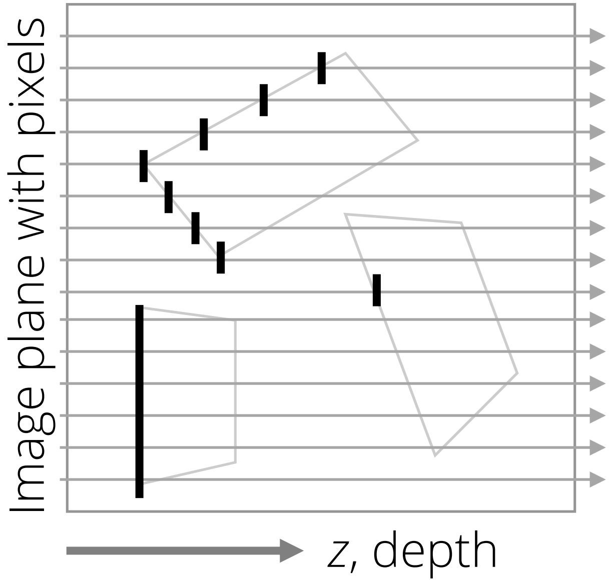
– Rasterization is typically implemented for canonical view volumes



Side view of a scene in a canonical view volume



Rasterization result

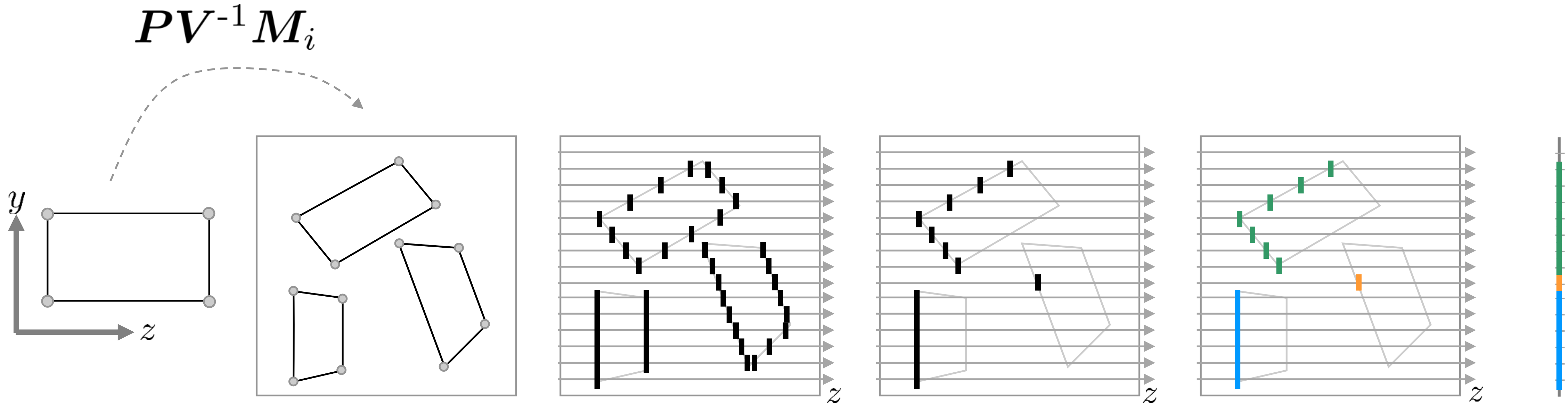


Resolved visibility

Rasterization and Rendering

- Rasterization is typically embedded in a complete rendering approach
  - Rendering pipeline
  - Rasterization-based rendering
  - Rasterization

Rasterization-based Rendering



Object in local space

Scene in the canonical view volume

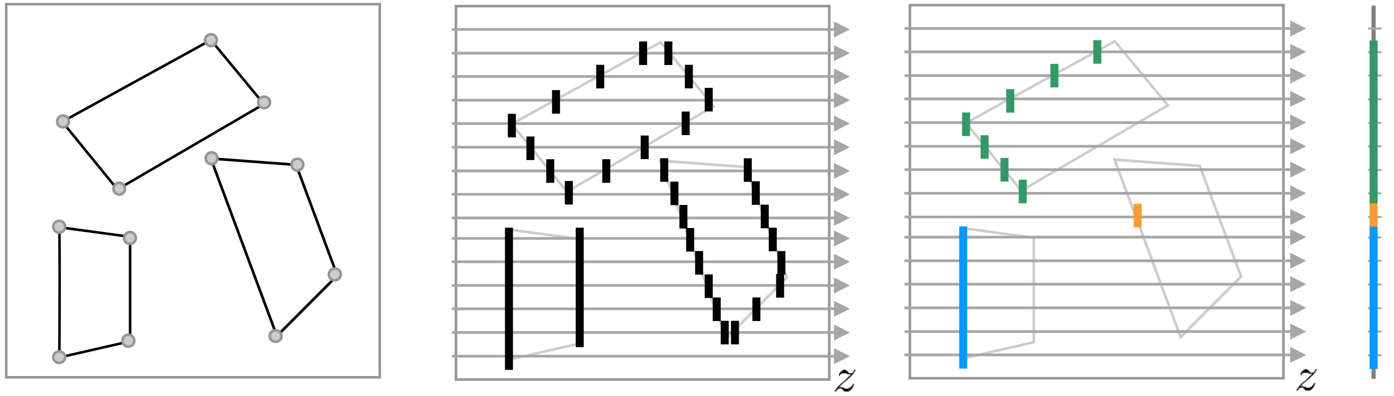
Rasterization

Visibility

Shading

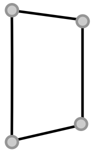
Image

Terms – 2D Illustration



Vertex
Primitive

Object with four vertices and four primitives



Fragment

Pixel

Vertices: have positions and other attributes.

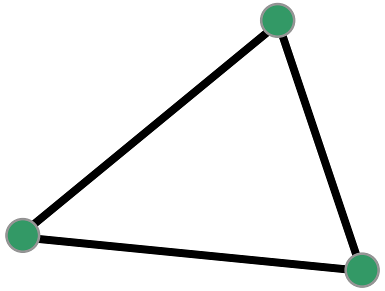
Primitives: are represented by vertices.

Fragments: are pixel candidates with pixel positions and other attributes.

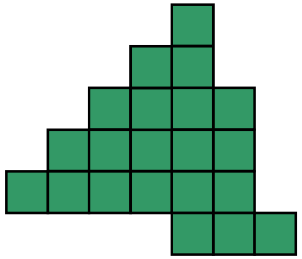
Pixels: have a position and other attributes, in particular color.

Framebuffer: consists of pixels.

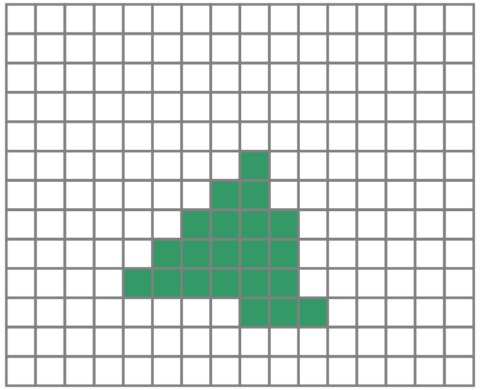
Terms - Illustration



Triangle 1 with three vertices

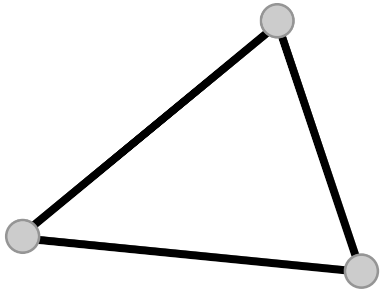


Rasterizer generates fragments.

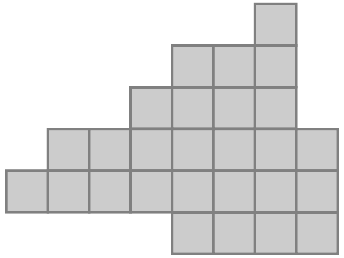


Pixels of the framebuffer

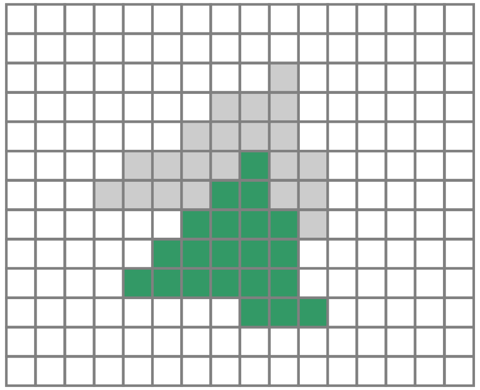
Fragment attributes are used to update pixel attributes in the framebuffer.



Triangle 2 with three vertices



Rasterizer generates fragments.



Pixels of the framebuffer

Framebuffer attri-butes can be upda-ted. Fragments can be discarded.

Outline

- Context
- Rasterization-based rendering
- Vertex processing
- Rasterization
- Fragment processing
- Framebuffer update

Main Stages

- Vertex processing
  - Input: Vertices
  - Output: Vertices
  - Transformations
  - Setting, computation, processing of vertex attributes, e.g. position, color (Phong), texture coordinates

Main Stages

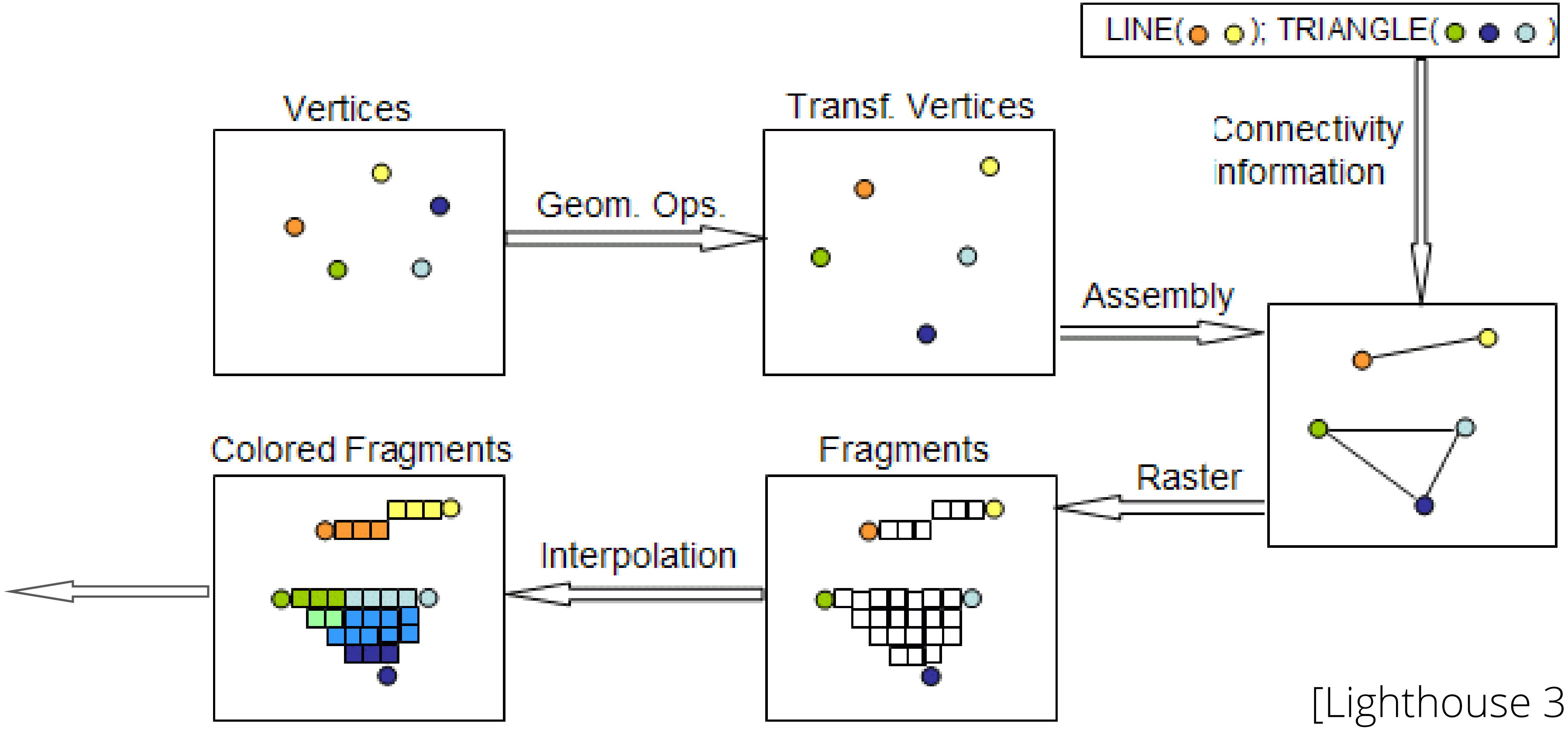
- Rasterization
  - Input: Vertices and connectivity information
  - Output: Fragments
  - Primitive assembly
  - Rasterization of primitives
    - Generates fragments from vertices and connectivity information
  - Sets or interpolates fragment attributes from vertex attributes, e.g. distance to viewer (depth), color, texture coordinates

Main Stages

- Fragment processing
  - Input: Fragments
  - Output: Fragments
  - Fragment attributes are processed, e.g. color
  - Fragments can be discarded
- Framebuffer update
  - Input: Fragments
  - Output: Framebuffer attributes
  - Fragment attributes update framebuffer attributes, e.g. color

Main Stages - Overview

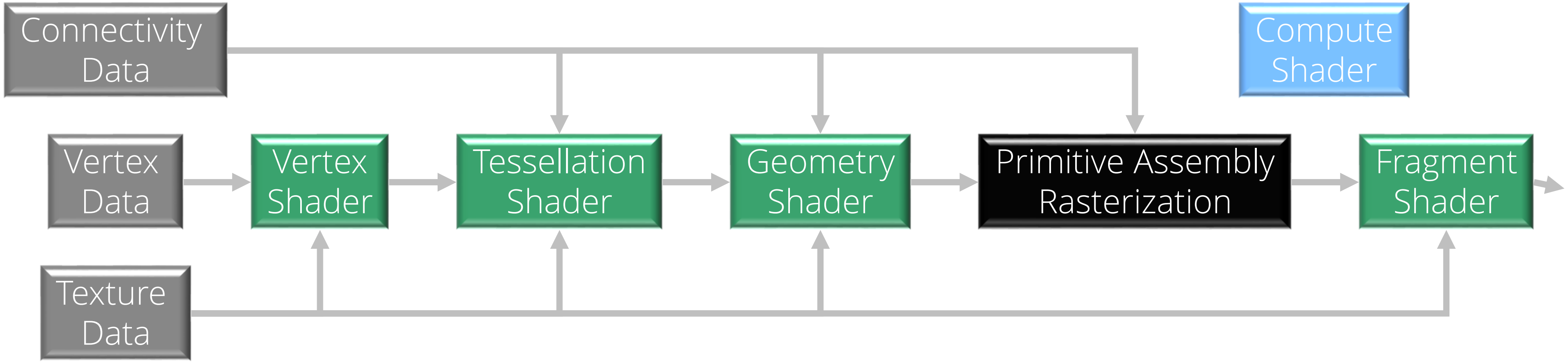
Framebuffer Update



[Lighthouse 3D]

Discussion

- Realization motivated by computational efficiency
  - Vertices and fragments are processed independently in the respective stages
- Stages are supported by graphics hardware GPU
  - OpenGL, DirectX, Vulkan are software interfaces to GPUs



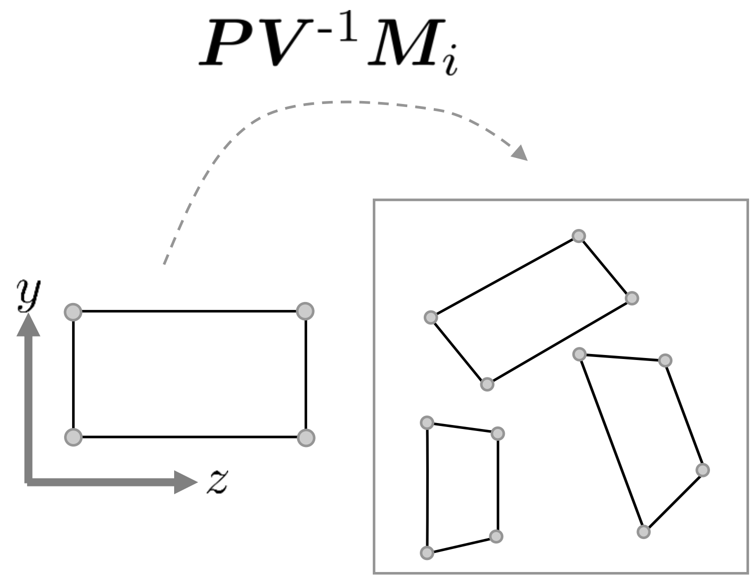
Outline

- Context
- Rasterization-based rendering
- Vertex processing
- Rasterization
- Fragment processing
- Framebuffer update

Transformations of Vertex Positions

- Scene modeling
  - Object placement $\boldsymbol{M}_i$
  - Camera placement $\boldsymbol{V}$
  - Internal camera parameters, i.e. focal length $\boldsymbol{P}$
- Vertices $\boldsymbol{p}$ of object *i* are transformed with

$$\boldsymbol{p}' = \boldsymbol{P}\boldsymbol{V}^{-1}\boldsymbol{M}_i\boldsymbol{p}$$



Object in local space

Scene in the canonical view volume

Transformations of Vertex Positions

- GPU rasterizers assume that all vertex positions are in clip / NDC space.
- Only vertices inside the canonical view volume, e.g. $(-1\ldots1, -1\ldots1, -1\ldots1)$, are processed
- Transformation $\boldsymbol{p}' = \boldsymbol{P}\boldsymbol{V}^{-1}\boldsymbol{M}_i\boldsymbol{p}$ can realize user-defined scene setups
- Alternatively, the scene can be setup within the canonical view volume and rendered with parallel projection. Then, transformations are not required.

Vertex Attributes

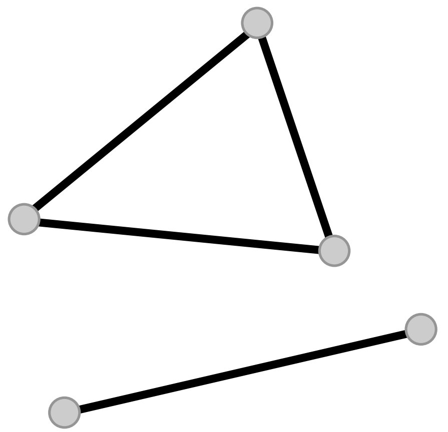
- Position $(p_x, p_y, p_z, 1)^\mathsf{T}$
  - *Z*-component in NDC space is referred to as depth value. Represents distance to the camera plane.
- Color $(R, G, B, A)^\mathsf{T}$
  - Can optionally be defined or computed with Phong, if surface normal, light and material properties are available
  - A can be used for rendering effects, e.g. transparency
- Texture coordinates, e.g. (u, v)
  - For lookup and processing of additional data, i.e. textures

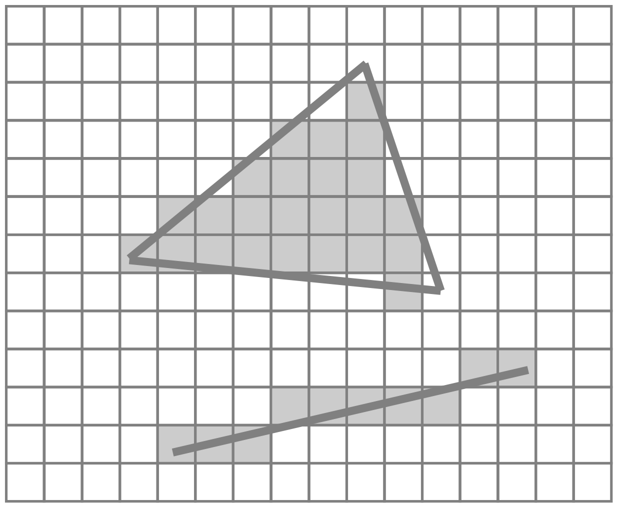
Outline

- Context
- Rasterization-based rendering
- Vertex processing
- Rasterization
- Fragment processing
- Framebuffer update

Rasterization

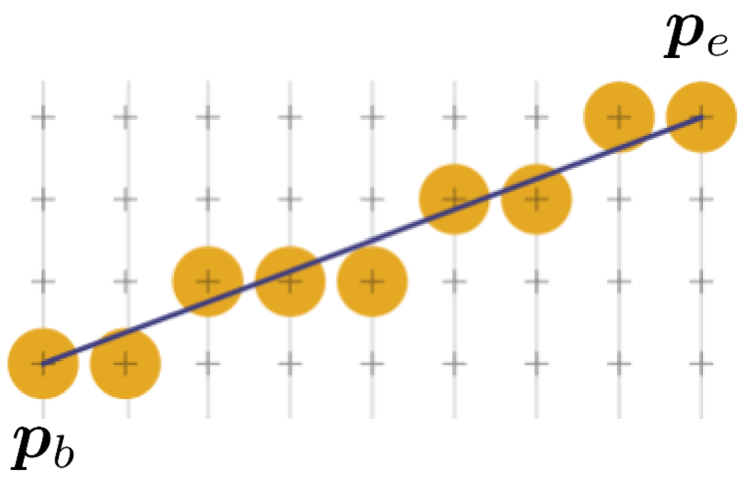
- Input
  - Vertices with connectivity information and attributes
    - Color, depth, texture coordinates
- Output
  - Fragments with attributes
    - Pixel position
    - Interpolated color, depth, texture coordinates





Line Rasterization

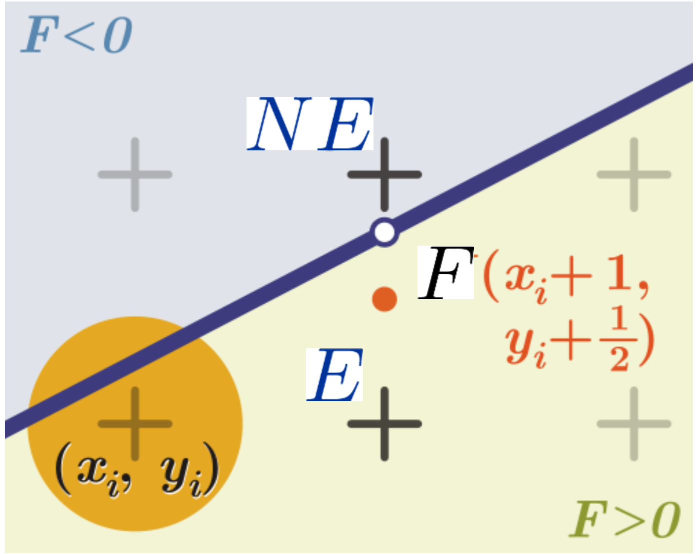
- Line begins and ends at integer-valued positions $\boldsymbol{p}_b = (x_b, y_b)$ and $\boldsymbol{p}_e = (x_e, y_e)$
- Algorithm defined for line slopes between 0 and 1
  - Generalized by employing symmetries
- One fragment per integer *x*-value
  - First fragment: (x_b, y_b)
  - Next fragment: $(x_b + 1, y_b)$ or $(x_b + 1, y_b + 1)$
  - Last fragment: (x_e, y_e)



[Wikipedia: Rasterung von Linien]

Bresenham Line Algorithm

- Based on the current fragment (x_i, y_i), the algorithm decides whether to choose $(x_i + 1, y_i)$ or $(x_i + 1, y_i + 1)$
- Line representation: $F(x, y) = ax + by + c = 0$
- F is evaluated at the midpoint between $(x_i + 1, y_i)$ and $(x_i + 1, y_i + 1)$
- $F(x_i + 1, y_i + \frac{1}{2}) > 0$
  choose *NE*, i.e. $(x_i + 1, y_i + 1)$
- $F(x_i + 1, y_i + \frac{1}{2}) \leq 0$
  choose *E*, i.e. $(x_i + 1, y_i)$



[Wikipedia: Rasterung von Linien]

Incremental Update of the Decision Variable

- Decision variable $d_i = F(x_i + 1, y_i + \frac{1}{2})$
- Incremental update from d_i to d_{i+1}
  - $d_i > 0 \Rightarrow$ choose *NE*, $d_{i+1} = F(x_i + 2, y_i + 1 + \frac{1}{2})$
  - $d_i \leq 0 \Rightarrow$ choose *E*, $d_{i+1} = F(x_i + 2, y_i + \frac{1}{2})$

$$F(x, y) = ax + by + c$$
$$a = \Delta y = y_e - y_b$$
$$b = -\Delta x = x_b - x_e$$

- In case of $d_i > 0$:

$$\Delta_{NE} = d_{i+1} - d_i = \Delta y \cdot (x_i + 2) - \Delta x \cdot (y_i + \tfrac{3}{2}) + c - (\Delta y \cdot (x_i + 1) - \Delta x \cdot (y_i + \tfrac{1}{2}) + c)$$
$$\Delta_{NE} = \Delta y - \Delta x$$

- In case of $d_i \leq 0$:

$$\Delta_E = d_{i+1} - d_i = \Delta y \cdot (x_i + 2) - \Delta x \cdot (y_i + \tfrac{1}{2}) + c - (\Delta y \cdot (x_i + 1) - \Delta x \cdot (y_i + \tfrac{1}{2}) + c)$$
$$\Delta_E = \Delta y$$

Bresenham Algorithm - Initialization

– For start fragment $\boldsymbol{p}_b = (x_b, y_b)$,
  the decision variable can be initialized as

$$\begin{aligned} d_1 &= F(x_b + 1, y_b + \tfrac{1}{2}) = \Delta y \cdot (x_b + 1) - \Delta x \cdot (y_b + \tfrac{1}{2}) + c \\ &= \Delta y \cdot x_b - \Delta x \cdot y_b + c + \Delta y - \tfrac{1}{2}\Delta x \\ &= F(x_b, y_b) + \Delta y - \tfrac{1}{2}\Delta x \\ &= \Delta y - \tfrac{1}{2}\Delta x \end{aligned}$$

– Floating-point arithmetic is avoided by
  considering $2 \cdot F(x, y)$:

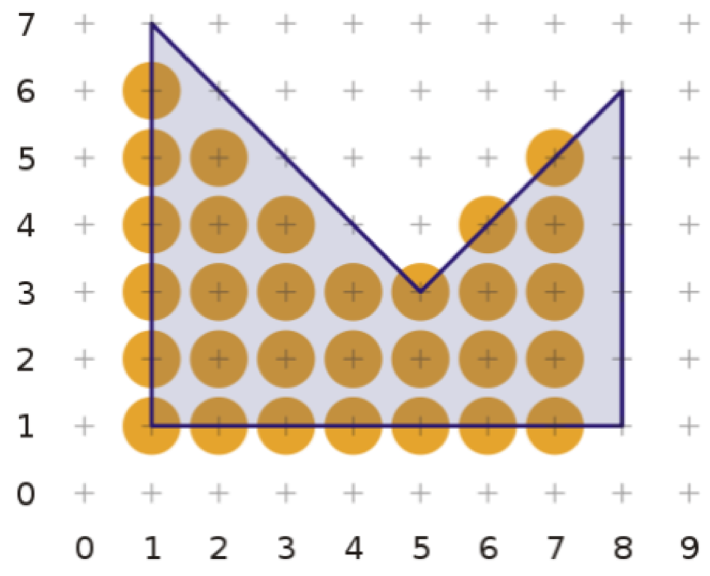
$$\begin{aligned} d_1 &= 2\Delta y - \Delta x \\ \Delta_E &= 2\Delta y \\ \Delta_{NE} &= 2\Delta y - 2\Delta x \end{aligned}$$

Bresenham Algorithm - Implementation

```
void BresenhamLine(int xb, int yb, int xe, int ye) {

    int dx, dy, incE, incNE, d, x, y;

    dx = xe - xb; dy = ye - yb;
    d = 2*dy - dx; incE = 2*dy; incNE = 2*(dy - dx);
    x = xb; y = yb;

    GenerateFragment(x, y);

    while (x < xe) {
        x++;
        if (d <= 0)    d += incE;          /* choose E */
        else {d += incNE; y++; }           /* choose NE */
        GenerateFragment(x, y);
    }
}
```

Polygon Rasterization

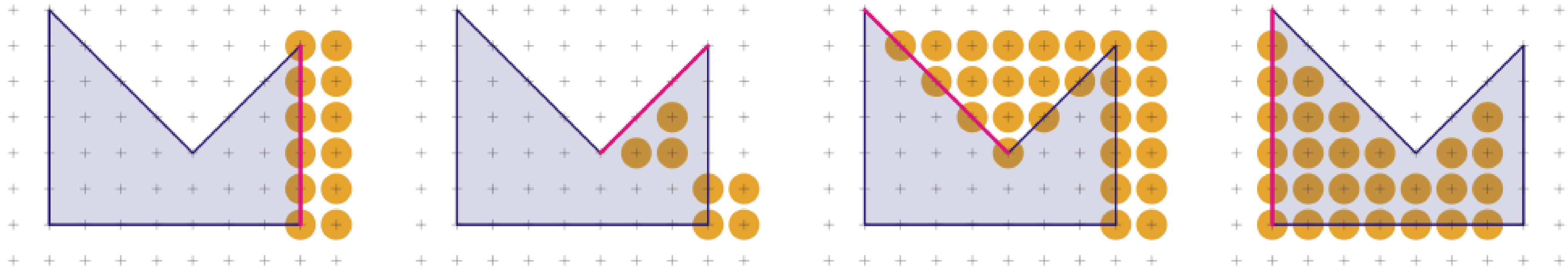
- Compute intersections of non-horizontal polygon edges with horizontal scanlines
- Intersections are computed for scanlines $y = y_i + 0.5$
- Fill pixel positions in-between two intersections with fragments
  - Scan from left to right
  - Enter the polygon at the first intersection, leave the polygon at the next intersection
- Works for closed polygons



[Wikipedia: Rasterung von Polygonen]

Polygon Rasterization

- For each polygon edge
  - Process all scanlines intersected by the edge
  - Invert all positions with an *x*-component larger than the intersection point

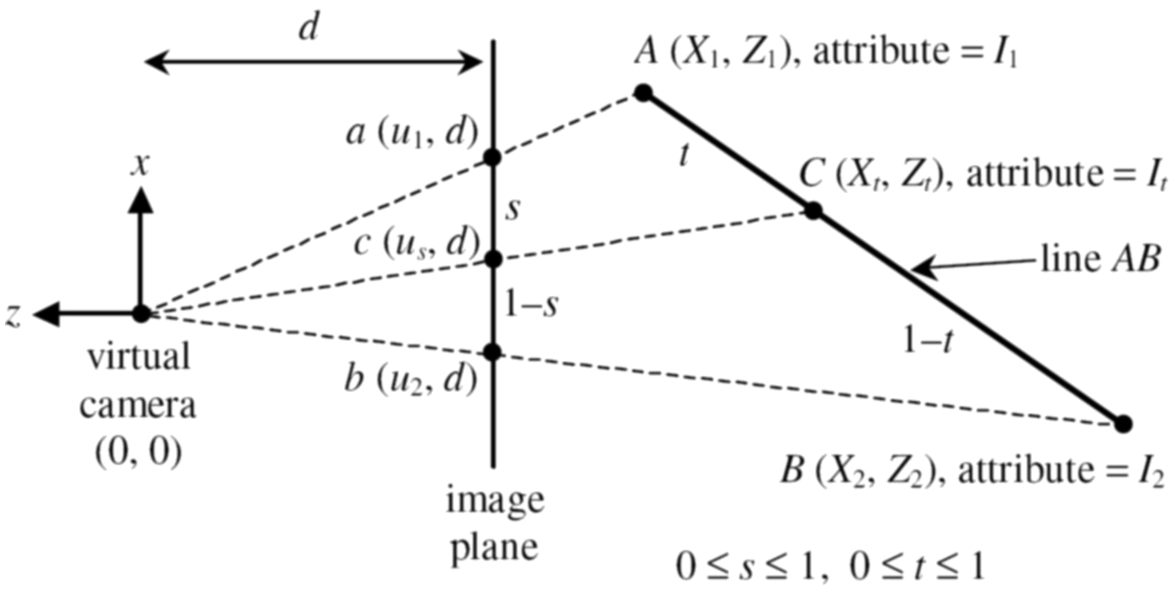


[Wikipedia: Rasterung von Polygonen]

Attribute Interpolation

- Attributes are interpolated from vertices to fragments
- Challenge in case of perspective projection: Linear interpolation in view space cannot be realized by linear interpolation in clip space

Attribute Interpolation



Perspective projection of a line *AB*. *t* / (1-*t*) is not equal to *s* / (1-*s*). Therefore, linear interpolation in clip space between a and b does not correspond to a linear interpolation between *A* and *B* in view space.

$$I_t = I_1 + t(I_2 - I_1)$$

Linear interpolation in view space

$$I_s = I_1 + \frac{sZ_1}{sZ_1 + (1-s)Z_2}(I_2 - I_1)$$

Non-linear interpolation in clip space

$$I_s = \frac{\frac{I_1}{Z_1} + s\left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1}\right)}{\frac{1}{Z_1} + s\left(\frac{1}{Z_2} - \frac{1}{Z_1}\right)}$$

Linear interpolation of *I / Z* and *1 / Z* in clip space

[Kok-Lim Low: Perspective-Correct Interpolation]

Attribute Interpolation

– Perspective projection transform

$$\begin{pmatrix} x_{\text{clip}} \\ y_{\text{clip}} \\ z_{\text{clip}} \\ w_{\text{clip}} \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_{\text{view}} \\ Y_{\text{view}} \\ Z_{\text{view}} \\ 1 \end{pmatrix}$$

– Linear relation between w_{clip} in clip space and z_{view} in view space $w_{\text{clip}} = Z_{\text{view}}$

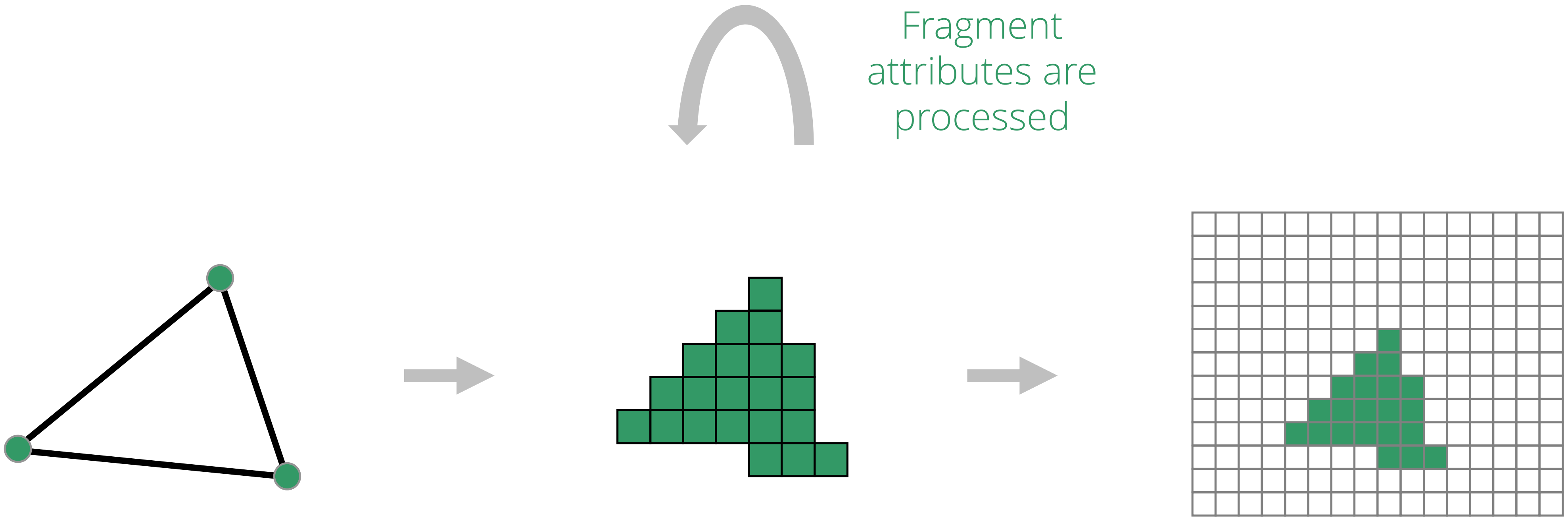
– Z_{view} or w_{clip} can be used in the interpolation

In view space: $I_s = \dfrac{\frac{I_1}{Z_1} + s\left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1}\right)}{\frac{1}{Z_1} + s\left(\frac{1}{Z_2} - \frac{1}{Z_1}\right)}$ In clip space: $I_s = \dfrac{\frac{I_1}{w_1} + s\left(\frac{I_2}{w_2} - \frac{I_1}{w_1}\right)}{\frac{1}{w_1} + s\left(\frac{1}{w_2} - \frac{1}{w_1}\right)}$

Outline

- Context
- Rasterization-based rendering
- Vertex processing
- Rasterization
- Fragment processing
- Framebuffer update

Fragment Processing



Vertices with connectivity and attributes

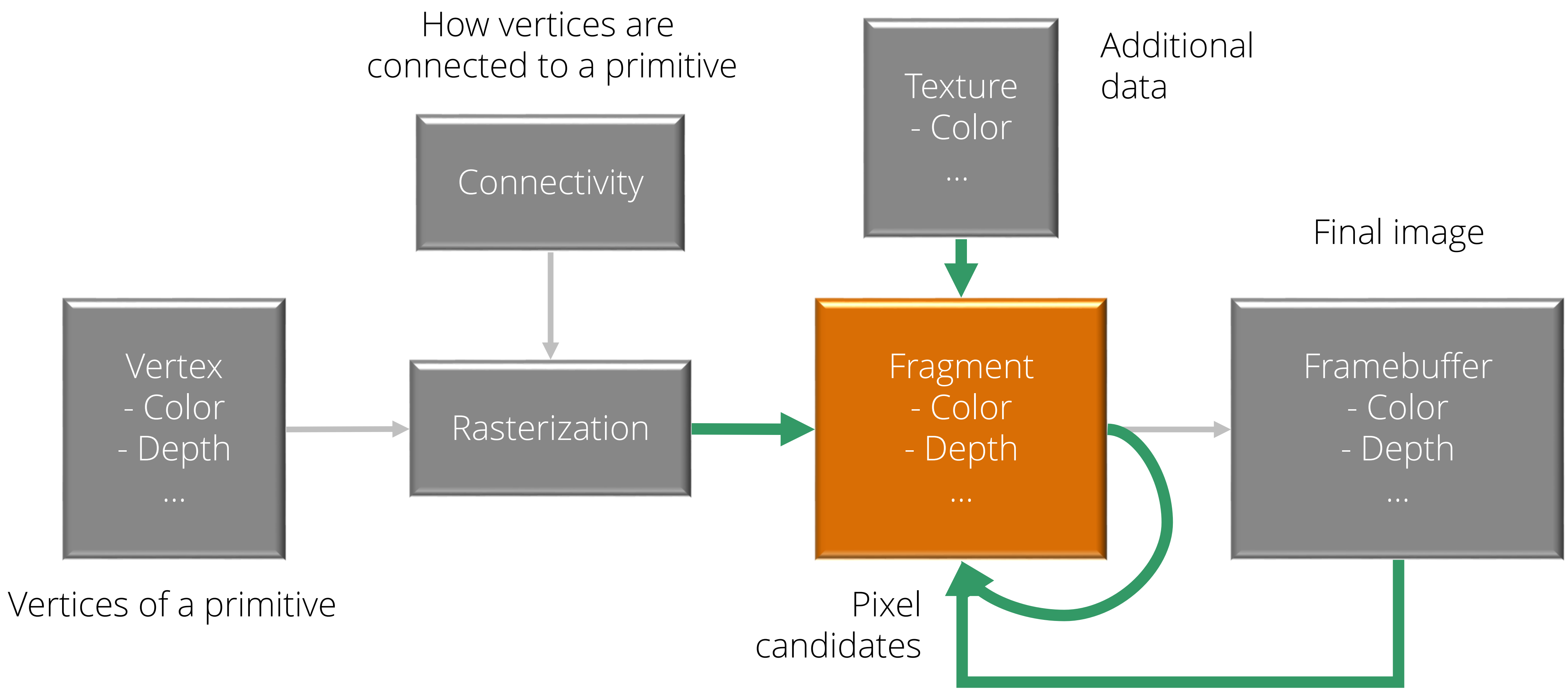
Fragments (pixel candidates) with attributes

Pixels of the framebuffer with attributes, in particular color

Fragment Processing

- Fragment attributes are processed
- Fragment attributes are tested
  - Fragments can be discarded
  - Fragments can pass a test and fragment attributes can be used to update framebuffer attributes
- Processing and testing make use of
  - Fragment attributes (position, color, depth, texture coord)
  - Textures (*n* dimensional arrays of data)
  - Framebuffer data that is available for each pixel position
    - Depth buffer, color buffer, stencil buffer, accumulation buffer

Fragment Processing

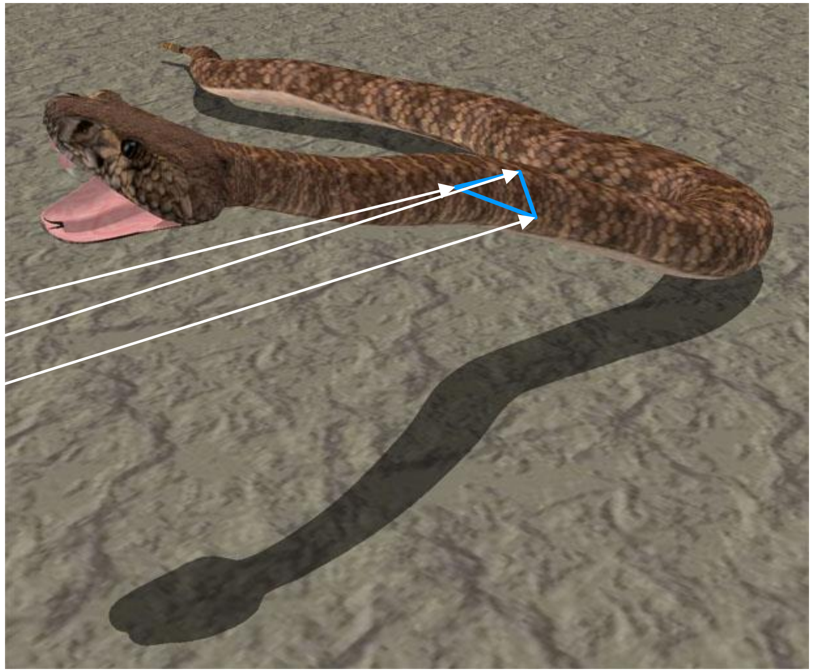


Attribute Processing - Examples

- Texturing
  - Combination of fragment color and texture data
- Fog
  - Adaptation of fragment color using fog color and fragment depth
- Antialiasing
  - Adaptation of fragment alpha value

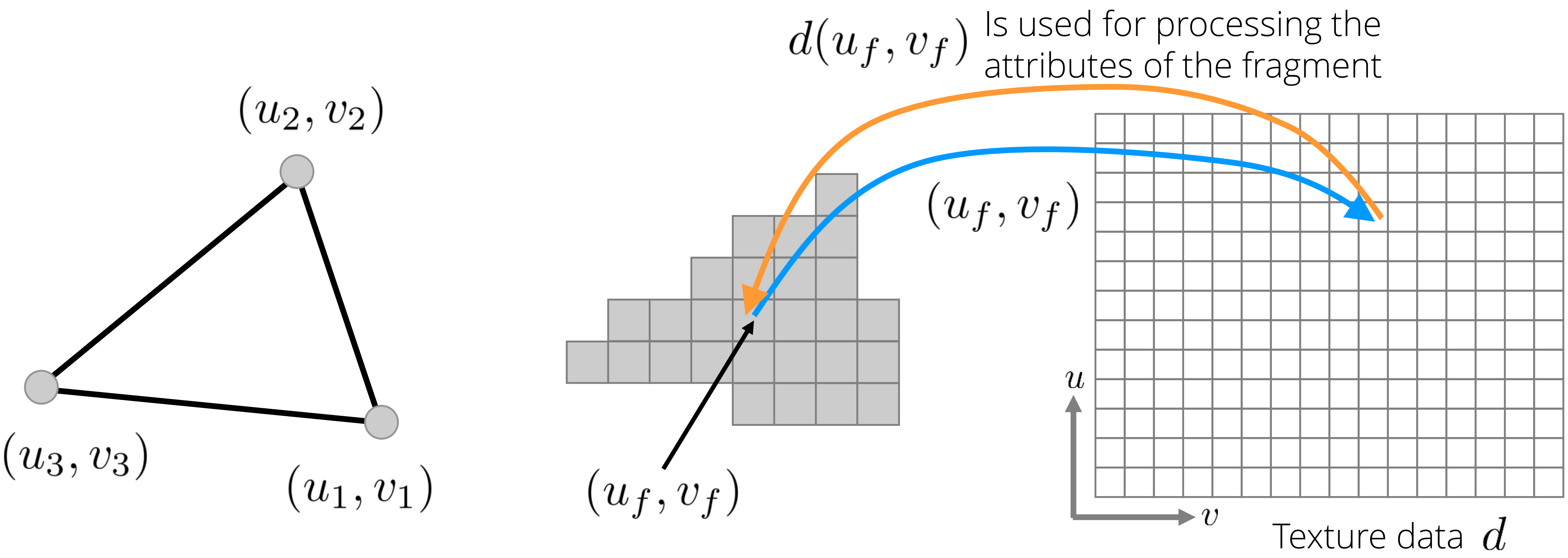
Texturing

Texture



Textured object

Texturing – 2D Example



Texture coordinates are typically defined or computed for vertices

Rasterizer interpolates texture coordinates from vertices to fragments

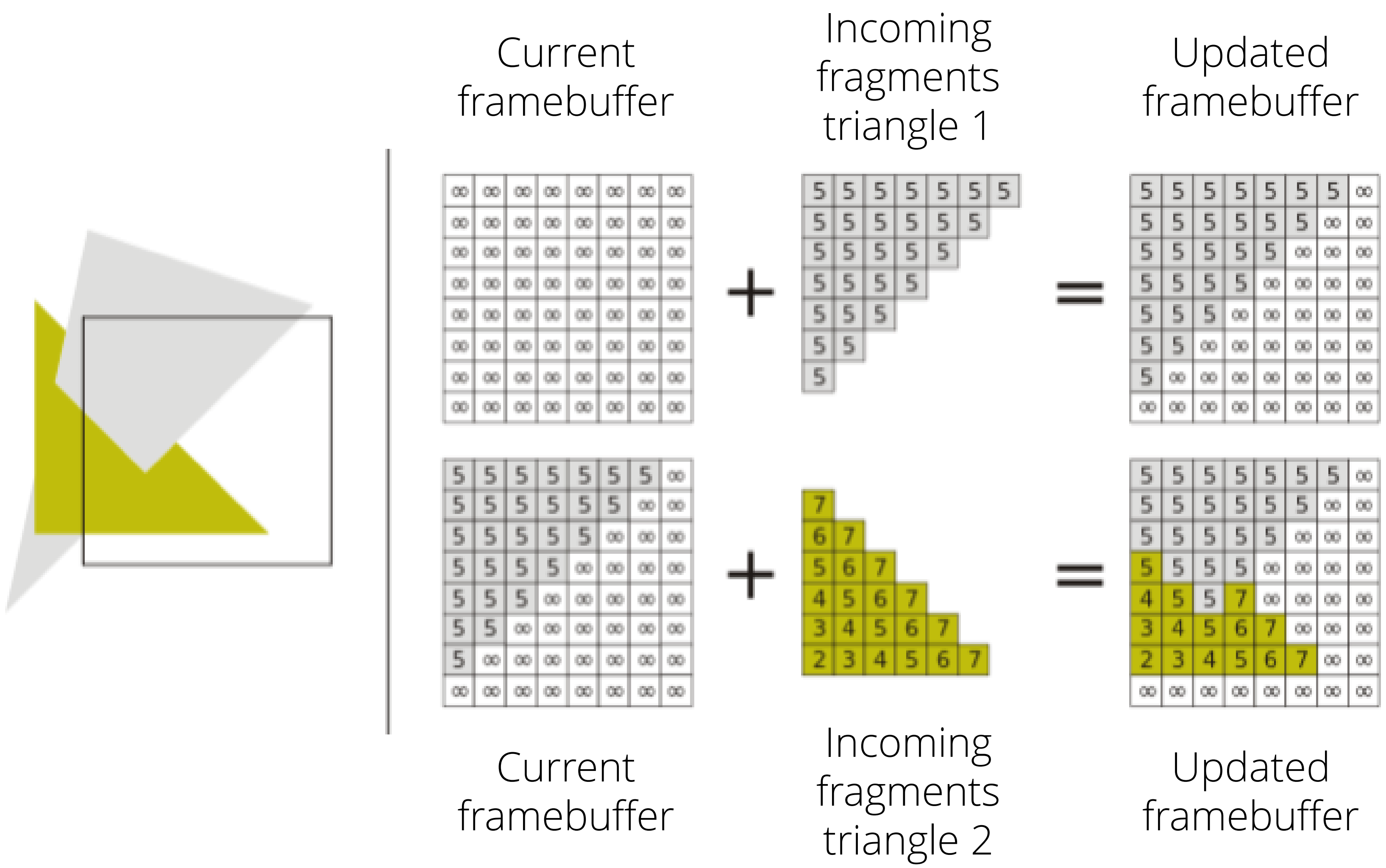
Tests - Examples

- Scissor test
  - Check if fragment position is inside a specified rectangle
- Alpha test
  - Check range of the fragment alpha value
  - Used for, e.g., transparency and billboarding
- Stencil test
  - Check if framebuffer stencil value at the fragment position fulfills a certain requirement
  - Used for, e.g., shadows

Depth Test – Resolving Visibility

- Depth test
  - Compare fragment depth value with the framebuffer depth value at the fragment position
  - If the fragment depth value is larger than the framebuffer depth value, the fragment is discarded
  - If the fragment depth value is smaller than the framebuffer depth value, the fragment passes and its attributes replace the current color and depth values in the framebuffer

Depth Test



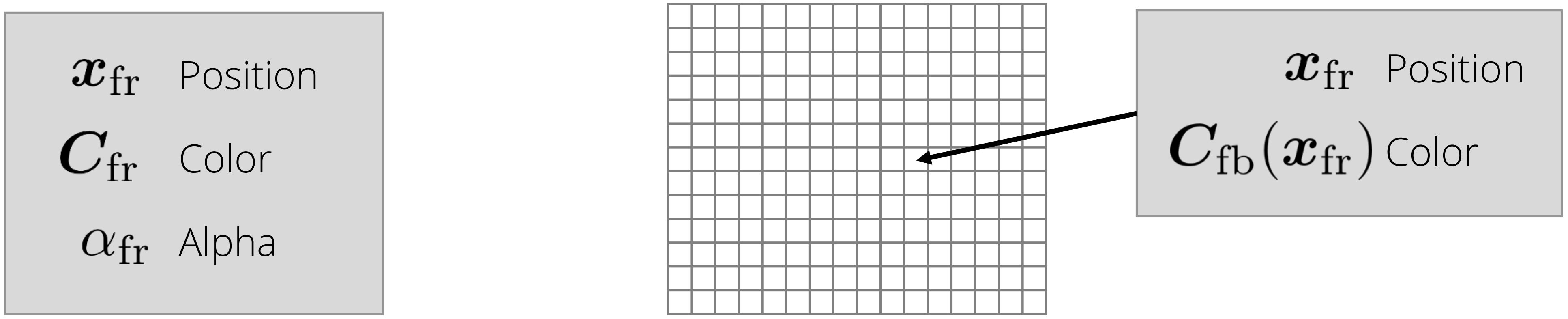
[Wikipedia]

Outline

- Context
- Rasterization-based rendering
- Vertex processing
- Rasterization
- Fragment processing
- Framebuffer update

Blending

- Combines the fragment color $\boldsymbol{C}_{\text{fr}}$ with the framebuffer color $\boldsymbol{C}_{\text{fb}}(\boldsymbol{x}_{\text{fr}})$at the fragment position $\boldsymbol{x}_{\text{fr}}$
- $\boldsymbol{C}_{\text{fb}}(\boldsymbol{x}_{\text{fr}}) = \alpha_{\text{fr}} \cdot \boldsymbol{C}_{\text{fr}} + (1 - \alpha_{\text{fr}}) \cdot \boldsymbol{C}_{\text{fb}}(\boldsymbol{x}_{\text{fr}})$



Fragment Framebuffer

Summary

- Rasterization combined with a depth test can resolve visibility
- Rendering pipeline employs rasterization
  - Vertex processing
  - Rasterization
  - Fragment processing
  - Framebuffer update
- Implemented on graphics hardware