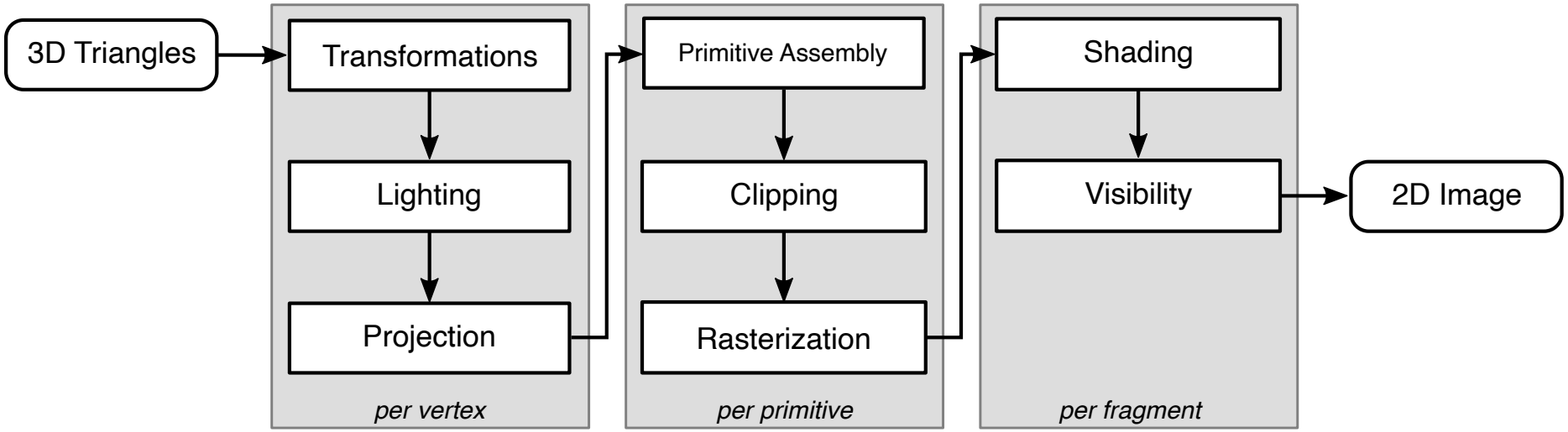
Computer Graphics

Rasterization Pipeline

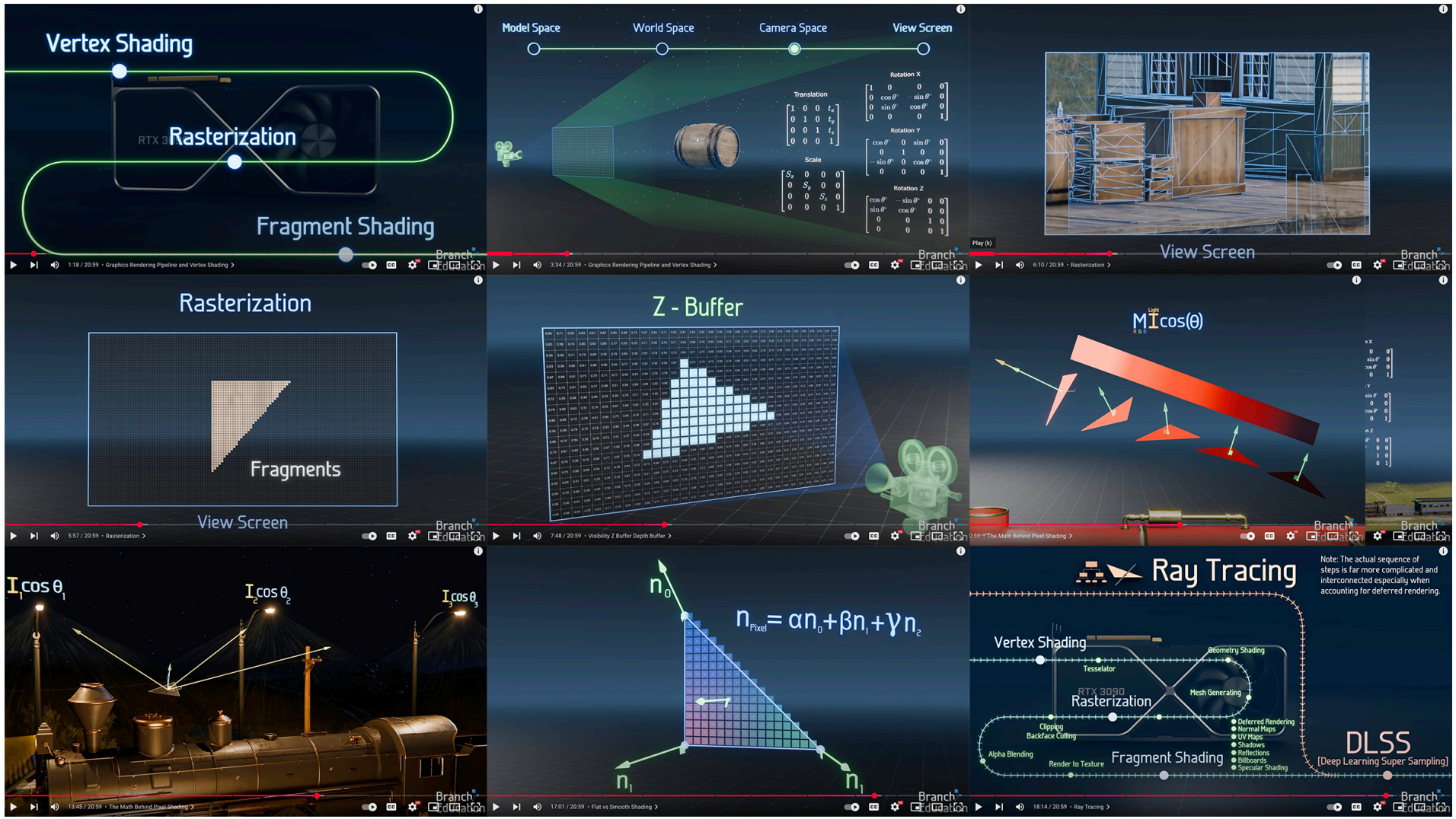
Mark Pauly

Geometric Computing Laboratory

Rasterization Pipeline

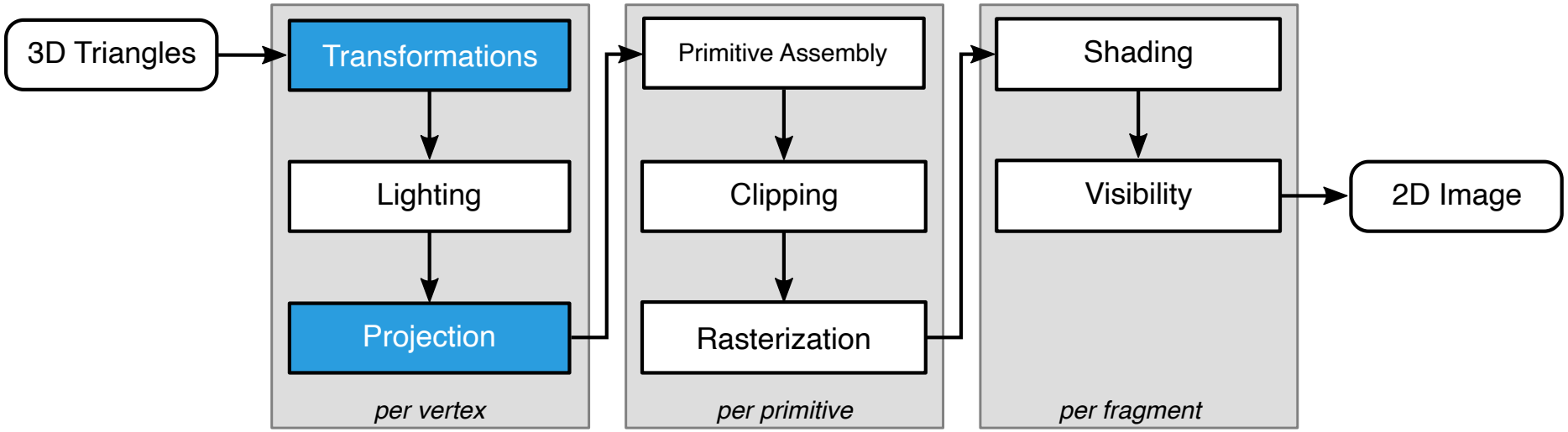


Rasterization Pipeline



Thanks to Amine and Haythem! https://www.youtube.com/watch?v=C8YtdC8mxTU

Transformations & Projections



Transformations & Projections

- Project edge or triangle from 3D to 2D
  - Simply transform/project its endpoints/vertices?
- Let us first analyze this for a line between $\mathbf{A}$ and $\mathbf{B}$

$$\mathbf{X}(\alpha) = (1-\alpha)\mathbf{A} + \alpha\mathbf{B}\,, \quad \alpha \in [0,1]$$

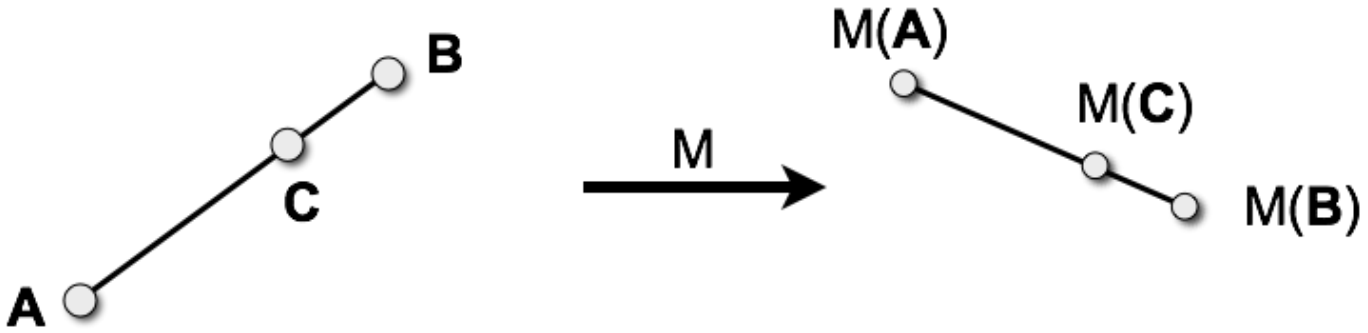
- For an affine/projective transformation $\mathbf{M}$ it has to hold

$$\forall \alpha \in \mathbb{R} \;\; \exists \beta \in \mathbb{R} \;:\; \mathbf{M}(\mathbf{X}(\alpha)) \;=\; (1-\beta)\mathbf{M}(\mathbf{A}) + \beta\mathbf{M}(\mathbf{B})$$

Affine Transformations of Lines

- Affine transformation $\mathbf{M}$ preserves affine combinations

$$\mathbf{M}((1-\alpha)\mathbf{A} + \alpha\mathbf{B}) \;=\; (1-\alpha)\mathbf{M}(\mathbf{A}) + \alpha\mathbf{M}(\mathbf{B})$$



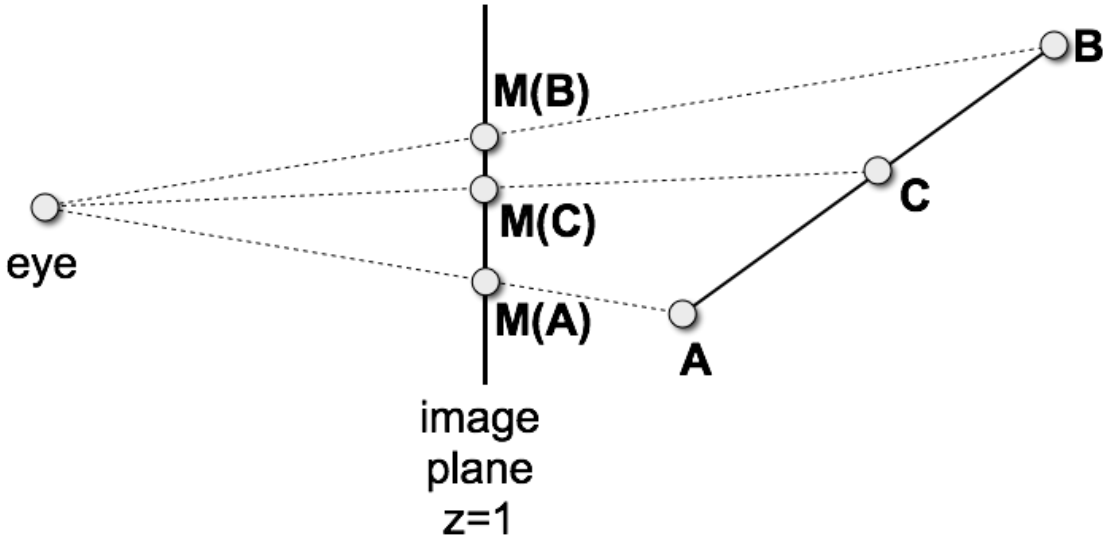
Projective Transformations of Lines

- For a central projection $\mathbf{P}(\mathbf{X}) = \frac{\mathbf{X}}{\mathbf{X}_z}$ we can show that

$$\mathbf{P}(\mathbf{X}(\alpha)) = \frac{(1-\alpha)\mathbf{A} + \alpha\mathbf{B}}{(1-\alpha)\mathbf{A}_z + \alpha\mathbf{B}_z} = (1-\beta)\frac{\mathbf{A}}{\mathbf{A}_z} + \beta\frac{\mathbf{B}}{\mathbf{B}_z}$$

$$\text{if we chose } \beta = \frac{\alpha\mathbf{B}_z}{(1-\alpha)\mathbf{A}_z + \alpha\mathbf{B}_z}$$

- Same argument holds for general projective transformations.



Transformations of Triangles

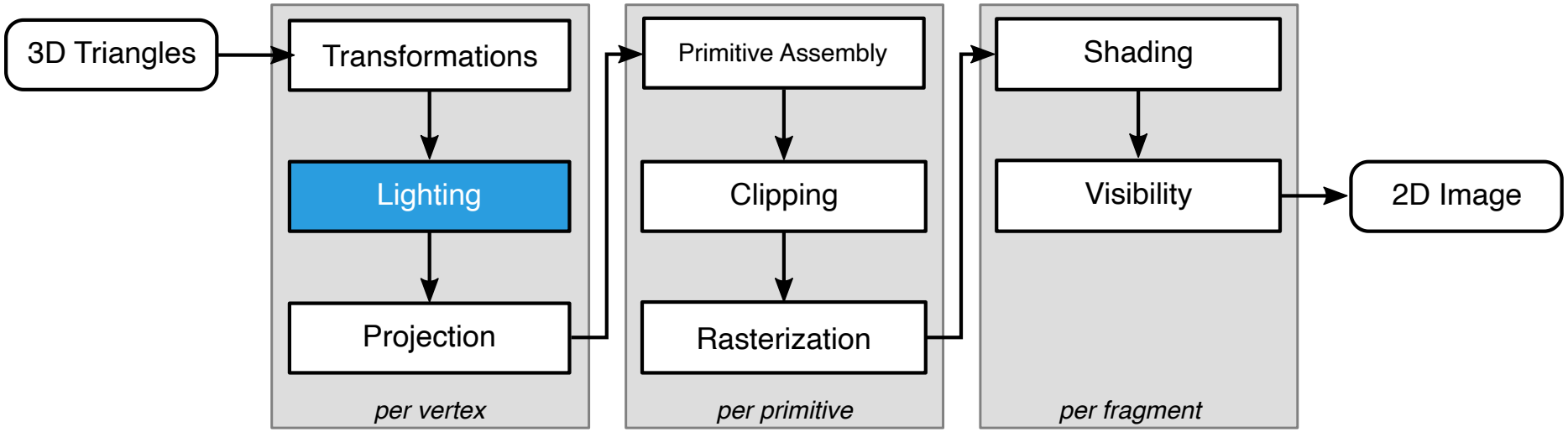
- A triangle is an affine combination of three points

$$\mathbf{X}(\alpha, \beta, \gamma) \;=\; \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C}$$

  with $\alpha + \beta + \gamma = 1$.

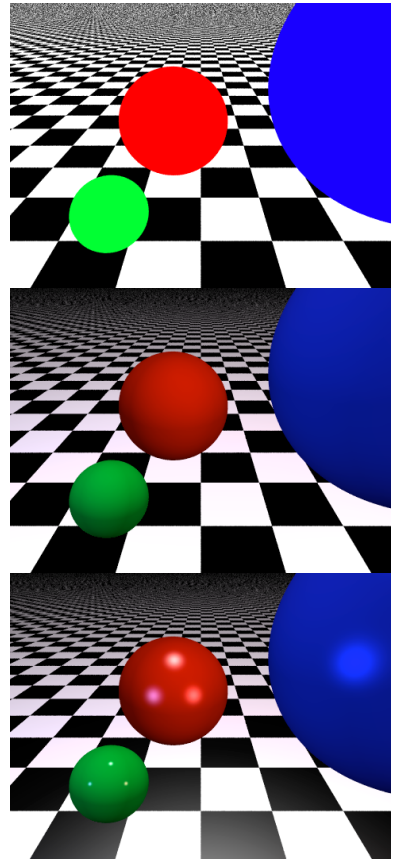
- Affine transformations preserve affine combinations
  - Triangles are transformed to triangles
- Planar projections map triangles to triangles
  - Similar derivation as for lines

Lighting


3D Triangles
Transformations
Lighting
Projection
per vertex
Primitive Assembly
Clipping
Rasterization
per primitive
Shading
Visibility
per fragment
2D Image

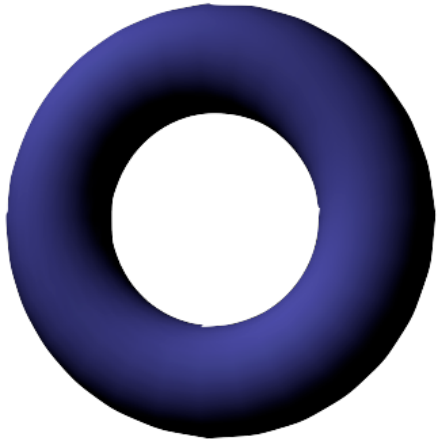
Phong Lighting Model

- **Ambient lighting**
  - approximate global light transport / exchange
  - uniform in, uniform out
- **Diffuse lighting**
  - dull / mat surfaces
  - directed in, uniform out
- **Specular lighting**
  - shiny surfaces
  - directed in, directed out

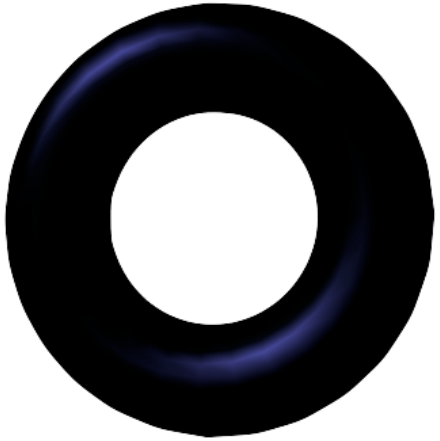


Phong Lighting Model

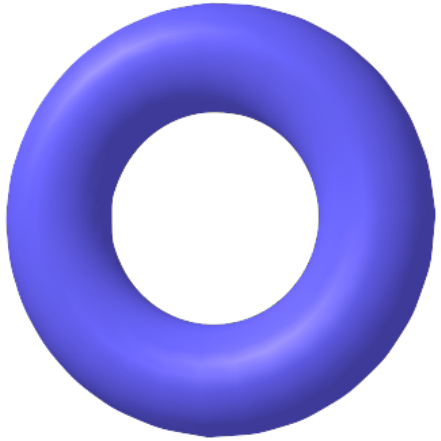
ambient: $I_a m_a$



diffuse: $I_l m_d \, (\mathbf{n} \cdot \mathbf{l})$



specular: $I_l m_s (\mathbf{r} \cdot \mathbf{v})^s$

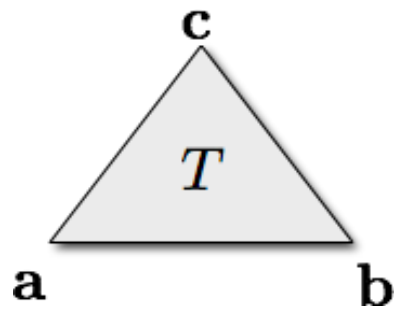


ambient+diffuse+specular

Normal Vectors

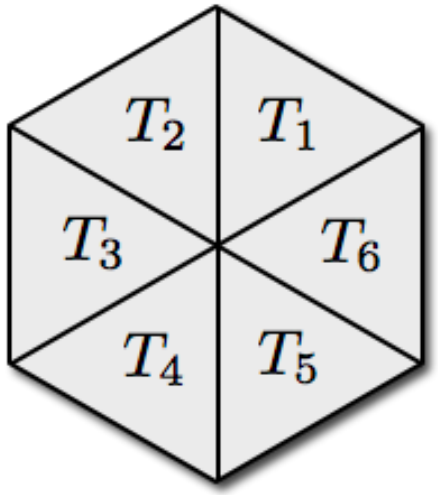
- Triangle normal

$$\mathbf{n}(T) = \frac{(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})}{\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|}$$



- Vertex normal
  - Average of incident triangles' normals $\mathbf{n}(T_i)$
  - Weighted by area or opening angle $w(T_i)$

$$\mathbf{n}(V) = \frac{\sum_{T_i \ni V} w(T_i)\,\mathbf{n}(T_i)}{\left\|\sum_{T_i \ni V} w(T_i)\,\mathbf{n}(T_i)\right\|}$$



How to transform normal vectors?

- Point $\mathbf{x} = (x, y, z, 1)^\mathsf{T}$ with normal $\mathbf{n} = (n_x, n_y, n_z, 0)^\mathsf{T}$ lies on its tangent plane

$$n_x x + n_y y + n_z z + d = 0 \quad \Leftrightarrow \quad \mathbf{p}^\mathsf{T}\mathbf{x} = 0$$

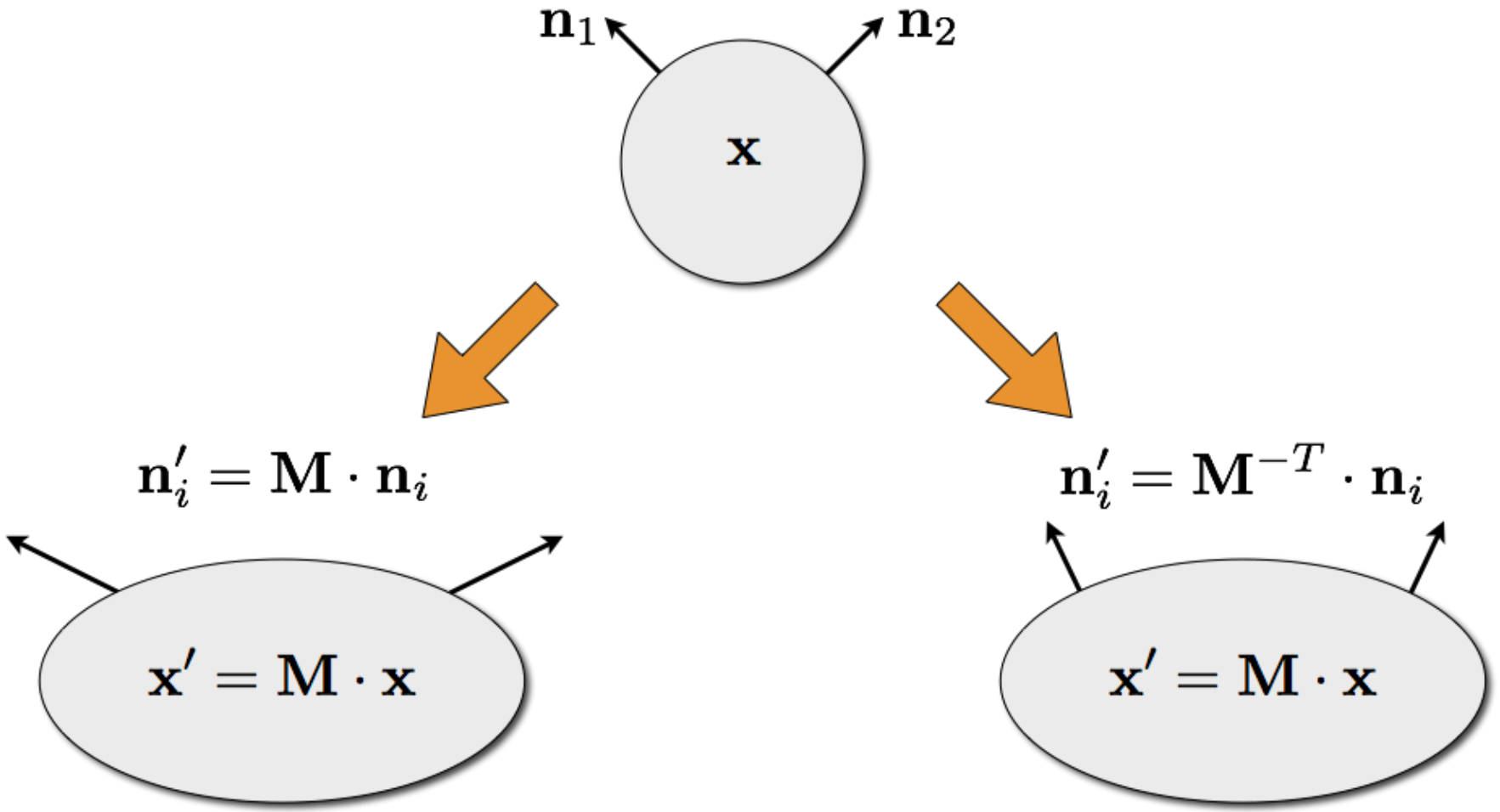
  with $d = \mathbf{n}^\mathsf{T}\mathbf{x}$ and $\mathbf{p} = (n_x, n_y, n_z, d)^\mathsf{T}$.

- The same equation should be satisfied after an affine transformation, represented by 4×4 matrix $\mathbf{M}$. Transformation maps $\mathbf{x}$ to $\mathbf{x}'$, $\mathbf{n}$ to $\mathbf{n}'$, and $\mathbf{p}$ to $\mathbf{p}'$:

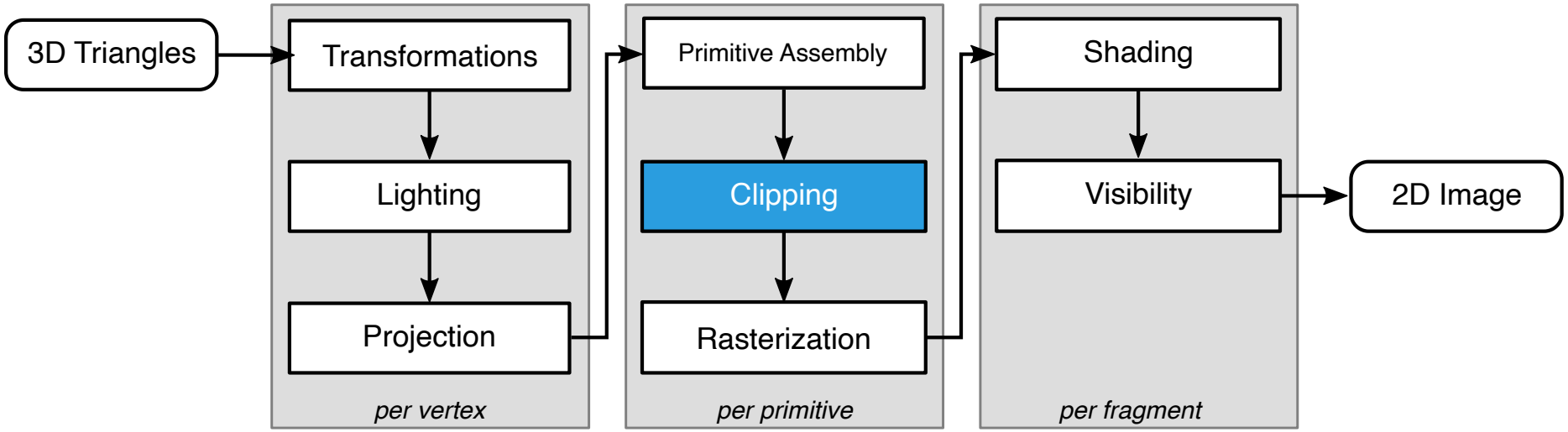
$$0 \;=\; \mathbf{p}'^\mathsf{T}\mathbf{x}' \;=\; \mathbf{p}'^\mathsf{T}\left(\mathbf{M}\mathbf{x}\right) \;=\; \left(\mathbf{M}^\mathsf{T}\mathbf{p}'\right)^\mathsf{T}\mathbf{x}$$

- Comparing the two equations yields $\mathbf{p}' = \mathbf{M}^{-\mathsf{T}}\mathbf{p}$, and therefore $\mathbf{n}' = \mathbf{M}^{-\mathsf{T}}\mathbf{n}$
  - In practice use the upper-left $\mathbf{3} \times \mathbf{3}$ block of $\mathbf{M}$ and build its transposed inverse
  - Then transform the 3D normal vector and re-normalize it

How to transform normal vectors?



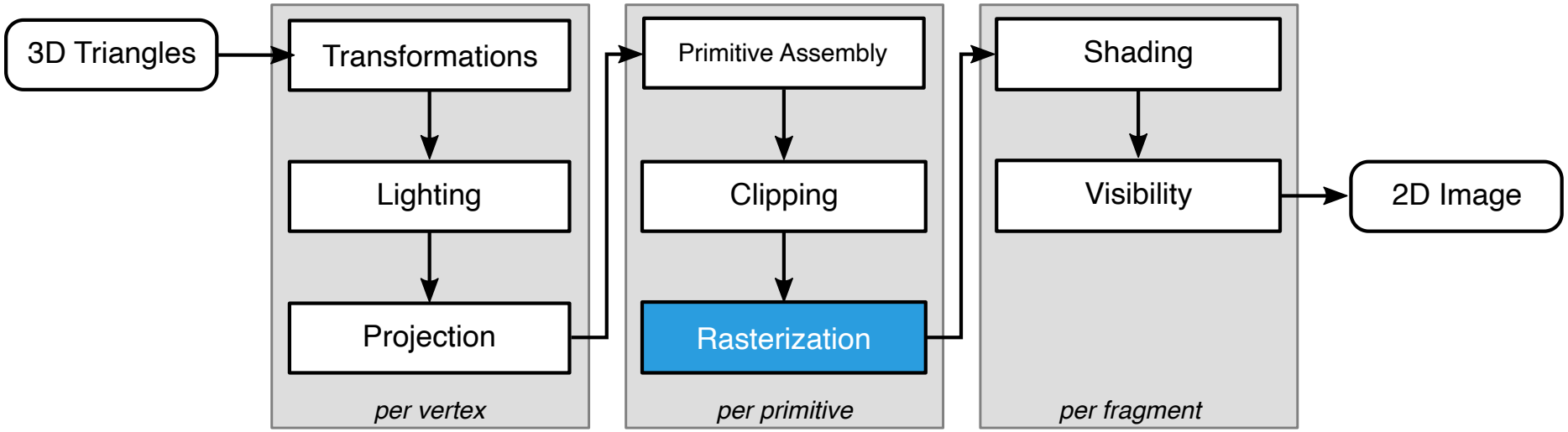
Clipping



Clipping removes everything outside the viewing frustum.

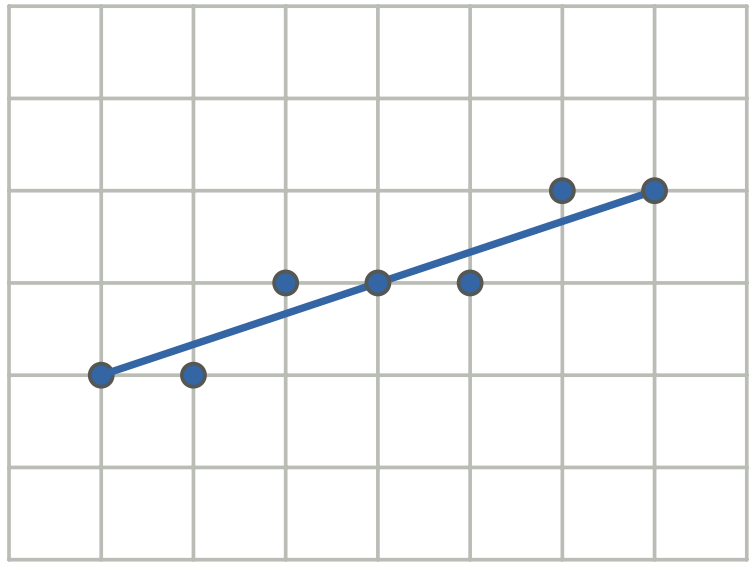
We don't discuss clipping here.

Rasterization



Line Rasterization

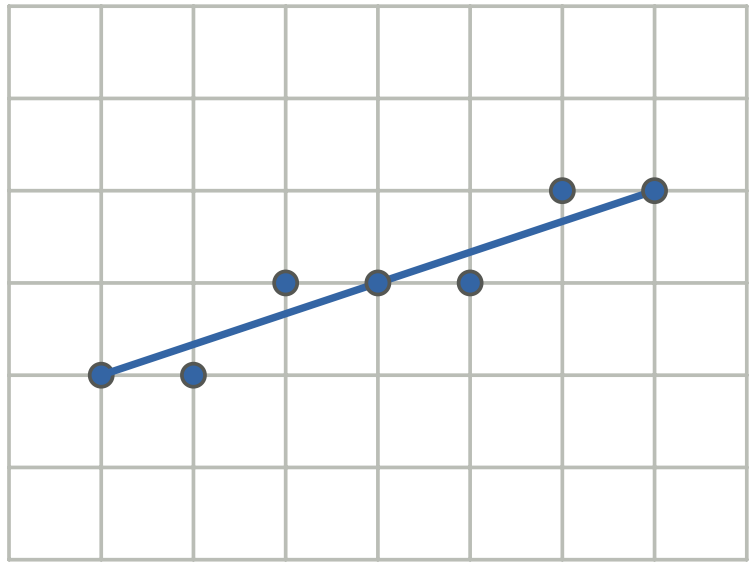
- Discretize line from (x_0, y_0) to (x_1, y_1) to pixel grid
  - Endpoints are integer coordinates (pixel positions)
  - Assume slope is in $[0, 1]$ and $x_0 < x_1$
  - Other cases follow by symmetry



How would you implement this?

Line Representation

- **Explicit:** Range of a function $y(x) = mx + t$ with
  - $m = \frac{y_1 - y_0}{x_1 - x_0} =: \frac{\Delta y}{\Delta x}$
  - $t = y_0 - m\,x_0$



First Guess

```
for (x=x0; x<=x1; ++x)
        set_pixel(x, round(m*x + t));
```

- ⊖ Multiplication
- ⊖ Rounding

Digital Differential Analysis

- Use an incremental algorithm!
- If the current pixel is (x_i, y_i) then
  - $x_{i+1} = x_i + 1$
  - $y_{i+1} = y_i + m$

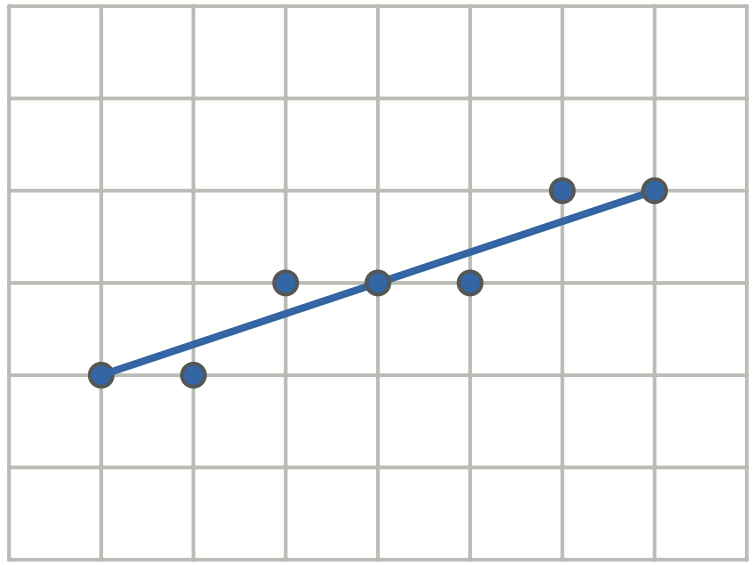
```
for (x=x0, y=y0; x<=x1; ++x, y+=m)
  set_pixel(x, round(y));
```

⊕ no multiplication

⊖ rounding, since y and m are floats

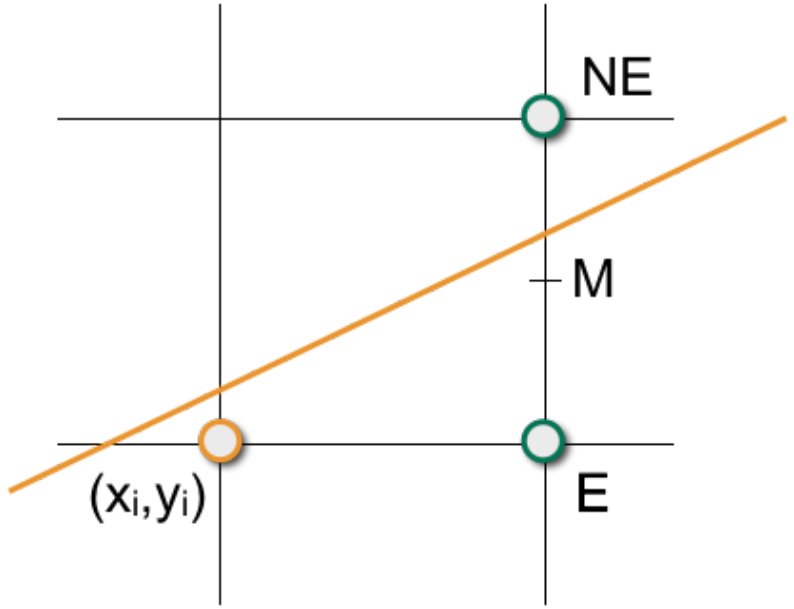
Line Representations

- **Explicit:** Range of a function $y(x) = mx + t$ with
  - $m = \frac{y_1 - y_0}{x_1 - x_0} =: \frac{\Delta y}{\Delta x}$
  - $t = y_0 - m\,x_0$
- **Implicit:** Zero-set of $F(x, y) = ax + by + c$ with
  - $a = (y_1 - y_0) = \Delta y$
  - $b = (x_0 - x_1) = -\Delta x$
  - $c = t\,(x_1 - x_0)$
  - $F(x, y) = 0 \;\Rightarrow$ point on line
  - $F(x, y) > 0 \;\Rightarrow$ point below line
  - $F(x, y) < 0 \;\Rightarrow$ point above line



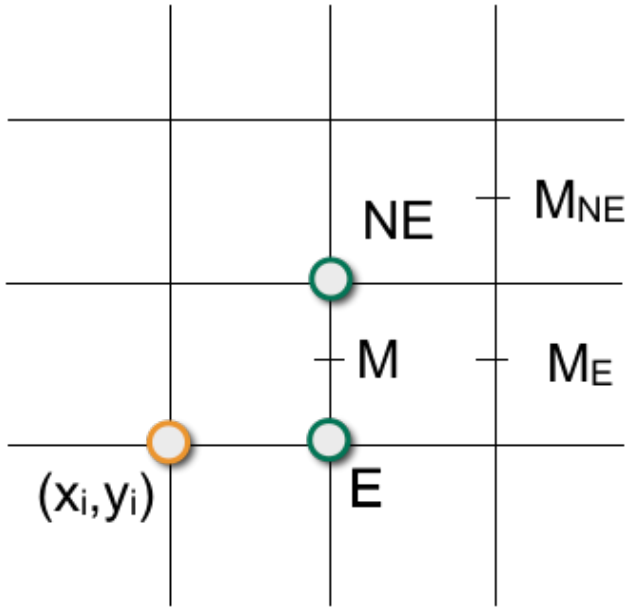
Bresenham Algorithm

- Next pixel can only be east (E) or north-east (NE)
  - Is the line below/above midpoint $\mathbf{M}$?
- Use a *decision variable* d:
  - $d = F(M) = F(x_i + 1, y_i + 0.5)$
  - If $d \leq 0$ then go east
  - If $d > 0$ then go north-east



Bresenham Algorithm

- Incrementally update decision variable d!
- If east was chosen:
  - $d_{\text{new}} = F(x_i + 2, y_i + 0.5) = d_{\text{old}} + a$
  - $\Delta\text{E} := a = \Delta y$
- If north-east was chosen:
  - $d_{\text{new}} = F(x_i + 2, y_i + 1.5) = d_{\text{old}} + a + b$
  - $\Delta\text{NE} := a + b = \Delta y - \Delta x$



Bresenham Algorithm

- Initialization of d
  - $d = F(x_0 + 1, y_0 + 0.5) = \Delta y - \Delta x/2$
  - Bad: $\Delta x/2$ may not be integer
- Use $2 \cdot F(x, y)$ instead of $F(x, y)$
  - Values of d, $\Delta\mathrm{E}$, $\Delta\mathrm{NE}$ are doubled
  - But sign of F and d remain unchanged

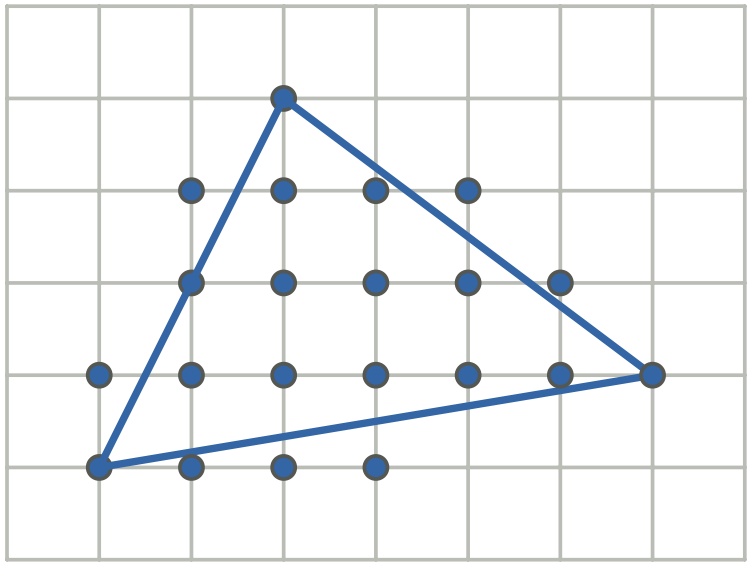
Bresenham Algorithm

```
Δx  = x1-x0;
Δy  = y1-y0;
d   = 2*Δy - Δx;
ΔE  = 2*Δy;
ΔNE = 2*(Δy - Δx);

set_pixel(x0, y0);

for (x = x0, y = y0; x <= x1;)
{
    if (d <= 0) { d += ΔE;  ++x;      }
    else        { d += ΔNE; ++x; ++y; }
    set_pixel(x, y);
}
```

⊕ Only integer arithmetic

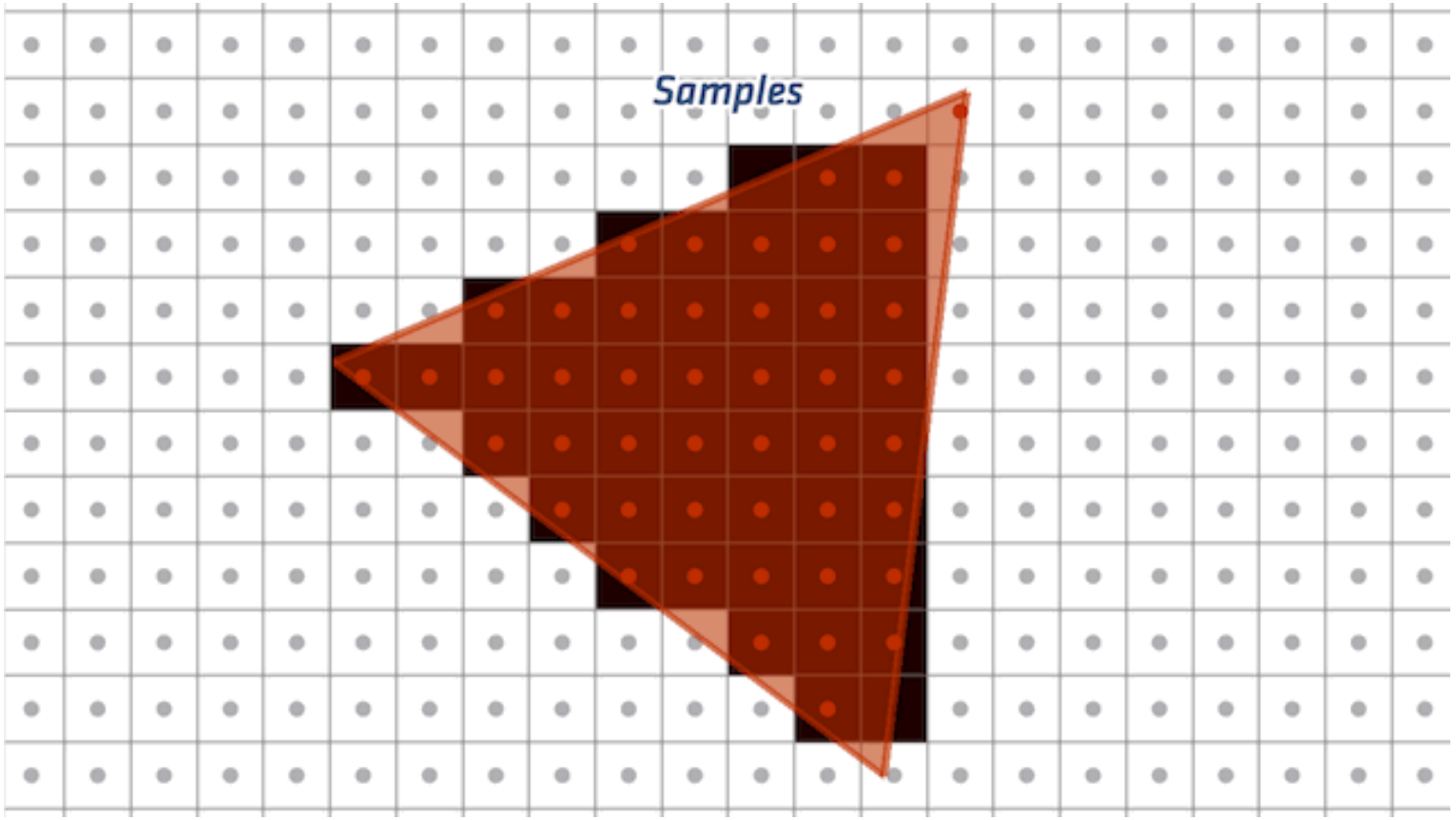
Triangle Rasterization

- Enumerate all pixels inside a 2D triangle
  - Inside test for every pixel is too expensive
- Compute horizontal spans in each scanline
  - Compute the intersections with triangle edges, fill all pixels inbetween

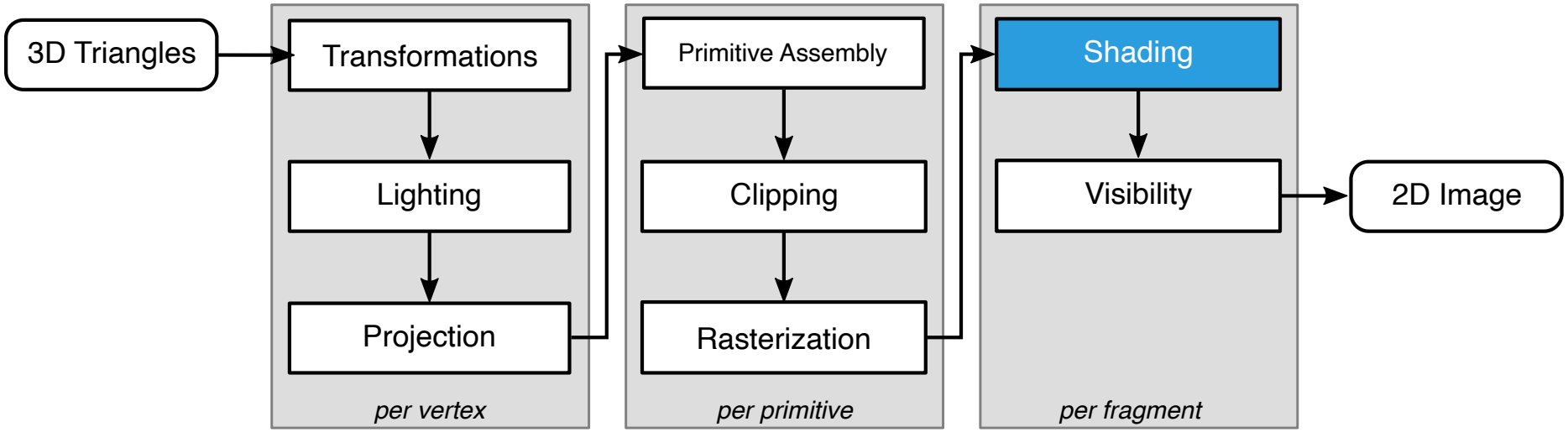


Many special cases to consider!

Triangle Rasterization



Shading


3D Triangles
Transformations
Lighting
Projection
per vertex
Primitive Assembly
Clipping
Rasterization
per primitive
Shading
Visibility
per fragment
2D Image

Shading: Fill the Interior

- **Flat Shading**
  - Compute lighting per face
  - Facetted appearance

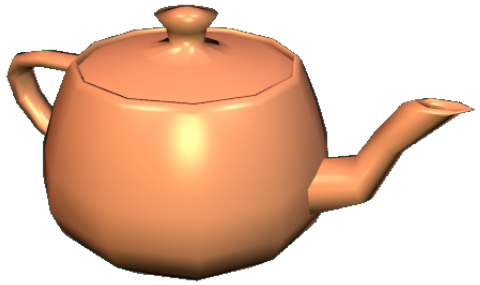
Mach band effect

Shading: Fill the Interior

- **Flat Shading**
  - Compute lighting per face
  - Facetted appearance

- **Gouraud Shading**
  - Compute lighting per vertex
  - Linear interpolation of colors
  - Might loose small highlights

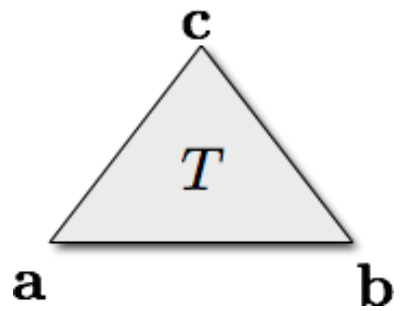
- **Phong Shading**
  - Linear interpolation of vertex normals
  - Compute lighting per pixel
  - Captures small highlights



Normal Vectors

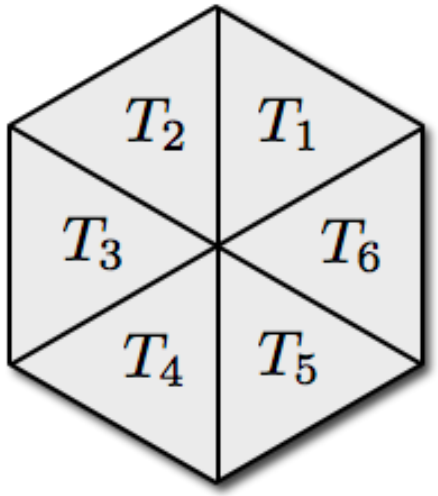
- Triangle normal

$$\mathbf{n}(T) = \frac{(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})}{\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|}$$



- Vertex normal
  - Average of incident triangles' normals $\mathbf{n}(T_i)$
  - Weighted by area or opening angle $w(T_i)$

$$\mathbf{n}(V) = \frac{\sum_{T_i \ni V} w(T_i)\,\mathbf{n}(T_i)}{\left\|\sum_{T_i \ni V} w(T_i)\,\mathbf{n}(T_i)\right\|}$$



Interpolate Vertex Normals

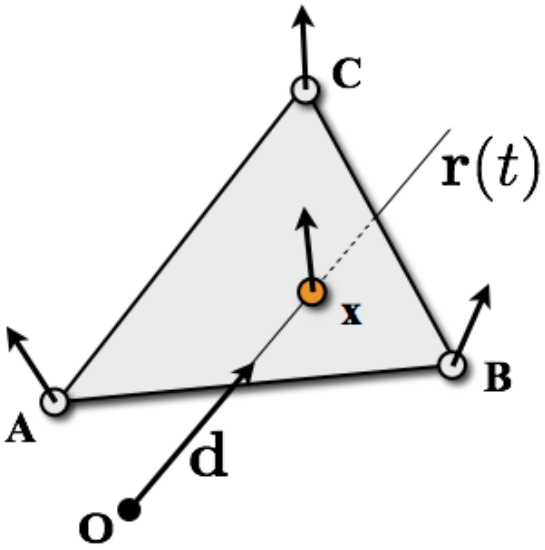
- Intersection point with barycentric coordinates

$$\mathbf{x} = \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C}$$

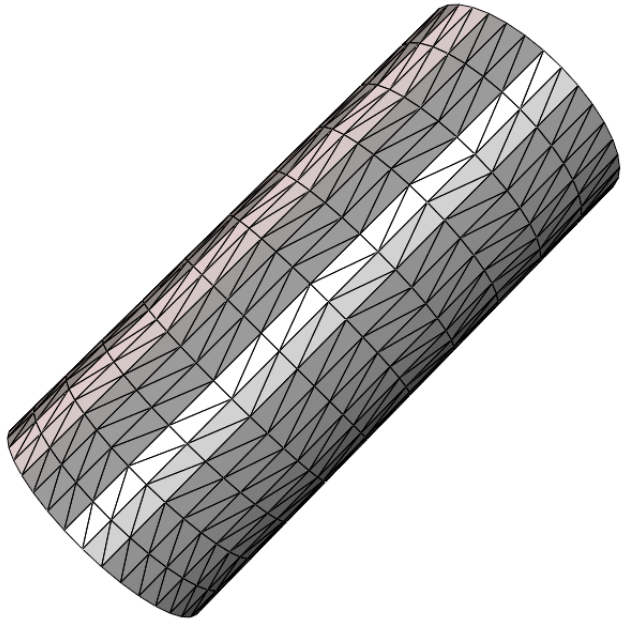
- Linearly interpolate vertex normals

$$\mathbf{n}(\mathbf{x}) = \alpha\mathbf{n}(\mathbf{A}) + \beta\mathbf{n}(\mathbf{B}) + \gamma\mathbf{n}(\mathbf{C})$$

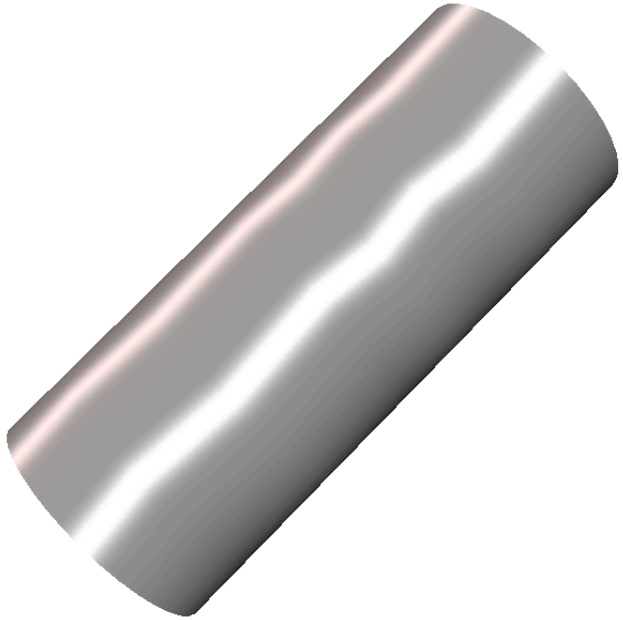
- Use $\mathbf{n}(\mathbf{x})$ to light point $\mathbf{x}$



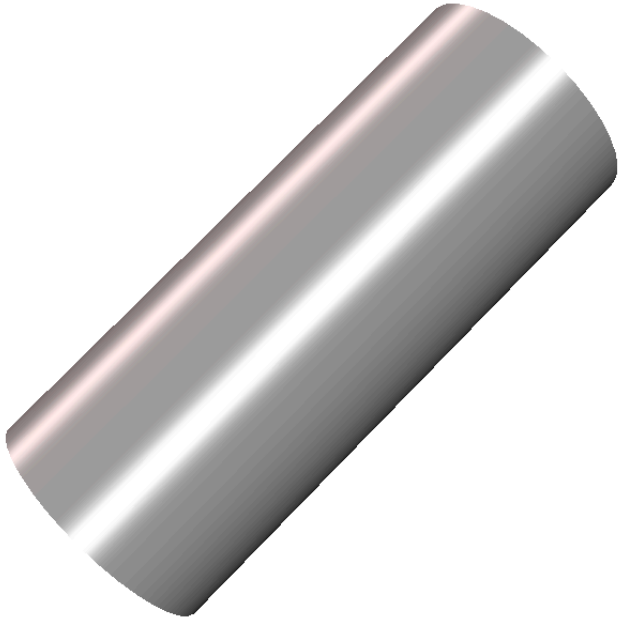
Influence of Vertex Normals



triangulation (flat shading)



no weighting



angle-weighted

Shading: Fill the Interior

Flat Shading

Gouraud Shading

Phong Shading

Shading: Fill the Interior

Quiz: Barycentric Interpolation

During rasterization normal vectors and texture coordinates for each pixel are computed by barycentric interpolation.

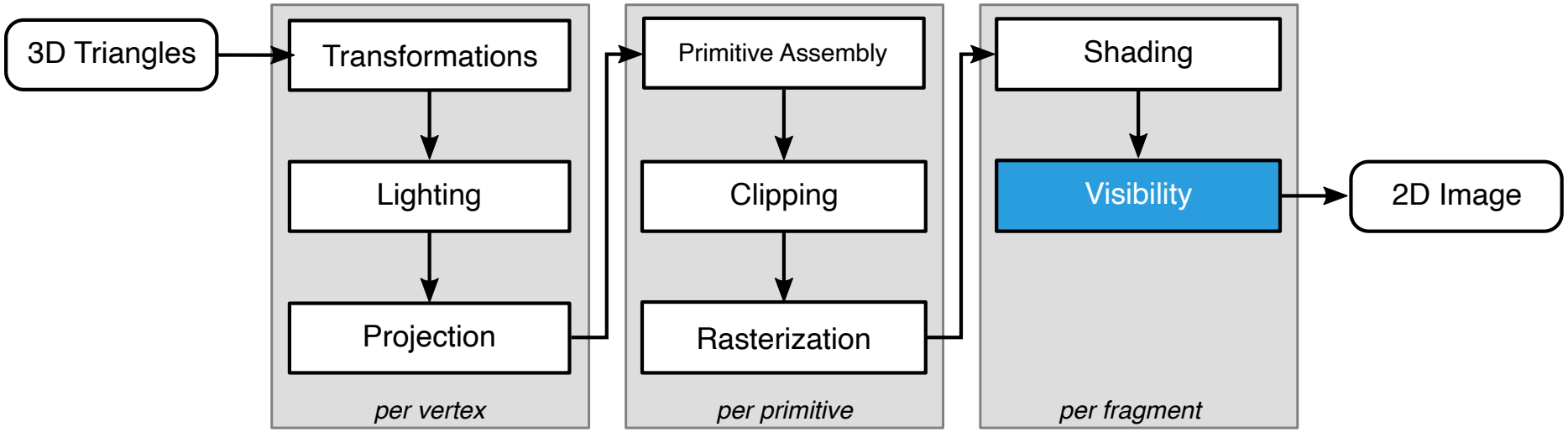
Where should we compute the barycentric coordinates of the pixels?

A: In 2D window coordinates

B: In 3D camera coordinates

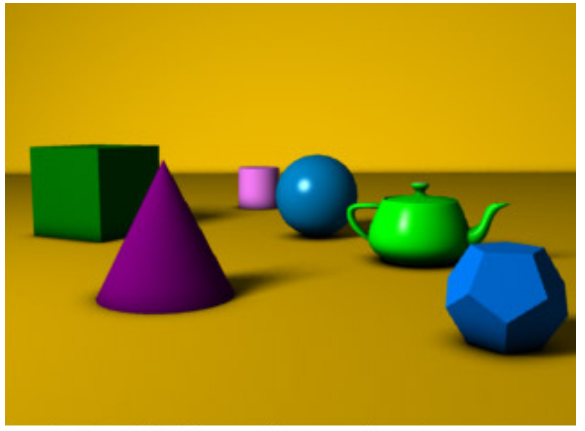
C: Makes no difference

Visibility


3D Triangles
Transformations
Lighting
Projection
per vertex
Primitive Assembly
Clipping
Rasterization
per primitive
Shading
Visibility
per fragment
2D Image

Z-Buffer

- Store current minimum z-value for each pixel
  - After model transformation, view transformation, projection transformation, and viewport transformation
- Additional buffer for depth values
  - Framebuffer stores RBG color values
  - Depth buffer (z-buffer) stores depth values
  - Storage: additional 16 to 32 bits per pixel



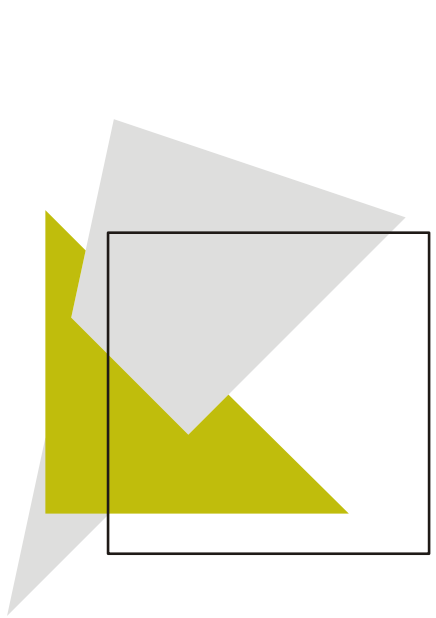
Images from Wikipedia

Z-Buffer

- Initialize depth buffer to ∞
- Rasterize triangles, interpolate depth value per fragment
  - each fragment (x,y) still has a depth value z
- Depth buffer test & update

```
// draw color rgb to pixel (x,y)
void set_pixel(x, y, z, rgb)
{
    // is current pixel closer than stored z-value?
    if (z < zbuffer[x,y])
    {
        // write pixel's color to frame buffer
        framebuffer[x,y] = rgb;

        // update depth buffer
        zbuffer[x,y] = z;
    }
}
```

Z-Buffer



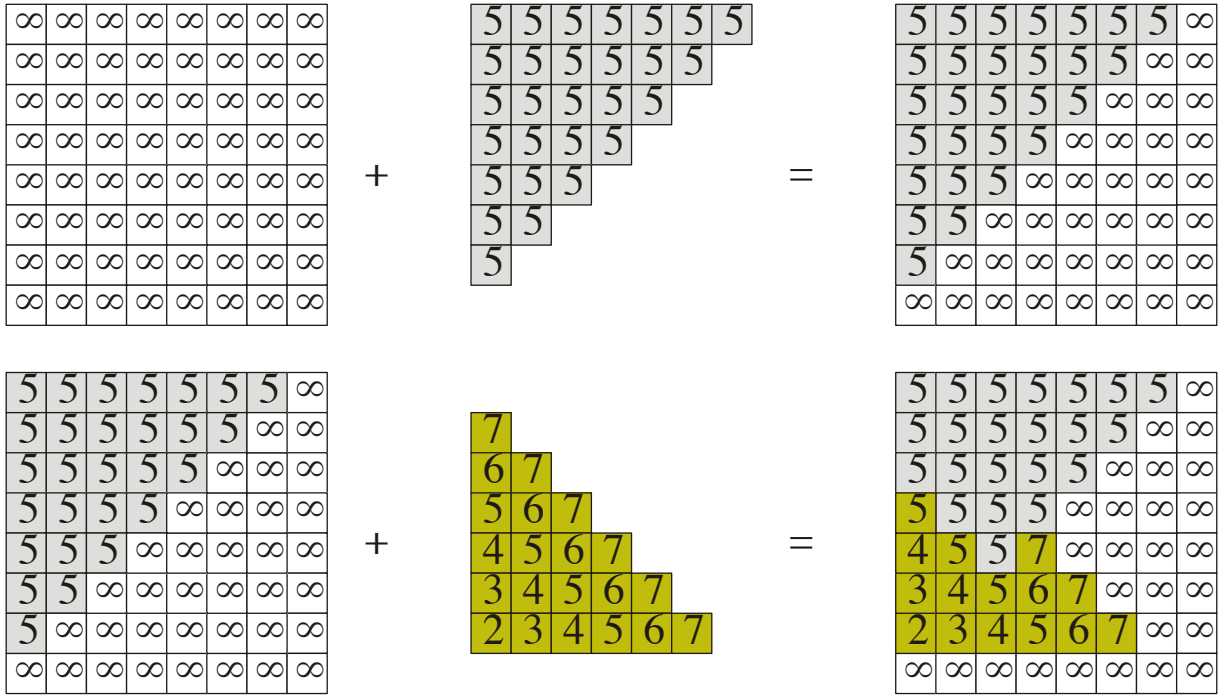


Image from Wikipedia

Quiz: Z-Buffer

- How will the z-buffer resolve visibility when two triangles A and B lie in the same plane and overlap? Assume triangle A is drawn before triangle B.

A: A will be completely visible.

B: B will be completely visible.

C: Some pixels from A and some pixels from B will be visible in the overlap region.

D: One triangle is completely visible, but it is random which one.

Precision Issues

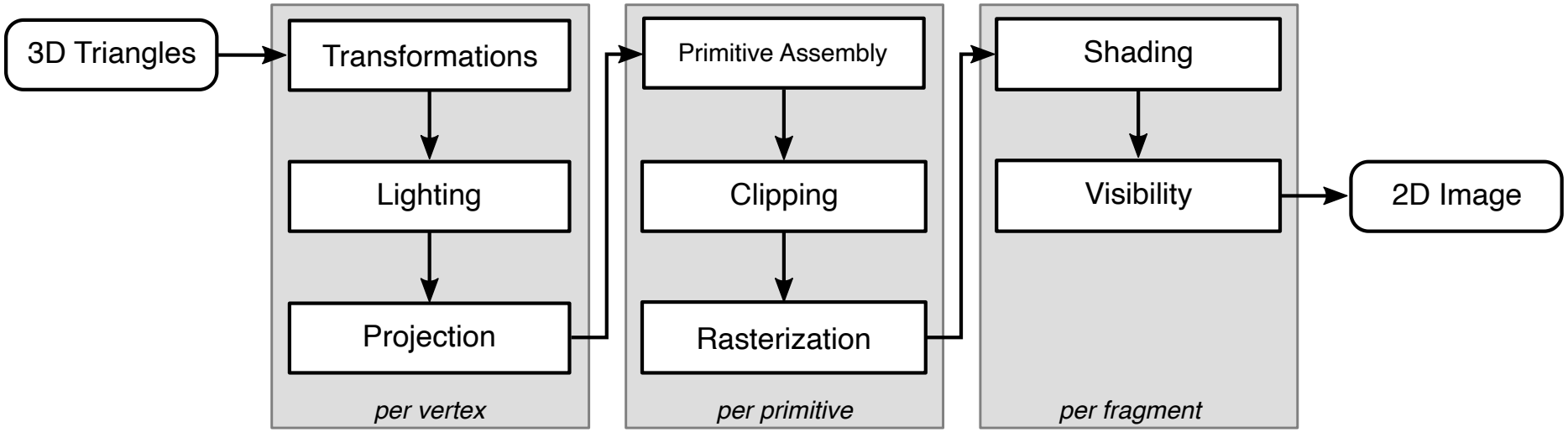
Logarithmic Z-Buffer

Quiz: Transform Normal Vectors

- If an object is affinely transformed by a matrix $\mathbf{M}$, which matrix is used to transform its normal vectors?

A: $\mathbf{M}$ B: $\mathbf{M}^{-1}$ C: $\mathbf{M}^{\mathsf{T}}$ D: $\mathbf{M}^{-\mathsf{T}}$

Rasterization Pipeline



Literature

- Marschner & Shirley: *Fundamentals of Computer Graphics*, 5th Edition, AK Peters, 2021.
  - Chapter 9