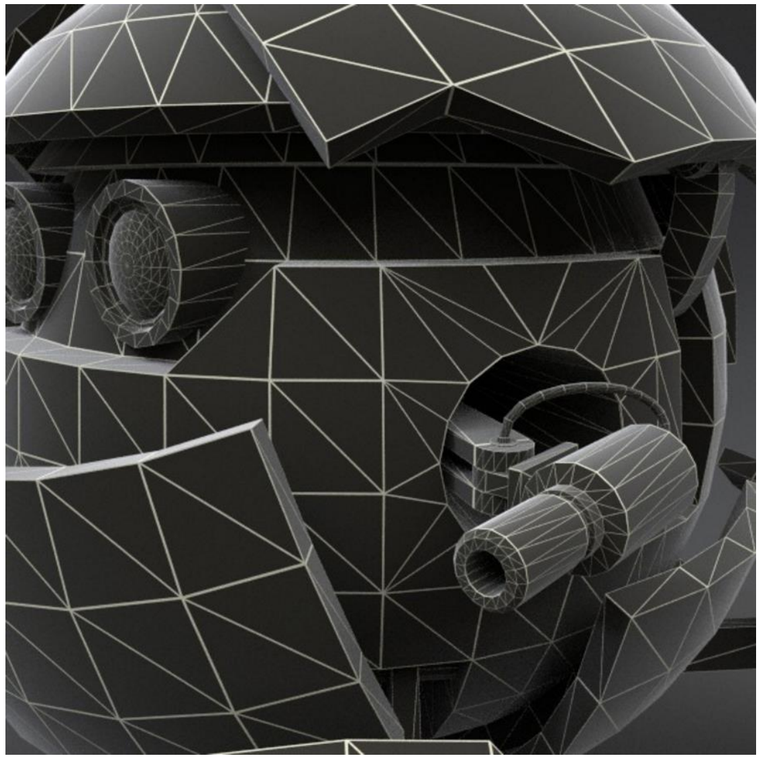
The Rasterization Pipeline

Computer Graphics
CMU 15-462/15-662, Fall 2016

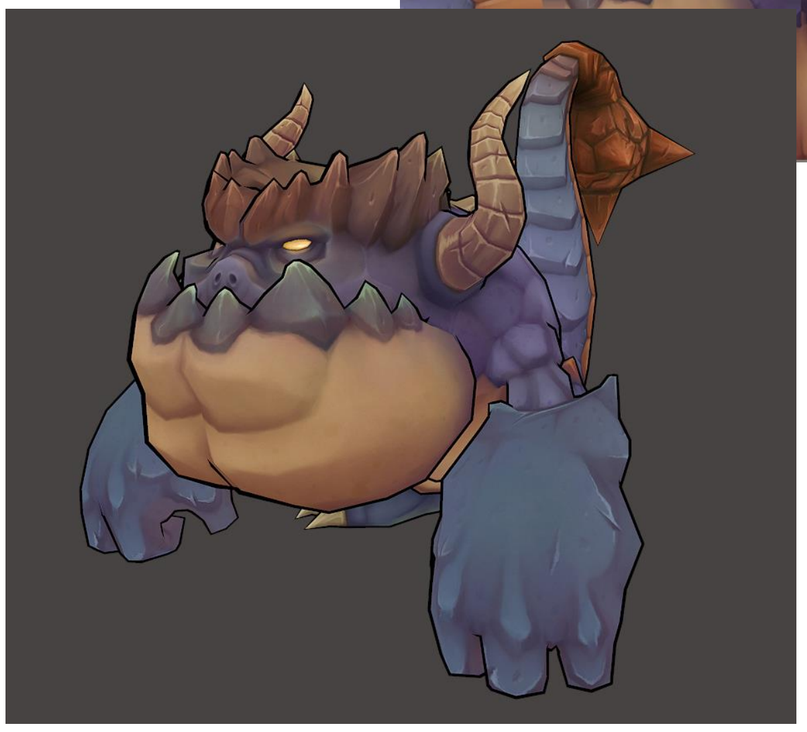
Announcements

- A1 due tomorrow
- A2 out tomorrow

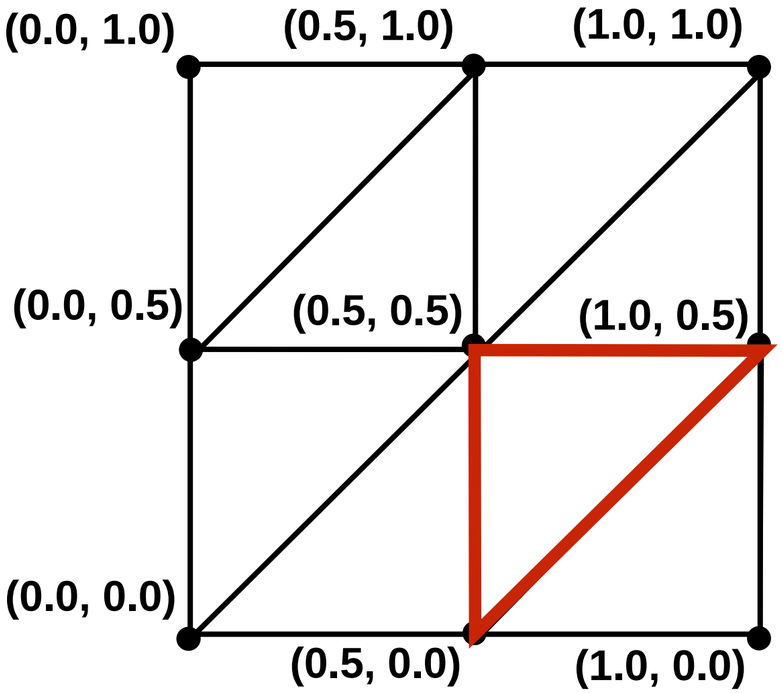
Texture mapping recap



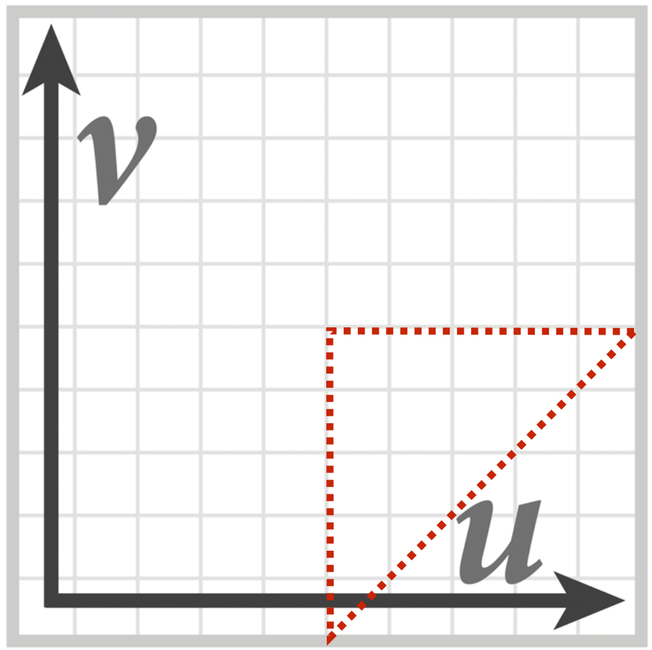
Texture mapping



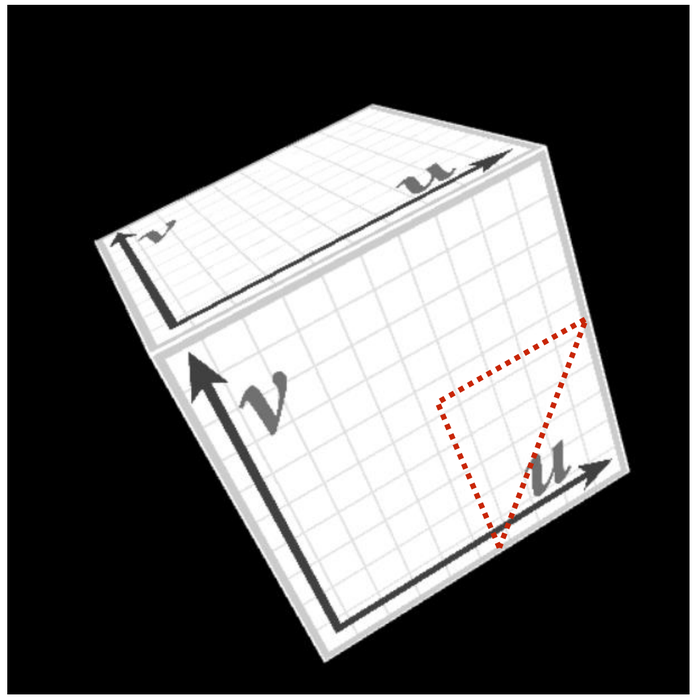
Texture coordinates



Geometry for one face of the cube



Texture (2D image)

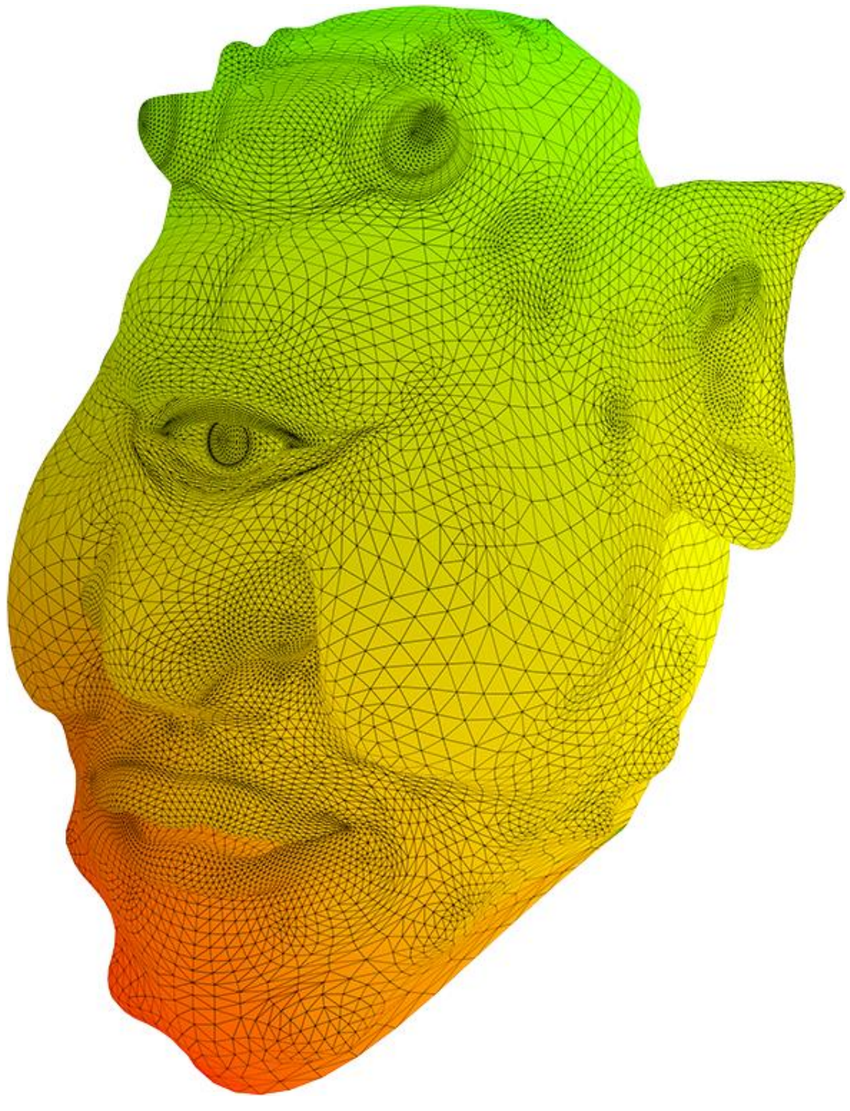


Rendered result

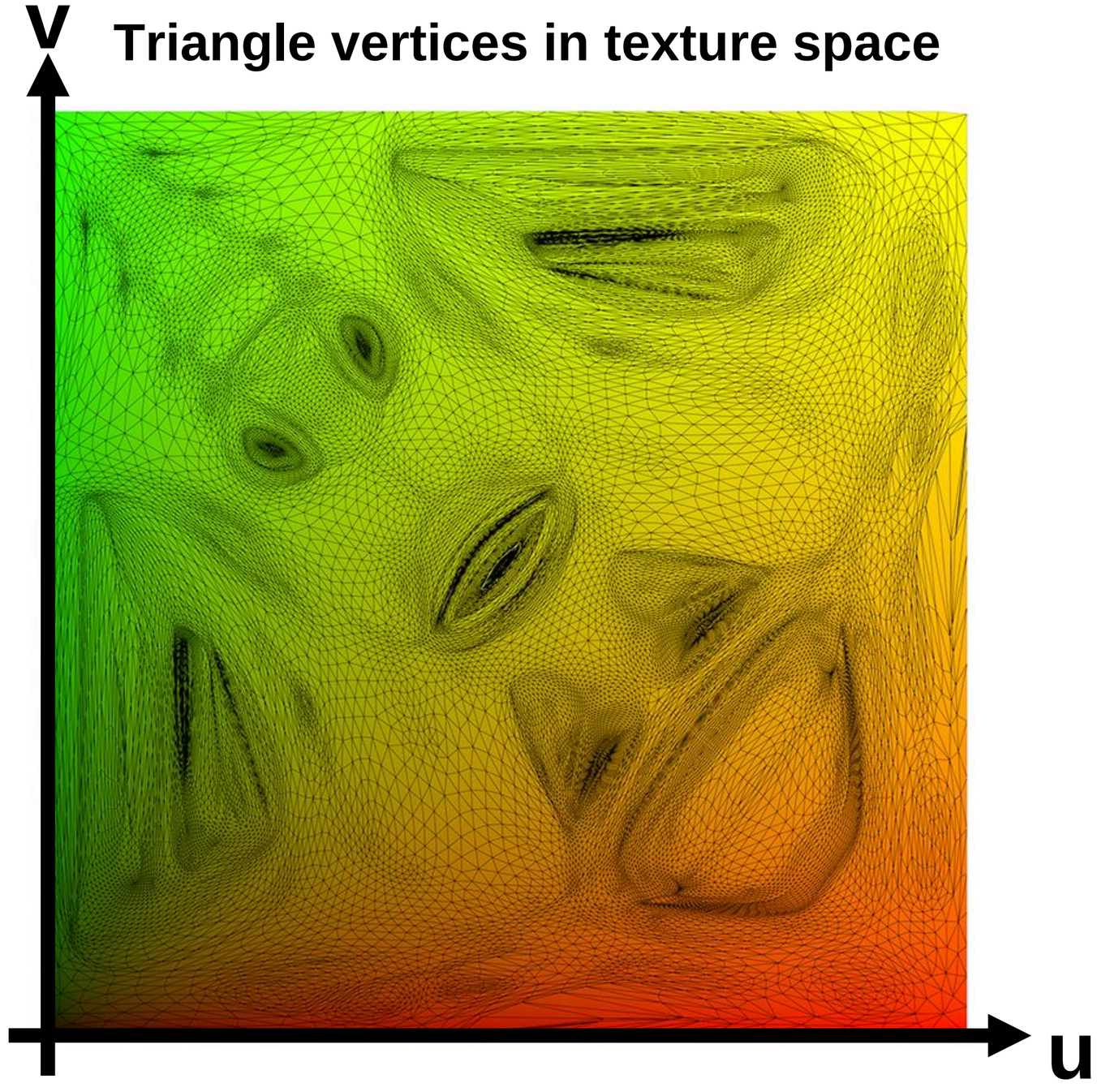
“Texture coordinates” define a mapping from 3D surface (triangle vertices) to points in texture domain.

Texture coordinates

Visualization of texture coordinates



Triangle vertices in texture space

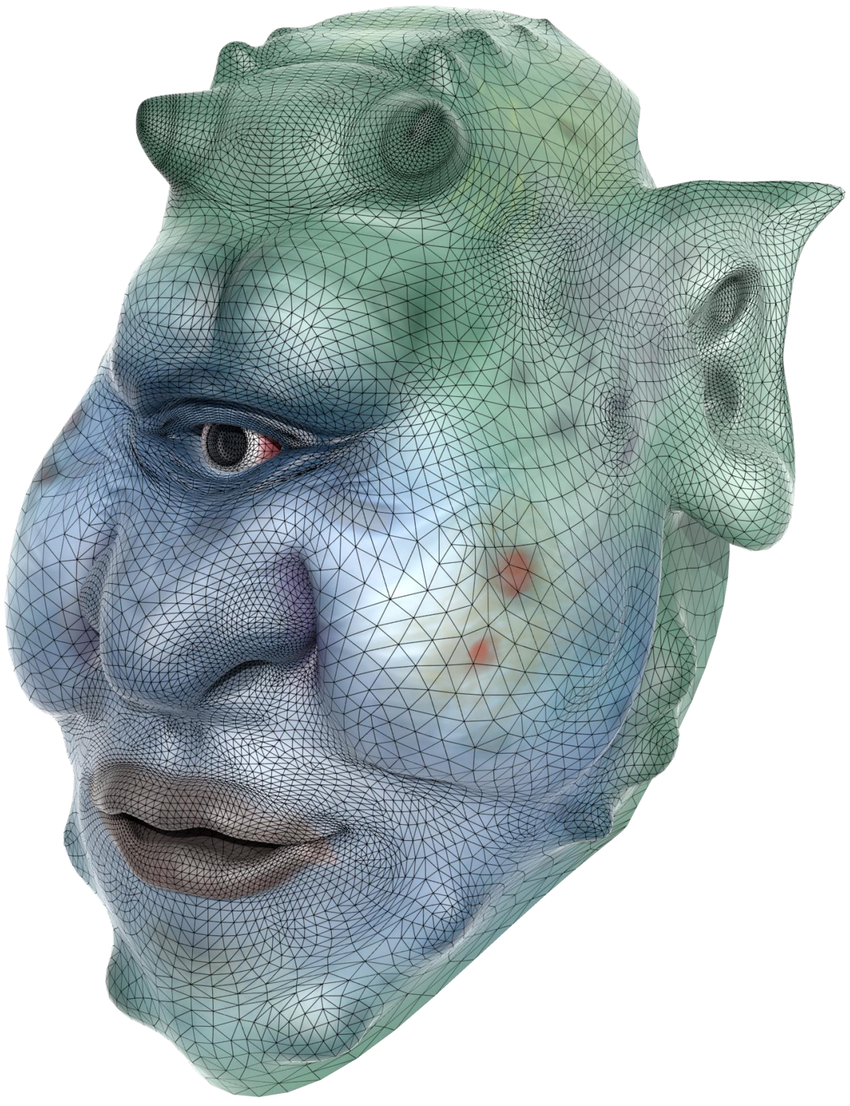


Each vertex has a coordinate (u,v) in texture space.
(Actually coming up with these coordinates is another story!)

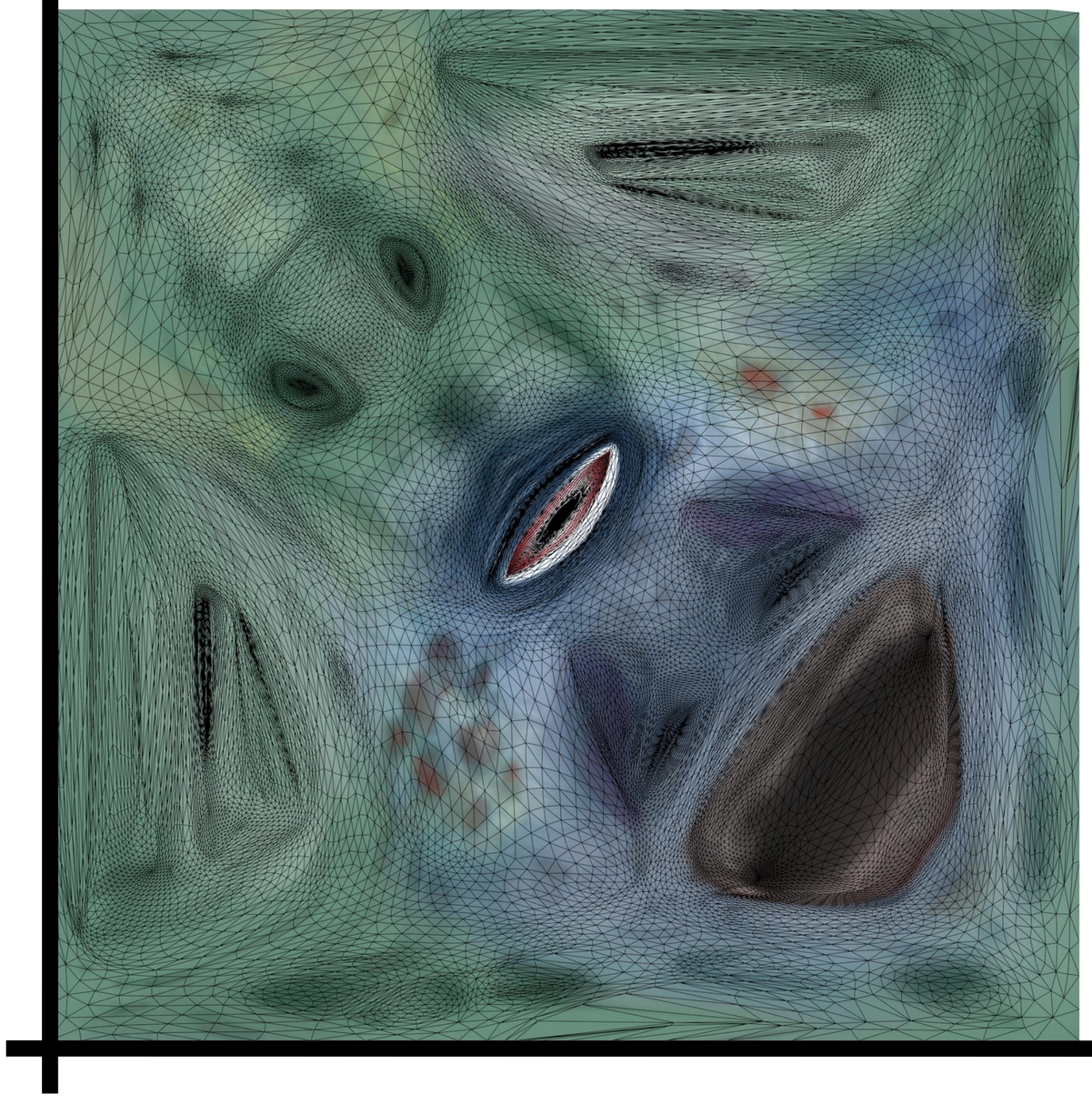
CMU 15-462/662, Fall 2016

Texture mapping adds detail

Rendered result



Triangle vertices in texture space

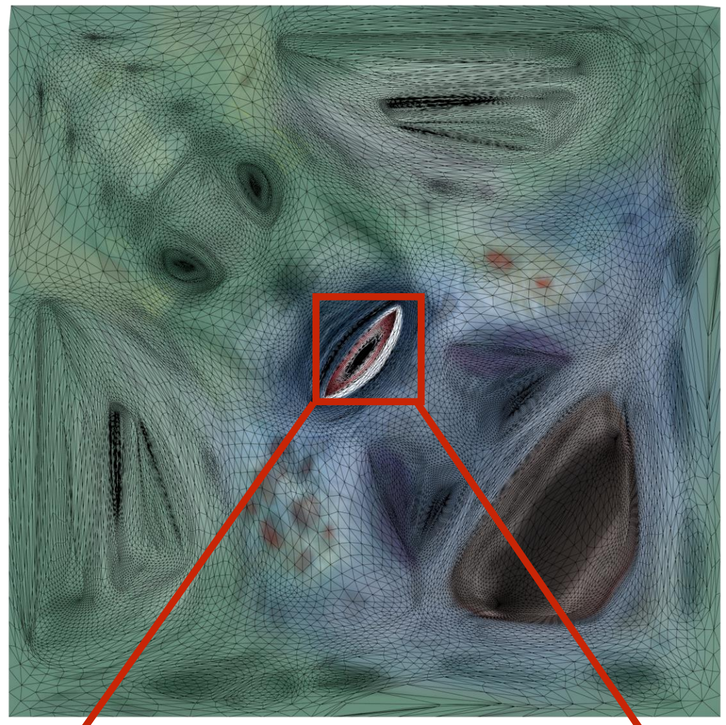


Texture mapping adds detail

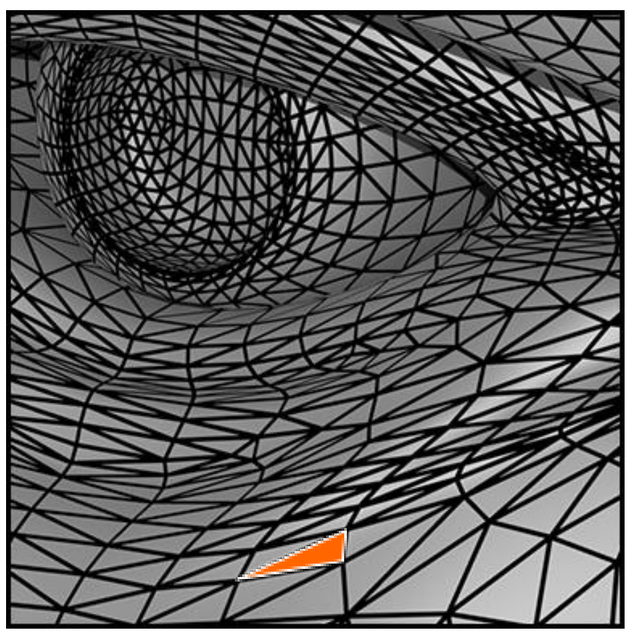
rendering without

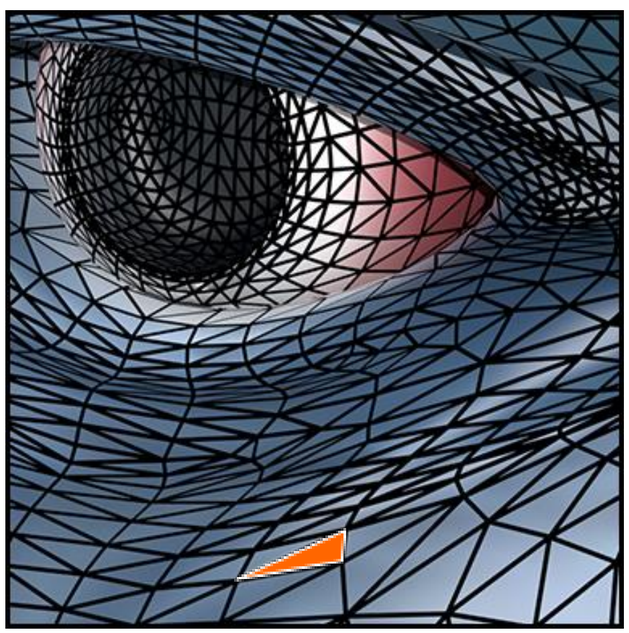
rendering with texture

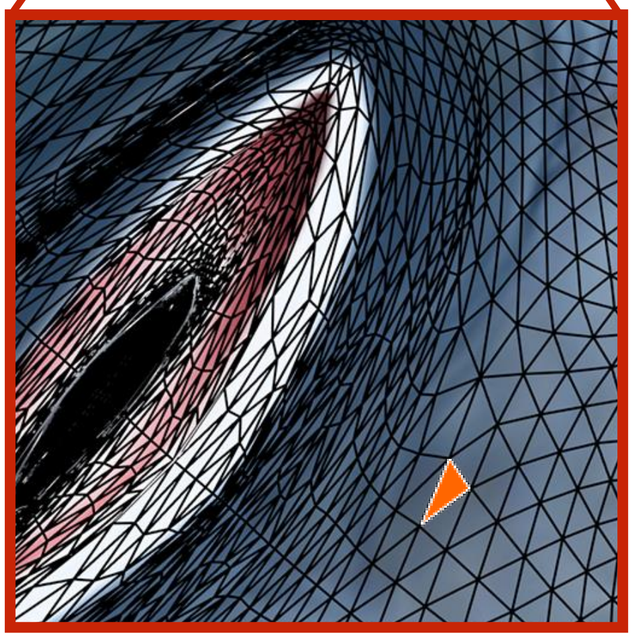
texture image



zoom







Each triangle “copies” a piece of the image back to the surface.

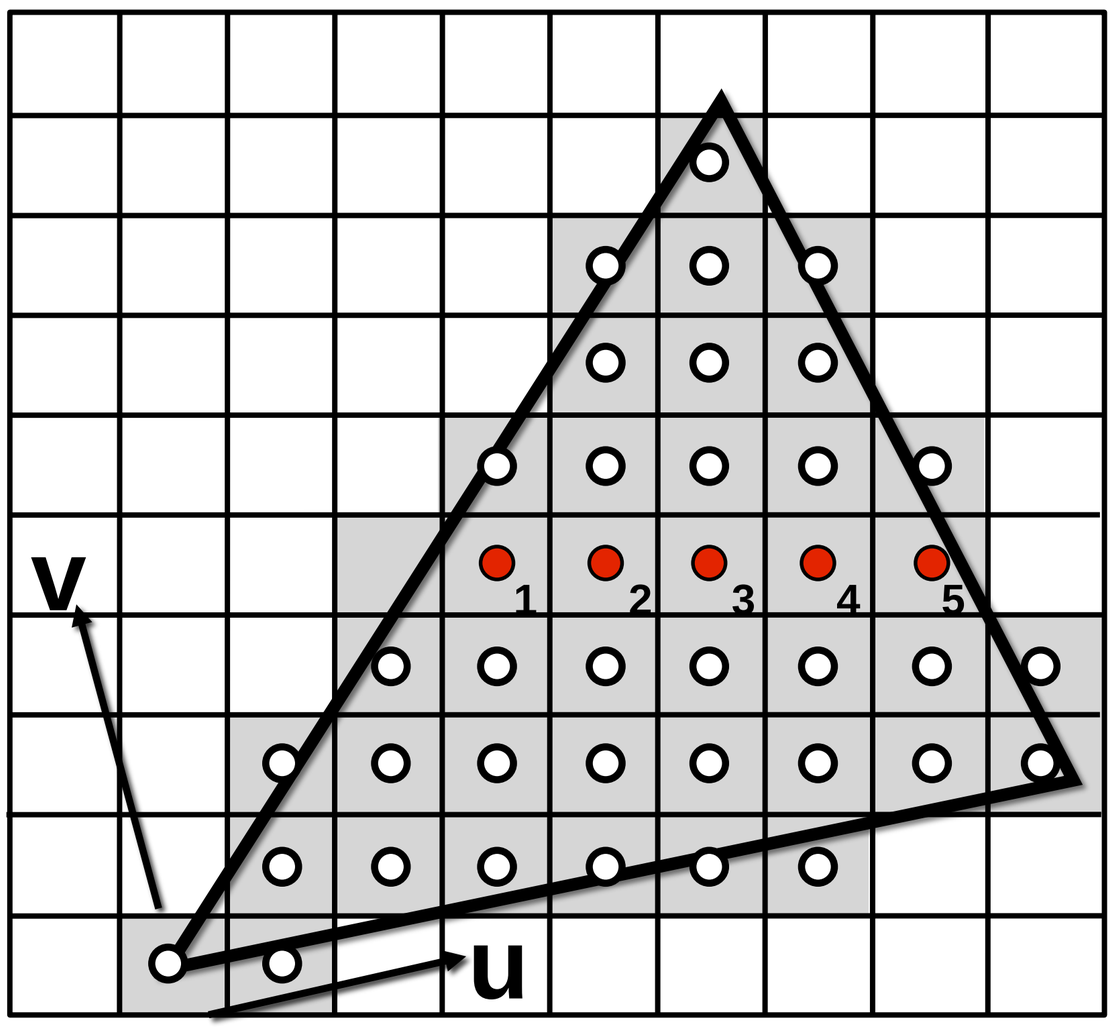
Simple texture mapping operation

```
for each covered screen sample (x,y):
    (u,v) = evaluate texcoord value at (x,y)
    float3 texcolor = texture.sample(u,v); ← "just" an image lookup…
    set sample's color to texcolor;
```

Q: How would you implement the texture.sample(u, v) routine?

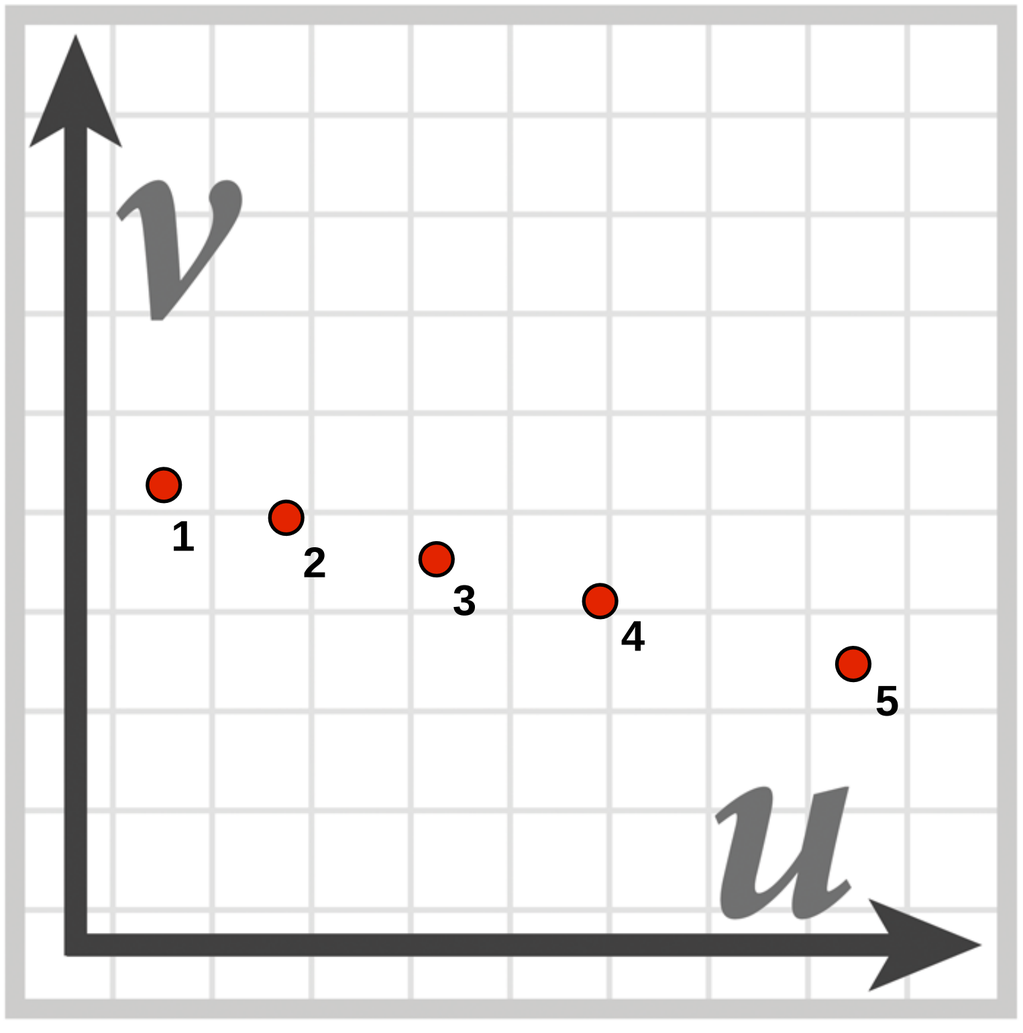
Texture space samples

Sample positions in XY screen space



Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

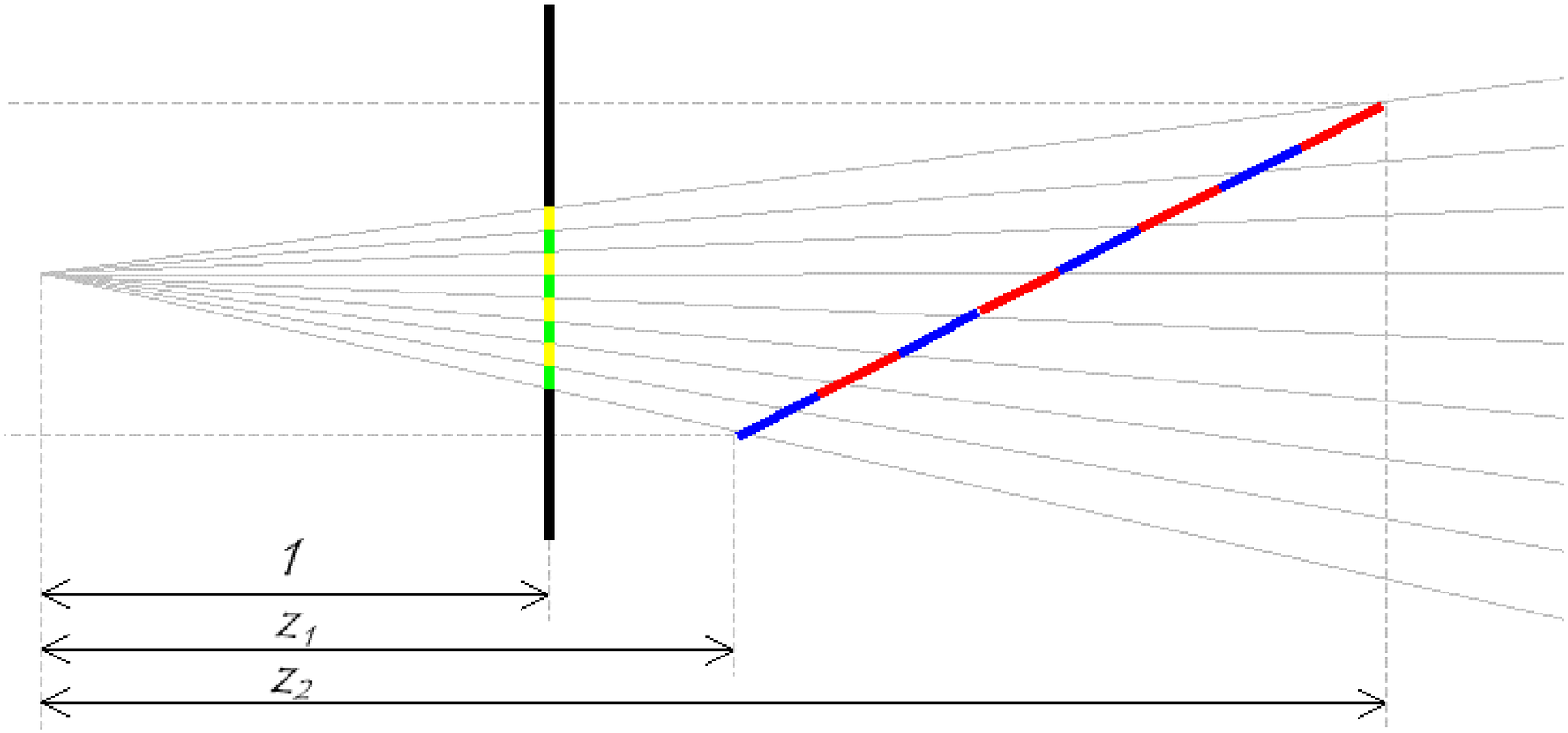
Sample positions in texture space



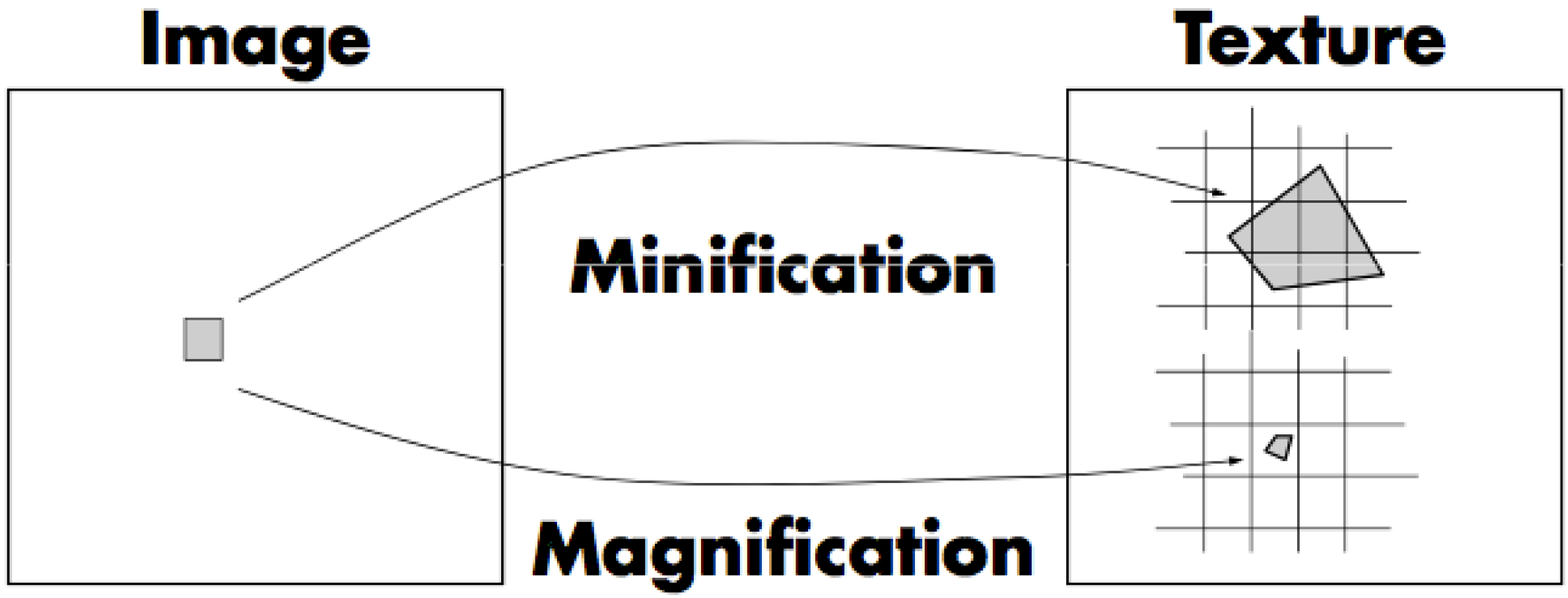
Texture sample positions in texture space (texture function is sampled at these locations)

Q: what does it mean that equally-spaced points in screen space move further apart in texture space?

Texture space samples



From pixels to texels



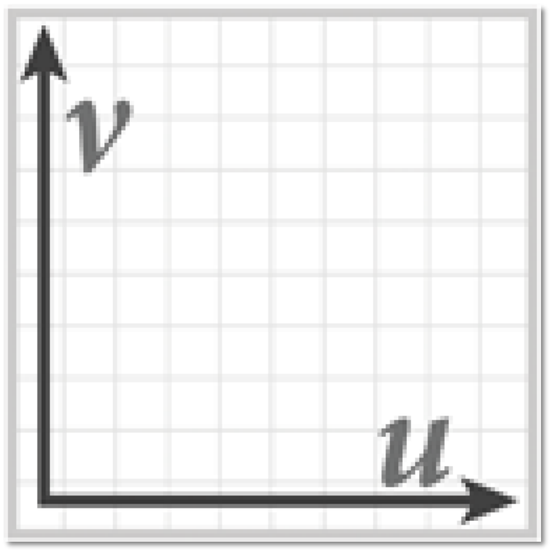
- **Minification:**
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away
  - Texture map is too detailed – if undersampled, leads to aliasing
- **Magnification:**
  - Area of screen pixel maps to tiny region of texture (interpolation required)
  - One texel maps to many screen pixels
  - Example: when camera is very close to scene object
  - Texture map is not detailed enough – leads to blurry/pixelated results

Figure credit: Akeley and Hanrahan

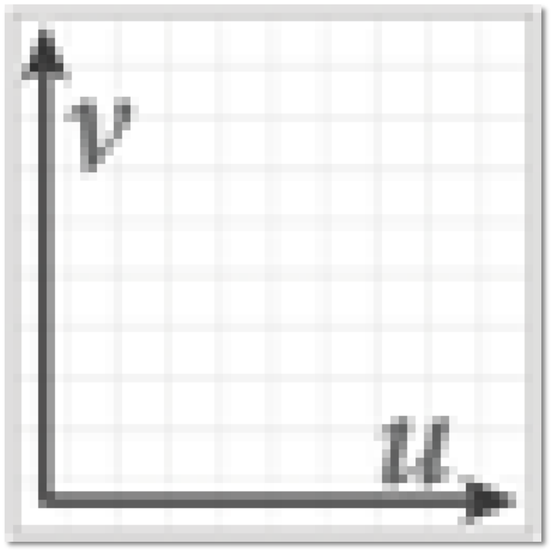
High-frequency texture: aliasing far from camera

Filtered texture: blurry image close to camera

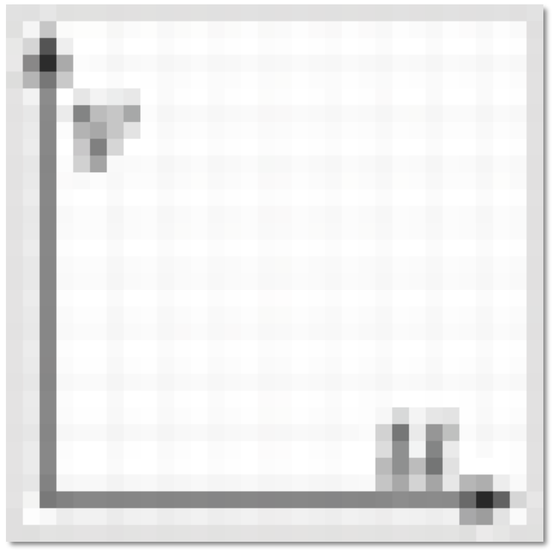
Mipmaps (L. Williams 83)



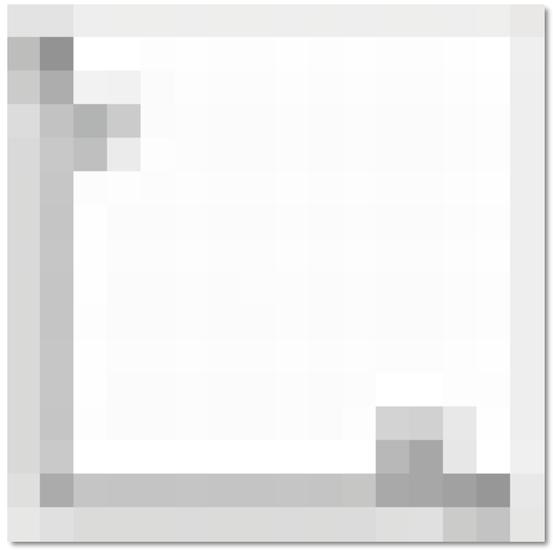
Level 0 = 128x128



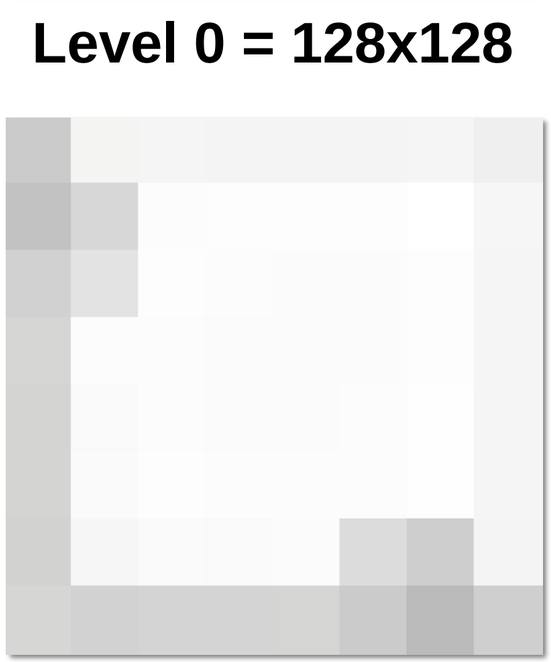
Level 1 = 64x64



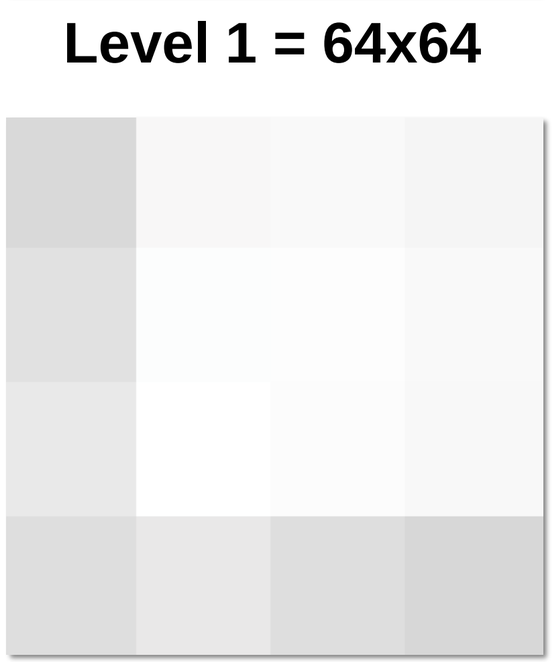
Level 2 = 32x32



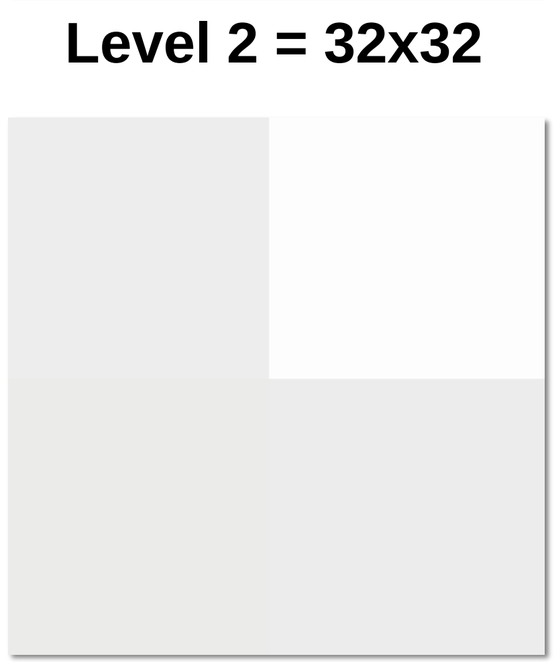
Level 3 = 16x16



Level 4 = 8x8



Level 5 = 4x4

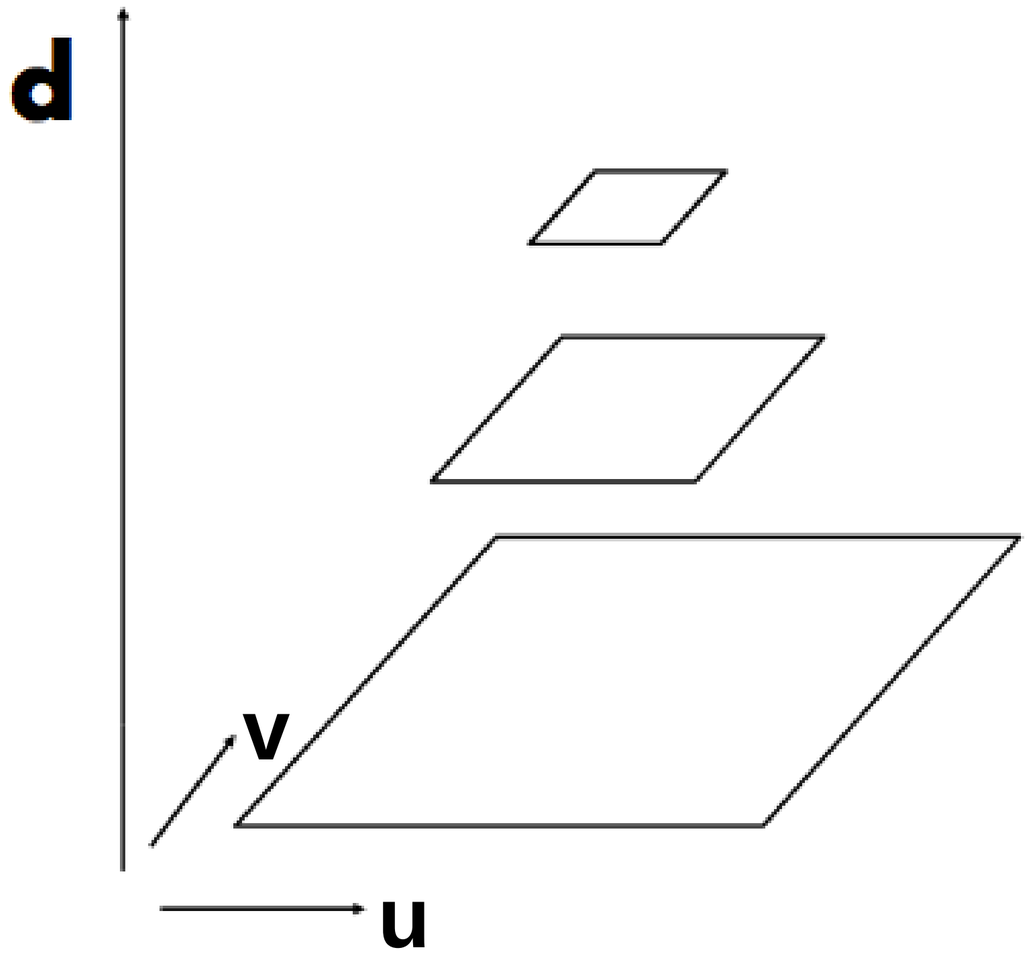


Level 6 = 2x2

Level 7 = 1x1

Texels at higher levels store integral of the texture function over a region of texture space
Texels at higher levels represent low-pass filtered version of original texture signal

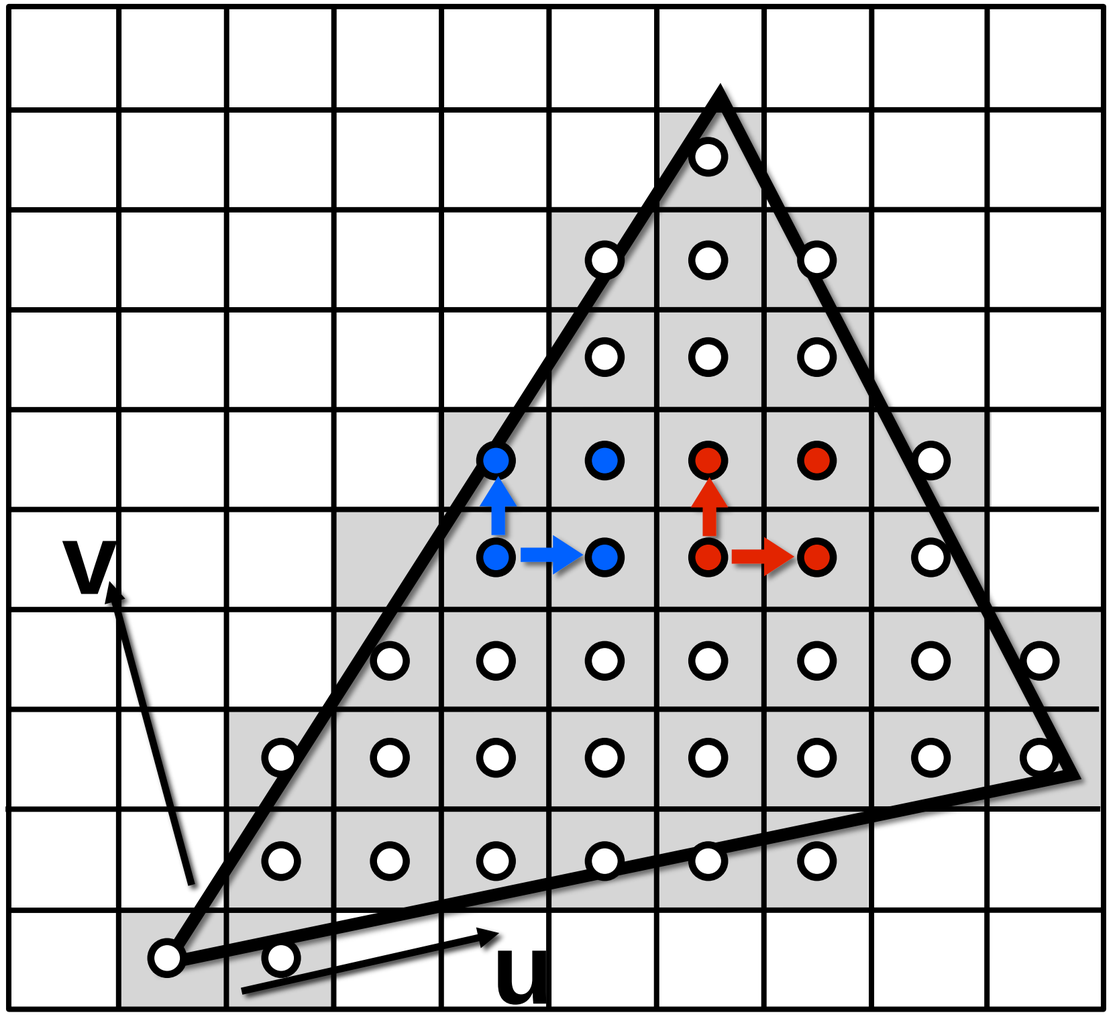
Mipmaps (L. Williams 83)



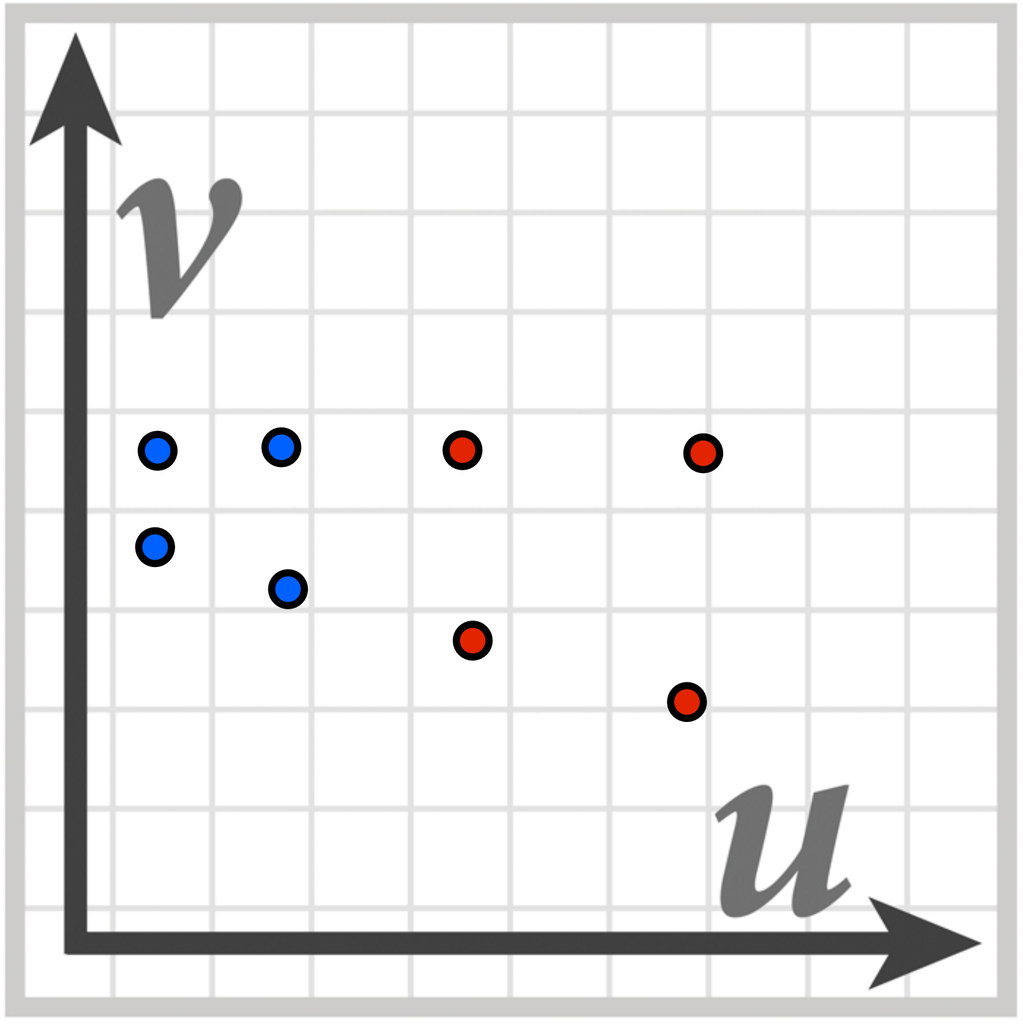
“Mip hierarchy”
level = *d*

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples



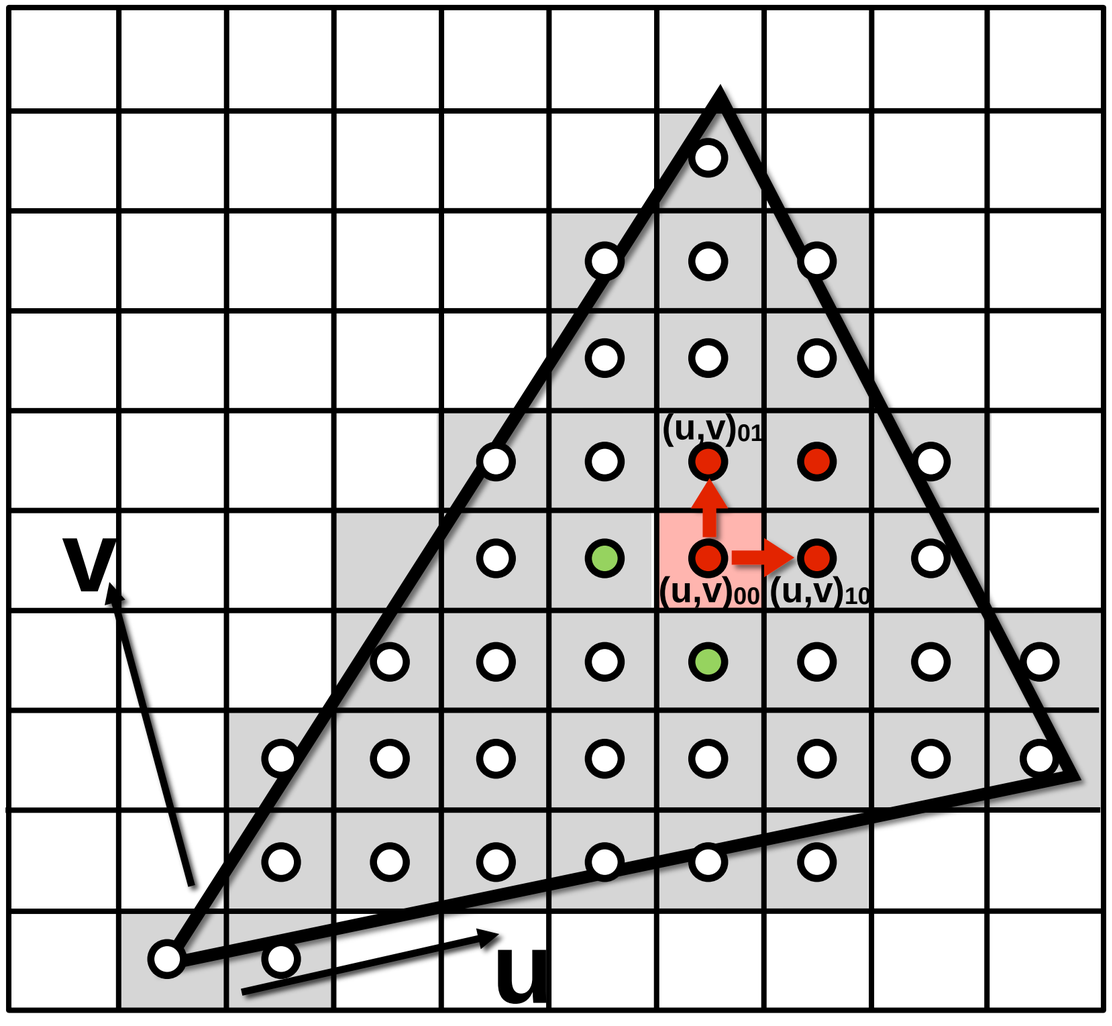
Screen space

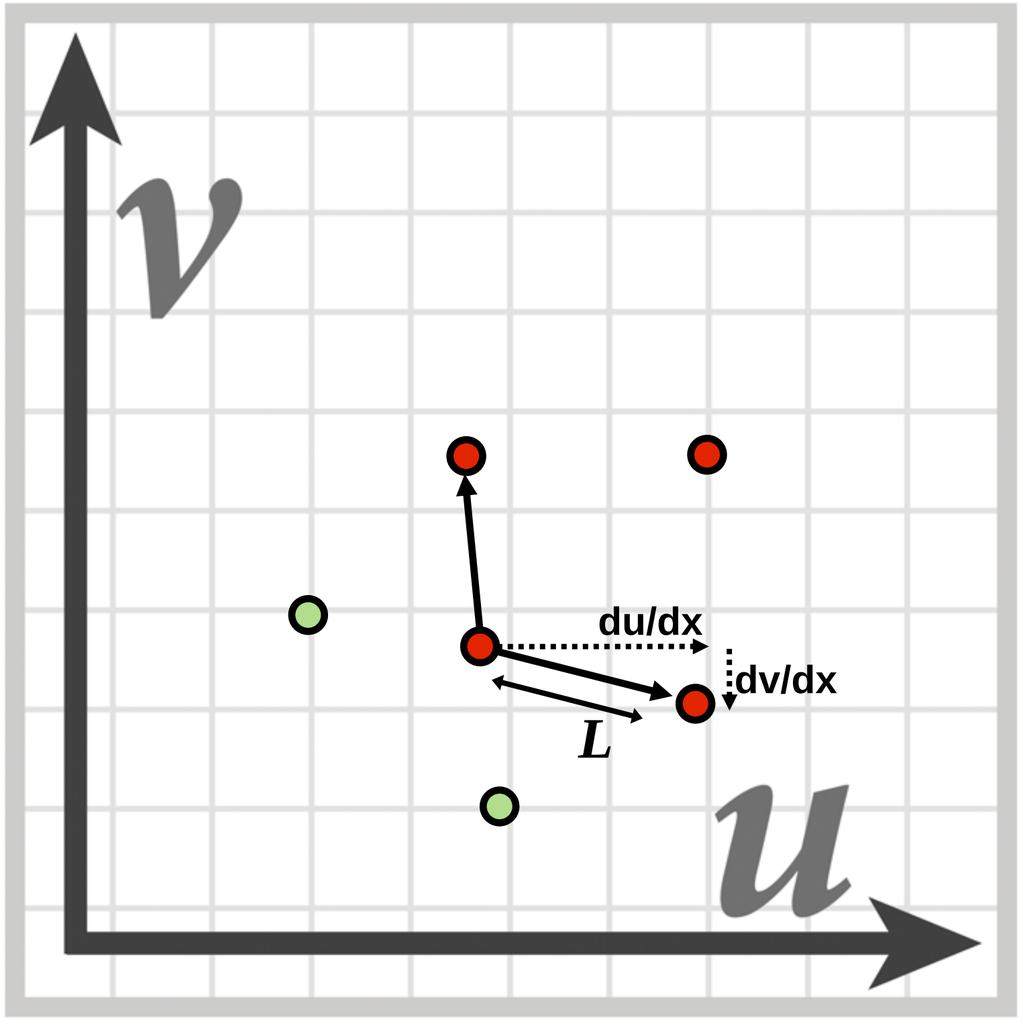


Texture space

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples





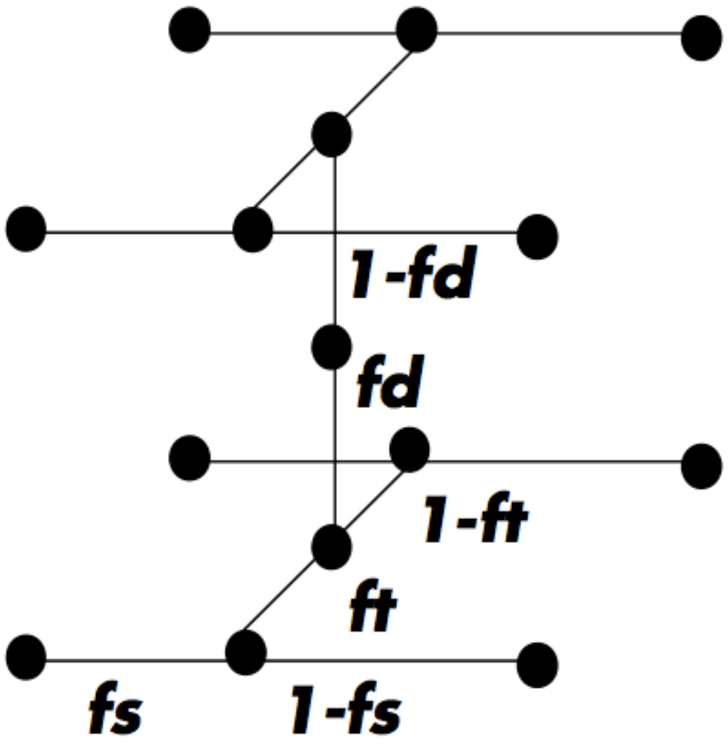
$du/dx = u_{10}-u_{00}$ $dv/dx = v_{10}-v_{00}$

$du/dy = u_{01}-u_{00}$ $dv/dy = v_{01}-v_{00}$

$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2+\left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2+\left(\frac{dv}{dy}\right)^2}\right)$$

$$\textit{mip-map } d = \log_2 L$$

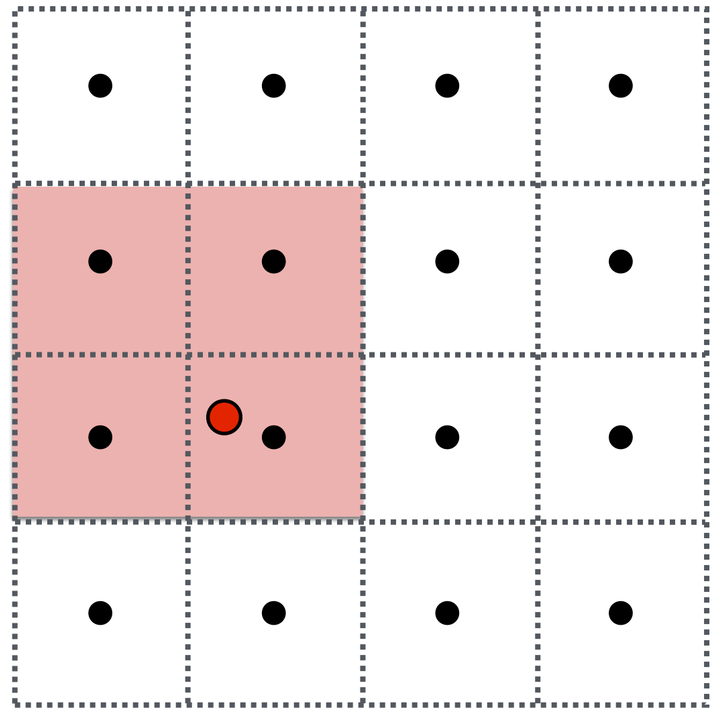
“Tri-linear” filtering



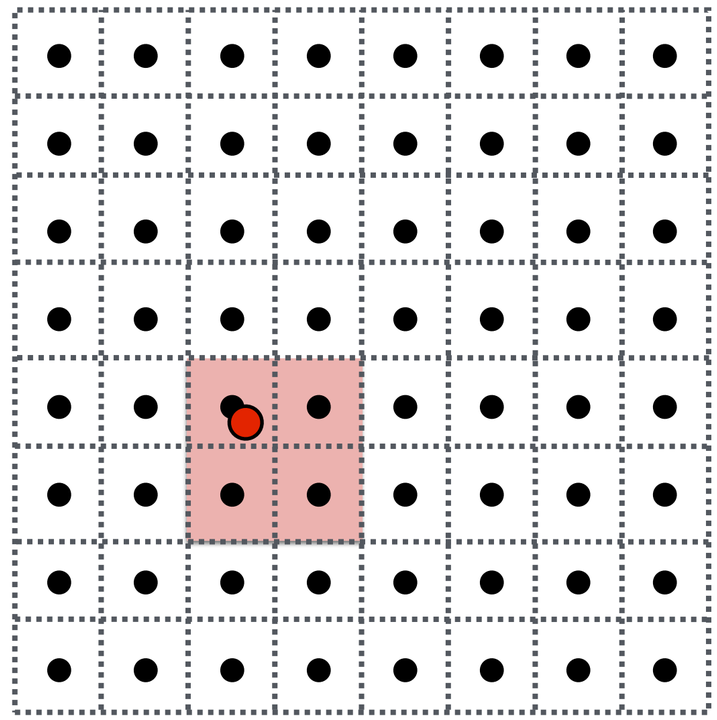
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

Bilinear resampling:
four texel reads
3 lerps (3 mul + 6 add)

Trilinear resampling:
eight telex reads
7 lerps (7 mul + 14 add)



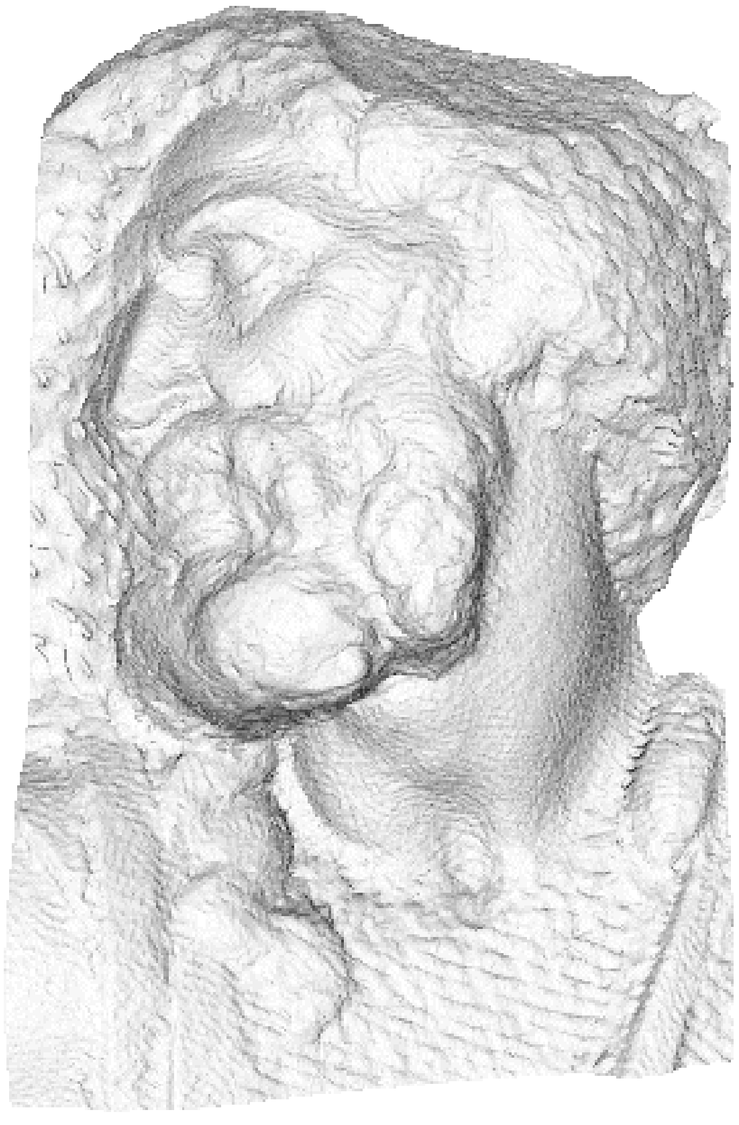
mip-map texels: level d+1



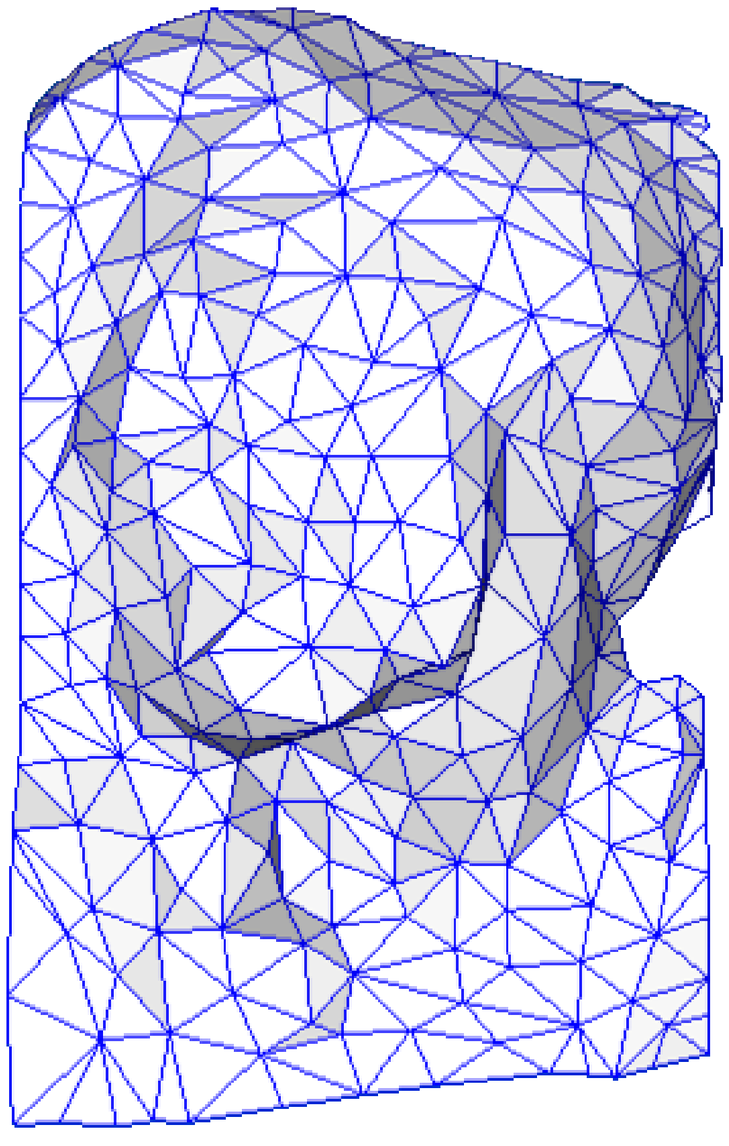
mip-map texels: level d

Figure credit: Akeley and Hanrahan

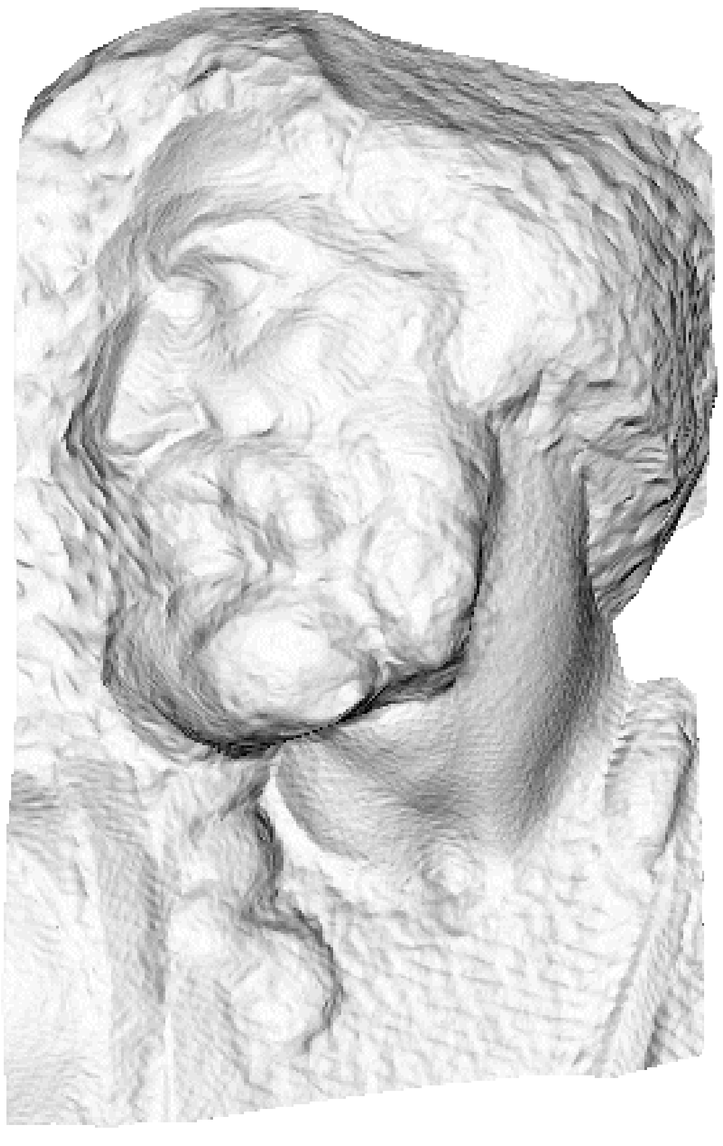
Texture Mapping: can be used to add other types of high-frequency details



original mesh
4M triangles

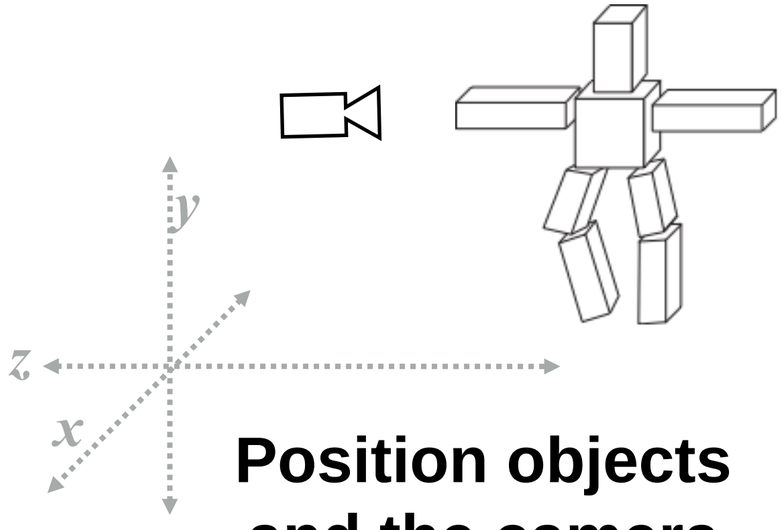


simplified mesh
500 triangles

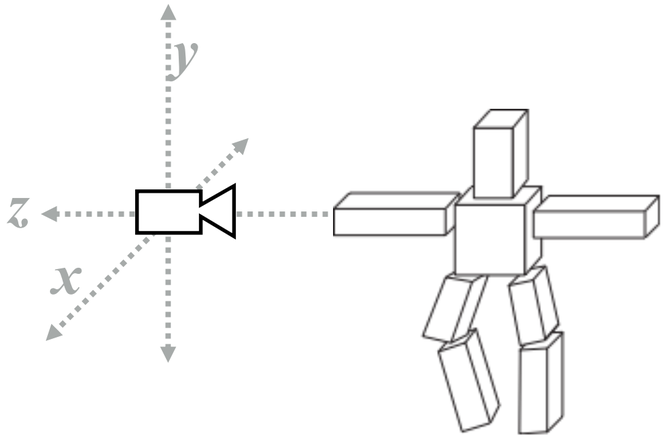


simplified mesh
and normal mapping
500 triangles

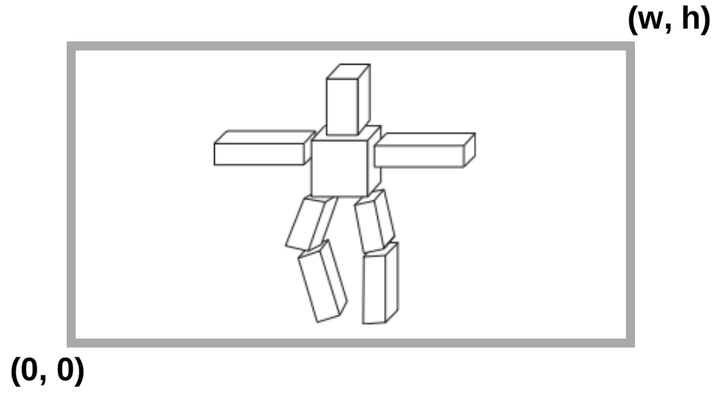
What you know how to do at this point in the course



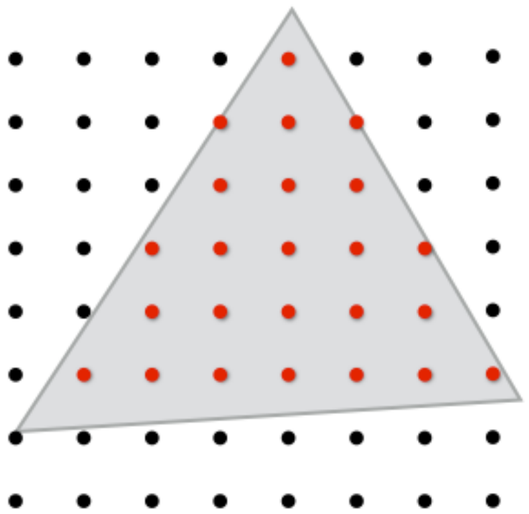
Position objects and the camera in the world



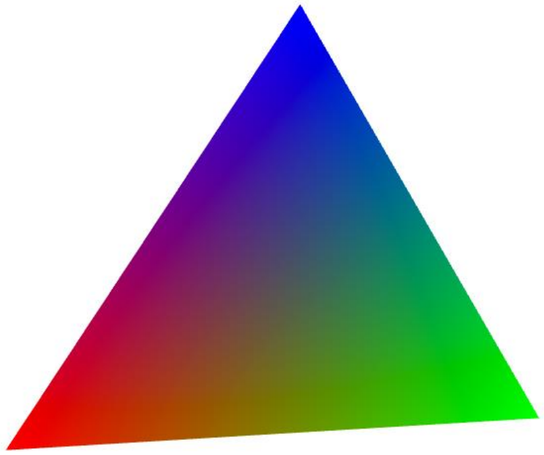
Determine the position of objects relative to the camera



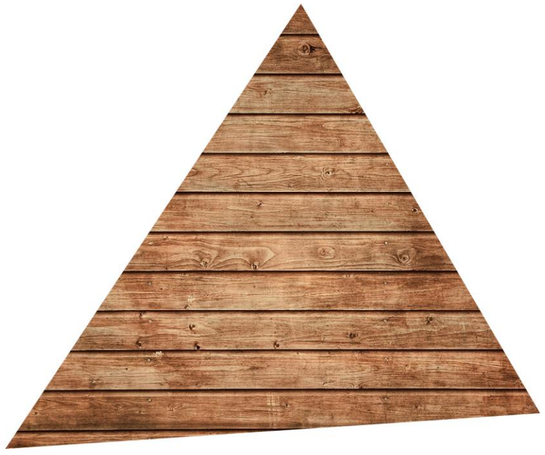
Project objects onto the screen



Sample triangle coverage



Compute triangle attribute values at covered sample points



Sample texture maps

What else do you need to know to render a picture like this?

Surface representation

How to represent/manipulate complex surfaces?

Occlusion

Which surface is visible to the camera at each sample point?

Lighting/materials

How to describe the way light interacts with objects in a scene?

Course roadmap

Drawing Things

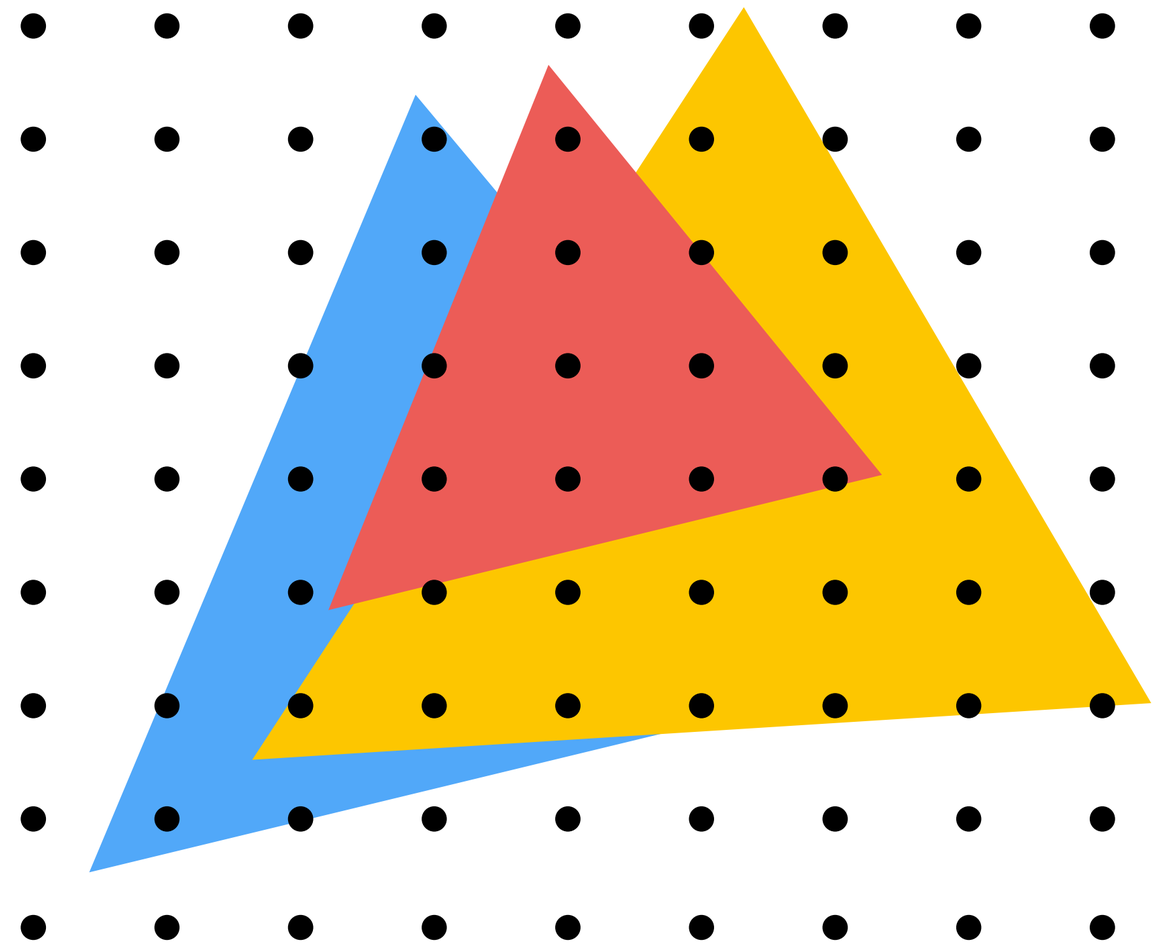
- Introduction
- Drawing a triangle (by sampling)
- Transforms and coordinate spaces
- Perspective projection and texture sampling
- **Today: putting it all together: end-to-end rasterization pipeline**

Geometry

Materials and Lighting

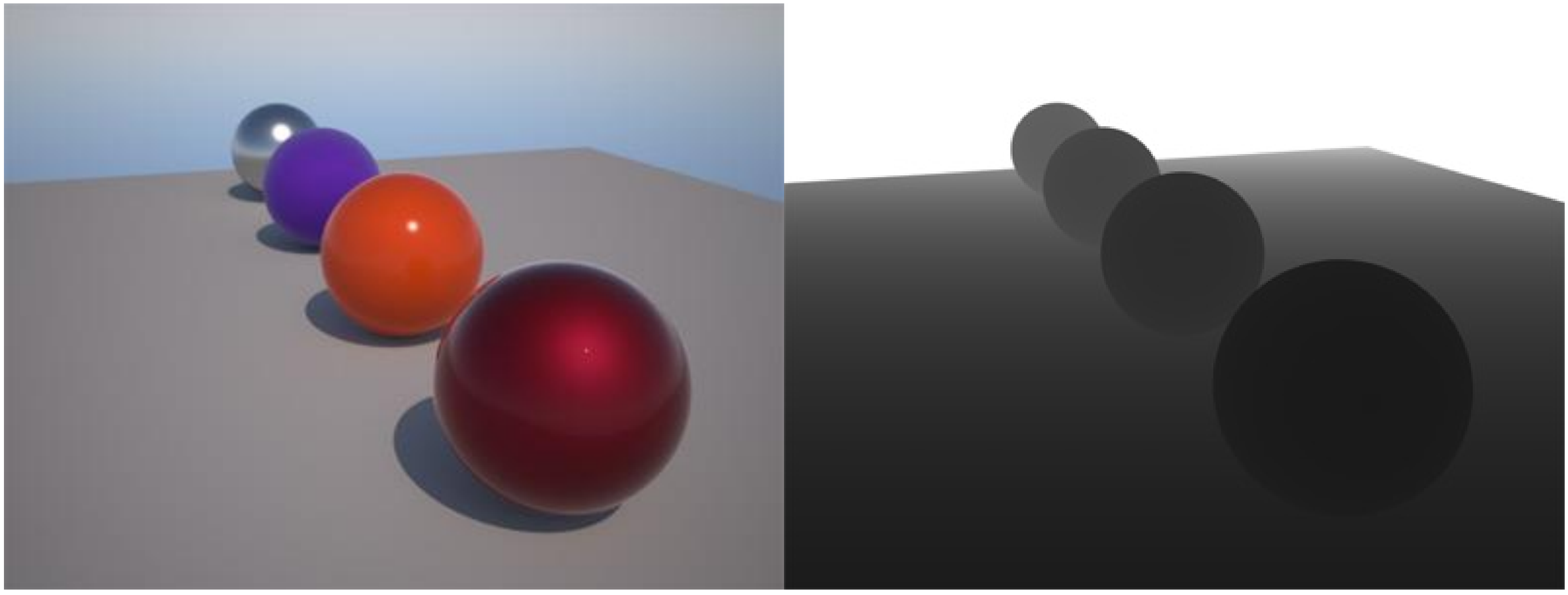
Occlusion

Which triangle is visible at each pixel?



Opaque Triangles

The depth buffer (Z-buffer)



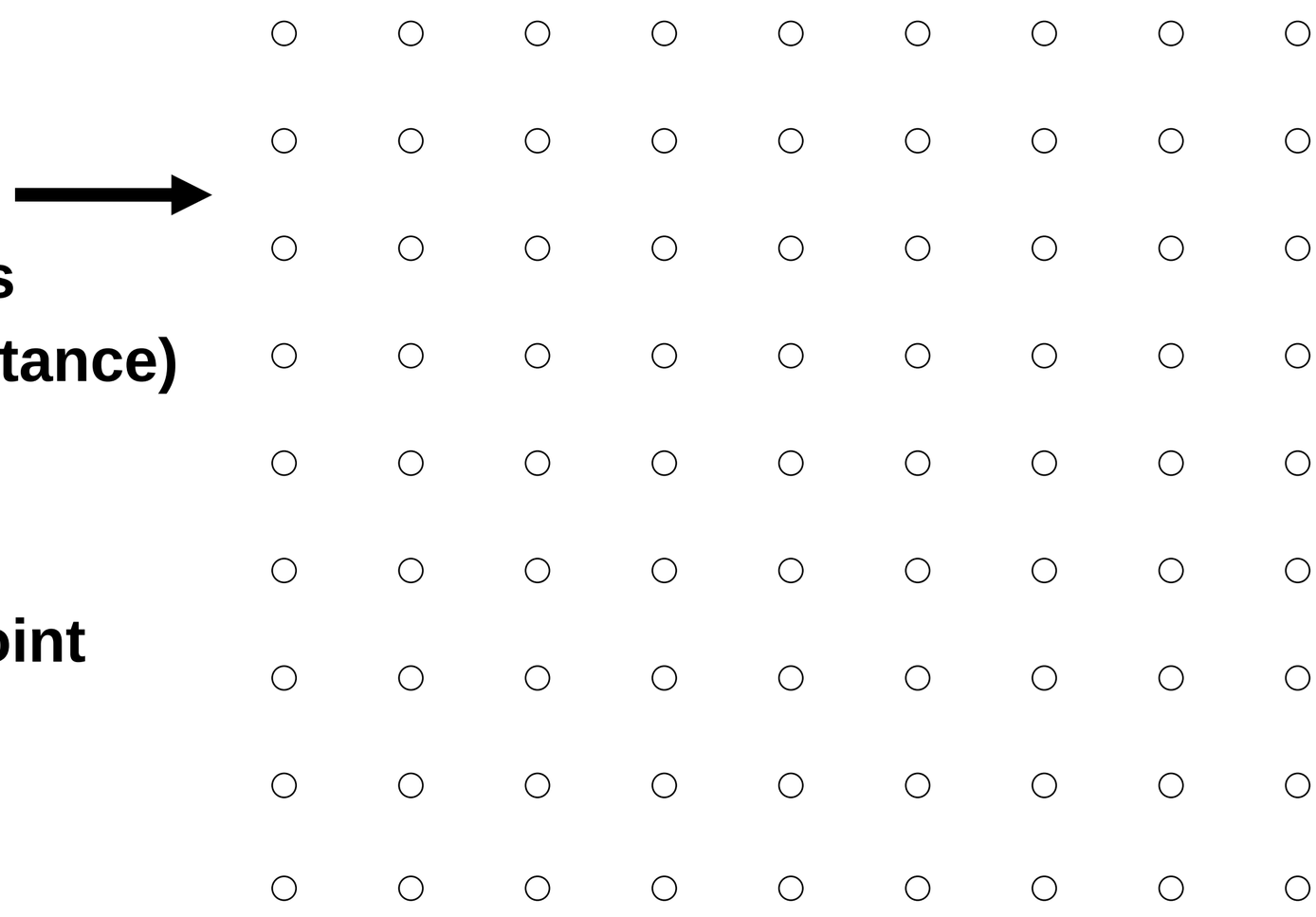
Q: How do we compute the depth of sampled points on a triangle?

Interpolate it just like any other attribute that varies linearly over the surface of the triangle.

Occlusion using the depth-buffer (Z-buffer)

For each coverage sample point, depth-buffer stores depth of closest triangle at this sample point that has been processed by the renderer so far.

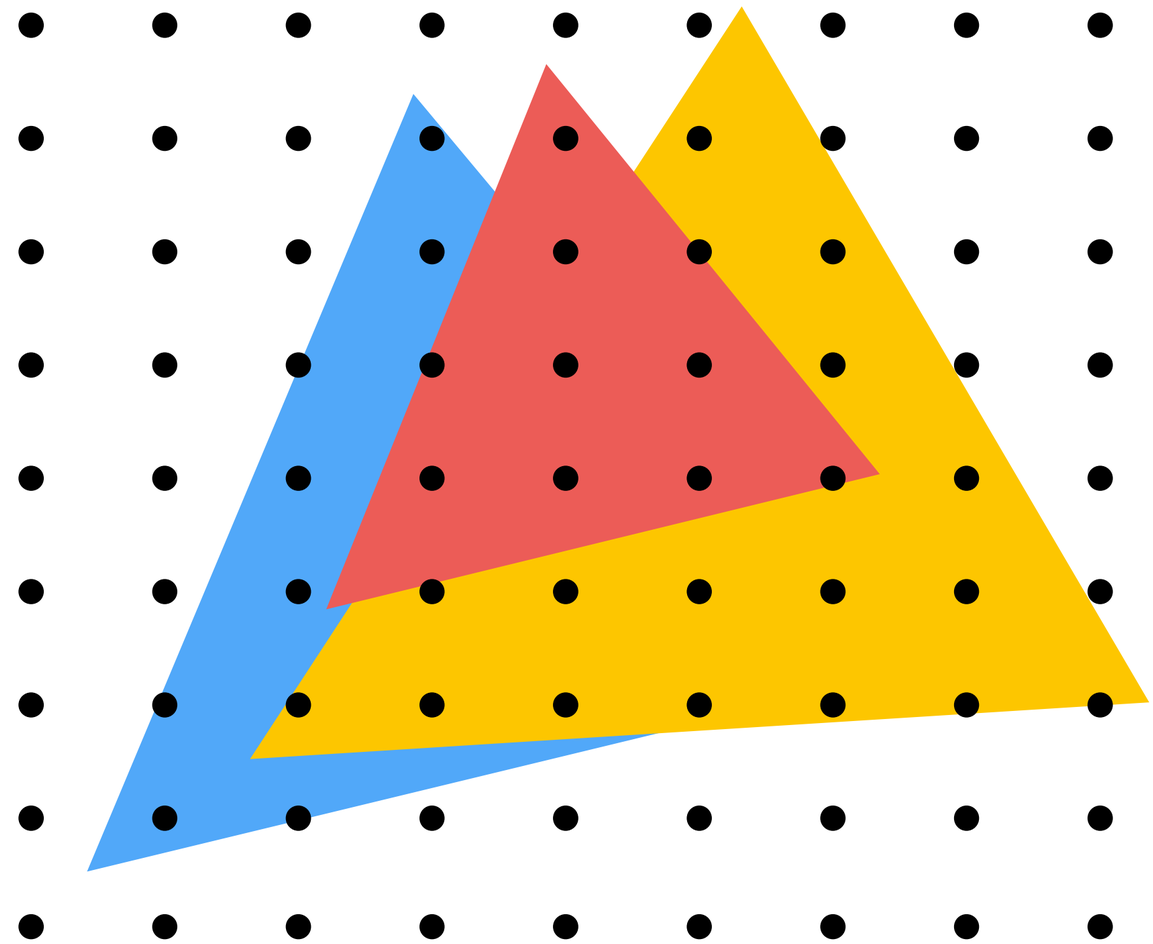
Initial state of depth buffer before rendering any triangles (all samples store farthest distance)



Grayscale value of sample point used to indicate distance

- **Black = small distance**
- **White = large distance**

Example: rendering three opaque triangles

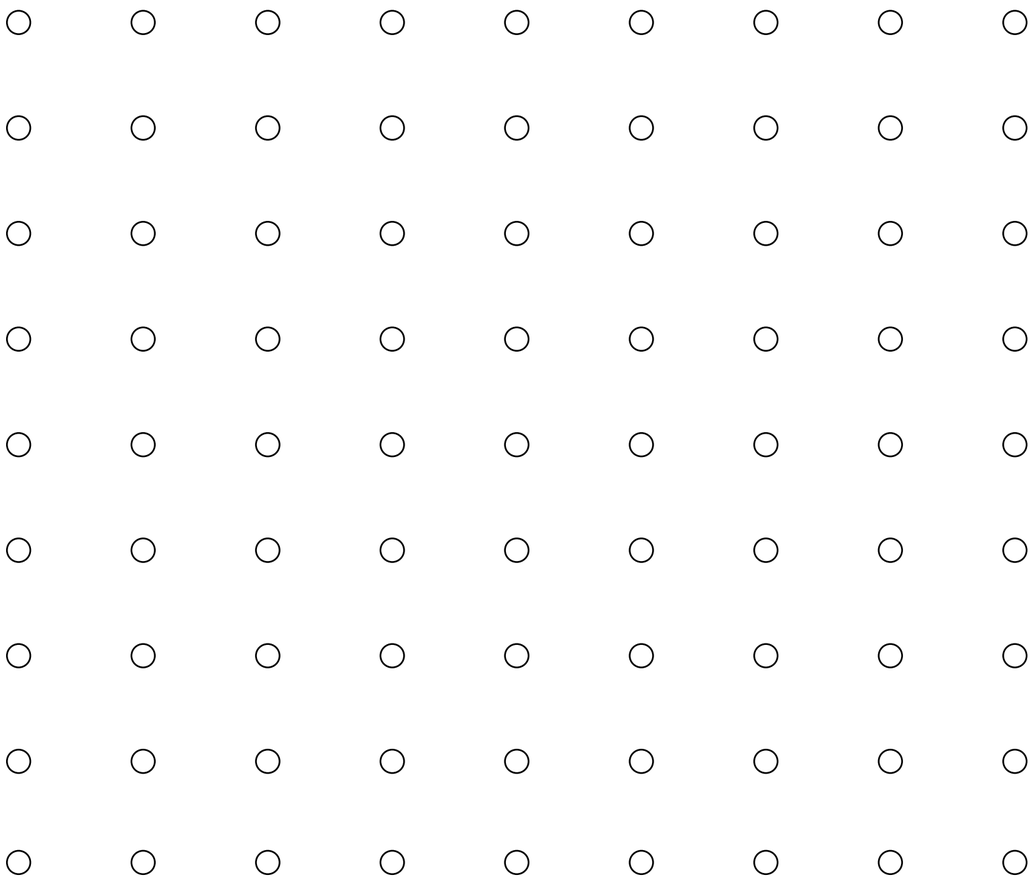


Occlusion using the depth-buffer (Z-buffer)

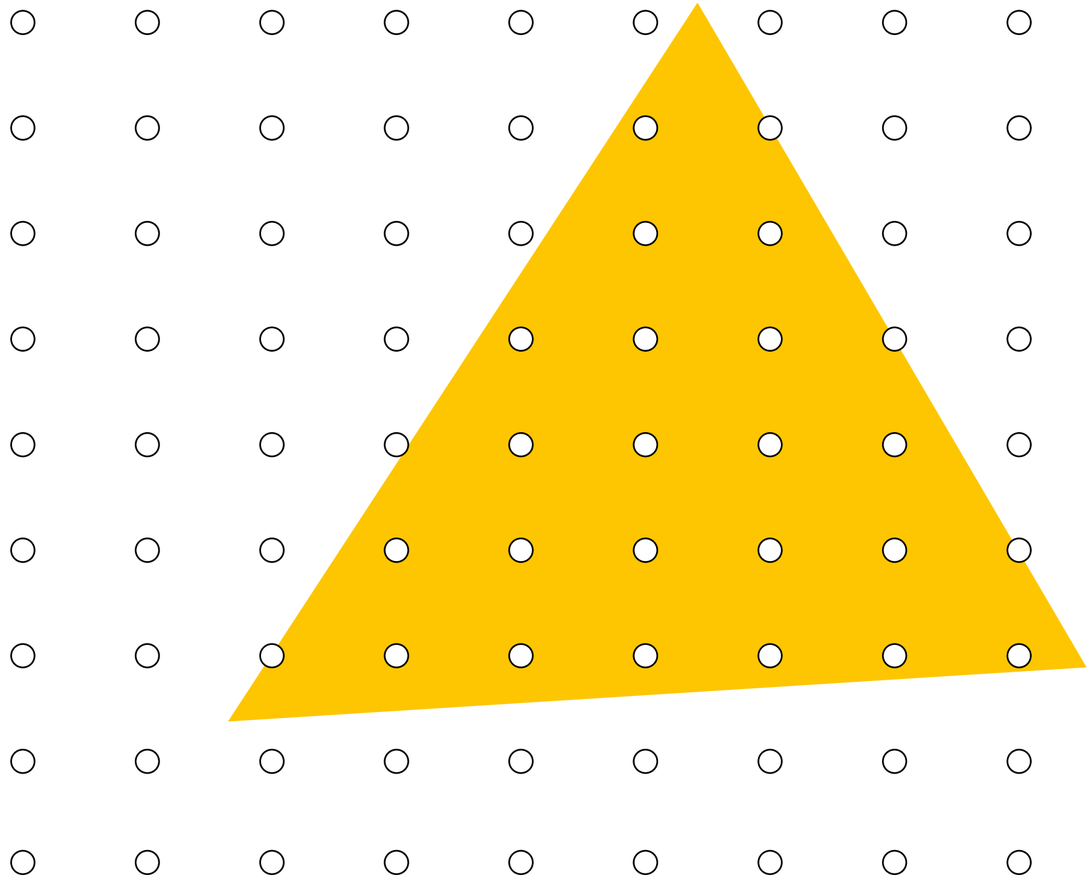
Processing yellow triangle:
depth = 0.5

Grayscale value of sample point used to indicate distance

White = large distance
Black = small distance
Red = sample passed depth test



Color buffer contents



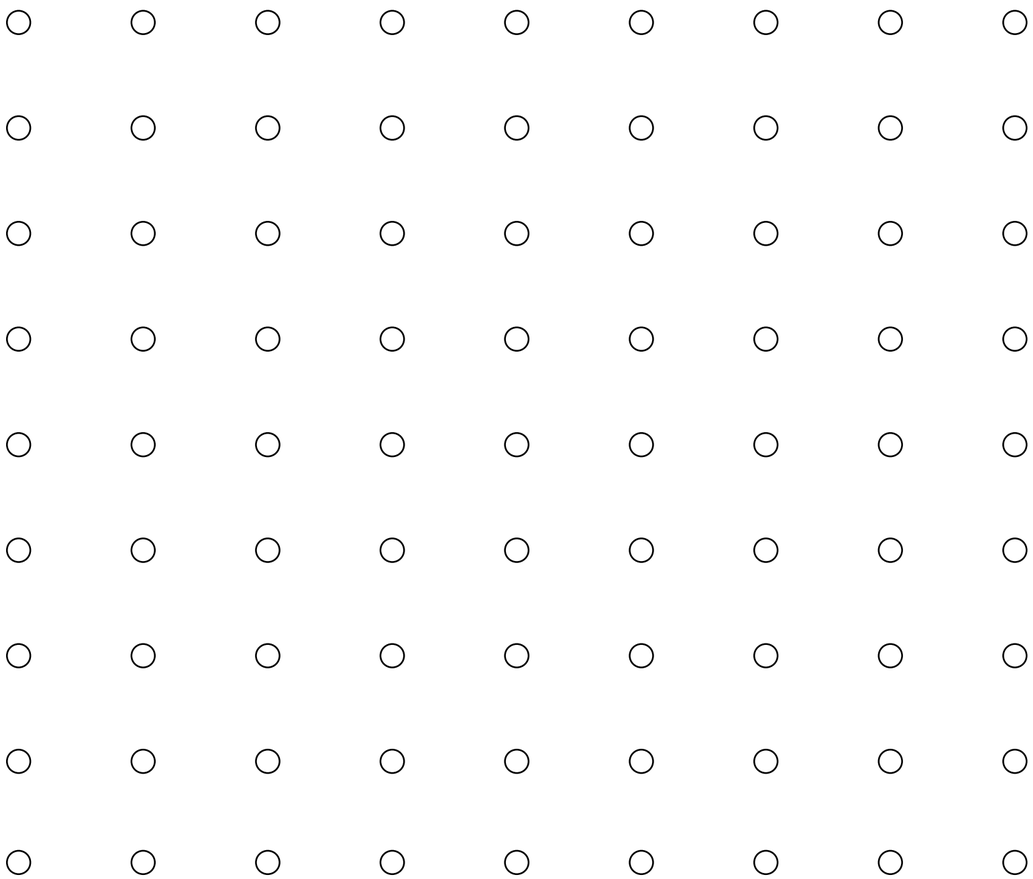
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

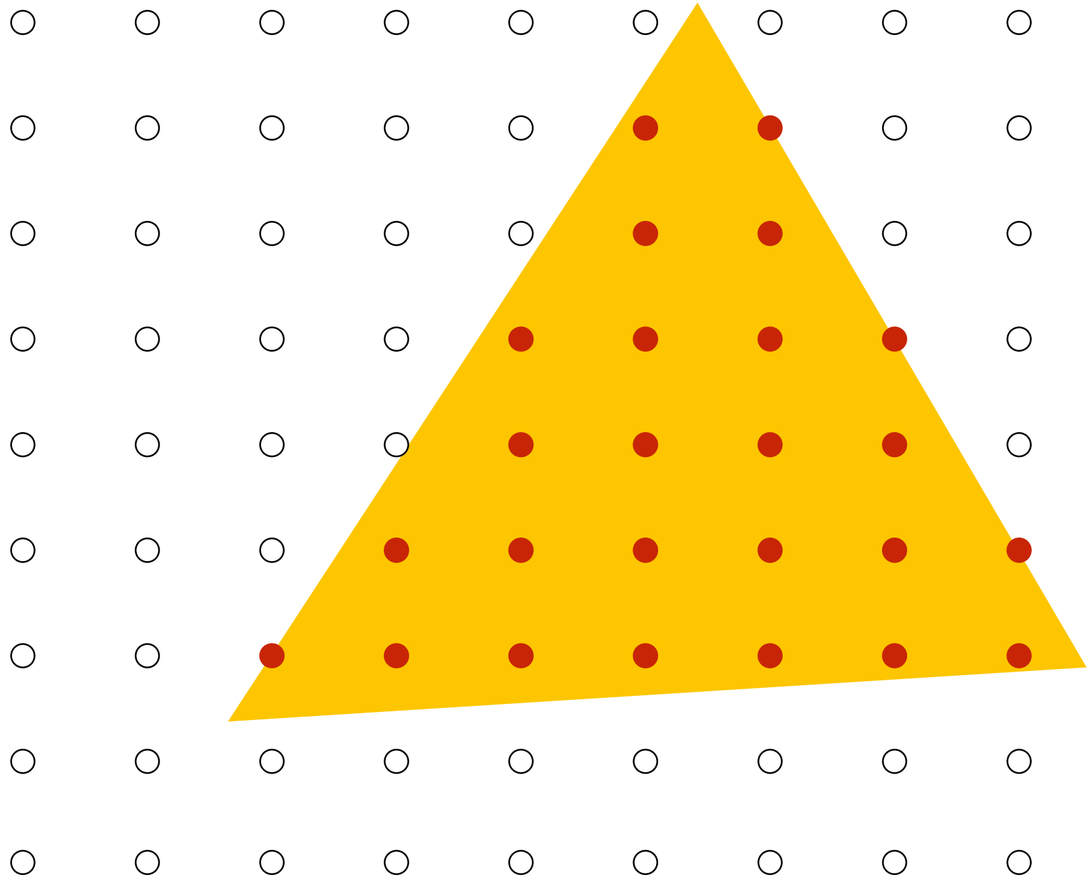
Processing yellow triangle:
depth = 0.5

Grayscale value of sample point used to indicate distance

White = large distance
Black = small distance
Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

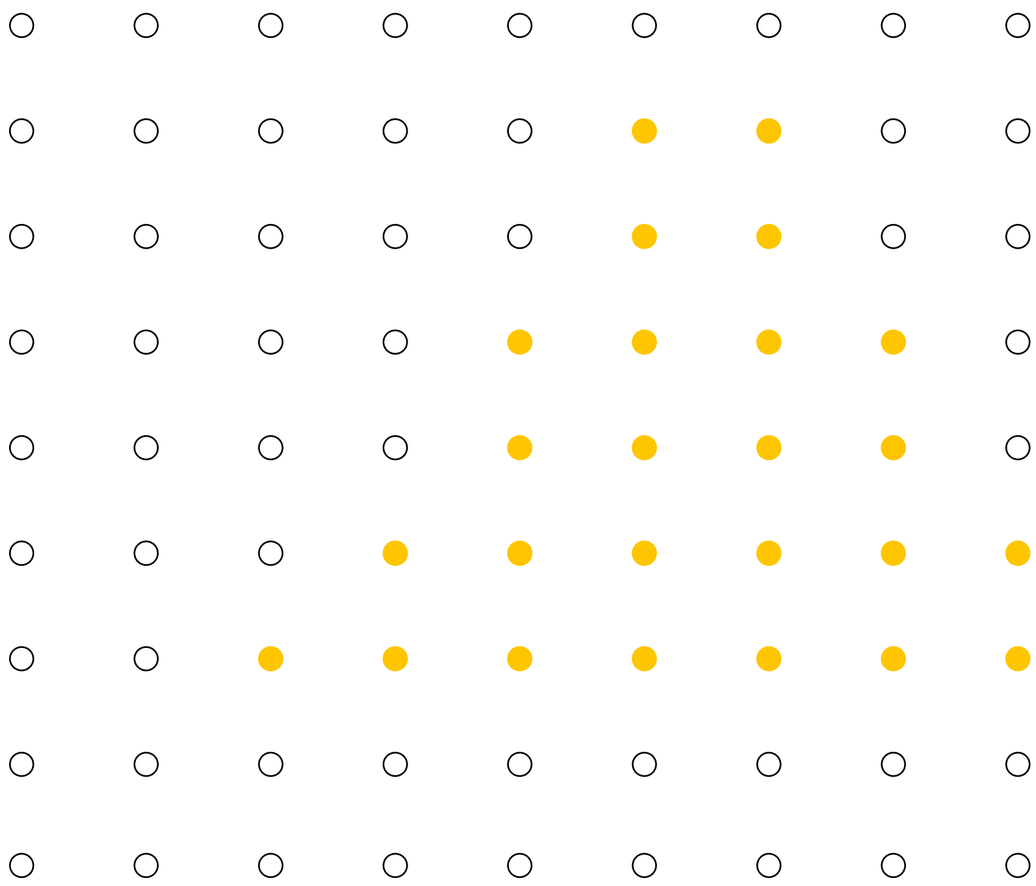
After processing yellow triangle:

Grayscale value of sample point used to indicate distance

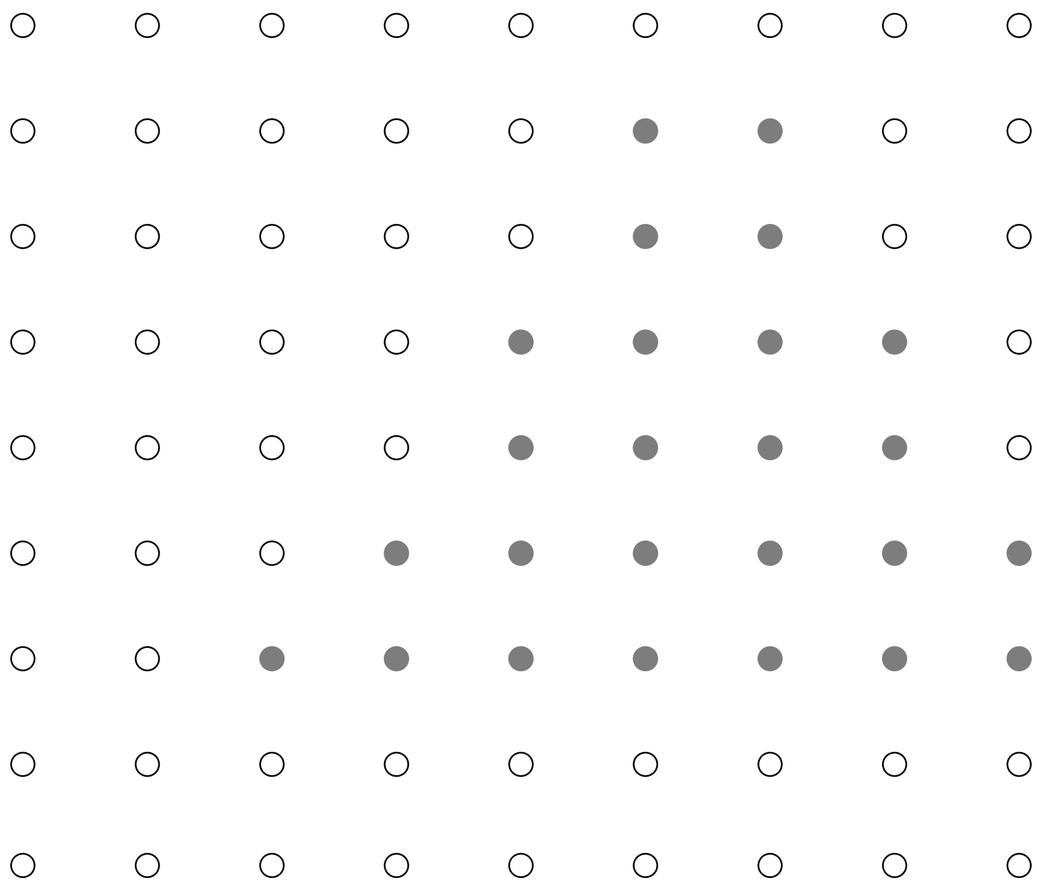
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

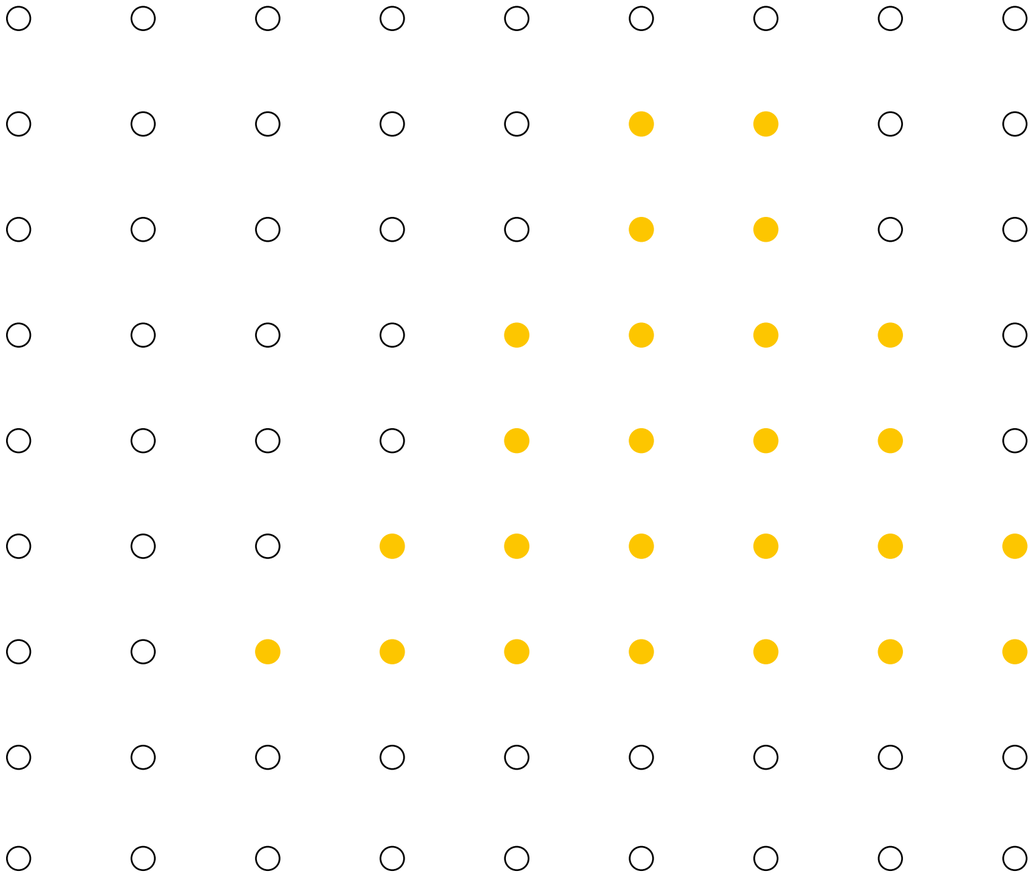
depth = 0.75

Grayscale value of sample point used to indicate distance

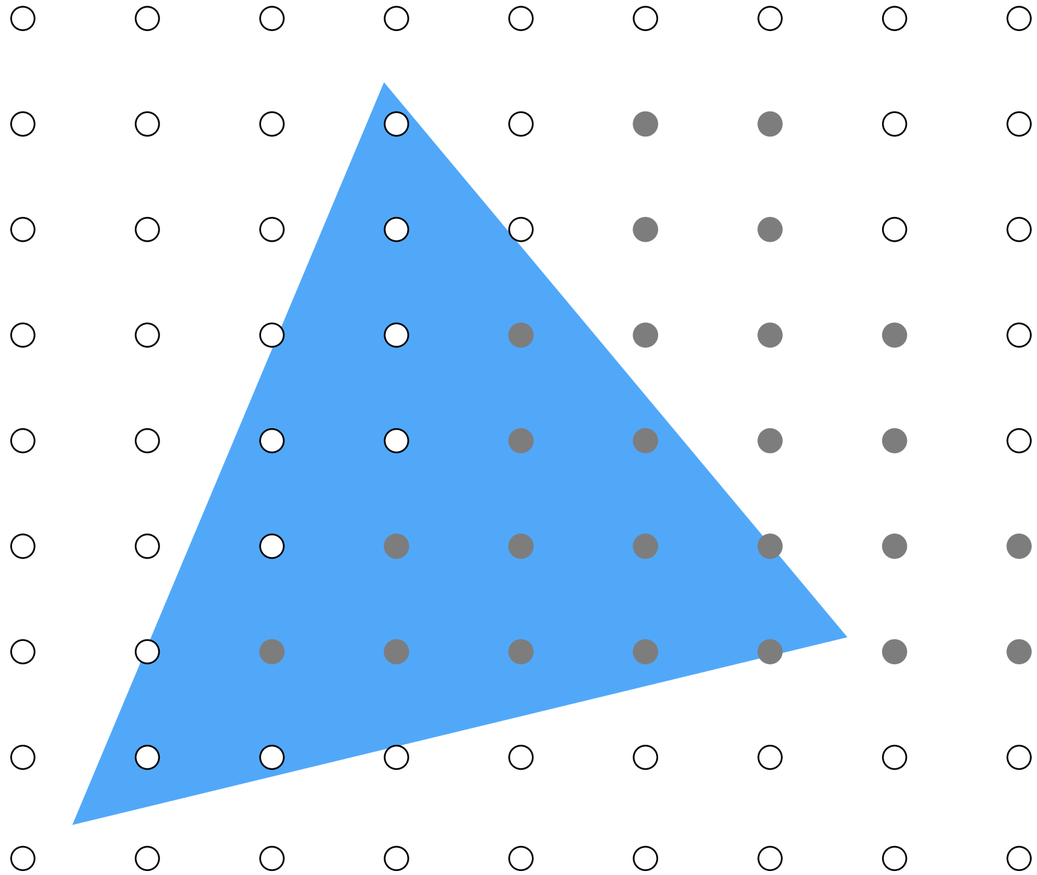
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

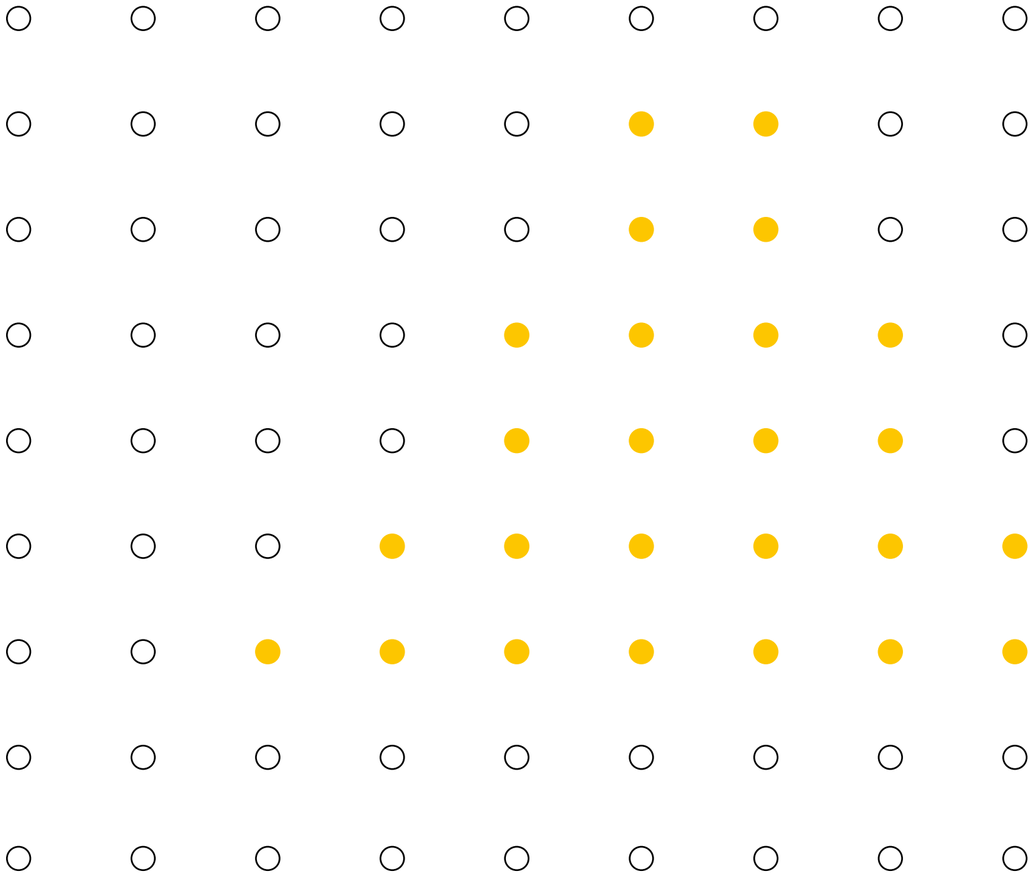
depth = 0.75

Grayscale value of sample point used to indicate distance

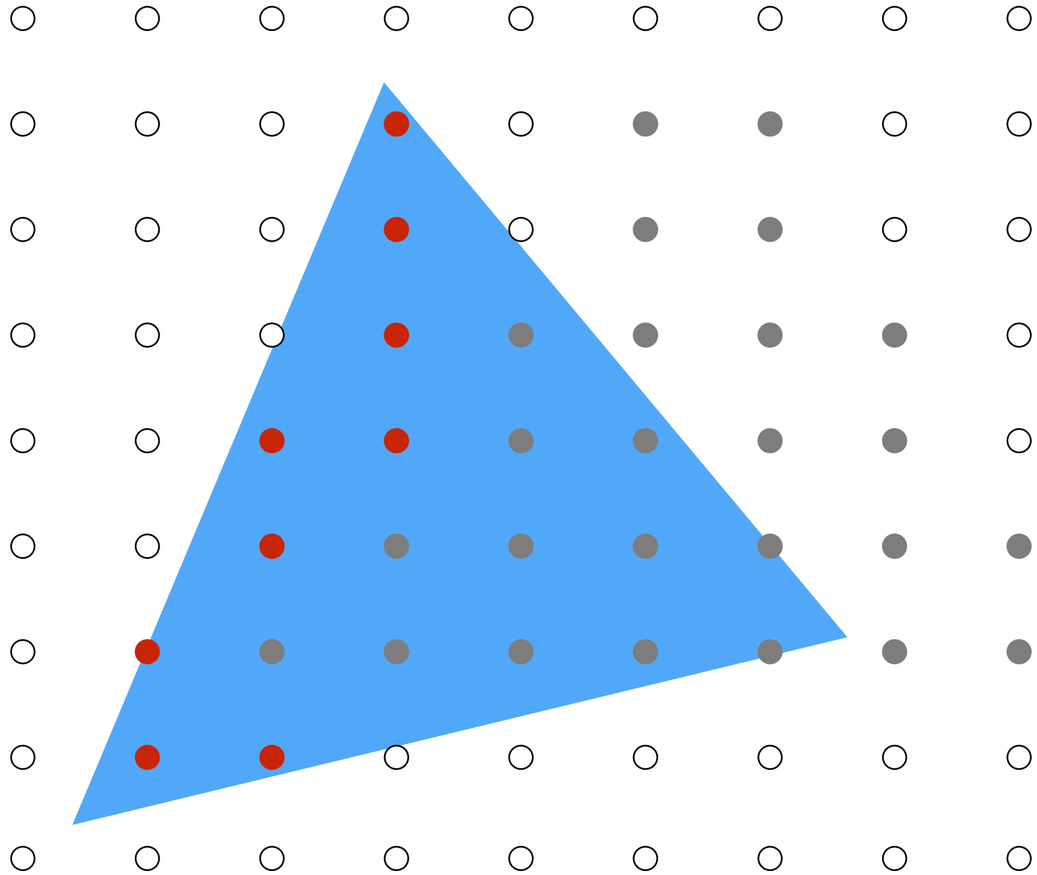
White = large distance

Black = small distance

Red = sample passed depth test



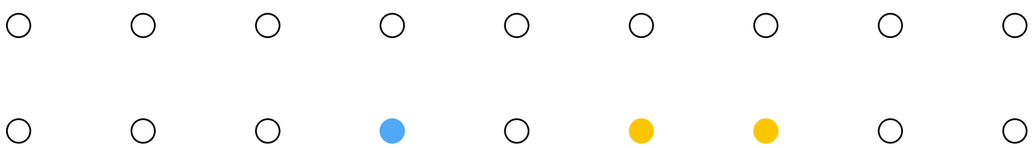
Color buffer contents

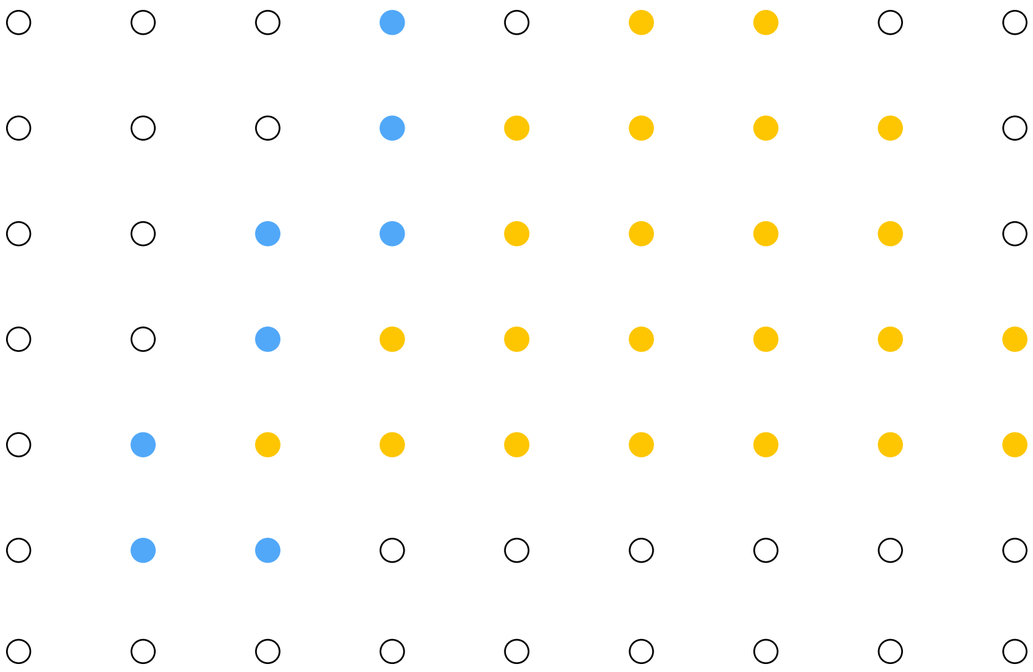


Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:





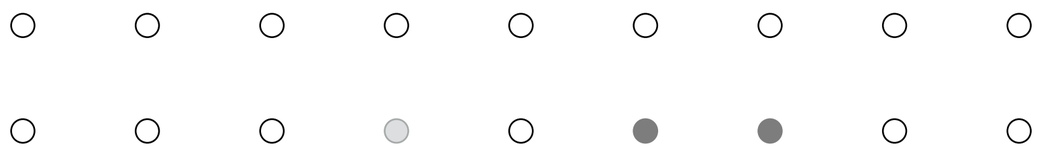
Color buffer contents

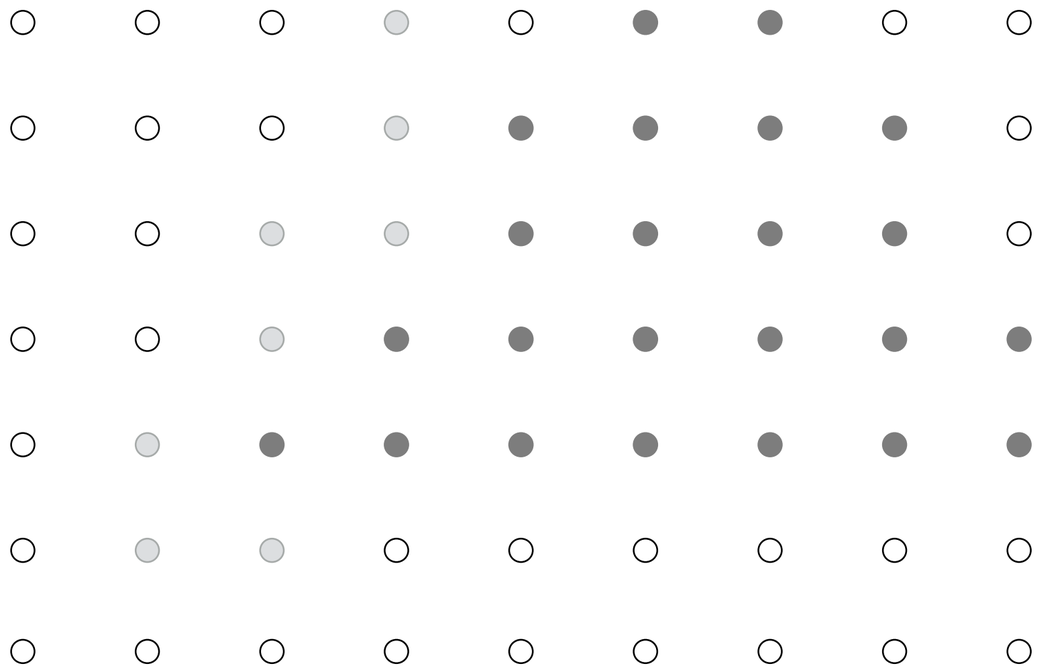
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test





Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

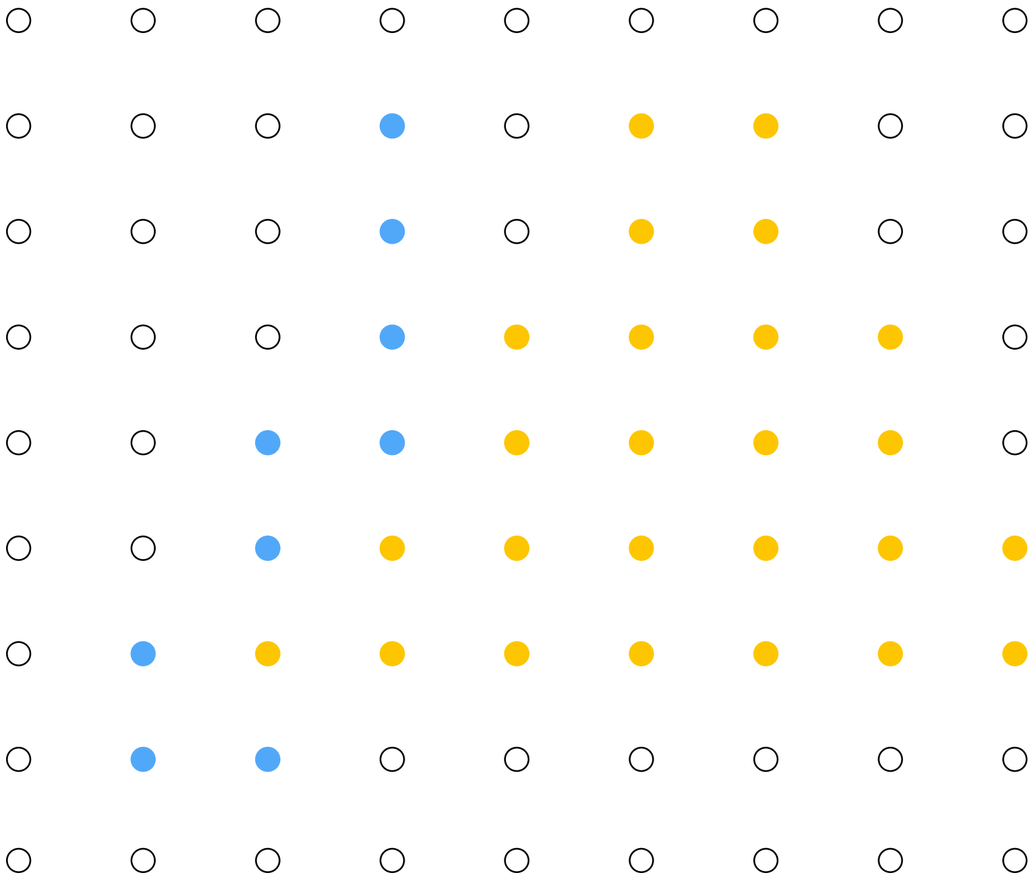
depth = 0.25

Grayscale value of sample point used to indicate distance

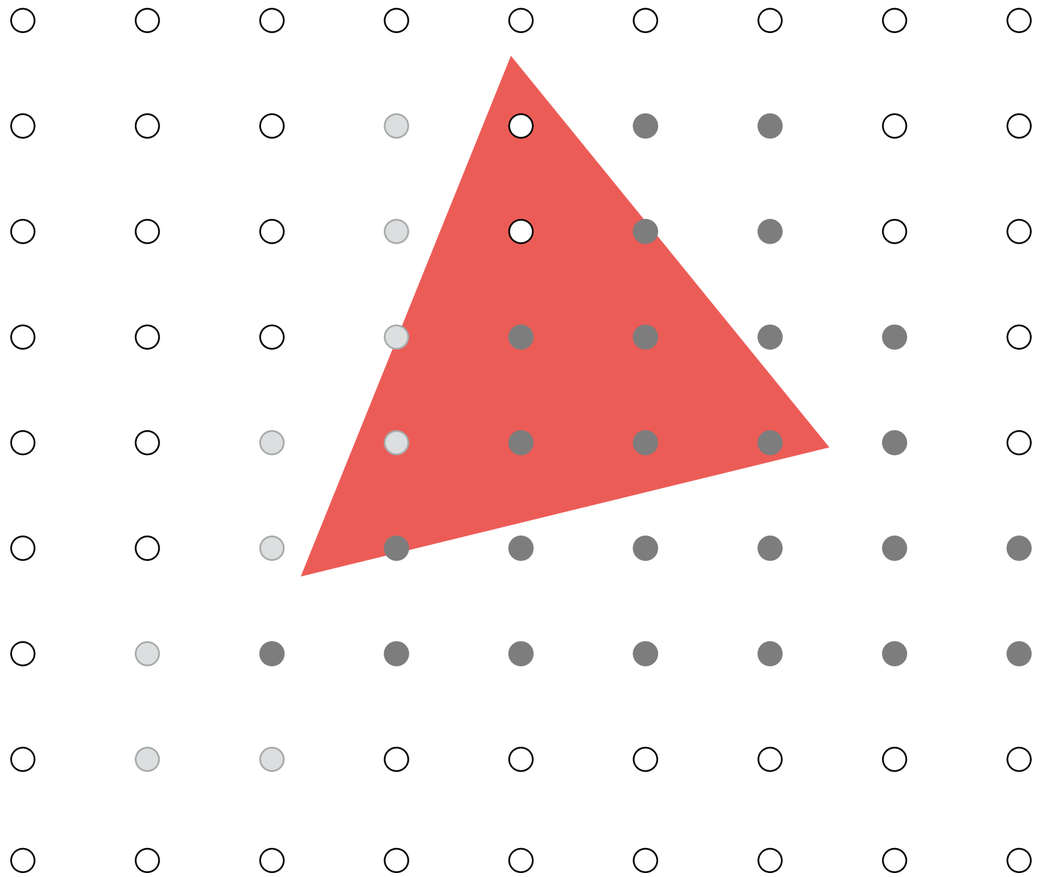
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

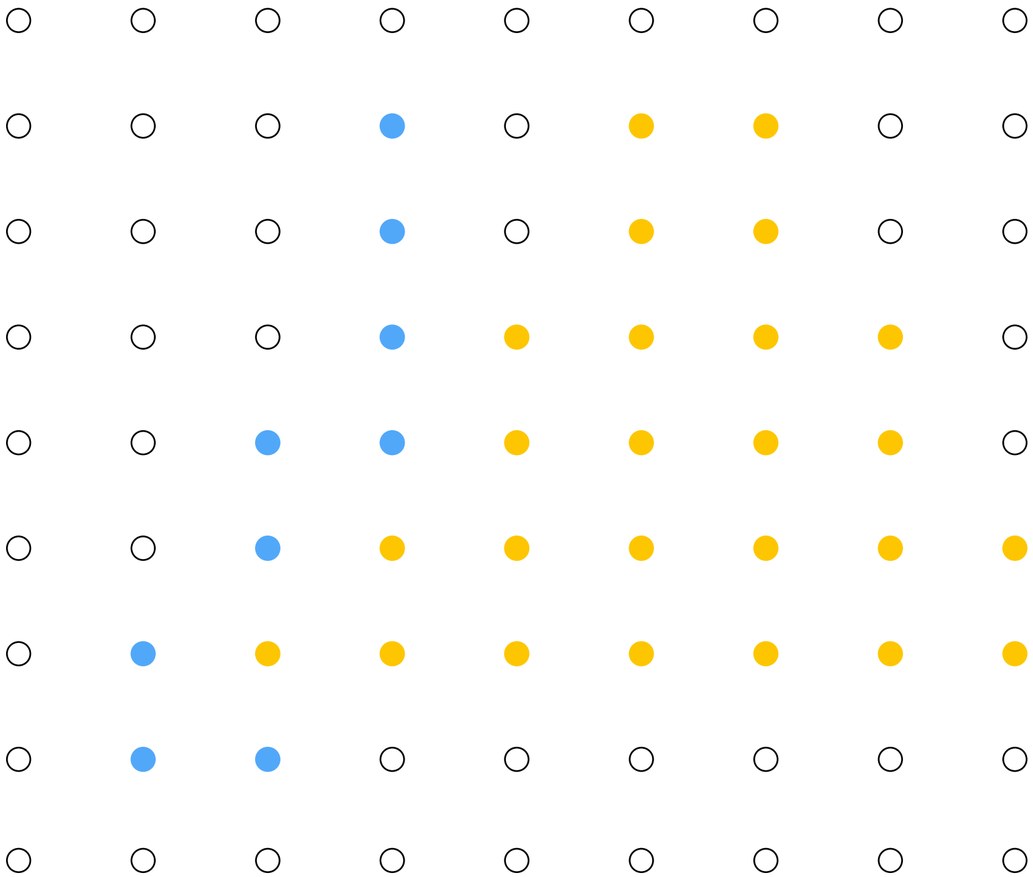
depth = 0.25

Grayscale value of sample point used to indicate distance

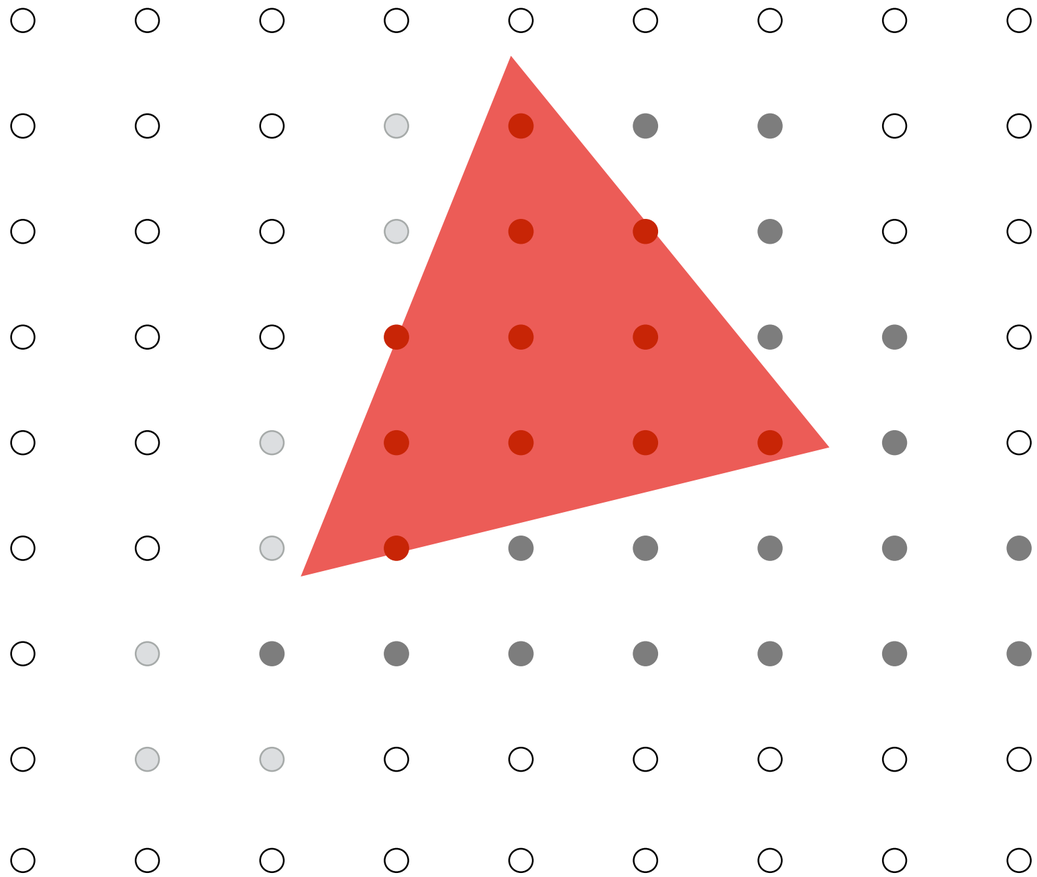
White = large distance

Black = small distance

Red = sample passed depth test



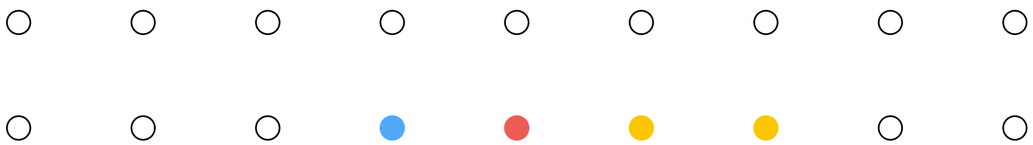
Color buffer contents

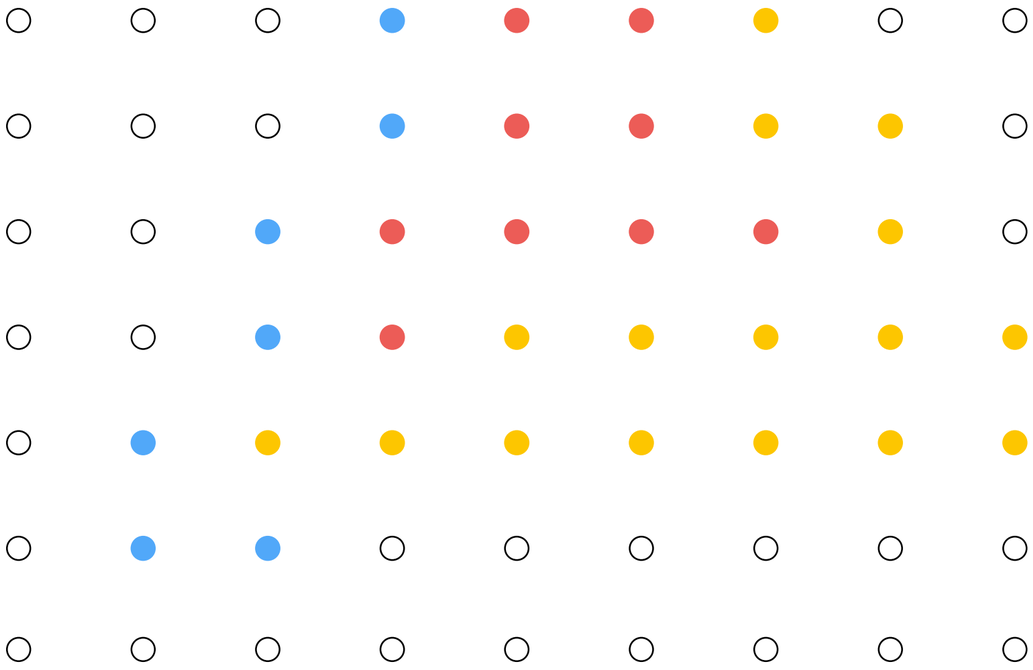


Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:





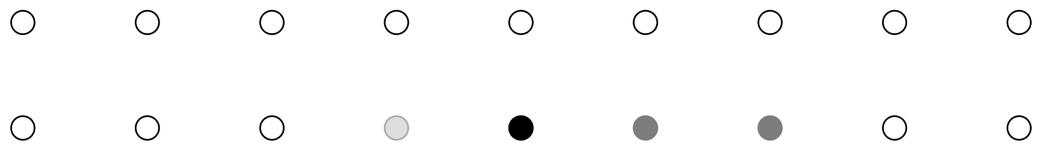
Color buffer contents

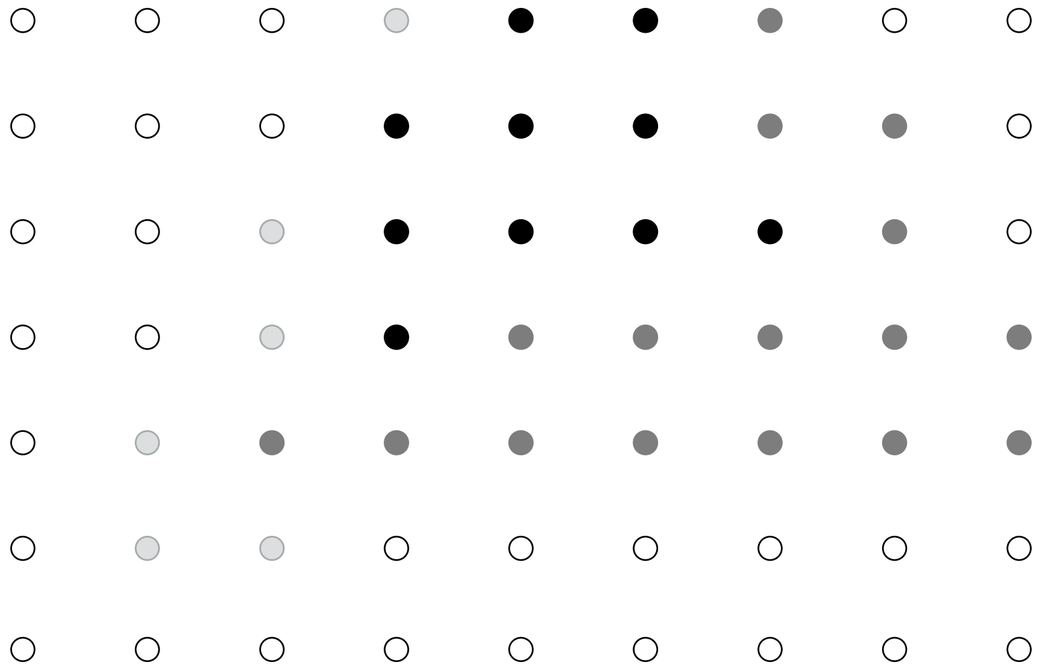
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

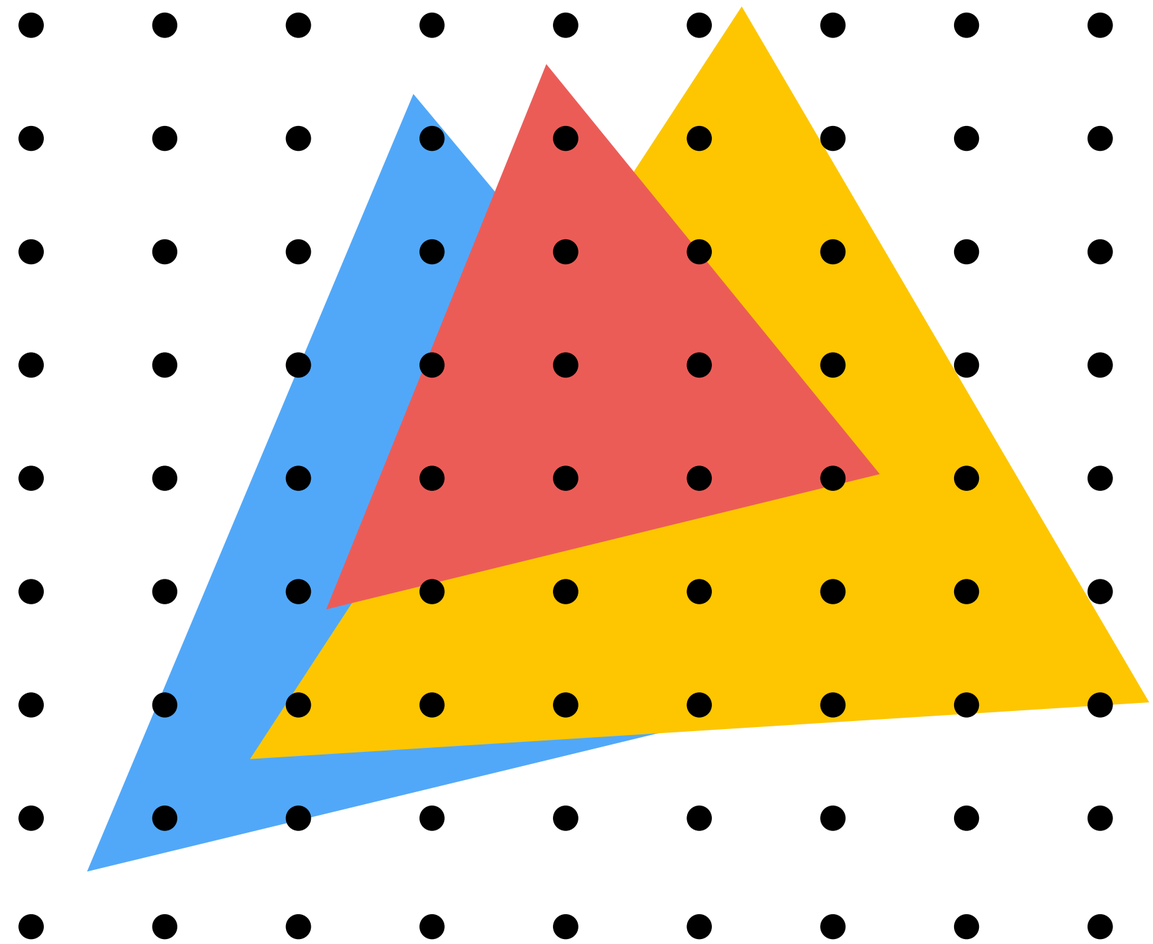
Red = sample passed depth test





Depth buffer contents

Example: rendering three opaque triangles

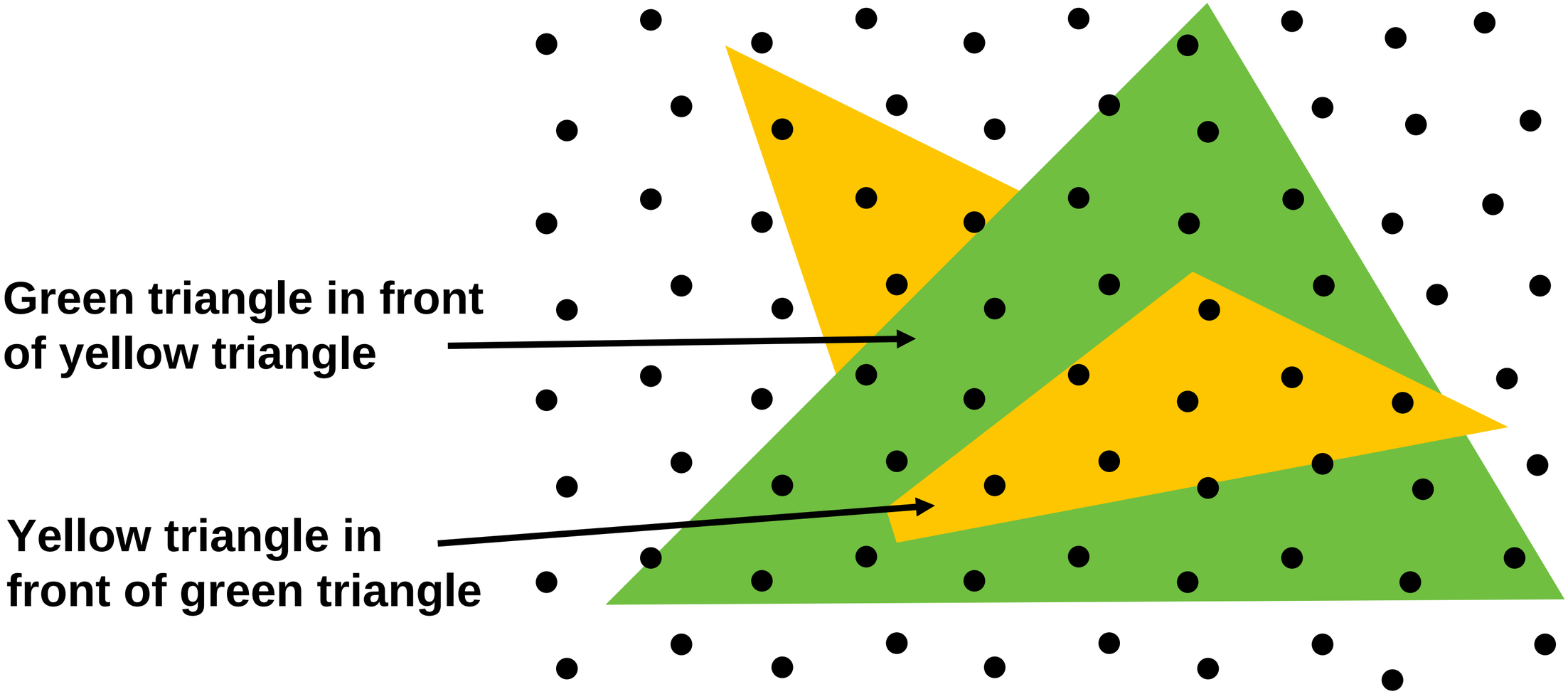


Does the depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles at a given sample point.

The relative depth of triangles may be different at different sample points.

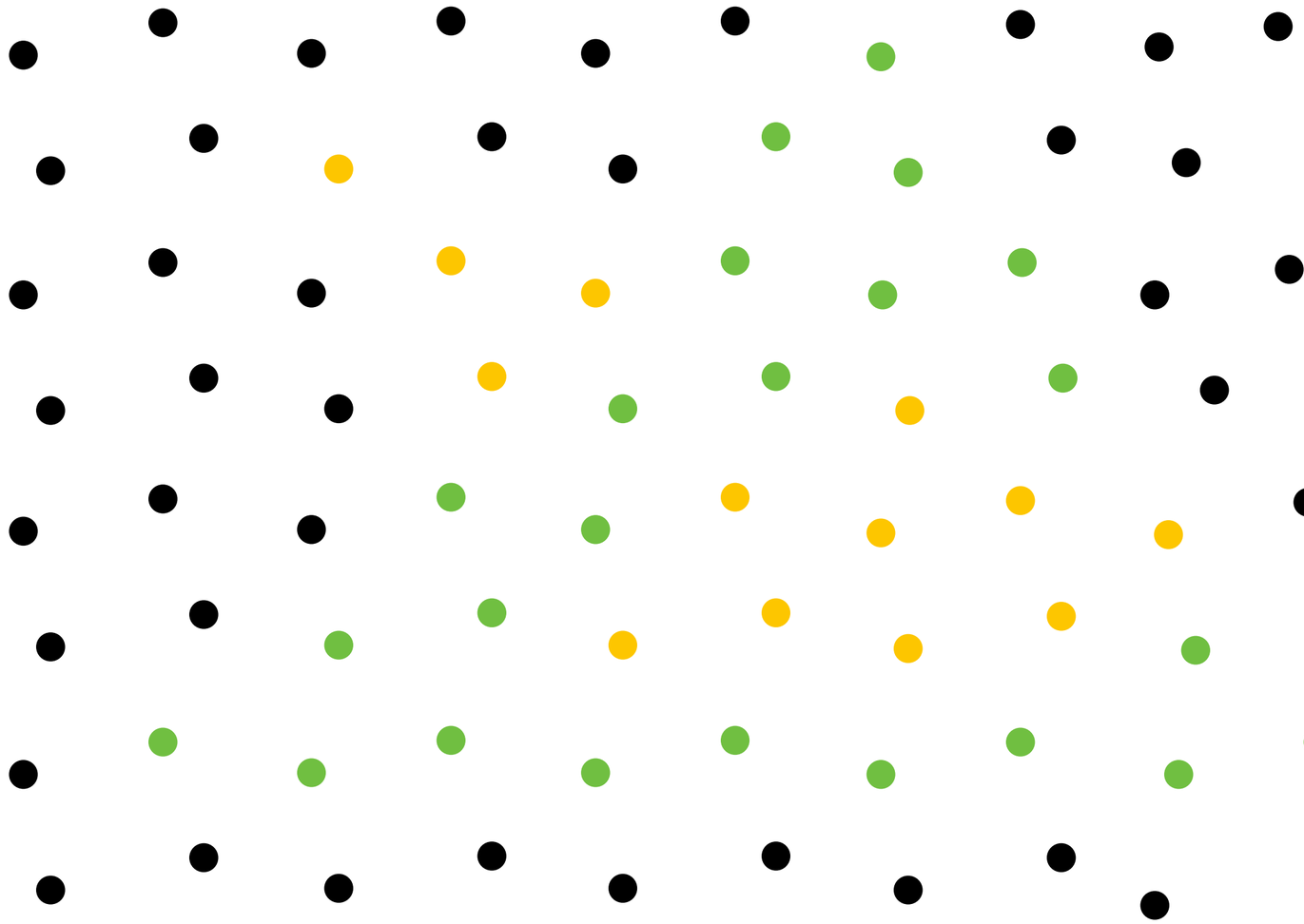


Does the depth-buffer algorithm handle interpenetrating surfaces?

Of course!

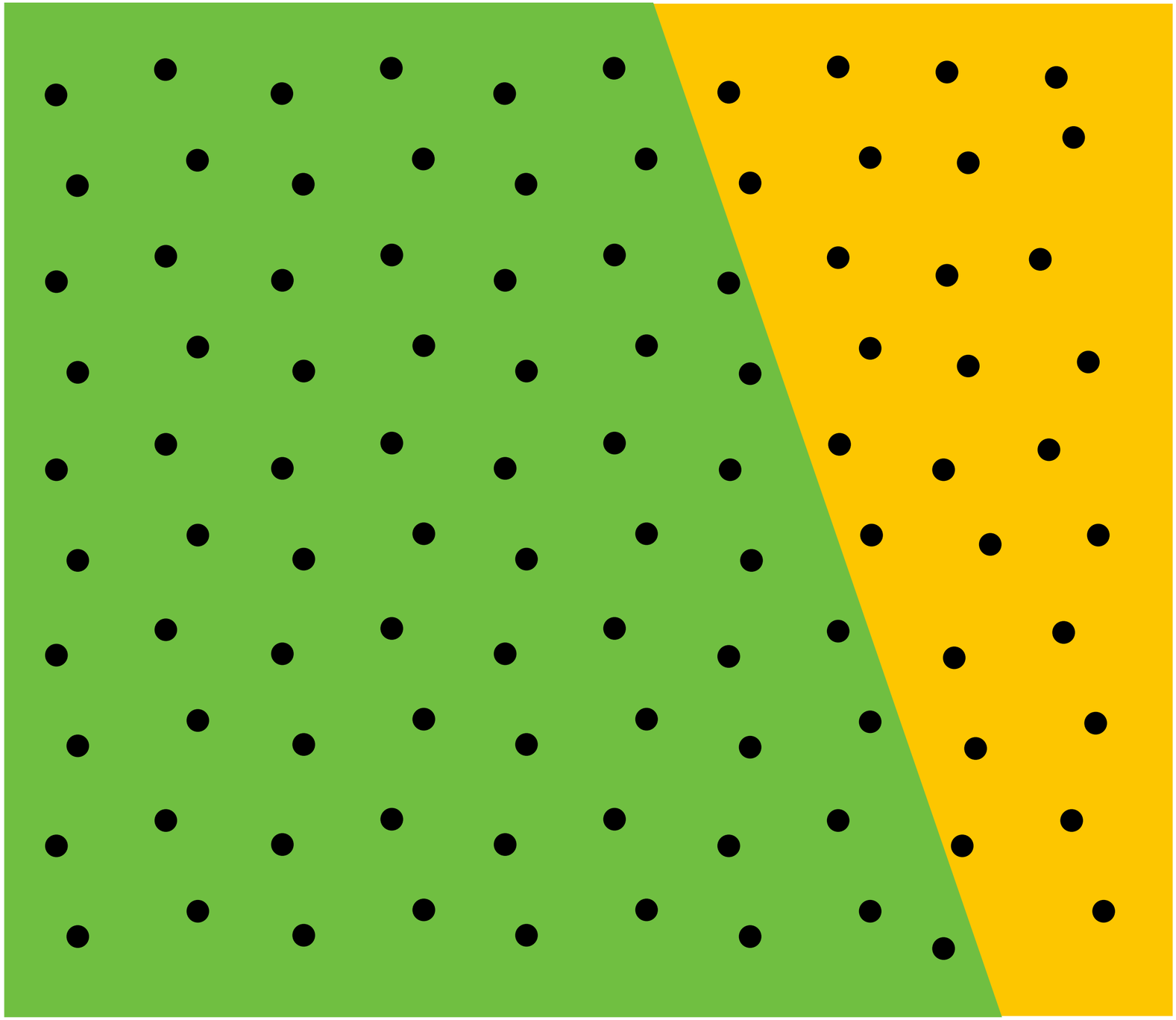
Occlusion test is based on depth of triangles at a given sample point.

The relative depth of triangles may be different at different sample points.



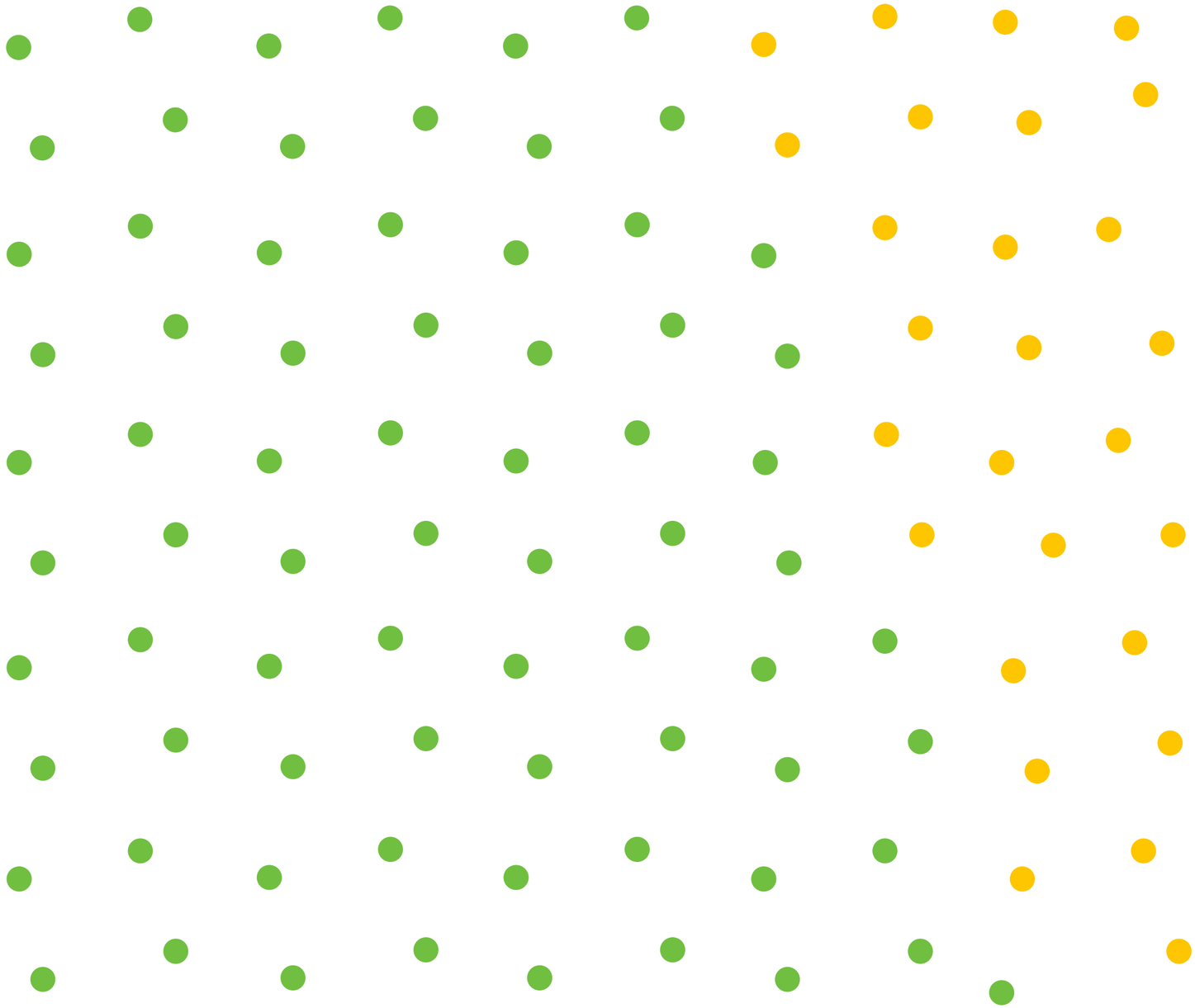
Does it work with super sampling?

Of course! Occlusion test is per sample, not per pixel!

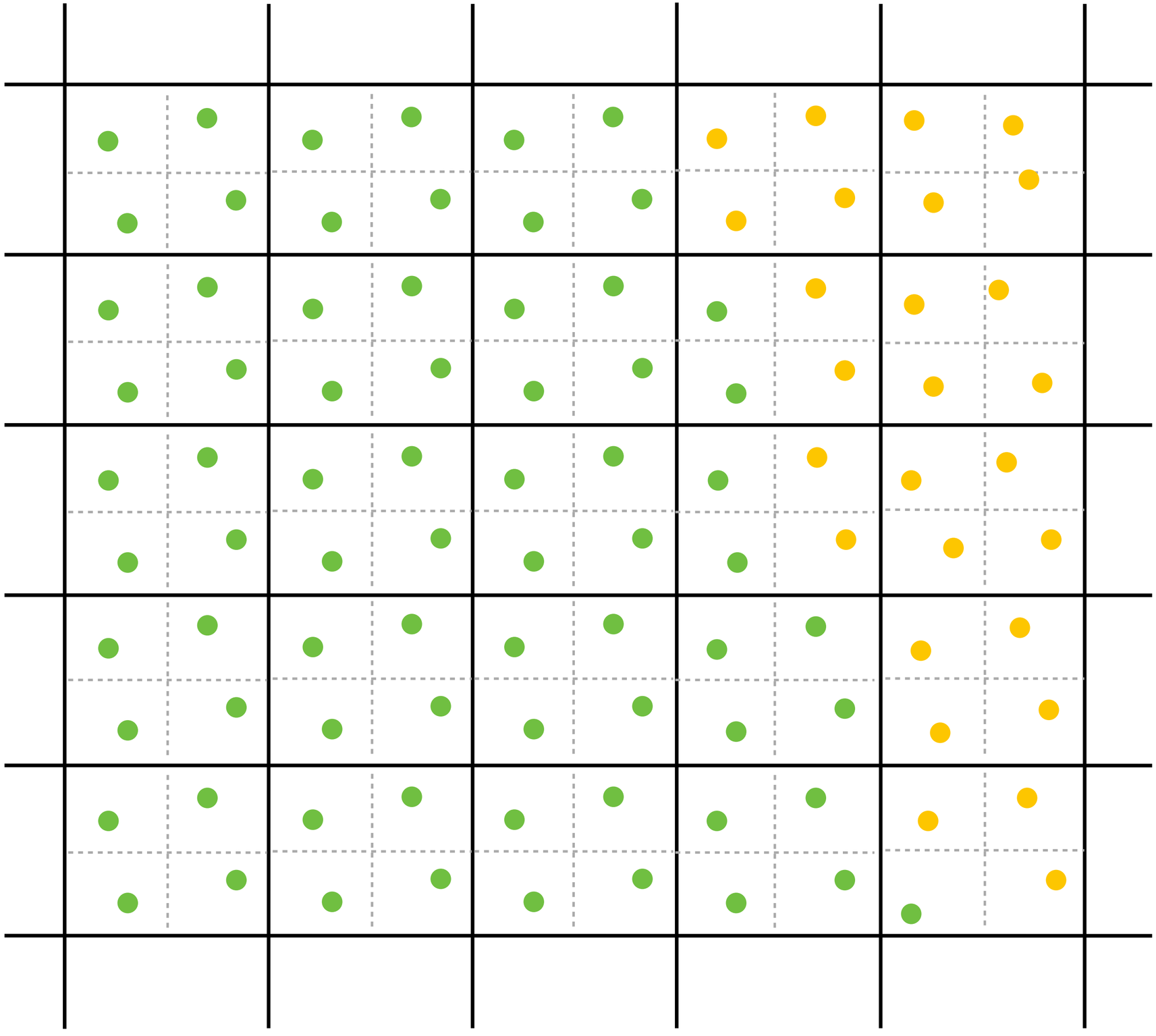


This example: green triangle occludes yellow triangle

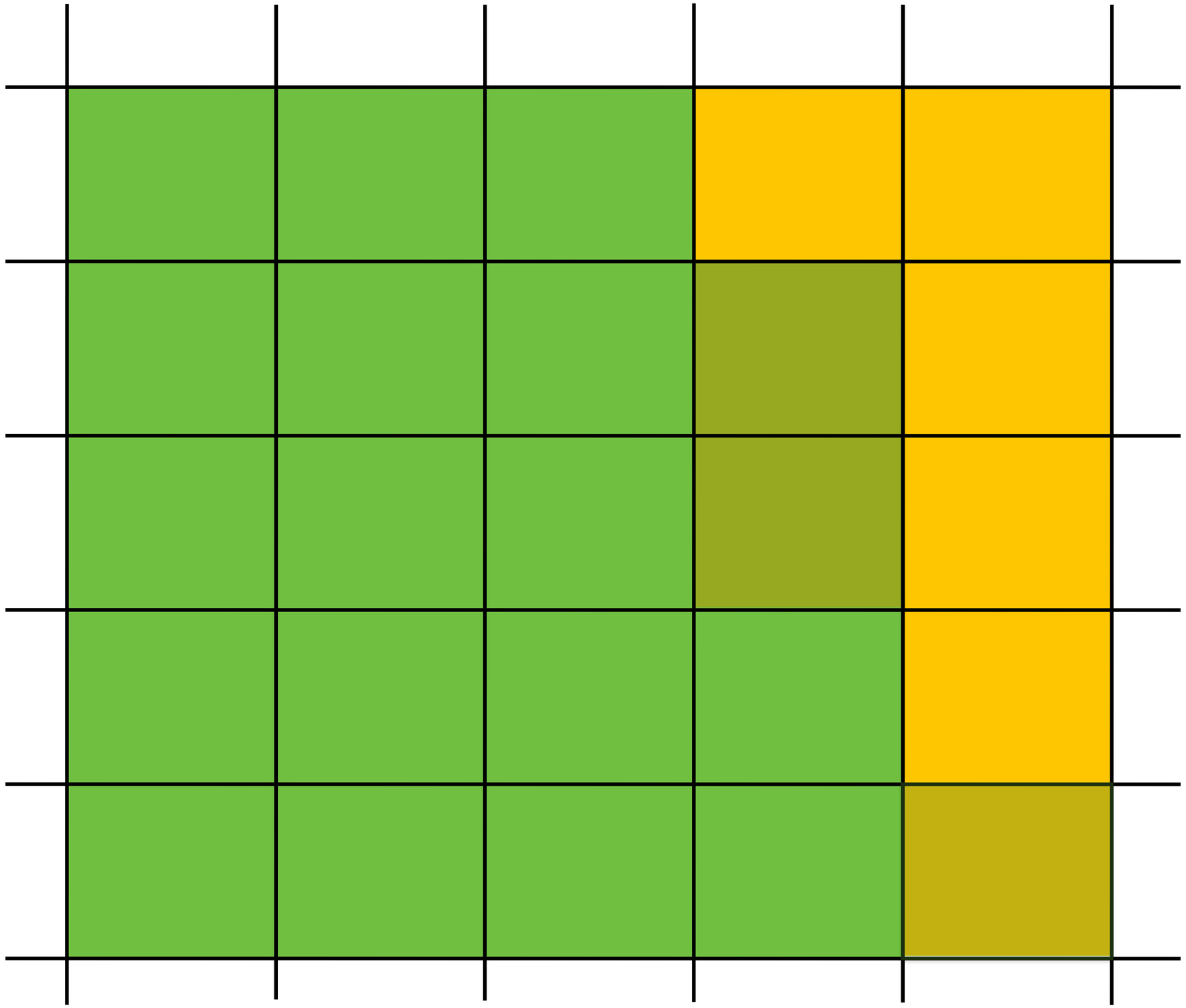
Color buffer contents



Color buffer contents (4 samples per pixel)



Final resampled result



Note anti-aliasing of edge due to filtering of green and yellow samples.

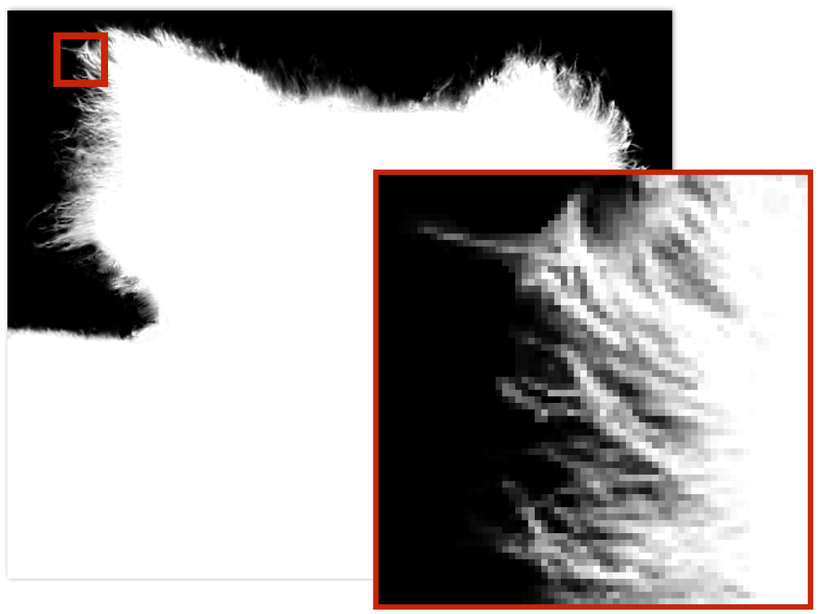
Summary: occlusion using a depth buffer

- Store one depth value per coverage sample (not per pixel!)
- Constant space per sample
  - Implication: constant space for depth buffer
- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if “pass” depth test
  - Just a read if “fail”
- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point
- Range of depth values is limited. That’s why the near and far planes are used in defining the view frustum!

But what about semi-transparent objects?

Compositing

Alpha: additional channel of image (rgba)



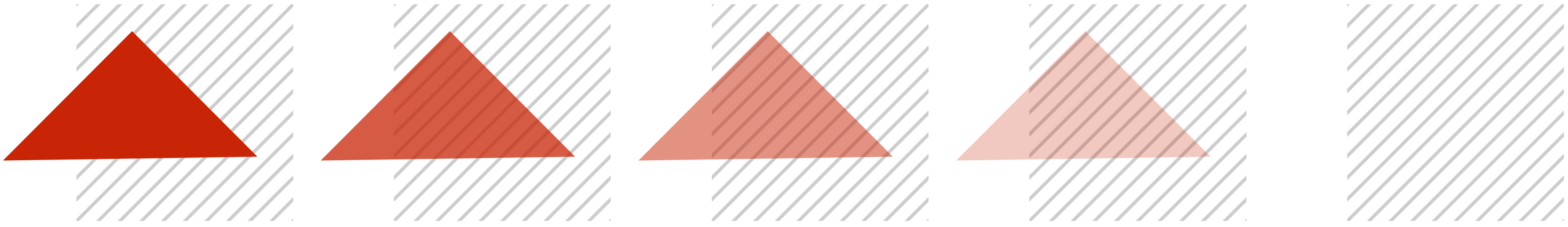
α of foreground object

Representing opacity as alpha

Alpha describes the opacity of an object

- **Fully opaque surface: α = 1**
- **50% transparent surface: α = 0.5**
- **Fully transparent surface: α = 0**

Red triangle with decreasing opacity



α = 1

α = 0.75

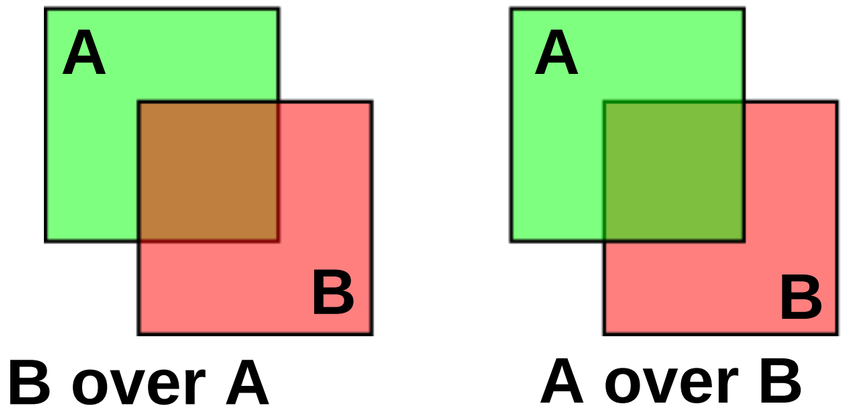
α = 0.5

α = 0.25

α =0

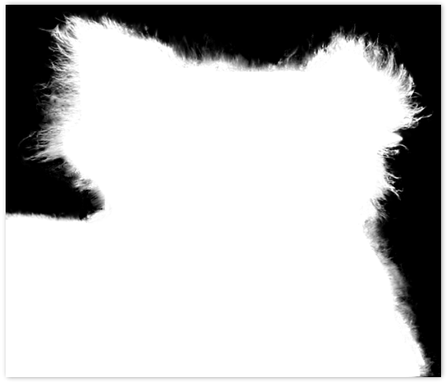
“Over" operator

Composite image B with opacity α_B over image A with opacity α_A



A over B != B over A

“Over” is not commutative



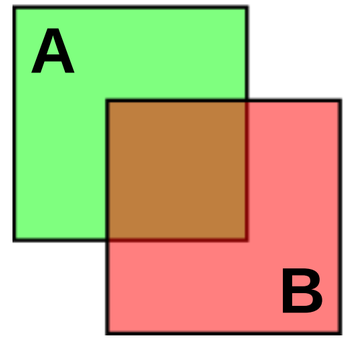
Koala over NYC

“Over" operator

Composite image B with opacity α_B over image A with opacity α_A

$$A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T$$

$$B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T$$



B over A

Composited color:

$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$

Appearance of semi-transparent B

What B lets through

Appearance of semi-transparent A

A over B != B over A

“Over” is not commutative

What is α_C?

$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$

“Over" operator

Composite image B with opacity α_B over image A with opacity α_A

First attempt:

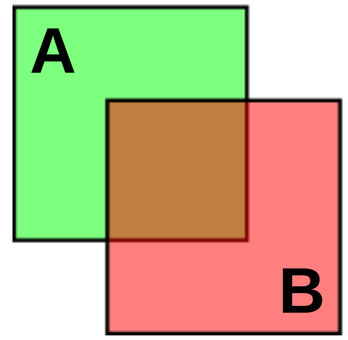
$$A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T$$

$$B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T$$

$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$

$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$

two multiplies, one add (referring to vector ops on colors)



B over A

Premultiplied alpha (equivalent):

$$A' = \begin{bmatrix} \alpha_A A_r & \alpha_A A_g & \alpha_A A_b & \alpha_A \end{bmatrix}^T$$

$$B' = \begin{bmatrix} \alpha_B B_r & \alpha_B B_g & \alpha_B B_b & \alpha_B \end{bmatrix}^T$$

$$C' = B + (1 - \alpha_B)A$$

one multiply, one add

Color buffer update: semi-transparent surfaces

Color buffer values and tri_color are represented with premultiplied alpha

```
over(c1, c2) {
   return c1 + (1-c1.a) * c2;
}

update_color_buffer(tri_d, tri_color, x, y) {

   if (pass_depth_test(tri_d, zbuffer[x][y]) {
      // update color buffer
      // Note: no depth buffer update
      color[x][y] = over(tri_color, color[x][y]);
  }
}
```

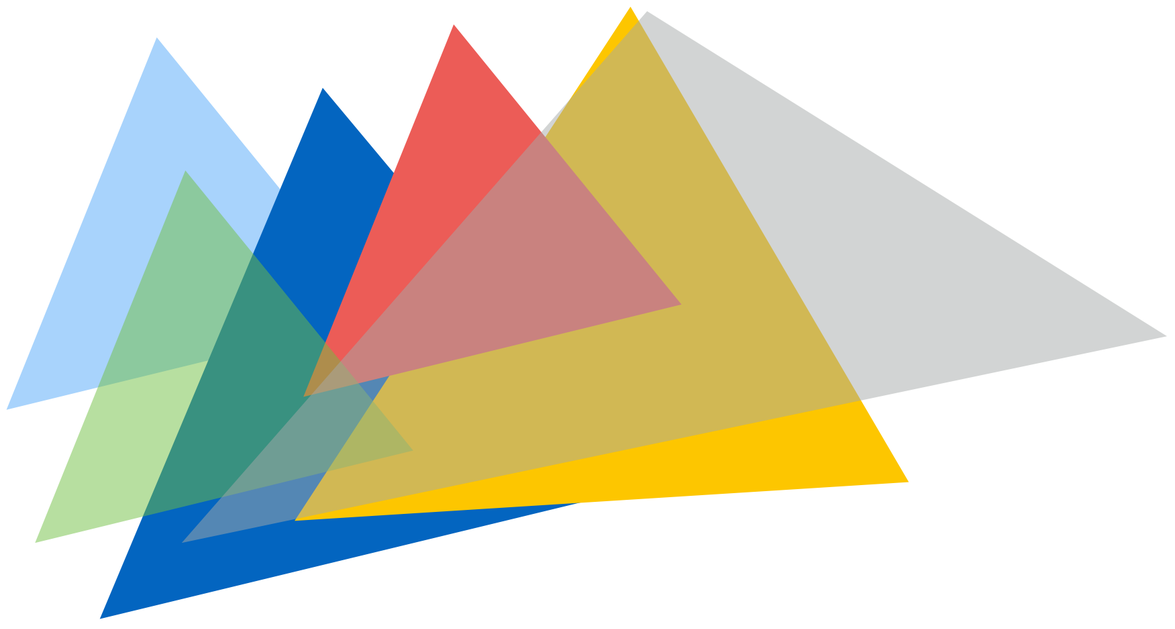
Q: What is the assumption made by this implementation?

Triangles must be rendered in back to front order!

Rendering a mixture of opaque and transparent triangles

Step 1: render opaque surfaces using depth-buffered occlusion (If pass depth test passed, triangle overwrites value in color buffer at sample)

Step 2: disable depth buffer update, render semi-transparent surfaces in back-to-front order. If depth test passed, triangle is composited OVER contents of color buffer at sample

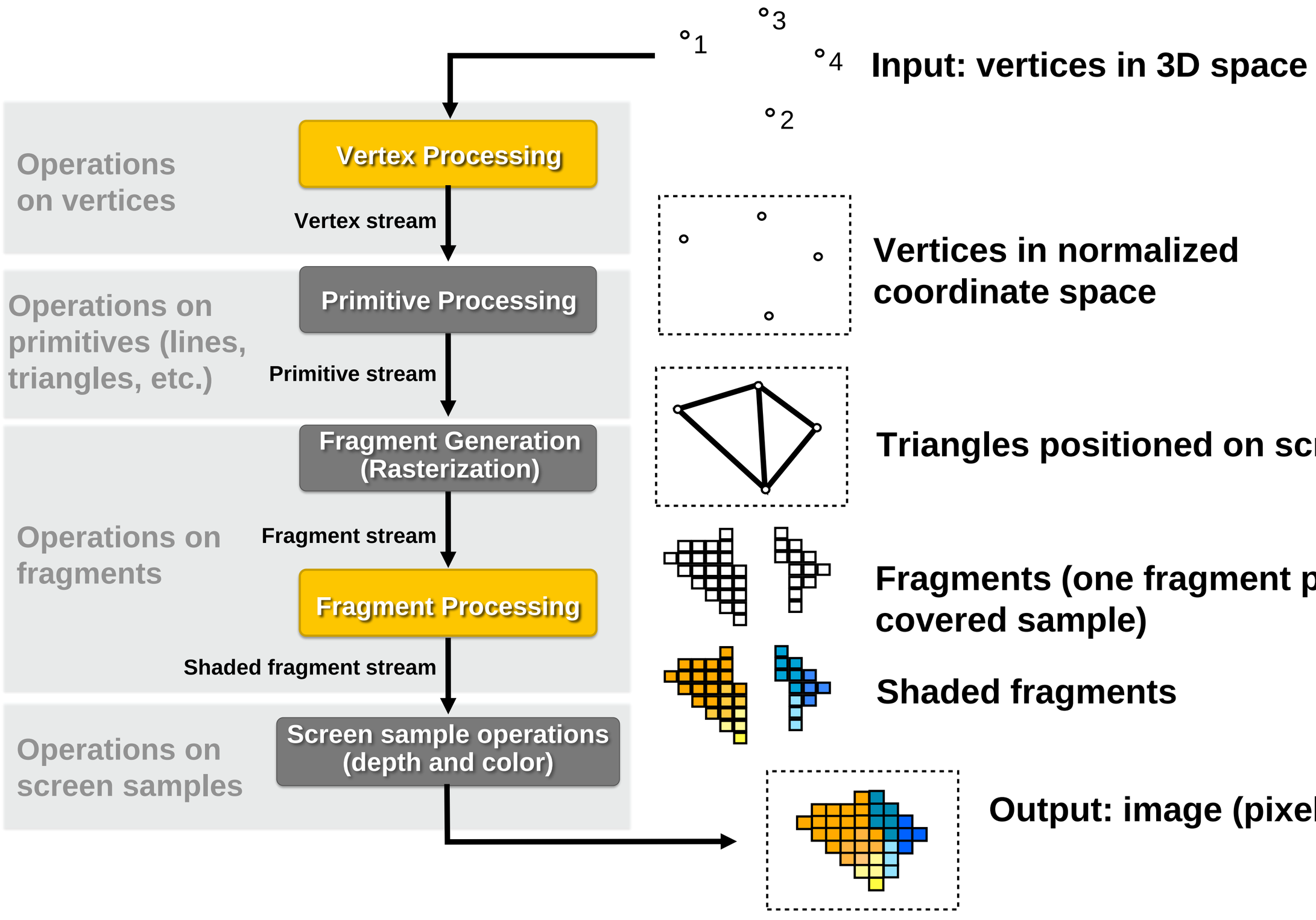


Putting it all together

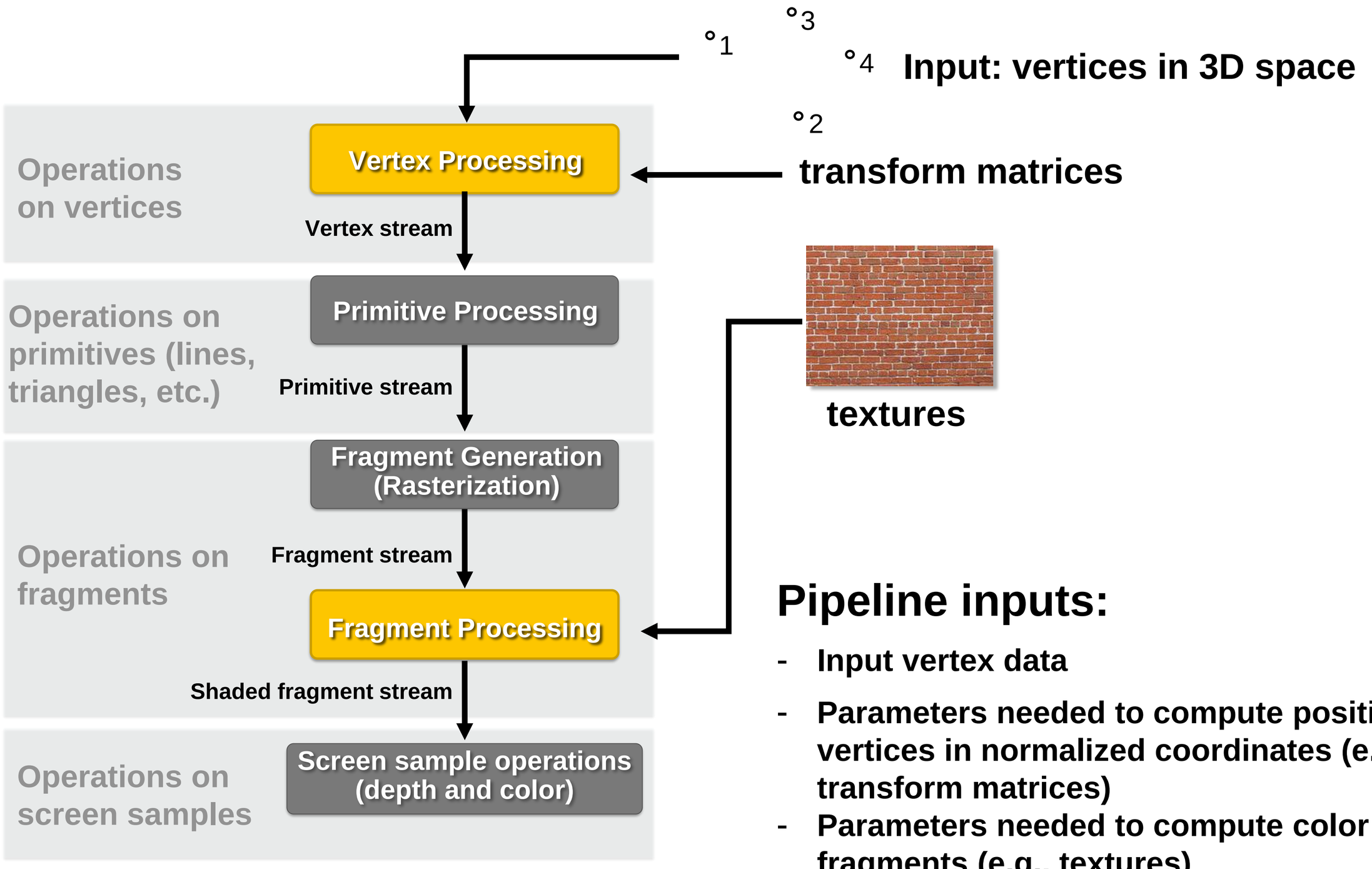
Unreal Engine Kite Demo (Epic Games 2015)

End-to-end rasterization pipeline (“real-time graphics pipeline”)

The real-time graphics pipeline



The real-time graphics pipeline



Pipeline inputs:

- Input vertex data
- Parameters needed to compute position of vertices in normalized coordinates (e.g., transform matrices)
- Parameters needed to compute color of fragments (e.g., textures)

Command: draw these triangles!

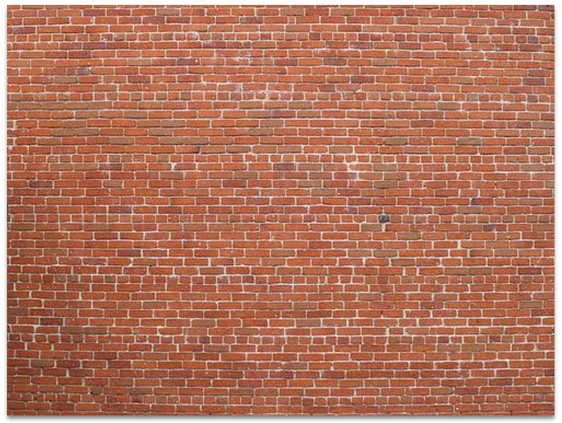
Inputs:

list_of_positions = {

v0x, v0y, v0z,
v1x, v1y, v1z,
v2x, v2y, v2z,
v3x, v3y, v3z,
v4x, v4y, v4z,
v5x, v5y, v5z };

list_of_texcoords = {

v0u, v0v,
v1u, v1v,
v2u, v2v,
v3u, v3v,
v4u, v4v,
v5u, v5v };



Texture map

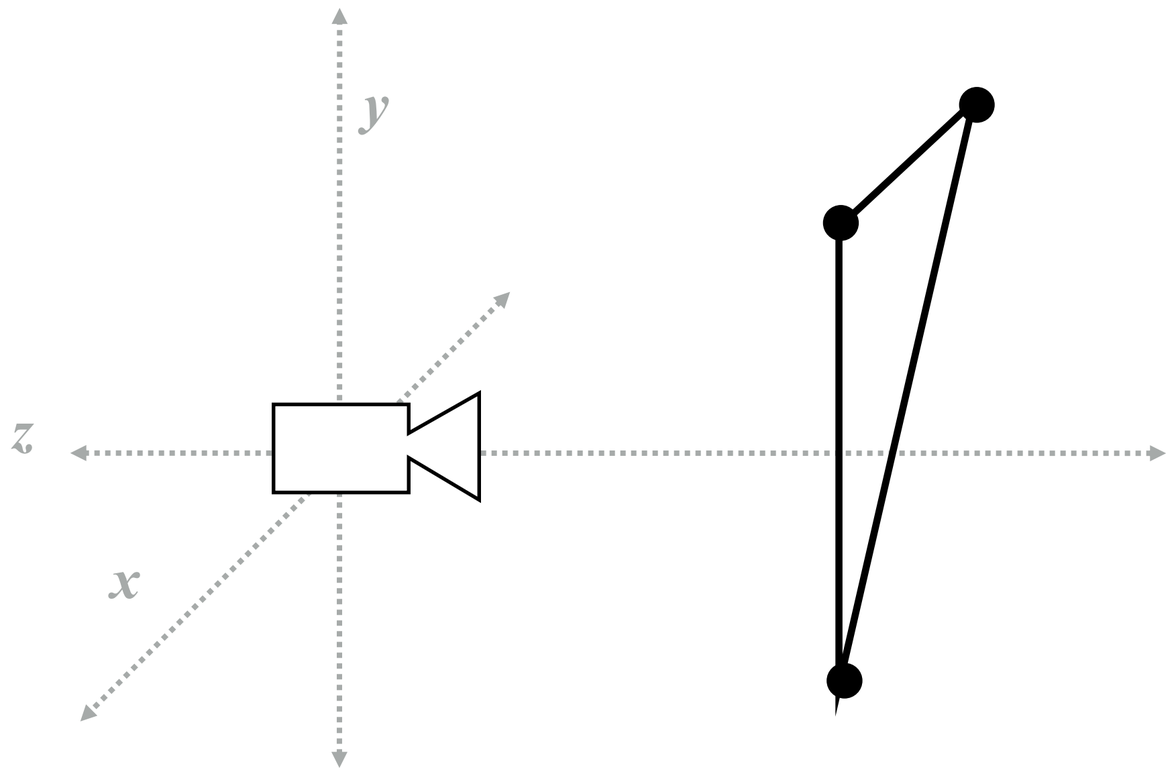
Object-to-camera-space transform *T*

Perspective projection transform *P*

Size of output image (W, H)

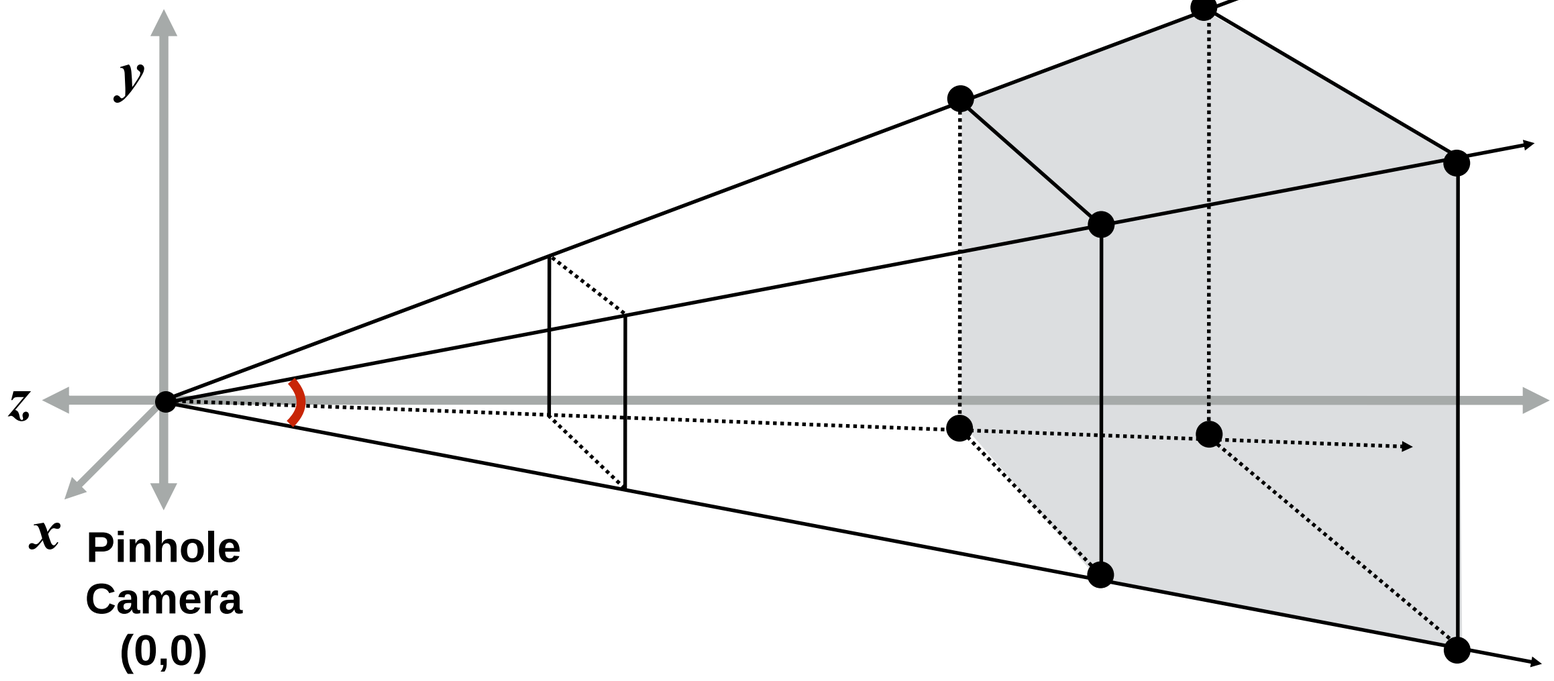
Step 1:

Transform triangle vertices into camera space

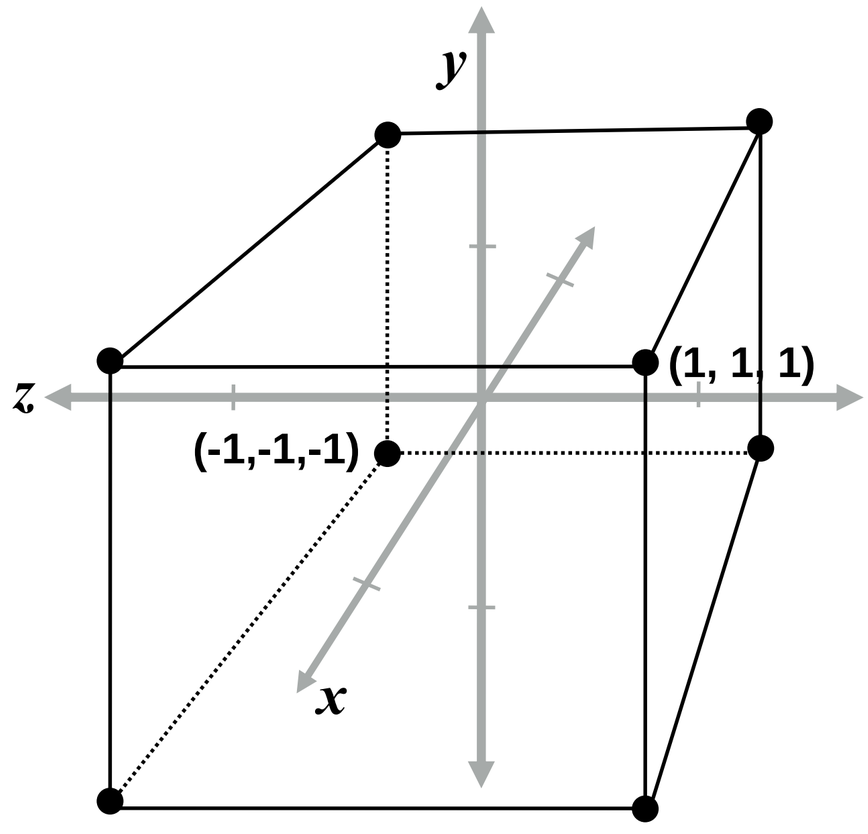


Step 2:

Apply perspective projection transform to transform triangle vertices into normalized coordinate space



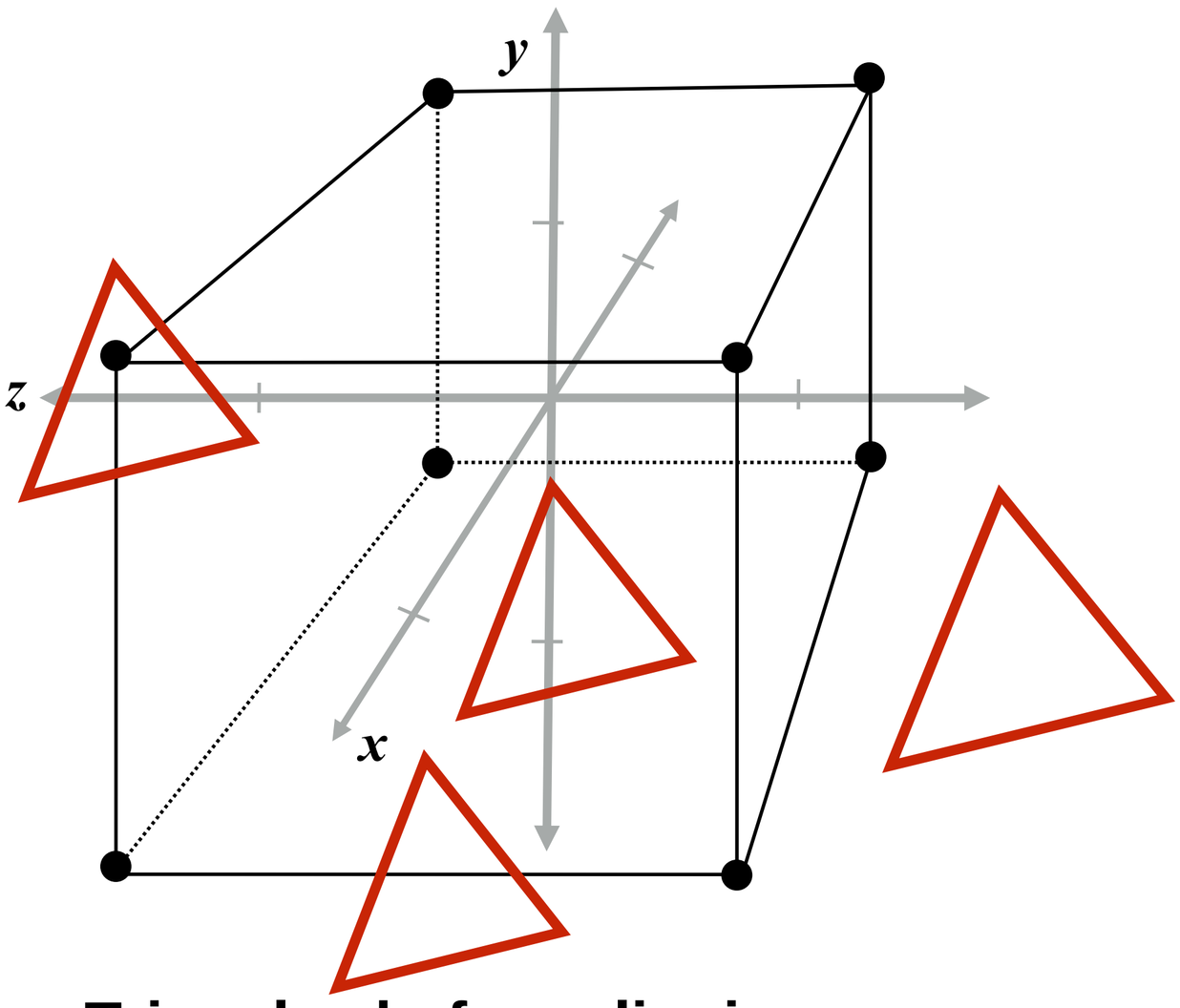
Camera-space positions: 3D



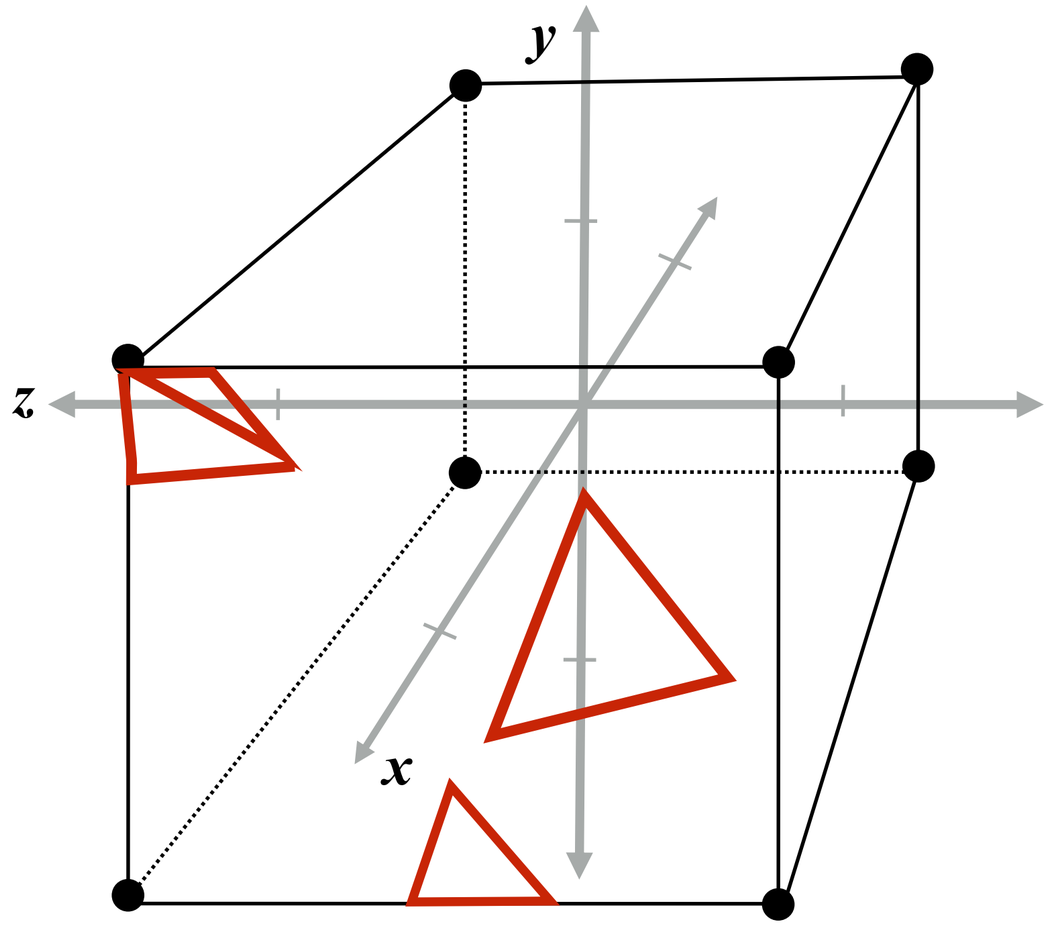
Normalized space positions

Step 3:

- Discard triangles that lie complete outside the unit cube (culling)
  - They are off screen, don't bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
  - Note: clipping may create more triangles



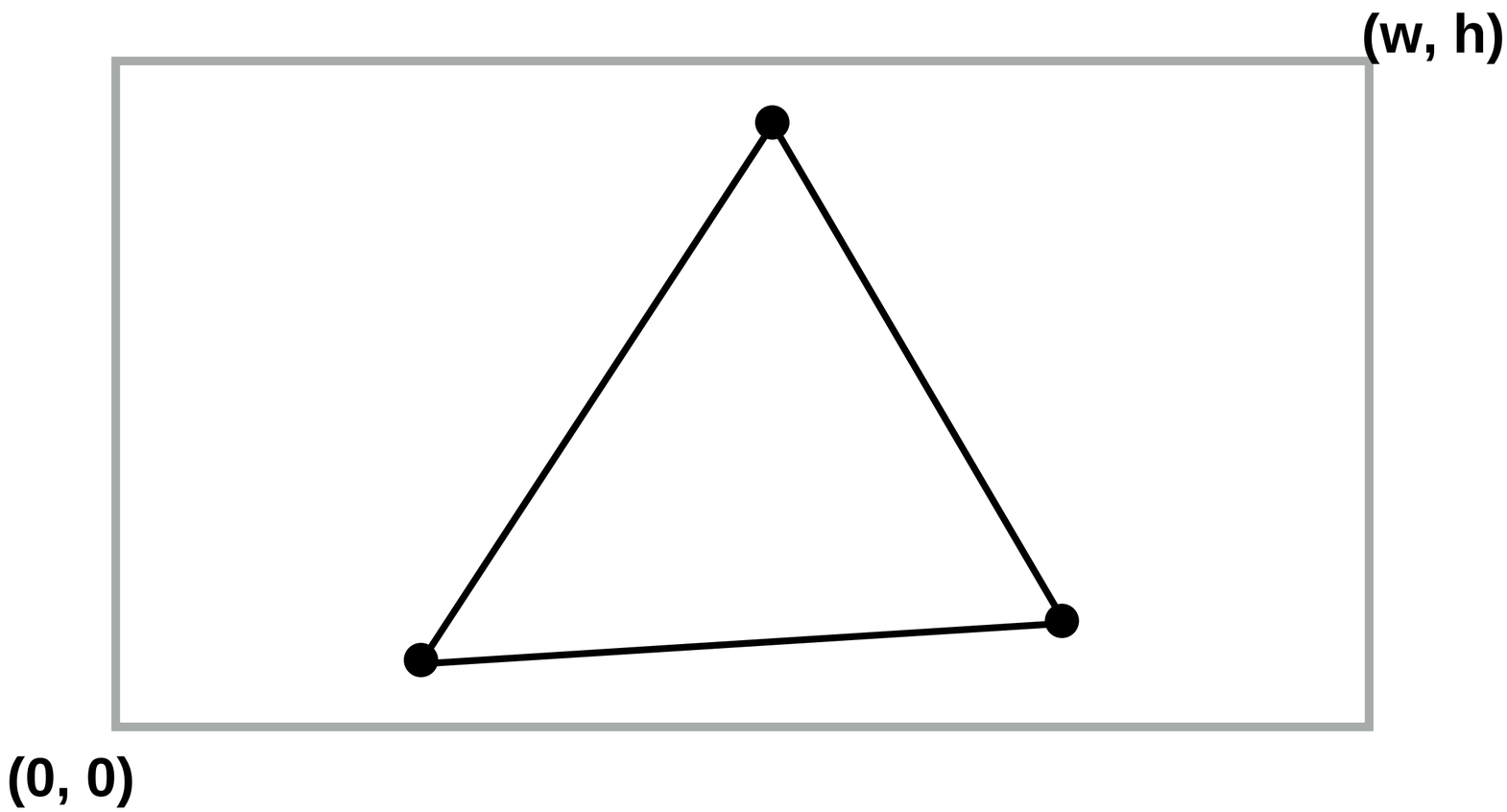
Triangles before clipping



Triangles after clipping

Step 4:

Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)



Step 5:

Triangle preprocessing

- **Compute triangle edge equations**
- **Compute triangle attribute equations**

$\mathbf{E}_{01}(x, y)$

$\mathbf{E}_{12}(x, y)$

$\mathbf{E}_{20}(x, y)$

$\frac{1}{\mathbf{w}}(x, y)$

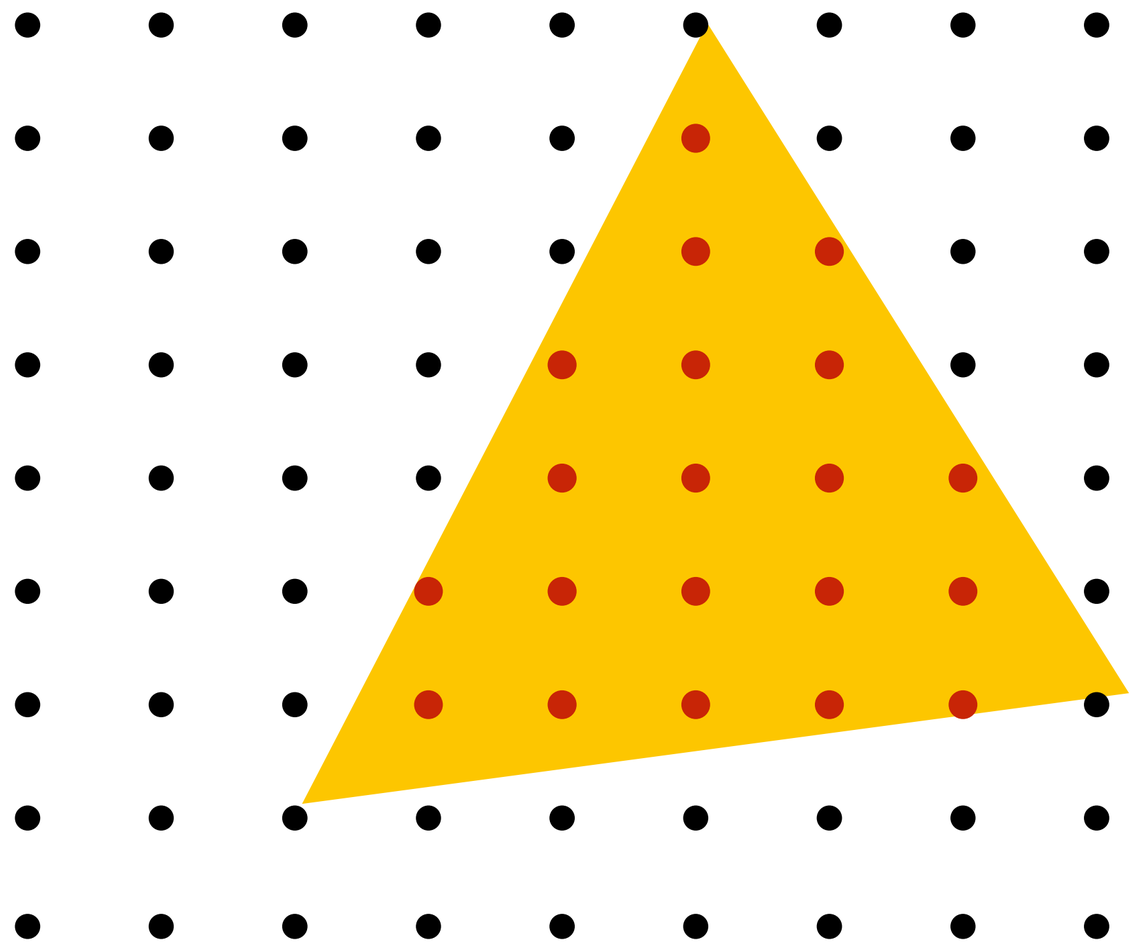
$\mathbf{Z}(x, y)$

$\mathbf{U}(x, y)$

$\mathbf{V}(x, y)$

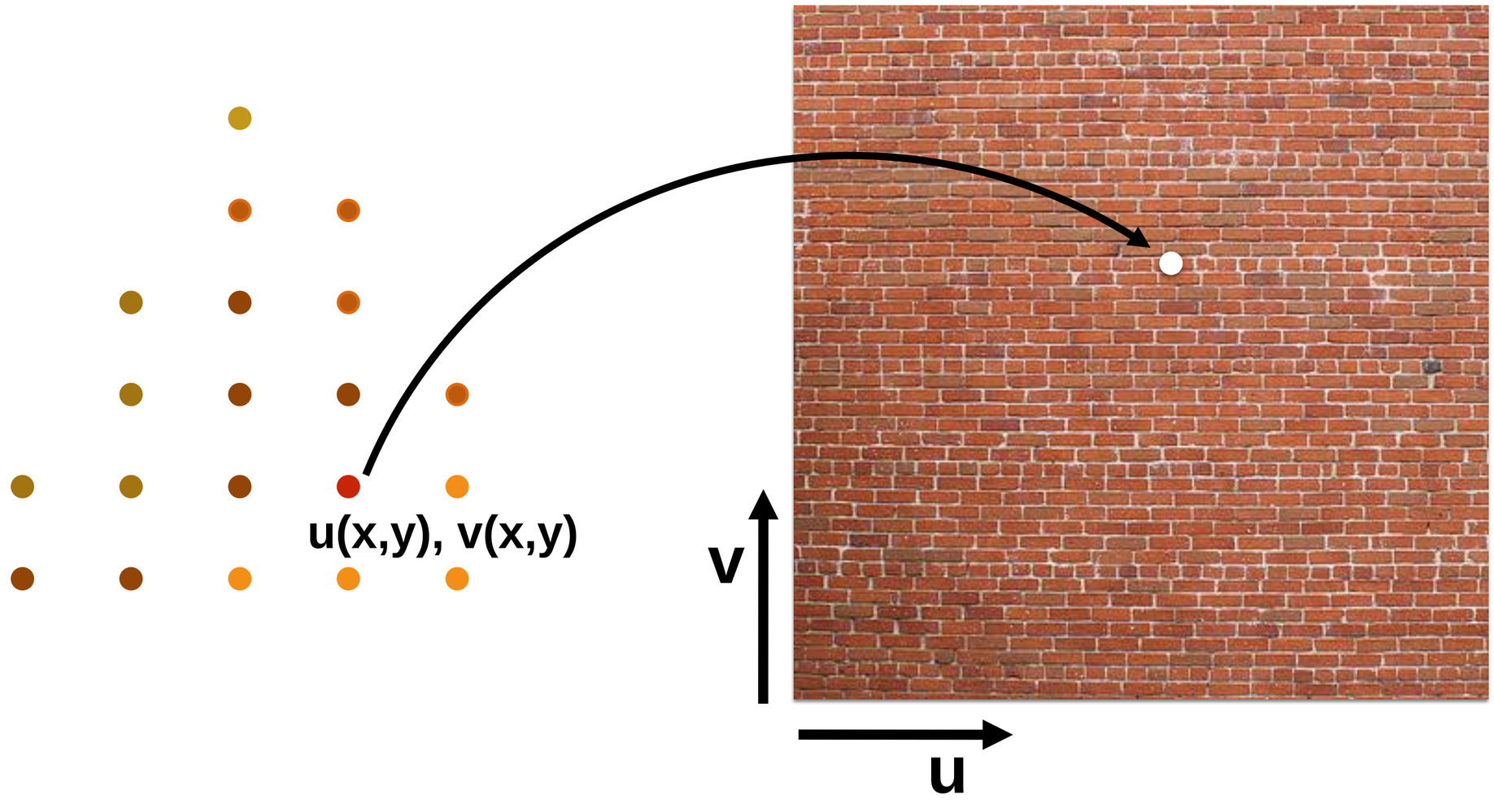
Step 6:

Sample coverage, evaluate attributes z, u, v at all covered samples



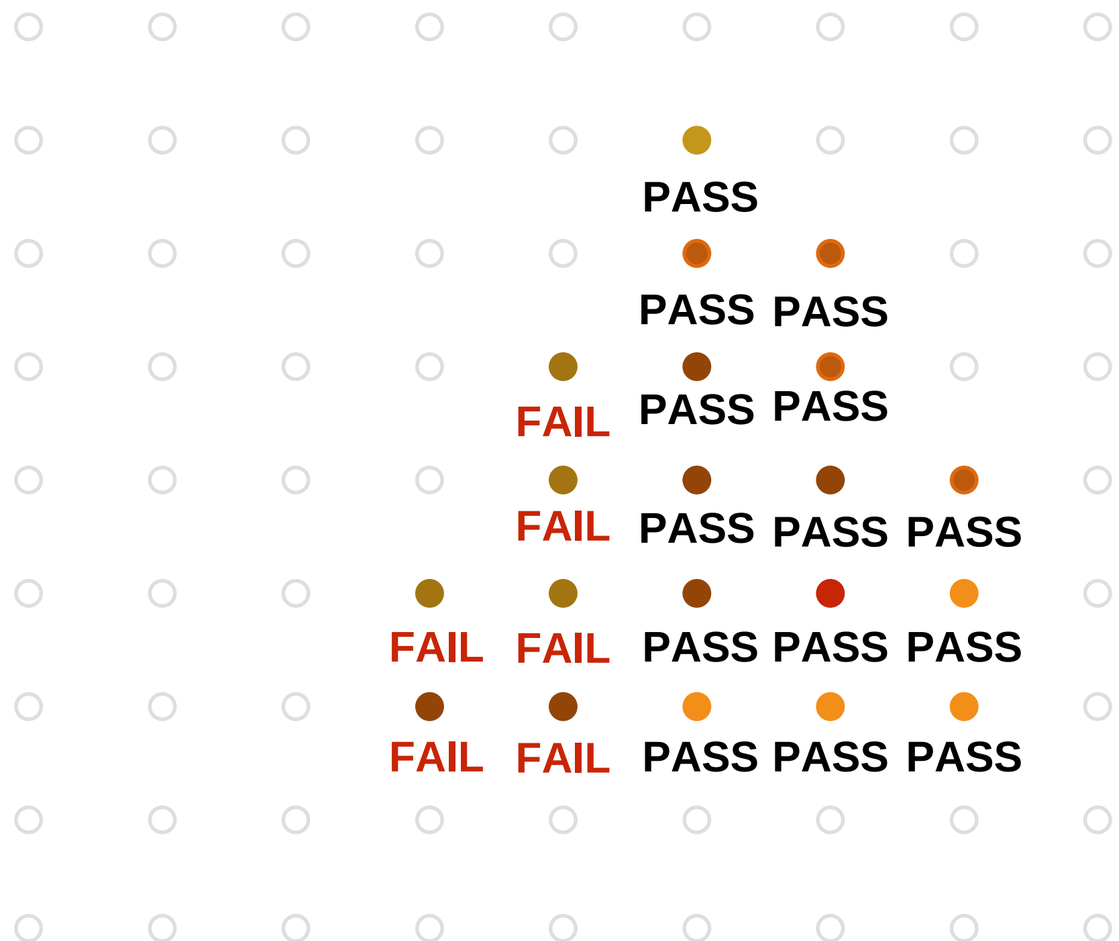
Step 7:

Compute triangle color at sample point (color interpolation, sample texture map, or more advanced shading algorithms)



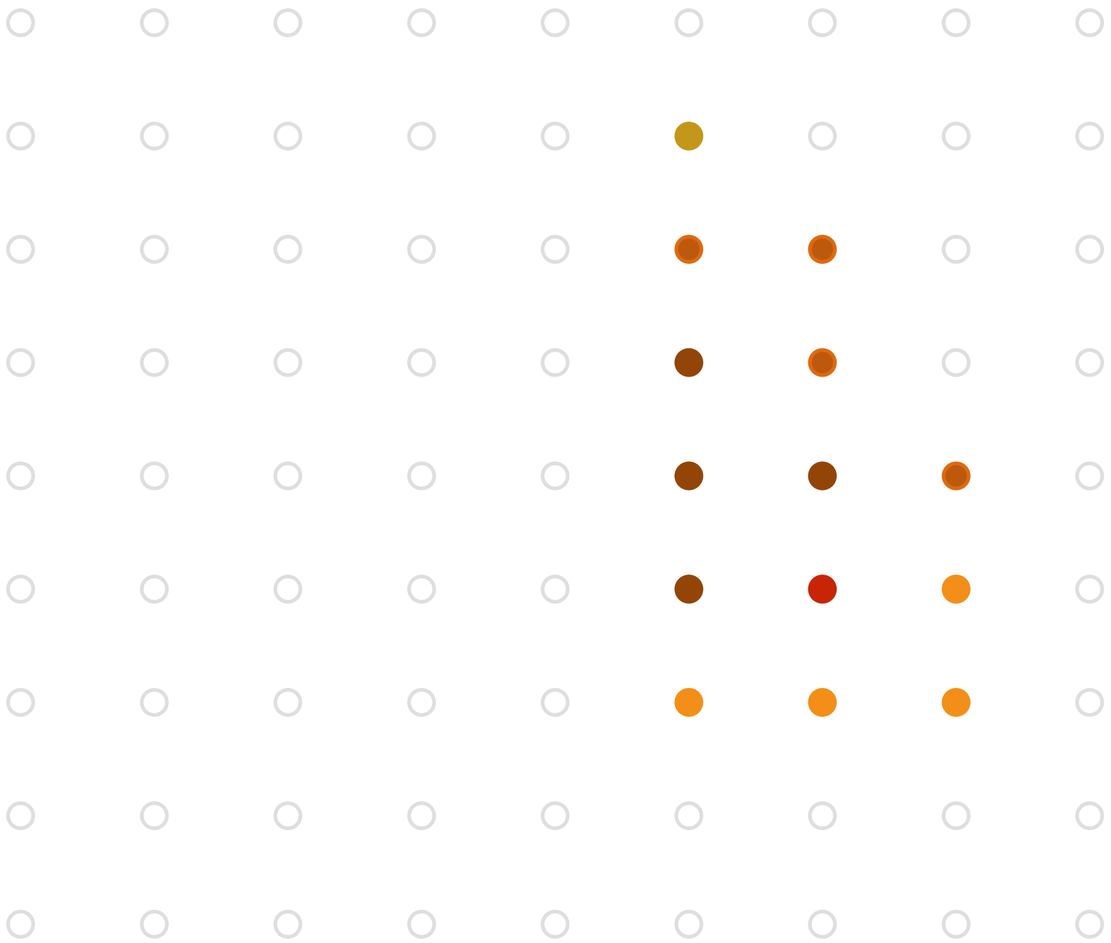
Step 8:

Perform depth test (if enabled) and update depth value at covered samples (if necessary)

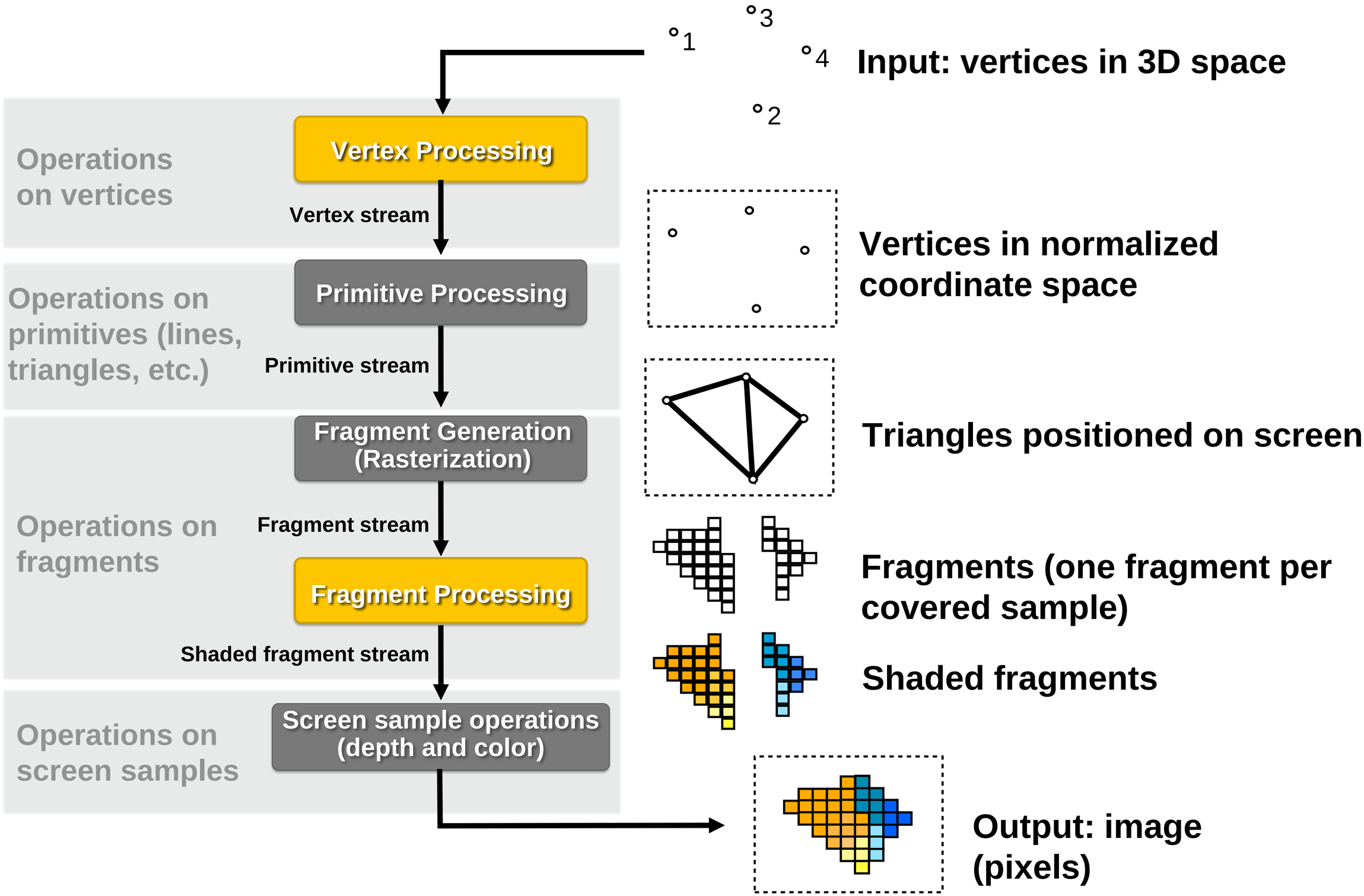


Step 9:

update color buffer (if depth test passed)



OpenGL/Direct3D graphics pipeline*



Note: “Shader” programs define behavior of vertex and fragment stages

*** Several stages of the modern OpenGL pipeline are omitted**

Shader programs

Define behavior of vertex processing and fragment processing stages
Describe operation on a single vertex (or single fragment)

Example GLSL fragment shader program

```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
  gl_FragColor = vec4(kd, 1.0);
}
```

Program parameters

Per-fragment attributes (interpolated by rasterizer)

Sample surface albedo (reflectance color) from texture

Shader outputs surface color

Modulate surface albedo by incident irradiance (incoming light)

Shader function executes once per fragment.

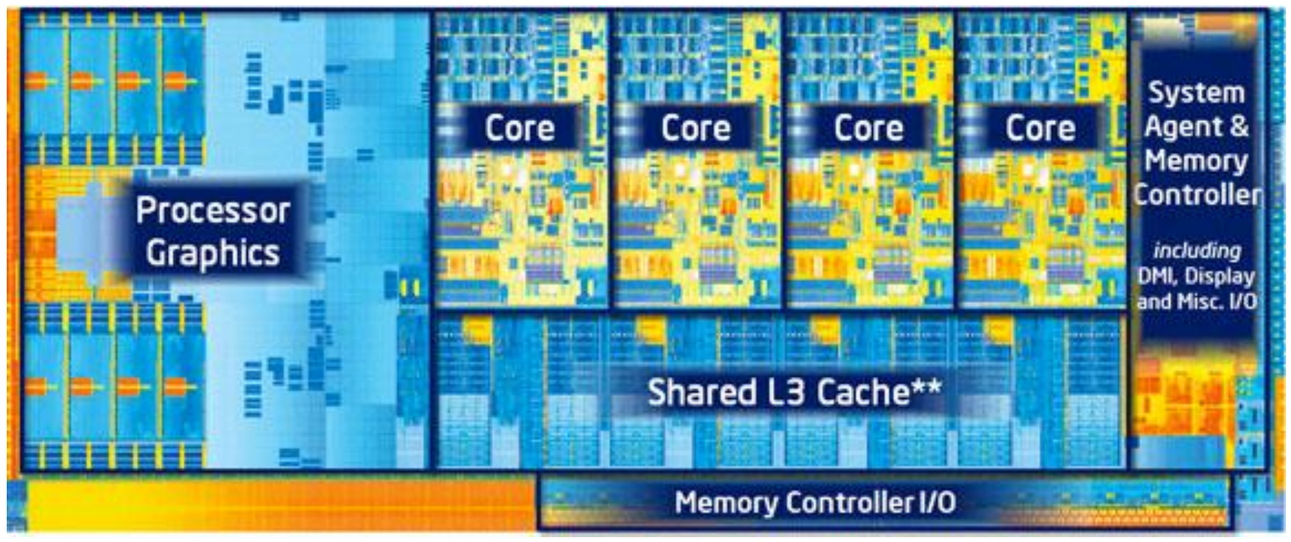
Outputs color of surface at sample point that corresponds to fragment.

(this shader performs a texture lookup to obtain the surface's material color at this point, then performs a simple lighting computation)

Graphics pipeline hardware implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU card (NVIDIA GeForce Titan X)

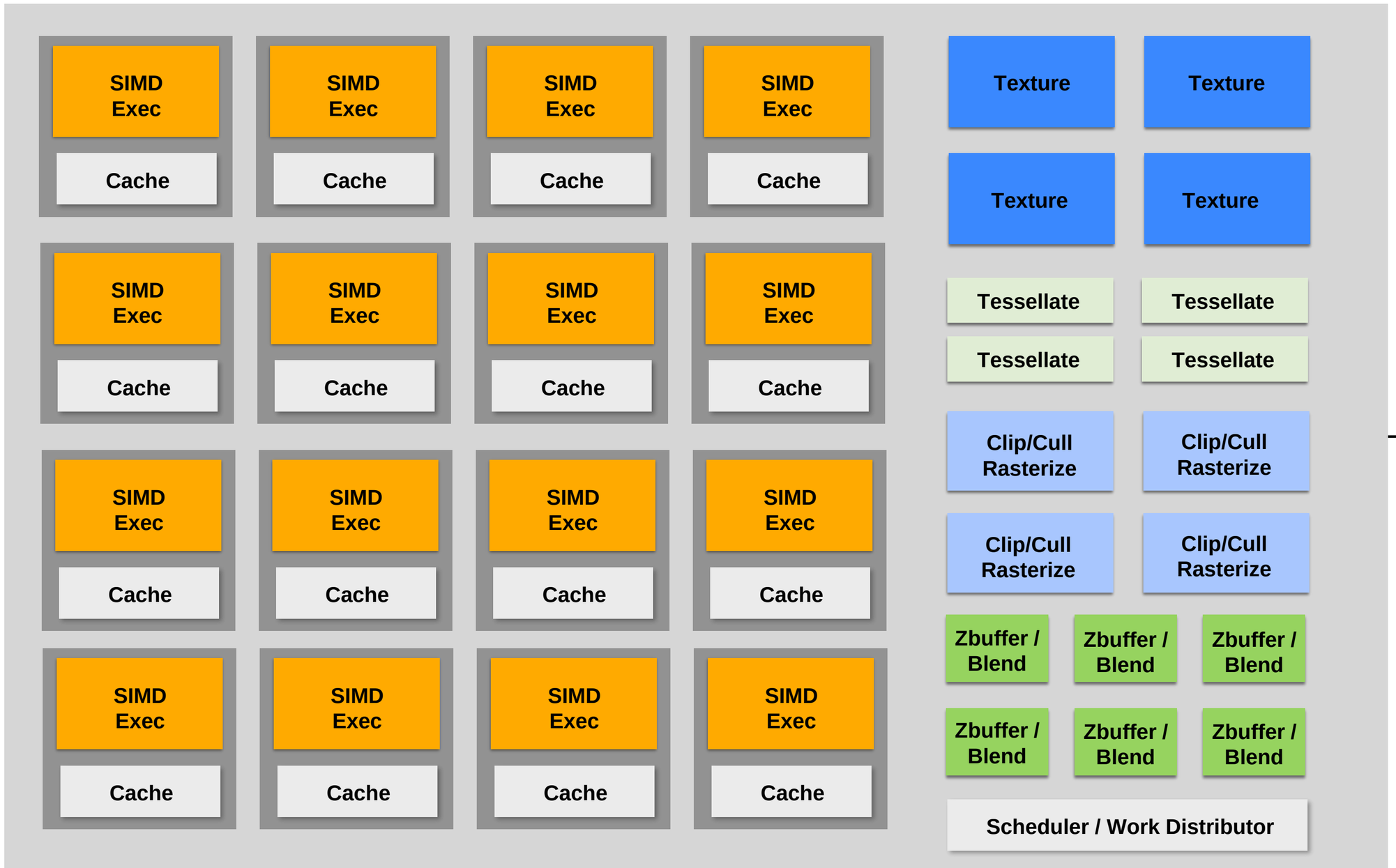


Integrated GPU: part of modern Intel CPU die

GPU: heterogeneous, multi-core processor

Modern GPUs offer ~2-4 TFLOPs of performance for executing vertex and fragment shader programs

T-OP's of fixed-function compute capability over here



Take Kayvon's parallel computing course (15-418) for more details!

GPUs render very high complexity 3D scenes

- 100’s of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (2-4 Mpixel + supersampling)
- 30-60 fps

Unreal Engine Kite Demo (Epic Games 2015)

Unreal Engine Kite Demo (Epic Games 2015)

Summary

- Occlusion resolved independently at each screen sample using the depth buffer

- Alpha compositing for semi-transparent surfaces
  - Premultiplied alpha forms simply repeated composition
  - “Over” compositing operations is not commutative: requires triangles to be processed in back-to-front (or front-to-back) order

- Graphics pipeline:
  - Structures rendering computation as a sequence of operations performed on vertices, primitives (e.g., triangles), fragments, and screen samples
  - Behavior of parts of the pipeline is application-defined using shader programs.
  - Pipeline operations implemented by highly optimized parallel processors and fixed-function hardware (GPUs)