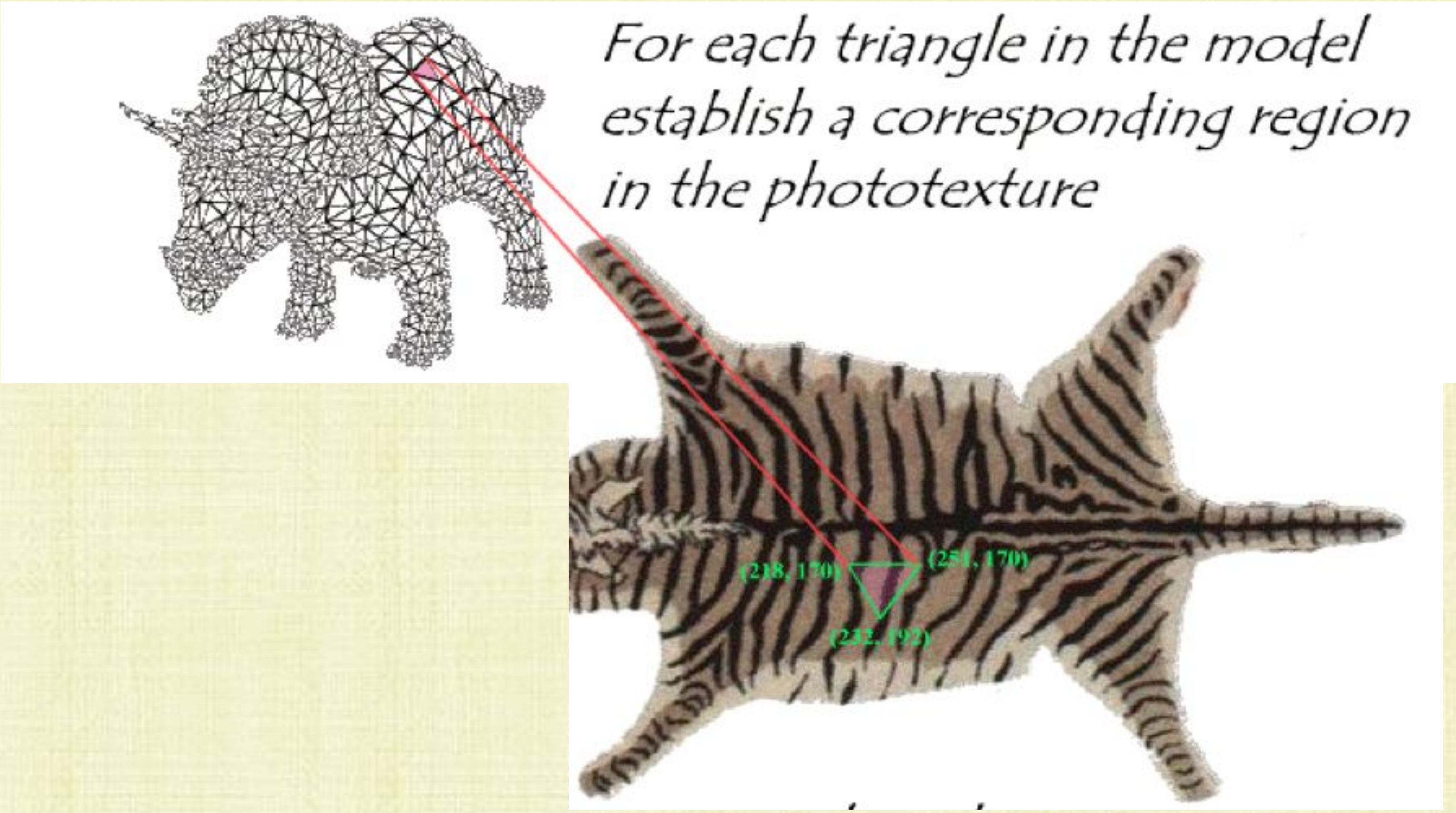
Texture Mapping

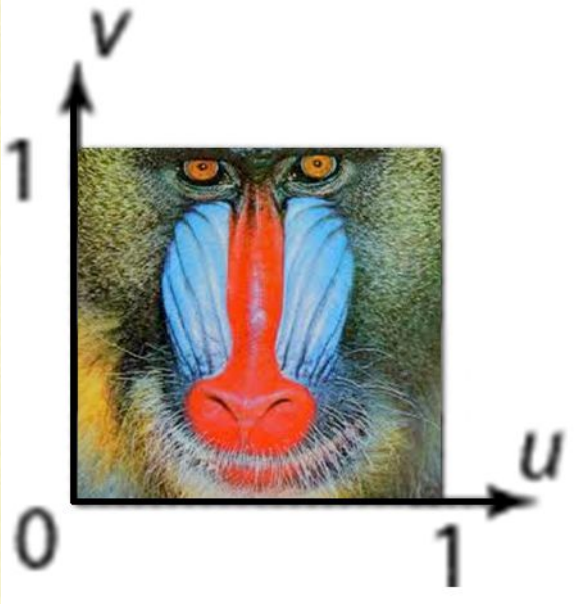
Texture Mapping

- Offsets the assumption that the BRDF doesn't change in u and v coordinates along an object's surface
- Store RGB reflectance as an image (called a texture)
- Map that image onto the object (one triangle at a time)

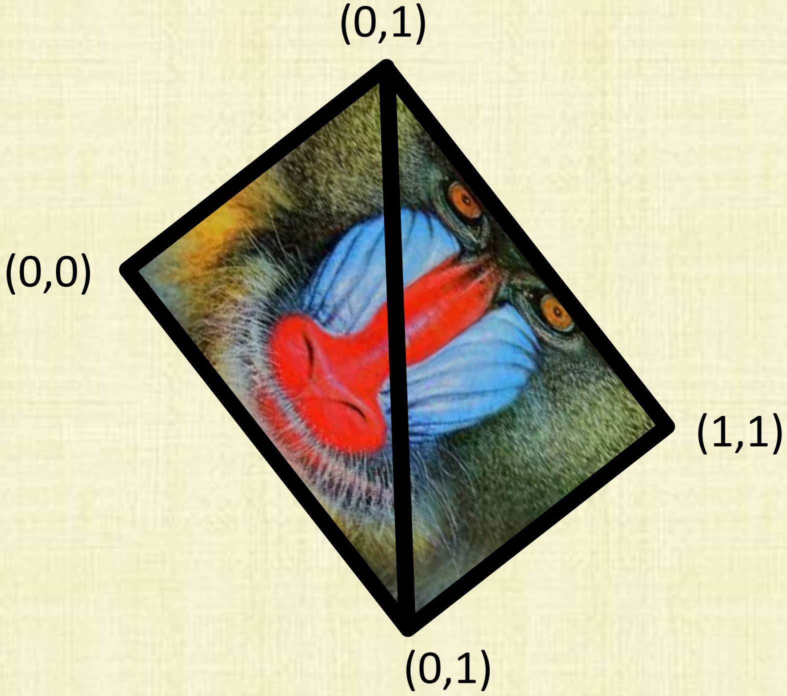


Texture Coordinates

- A texture image is defined in a 2D coordinate system: (u, v)
- Texture mapping assigns each triangle vertex a (u, v) coordinate
- Then, the texture is "stuck" onto the triangle:
  - Let p be a point inside the triangle, with barycentric weights $\alpha_0, \alpha_1, \alpha_2$
  - The reflectance color at p is the texture color at $\big(u(p), v(p)\big) = \alpha_0(u_0, v_0) + \alpha_1(u_1, v_1) + \alpha_2(u_2, v_2)$
  - That is, texture coordinates are barycentrically interpolated



texture image

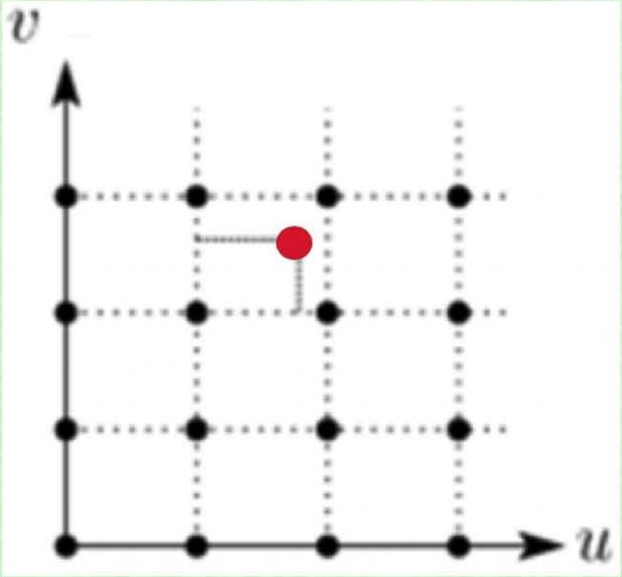


2 triangles

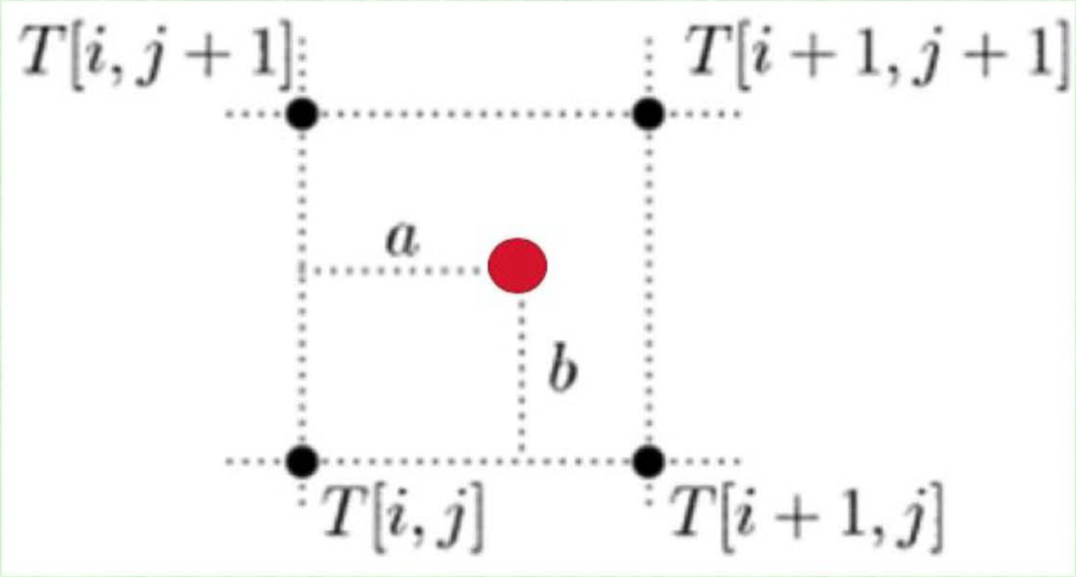
Interpolating RGB Color

- $(u(p), v(p))$ is surrounded by 4 pixels in the texture image
- Use bilinear interpolation for T = R, G, B, α, etc.
- First, linearly interpolate in the u direction; then, in the v direction (or vice versa)

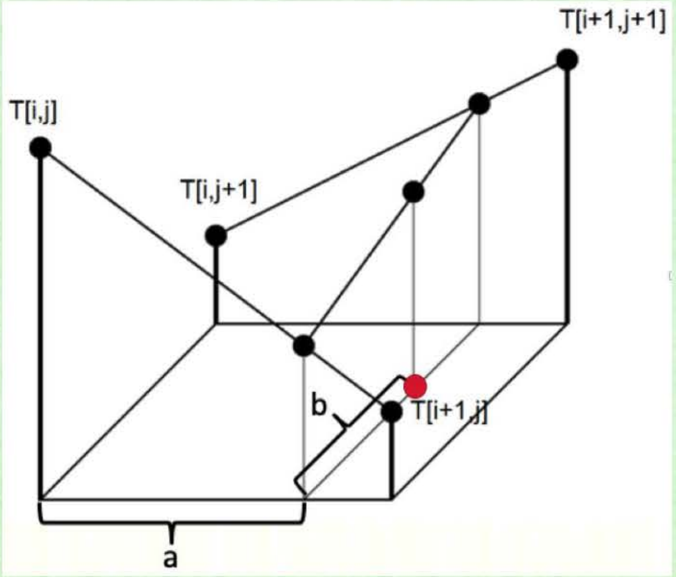
$$T(u, v) = (1-a)(1-b)T_{i,j} + a(1-b)T_{i+1,j} + (1-a)bT_{i,j+1} + abT_{i+1,j+1}$$



uv texture space



close up view (of 4 surrounding pixels)



bilinear interpolation

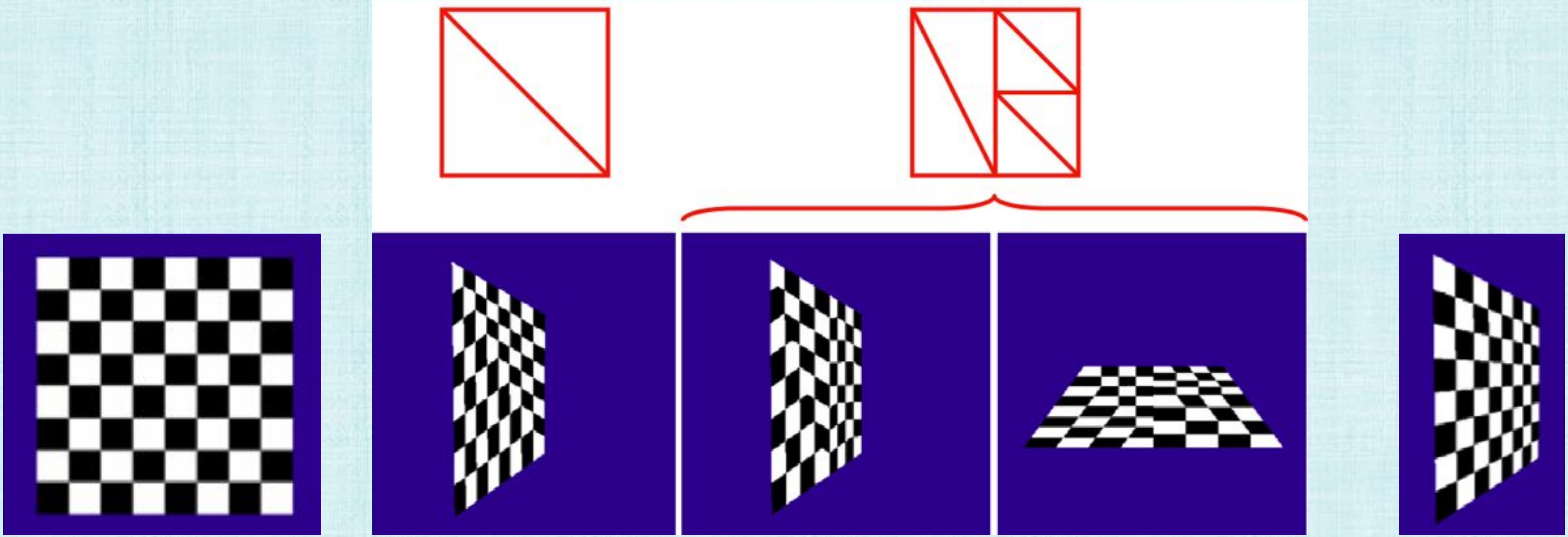
Recall: Perspective Projection

- Project a world space triangle (vertices p_0, p_1, p_2) into screen space (vertex by vertex) to obtain p'_0, p'_1, p'_2 via $x' = \frac{hx}{z}$ and $y' = \frac{hy}{z}$ for each vertex (x, y, z)
- A point $p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$ on a world space triangle is projected into screen space to a corresponding point p'
- Notably, $p' \neq \alpha_0 p'_0 + \alpha_1 p'_1 + \alpha_2 p'_2$ because the perspective projection is highly nonlinear
- The barycentric weights that describe the interior of the triangle in world space do not still hold after projecting the vertices into screen space

- Need a way of computing z' at a pixel from the z' values at the vertices of the screen space triangle
- The z' values are not linear with respect to the triangle vertices in screen space, only in world space (so can't use barycentric interpolation!)
- However, if we knew the location of the pixel on the world space triangle, we could use barycentric interpolation on the world space triangle to compute z and z' for the pixel

Screen Space vs. World Space

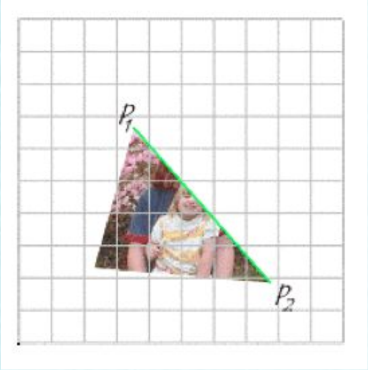
- Perspective transformation nonlinearly changes a triangle's shape, leading to different barycentric weights before/after (as we have seen)
- Interpolating texture coordinates in screen space results in texture distortion

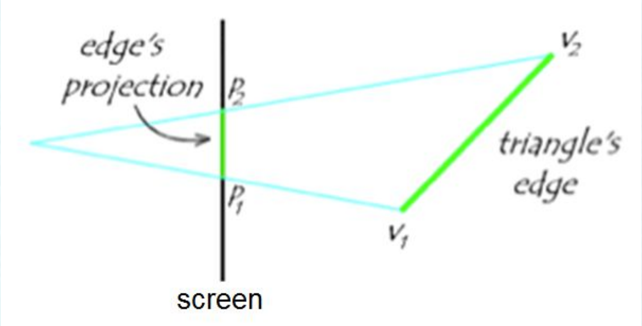


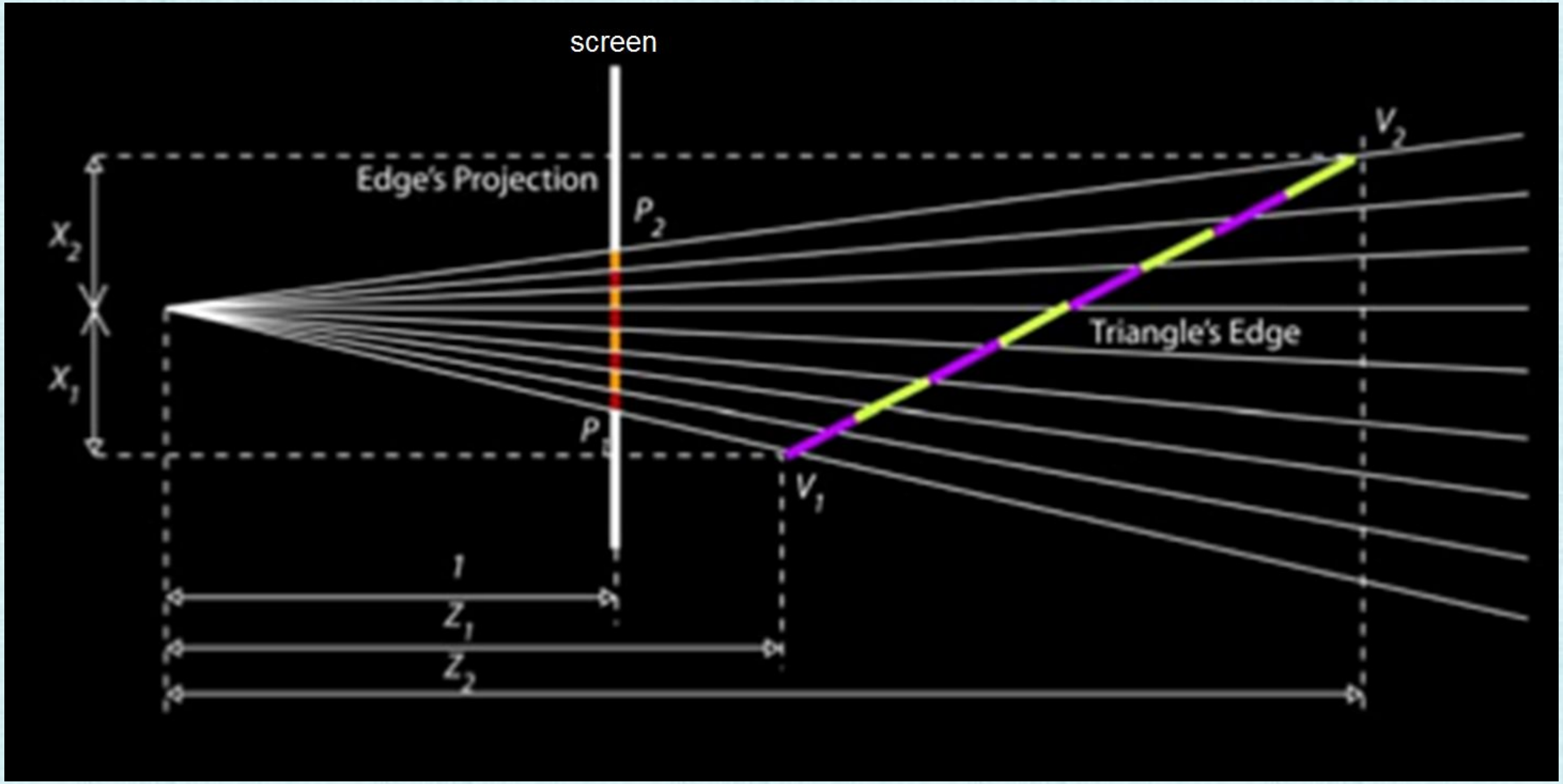
texture | screen space | mesh refinement helps (less z variance per triangle) | world space

Texture Distortion

- Consider one triangle edge
- Uniform increments along the edge in world space do not correspond to uniform increments in screen space (linear barycentric interpolation cannot account for this nonlinearity)







Recall: Screen Space Barycentric Weights

- Starting from $\frac{1}{\alpha_0(z_0-z_2)+\alpha_1(z_1-z_2)+z_2}\begin{pmatrix} z_0\alpha_0 \\ z_1\alpha_1 \end{pmatrix} = \begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$ or $\begin{pmatrix} z_0\alpha_0 \\ z_1\alpha_1 \end{pmatrix} = (\alpha_0(z_0 - z_2) + \alpha_1(z_1 - z_2) + z_2)\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$
- Rewrite to $\begin{pmatrix} z_0 + (z_2 - z_0)\alpha_0' & (z_2 - z_1)\alpha_0' \\ (z_2 - z_0)\alpha_1' & z_1 + (z_2 - z_1)\alpha_1' \end{pmatrix}\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = z_2\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$
- The determinant of this 2x2 matrix is $z_0z_1 + z_1(z_2 - z_0)\alpha_0' + z_0(z_2 - z_1)\alpha_1'$
- Thus the inverse is $\frac{1}{z_0z_1+z_1(z_2-z_0)\alpha_0'+z_0(z_2-z_1)\alpha_1'}\begin{pmatrix} z_1 + (z_2 - z_1)\alpha_1' & (z_1 - z_2)\alpha_0' \\ (z_0 - z_2)\alpha_1' & z_0 + (z_2 - z_0)\alpha_0' \end{pmatrix}$
- Note that $\begin{pmatrix} z_1 + (z_2 - z_1)\alpha_1' & (z_1 - z_2)\alpha_0' \\ (z_0 - z_2)\alpha_1' & z_0 + (z_2 - z_0)\alpha_0' \end{pmatrix}\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix} = \begin{pmatrix} z_1\alpha_0' \\ z_0\alpha_1' \end{pmatrix}$
- Thus, $\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \frac{z_2}{z_0z_1+z_1(z_2-z_0)\alpha_0'+z_0(z_2-z_1)\alpha_1'}\begin{pmatrix} z_1\alpha_0' \\ z_0\alpha_1' \end{pmatrix}$

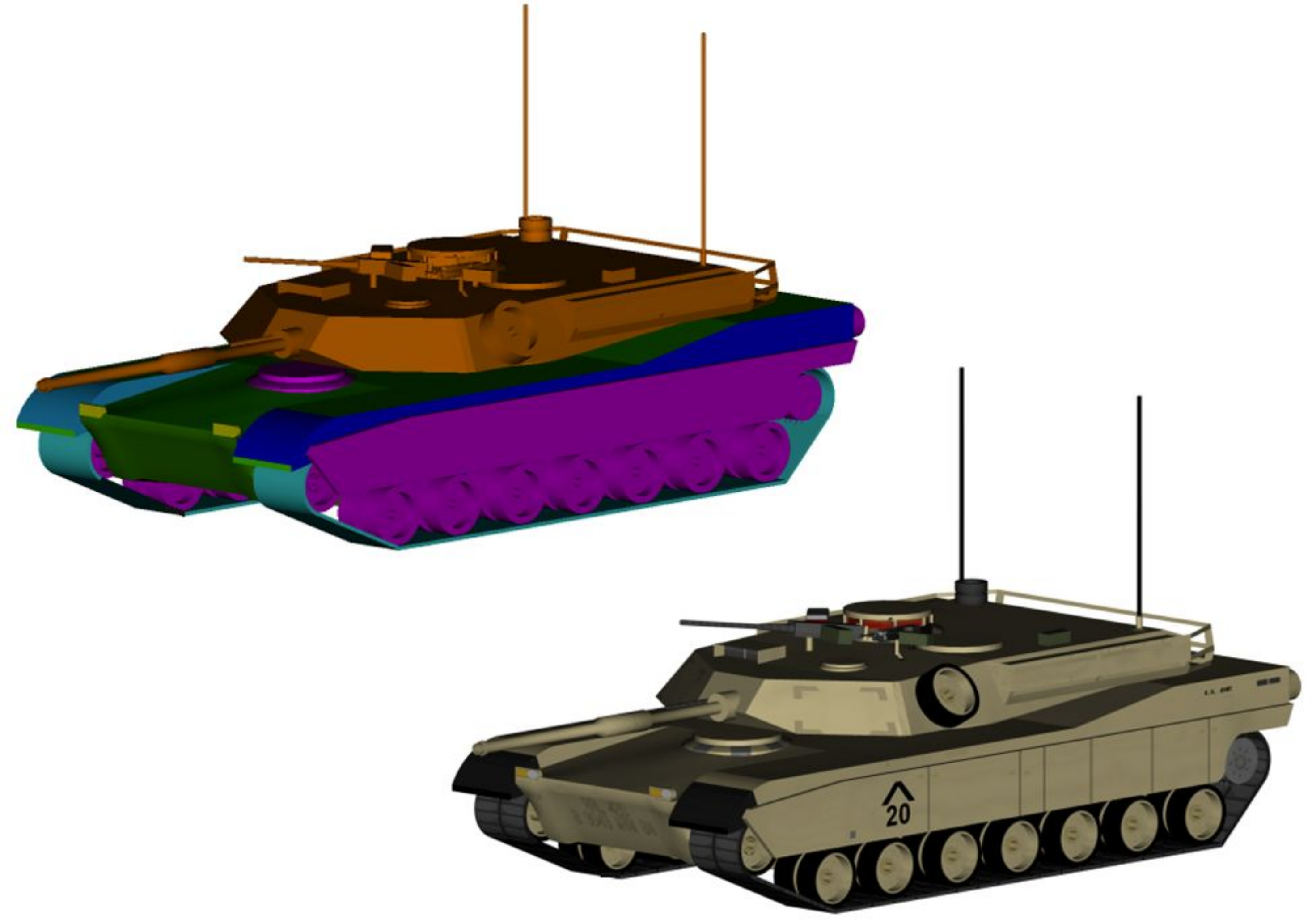
- So, given barycentric coordinates of the pixel, α_0' and α_1', we can compute:

$$\alpha_0 = \frac{z_1z_2\alpha_0'}{z_0z_1+z_1(z_2-z_0)\alpha_0'+z_0(z_2-z_1)\alpha_1'} \quad \text{and} \quad \alpha_1 = \frac{z_0z_2\alpha_1'}{z_0z_1+z_1(z_2-z_0)\alpha_0'+z_0(z_2-z_1)\alpha_1'}$$

- Then α_0 and α_1 (and α_2) can be used to find the (unknown) corresponding point p on the world space triangle
- We use α_0 and α_1 to compute z (as well as $z' = n + f - \frac{fn}{z}$) for the pixel (not α_0' and α_1')

Assigning Texture Coordinates

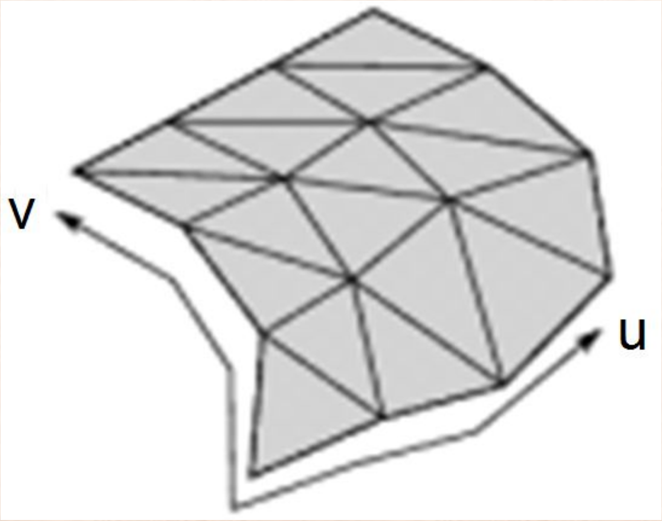
- Assign texture coordinates on complex objects one part/component at a time

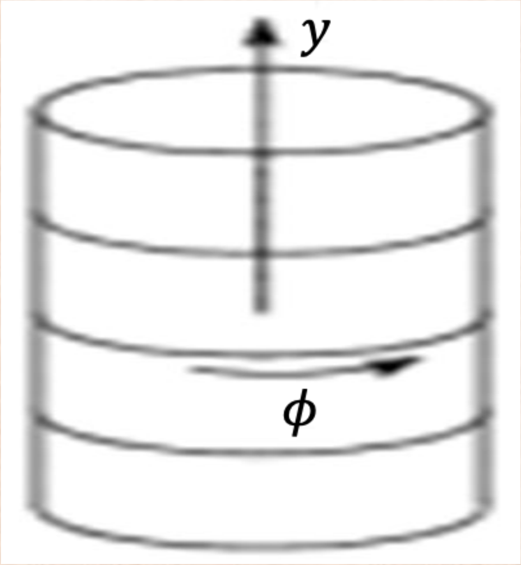


Assigning Texture Coordinates

- For complex surfaces, manually assigning (u, v) one vertex at a time can be tedious

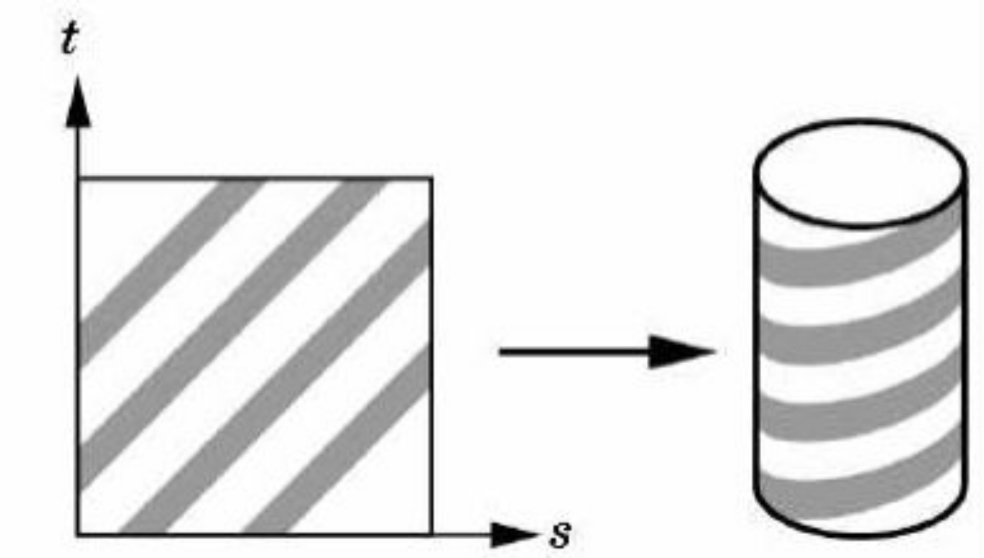
- For some surfaces, the (u, v) texture coordinates can be generated procedurally
- E.g. Cylinder (wrap the image around the outside)
    - map the $[0,1]$ values of the u coordinate to $[0,2\pi]$ for ϕ
    - map the $[0,1]$ values of the v coordinate to $[0, h]$ for y

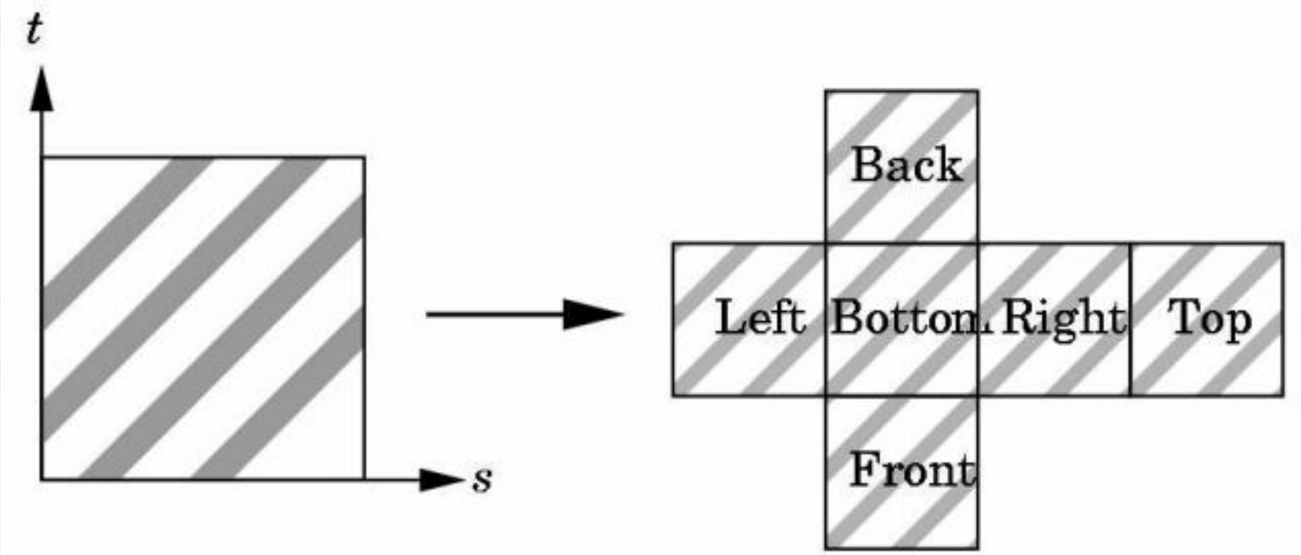




Proxy Objects – Step 1

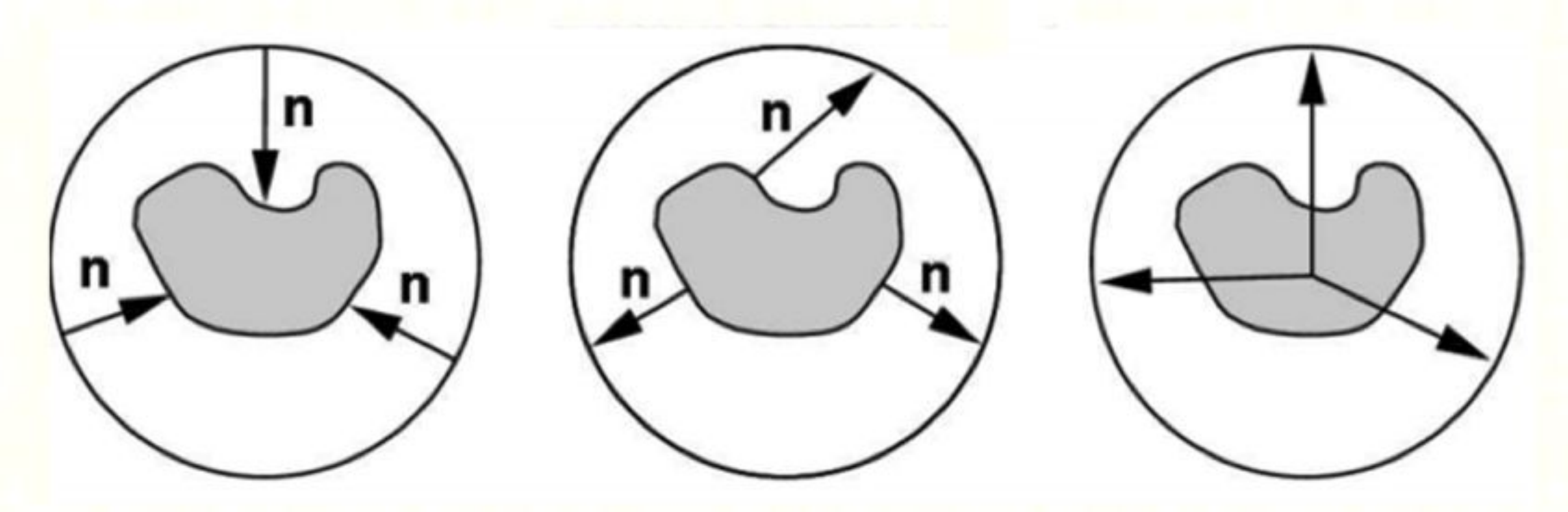
- Assign texture coordinates to intermediate/proxy objects:
  - Example: Cylinder
    - wrap texture coordinates around the outside of the cylinder
    - not the top or bottom (to avoid distorting the texture)
  - Example: Cube
    - unwrap cube, and map texture coordinates over the unwrapped cube
    - texture is seamless across some of the edges, but not necessarily other edges





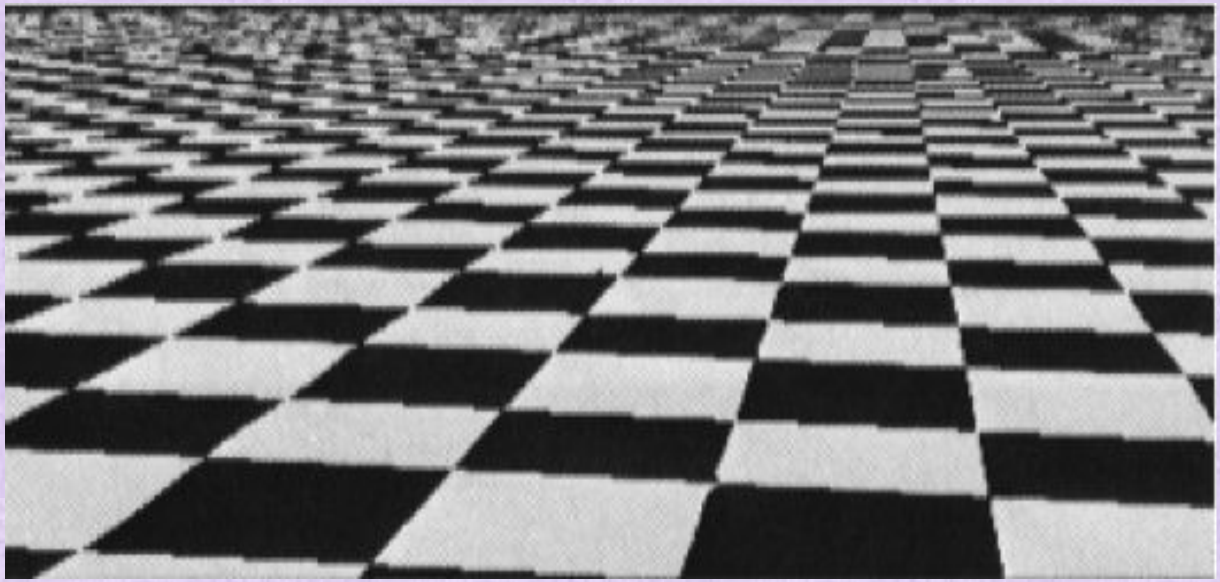
Proxy Objects – Step 2

- Next, map the texture coordinates from the intermediate/proxy object to the final object
- Three ways of doing this:
  - Use the intermediate/proxy object's surface normal
  - Use the target object's surface normal
  - Use rays emanating from a "center"-point or "center"-line of the target object

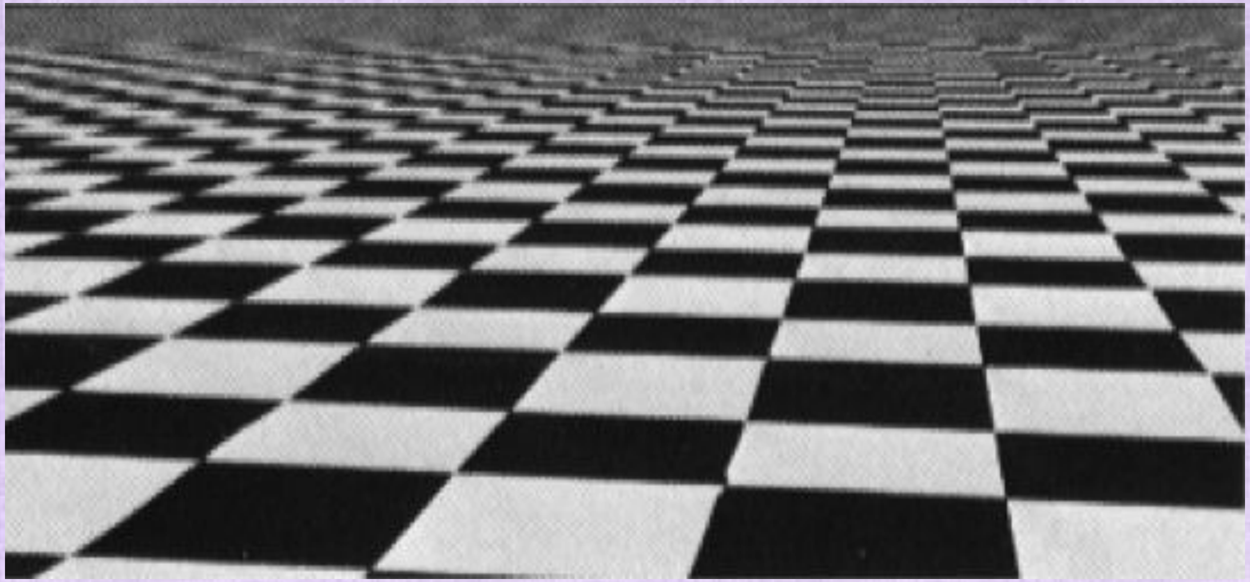


Aliasing

- When textures are viewed from a distance, they may alias



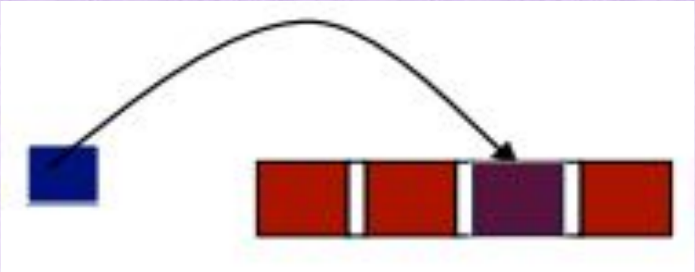
incorrect



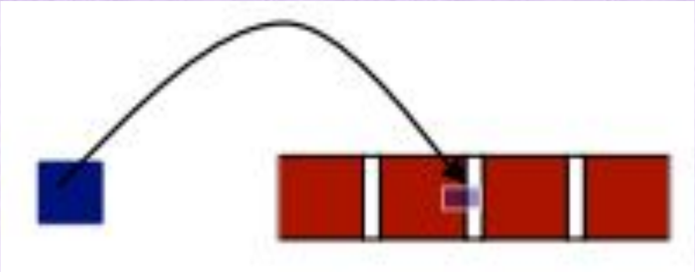
correct

Aliasing

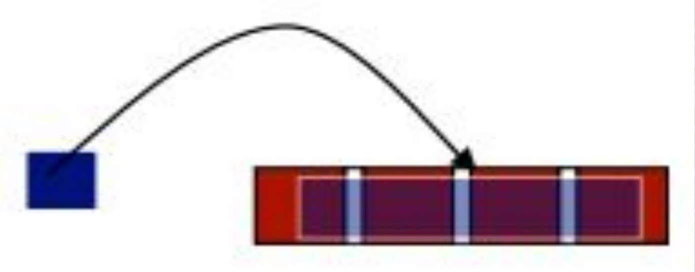
- Texture mapping is point sampling:
    - If the pixel sampling frequency is too low compared to the signal frequency (the texture resolution), aliasing can occur
- At an optimal distance, there is a 1 to 1 mapping from triangle pixels to texture pixels (texels)
- At closer distances, each triangle pixel maps to a small part of a texture pixel, and there are multiple triangle pixels per texel (oversampling is fine)
- At far distances, each triangle pixel <u>should</u> map to several texture pixels, but interpolation ignores all but the nearest texels (information is lost)
    - averaging would be better!



1 to 1



pixels super-sample texels



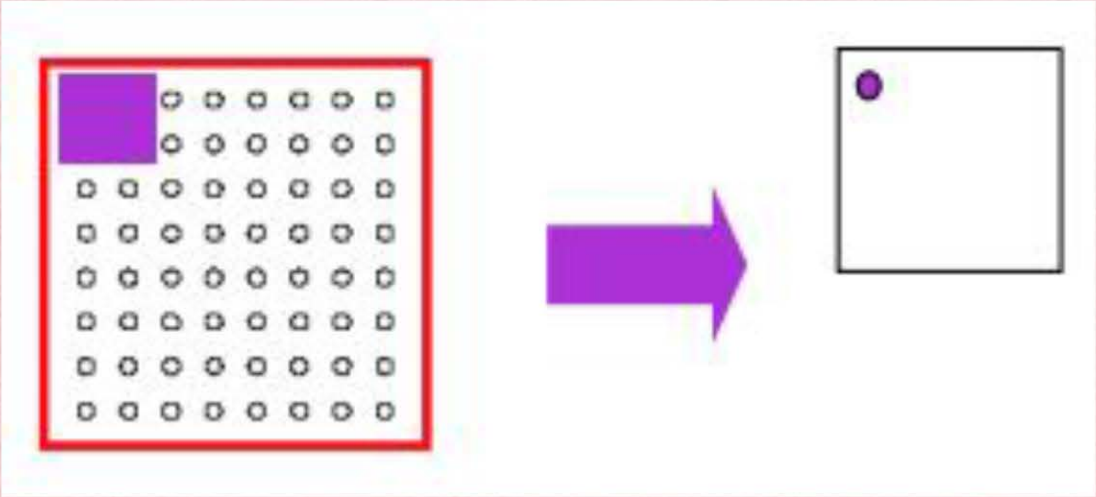
pixels under-sample texels

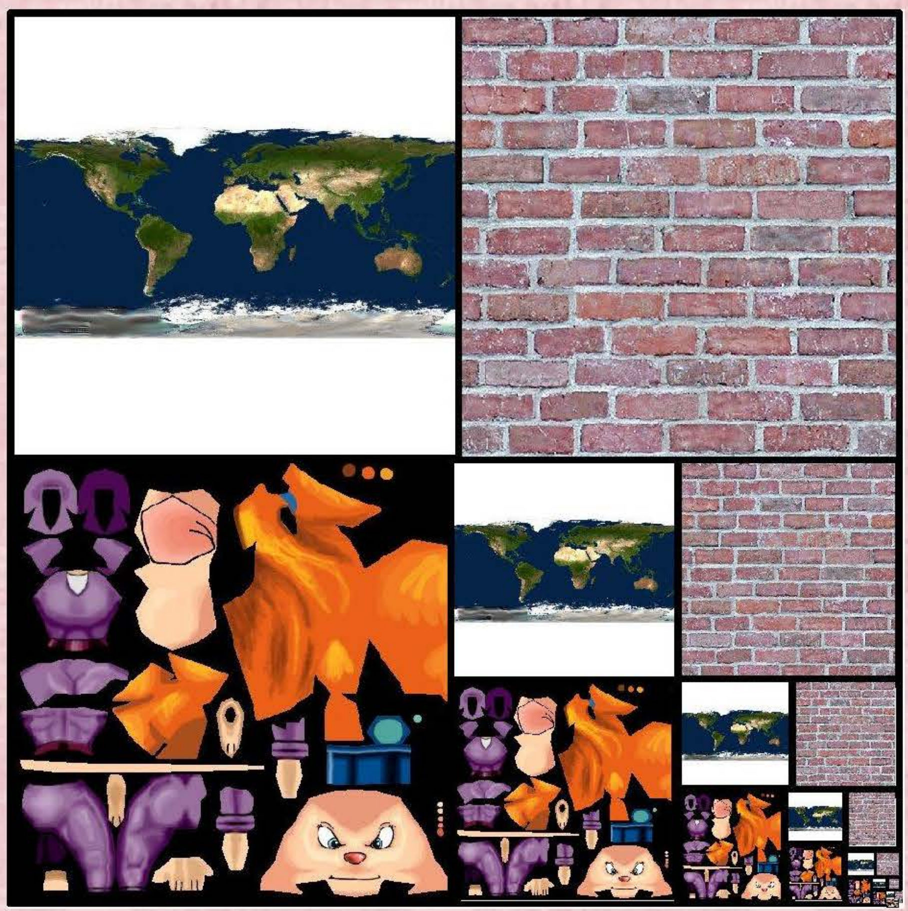
MIP Maps

- Multum in Parvo: Much in little, many in small places
- Precompute texture maps at multiple resolutions, using <u>averaging</u> as a low pass filter
- When texture mapping, choose the image size that approximately gives a 1 to 1 pixel to texel correspondence
- The averaging "bakes-in" all the nearby pixels that otherwise would not be sampled correctly

MIP Maps

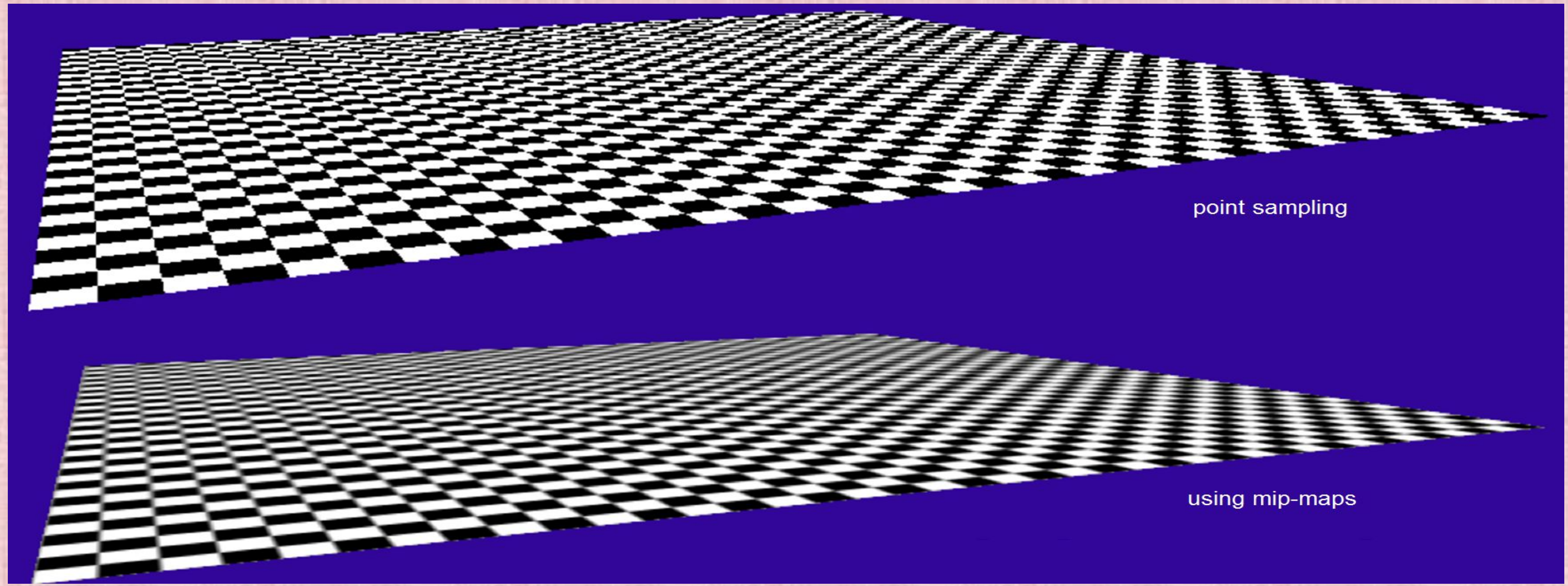
- 4 neighboring pixels of one level are averaged to form a single pixel at the next lower level
- Starting at a base resolution, can store EVERY coarser resolution in powers of 2 using only 1/3 additional space: $1 + \frac{1}{4} + \frac{1}{16} + \cdots = \frac{4}{3}$





Using MIP Maps

- Find the MIP map image just above and the image just below the screen space pixel resolution
- Use bilinear interpolation on <u>both</u> the higher/lower resolution MIP map images
- Use linear interpolation between those two bilinearly interpolated texture values, where the weights come from comparing the screen space resolution to that of the two MIP maps



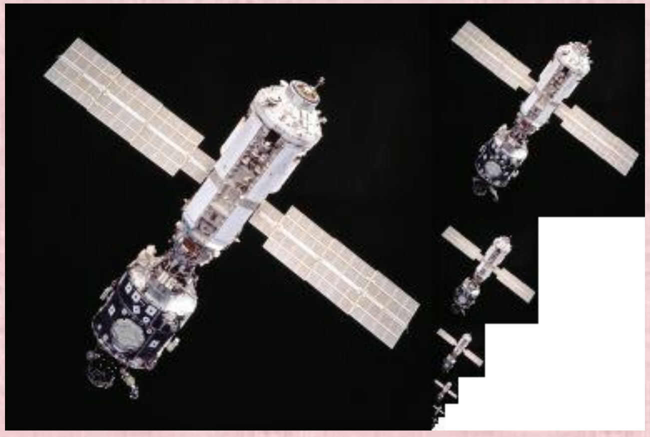
RIP Maps

- A horizontal plane at an oblique angle to the camera will have a texel sampling rate much smaller vertically than horizontally
- Using an averaged MIP map created to avoid aliasing in the vertical direction also averages in the horizontal direction (causing unwanted blurring)
- RIP mapping is an anisotropic improvement to isotropic MIP mapping that coarsens both axes separately

RIP Maps

- RIP maps require 4 times the storage if we store every coarser resolution in each axial direction:

$$\left(1+\frac{1}{4}+\frac{1}{16}+\cdots\right)\left[1+2\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots\right)\right]=4$$



MIP map

RIP map

DEBUG with checkerboard textures