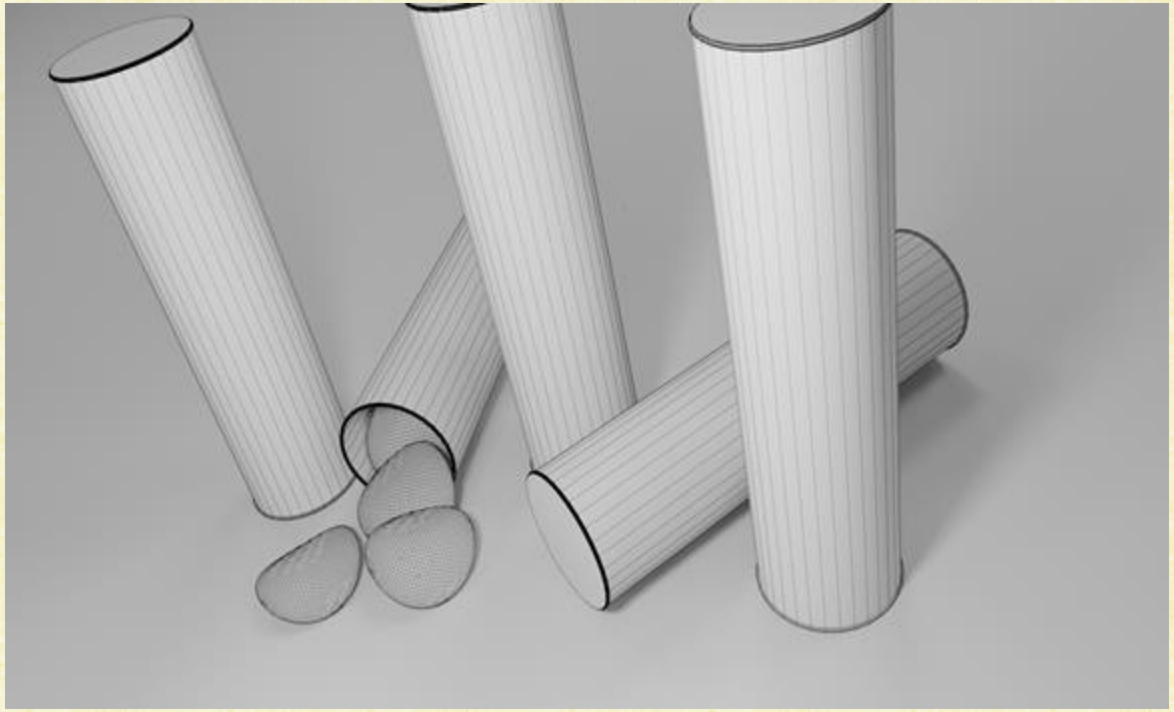
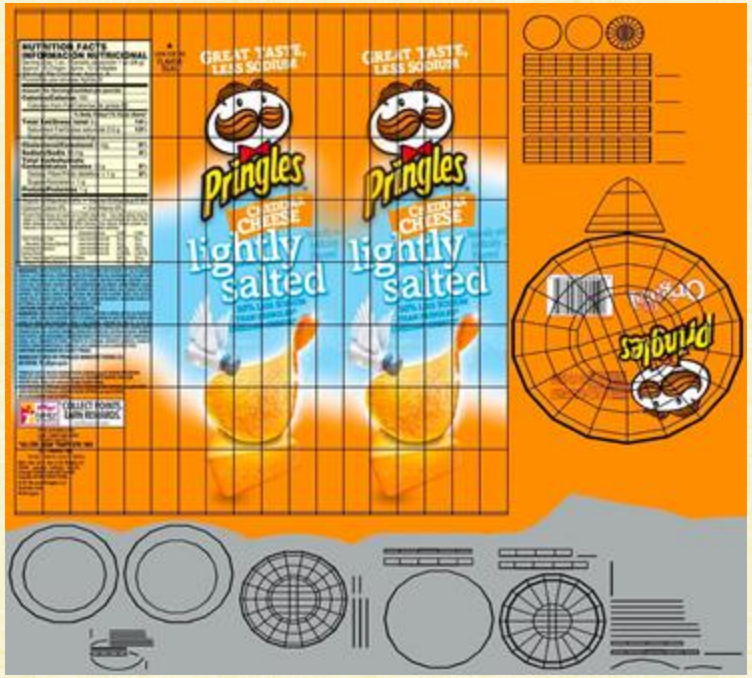
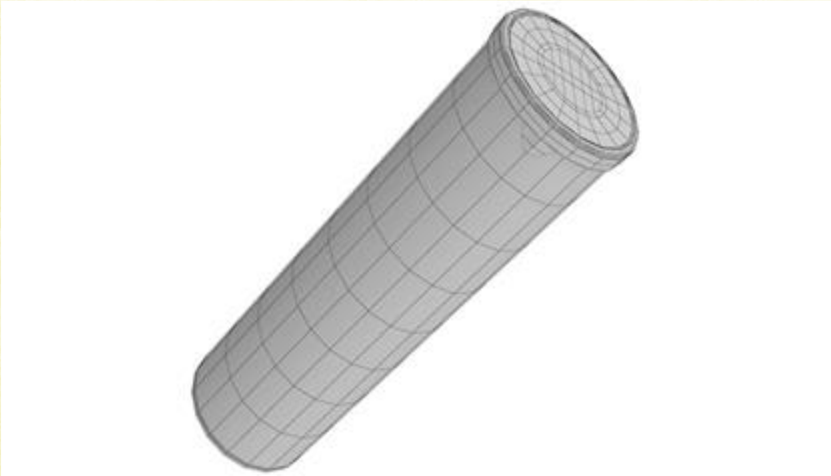
Texture Mapping



Similar to Putting Stickers onto Geometry



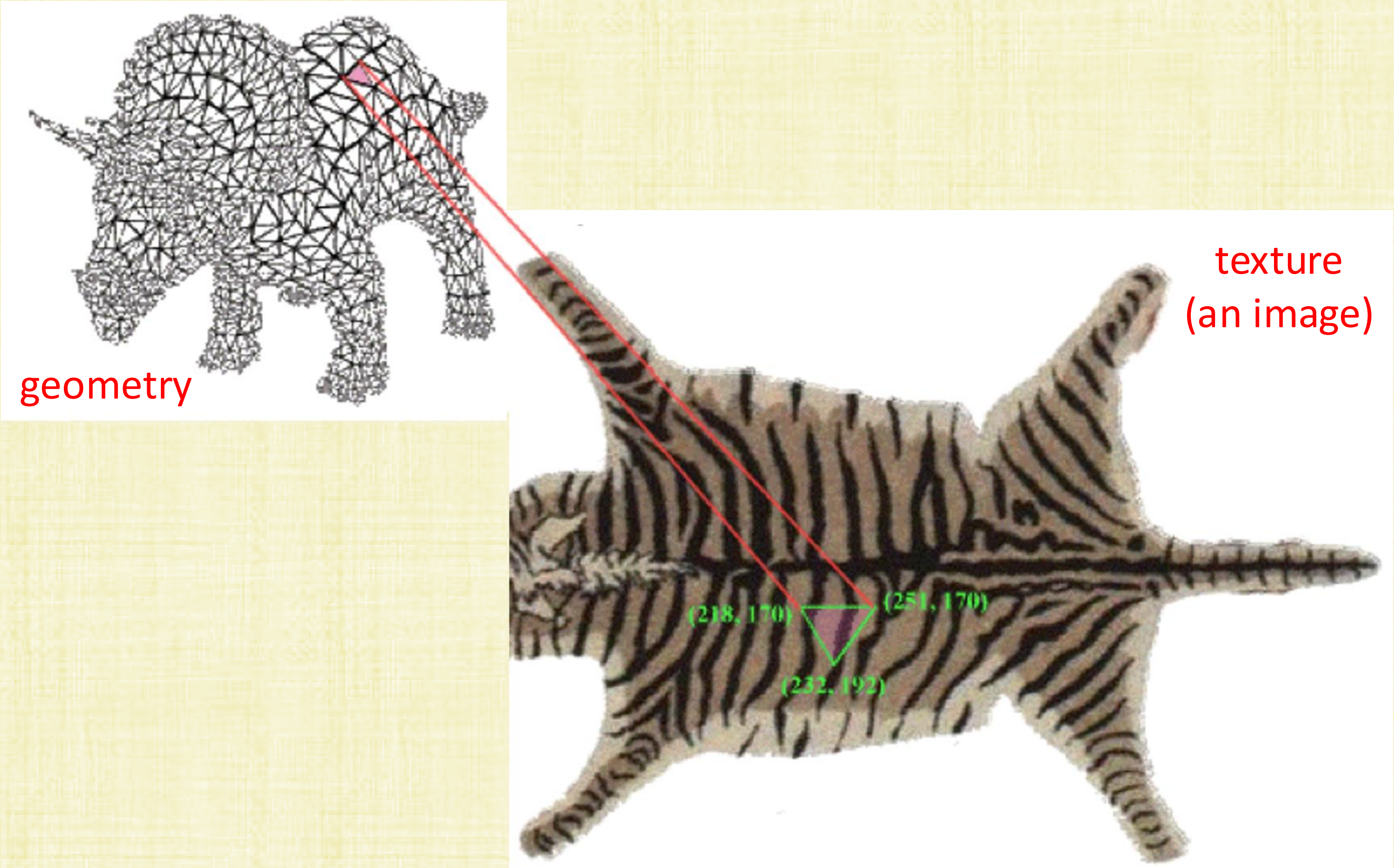
texture



geometry

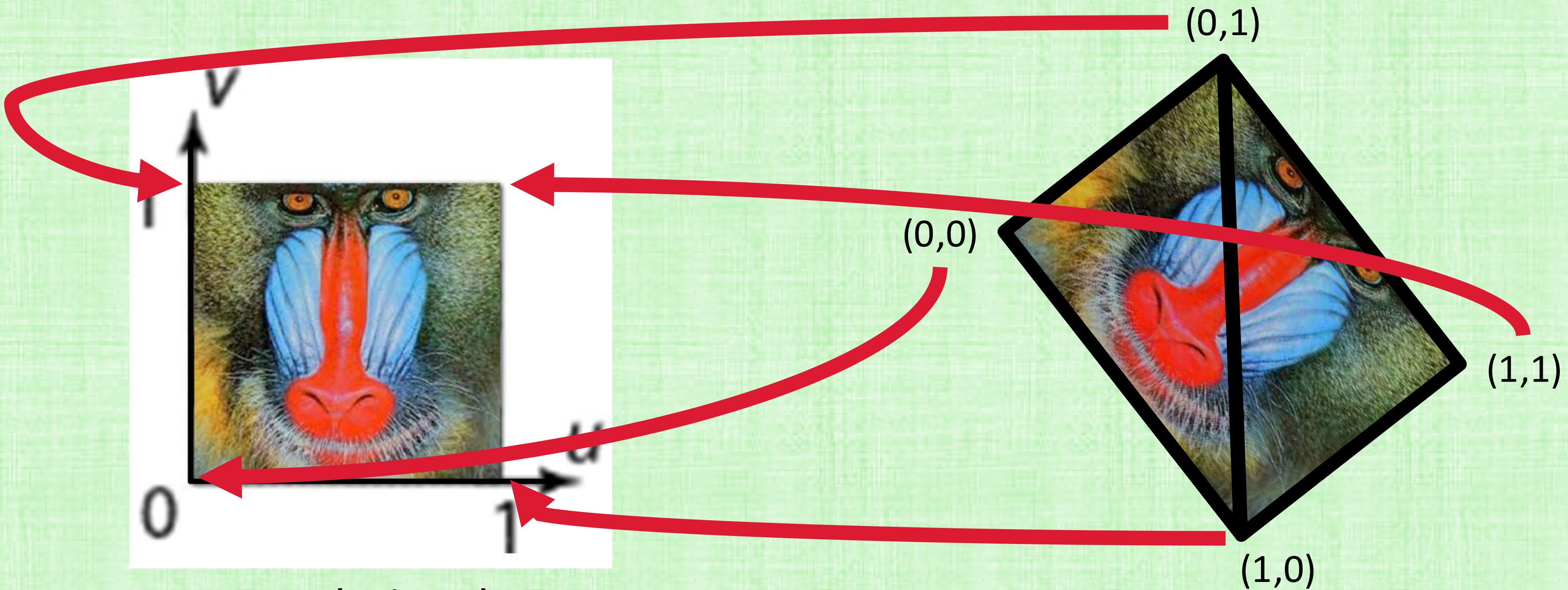
Texture Mapping

- Adds back details lost by treating the BRDF as non-varying across an object's surface
- RGB reflectance modifications are stored in an image, referred to as a texture
- Image colors are mapped to the object's surface, one triangle at a time



Texture Coordinates

- A texture is defined by an image in a 2D coordinate system: (u, v)
- Texture Mapping assigns a (u, v) coordinate to each triangle vertex
- The texture is then "stuck" onto the triangle (potentially, with distortion):
  - Let p be a point inside the triangle, with barycentric weights $\alpha_0, \alpha_1, \alpha_2$
  - The <u>color</u> assigned to p is the <u>texture color</u> at $\left(u_p, v_p\right) = \alpha_0(u_0, v_0) + \alpha_1(u_1, v_1) + \alpha_2(u_2, v_2)$
    - That is, texture coordinates are barycentrically interpolated



texture (an image)

geometry (2 triangles)

Recall: Screen Space vs. World Space Barycentric Weights

(see Class 4: purple slides)

- Express the pixel p' terms of its (computable) screen space barycentric weights: $\alpha'_0, \alpha'_1, \alpha'_2$
- Express the point p that projects to p' in terms of its unknown world space barycentric weights: $\alpha_o, \alpha_1, \alpha_2$
- Project p into screen space and set the result equal to p'
- Can solve for $\alpha_o, \alpha_1, \alpha_2$ to obtain:

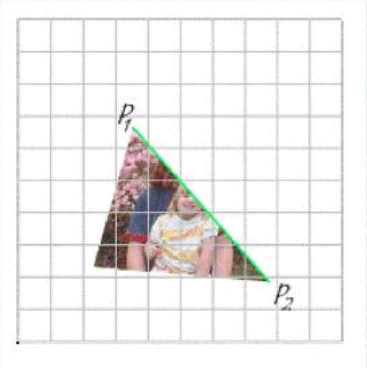
$$\alpha_0 = \frac{z_1 z_2 \alpha'_0}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2}$$

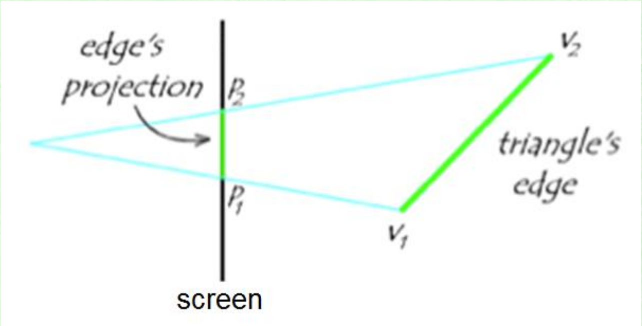
$$\alpha_1 = \frac{z_0 z_2 \alpha'_1}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2}$$

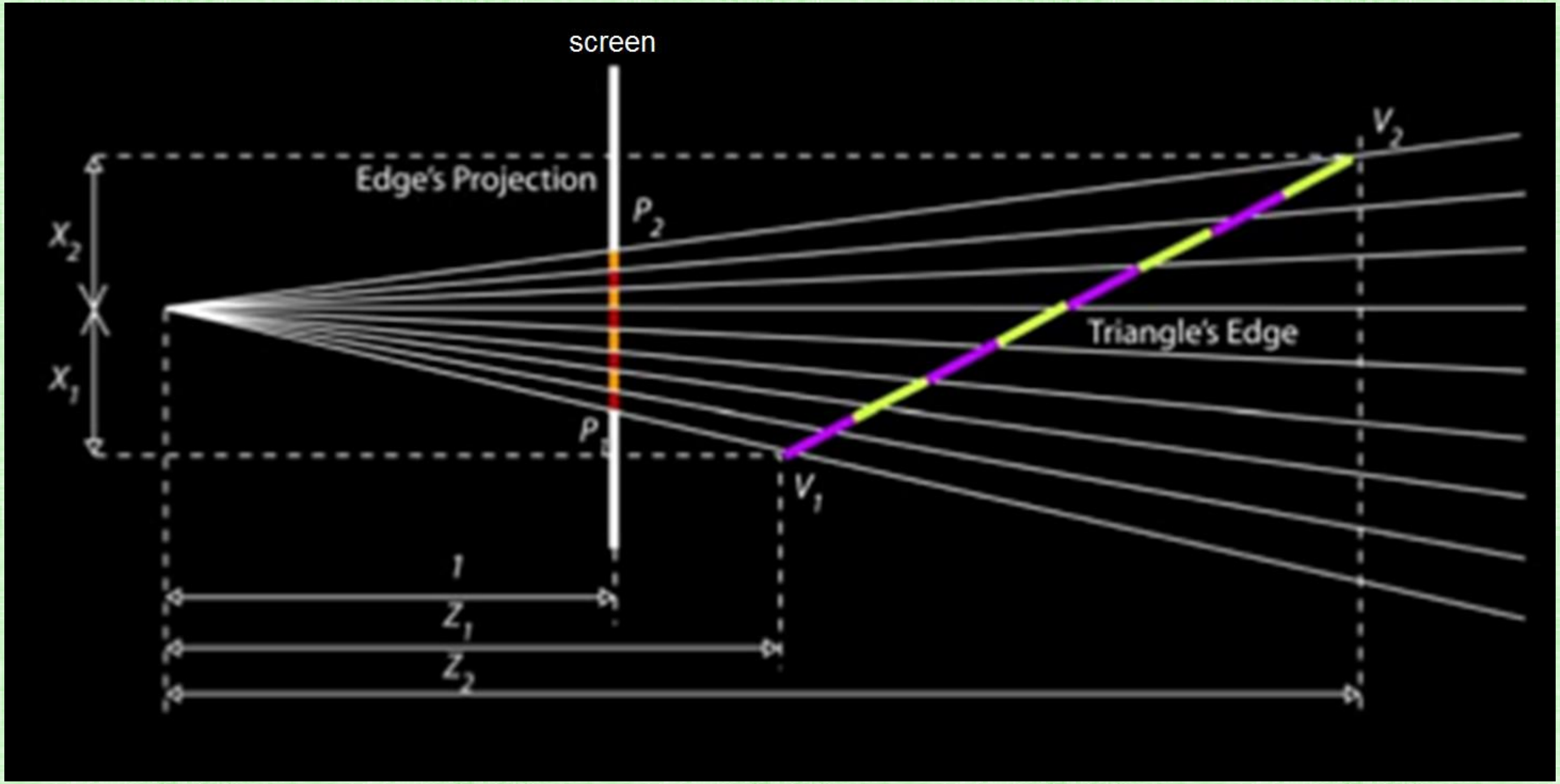
$$\alpha_2 = \frac{z_0 z_1 \alpha'_2}{z_1 z_2 \alpha'_0 + z_0 z_2 \alpha'_1 + z_0 z_1 \alpha'_2}$$

Distortion from Screen Space Projection

- Consider a single edge of one triangle
- Uniform increments along the edge in screen space (orange/red) do not correspond to uniform increments along the edge in world space (green/purple)

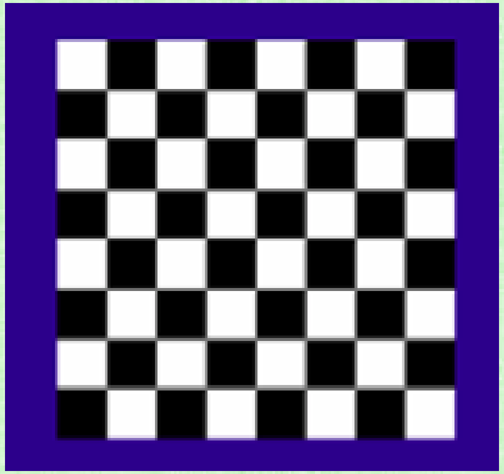




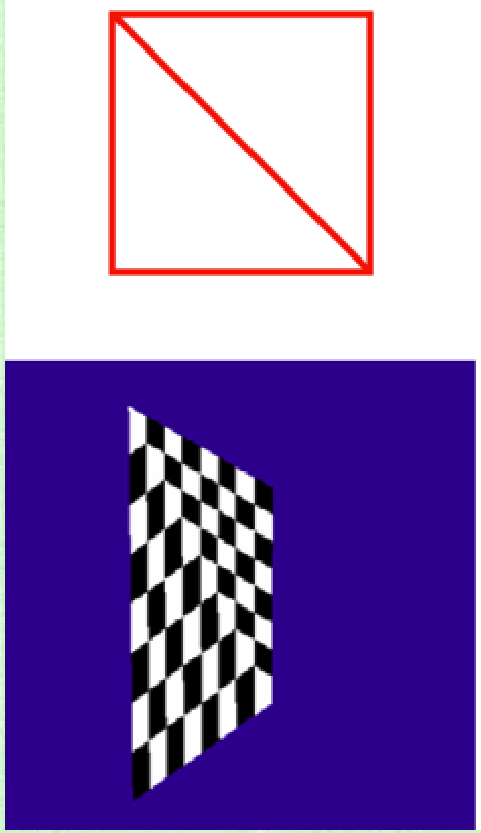


Screen Space vs. World Space Barycentric Weights

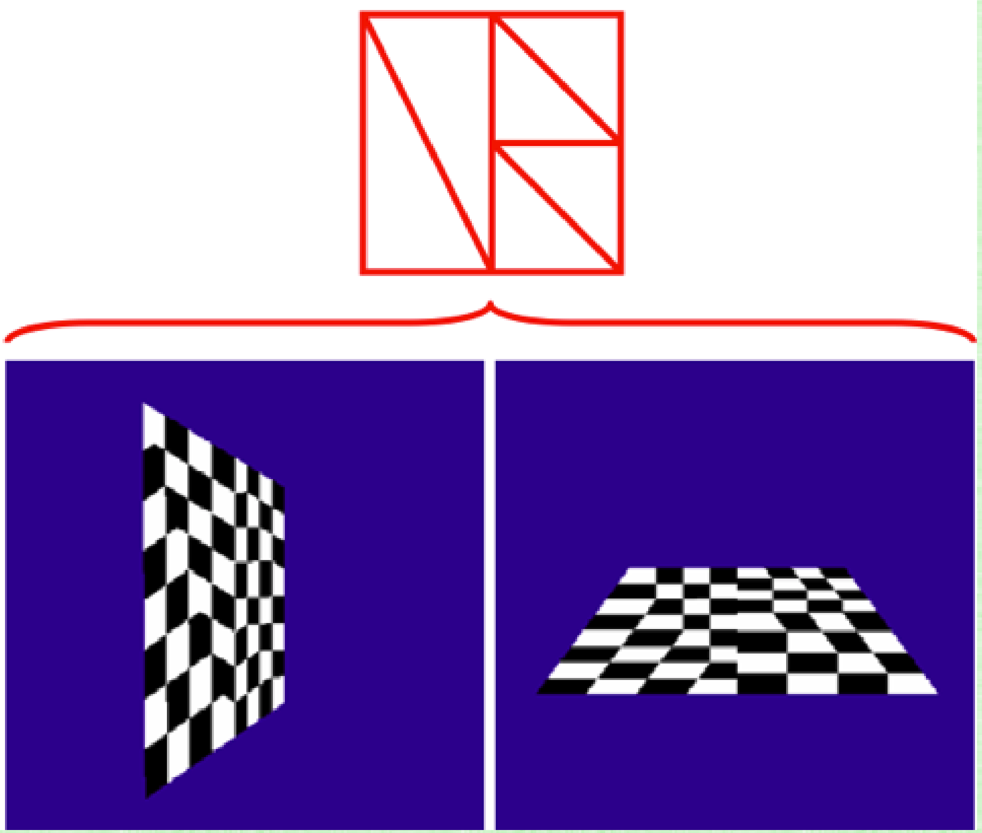
- Perspective transformation (nonlinearly) changes triangle shape
- So, interpolating texture coordinates in screen space (nonlinealy) distorts textures



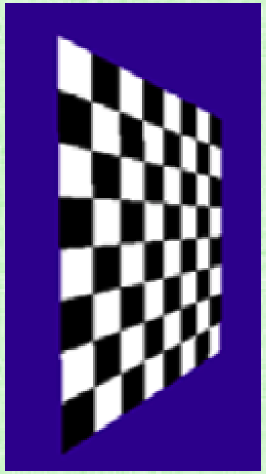
texture



screen space
barycentric weights



mesh refinement helps
(less $\frac{1}{z}$ variance per triangle)

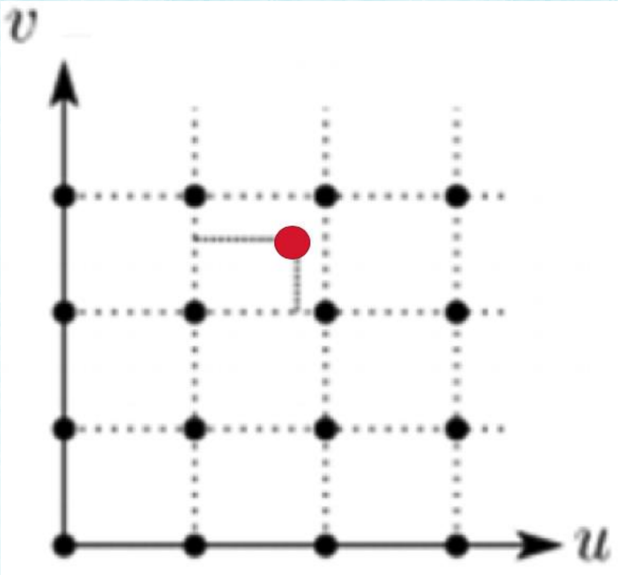


world space
barycentric weights

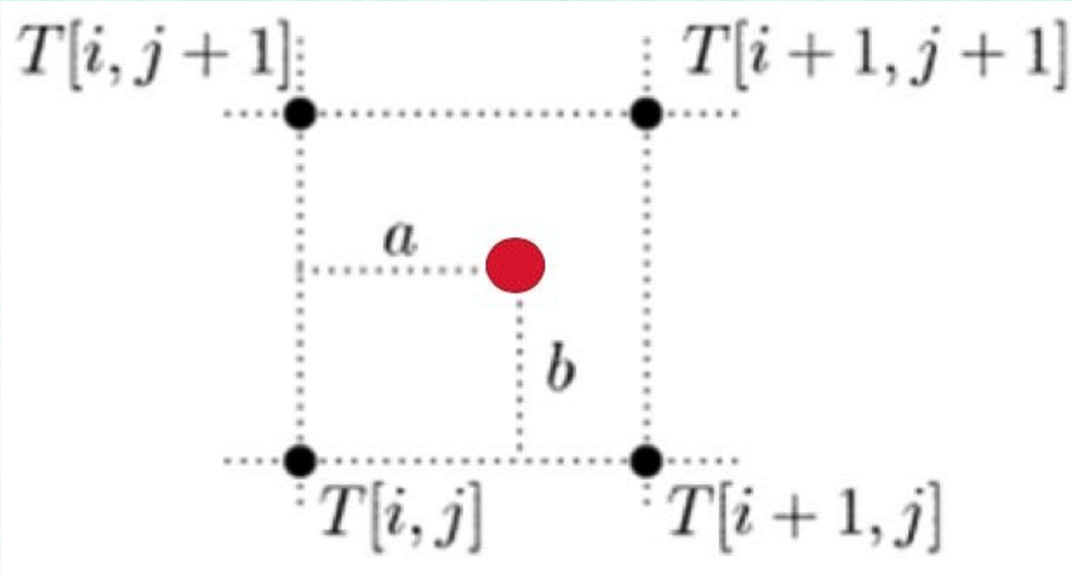
Interpolating within the Texture

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

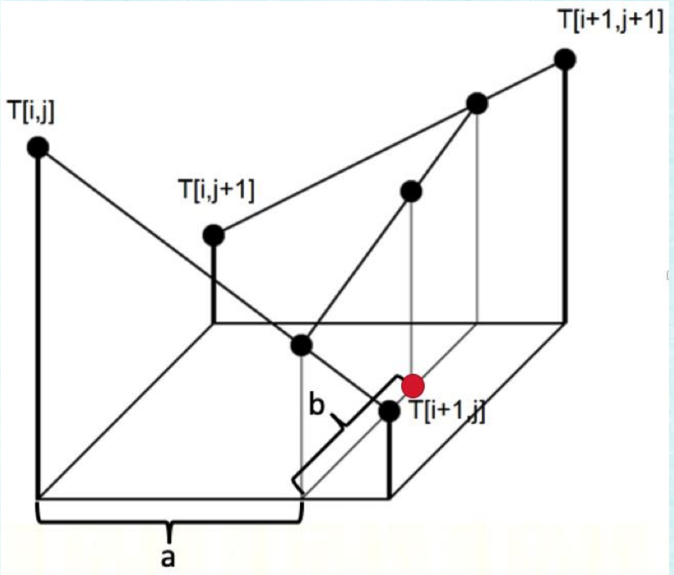
$$T(u_p, v_P) = (1-a)(1-b)T_{i,j} + a(1-b)T_{i+1,j} + (1-a)bT_{i,j+1} + abT_{i+1,j+1}$$



texture image



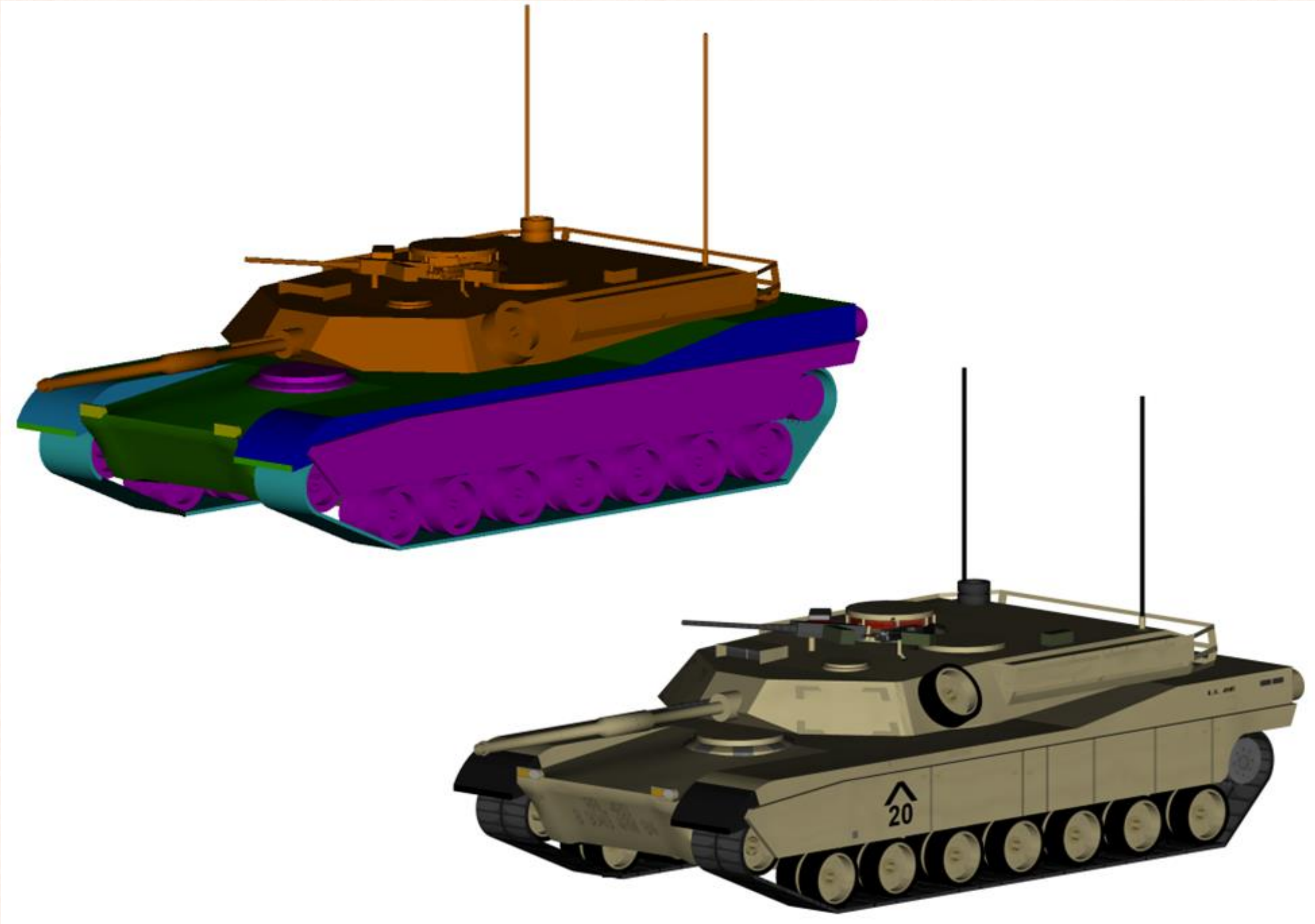
close-up view (of the 4 surrounding pixels)



bilinear interpolation

Assigning Texture Coordinates

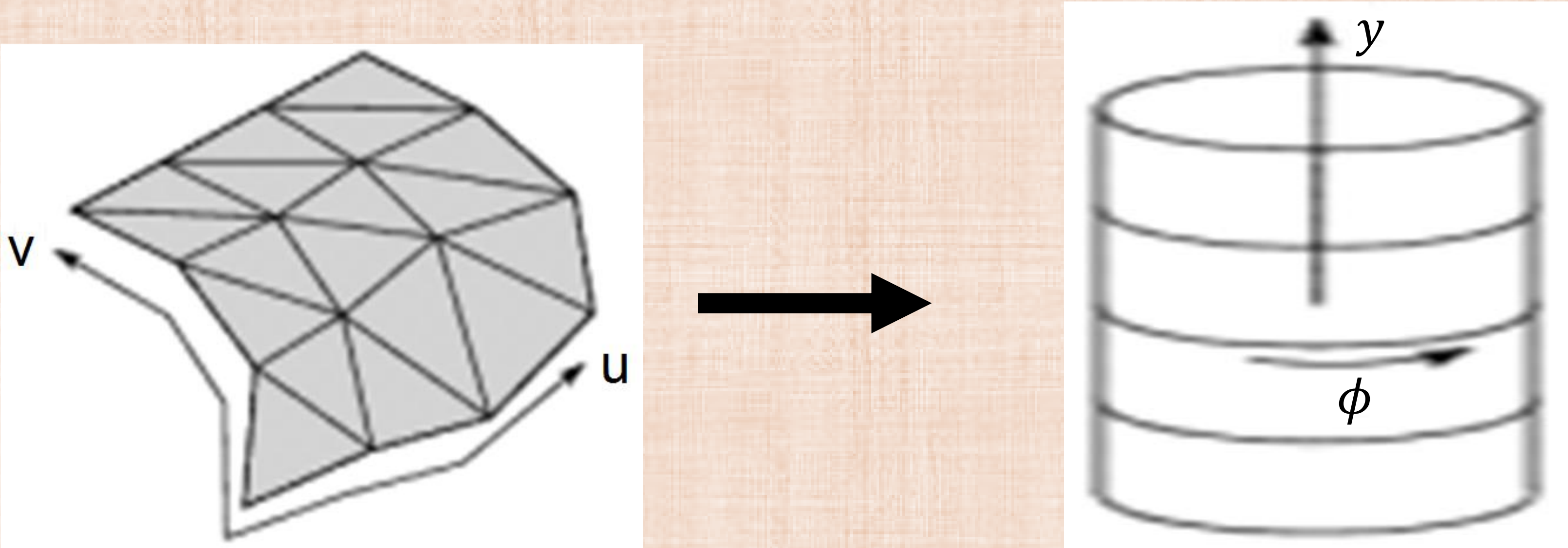
- Assign texture coordinates on complex objects one part/component at a time



Assigning Texture Coordinates

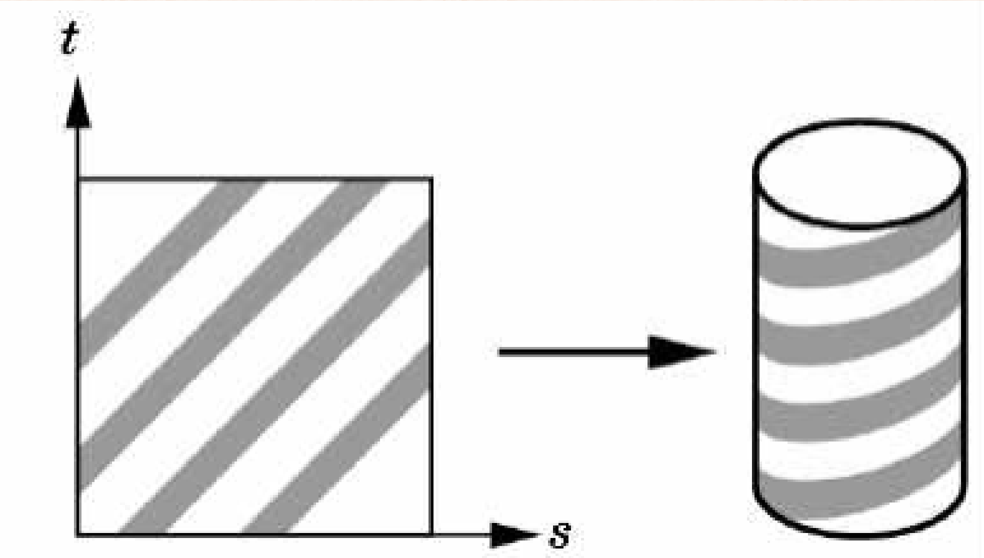
- Manually assigning (u, v) one vertex at a time can be tedious

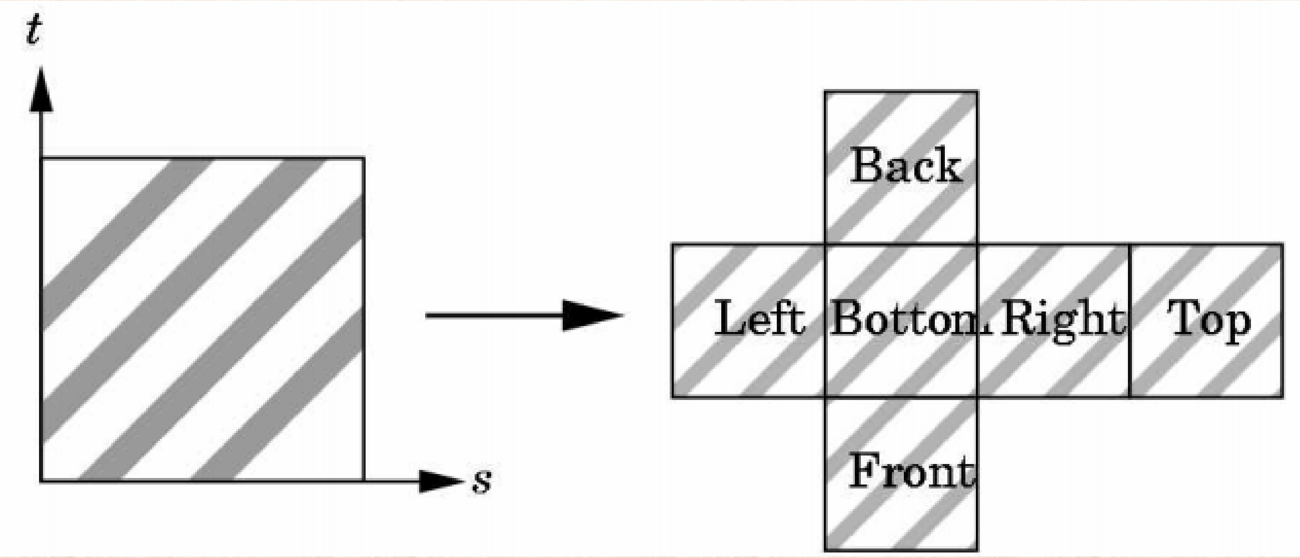
- For some surfaces, the (u, v) texture coordinates can be generated procedurally
- E.g. Cylinder - wrap the image around the outside
  - map the $[0,1]$ values of the u coordinate to $[0,2\pi]$ for ϕ
  - map the $[0,1]$ values of the v coordinate to $[0, h]$ for y



Using Proxy Objects – Step 1

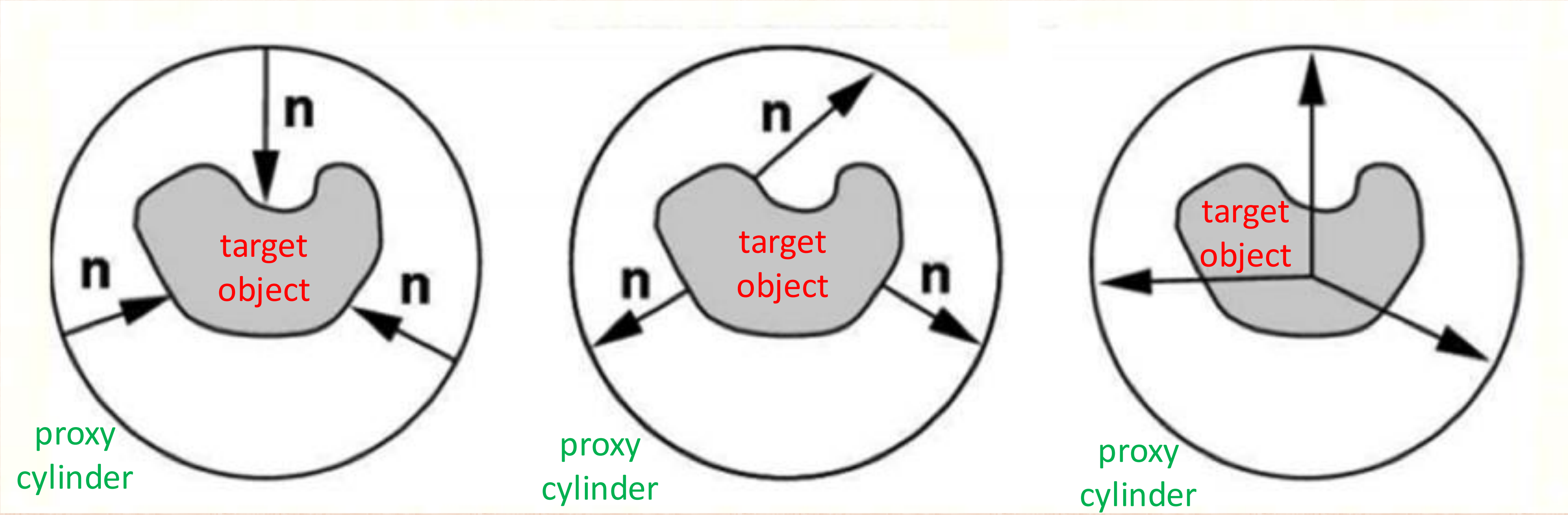
- Assign texture coordinates to proxy objects:
  - Example: Cylinder
    - wrap texture coordinates around the outside of the cylinder
    - not the top or bottom (to avoid texture distortion)
  - Example: Cube
    - unwrap cube, and map texture coordinates over the unwrapped cube
    - texture is seamless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





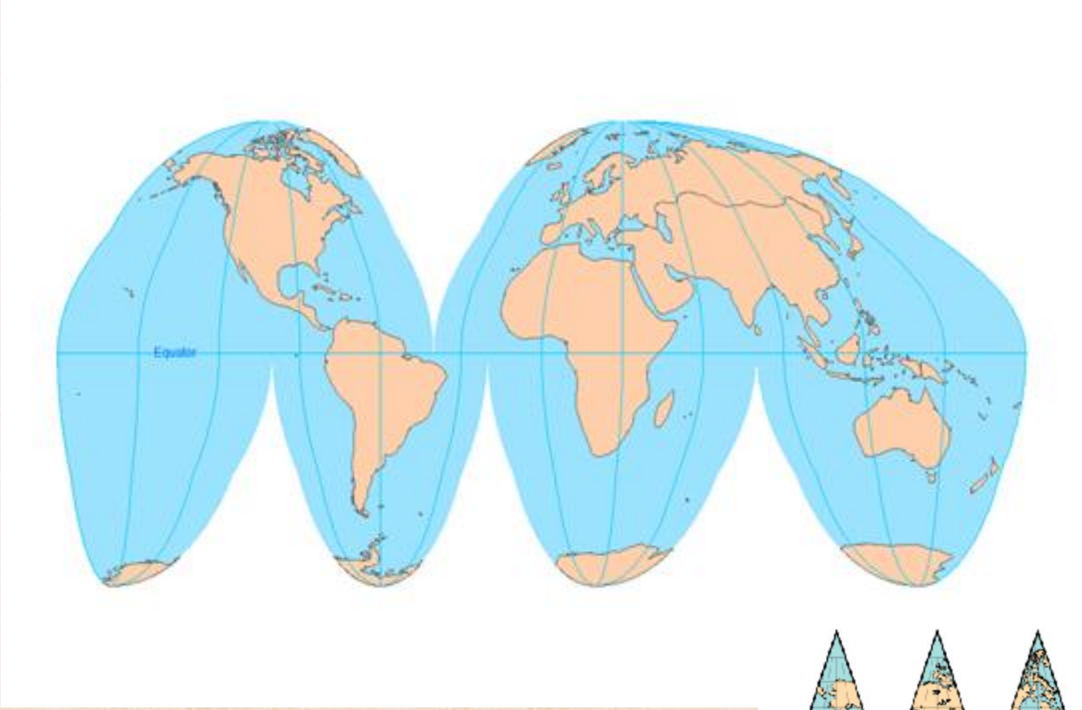
Using Proxy Objects – Step 2

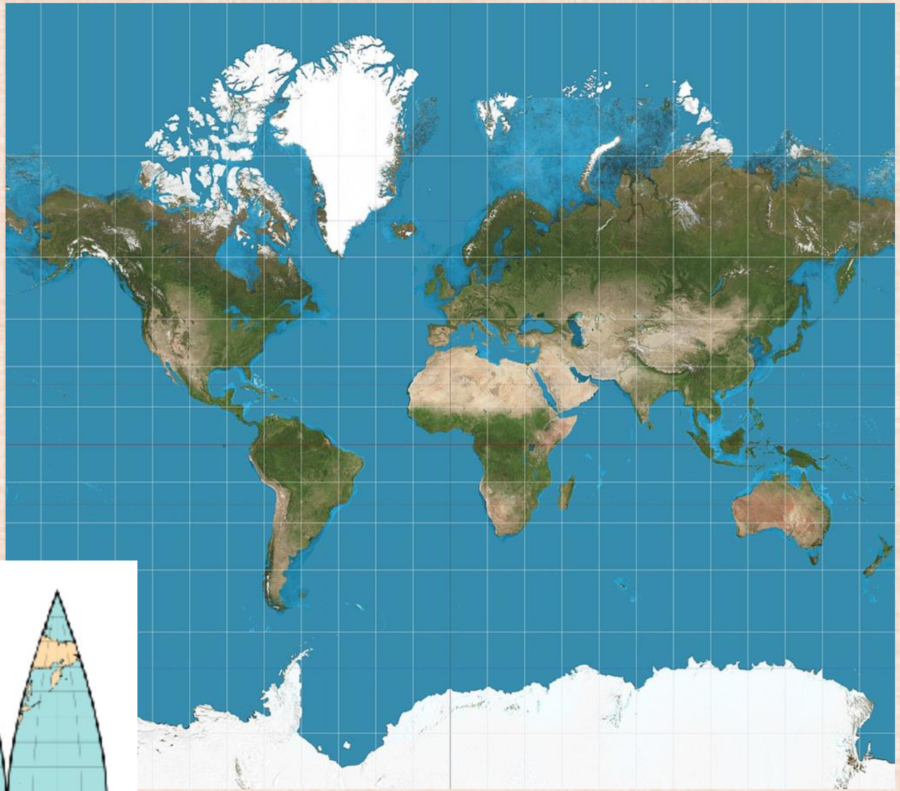
- Transfer texture coordinates from the proxy object to the final object
- Various approaches:
  - Use the proxy object's surface normal
  - Use the target object's surface normal
  - Use rays emanating from a "center"-point/line of the target or proxy object

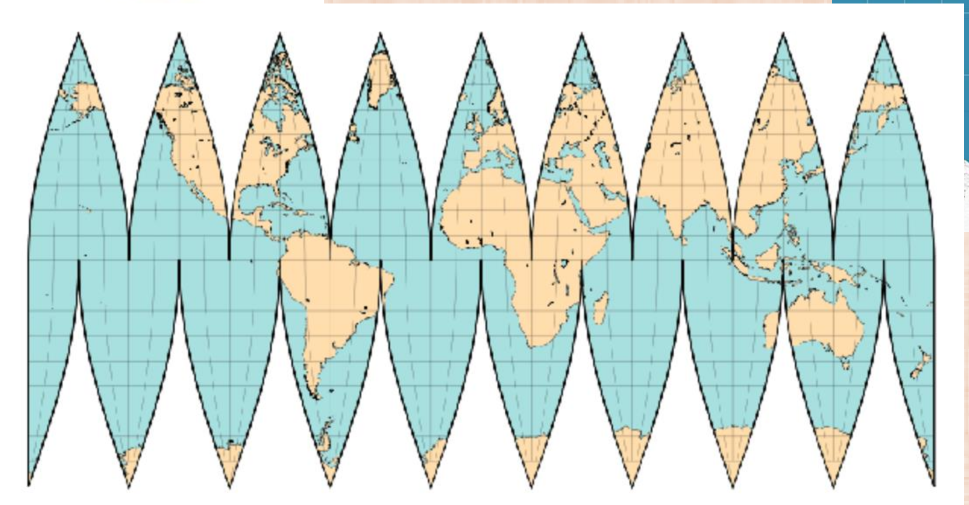


Distortion

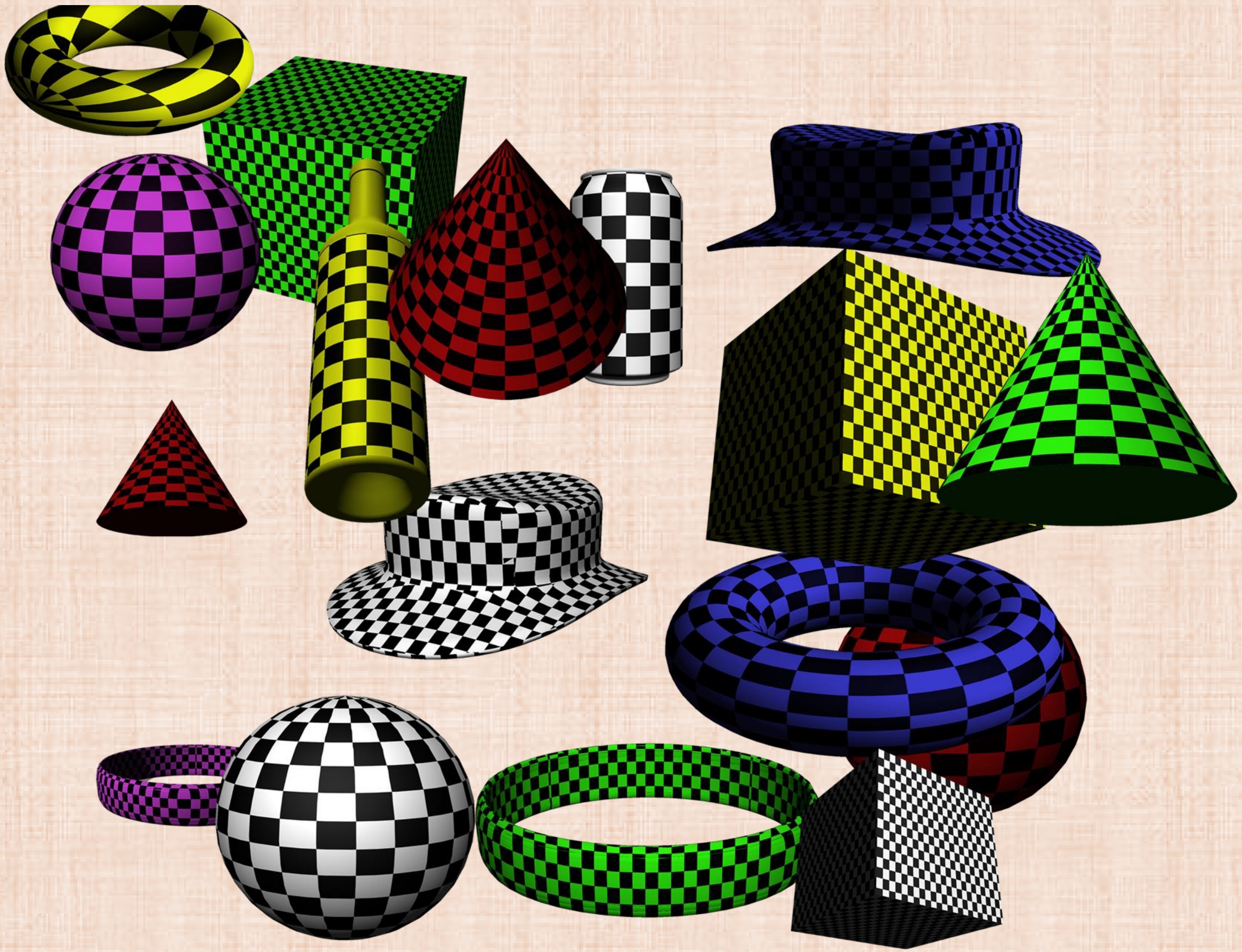
- It’s difficult to find low-distortion mappings (back and forth) from a 2D plane to 3D surfaces





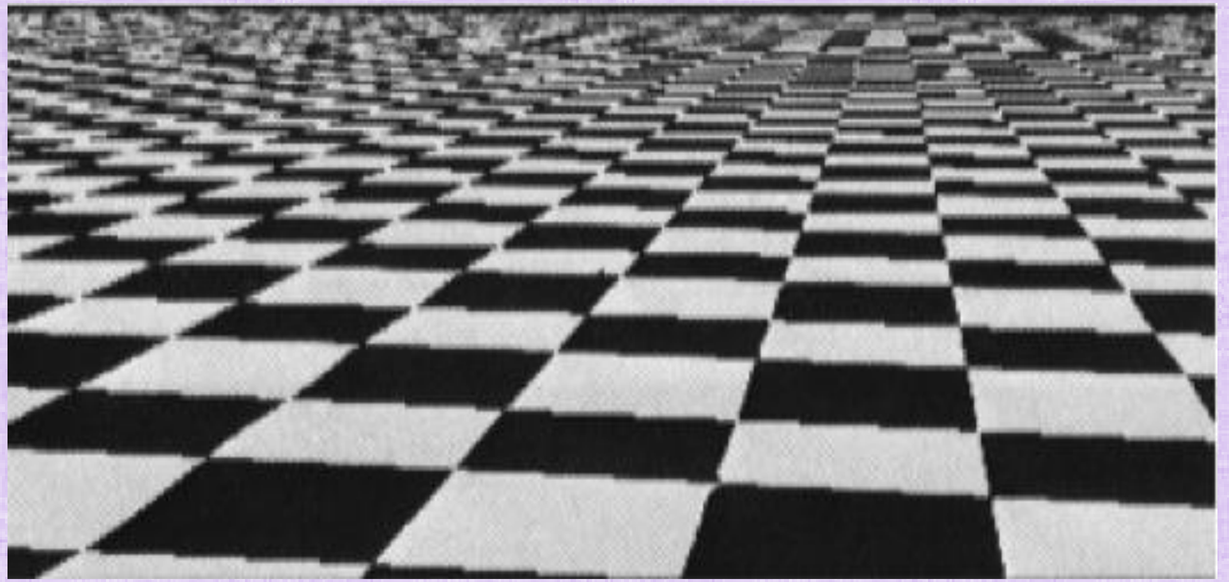


DEBUG with checkerboard textures

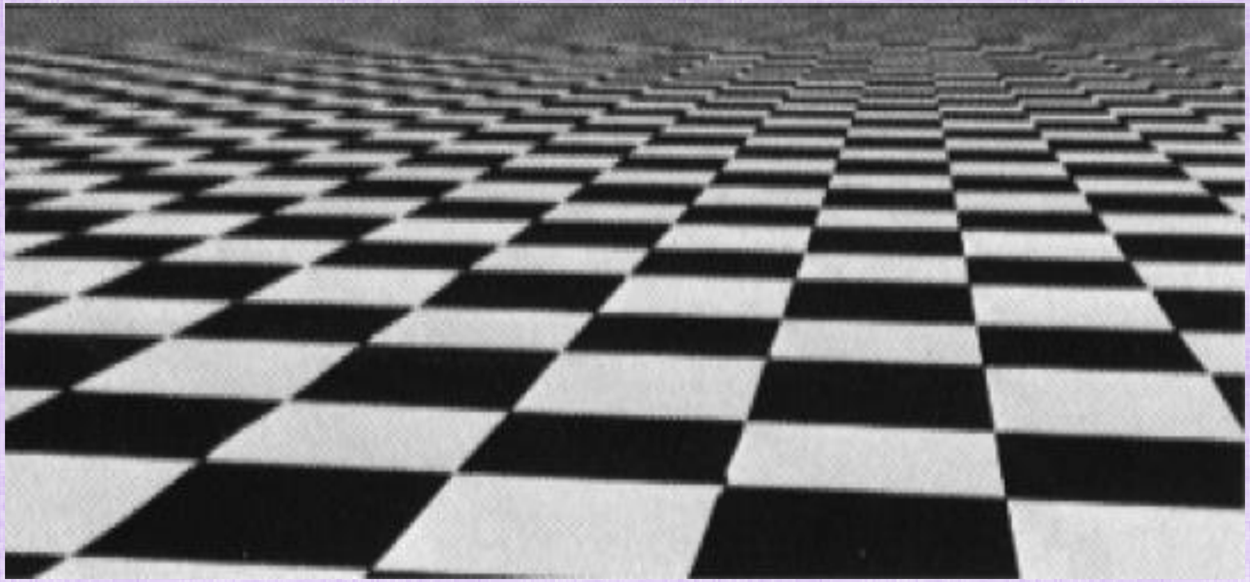


Aliasing

- Textures often alias when viewed from a distance



incorrect

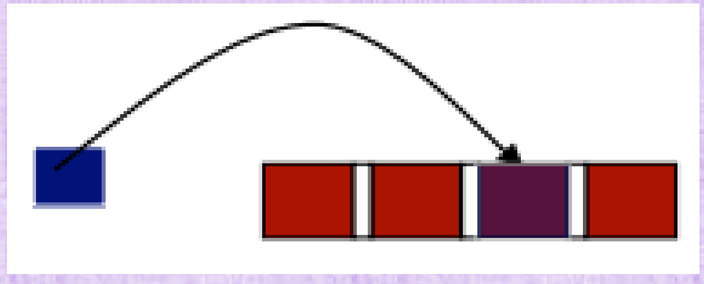


correct

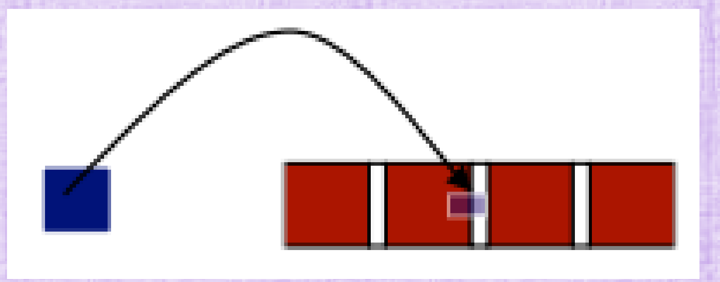
Aliasing

• Recall: aliasing occurs when the sampling frequency is too low compared to the signal frequency (which is the texture resolution here)

• At an optimal distance, there is a 1 to 1 mapping from triangle pixels to texels (texture pixels)



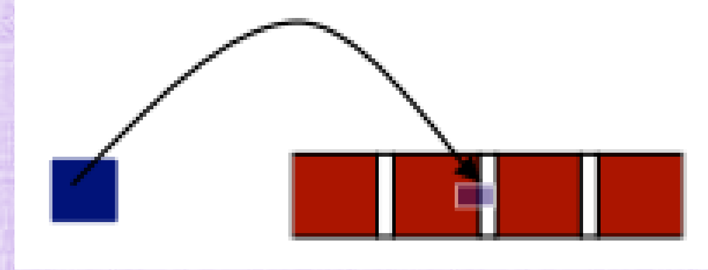
• At closer distances, the sampling frequency is higher than the texture resolution, which is the signal frequency; so, there is no aliasing
• In this case, triangle pixels (correctly) interpolate from the surrounding texels



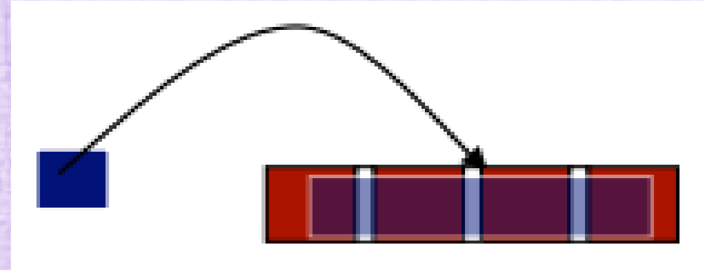
As was discussed in the prior (blue) slide

Aliasing

- At far distances, the sampling frequency is too low compared to the texture resolution
- Thus, the interpolation will result in aliasing
- In this case, interpolating values for triangle pixels from the surrounding texels causes aliasing



- Instead, a triangle pixel <u>should</u> contain (and average together) all the information from the several texels that it overlaps



- Interpolation ignores all but the neighboring texels, resulting in aliasing)

Anti-Aliasing

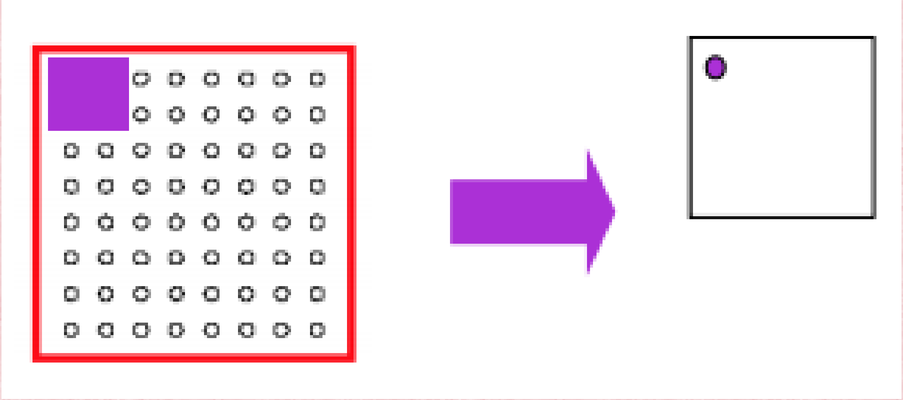
- Need to pre-process the texture image to remove the frequencies that are too high for the sampling rate to properly capture, before doing interpolation
- This is accomplished via MIP and RIP maps…

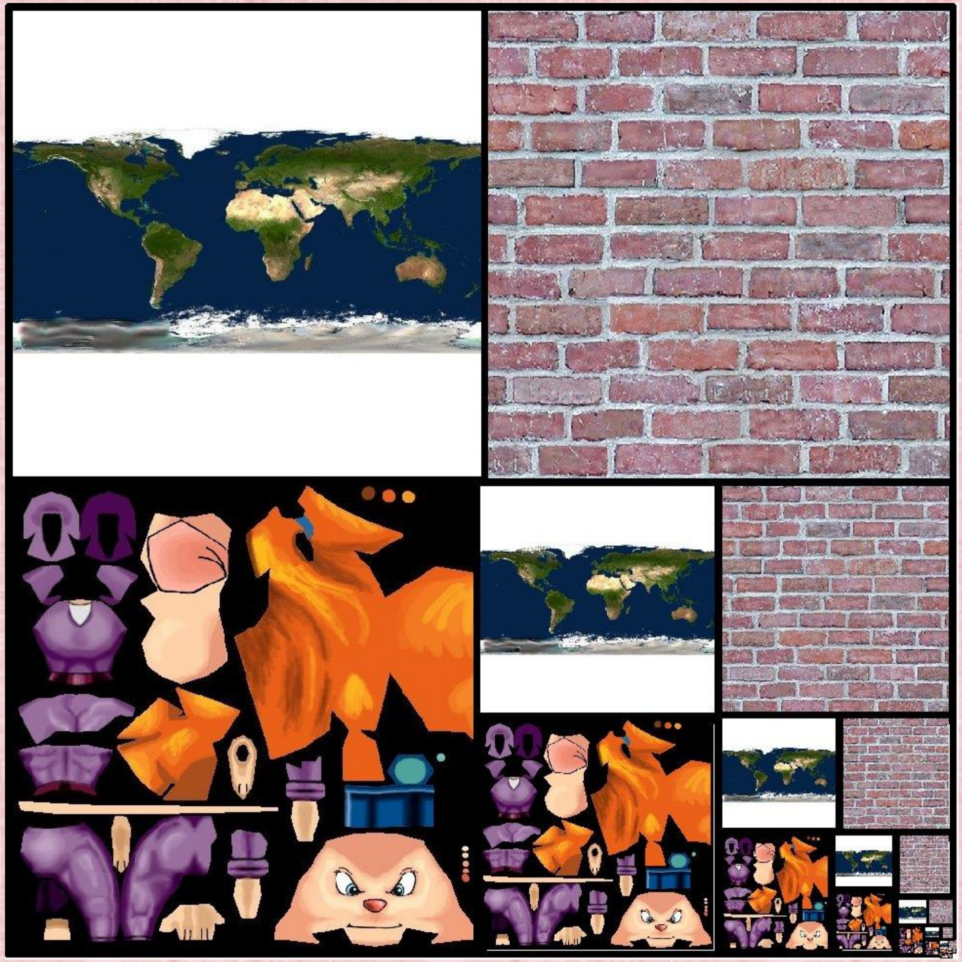
MIP Maps

- Multum in Parvo (much in little)
- Precompute texture images at multiple resolutions, using averaging as a low pass filter
- Averaging "bakes-in" nearby texels that would otherwise be ignored by the interpolation
- When texture mapping, choose the image size that (approximately) gives a 1 to 1 pixel to texel correspondence

Storing MIP Maps

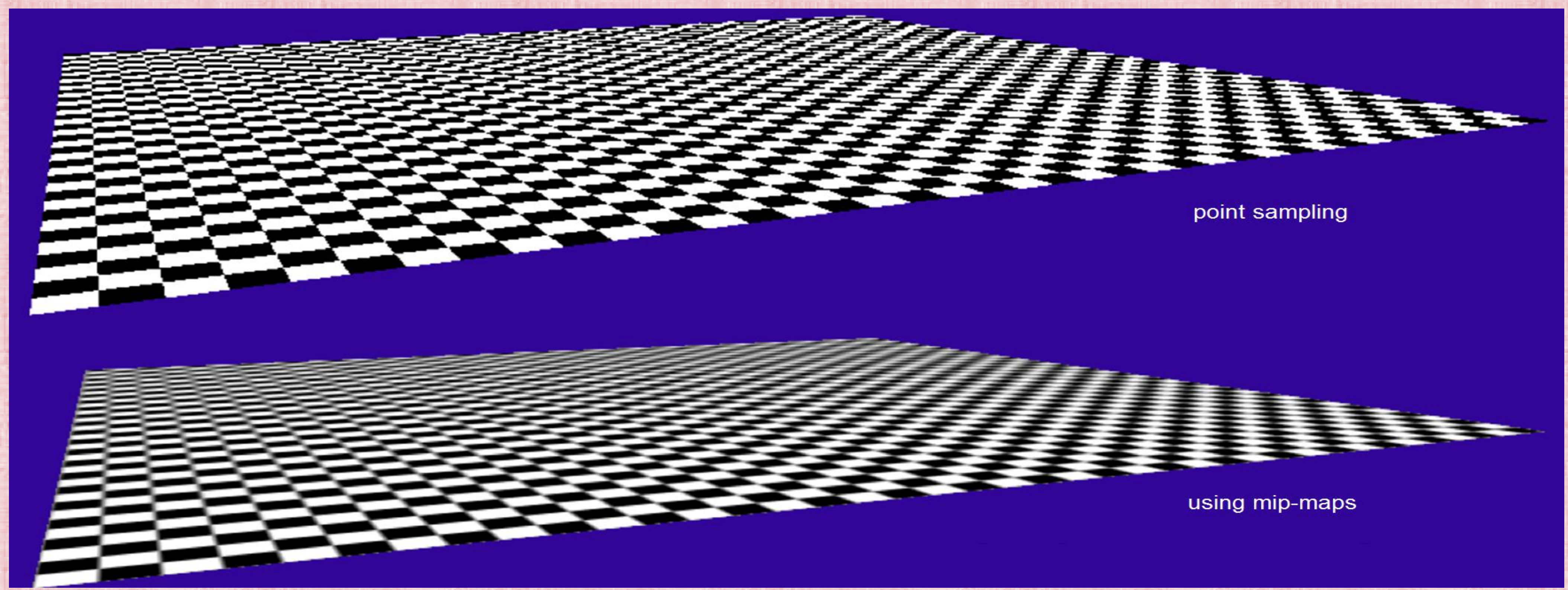
- 4 neighboring texels of one level are averaged to form a single texel at the next level
- Since $1 + \frac{1}{4} + \frac{1}{16} + \cdots = \frac{4}{3}$, can store all coarser resolutions with 1/3 additional space





Using MIP Maps

- Find the MIP map image just above and just below the screen space pixel resolution
- Use bilinear interpolation on both of the chosen MIP map images
- Then, linearly interpolate between the two results (using weights that relate the screen space resolution to that of the two MIP map images)



RIP Maps

- A triangle tilted away from the camera has different texel sampling rates in the horizontal and vertical directions
- MIP map images can only match one of those two sampling rates
- Anisotropic RIP maps are designed to account for this
- RIP maps require 4 times the storage:

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

RIP map

MIP map