A1.5: Rasterizer

- Screen Space Smooth Triangles
- Perspective Correct Interpolation
- Mip Map
- Supersampling

```
* - frag.attributes = depends on Interp_* flag in flags:
*   - if Interp_Flat: copy from va.attributes
*   - if Interp_Smooth: interpolate as if (a,b,c) is a 2D triangle flat on the screen
*   - if Interp_Correct: use perspective-correct interpolation
```

A1T3: Flat shaded

- Copied vertex attributes of the first vertex (va)

A1T5: Smooth shaded

- Interpolate the attributes (like color) using barycentric coordinates!

Screen Space Smooth Triangles

- We need to fill the Fragment struct correctly
  - Frag.attributes: The interpolated values (e.g., UVs, Normals)
  - Frag.derivatives: How those attributes change w.r.t. x and y screen coordinates
- You will need barycentric coordinates constantly, consider making a lambda helper function for this!

```
auto compute_barycentric(float x, float y) -> Vec3.
```

- Screen space smooth triangles means standard linear interpolation

```
attr = alpha * va.attr + beta * vb.attr + gamma * vc.attr
```

- Alpha beta gamma are barycentric coordinates

Computing Derivatives

- Needed for mip mapping and computing level of detail (task 6)
- Represent the rate of change of attributes per screen pixel

```
d_dx = attr(x+1, y) - attr(x,y)
d_dy = attr(x, y+1) - attr(x,y)
```

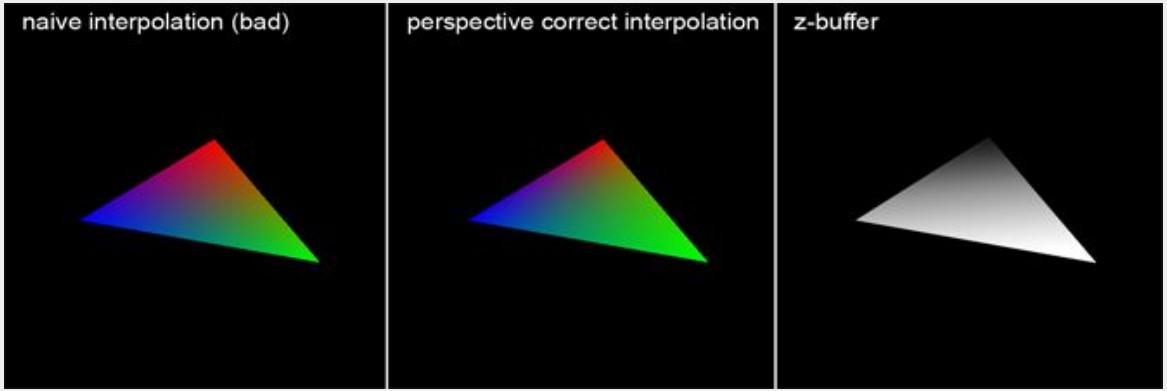
- Use your barycentric helper!
  - Call it for the current pixel (x,y) -> compute attributes = attr_val
  - Call it for the neighbor (x+1, y) -> compute attributes = attr_val_dx
  - Call it for the other neighbor (x, y+1) -> compute attributes = attr_val_dy

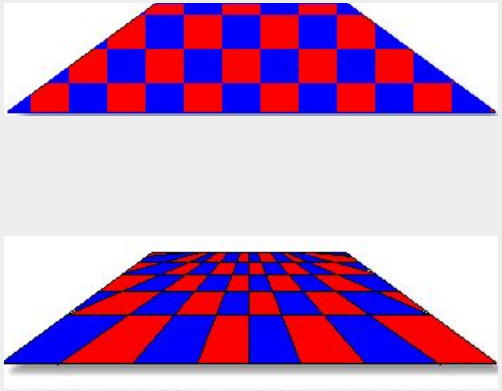
```
out.derivatives[i].x=attr_val_dx - attr_val
```

- Screen Space Smooth Triangles
- Perspective Correct Interpolation
- Mip Map
- Supersampling

Perspective Correct Interpolation

- Naively shading our fragments makes us lose depth information!
- Perspective correct interpolation fixes this





Quick Note…

- In lecture you saw perspective correct interpolation defined in terms of:
  - P = v/z = interpolated vertex positions
  - Z = 1/z = inverse depth
  - ϕ = barycentric coordinates
- But in the writeup, you'll see it in terms of ϕ and ω
  - ϕ = variable for different vertex attributes (not just position)
  - ω = homogeneous coordinate presented in lecture
  - You calculate barycentric coordinates in the code
- They're actually the same thing, but represented in different ways!

Pseudocode

Final interpolated result:

$$\frac{\text{Interpolate}(\Phi/w)}{\text{Interpolate}(1/w)}$$

interpolate(1/w) = Linear interpolation of each of our vertices' **inv_w** with the barycentric coords

interpolate(Φ/w) = Same as above, except we also interpolate the vertex's attribute

Divide them and you have your interpolated attributes!

Note: For derivatives, apply the same logic as before but interpolate neighbors using perspective math first

Triangle Interpolation Summary

- **Flat Shading:** copy attributes of one of the vertices:

$$a(\phi_x, \phi_y) = a(v_1) \qquad \frac{da}{dx} = 0$$

- **Smooth Shading:** barycentric interpolate vertices:

$$a(\phi_x, \phi_y) = a(v_1) * \phi_x + a(v_2) * \phi_y + a(v_3) * (1 - \phi_x - \phi_y)$$

$$\frac{da}{dx} = a(\phi_{x+1}, \phi_y) - a(\phi_x, \phi_y)$$

- **Perspective Correct Shading:** barycentric interpolate vertices with depth:

$$z(\phi_x, \phi_y) = 1/z_1 * \phi_x + 1/z_2 * \phi_y + 1/z_3 * (1 - \phi_x - \phi_y)$$

$$a(\phi_x, \phi_y) = \frac{\left[\frac{a(v_1)}{z_1} * \phi_x + \frac{a(v_2)}{z_2} * \phi_y + \frac{a(v_3)}{z_3} * (1 - \phi_x - \phi_y)\right]}{z(\phi_x, \phi_y)}$$

$$\frac{da}{dx} = a(\phi_{x+1}, \phi_y) - a(\phi_x, \phi_y)$$

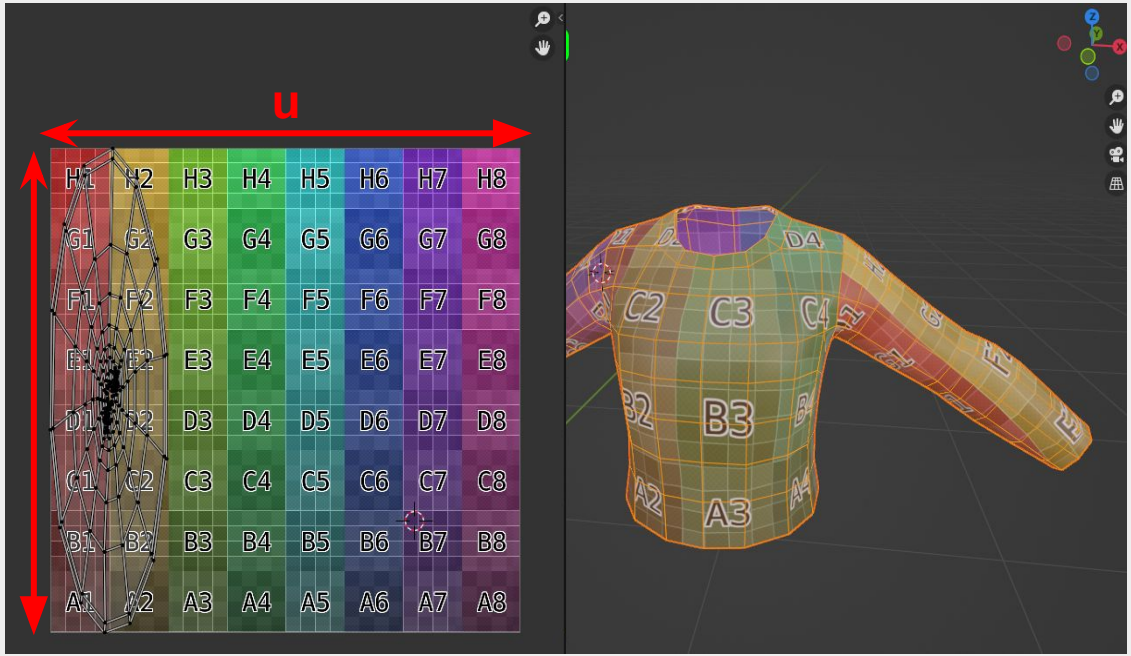
Object Shading in Unity (2018) LMHPOLY

* Remember that the writeup uses different notation

- Screen Space Smooth Triangles
- Perspective Correct Interpolation
- Mip Map
- Supersampling

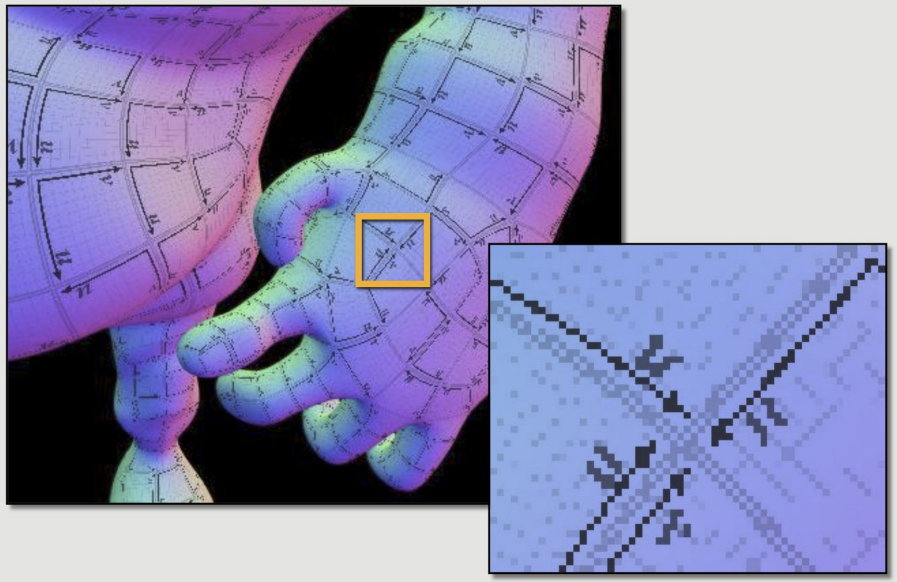
Textures

- Textures are indexed by u,v coordinates
- Each vertex stores its own u,v coordinates that determines where to sample
  - Triangle interpolation is used to fill in between to “wrap” a texture around a mesh
- u,v coordinates are normalized, but meshes are not, so textures may appear stretched/compressed undesirably
  - If only there was something we could do…

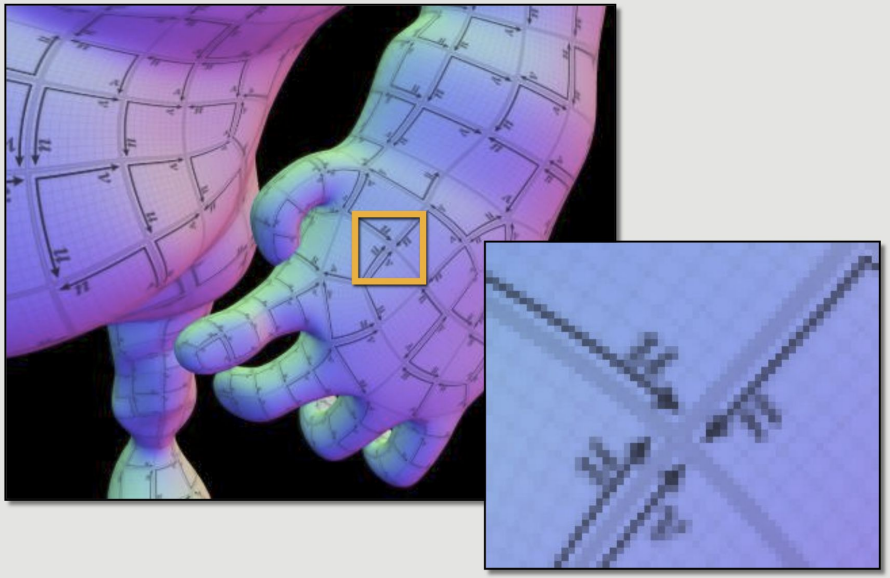


Why do we need mip maps?

Aliasing Due To Minification



Pre-Filtering Texture



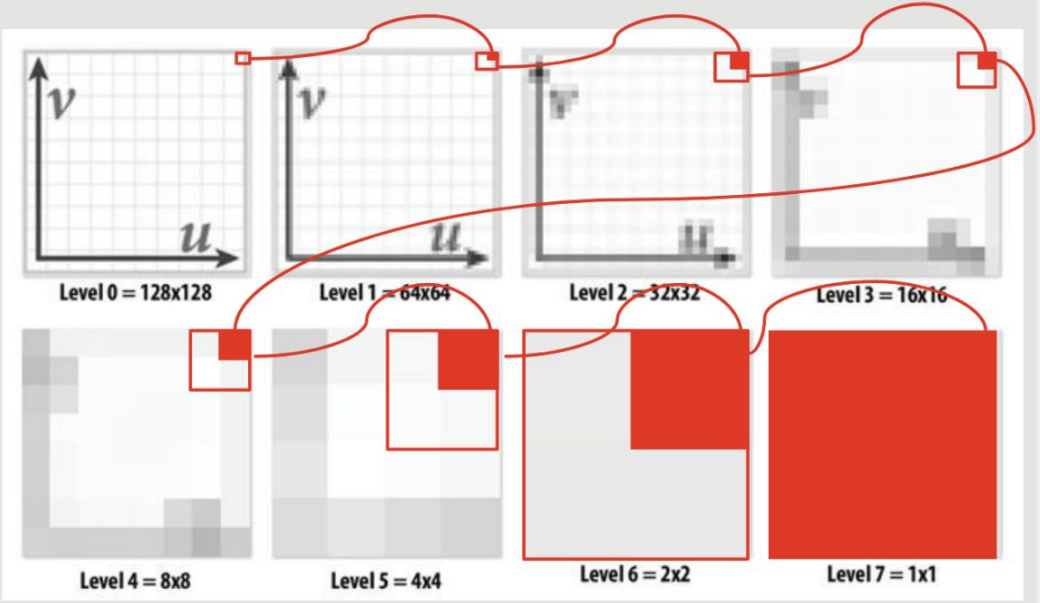
Mip maps address **minification** (aliasing from trying to map many texels onto a small number of pixels)

Mip-Map [L. Williams '83]

- **Rough idea:** precompute a prefiltered image at every possible scale
  - The image at depth d is the result of applying a 2x2 avg filter on the image at depth d-1
    - The image at depth 0 is the base image

- Mip-Map generates $log_2[\min(wth, hgt)] + 1$ levels
  - Each level the width and height gets halved

- Memory overhead: (1+1/3)x original texture
  - $1 + \frac{1}{4} + \frac{1}{16} + \cdots = \sum \frac{1}{4}^j = \frac{1}{1-\frac{1}{4}} = \frac{4}{3}$



Generate MipMap

- You need to fill in the levels vector
  - Level 0 is base image, level 1 is half size, etc.

- Downsampling logic
  - For every pixel (x,y) in dst, you need to calculate the corresponding UV coordinate in src
  - Map the center of the destination pixel (x + 0.5, y + 0.5) back to the normalized [0,1] range
  - Use sample_bilinear(u,v) to get the average color for that location

- Note: Be careful with integer division here! Make sure you are doing floating point math for the UVs.

Computing MipMap Depth

More formally:

$\frac{du}{dx} =$ `fdx_texcoord.x` $\quad \frac{du}{dy} =$ `fdy_texcoord.x`

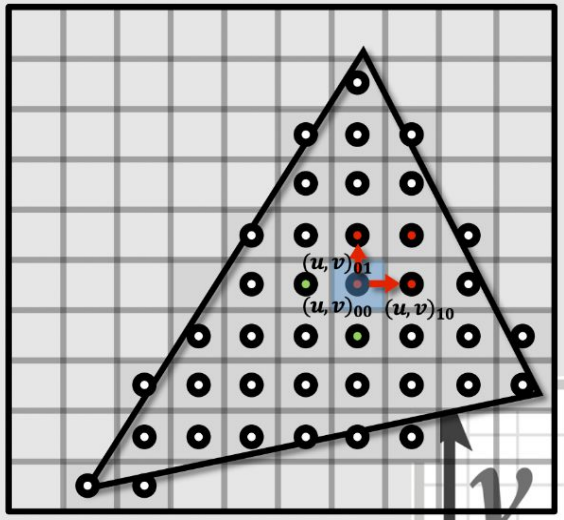
$\frac{dv}{dx} =$ `fdx_texcoord.y` $\quad \frac{dv}{dy} =$ `fdy_texcoord.y`

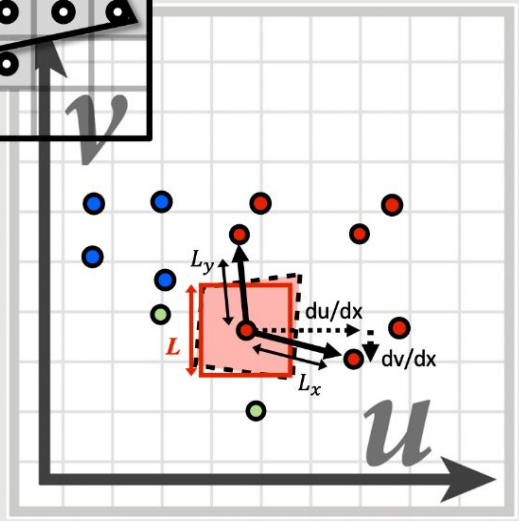
Where dx and dy measure the change in screen space and du and dv measure the change in texture space

$$L_x^2 = \left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2 \qquad L_y^2 = \left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2$$

$$L = \sqrt{max(L_x^2, L_y^2)}$$

L measures the Euclidean distance of the change.
We take the max to get a single number.





$d = \log_2 L$

[final level d]

Trilinear Sampling

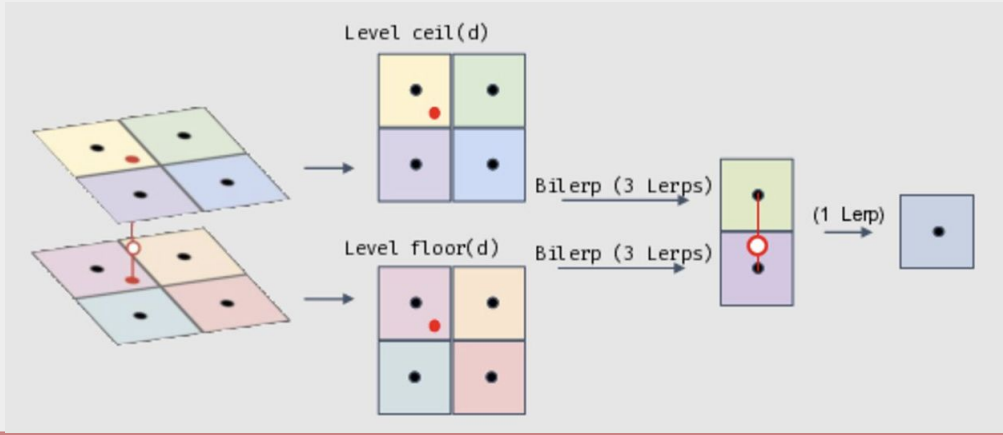
- Determine the two levels to interpolate between
  - d = LOD
  - floor(d) is the higher-res level
  - ceil(d) is the lower-res level
- Make sure to handle boundaries! (lod < 0, lod >= max level)
- Interpolate!
  - Sample level d_floor using bilinear
  - Sample level d_ceil using bilinear
  - Linearly interpolate results based on fractional part of lod

$$L_x{}^2 \leftarrow \frac{du^2}{dx} + \frac{dv^2}{dx}$$
$$L_y{}^2 \leftarrow \frac{du^2}{dy} + \frac{dv^2}{dy}$$

$$L \leftarrow \sqrt{\max(L_x{}^2, L_y{}^2)}$$
$$d \leftarrow log_2\, L$$

$$d' \leftarrow floor(d)$$
$$\Delta d \leftarrow d - d'$$

$$t_d \leftarrow tex[d'].bilinear(x, y)$$
$$t_{d+1} \leftarrow tex[d'+1].bilinear(x, y)$$
$$t \leftarrow (1 - \Delta d) * t_d + \Delta d * t_{d+1}$$



Quick Note…

- In the lecture pseudocode, we assume that all the levels of the MipMap are stored in the same array
  - So we make the implicit assumption that L=0 contains our original texture at index 0 of the array
- **But** in Scotty3D, our original texture is stored in **base**
  - So L=0 would refer to the texture stored in **base**
  - and then L=1,2,... would refer to index L-1 in the rest of the MipMap array
- Also, our code uses fdx_texcoord and fdy_texcoord to represent the changes in texture space (recall that uv coordinates are in [0,1]^2). Think about how you would want to use wh to get the actual amount of change in the texture image.

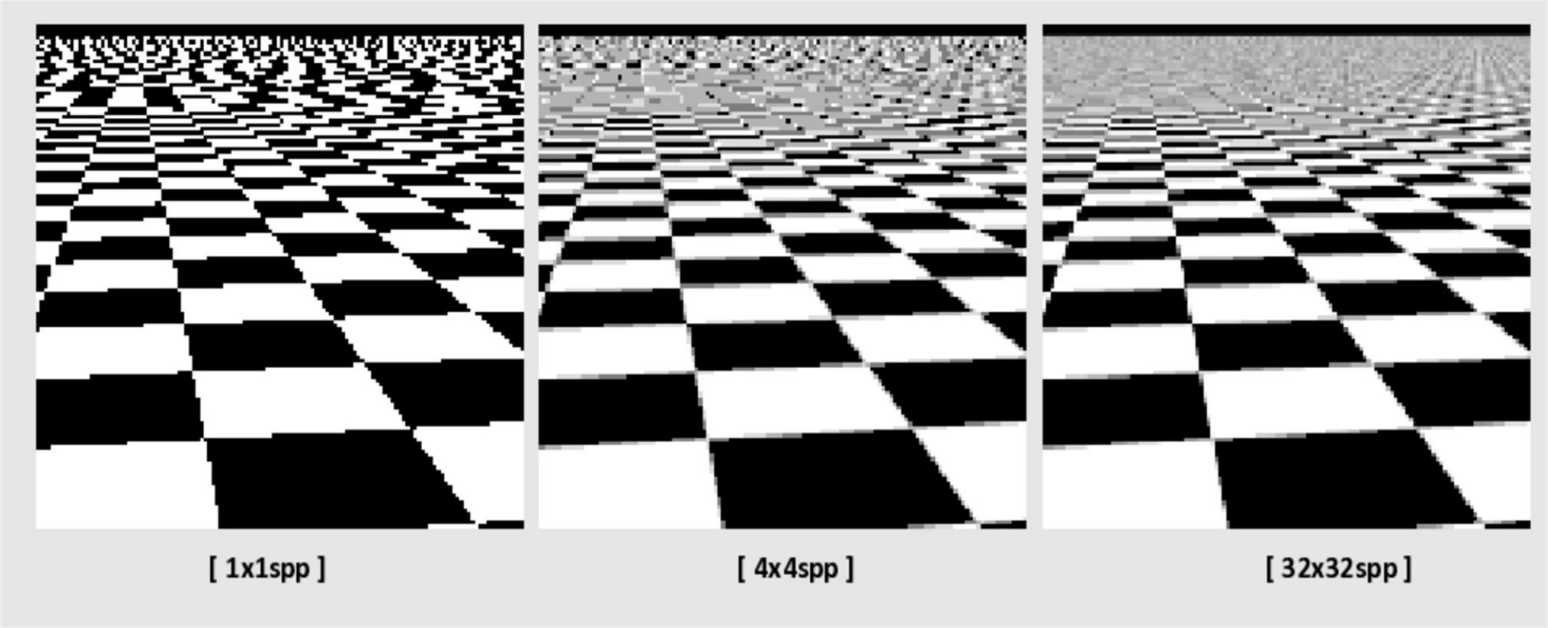
- Screen Space Smooth Triangles
- Perspective Correct Interpolation
- Mip Map
- Supersampling

Supersampling

Ugly artifacting (Moire)

Slightly better…

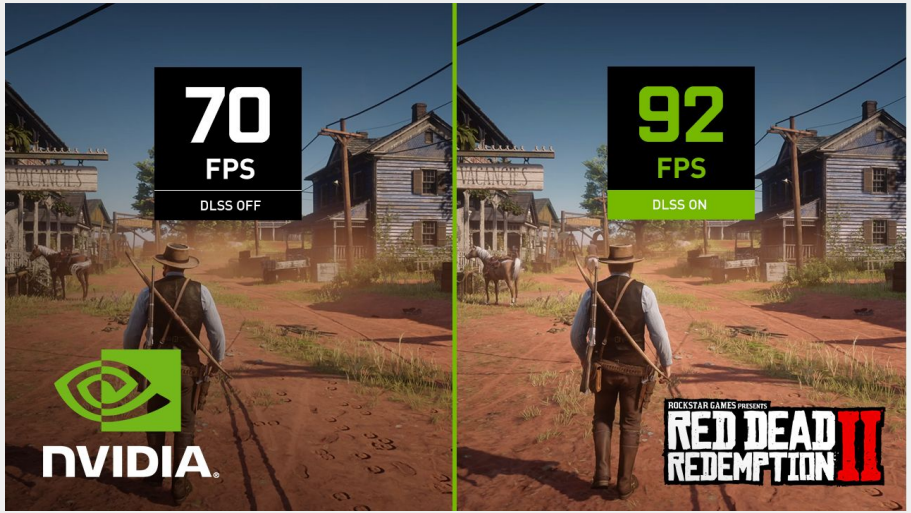
Much better



Is there a way to “soften” the edges to reduce artifacting?

How is this used today?

- DLSS, TAA, etc.

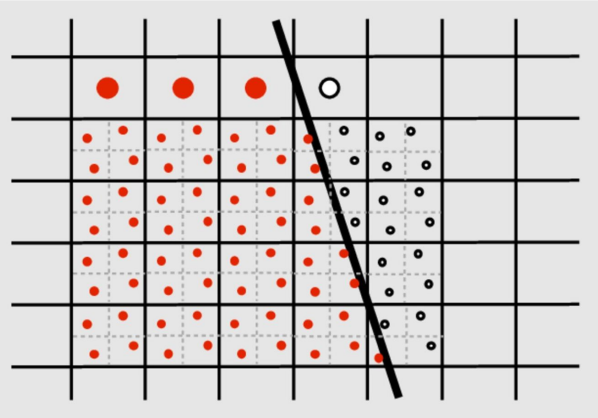


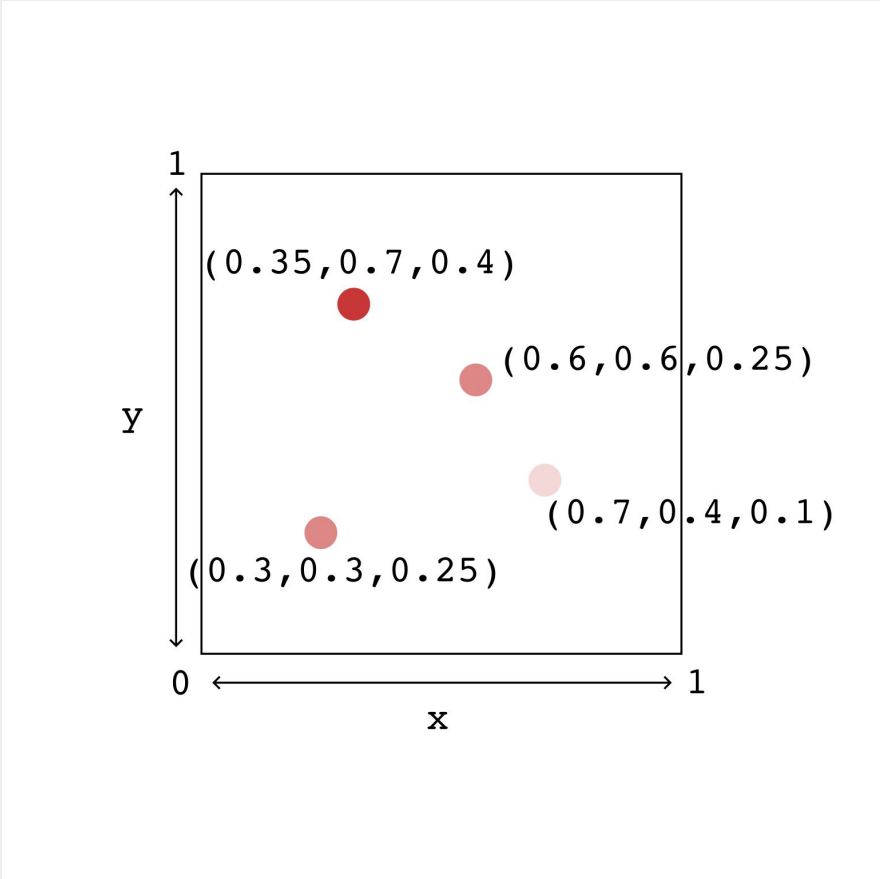
Minecraft! (Win10 version)

Supersampling

Rough idea:

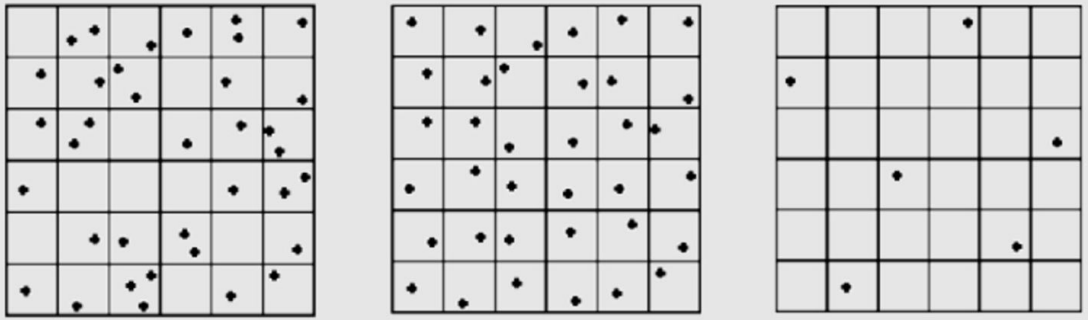
- How do we split up a pixel?
  - Next slide
- How do we figure out what the new color is?
  - Averages with weights!





How do we know what to sample?

Resampling



[Random] [Jittered] [N Rooks]

← Way too hard, implement for extra credit

Many different ways to take n samples.

In Scotty3D, we will provide a vector of sample offsets (dx, dy) and weights w per pixel. Render each sample and perform weighted blending:

$$image(x, y) = \sum_{i=0}^{n} shade(x + dx_i - 0.5, y + dy_i - 0.5) * w_i$$

Impacts

But… our rasterizers were built with only one sample in mind!

- No need to change functions directly, what can we do differently with the ones we currently have?
- Hint: loops