

Image Compositing

Often want to combine a sequence of images together

- different parts of final image can come from different sources
- TV stations have been doing this for a long time

Introduce a new alpha channel in addition to RGB channels

- the α value of a pixel indicates its opacity
 - if $\alpha=0$, pixel is totally transparent
 - if $\alpha=1$, pixel is totally opaque
- alternatively, can think of α as the fraction of the pixel actually covered by the stored color
- convenient to store colors with α premultiplied

$$(r, g, b, \alpha) \Rightarrow (\alpha r, \alpha g, \alpha b, \alpha)$$



Example: Image Compositing

Read RGB α values from frame buffer

Given RGB colors $A = (0.8, 0.6, 1.0)$ and $B = (1, 1, 1)$; $\alpha_A = 0.5$; $\alpha_B = 0.2$

Premultiply: $A' = \alpha_A A = (0.4, 0.3, 0.5)$ $B' = \alpha_B B = (0.2, 0.2, 0.2)$

$$C = A \text{ over } B$$

$$C' = F_A A' + F_B B'$$

$$= (1)A' + (1 - \alpha_A)B'$$

$$= (0.5, 0.4, 0.6)$$

$$\alpha_C = F_A \alpha_A + F_B \alpha_B$$

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

$$= 0.5 + (0.5)0.2 = 0.6$$

De-premultiply: $C = C' / \alpha_C = (0.83, 0.67, 1.0)$

Write RGB α values back into frame buffer

Image Compositing

Compositing one image over another is most common choice

- can think of each image drawn on a transparent plastic sheet
- the final image is formed by stacking layers together

Given images A & B , we can compute $C = A \text{ over } B$

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

- if we pre-multiply α values, this simplifies to

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

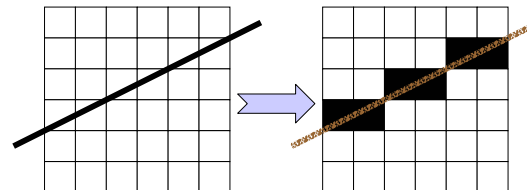
This is only one possible compositing operator

- there are in fact 12 possible ways of combining 2 images

Aliasing

We are often beset by the problem of **aliasing**

- classic examples come up in rasterization (jaggies)



But aliasing can arise in other contexts

- when converting from continuous to discrete representations

We want to get rid of this problem

- we need methods for **antialiasing**

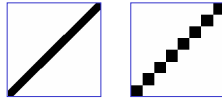
Jaggies: Spatial Aliasing

Jaggies are a particular instance of **spatial aliasing**

- converting a spatial function to discrete representation

Note that increasing resolution decreases jaggies

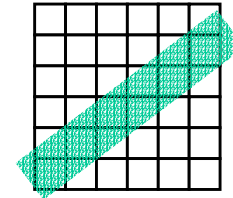
- consider the following two lines



- they come from geometrically identical line descriptions
- but are drawn on 300x300 vs. 8x8 grid
- the higher resolution one is obviously better

Where Do Jaggies Come From?

Our primitives don't evenly cover all the pixels they touch



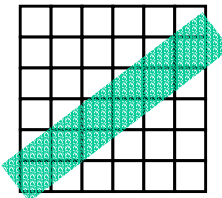
- higher resolution helps because the pixels are smaller
- and the amount of fractional coverage is smaller

What pixels do we fill in?

- all that are completely covered — artificially shrinks object
- all that are touched — artificially expands object

Getting Rid of Jaggies

The key idea is to use **area-weighted sampling**

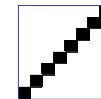


- instead of simply filling in a pixel or not
- compute how much of the pixel is covered by the object
- and fill in with an appropriately scaled color
 - e.g., 35% coverage = RGB (0.35, 0.35, 0.35) for a black object
 - sounds familiar — a lot like alpha values

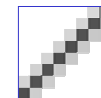
Getting Rid of Jaggies

When we introduce **area-weighted sampling**

- we transition from solid color jagged objects



- to more smoothly colored objects with multiple tones



- when viewed from a proper distance is hopefully smoother

Another Kind of Aliasing

Aliasing also arises when we try to sample small objects

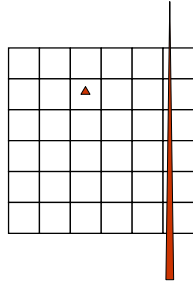
- typically, pixels reflect samples taken at their centers
- when pixels are much smaller than objects, this matters little

Objects may be smaller than pixels

- very small fragments
- very thin slivers

Area sampling can fix this

- but we need to know what pixels they hit
- this will be a problem later with ray tracing



Common Method for Antialiasing

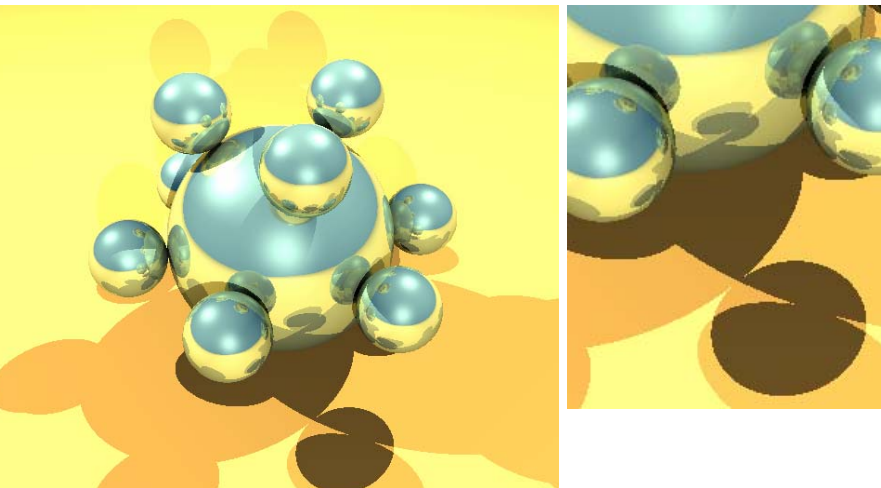
Don't usually want to analytically calculate pixel coverage

- that would be expensive and slow down rendering
- non-trivial computation for higher-order objects

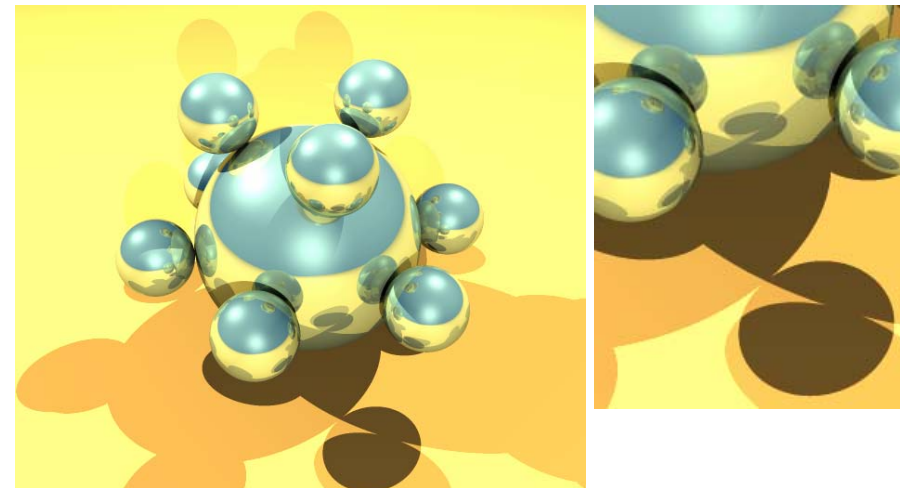
Instead, our most common attack is **super-sampling**

- suppose we want to produce a 256x256 image
 - first, generate a 1024x1024 image
 - then downsample to 256x256 by 4x4 averaging
- each output pixel is computed from 16 subpixel samples
- this reduces efficiency too (by a factor of 16)
- but at least we can control it, by the supersampling ratio

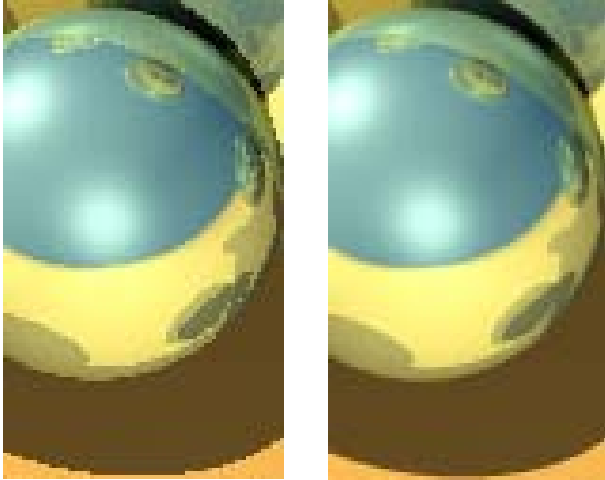
Aliasing in Action



Antialiasing in Action



A Closer Comparison



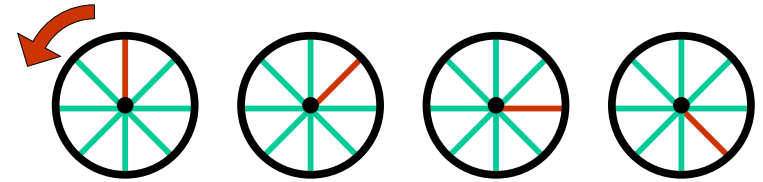
Aliasing in Time

Our animations can also experience **temporal aliasing**

- the motion of objects does not appear as it should

Consider a **spoked wagon wheel**

- suppose it rotates $7/8$ around in 1 second
- and we take a picture of it every second



- it appears to be rotating backwards!

Aliasing in Time

In principle, this is the same phenomenon as **spatial aliasing**

- converting from continuous to discrete representation
- unlucky choice of discrete samples will give us bad results

Can solve the problem in the same way as **spatial aliasing**

- temporal supersampling and average
- in other words, motion blur

Final Thoughts on Aliasing

Unfortunately, we can never really win

- no matter how much we supersample, aliasing remains
- it just gets less and less apparent
- and at some point, humans can't detect it anymore

We've skipped over vast amounts of theory

- can develop rigorous theory of (anti)aliasing
 - based upon signal processing theory (see Foley)
- in this course, focus on simple antialiasing in practice

Image Processing

Construction of an image B as a function of an image A

- **point processing**: function of corresponding pixel only
example: $B[x,y] = \text{sqrt}(A[x,y])$
- **filtering**: function of local neighborhood
example: $B[x,y] = \text{average of neighbors of } A[x,y]$
- largely based on signal processing theory

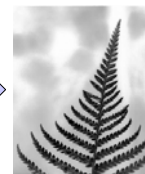
Image processing is a key component in:

- retouching scanned photos (e.g., sharpening)
- automatic segmentation (e.g., foreground vs. background)
- image compression, particularly lossy schemes like JPEG
- and many others ...

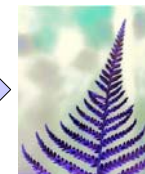
Simple Point Processing Examples

Invert image: $f(p) = 1-p$

- for grayscale images, maps black to white and white to black
- affect on RGB images is a little less obvious



Grayscale Inversion

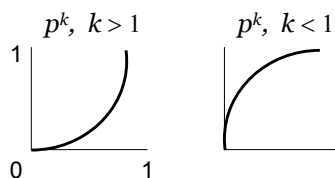


RGB Inversion

Simple Point Processing Examples

Power law transformation: $f(p) = p^k$

- brightens if $k < 1$
- identity if $k = 1$
- darkens if $k > 1$



$k = 2.8$



$k = 0.4$



Image Warping

Instead of modifying pixel values, map pixels to new locations

$$(x, y) \rightarrow (x' = f(x, y), y' = g(x, y))$$

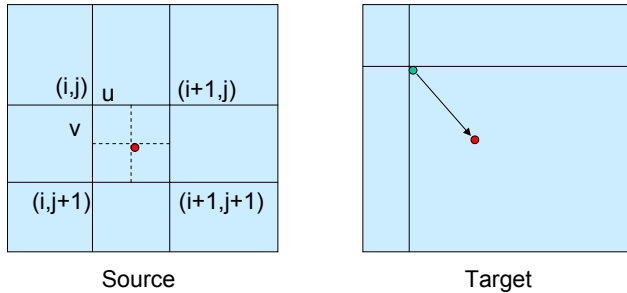
For example, we might apply a horizontal shear



Need to be careful when computing new image

- simply moving pixels from source to target can leave holes
- typically loop over target pixels and map them backwards into the source image

Backward Image Warping



Bilinear Interpolation

$$(1-u)(1-v)C(i, j) + u(1-v)C(i+1, j) + (1-u)vC(i, j+1) + uvC(i+1, j+1)$$

Filtering Images

Simple filters are generally formulated in terms of convolution

- written as $B = A \otimes f = f \otimes A$
- these are probably the most common type

$$\text{in 1-D: } B[x] = \sum_{t=-\infty}^{\infty} A[t]f[x-t]$$

$$\text{in 2-D: } B[x, y] = \sum_{s=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} A[s, t]f[x-s, y-t]$$

Things to notice about convolution formulas

- they're linear functions
- the result at a point is a function of its neighborhood
- nominally, this neighborhood covers all available data

Triangular Barycentric Coordinates

All points in triangle satisfy equation

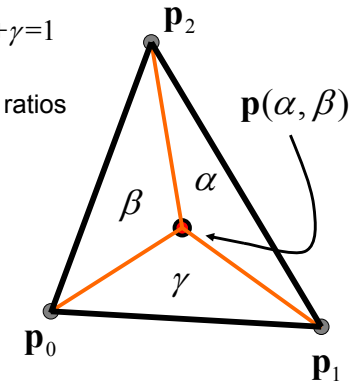
$$\mathbf{p} = \alpha \mathbf{p}_0 + \beta \mathbf{p}_1 + \gamma \mathbf{p}_2 \quad \text{where } \alpha + \beta + \gamma = 1$$

- These coefficients are triangle area ratios
- For triangle of A:

$$\alpha = \frac{\text{Area}(\mathbf{p}, \mathbf{p}_1, \mathbf{p}_2)}{A}$$

$$\beta = \frac{\text{Area}(\mathbf{p}, \mathbf{p}_2, \mathbf{p}_0)}{A}$$

$$\gamma = \frac{\text{Area}(\mathbf{p}, \mathbf{p}_0, \mathbf{p}_1)}{A} = 1 - \alpha - \beta$$



Filtering Images

Naturally, we never want to sum over all the data

- want filter function f to be non-zero over a small area
- this area is the **support** of the filter

It is common practice to represent filters with block templates

- an array of weights applied to local neighborhood
- it looks like a matrix but *it's not*

Here's an example: a 3x3 grid, each cell having value 1/9

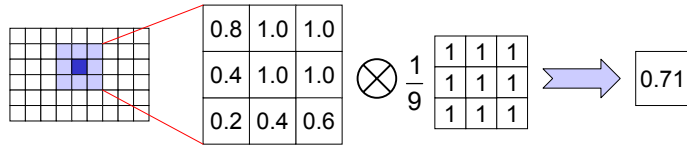
- it replaces a pixel by the equally weighted average of its 3x3 neighborhood
- applying this will blur the image

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Using Filter Templates

To compute the pixel at location (x,y) of the output image

1. find corresponding (x,y) location of input image
2. pick up local neighborhood matching filter template size
3. weight each value of the input according to the value in the template
4. add all the weighted values together



Blurring Filter Example



3x3



5x5



9x9

Blurring Filter Example: A Closer Look



3x3



5x5



9x9

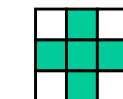
Image Region Filling

Often want to fill a connected region with a specific color

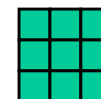
- especially in interactive paint programs
- typically use **flood fill**: start at a given point and fill all neighbors which have the same color as the seed point; recurse
- occasionally use **boundary fill**: keep filling neighbors until pixels are reached which have a specified boundary color

Can define neighborhood in different ways

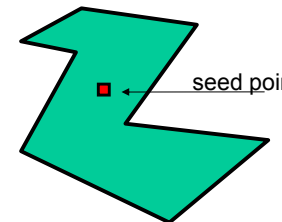
- 8-connected: all neighboring pixels
- 4-connected: only horizontal & vertical



4-connected neighborhood



8-connected neighborhood



seed point

Example Code for Recursive Flood Fill

```
void FloodFill4(int x, int y, color oldValue, color newValue)
{
    if( ReadPixel(x, y) == oldValue )
    {
        WritePixel(x, y, newValue);
        FloodFill4(x, y-1, oldValue, newValue);
        FloodFill4(x, y+1, oldValue, newValue);
        FloodFill4(x-1, y, oldValue, newValue);
        FloodFill4(x+1, y, oldValue, newValue);
    }
}
```

taken from Foley *et al* 19.5

Problem: Recursion depth can be very high

- various more advanced versions
- example: fill entire spans (connected horizontal sequences) at once