Sampling

- our scenes are described with triangles giving a continuous 2d color field
- our images are digital/discrete made up of a grid of dots
- need to make a bridge between these two worlds
- else we will get some unnecessary artifacts called "aliasing" artifacts.
  - jaggies, moire patterns, flickering
- these occur when there is too much detail to fit in one pixel
- we can mitigate these artifacts by averaging up the colors within a pixel's square
  - this is called anti-aliasing
- see CSAA dx10 demo: .ProgramData.NVIDIA Corporation.NVIDIA Direct3D SDK 10.Bin

Two Models

- a *continuous image*, $I(x_w, y_w)$, is a bivariate function.
  - range is a linear color space
- A *discrete image* `I[i][j]` is a two dimensional array of color values.
- We associate each pair of integers `i,j`, with the continuous image coordinates $x_w = i$ and $y_w = j$,

aliasing

- The simplest and most obvious method to go from a continuous to a discrete image is by *point sampling*,
- to obtain the value of a pixel `i,j`, we sample the continuous image function at a single integer valued domain location:

$$\texttt{I[i][j]} \leftarrow I(i, j)$$

- this can results in unwanted artifacts.
- scene made up of black and white triangles: jaggies at boundaries
  - jaggies will crawl during motion
- if triangles are small enough then we get random values or weird patterns
  - will flicker during motion
- the heart of the problem: too much info in one pixel

antialiasing

- intuitively: the single sample is a bad value, we would be better off setting the pixel value using some kind of average value over some appropriate region.
  - in the above examples, perhaps some gray value
- mathematically this can be modeled using Fourier analysis
  - breaks up the data by "frequencies" and figures out what to do with the unrepresentable high frequencies
- we can also model this as an optimization problem
- these approaches lead to:

$$\texttt{I[i][j]} \leftarrow \int\int_{\Omega} dx\, dy\, I(x, y) F_{i,j}(x, y) \tag{1}$$

- where $F_{i,j}(x, y)$ is some function that tells us how strongly the continuous image value at $[x, y]^t$ should influence the pixel value `i,j`.
- In this setting, the function $F_{i,j}(x, y)$ is called a *filter*.
  - In other words, the best pixel value is determined by performing some continuous weighted averaging near the pixel's location.
  - Effectively, this is like blurring the continuous image before point sampling it

box filter

- we often choose the filters $F_{i,j}(x, y)$ to be something non-optimal, but that can more easily computed with.
- The simplest such choice is a *box filter*, where $F_{i,j}(x, y)$ is zero everywhere except over the 1-by-1 square centered at $x = i$, $y = j$.
- Calling this square $\Omega_{i,j}$, we arrive at

$$\texttt{I[i][j]} \leftarrow \int \int_{\Omega_{i,j}} dx\ dy\ I(x, y) \tag{2}$$

- In this case, the desired pixel value is simply the average of the continuous image over the pixel's square domain.

oversampling

- even that integral is not really reasonable to compute
- Instead, it is approximated by some sum of the form:

$$\texttt{I[i][j]} \leftarrow \frac{1}{n} \sum_{k=1}^{n} I(x_k, y_k) \tag{3}$$

- where k indexes some set of locations (x_k, y_k) called the

sample

locations.

- the renderer first produces a "high resolution" color and z-buffer "image",
  - where we will use the term *sample* to refer to each of these high resolution pixels.
- Then, once rasterization is complete, groups of these samples are averaged together, to create the final, lower resolution image.

supersampling

- If the sample locations for the high resolution image form a regular, high resolution grid, then this is called *super sampling*.
- We can also choose other sampling patterns for the high resolution "image",
  - Such less regular patterns can help us avoid systematic errors that can arise when using the sum to replace the integral

multisampling

- render to a "high resolution" color and z-buffer
- During the rasterization of each triangle, "coverage" and z-values are computed at this sample level.
- But for efficiency, the fragment shader is only called

only once per final resolution pixel

.

  - This color data is shared between all of the samples hit by the triangle in a single (final resolution) pixel.
- once rasterization is complete, groups of these high resolution samples are averaged together,

- Multisampling can be an effective anti-aliasing method, since, without texture mapping, colors tend to vary quite slowly over each triangle, and thus they do not need to be computed at high spatial resolution.
- To deal with aliasing that occurs during texture mapping, we have the advantage of possessing the texture image in hand at the outset of the rendering process.
  - This leads to specialized techniques such as *mip mapping*
- csaa demo uses multisampling, see book image

In the Wild

- In digital cameras, antialiasing is accomplished by a combination of the spatial integration that happens over the extent of each pixel sensor, as well as by the optical blurring that happens at due to the lens. Some cameras also include additional optical elements specifically to blur the continuous image data before it is sampled at the sensors.
- In human vision, aliasing artifacts are not typically encountered.
  - Most of the antialiasing, at least in the foveal (central) region of vision, is due to the optical blurring of light, which happens well before it hits the receptor cells.
  - The irregular spatial layout of the sensor cells in the retina also helps by effectively providing spatial jitter (randomness) which turns noticeable aliasing into less conspicuous noise.

image compositing

- given two discrete images, a foreground, $\mathtt{I^f}$, and background, $\mathtt{I^b}$, that we want to combine into one image $\mathtt{I^c}$.
- simple: in composite, use foreground pixels where they are defined. else use background pixels.
- this will give us a jagged boundary (see book fig)
- real image would have "boundary" pixels with blended colors
- but this requires using "sub-pixel" information

alpha blending

- associate with each pixel in each image layer, a value, $\alpha[\mathtt{i}][\mathtt{j}]$, that describes the overall *opacity* or *coverage* of the image layer at that pixel.
  - An alpha value of 1 represents a fully opaque/occupied pixel, while a value of 0 represents a fully transparent/empty one.
  - A fractional value represents a partially transparent (partially occupied) pixel.
- alpha will be used during compositing

alpha definition

- More specifically, let $I(x, y)$ be a continuous image, and let $C(x, y)$ be a binary valued *coverage function* over the continuous (x, y) domain, with a value of 1 at any point where the image is "occupied" and 0 where it is not.
- Let us store in our discrete image the values

$$\begin{aligned} \mathtt{I}[\mathtt{i}][\mathtt{j}] &\leftarrow \int\int_{\Omega_{i,j}} dx\, dy\, I(x,y)C(x,y) \\ \alpha[\mathtt{i}][\mathtt{j}] &\leftarrow \int\int_{\Omega_{i,j}} dx\, dy\, C(x,y) \end{aligned}$$

over operation

- to compose $\mathtt{I^f}[\mathtt{i}][\mathtt{j}]$ *over* $\mathtt{I^b}[\mathtt{i}][\mathtt{j}]$, we compute the composite image colors, $\mathtt{I^c}[\mathtt{i}][\mathtt{j}]$, using

$$\mathtt{I^c}[\mathtt{i}][\mathtt{j}] \leftarrow \mathtt{I^f}[\mathtt{i}][\mathtt{j}] + \mathtt{I^b}[\mathtt{i}][\mathtt{j}](\mathtt{1}-\alpha^f[\mathtt{i}][\mathtt{j}]) \tag{4}$$

- That is, the amount of observed background color at a pixel is proportional to the transparency of the foreground layer at that pixel.

- Likewise, alpha for the composite image can be computed as

$$\alpha^c[\mathtt{i}][\mathtt{j}] \leftarrow \alpha^f[\mathtt{i}][\mathtt{j}] + \alpha^b[\mathtt{i}][\mathtt{j}](1 - \alpha^f[\mathtt{i}][\mathtt{j}]) \tag{5}$$

- if background is opaque, so is the composite pixel
- but we can model more general case as part of blending multiple layers
- see book figure again, also marshner,p13

over properties

- this provides a reasonable approximation to the correctly rendered image (see book for some math)
- One can easily verify that the over operation is associative but not commutative. That is,

$$\mathtt{I}^{\mathtt{a}} \text{ over } (\mathtt{I}^{\mathtt{b}} \text{ over } \mathtt{I}^{\mathtt{c}}) = (\mathtt{I}^{\mathtt{a}} \text{ over } \mathtt{I}^{\mathtt{b}}) \text{ over } \mathtt{I}^{\mathtt{c}}$$

- but

$$\mathtt{I}^{\mathtt{a}} \text{ over } \mathtt{I}^{\mathtt{b}} \neq \mathtt{I}^{\mathtt{b}} \text{ over } \mathtt{I}^{\mathtt{a}}$$

- note: pulling mattes from real images against controlled or esp. uncontrolled backgrounds is a area of research.

alpha blending in openGL

- used for general purpose monochromatic blending
  - simulate transparent, non refractive, geometry
- We can assign an alpha value as the fourth component of `fragColor` in a fragment shader, and this is used to combine with whatever is currently present in the framebuffer.
- Blending is enabled with the call `glEnable(GL_BLEND)` and the blending calculation to be used is specified by the arguments in a call to `glBlendFunc`.
- due to non-commutativity, this requires back to front ordered rendering
- there are other ideas for doing transparency in stcastic demo: openGL .ProgramData.NVIDIA Corporation.NVIDIA Direct3D SDK 11.Bin

Reconstruction

- given a discrete image `I[i][j]`, how do we create a continuous image $I(x, y)$?
- is central to resizing images and to texture mapping.
  - how to get a texture colors that fall in between texels
- This process is called *reconstruction*.

Constant reconstruction

- a real valued image coordinate is assumed to have the color of the closest discrete pixel. This method can be described by the following pseudo-code

```
color constantReconstruction(float x, float y, color image[][]){
 int i = (int) (x + .5);
 int j = (int) (y + .5);
 return image[i][j]
}
```

  The "`(int)`" typecast rounds a number p to the nearest integer not larger than p.

- the resulting continuous image is made up of little squares of constant color.

- Each pixel has an influence region of 1 by 1.
- see constVSlin demo

Bilinear

- can create a smoother looking reconstruction using *bilinear interpolation*.
- Bilinear interpolation is obtained by applying linear interpolation in both the horizontal and vertical directions.

```
color bilinearReconstruction(float  x, float y, color image[][]){
  int intx = (int) x;
  int inty = (int) y;
  float fracx = x - intx;
  float fracy = y - inty;

  color  colorx1 = (1-fracx)* image[intx]  [inty] +
                   (fracx)  * image[intx+1][inty];
  color  colorx2 = (1-fracx)* image[intx]  [inty+1] +
                   (fracx)  * image[intx+1][ inty+1];

  color  colorxy   = (1-fracy)* colorx1 +
                       (fracy)  * colorx2;
  return(colorxy)
}
```

properties

- At integer coordinates, we have $I(i,j) = \texttt{I[i][j]}$; the reconstructed continuous image I agrees with the discrete image `I`.
- In between integer coordinates, the color values are blended continuously.
- Each pixel in the discrete image influences, to a varying degree, each point within a 2-by-2 square region of the continuous image.
- tent demo
- the horiz/vert ordering is irrelevant
- color over a square is billinear function of (x,y).
- can be modeled as

$$I(x,y) \leftarrow \sum_{i,j} B_{i,j}(x,y)\texttt{I[i][j]} \tag{6}$$

- there are also higher order methods, see recon demo

Resampling

- lets revisit texture mapping
- we start with a discrete image and end with a discrete image.
- the mapping technically involves both a reconstruction and sampling stage.
- In this context, we will explain the technique of mip mapping used for anti-aliased texture mapping.

Resampling equation

- Suppose we start with a discrete image or texture `T[k][l]` and apply some 2D warp to this image to obtain an output image `I[i][j]`.
- Reconstruct a continuous texture $T(x_t, y_t)$
- Apply the geometric warp M^{-1} to the continuous image.

  - texture to window warp
- Integrate against a box filter on the screen

...math... we get

- we need to integrate the reconstructed texture over the region $M(\Omega_{i,j})$ on the texture domain,
- see fig 18.1
- When the transformation M^{-1} effectively shrinks the texture, then $M(\Omega_{i,j})$ has a large footprint over $T(x_t, y_t)$.
- If M^{-1} is blowing up the texture, then $M(\Omega_{i,j})$ has a very narrow footprint over $T(x_t, y_t)$.
- During texture mapping, M^{-1} can also do funnier things, like shrink in one direction only.

Blow up

- in the case that we are blowing up the texture, the filtering component has minimal impact on the output.
- In particular, the footprint of $M(\Omega_{i,j})$ may be smaller than a pixel unit in texture space, and thus there is not much detail that needs blurring/averaging.
- As such, the integration step can be dropped, and the resampling can be implemented as

$$\texttt{I[i][j]} \quad \leftarrow \quad \sum_{k,l} B_{k,l}(x_t, y_t)\texttt{T[k][l]} \tag{7}$$

  where $(x_t, y_t) = M(i, j)$.
- such as constant or billinear
-
- We tell OpenGL to do this using the call `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)`.
- For a single texture lookup in a fragment shader, the hardware needs to fetch 4 texture pixels and blend them appropriately.

shrinking

- In the case that a texture is getting shrunk down, then, to avoid aliasing, the filtering component should not be ignored.
- Unfortunately, there may be numerous texture pixels under the footprint of $M(\Omega_{i,j})$, and we may not be able to do our texture lookup in constant time.

mip mapping

- In mip mapping, one starts with an original texture $\texttt{T}^0$ and then creates a series of lower and lower resolution (blurrier) textures $\texttt{T}^i$.
- Each successive texture is twice as blurry. And because they have successively less detail, they can be represented with $1/2$ the number of pixels in both the horizontal and vertical directions.
- This collection, called a mip map, is built before the any triangle rendering is done.
- Thus, the simplest way to construct a mip map is to average two-by-two pixel blocks to produce each pixel in a lower resolution image.
- see webimage.

more

- During texture mapping, for each texture coordinate (x_t, y_t), the hardware estimates how much shrinking is going on.
  - how big is the pixel footprint on the geometry.

- This shrinking factor is then used to select from an appropriate resolution texture T^i from the mip map. Since we pick a suitably low resolution texture, additional filtering is not needed, and again, we can just use reconstruction
- To avoid spatial or temporal discontinuities where/when the texture mip map switches between levels, we can use so-called trilinear interpolation. We use bilinear interpolation to reconstruct one color from T^i and another reconstruction from T^{i+1}. These two colors are then linearly interpolated. This third interpolation factor is based on how close we are to choosing level i or $i+1$.
- Mip mapping with trilinear interpolation is specified with the call
`glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR)`.
- Trilinear interpolation requires OpenGL to fetch 8 texture pixels and blend them appropriately for every requested texture access.
- see aliasing demo

properties

- It is easy to see that mip mapping does not do the exactly correct computation.
- First of all, each lower resolution image in the mip map is obtained by isotropic shrinking, equally in every direction. But, during texture mapping, some region of texture space may get shrunk in only one direction (see book Figure).
- even for isotropic shrinking, the data in the low resolution image only represents a very specific, dyadic, pattern of pixel averages from the original image.
- filtering can be better approximated at the expense of more fetches from various levels of the mip map to approximately cover the area $M(\Omega_{i,j})$ on the texture.
- This approach is often called anisotropic filtering, and can be enabled in an API or using the driver control panel.
- see book images