

Chapter 1

Image sampling

So far, we have thought of an image in two different ways, continuous and discrete. A continuous greyscale image is a continuous intensity function over a two dimensional domain represented in image coordinates

$$I(x_p, y_p)$$

We have assumed that I is defined over $-0.5 < x_p < w - 0.5$ and $-0.5 < y_p < h - 0.5$. A color image is simply three such images, one for red, green and blue.

A discrete greyscale image is a w by h array of bytes, and a discrete color image is a w by h array of triplets of bytes.

$$I[i][j]$$

where i is an integer in $[0..w - 1]$ and j is an integer in $[0..h - 1]$.

We have assumed that the relationship between these two kinds of images is that the value of the discrete image $I[i][j]$ is a single sample of the continuous function at the integer valued location $x_p = i$, $y_p = j$. This was our assumption, for example, when we described our scan conversion algorithm for rendering triangles.

In this chapter we will start by looking at two questions. Given a discrete image, how should we create a continuous function? This is an important question that comes up for example during texture mapping. In short, the answer will be some type of interpolation.

The other question is given a continuous function, is single sampling really the best way to create a discrete image? In short the answer will be no. We can do better by looking at more than just $w \times h$ function values, even if in the end, we are only going to store $w \times h$ pixels. By doing this, we can remove certain artifacts, like jagged edges and animation flicker.

One application where jagged edges are often very distracting is with image compositing. In image compositing, one places part of an image (say of a weatherman)

ontop of a background image, say of a weathermap. We will discuss the notion of alpha composition as a method for reducing some of the jagged edges that can appear.

In this chapter, we will see that certain of the operations that we would like to do involve computing integrals. Since computing integrals using symbolic manipulation, like from calculus class, can be impractical, we will look at numerical methods for approximating integrals. In particular we will look at the simple but powerful technique of monte carlo sampling. This method will also be at the heart of the distribution ray-tracing algorithm for high quality rendering that we will see in chapter

1.1 Image Reconstruction

The first question we will look at is: given a discrete image $I[i][j]$, how does one create a continuous image $I(x_p, y_p)$? This question can arise in a number of contexts. Suppose I give you a discrete image, and I wish to blow it up. To do this, I need more samples. One way to approach this problem is to first create a continuous image; then one can generate the extra pixels for the blown up image from this continuous image. Another context where this question arises is texture mapping. Texture mapping creates a mapping from an output pixel (with interger coordinates) to some real valued texture coordinate $[x_t, y_t]^t$. If these texture coordinate are non integers, we must decide how to extract a value from discrete texture image. One approach to first create a continuous image; the one can generate values for any requested texture coorindates.

In order to approach the problem one needs to make some assumptions. Otherwise there is really no way to pick a good continuous function with an infinite amount of information when given only finite discrete input. The most typical assumption that is made is that we expect the continuous function to be in some way *smooth*. We do not expect it to have all kinds of random hills and vallys and extra colors that were not seen in the discrete input. This leads us to a simple family of reconstruction algorithms.

1.1.1 Constant reconstruction

Perhaps the easiest image reconstruction approach is the constant reconstruction method. In this method, a real valued image coordinate is assumed to have the color of the closest discrete pixel. This method can be described by the following code

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

The reason this is called constant reconstruction is that the continuous image is made up of little squares of constant color. For example, the image will have the

constant value of $I[0][0]$ of the square shaped region $-.5 < x_p < .5$, and $-.5 < y_p < .5$. Each pixel has an influence region of 1 by 1.

1.1.2 Bilinear reconstruction

One can create a smoother reconstruction using bilinear interpolation. Bilinear interpolation is obtained using linear interpolation in the horizontal and vertical directions. It can be described by the following code:

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

In this code, we first applied linear interpolation in x followed by linear interpolation of that result in y . It can be shown that if we were to reverse this order, we would obtain the same result. This reconstructed image will have the colors from the discrete image at the interger valued coordinates. In between these coordinates, the colors will be smoothly blended. Each pixel has an influence region of 2 by 2, but the influence gets smaller as one moves away from the integer valued image coordinate.

This method is called bilinear because the continuouse image over a square shaped domain region $i < x < i + 1$, and $j < y < j + 1$ will be a bilinear funcion. A bilinear function is a function of the form $ax + by + cx + d$.

1.1.3 Basis functions

The constant and bilinear reconstruction algorithms are just two examples of image reconstruction using linear combination of basis functions. In this method, one first chooses a set of continuous basis functions $B_{i,j}(x, y)$. These functions are then weighted by the pixel values and summed up to create the continuous reconstruction

$$I(x, y) = \sum_{i,j} B_{i,j}(x, y)I[i][j]$$

At each x, y , the function $B_{i,j}(x, y)$ tells us how much of pixel $I[i][j]$ to blend in. If the value is 1 then we use all of pixel i, j at that x, y location. If the value is 0, we use none of it.

We can express the constant reconstruction algorithm as a simple example of basis function reconstruction. In this case the basis function $B_{i,j}(x, y)$ is a function that is zero everywhere except over the square domain $i - .5 < x < i + .5$, and $j - .5 < y < j + .5$. Over this square, the value is one. See figure !!.

Some of these concepts are easier to visualize over a one dimensional domain. You can think of this as reconstructing a continuous image over a single scanline j . Figure !! shows constant reconstruction over such a one dimensional domain.

Over a one dimensional domain, if one wanted to perform linear interpolation of the pixel values, one can achieve this using the “hat” basis functions

$$H_i(x) = \begin{cases} x - i & \text{for } i - 1 < x < i \\ -x - i + 1 & \text{for } i < x < i + 1 \\ 0 & \text{else} \end{cases}$$

See figure !!. A hat function has value one at its center, and falls off linearly to 0 at a distance of 1 in both directions. Outside this region, the function is 0. In two dimensions one can perform bilinear interpolation using the Tent basis function

$$B_{i,j}(x, y) = T_{i,j}(x, y) = H_i(x)H_j(y)$$

This is a tent shaped function that is 1 at its center and falls off bilinearly to 0 outside of a sidelength 2 domain square. See figure !!). The process of creating a bivariate function by multiplying two univariate functions is called creating a tensor product, or separable function.

In general one can choose basis functions with all kinds of sizes (wider support) and shapes. In particular people often choose radially symmetric basis functions (with circular cross section) instead of tensor product ones. In this sense a pixel is not really a little square. It is simply a discrete value that is used in conjunction with a set of basis functions to obtain a continuous function.

1.1.4 In practice

In practice, for texture mapping, the most common reconstruction method is bilinear reconstruction. Occasionally for efficiency, one might use constant reconstruction. For blowing up images, a variety of filters are used. The most common are bilinear and constant, but high quality image editing programs usually provide smoother interpolation based on piecewise cubic functions.

1.1.5 Scattered Data

In our domain, we are trying to reconstruct a continuous function given a regularly spaced set of data samples. A related problem is obtaining a function when the data samples are irregularly spaced. This is often called the “scattered data interpolation” problem. This problem is typically harder than our problem, but the same principles apply. One makes some assumptions about the missing data (usually that it is smooth, in some precise mathematical sense), and then solves for the smoothest function that agrees with the data.

Although this sounds very mathematical, these problems often arise often in computer graphics. It comes up in places like surface fitting, image warping, and animation. A lengthy discussion of this topic is beyond the scope of this book.

1.2 Sampling

The dual of the reconstruction problem is important as well. Given a continuous image $I(x, y)$, what is the best way to pick values for the discrete representation $I[i][j]$? This is a fundamental problem in computer graphics. If one describes the computer graphics imaging process by modeling the scene as triangles, and the camera as a projection matrix, this will define a color value for every real valued domain coordinate $[x_p, y_p]^t$. During the rendering process, we must turn this into a discrete image.

Thusfar, we have answered this question by using “point sampling”, that is, to obtain the value of a pixel i, j , we sample the continuous image function at integer valued domain locations

$$I[i][j] = I(i, j)$$

For example during scan conversion, we simply determined which triangle is non occluded at a single integer location single point, and computed the appropriate color for that single point. The question remains: is this the best thing we can do?

There are many examples when sampling can result in various artifacts in the generated discrete image. For example, when point sampling a triangle, the boundary between the interior and the exterior of the triangle will usually take on some sort of staircased shape in the sampled representation. This is often referred to as “jaggies”. During an animation, if one moves the triangle slowly, there will be no change in the sampled image until some new sample location gets covered or uncovered. At that point the jagged edge will pop to a new shape, this is called “crawling”.

One can get also very bad artifacts when dealing with very small triangles. Suppose one is rendering a picture of a herd of zebra. The zebras are made up of black and white colored triangles. Suppose one of the zebra is far away in the background, so far away that it covers just a few pixels. What will the color of those pixels be? If a sample happens to land on a black triangle, then the pixel will be black, while if it happens to land on a white triangle, the pixel will be white. The results is essentially

random. During animation, the zebra may move a bit, and these pixels can then take on a different set of random values. The result is that we will see “flickering” during the animation.

These types of errors appear when the continuous function has lots of detail in a small region. In the zebra example, the continuous image has a whole lot of zebra in the space of a few pixels. With the jagged triangle example, the continuous image has an exact sharp edge with an immediate transition, not a gradual one. For technical reasons, this type of loss is called “aliasing”.

When there is a lot of detail in a small space, it is unlikely that we will be able to keep all of the information in our discrete representation. But by taking only a single sample to determine the pixel’s color we may be making matters even worse for ourselves, possibly just picking a random value. Perhaps we can do better if we take more than just one sample before determining the pixel’s color.

In general, if one starts with a continuous function I , then samples it, and finally uses basis functions to obtain a reconstructed continuous function, one will generally not be able to exactly reproduce the original function.

$$\hat{I}(x, y) \neq \sum_{i,j} B_{i,j}(x, y) \mathbb{I}[i][j]$$

Given that we will be reconstructing the continuous image using the chosen basis functions, what is the best choice for the pixel values $\mathbb{I}[i][j]$ such that the error in the approximation is minimized. There are many ways such an error can be quantified. Ideally one would like a metric that measured how perceptually similar two images are. These metrics are usually hard to come by. If we use a more straightforward error metric (called the L^2 metric), it can be shown that the best pixel value is obtained using

$$\mathbb{I}[i][j] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x, y) F_{i,j}(x, y) dx dy$$

In other words, the best pixel value is determined by integrating, or performing some continuous weighted average over the pixel domain using a function F to determine the weights. The function $F_{i,j}(x, y)$ is sometimes called a filter. The optimal choice for $F_{i,j}$ turns out to be something called the dual basis of $B_{i,j}(x, y)$. For some kinds of bases (called orthonormal bases), the dual basis is the same as the original basis $F_{i,j} = B_{i,j}$. The box basis is orthonormal, but the tent basis is not. Often we do not require complete optimality, and we choose the $F_{i,j}$ to be something convenient, even if it is not the dual basis.

1.2.1 In practice

In practice to avoid aliasing artifacts usually box functions are used as the filters $F_{i,j}$. In this case, the desired pixel value is simply the unweighted average of the continuous

image over the filter's square domain. When scan converting lines, there are algorithms to directly compute what fraction of the pixel filter square domain is covered by a line with some finite width.

When scan converting triangles, exact integration is typically avoided, and numerical approximation algorithms are employed. The most common numerical approach is to create a high resolution image; these high resolution samples are then used to numerically approximate the integration. For example, one may scan convert an image at twice the resolution horizontally and vertically. Then for each output pixel, four high resolution samples are averaged together. Later in this chapter we will discuss monte carlo intergration. This is a numerical integration technique based on random samples that can be used when ray tracing images.

1.2.2 Relationship to Fourier analysis

Many of these concepts arise in the field of signal processing. We will briefly discuss the most famous theorem in that field, the Shannon sampling theorem.

Suppose that as our reconstruction basis functions we choose shifted "sinc" functions

$$B_{i,j} = \text{sinc}(x - i)\text{sinc}(y - j)$$

where

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

The Shannon sampling theorem states that if I is "smooth enough", then it can be exactly reconstructed using the sinc basis functions. Moreover we only need to take single function samples for each pixel. In other words:

$$I(x, y) = \sum_{i,j} B_{i,j}(x, y)I(i, j)$$

For the Shannon sampling theorem, smoothness is defined by looking at something called the Fourier transform of I . The Fourier transform is a way of measuring the different frequencies that make up the function I . If there are no "high frequency" terms, then it can be exactly reconstructed. This type of smoothness is invariant to shifts, so even if we shift the function by some amount, single sample pixels used to weight shifted sines will still reconstruct the exact function.

When there are high frequency terms, then exact reconstruction cannot be achieved. To obtain the least error, as above, one must perform integration against a filter to obtain the pixel values. Since sines are orthonormal basis functions, they serve as their own duals, and the best pixel values are obtained as

$$\mathbb{I}[i][j] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x, y)B_{i,j}(x, y)dx dy$$

In practice, one often uses filters that are more convenient than the sinc function, such as the box or tent function.

In signal processing, the process of obtaining pixels by integrating with filters is often interpreted as a two stage process. In the first stage the continuous image is blurred to obtain a smoother continuous function

$$\hat{I}(i, j) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x, y) B_{0,0}(x - i, y - j) dx dy$$

The blurred function now satisfies the requirements of the Shannon sampling theorem and so a discrete image can be created using point sampling

$$I[i][j] = \hat{I}(i, j)$$

1.2.3 Blurring and shrinking

If one starts with a high resolution image I^0 , one can create a blurrier version of the image I^1 by using wider filter $F_{i,j}^1(x, y)$. The proper steps to do this are

- reconstruct the continuous sharp image $I^0(x, y) = \sum_{k,l} B_{k,l}^0(x, y) I^0[k][l]$
- create the blurry discrete image using a wide filter F^1

$$I^1[i][j] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I^0(x, y) F_{i,j}^1(x, y) dx dy$$

The two steps can be written together as:

$$\begin{aligned} I^1[i][j] &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F_{i,j}^1(x, y) \sum_{k,l} B_{k,l}^0(x, y) I^0[k][l] dx dy \\ &= \sum_{k,l} I^0[k][l] \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F_{i,j}^1(x, y) B_{k,l}^0(x, y) dx dy \\ &= \sum_{k,l} I^0[k][l] T_{i,j,k,l} \end{aligned}$$

The last line says that to blur an image, we only need the original discrete image $I^0[k][l]$, and the discrete blurring numbers $T_{i,j,k,l}$. We never have to actually create the continuous image, nor take any integrals of images. Computing the $T_{i,j,k,l}$ does require integration, but these are integrals of known simple functions B and F , and not I^0 , and so they can be often precomputed in closed form.

Shrinking an image should be done using the exact same methodology. If one just shrinks an image by dropping some of the pixels, one is simply throwing away detail.

The particular pixels that remain are somewhat random. The proper way to shrink an image is to make sure that the continuous shrunken function is properly filtered to obtain the discrete shrunken function. As in the image blurring case, one can do this without actually going to the continuous domain.

To perform shrinking, one needs a simple affine mapping M to map between the big image pixel coordinates and the little image pixel coordinates

$$(x_b, y_b) = M(x_l, y_l)$$

One then goes through the following steps

- reconstruct the continuous sharp image $I^0(x_b, y_b) = \sum_{k,l} B_{k,l}^0(x_b, y_b) I^0[k][1]$
- create a smaller continuous sharp image $I^1(x_l, y_l) = I^0(M(x_l, y_l))$
- create the discrete small discrete image using the standard filter F

$$I^1[i][j] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I^0(M(x_l, y_l)) F_{i,j}(x_l, y_l) dx_l dy_l$$

The two steps can be written together as:

$$\begin{aligned} I^1[i][j] &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F_{i,j}^1(x_l, y_l) \sum_{k,l} B_{k,l}^0(M(x_l, y_l)) I^0[k][1] dx_l dy_l \\ &= \sum_{k,l} I^0[k][1] \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F_{i,j}(x_l, y_l) B_{k,l}^0(M(x_l, y_l)) dx_l dy_l \\ &= \sum_{k,l} I^0[k][1] T_{i,j,k,l} \end{aligned}$$

In the output image, each pixel will actually be some linear combination of the high resolution pixels, not just some subset of them.

The simplest example arises when one uses box functions as the basis functions and the filters, and shrinks the image by a factor of 2. In this case, one simply takes the average of 4 input pixels for each output pixel.

1.2.4 Texture Filtering

In texture mapping, we must deal with both problems: going from discrete to continuous, and from continuous to discrete. As we saw above, if I need to sample some non integer texture coordinate location, I need a continuous texture function to do so. This requires a reconstruction step. When creating the output image, I must properly filter the continuous output image to obtain the output pixels. If this is not done, we can get aliasing. For example, if my texture is a high resolution picture of a zebra and it

is mapped on to a polygon that is far from the camera, and takes up only a few pixels, we will need to filter down the zebra so we don't just get a random color at the output pixel. We will first look at the proper way to accomplish this, and then look at how it is done in practice.

Texture mapping produces an invertible mapping between image pixel coordinates and texture pixel coordinates

$$(x_t, y_t) = M(x_p, y_p)$$

where M is a projective transformation. The output image function (over the domain of the texture mapped triangle) is determined by the texture image

$$I(x_p, y_p) = T(M(x_p, y_p))$$

In a discrete setting, the texture is represented as a discrete texture image $T[i][j]$, and the output that we want to create is a discrete image $I[i][j]$.

The proper steps to create the output can be derived from our earlier discussion. It can be broken down into four steps

- reconstruct a continuous texture image from the discrete one $T(x_t, y_t) = \sum_{k,l} B_{k,l}(x_t, y_t)T[k][l]$.
- create the continuous output image by copying to colors at the proper texture coordinate locations $I(x_p, y_p) = T(x_t, y_t) = T(M(x_p, y_p))$.
- create the discrete output pixels by integrating against the filter

$$I[i][j] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x_p, y_p) F_{i,j}(x_p, y_p) dx_p dy_p$$

Putting this all together we get

$$I[i][j] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F_{i,j}(x_p, y_p) \sum_{k,l} B_{k,l}(M(x_p, y_p)) T[k][l] dx_p dy_p$$

This can be rearranged as

$$\begin{aligned} I[i][j] &= \sum_{k,l} T[k][l] \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F_{i,j}(x_p, y_p) B_{k,l}(M(x_p, y_p)) dx_p dy_p \\ &\equiv \sum_{k,l} T[k][l] T_{i,j,k,l} \end{aligned}$$

Because M is a projective transformation, computing the $T_{i,j,k,l}$ can be complicated. And so drastic approximations are usually done.

In practice, the approximation often used is a method called mip mapping. In mip mapping, one starts with an original texture T^0 creates a series of lower and lower

resolution textures 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 is done using the shrinking method described above.

During texture mapping, for each texture coordinate (x_t, y_t) , one computes how much shrinking the texture locally undergoes under the mapping M . This computation is used to select from an appropriately blurry texture T^i . When non integer coordinates are needed, bilinear interpolation is used to obtain a color value from T^i . To avoid “popping” that occurs when one switches the texture mip map levels i and $i + 1$, two colors are actually extracted, one from T^i and one from T^{i+1} . These two colors are then linearly interpolated. The interpolation factor is based on how close we are to choosing level i or $i + 1$. Using this extra linear interpolation gives what is called in the lingo trilinear interpolation.

It is easy to see that mip mapping does not do the exactly correct computation. Mip mapping implicitly assumes that locally, M looks like a simple image shrinking operation. But during texture mapping, some region of texture space may get blown up in one direction and shrunk in another. For example suppose i wish to texture map a checkerboard pattern on to a floor. If I look at some point on the floor up close, but at a very glancing angle, the texture will get blown up in the horizontal direction but shrunk in the vertical direction. Even with its deficiencies, mip mapping is the most popular way of producing antialiased textured images.

1.3 the alpha channel

Suppose i have two pictures, a foreground and background image, that i want to combine into one picture. For example I may want to superimpose the forground picture of a weatherman over the background image of a weathermap. The most obvious solution is to cut out the weatherman pixels from his image. In the composite image, we could simply place the weatherman pixels over the weathermap pixels.

Unfortunately, this may give us a jagged boundary between the weatherman/map transition regions. In a real image, with the man in front of the map, since the pixels are filtered, the boundary pixels will really be some average of the man and map colors.

This problem can be alleviated somewhat using an alpha channel. An alpha channel says how much of each pixel domain is covered. A 0 means no weatherman, a 1 means all weatherman. A fractional alpha value, say at the boundary of the man, means that the man covers part of the pixel. If the pixel is then treated as fractionally tranparant, then the composite pixel will be a blend between the forground and background pixel.

Lets look at how this approximation works Suppose for simplicity we are using a box function for the filter F , and so

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x, y) F_{i,j}(x, y) dx dy = \int \int_{\Omega_{i,j}} I(x, y) dx dy$$

where $\Omega_{i,j}$ is the support of pixel i, j . Let us define the coverage function $C(x, y)$ as the binary valued function that is 1 where the weatherman is, and 0 where he is not. At a pixel i, j , let us store not just an intensity value (or color value) $I[i][j]$ but also an alpha value $\alpha[i][j]$ defined as follows

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

Suppose i wish to create a composite image $I^c[i][j]$ by placing a foreground image $I^f(x, y)$ over a background image $I^b(x, y)$. The correct computation for the output pixel can be computed as by simply integrating both the foreground and background images, using their masks to determine where they are valid. Then we remove the background where it is covered by the foreground

$$\begin{aligned} I^c[i][j] &= \int \int_{\Omega_{i,j}} dx dy I^f(x, y)C^f(x, y) + I^b(x, y)C^b(x, y) - I^b(x, y)C^b(x, y)C^f(x, y) \\ &= \int \int_{\Omega_{i,j}} dx dy I^f(x, y)C^f(x, y) + \int \int_{\Omega_{i,y}} dx dy I^b(x, y)C^b(x, y) - \\ &\quad \int \int_{\Omega_{i,y}} dx dy I^b(x, y)C^b(x, y)C^f(x, y) \\ &\approx \int \int_{\Omega_{i,j}} dx dy I^f(x, y)C^f(x, y) + \int \int_{\Omega_{i,j}} dx dy I^b(x, y)C^b(x, y) - \\ &\quad \int \int_{\Omega_{i,j}} dx dy I^b(x, y)C^b(x, y) \cdot \int \int_{\Omega_{i,j}} dx dy C^f(x, y) \\ &= \int \int_{\Omega_{i,j}} dx dy I^f(x, y)C^f(x, y) + \left(\int \int_{\Omega_{i,j}} dx dy I^b(x, y)C^b(x, y) \right) \left(1 - \int \int_{\Omega_{i,j}} dx dy C^f(x, y) \right) \\ &= I^f[i][j] + I^b[i][j](1 - \alpha^f[i][j]) \end{aligned}$$

and likewise, alpha can be computed as

$$\alpha^c[i][j] = \alpha^f[i][j] + \alpha^b[i][j](1 - \alpha^f[i][j])$$

All of the above manipulation was straightforward except for the line beginning with the (\approx). That approximation is only true if the coverage masks of the foreground and background are uncorrelated. This is an assumption which we have to accept when using alpha blending.

Under this assumption, I can obtain the correct composite discrete image simply by applying the pixel updates described above. This operation is called the discrete “over” operation. Because alpha is such an important part of representing images when doing image manipulation that it is often considered a fourth channel: red, green, blue,

alpha. The over operation is associative but not commutative. So one can composite a set of layers using any choice for parenthesis (reordering the operations), but one cannot reorder the layers. Because of this, alpha compositing requires that the layers are sorted in depth.

Because a pixel with a fractional alpha value behaves as if it is partially transparent, alpha is also often used to represent not coverage but transparency. This is just an approximation to the ways that real transparent objects behave. In the real world, transparent objects have different opacity coefficients for each wavelength of light (some glass only lets blue through). In the real world, transparent objects usually refract the light that passes through it.

Graphics APIs, like OpenGL, let you use alpha when describing pixel values. They also let one specify the α values of vertices as if it were a color. But since the over operation is not commutative, the user is responsible for sorting the layers. It is very difficult to use alpha blending together with the z-buffer algorithm.

Note, some image formats instead of storing I and α store I/α and α . This is called the “non premultiplied” format. The reason for using this format has to do with data precision. When a pixel has a small coverage value, then I will be a small number, using only a few bits. In the non premultiplied form, one can use all of the bits to describe the color independent of the coverage. So images are often stored with 8 bits per channel in non premultiplied format. When they are manipulated in a compositing program, floating point numbers are used, and the values are transferred to the usual premultiplied format.

1.4 Monte Carlo integration

As described above, when computing the color for a pixel during the rendering of a 3D scene, one must compute a definite integral. Even if one only uses box functions as the filters, one still must compute an unweighted continuous average over the pixel domain. Computing these integrals analytically is usually not possible, and we must resort to numerical approximations.

The basic way to numerically approximate an integral is to turn it into a discrete sum over a set of discrete samples.

$$I[i][j] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x, y) F_{i,j}(x, y) dx dy \approx \sum_k I(x_k, y_k) F_{i,j}(x_k, y_k) w_{i,j,k}$$

where w_k is some weight. In general, this approach is called quadrature, and there is a vast literature on various quadrature schemes. The simplest is to use an evenly spaced set of samples. This is essentially what we did when we used supersampling and averaging to obtain the pixel value.

Deterministic quadrature schemes use a fixed set of sample locations x_k, y_k . There

are some drawbacks to these deterministic methods. First of all, since we are committing to use some fixed sampling pattern, there are “worst cases” where our samples will do a bad job of approximating the function. Second of all, if I use a regular pattern of samples, error tends to show up as patterned noise. Third of all, basic quadrature methods are very expensive when the dimension of the domain is high. If I want m evenly spaced samples in each of d dimensions, I will need m^d samples. In general, the best deterministic methods have a worst case error of $\Omega(n^{-r/d})$ where n is the number of samples, d is the dimension and r is the degree of continuity. For image antialiasing, the dimension is just two, but later on, when we study realistic simulation of light transport, we will need to perform integrals over much higher dimensional domains.

In contrast to the deterministic methods, monte carlo integration is a method that uses randomness to determine the sample locations. When using randomness, there is no function that is a “worst case” problem to integrate. Secondly, any error that does occur will appear as noise instead of as a pattern of error. Noise is typically considered more acceptable perceptually than patterned error. Finally, monte carlo methods are not penalized by high dimensional domains. For monte carlo, the expected error is $O(n^{-1/2})$.

1.4.1 Basic monte carlo

Here is how monte carlo works. Suppose I want to compute the following definite integral

$$\text{ans} = \int_{\Omega} dx f(x)$$

To perform monte carlo integration, I first pick any random distribution $p(x)$ over the domain Ω . A random distribution is a function that is positive everywhere, and has unit area.

$$1 = \int_{\Omega} dx p(x)$$

The next step is to generate a random sample x using the distribution $p(x)$. The monte carlo method returns as its estimate of the integral

$$\text{est} = f(x)/p(x)$$

It is just that simple.

To see how this estimate behaves, we can compute the expected value of “est”. We do this by integrating over all possible x values, weighted by their probability and their outcome.

$$E \left[\frac{f(x)}{p(x)} \right] = \int_{\Omega} dx \frac{f(x)}{p(x)} p(x) = \int_{\Omega} dx f(x)$$

So we see that the expected value of this monte carlo integral is the desired answer. Such an estimate is called unbiased. Amazingly, the monte carlo estimate is unbiased for any choice of $p(x)$.

The meaning of unbiased is that if we were to run the estimate a zillion times, the average value obtained would be the correct one. It does not mean that if we run the estimator once, that we don't expect any error. In fact one can calculate what the expected squared error will be, this is called the variance.

$$V = E[(\text{est} - \text{ans})^2]$$

The variance of the estimator turns out not to be independent of the choice for p . Better choices for p are ones that are more likely to sample regions of x where the function value $f(x)$ is large. This approach is called importance sampling.

The simplest way to obtain an estimator with less variance is to use more samples. The easiest way to do this is to pick n independent samples x_p under the distribution $p(x)$, run the single sample estimator n times, and simply average the result.

$$\text{est} = \frac{1}{n} \sum_p \frac{f(x_p)}{p(x_p)}$$

It can be shown that this n sample estimator has variance V/n where V is the variance of the single sample estimator. Therefore the expected error of the n sample estimator is $O(n^{-1/2})$.

In some cases, taking n independent samples can be less than optimal. In particular, n random samples will typically be clumped up in some regions and sparse in other regions. One way to compromise between randomness, and even coverage is to use "jittered" or stratified sampling. The basic idea is to break up the domain into r disjoint regions. Then within each region, an independent monte carlo estimate is run to compute the integral over each region. These estimates are then combined to create the estimate for the whole region. For example, one may break up a pixel square domain into four subsquares, pick a random sample in each subsquare and average the four resulting numbers together.

Monte carlo integration isn't very applicable to scan conversion rendering algorithms, which rely on using incremental computation while taking even steps over the image domain. Later we will discuss the ray tracing approach to rendering. In this case, one is free to sample the image plane in any particular way one wishes.

Monte carlo integration is a rich topic within computer graphics and once again, we have only scratched the surface of these methods.