CS380: Introduction to Computer Graphics
Reconstruction & Resampling
Chapter 17 & 18

Min H. Kim
KAIST School of Computing

Sampling (continuous → discrete)

SUMMARY

Aliasing

- 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.
  - Jaggies will crawl during motion

Over-sampling

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

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

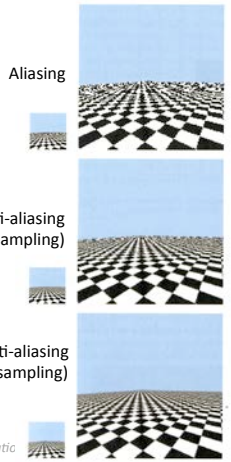
where *k* indexes some set of locations (x_k, y_k) called the sample locations.

- 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.

Multi-sampling

- 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.

Aliasing vs. anti-aliasing

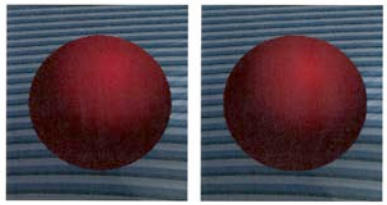


Over operation

- To compose $I^f[i][j]$ over $I^b[i][j]$, we compute the composite image colors, $I^c[i][j]$, using

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

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



Chapter 17

RECONSTRUCTION
(DISCRETE → CONTINUOUS)

Reconstruction

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

Constant reconstruction

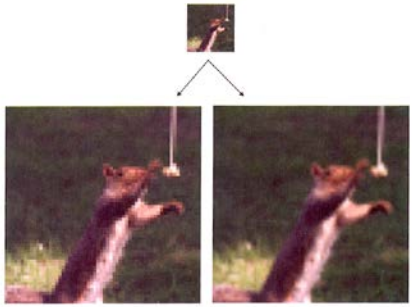
- A real valued image coordinate is assumed to have the color of the closest discrete pixel. This method can be described by the following pseudo-code:

```
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.*

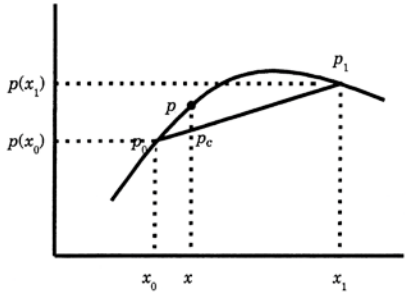
Constant reconstruction

- The resulting continuous image is made up of little squares of constant color.
- Each pixel has an influence region of 1-by-1



linear interpolation

- Linear interpolation (1D):
  $$p_c(x) = p(x_0) + [(x - x_0)/(x_1 - x_0)][p(x_1) - p(x_0)].$$
- Interpolation error:



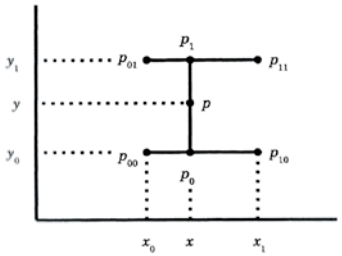
Bilinear interpolation

- Bilinear interpolation (2D):

$$p_0(x) = p_{00} + [(x - x_0)/(x_1 - x_0)](p_{10} - p_{00}).$$

$$p_1(x) = p_{01} + [(x - x_0)/(x_1 - x_0)](p_{11} - p_{01}).$$

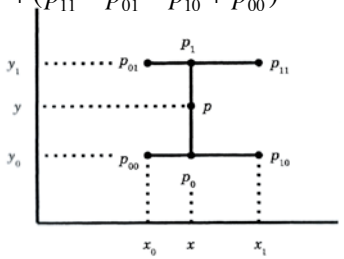
$$p(x,y) = p_0 + [(y - y_0)/(y_1 - y_0)](p_1 - p_0).$$



Bilinear interpolation

- Bilinear interpolation (2D):

$$\begin{aligned} p(x,y) = p_{00} &+ [(x - x_0)/(x_1 - x_0)](p_{10} - p_{00}) \\ &+ [(y - y_0)/(y_1 - y_0)](p_{01} - p_{00}) \\ &+ [(x - x_0)/(x_1 - x_0)][(y - y_0)/(y_1 - y_0)] \\ &+ (p_{11} - p_{01} - p_{10} + p_{00}) \end{aligned}$$



Bilinear reconstruction

- 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);
}
```


Bilinear properties

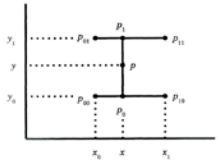
- At integer coordinates, we have *I(x,y)*=I[i][j]; the reconstructed continuous image *I* agrees with the discrete image I.
- In between integer coordinates, the color values are blended continuously.
- Each pixel in the discrete image influences, to a varying degree, each point within a 2-by-2 square region of the continuous image.
- The horizontal/vertical ordering is irrelevant.
- Color over a square is bilinear function of (x,y).

Bilinear function

- 1 by 1 square with coordinates $i < x < i+1$ $j < y < j+1$ for some fixed i and j .

$$I(i+x_f, j+y_f) \leftarrow (1-y_f)\{(1-x_f)\mathrm{I[i][j]}+(x_f)\mathrm{I[i+1][j]}\} (y_f)\{(1-x_f)\mathrm{I[i][j+1]}+(x_f)\mathrm{I[i+1][j+1]}\},$$

- where x_f and y_f are the fracx and fracy above.



Bilinear function

- Rearranging the terms,

$$\begin{aligned} I(i+x_f, j+y_f) \leftarrow\ & \mathrm{I[i][j]} \\ & +(-\mathrm{I[i][j]} + \mathrm{I[i+1][j]})x_f \\ & +(-\mathrm{I[i][j]} + \mathrm{I[i][j+1]})y_f \\ & +(\mathrm{I[i][j]} - \mathrm{I[i][j+1]} - \mathrm{I[i+1][j]} + \mathrm{I[i+1][j+1]})x_f y_f \end{aligned}$$

- This function has terms that are constant, linear, and bilinear terms in the variables (x_f, y_f)

Bilinear basis function

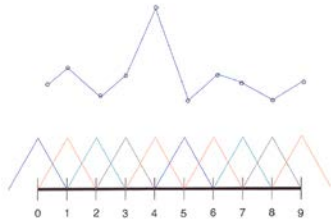
- Rearrange the bilinear function to obtain:

$$\begin{aligned} I(i+x_f, j+y_f) \leftarrow\ & (1-x_f-y_f+x_f y_f)\mathrm{I[i][j]} \\ & +(x_f - x_f y_f)\mathrm{I[i+1][j]} \\ & +(y_f - x_f y_f)\mathrm{I[i][j+1]} \\ & +(x_f y_f)\mathrm{I[i+1][j+1]} \end{aligned}$$

- For a fixed position (x_f, y_f) , the color of the continuous reconstruction is linear in the discrete pixel values of I: $I(x,y) \leftarrow \sum_{i,j} B_{i,j}(x,y)\mathrm{I[i][j]}$

Bilinear basis function

- These *B* are called *basis functions (tent functions)*
- They describe how much pixel i, j influences the continuous image at $[x,y]^t$.
- In 1D, we can define a univariate *hat function* $H_i(x)$.



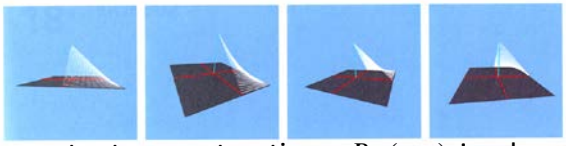
$$H_i(x) = \begin{cases} x-i+1 & \text{for } i-1 < x < i \\ -x+i+1 & \text{for } i < x < i+1 \\ 0 & \text{else} \end{cases}$$

Bilinear basis function

- In 2D (bilinear function), let $T_{i,j}(x,y)$ be a bivariate function:

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

- This is called a tent function



- In constant reconstruction, $B_{i,j}(x,y)$ is a box function that is zero everywhere except for the unit square surrounding the coordinates (i,j), where it has constant value 1.

Chapter 18

RESAMPLING

(RECONSTRUCTION+SAMPLING, DISCRETE→CONTINUOUS→DISCRETE)

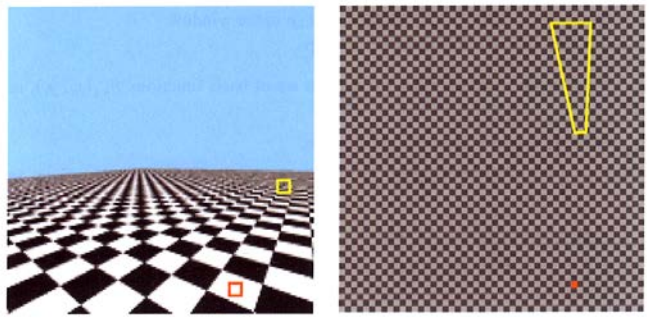
Resampling

- 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 texture image (discrete) T[k][l] and apply some 2D warp to this image to obtain an output image I[i][j].
- Reconstruct a continuous texture $T(x_t, y_t)$ using a set of basis functions $B_{k,l}(x_t, y_t)$.
- Apply the geometric wrap (at the view point) to the continuous image.
- Integrate against a set of filters $F_{k,l}(x_w, y_w)$ (a box filter) to obtain the discrete output image.

Resampling equation



Resampling equation

- Let the geometric transform be described by a mapping $M(x_w, y_w)$ which maps from continuous window to texture coordinates.
- We obtain:

$$\mathrm{I[i][j]} \leftarrow \iint_{\Omega} F_{i,j}(x_w, y_w)\left(\sum_{k,l} B_{k,l}[M(x_w, y_w)]\mathrm{T[k][l]}\right) dx_w dy_w$$

$$= \sum_{k,l} \mathrm{T[k][l]}\left(\iint_{\Omega} F_{i,j}(x_w, y_w)\left(B_{k,l}[M(x_w, y_w)]\right) dx_w dy_w\right)$$

(we could obtain an output pixel as a linear combination of the input texture pixels.)

Resampling equation

- We can rewrite the integration over the texture domain, instead of the window domain.

$$\mathrm{I[i][j]} \leftarrow \iint_{M(\Omega_{i,j})} \left(\left|\det(D_N)\right| F_{i,j}[N(x_t, y_t)]\right)\left(\sum_{k,l} B_{k,l}\mathrm{T[k][l]}\right) dx_t dy_t$$

$$= \iint_{M(\Omega_{i,j})} \left(\left|\det(D_N)\right| F'_{i,j}(x_t, y_t)\right)\left(\sum_{k,l} B_{k,l}\mathrm{T[k][l]}\right) dx_t dy_t$$

- where
  $N = M^{-1}$ is the inverse mapping $(x_w, y_w) = M^{-1}(x_t, y_t)$
  D_N is the Jacobian matrix of *N* (changes of variables)
  $F' = F \circ N$ (continuity of composition of *F* and *N*)

Resampling equation

- In summary, when *F* is a box filter

$$\mathrm{I[i][j]} \leftarrow \iint_{\Omega}\left(\sum_{k,l} B_{k,l}[M(x_w, y_w)]\mathrm{T[k][l]}\right) dx_w\, dy_w$$

$$= \iint_{M(\Omega_{i,j})} \left|\det(D_N)\right|\left(\sum_{k,l} B_{k,l}\mathrm{T[k][l]}\right) dx_t dy_t$$

- where $B_{k,l}(p)$ is how much texel *k,l* blends to obtain the value at continuous texture location *p,* from reconstruction step.

Summary

- That is, we need to integrate over the region $M(\Omega_{i,j})$ on the texture domain, and blend that data together.
- When our transformation *M* effectively shrinks the texture, then $M(\Omega_{i,j})$ has a large footprint over $T(x_t, y_t)$
- If *M* 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* 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

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

where $(x_t, y_t) = M(i, j)$

Blow up

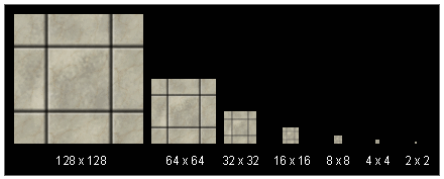
- We tell OpenGL to do this using the call glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR).
- For a single texture lookup in a fragment shader, the hardware needs to fetch 4 texture pixels and blend them appropriately.

Shrinking

- In the case that a texture is getting shrunk down, then, to avoid aliasing, the filter 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 T^0 and then creates a series of lower and lower resolution (blurrier) texture T^i.
- Each successive texture is twice as blurry. And because they have successively less detail, they can be represented with ½ the number of pixels in both the horizontal and vertical directions.



Mip mapping

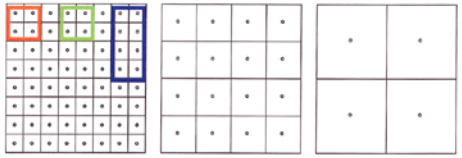
- This collection, called a mip map, is built before the any triangle rendering is done.
- Thus, the simplest way to construct a mip map is to average two-by-two pixel blocks to produce each pixel in a lower resolution image.
- During texture mapping, for each texture coordinate (x_t, y_t), the hardware estimates how much shrinking is going on.
  - How big is the pixel footprint on the geometry.

Mip mapping

- 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/ where the texture mip map switches between levels, we can 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

- 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.



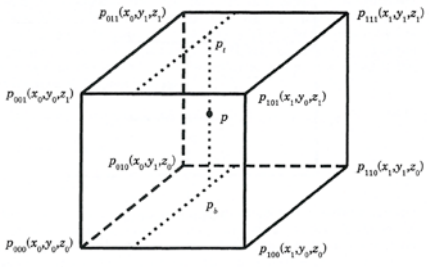
Trilinear interpolation

- Trilinear interpolation (3D):

$$p(x,y,z) = c_0 + c_1\Delta x + c_2\Delta y + c_3\Delta z + c_4\Delta x\Delta y + c_5\Delta x\Delta z$$
$$c_6\Delta y\Delta z + c_7\Delta x\Delta y\Delta z$$

$p_0 + [0,1,n,n+1,n^2,$
$n^2+1,n^2+n,n^2+n+1]$

The dimension of CLUT is (2n+1)x(2n+1)x(2n+1)



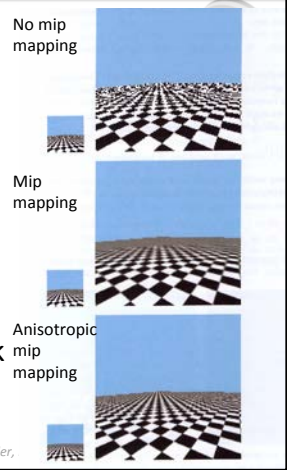
Trilinear interpolation

- where

$$\Delta x = x - x_0$$
$$\Delta y = y - y_0$$
$$\Delta z = z - z_0$$
$$c_0 = p_{000}$$
$$c_1 = (p_{100} - p_{000})/(x_1 - x_0)$$
$$c_2 = (p_{010} - p_{000})/(y_1 - y_0)$$
$$c_3 = (p_{001} - p_{000})/(z_1 - z_0)$$
$$c_4 = (p_{110} - p_{010} - p_{100} + p_{000})/[(x_1 - x_0)(y_1 - y_0)]$$
$$c_5 = (p_{101} - p_{001} - p_{100} + p_{000})/[(x_1 - x_0)(z_1 - z_0)]$$
$$c_6 = (p_{011} - p_{001} - p_{010} + p_{000})/[(y_1 - y_0)(z_1 - z_0)]$$
$$c_7 = (p_{111} - p_{011} - p_{101} - p_{110} + p_{100} + p_{001} + p_{010} - p_{000})/$$
$$[(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)].$$

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.



Properties

- 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 ansiotropic filtering and can be abled in an API or using the driver control panel.

Announcement

- Homework7 deadline:
  - 2015.05.24 (Sun) 11:55pm