Announcement

- Homework #5 Due June 14 (Midnight)
  - Late submission allowed only upto *June 21*.

	2	Sampling	Chap 16
today	7	Samplling/Reconstruction	Chap 16/17/18
	8	Open Lab	
	9	Geometirc modeling	Chap 22
	14	Animation	Chap 23
	21	Final Exam	

- Final Exam
  - Tuesday June 21 4PM E3-1, #1501
  - Reading: all chapters covered - omit 19 (Color)

Contest

<optional but highly-recommended!>

- Best image of a virtual floppy cube
  - Created by you
  - 512 x 512 (any standard image format is fine)
  - High quality rendered output
  - Title & short description will be nice to have

  - Enter by June 15 Midnight to KLMS

  - **Winner(s)** will be announced at the final on June 21 and receive a prize!

Reconstruction Resampling

Chapter 17

Chapter 18

Last lecture: Sampling

- 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

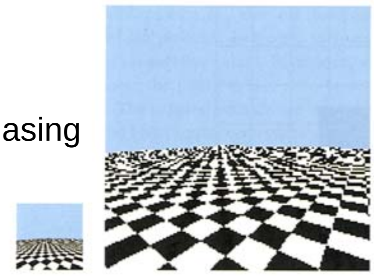
https://www.youtube.com/watch?v=Uan1L3NuWSY (3:00~)

Anti-aliasing

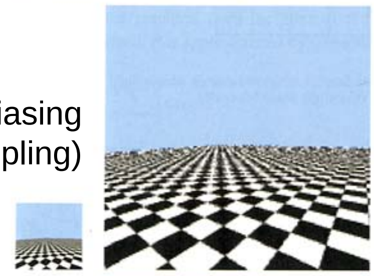
- Oversampling
- Supersampling
- Multisampling

- Over operation
  - Blending

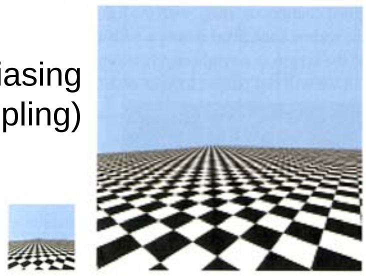
Aliasing



Anti-aliasing
(multi-sampling)

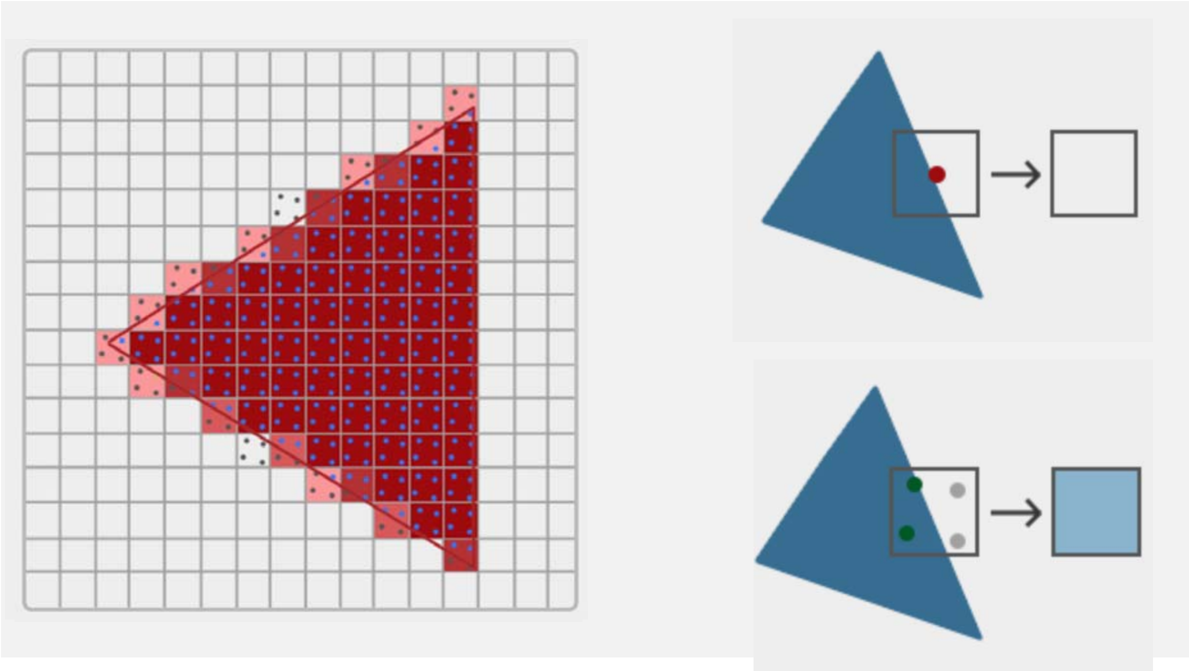


Anti-aliasing
(super-sampling)

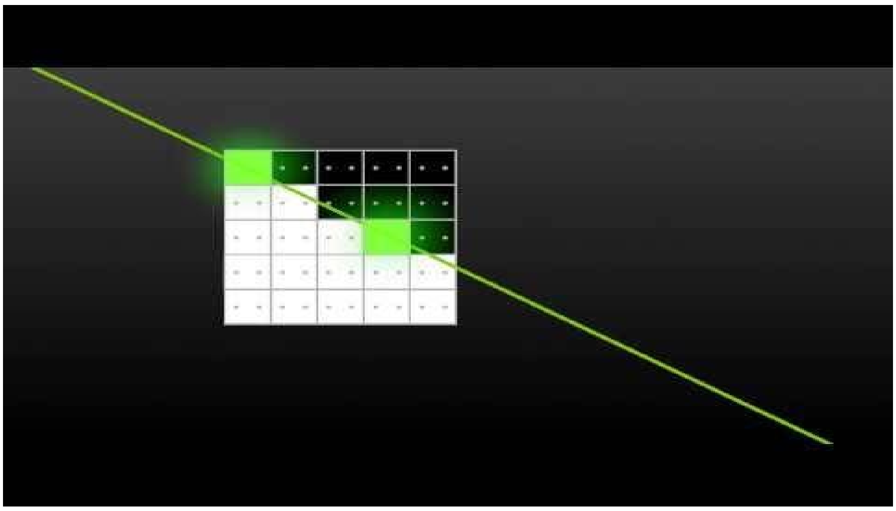


http://www.learnopengl.com/#!Advanced-OpenGL/Anti-Aliasing

Multisampling



GeForce: MFAA



Chapter 16 (Sampling)

- This chapter deals with the '*image*' to 'screen'
  - Picture → collection of pixels
    - '.. is and artifact that depicts or records visual perception'
  - Continuous image $I(x_w, y_w)$: a bivariate function
  - Discrete image `I[i][j]` : two dimensional array of color values
    - We associate each pair of integers `i,j`, with the continuous image coordinates $x_w = i$ and $y_w = j$

Chapter 17 (Reconstruction)

- Texture to *Image*
- Given a discrete image `I[i][j]` how do we create a continuous image $I(x, y)$?
- 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

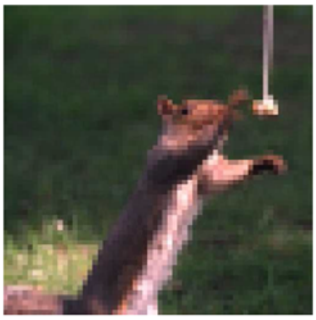
- The resulting continuous image is made up of little squares of constant color.
- Each pixel has an influence region of 1-by-1

"magnification problem'

Solution:

Nearest-neighbor

Bilinear interpolation

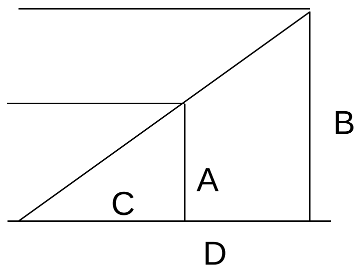


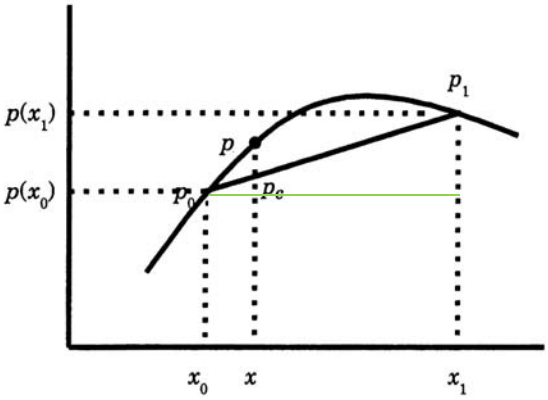
Linear interpolation

- Linear interpolation (1D):

$$p_c(x) = p(x_0) + [(x - x_0)/(x_1 - x_0)][p(x_1) - p(x_0)].$$

- Interpolation error:

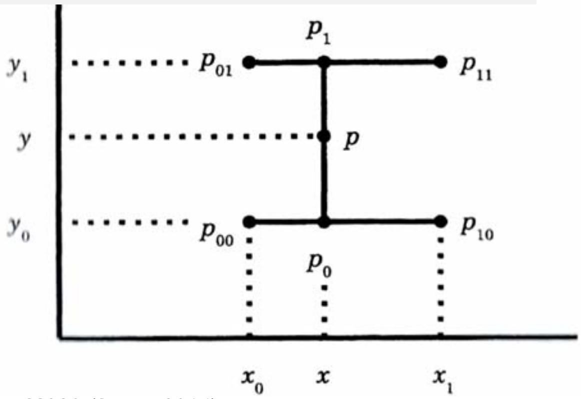




Bilinear interpolation

- Bilinear interpolation (2D):

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



Bilinear interpolation

- 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 interpolation

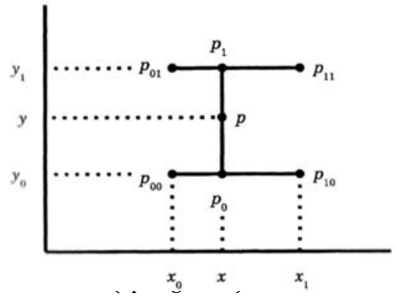
- At integer coordinates, we have $I(i,j) = \mathtt{I[i][j]}$; the reconstructed continuous image I agrees with the discrete image $\mathtt{I}$.
- In between integer coordinates, the color values are blended *continuously*.
- Each pixel in the discrete image influences, to a varying degree, each point 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 interpolation

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

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

where x_f and y_f are the fracx and fracy in the code.



Bilinear interpolation

- Rearranging the terms,

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

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

Bilinear basis function

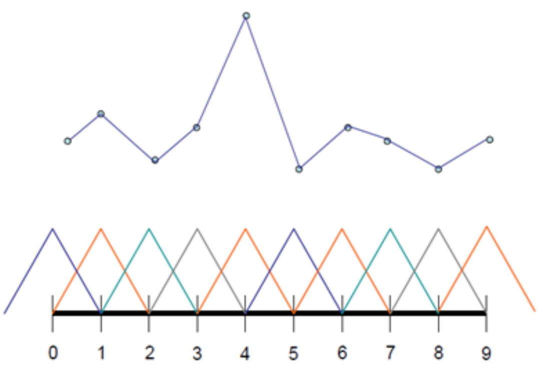
- Rearrange the bilinear function to obtain:

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

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

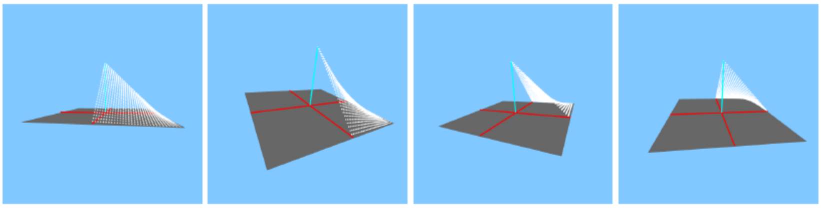
Bilinear basis function

- 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* $H_i(x)$.



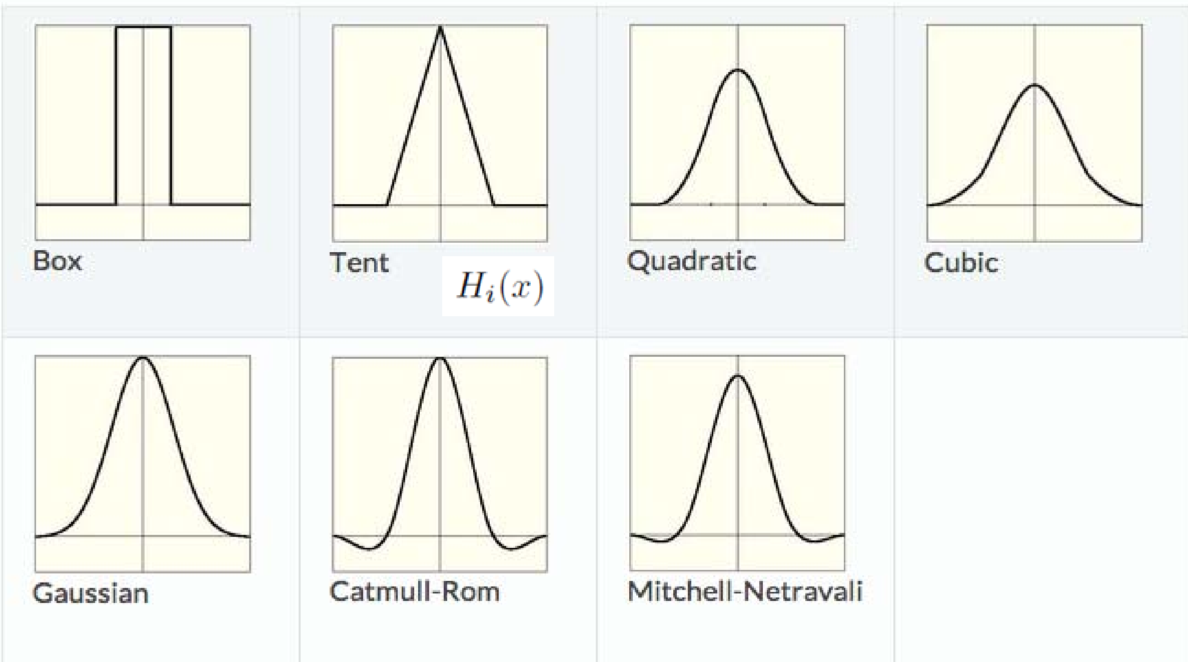
$$H_i(x) = \begin{cases} x-i+1 & \text{for} \quad i-1 < x < i \\ -x+i+1 & \text{for} \quad i < x < i+1 \\ 0 & \text{else} \end{cases}$$

- 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.

Filters (from last lecture)



Chapter 19 Resampling

- Reconstruction + Sampling
  - discrete → continuous → discrete
    (texture) (image screen)
- 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 $B_{k,l}(x_t, y_t)$
  - Texture to *image* (chapter 17)
- Apply *the geometric wrap* (at the view point) to the continuous image.
- Integrate against a set of filters $F_{i,j}(x_w, y_w)$ (a box filter) to obtain the discrete output image.
  - *Image* to screen (chapter 16)

Resampling equation

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

$$\texttt{I[i][j]} \leftarrow \int\int_{\Omega} dx_w\, dy_w\, F_{i,j}(x_w, y_w) \sum_{k,l} B_{k,l}(M(x_w, y_w)) \texttt{T[k][l]}$$

$$= \sum_{k,l} \texttt{T[k][l]} \int\int_{\Omega} dx_w\, dy_w\, F_{i,j}(x_w, y_w) B_{k,l}(M(x_w, y_w))$$

$\boxed{T(x_t, y_t)}$

Resampling equation

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

$$\texttt{I[i][j]} \leftarrow \int\int_{M(\Omega)} dx_t\, dy_t\, |\det(\mathrm{D_N})|\;\; F_{i,j}(N(x_t, y_t)) \sum_{k,l} B_{k,l}(x_t, y_t) \texttt{T[k][l]}$$

$$= \int\int_{M(\Omega)} dx_t\, dy_t\, |\det(\mathrm{D_N})|\;\; F'_{i,j}(x_t, y_t) \sum_{k,l} B_{k,l}(x_t, y_t) \texttt{T[k][l]}$$

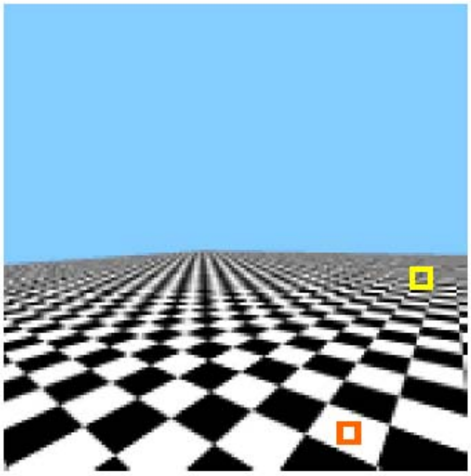
where D_N is the Jacobian of N and $F' = F \circ N$.

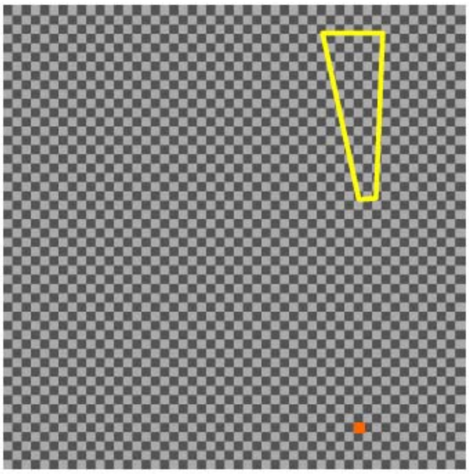
When F is a box filter, this becomes

$$\texttt{I[i][j]} \leftarrow \int\int_{M(\Omega_{i,j})} dx_t\, dy_t\, |\det(\mathrm{D_N})| \sum_{k,l} B_{k,l}(x_t, y_t) \texttt{T[k][l]}$$

R

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 $T(x_t, y_t)$.





Blow up

- In the case that we are blowing up (zooming in) 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, <u>the integration step can be dropped</u>, and the resampling can be implemented as

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

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)
```

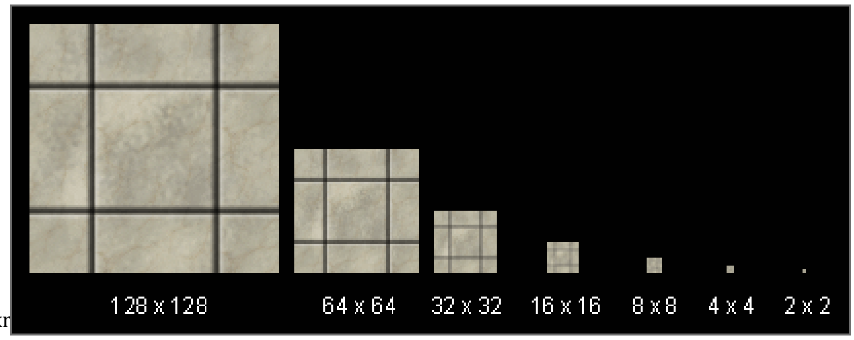
- For a single texture lookup in a fragment shader, the hardware needs to fetch 4 texture pixels and blend them appropriately.

Shirking

- 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

- 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.

Mip mapping

- 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 with trilinear interpolation is specified with the call

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER.
                GL_LINEAR_MIPMAP_LINEAR).
```


Trilinear interpolation

- Trilinear interpolation (3D):

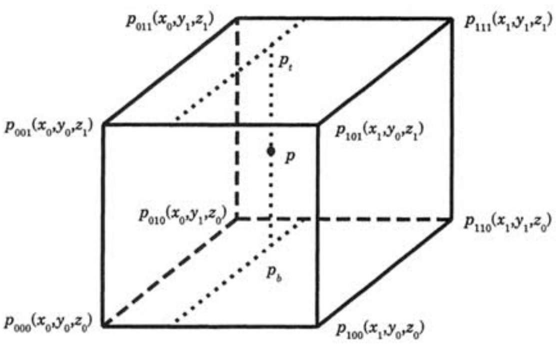
$$p(x,y,z) = c_0 + c_1\Delta x + c_2\Delta y + c_3\Delta z + c_4\Delta x\Delta y + c_5\Delta x\Delta z$$
$$c_6\Delta y\Delta z + c_7\Delta x\Delta y\Delta z$$

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

$$\Delta x = x - x_0$$
$$\Delta y = y - y_0$$
$$\Delta z = z - z_0$$

where

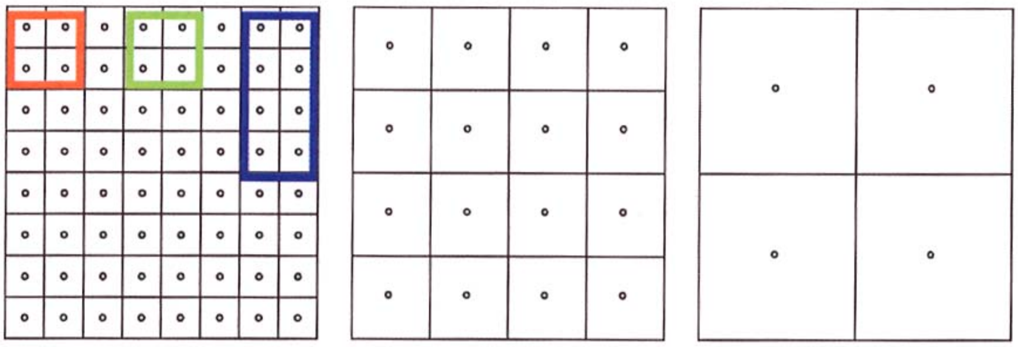
$$c_0 = p_{000}$$



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

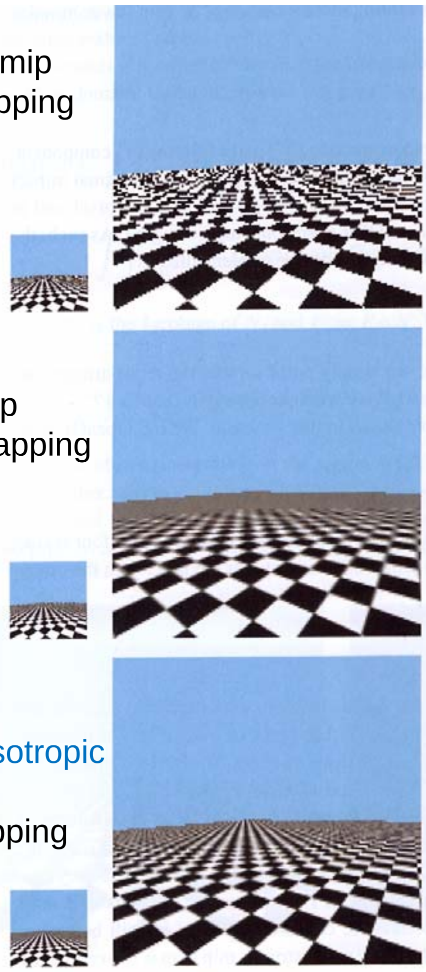
Mip mapping

- Trilinear interpolation requires OpenGL to <u>fetch 8 texture pixels </u>and blend them appropriately for every requested texture access.



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 <u>isotropic shrinking</u>, 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 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 abled in an API or using the driver control panel.

Summary

- https://www.youtube.com/watch?v=FT6WYIXNTsA (1:40, 2:10, 3:07 2:20)