Introduction to Computer Graphics

9. Buffers and Mapping techniques

I-Chen Lin

National Chiao Tung University

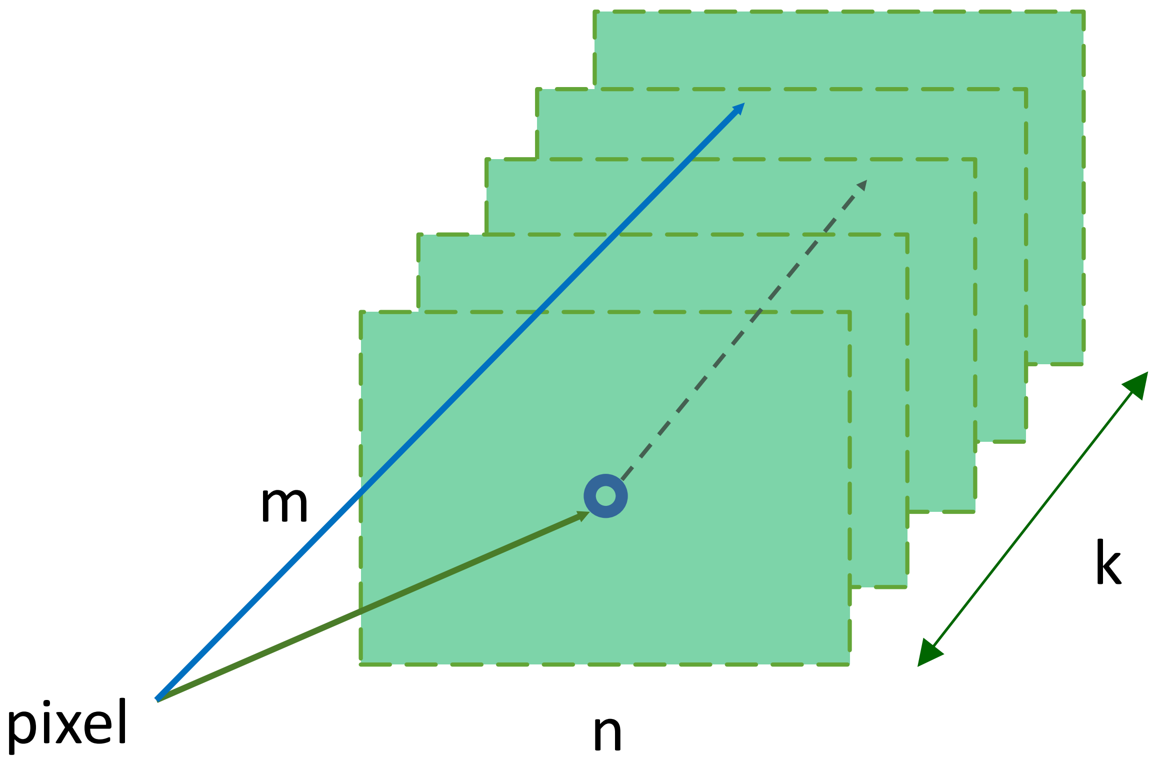
Textbook: E.Angel, D. Shreiner Interactive Computer Graphics, 6th Ed., Pearson
Ref: D.D. Hearn, M. P. Baker, W. Carithers, Computer Graphics with OpenGL, 4th Ed., Pearson

Outline

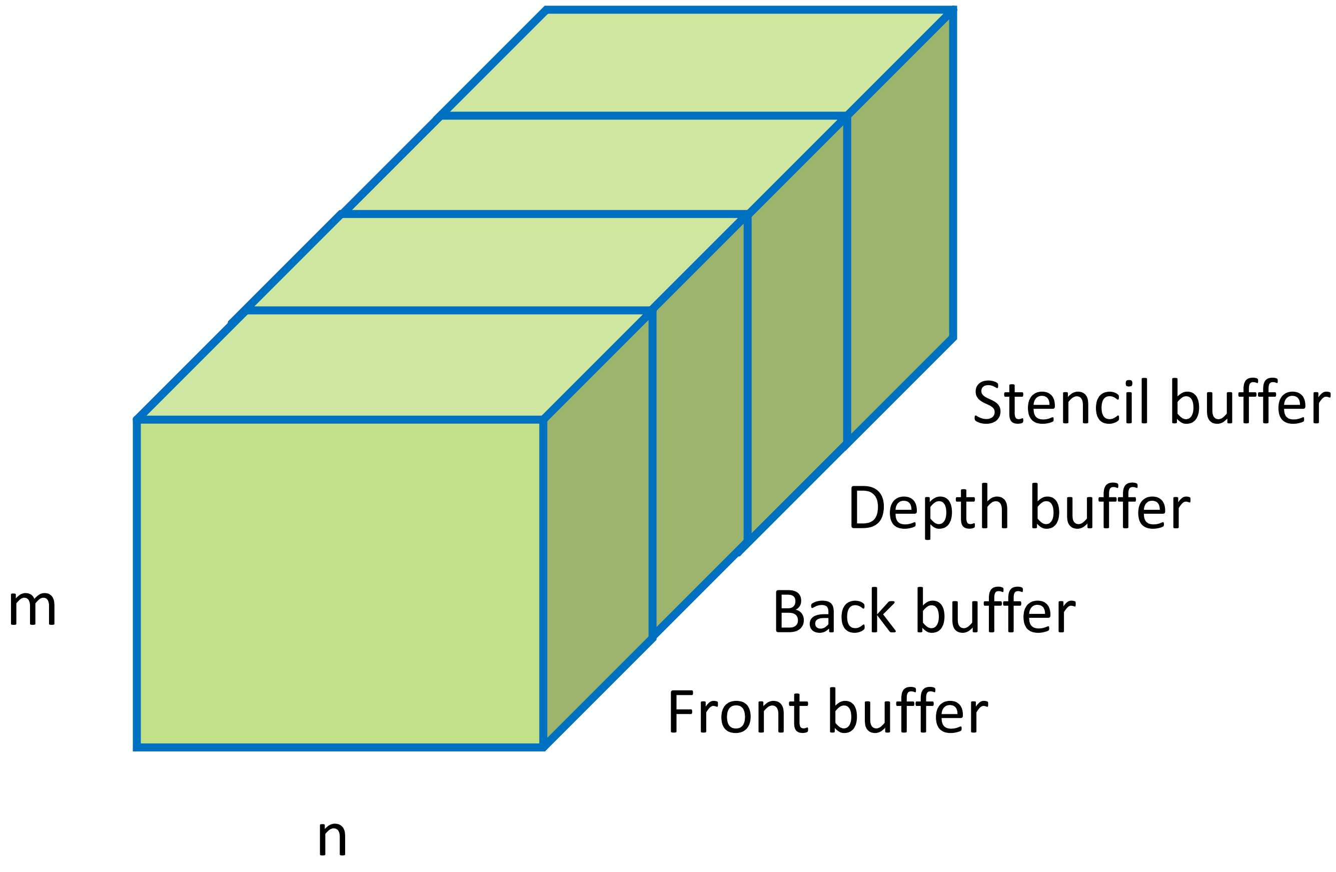
- Buffers
- Mapping techniques
- Anti-aliasing

Buffer

- Define a buffer by its spatial resolution (*n* x *m*) and its depth (or precision) *k*, the number of bits/pixel



OpenGL Frame Buffer



Buffers

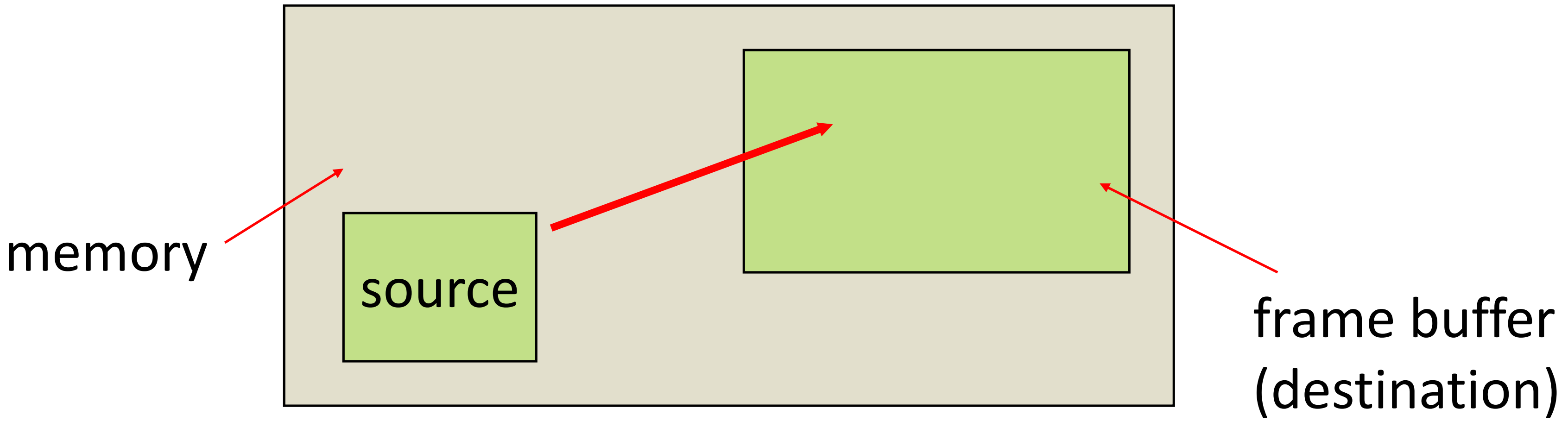
- Color buffers can be displayed
  - Front
  - Back
  - ……
- Depth
- Accumulation

Note: glAccum deprecated in newer OpenGL versions

- Stencil

Writing in Buffers

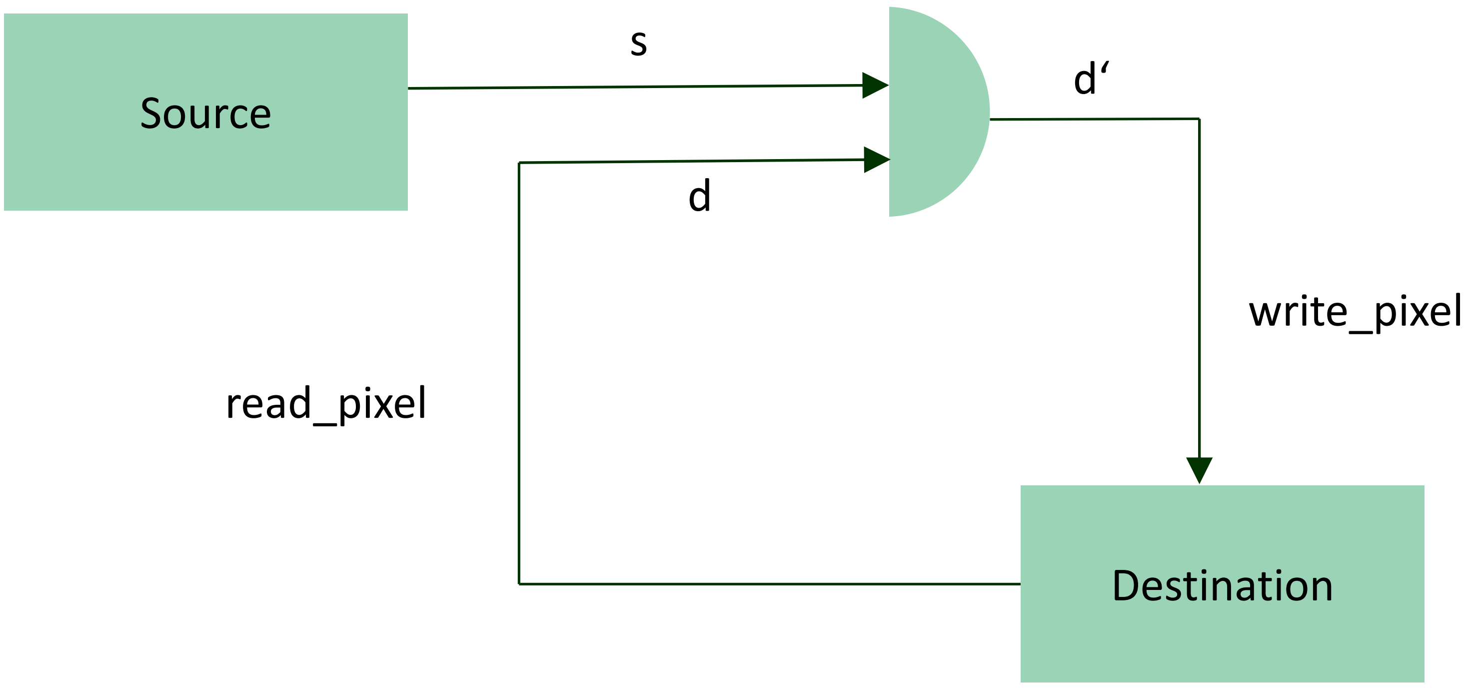
- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
  - Bit block transfer (bitblt) operations
- The frame buffer is part of this memory



writing into frame buffer

Writing Model

- Read destination pixel before writing source



Bit Writing Modes

- Source and destination bits are combined bitwise
- 16 possible functions (one per column in table)

replace XOR OR

s	d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Mapping Methods

- Texture Mapping
- Environment Mapping
- Normal and Bump Mapping

The Limits of Geometric Modeling

- Although graphics cards can render over 10 million polygons per second, the number is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin

Modeling an Orange

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Does not capture surface characteristics (small dimples)
  - Takes too many polygons to model all the dimples

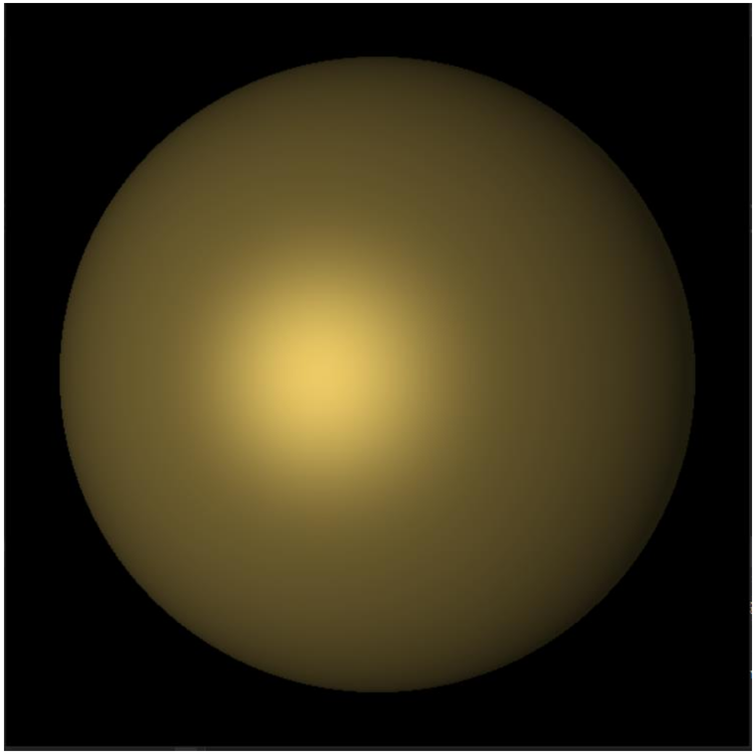
Modeling an Orange (cont.)

- Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  - This process is known as *texture mapping*
- Still might not be sufficient because the resulting surface will be smooth
  - Need to change local shape
  - Bump mapping

Three Types of Mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environment (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during the rendering process

Texture Mapping

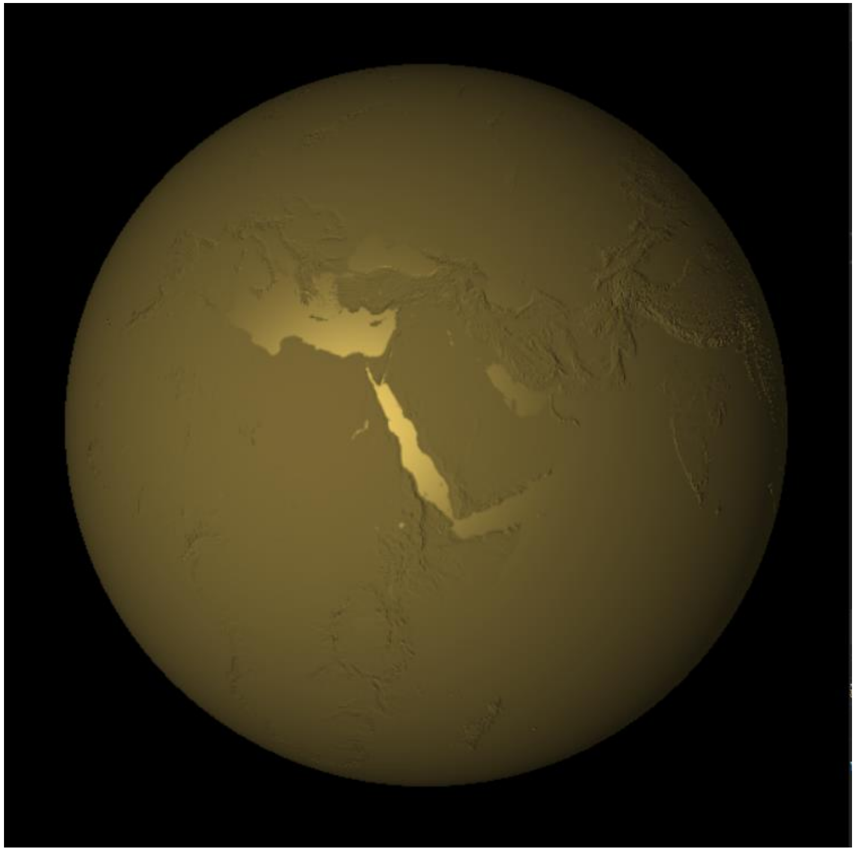


Geometric model

Texture-mapped

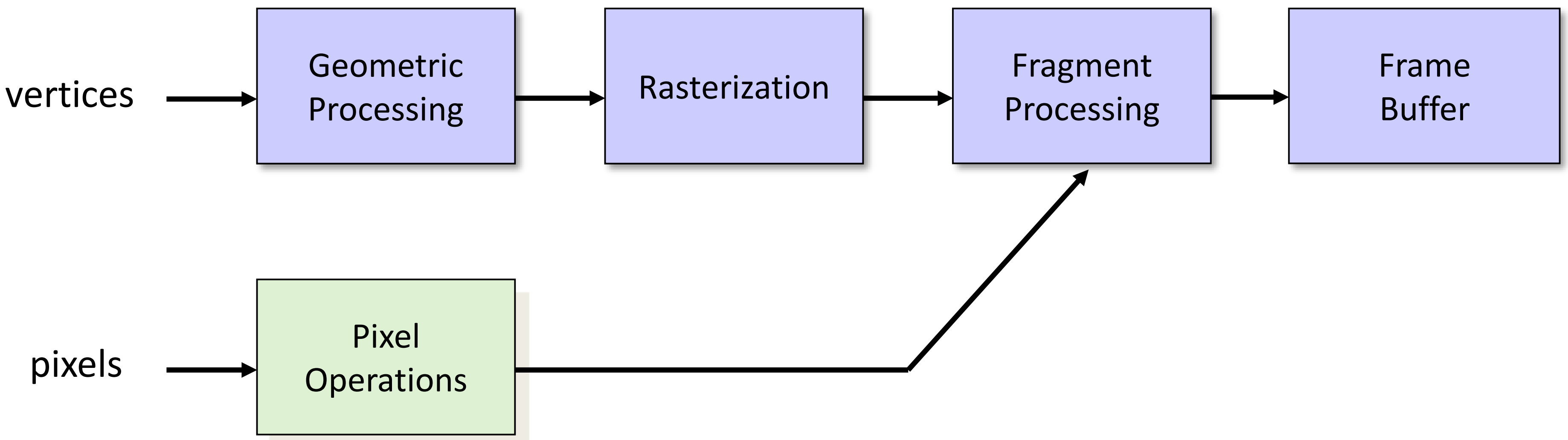
Environment Mapping

Bump Mapping



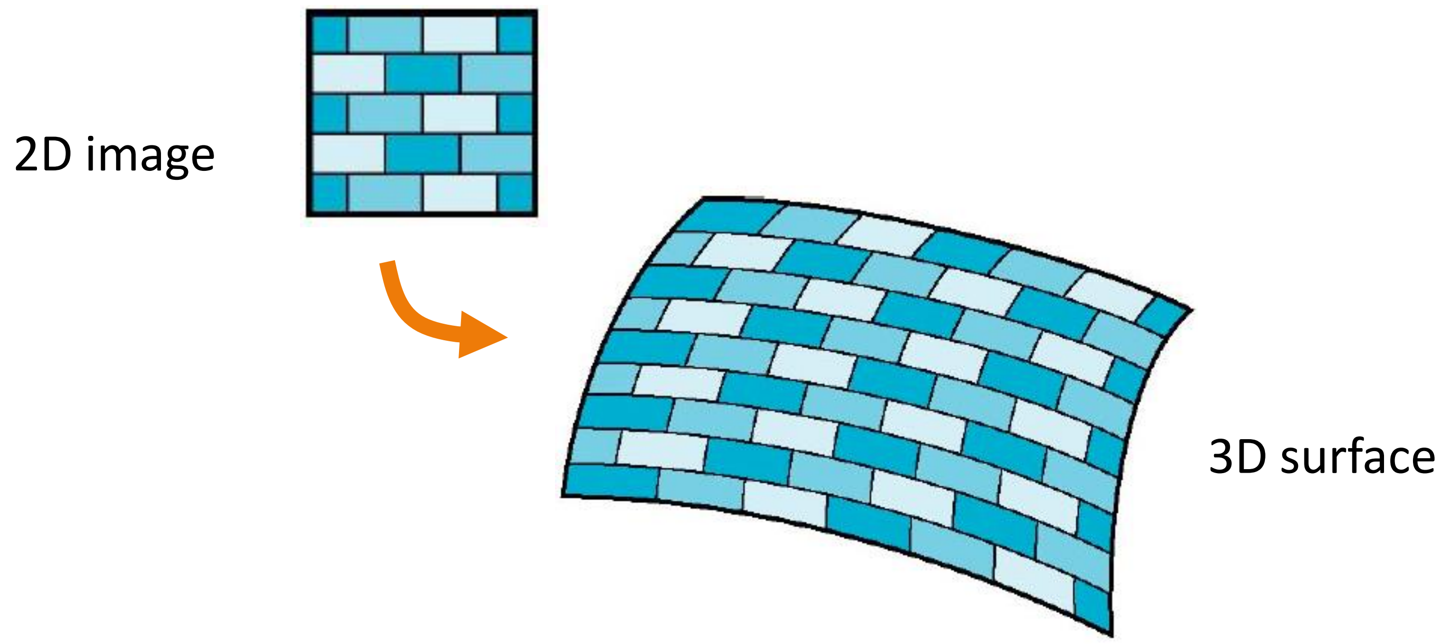
Where Does Mapping Take Place?

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons make it past the clipper



Is it Simple?

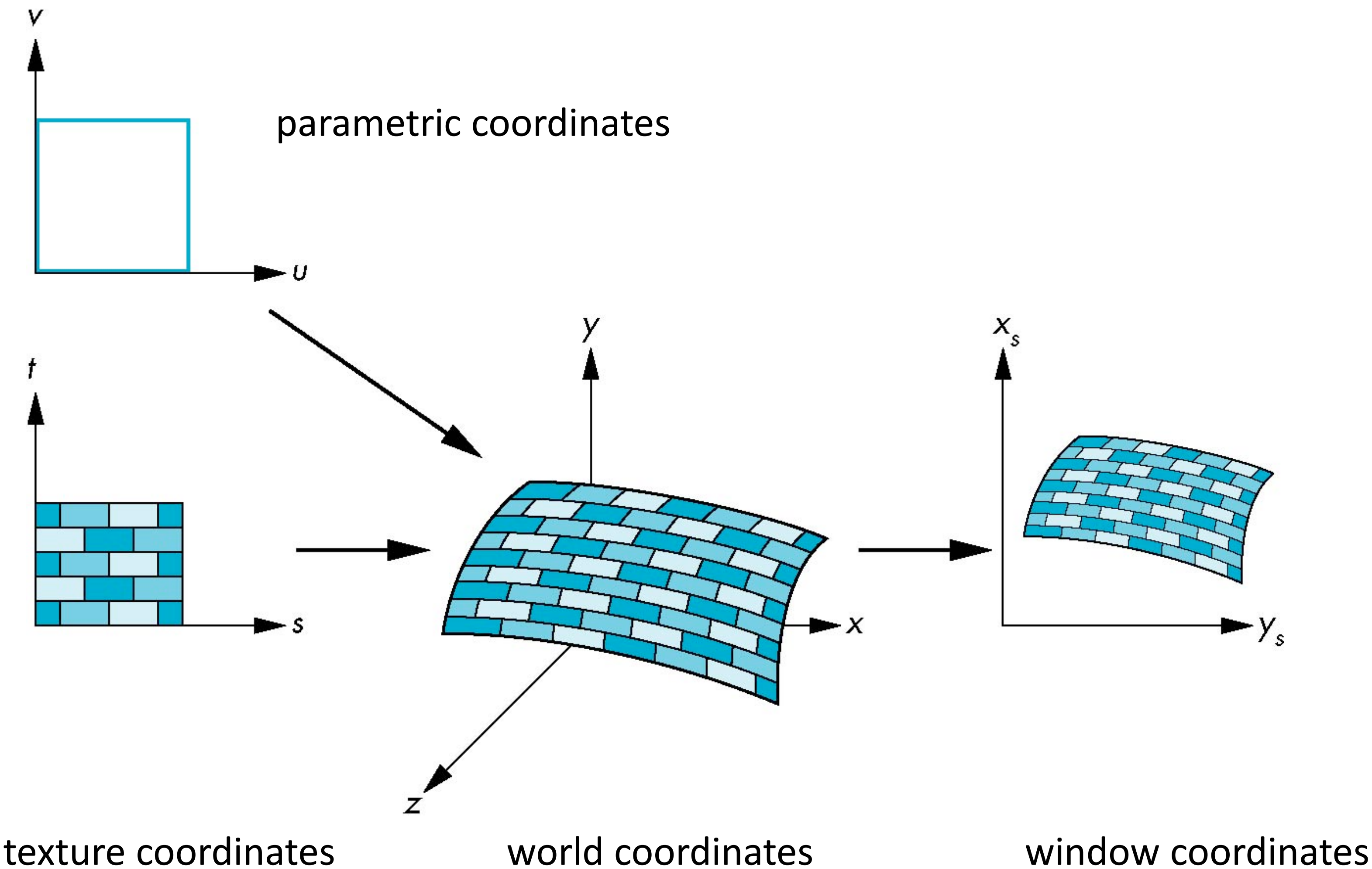
- Although the idea is simple
  - map an image to a surface---there are 3 or 4 coordinate systems involved



Coordinate Systems

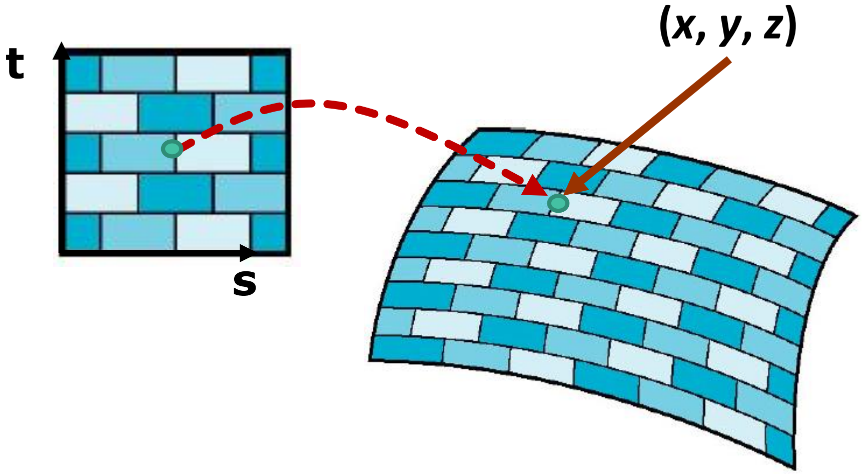
- Parametric coordinates
  - May be used to model curves and surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- Object or World Coordinates
  - Conceptually, where the mapping takes place
- Window Coordinates
  - Where the final image is really produced

Texture Mapping



Mapping Functions

- The basic problem is how to find the maps
- Consider mapping from texture coordinates to a point a surface
- Appear to need three functions
  - x = x(s,t)
  - y = y(s,t)
  - z = z(s,t)



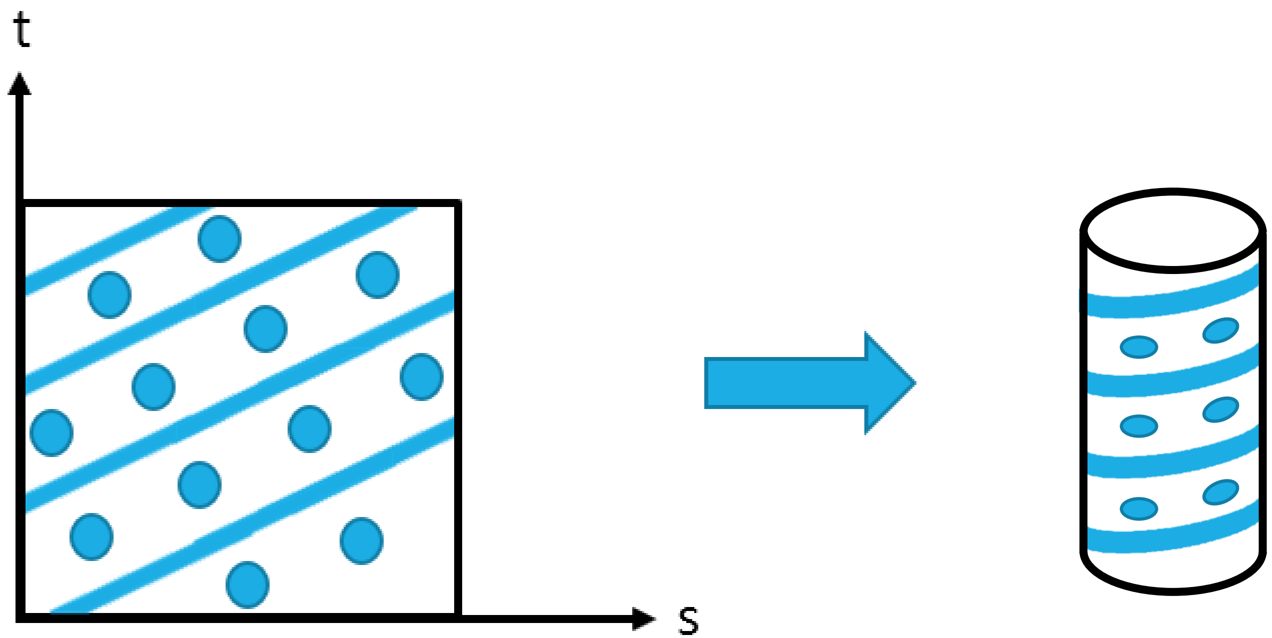
- But we really want to go the other way

Backward Mapping

- We really want
  - Given a point on an object, we want to know to which point in the texture it corresponds
- Need a backward map of the form
  - $s = \mathbf{s}(x, y, z)$
  - $t = \mathbf{t}(x, y, z)$
- Such functions are difficult to find in general

Two-part Mapping

- One solution to the mapping problem is to first map the texture to a simple intermediate surface
  - Example: map to cylinder



Cylindrical Mapping

- parametric cylinder
  - $x = r \cos 2\pi u$
  - $y = r \sin 2\pi u$
  - $z = v/h$

u, v: 0~1

- maps rectangle in *u*,*v* space to cylinder of radius r and height h in world coordinates
  - $s = u$
  - $t = v$
- maps from texture space

Spherical Map

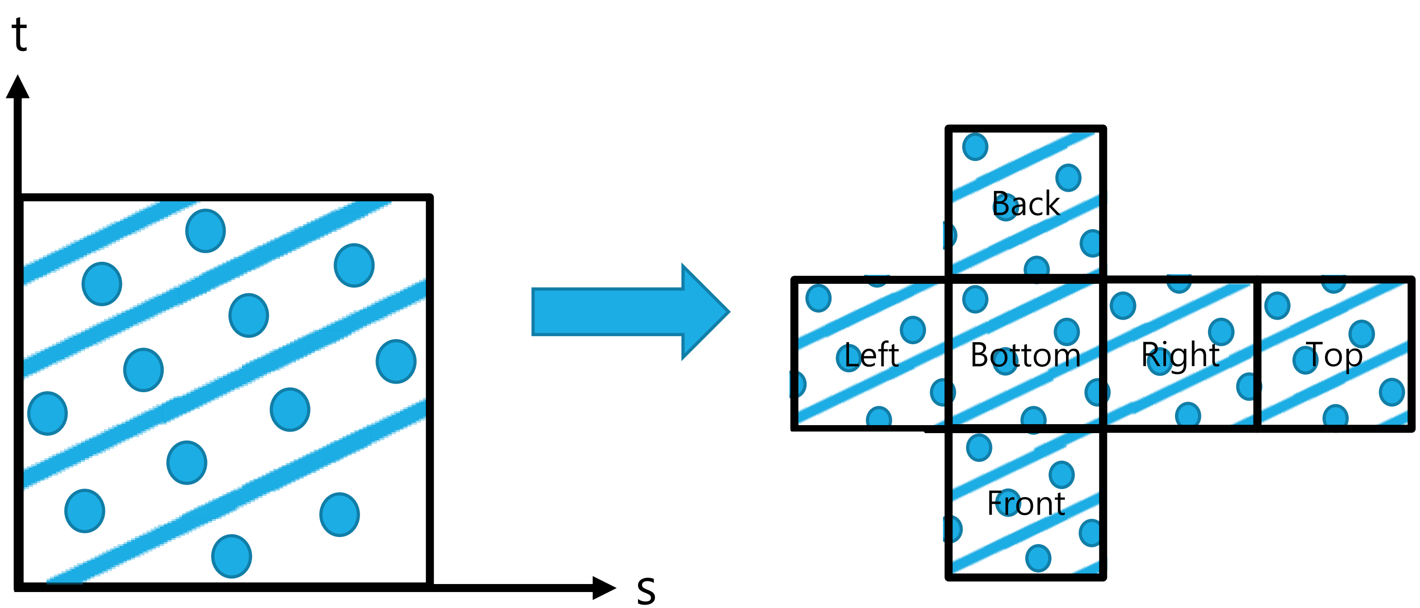
- We can use a parametric sphere
  - $x = r \cos 2\pi u$
  - $y = r \sin 2\pi u \cos 2\pi v$
  - $z = r \sin 2\pi u \sin 2\pi v$

- in a similar manner to the cylinder but have to decide where to put the distortion

- *Spheres are used in environmental maps*

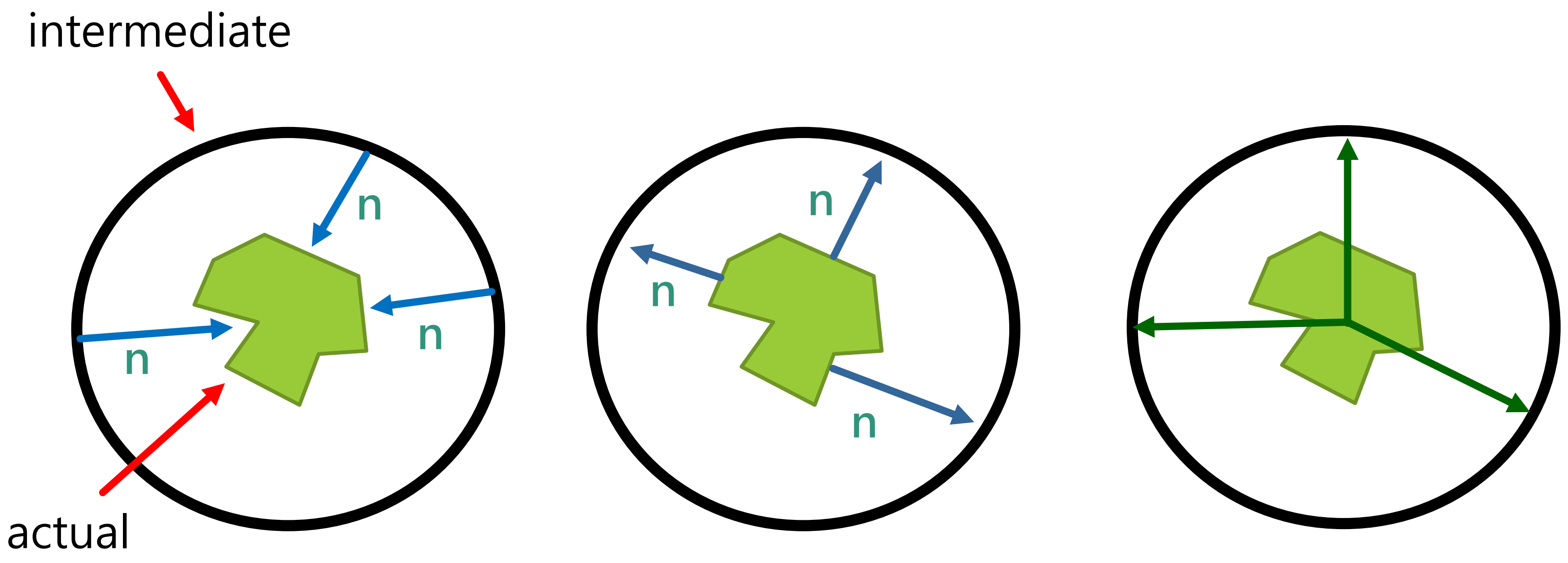
Box Mapping

- Easy to use with simple orthographic projection
- Also used in environment maps (Cube mapping)



Second Mapping

- Map from an intermediate object to an actual object
  - Normals from the intermediate to the actual
  - Normals from the actual to the intermediate
  - Vectors from the center of the intermediate

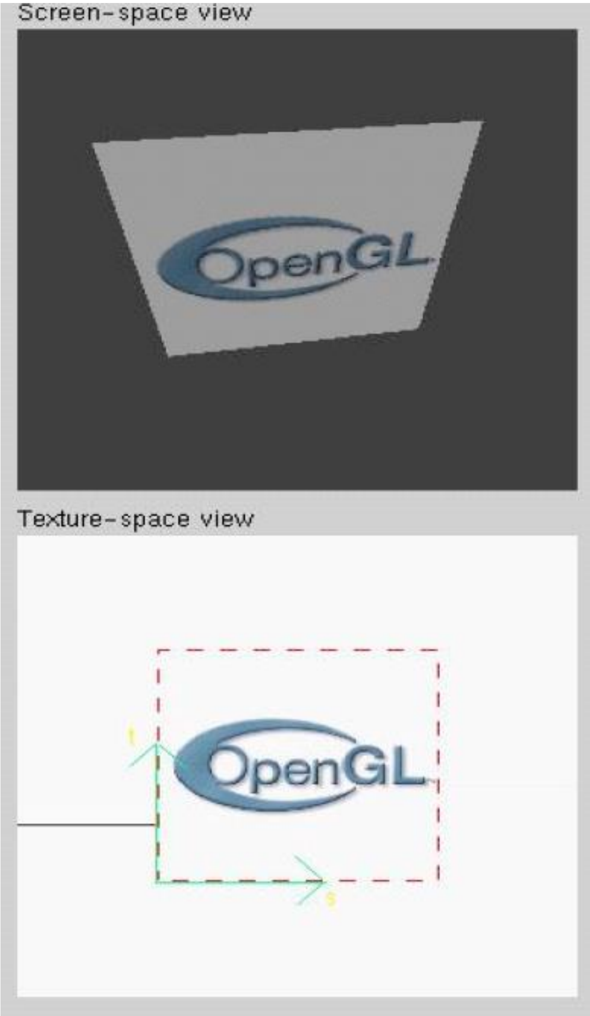


Two-part Mapping

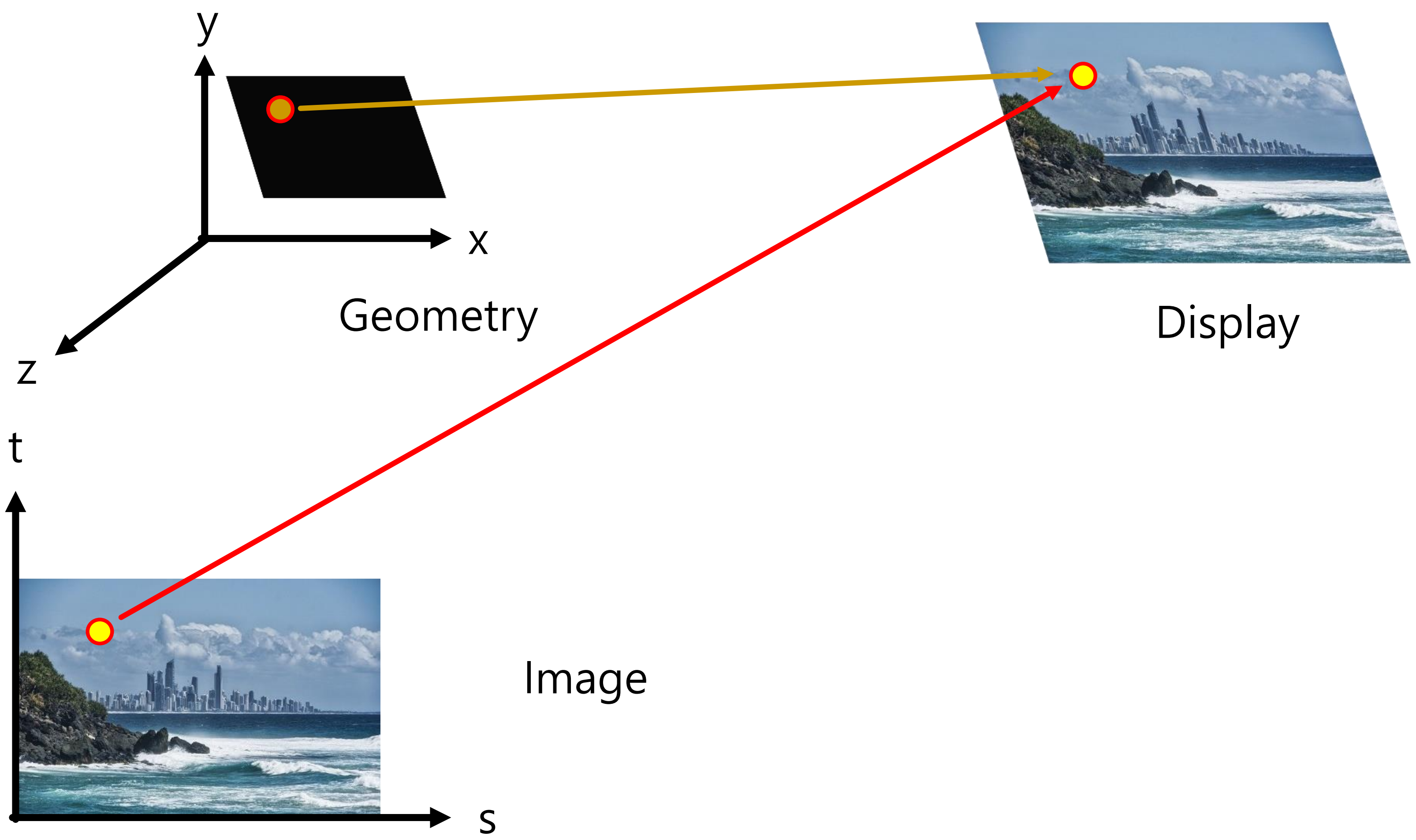
Texture Example

- The texture (below) is a 256 x 256 image, mapped to a rectangular polygon which is viewed in perspective.

- OpenGL requires texture dimensions to be powers of 2

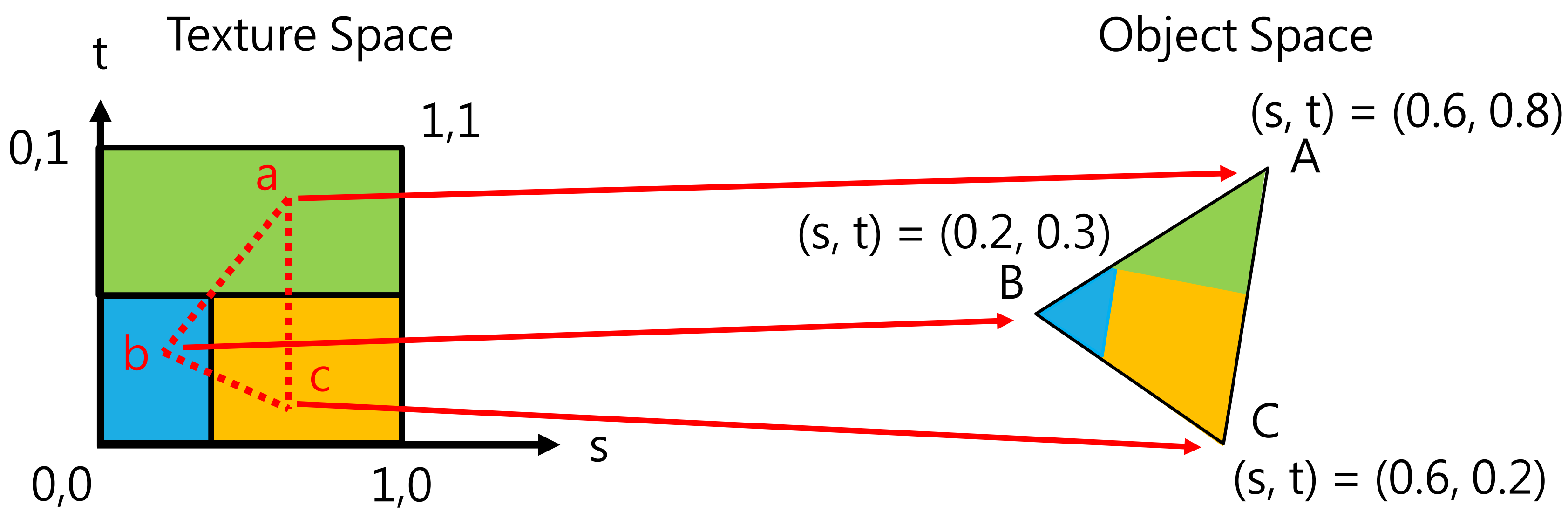


Texture Mapping



Texture Mapping for Polygons

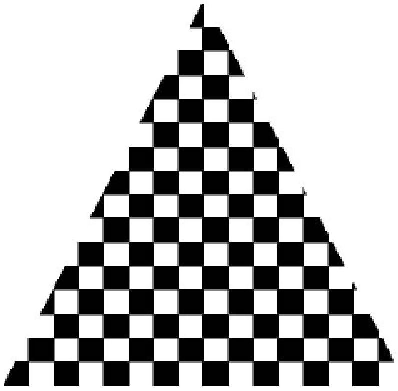
- Based on parametric texture coordinates
  - glTexCoord*() specified at each vertex



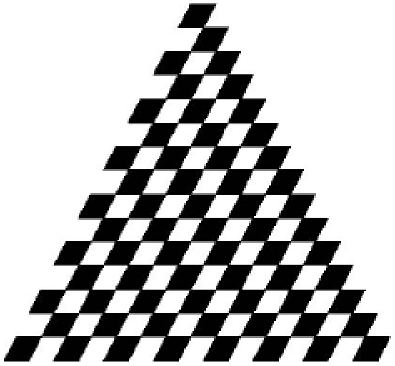
Interpolation

- OpenGL uses interpolation to find proper texels from specified texture coordinates
  - Can be distortions

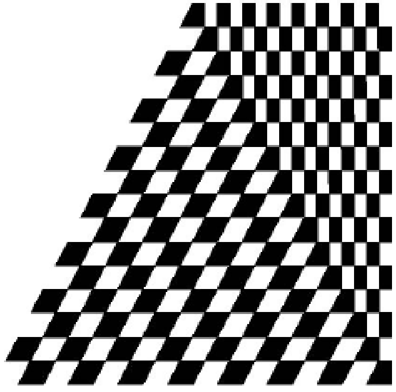
good selection
of tex coordinates



poor selection
of tex coordinates



texture stretched
over trapezoid
showing effects of
bilinear interpolation



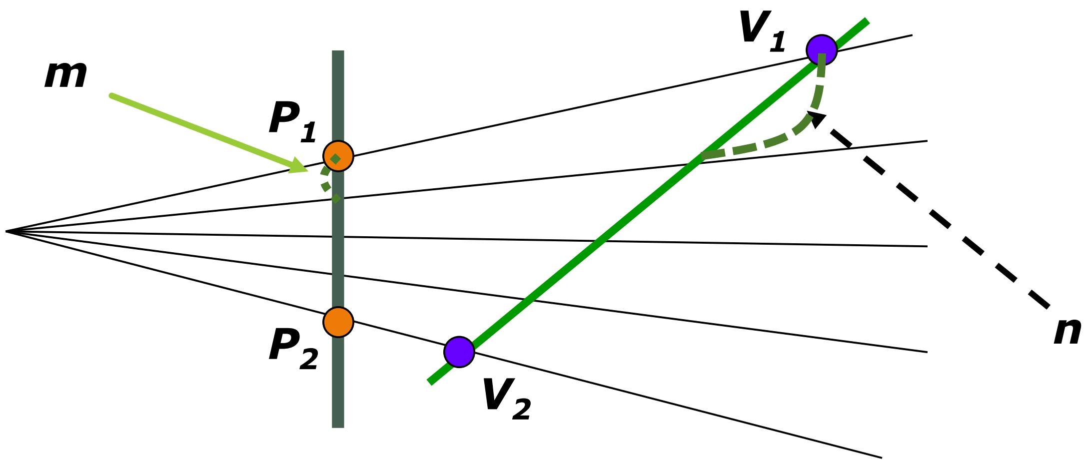
Interpolation

- Can we directly use projected x, y for texture coordinate interpolation?

Reduction of the flaws

- Subdivide the texture-mapped triangles into smaller triangles.
- Is it correct?
- How to correct this issue?

Reminder: Screen Space vs. 3D space



- Interpolation in screen space
  - $P(m) = P_1 + m(P_2 - P_1)$
- Interpolation in 3D space
  - $V(n) = V_1 + n(V_2 - V_1)$
  - $P_y(n) = V_y(n) / V_z(n)$

Reminder: Mapping from Screen Space to 3D Space

$$P_y = \frac{y_1}{z_1} + m\left(\frac{y_2}{z_2} - \frac{y_1}{z_1}\right) = \frac{y_1 + n\left(y_2 - y_1\right)}{z_1 + n\left(z_2 - z_1\right)}$$

n in terms of ***m***

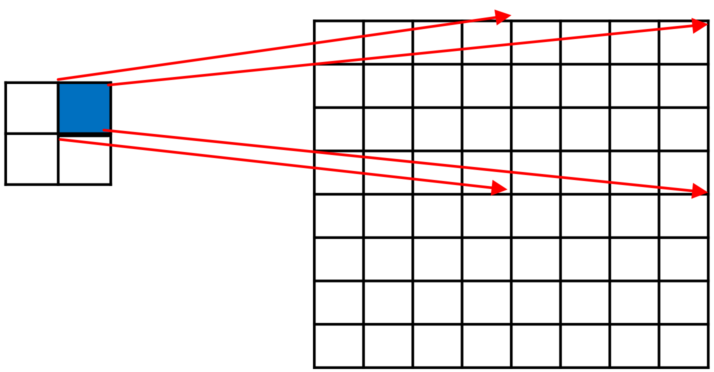
$$n = \frac{mz_1}{z_2 + m\left(z_1 - z_2\right)}$$

$$T(n) = T_1 + n(T_2 - T_1)$$

Also think about the normalized projection space….

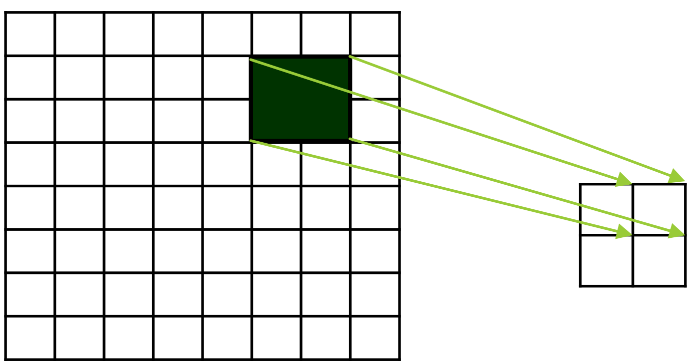
Magnification and Minification

- Minification
  - More than one texel can cover a pixel
- Magnification
  - More than one pixel can cover a texel



Texture Polygon

Magnification



Texture Polygon

Minification

point sampling (nearest texel) is the most efficient approach, but …

Aliasing

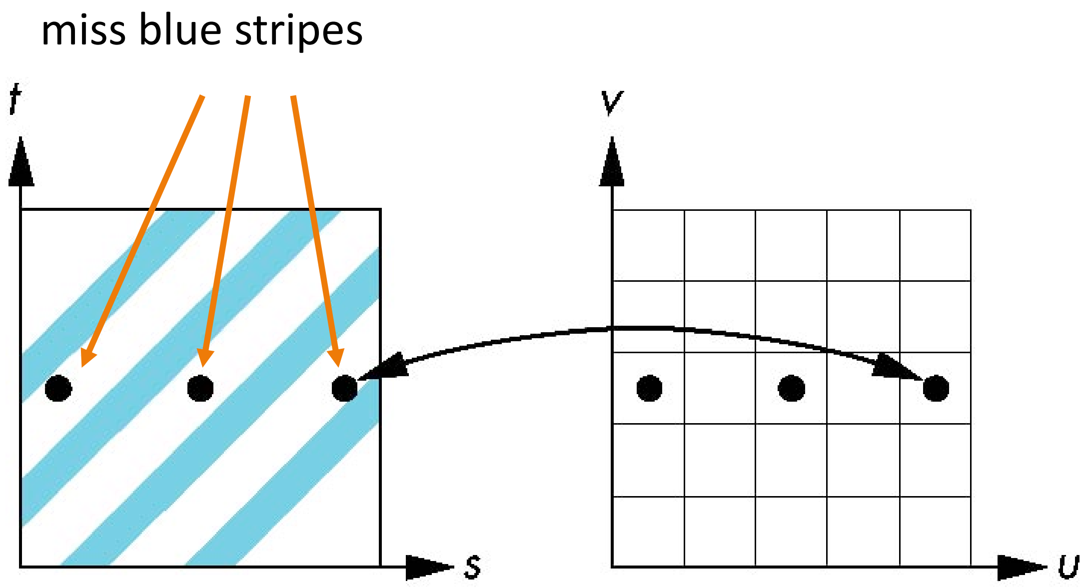
Original image

Sample one for each 5x5 pixels

Ref: www.relisoft.com/Science/Graphics/alias.html

Aliasing

- Point sampling of the texture can lead to aliasing errors

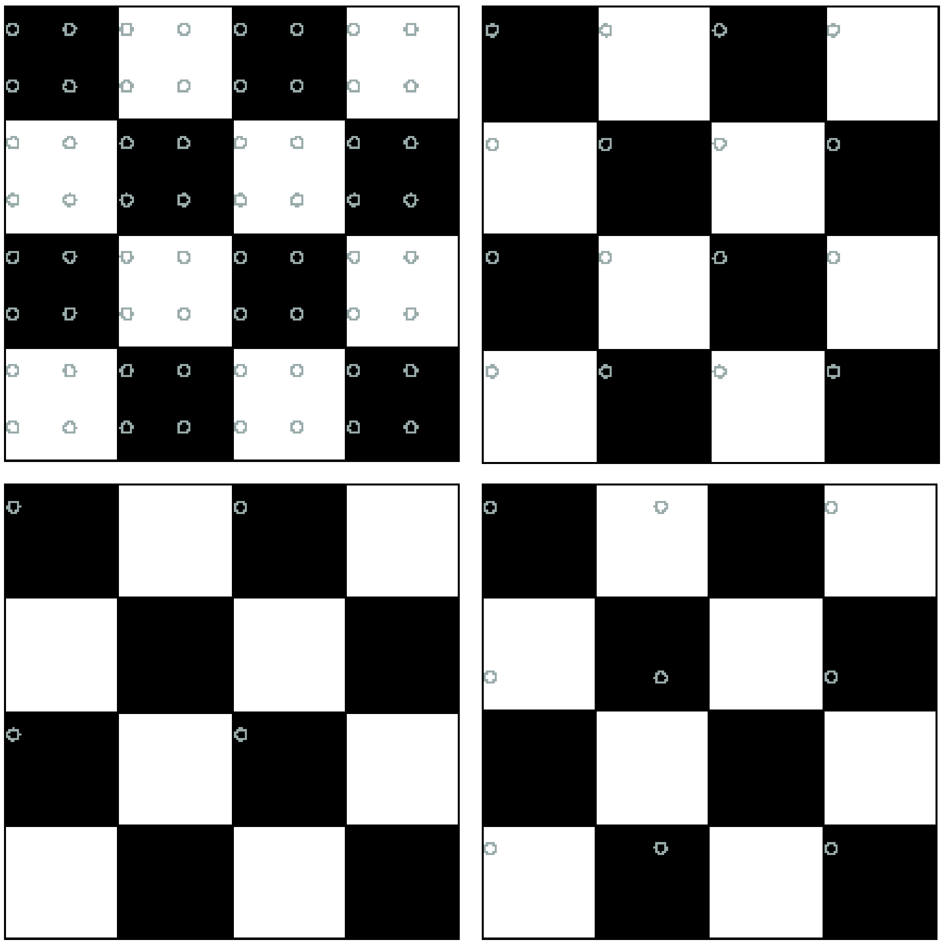


point samples in texture space

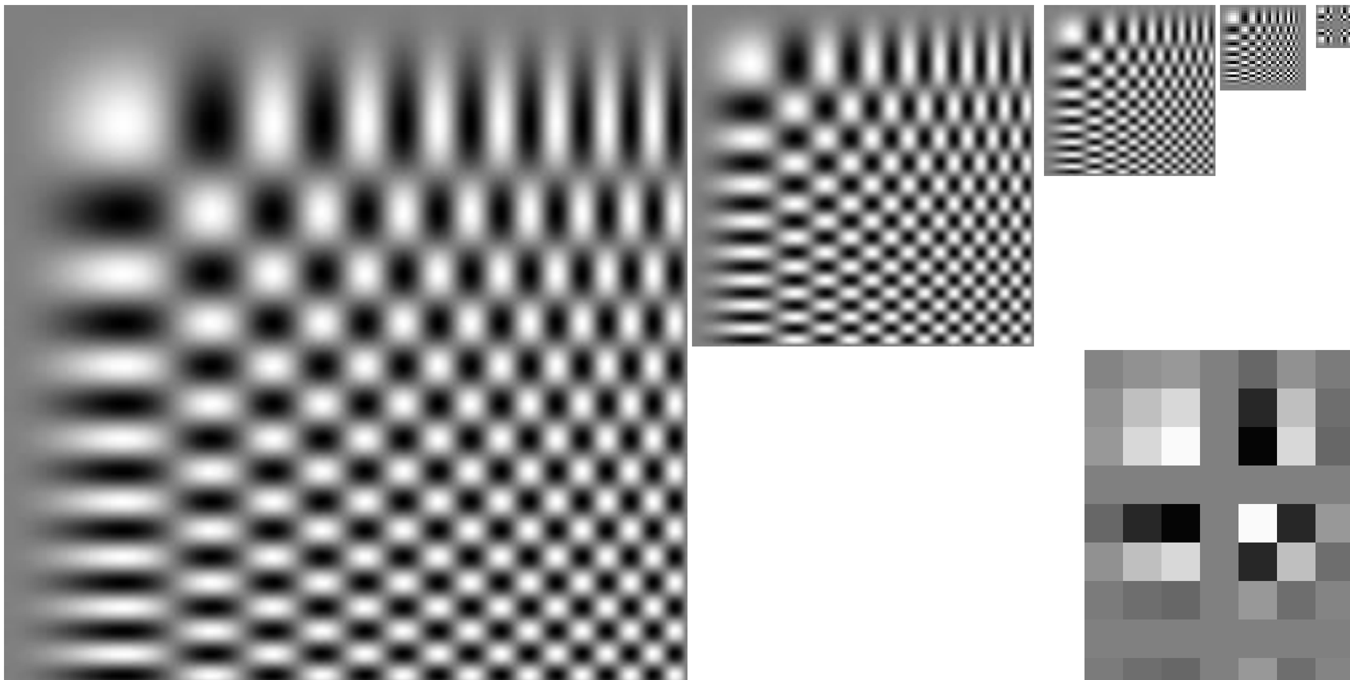
point samples in u,v (or x,y,z) space

Re-sampling

- Resample the checkerboard by taking one sample at each circle.



Simple sampling, but …



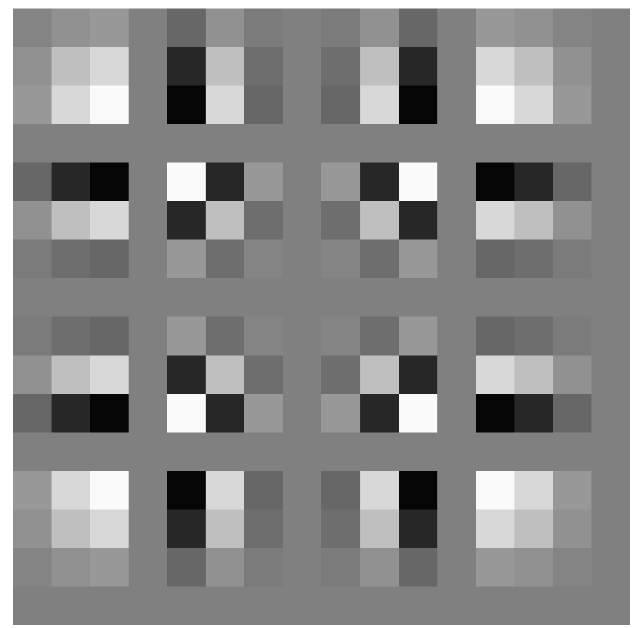
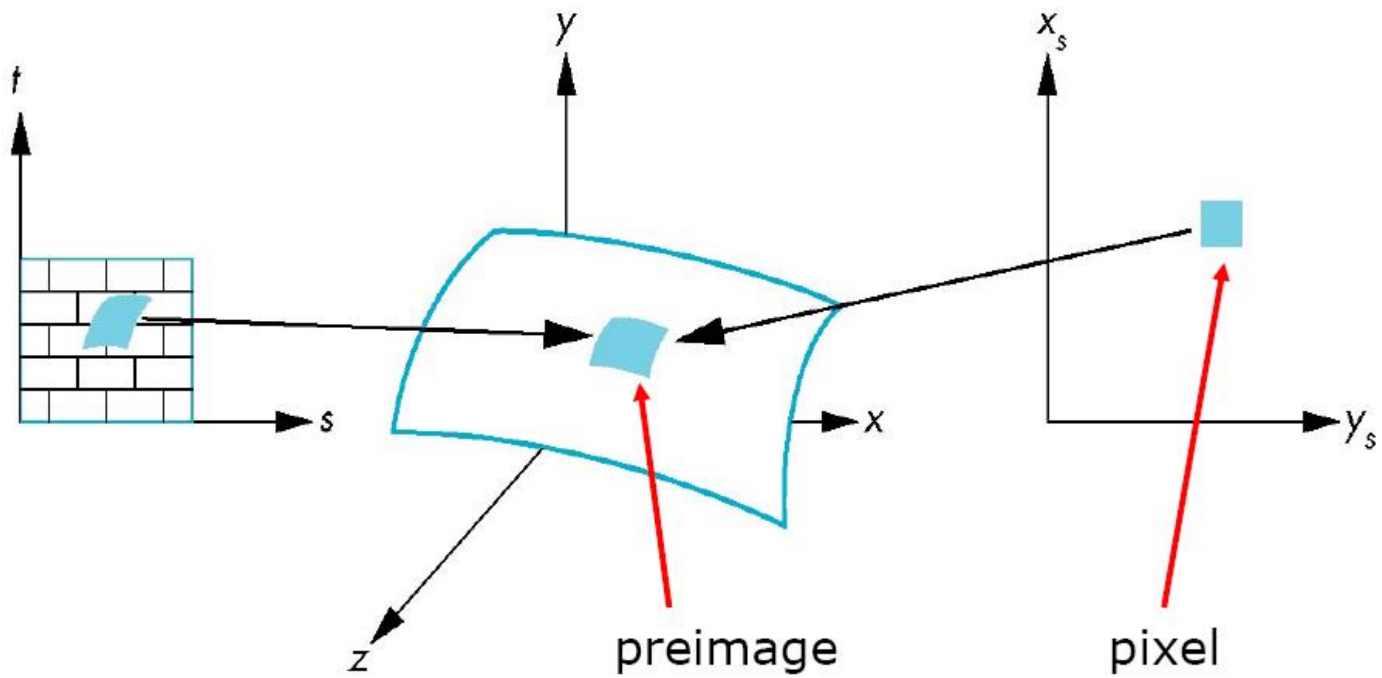


Fig. from: David A. Forsyth and Jean Ponce, Computer Vision: A Modern Approach, Prentice Hall, New Jersey, 2003.

Area Averaging

- A better but slower option is to use area averaging



Area Averaging

Original image

Sampling every 5x5 pixels

Applying a 5x5 box filter

Sampling every 5x5 pixels

www.relisoft.com/Science/Graphics/alias.html

Mipmapped Textures

- On-line processing or pre-filtering?
- Mipmapping allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects

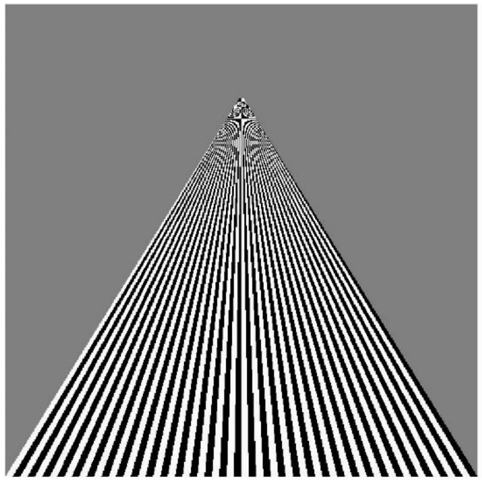
MipMap

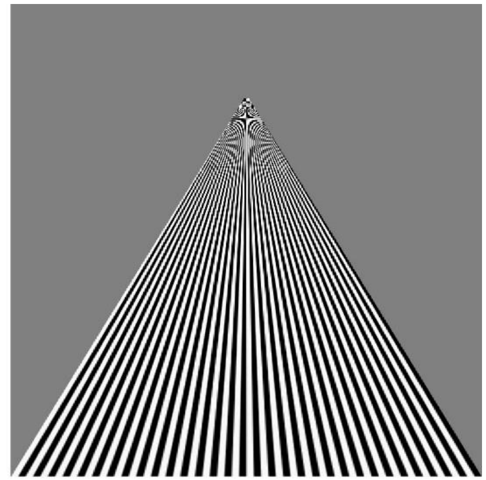
Mipmapping

- 1/3 overhead of maintaining the MIP map.

Example

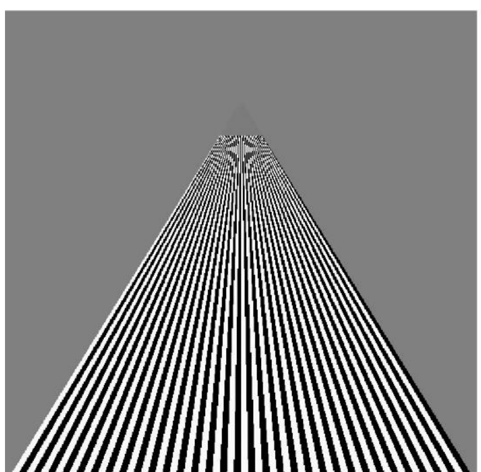
point sampling

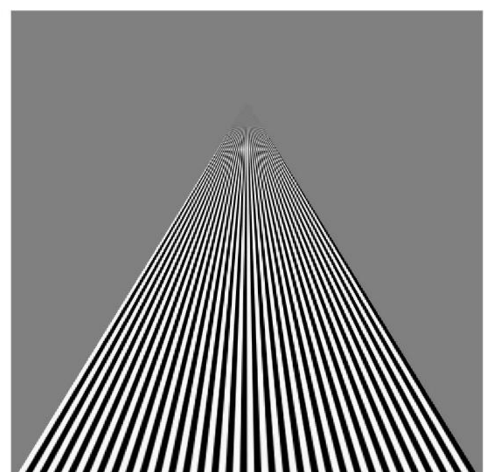




linear filtering

mipmapped point sampling





mipmapped linear filtering

Examples of Highly Reflected Models

T1000 from movie “Terminator 2”

Silver Surfer from movie “Fantastic 4:Rise of the Silver Surfer”

How to Handle Highly Specular Surfaces?

- How to render a flat mirror?
- How to render a mirror-like object in a virtual scene?
- How about rendering such an object in a real scene?

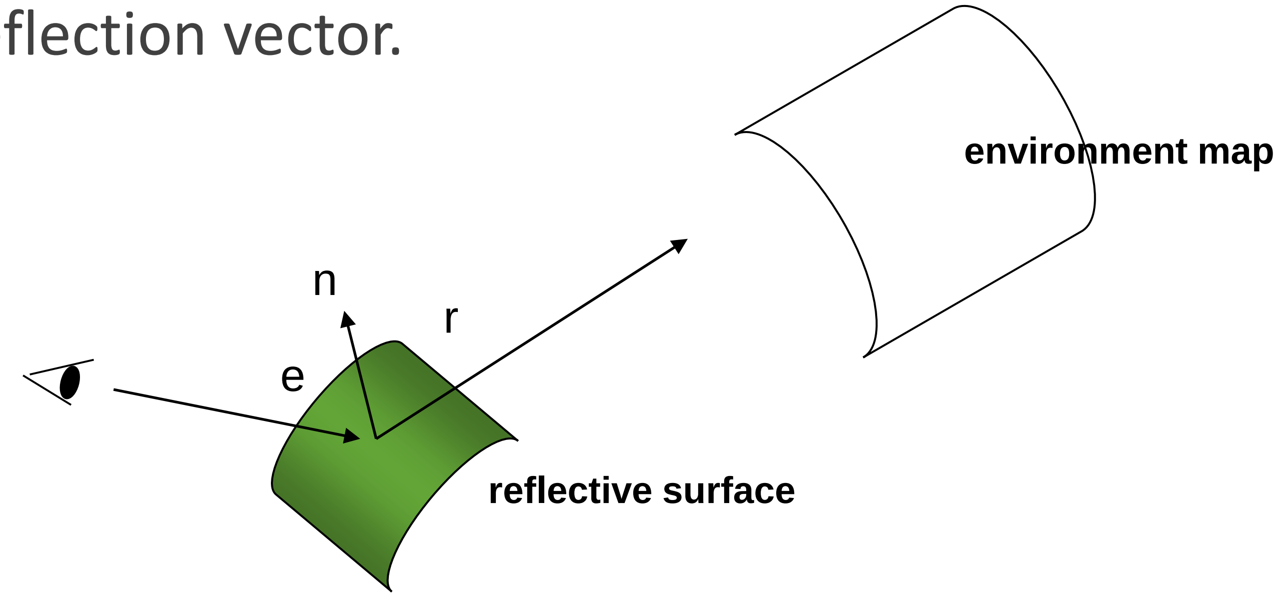
Environment Mapping

- For real-time applications
- A.k.a reflection mapping
- First proposed by Blinn and Newell.
- A efficient way to create reflections on curved surfaces
  - can be implemented using texture mapping supported by graphics hardware

Environment Mapping

- Assume the environment is far away and there's no self-reflection

- The reflection at a point can be solely decided by the reflection vector.



Environment Mapping

Sphere Mapping

- The image texture is taken from a perfectly reflective sphere.
- Assume the size of the sphere →0. Map the rays to the environment
- Using orthogonal projection.

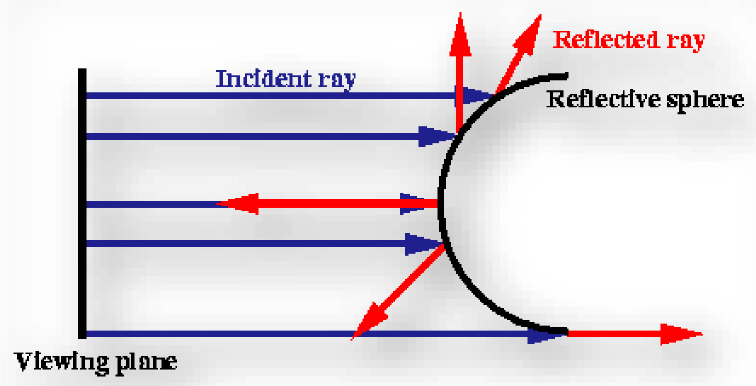
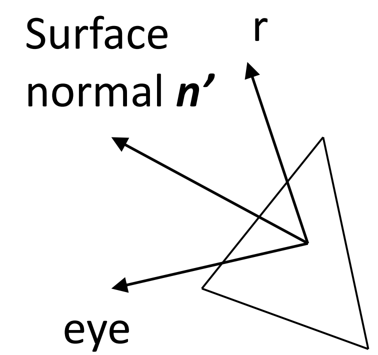


Figure 27. Creating a sphere map


Figure 31. Sphere map generated from image cube faces in Figure 30

Pictures from OpenGL tutorial. http://www.opengl.org

Sphere Mapping



- To access the sphere map texture
  - Compute the reflection vector *r* on the object surface by *e* and *n'*.

    $(r = (r_x, r_y, r_z) = -e' + 2(n' \cdot e')n')$

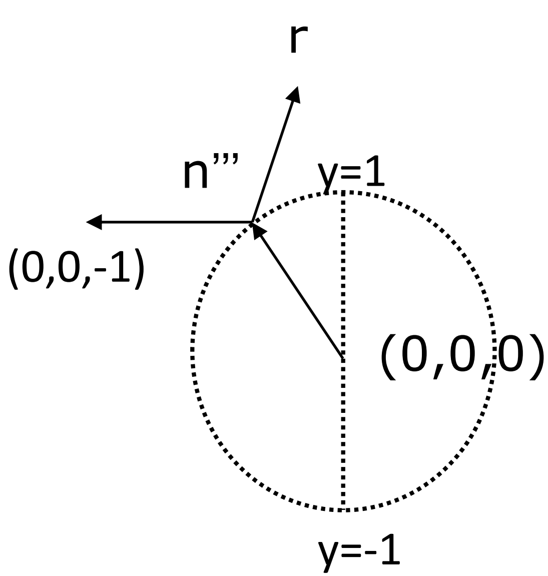
  - Access the texture: compute the sphere normal in the local space
    $n'' = (r_x, r_y, r_z) + (0,0,-1)$

$$n''' = \left(\frac{r_x}{m}, \frac{r_y}{m}, \frac{r_z - 1}{m}\right) \qquad m = \sqrt{r_x^{\ 2} + r_y^{\ 2} + (r_z - 1)^2}$$

  - Normalized the screen space from [-1,1] to [0,1]

$$s = \frac{r_x}{2m} + \frac{1}{2} \qquad t = \frac{r_y}{2m} + \frac{1}{2}$$

  - (s, t) is the target texture coordinate



Sphere Mapping

Samples from DirectX SDK

Cubemap in OpenGL

- In modern OpenGL, A special kind of texture, Cube Map, consists of six images, can be indexed by (s, t, r).

```
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
………………….
for(unsigned int i = 0; i < 6; i++) {
    glTexImage2D(
        GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
        0, GL_RGB, width, height, 0, GL_RGB,
        GL_UNSIGNED_BYTE, data[i]    ); }
```

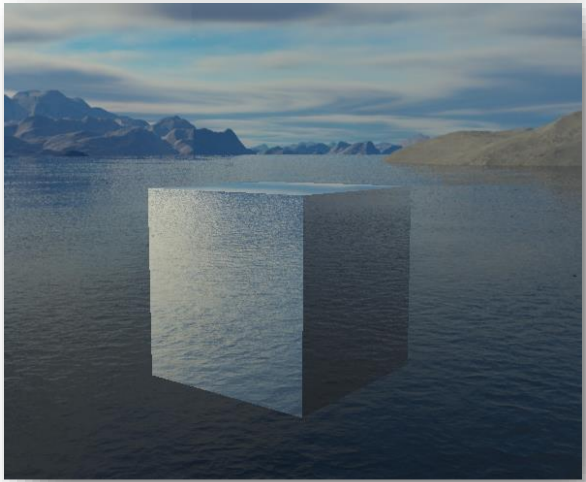
Cubemap for Environment mapping

```
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 Position;

uniform vec3 cameraPos;
uniform samplerCube skybox;

void main()
{
    vec3 I = normalize(Position - cameraPos);
    vec3 R = reflect(I, normalize(Normal));
    FragColor = vec4(texture(skybox, R).rgb, 1.0);
}
```



The example is extracted from leanopengl.com

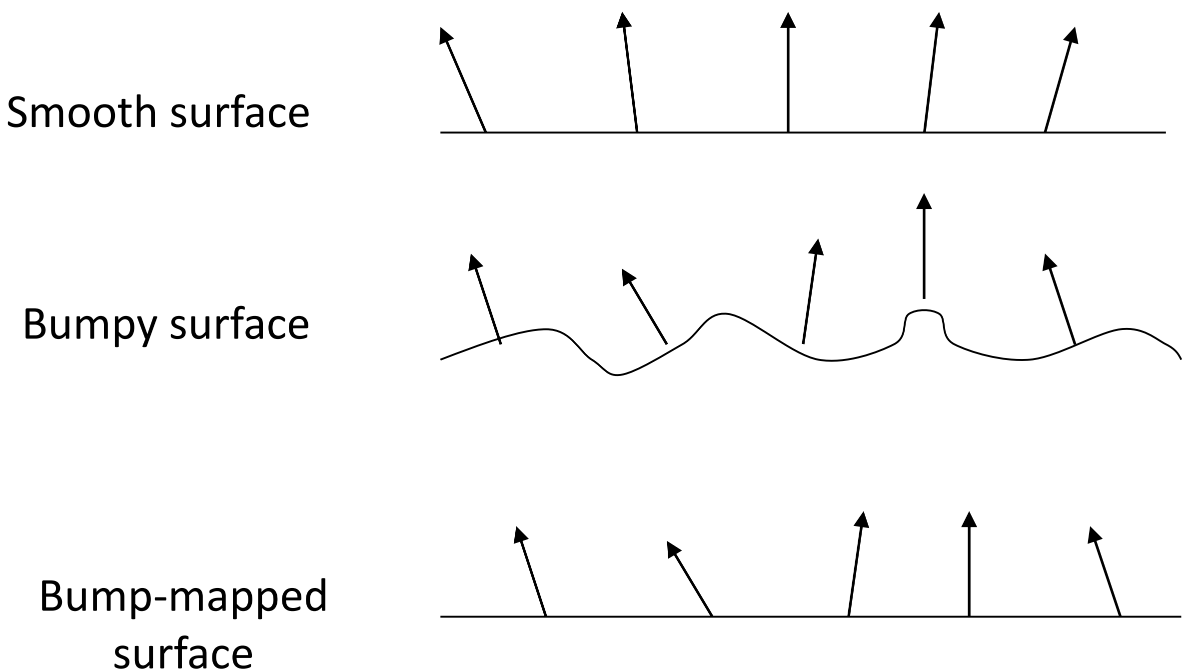
Bump and Normal Mapping

- Represent surface details and avoid heavy geometric computation.

http://www.ozone3d.net/tutorials/bump_mapping.php

Bump and Normal Mapping

- Calculate reflection (Phong Shading) with a normal map. [normal mapping]
- Or with a height map. [bump mapping]

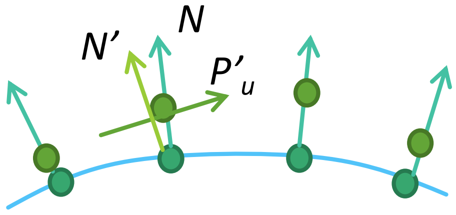


Bump Mapping

- Let $P = P(u,v)$ be a smooth parametric surface, with normals $N = N(u,v)$.
- Apply a bump map $b = b(u,v)$:

$$P' = P + bN$$

$$N' = P'_u \times P'_v$$



$$P'_u = \frac{\partial}{\partial u}(P + bN) = P_u + b_u N + bN_u \approx P_u + b_u N$$

$$P'_v = \frac{\partial}{\partial v}(P + bN) = P_v + b_v N + bN_v \approx P_v + b_v N$$

P_u – Tangent at P in u direction
P_v – Tangent at P in v direction

Bump Mapping (cont.)

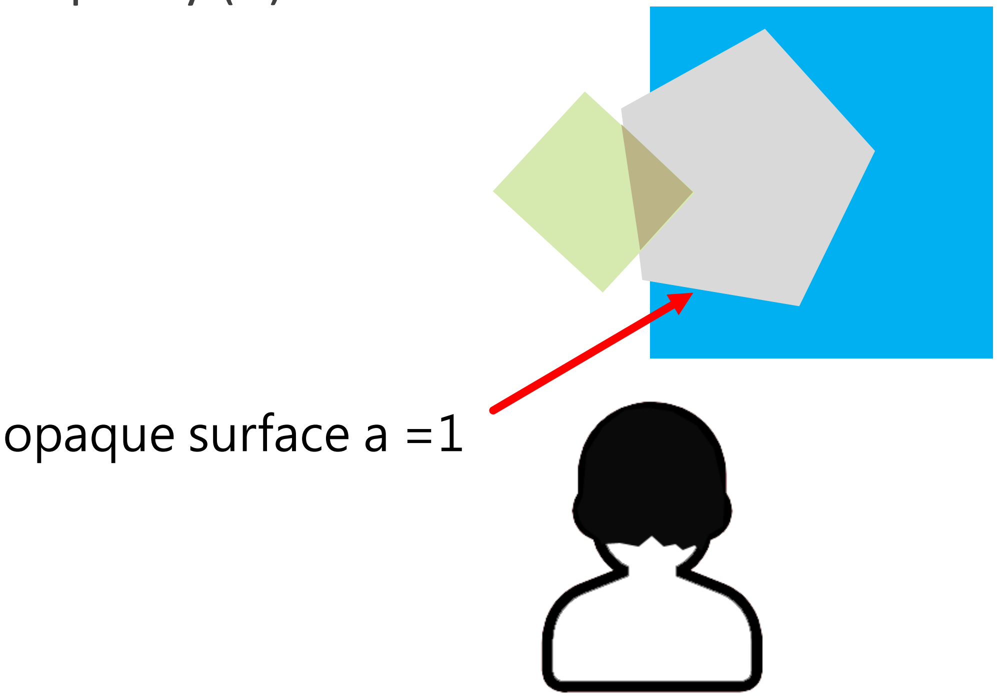
$$N' \approx (P_u + b_u N) \times (P_v + b_v N)$$
$$= P_u \times P_v + b_u (N \times P_v) + b_v (P_u \times N) + b_u b_v (N \times N)$$
$$= N + b_u (N \times P_v) + b_v (P_u \times N)$$

E.g.
When $N = (0, 0, 1)$, $N \times P_v = (-1, 0, 0)$, $P_u \times N = (0, -1, 0)$,
N' becomes $(-b_u, -b_v, 1)$

Compositing, Blending and Accumulation Buffer

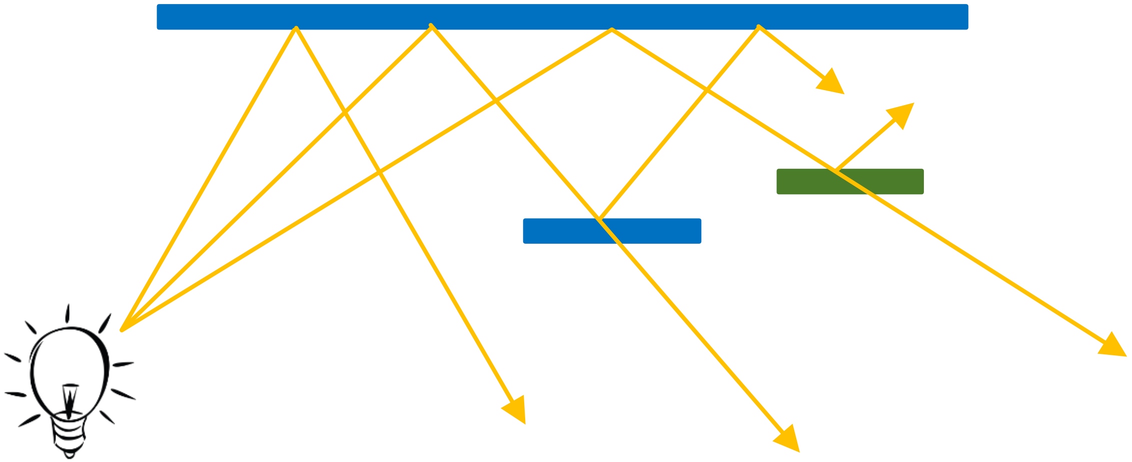
Opacity and Transparency

- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light
  - translucency = 1 – opacity (α)



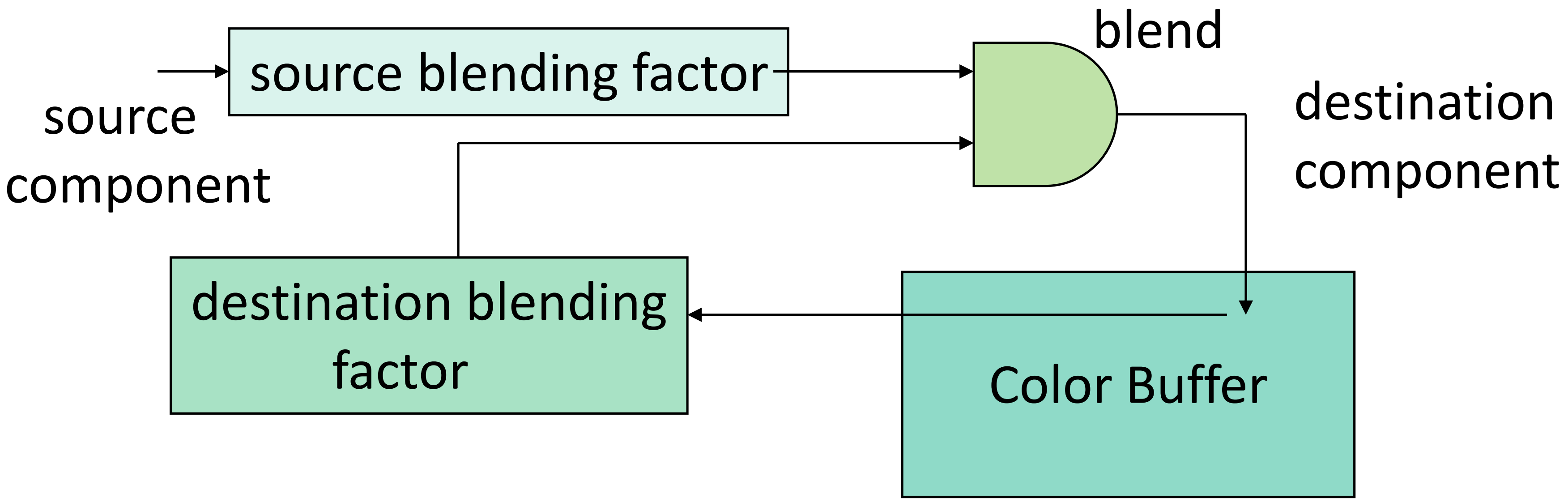
Physical Models

- Dealing with translucency in a physically correct manner is difficult due to
  - the complexity of the internal interactions of light and matter
  - Using a pipeline renderer



Writing Model

- Use A component of RGBA (or RGBα) color to store opacity
- During rendering we can expand our writing model to use RGBA values



Blending Equation

- We can define source and destination blending factors for each RGBA component
  - $s = [s_r, s_g, s_b, s_\alpha]$
  - $d = [d_r, d_g, d_b, d_\alpha]$

- Suppose that the source and destination colors are
  - $b = [b_r, b_g, b_b, b_\alpha]$
  - $c = [c_r, c_g, c_b, c_\alpha]$

- Blend as
  - $c' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g , b_b s_b + c_b d_b , b_\alpha s_\alpha + c_\alpha d_\alpha]$

Blending in practice

- glEnable(GL_BLEND);
  glBlendFunc(source_factor, destination_factor)
- Only certain factors supported: Ws + Wd = 1
  (要限制範圍，才不會爆亮=>數值超過1)
  GL_ZERO, GL_ONE, 要嘛是0要嘛是1
  GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, alpha / (1 - alpha)
  GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA (1 - beta) / beta

While we the source **α** as the source blending factor and **1– α** for the destination factor

$$(R_d', G_d', B_d', \alpha_d') = (\alpha_s R_s + (1-\alpha_s)R_d,\ \alpha_s G + (1-\alpha_s)G_d,\ \alpha_s B_s + (1-\alpha_s)B_d,\ \alpha_s\alpha_d + (1-\alpha_s)\alpha_d).$$

把前景後景疊在一起

It ensures that neither colors nor opacities can saturate, but … order dependent

Alpha Blending for Particles

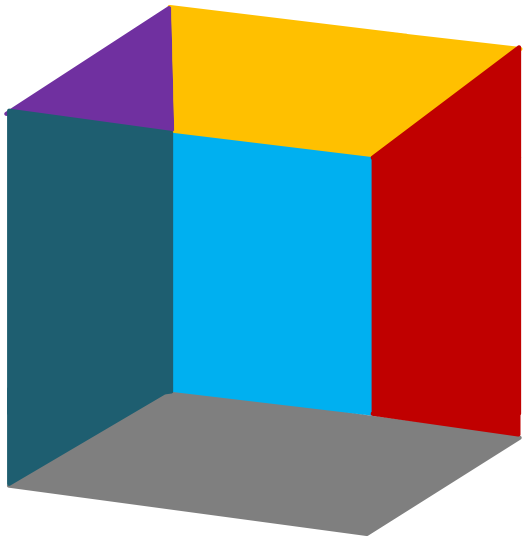
Figure from: http://www.gamedev.net/topic/592399-particle-visual-artifacts/

Order Dependency

疊顏色片的順序會影響最終呈現

用不透明片擋住透明片，透明片的顏色會完全被擋掉；若透明片在不透明片上=>blend color

- Is this image correct?
  - Probably not
  - Polygons are rendered in the order they pass down the pipeline
  - Blending functions are order dependent



Opaque and Translucent Polygons

- Suppose that we have a group of polygons some of which are opaque and some translucent
- Opaque polygons block all polygons behind them and affect the depth buffer

不透明片要去跟z buffer去看有沒有遮蔽問題，要用透明片的時候先關掉depth test、要排序

- Translucent polygons should not affect depth buffer
  - Render with **`glDepthMask(GL_FALSE)`** which makes depth buffer read-only
- Sort polygons first to remove order dependency

Fog

- We can composite with a fixed color and have the blending factors depend on depth
  - Simulates a fog effect
  - Blend source color Cs and fog color C_f by
  - $C_s' = f\ C_s + (1-f)\ C_f$
- *f* is the *fog factor*
  - Exponential
  - Gaussian
  - Linear

Fog Functions

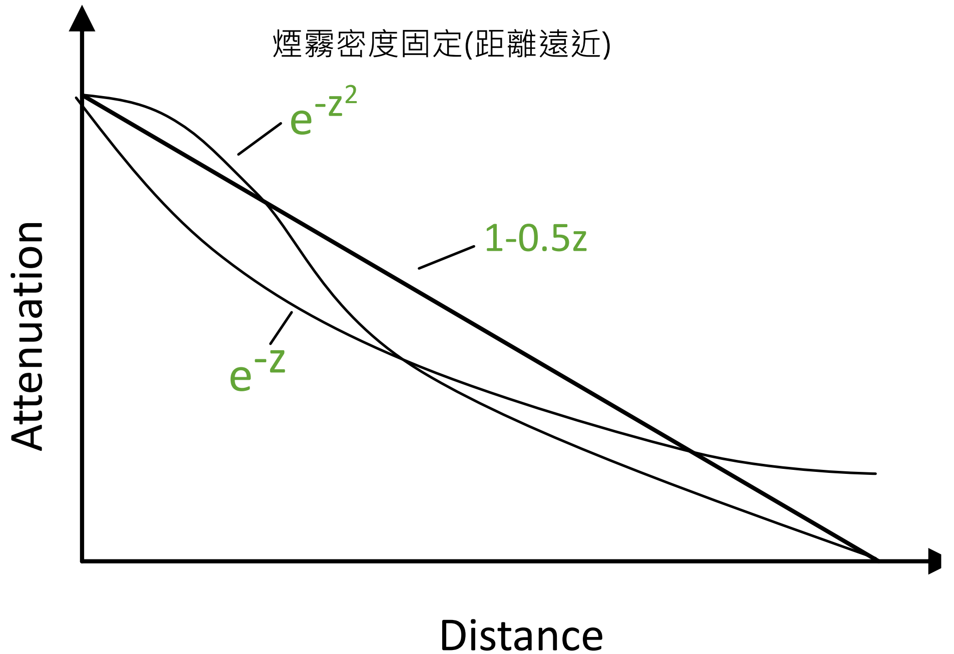


Figure from: http://www.directxtutorial.com

Accumulation Buffer

現在沒有這東西了，因為現在的frame buffer可以開float、也可以產生texture

- Compositing and blending are limited by resolution of the frame buffer
  - Typically 8 bits per color component
- The accumulation buffer is a high resolution buffer
  - 16 or more bits per component
  - Write into it or read from it with a scale factor
- Slower than direct compositing into the frame buffer
- Now deprecated but can do techniques with floating point frame buffers

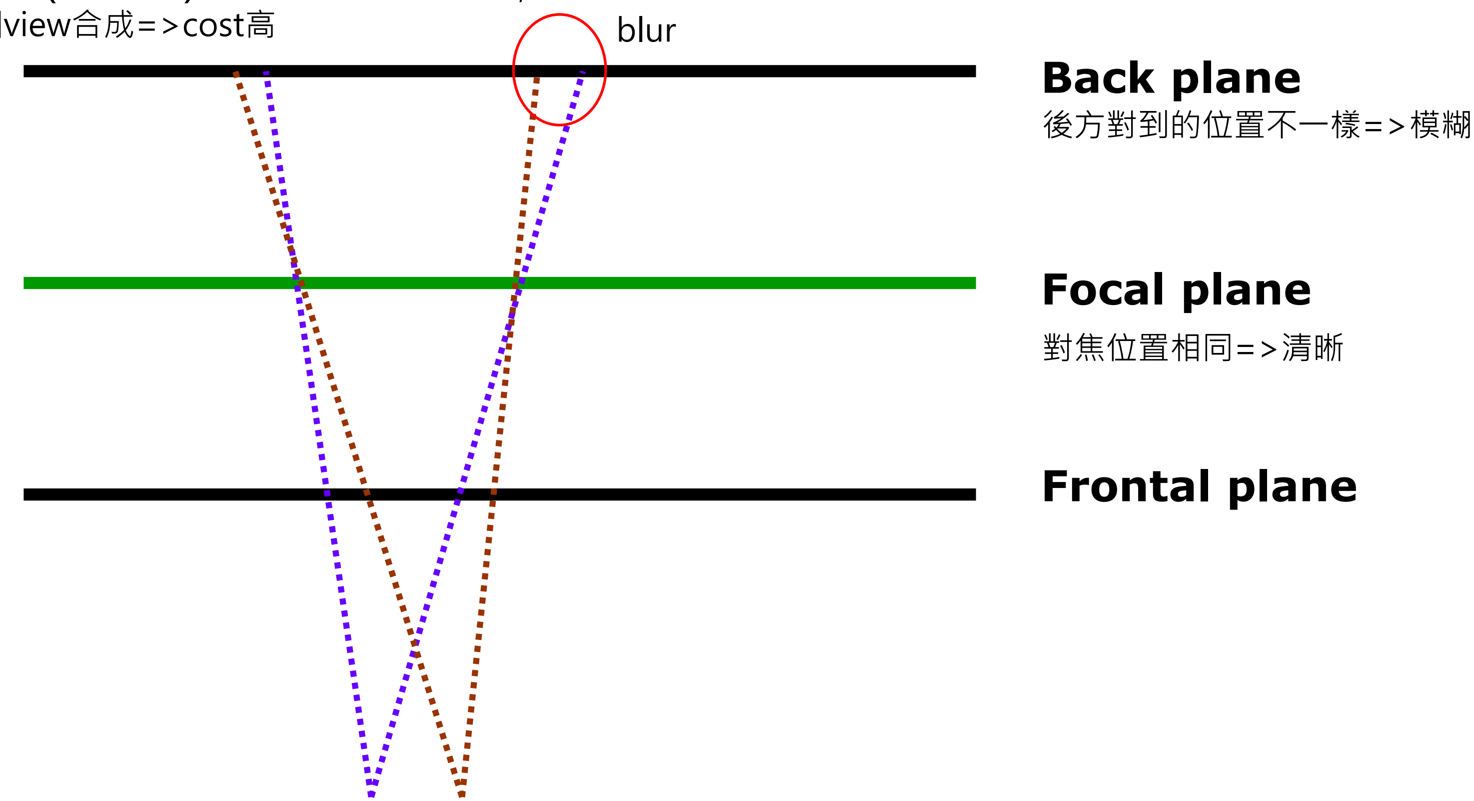
Applications

- Compositing 要用alpha浮點數、整數、字元來計算
- Image Filtering 多張圖疊在一起=>視角不同、深度值不同
- Motion effects
- Full screen antialiasing
- .....

Depth of Focus

EX : 人像對焦(前面清晰背後模糊)
調焦距=>使用透鏡時

guassian rate(高斯模糊) : 把周圍顏色疊在一起；
render多個view合成=>cost高



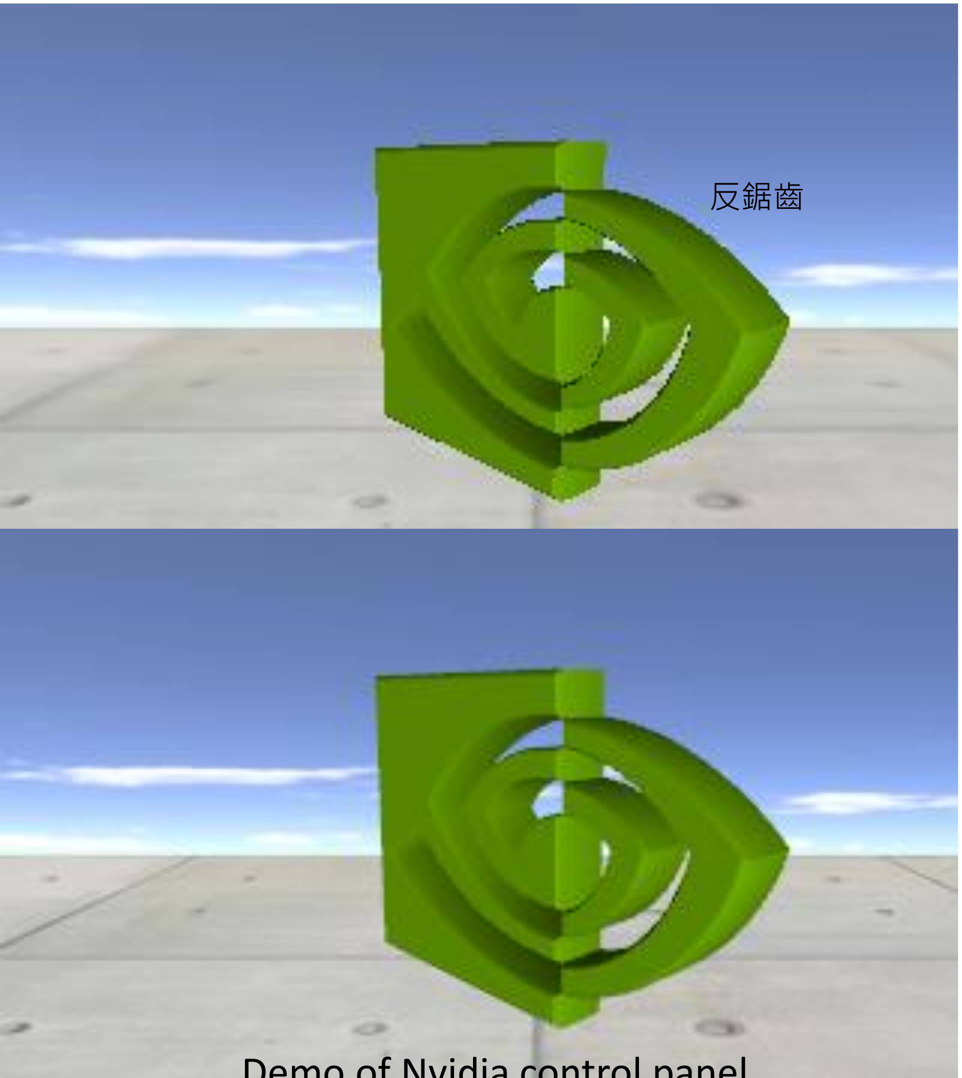
Motion blur

揮劍會有殘影、隨著時間decay、將連續兩三個frame疊在一起製作軌跡效果；
曝光大，畫面清晰；場景暗、動作快速畫面模糊=>殘影

http://www.eml.hiroshima-u.ac.jp/gallery/ComputerGraphics/motion_blur/

Anti-aliasing (Full screen and Multiple samples)

*full screen anti-aliasing(FSAA) : 把圖render大一點(拉高解析度)，cover一個位置的pixel數增加，把顏色疊在一起=>cost高



Demo of Nvidia control panel

Multisampling Anti-aliasing (MSAA)

現在比較常用MSAA

- The fragment shader still runs once per pixel for each primitive.
- MSAA then uses a larger depth/stencil buffer to determine subsample coverage. 邊緣和背景混合

sampling在圖案斜角上做，視覺效果較佳；
sample點沒通過圖案=>白色(沒有顏色)

開DLSS、FPS會上升

Deep learning super sampling (DLSS) 2.0

先render鋸齒狀的圖=>DL處理雜訊(PLS)：提高解析度 + 反鋸齒=>美美的圖(4K)

- **Convolutional autoencoder** takes the *low resolution current frame* (the aliased image and motion vectors), and the *high resolution previous frame*, to generate a higher quality current frame.

motion vector：上一張圖到下一張圖差距很大的時候，
因為深度不同，遠近動的比例不同，anti-aliased較容易使用，
motion vector可以提供顏色資訊，但不需要每個物件都train

noise=>clear：操作時會先把畫值下壓(subspace)，踢掉不必要的東西之後再放大；
圖像自由度不高、圖片有相依性

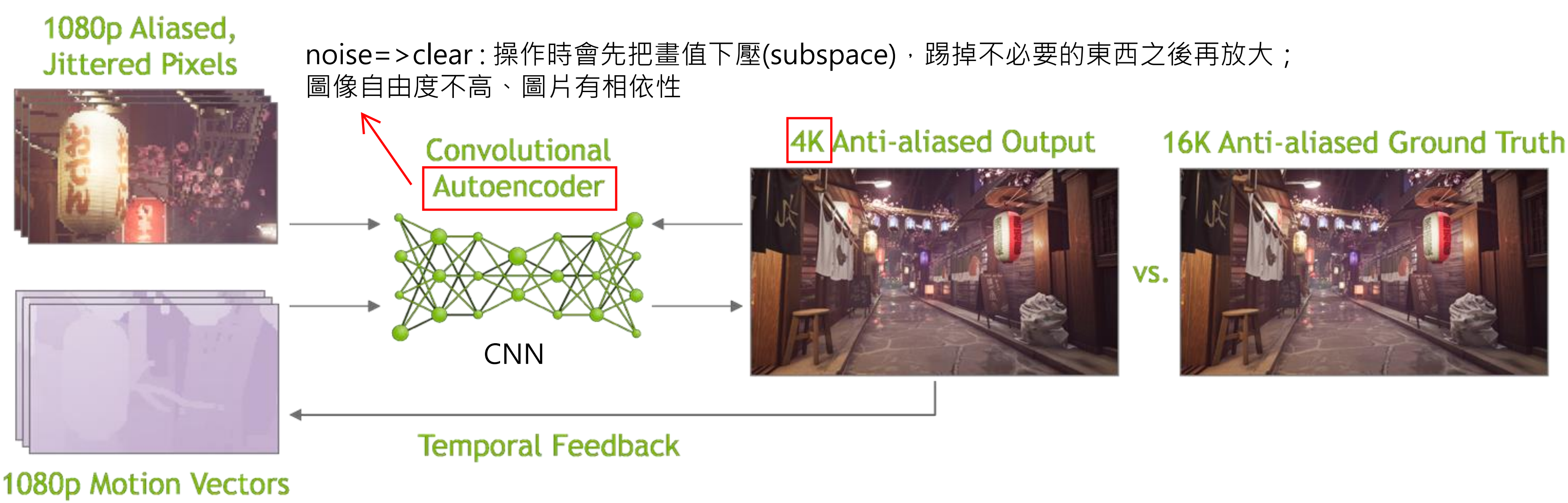


Fig. from NVIDIA DLSS 2.0

The End of Chapter 9