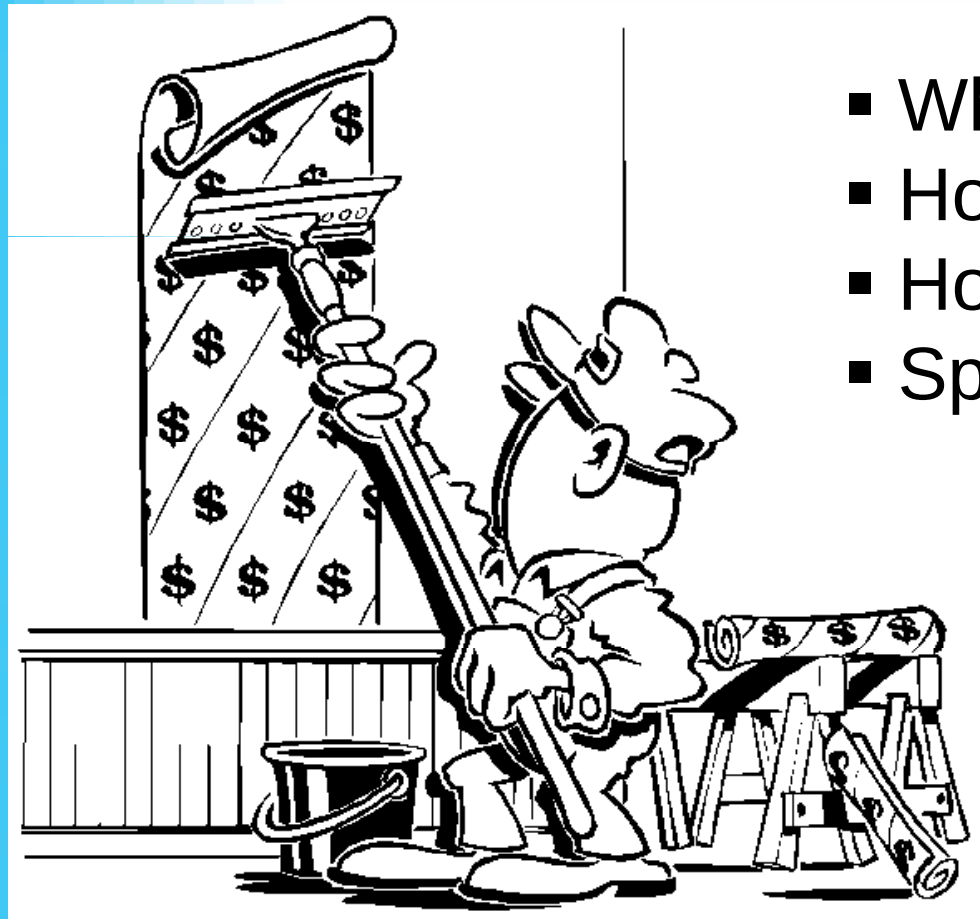
Texture Mapping – Part 1-4



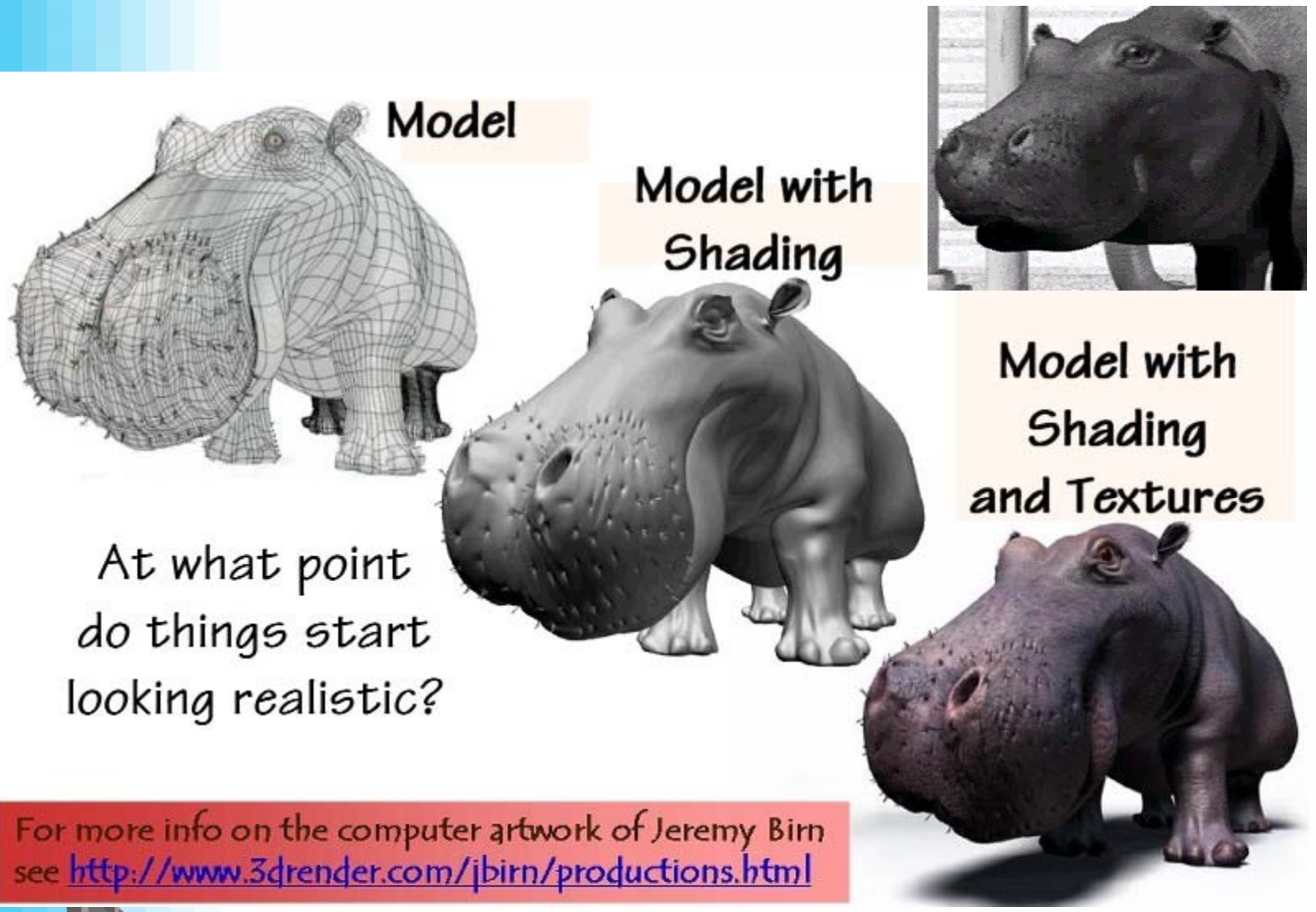
- Why Texture Map?
- How to do it
- How to do it right
- Spilling the beans

Lecture 20
CISC 440/640
Spring 2015

COMPUTER GRAPHICS

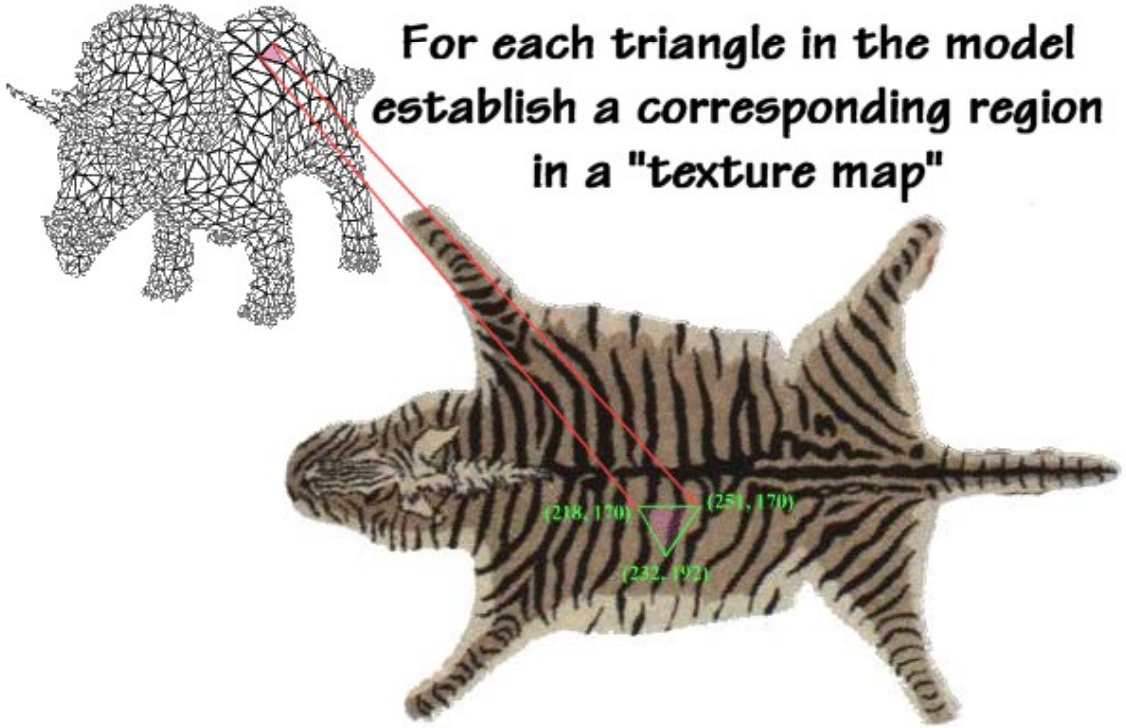
Department of Computer and Information Science

The Quest for Visual Realism



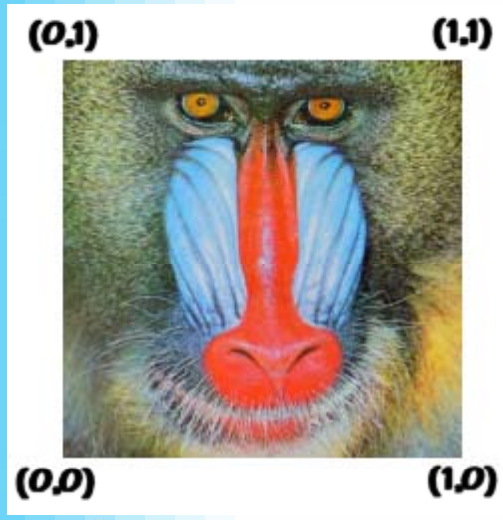
Decal Textures

The concept is very simple!



Simple OpenGL Example

- Specify a texture coordinate at each vertex *(s, t)*
- Canonical coordinates where *s* and *t* are between 0 and 1



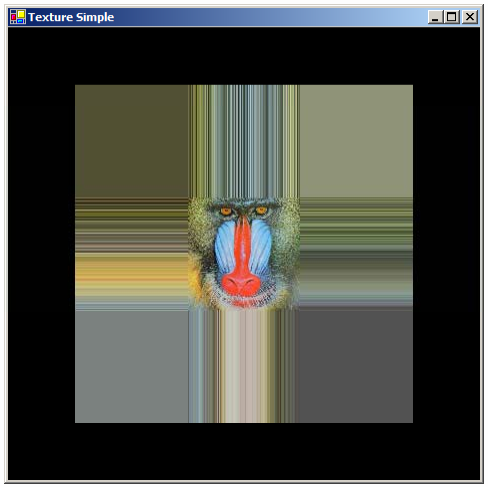

```
public override void Draw() {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFEI
        glLoadIdentity();
        glTranslated(centerx, centery, depth);
        glMultMatrixf(Rotation);
              :
        // Draw Front of the Cube
        glEnable(GL_TEXTURE_2D);
        glBegin(GL_QUADS);
                glColor3d(0.0, 0.0, 1.0);
                glTexCoord2d(0, 1);
                glVertex3d( 1.0, 1.0, 1.0);
                glTexCoord2d(1, 1);
                glVertex3d(-1.0, 1.0, 1.0);
                glTexCoord2d(1, 0);
                glVertex3d(-1.0,-1.0, 1.0);
                glTexCoord2d(0, 0);
                glVertex3d( 1.0,-1.0, 1.0);
        glEnd();
        glDisable(GL_TEXTURE_2D);
              :
        glFlush();
}
```

OpenGL Texture Peculiarities

- The width and height of Textures in OpenGL must be powers of 2
- The parameter space of each dimension of a texture ranges from [0,1) regardless of the texture's actual size.
- The behavior of texture indices outside of the range [0,1) is determined by the texture wrap options.

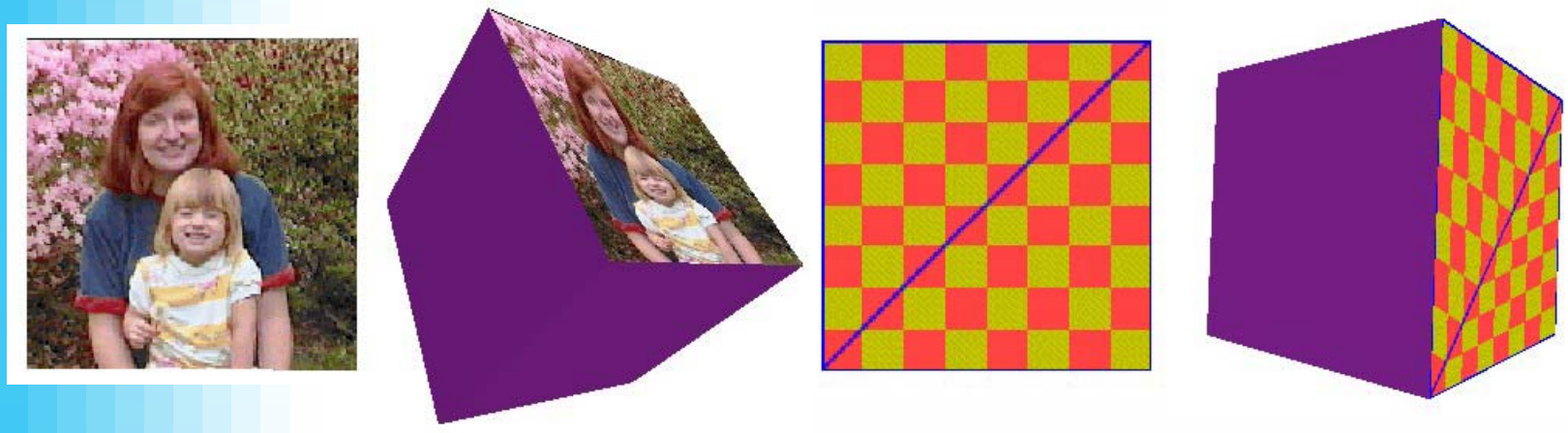
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```


```
// Draw Front of the Cube
glEnable(GL_TEXTURE_2D);
glBegin(GL_QUADS);
glColor3d(0.0, 0.0, 1.0);
glTexCoord2d(-1, 2);
glVertex3d( 1.0, 1.0, 1.0);
glTexCoord2d(2, 2);
glVertex3d(-1.0, 1.0, 1.0);
glTexCoord2d(2, -1);
glVertex3d(-1.0,-1.0, 1.0);
glTexCoord2d(-1, -1);
glVertex3d( 1.0,-1.0, 1.0);
glEnd();
glDisable(GL_TEXTURE_2D);
```




```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

Linear Interpolation of Textures



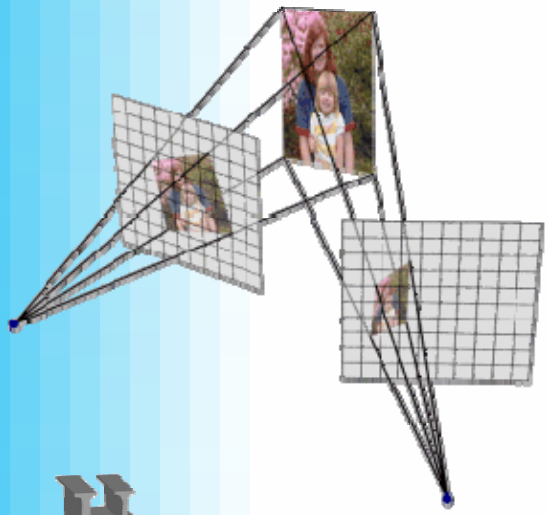
At first, you might think that we could simply apply the linear interpolation methods that we used to interpolate colors in our triangle rasterizer. However, if you implement texturing this way, you don't get the expected results.

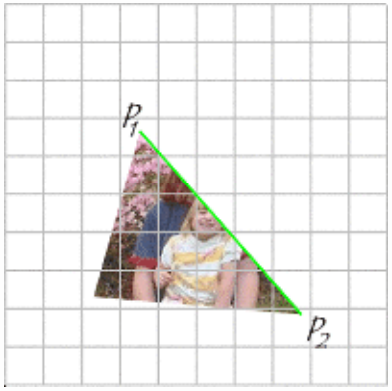
Notice how the texture seems to bend and warp along the diagonal triangle edges. Let's take a closer look at what is going on.

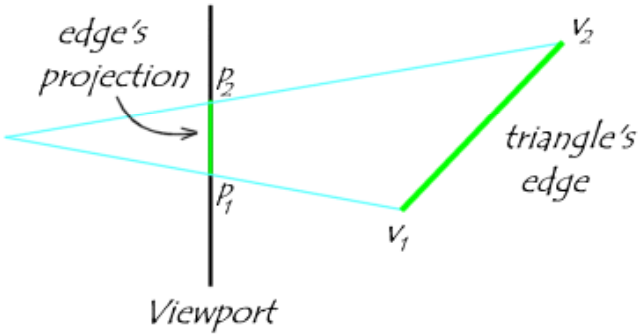
Texture Index Interpolation

Interpolating texture indices is not as simple as the linear interpolation of colors that we discussed when rasterizing triangles. Let's look at an example.

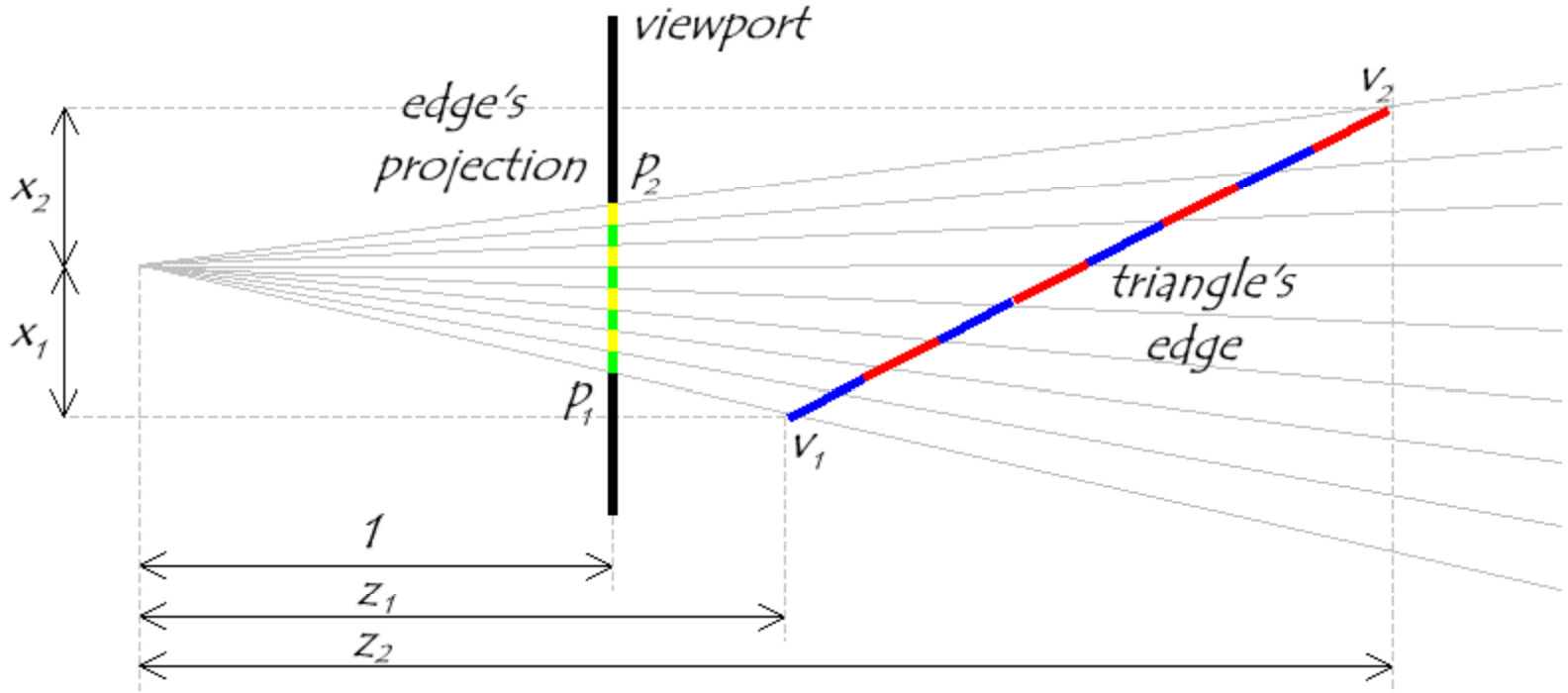
First, let's consider one edge from a given triangle. This edge and its projection onto our viewport lie in a single common plane. For the moment, let's look only at that plane, which is illustrated below:







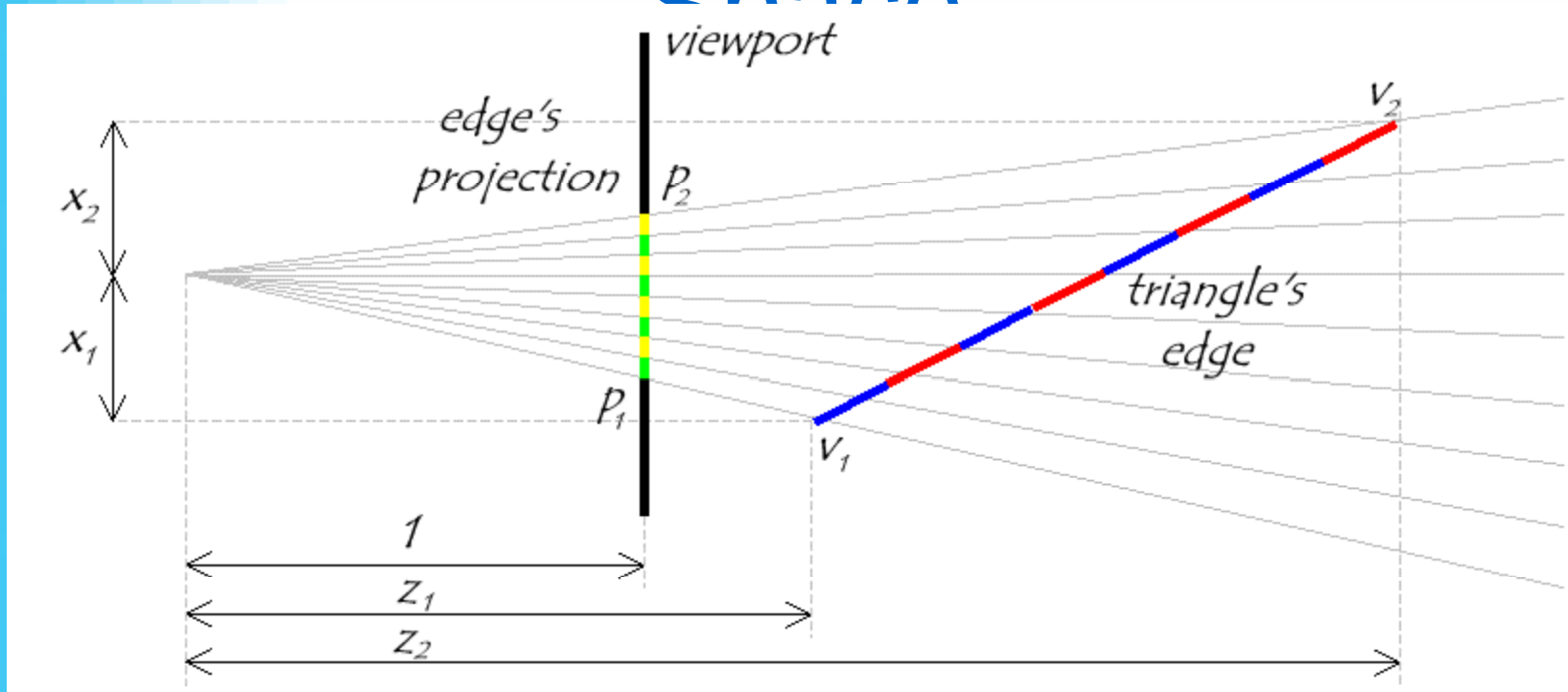
Texture Interpolation Problem



Notice that uniform steps on the image plane do not correspond to uniform steps along the edge.

Without loss of generality, let's assume that the viewport is located 1 unit away from the center of projection.

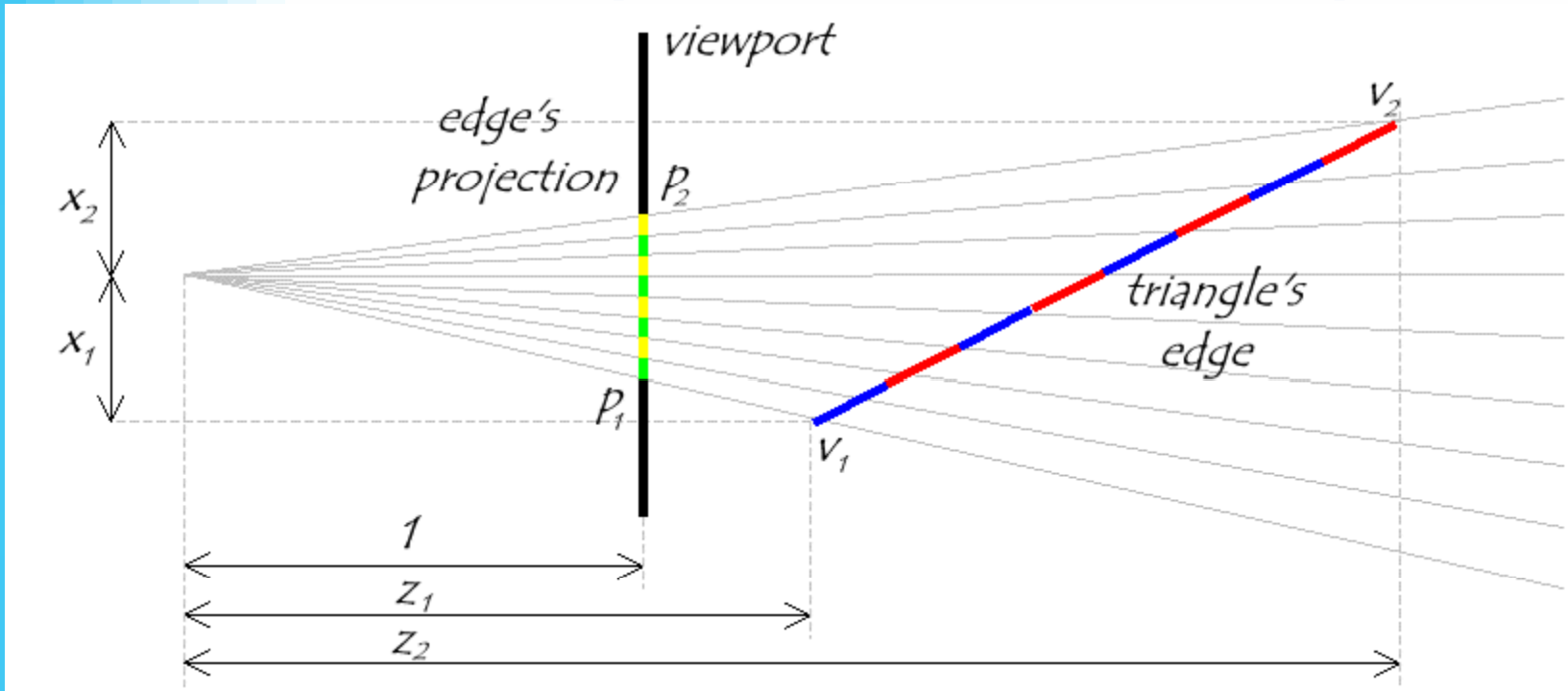
Linear Interpolation in Screen Space



Compare linear interpolation in screen space

$$p(t) = p_1 + t(p_2 - p_1) = \frac{x_1}{z_1} + t\left(\frac{x_2}{z_2} - \frac{x_1}{z_1}\right)$$

Linear Interpolation in 3-Space



to interpolation in 3-space:

$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} x_1 \\ z_1 \end{bmatrix} + s\left(\begin{bmatrix} x_2 \\ z_2 \end{bmatrix} - \begin{bmatrix} x_1 \\ z_1 \end{bmatrix}\right) \qquad P\left(\begin{bmatrix} x \\ z \end{bmatrix}\right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

How to make them Mesh

Still need to scan convert in screen space... so we need a mapping from *t* values to *s* values. We know that the all points on the 3-space edge project onto our screen-space line. Thus we can set up the following equality:

$$\frac{x_1}{z_1} + t\left(\frac{x_2}{z_2} - \frac{x_1}{z_1}\right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

and solve for *s* in terms of *t* giving:

$$s = \frac{t\, z_1}{z_2 + t\,(z_1 - z_2)}$$

Unfortunately, at this point in the pipeline (after projection) we no longer have z_1 and z_2 lingering around (Why?). However, we do have $w_1 = 1/z_1$ and $w_2 = 1/z_2$.

$$s = \frac{t\,\frac{1}{w_1}}{\frac{1}{w_2} + t\,\left(\frac{1}{w_1} - \frac{1}{w_2}\right)} = \frac{t\, w_2}{w_1 + t\,(w_2 - w_1)}$$

Interpolating Parameters

We can now use this expression for *s* to interpolate arbitrary parameters, such as texture indices (*u, v)*, over our 3-space triangle. This is accomplished by substituting our solution for *s* given *t* into the parameter interpolation.

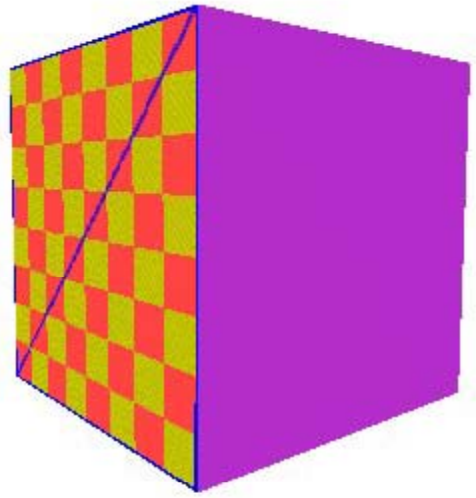
$$u = u_1 + s(u_2 - u_1)$$

$$u = u_1 + \frac{t\,w_2}{w_1 + t\,(w_2 - w_1)}(u_2 - u_1) = \frac{u_1 w_1 + t\,(u_2 w_2 - u_1 w_1)}{w_1 + t\,(w_2 - w_1)}$$

Therefore, if we **premultiply all parameters that we wish to interpolate in 3-space by their corresponding *w* value** and add a new plane equation to interpolate the *w* values themselves, we can interpolate the numerators and denominator in screen-space. We then need to perform a divide a each step to get to map the screen-space interpolants to their corresponding 3-space values. This is a simple modification to the triangle rasterizer that we developed in class.

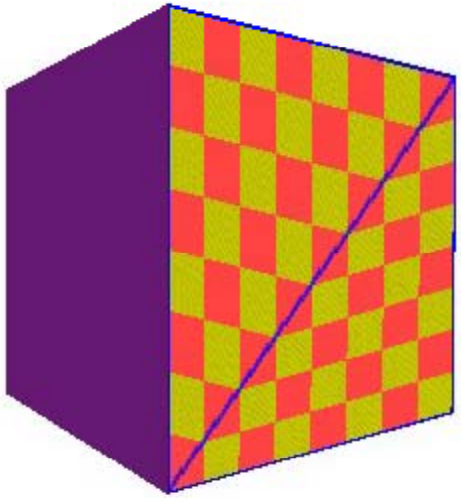
Demonstration

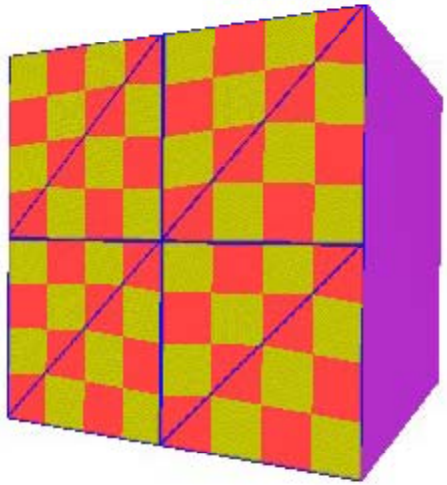
For obvious reasons this method of interpolation is called *perspective-correct interpolation*. The fact is, the name could be shortened to simply *correct interpolation*. You should be aware that not all 3-D graphics APIs implement perspective-correct interpolation.



Dealing with Incorrect Interpolation

You can reduce the perceived artifacts of non-perspective correct interpolation by subdividing the texture-mapped triangles into smaller triangles (why does this work?). But, fundamentally the screen-space interpolation of projected parameters is .





Wait a Minute!

When we did Gouraud shading, and discussed interpolating normals for Phong shading didn't we interpolate the values that we found at each vertex using screen-space interpolation? Didn't we just say that screen-space interpolation is wrong (I believe "inherently flawed" were my exact words)?

Does that mean that Gouraud shading is wrong?

Is everything that I've been telling you all one big lie?

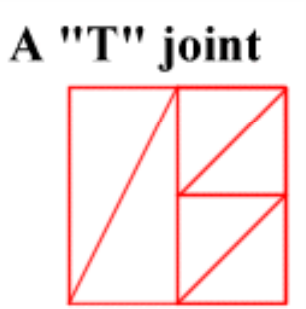
*Has **CISC 440** amounted to a total waste of time?*

Yes, Yes, Yes, Maybe

No, you've been exposed to a lot of nice purple cows!

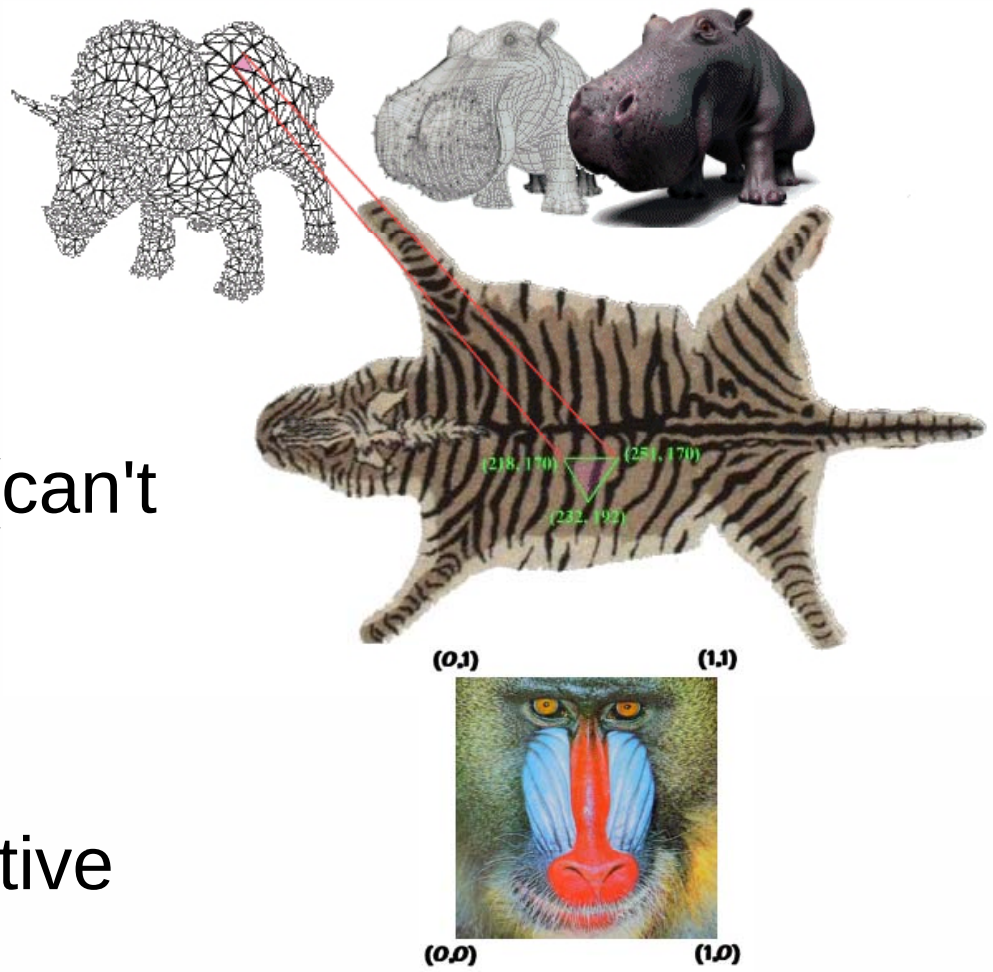
Traditional screen-space Gourand shading is wrong. However, you usually will not notice because the transition in colors is very smooth (And we don't know what the right color should be anyway, all we care about is a pretty picture). There are some cases where the errors in Gourand shading become obvious.

- When switching between different levels-of-detail representations
- At "T" joints.



Review of *Label* Textures

- Increases the apparent complexity of simple geometry
- Must specify texture coordinates for each vertex
- Projective correction (can't linearly interpolate in screen space)
- Specify variations in shading within a primitive



Texture

- Texture depicts spatially repeating patterns
- Many natural phenomena are textures

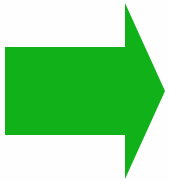
radishes

rocks

yogurt

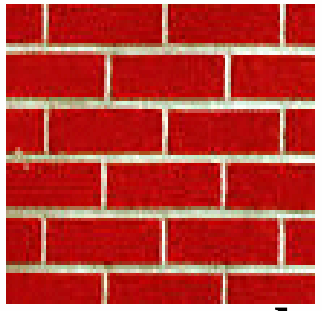
Texture Synthesis

- Goal of Texture Synthesis: create new samples of a given texture
- Many applications: virtual environments, hole-filling, texturing surfaces

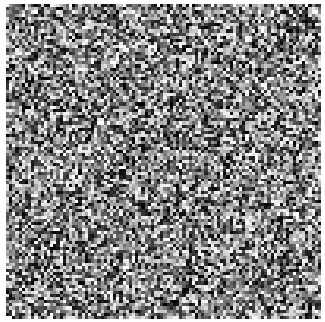


The Challenge

- Need to model the whole spectrum: from repeated to stochastic texture



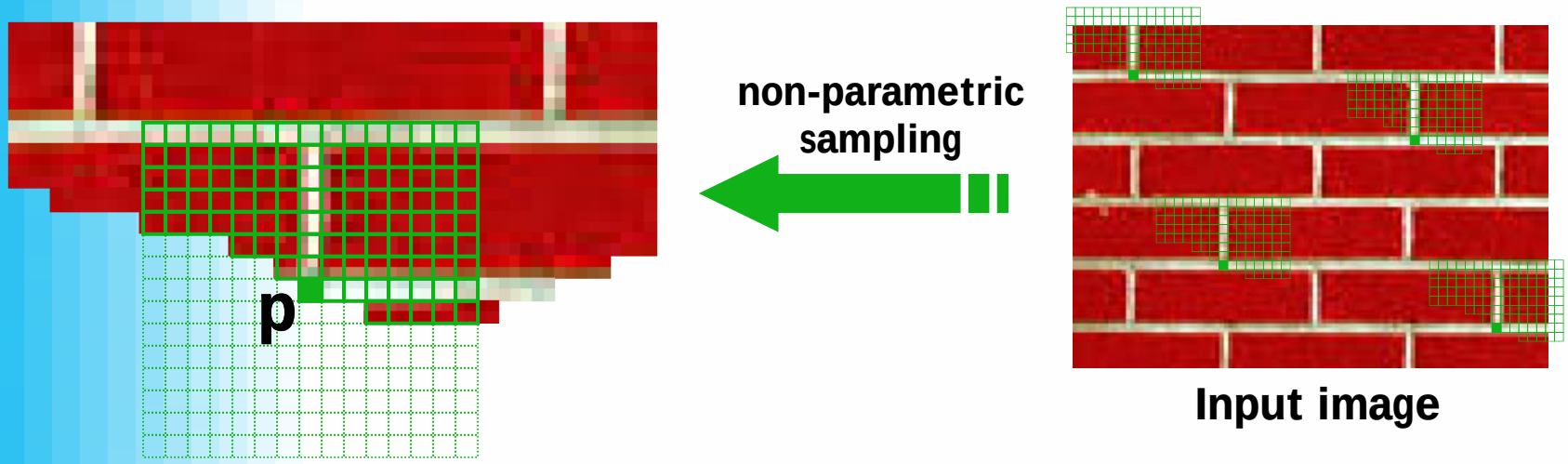
repeated



stochastic

Both?

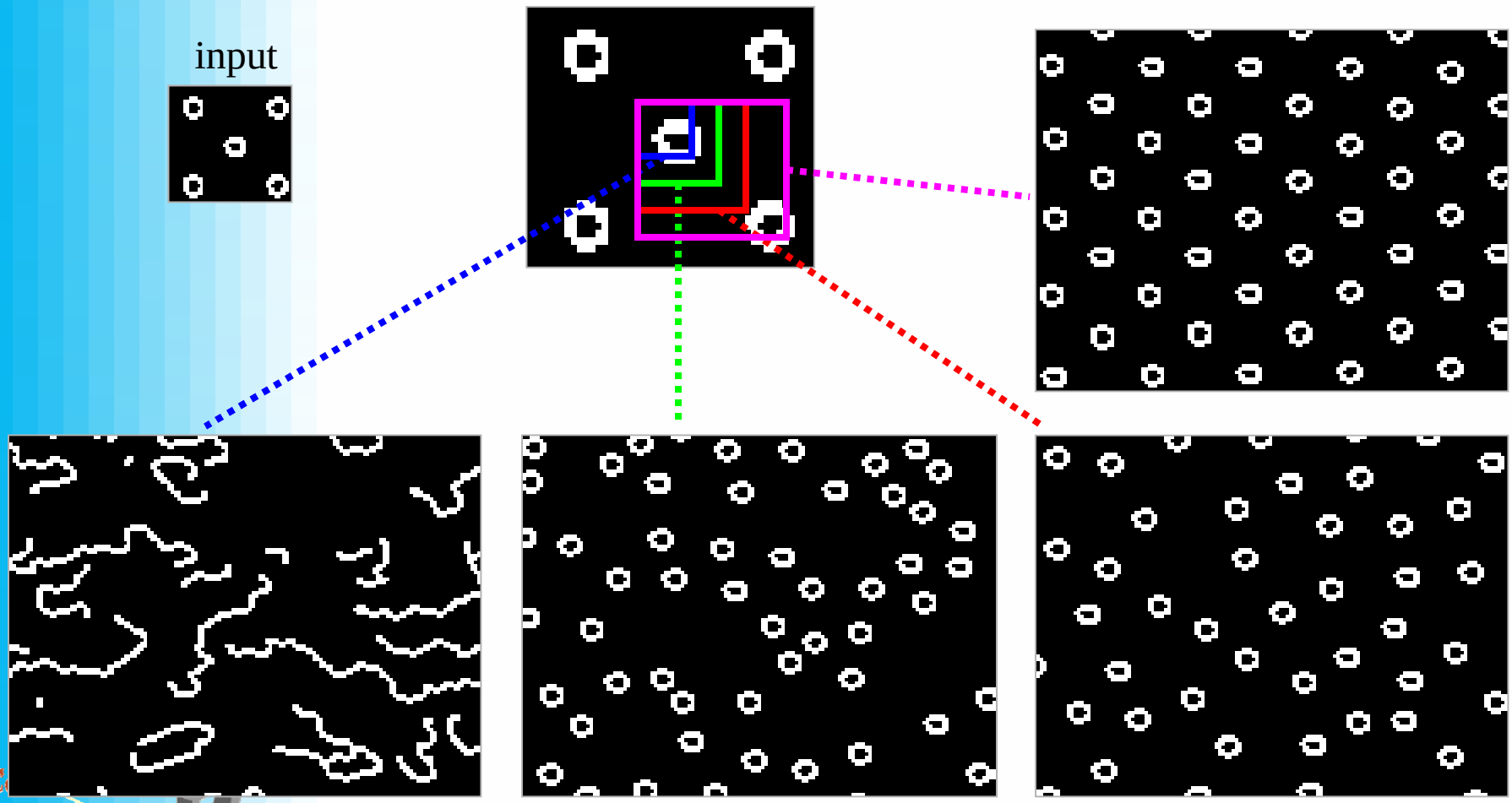
Efros & Leung Algorithm



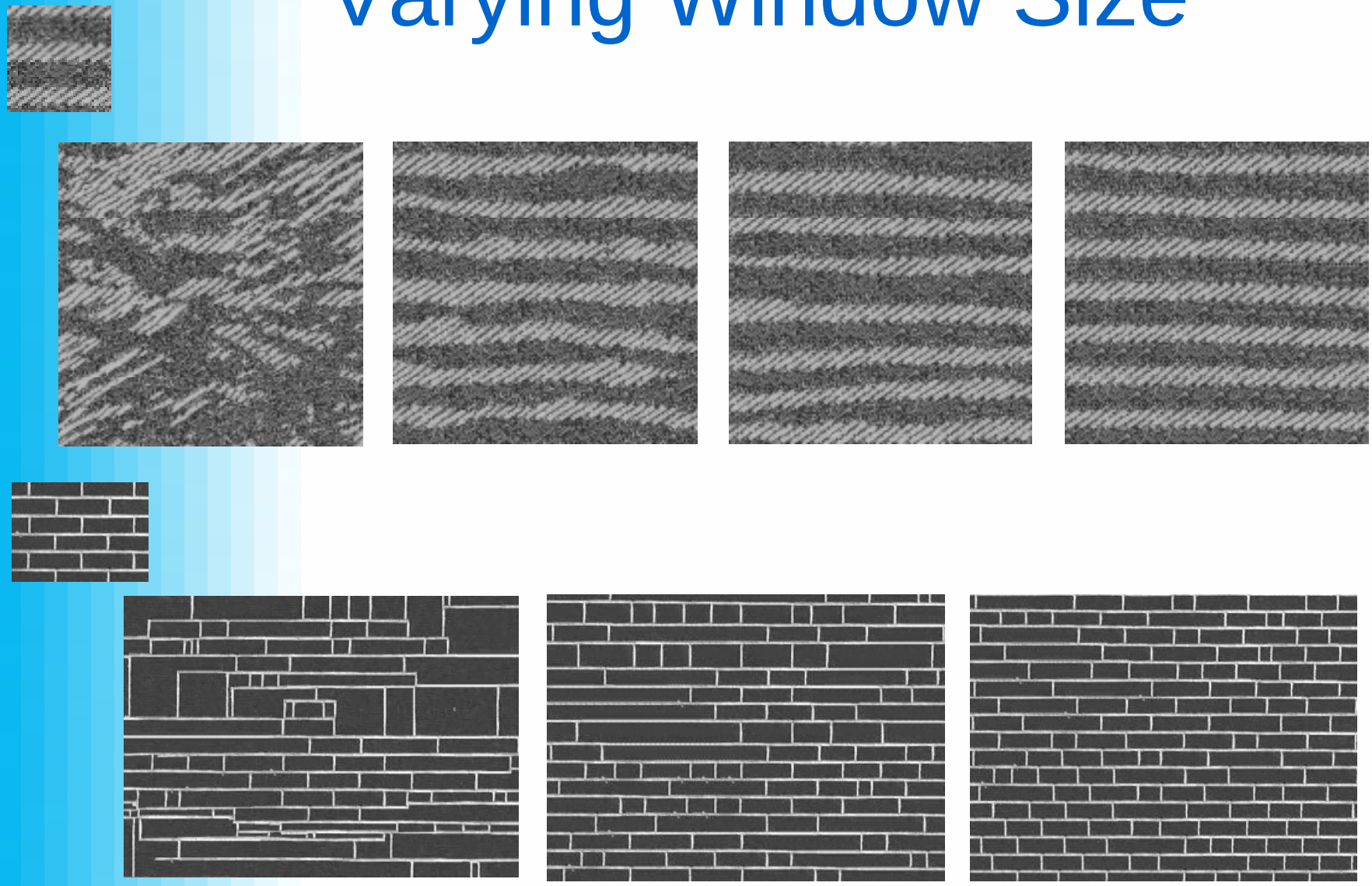
Synthesizing a pixel

– Assuming Markov property, compute P(**p**|N(**p**))
  - Building explicit probability tables infeasible
– Instead, we *search the input image* for all similar neighborhoods — that's our pdf for **p**
– To sample from this pdf, just pick one match at random

Neighborhood Window



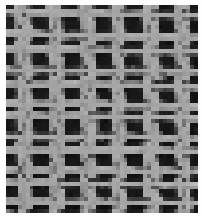
Varying Window Size

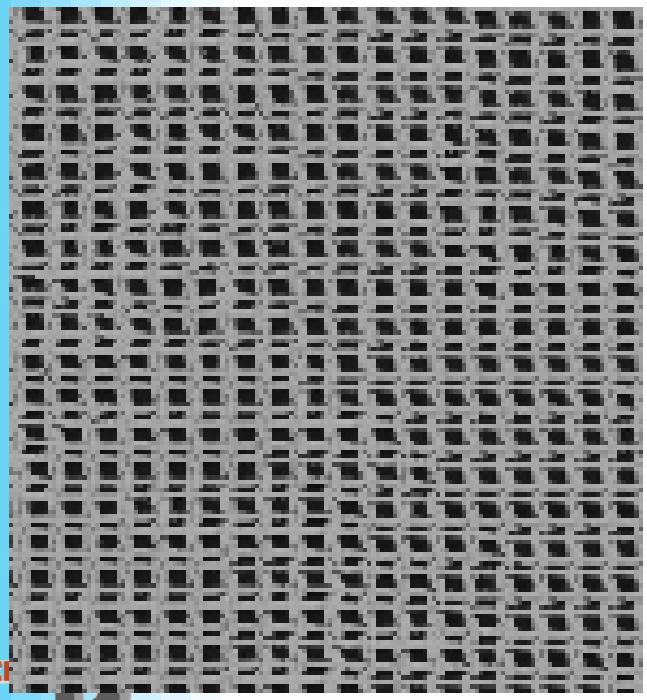


Increasing window size

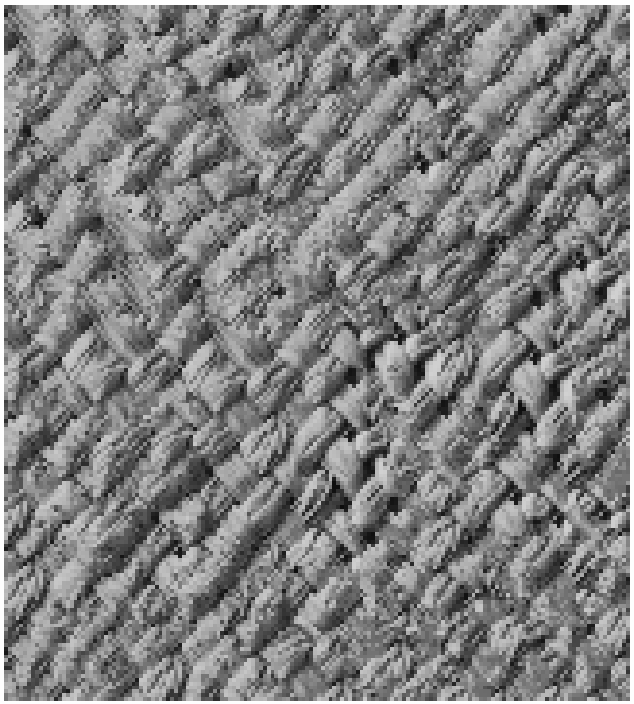
Synthesis Results

french canvas





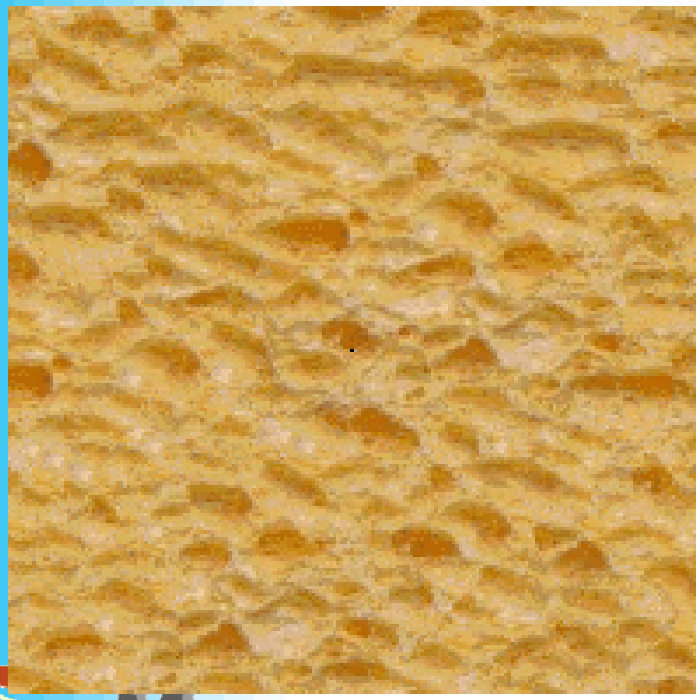
rafia weave



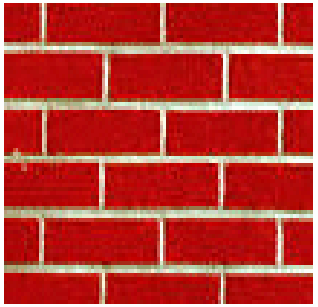
COMPUTER GRAPHICS

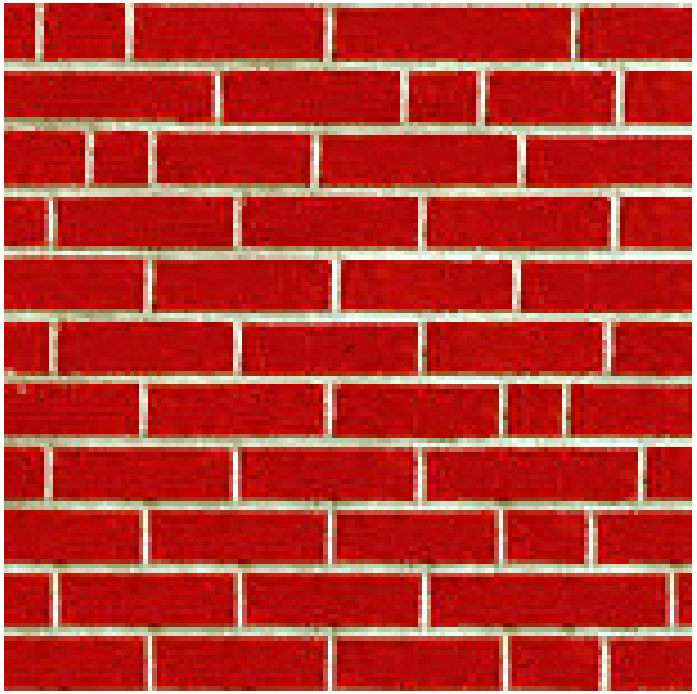
More Results

white bread

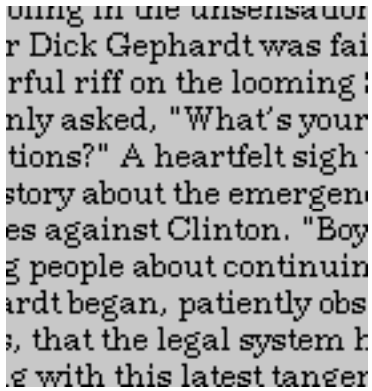


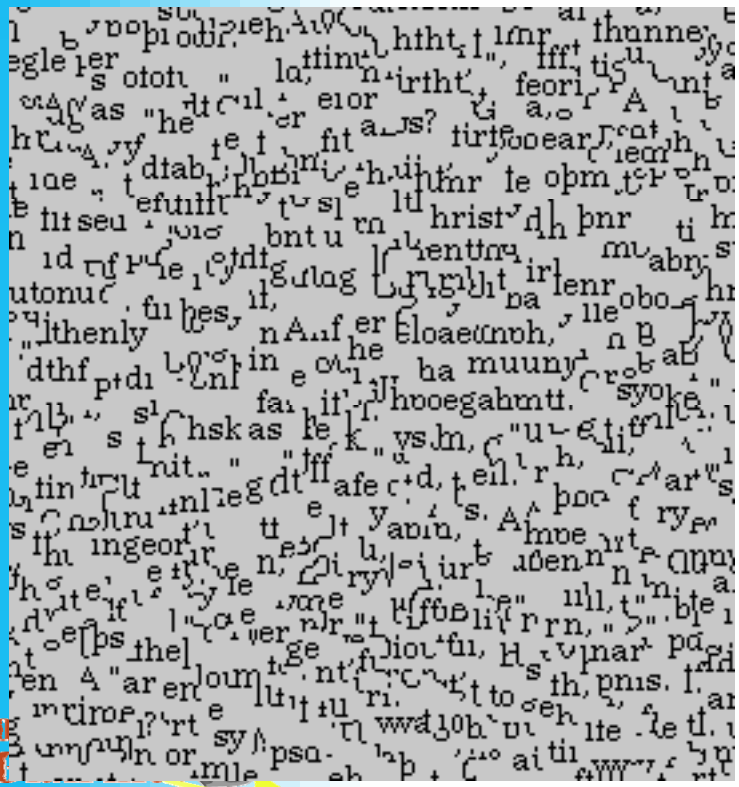
brick wall

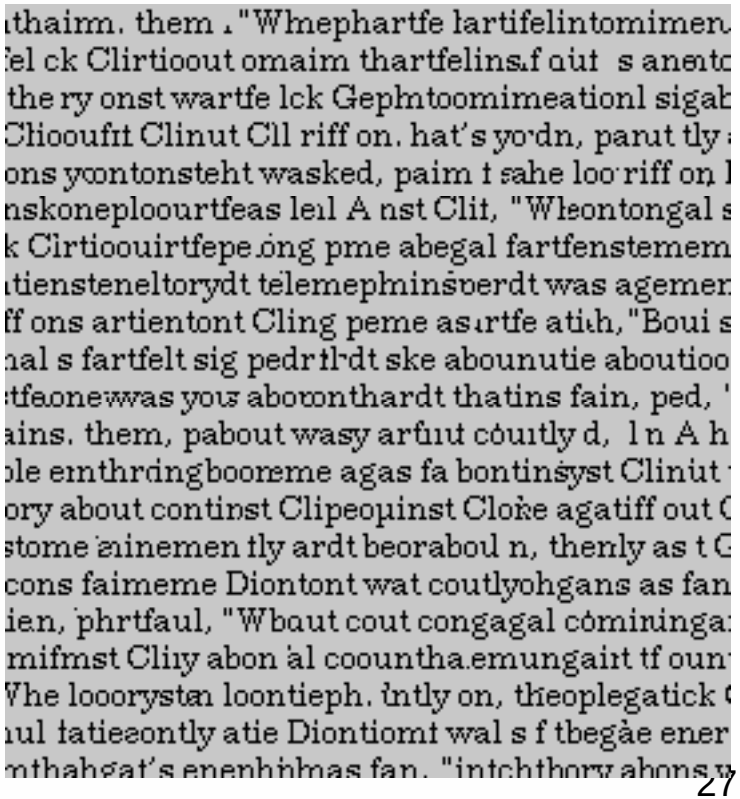




Homage to Shannon

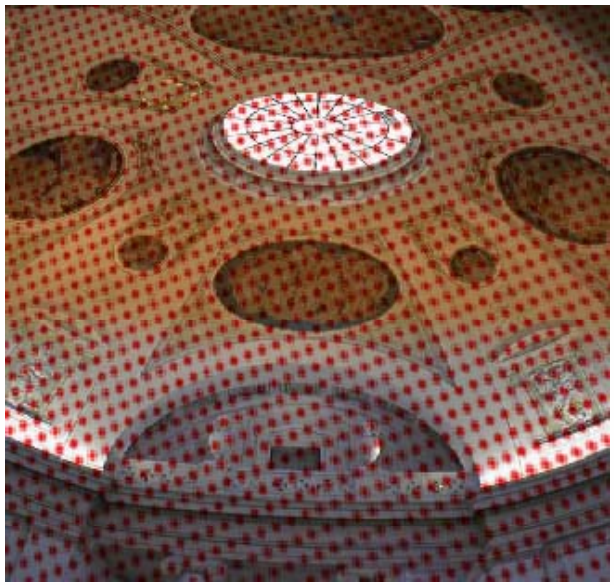

r Dick Gephardt was fai
rful riff on the looming
nly asked, "What's your
tions?" A heartfelt sigh
story about the emergen
es against Clinton. "Boy
g people about continuin
ardt began, patiently obs
, that the legal system h
g with this latest tanger




thaim. them ."Whephartfe lartifelintomimen
el ck Clirtioout omaim thartfelins.f oiit s anento
the ry onst wartfe lck Gephntoomimeationl sigab
Clioouftt Clinut Cll riff on. hat's yo·dn, parut tly
ons ycontonsteht wasked, paim t sahe loo·riff on
nskoneploourtfeas leıl A nst Clit, "Wleontongal s
k Cirtioouirtfepe.ong pme abegal fartfenstemem
tienstenelt orydt telemephninsverdt was agemen
ff ons artientont Cling peme asurtfe atiih,"Boui s
nal s fartfelt sig pedr tlrdt ske abounutie aboutioo
tfeonewas yous abotonthardt thatins fain, ped,
ains. them, pabout wasy arfiut cóuitly d, 1 n A h
ple emthrdngbooneme agas fa bontinsyst Cliniut
ory about continst Clipeopinst Cloke agatiff out
stome ainemen tly ardt beoraboul n, thenly as t G
cons faimeme Diontont wat coutlyohgans as fan
ien, phrtfaul, "Wbaut cout congagal cóminingai
mifmst Cliiy abon al coountha.emungairt tf oun
Vhe looorysten loontieph. intly on, theoplegatick
ul tatieeontly atie Diontiomt wal s f tbegàe ener
nthahgat's enenhhmas fan. "intchthory ahons v

Sampling Texture Maps

- When texture mapping it is rare that the screen-space sampling density matches the sampling density of the texture. Typically one of two things can occur:

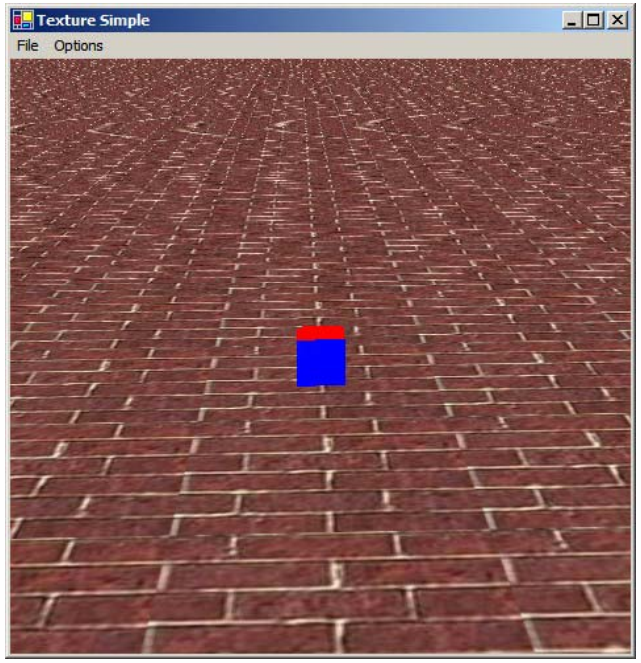


Oversampling of the texture or **Undersampling of the texture**

- In the case of undersampling we already know what to do... interpolation. But, how do we handle oversampling?

How Bad Does it Look?

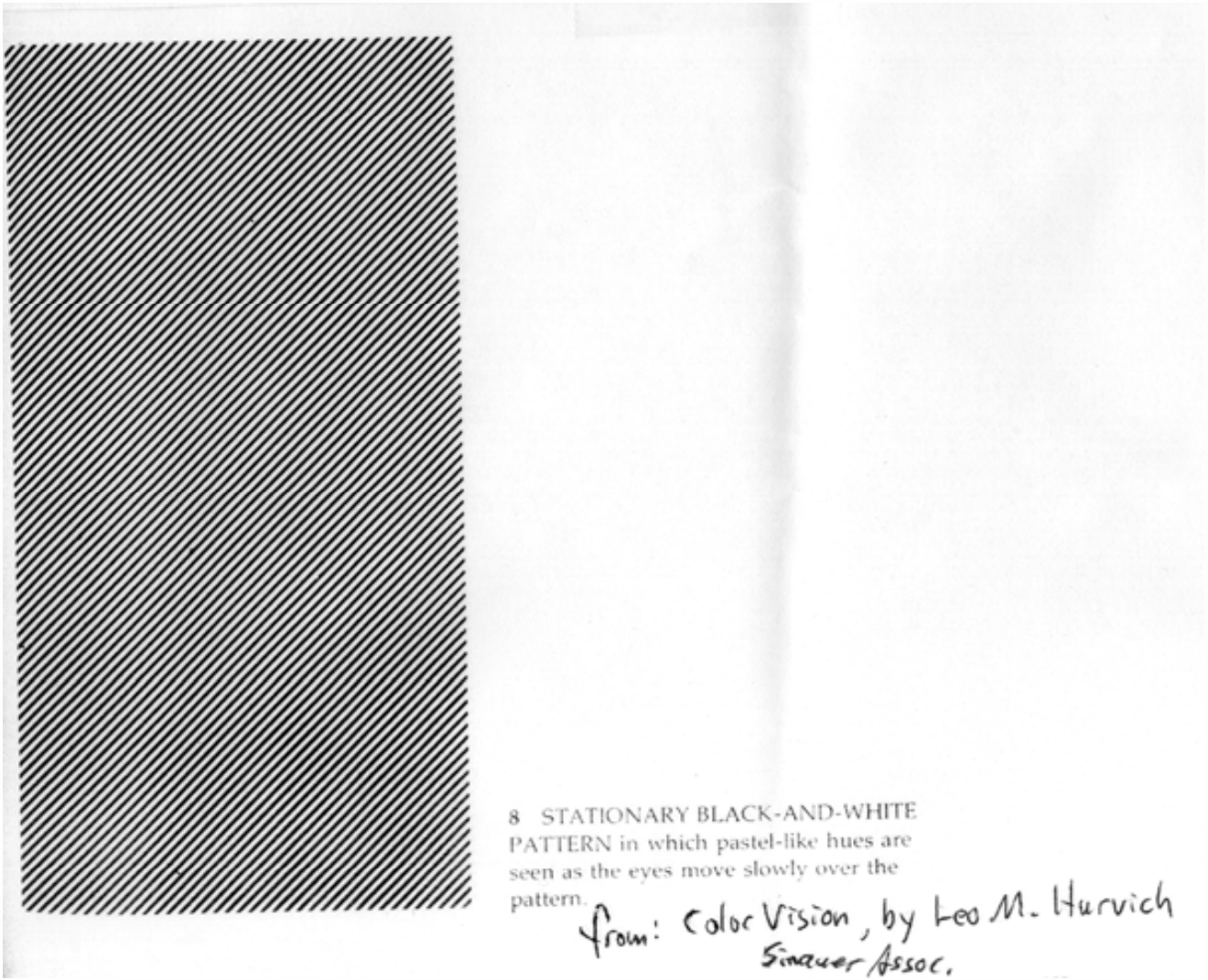
- Let's take a look at what oversampling looks like:



- Notice how details in the texture, in particular the mortar between the bricks, tend to pop (disappear and reappear). This popping is most noticeable around details (parts of the texture with a high-spatial frequency). This is indicative of aliasing (high-frequency details showing up in areas where we expect to see low frequencies).

Our Eyes Also Observe Aliasing

Scale relative to human photoreceptor size: each line covers about 7 photoreceptors.

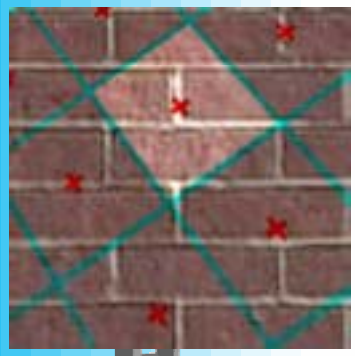


8 STATIONARY BLACK-AND-WHITE PATTERN in which pastel-like hues are seen as the eyes move slowly over the pattern.

from: Color Vision, by Leo M. Hurvich
Sinauer Assoc.

Spatial Filtering

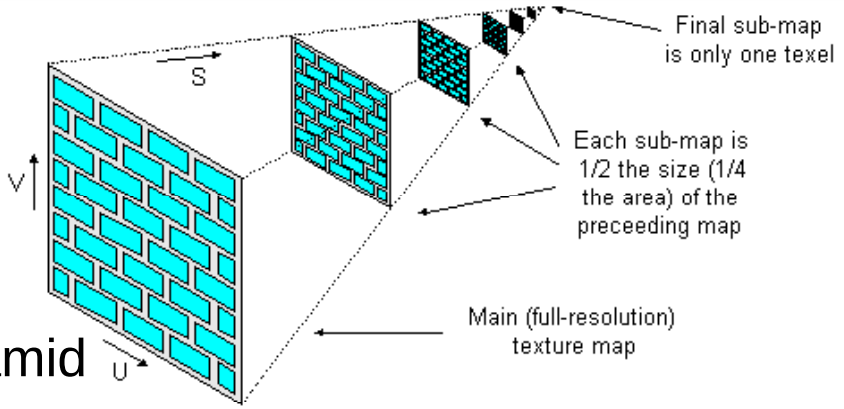
- In order to get the sort of images the we expect, we must *prefilter* the texture to remove the high frequencies that show up as artifacts in the final rendering. The prefiltering required in the undersampling case is basically a spatial integration over the extent of the sample.

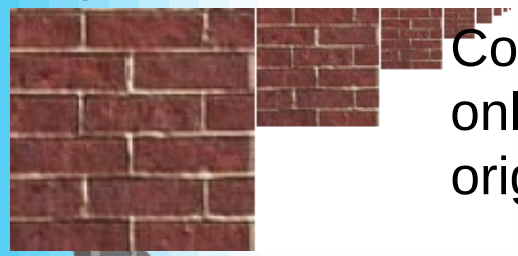


We could perform this filtering while texture mapping (during rasterization), by keeping track of the area enclosed by sequential samples and performing the integration as required. However, this would be expensive. The most common solution to undersampling is to perform prefiltering prior to rendering.

MIP Mapping

- MIP Mapping is one popular technique for precomputing and performing this prefiltering. MIP is an acronym for the Latin phrase *multium in parvo*, which means "many in a small place". The technique was first described by Lance Williams. The basic idea is to construct a *pyramid of images* that are prefiltered and resampled at sampling frequencies that are a binary fractions (1/2, 1/4, 1/8, etc) of the original image's sampling.
- While rasterizing we compute the index of the decimated image that is sampled at a rate closest to the density of our desired sampling rate (rather than picking the closest one can in also interpolate between pyramid levels).

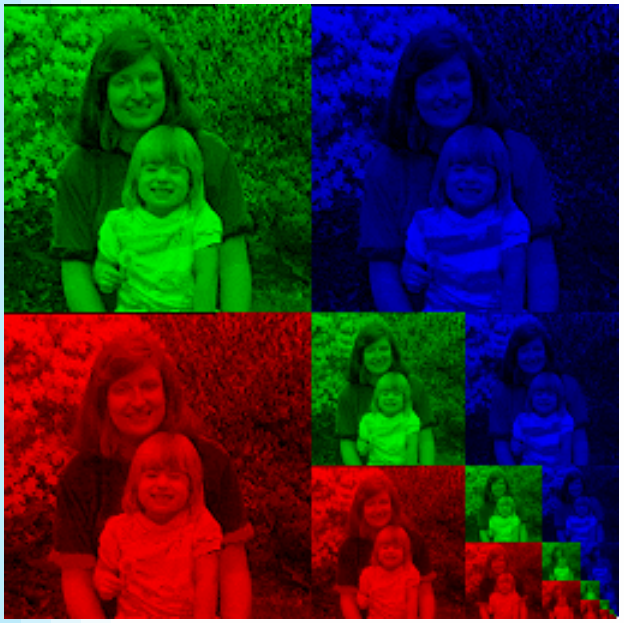


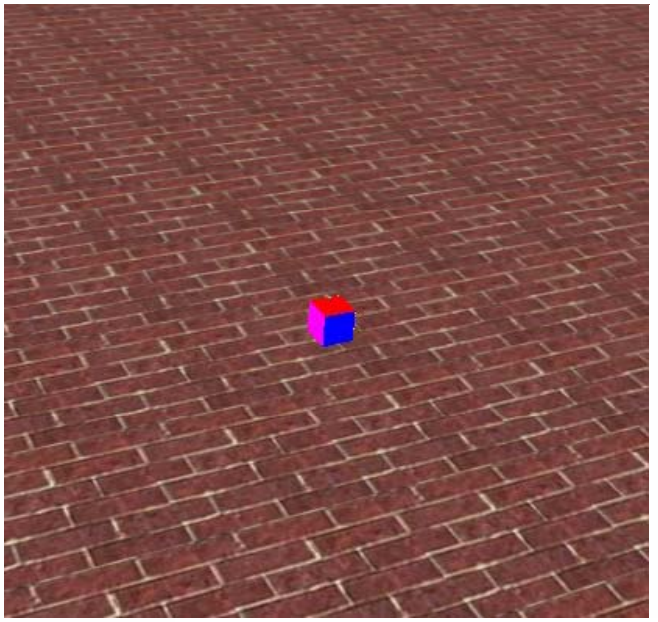


Computing this series of filtered images requires only a small fraction of additional storage over the original texture (How small of a fraction?).

Storing MIP Maps

- One convienent method of storing a MIP map is shown below (It also nicely illustrates the 1/3 overhead of maintaining the MIP map).





- The rasterizer must be modified to compute the MIP map level. Remember the equations that we derived last lecture for mapping screen-space interpolants to their 3-space equivalent.

$$u = u_1 + s(u_2 - u_1)$$

$$s = \frac{t\, w_2}{w_1 + t(w_2 - w_1)}$$

Finding the MIP level

- What we'd like to find is the step size that a uniform step in screen-space causes in three-space, or, in other words how a screen-space change relates to a 3-space change. This sounds like the derivatives, *(du/dt, dv/dt)*. They can be computed simply using the chain rule:

$$\frac{du}{dt} = \frac{du}{ds}\frac{ds}{dt} = (u_2 - u_1)\frac{w_1\, w_2}{(w_1 + t(w_2 - w_1))^2}$$

$$\frac{dv}{dt} = (v_2 - v_1)\frac{w_1\, w_2}{(w_1 + t(w_2 - w_1))^2}$$

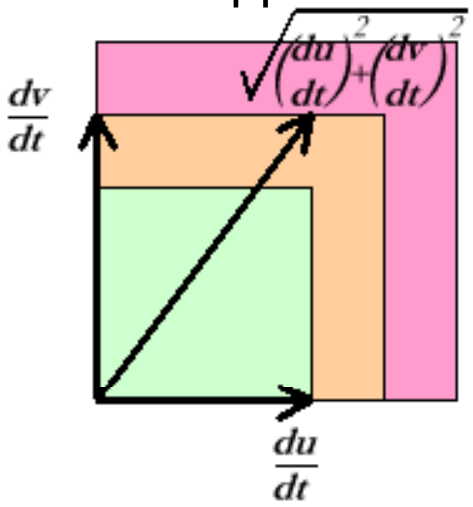
- Notice that the term being squared under the numerator is just the *w* plane equation that we are already computing. The remaining terms are constant for a given rasterization. Thus, all we need to do to compute the derivative is a square the *w* accumulator and multiply it by a couple of constants.
- Now, we know how a step in screen-space relates to a step in 3-space. So how do we translate this to an index into our MIP table?

MIP Indices

- Actually, you have a choice of ways to translate this **gradient value** into a MIP level. This also brings up one of the shortcomings of MIP mapping. MIP mapping assumes that both the *u* and *v* components of the texture index are undergoing a uniform scaling, while in fact the terms *du/dt* and *dv/dt* are relatively independent. Thus, we must make some sort of compromise. Two of the most common approaches are given below:

$$level = \log_2\left(\sqrt{\left(\frac{du}{dt}\right)^2 + \left(\frac{dv}{dt}\right)^2}\right)$$

$$level = \log_2\left(Max\left(\left|\frac{du}{dt}\right|, \left|\frac{dv}{dt}\right|\right)\right)$$



The differences between these level selection methods is illustrated in the accompanying figure.

OpenGL Code Example

Incorporating MIPmapping into OpenGL applications is surprisingly easy.

```
// Boilerplate Texture setup code

glTexImage2D(GL_TEXTURE_2D, 0, 4, texWidth, texHeight, 0,
GL_RGBA,GL_UNSIGNED_BYTE, data);
gluBuild2DMipmaps(GL_TEXTURE_2D, 4, texWidth, texHeight, GL_RGBA,
GL_UNSIGNED_BYTE, data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
                                                GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

OpenGL also provides a facility for specifying the MIPmap image at each level using multiple calls to the glTexImage*D() function. This approach provides more control over filtering in the MIPmap construction and enables a wide range of tricks.

The gluBuildMipmaps() utility routine will automatically construct a mipmap from a given texture buffer. It will filter the texture using a simple box filter and then subsample it by a factor of 2 in each dimension. It repeats this process until one of the texture's dimensions is 1. Each texture ID, can have multiple levels associated with it. GL_LINEAR_MIPMAP_LINEAR trilinearly interploates between texture indices and MIPmap levels. Other options include GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, and GL_LINEAR_MIPMAP_NEAREST.

Summed-Area Tables

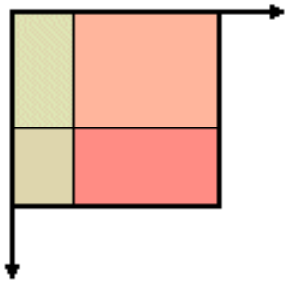
There are other approaches to computing this prefiltering integration on the fly. One, which was introduced by Frank Crow is called a *summed-area table*. Basically, a summed-area table is a tabularized two-dimensional cumulative distribution function. Imagine having a 2-D table of numbers the cumulative distribution function could be found as shown below.

1	6	8	3
0	0	3	7
4	7	8	8
5	0	9	9

1	7	15	18
1	7	18	28
5	18	37	55
10	23	51	78

- To find the sum of region contained in a box bounded by (x_0, y_0) and (x_1, y_1):

$$\mathbf{T}(x_1, y_1) - \mathbf{T}(x_0, y_1) - \mathbf{T}(x_1, y_0) + \mathbf{T}(x_0, y_0)$$



- This approach can be used to compute the integration of pixels that lie under a pixel by dividing the resulting sum by the area of the rectangle,

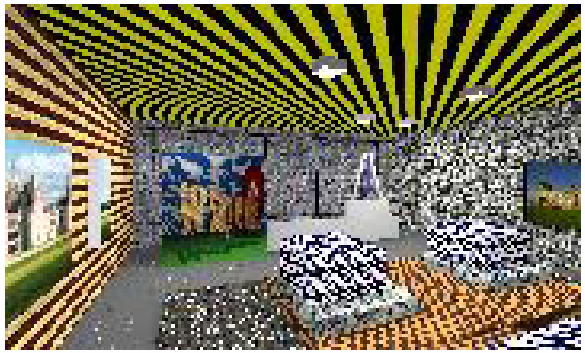
$$(y_1 - y_0)(x_1 - x_0).$$

- With a little more work you can compute the area under any four-sided polygon (How?).

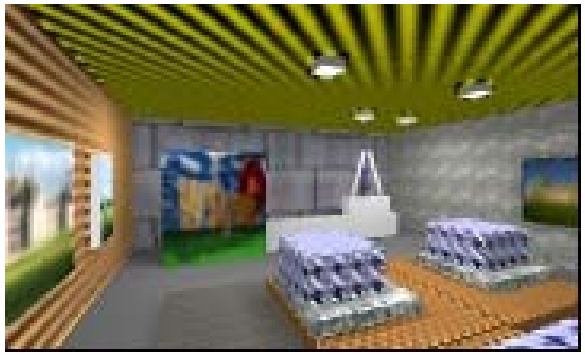
Summed-Area Tables

- How much storage does a summed-area table require?
- Does it require more or less work per pixel than a MIP map?
- What sort of low-pass filter does a summed-area table implement?
- What do you remember about this sort of filter if you have taken a signal processing class?

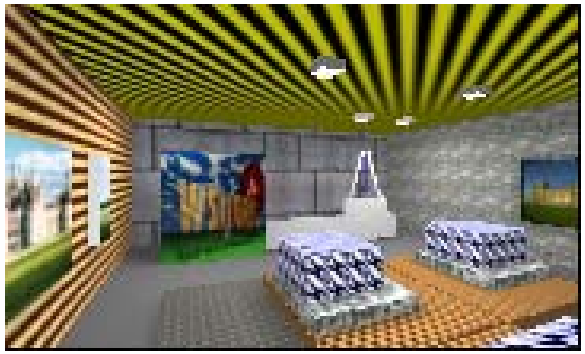
No Filtering



MIP mapping

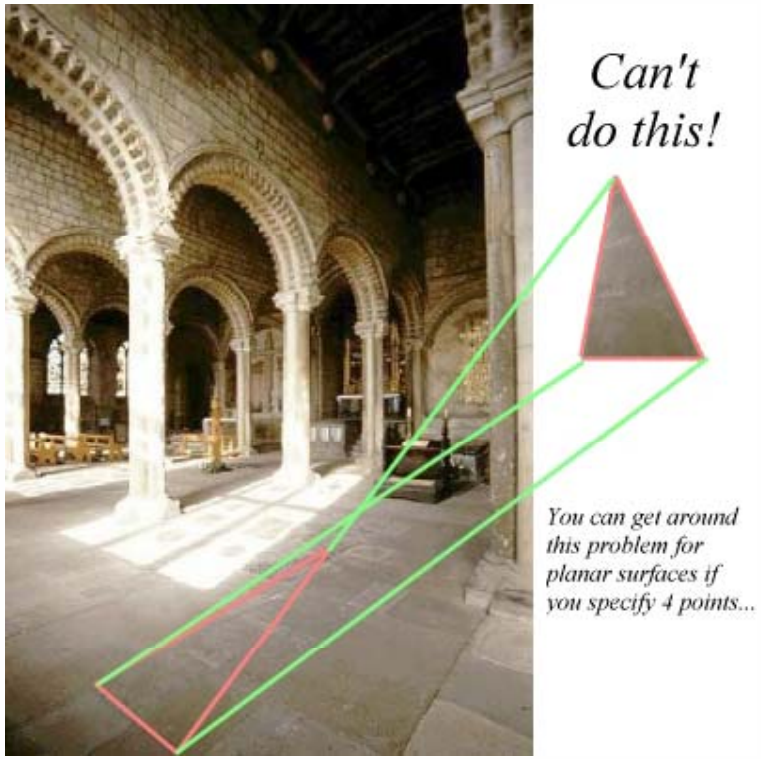


Summed-Area Table



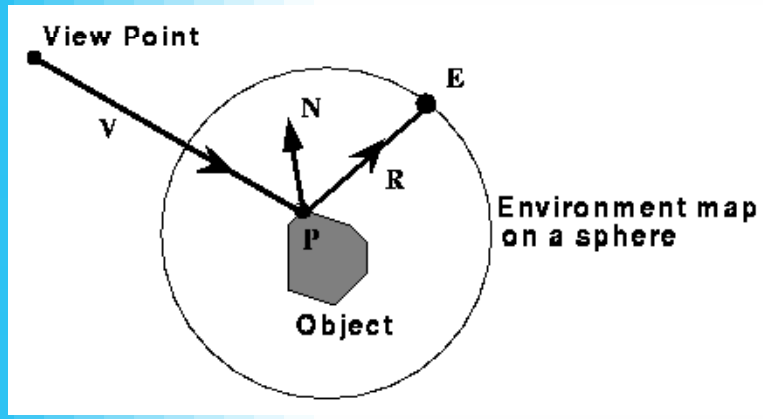
Other Problems with Label Textures

- Tedious to specify texture coordinates for every triangle
- Textures are attached to the geometry
- Easier to model variations in reflectance than illumination
- Can't use just any image as a label texture
- **The "texture" can't have projective distortions**
  Reminder: linear interpolation in image space is not equivalent to linear interpolation in 3-space (This is why we need "perspective-correct" texturing).
  The converse is also true.
- Textures are attached to the geometry
- Easier to model variations in reflectance than illumination
- Makes it hard to use pictures as textures



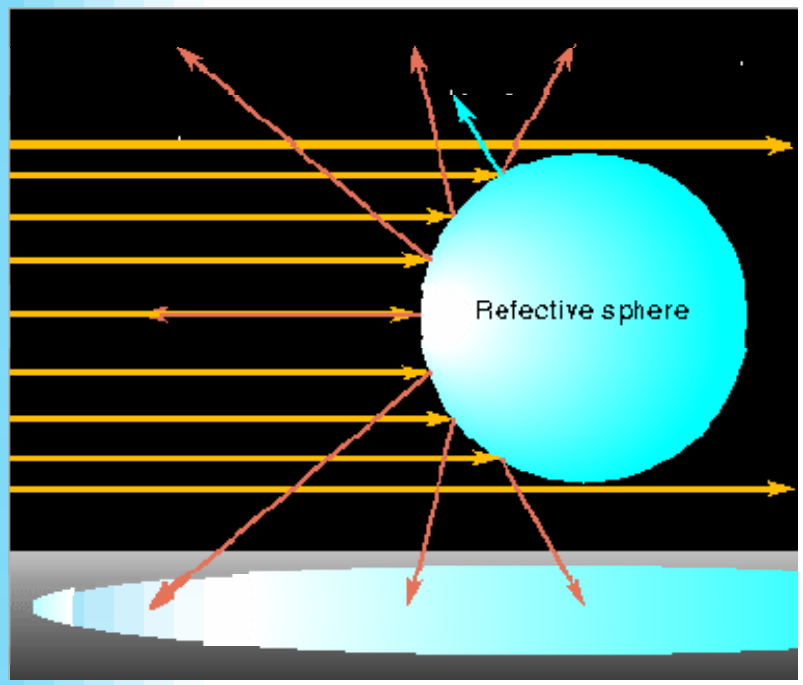
Environment Maps

Instead of using transformed vertices to index the projected texture', we can use transformed *surface normal* s to compute indices into the texture map. These sorts of mapping can be used to simulate reflections, and other shading effects. This approach is not completely accurate. It assumes that all reflected rays begin from the same point, and that all objects in the scene are the same distance from that point.



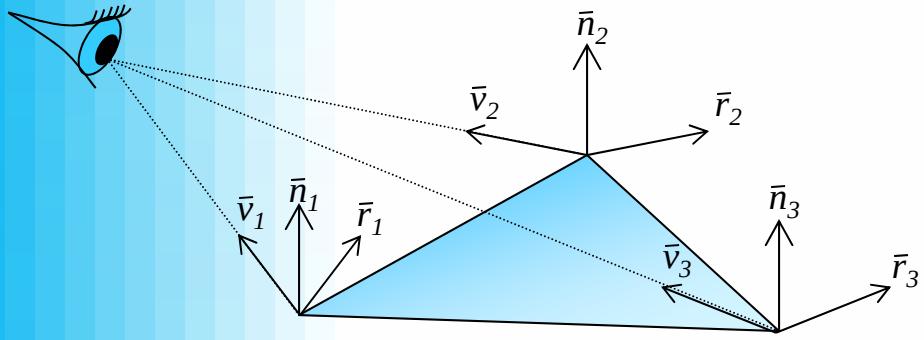
Sphere Mapping Basics

- OpenGL provides special support for a particular form of Normal mapping called sphere mapping. It maps the normals of the object to the corresponding normal of a sphere. It uses a texture map of a sphere viewed from infinity to establish the color for the normal.



Sphere Mapping

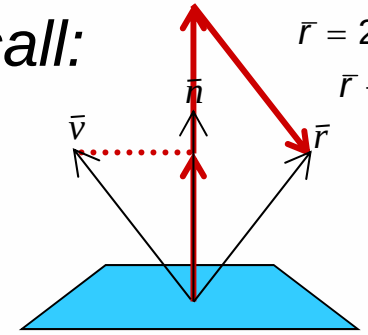
- Mapping the normal to a point on the sphere



Recall:

$$\bar{r} = 2(\bar{n}\cdot\bar{v})\bar{n} - \bar{v}$$

$$\bar{r} + \bar{v} = \alpha\bar{n}$$

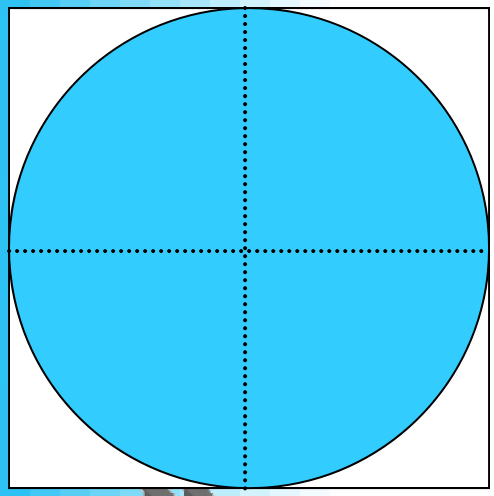


$$\alpha\bar{n} = \bar{r} + \bar{v} = \begin{bmatrix} r_x \\ r_y \\ r_z \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\frac{\alpha\bar{n}}{\|\alpha\bar{n}\|} = \begin{bmatrix} \frac{r_x}{p} \\ \frac{r_y}{p} \\ \frac{r_z+1}{p} \\ 0 \end{bmatrix}$$

$$p = \sqrt{r_x^2 + r_y^2 + (r_z+1)^2}$$

(1,1)



(-1,-1)

$$\bar{n} = \begin{bmatrix} s \\ t \\ \sqrt{1 - s^2 - t^2} \\ 0 \end{bmatrix}$$

$$s = \frac{r_x}{p} \quad t = \frac{r_y}{p}$$

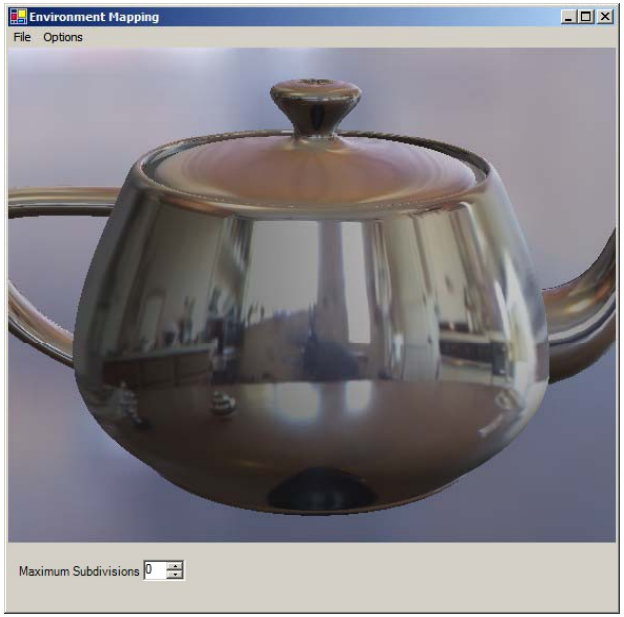
$$s' = \frac{s}{2} + \frac{1}{2} \quad t' = \frac{t}{2} + \frac{1}{2}$$

$$s' = \frac{r_x}{2p} + \frac{1}{2} \quad t' = \frac{r_y}{2p} + \frac{1}{2}$$

OpenGL code Example

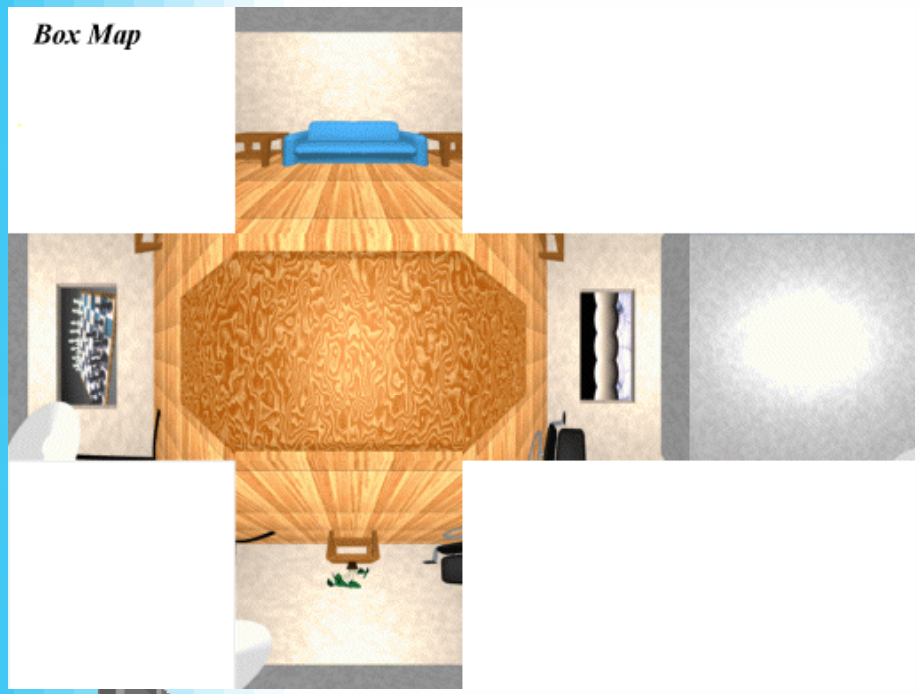
```
// this gets inserted where the texture is created

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, (int) GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, (int) GL_SPHERE_MAP);


 // Add this before rendering any primatives

 if (texWidth > 0) {
  glEnable(GL_TEXTURE_2D);
  glEnable(GL_TEXTURE_GEN_S);
  glEnable(GL_TEXTURE_GEN_T);
 }
```

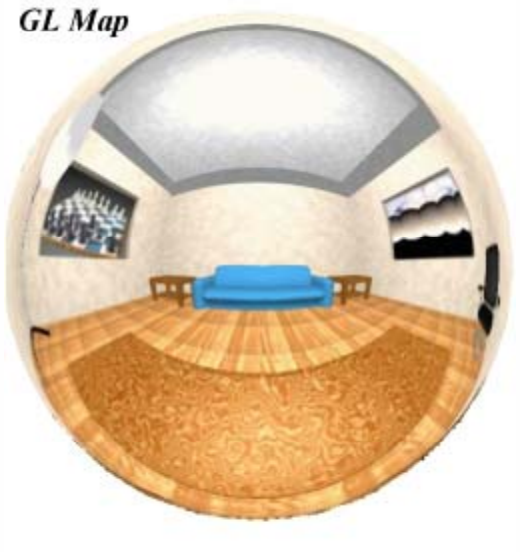
This was a very special purpose hack in OpenGL, however, we have it to thank for a lot of the flexibility in today's graphics hardware… this hack was the genesis of programmable vertex shading.



What's the Best Map?

A sphere map is not the only representation choice for environment maps. There are alternatives, with more uniform sampling properties, but they require different normal-to-texture mapping functions.





Shadow Map

- Render an image from the *light's* point of view
  - Camera look-from point is the light position
  - Aim camera to look at objects in scene
  - Render only the z-buffer depth values
    - Don't need colors
    - Don't need to compute lighting or shading
      - (unless a procedural shader would make an object transparent)
- Store result in a *Shadowmap* AKA *depth map*
  - Store the depth values
  - Also store the (inverse) camera & projection transform

- Remember, z-buffer pixel holds depth of closest object to the camera
  - A shadow map pixel contains the distance of the closest object to the light

Shadow Map

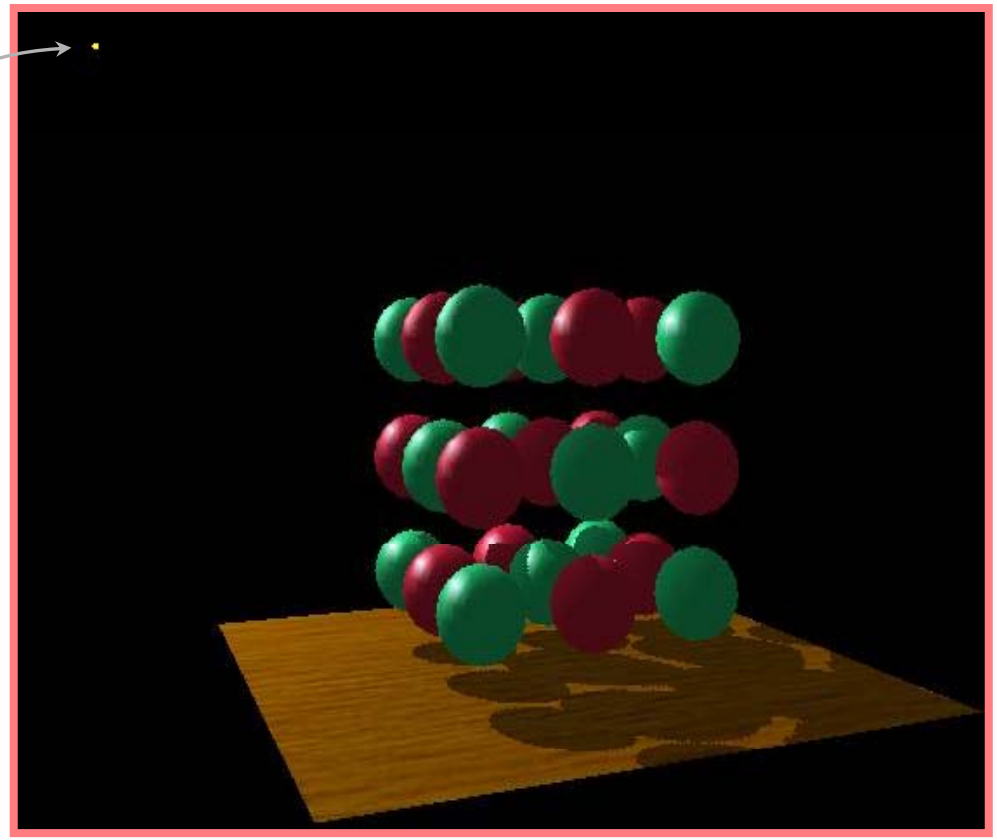
- Directional light source
  - Use orthographic shadow camera
- Point light source
  - Use perspective shadow camera

Shadow Mapping

- When lighting a point on a surface
  - For each light that has a shadowmap…
  - Transform the point to the shadowmap's image space
    - Get X,Y,Z values
    - Compare Z to the depth value at X,Y in the shadowmap
    - If the shadowmap depth is less than Z
      - some other object is closer to the light than this point
      - this light is blocked, don't include it in the illumination
    - If the shadowmap is the same as Z
      - this point is the one that's closest to the light
      - illuminate with this light
      - (because of numerical inaccuracies, test for almost-the-same-as Z)

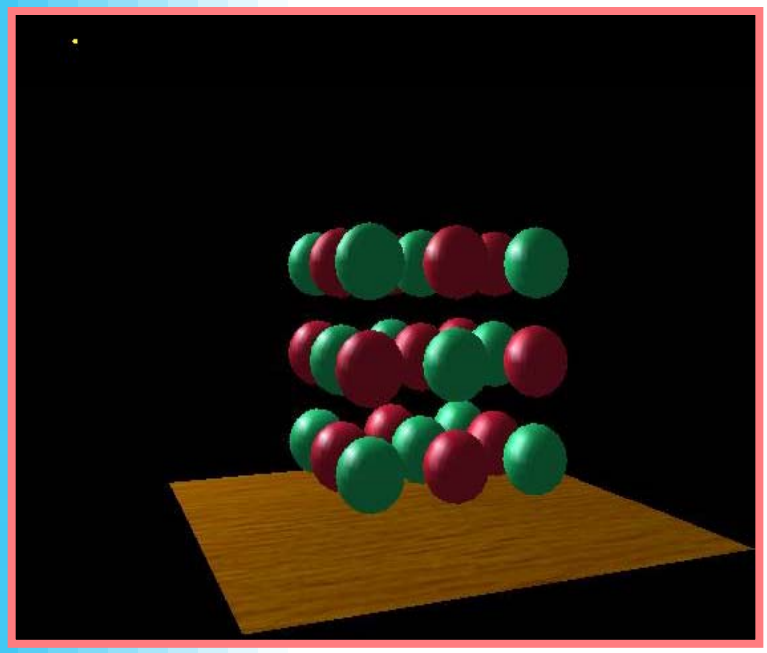
Shadow Mapping

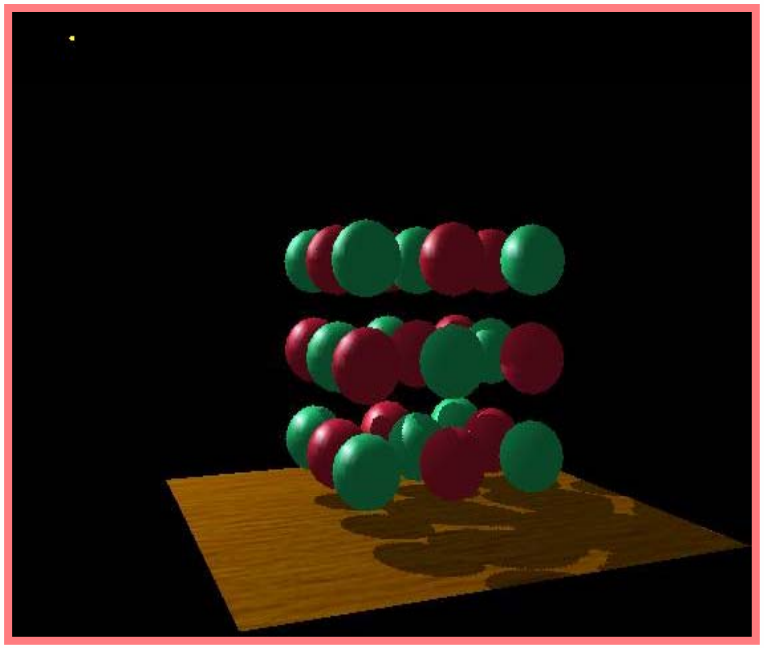
- A scene with shadows



Shadow Mapping

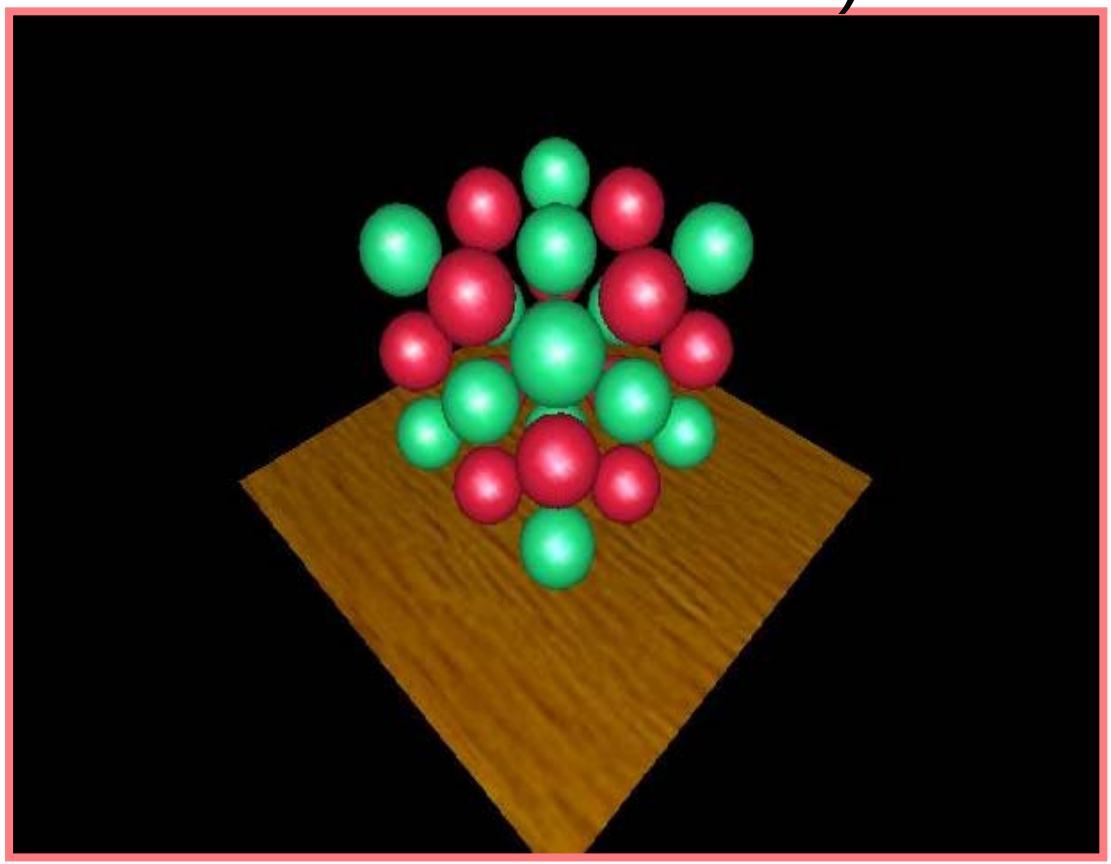
- Without and with shadows





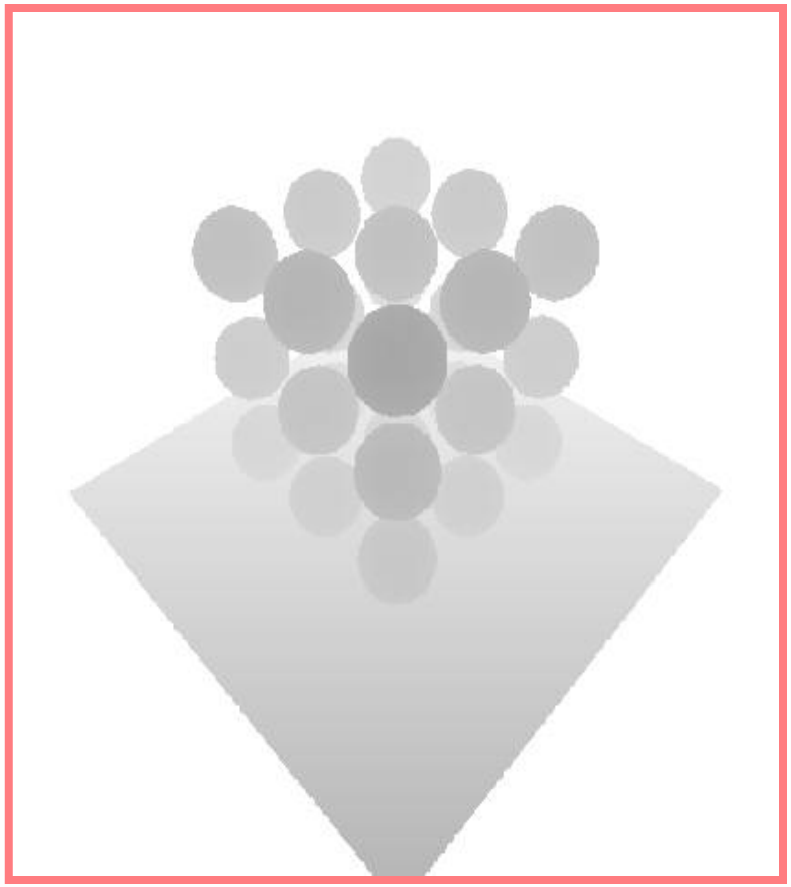
Shadow Mapping

- The scene from the shadow camera
  - (just FYI -- no need to save this)



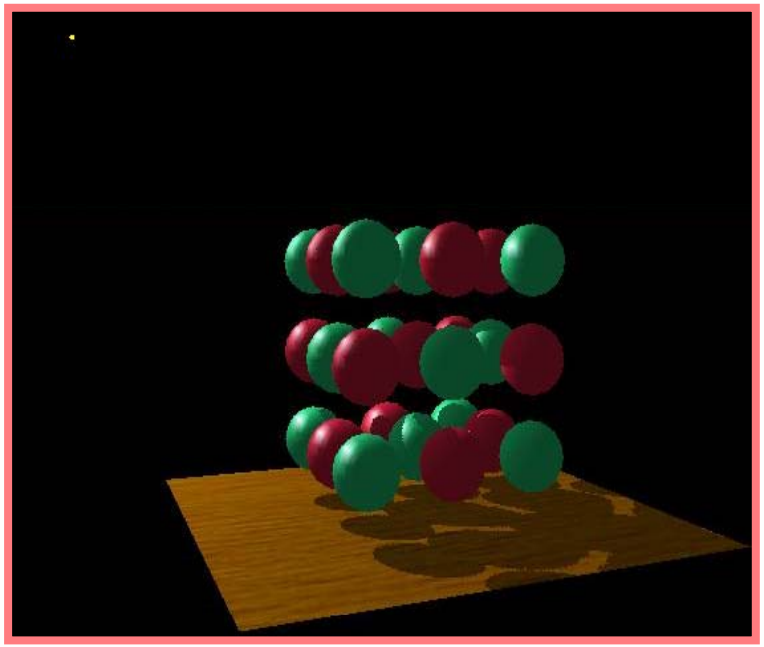
Shadow Mapping

- The shadowmap depth buffer
  - Darker is closer to the camera



Shadow Mapping

- Visualization…
  - Green: surface light Z is (approximately) equal to depth map Z
  - Non-green: surface is in shadow



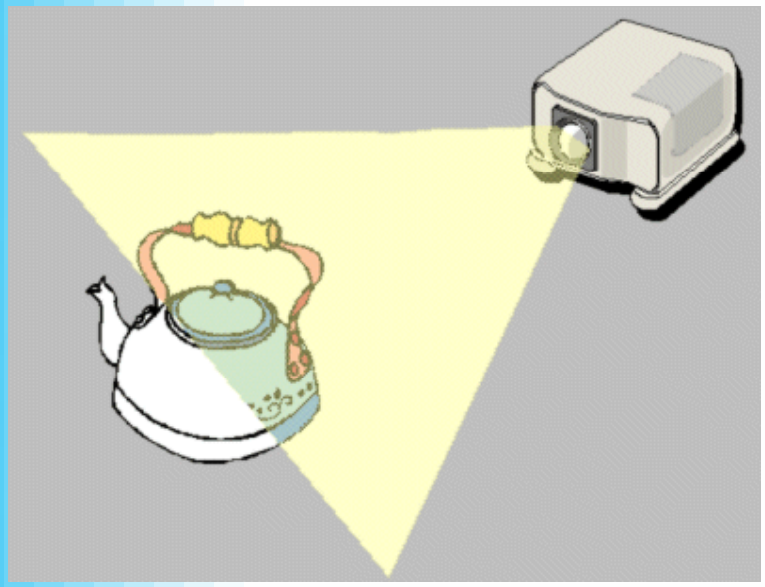
COMPUTER GRAPHICS

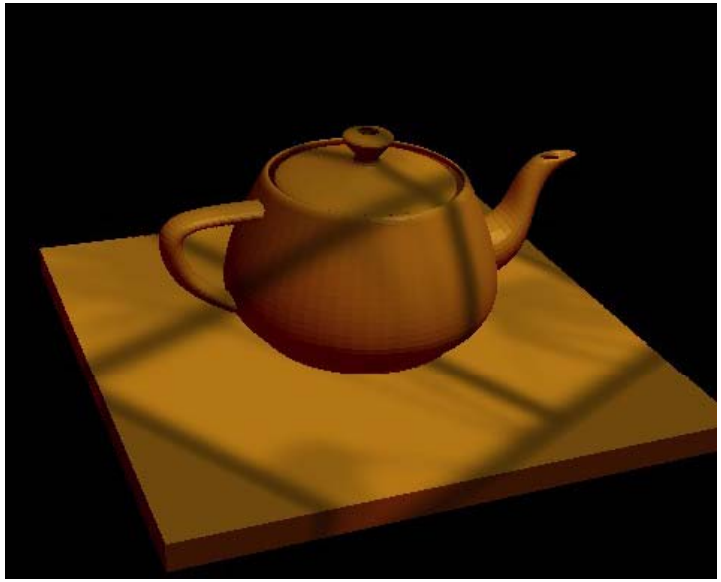
Shadow Mapping Notes

- Very commonly used
- Problems:
  - Blocky shadows, depending on resolution of shadowmap
  - Shadowmap pixels & image samples don't necessarily line up
    - Hard to tell if object is really the closest object
    - Typically add a small bias to keep from self-interfering
    - But the bias causes shadows to separate from their objects
  - No great ways to get soft shadows (Penumbras)

Projective Textures

- Treat the texture as a light source (like a slide projector)
- No need to specify texture coordinates explicitly
- A good model for shading variations due to illumination (cool spotlights)
- A fair model for view-dependent reflectance (can use pictures)



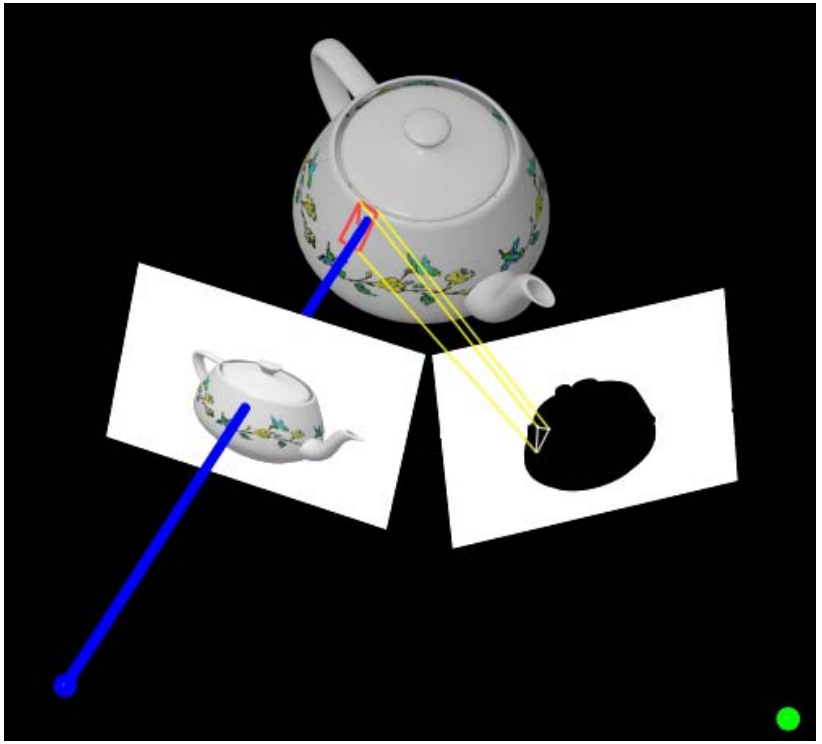


The Mapping Process

During the Illumination process:

For each vertex of triangle
(in world or lighting space)
Compute ray from the projective texture's origin to point
Compute homogeneous texture coordinate, $[ti, tj, t]^T$

During scan conversion
(in projected screen space)
Interpolate all three texture coordinates in 3-space
(premultiply by *w* of vertex)
Do normalization at each rendered pixel
$i = t\,i / t, \quad j = t\,j / t$
Access projected texture



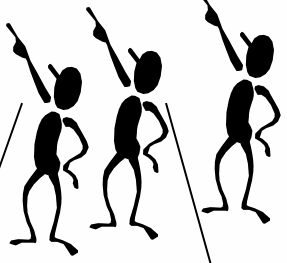
This is the same process, albeit with an additional transform, as perspective correct texture mapping. Thus, we can do it for free! Almost.

Another Frame "Texture Space"

- OpenGL is able to insert this extra projection transformation for textures by including another matrix stack, called GL_TEXTURE.
- The transform we want is:

$$T_{eye} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} P_{proj} V_{proj} M_{eye-to-world}$$

This extra matrix maps from normalized device coordinates ranging from [-1,1] to valid texture coordinates ranging from [0,1].



This matrix undoes the world-to-eye transform on the MODEL_VIEW matrix stack., so the projective texture can be specified in world coordinates. Note: If you specify your lights in eye-space then this matrix is identity.

This matrix specifies the frustum of the projector . It is a non-affine, projection matrix. You can use any of the projection transformations to establish it, such as glFrustum(), glOrtho() or gluPerspective () .

This matrix positions the projector in the world, much like the viewing matrix positions the eye within the world. (HINT, you can use gluLookAt() to set this up if you want.

OpenGL Example

Here is a code fragment implementing projective textures in OpenGL

```
// The following information is associated with the current active texture
// Basically, the first group of setting says that we will not be supplying texture coordinates.
// Instead, they will be automatically established based on the vertex coordinates in “EYE-SPACE”
// (after application of the MODEL_VIEW matrix).

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);

// These calls initialize the TEXTURE_MAPPING function to identity. We will be using
// the Texture matrix stack to establish this mapping indirectly.

float [] eyePlaneS = { 1.0f, 0.0f, 0.0f, 0.0f };
float [] eyePlaneT = { 0.0f, 1.0f, 0.0f, 0.0f };
float [] eyePlaneR = { 0.0f, 0.0f, 1.0f, 0.0f };
float [] eyePlaneQ = { 0.0f, 0.0f, 0.0f, 1.0f };

glTexGenfv(GL_S, GL_EYE_PLANE, eyePlaneS);
glTexGenfv(GL_T, GL_EYE_PLANE, eyePlaneT);
glTexGenfv(GL_R, GL_EYE_PLANE, eyePlaneR);
glTexGenfv(GL_Q, GL_EYE_PLANE, eyePlaneQ);
```

OpenGL Example (cont)

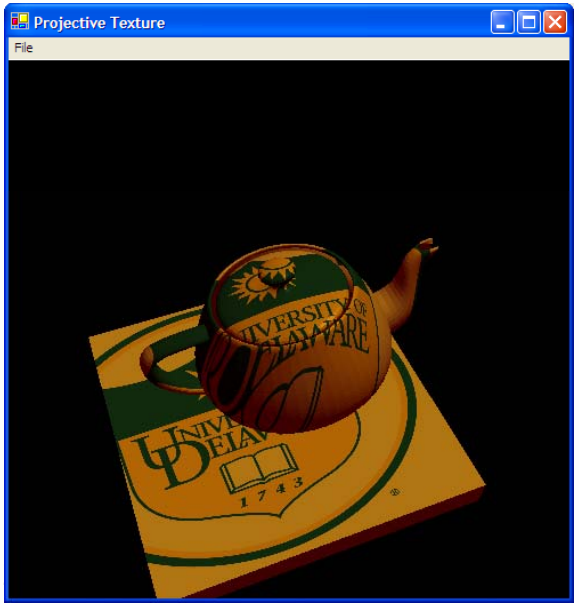
The following code fragment is inserted into Draw( ) or Display( )

```
if (projTexture) {
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    glEnable(GL_TEXTURE_GEN_R);
    glEnable(GL_TEXTURE_GEN_Q);
    projectTexture();
}

// … draw everything that the texture is projected onto

if (projTexture) {
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_Q);
 }
```

OpenGL Example (cont)

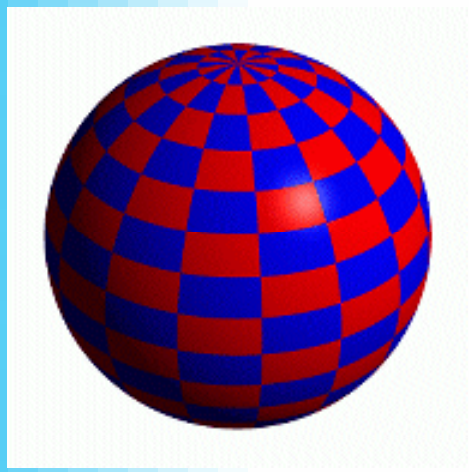
Here is where the extra “Texture” transformation on the vertices is inserted.

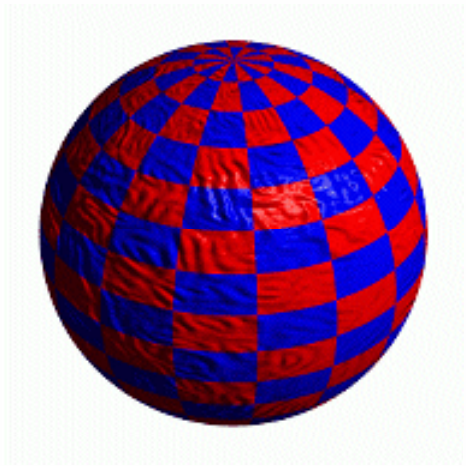
```
private void projectTexture() {

    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();
    glTranslated(0.5, 0.5, 0.5);    // Scale and bias the [-1,1] NDC values
    glScaled(0.5, 0.5, 0.5);        // to the [0,1] range of the texture map
    gluPerspective(15, 1, 5, 7);      // projector "projection" and view matrices
    gluLookAt(lightPosition[0],lightPosition[1],lightPosition[2], 0,0,0, 0,1,0);
    glMatrixMode(GL_MODELVIEW);

}
```



Bump Mapping

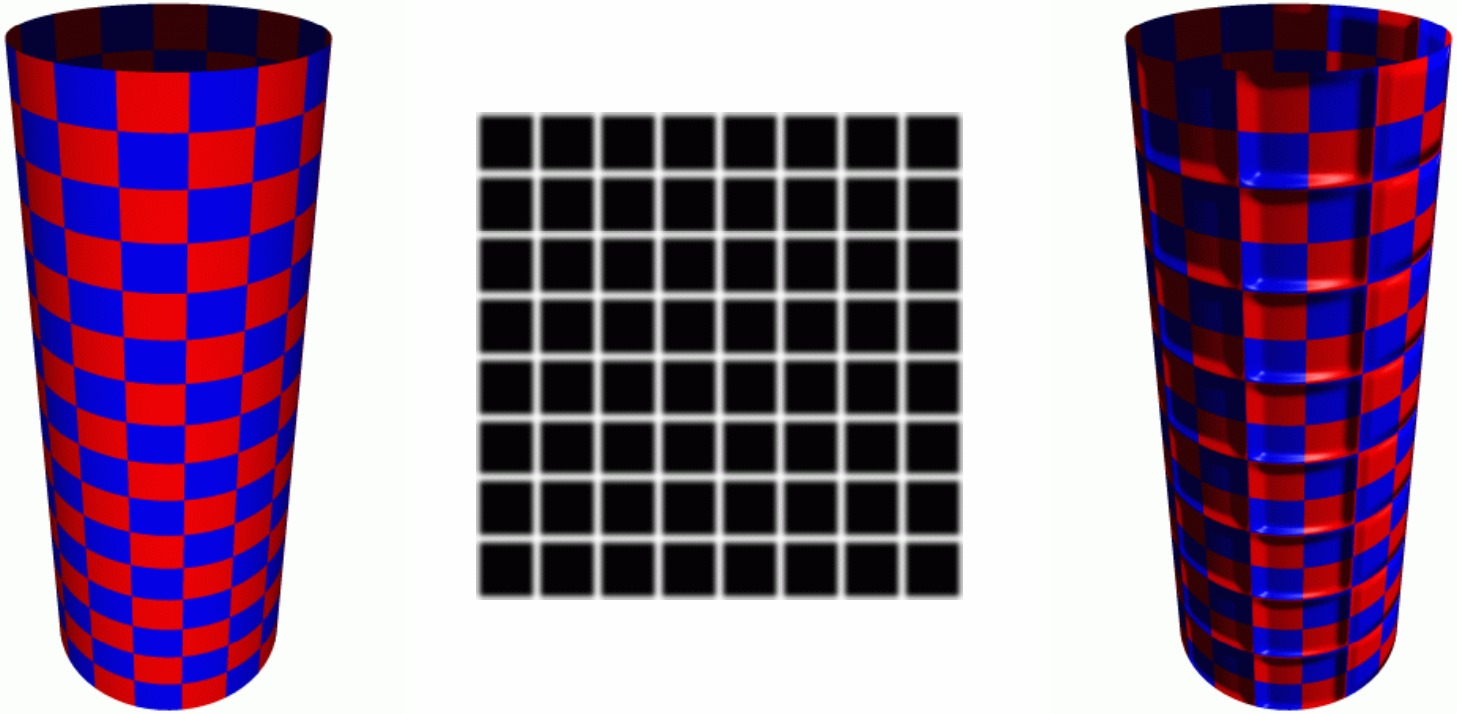
Textures can be used to alter the surface normals of an object. This does not actual shape of the surface -- we are only shading it as if it were a different shape! This technique is called *bump mapping*. The texture map is treated as a single-valued height function. The value of the function is not actually used, just its partial derivatives. The partial derivatives tell how to alter the true surface normal at each point on the surface to make the object appear as if it were deformed by the height function.



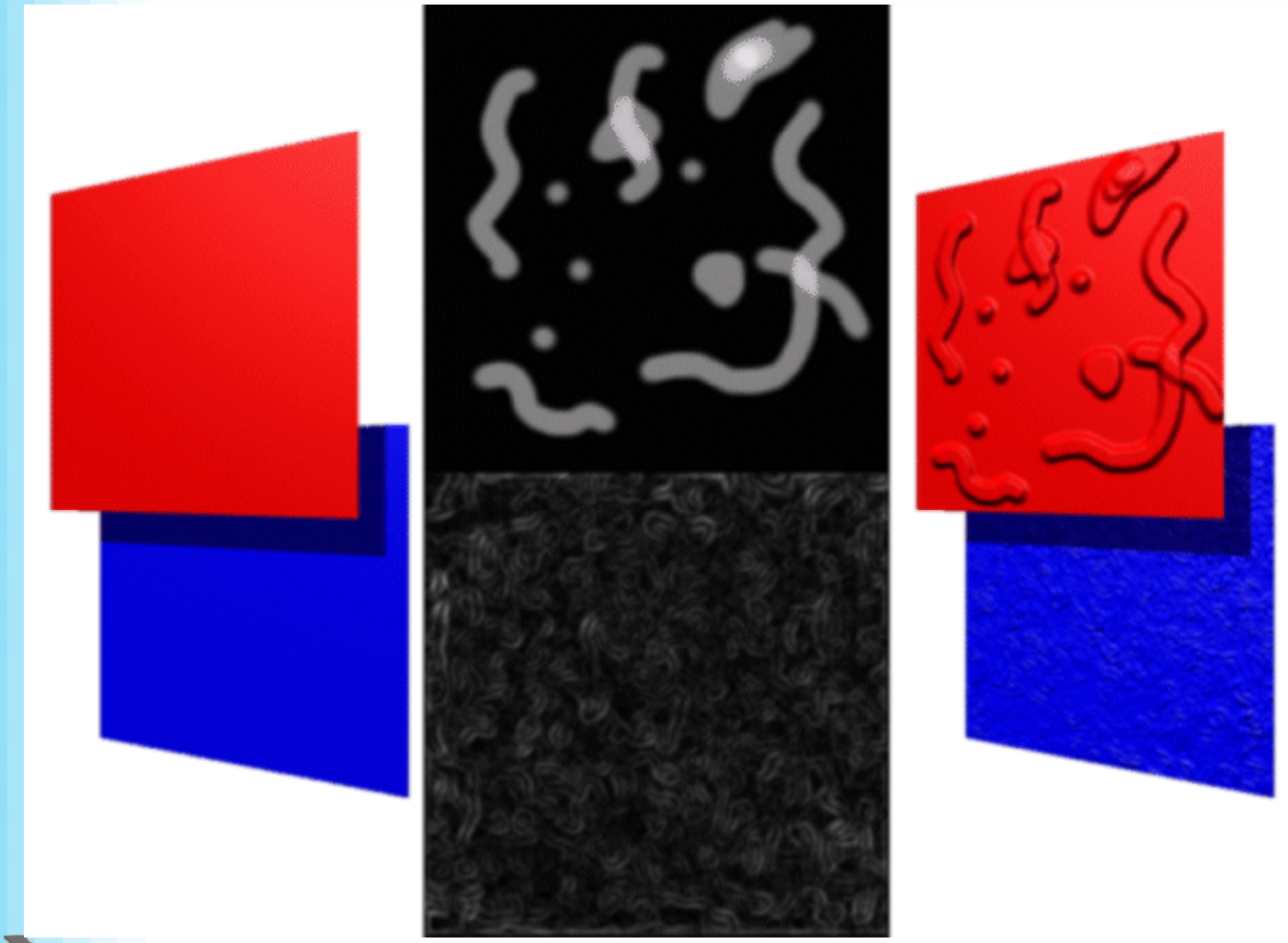


More Bump Map Examples

Since the actual shape of the object does not change, the silhouette edge of the object will not change. Bump Mapping also assumes that the Illumination model is applied at every pixel (as in Phong Shading or ray tracing).

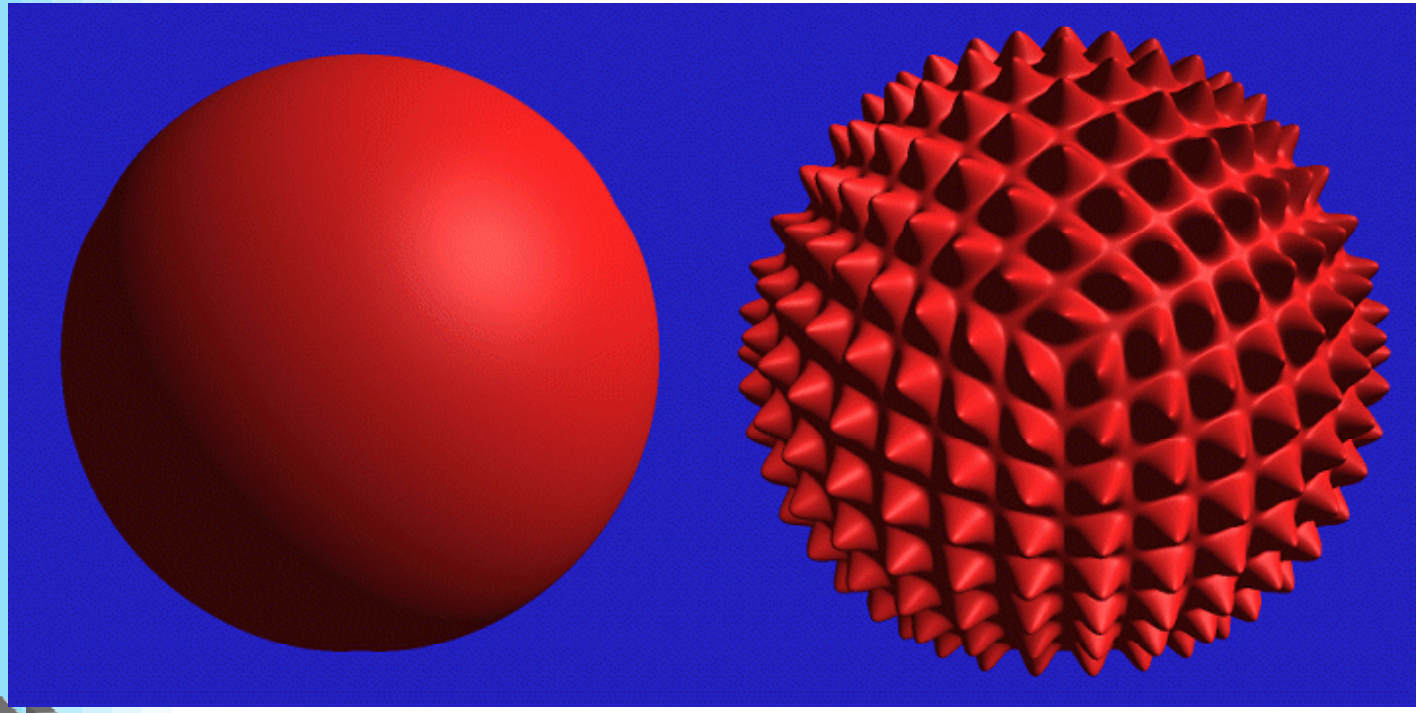


One More Bump Map Example



Displacement Mapping

Texture maps can be used to actually move surface points. This is called *displacement mapping*. How is this fundamentally different than bump mapping?

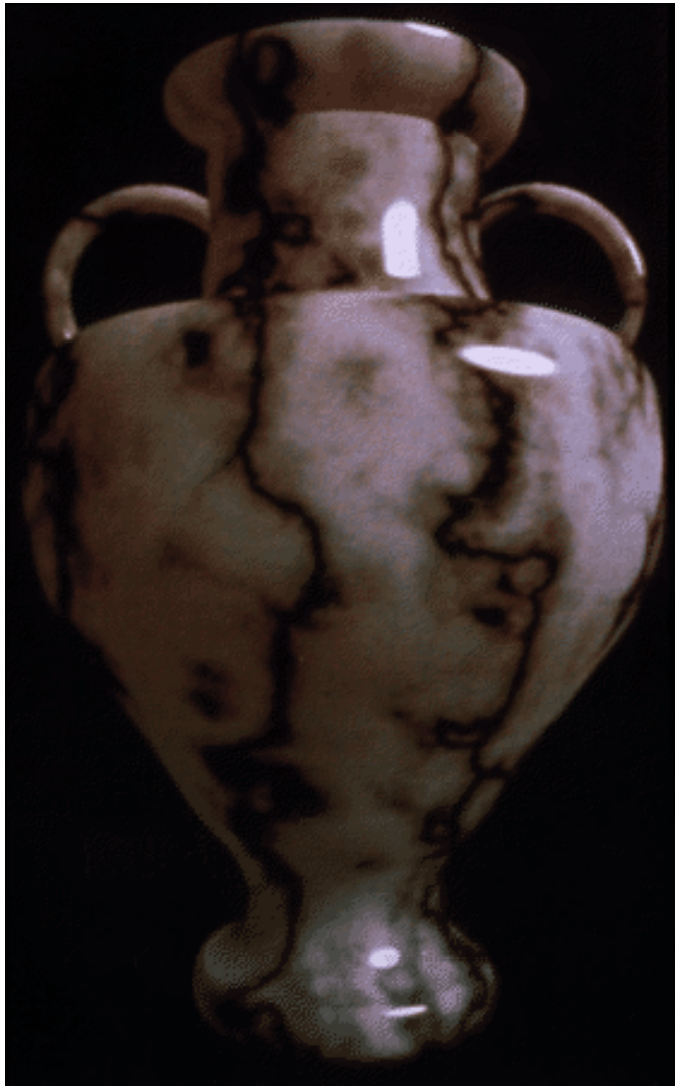


Three Dimensional or Solid Textures

The textures that we have discussed to this point are two-dimensional functions mapped onto two-dimensional surfaces. Another approach is to consider a texture as a function defined over a three-dimensional surface. Textures of this type are called *solid textures*.

Solid textures are very effective at representing some types of materials such as marble and wood. Generally, solid textures are defined procedural functions rather than tabularized or sampled functions as used in 2-D

The approach that we will explore is based on *An Image Synthesizer*, by Ken Perlin, SIGGRAPH '85. The vase to the right is from this paper.



Multi-Texturing

- Shaders will often have uses for multiple textures, how do you bind them?
- How do you specify multiple texture coordinates for a vertex?

Fragment Program with Multiple Textures

```
uniform sampler2D dayTexture;
uniform sampler2D nightTexture;

void main ()
{
    ...
    vec4 dayColor = texture2D (dayTexture, gl_TexCoord[0].xy)
    ...
}
```

Multi-Texture: OpenGL

```
// load two textures
GLuint texture0 = glLoadTexture ("tex0.jpg");
GLuint texture1 = glLoadTexture ("tex1.jpg");

// bind and enable texture unit 0
glActiveTexture (GL_TEXTURE0);
glBindTexture (GL_TEXTURE_2D, texture0);
glEnable (GL_TEXTURE_2D);

// bind and enable texture unit 1
glActiveTexture (GL_TEXTURE1);
glBindTexture (GL_TEXTURE_2D, texture1);
glEnable (GL_TEXTURE_2D);

// specify two sets of texture coordinates for each vertex
glBegin (GL_TRIANGLES);
    glMultiTexCoord2f (GL_TEXTURE0, 0.0, 1.0);
    glMultiTexCoord2f (GL_TEXTURE1, 1.0, 0.0);
    glVertex3f (...)
    ...
glEnd ();
```

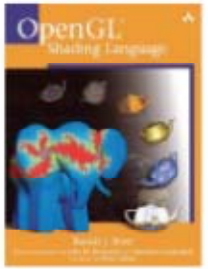
Textures in GLSL

- Textures are just uniforms of type sampler2D
- Tell GLSL that you want one of these sampler2Ds to be GL_TEXTURE*i* by setting the corresponding uniform to i

```
uniform sampler2D dayTexture;
uniform sampler2D nightTexture;
```

Night and Day

Idea Stolen From



Night and Day

- Use one texture for day, one for night
- Use the same texture coordinates for both texture
- Components
  - Vertex program
  - Fragment program
  - OpenGL application

Night and Day Vertex Program

```
/* this vector will store the normal in eye coordinates */
varying vec3 normal;

void main ()
{
    /* store the transformed normal in normal */
    normal = gl_NormalMatrix * gl_Normal;

    /* pass texture coordinate to the fragment program */
    gl_TexCoord[0] = glMultiTexCoord0;

    /* use the standard OpenGL transformation matrix */
    gl_Position = ftransform ();
}
```

Night and Day Fragment Program

```
/* this vector will store the normal in eye coordinates */
varying vec3 normal;

/* the fraction of overlap between textures */
uniform float overlap;

/* uniform texture map for the day's texture */
uniform sampler2D dayTexture;

/* uniform texture map for the night's */
uniform sampler2D nightTexture;

void main ()
{
    /* make sure we have unit normals while interpolating */
    vec3 N = normalize (normal);

    /* unit light direction, (assume) stored in eye coordinates */
    vec3 L = normalize (gl_LightSource[0].position.xyz);
    ...
```

Night and Day

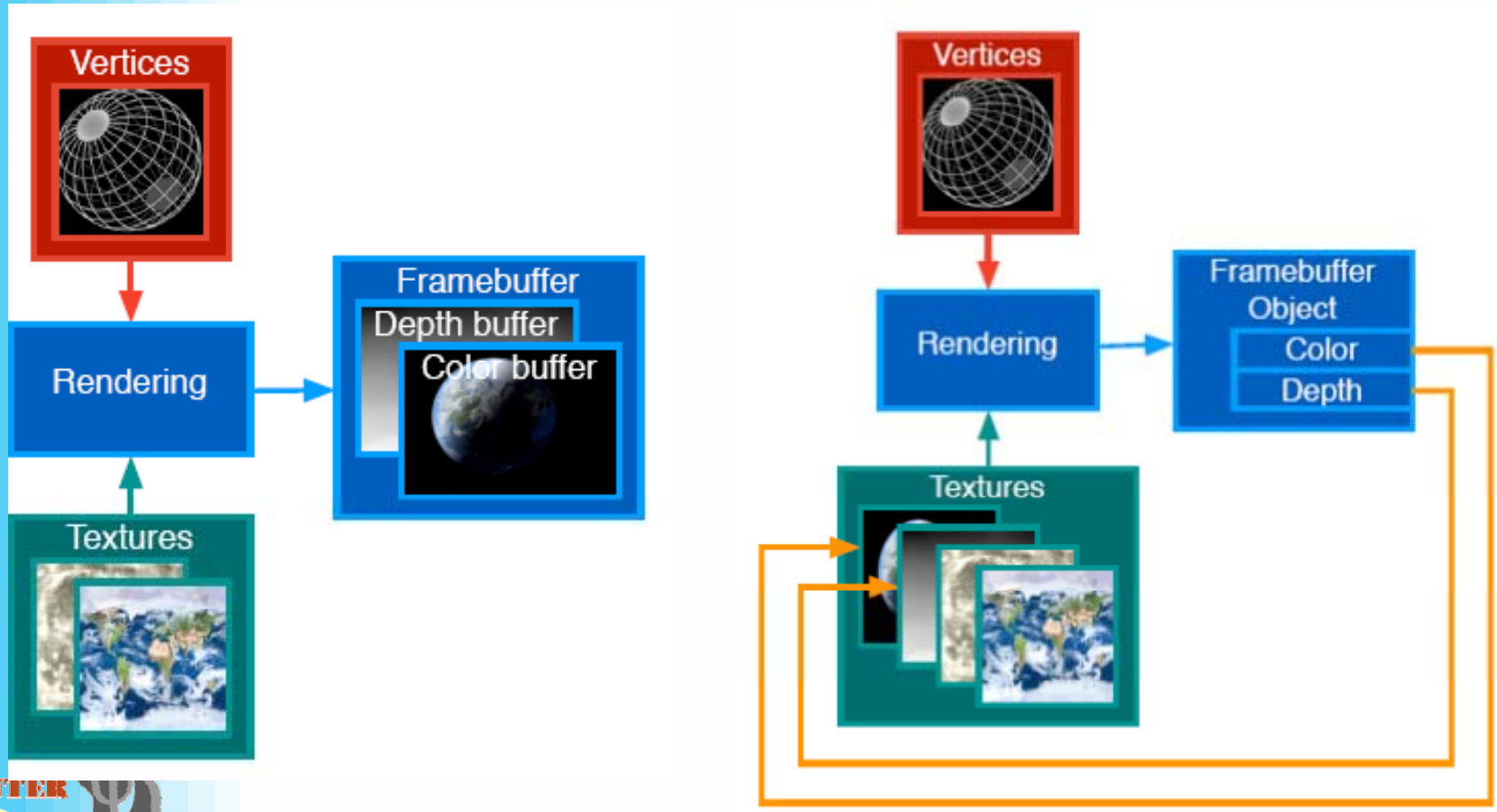
```
    float NdotL =  dot (N, L);

    /* calculate the and night's weights */
    vec4 dayWeight =
        gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse *
        max (NdotL + overlap, 0.0) / (1.0 + overlap);
    vec4 nightWeight = max (overlap - dayWeight, 0.0) / overlap;

    /* sample day's color */
    vec4 dayColor = texture2D (dayTexture, gl_TexCoord[0].xy);

    /* sample explosion color */
    vec4 nightColor = texture2D (nightTexture, gl_TexCoord[0].xy);

    /* set output color to the weighted combination */
    gl_FragColor =
              dayColor    * dayWeight +
        2.5 * nightColor *  nightWeight;

    /* make alpha always 1 */
    gl_FragColor.w = 1.0;
}
```

Night and Day OpenGL Program

```
// bind and enable texture unit 0, set it to be the day's texture map
glActiveTexture (GL_TEXTURE0);
glBindTexture (GL_TEXTURE_2D, dayTexture);
glEnable (GL_TEXTURE_2D);

// tell the program that earthTexture is in GL_TEXTURE0
GLuint dayLoc = glGetUniformLocation (program, "dayTexture");
glUniformi (dayLoc, 0);

// bind and enable texture unit 1, set it to be night's texture map
glActiveTexture (GL_TEXTURE1);
glBindTexture (GL_TEXTURE_2D, nightTexture);
glEnable (GL_TEXTURE_2D);

// tell the program that nightTexture is in GL_TEXTURE1
GLuint nightLoc = glGetUniformLocation (program, "nightTexture");
glUniformi (nightLoc, 1);

// draw the earth
TexturedSphere ();
```

Render to Texture

- GL_EXT_framebuffer_object
- Render intermediate result into a texture, and then render using the texture



Render to Texture

```
/* create a framebuffer object */
GLuint fb;
glGenFramebuffersEXT (1, &fb);
glBindFramebufferEXT (GL_FRAMEBUFFER_EXT, fb);

/* attach a texture (assume tex is an allocated texture)
glFramebufferTexture2DEXT(
    GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
    GL_TEXTURE_2D, texture, 0);
```

```
/* Switch back to drawing to screen */
glBindFramebufferEXT (GL_FRAMEBUFFER_EXT, fb);

// draw stuff into texture

/* Switch back to drawing to screen */
glBindFramebufferEXT (GL_FRAMEBUFFER_EXT, 0);
```