11. Texture Mapping

Can we do these?
11. Texture Mapping

Instead of taking an enormous amount of effort to model every detail of every 3D shape in your scene, we can create the illusion of detail:

- take a photograph of the “real thing”
- paste that photo onto simple 3D geometry
- increase realism without increasing the amount of geometry to draw
11. Texture Mapping

Texture mapping is the process of
- taking a 2D image (texture; in general, a pre-defined multi-dimensional table) and mapping onto a polygon in the scene
11. Texture Mapping

This texture acts like a painting (or, wall paper), adding 2D detail to the 2D polygon.

Instead of filling a polygon with a color in the scan conversion process, fill the pixels of the polygon with the pixels of the texture (texels).
11. Texture Mapping

Used to:
- add detail
- add 'roughness'
- add patterns

texture space

object space

image space
**Example**: an increasingly complex sphere texture mapped with the following image of Mars.
Photo Textures

(Slide Courtesy of Leonard McMillan & Jovan Popovic, MIT)

The concept is very simple!

For each triangle in the model establish a corresponding region in the phototexture.

During rasterization interpolate the coordinate indices into the texture map.
Textures

- Textures may be either digitized or synthesized.

- Unlike bitmaps and stipple patterns, textures rotate, translate, and scale.

- Textures may be applied to all primitives.

- Texture mapping is computationally expensive (specialized hardware support is usually necessary).
Textures

Texture is also referred to as *texture map*

- Texture maps are composed of *discrete elements* called *texels*

- Each *texel* contains information describing the texture at the corresponding point (*color, luminance, color+alpha*)

- The information can be stored in different *formats*
  - 1, 2, 3 or 4 elements per texel
  - RGBA color, modulation constant, etc.
Textures

- Texture maps can be thought of as a **rectangular array of texels**
- The array lies in **texture coordinate space**
- Texture coordinate components are referred to as $s$, $t$, $r$, and $q$
- One texel **does not necessarily** correspond to one pixel in the final image
Texture Files under Win32

Stored in device independent bitmap (DIB) files

- `.bmp` extension

- Two headers:
  BITMAPFILEHEADER
  - BM signature; file size; bitmap offset
  Second Header
  - depth of color array; compression method

- Color formats: 1, 4, 8, 16, 24, or 32 bits per texel

- DIB color triples are stored in blue-green-red order
Texture Mapping in OpenGL

Upload texture (raw RGB image data) to video memory:
- some setup must take place first
- same calls must be made for each texture

Apply texture onto geometry:
- enable texturing
- bind a texture before glBegin/glEnd
- specify a texture coordinate before each vertex of a polygon
Uploading Textures

`glBindTexture` : sets texture “id”

`glPixelStorei` : tells OpenGL how data is aligned

`glTexParameteri` : sets the various parameters

`glTexEnvf` : sets environment variables

`glTexImage2D` : uploads texture to video memory

Tells OpenGL which texture “id” we will be working with
Uploading Textures

```c
GLenum texTarget = GL_TEXTURE_2D;
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, imageWidth, imageHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, imageData);
```
glTexImage2D

C Specification

```c
void glTexImage2D (GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid * data);
```
**Target:**
Specifies the *target texture*.

**Level:**
Specifies the level-of-detail *number*.  
Level 0 is the base image level.  
Level n is the n-th *mipmap* reduction image.

*For now*
**internalFormat:**

*internal component parameter.*

*This tells OpenGL how many color components to represent internally from the texture that is uploaded.*

*There are many symbolic constants for this parameter but the one which is most widely used is GL_RGB; this constant is equal to 3.*
width & height:
Width and of the image data. These must be integers that are equal to $2^{**n+2}$ (border) for some integer $n$. This means the texture width and height must be a power of 2 ($2, 4, 8, 16, 32, 64, 128, 256, 512, \text{etc}$)
**border:**

*Image border, must be 0 or 1. Use 0 in your code if you do not use image borders.*

**format:**

*Format of the pixel data that will be uploaded. GL_RGB is the value that is widely used.*
**type:**
*Type of data that will be uploaded.*
*A good choice is* GL_UNSIGNED_BYTE.

**pixels:**
*Pointer to the image data to be uploaded to the video memory.*
*This memory can be freed once glTexImage2D is called since the texture is already uploaded into video memory.*
Applying Textures

Enable texturing:

with `glEnable(GL_TEXTURE_2D)` call

Bind a texture before `glBegin/glEnd`:

`glBindTexture(GL_TEXTURE_2D, 13);`
`glBegin(GL_QUADS);`
...
`glEnd();`
Applying Textures

Specify a **texture coordinate** before each vertex of a polygon:

```c
glBegin(GL_Quads);
glTexCoord2f(0.0, 0.0);
glVertex3f(0.0, 0.0, 0.0);
glTexCoord2f(1.0, 0.0);
glVertex3f(10.0, 0.0, 0.0);
```

```c
  glTexCoord2f(1.0, 1.0);
  glVertex3f(10.0, 10.0, 0.0);
  glTexCoord2f(0.0, 1.0);
  glVertex3f(0.0, 10.0, 0.0);
  glEnd( );
```
mipMap

- pre-calculated, optimized collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing artifacts.

- Widely used in 3D computer games, flight simulators and ...
mipmap

- Each bitmap image of the mipmap set is a version of the main texture, but at a certain reduced level of detail.

An example of mipmap image storage: the principal image on the left is accompanied by filtered copies of reduced size.
mipMap

The **main texture** is used when the view is sufficient to render it in full detail.

The renderer will **switch** to a suitable mipmap image (or **interpolate** between the two nearest, if **trilinear filtering** is activated) when the texture is viewed from a **distance** or at a **small size**.
mipMap

- If the texture has a basic size of 256 by 256 pixels, then the associated mipmap set may contain a series of 8 images, each one-fourth the total area of the previous one:
  - 128×128 pixels,
  - 64×64,
  - 32×32,
  - 16×16,
  - 8×8,
  - 4×4,
  - 2×2,
  - 1×1 (a single pixel).
mipMap

- Rendering speed increases since the number of texture pixels ("texels") being processed can be much lower than with simple textures.
- Artifacts are reduced since the mipmap images are effectively already anti-aliased, taking some of the burden off the real-time renderer.
- Scaling down and up is made more efficient with mipmaps as well.
mipMap

- Simplest way to generate these textures: successive averaging

(Courtesy of John Hart)
mipMap

- Can you get the following mipmaps using successive averaging?

- more sophisticated algorithms (based on signal processing and Fourier transforms) should be used to get mipmaps with better quality.
mipMap

- The increase in storage space required for all of these mipmaps is \( \frac{1}{3} \) the original texture
  
  (the sum of the areas \( \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \cdots \) converges to \( \frac{1}{3} \))

- For an RGB image with three channels stored as separate planes, the total mipmap can be visualized as fitting neatly into a square area twice as large as the dimensions of the original image on each side
mipMap

In the case of an RGB image with three channels stored as separate planes, the total mipmap can be visualized as fitting neatly into a square area twice as large as the dimensions of the original image on each side.
Recall that we have to specify a texture coordinate \((U, V)\) before each vertex of a polygon:

```cpp
glBegin(GL_Quads);
// Set texture coordinates
glTexCoord2f(0.0, 0.0);
glVertex3f(0.0, 0.0, 0.0);

// Repeat for all vertices
```

How? through UV-mapping
UV-Mapping

• Each Texel is associated with a \((u,v)\) 2D texture coordinate
• The range of \(u, v\) is \([0.0,1.0]\) due to normalization
UV-Mapping

The UV Mapping process at its simplest requires three steps:

- unwrapping the mesh,
- building the correspondence, and
- applying the texture.
UV-Mapping

- When creating a **polygon mesh**, generating UV coordinates for each vertex in the mesh.
- One way is for the 3D modeler to **unfold** the triangle mesh at the seams, laying out the triangles on a flat page.
- Then **paint** a texture on each triangle individually, using the unwrapped mesh as a **template**.
- When the scene is rendered, **each triangle will map** to the appropriate **texture** from the "decal sheet".
UV-Mapping

An example Of unwrapping:

Basically, a 3D to 2D mapping technique

How?

A representation of UV mapping of a cube. The flattened cube net may then be textured to Texture the cube.
What happens when outside the 0~1 range?

- (u,v) should be in the range of 0~1
- What happens when you request (1.4, 2.3)?
  - **Tile**: repeat (OGL); the integer part of the value is dropped, and the image repeats itself across the surface
  - **Mirror**: the image repeats itself but is mirrored (flipped) on every other repetition
  - **Clamp to edge**: values outside of the range are clamped to this range
  - **Clamp to border**: all those outside are rendered with a separately defined color of the border

Horizontally, (1.4, 2.3) -> (0.0, 0.3) -> (0.6, 0.3)
Vertically, (1.4, 2.3) -> (0.4, 0.0) -> (0.4, 0.7)
What happens when outside the 0~1 range?
Methods for modifying surface

- After a texture value is retrieved (may be further transformed), the resulting values are used to modify one or more surface attributes
- Called *combine functions* or *texture blending operations*
  - Replace (Decal): replace surface color with texture color
Methods for modifying surface

- **Modulate**: multiply the surface color by the texture color (shaded + textured surface). Need this for multi-texturing (i.e., lightmaps).
Methods for modifying surface

- **Blend**: similar to modulation, but add alpha texture value

  (the alpha component in the framebuffer is not modified)
Methods for modifying surface

- texture blending operations
Okay, then how can you implement?

(Slide Courtesy of Leonard McMillan & Jovan Popovic, MIT)

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:
(u,v) tuple

- For any (u,v) in the range of (0-1, 0-1) multiplied by texture image width and height, we can find the corresponding value in the texture map.
How do we get $F(u,v)$?

- We are given a discrete set of values:
  - $F[i,j]$ for $i=0,...,N$, $j=0,...,M$

- Nearest neighbor:
  - $F(u,v) = F[\text{round}(N*u), \text{round}(M*v)]$

- Linear Interpolation:
  - $i = \text{floor}(N*u)$, $j = \text{floor}(M*v)$
  - Interpolate from $F[i,j]$, $F[i+1,j]$, $F[i,j+1]$, $F[i+1,j+1]$

- Filtering in general!
  Bilinear, trilinear, and anisotropiopic filtering
Interpolation

Nearest Neighbor

Bilinear Interpolation
Filtering Textures

Footprint changes from pixel to pixel: no single filter resampling theory:
  - Magnification: Interpolation
  - Minification: Averaging

We would like a constant cost per pixel
Texture Coordinates

- Specify a texture coordinate at each vertex \((s, t)\) or \((u, v)\)
- Canonical coordinates where \(u\) and \(v\) are between 0 and 1
- Simple modifications to triangle rasterizer

```
Void EdgeRec : : init() {
  ...
  // Note: here we use \(w=1/z\)
  wstart=1./z1;  wend=1./z2;  dw=wend-wstart;
  wcurr=wstart;
  sstart=s1;  send=s2;  ds=send-sstart;
  scurr=sstart;
  tstart=t1;  tend=t2;  dt=tend-tstart;
  tcurr=tstart;
}

Void EdgeRec : : update( ) {
  ycurr+=1;  xcurr+=dx;  wcurr+=dw;
  scurr+=ds;  tcurr+=dt;
}
```
static void RenderScanLine ( ... ) {

    ... 
    for (e1 = AEL->ToFront ( ) ; e1 != NULL; e1 = AEL->Next() ) {
        e2=AEL->NextPolyEdge(e1->poly);
        x1=[e1->xcurr];  x2=[e2->xcurr];  dx=x2-x1;
        w1=e1->wcurr;    w2=e2->wcurr;  dw=(w2-w1)/dx;
        s1=e1->scurr;     s2=e2->scurr;    ds=(s2-s1)/dx;
        t1=e1->tcurr;       t2=e2->tcurr;     dt=(t2-t1)/dx;
        for (int x=x1;  x<x2;  x++)  {
            w+=dw;
            s+=ds;   t+=dt;
            if  ( w<wbuffer[x] ) {
                wbuffer[x]=w;
                raster.setPixel(x,  texturemap[s, t])
            }
        }
    }
}

raster->write(y);
Was it working correctly?

Let's assume that the viewport is located 1 unit away from the center of projection.

So, \( d = 1 \)
No perspective correction  Perspective correction
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 interpolate in 3-space

\[
\begin{bmatrix}
x \\
z
\end{bmatrix} = \begin{bmatrix} x_1 \\ z_1 \end{bmatrix} + s \left( \begin{bmatrix} x_2 \\ z_2 \end{bmatrix} - \begin{bmatrix} x_1 \\ z_1 \end{bmatrix} \right)
\]

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

$$s = \frac{t \frac{z_1}{z_2}}{z_2 + t(\frac{z_1}{z_2} - 1)}$$

Unfortunately, at this point in the pipeline (after projection) we no longer have $z_1$ lingering around (Why?). However, we do have $\omega_1 = 1/z_1$ and $\omega_2 = 1/z_2$.

$$s = \frac{t}{\frac{1}{\omega_1} + t\left(\frac{1}{\omega_1} - \frac{1}{\omega_2}\right)} = \frac{t \omega_2}{\omega_1 + t(\omega_2 - \omega_1)}$$
Perspective Texturing

- The most common form of perspective texturing is done via a divide by Z.
- Instead of interpolating U and V, we interpolate U/Z and V/Z. 1/Z is also interpolated.
- At each pixel, we take our texture coords, and divide them by Z. Note that Z is also interpolated, so we're not dividing by the same Z twice. We then take the new U and V values, index into our texture map, and plot the pixel.
Perspective Texturing

Pseudo-code might be:

\[
\begin{align*}
su &= \text{Screen-}U = U/Z \\
sv &= \text{Screen-}V = V/Z \\
sz &= \text{Screen-}Z = 1/Z \\
\text{for } x = \text{startx to endx} \\
&\quad u = su / sz \\
&\quad v = sv / sz \\
&\quad \text{PutPixel}(x, y, \text{texture}[v][u]) \\
&\quad su += \text{deltasu} \\
&\quad sv += \text{deltasv} \\
&\quad sz += \text{deltasz}
\end{align*}
\]

Very simple and very slow!
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 of $s$ for the given $t$ into the parameter interpolation.

\[
\begin{align*}
    u &= u_1 + s(u_2 - u_1) \\
    u &= u_1 + \frac{t w_2}{w_1 + t (w_2 - w_1)} (u_2 - u_1) = \frac{u_1 w_1 + t (u_2 w_2 - u_1 w_1)}{w_1 + t (w_2 - w_1)}
\end{align*}
\]
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 of $s$ for the given $t$ into the parameter interpolation.

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

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

Therefore if we pre-multiply all parameters that we wish to interpolate in 3-space by their corresponding w values and add a new place equation to interpolate the w values themselves, we can interpolate the numerators and the denominators in the screen space.

We then need to perform a divide at each step to map the screen space interpolants to their corresponding 3-space values.

This is a simple modification to the triangle rasterizer.
Void EdgeRec : : init( ) {
    ...  // Note: here we use w=1/z
    wstart=1./z1;  wend=1./z2;  dw=wend-wstart;
    wcurr=wstart;
    swstart=s1*w1;  send=s2*w2;
    dsw=swstart-swend;  swcurr=swstart;
    twstart=t1*w1;  twend=t2*w2;
    dtw=twstart-twend;  twcurr=twstart;
}

Void EdgeRec : : update( ) {
    ycurr+=1;  xcurr+=dx;  wcurr+=dw;
    swcurr+=dsw;  twcurr+=dtw;
}
static void RenderScanLine ( ... ) {
    ...
    for (e1 = AEL->ToFront ( ) ; e1 != NULL; e1 = AEL->Next() ) {
        e2=AEL->NextPolyEdge(e1->poly);
        x1=[e1->xcurr];  x2=[e2->xcurr];  dx=x2-x1;
        w1=e1->xcurr;    w2=e2->wcurr;  dw=(w2-w1)/dx;
        sw1=e1->swcurr;  sw2=e2->swcurr;  dsw=(sw2-sw1)/dx;
        tw1=e1->twcurr;  tw2=e2->twcurr;  dtw=(tw2-tw1)/dx;
        for (int x=x1;  x<x2;  x++) {
            w+=dw;
            float denom = 1.0f / w;
            sw+=dsw;  tw+=dtw;
            correct_s = sw*denom;  correct_t = tw*denom;
            if ( w<wbuffer[x] ) {
                wbuffer[x]=w;
                raster.setPixel(x,  texturemap[correct_s, correct_t])
            }  
        }
    }
    raster->write(y);
}
Useful links (Google - perspective correct texture)

- http://www.whisqu.se/per/docs/graphics16.htm
- http://easyweb.easynet.co.uk/~mrmeanie/tmap/tmap.htm
End of 11-1

Slide/photo Courtesy of
Leonard McMillan & Jovan Popovic, MIT
John Hart, UIUC
Ilmi Yoon, SFST
Taku Komura, Edinburgh University