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
- increases realism without increasing the amount 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

This texture acts like a painting, 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
**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.
OpenGL Functions - demo

● During initialization read in or create the texture image and place it into the OpenGL state.

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

● Before rendering your textured object, enable texture mapping and tell the system to use this particular texture.

```c
glBindTexture(GL_TEXTURE_2D, 13);
```
glTexImage2D

**Name**

gTexImage2D — specify a two-dimensional texture image

**C Specification**

```c
void glTexImage2D(GLenum target, GLint level, GLsizei width, GLsizei height, GLenum format, GLenum type, const void *data);
```
Target:

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

**InternalFormat:**
Specifies the internal format of the texture.
Must be one of: GL_ALPHA, GL_LUMINANCE,
GL_LUMINANCE_ALPHA, GL_RGB, GL_RGBA.
Width:
Specifies the width of the texture image. At least 64 texels wide for 2D texture images and 16 texels for cube-mapped texture images.

Height:
Specifies the height of the texture image. At least 64 texels wide for 2D texture images and 16 texels for cube-mapped texture images.
Cube-mapped texture images:
border: Specifies the width of the border. Must be 0.

format: Specifies the format of the texel data (how the color is represented in RAM). Note that BMP does not store RED->GREEN->BLUE, but BLUE->GREEN->RED
**type:**
Specifies the data type of the texel data, such as: GL_UNSIGNED_BYTE,
GL_UNSIGNED_SHORT_5_6_5,
GL_UNSIGNED_SHORT_4_4_4_4, ...

**data:**
Specifies a pointer to the image data in memory.
mipMap

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

Acronym of the Latin phrase *multum in parvo*, meaning “much in little”.

Main texture

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

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

- main texture is used when the view is sufficient to render it in full detail
- the renderer switches to a suitable mipmap image when the texture is viewed from a distance or at a small size.
- interpolate between the two nearest, if trilinear filtering is activated
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

- These textures can be generated by successive averaging
- more sophisticated algorithms (using signal processing or Fourier transforms) can be used
mipMap

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

- 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.
mipMap

The original RGB image

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.
Texture and Texel

- Each pixel in a texture map is called a **Texel**
- Each Texel is associated with a \((u,v)\) 2D texture coordinate
- The range of \(u\), \(v\) is \([0.0,1.0]\) due to normalization
OpenGL Functions - demo

- During rendering, give the **cartesian coordinates** and the **texture coordinates** for each vertex.

```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);
    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 ();
```
UV-Mapping

When a model is created as a polygon mesh using a 3D modeler, UV coordinates can be generated for each vertex in the mesh. One way is for the 3D modeler to unfold the triangle mesh at the seams, automatically laying out the triangles on a flat page. If the mesh is a UV sphere, for example, the modeler might transform it into an equirectangular projection. Once the model is unwrapped, the artist can 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".

e.g., \((U, V) = (g, t)\)
\[g : \text{latitude} \quad t : \text{longitude}\]
\[(X, Y, Z) = (\cos(t)\cos(g), \sin(t), \sin(g))\]
UV-Mapping

e.g., (U, V) = (g, t)
g: longitude  \ t: latitude

(X, Y, Z) = (\cos(t)\cos(g), \sin(t), \sin(g))
UV-Mapping

An example Of unwrapping:

Basically, a 3D to 2D mapping technique

A representation of UV mapping of a cube. The flattened cube net may then be textured to Texture the cube.
UV-Mapping

The UV Mapping process at its simplest requires three steps:

- unwrapping the mesh,
- building the correspondence, and
- applying the texture.
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, \((1.4, 2.3) \rightarrow (0.4, 0.3) \rightarrow (0.6, 0.3)\)
Vertically, \((1.4, 2.3) \rightarrow (0.4, 0.3) \rightarrow (0.4, 0.7)\)
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**: replace surface color with texture color
  - **Decal**: replace surface color with texture color, blend the color with underlying color with an alpha texture value, but the alpha component in the framebuffer is not modified
  - **Modulate**: multiply the surface color by the texture color (shaded + textured surface)
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!
Interpolation

Nearest Neighbor

Linear 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
MipMapping [Williams] again:

MIP = Multim In Parvo = Many things in a small place

Trilinear interpolation
MipMapping – Another Example

(Courtesy of John Hart)

or, successive averaging
Texture Coordinates

Void EdgeRec::init() {
    wstart=1./z1; wend=1./z2; dw=wend-wstart; wcurn=wstart;
    sstart=s1; send=s2; ds=sstart-send; scurr=sstart;
    tstart=t1; tend=t2; dt=tstart-tend; tcurn=tstart;
}

Void EdgeRec::update () {
    ycurr+=1; xcurr+=dx; wcurn+=dw;
    scurr+=ds; tcurn+=dt;

- Specify a texture coordinate at each vertex (s, t) or (u, v)
- Canonical coordinates where u and v are between 0 and 1
Texture Coordinates

```java
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);
}
```

- Simple modifications to triangle rasterizer
Was it working correctly?

Let's assume that the viewport is located 1 unit away from the center of projection.
No perspective correction

Perspective correction
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 interpolate in 3-space and then do projection

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

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

$$\frac{x_1}{z_1} + t \left( \frac{x_2 - x_1}{z_2} \right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$
How to Make Them Mesh

and solve $s$ in terms of $t$ giving:

$$s = \frac{t}{z_2} - \frac{1}{z_1} + t \left( \frac{1}{z_2} - \frac{1}{z_1} \right) = \frac{t z_1}{z_2 + t(z_1 - z_2)}$$
How to Make Them Mesh

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

$$s = \frac{t/w_1}{1/w_2 + t(1/w_1 - 1/w_2)}$$

$$= \frac{tw_2}{w_1 + t(w_2 - w_1)}$$
The most common form of perspective texturing is done via a divide by Z.

Instead of interpolating U and V, we interpolate \( \frac{U}{Z} \) and \( \frac{V}{Z} \). \( \frac{1}{Z} \) 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:

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

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.
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.
Texture Coordinates

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

Void EdgeRec::update() {
    ycurr+=1; xcurr+=dx; wcurr+=dw;
    swcurr+=ds; 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->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);
}
Texture Coordinates

static void RenderScanLine ( ...) {
    ...
    e2=AEL->NextPolyEdge(e1->poly);
    x1=[e1->xcurr]; x2=[e2->xcurr]; dx=x2-x1;
    w1=e1->wcurr; 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->write(y);
    }

Are correct_s and correct_t integer?
Useful links (Google - perspective correct texture)

- [http://www.whisqu.se/per/docs/graphics16.htm](http://www.whisqu.se/per/docs/graphics16.htm) (Perspective correction with Z-buffering)
- [http://easyweb.easynet.co.uk/~mrmeanie/tmap/tmap.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