Elements of Realistic Rendering

Realism in Rendering:
- Perspective projection
- Illumination and shading
  - Texture mapping and surface detail
  - Hidden surface removal
  - Color
  - Reflection and other effects

Overview:

- Texture mapping
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures and Perlin noise

Texture and Surface Detail

We have seen how to provide color to objects using:
- Solid colors: using glColor().
- Lighting and shading: through various lighting and shading models.

Today we consider how to add realism through surface detail.
Examples:
- Natural surfaces: stone, wood, gravel, grass.
- Printing and painting: printed labels, billboards, newspapers.
- Clothing and fabric: woven and printed patterns, upholstery.

Image Texturing

Image courtesy, Foley, van Dam, Feiner, Hughes

Overview

- The texture-mapping process
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
- Perlin noise

Texture and Surface Detail

Texture Mapping:
  - Conceptually, a texture is an image pasted on the surface of an object.

Textures as Micro-Geometry:
- Textures provide a way to model repetitive and high-resolution color and geometric features, without complex geometry.

Generalizations: There are elements other than color that can be mapped onto a surface.
- Surface normals (used for lighting computations)
- Bumps and surface displacements
- Environments (used to "fake" reflective surfaces)
- Shadows, …
Consider a point with coordinates \((x, y, z)\) on this sphere.

Then:

\[
\begin{align*}
\theta &= \arctan(y/x)/2, \\
\phi &= \arcsin(z/r). \\
\end{align*}
\]

These encode something like the longitude and latitude of the point.

Inverse Wrapping Function: Assuming that texture coordinate range is \(0 \leq s \leq 1\), let

\[
\begin{align*}
s(x,y,z) &= \theta(x,y,z)/2 - (\arctan(y/x))/2, \\
t(x,y,z) &= \phi(x,y,z)/\pi - (\arctan(y/x))/\pi; \\
\end{align*}
\]

Example: Texture Mapping a Sphere

**Example:** Mapping a cylinder. Assume that \((s, t)\) space has been normalized to \(0 \leq s, t \leq 1\).

**Texture Mapping a Cylinder (cont)**

**Parametric Equation of a Cylinder:** of radius \(r\), with base centered at the origin, and height \(H\).

\[
\begin{align*}
x &= r \cos \theta, & 0 \leq \theta \leq 2\pi, \\
y &= r \sin \theta, & 0 \leq \theta \leq 2\pi, \\
z &= h, & 0 \leq \theta \leq 2\pi. \\
\end{align*}
\]

**Surface Parameterization:** \((g, h)\), where \(g(x,y,z) = \arctan(y/x), \quad 0 \leq x \leq 2\pi, \quad h(x,y,z) = z, \quad 0 \leq z \leq H\).

**Inverse wrapping function:** Assuming that texture coordinate range is \(0 \leq g, h \leq 1\), let:

\[
\begin{align*}
s(x,y,z) &= g(x,y,z)/\pi - (\arctan(y/x))/\pi, \\
t(x,y,z) &= h(x,y,z)/H - (z/H). \\
\end{align*}
\]
General Texture Mapping

Overview

- The texture-mapping process
  - Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
- Perlin noise

Create Texture Object(s)

Create Texture Object(s)

Specifying a 2-d Texture Object

OpenGL provides a flexible way to define a texture, but there are many parameter values to be specified:

- **External Format**: The format in which you present your texture to OpenGL (e.g., RGB, RGBA, ...)
- **Internal Format**: The format in which OpenGL stores the texture internally and the number of bits per pixel.
- **Width, Height**: Size of the image. For technical reasons, OpenGL requires image sizes to be a power of 2, but you can always pad your images out with extra pixels to satisfy this requirement.
- **Level of detail**: It is possible to provide images in different levels of detail. We will discuss this when we talk about mip-mapping.
- **Image border**: For purposes of smoothing, OpenGL interpolates a pixel's colors with its neighbors. This is a problem along the image edge, since there are no neighbors. Thus, a thin border around the image can be provided for smoothing purposes.
Specify a 2-d Texture Object

gTexImage2D ( GLenum (target), GLint (level),
  GLint (internalFormat), GLint (width), GLint (height),
  GLint (border), GLenum (format), GLenum (type),
  const GLvoid* (texels奀 );

Sample:
gTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB8, GL_UNSIGNED_BYTE, myImage):

• (format) and (type) are used to specify the way in which the
  texels are stored in your image array.
• (internalFormat) specifies how OpenGL should store the data
  internally.
• (width) and (height) give the image size. They must be powers
  of 2. You can use gluScaleImage( ) to scale your image.
• (level) and (border) have other uses (see documentation).

Specify how Texture is Applied

How is the color of the texture pixel combined with the existing
pixel?
The main issue to do with whether the texture color is
combined with existing object color after lighting (modulation)
or is just painted on (decal).

gTexEnv(f) ( GLenum (target), GLenum (name), (TYPE) (value) )

where (target) is GL_TEXTURE_ENV.

(pname) value
GL_TEXTURE_ENV_MODE GL_MODULATE (mix with lighting) or,
GL_REPLACE (just paint this color).

Enable the Texture and Draw

gEnable ( GL_TEXTURE_2D );
  - Enable 2-d texturing.

gTexCoord2f ( GL_FLOAT s, GL_FLOAT t );
  - Specify texture coordinates for the next vertex. (Applies to all
    subsequent vertices until changed, just the same as glNormal( ),
    and glColor( ).)
  - Indexing is relative to the lower left corner, and (irrespective of
    the image size), s and t range from 0 to 1.

gDisable ( GL_TEXTURE_2D );
  - Disable 2-d texturing, to return to
    simple coloring.

Example

Texture Initialization:
gGenTextures ( ... ); // create new texture objects
gBindTexture ( ... ); // make this texture active
gTexParameter ( ... ); // define texture properties
  // ... input texture array from file or generate ...
gTexImage2D ( ... ); // provide the texture to OpenGL

Displaying a Textured Object:
gEnable ( GL_TEXTURE_2D ); // enable texturing
  gBegin ( GL_TRIANGLES ); // draw the object
  gTexCoord2f ( ... ); // draw other vertices in the same way
  gEnd ( );
gDisable ( GL_TEXTURE_2D ); // done

In-Class Exercise

Skybox:
  - You want to enclose your world within a huge cube (skybox), which
    will be texture mapped with a picture of the sky.
  - The skybox is centered about the origin. It is 50,000 units high
    and 100,000 units wide.
  - You want to paste the 1024 x 1024 texture shown below onto the
    top and sides of your skybox.
  - Give the OpenGL commands to draw one of the sides (say the East
    side) and provide the texture coordinates. Draw the side so that
    (when viewed from inside) the vertices are given in CCW order.
**Minimization Filtering and MIP-mapping**

What if one screen-space pixel overlaps many texture pixels? Ideally we should average these pixels, but this takes time. So OpenGL just takes one.

Result: A jagged appearance, aliasing.

MIP-mapping: Precompute averages and build hierarchy based on powers of 2.

To render: find the appropriate level in the MIP-map, and use this pixel. This smooths out the jagged lines.

---

**Magnification Filtering**

What if many screen-space pixels overlap one texture pixel?

- This results in a blocky appearance.
- Can we do anything to smooth things out?

**Bilinear Interpolation**

Find four nearest texels to fragment center \( (a, b, c, d) \):

Interpolate between \( a \) and \( b \) in \( s \) (\( p_0 \)):

\[
p_0 = (1-s) \cdot a + s \cdot b.
\]

Interpolate between \( c \) and \( d \) in \( s \) (\( p_1 \)):

\[
p_1 = (1-s) \cdot c + s \cdot d.
\]

Interpolate between \( p_0 \) and \( p_1 \) in \( t \) (\( p' \)):

\[
p' = (1-t) \cdot p_0 + t \cdot p_1.
\]

**Perspective Foreshortening**

Linear Interpolation is an affine operation, but perspective is not. Thus, linear interpolation does not account for perspective foreshortening.

Result: A line’s midpoint in object space is not mapped to midpoint in screen space.

**Solution: Perspective-Correct Interpolation**

Perspective-Correct Interpolation:

Let \( p_0 \) and \( p_1 \) be two points to interpolate, and let \( z_0 \) and \( z_1 \) denote their respective depth values.

Let \( q_0 = \frac{p_0}{z_0} \) and \( q_1 = \frac{p_1}{z_1} \).

The Trick: Interpolate \( q_0 \) and \( q_1 \) and then divide by the interpolant of \( 1/z_0 \) and \( 1/z_1 \). Effectively this is the same as doing linear interpolation prior to perspective normalization, and then projecting. Letting \( 0 \leq \alpha \leq 1 \) be the interpolation parameter, we have:

\[
p = \left( \begin{array}{c}
1-a \\
1-a
\end{array} \right) q_0 + a q_1
\]

**Overview**

- The texture-mapping process
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
- Perlin noise
Bump Mapping:
- Introduced by Blinn in 1978.
- We perceive bumps and surface roughness because of variations in diffuse and specular illumination.
- In turn, these are functions of the surface normal vector: $(\mathbf{n} \cdot \mathbf{l})$ and $(\mathbf{n} \cdot \mathbf{h})$.
- Remarkably by modifying just the surface normal $\mathbf{n}$, we can create appearance of bumps without actually changing the surface geometry.
- Note: These are not “real” bumps. Silhouettes remain smooth.

Displacement Mapping:
- In contrast to bump mapping, the actual geometry is displaced.
- The surface as well as silhouettes appear non-smooth.
- Presents a challenge for deferred shading systems (when shading is performed after visibility).

Environment Mapping:
- Invented by Blinn and Newell 1976.
- Index into texture map not by $(u, v)$ but by reflection rays.
- Creates illusion of reflectivity.

Computing the Reflection Vector:
- Given $\mathbf{v}$ and $\mathbf{n}$ compute $\mathbf{r}$.
- Assume $\mathbf{v}$ and $\mathbf{n}$ normalized.
- From properties of the dot product, recall that the orthogonal projection of $\mathbf{v}$ onto $\mathbf{n}$ is:
  \[
  \mathbf{n}' = (\mathbf{v} \cdot \mathbf{n}) / (\mathbf{n} \cdot \mathbf{n}) \mathbf{n} - (\mathbf{v} \cdot \mathbf{n}) \mathbf{n}.
  \]
- The vector from the head of $\mathbf{v}$ to head of $\mathbf{n}'$ is $\mathbf{u} = \mathbf{n}' - \mathbf{v}$.
- The reflection vector is:
  \[
  \mathbf{r} = \mathbf{v} + 2 \mathbf{u} = \mathbf{v} + 2 (\mathbf{n}' - \mathbf{v}) = 2 (\mathbf{v} \cdot \mathbf{n}) \mathbf{n} - \mathbf{v}.
  \]
Environment Mapping

Computing the Color in Direction \( r \):
- If we had a full 3-d environment model, we could shoot a ray starting at \( p \) in the direction \( r \) and determine what is hit.
- But this too computationally expensive for interactive speeds.
- Instead, we pre-compute a panoramic image of the environment from a point centered inside the object. Can be stored as 6 images projected onto the faces of a cube.

How to compute \( s \) and \( t \)?
- Let \( r = (a, b, c) \). The component of maximum absolute value \( \max(|a|, |b|, |c|) \) determines which face of the cube \( r \) hits.
- Suppose that \( |a| \) is maximum and positive, that is, the top of the cube. (There are 5 other analogous cases.)
- Texture coordinates \( s \) and \( t \) are determined by the slopes, \( \frac{b}{a} \) and \( \frac{c}{a} \). These slopes are in the interval \([-1,1] \), and so we transform to map to the range \([0,1] \), yielding:

\[
\begin{align*}
s &= \frac{1}{2} \left( \frac{b}{a} \right) - 1 \\
t &= \frac{1}{2} \left( \frac{c}{a} \right) - 1
\end{align*}
\]

Overview

- The texture-mapping process
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
- Perlin noise

Procedural Textures

Making the Checkerboard:
Consider with a 1-d vertical checkerboard pattern with 8 columns.
The column subdivisions are at multiples of 1/8.
So we have:
- Column 0: \( 0 \leq s < 1 \) (white)
- Column 1: \( 1 \leq s < 2 \) (black)
- Column 2: \( 2 \leq s < 3 \) (white)
- Column 3: \( 3 \leq s < 4 \) (black)
Final 1-d Rule:
- \( [s \mod 4] = 0 \rightarrow \text{white} \)
- \( [s \mod 4] = 1 \rightarrow \text{black} \)
Procedural Textures

Making the Checkerboard:
Consider with a 1-d horizontal checkerboard pattern with 8 columns.
By analogy we have:
\[
\lfloor 8 \cdot t \rceil \mod 2 = 0 \Rightarrow \text{white}
\]
\[
\lfloor 8 \cdot t \rceil \mod 2 = 1 \Rightarrow \text{black}
\]

Final 2-d Checkboard Rule:
To combine these we see that a square is white if and only if both the s-rule and t-rule generate the same color, and black otherwise. We have the following final rule (where \( \oplus \) denotes exclusive-or):
\[
f(s, t) = \text{white} \quad \text{if } (\lfloor 8 \cdot s \rceil \mod 2 \oplus \lfloor 8 \cdot t \rceil \mod 2) = 0 \quad \text{and black otherwise.}
\]
Of course this is easy to generalize to values other than 8.

Examples of Procedural Models

Overview
- The texture-mapping process
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
  - Perlin noise

Perlin Noise

Natural phenomena: derive their richness from random variations.
- But we don’t want white noise!
- Perlin noise provides nicely “structured” noise.
- Developed by Ken Perlin in 1980’s.

Properties:
- Reproducible. (Generated by a function so there is no need to store large texture images.)
- No repetitions.
- Band limited (smoothly changing).
- User-specifiable dynamic range.

Perlin noise: Is somewhat like like a fractal.
The overall noise function is a layering of many simpler noise functions. These functions are structurally similar, but vary in frequency and amplitude.

Terms from Harmonic Analysis:
- Wavelength: The distance between successive crests.
- Frequency: The number of crests per unit distance, \(1/\text{wavelength}.\)
- Amplitude: Height of the crests.
Harmonics

Periodic Functions: can be generated with a given frequency \( \omega \) and amplitude \( A \) by an appropriate transformation of the sine function:
\[
\sin(t) \rightarrow f_n(t) = A \sin(\omega t)
\]

Periodic Noise: Suppose that we are given a pseudo-random noise function that is periodic in nature. (See below.) We can transform its frequency and amplitude in the same manner:
\[
\text{noise}(t) \rightarrow \text{noise}_n(t) = \text{noise}(\omega t)
\]

Perlin: Layering and Dampening

How to Layer? It is common to layer functions by increasing frequency by doubling, and decreasing amplitude by halving:
\[
\text{noise}(t) = \frac{1}{2} \sum_{i=0}^{\infty} \text{noise}(2^i t)
\]

Dampening: The above is rather limited. We can generalize this by providing a parameter called dampening factor:
\[
\text{frequency}(i) = 2^i, \quad \text{amplitude}(i) = \frac{1}{d}, \quad d \text{ is the dampening factor. Thus we have:}
\]
\[
\text{noise}(t) = \frac{1}{d} \sum_{i=0}^{\infty} \text{noise}(2^i t)
\]

As the dampening factor increases, the noise smooths out.

Perlin: Octaves

Octaves:
- Each successive layer has twice the frequency as the previous one.
- The individual noise layers are called octaves.
- Why is it called an octave?
  - In music an octave is the span between 8 whole notes.
  - Doubling the frequency increases the pitch by one octave.

How many octaves should I use?
- Up to you. The more octaves the higher the final frequency.
- No need to generate frequencies higher than the pixel sampling rate on your image, since such linear variations will not be visible.

Perlin: Basic Noise Function

All this assumes that we have a basic noise function \( \text{noise}(x) \).

How do we generate it?
- Random?... but this would not reproduce the same function each time. (Our texture would vary from frame to frame.)
- Pseudo-random: A function that appears random, but produces the same value each time it is invoked with the same argument.

See http://freespace.virgin.net/hugo.elias/models/m_perlin.htm for a sample pseudo-random noise function that maps a 32-bit integer to a pseudo-random float in the interval \([1,\infty]\).

float noise (int x) {
    x -= (x << 13); x;
    return ( 1.0 - ((x * (x * x * 15731 + 789221) + 1376312589) & 7fffffff) / 1073741824.0);
}

Perlin: Smoothing the Basic Noise

Smoothing:
- To avoid a blocky look, we should smooth the noise out, by interpolating between successive values.
- Linear interpolation: (or Lerp) is fast, but produces a rather jagged results:
  - \( \text{Lerp}(a, b, x) = (1 - x) \cdot a + x \cdot b \), where \( 0 \leq x \leq 1 \).
- Cubic interpolation: is very smooth, but a bit complicated.
- Cosine interpolation: is a simple compromise. Idea:
  - The interpolant is \( f(x) = a + f(x) \cdot b \), where \( 0 \leq x \leq 1 \).
  - To make the transitions gradual, select a function \( f(x) \) that it covers the same interval \([0,1]\), but its derivative is zero at \( x = 0 \) and \( x = 1 \).
  - \( f(x) = (1 - \cos(\pi \cdot x)) / 2 \) does the trick.

1-d Perlin Noise: Octaves and Sum

Various octaves of 1-dimensional Perlin noise and their sum.
Perlin: Generalization to 2-dimensions

2-d Noise: Can be done by making the noise a function of x and y:
- \( \text{noise2D}(x, y) \leftarrow \text{noise}(x + P \cdot y) \),
  where \( P \) is a large prime number. (Watch out for arithmetic overflow.)

2-d Interpolation: a generalization of bilinear interpolation:
- Interpolate along x:
  \( p_0 \leftarrow \text{interp}(a, b, x) \)
  \( p_1 \leftarrow \text{interp}(c, d, x) \)
- and then interpolate along y:
  \( p' \leftarrow \text{interp}(p_0, p_1, y) \)

Perlin: Further Smoothing

Further Smoothing:
- Prior to interpolation, we can further smooth the noise out by averaging neighboring noise values together.

1-d smoothing:
- \( \text{smooth}_\text{noise}(x) = \text{noise}(x)/2 + \text{noise}(x+1)/4 + \text{noise}(x+1)/4 \)
- \( 1/4 \ 1/2 \ 1/4 \)

2-d smoothing: Apply this same idea to x, and then y.
- \( 1/16 \ 1/8 \ 1/16 \)
- \( 1/8 \ 1/4 \ 1/8 \)
- \( 1/16 \ 1/8 \ 1/16 \)

Perlin: Putting it Together

1-d Perlin Noise Function: (The 2-d case is an exercise.)

```c
float perlin_noise(float x) {
  float dampening_factor = dampening_factor;
  float freq = freq; 
  float total = 0;
  for ( 0 <= \( i < n \) )
    total += (1/di) * interp_noise( \( 2^i \cdot x \) )
  return total;
}
```

Summary

Summary:
- Texture mapping
- Texture mapping in OpenGL
- Other types of maps: Bump, Displacement, Environment
- Procedural textures
- Perlin noise

What’s Next?
- Shadows and Reflection