More Texture Mapping
Perturbing Normals
Perturbing Normals

- Instead of fetching a texture for color, fetch a new perturbed normal vector
- Creates the appearance of more geometric detail than from a single flat triangle
Defining a Coordinate System

- Use the mapping between each vertex \((x, y, z)\) and the texture coordinates \((u, v)\) to calculate a Tangent, Binormal, and Normal.
- Let \(u\) be the Tangent direction and \(v\) be the Binormal direction.
- Assume that \(x, y,\) and \(z\) are each linear functions of \(u\) and \(v\) in each triangle:
  \[
  x = a_0 u + b_0 v + c_0 \\
  y = a_1 u + b_1 v + c_1 \\
  z = a_2 u + b_2 v + c_2
  \]
- For each of the 3 vertices, plug the known values of \(x, y, z, u,\) and \(v\) into the above equations.
- This gives 9 total equations, or 3 sets of 3 equations.
- That is, there is a 3x3 system of equations for \(a_0, b_0, c_0,\) and likewise a 3x3 system for \(a_1, b_1, c_1,\) and likewise a 3x3 system for \(a_2, b_2, c_2\).
- After solving, we set \(\vec{T} = \left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right) = (a_0, a_1, a_2),\) \(\vec{B} = \left(\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}\right) = (b_0, b_1, b_2),\) and \(\vec{N} = \vec{T} \times \vec{B}\).
- (Having a local coordinate system is useful for various other operations.)
Perturbing the Normal

- The normal would typically be \((0, 0, 1)\) in the triangle’s local coordinate system \((\vec{T}, \vec{B}, \vec{N})\) that is \(0\vec{T} + 0\vec{B} + 1\vec{N} = \vec{N}\)

- Instead use a new normal vector \((n_T, n_B, n_N)\) stored in the “texture” or computed from the “texture”

Aside on the light direction:

- The light direction can be computed on the fly at each point on the triangle that corresponds to a pixel
- Or alternatively, it can be computed only at the vertices of the triangle
- In this later case, we use (perspective correct) barycentric interpolation in order to interpolate the light direction to interior points of the triangle

Caution: Before computing dot products between the light direction and the normal direction, be sure to transform the light direction into the triangle’s local coordinate system \((\vec{T}, \vec{B}, \vec{N})\)
Bump Maps

- A bump map is a single-channel (grey-scale) height map $h(u, v)$
- Points on the three dimensional surface are given by $(u, v, h(u, v))$
- The tangent plane at the point $(u_0, v_0, h(u_0, v_0))$ is given by
  \[
  - \frac{\partial h(u, v)}{\partial u} \bigg|_{(u_0, v_0)} (u - u_0) - \frac{\partial h(u, v)}{\partial v} \bigg|_{(u_0, v_0)} (v - v_0) + (h - h(u_0, v_0)) = 0
  \]
- The partial derivatives can be computed using finite differences, e.g.,
  \[
  \frac{\partial h(u,v)}{\partial u} \bigg|_{(u_0,v_0)} = \frac{h(u_{\text{right}},v_{\text{center}}) - h(u_{\text{left}},v_{\text{center}})}{u_{\text{right}} - u_{\text{left}}}
  \]
  \[
  \frac{\partial h(u,v)}{\partial v} \bigg|_{(u_0,v_0)} = \frac{h(u_{\text{center}},v_{\text{top}}) - h(u_{\text{center}},v_{\text{bottom}})}{v_{\text{top}} - v_{\text{bottom}}}
  \]
- The outward (non-unit) normal direction to this tangent plane is
  \[
  \left( - \frac{\partial h(u,v)}{\partial u} \bigg|_{(u_0,v_0)}, - \frac{\partial h(u,v)}{\partial v} \bigg|_{(u_0,v_0)}, 1 \right)
  \]
- Normalize to get a unit normal $(n_T, n_B, n_N)$
Bump Maps

without bump mapping

with bump mapping
Normal Maps

- A normal map stores “color-codes” of normal vectors

- \((n_T, n_B, n_N)\) is a normalized vector, so each element is in the range of \([-1, 1]\)

- They are converted to a range of \([0, 255]\) and stored as colors in the texture

\[
(R, G, B) = 255 \times \frac{(n_T, n_B, n_N) + (1, 1, 1)}{2}
\]

\[
(n_T, n_B, n_N) = 2 \times (R, G, B)/255 - (1, 1, 1)
\]

- Larger storage for an RGB image as opposed to single channel black and white for a height field, but less computation to compute the normal
Normal Maps

- normal mapping on a plane
- note the variation of specular highlights created by the variation of normals
Perturbing Geometry
Displacement Mapping

Bump mapping

Displacement mapping

*note the difference in the silhouette*
Displacement Mapping

- Perturb the surface geometry
  - Makes new (temporary) geometry on-the-fly at render time
- The texture stores a height map $h(u, v)$. The texel values are used to perturb the vertices along the normal direction of the original geometry

Pros:
- self-occlusion, self-shadowing
- silhouettes and silhouette shadows appear correctly

Cons:
- expensive
- requires adaptive tessellation to ensure that the surface mesh is fine enough near perturbations
- still need to bump/normal map for sub-triangle lighting variations
**Displacement Mapping**

- **Compute normals on the vertices of the original mesh**
- **Move each vertex in the original surface normal direction for a distance determined by** $h(u, v)$
- **Compute new normals from the new mesh**

\[
\text{original geometry} + h(u, v) = \text{perturbed geometry}
\]
Displacement Mapping

Input texture:

Displacement map over rectangular surface:
Displacement Mapping

- Notice the self-occlusion, self-shadows, and silhouette shadows
Environment Mapping
Environment Mapping

- Objects reflect light transmitted to them from other objects
  - e.g. the reflection of the tree on the car
- An environment texture map stores the reflected image of the environment as a texture image
Environment Mapping

- A photograph of a chrome sphere (light probe) contains the intensities of the environmental light shone onto the sphere from almost all directions

Miller and Hoffman, 1984, “Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments”
Assume objects in the environment are infinitely far away, so that the intensity of environmental light depends only on the direction $I$ (not on position).

$R$ is the direction from the light probe to the camera. $I$, $R$, $N$ are coplanar. $I$ and $R$ are equal-angle from $N$ according to mirrored reflection.

Thus, $N$ has a one-to-one correspondence with $I$.

In the texture (in the red square below), each texel stores light from one direction $I$ corresponding to one surface normal $N$. 

Side view of a light probe.
Environment Mapping

- Placing a coordinate system at the center of the sphere gives the surface normal \( N = (n_x, n_y, n_z) = (x, y, z) / \sqrt{x^2 + y^2 + z^2} \)

- \( n_x \) and \( n_y \) are in the range of \([-1, 1]\) and are transformed into the range \([0, 1]\) to get the corresponding texture coordinates \((u, v) = (n_x + 1, n_y + 1)/2\)

- when we render a CG object, we use the CG object’s local surface normal to compute the texture coordinates in order to fetch the color of the incoming light
Environment Mapping
Making Textures
Texture Tiling

- Creates a large image from small texture samples by repeating the small samples side by side
Texture Tiling

- Neighboring tiles should match across boundaries
- Can be hard/impossible if the texture doesn’t have natural periodic properties
- May look artificial because of the repetitive patterns

(From: http://procworld.blogspot.com/2013/01/introduction-to-wang-tiles.html)
Texture Synthesis

- Create a large non-repetitive texture from a small sample by using its structural content

- Algorithms for texture synthesis are typically either pixel based or patch based
  - Pixel-based algorithms - generate one pixel at a time
  - Patch-based algorithms - generate one block/patch at a time
Texture Synthesis: Pixel-based

The search stencil is shaped to contain the pixels that are already generated in a neighborhood around the pixel under consideration.

- To generate the texture for pixel $p$
  - compare $p$'s neighboring pixels in the (red) stencil with all potential choices in the input texture sample
  - choose the one with the smallest difference to fill pixel $p$
- Generate the texture in a raster scan ordering
- When the stencil tries to look up values outside the domain, periodic boundary conditions are applied (so at the beginning, the last few rows and columns of pixels are filled up with random values)

Search stencil
Texture Synthesis: Pixel-based

- Greatly reduces repetitive patterns compared to texture tiling
- The generated texture has similar content to the input texture sample
- However, it may lose too much structural content and/or create noisy or poorly structured textures
Texture Synthesis: Patch-based

Similar to texture tiling, but ...
- only uses a subset of the original texture sample to avoid repetitive blocks
- blends the overlapped regions to remove “seams”

For each patch being considered,
- search the original sample to find candidates which best match the overlap regions on the boundary
- choose from the good candidates

Advantages
- uses some structural content directly, so noise is less problematic
Texture Synthesis: Patch-based
Procedural Textures

- Created/generated using a mathematical/computational algorithm

- Good for generating natural elements
  - e.g. wood, marble, granite, stone, etc.

- The natural look is achieved using noise or turbulence functions

- These turbulence functions are used as a numerical representation of the “randomness” found in nature
Marble Texture

- Marble is metamorphosed limestone
- Typically contains a variety of material impurities that are chaotically distributed during metamorphosis
Marble Texture

- Predefine layers of different color marble
- Use a function to map \((u, v)\) locations to layers
- E.g. a sine function:

\[
\text{marble}(u, v) = \text{layer\_color}(\sin(k_u u + k_v v))
\]
Marble Texture

\[ marble(u, v) = layer\_color(sin(k_u u + k_v v)) \]

- \( k_u, k_v \) are spatial frequencies set by the user
- \( 2\pi / \sqrt{k_u^2 + k_v^2} \) determines the spatial periodicity
  - e.g. \( \sqrt{k_u^2 + k_v^2} \) is larger on the left figure
- The vector \((k_u, k_v)\) determines the direction of the patterns
- Problem: too regular, most things in nature have a degree of randomness
- Solution: add noise...
Perlin Noise

- To add noise to the marble texture, one could call a random number generator at every point of the texture
  - But this is “white noise” and has no structure
- We want to make the noise smoother and more structured

- Make a large grid with random numbers on each grid node
  - Interpolate the noise to the points inside the lattice cells
  - This gives spatial coherency
- Ken Perlin proposed a specific method for implementing this
Perlin Noise

- Lay out a 2D grid over the image, and assign a pseudo-random unit gradient to each grid point
- For each pixel with texture coordinates \((u, v)\), find out which grid cell it is inside
- Compute a weighted average of the gradients dot-product(ed) with the distance to that corner to get a noise value at \((u, v)\)

\[
n(u, v) = \sum_{i,j} w\left(\frac{u - u_i}{\Delta u}\right) w\left(\frac{v - v_j}{\Delta v}\right) \left(g(u_i, v_j) \cdot ((u, v) - (u_i, v_j))\right)
\]

- The cubic weighting function is widely used
  \[
  w(t) = 2|t|^3 - 3|t|^2 + 1 \text{ (for } -1 < t < 1)\]

The 4 pseudorandom gradients associated with the grid points
Perlin Noise

- Many natural textures contain a variety of feature sizes in the same texture.
- The Perlin Noise function recreates this by adding up noises with different frequencies and amplitudes
  \[
  \text{perlin}(u, v) = \sum_{k} n\left(\text{frequency}(k) \times (u, v)\right) \times \text{amplitude}(k)
  \]
- Typically, frequencies and amplitudes are chosen via
  \[
  \text{frequency}(k) = 2^k \\
  \text{amplitude}(k) = \text{persistence}^k
  \]
- Each successive noise function we add is called an octave, because it is twice the frequency of the previous one.
- Persistence is a parameter (\(\leq 1\)) diminishing the relative amplitudes of higher frequencies.
Perlin Noise

- **1D example**
- Smaller persistence = smaller higher frequency noise = smoother result

<table>
<thead>
<tr>
<th>Frequency</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence = 1/4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amplitude:</td>
<td>1</td>
<td>1/4</td>
<td>1/16</td>
<td>1/64</td>
<td>1/256</td>
<td>1/1024</td>
</tr>
<tr>
<td>Persistence = 1/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amplitude:</td>
<td>1</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/32</td>
</tr>
<tr>
<td>Persistence = 1 / \sqrt{2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amplitude:</td>
<td>1</td>
<td>1/1.414</td>
<td>1/2</td>
<td>1/2.828</td>
<td>1/4</td>
<td>1/5.656</td>
</tr>
<tr>
<td>Persistence = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amplitude:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(from: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)
Perlin Noise

- Used to generate height-fields, marble textures, etc.
- Can rescale it and add to itself to get a variety of natural looking maps
Perlin Noise

- Can be scaled and added to the marble texture
  \[ marble(u, v) = layer_{\text{color}}(\sin(k_u u + k_v v + A \times \text{perlin}(u, v))) \]
- Set the value of the parameter \( A \) to scale the noise
3D Textures

- Typically generated procedurally, 3D images are rare
- Although, one could slice up a 3D object and take a bunch of 2D pictures to make a 3D texture
- Or use some sort of 3D imaging technology

- Human Slices:

  Vertical

  Horizontal

  Brain
3D Textures

- Generate a 3D texture representing the material
- “Carve” the object out of this 3D texture
- Eliminate the difficulty of wrapping a 2D texture over a complex 3D object
- No need to worry about matching up the texture at the seams
Marble texture function with Perlin noise for a 3D texture

\[ marble(u, v, w) = layer\_color(sin(k_u u + k_v v + k_w w + A \times perlin(u, v, w))) \]
Wood texture is caused by tree rings.

Object point \((x, y, z)\)

Compute cylindrical coordinates for \((x, y, z)\) object points:

\[
R = \sqrt{x^2 + z^2} \\
\theta = \tan^{-1}\left(\frac{z}{x}\right) \\
H = y
\]
3D Textures

- Results
Homework Tip
DEBUG with checkerboard textures