THE PROBLEM

- There are a lot of phenomena that:
  - Change the surface appearance
  - Don’t really change the shape in a meaningful way

- Some examples:
  - Wrinkles, dents, scratches, woven texture, etc.
    - COULD model as geometry, but introduces WAY too many additional polygons
  - Stains, painted-on patterns, dust, dirt, fingerprints, etc.
    - Using geometry starts to become ridiculous...

- So, how do we render these phenomena WITHOUT using extra geometry?

http://cliparts.co/cliparts/6cr/5Ek/6cr5EkRzi.jpg
http://dented.co/wp-content/uploads/2012/10/Dent-Repair-300x228.jpg
http://www.illusionware.it/pc/doom3-05.jpg
TEXTURING

• **Texturing** = process that takes a surface and modifies its appearance at each location using some image, function, or other data source
  
  • *Example*: 2D image uses as diffuse color for surface

DIFFERENT KINDS OF TEXTURE MAPPING

• Textures can be used to alter many different properties:
  • **Diffuse texture mapping** → modifies *diffuse* color at each pixel ($k_d$)
    • *Example*: brick wall texture
  • **Specular texture mapping** → modifies *specular* color/intensity at each pixel ($k_s$)
    • *Example*: whether the bricks/mortar should reflect light or not
  • **Gloss texture mapping** → modifies *shininess* at each pixel ($\omega$)
    • *Example*: mortar should be glossy, but bricks are more matte
  • **Normal map texture mapping / bump mapping** → modifies *normal* at each pixel (N)
    • *Example*: makes little bumps/irregularities for bricks
  • **Parallax and relief mapping** → pretends to deform flat surface while rendering it
    • *Example*: gives illusion that bricks stick out from mortar and also cast shadows on mortar
  • **Displacement mapping** → actually displaces/changes surface (creating triangles between texels)
  • **Light mapping** → precompute complicated lighting calculations → use texture instead of lighting equation!
  • **Shadow mapping** → render view from light’s perspective → point in shadow if not visible on rendered image!
TEXTURING EXAMPLE

THE TEXTURING PIPELINE
THE TEXTURING PIPELINE

- Let’s think about how we get the texture values for a single pixel

  * There are four basic steps:
    - Projector function
    - Corresponder function(s)
    - Obtain value
    - Value transform function (optional)

- From here on out:
  - **Pixel** = single location on SCREEN
  - **Texel** = single location on TEXTURE
TEXTURE LOOKUP

- **Texture lookup** = getting the values from the texture for a given point on the model
  - Pretty much all the preceding steps in the texturing pipeline
PROJECTOR FUNCTION
PROJECTOR FUNCTION

- **Projector function**
  - Converts from OBJECT SPACE location $\rightarrow$ PARAMETER SPACE COORDINATES
    - Often 3D coordinates $(x,y,z)$ to texture coordinates
  - *Example*: **model coordinates** $(x,y,z) \rightarrow (u,v)$ **texture coordinates**
    - Can also start with world coordinates, but usually use model coordinates since texture moves with model

- **Texture coordinates** are:
  - In **texture space**
  - Usually normalized $\rightarrow [0,1]$
The projector function is also referred to as:
  - *Texture coordinate function*
  - *UV mapping*
  - *Surface parameterization*
WHAT PROJECTION DO WE USE?

• We’ll assume that we have:
  • Model coordinates \((x,y,z)\) for a point on the model
  • A 2D texture image
  • 2D texture coordinates \((u,v)\) we need to find
• How do we pick the best projection?
  • I.e., looking for best mapping \((x,y,z) \rightarrow (u,v)\)
PROJECTION

• **Parametric surface**
  - Actually get \((u,v)\) coordinates for free \(\Rightarrow\) same as our parameters used to define our line/surface 😊

• **Implicit surface** or **triangle mesh**
  - Need to careful pick projection function to get good results...
PROJECTION GOALS

• Trying to find a balance between:
  • **Bijectivity**
    • Want one-to-one mapping: each point \((x,y,z)\) \(\rightarrow\) maps to ONLY ONE point \((u,v)\)
    • Otherwise, one texel \(\rightarrow\) affects lots of points on model
    • *Exception*: repeating texture does not follow this goal, but this is INTENTIONAL
  • **Low size distortion**
    • Want distances between points in \((x,y,z)\) \(\rightarrow\) similar to distances between \((u,v)\) coordinates
    • Scale of texture approximately constant across surface
  • **Low shape distortion**
    • Don’t want texture squished or stretched too much
      • E.g., circle on texture \(\rightarrow\) should look (mostly) like a circle on the object (not a really narrow ellipse)
  • **Continuity**
    • Avoid seams in texture
      • I.e., neighbor points on object \(\rightarrow\) correspond to neighbor points on texture
    • If we HAVE seams/discontinuities, make them inconspicuous

Cartographers have been dealing with these issues for a while…

Let’s start with one of the simplest projection functions → **planar projection**

- Basically, you are splatting your texture onto a planar surface

If we have a plane that extends in the X and Z directions (say, a floor), then we can use the following to get our texture coordinates:

\[
\begin{align*}
u &= ax \\
v &= bz
\end{align*}
\]

- ...where a and b are some constants

**Problem:** we’re assuming the surface is an axis-aligned plane → what if it’s slanted, like a roof?

[Image of a slanted surface]
PLANAR PROJECTION: ANY PLANAR SURFACE

- Basically, planar projection \(\rightarrow\) same as applying the following:
  - **View transform**
    - Viewing direction = normal of plane surface
  - **Orthographic projection** where we discard the z coordinate!
    - Can scale and shift as we see fit as well

- **Example**: have a cube \([-1,1]\) \(\rightarrow\) want to use plane with normal \((1,1,1)\)
  - Also, want:
    - \("X"/"Y"\) distance of -1 to map to \(U/V = 0\)
    - \("X"/"Y"\) distance of 1 to map to \(U/V = 1\)
  - Thus, left = -3, right = 1, bottom = -3, top = 1

\[
V = \begin{bmatrix}
0.71 & 0 & -0.71 & 0 \\
-0.41 & 0.82 & -0.41 & 0 \\
0.58 & 0.58 & 0.58 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
\[
P_o = \begin{bmatrix}
0.5 & 0 & 0 & 0.5 \\
0 & 0.5 & 0 & 0.5 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
PROBLEMS WITH PLANAR PROJECTION

- **Planar projection** → works well if surface flat or nearly flat

- **Problems**: for a closed shape → **not bijective**!
  - Front and back will map to the same texture coordinates
  - If sides parallel to projection direction → smears same value across side surface
If we swap out the orthographic projection for a perspective projection → **projective texture coordinates**!

- Used in shadow mapping (which we’ll talk about later)

\[
\begin{bmatrix}
  u_w \\
  v_w \\
  * \\
  w
\end{bmatrix}
= P_V
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]
SPHERICAL COORDINATES

- Spherical coordinates
  - Describe \((u,v)\) coordinates with latitude and longitude
    - Alternative description: get spherical coordinates \((p, \theta, \phi)\), then discard \(p\)
  - If \(y\) = up, then:

\[
\begin{align*}
  u &= \left[ \pi + \arctan\left( \frac{z}{x} \right) \right] / 2\pi \\
  v &= \left[ \pi - \arctan\left( \sqrt{x^2 + z^2}, y \right) \right] / \pi
\end{align*}
\]
If your object is more column-like than spherical → use cylindrical coordinates

Examples: a pillar, an arm or leg, a vase, etc.

Compute full cylindrical coordinates → then discard radius:

Note: $u$ calculation same as spherical

\[
\begin{align*}
  u & = \left[ \pi + \arctan(z, x) \right] / 2\pi \\
  v & = (1 + y) / 2
\end{align*}
\]
PROBLEMS WITH SPHERICAL PROJECTIONS

• *Problems with spherical projections:*
  • More distortion as you get near the poles
  • At the poles themselves \(\rightarrow\) not bijective
  • Seam at one line of longitude

• The first two problems can be addressed with *cubemaps*
  • However, this does introduce additional discontinuities...
CUBE MAPS

- **Cube maps** or **cube textures**
  - Six square textures → one for each face of cube
  - Use 3D texture coordinates = ray from center of cube
    - Use largest component to pick face (e.g., (-3.2, 5.1, -8.4) → use -Z face)
    - Divide remaining two components by absolute value of largest side (-3.2/8.4, 5.1/8.4)
    - Remap from [-1,1] to [0,1] → compute (u,v) coordinates for face
  - Most often used for **environment mapping**
    - E.g., simulation reflections of room, skyboxes, etc.
  - Problem when **sampling at seams**
    - SHOULD sample across boundary → however, almost all graphics hardware can’t do this
    - Larger area covered

![Cube Map Diagram](http://upload.wikimedia.org/wikipedia/commons/b/b4/Skybox_example.png)
![Environment Mapping](http://upload.wikimedia.org/wikipedia/commons/3/31/Environment_mapping.png)
![NVIDIA CG Tutorial](http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter07.html)
PROJECTOR FUNCTION: DIFFERENT TYPES

Spherical | Cylindrical | Planar | Natural \((u,v)\) from parametric surface
PROJECTOR FUNCTION: MULTIPLE PROJECTIONS

• When creating a model, a different projector function might be used for each part
• However, for real-time rendering, usually all textures combined into one texture
• Store (u,v) coordinates per vertex
PROJECTION FUNCTION: DIFFERENT DIMENSIONS

• Texture coordinates can be:
  • 1D (u)
    • Coloration based on altitude of terrain
    • Texturing lines of rain with semi-transparent texture
  • 2D (u,v)
  • 3D (u,v,w)
    • Volume textures
    • Directional ➔ point to cube map
PROJECTOR FUNCTION: NON-INTERACTIVE VS. REAL-TIME

• **Non-interactive** → projector function often part of rendering process (i.e., calculated on the fly)

• **Interactive, real-time** →
  • In 3D modeling program, use projector function(s) to get \((u,v)\) coordinates for vertices
  • Store per-vertex \((u,v)\) coordinates
  • At render time, interpolate values for each fragment
  • **EXCEPTIONS**: texture animation (e.g., water flowing), environment mapping (i.e., which cube face to use)

• Note: NOT interpolated until we get to the corresponder function
CORRESPONDER FUNCTION
CORRESPONDER FUNCTION

- **Corresponder function**
  - Converts PARAMETER-SPACE COORDINATES $\rightarrow$ TEXTURE-SPACE LOCATIONS
  - *Example*: $(u,v)$ texture coordinates $\rightarrow$ $(x,y)$ actual texel coordinates
  - May perform scaling, translation, rotation, shearing, etc.
CORRESPONDER FUNCTION: OUT OF BOUNDS?

• Corresponder functions also determine behavior when desired texture coordinates are OUTSIDE of the texture → function called wrapping mode (OpenGL) or texture addressing mode (DirectX)

• Common wrapping mode functions:

<table>
<thead>
<tr>
<th>Wrap (DirectX)</th>
<th>Repeat (OpenGL)</th>
<th>Tile</th>
<th>Mirror</th>
<th>Clamp (DirectX)</th>
<th>Clamp to Edge (OpenGL)</th>
<th>Border (DirectX)</th>
<th>Clamp to Border (OpenGL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R R R R</td>
<td>R R R</td>
<td>R R R</td>
<td>R K K K</td>
<td>R R R</td>
<td>R R R R R R R R R R R</td>
<td>R K K K</td>
<td>R R R R R R R R R R R</td>
</tr>
</tbody>
</table>
CORRESPONDER FUNCTION: OUT OF BOUNDS?

- **Wrap** (DirectX) / **Repeat** (OpenGL) / **Tile**
  - Often the default
  - Algorithmically, removes integer part of coordinate
  - Good for texture that repeatedly covers surface

- **Mirror**
  - Provides some continuity along edges of texture

- **Clamp** (DirectX) / **Clamp to Edge** (OpenGL)
  - Values outside of [0,1) clamped to this range → repetition of edges of texture
  - Avoids accidentally taking samples from opposite edge of texture when bilinear interpolation happens near texture’s edge

- **Border** (DirectX) / **Clamp to Border** (OpenGL)
  - Values outside of [0,1) are rendered with separately defined border color
  - Good for decals on surfaces → blends smoothly with border color
GETTING TEXTURE VALUES
OBTAINING TEXTURE VALUES

• Now we can get our texture values from the texture-space coordinates

• Most of the time, grab data from image (or maybe volume)
  • Example: 2D Texture → use texel coordinates and grab value from image

• However, can also generate textures procedurally
  • Use texture-space coordinates → compute function that outputs color

• Often return RGB or RGBA values
  • However, can also returns other data (e.g., normals with bump mapping)
VALUE TRANSFORM FUNCTION (OPTIONAL)

• One can also optionally transform the texture values

• Examples:
  • Remap from (0,1) to (-1,1)
  • Checking texture value vs. reference value → returning flag (often used in shadow mapping)
PERSPECTIVE CORRECT INTERPOLATION
INTERPOLATING VALUES

- Hitherto, if we’ve had to interpolate values between vertices → use barycentric coordinates
  - E.g., color at each vertex

- HOWEVER, if we’ve already rasterized the polygon → vertices in screen space → interpolating across screen space

- PROBLEM: Because of perspective projection → should be interpolating with perspective taken into account!

SIMPLE CASE: INTERPOLATING A LINE

• Let’s say this is our original line in world space:

\[ Q(t) = Q_1 + t(Q_2 - Q_1) \]

• Note also that our \((u,v)\) coordinates are relative to the world/model coordinate space

• After a perspective projection, our new line is:

\[ S(a) = S_1 + a(S_2 - S_1) \]

• NOTE: OUR PARAMETERS ARE NOT THE SAME!!!

• \( a \neq t \)

• We can’t just use \( a \) to interpolate \((u,v)\) \(\rightarrow\) so, how to we get the corresponding \( t \)?
• Our original line is this: \( Q(t) = Q_1 + t(Q_2 - Q_1) \)

• If our transformation matrix is \( M \), our transformed line is:

\[
MQ_1 + t(MQ_2 - MQ_1) = \frac{S_1 + t(S_2 - S_1)}{w_1 + t(w_2 - w_1)}
\]

• We can alternatively consider our line to be:

\[
\frac{S_1}{w_1} + a\left( \frac{S_2}{w_2} - \frac{S_1}{w_1} \right)
\]
INTERPOLATION: THE HARD WAY

- If you go through some mathematical gyrations, you can get the parameters as functions of each other:

\[
\begin{align*}
t(a) &= \frac{w_1a}{w_2 + a(w_1 - w_2)} \\
a(t) &= \frac{w_2t}{w_1 + t(w_2 - w_1)}
\end{align*}
\]

- So, if you have an interpolated point \(S'\) in screen space:

\[
S' = S_1 + a(S_2 - S_1)
\]

- You can figure out \(t\), and then get \((u,v)\) coordinates:

\[
\begin{align*}
u' &= u_1 + t(a)(u_2 - u_1) \\
v' &= v_1 + t(a)(v_2 - v_1)
\end{align*}
\]

- **PROBLEM**: Evaluating \(t(a)\) for each fragment is slow → is there a better way?
INTERPOLATION USING HOMOGENEOUS COORDINATES

- Remember that, BEFORE we do the homogenous divide, we had a nice, clean linear interpolation between points
  - Example: z values AFTER projection but BEFORE homogeneous divide
- The trick, then, is to do the following:
  - Perform projection on points
  - BEFORE doing the homogeneous divide → append (u, v, 1) to your vector
  - Perform the homogeneous divide
  - Get interpolation weights (barycentric coordinates) using screen space coordinates, BUT use (u/w) and (v/w) values
    - Effectively “screen space” (u,v) coordinates → (u’,v’)
  - Divide interpolated (u’, v’) values by interpolated 1/w → cancels out w to get correct original world (u,v) coordinates
OPENGL AND GLSL

• OpenGL by default will correctly interpolate any values you send out of the vertex shader using perspective-correct interpolation
  • Remember that gl_Position contains the w component, so the GPU has everything it needs
• You can actually modify this by prefixing your variable with one of the following:
  • flat
    • No interpolation; use “provoking vertex” of primitive to get value (by default the last vertex)
  • noperspective
    • Interpolate WITHOUT perspective correction
  • smooth
    • Interpolate with perspective correction
    • Default
IMAGE TEXTURING
INTRODUCTION

• For now, let’s focus on 2D image texturing (most common type of texturing in real-time graphics)
  • Texture = 2D image

• Typically, texture sizes are in powers of 2: $2^m \times 2^n$
  • As we’ll see, this related to mipmapping hardware
  • Modern GPUs can handle textures of arbitrary size
SIZING PROBLEMS

• Let’s say a texture covers a square projected on the screen
• If the projected area (in pixels)
  • Same size as texture → life is good
  • LARGER than texture → MAGNIFICATION
  • SMALLER than texture → MINIFICATION
• We will now cover some approaches to address these problems
Because the projected area is larger than the texture, we will sometimes need to get values in between texels.

Different approaches for this:

- **Nearest-Neighbor (i.e., Box Filter)**
  - Pick nearest whole texel
  - Fast, simple, hardware-supported
  - Terrible results $\rightarrow$ pixelation (individual texels are easy to see)

- **Bilinear Interpolation**
  - Interpolation between 4 nearest texel $\rightarrow$ weighted average
  - Not as fast, still hardware-supported
  - Less pixelation, more blurry

- **Bicubic Interpolation**
  - Use 4x4 or 5x5 weighted sum of texels
  - Slower still, not usually hardware-supported (although can code in pixel/fragment shader)
  - Better results
MAGNIFICATION

48 x 48 $\rightarrow$ 320 x 320

Nearest-Neighbor

Bilinear interpolation

Cubic interpolation (5x5)
Let’s say we want to get the value at texture coordinates \((p_u, p_v) = (81.92, 74.24)\).

Drop the integer parts \(\rightarrow (0.92, 0.24) \rightarrow (u', v')\)

Get weighted average of surrounding 4 values (where \(t(x,y)\) returns the texel value):

\[
b(p_u, p_v) = (1-u')(1-v')t(x_l, y_b) + u'(1-v')t(x_r, y_b) + (1-u')v't(x_l, y_t) + u'v't(x_r, y_t)
\]
What if you have a texture that should keep sharp edges (e.g., checkboard, text, etc.)?

- Bilinear interpolations \(\rightarrow\) blurs boundaries

  - Possible solution: use some kind of grayscale remapping scheme (if value < 0.4 \(\rightarrow\) value = 0)

- Nearest Neighbor (boxes different sizes)
- Bilinear Interpolation
- Bilinear Interpolation (with remapping)
This is when the pixel’s area covers several texels \( \rightarrow \) have to integrate the effects of all texels

- Effectively impossible to do this completely correctly in real-time
- \textit{Nearest-neighbor} \( \rightarrow \) severe aliasing
- \textit{Bilinear interpolation} \( \rightarrow \) good as long as more than 4 texels don’t influence the pixel cell; otherwise, aliasing again

Several approaches:
- Mipmapping
- Summed Area Tables
- Unconstrained Anisotropic Filtering
MINIFICATION: MIPMAPPING

- Mipmapping
  - Most popular method of antialiasing for textures
    - “Mip” = multum in parvo = Latin for “many things in a small place”
  - Start with original texture (Level 0)
    - Downsampling to a quarter of the area → Level 1 (subtexture of original)
    - Downsampling Level 1 texture → Level 2 texture
    - Repeat...
  - **Mipmap chain** = set of all the images
  - **d axis** = level of detail (also called \( \lambda \))
MINIFICATION: MIPMAPPING

• To do mipmapping properly, you need:
  • **Good filtering** → i.e., how you get your downsampled texture
    • Box filter = average of 4 texels from Level N to get Level N+1 texel → one of the worst filters possible
    • Gaussian filter → better choice
  • **Gamma correction**
    • Nonlinear relationship between actual intensity and value representing intensity → when averaging, need to do gamma correction to make sure upper levels of mipmaps do not get darker
MINIFICATION: MIPMAPPING

To use mipmapping:

- Compute $d \rightarrow$ floating-point number
  - Use longer edge of quadrilateral formed by pixel’s cell on texture space
  - OR
  - Compute differentials (i.e., change in texture space across screen)
- May also add level of detail bias (LOD bias) $\rightarrow$ starting $d$
  - User sets this
  - Increase bias $\rightarrow$ blurrier textures
  - Decrease bias $\rightarrow$ sharper textures
- $d_{\text{low}} = \text{floor}(d)$, $d_{\text{high}} = \text{ceiling}(d)$
- Use bilinear interpolation to get values from $V_{\text{low}} = (u,v,d_{\text{low}})$ and $V_{\text{high}} = (u,v,d_{\text{high}})$
- Interpolate values $V_{\text{low}}$ and $V_{\text{high}}$ based on $d \rightarrow \text{trilinear interpolation}$
MINIFICATION: MIPMAPPING

• Disadvantage of mipmapping → overblurring
  • If you cover a lot of pixels in one direction but not the other → uses largest side to compute d
  • Partial solution: ripmap → encodes rectangular areas of texture as well
    • Still doesn’t help when pixel cell covers diagonal area
MINIFICATION: SUMMED-AREA TABLES

- Summed-area table (SAT) → a method to avoid overblurring problem of mipmaps
  - Create array S that is the size of the texture, BUT contains more bits of precision (e.g., 16 bits per color per texel instead of the usual 8 bits)
  - Each location S(x,y) contains the sum of all pixels from S(0,0) to S(x,y) inclusive
  - To get the average color value for a rectangle in the texture:

\[
c = \frac{S[x_{ur}, y_{ur}] - S[x_{ur}, y_{ll}] - S[x_{ll}, y_{ur}] + S[x_{ll}, y_{ll}]}{(x_{ur} - x_{ll})(y_{ur} - y_{ll})}
\]
MINIFICATION: COMPARISON OF APPROACHES SO FAR

• **Anisotropic filtering algorithms** = can retrieve texel values over areas that are not square
  - E.g., ripmaps and summed-area tables (but best in horizontal/vertical directions)
  - Regular mipmapping is NOT an anisotropic filtering algorithm

• **Memory**:
  - **Mipmaps** $\rightarrow$ + $\frac{1}{3}$ of original texture memory
  - **Ripmaps** $\rightarrow$ + $3x$ original texture memory
  - **Summed-area tables** $\rightarrow$ + at least $2x$ for texture of size 16 x 16 or less $\rightarrow$ more precision needed for larger textures
MINIFICATION: COMPARISON OF APPROACHES SO FAR

Nearest neighbor

Mipmapping

Summed area tables
MINIFICATION: UNCONSTRAINED ANISOTROPIC FILTERING

• Mipmapping hardware already in place

• **Unconstrained anisotropic filtering**
  • Find axis of projected pixel area → **line of anisotropy**
  • Take *multiple square mipmap samples* along line of anisotropy
    • Common number of samples: 2
    • Use shorter side of quad to get d → less blurry
  • **Advantages:**
    • Line of anisotropy can run in any direction → not limited like ripmaps and summed-area tables
    • Same memory requirements of mipmapping
MINIFICATION:
UNCONSTRAINED ANISOTROPIC FILTERING

Mipmapping

Unconstrained anisotropic filtering
(16:1) → 16 samples
BUMP MAPPING
INTRODUCTION

- **Bump mapping** = family of techniques that give surfaces a more 3D appearance than garden-variety 2D texturing (but less than actually creating geometry)
  - Modify the geometric normal using values from a texture
  - Normal must be modified relative to the **tangent space basis** at the point (i.e., local frame of reference)
• **Tangent space basis** = normal, tangent, and bitangent (binormal) vectors
  - Tangent and bitangent → axes of normal map in object space
  - Need all three at each vertex
    - Store them all explicitly
    - OR
    - Store tangent and bitangent → compute normal using cross product
      - Issue of “handedness” → either make sure it’s always the same or store a bit indicating left or right hand rule!
  - Often transform light vector, view vector, etc. in tangent space before doing lighting calculations
    - Easier to modify normal from normal map values, since normal will now be (0,0,1)
BLINN’S METHODS

• The original bump mapping approach proposed by Jim Blinn:
  • Store two signed values, $b_u$ and $b_v$, in texture → vary normal by that much in $u$ and $v$ axes
  • Bump map texture called offset vector bump map or offset map

• Alternative: use a heightfield
  • $(0,1) →$ lower and higher
  • Compute $b_u$ and $b_v$ by getting differences of neighboring pixels → edge detection

“All it takes is for the rendered image to look right.”
-- Jim Blinn

http://old.siggraph.org/awards/1999/Coons.html
BLINN’S METHODS
NORMAL MAPPING

• Normal map
  • Stores the (x,y,z) values of the perturbed normal
    • Fewer calculations (if not converting to tangent space), greater storage size
    • Preferred method on modern hardware
  • Each value:
    • [-1,1] → [0, 255]
  • Demo: http://jsoserson.github.io/texture.js/demo/

http://jsoserson.github.io/texture.js/demo/images/brickwork-normal.jpg
NORMAL MAPPING: WHAT SPACE?

• What space is the normal from the normal map in?
  • World space
    • Original approach, but rarely used now
    • Doesn’t work if object is rotated
  • Object space
    • Valid under rigid transformations (but not other kinds of deformations)
    • Light vector must be transformed to object space
  • Tangent space
    • Can compress texture a little better → $z$ is always positive
    • More transformations required for shading → tangent space basis can change across surface
• When we do shading, the lighting and surface MUST be in the same space: world, tangent, or object
  • Tangent → can compute light vector per vertex and interpolate across surface
    • More efficient for single light (not transforming normal to world space every pixel)
  • World → convert normal map value to world space
    • More efficient for multiple lights; needed if doing reflection calculations
**Parallax Mapping**

- **Parallax mapping**
  - Addresses problem with normal mapping (bumps do not block each other)
    - *Example*: bricks should stick out from surface and block view of mortar from some angles
  - **Parallax** = position of objects move relative to one another as observer moves
  - Bumps are stored in heightfield texture
  - When viewing surface at given pixel:
    - Get heightfield value
    - Shift texture coordinate to retrieve different part of surface → based on height and viewing angle
  - So:
    - \( h \) = heightfield value
    - \( v = \text{normalized view vector} \rightarrow v_z = \text{height}, v_{xy} = \text{horizontal component} \) (MUST BE IN TANGENT SPACE)
    - \( p_{adj} \) = new parallax-adjusted position:
      \[
p_{adj} = p + \frac{h \cdot v_{xy}}{v_z}\]
PARALLAX MAPPING

• **Advantages:**
  - Simple; works well in practice → practical standard for bump mapping

• **Disadvantages:**
  - Illusion falls apart at shallow viewing angles → small height changes, LARGE coordinate shift
  - Partial solution: use **offset limiting** → limits offset to retrieved height:
    \[ p'_{adj} = p + h \cdot v_{xy} \]
  - Texture swimming → texture swirls around when view changes
  - Stereo rendering → need to make sure depth cues are consistent
MAIN PROBLEM WITH PARALLAX MAPPING

• Main problem with parallax mapping → we want when the view vector FIRST intersects with the heightfield

• Almost all approaches to do this use some approximation of ray tracing...which brings us to relief mapping...
• Relief mapping
  • Different algorithms for this: 
    *parallax occlusion mapping (POM)*, relief mapping, steep parallax mapping
  • Basic idea:
    • Test a fixed number of texture samples along the projected vector
      • Use more samples if view rays are at grazing angles
    • Once a sample below the heightfield is found → use 1) distance below heightfield and 2) distance above previous sample to find intersection point
    • Use intersection point to get diffuse texture, normal map, etc.
  • Can use multiple heightfields for overhangs, independent overlapping surfaces, etc.
  • Can also be used to do *self-shadowing* (where the object casts shadows on itself)
RELIEF MAPPING

• **Challenges:**
  - Finding the actual intersection point
    - Sample a lot in parallel → more texture accesses
  - Use iterative approach → fewer texture accesses, but dependent on each other
  - Must sample heightfield frequently enough → problematic when mipmapping/anisotropic filtering comes into play

• **Problem:**
  - Illusion breaks down at silhouette edge

https://chetanjags.files.wordpress.com/2013/05/pom_pom.png
NORMAL VS. RELIEF MAPPING

http://s140.photobucket.com/user/ktbluear/media/Fig25.jpg.html
NORMAL MAPPING VS. PARALLAX MAPPING VS. RELIEF MAPPING
DISPLACEMENT MAPPING

• **Displacement mapping**
  • Uses heightfield to modify vertex locations → models bumps as true geometry
  • More memory and computation intensive → becoming more popular because of *geometry shaders* (create geometry on the fly)
  • Collision detection can be a problem → don’t have geometry to intersect with at application level
3D TEXTURES
SOLID OBJECTS?

- Up to this point, we’ve mapped a 3D point \((x, y, z)\) \(\rightarrow\) 2D position \((u, v)\) in texture
  - Really just concentrated on the surface of an object

- What if we want to model/texture the interior?
  - E.g., what if we had a deformable/modifiable object, like wood or marble?
  - Possible solutions:
    - Volume (3D) textures
    - Procedural textures
VOLUME TEXTURES

• **Volume textures**
  • Use \((u,v,w)\) coordinates to access
  • **Advantages:**
    • Could use 3D coordinates directly as texture coordinates if desired
    • Can use to represent volumetric structure of a material → e.g., wood, marble, etc.
  • **Disadvantages:**
    • Significantly higher storage requirements
    • More expensive to filter
    • Overkill for surface texturing
**PROCEDURAL TEXTURING**

- **Procedural Texturing** = use function to compute texture VALUES on the fly
  - Good for volumetric materials (e.g., wood, marble)

http://modo.docs.thefoundry.co.uk/modo/701/help/pages/shaderendering/ShaderItems/Wood.html
COMMON TYPES OF PROCEDURAL TEXTURES

• There are a number of different patterns/functions that can be used alone or in combination to generate good procedural textures:
  • 3D stripes
  • Solid noise (specifically, *Perlin noise*)
  • Turbulence
3D STRIPE TEXTURES

• Assuming we have two colors, $C_0$, and $C_1$, we need some oscillating function to switch between them (like sine):

• If we want to make the width $w$ adjustable:

• If we want smooth interpolation, then we can compute a parameter $t$ with the sin function and then use linear interpolation between the two colors:

```cpp
vec3 stripe(vec3 point) {
    if(sin(point.x) > 0)
        return c0;
    else
        return c1;
}
```

```cpp
vec3 stripeWidth( vec3 point, float w) {
    if(sin(PI*point.x/w) > 0)
        return c0;
    else
        return c1;
}
```

```cpp
vec3 stripeSmooth( vec3 p, float w) {
    float t = (1 + sin(PI*p.x/w))/2;
    return (1-t)*c0 + t*c1;
}
```
SOLID NOISE

- Usually, we want some kind of pattern that looks “mottled”
  - E.g., bird’s eggs, marble, etc.
- Best to use some kind of solid noise
  - *Randomize each point?*
    - Looks like TV static → not what we’re looking for
  - *Large lattice → random number at each point in lattice → interpolate values?*
    - Lattice will be too obvious
- Perlin noise
  - Most commonly used noise for this purpose
    - Perlin received a technical Academy Award for this → impact in film industry
  - Uses lattice technique, but uses some tricks to make it less obvious…
PERLIN NOISE

- Perlin noise:

\[ n(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor + 1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor + 1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor + 1} \Omega_{ijk} (x - i, y - j, z - k) \]

- ...where:
  - \((x,y,z) = \text{position in model}\)
  - \((i,j,k) = \text{lattice point indices}\)

- **Major changes from basic lattice method:**
  - Use Hermite interpolation between lattice points
  - Use random VECTORS instead of single values at each point \(\Rightarrow\) use dot product to get value
  - Use 1D array and hashing to create virtual 3D array of random vectors
PERLIN NOISE

• Let’s look at the inner part of the equation; at this point, we have some fractional interpolation parameters \((u,v,w)\) between lattice points:

\[
\Omega_{ijk}(u,v,w) = \omega(u)\omega(v)\omega(w)(\Gamma_{ijk} \cdot (u,v,w))
\]

• The actual interpolation weight is not linear, but is instead based on a cubic weighting function (Hermite):

\[
\omega(t) = \begin{cases} 
2|t|^3 - 3|t|^2 + 1 & \text{if } |t| < 1 \\
0 & \text{otherwise}
\end{cases}
\]

\[
n(x, y, z) = \sum_{i=x}^{x+1} \sum_{j=y}^{y+1} \sum_{k=z}^{z+1} \Omega_{ijk}(x-i, y-j, z-k)
\]
PERLIN NOISE

• To get a random vector, Perlin uses a pseudorandom table:

\[
n(x, y, z) = \sum_{i=x}^{x+1} \sum_{j=y}^{y+1} \sum_{k=z}^{z+1} \Omega_{ijk}(x-i, y-j, z-k)
\]

\[
\Omega_{ijk}(u, v, w) = \omega(u)\omega(v)\omega(w)(\Gamma_{ijk} \cdot (u, v, w))
\]

\[
\Gamma_{ijk} = G(\phi(i + \phi(j + \phi(k))))
\]

• ...where:
  • G = precomputed array of n random unit vectors
  • \(\phi(i) = P[i \mod n]\) \(\rightarrow\) P is array of length n containing a permutation of integers 0 through n-1
    • Perlin used n = 256
PERLIN NOISE

• Each random vector in G is generated in the following fashion:
  • **Step 1:** Get 3 canonical random numbers $E_1, E_2, E_3$
    • Just means the number is in the range [0, 1)
  • **Step 2:** Set your possible random vector to the following:
    
    $$
    \begin{align*}
    v_x &= 2E_1 - 1 \\
    v_y &= 2E_2 - 1 \\
    v_z &= 2E_3 - 1
    \end{align*}
    $$
  • **Step 3:** If the dot product is less than 1 $\rightarrow$ normalize vector and keep it
  • Otherwise, go back to step 1
• Basically trying to get random points on the unit sphere
PERLIN NOISE

- The noise values by default are in the range $[-1, 1]$ → need to transform if using as output intensity or weights for interpolation
  - Also can play around with X and Y scale

- *For smoother output* → scale from $[0, 1]$
TURBULENCE

• Many natural textures contain a variety of feature sizes in the same texture
  • I.e., smaller details/perturbations in addition to the overarching pattern

• Perlin uses a pseudofractal “turbulence” function:

\[
\text{n}_i(P) = \sum_{i} \left[ \frac{n(2^i P)}{2^i} \right]
\]

• Basically repeatedly adds scaled copies of the function onto itself
TURBULENCE ON PERLIN NOISE

1 iteration (no turbulence)

2 iterations

4 iterations
ALPHA MAPPING
Often we have a texture we want to:
- Put on top of another object that’s already textured/colored \rightarrow \textbf{decals}
- Render on a single polygon to create the illusion of a complicated object (e.g., tree, plant) \rightarrow \textbf{used in billboard}

Use full alpha blending \rightarrow have to worry about order of rendering

Use \textbf{alpha testing} \rightarrow just check if alpha is 1 or 0
- Rendering order not a problem
- Can use 1-bit alpha map
- Can get aliasing on edges of decal
OPENGL/GLSL AND TEXTURING
To do basic 2D texturing in OpenGL/GLSL, we have to do a few things:

1) Get texture coordinates for the model; either:
   - A) Create buffer for texture coordinates, attach it to an attribute index, set up shaders
   - OR
   - B) Generate texture coordinates procedurally in the shader

2) Add sampler type to shaders

3) Get and store the uniform location of the sampler

4) Load texture image from a file, create texture object, and copy data in

5) Perform necessary calls in OpenGL to use texture for rendering
GLSL: SAMPLER2D

• GLSL uses the sampler* types for texture information
  • These have to be uniform variables
• In the fragment shader:
  • Add the sampler2D variable to the global section

```glsl
uniform sampler2D textureID;
```

• Get the color from the texture using your UV coordinates

```glsl
texColor = texture( textureID, UV);
```

vec2 UV;
...
GETTING THE SAMPLER LOCATION

• Since this is a uniform variable, we need to grab the location:

```cpp
uniformTextureID = glGetUniformLocation(shaderProgram, "textureID");
```
CREATING TEXTURE OBJECT

• To create a texture WITHOUT SOIL:
  • Need image loading code to get the image data in
    
    ```cpp
    glm::vec3 *imageData = new glm::vec3[textureWidth*textureHeight];
    ```
  • Create texture ID
  • Bind the texture ID to GL_TEXTURE_2D
    • ...since we're doing 2D texturing
      ```cpp
      GLuint textureID;
      glGenTextures(1, &textureID);
      glBindTexture(GL_TEXTURE_2D, textureID);
      ```
  • Allocate space for texture (and copy in data if we want)
    ```cpp
    glTexImage2D(GL_TEXTURE_2D, 0,
                 GL_RGB,
                 textureWidth,
                 textureHeight,
                 0, GL_RGB, GL_FLOAT,
                 imageData);
    ```
A CLOSER LOOK AT GLTEXIMAGE2D()

```c
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, textureWidth, textureHeight, 0, GL_RGB, GL_FLOAT, imageData);
// Target
// Mipmap level
// Internal format of texels (Red-Green-Blue color model)
// Width and height of texture
// Border around texture (setting to 0 because we don't want one)
// Format of texels (must match internal format)
// Data type of each texels
// Can also pass in NULL if we just want to allocate space
```
GLTEXIMAGE2D() VS. GLTEXSUBIMAGE2D

• `glTexImage2D()`
  • (Re)allocates memory
  • Also can copy in texture data

• `glTexSubImage2D()`
  • Just copies in memory
SETTING TEXTURE PARAMETERS

• While a texture is bound, there are several parameters you can set:

  • Filtering method
    - GL_NEAREST = nearest neighbor
    - GL_LINEAR = weight average of nearest 4 texels

  • Wrapping mode
    - GL_CLAMP_TO_EDGE
    - GL_CLAMP_TO_BORDER
    - GL_MIRRORED_REPEAT
    - GL_REPEAT
    - GL_MIRROR_CLAMP_TO_EDGE = mirror once, then clamp

```c
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```
MULTI-TEXTURING

- You can have multiple textures active at one time
  - You would need separate sampler variables for each one
  - You need to set the uniform value to the texture unit the sampler is associated with:
    
    ```
    glUniform1i(uniformTextureID, 0);
    ```
  - You also need to activate each texture unit you want to use and THEN bind the 2D texture to be associated with that texture unit
    
    ```
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, textureID);
    ```

IN YOUR RENDER LOOP

• Set the texture unit(s) you need to be active
• Bind the texture you want
  
  ```
  glEnableTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, textureID);
  ```
• After you activate the shader program, set the uniform value:
  
  ```
  glUniform1i(uniformTextureID, 0);
  ```
DELETING A TEXTURE

• To delete a texture:

```c
glBindTexture(GL_TEXTURE_2D, 0);
glDeleteTextures(1, &renderedTexture);
```