• **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 ($n_g$)
    • *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)
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
• 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
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
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 part of rendering process

• **Interactive, real-time** → use projector function to get \((u,v)\) coordinates for vertices, then interpolate values for each fragment
  - **EXCEPTIONS**: texture animation, environment mapping

• Note: NOT interpolated until we get to the 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 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:**
  - **Wrap** (DirectX) / **Repeat** (OpenGL) / **Tile**
  - **Mirror**
  - **Clamp** (DirectX) / **Clamp to Edge** (OpenGL)
  - **Border** (DirectX) / **Clamp to Border** (OpenGL)
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 \(\rightarrow\) 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 \(\rightarrow\) blends smoothly with border color
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)
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
MAGNIFICATION

- 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 → 320 x 320

Nearest-Neighbor

Bilinear interpolation

Cubic interpolation (5x5)
MAGNIFICATION:  
BILINEAR INTERPOLATION

- 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):

\[
b(p_u, p_v) = (1 - u')(1 - v')t(x_l, y_b) \\
+ u'(1 - v')t(x_r, y_b) \\
+ (1 - u')v't(x_l, y_t) \\
+ u'v't(x_r, y_t)
\]
MAGNIFICATION:
PROBLEM WITH BILINEAR INTERPOLATION

• 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)
This is when the pixel’s area covers several texels → have to integrate the effects of all texels

- Effectively impossible to do this completely correctly in real-time
- **Nearest-neighbor** → severe aliasing
- **Bilinear interpolation** → 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)
    - Downsample to a quarter of the area → Level 1 (subtexture of original)
    - Downsample Level 1 texture → Level 2 texture
    - Repeat...
- Mipmap chain = set of all the images
- d axis = level of detail (also called λ)
MINIFICATION: MIPMAPPING

• To do mipmapping properly, you need:
  • **Good filtering** → i.e., how you get your downsampling 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 \( d \) \( \rightarrow \) 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** → + 1/3 of original texture memory
  - **Ripmaps** → + 3x original texture memory
  - **Summed-area tables** → + at least 2x for texture of size 16 x 16 or less → 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 \( \rightarrow \) line of anisotropy
  • Take *multiple square mipmap samples* along line of anisotropy
    • Common number of samples: 2
    • Use shorter side of quad to get \( d \rightarrow \) less blurry
  • **Advantages:**
    • Line of anisotropy can run in any direction \( \rightarrow \) 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

- **Tangent space basis** = normal, tangent, and bitangent (binormal) vectors
  - Tangent and bitangent \(\rightarrow\) axes of normal map in object space
  - Need all three at each vertex
    - Store them all explicitly
    - OR
    - Store tangent and bitangent \(\rightarrow\) compute normal using cross product
      - Issue of "handedness" \(\rightarrow\) 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 \((0,0,1)\)

\[
\begin{bmatrix}
  t_x & t_y & t_z & 0 \\
  b_x & b_y & b_z & 0 \\
  n_x & n_y & n_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]
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

Heightfield
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:
    - \([-1,1] \rightarrow [0, 255]\)
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
    - \( v = \) normalized view vector → \( v_z = \text{height}, v_{xy} = \text{horizontal component} \) (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

• 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
NORMAL VS. RELIEF MAPPING

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

(a) Normal mapping
(b) Parallax mapping
(c) Relief 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
OTHER KINDS OF 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
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
PROCEDURAL TEXTURING

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

[Image of a wood-textured sphere with parameters for Ring Scale, Waviness, and Wave Scale]

[Links to shader documentation: http://modo.docs.thefoundry.co.uk/modo/701/help/pages/shaderrendering/ShaderItems/Wood.html]
• Often we have a texture we want to:
  • Put on top of another object that’s already textured/colored → decals
  • Render on a single polygon to create the illusion of a complicated object (e.g., tree, plant) → used in billboard

• Use full alpha blending → have to worry about order of rendering
• Use alpha testing → 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