Texture mapping and compositing

Coarse vs. fine modeling

- Coarse models (both geometry and color)
  - not realistic, obviously computer generated
  - fast to draw
  - easy to model
- Detailed models
  - models may look less blocky, so look more natural
  - slow to draw
  - hard to model (unless you scan real objects)
- Texture mapping takes the best of both
  - the base models don’t need as much detail
  - texturing (either color or geometry) dramatically increases realism
  - relatively easy modeling and fast drawing

Some images, such as the one on this page, come from "Teaching Texture Mapping Visually" by R. Wolfe
http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe/r_wolfe_mapping_1.htm

Parametric vs. nonparametric

- Nonparametric texturing
  - just show a portion of image through the shape
- Parametric texturing
  - attach the image onto the shape
- It’s really about the coordinate system of texture
  - in object’s local coordinates (parametric)
  - or in world coordinates (non-parametric)

Texture coordinates

- ST-coordinate system [0,1]x[0,1] covers the image
- Need a mapping from objects to texture
  - (x,y,z)->(s,t)
- Assign a texture coordinate for each vertex
  - attaches the corresponding image point to the vertex
- What if tex coords outside [0,1]?
  - need to specify whether texture wraps or clamps
Parameterization onto surfaces

- Find a mapping from a surface \((x, y, z)\) to \((s, t) [0,1] \times [0,1]\)

  - Plane
    \[
    (x, y, z) = \left(\frac{x - x_0}{x_1 - x_0}, \frac{y - y_0}{y_1 - y_0}\right)
    \]
    \[
    (s, t) = \left(\frac{x - x_0}{x_1 - x_0}, \frac{y - y_0}{y_1 - y_0}\right)
    \]

  - Sphere
    \[
    (x, y, z) = (\cos \phi \sin \theta, \sin \phi \sin \theta, \cos \theta)
    \]
    \[
    (s, t) = \left(\frac{\tan^{-1}(y/x)}{2\pi}, \frac{\cos^{-1}(z)}{\pi}\right)
    \]

  - Cylinder
    \[
    (x, y, z) = (\cos \theta, \sin \theta, z)
    \]
    \[
    (s, t) = \left(\frac{\theta}{2\pi}, \frac{z - z_0}{z_1 - z_0}\right)
    \]

- Mesh assign values at vertices

Two-stage mapping

- Figuring out the \((x, y, z)\rightarrow(s, t)\) mapping can be difficult
- Map the texture first to a simple shape: plane, sphere, cylinder, cube
- Then choose a parameterization between shapes
  - For each point, use:
    - position relative to bounding box
    - follow normal, where does it meet shape?
    - extend vector from centroid to meet shape
    - reflect vector by normal view point dependent!

Different mappings

- Plane
  - \(x \rightarrow s\)
  - \(y \rightarrow t\)

- Cylinder
  - \(xyz \rightarrow r, \theta, z\)
  - \(\theta \rightarrow s\)
  - \(z \rightarrow t\)

- Sphere
  - \(\text{latitude} \rightarrow s\)
  - \(\text{longitude} \rightarrow t\)

Different parametrizations

- Plane
- Cylinder
- Sphere
- Shape
Reflections from environment map

- Ray tracing can do reflections, OpenGL cannot
- Map a view of the environment to sphere or cube, then to object
- Difference to ray tracing
  - much faster
  - works if object planar or convex enough and not too many straight lines in the scene
  - usually approximation is good enough

Spherical environment maps

- Important observation
  - for correct results you need correct view point, contact point at the surface, normal vector
  - but if the reflecting object is small enough compared to the distance to the environment, the incoming light only depends on the direction of the reflected ray
  - ignore the actual surface position
  - use for all points same eye ray, use the normal ray to calculate reflection ray
  - access precomputed reflections from an environment map

Spherical maps

- Obtain a spherical map by taking a photo
  - with orthographic projection
- Problematic sampling
  - resolution of the texture varies spatially
    - samples close to boundary cover large area
    - singularity: at the boundary all pixels correspond to the direction behind the sphere
  - have to recreate the map for each viewpoint

Cubical environment maps

- Easy to obtain:
  - rendering system
  - photograph
**Cubical...**

- **Good things**
  - resolution is more uniform
    - varies at most by a factor of $3\sqrt{3} \approx 5.2$
  - don’t need to redo for each frame
- **HW implementation difficult**
  - need to choose which of the six maps to access
  - interpolation across maps difficult
- **But there is some HW**
  - accessible through OpenGL extension
- **How is it done otherwise?**
  - resample from the cube map to spherical map
  - need to redo every frame if the camera moves

**Dual parabolic environment maps**

- Instead of one spherical map
  - use two parabolic maps
    - front-facing and back-facing
  - get even better sampling than in cube maps (only a factor of 4)
- **Implementation**
  - won’t give details here, but…
  - render twice
    - or use multitexturing
  - ignore the parts outside of circles using alpha testing
  - if assume camera is far (and object not too flat)
    - can do all using linear matrix transforms
    - otherwise need to use glGenTex() to generate spherical reflectance coords

**Filtering textures**

- Project texture on image
  - how does projection relate to texels?
- **Projected area is larger**
  - we need to “magnify” the texel
  - just point sampling, or
  - bilinear interpolation for smoother output
    (here along both rows, then between them)
- **Projected area is smaller**
  - we need to “minify” the texels corresponding to the pixel
  - can also use point sampling
  - but filtering is much better (e.g., averaging)

**Aliasing with textures**

- **Point sampling creates artifacts due to aliasing**
  - jaggies nearby
  - random patterns far away
- **Better filtering**
  - first filter to remove high frequency data
  - then sample

Heidrich & Seidel 1998
Graphics HW Workshop
Mip mapping

- MIP = multum in parvo
  = many things in a small place
- Precalculated filtering for minification
  - usually done by successive averaging
  - Gaussians would give better result
- Trilinear interpolation
  - first with mipmap levels
  - then across levels

Calculating mipmap level

- Calculate mipmap level so that a unit step in pixels corresponds to a unit step in texels

\[ \log_2 \left( \max \left( \frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial y} \right) \right) \]

Example of mipmap level calculation

- Environment map on a teapot
- Approximately 9 texels will contribute to 1 pixel
- Get the texels from levels surrounding 9:1, that is, from level 1 (4:1) and 2 (16:1)
- Linearly interpolate those

Quality vs. time

- The cheapest “filtering” is no filtering
  - just do point sampling, that is, choose the texel that is nearest to current the pixel
- The next cheapest is to use prefiltering
  - choose the closest mipmap level, perform point sampling there
- The next one is to do bilinear interpolation
  - can actually do for both minification and magnification
  - need to fetch 4 texels => more memory accesses => slower
- Combine nearest mipmap and bilinear interpolation
  - a bit more calculation, again fetch 4 texels
- Trilinear filtering gives best results, but has the highest cost
  - calculate mipmap, do bilinear filtering on both, then linear
  - need to fetch 8 texels, big burden on memory bandwidth!
Non-parametric texturing

- You can do a block copy into the frame buffer
  - first, need to set raster position
    `glRasterPos3(x, y, z)`
  - if that projects within viewport, the projected location is the reference point (lower left corner) for
    `glDrawPixels(w, h, format, type, data)`
- You can also read from the buffers
  - `glReadPixels(x, y, w, h, format, type, data)`
  - notice that now the reference is given as the 2 first args
- Various pixel transfer modes affect these operations
  - `glPixelStore()`, `glPixelTransfer()`, `glReadBuffer()`, `glDrawBuffer()`

Parametric texturing in OpenGL

- `glEnable(GL_TEXTURE_2D)`
  - recall also to disable when you don't want textures
- `glTexImage2d(target, level, internalformat, width, height, border, format, type, data)`
  - `target` = `GL_TEXTURE_2D`
  - `level` = MIPMAP level, start from 0
  - `internalformat` = 38 choices, e.g., `GL_RGB`
  - `width`, `height`, `border` = sizes must be \( b^m + 2b \)
  - `format, type` = same as with `glDrawPixels()`, e.g., `GL_RGBA`, `GL_UNSIGNED_BYTE`
  - `data` = the raw image data (pixels)

Texturing commands in OpenGL

- There is a helper function for setting the mipmaps
  - `gluBuild2DMipmaps()`
- The pixel transfer modes affect texturing the same as with `glDrawPixels()`, so `glPixelStore()` and `glPixelTransfer()` apply
  - e.g., don't expect that the image data would be word (4 byte) aligned:
    - `glPixelStore(GL_UNPACK_ALIGNMENT, 1)`
- `glTexCoord2(s, t)`
  - for setting up the texture coordinates
- `glTexParameteri(target, pname, value)`
  - `target` = `GL_TEXTURE_2D`
  - `pname + value`:
    - `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`
      - `GL_CLAMP`, `GL_REPEAT`
    - `GL_TEXTURE_MAG_FILTER` magnify a texel over many pixels
      - `GL_NEAREST`, `GL_LINEAR`
    - `GL_TEXTURE_MIN_FILTER` minify many texels into a pixel
      - `GL_NEAREST`, `GL_LINEAR`, `GL_X_MIPMAP_Y`, `GL_NEAREST_MIPMAP_LINEAR`
    - `GL_TEXTURE_BORDER_COLOR` [R,G,B,A] = [0,0,0,0]
    - `GL_TEXTURE_PRIORITY` number clamped to \([0,1]\)
Texturing modes

- Terms
  - $C_f, A_f$ color and alpha of incoming fragment (underlying object)
  - $C_t, A_t$ color and alpha of a texel
  - $C_e$ "texture environment color"

- Decal
  - $C_t(1-A_t)+C_eA_t, A_f$
  - blend a partially transparent label over the surface
  - texture cannot make holes

- Replace
  - $C_t, A_t$
  - paste texture on the object, overriding color and transparency
  - texture can make holes

- Modulate
  - $C_tC_t, A_tA_t$
  - object's color is modulated or filtered with the texture map

- Blend
  - $C_f(1-C_t)+C_eC_t, A_tA_t$
  - no env color: modulate with inverse texture
  - env color: use texture a bit like alpha to blend object and env colors
  - can create many texture maps from a single texture image

Texturing commands in OpenGL

- `glTexEnv(target, pname, value)`
  - `target = GL_TEXTURE_ENV`
  - `pname + value`:
    - `GL_TEXTURE_ENV_MODE`
      - `GL_DECAL` — add a (transparent) label on top of object
      - `GL_REPLACE` — just replace
      - `GL_MODULATE` — modulate object’s color with texture
      - `GL_BLEND` — blend object color and env. color by texture
    - `GL_TEXTURE_ENV_COLOR` — [R,G,B,A] (only with `GL_BLEND`) [0,0,0,0]

Texturing example

Correct Perspective

Incorrect Perspective

Linear interpolation doesn't work

Correct Perspective

Incorrect Perspective

Linear interpolation doesn't work, need perspective interpolation

http://www.mic.atr.co.jp/~gulliver/PyOpenGL/
**Perspective correction**

- Interpolate A,B by $f$ in screen, what are texture coordinates?
- Homogeneous coordinates to help:
  - $\text{homog} \rightarrow \text{div } z \rightarrow \text{lerp} \rightarrow \text{div } w$

**Example**

- $A = (s,1)$
- divide by $z$: $A' = (s/z,1/z)$
- same for B
- linearly interpolate $C' = (s',q) = \text{lerp}(A',B',f)$
- homogeneous divide $C = (s'/q,1)$

**Expanded for 2D:**

$$s = \left(1 - f\right)\frac{s_a}{z_a} + f\frac{s_b}{z_b}$$
$$t = \left(1 - f\right)\frac{t_a}{z_a} + f\frac{t_b}{z_b}$$

**Bump mapping**

- Want to model a strawberry?
- don't want too much detail
- textures will work only with static lights and view point
- how to capture changing shadows and highlights when lights or view point moves?

- Map a normal perturbation onto surface
  - taken into account in shading
  - bump mapping

- Where/when do we see this is fake?

Silhouettes remain smooth

**Bump mapping maths**

parametric surface $p(u, v)$

$$n = \frac{p_u \times p_v}{|p_u \times p_v|}$$

suppose we perturb surface by $p' = p + d(u, v)n$

new normal $n' = p'_u \times p'_v$, where

$$p'_u = p_u + \frac{\partial d}{\partial u}n + d(u,v)n_u$$
$$p'_v = p_v + \frac{\partial d}{\partial v}n + d(u,v)n_v$$

with small d, we get $n' = n + \frac{\partial d}{\partial u}p_v \times n + \frac{\partial d}{\partial v}p_u \times n$

precalculate and store $\frac{\partial d}{\partial u}p_v \times n$ and $\frac{\partial d}{\partial v}p_u \times n$ for each texel

if the perturbations also change geometry, it's called displacement mapping

**Solid textures**

- Define a texture in 3D
  - need also 3D coordinates
  - use scaled and shifted $(x,y,z)$ surface coordinate as the texture coordinate
  - avoid distortions, and discontinuities between 2D-texture patches

- Represent as
  - voxel grid
  - stack of images
  - procedural

"Carve the shape out of the texture matter"
Glass knight in "Young Sherlock Holmes"

Layered effects

(a) The basic shape of the shoulder guard, with a color map
(b) The piece with an environment map
(c) The environment map modified by a bump map and illumination function
(d) Combine (a) and (c), add some dirt
(e) Add more details for the seams and rivets, add an extra specular layer
(f) A detail

Uses of alpha channel

- Transparency
- Blending and color filtering
- Anti-aliasing
- Compositing
The alpha channel

- In addition to RGB, we store an alpha value for every pixel
  - the set of alpha values for an image is called the alpha channel

- Two interpretations for $\alpha$
  - coverage: how large portion of a pixel is covered
  - opaqueness: how much light is blocked to pass through a pixel (opaqueness = 1-transparency)

- Values
  - transparent / no coverage when $\alpha = 0$
  - opaque / full coverage when $\alpha = 1$
  - otherwise partially transparent / covered

- Relationship between $\alpha$ and RGB:
  - computed at same time
  - need comparable resolution
  - can manipulate in almost exactly the same way

Transparency in OpenGL

- Draw opaque objects first
- Then draw transparent objects
  - sorted, farthest first
  - enable blending and set the blending function:
    - glEnable(GL_BLEND)
    - glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
  - Creates a blend
    - incoming pixel (src) gets weight of incoming alpha $\alpha$
    - $1-\alpha$ of the pixel in the frame buffer (dst) remains

How does blending work?

- Source
  - (Rs, Gs, Bs, As)
  - the new value
- Destination
  - (Rd, Gd, Bd, Ad)
  - in the frame buffer
- Blending factors
  - (Sr, Sg, Sb, Sa)
  - (Dr, Dg, Db, Da)
- Result
  - multiply with factors, add
    - (Rs Sr, Gs Sg, Bs Sb, As Sa) + (Rd Dr, Gd Dg, Bd Db, Ad Da)
    - = (RsSr+Rd Dr, GsSg+Gd Dg, BsSb+Bd Db, AsSa+Ad Da)

Blending factors

- Set both for source and destination
- glBlendFunc(sfactor, dfactor)
  - GL_ZERO s,d (0,0,0,0)
  - GL_ONE s,d (1,1,1,1)
  - GL_SRC_ALPHA s,d (As, As, As, As)
  - GL_DST_ALPHA s,d (Ad, Ad, Ad, Ad)
  - GL_ONE_MINUS_SRC_ALPHA s,d (1-As, 1-As, 1-As, 1-As)
  - GL_ONE_MINUS_DST_ALPHA s,d (1-Ad, 1-Ad, 1-Ad, 1-Ad)
  - GL_SRC_COLOR d (Rs, Gs, Bs, As)
  - GL_DST_COLOR s (Rd, Gd, Bd, Ad)
  - GL_ONE_MINUS_SRC_COLOR d (1-Rs, 1-Gs, 1-Bs, 1-As)
  - GL_ONE_MINUS_DST_COLOR s (1-Rd, 1-Gd, 1-Bd, 1-Ad)
  - GL_SRC_ALPHA_SATURATE s (i, i, i, 1)
    - $i = \min(As, 1-Ad)$
**Blending examples**

- **Blend 2 images 50% : 50%**
  - draw first image with
    src factor `GL_ONE` \((1,1,1,1)\)
    dst factor `GL_ZERO` \((0,0,0,0)\)
  - draw the second image with an alpha of \(.5\)
    src factor `GL_SRC_ALPHA` \((.5, .5, .5, .5)\)
    dst factor `GL_SRC_ALPHA` \((.5, .5, .5, .5)\)

- **Blend 2 images 75% : 25%**
  - draw first image with
    src factor `GL_ONE` \((1,1,1,1)\)
    dst factor `GL_ZERO` \((0,0,0,0)\)
  - draw the second image with an alpha of \(.25\)
    src factor `GL_SRC_ALPHA` \((.25, .25, .25, .25)\)
    dst factor `GL_ONE_MINUS_SRC_ALPHA` \((.75, .75, .75, .75)\)

**Blending examples**

- **Blend \(n\) images equally**
  - draw all images with alpha \(1/n\)
    src factor `GL_SRC_ALPHA` \((1/n, 1/n, 1/n, 1/n)\)
  - first image
    dst factor `GL_ZERO` \((0,0,0,0)\)
  - other images
    dst factor `GL_ONE` \((1,1,1,1)\)
  - gradually adds up

- **Apply a color filter over an image**
  - Draw filter image \((R_f, G_f, B_f, 1)\) with
    src factor `GL_ZERO` \((0,0,0,0)\)
    dst factor `GL_SRC_COLOR` \((R_f, G_f, B_f, 1)\)
  - result: \((R_d, R_f, G_d, G_f, B_d, B_f, A_d)\)

**Compositing**

- **Sometimes, a single image needs to be constructed out of parts**
  - mixing 3D graphics with film
  - adding a backdrop to a scene
  - painting objects into a scene

- **Sometimes, it’s just better to do things in parts**
  - can save time in ray tracing
  - a small problem in one part can easily be fixed in the final image

- **Need a method for building up an image from a set of components**
  - ideally, invent a general “algebra” of **compositing**
**Aliasing in compositing**

- Assemble images from parts, each part has a **matte**
  - record whether a pixel belongs to the foreground / background
  - discard background pixels when assembling

- **Problem**
  - at the boundary, pixels have only **partial coverage**
  - the matte must record more than a single bit of information per pixel

- **Solution**
  - determine coverage and use alpha

**General compositing operator**

- **Notation**
  - compositing with operator \( \oplus \)
  - two images \( F, G \), with factors \( \alpha, \beta \), over background \( B \)

- Can you composite images in arbitrary order?
  - \( G \oplus (F \oplus B) = (G \oplus F) \oplus B? \) (associativity)
  - Left side: add layers one at a time over background
  - Right side: composite two layers first, then lay over the background

- Calculate \( H = (G \oplus F) \) and the corresponding factor \( \gamma \)
  - \( \beta G + (1-\beta) (\alpha F + (1-\alpha)B) = (\beta G + (1-\beta)\alpha F) + (1-\beta) (1-\alpha) B \)
  - first solve \( \gamma \) \( (1-\beta) (1-\alpha) = (1-\gamma) \) \( \Rightarrow \gamma = \beta + (1-\beta)\alpha \)
  - then solve \( H \): \( H = (\beta G + (1-\beta)\alpha F) / \gamma \)

- Yes, you can, but alpha and colors are calculated differently

**Simpler definition**

- The "normal" interpretation of \((R,G,B,\alpha)\) is that the pixel has color \(R,G,B\) with transparency \(\alpha\)
  - postmultiplied alpha
- For compositing we should premultiply the color with alpha:
  - \((R, G, B, \alpha) \Rightarrow (\alpha R, \alpha G, \alpha B, \alpha)\)
- Now, if we premultiply \(F, G,\) and \(H\), (denote with underline) we get
  - \(H = G + (1-\beta)F\)
  - \(\gamma = \beta + (1-\beta)\alpha\)
- So much simpler! And alpha and color are treated equally!
  - And composition can be done before combining the result over a background.
- What do premultiplied \((0,0,0,1), (0,0,0,0), (.5,.25,.25,.5)\) represent?
  - opaque black
  - fully transparent
  - half transparent bright pink
  - \((1,.5,.5)\)

**Some useful compositing operations**

- **A over B**
  - foreground A over background B
- **A in B**
  - A inside the picture formed by B
- **A out B**
  - out = held out by
  - show only the part of A outside of B
- Of course the same reversed
  - and some other bit less useful operations which are really combinations of those three
Compositing possibilities

- The contributions of A and B divide the area into 4 regions
  - choose what will be visible in each region

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Possibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A \cap \neg B</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>A \cap \neg B</td>
<td>0, A</td>
</tr>
<tr>
<td>B</td>
<td>\neg A \cap B</td>
<td>0, B</td>
</tr>
<tr>
<td>AB</td>
<td>A \cap B</td>
<td>0, A, B</td>
</tr>
</tbody>
</table>

- According to this enumeration, how many binary compositing operators are there? 12 = 1 x 2 x 2 x 3

Some additional operators

- **darken**(A, t) := (tR, tG, tB, α), 0 <= t <= 1
  - just change the color

- **opaque**(A, t) := (R, G, B, tα), 0 <= t <= 1
  - just change transparency

- **fade**(A, t) := (tR, tG, tB, tα), 0 <= t <= 1
  - fade is a combination of darken and opaque

- **plus**(A, B) := (Ra+Rb, Ga+Gb, Ba+Bb, αa+αb)
  - combine two images

- Example of use:
  - blend smoothly between A and B: **fade**(A, t) **plus** fade(B, 1-t)

Example: Genesis effect

(FFire **plus** (BFire out Planet))
over **darken**(Planet, 0.8)
over Stars
**Example: Making of 405, the movie**

- www.405themovie.com/makingofHome.asp
  - viewing the film and reading the story (8 web pages) is considered required reading (not replicated here)

---

**Multipass rendering, multitexturing**

- Idea: you can generate many special effects and more sophisticated lighting models by rendering in several passes
- Starting from OpenGL 1.3, OpenGL also supports multitexturing
  - but you can emulate it with multipassing in earlier versions
  - render on top of prev. image, or to off-screen buffers and combine
- Example:
  - base texture
  - labels (Coated, Bruns, Circle)
  - dirt (Marks)
  - diffuse lighting (Cd)
  - specular lighting (Cs)
  - composite: Circle over (Bruns over (Coated over Base)) * Marks * Cd + Cs

---

**Review of OpenGL pipeline**

1. **Geometry**
   - Vertex data
   - Floating point

2. **Rasterizer**
   - Pixel data
   - Fixed point

3. **Model & view transform**

4. **Projection**

5. **Clipping**

6. **Persp. divide**

7. **Screen mapping**

8. **Scan conversion**

9. **Visibility solving**

10. **Texture mapping**

11. **Blending, etc.**

12. **Object coordinates**

13. **Eye coordinates**

14. **Clip coordinates**

15. **Norm.device coords**

16. **Screen coordinates**

17. **Single pixel**

18. **Frame buffer**