Texture Mapping

Computer Graphics
CMU 15-462/15-662
Texture Mapping
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

Wood grain on floor
Describe surface material properties

Multiple layers of texture maps for color, logos, scratches, etc.
Normal & Displacement Mapping

normal mapping

Use texture value to perturb surface normal to “fake” appearance of a bumpy surface

displacement mapping

dice up surface geometry into tiny triangles & offset positions according to texture values (note bumpy silhouette and shadow boundary)
Represent precomputed lighting and shadows

Grace Cathedral environment map

Environment map used in rendering
Texture coordinates

- "Texture coordinates" define a mapping from surface coordinates to points in texture domain.
- Often defined by linearly interpolating texture coordinates at triangle vertices.

Suppose each cube face is split into eight triangles, with texture coordinates \((u,v)\) at each vertex.

A texture on the \([0,1]^2\) domain can be specified by a 2048x2048 image.

Linearly interpolating texture coordinates & “looking up” color in texture gives this image:

(location of highlighted triangle in texture space shown in red)
Visualization of texture coordinates

Associating texture coordinates \((u, v)\) with colors helps to visualize mapping

\[(0,1) \quad \text{green} \]
\[(1,0) \quad \text{red} \]

black \((0,0)\)
More complex mapping

Visualization of texture coordinates

Triangle vertices in texture space

Each vertex has a coordinate \((u,v)\) in texture space

(Actually coming up with these coordinates is another story!)
Texture mapping adds detail

Each triangle “copies” a piece of the image back to the surface
Texture mapping adds detail

- Rendering without texture
- Rendering with texture
- Texture image

Zoom
Another example: periodic coordinates

Q: Why do you think texture coordinates might repeat over the surface?
Textured Sponza

A: Want to tile a texture many times (rather than store a huge image!)
Texture Sampling 101

Basic algorithm for texture mapping:
- for each pixel in the rasterized image:
  - interpolate $(u, v)$ coordinates across triangle
  - sample (evaluate) texture at interpolated $(u, v)$
  - set color of fragment to sampled texture value

...sadly not this easy in general!
Recall: aliasing

Undersampling a high-frequency signal can result in aliasing.

\[ f(x) \]

\[ x \]
Visualizing texture samples

Since triangles are projected from 3D to 2D, pixels in screen space will correspond to regions of varying size & location in texture.

Sample positions are uniformly distributed in screen space (rasterizer samples triangle’s appearance at these locations).

Sample positions in texture space are not uniform (texture function is sampled at these locations).

Irregular sampling pattern makes it hard to avoid aliasing!
Magnification vs. Minification

- **Magnification (easier):**
  - Example: camera is very close to scene object
  - Single screen pixel maps to tiny region of texture
  - Can just interpolate value at screen pixel center

- **Minification (harder):**
  - Example: scene object is very far away
  - Single screen pixel maps to large region of texture
  - Need to compute average texture value over pixel to avoid aliasing
Bilinear interpolation (magnification)

How can we “look up” a texture value at a non-integer location \((u, v)\)?

**Fast but ugly:**
just grab value of nearest “texel” (texture pixel)

**Linear (each row):**

\[
\begin{align*}
(1-s)f_{01} + sf_{11} \\
(1-s)f_{00} + sf_{10}
\end{align*}
\]

**Bilinear:**

\[
(1-t)((1-s)f_{00} + sf_{10}) + t((1-s)f_{01} + sf_{11})
\]

Q: What happens if we interpolate vertically first?
Aliasing due to minification
“Pre-filtering” texture (minification)
Texture prefiltering — basic idea

- Texture aliasing often occurs because a single pixel on the screen covers many pixels of the texture.

- If we just grab the texture value at the center of the pixel, we get aliasing (get a “random” color that changes if the sample moves even very slightly).

- Ideally, would use the average texture value—but this is expensive to compute.

- Instead, we can pre-compute the averages (once) and just look up these averages (many times) at run-time.

But which averages should we store? Can’t precompute them all!
Prefiltered textures

Actual texture: 700x700 image
(only a crop is shown)

Actual texture: 64x64 image

Q: Are two resolutions enough?  A: No…
MIP map (L. Williams 83)

- Rough idea: store prefiltered image at “every possible scale”
- Texels at higher levels store average of texture over a region of texture space (downsampled)
- Later: look up a single pixel from MIP map of appropriate size
Mipmap (L. Williams 83)

Williams' original proposed mip-map layout

“Mip hierarchy”
level = \( d \)

Q: What's the storage overhead of a mipmap?
Computing MIP Map Level

Even within a single triangle, may want to sample from different MIP map levels:

Q: Which pixel should sample from a coarser MIP map level: the blue one, or the red one?
Computing Mip Map Level

Compute differences between texture coordinate values at neighboring samples

\[
\frac{du}{dx} = u_{10} - u_{00} \quad \frac{dv}{dx} = v_{10} - v_{00}
\]

\[
\frac{du}{dy} = u_{01} - u_{00} \quad \frac{dv}{dy} = v_{01} - v_{00}
\]

\[
L_x^2 = \left( \frac{du}{dx} \right)^2 + \left( \frac{dv}{dx} \right)^2 \quad L_y^2 = \left( \frac{du}{dy} \right)^2 + \left( \frac{dv}{dy} \right)^2
\]

\[
L = \sqrt{\max(L_x^2, L_y^2)}
\]

mip-map level: \( d = \log_2 L \)
Visualization of mip-map level

d clamped to nearest level
Sponza (bilinear resampling at level 0)
Sponza (bilinear resampling at level 2)
Sponza (bilinear resampling at level 4)
Sponza (MIP mapped)

nicely filters the background

retains detail in the foreground
Problem with basic MIP mapping

- If we just use the nearest level, can get artifacts where level “jumps”—appearance sharply transitions from detailed to blurry texture

- **IDEA:** rather than clamping the MIP map level to the closest integer, use the original (continuous) MIP map level $d$

- **PROBLEM:** we only computed a fixed number of MIP map levels. How do we interpolate between levels?
Trilinear Filtering

- Used **bilinear** filtering for 2D data; can use **trilinear** filtering for 3D data

- Given a point \((u, v, w) \in [0, 1]^3\), and eight closest values \(f_{ijk}\)

- Just iterate linear filtering:
  - weighted average along \(u\)
  - weighted average along \(v\)
  - weighted average along \(w\)

\[
\begin{align*}
g_{00} &= (1 - u)f_{000} + uf_{100} \\
g_{10} &= (1 - u)f_{010} + uf_{110} \\
g_{01} &= (1 - u)f_{001} + uf_{101} \\
g_{11} &= (1 - u)f_{011} + uf_{111} \\
h_0 &= (1 - v)g_{00} + vg_{10} \\
h_1 &= (1 - v)g_{01} + vg_{11} \\
(1 - w)h_0 + wh_1 &= (1 - w)(1 - v)g_{00} + w(vg_{10} + vg_{11})
\end{align*}
\]
MIP Map Lookup

- MIP map interpolation works essentially the same way
  - not interpolating from 3D grid
  - interpolate from two MIP map levels closest to $d \in \mathbb{R}$
  - perform bilinear interpolation independently in each level
  - interpolate between two bilinear values using $w = d - \lfloor d \rfloor$

Starts getting expensive! (⇒ specialized hardware)

Bilinear interpolation:
  four texel reads
  3 linear interpolations (3 mul + 6 add)

Trilinear/MIP map interpolation:
  eight texel reads
  7 linear interpolations (7 mul + 14 add)
Anisotropic Filtering

At grazing angles, samples may be stretched out by (very) different amounts along \( u \) and \( v \)

Overblurring in \( u \) direction

Common solution: combine multiple MIP map samples (even more arithmetic/bandwidth!)

Overblurring in \( v \) direction

texture space viewed from camera w/ perspective projection
Texture Sampling Pipeline

1. Compute $u$ and $v$ from screen sample $(x, y)$ via barycentric interpolation
d
4. Convert normalized $[0, 1]$ texture coordinate $(u, v)$ to pixel locations $(U, V) \in [W, H]$ in texture image
3. Compute mip map level $d$
5. Determine addresses of texels needed for filter (e.g., eight neighbors for trilinear)
6. Load texels into local registers
7. Perform tri-linear interpolation according to $(U, V, d)$
8. (…even more work for anisotropic filtering…)

Takeaway: high-quality texturing requires far more work than just looking up a pixel in an image! Each sample demands significant arithmetic & bandwidth

For this reason, graphics processing units (GPUs) have dedicated, fixed-function hardware support to perform texture sampling operations.
Moving to: Depth & Transparency
Depth and Transparency
Wrap up the rasterization pipeline!

**Remember our goal:**
- Start with INPUTS (triangles)
  - possibly w/ other data (e.g., colors or texture coordinates)
- Apply a series of transformations: STAGES of pipeline
- Produce OUTPUT (final image)

**INPUT (TRIANGLES)**

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Input Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(1, 1, 1)</td>
</tr>
<tr>
<td>B</td>
<td>(-1, 1, 1)</td>
</tr>
<tr>
<td>C</td>
<td>(1, -1, 1)</td>
</tr>
<tr>
<td>D</td>
<td>(-1, -1, 1)</td>
</tr>
<tr>
<td>E</td>
<td>(1, 1, -1)</td>
</tr>
<tr>
<td>F</td>
<td>(-1, 1, -1)</td>
</tr>
<tr>
<td>G</td>
<td>(1, -1, -1)</td>
</tr>
<tr>
<td>H</td>
<td>(-1, -1, -1)</td>
</tr>
</tbody>
</table>

**Triangles**
- EHF, GFH, FGB, CBG
- GHC, DCH, ABD, CDB
- HED, ADE, EFA, BAF

**Rasterization Pipeline**

**Output (Bitmap Image)**
What we know how to do so far...

position objects in the world (3D transformations)

project objects onto the screen (perspective projection)

sample triangle coverage (rasterization)

sample texture maps (filtering, mipmapping)

interpolate vertex attributes (barycentric coordinates)

put samples into frame buffer (depth & alpha)
Occlusion
Occlusion: which triangle is visible at each covered sample point?

Opaque Triangles

50% transparent triangles
Sampling Depth

Assume we have a triangle given by:
- the projected 2D coordinates $(x_i, y_i)$ of each vertex
- the “depth” $d_i$ of each vertex (i.e., distance from the viewer)

Q: How do we compute the depth $d$ at a given sample point $(x, y)$?

A: Interpolate it using barycentric coordinates—just like any other attribute that varies linearly over the triangle
The depth-buffer (Z-buffer)

For each sample, depth-buffer stores the depth of the closest triangle seen so far.

Initialize all depth buffer values to “infinity” (max value)
Depth buffer example
Example: rendering three opaque triangles
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5

---

Sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:

Color buffer contents

Depth buffer contents

— sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:
depth = 0.75

Color buffer contents

Depth buffer contents

near — sample passed depth test

far
Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:

<table>
<thead>
<tr>
<th>Color buffer contents</th>
<th>Depth buffer contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>near</td>
</tr>
<tr>
<td></td>
<td>— sample passed depth test</td>
</tr>
</tbody>
</table>

- After processing the blue triangle, the color buffer contents are updated accordingly.
- The depth buffer contents are also updated to reflect the depth values of the objects.
- The sample passed the depth test, indicating that it is not occluded by any other objects in the scene.
Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:
depth = 0.25

Color buffer contents

Depth buffer contents

— sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:

Color buffer contents

Depth buffer contents

near — sample passed depth test
Occlusion using the depth buffer

```c
bool pass_depth_test(d1, d2)
{
    return d1 < d2;
}
```

```c
void draw_sample(x, y, d, c) //new depth d & color c at (x,y)
{
    if( pass_depth_test( d, zbuffer[x][y] ) )
    {
        // triangle is closest object seen so far at this
        // sample point. Update depth and color buffers.
        zbuffer[x][y] = d; // update zbuffer
        color[x][y] = c; // update color buffer
    }
    // otherwise, we’ve seen something closer already;
    // don’t update color or depth
}
```
Q: Does depth-buffer algorithm handle interpenetrating surfaces?
A: Of course!

Occlusion test is based on depth of triangles at a given sample point. Relative depth of triangles may be different at different sample points.

Green triangle in front of yellow triangle

Yellow triangle in front of green triangle
Intersection

Q: Does depth-buffer algorithm handle interpenetrating surfaces?

A: Of course!

Occlusion test is based on depth of triangles at a given sample point. Relative depth of triangles may be different at different sample points.
Q: Does depth buffer work with super sampling?  
A: Yes! If done per (super) sample.

(Here: green triangle occludes yellow triangle)
Depth + Supersampling

Color of super samples after rasterizing w/ depth buffer
Color buffer contents (4 samples per pixel)
Final resampled result

Note anti-aliasing of edge due to filtering of green and yellow samples
Summary: occlusion using a depth buffer

- Store one depth value per (super) sample—not one per pixel!

- **Constant additional space per sample**
  - Hence, **constant space for depth buffer**
  - Doesn’t depend on number of overlapping primitives!

- **Constant time occlusion test per covered sample**
  - Read-modify write of depth buffer if “pass” depth test
  - Just a read if “fail”

- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point

But what about semi-transparent surfaces?
Compositing
Representing opacity as alpha

An “alpha” value $0 \leq \alpha \leq 1$ describes the opacity of an object.

- $\alpha = 1$ (fully opaque)
- $\alpha = \frac{3}{4}$
- $\alpha = \frac{1}{2}$
- $\alpha = \frac{1}{4}$
- $\alpha = 0$ (fully transparent)
Key idea: can use $\alpha$ channel to composite one image on top of another.
Fringing

Poor treatment of color/alpha can yield dark “fringing”:

- Foreground color
- Foreground alpha
- Background color

- Fringing
- No fringing
No fringing
Fringing (…why does this happen?)
Over operator:

Composites image $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

Informally, captures behavior of “tinted glass”

Notice: “over” is not commutative

$A$ over $B \neq B$ over $A$

Over operator: non-premultiplied alpha

Composite image $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

A first attempt:

$$A = (A_r, A_g, A_b)$$
$$B = (B_r, B_g, B_b)$$

**Composite color:**

$$C = \alpha_B B + (1 - \alpha_B) \alpha_A A$$

**Composite alpha:**

$$\alpha_C = \alpha_B + (1 - \alpha_B) \alpha_A$$
Over operator: premultiplied alpha

Composite image $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

Premultiplied alpha—multiply color by $\alpha$, then composite:

$$A' = (\alpha_A A_r, \alpha_A A_g, \alpha_A A_b, \alpha_A)$$

$$B' = (\alpha_B B_r, \alpha_B B_g, \alpha_B B_b, \alpha_B)$$

$$C' = B' + (1 - \alpha_B)A'$$

Notice premultiplied alpha composites alpha just like how it composites rgb.
(Non-premultiplied alpha composites alpha differently than rgb.)

“Un-premultiply” to get final color:

$$(C_r, C_g, C_b, \alpha_C) \implies (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)$$

Q: Does this division remind you of anything?
Compositing with & without premultiplied $\alpha$

Suppose we upsample an image with an $\alpha$ channel, then composite it onto a background:

Q: Why do we get the "green fringe" when we don’t premultiply?
Similar problem with non-premultiplied $\alpha$

Consider pre-filtering (downsampling) a texture with an alpha matte.

- Input color
- Input $\alpha$
- Filtered color
- Filtered $\alpha$
- Composited over white

- Premultiplied color
- Premultiplied $\alpha$
- Filtered color
- Filtered $\alpha$
- Composited over white
More problems: applying “over” repeatedly

Composite image $C$ with opacity $\alpha_C$ over $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

Premultiplied alpha is closed under composition; non-premultiplied alpha is not!

Example: composite 50% bright red over 50% bright red (where “bright red” = $(1,0,0)$, and $\alpha = 0.5$)

<table>
<thead>
<tr>
<th>Non-premultiplied</th>
<th>Premultiplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>Color</td>
</tr>
<tr>
<td>$0.5(1,0,0) + (1-0.5).5(1,0,0)$</td>
<td>$(0.5,0,0,.5)+(1-.5)(.5,0,0,.5)$</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$(0.75,0,0)$</td>
<td>$(0.75,0,0.75)$</td>
</tr>
<tr>
<td><strong>too dark!</strong></td>
<td>divide by $\alpha$</td>
</tr>
<tr>
<td>Alpha</td>
<td>Alpha</td>
</tr>
<tr>
<td>$0.5 + (1-0.5).5 = 0.75$</td>
<td>$\alpha = 0.75$</td>
</tr>
</tbody>
</table>
Summary: advantages of premultiplied alpha

- Compositing operation treats all channels the same (color and $\alpha$)
- Fewer arithmetic operations for “over” operation than with non-premultiplied representation
- Closed under composition (repeated “over” operations)
- Better representation for filtering (upsampling/downsampling) images with alpha channel
- Fits naturally into rasterization pipeline (homogeneous coordinates)
Strategy for drawing semi-transparent primitives

Assuming all primitives are semi-transparent, and color values are encoded with premultiplied alpha, here's a strategy for rasterizing an image:

```c
over(c1, c2)
{
    return c1.rgba + (1-c1.a) * c2.rgba;
}
```

```c
update_color_buffer( x, y, sample_color, sample_depth )
{
    if (pass_depth_test(sample_depth, zbuffer[x][y])
    {
        // (how) should we update depth buffer here??
        color[x][y] = over(sample_color, color[x][y]);
    }
}
```

Q: What is the assumption made by this implementation?

Triangles must be rendered in back to front order!
Putting it all together

What if we have a mixture of opaque and transparent triangles?

Step 1: render opaque primitives (in any order) using depth-buffered occlusion
If pass depth test, triangle overwrites value in color buffer at sample

Step 2: disable depth buffer update, render semi-transparent surfaces in back-to-front order.
If pass depth test, triangle is composited OVER contents of color buffer at sample
End-to-end rasterization pipeline
Goal: turn inputs into an image!

Inputs:

positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x
};

texcoords = {
    v0u, v0v,
    v1u, v1v,
    v2u, v2v,
    v3u, v3v,
    v4u, v4v,
    v5u, v5v
};

Object-to-camera-space transform $T \in \mathbb{R}^{4\times4}$

Perspective projection transform $P \in \mathbb{R}^{4\times4}$

Size of output image $(W, H)$

At this point we have all the tools we need to make an image... Let's review!
Step 1:
Transform triangle vertices into camera space
Step 2:

Apply perspective projection transform to transform triangle vertices into normalized coordinate space

Camera-space positions: 3D

Normalized space positions
Step 3: clipping

- Discard triangles that lie complete outside the unit cube (culling)
  - They are off screen, don’t bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
  - (possibly generating new triangles)
Step 4: transform to screen coordinates

Perform homogeneous divide, transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)
Step 5: setup triangle (triangle preprocessing)

Before rasterizing triangle, can compute a bunch of data that will be used by all fragments, e.g.,

- triangle edge equations
- triangle attribute equations
- etc.

\[
\begin{align*}
E_{01}(x, y) &= U(x, y) \\
E_{12}(x, y) &= V(x, y) \\
E_{20}(x, y) \\
\frac{1}{w}(x, y) \\
Z(x, y)
\end{align*}
\]
Step 6: sample coverage

Evaluate attributes $z$, $u$, $v$ at all covered samples
Step 6: compute triangle color at sample point
e.g., sample texture map *

*Not the only way to get a color! Later we’ll talk about more general models of materials…
Step 7: perform depth test (if enabled)

Also update depth value at covered samples (if necessary)
Step 8: update color buffer* (if depth test passed)

* Possibly using OVER operation for transparency
OpenGL/Direct3D graphics pipeline

Our rasterization pipeline doesn’t look much different from “real” pipelines used in modern APIs / graphics hardware.

1. Input: vertices in 3D space
2. Vertices in positioned in 3D normalized coordinate space
3. Triangles projected to 2D screen
4. Fragments (one fragment per covered sample)

* Several stages of the modern OpenGL pipeline are omitted
Goal: render very high complexity 3D scenes

- 100’s of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (~10Mpixel + supersampling)
- 30-120 fps
Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

- Discrete GPU card
- Smartphone GPU (integrated)
- Integrated GPU: part of modern CPU die
Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs ("compute")

This part (mostly) not used by CUDA/OpenCL; raw graphics horsepower still greater than compute!
Modern Rasterization Pipeline

- Trend toward more generic (but still highly parallel!) computation:
  - make stages programmable
  - replace fixed function vertex, fragment processing
  - add geometry, tessellation shaders
  - generic “compute” shaders (whole other story…)
  - more flexible scheduling of stages
Ray Tracing in Graphics Pipeline

- More recently: specialized pipeline for ray tracing (NVIDIA RTX)

GPU Ray Tracing Demo ("Marbles at Night")
What else do we need to know to generate images like these?

**GEOMETRY**

How do we describe complex shapes (so far just triangles...)

**RENDERING**

How does light interact w/ materials to produce color?

**ANIMATION**

How do we describe the way things move?

("Moana", Disney 2016)
Course roadmap

- Introduction
- Drawing a triangle (by sampling)
- Transforms and coordinate spaces
- Perspective projection and texture sampling
- Today: putting it all together: end-to-end rasterization pipeline

Geometry

Up next after the Midterm!