# Lecture 20?
# Drawing in 3D

# Recap...

- Lecture 17: Meshes and appearance

- "Lecture 17" (old 17/18): Texture Basics
  - object appearence by **color maps**
  - texture coordinates, lookups
  - filtering

- "Lecture 18" (old 18/19): Advanced Textures
  - object shape by bump, normal, displacement **maps**
  - tricks for maps (wrapping, scaling, layering, ...)
  - complex lighting by environment and shadow **maps**

# Today: Why/How?

- Why just triangles?

- Why maps not more triangles?

- Why **hacks** to get shape and lighting?

1. Really understand the model
    - need it to use it well / efficiently

    - need it to understand shaders

2. See some of the algorithms underneath
    - they explain why things work the way they do

# The Assumptions of Interactive 3D

1. Triangles (primitives)

2. Triangles are **independent** (Local Shading)

# What controls appearance

At a point:

    1. Surface Color (and properties)

    2. Light Color (and properties)

# Lighting: Local vs. Global

## Local:

- consider what happens at one point

- given what light arrives, what color do we see

## Global:

- what light gets to the point

- how do different points interact

# Global Lighting Effects

- Shadows

- Reflections (mirror)

- Spill

- Indirect lighting

- Refraction

- Complex combinations (e.g., caustics)

1. Simulate **transport**

2. Use Hacks

# Local Lighting Model

Consider one point

Each light contributes

- simple: each light gives 1 direction/colors

- complex: light over a range of directions

Bi-directional Reflectance Distribution Function (BRDF)

# Why local lighting?

Compute each triangle independently
(each point on each triangle)

Need:

- Light at point
- Color (surface properties at point)

**Never consider all triangles together**

# But why and how triangles?

# Triangles: the primitive

Possibly points and lines as well

1. Projection of a triangle is a triangle
    - just transform the vertices (insides might change)

2. Barycentric coordinates

3. Fast Algorithms to Draw

# Colors per triangle

- 1 (face, split vertices)

- 3 (vertex, interpolation)

- lots (texture)

# Why not model everything?

Actual geometry?

Dot on dice:

- triangles on top of each other?

- floating dot?

- divide up face?

Very small triangles are bad:

- many triangles per pixel (how to pick?)

- hard to know which is in front

- hard to author (design, store, maintain, ...)

# Texture Mapping Review

1. Need coordinates **per pixel**

2. UV mapping
   - specify at vertices and interpolate

3. other coordinates: computed (environment), solid

4. lookup color based on coordinates

5. Filter/Interpolate (each pixel is a region of the texture)
   - use mip maps for speed

# How do we actually draw?

(or, how does the hardware do it for us?)

Categories of methods (rough, one way to divide it) ...

    1. Per-primitive methods (used for interactive graphics)

    2. Image-space methods

    3. World-space methods

Most (all?) graphics hardware (and high-performance interactive) rendering is done with **per-primitive** methods.

# The Process of Drawing in 3D

1. Triangles (in 3D)

2. Transform (the triangles to 2D) - **viewing**

3. Convert triangles to pixels - **rasterization**

4. Color Each Pixel - **shading** (lighting, texturing, ...)

along the way...

- decide if the triangle is on the screen - **clipping**

- decide if another triangle blocks the pixel - **visibility**

# Transformation (Modeling, Viewing)

Convert the 3D Triangle to a 2D triangle

# Rasterization

What pixels does a triangle cover

- since we will "fill" these pixels

# Do we always draw all triangles?

# Clipping vs. Visibility

**Clipping** is skipping triangles that are off screen

- clip to the frustum

- includes near and far

**Visibility** is having near things block farther ones

- can't know until you look at all of the triangles

- immediate mode might draw in any order

**Culling** is skipping primitives based on fast decisions

# When do we get rid of things?

**Clipping** can happen after the triangle is transformed

**Visibility** depends on the algorithm (often per-pixel, after coloring)

**Culling** should happen early

- discard a whole group of triangles (triangles in another room)
- backface culling (after the normal is transformed)
- and many others

# Visibility: The Problem

Assume objects are **solid** and **not-transparent**

Closer objects occlude farther ones

This is not **clipping** - we assume things are in-view

# Visibility is important!

An important cue to making things look realistic!

# Visibility Algorithms

- Painters Algorithm - important in concept

- Z-Buffer Algorithm - used in practice

We often use ideas from the painter's algorithm

# How does this work in the real world?

# How does a painter do it?

Paint each object

Paint new objects **on top of** older ones

Order matters

# The Painters Algorithm

1. Collect all objects

2. Sort from back to front

3. Draw objects in order (back to front)

# Problems with the Painter's Algorithm

1. Need all objects to sort (not immediate mode)

2. What happens with Ties?

3. What happens with intersecting objects?

4. Inefficiency - resort when camera moves

5. Inefficiency - draw things that get covered

# Dealing with Ties: Cutting Objects

If two objects (triangles) intersect, cut them

# Avoiding Re-Sorting

Use fancy data structures!

Binary Space Partitioning Tree (BSP-Trees)

Very important in old (pre-graphics hardware) video games
Not very important now (we use hardware)

# The Z-Buffer Algorithm

Goals:

- order independent

- immediate mode (1 triangle at a time)

Idea:

- Store the depth per-pixel

# The Z-Buffer

An extra number per-pixel

color buffer (RGB)

z-Buffer (Z)

# Clearing the Z-Buffer

Start with all pixels at max distance

Anything that will be drawn will be in front of the background

Why the **far** distance of the camera is important

Historically: limited precision (16 bits), avoid making far too big

# Drawing

Draw pixel (x,y) with color (c) and depth (z)

Frame buffer FB, Z-Buffer (ZB)

Old - no Z-Buffer (write pixel):

    FB(x,y) = c

New - Z-buffer (read/test/write):

    pz = ZB(x,y)

    if z < pz:

        FB(x,y) = c

        ZB(x,y) = z

# Order independent

**Far object first**

**Close object first**

# What could go wrong?

- Requires memory

- Requires read/modify/write

  - memory reads can be slow

  - need to wait until prior writes are done

- Requires storing distances

  - old days: 16 bits (scaling issues)

- things we care about...

# What happens in a tie?

Decide if < or <=

Order matters

## Z-Fighting

Two objects are very similar distances

- might be a tie (drawing order matters)

- numerical noise might break the tie

# What if a triangle partially fills a pixel?

Z-Buffer test is yes/no

Doesn't consider partial filling

## Aliasing!

Anti-aliasing the edges of objects is hard

(warning - we haven't discussed aliasing yet)

# Overdraw Efficiency

- Throw away pixel after it is computed

- After we have computed the color!

- Wasted effort!

- Overwrite is even more wasteful than Z-Fail

Future: ways to avoid the inefficiency

# Semi-Transparent Objects?

Alpha-Blending?

Back object needs to be drawn first:

- need to blend with correct background

- close object prevents objects behind from being drawn

Solution:

- sort objects (basically painters algorithm)

- transparent objects last

# Z-Buffer Summary

1. Clear Z-Buffer to Max Distance

2. Replace pixel write with read/test/write

## Good Parts

- Simple

- Uses memory (now cheap)

- Generally order independent

- Don't need objects ahead of time

- Easy to implement in hardware

- Invented by Ed Catmull

## Problems

1. Sometimes order matters

2. Can have efficiency issues

3. Z-Fighting Problems

4. Aliasing

5. Doesn't handle transparency

41

# The Drawing Process ("Pipeline")

Draw each triangle - somewhat order independent (parallel)

1. Transform triangles into 2D (with Z values)

2. Rasterize triangles (into fragments (pixels), with Z-values)

3. Figure out the color of those fragments

4. Write those fragments to the image (with Z-Test)

## What's next?

- program steps 1 and 3! (shaders)