CIS 441/541: Introduction to Computer Graphics
Lecture 3: Interpolation and the Z-Buffer

April 10th, 2013
Outline

- Project 1B revisited
- Interpolation along a triangle
- The Z-Buffer: How to resolve when triangles overlap on the screen
- Project 1C
- Parallel Rendering
Where we are…

- We haven’t talked about how to get triangles into position.
  - Arbitrary camera positions through linear algebra
- We haven’t talked about shading
- On Friday, we tackled this problem:
  How to deposit triangle colors onto an image?

Still don’t know how to:
1) Vary colors (easy)
2) Deal with triangles that overlap
Outline

- Project 1B revisited
- Interpolation along a triangle
- The Z-Buffer: How to resolve when triangles overlap on the screen
- Project 1C
- Parallel Rendering
- Cout/cerr can be misleading:

```cpp
fawcett:Downloads child$ cat t2.C
#include <iostream.h>
#include <iomanip>

int main()
{
    double X=188;
    X-=1e-12;
    cerr << X << endl;
    cerr << std::setprecision(16) << X << endl;
}
fawcett:Downloads child$ ./a.out
188
187.9999999999999
```
Project 1B

- The limited accuracy of cerr/cout can cause other functions to appear to be wrong:

```c
fawcett:Downloads child$ cat t3.C
#include <iostream.h>
#include <iomanip>
#include <math.h>

int main()
{
    double X=188;
    X-=1.e-12;
    cerr << "The floor of " << X << " is " << floor(X) << endl;
}

fawcett:Downloads child$ ./a.out
The floor of 188 is 187
```
Floating point precision is an approximation of the problem you are trying to solve.

Tiny errors are introduced in nearly every operation you perform.
- Exceptions for integers and denominators that are a power of two.

Fundamental problem:
- Changing the sequence of these operations leads to *different* errors.
- Example: \((A+B)+C \neq A+(B+C)\)
For project 1B, we are making a binary decision for each pixel: should it be colored or not?

Consider when a triangle vertex coincides with the bottom left of a pixel:

We all do different variations on how to solve for the endpoints of a line, so we all get slightly different errors.
Our algorithm incorporates floor and ceiling functions.

- This is the right place to bypass the precision problem.
- See “floor441” and “ceil441” in homework.
Outline

- Project 1B revisited
- Interpolating a field over a triangle
- The Z-Buffer: How to resolve when triangles overlap on the screen
- Project 1C
- Parallel Rendering
Every triangle has one color.

Any pixel covered by a triangle gets that triangle’s color.

We can think of the color as a constant field over the triangle.
What about triangles that have more than one color?
The color is in three channels, hence three scalar fields defined on the triangle.

- Red channel
- Green channel
- Blue channel
Consider a single scalar field defined on a triangle.
Consider a single scalar field defined on a triangle.

F(V1) = 10
F(V2) = 2
F(V3) = -2
What is $F(V4)$?

- $F(V1) = 10$
- $F(V2) = 2$
- $F(V3) = -2$

V4, at (0.5, 0.25)
Visualization of F

How do you think this picture was made?
We can interpolate colors over a triangle via fields $F_R$, $F_G$, and $F_B$. 

Red channel

Green channel

Blue channel
Scanline algorithm

- Determine rows of pixels triangles can possibly intersect
  - Call them rowMin to rowMax
    - rowMin: ceiling of smallest Y value
    - rowMax: floor of biggest Y value
- For r in [rowMin → rowMax] ; do
  - Find end points of r intersected with triangle
    - Call them leftEnd and rightEnd
  - For c in [ceiling(leftEnd) → floor(rightEnd) ] ; do
    - ImageColor(r, c) ← triangle color
Scanline algorithm w/ Color

- Determine rows of pixels triangles can possibly intersect
  - Call them rowMin to rowMax
    - rowMin: ceiling of smallest Y value
    - rowMax: floor of biggest Y value

- For r in [rowMin → rowMax] ; do
  - Find end points of r intersected with triangle
    - Call them leftEnd and rightEnd
  - Calculate Color(leftEnd) and Color(rightEnd) using interpolation from triangle vertices

- For c in [ceiling(leftEnd) → floor(rightEnd) ] ; do
  - Calculate Color(r, c) using Color(leftEnd) and Color(rightEnd)
  - ImageColor(r, c) ← Color(r, c)
Outline

- Project 1B revisited
- Interpolating a field over a triangle
- The Z-Buffer: How to resolve when triangles overlap on the screen
- Project 1C
- Parallel Rendering
Imagine you have a cube where each face has its own color.

<table>
<thead>
<tr>
<th>Face</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>Blue</td>
</tr>
<tr>
<td>Right</td>
<td>Green</td>
</tr>
<tr>
<td>Top</td>
<td>Red</td>
</tr>
<tr>
<td>Back</td>
<td>Yellow</td>
</tr>
<tr>
<td>Left</td>
<td>Purple</td>
</tr>
<tr>
<td>Bottom</td>
<td>Cyan</td>
</tr>
</tbody>
</table>

View from “front/top/right” side
Imagine you have a cube where each face has its own color.

How do we render the pixels that we want and ignore the pixels from faces that are obscured?

View from “front/top/right” side

View from “back/bottom/left” side
Consider a scene from the right side

Camera/eyeball

Camera oriented directly at Front face, seen from the Right side

<table>
<thead>
<tr>
<th>Face</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>Blue</td>
</tr>
<tr>
<td>Right</td>
<td>Green</td>
</tr>
<tr>
<td>Top</td>
<td>Red</td>
</tr>
<tr>
<td>Back</td>
<td>Yellow</td>
</tr>
<tr>
<td>Left</td>
<td>Purple</td>
</tr>
<tr>
<td>Bottom</td>
<td>Cyan</td>
</tr>
</tbody>
</table>
Consider the scene from the top side

Camera/eyeball

Camera oriented directly at Front face, seen from the Top side

<table>
<thead>
<tr>
<th>Face</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>Blue</td>
</tr>
<tr>
<td>Right</td>
<td>Green</td>
</tr>
<tr>
<td>Top</td>
<td>Red</td>
</tr>
<tr>
<td>Back</td>
<td>Yellow</td>
</tr>
<tr>
<td>Left</td>
<td>Purple</td>
</tr>
<tr>
<td>Bottom</td>
<td>Cyan</td>
</tr>
</tbody>
</table>
What do we render?

Green, Red, Purple, and Cyan all “flat” to camera. Only need to render Blue and Yellow faces (*).

Camera oriented directly at Front face, seen from the Top side

<table>
<thead>
<tr>
<th>Face</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>Blue</td>
</tr>
<tr>
<td>Right</td>
<td>Green</td>
</tr>
<tr>
<td>Top</td>
<td>Red</td>
</tr>
<tr>
<td>Back</td>
<td>Yellow</td>
</tr>
<tr>
<td>Left</td>
<td>Purple</td>
</tr>
<tr>
<td>Bottom</td>
<td>Cyan</td>
</tr>
</tbody>
</table>
What do we render?

What should the picture look like?
What’s visible? What’s obscured?

<table>
<thead>
<tr>
<th>Face</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>Blue</td>
</tr>
<tr>
<td>Right</td>
<td>Green</td>
</tr>
<tr>
<td>Top</td>
<td>Red</td>
</tr>
<tr>
<td>Back</td>
<td>Yellow</td>
</tr>
<tr>
<td>Left</td>
<td>Purple</td>
</tr>
<tr>
<td>Bottom</td>
<td>Cyan</td>
</tr>
</tbody>
</table>

Camera/eyeball

Camera oriented directly at Front face, seen from the Top side
New field associated with each triangle: depth

- Project 1B,1C:
  ```
  class Triangle
  {
    public:
      Double X[3];
      Double Y[3];
      ...
  };
  ```

- Now...
  ```
  Double Z[3];
  ```
What do we render?

Camera/eyeball

<table>
<thead>
<tr>
<th>Face</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>Blue</td>
</tr>
<tr>
<td>Right</td>
<td>Green</td>
</tr>
<tr>
<td>Top</td>
<td>Red</td>
</tr>
<tr>
<td>Back</td>
<td>Yellow</td>
</tr>
<tr>
<td>Left</td>
<td>Purple</td>
</tr>
<tr>
<td>Bottom</td>
<td>Cyan</td>
</tr>
</tbody>
</table>

Camera oriented directly at Front face, seen from the Top side.
Using depth when rendering

- Use Z values to guide which geometry is displayed and which is obscured.

- Example....
Consider 4 triangles with constant Z values:

- $Z = -0.35$
- $Z = -0.5$
- $Z = -0.65$
- $Z = -0.8$
Consider 4 triangles with constant Z values

How can we accomplish this picture?
Idea #1

- Sort triangles “back to front” (based on Z)
- Render triangles in back to front order
  - Overwrite existing pixels
Idea #2

- Sort triangles “front to back” (based on Z)
- Render triangles in front to back order
  - Do not overwrite existing pixels.
But there is a problem...
The Z-Buffer Algorithm

- The preceding 10 slides were designed to get you comfortable with the notion of depth/Z.

- The Z-Buffer algorithm is the way to deal with overlapping triangles when doing rasterization.
  - It is the technique that GPUs use.

- It works with opaque triangles, but not transparent geometry, which requires special handling
  - Transparent geometry discussed week 7.
  - Uses the front-to-back or back-to-front sortings just discussed.
The Z-Buffer Algorithm: Data Structure

- Existing: for every pixel, we store 3 bytes:
  - Red channel, green channel, blue channel

- New: for every pixel, we store a floating point value:
  - Depth buffer

- Now 7 bytes per pixel (*)
  - (*): 8 with RGBA
The Z-Buffer Algorithm: Initialization

- **Existing:**
  - For each pixel, for each R/G/B, set to 0.

- **New:**
  - For each pixel, for each depth value, set to -1.

  - Valid depth values go from -1 (back) to 0 (front)
  - This is partly convention and partly because it “makes the math easy” when doing transformations.
Scanline algorithm

- Determine rows of pixels triangles can possibly intersect
  - Call them rowMin to rowMax
    - rowMin: ceiling of smallest Y value
    - rowMax: floor of biggest Y value
- For r in [rowMin → rowMax] ; do
  - Find end points of r intersected with triangle
    - Call them leftEnd and rightEnd
  - For c in [ceiling(leftEnd) → floor(rightEnd)] ; do
    - ImageColor(r, c) ← triangle color
Scanline algorithm w/ Z-Buffer

- Determine rows of pixels triangles can possibly intersect
  - Call them rowMin to rowMax
    - rowMin: ceiling of smallest Y value
    - rowMax: floor of biggest Y value

- For \( r \) in \([\text{rowMin} \rightarrow \text{rowMax}]\); do
  - Find end points of \( r \) intersected with triangle
    - Call them leftEnd and rightEnd
  - Interpolate \( z(\text{leftEnd}) \) and \( z(\text{rightEnd}) \) from triangle vertices

- For \( c \) in \([\text{ceiling}(\text{leftEnd}) \rightarrow \text{floor}(\text{rightEnd})]\); do
  - Interpolate \( z(r,c) \) from \( z(\text{leftEnd}) \) and \( z(\text{rightEnd}) \)
  - If \( (z(r,c) > \text{depthBuffer}(r,c)) \)
    - ImageColor\((r, c) \leftarrow \text{triangle color}
    - \text{depthBuffer}(r,c) = z(r,c) \)
The Z-Buffer Algorithm: Example
The Z-Buffer Algorithm: Example
Interpolation and Triangles

- We introduced the notion of interpolating a field on a triangle
- We used the interpolation in two settings:
  - 1) to interpolate colors
  - 2) to interpolate depths for z-buffer algorithm
- Project 1D will be discussed on Friday, but…
  - You will be adding color interpolation and the z-buffer algorithm to your programs.
Outline

- Project 1B revisited
- Interpolation along a triangle
- The Z-Buffer: How to resolve when triangles overlap on the screen
- Project 1C
- Parallel Rendering
Arbitrary Triangles

- The description of the scanline algorithm from Lecture 2 is general.
- But the implementation for these three triangles vary:
Arbitrary Triangles

- Project #1B: implement the scanline algorithm for triangles with “flat bottoms”
- Project #1C: arbitrary triangles
Goal: apply the scanline algorithm to arbitrary triangles and output an image.

Extend your project1B code

File proj1c_geometry.vtk available on web (80MB)

File “reader.cxx” has code to read triangles from file.
What do I do if I run into trouble?

1) Start with the class forum
2) OH this week
   - Weds: 2-3:30
   - Fri: 10-11:30, 2-3:30
3) Email me: hank@cs.uoregon.edu
Outline

- Project 1B revisited
- Interpolation along a triangle
- The Z-Buffer: How to resolve when triangles overlap on the screen
- Project 1C
- Parallel Rendering
Parallel Rendering

- Three forms:
  - Sort first
  - Sort middle
  - Sort last

- We know enough to discuss “sort last”
Sort last rendering

Assumptions:
- So many triangles that you need multiple computers to render them
- The triangles are partitioned over the computers

CPU 0’s image

CPU 1’s image

Final composited image (done w/ z-buffer)