Lecture 22: General Position, Robustness & Exact Computation
Outline for Today

- Homework 6 Posted
- Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
- General Position, Floating Point Equality
- Numerical Computing, Divide by Zero, & Gaussian Elimination
- Floating Point Bugs in Computational Geometry
- Floating Point Bugs in Computer Graphics
- Real RAM vs. IEEE Floating Point
- Arbitrary Arithmetic with Rational and Algebraic Numbers
- Symbolic Computation, Floating Point Filters, Interval Computation
- Next Time: ?
Outline for Today

- Homework 6 Posted
- Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
- General Position, Floating Point Equality
- Numerical Computing, Divide by Zero, & Gaussian Elimination
- Floating Point Bugs in Computational Geometry
- Floating Point Bugs in Computer Graphics
- Real RAM vs. IEEE Floating Point
- Arbitrary Arithmetic with Rational and Algebraic Numbers
- Symbolic Computation, Floating Point Filters, Interval Computation
- Next Time: ?
Necker Cube

- A two dimensional representation of a three dimensional wire frame cube
- Viewer’s perception can flips back and forth between equally possible perspectives

https://www.newworldencyclopedia.org/entry/necker_cube
https://commons.wikimedia.org/wiki/File:Necker%27s_cube.svg
Hidden Line Drawing / Depth Buffer (z-Buffer)

- Given a primitive's vertices & the color / illumination at each vertex:
- Figure out which pixels to "turn on" to render the primitive
- Interpolate the color / illumination values to "fill in" the primitive
- At each pixel, keep track of the closest primitive (depth buffer / z-buffer)

```
glBegin(GL_TRIANGLES)
glNormal3f(...)  
glVertex3f(...)   
glVertex3f(...)   
glVertex3f(...)   
glEnd();
```
Hidden Line Drawing: Painter’s Algorithm

- Let’s order the primitives by how close they are to the camera
- Draw the primitives from back to front
- Then we don’t need to keep track of the depth!

\textit{Save memory!}
Definition: Binary Space Partition

- Place items in a binary tree, each node stores a half plane.
- Primitives that are collinear with the half plane are stored in the node.
- Items overlapping a half plane are copied/split into two primitives.
- We recurse until exactly one item is left, it is stored in the leaf.

*Computational Geometry Algorithms and Applications*, de Berg, Cheong, van Kreveld and Overmars, Chapter 12
Discussion - Quad Tree, kD Tree, BSP

- k-D trees are a special case of BSP (where splits are always axis aligned)
- Quad trees are a special case of k-D trees (where splits are always at the midpoints)
Outline for Today

- Homework 6 Posted
- Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
- **General Position, Floating Point Equality**
- Numerical Computing, Divide by Zero, & Gaussian Elimination
- Floating Point Bugs in Computational Geometry
- Floating Point Bugs in Computer Graphics
- Real RAM vs. IEEE Floating Point
- Arbitrary Arithmetic with Rational and Algebraic Numbers
- Symbolic Computation, Floating Point Filters, Interval Computation
- Next Time: ?
“Assuming General Position… ”

- Degeneracies in the input data cause problems
- To avoid problems in developing algorithms and to prove the correctness and performance of those algorithms, we will often make assumptions, e.g.:
  - No 2 points have the same \( x \) and same \( y \) coordinates
  - No 3 points are collinear
  - No 2 points lie on the same vertical line
  - No 4 points lie on the same circle
Robustness: Floating Point Equality

- Programming Mantra: Never compare two floats or doubles with ==
- Why not?

```c
double a = 2/float(3);
double b = (5/3.0) * (7.0/5.0f) * (2/double(7));
assert (a == b);
```
Robustness: Floating Point Equality

- Programming Mantra: Never compare two floats or doubles with ==
- Why not?
  ```
  double a = 2/float(3);
  double b = (5/3.0) * (7.0/5.0f) * (2/double(7));
  assert (a == b); ← this will probably fail!
  ```
- Even if we’re more careful and use float & double consistently, the compiler is still free to use extra precision for the values of intermediate expressions.
- Optimized code might use registers for intermediate expressions (which often have higher precision than required by the type).
Common Robustness Workaround

- Instead compare to a tolerance or epsilon value, e.g.,
  
  ```c
  double a = 2/float(3);
  double b = (5/3.0) * (7.0/5.0f) * (2/double(7));
  assert (fabs(a-b) < 0.00001);
  ```

- But what is the right value for epsilon?
Common Robustness Workaround

- Instead compare to a tolerance or epsilon value, e.g.,
  ```
  double a = 2/float(3);
  double b = (5/3.0) * (7.0/5.0f) * (2/double(7));
  assert (fabs(a-b) < 0.00001);
  ```

- But what is the right value for epsilon?
  It depends on the application, data type, & overall scale of the data!

  - epsilon way too big → we risk computing the wrong answer
  - epsilon too small → the original equality rounding error issue
  - What if roundoff error will accumulate or compound over time?
    It will likely be impossible to appropriately set an epsilon!
Outline for Today

● Homework 6 Posted
● Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
● General Position, Floating Point Equality
● Numerical Computing, Divide by Zero, & Gaussian Elimination
● Floating Point Bugs in Computational Geometry
● Floating Point Bugs in Computer Graphics
● Real RAM vs. IEEE Floating Point
● Arbitrary Arithmetic with Rational and Algebraic Numbers
● Symbolic Computation, Floating Point Filters, Interval Computation
● Next Time: ?
Numerical Computing & Divide by Zero

- For numerical computing, divide by zero is the most common (is the only?) precision / rounding error that may cause a program to crash.

- Otherwise, the program will always return a result:
  - The result will be a good answer.
  - It may be slightly off due to rounding error (the error is proportional to the types – e.g., float/double), but it is generally acceptable.
Factorization by Gaussian Elimination

- A pivot or row swap is necessary if the value in the target position is zero and would lead to a divide-by-zero when we try to compute the row multiplier necessary to produce zeros in that column in the lower rows.
Factorization by Gaussian Elimination

- Divide by zero is not the only concern…
- We should also avoid division by very small values, e.g., epsilon:

\[
A = \begin{bmatrix}
-\epsilon & 1 \\
1 & -1
\end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 - \epsilon \\ 0 \end{bmatrix},
\]

\[
\begin{bmatrix}
-\epsilon & 1 & 1 - \epsilon \\
0 & -1 + \epsilon^{-1} & \epsilon^{-1} - 1
\end{bmatrix}
\]

\[
x_2 = 1 \quad \Rightarrow \quad x_1 = \frac{(1 - \epsilon) - 1}{-\epsilon}
\]

Correct answer: \( x_1 = 1 \)

But we will have robustness problems if \( \epsilon \) is very small!

Fundamentals of Numerical Computation, Driscoll & Braun, 2017
https://fncbook.github.io/v1.0/linsys/pivoting.html
Factorization by Gaussian Elimination

- Divide by zero is not the only concern...
- We should also avoid division by very small values, e.g., epsilon:

\[
\begin{align*}
\text{Correct answer: } x_1 &= 1 \\
\text{But we will have robustness problems if } \epsilon \text{ is very small!}
\end{align*}
\]

It's better to pivot / swap rows for the row with the largest value in this column

Fundamentals of Numerical Computation, Driscoll & Braun, 2017
https://fncbook.github.io/v1.0/linsys/pivoting.html
Numerical vs. Combinatorial

- Use of a tolerance or epsilon is an appropriate approach for *numerical computing* (e.g. solving linear systems), where answers being slightly off is acceptable.

- However in geometry, the goal is not to compute numbers but rather structures (convex hull, Delaunay triangulation, etc).

- It is a *combinatorial problem*, not a numerical problem.

https://www.cgal.org/exact.html
Outline for Today

● Homework 6 Posted
● Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
● General Position, Floating Point Equality
● Numerical Computing, Divide by Zero, & Gaussian Elimination
● Floating Point Bugs in Computational Geometry
● Floating Point Bugs in Computer Graphics
● Real RAM vs. IEEE Floating Point
● Arbitrary Arithmetic with Rational and Algebraic Numbers
● Symbolic Computation, Floating Point Filters, Interval Computation
● Next Time: ?
Ramshaw’s Braided Lines

- Consider 2 lines,
  \[ l_1 : y = \frac{9833x}{9454} \]
  \[ l_2 : y = \frac{9366x}{9005} \]
  both pass through the origin,
slope of \( l_1 \) is slightly larger than \( l_2 \)
- This program computes and compares the y-value for each line at multiples of 0.001 between 0 and 1
- The program outputs that \( l_1 \) and \( l_2 \) intersect 24 times !?!?!
- If we switch float → double, it still prints 1 false intersection (not the origin)

http://www.algorithmic-solutions.info/leda_guide/geometry/dangerfloat.html

```cpp
#include <iostream.h>

int main()
{
    cout.precision(12);
    float delta=0.001f;
    int last_comp=-1;
    float a=9833,b=9454,c=9366,d=9005;

    float x;
    for (x=0;x<0.1;x=x+delta) {
        float y1=a*x/b;
        float y2=c*x/d;

        int comp;
        if (y1<y2) comp=-1;
        else if (y1==y2) comp=0;
        else comp=1;

        if (comp!=last_comp) {
            cout << endl << x << " : ";
            if (comp==-1) cout << "l1 is below l2";
            if (comp==0) cout << "l1 intersects l2";
            else cout << "l1 is above l2";
        }
    }
    last_comp=comp;
}

    cout << endl << endl;
    return 0;
}```
Using floating point arithmetic:
- Take two random lines $l_1$ and $l_2$
- Compute intersection point $p_{12}$
- assert (point $p_{12}$ lies on line $l_1$)
- assert (point $p_{12}$ lies on line $l_2$)

Orange dots = 1 assertion fails
Red dots = both assertion fails

Invited Lecture: “Real Numbers and Robustness in Computational Geometry”, Real Numbers and Computers 2004, Stefan Schirra
Incremental Convex Hull Construction

- Make a triangle with the first 3 points

- For each additional point $r$
  - Find an outside edge that is “visible” from $r$
  - Expand to a sequence of connected edges
    \[ v_i \rightarrow v_{i+1} \rightarrow v_{i+2} \rightarrow \ldots \rightarrow v_j \]
  - Remove middle points (e.g., $v_{i+1}$ & $v_{i+2}$) from hull, add point $r$ to hull
Incremental Convex Hull Construction

- Make a triangle with the first 3 points
- For each additional point $r$
  - Find an outside edge that is “visible” from $r$
  - Expand to a sequence of connected edges $v_i \rightarrow v_{i+1} \rightarrow v_{i+2} (\rightarrow \ldots ) \rightarrow v_j$
  - Remove middle points (e.g., $v_{i+1} \& v_{i+2}$) from hull, add point $r$ to hull


Algorithm looks great!
So how could this be a program output????
- the hull of \( p_1 \) to \( p_4 \) is correctly computed
- \( p_5 \) lies close to \( p_1 \), lies inside the hull of the first four points, but float-sees the edge \((p_1, p_4)\).
  Concave corner at \( p_5 \).
- point \( p_6 \) sees the edges \((p_1, p_2)\) and \((p_4, p_5)\), but does not see the edge \((p_5, p_1)\).
- we obtain ...
This concave angle is a small, “acceptable” numerical error

- the hull of $p_1$ to $p_4$ is correctly computed
- $p_5$ lies close to $p_1$, lies inside the hull of the first four points, but float-sees the edge $(p_1, p_4)$. Concave corner at $p_5$.
- point $p_6$ sees the edges $(p_1, p_2)$ and $(p_4, p_5)$, but does not see the edge $(p_5, p_1)$.
- we obtain ...

This concave angle is a small, "acceptable" numerical error

But it causes a large, unacceptable logical error later!

- the hull of $p_1$ to $p_4$ is correctly computed
- $p_5$ lies close to $p_1$, lies inside the hull of the first four points, but float-sees the edge $(p_1, p_4)$. Concave corner at $p_5$.
- point $p_6$ sees the edges $(p_1, p_2)$ and $(p_4, p_5)$, but does not see the edge $(p_5, p_1)$.
- we obtain ...

Outline for Today

● Homework 6 Posted
● Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
● General Position, Floating Point Equality
● Numerical Computing, Divide by Zero, & Gaussian Elimination
● Floating Point Bugs in Computational Geometry
● **Floating Point Bugs in Computer Graphics**
● Real RAM vs. IEEE Floating Point
● Arbitrary Arithmetic with Rational and Algebraic Numbers
● Symbolic Computation, Floating Point Filters, Interval Computation
● Next Time: ?
Hue indicates elevation

Possible Hidden / Probably Hidden: If height is changed by epsilon, the visibility flips!

The visibility of one half of the points is uncertain!
Rounding Errors in Graphics Ray Tracing

- To correctly simulate light as it bends/refracts through a medium denser than air (e.g., glass), we must know when a ray enters and when a ray exits an object.

  - $r_a$ intersects $f_1$
  - $r_b$ must intersect $f_1$ or $f_2$ but NOT both or neither!
  - *We cannot miss or double count intersections!*

Jietong Chen

https://cjt-jackton.github.io/RayTracing/
Epsilon in Ray Tracing

Image from Zachary Lynn
Epsilon a.k.a. Bias for Shadow Maps

Correct image  Not enough bias  Way too much bias
Outline for Today

● Homework 6 Posted
● Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
● General Position, Floating Point Equality
● Numerical Computing, Divide by Zero, & Gaussian Elimination
● Floating Point Bugs in Computational Geometry
● Floating Point Bugs in Computer Graphics
● **Real RAM vs. IEEE Floating Point**
● Arbitrary Arithmetic with Rational and Algebraic Numbers
● Symbolic Computation, Floating Point Filters, Interval Computation
● Next Time: ?
Real RAM

- “A real RAM (random-access machine) is a mathematical model of a computer that can compute with exact real numbers instead of the binary fixed point or floating point numbers used by most actual computers.”

- Computers can only approximate a real RAM using floating point types.

- CGAL (The Computational Geometry Algorithms Library) and LEDA (A Library of Efficient Data Types and Algorithms) provide tools that allow us to write programs that work like they were running on a real RAM.

https://en.wikipedia.org/wiki/Real_RAM
The IEEE Floating Point Standard

- **IEEE binary32** = C/C++ float
  
  ![Binary32 Diagram]

  $0.15625 = 0.00011111100_2$ (bit index)

- **IEEE binary64** = C/C++ double
  
  ![Binary64 Diagram]

- **IEEE binary128** = [ not (yet?) support by most hardware ]

Avoid Creating Irrational Numbers

- Problem: Given 5 points with integer coordinates, find the nearest neighbor to point $a$
- Compute the length of lines $ab$, $ac$, $ad$, $ae$
  - $\text{length}(ab) = \sqrt{(x_a-x_b)(x_a-x_b) + (y_a-y_b)(y_a-y_b)}$
  - Note: the $\sqrt{}$, will likely create irrational numbers!
- Sort the lengths, return endpoint for shortest line length

- Instead… compute & sort the squares of the line lengths
  - $\text{squared\_length}(ab) = (x_a-x_b)(x_a-x_b) + (y_a-y_b)(y_a-y_b)$
  - This is an integer!
- This will always return the correct answer to the original question.

*WITHOUT creating irrational numbers!*
Outline for Today

● Homework 6 Posted
● Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
● General Position, Floating Point Equality
● Numerical Computing, Divide by Zero, & Gaussian Elimination
● Floating Point Bugs in Computational Geometry
● Floating Point Bugs in Computer Graphics
● Real RAM vs. IEEE Floating Point
● **Arbitrary Arithmetic with Rational and Algebraic Numbers**
● Symbolic Computation, Floating Point Filters, Interval Computation
● Next Time: ?
Arbitrary Precision Arithmetic

- If we do not have irrational numbers in our program...
- We can store integers using a “BigNum” infinite precision integer type

- 64 bit binary integer = ~19 bit base 10 integer
- RSA Security requires at least 100 binary digits, but recommends 1000+ binary digits

https://patshaughnessy.net/2014/1/9/how-big-is-a-bignum
Arbitrary Precision Arithmetic

- If we do not have irrational numbers in our program...
- We can store rational numbers as a ratio of two BigNums
- Reduce fractions whenever possible to minimize storage:

\[
\frac{49578291287491495151508905425869578}{74367436931237242727263358138804367} = \frac{2}{3}
\]

https://algorist.com/problems/Arbitrary-Precision_Arithmetic.html
What if we *cannot* avoid Irrational Numbers?

- Hippasos used a geometric analog of Euclid's algorithm to show that the ratio $d_0/s_0$ is an irrational number.

Algebraic Number

- A number that is a root of a non-zero polynomial in one variable with integer (or, equivalently, rational) coefficients.
- \( \sqrt{2} \) is an algebraic number
- The golden ratio, \( \varphi = (1 + \sqrt{5}) / 2 \approx 1.61803 \) is an algebraic number, it is a root of \( x^2 - x - 1 \)
- All rational numbers are algebraic
- Some irrational numbers (e.g., \( \sqrt{2} \) & \( \varphi \)) are algebraic
- Some irrational numbers are NOT algebraic
  - \( \pi \approx 3.14159 \) is not algebraic
  - \( e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \ldots \approx 2.71828 \) is not algebraic

https://en.wikipedia.org/wiki/Algebraic_number
Outline for Today

- Homework 6 Posted
- Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
- General Position, Floating Point Equality
- Numerical Computing, Divide by Zero, & Gaussian Elimination
- Floating Point Bugs in Computational Geometry
- Floating Point Bugs in Computer Graphics
- Real RAM vs. IEEE Floating Point
- Arbitrary Arithmetic with Rational and Algebraic Numbers
- Symbolic Computation, Floating Point Filters, Interval Computation
- Next Time: ?
So, What’s done in Practice?

- Input point coordinates are rational
- If we can limit to linear primitives – straight lines, not curves…
  then the computations for most geometric problems will be rational, or at least algebraic.
- We can write software to implement & use basic arithmetic operations with all of the necessary types:
  integers, big-nums, rational numbers, and even algebraic numbers
  - And this is exactly what CGAL is doing with all of those C++ templates & typedefs!
- Much of this can also be made to work with nonlinear primitives too!
  - Avoid creating irrational numbers by working symbolically until output
Improving Performance

- The challenge is efficiency.
  - CGAL: overhead for exact computation = 25% - 80% (depending on algorithm)
  - See also https://www.cgal.org/exact.html
- User is responsible for understanding exact vs. inexact computation
  - Writing good CGAL code (non-buggy, robust, accurate, and fast) takes skill
  - Leverage both non-exact and exact kernels in different places in same program
- Implementation of CGAL (& other libraries) is clever…
  - Don’t use exact computation unless necessary
  - Work with floating point approximations most of the time
  - “Floating point filters”: Automatically switch from a floating point representation to exact computation when the numbers are close to a floating point tolerance.
  - Use symbolic / lazy adaptive evaluation to delay exact computation until and only if it is actually necessary
Alternative for Real Number Computation?

- Take imprecision into account when designing and proving the algorithms
- “Topology oriented implementation”
  - Program will always return an answer, even if all computations are replaced by random numbers
  - Never crashes because of inconsistencies
- Has been done for some problems & algorithms in Computational Geometry

- However, because most proofs rely on “assume general position” or small tricks like “rotate everything a tiny amount” to break ties

Most work in Computational Geometry would need to be redone!
Outline for Today

- Homework 6 Posted
- Last Time: Hidden Line Drawing, Painter’s Algorithm, & BSP
- General Position, Floating Point Equality
- Numerical Computing, Divide by Zero, & Gaussian Elimination
- Floating Point Bugs in Computational Geometry
- Floating Point Bugs in Computer Graphics
- Real RAM vs. IEEE Floating Point
- Arbitrary Arithmetic with Rational and Algebraic Numbers
- Symbolic Computation, Floating Point Filters, Interval Computation
- Next Time: ?