GDC

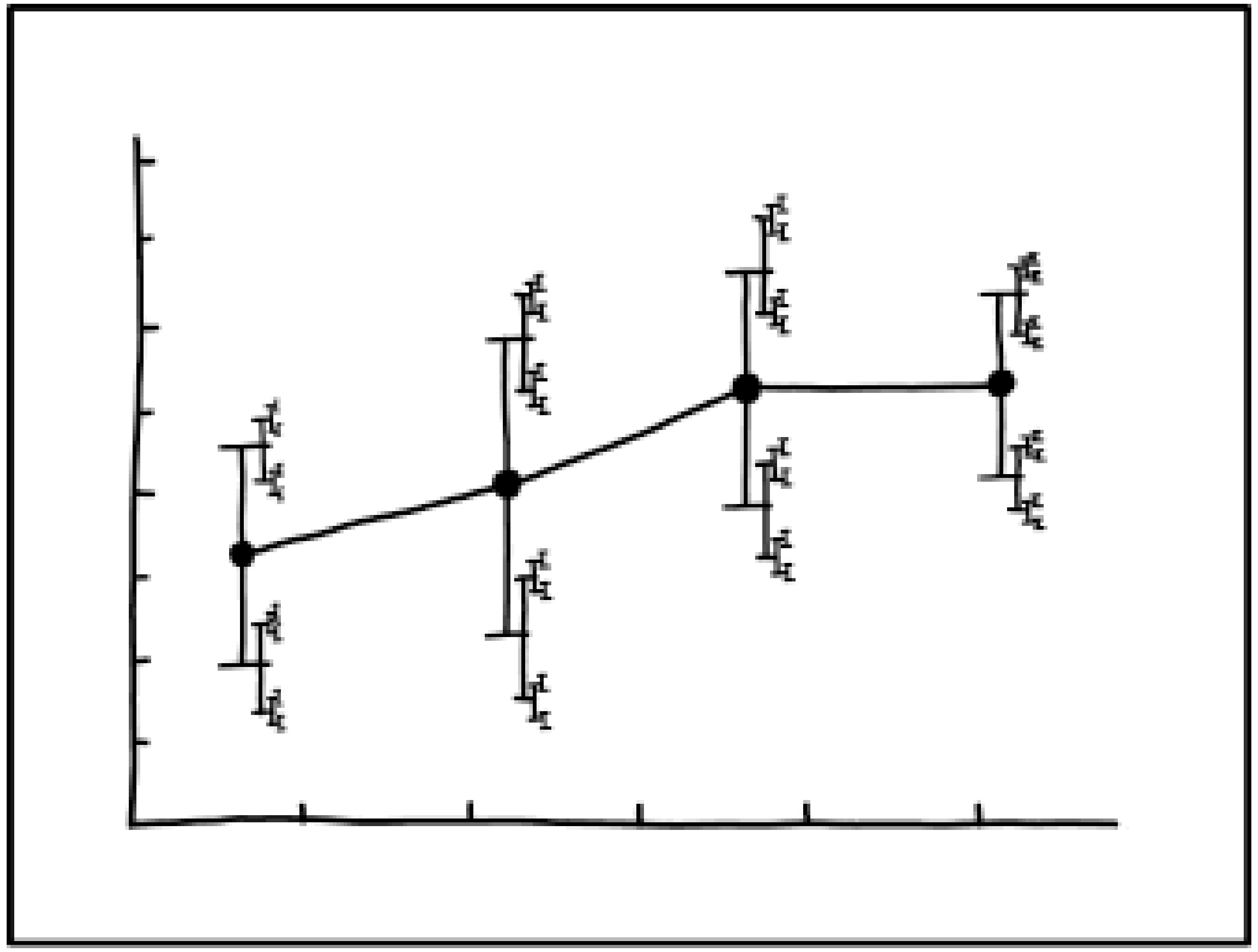
Math for Game Developers:

Understanding and Tracing Numerical Errors in C++

Gino van den Bergen
Dtecta

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

Know Thy Error

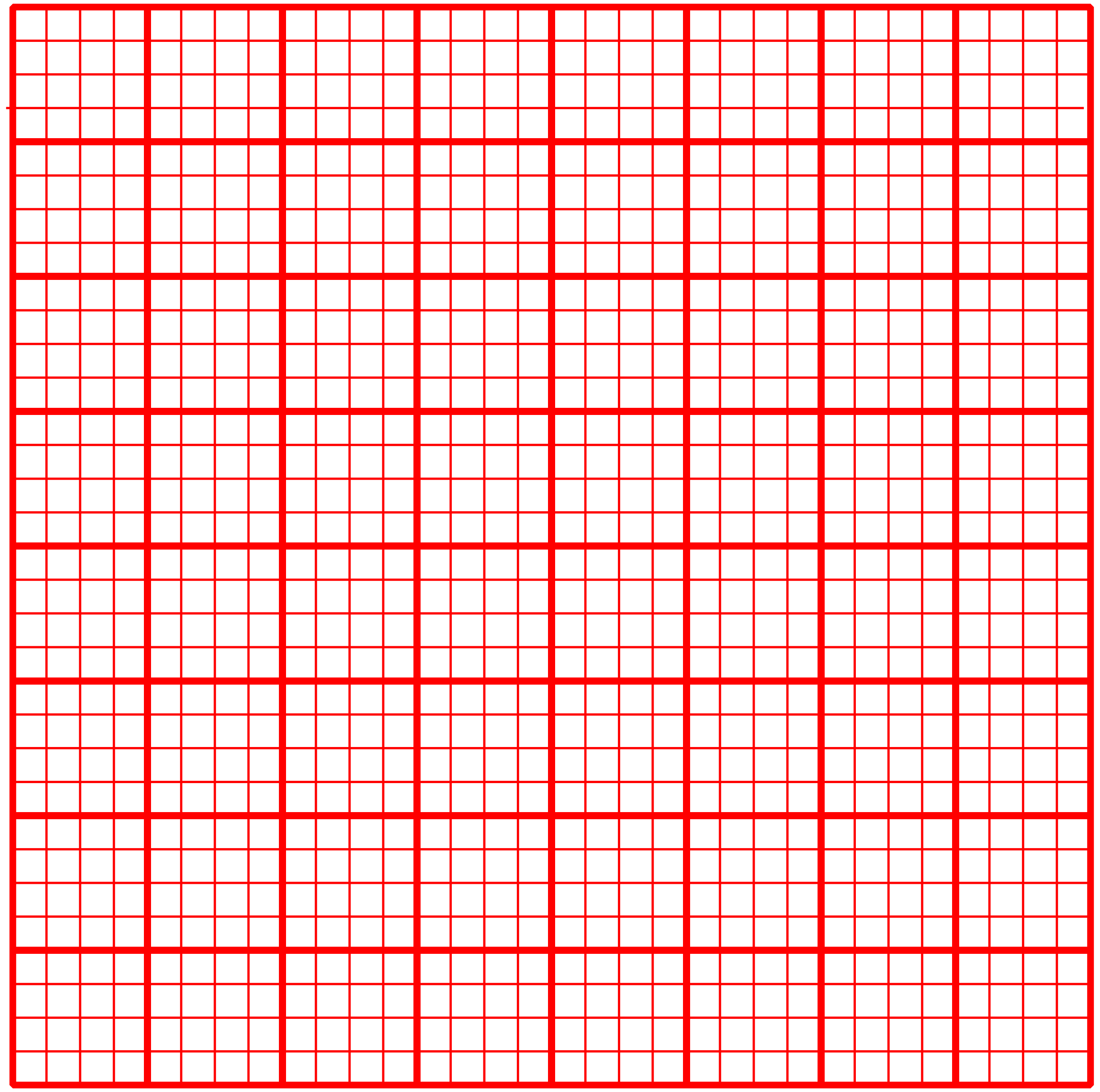


I DON'T KNOW HOW TO PROPAGATE
ERROR CORRECTLY, SO I JUST PUT
ERROR BARS ON ALL MY ERROR BARS.

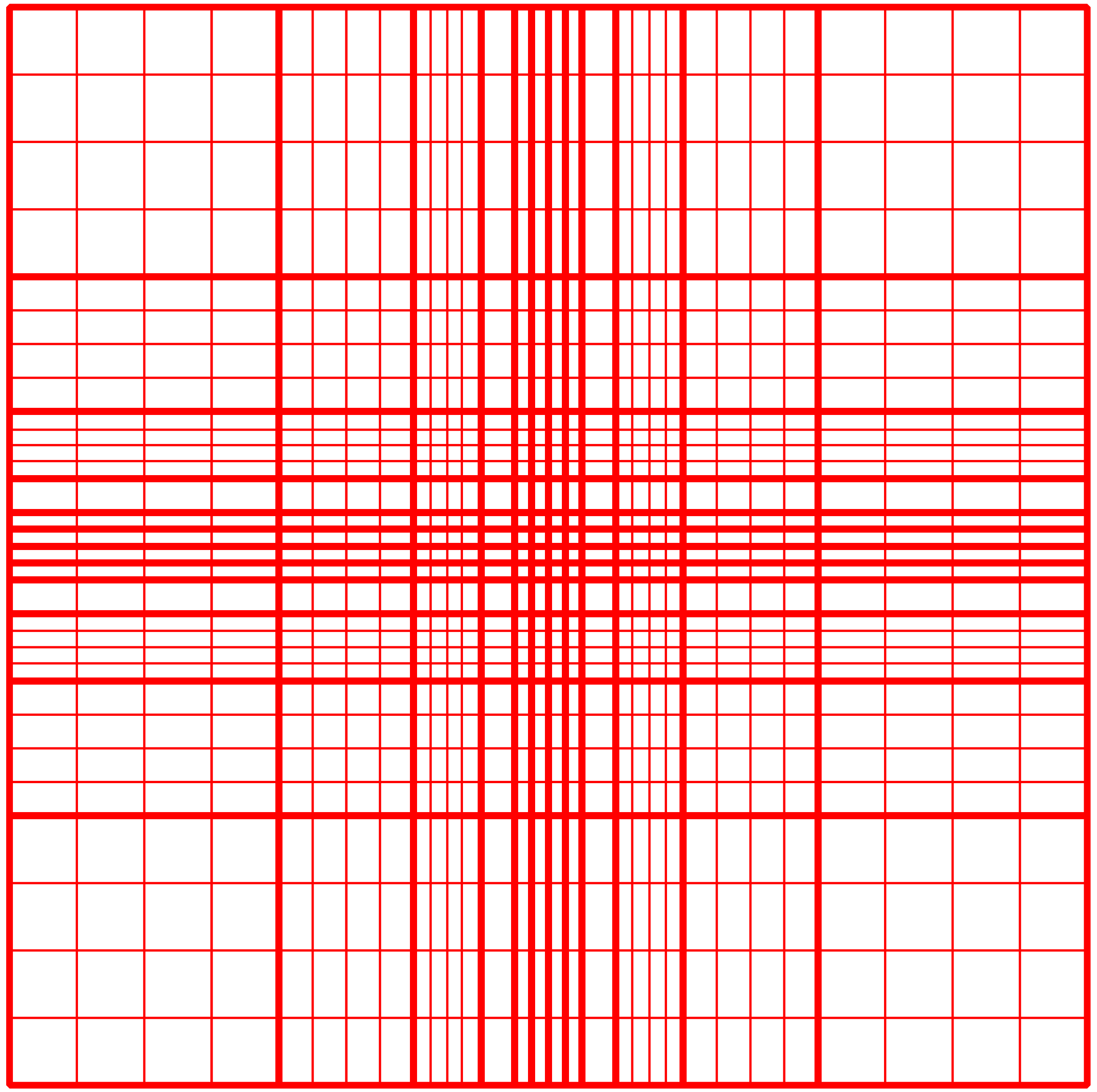
Computer Numbers

- Digital computer number formats have limited precision.
- Results of arithmetic operations are rounded to the nearest representable value.

Fixed-point Numbers



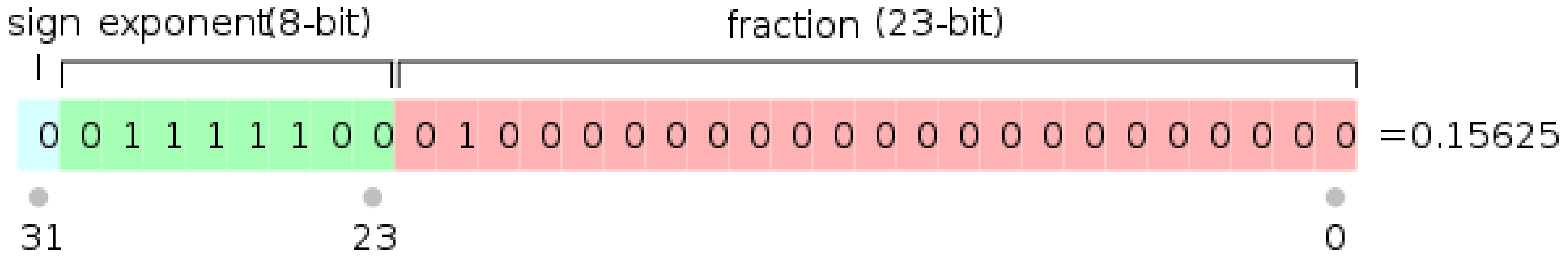
Floating-point Numbers



Floating-point Format

- IEEE 754 single-precision (32-bit) format:

$$(-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exponent}-127}$$



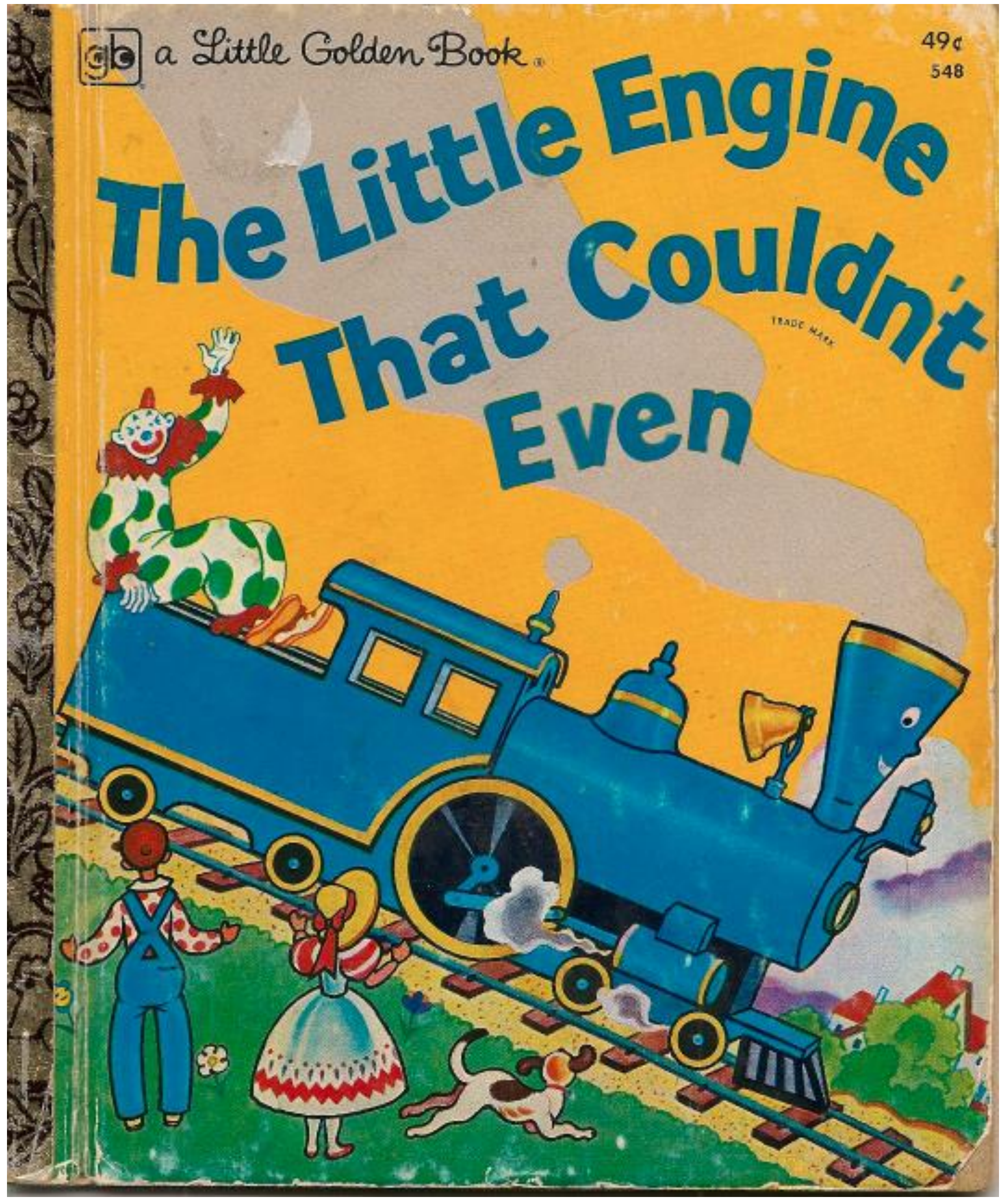
Floating-point Format (cont'd)

- Zero is a special case: *exponent* and *fraction* are zero. Both +0 and –0 exist.
- Subnormal numbers: *exponent* is zero.

$$(-1)^{\text{sign}} \times 0.\text{fraction} \times 2^{-126}$$

Fills the gap between 0 and 2^{-126}.

A Little Story…



A Little Story… (cont’d)

- World coordinates are single-precision floats.
- The top of the mountain is far, far away (300km) from the world coordinate origin.
- The little blue engine moves by forward Euler:

$$\mathbf{p}_{n+1} = \mathbf{p}_n + \mathbf{v}h$$

A Little Story… (cont'd)

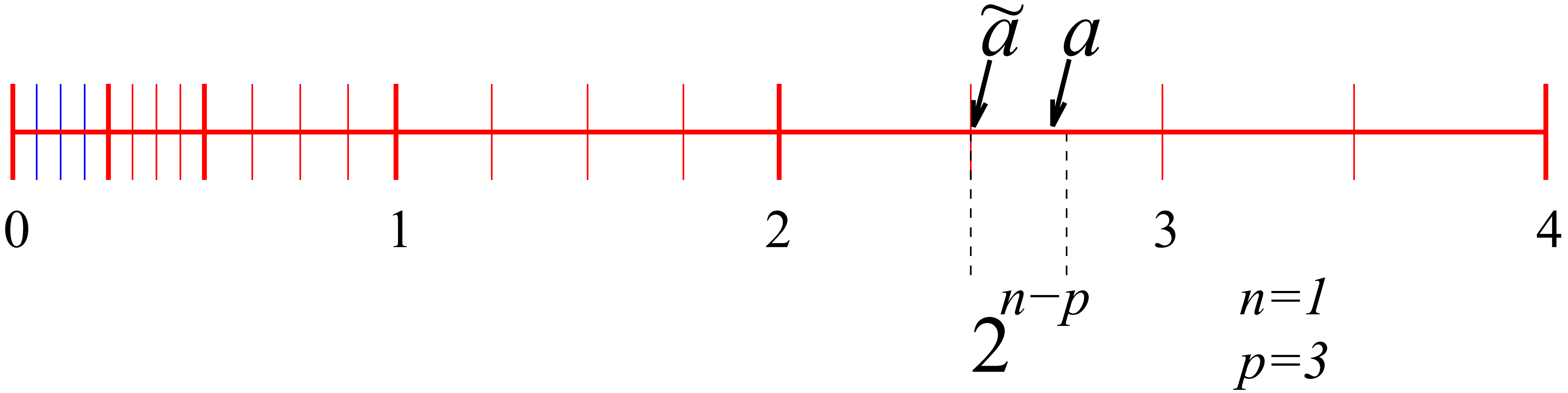
- The little engine tugged and pulled up the mountain and slowly, slowly, slowly, …
- … came to a grinding halt.
- What happened?
- At this distance from the origin $\mathbf{p}_n + \mathbf{v}h$ is rounded to $\mathbf{p}_n$ even though $\mathbf{v}h$ is not zero.

Big Worlds

- Prefer fixed-point for storing world coordinates.
- Fixed-point warrants same numerical behavior anywhere in your game world.
- Optionally, keep a float for storing the remainder after rounding to fixed-point unit.
- Also, prefer fixed-point for absolute time.

Relative Error

- For each real number $a \in [2^n, 2^{n+1}]$, there exists a floating-point number $\tilde{a} \in [2^n, 2^{n+1}]$, such that $|a - \tilde{a}| \le 2^{n-p}$, where p is the precision (bit-width of *fraction* plus one).



Relative Error (cont'd)

- There exists an r, such that $\tilde{a} = a(1 + r)$, and $|r| \leq 2^{-p}$.
- $\varepsilon = 2^{-p}$ is the *machine epsilon*, an upper bound on the *relative error*.
- For single-precison, $\varepsilon = 2^{-24}$, which is half `FLT_EPSILON` (the difference between 1 and the smallest float > 1).

Relative Error (cont’d)

- A single rounding operation results in a relative error that is no greater than ε.
- Errors accumulate with each operation.
- Notably subtracting two almost equal floating-point values introduces a large relative error.

Cancellation

- We have numbers $\tilde{a} = a(1 + r_a)$ and $\tilde{b} = b(1 + r_b)$ already contaminated by rounding.
- The difference $d = a - b$ is computed as $\tilde{d} = (\tilde{a} - \tilde{b})(1 + \varepsilon) = (a - b)(1 + r_d)$, where

$$|r_d| \le \frac{|a\, r_a| + |b\, r_b|}{|a - b|} + \varepsilon$$

Cancellation (cont'd)

- Suppose that a and b are almost equal. Then, $|r_d|$ can be huge.

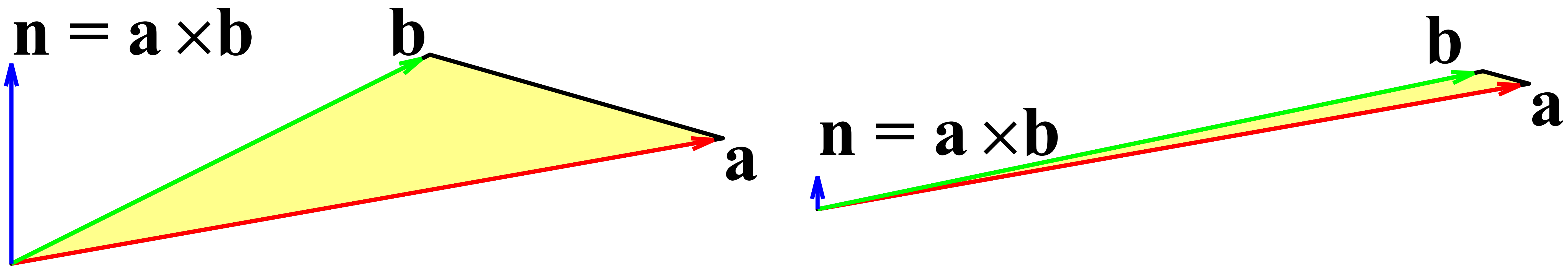
$$
\begin{array}{rl}
 & 1.11111111000101010110110110 \times 2^{-5} \\
- & 1.11111111000101010110011110 \times 2^{-5} \\
\hline
 & 1.1000\color{red}{0000000000000000000000} \times 2^{-25}
\end{array}
$$

Cancellation (cont'd)

- In this example, the 20 least-significant bits (red zeroes) in the fraction are garbage.
- This loss of significant bits is called **cancellation**, and is the main source of numerical issues.

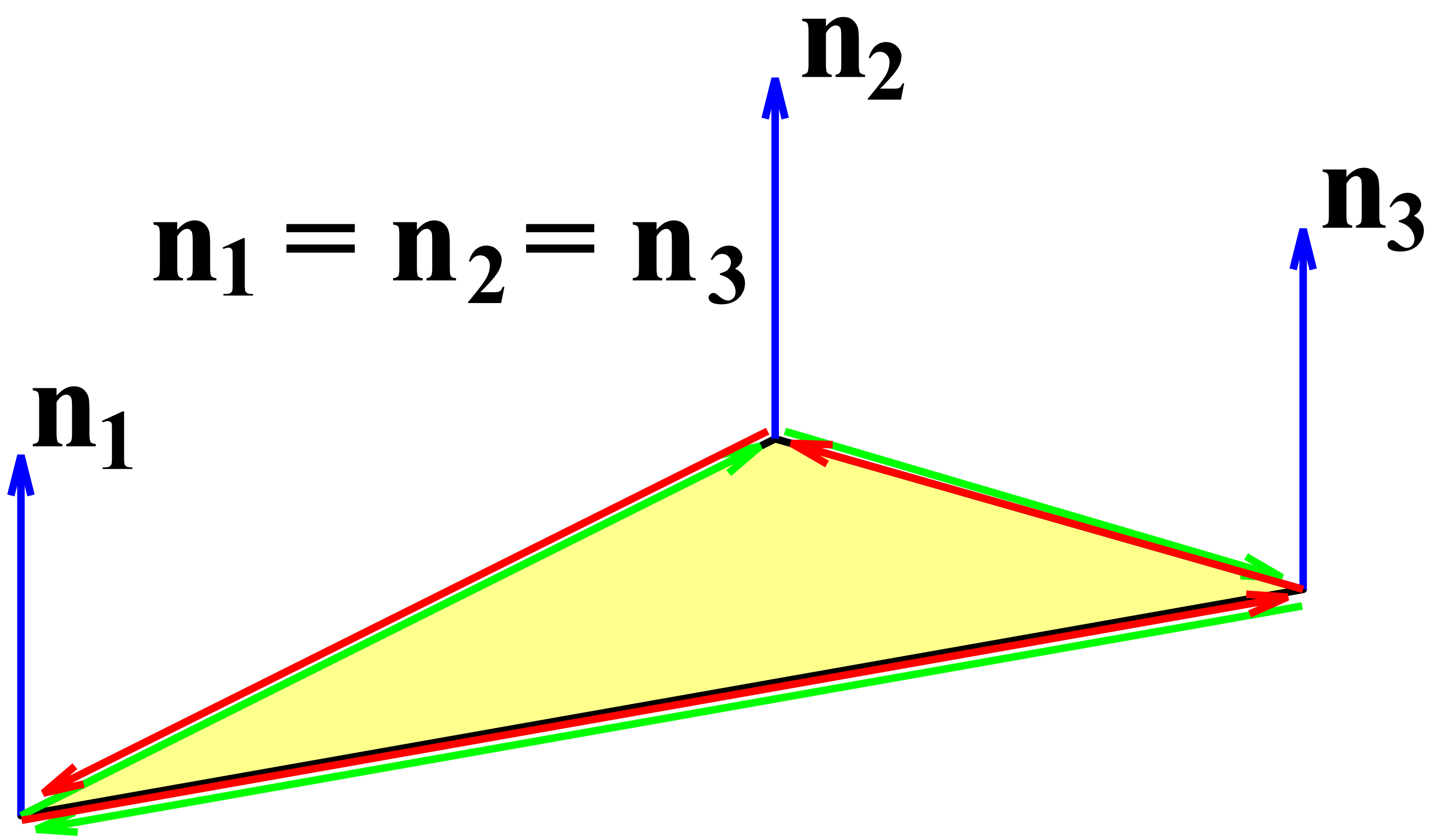
Example: Face Normals

- Compute normal of triangle by taking the cross product of two of its edges.



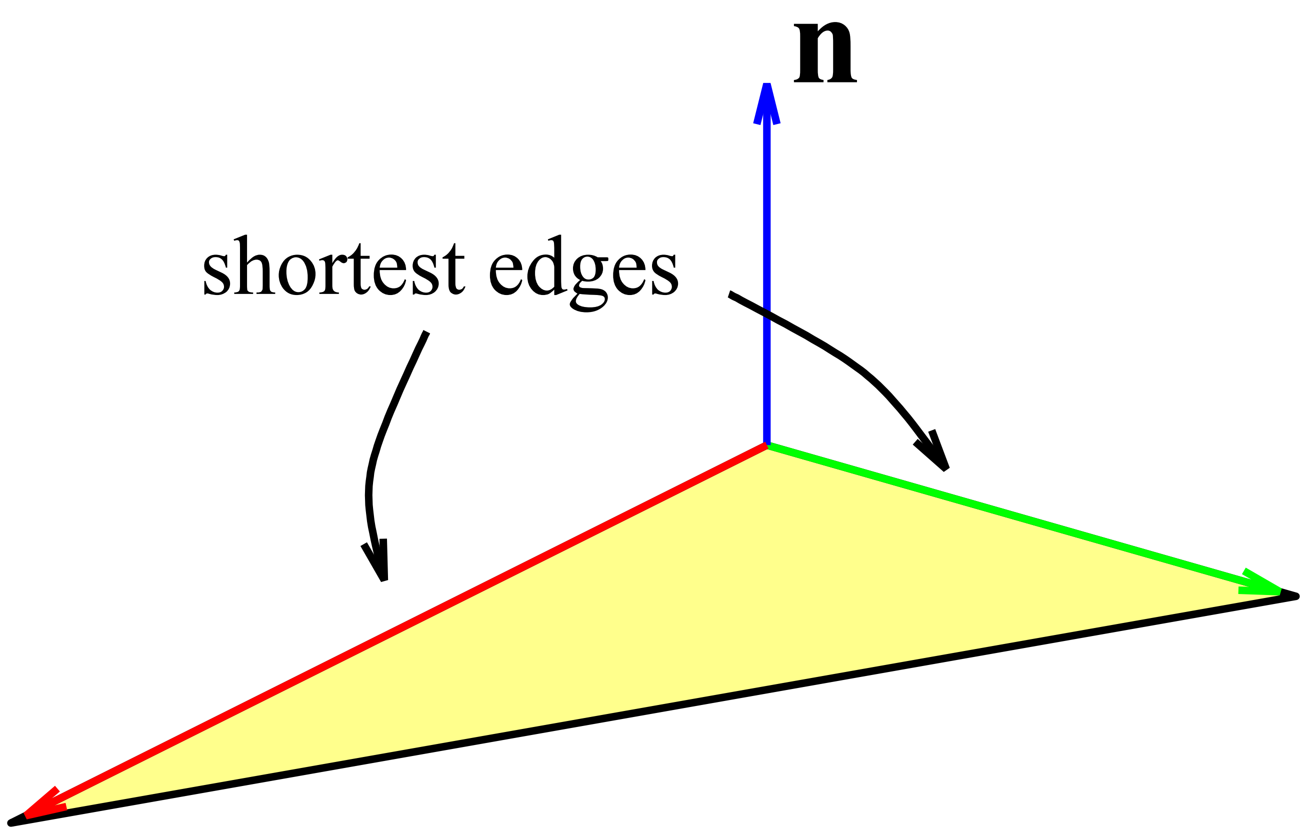
Example: Face Normals (cont'd)

- Choice of edges is arbitrary. Length of cross product is always twice the triangle's area.



Example: Face Normals (cont'd)

- Pick the two shortest edges for the smallest round-off error.



Order of Operations

- In floating-point arithmetic the following may not be true!

$$a + (b + c) = (a + b) + c$$
$$a(b + c) = ab + ac$$

- The order in which operations are evaluated can have a great effect on the error.

Example: Determinants in GJK

- Johnson's algorithm in GJK computes determinants as products of $\mathbf{y}_i \cdot (\mathbf{y}_j - \mathbf{y}_k)$.
- Expressing these factors as $\mathbf{y}_i \cdot \mathbf{y}_j - \mathbf{y}_i \cdot \mathbf{y}_k$ is way less robust!
- Factorize! Always try to perform additions and subtractions before multiplications.

Automatic Error Tracing in C++

- Make floating-point types abstract types.
- Quickly tell a numerical issue from a bug by substituting double or higher precision.
- Maintain a bound for the relative error by substituting the *ErrorTracer* proxy class.

Abstract Numerical Types

- Never use built-in floating-point types, such as `float` or `double`, explicitly.
- Rather, use a type name, e.g. `Scalar`:

```
using Scalar = float;
```

And hide the actual float type in your code.

Abstract Numerical Types (cont'd)

- Never use `float` literals, C-style casts, or `static_cast` for initialization or conversion, e.g. use

  ```
  Scalar(2),
  ```

  rather than `2.0f`, `(Scalar)2`, or `static_cast<Scalar>(2)`.

Abstract Numerical Types (cont'd)

- Use a traits class for type-dependent constants, e.g. use

  `std::numeric_limits<Scalar>::epsilon()`

  rather than `FLT_EPSILON`.

Abstract Numerical Types (cont’d)

- Use the overloaded C++ math functions from `<cmath>` rather than the C math functions from `<math.h>`, e.g use

  `sqrt(x)` or `std::sqrt(x)`,

  rather than `sqrtf(x)` or `std::sqrtf(x)`.

ErrorTracer<T>

```
template <typename T>
class ErrorTracer
{
   …
private:
     T mValue; // value of the scalar
     T mError; // max. relative error
};
```

ErrorTracer<T>: Operators

```
template <typename T>
ErrorTracer<T> operator-(const ErrorTracer<T>& x,
                         const ErrorTracer<T>& y)
{
    T value = x.value() - y.value();
    T error = abs(x.value()) * x.error() +
              abs(y.value()) * y.error();
    return ErrorTracer<T>(value,
          !iszero(value) ? error / abs(value) + T(1) : T());
}
```

ErrorTracer<T>: Math Functions

```
template <typename T>
ErrorTracer<T> sqrt(const ErrorTracer<T>& x)
{
    return ErrorTracer<T>(sqrt(x.value()),
                          x.error() * T(0.5) + T(1));
}
```

ErrorTracer<T>

- ErrorTracer transparently replaces built-in types:

```
using Scalar = ErrorTracer<float>;
```

ErrorTracer<T> Reporting

- ErrorTracer reports the relative error

```
float r = x.maxRelativeError();
```

- And the number of contaminated bits

```
float b = x.dirtyBits();
```

True Relative Error

- FPUs may use higher precision for intermediate results (`FLT_EVAL_METHOD`).
- Therefore, the error returned by ErrorTracer may be hugely overestimated.
- Great for checking where precision is lost.
- YMMV, if you need tight upper bounds for error.

Conclusions

- Caution with floating-point types for position and absolute time.
- Choose a formulation that uses the smallest input values.
- Factorize! Additions and subtractions first.
- Abstract from numerical types in C++ code.
- Know the cause of precision loss.

References

- D. Goldberg. *What every computer scientist should know about floating-point arithmetic*. ACM Computing Surveys, 23(1):5-48, March 1991.
- C. Ericson. *Numerical Robustness for Geometric Calculations*. GDC 2005 Tutorial.
- G. van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann Publishers, 2003.
- G. van den Bergen. *Math for Game Programmers: Dual Numbers*. GDC 2013 Tutorial.

Thank You!

Check me out on

- Web: www.dtecta.com
- Twitter: @dtecta
- ErrorTracer C++ code available in MoTo: https://github.com/dtecta/motion-toolkit

Interval Arithmetic (bonus)

- Maintain an upper and lower bound of a computed value (true value included).
- Requires changing of FPU rounding policy.
- Tighter, yet computationally way more expensive, than ErrorTracer.
- Boost Interval Arithmetic Library implements this for C++.