/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2019 - Lecture 11: “Fixed Point Math”

Welcome!

Today's Agenda:

- Introduction
- Float to Fixed Point and Back
- Operations
- Fixed Point & Accuracy
- Demonstration

Introduction

The Concept of Fixed Point Math

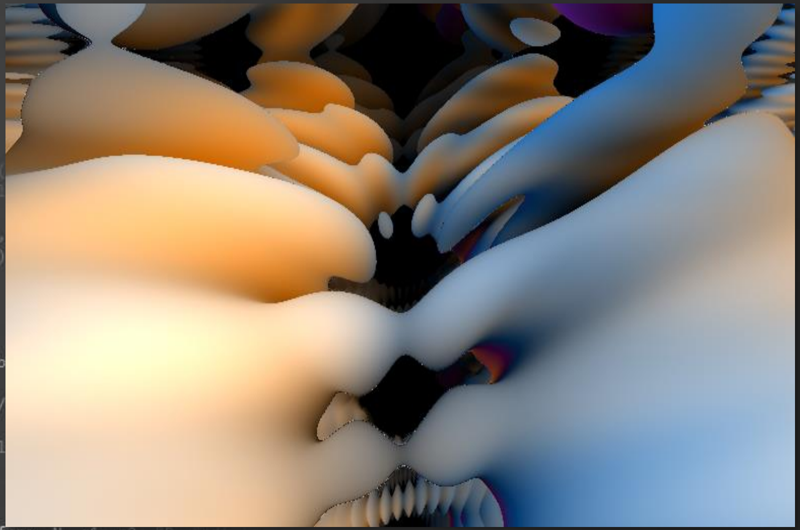
Basic idea: *emulating floating point math using integers*.

Why?

- Not every CPU has a floating point unit.
- Specifically: cheap DSPs do not support floating point.
- Mixing floating point and integer is Good for the Pipes.
- Some floating point ops have long latencies (div).
- Data conversion can be a significant part of a task.
- Fixed point can be more accurate.

Introduction

Turing introduces a new processor architecture, the **Turing SM**, that d
shading efficiency, achieving 50% improvement in delivered performa
compared to the Pascal generation. These improvements are enabled
changes. First, the Turing SM adds a new independent integer datapat
instructions concurrently with the floating-point math datapath. In pre
executing these instructions would have blocked floating-point instruc
the SM memory path has been redesigned to unify shared memory, te
load caching into one unit. This translates to 2x more bandwidth and r
available for L1 cache for common workloads.



```
float f(vec3 p)
{
    p.z+=iTime;return length(.05*cos(9.*p.y*p.x)+cos(p)-.1*cos(9.*(p.z+.3*p.x-p.y)))-1.;
}
void mainImage( out vec4 c, vec2 p )
{
    vec3 d=.5-vec3(p,1)/iResolution.x,o=d;for(int i=0;i<128;i++)o+=f(o)*d;
    c.xyz = abs(f(o-d)*vec3(0,1,2)+f(o-.6)*vec3(2,1,0))*(1.-.1*o.z);
}
```

Could we evaluate function f *without using floats?*

Introduction

The Concept of Fixed Point Math

Basic idea: we have π: 3.1415926536.

- Multiplying that by 10^{10} yields 31415926536.
- Adding 1 to π yields 4.1415926536.
- But, we scale up 1 by 10^{10} as well:
  adding $1 \cdot 10^{10}$ to the scaled up version of π yields 41415926536.

➔ In base 10, we get N digits of fractional precision if we multiply our numbers by 10^N (and remember where we put that dot).

Introduction

The Concept of Fixed Point Math

Addition and subtraction are straight-forward with fixed point math.

We can also use it for interpolation:

```
void line( int x1, int y1, int x2, int y2 )
{
   int dx = (x2 - x1) * 10000;
   int dy = (y2 - y1) * 10000;
   int pixels = max( abs( x2 - x1 ), abs( y2 - y1 ) );
   dx /= pixels;
   dy /= pixels;
   int x = x1 * 10000, y = y1 * 10000;
   for( int i = 0; i < pixels; i++, x += dx, y += dy )
      plot( x / 10000, y / 10000 );
}
```

Introduction

The Concept of Fixed Point Math

For multiplication and division things get a bit more complex.

- $\pi \cdot 2 \equiv 31415926536 * 20000000000 = 628318530720000000000$
- $\pi / 2 \equiv 31415926536 / 20000000000 = 1$ (or 2, if we use proper rounding).

Multiplying two fixed point numbers yields a result that is 10^{10} too large (in this case).
Dividing two fixed point numbers yields a result that is 10^{10} too small.

Introduction

The Concept of Fixed Point Math

On a computer, we obviously do not use base 10, but base 2. Starting with π again:

- Multiplying by 2^{16} yields 205887.
- Adding $1 \cdot 2^{16}$ to the scaled up version of π yields 271423.

In binary:

- $205887 = 00000000\ 00000011\ 00100100\ 00111111$
- $271423 = 00000000\ 00000100\ 00100100\ 00111111$

Looking at the first number (205887), and splitting in two sets of 16 bit, we get:

- 0000000000000011 (base 2) = 3 (base 10);
- 10010000111111 (base 2) = 9279 (base 10); $\frac{9279}{2^{16}} = 0.141586304$.

Introduction

The Concept of Fixed Point Math

Interpolation, base 10:

```
void line( int x1, int y1, int x2, int y2 )
{
   int dx = (x2 – x1) * 10000;
   int dy = (y2 – y1) * 10000;
   int pixels = max( abs( x2 – x1 ), abs( y2 – y1 ) );
   dx /= pixels;
   dy /= pixels;
   int x = x1 * 10000, y = y1 * 10000;
   for( int i = 0; i < pixels; i++, x += dx, y += dy )
      plot( x / 10000, y / 10000 );
}
```

Introduction

The Concept of Fixed Point Math

Interpolation, base 2:

```
void line( int x1, int y1, int x2, int y2 )
{
   int dx = (x2 – x1) * 65536;
   int dy = (y2 – y1) * 65536;
   int pixels = max( abs( x2 – x1 ), abs( y2 – y1 ) );
   dx /= pixels;
   dy /= pixels;
   int x = x1 * 65536, y = y1 * 65536;
   for( int i = 0; i < pixels; i++, x += dx, y += dy )
      plot( x / 65536, y / 65536 );
}
```

Introduction

```
void line( int x1, int y1, int x2, int y2 )
{
   int dx = (x2 - x1) * 65536;
   int dy = (y2 - y1) * 65536;
   int pixels = max( abs( x2 - x1 ), abs( y2 - y1 ) );
   dx /= pixels;
   dy /= pixels;
   int x = x1 * 65536, y = y1 * 65536;
   for( int i = 0; i < pixels; i++, x += dx, y += dy )
      plot( x / 65536, y / 65536 );
}
```

The Concept of Fixed Point Math

How many bits do we need?

- The number 10.3 (base 10) has a maximum error of 0.05: $10.25 \leq 10.3 < 10.35$.
- So, the error is at most $\frac{1}{2}10^{-X}$ for x fractional digits.
- A fixed point number with 16 fractional bits has a maximum error of $\frac{1}{2}2^{-16}$.

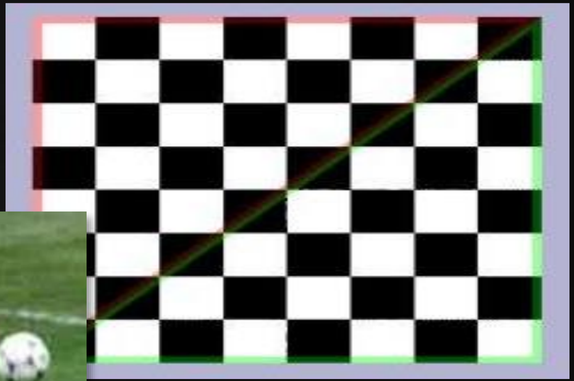
During interpolation:

If our longest line is Y pixels, the maximum error with X fractional bits is $\frac{1}{2}Y\ 2^{-X}$.
If the maximum error exceeds 1, the line may differ from ‘ground truth’.

Introduction

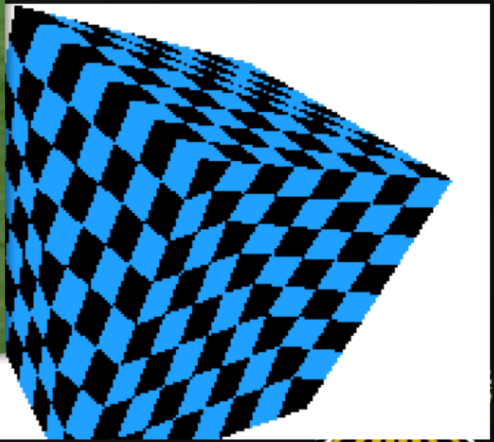
Practical example

Texture mapping in Quake 1: Perspective Correction



- Affine texture mapping: interpolate u/v linearly over polygon
- Perspective correct texture mapping: interpolate 1/z, u/z and v/z.
- Reconstruct u and v per pixel using the reciprocal of 1/z.

Instruction	Operand	Clock cycles	Pairability	i-ov	fp-ov
FLD	r/m32/m64	1	0	0	0
FLD	m80	3	np	0	0
FBLD	m80	48-58	np	0	0
FST(P)	r	1	np	0	0
FST(P)	m32/m64	2 m)	np	0	0
FST(P)	m80	3 m)	np	0	0
FBSTP	m80	148-154	np	0	0
FILD	m	3	np	2	2
FIST(P)	m	6	np	0	0
FLDZ FLD1		2	np	0	0
FLDPI FLDL2E etc.		5 s)	np	2	2
FNSTSW	AX/m16	6 q)	np	0	0
FLDCW	m16	8	np	0	0
FNSTCW	m16	2	np	0	0
FADD(P)	r/m	3	0	2	2
FSUB(R)(P)	r/m	3	0	2	2
FMUL(P)	r/m	3	0	2	2 n)
FDIV(R)(P)	r/m	19/33/39 p)	0	38 o)	2



Introduction

Practical example

Texture mapping in Quake 1: Perspective Correction

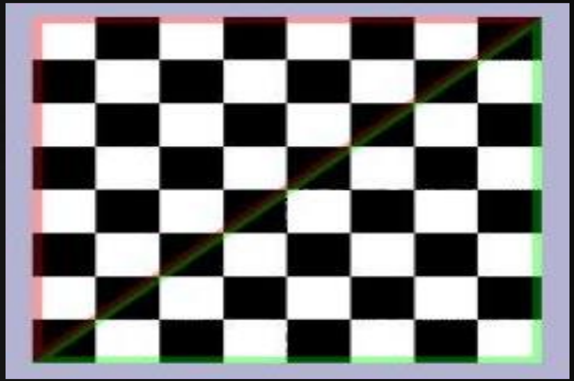
- Affine texture mapping: interpolate u/v linearly over polygon
- Perspective correct texture mapping: interpolate 1/z, u/z and v/z.
- Reconstruct u and v per pixel using the reciprocal of 1/z.

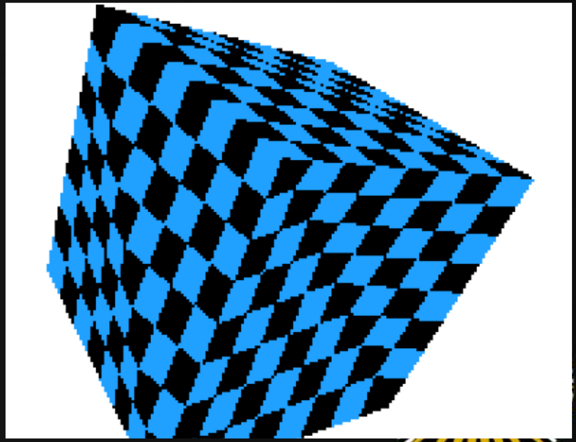
Quake’s solution:

- Divide a horizontal line of pixels in segments of 8 pixels;
- Calculate u and v for the start and end of the segment;
- Interpolate linearly (fixed point!) over the 8 pixels.

And:

Start the floating point division (39 cycles) for the next segment, so it can complete while we execute integer code for the linear interpolation.





Today's Agenda:

- Introduction
- Float to Fixed Point and Back
- Operations
- Fixed Point & Accuracy
- Demonstration

Conversions

Practical Things

Converting a floating point number to fixed point:

Multiply the float by a power of 2 <u>represented by a floating point value</u>, and cast the result to an integer. E.g.:

```
int fp_pi = (int)(3.141593f * 65536.0f); // 16 bits fractional
```

After calculations, cast the result to int by discarding the fractional bits. E.g.:

```
int result = fp_pi >> 16; // divide by 65536
```

Or, get the original float back by casting to float and dividing by $2^{fractionalbits}$:

```
float result = (float)fp_pi * (1.0f / 65536.0f);
```

Note that this last option has significant overhead, which should be outweighed by the gains.

Conversions

Practical Things - Considerations

Example: precomputed sin/cos table

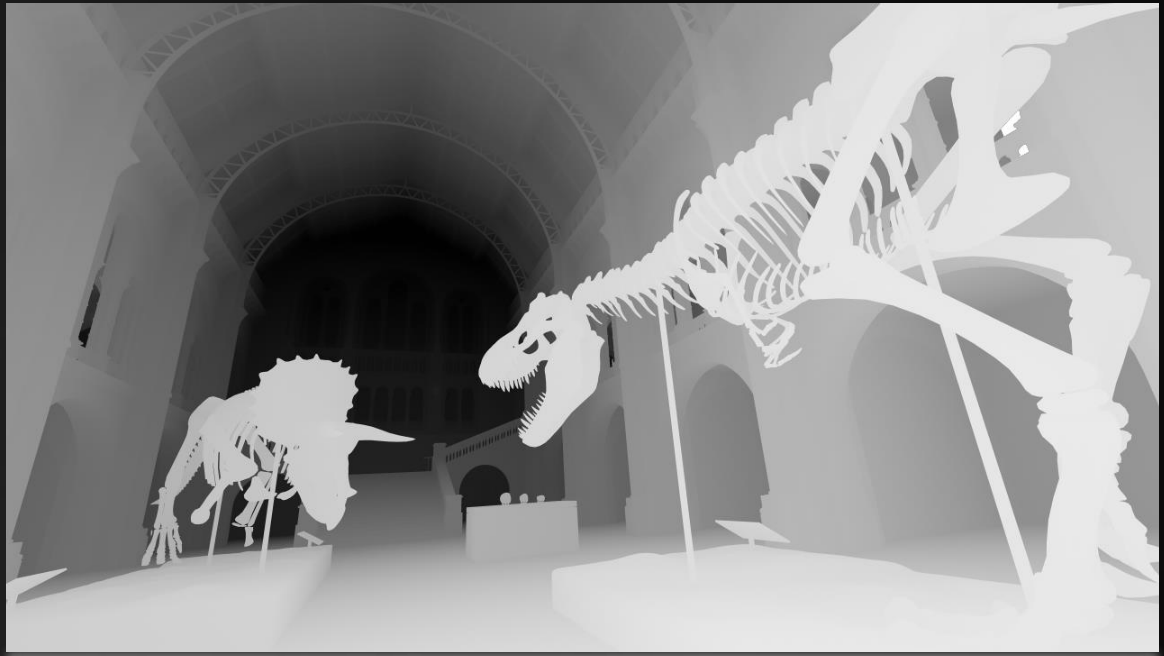
```
#define FP_SCALE 65536.0f   1073741824.0f
int sintab[256], costab[256];
for( int i = 0; i < 256; i++ )
   sintab[i] = (int)(FP_SCALE * sinf( (float)i / 128.0f * PI )),
   costab[i] = (int)(FP_SCALE * cosf( (float)i / 128.0f * PI ));
```

What is the best value for FP_SCALE in this case? And should we use `int` or `unsigned int` for the table?

Sine/cosine: range is [-1, 1]. In this case, we need 1 sign bit, and 1 bit for the whole part of the number. So:

➔ We use 30 bits for fractional precision, 1 for sign, 1 for range.
In base 10, the fractional precision is ~10 digits (float has 7).

Conversions



Practical Things - Considerations

Example: values in a z-buffer

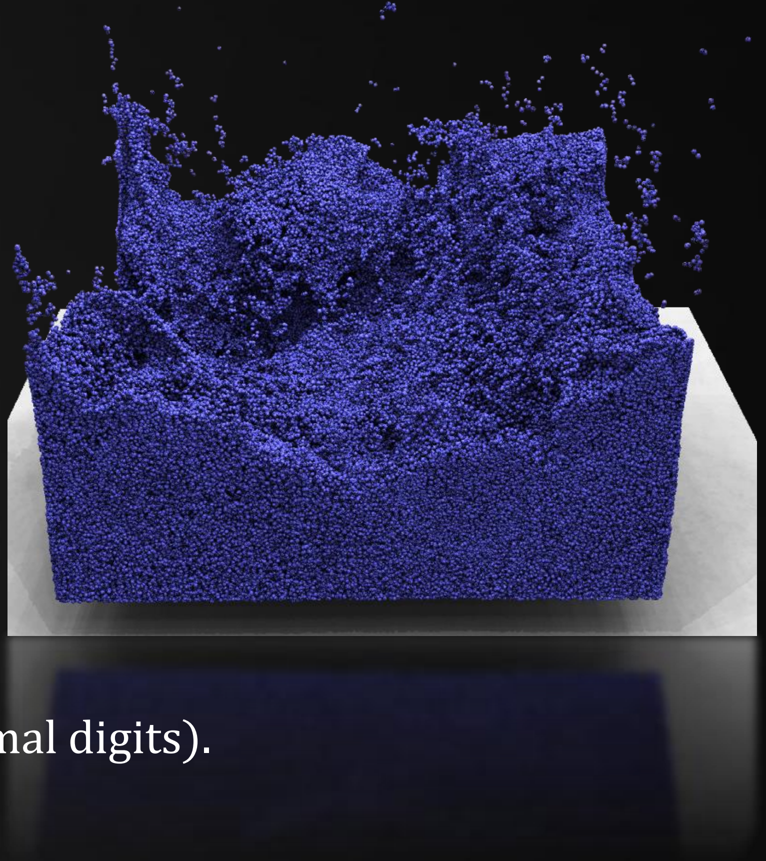
A 3D engine needs to keep track of the depth of pixels on the screen for depth sorting. For this, it uses a z-buffer.

We can make two observations:

1. All values are positive (no objects behind the camera are drawn);
2. Further away we need less precision.

By adding 1 to z, we guarantee that z is in the range [1..infinity].
The reciprocal of z is then in the range [0..1].
We store 1/(z+1) as a 0:32 unsigned fixed point number for maximum precision.

Conversions



Practical Things - Considerations

Example: particle simulation

Your particle simulation operates on particles inside a 100x100x100 box centered around the origin. What fixed point format do you use for the coordinates of the particles?

1. Since all coordinates are in the range [-50,50], we need a sign.
2. The maximum integer value of 50 fits in 6 bits.
3. This leaves 25 bits fractional precision (a bit more than 8 decimal digits).

➔ We use a 6:25 signed fixed point representation.

Better: scale the simulation to a box of 127x127x127 for better use of the full range; this gets you ~8.5 decimal digits of precision.

Conversions

Practical Things - Considerations

We pick the right precision based on the problem at hand.

Sin/cos: original values [-1..1];
➔ sign bit + 31 fractional bits;
➔ 0:31 signed fixed point.

Storing 1/(z+1): original values [0..1];
➔ 32 fractional bits;
➔ 0:32 unsigned fixed point.

Particles: original values [-50..50];
➔ sign bit + 6 integer bits, 32-7=25 fractional bits;
➔ 6:25 signed fixed point.

In general:

- first determine if we need a sign;
- then, determine how many bits are need to represent the integer range;
- use the remainder as fractional bits.

Today's Agenda:

- Introduction
- Float to Fixed Point and Back
- Operations
- Fixed Point & Accuracy
- Demonstration

Operations

Basic Operations on Fixed Point Numbers

Operations on mixed fixed point formats:

- A+B $(I_A:F_A + I_B:F_B)$

To be able to add the numbers, they need to be in the same format.

Example: $I_A:F_A$=4:28, $I_B:F_B$=16:16

Option 1: A >>= 12 (to make it 16:16)
Option 2: B <<= 12 (to make it 4:28)

Problem with option 2: we do not get 4:28, we get 16:28!
Problem with option 1: we drop 12 bits from A.

Operations

Basic Operations on Fixed Point Numbers

Operations on mixed fixed point formats:

- A∗B $(I_A : F_A * I_B : F_B)$

We can freely mix fixed point formats for multiplication.

Example: $I_A : F_A$=18:14, $I_B : F_B$=14:18
Result: 32:32, shift to the right by 18 to get a ..:14 number, or by 14 to get a ..:18 number.

Problem: the intermediate result doesn't fit in a 32-bit register.

Operations

Multiplication

Color scaling, base 2:

```
uint ScaleColor( const uint c, const uint x ) // x = 0..255
{
   uint redblue = c & 0x00FF00FF;
   uint green   = c & 0x0000FF00;
   redblue = (redblue * x) & 0xFF00FF00;
   green = (green * x) & 0x00FF0000;
   return (redblue + green) >> 8;
}
```

31 24	23 16	15 8	7 0
31 24	23 16	15 8	7

Operations

Multiplication

- *“Ensure that intermediate results never exceed 32 bits.”*

Suppose we want to multiply two 20:12 unsigned fixed point numbers:

```
1.  (fp_a * fp_b) >> 12;          // good if fp_a and fp_b are very small
2.  (fp_a >> 12) * fp_b;          // good if fp_a is a whole number
3.  (fp_a >> 6) * (fp_b >> 6);  // good if fp_a and fp_b are large
4.  ((fp_a >> 3) * (fp_b >> 3)) >> 6;
```

Which option we chose depends on the parameters:

```
fp_a = PI;
fp_b = 0.5f * 2^12;
int fp_prod = fp_a >> 1; // ☺
```

Operations

Division

- *“Ensure that intermediate results never exceed 32 bits.”*

Dividing two 20:12 fixed point numbers:

```
1. (fp_a << 12) / fp_b;         // good if fp_a and fp_b are very small
2. fp_a / (fp_b >> 12);         // good if fp_b is a whole number
3. (fp_a << 6) / (fp_b >> 6);  // good if fp_a and fp_b are large
4. ((fp_a << 3) / (fp_b >> 3)) << 6;
```

Note that a division by a constant can be replaced by a multiplication by its reciprocal:

```
fp_reci = (1 << 12) / fp_b;
fp_prod = (fp_a * fp_reci) >> 12; // or one of the alternatives
```

Operations

Multiplication, Take 2

- *"Use a 64-bit intermediate result."*

A∗B $(I_A : F_A * I_B : F_B)$

Example: $I_A : F_A$=16:16, $I_B : F_B$=16:16
Result: 32:32

Calculate a 64-bit result (with enough room for 32:32), throw out 32 bits afterwards.

x86 MUL instruction:

MUL EDX

Functionality:

multiplies EDX by EAX, stores the result in EDX:EAX.

➔ Tossing 32 bits: ignore EAX.
➔ x86 is designed for 16:16.

Operations

Multiplication

Special case: multiply by a 32:0 number.

```
int fp_pi = (int)(3.141593f * 65536.0f); // 16 bits fractional
int fp_2pi = fp_pi * 2;                  // 16 bits fractional
```

We did this in the line function:

```
dx /= pixels;  // dx is 16:16, pixels is 32:0
dy /= pixels;
```

Operations

Square Root

For square roots of fixed point numbers, optimal performance is achieved via _mm_rsqrt_ps (via float). If precision is of little concern, use a lookup table, optionally combined with interpolation and / or a Newton-Raphson iteration.

Sine / Cosine / Log / Pow / etc.

Almost always a LUT is the best option*.

*: Not on the GPU however. Alternative: https://www.coranac.com/2009/07/sines

Operations

Fixed Point & SIMD

For a world of hurt, combine SIMD and fixed point:

```
_mm_mul_epu32
_mm_mullo_epi16
_mm_mulhi_epu16
_mm_srl_epi32
_mm_srai_epi32
```

See MSDN for more details.

Today’s Agenda:

- Introduction
- Float to Fixed Point and Back
- Operations
- Fixed Point & Accuracy
- Demonstration

Accuracy

Range versus Precision

Looking at the line code once more:

```
void line( int x1, int y1, int x2, int y2 )
{
   int dx = (x2 - x1) << 16;
   int dy = (y2 - y1) << 16;
   int pixels = max( abs( x2 - x1 ), abs( y2 - y1 ) );
   dx /= pixels;
   dy /= pixels;
   int x = x1 << 16, y = y1 << 16;
   for( int i = 0; i < pixels; i++, x += dx, y += dy )
      plot( x >> 16, y >> 16 );
}
```

dx=15:16, range is 32767.

precision: 16 bits,
maximum error: $\frac{1}{2^{16}} * 0.5 = \frac{1}{2^{17}}$.
Interpolating a 1024 pixel line, the maximum cumulative error is $2^{10} \cdot \frac{1}{2^{17}} = \frac{1}{2^{7}} \approx 0.008$.

Accuracy

Range versus Precision: Error

In base 10, error is clear:

PI = 3.14 means: $3.145 > PI > 3.135$

The maximum error is thus $\frac{1}{2}\frac{1}{10^2} = 0.005$.

In base 2, we apply the same principle:

16:16 fixed point numbers have a maximum error of $\frac{1}{2}\frac{1}{2^{16}} = \frac{1}{2^{17}} \approx 7.6 \cdot 10^{-6}$.

➔ We get slightly more than 5 digits of decimal precision.

For reference: 32-bit floating point numbers:

- 1 sign bit, 8 exponent bits, 23 mantissa bits
- $2^{23} \approx 8{,}000{,}000$; floats thus have ~7 digits of decimal precision.

Accuracy

Range versus Precision: Error

During some operations, precision may suffer greatly:

$$x = y/z$$

$$fp\_x \;=\; (fp\_y \;<<\; 8)\,/\,(fp\_z \;>>\; 8)$$

Assuming 16:16 input, fp_z briefly becomes 16:8, with a precision of only 2 decimal digits.

Similarly:

$$fp\_x \;=\; (fp\_y \;>>\; 8) \,*\, (fp\_z \;>>\; 8)$$

Here, both fp_y and fp_z become 16:8, and the cumulative error may exceed $1/2^9$.

Accuracy

Error

Careful balancing of range and precision in fixed point calculations can reduce this problem.

Note that accuracy problems also occur in float calculations; they are just exposed more clearly in fixed point. And: this time we can do something about it.

Today's Agenda:

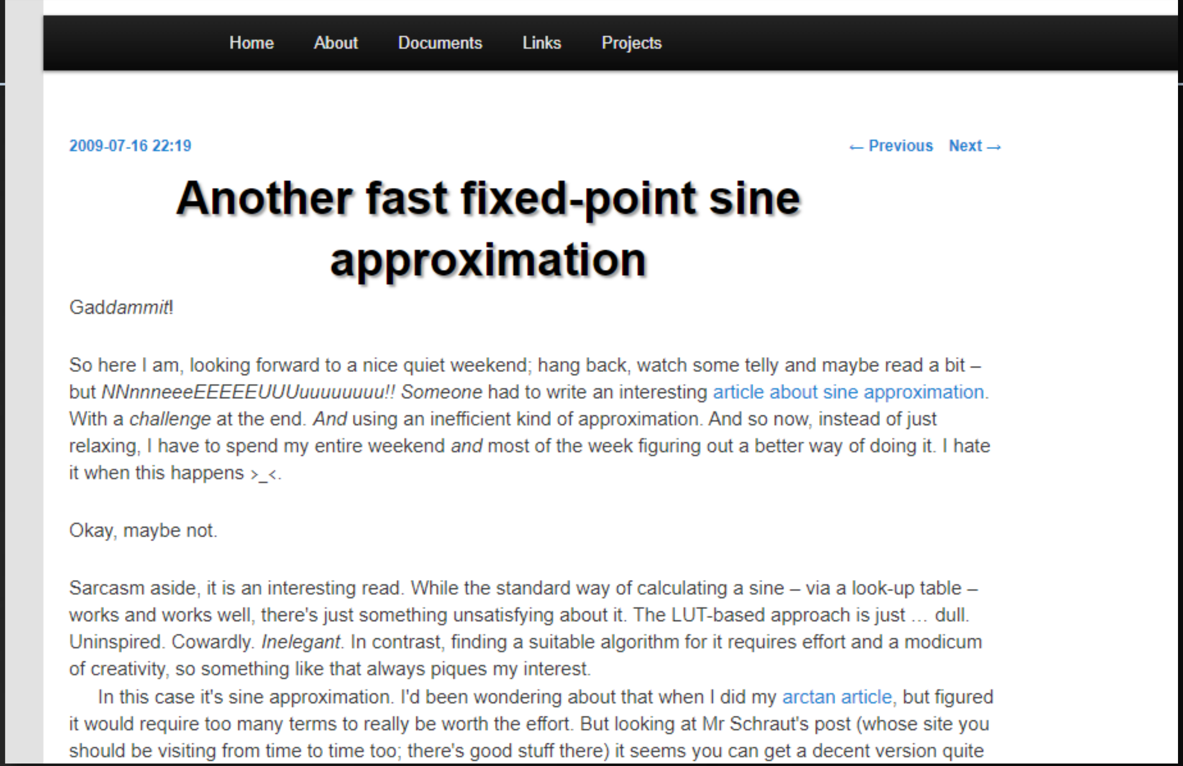
- Introduction
- Float to Fixed Point and Back
- Operations
- Fixed Point & Accuracy
- Demonstration

Demonstration

That Shader

Could it be done?

- Length means sqrt
- Cos, sin (LUT on GPU?)
- Vectors
- …



Home About Documents Links Projects

2009-07-16 22:19 ← Previous Next →

Another fast fixed-point sine approximation

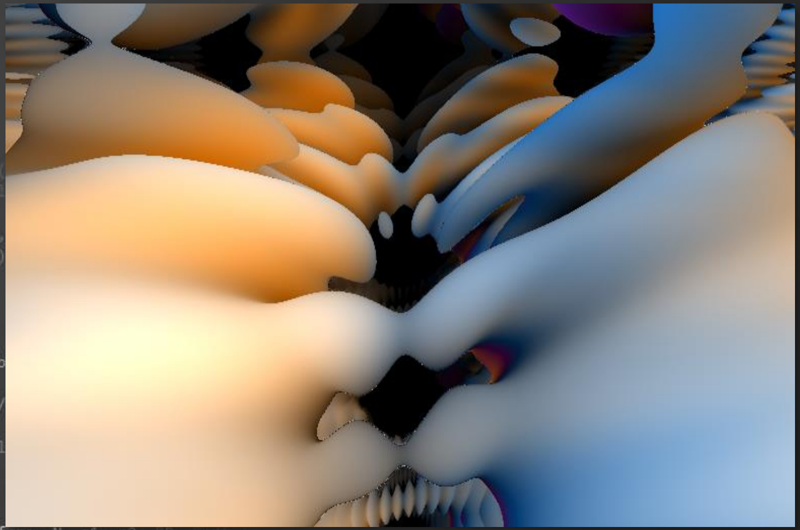
Gaddammit!

So here I am, looking forward to a nice quiet weekend; hang back, watch some telly and maybe read a bit – but *NNnnneeeEEEEEUUUuuuuuuuu!! Someone* had to write an interesting article about sine approximation. With a *challenge* at the end. *And* using an inefficient kind of approximation. And so now, instead of just relaxing, I have to spend my entire weekend *and* most of the week figuring out a better way of doing it. I hate it when this happens >_<.

Okay, maybe not.

Sarcasm aside, it is an interesting read. While the standard way of calculating a sine – via a look-up table – works and works well, there's just something unsatisfying about it. The LUT-based approach is just … dull. Uninspired. Cowardly. *Inelegant*. In contrast, finding a suitable algorithm for it requires effort and a modicum of creativity, so something like that always piques my interest.

In this case it's sine approximation. I'd been wondering about that when I did my arctan article, but figured it would require too many terms to really be worth the effort. But looking at Mr Schraut's post (whose site you should be visiting from time to time too; there's good stuff there) it seems you can get a decent version quite



```
float f(vec3 p)
{
    p.z+=iTime;return length(.05*cos(9.*p.y*p.x)+cos(p)-.1*cos(9.*(p.z+.3*p.x-p.y)))-1.;
}
void mainImage( out vec4 c, vec2 p )
{
    vec3 d=.5-vec3(p,1)/iResolution.x,o=d;for(int i=0;i<128;i++)o+=f(o)*d;
    c.xyz = abs(f(o-d)*vec3(0,1,2)+f(o-.6)*vec3(2,1,0))*(1.-.1*o.z);
}
```

Today's Agenda:

- Introduction
- Float to Fixed Point and Back
- Operations
- Fixed Point & Accuracy
- Demonstration

/INFOMOV/

END of “Fixed Point Math”

next lecture: “Snippets”