



```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

at a = nt - nc; b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn *
E * diffuse;
= true;
-
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth <
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

# Today's Agenda:

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



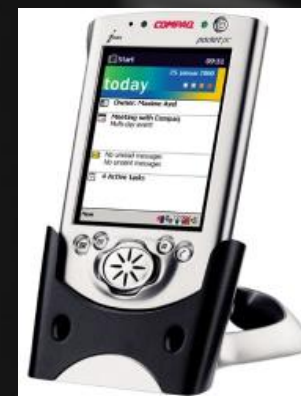
# Introduction

## 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 delivers a dramatic boost in shading efficiency, achieving 50% improvement in delivered performance per CUDA Core compared to the Pascal generation. These improvements are enabled by two key architectural changes. First, the Turing SM adds a new independent integer datapath that can execute instructions concurrently with the floating-point math datapath. In previous generations, executing these instructions would have blocked floating-point instructions from issuing. Second, the SM memory path has been redesigned to unify shared memory, texture caching, and memory load caching into one unit. This translates to 2x more bandwidth and more than 2x more capacity available for L1 cache for common workloads.

```
vec3 shade( vec3 V, vec3 R )
{
    float spec = pow( max( dot( V, R ), 0 ), 32 );
    return spec * lightColor;
}
```

**Could we evaluate function shade *without using floats*?**



# Introduction

## The Concept of Fixed Point Math

Basic idea: we have  $\pi$ : 3.1415926536.

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

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

```

ics
& (depth < MAXDEPTH)
{
    int = inside ? 1 : 0;
    nt = nt / nc; ddn = ddn * 2;
    ps2t = 1.0f - nnt * nnt;
    D, N );
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (ddn
E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light;
e.x + radiance.y + radiance.z) > 0) && (survive)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```





# 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.

```

ics
& (depth < MAXDEPTH)
{
    int inside = 1;
    int nt = nt / nc;
    double nnt = 1.0f * nnt;
    double N;
    double R;
    double a = nt - nc;
    double Tr = 1 - (R0 + (1 - R0) * R);
    double R = (D * nnt - N * (dd));
    E * diffuse;
    = true;
    refl + refr)) && (depth < MAXDEPTH)
    D, N);
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &align);
    e.x + radiance.y + radiance.z) > 0) && (abs(
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following
    ive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```



# Introduction

## The Concept of Fixed Point Math

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

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

In binary:

- $205887 = 00000000\ 00000011\ 00100100\ 00111111$
- $271423 = 00000000\ 00001000\ 00100100\ 00111111$

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

- $00000000000011$  (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

## 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  $\frac{1}{2} 2^{-16} = 2^{-17}$ .
- ...or  $2^{-16}$  if we always round down (twice as much!).
- This can be prevented by adding  $\frac{1}{2} 2^{-16}$  before flooring:  
 $\text{round}(10.7) = \text{floor}(10.7 + 0.5) = 11$ .

During interpolation:

If our longest line is Y pixels,

→ the maximum accumulated error with x fractional bits is:  $Y 2^{-(x+1)}$ .

**\*Only\*** if the maximum error exceeds 1, the line may differ from ‘ground truth’.

```
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

$$u = u\_over\_z / z \quad rz = 1 / z$$

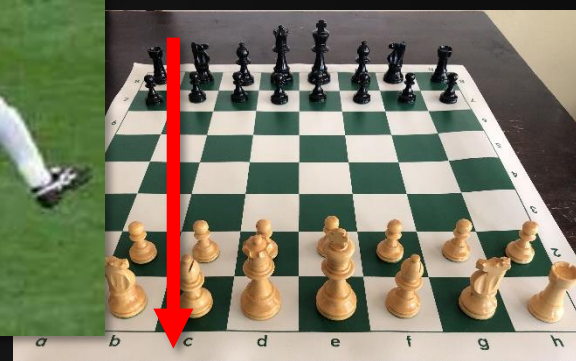
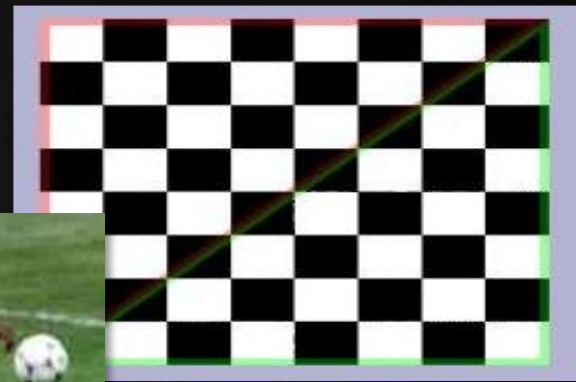
$$v = v\_over\_z / z \quad u = u\_over\_z * rz$$

$$v = v\_over\_z * rz$$

## 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)
<b>FDIV(R)(P)</b>	<b>r/m</b>	<b>19/33/39 p)</b>	0	38 o)	2





# Introduction

## Practical example

### Texture mapping in Quake 1: Perspective Correction

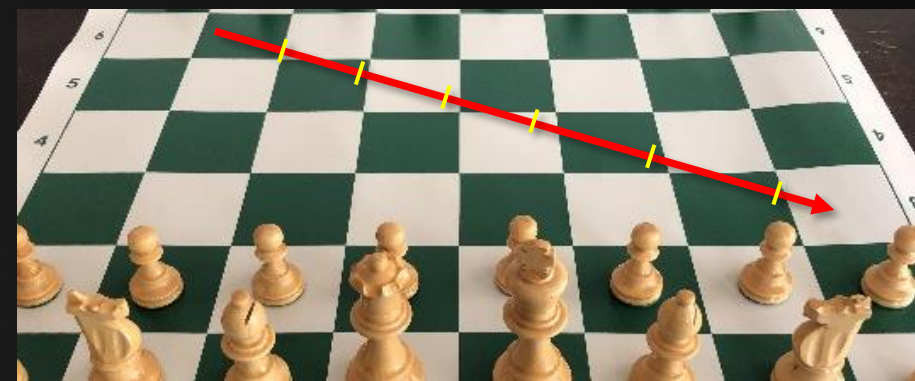
- Affine texture mapping: interpolate  $u/v$  linearly over polygon
- Perspective correct texture mapping: interpolate  $1/z$ ,  $u/z$  and  $v/z$ .
- Reconstruct  $u$  and  $v$  per pixel using the reciprocal of  $1/z$ .

### 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.



```

...ics
& (depth < MAXDEPTH)
...
c = inside ? 1 : -1;
nt = nt / nc; ddn = ddn * c;
cos2t = 1.0f - nnt * nnt;
D, N );
)
...
at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * c);
Tr) R = (D * nnt - N * (ddn * c));
...
E * diffuse;
= true;
...
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, closely following
df;
radiance = SampleLight( &rand, I, &L, &light,
e.x + radiance.y + radiance.z) > 0) && (survive);
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurf;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) *
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn * ddn));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

# 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 represented by a floating point value, and cast the result to an integer. E.g.:

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

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<sup>fractionalbits</sup>:

```
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.

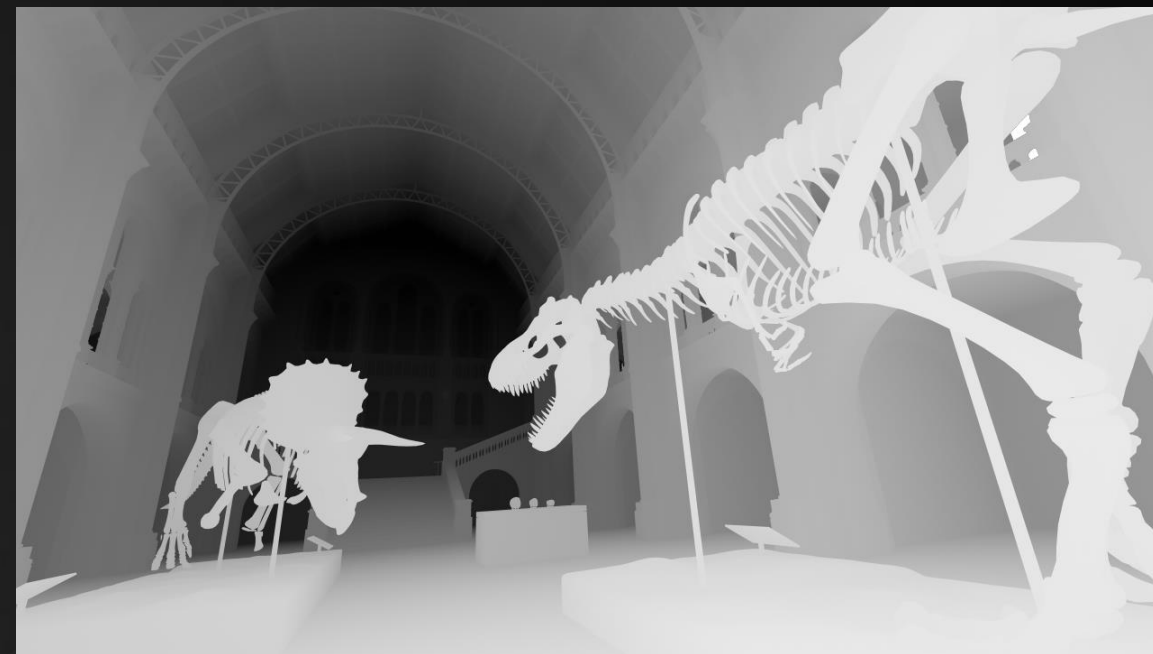
We can make two observations:

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.



```

ics
& (depth < MAXDEPTH)
c = inside ? 1.0f : 0.0f;
nt = nt / nc; ddn = ddn * c;
ps2t = 1.0f - nnt * ddn;
D, N );
0);
at a = nt - nc, b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * c);
Tr) R = (D * nnt - N * (ddn *
E * diffuse;
= true;
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, closely
if;
radiance = SampleLight( &rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (survive
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
andom walk - done properly, closely following
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Conversions

## Practical Things - Considerations

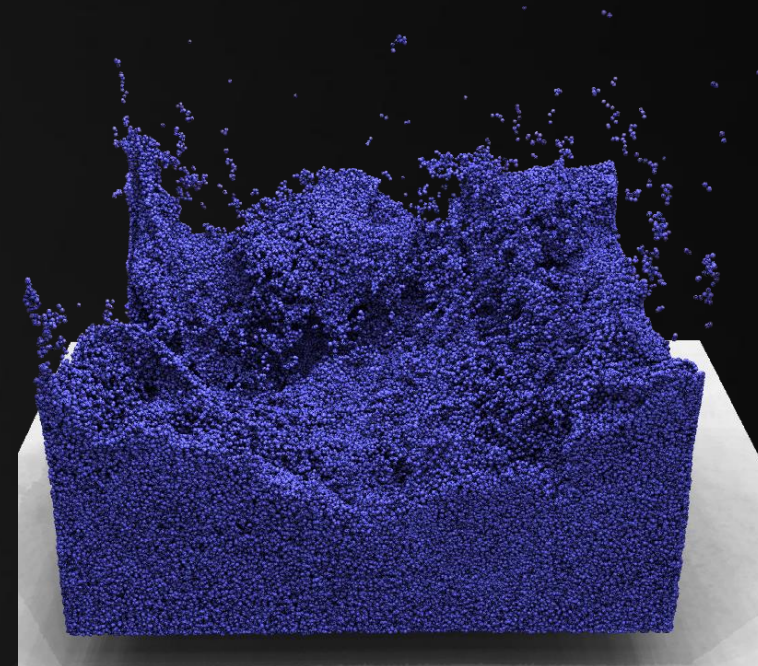
### Example: particle simulation

A 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.



```

ics
& (depth < MAXDEPTH)
{
    int = inside ? 1 : 0;
    nt = nt / nc; ddn = ddn * ddn;
    cos2t = 1.0f - nnt;
    D, N );
    )
    at a = nt - nc; b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
    E * diffuse;
    = true;
    -
    efl + refr) && (depth < MAXDEPTH)
    D, N );
    efl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, closely
    if;
    radiance = SampleLight( &rand, I, &L, &light
    e.x + radiance.y + radiance.z) > 0) && (rand
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psum
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Section
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```





# 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.
- If too imprecise: use 64-bit integers (use sparsely on platforms that do not support this natively!).



```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn * ddn));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

# 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 must be in the same format.

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

Option 1:  $A \gg= 12$  (to make it 16:16)

Option 2:  $B \ll= 12$  (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:

- $A * B (I_A : F_A * I_B : F_B) \rightarrow \text{yields } (I_A + I_B) : (F_A + F_B)$

We can however freely mix fixed point formats for multiplication.

Example:

$$I_A : F_A = 16 : 16, I_B : F_B = 16 : 16$$

Result: 32:32, shift to the right by 32 to get a 32:16 number.

$$I_A : F_A = 18 : 14, I_B : F_B = 14 : 18$$

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.
- The answer may not fit in a 32-bit register.



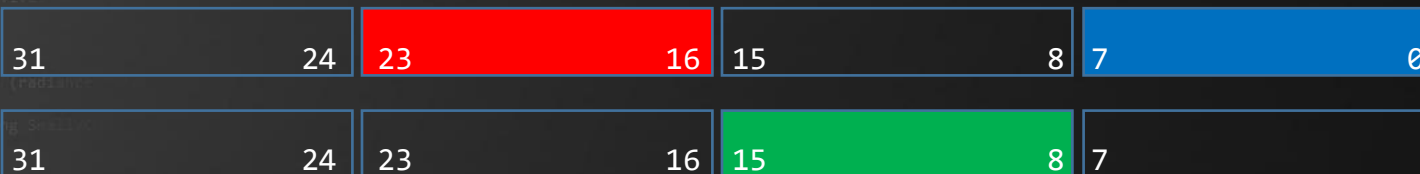
# Operations

## 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;
}
```

We did 0:8 \* 0:8 fixed point multiplication here, which yields 0:16 results.



# 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_reciprocal = (1 << 12) / fp_b;`
- `fp_prod = (fp_a * fp_reciprocal) >> 12;` // or one of the alternatives



# Operations

## Multiplication, Take 2

- *“Use a 64-bit intermediate result.”*

$$A * B \quad (I_A : F_A * I_B : F_B)$$

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 or divide 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\*.

...But, if you must: <https://github.com/chmike/fpsqrt>

```
int sqrt_i32( int v )
{
    uint b = 1<<30, q = 0, r = v;
    while (b > r) b >>= 2;
    while (b > 0)
    {
        uint t = q + b;
        q >>= 1;
        if (r >= t) r -= t, q += b;
        b >>= 2;
    }
    return q;
}
```

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



# Operations

## Fixed Point & SIMD

For a delicious 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 / Intel Intrinsic list for more details.

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn / nc;
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }

    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
    E * diffuse;
    = true;
    -
    refl + refr) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse, i);
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &light);
    e.x + radiance.y + radiance.z) > 0) && (cosThetaOut > 0)
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Serfling's
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

at a = nt - nc; b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn * ddn));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

# 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  $2^{10} * \frac{1}{2^{17}} = \frac{1}{2^7} \approx 0.008$ .

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn / nc;
        ps2t = 1.0f - nnt * nnt;
        D, N );
    }
}

at a = nt - nc, b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * nnt);
Tr) R = (D * nnt - N * (ddn * nnt + ps2t)) / (D * nnt - N * (ddn * nnt + ps2t));
E * diffuse;
= true;
refl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse, nnt );
estimation - doing it properly, closely following the path of the photon
if;
radiance = SampleLight( &rand, I, &L, &align );
e.x + radiance.y + radiance.z > 0) && (survive)
}

v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance.x + radiance.y + radiance.z);
random walk - done properly, closely following the path of the photon
survive)
};
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```



# 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

## 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.

```

ics
& (depth < MAXDEPTH)
c = inside ? 1.0f : 0.0f;
nt = nt / nc; ddn = ddn * ddn;
os2t = 1.0f - nnt * nnt;
D, N );
);
at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * c);
Tr) R = (D * nnt - N * (ddn *
E * diffuse;
= true;
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse, i);
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (cosThetaOut > 0)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn * ddn));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

# Today's Agenda:

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



/INFOMOV/

# END of “Fixed Point Math”

next lecture: “Guest Lecture”

```
ics
& (depth < MAXDEPTH)
{
    if ( ! inside )
    {
        nt = nt / nc; ddn = ddn * 2;
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }
}

at a = nt - nc; b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn * ddn));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &align,
e.x + radiance.y + radiance.z) > 0) && (abs(
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

