Advanced .NET Performance Optimization Techniques

Erik Freed

2012.11.07

Why do we care?

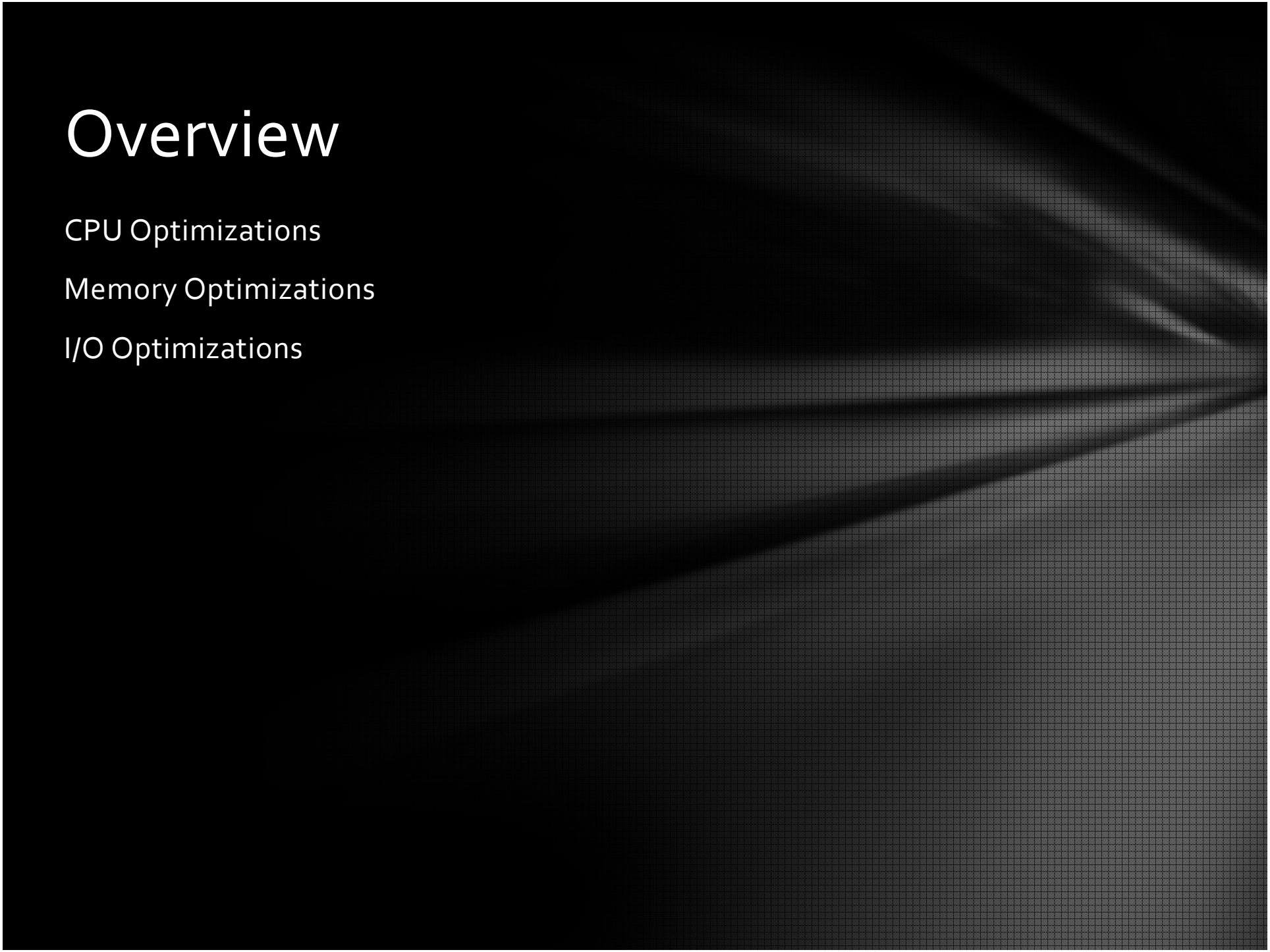
- Ground Guidance is very processor- and disk-intensive
- Ground Guidance runs on limited hardware
- 90-10 Rule

Overview

CPU Optimizations

Memory Optimizations

I/O Optimizations



Measure, Measure, Measure!

- Many variables affect performance significantly
  - Hardware
    - Memory, CPU, HDD/SSD
  - Debug vs. Release

The Elegant .NET Compiler

Constant folding

Code motion of loop invariants

Dead store and dead code elimination

Register allocation

Method inlining

Loop unrolling

Range Check Elimination

CPU Operations

Arithmetic Operations

Conditional Operations

Trigonometric Functions

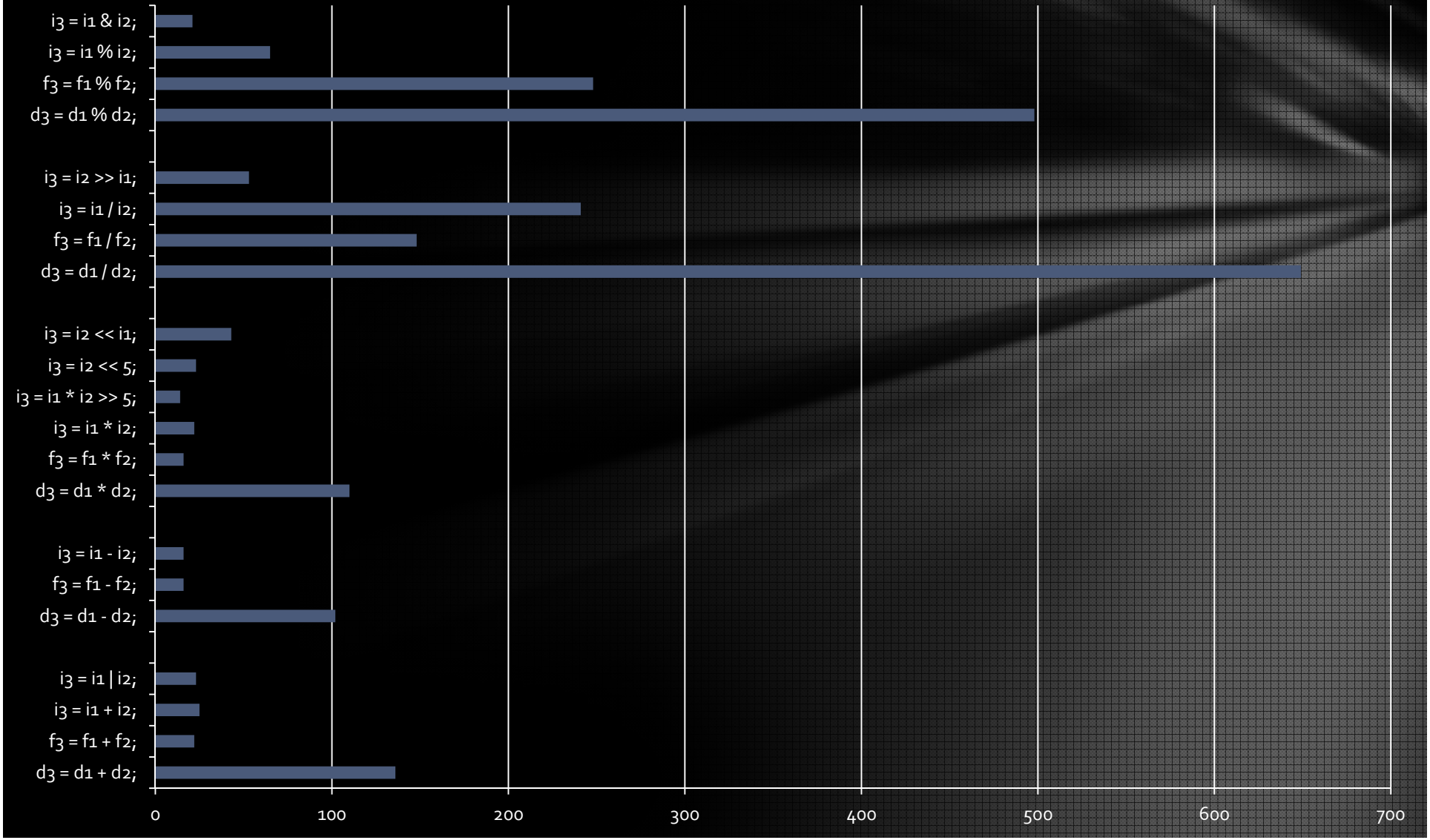
Power Functions

Type Conversion

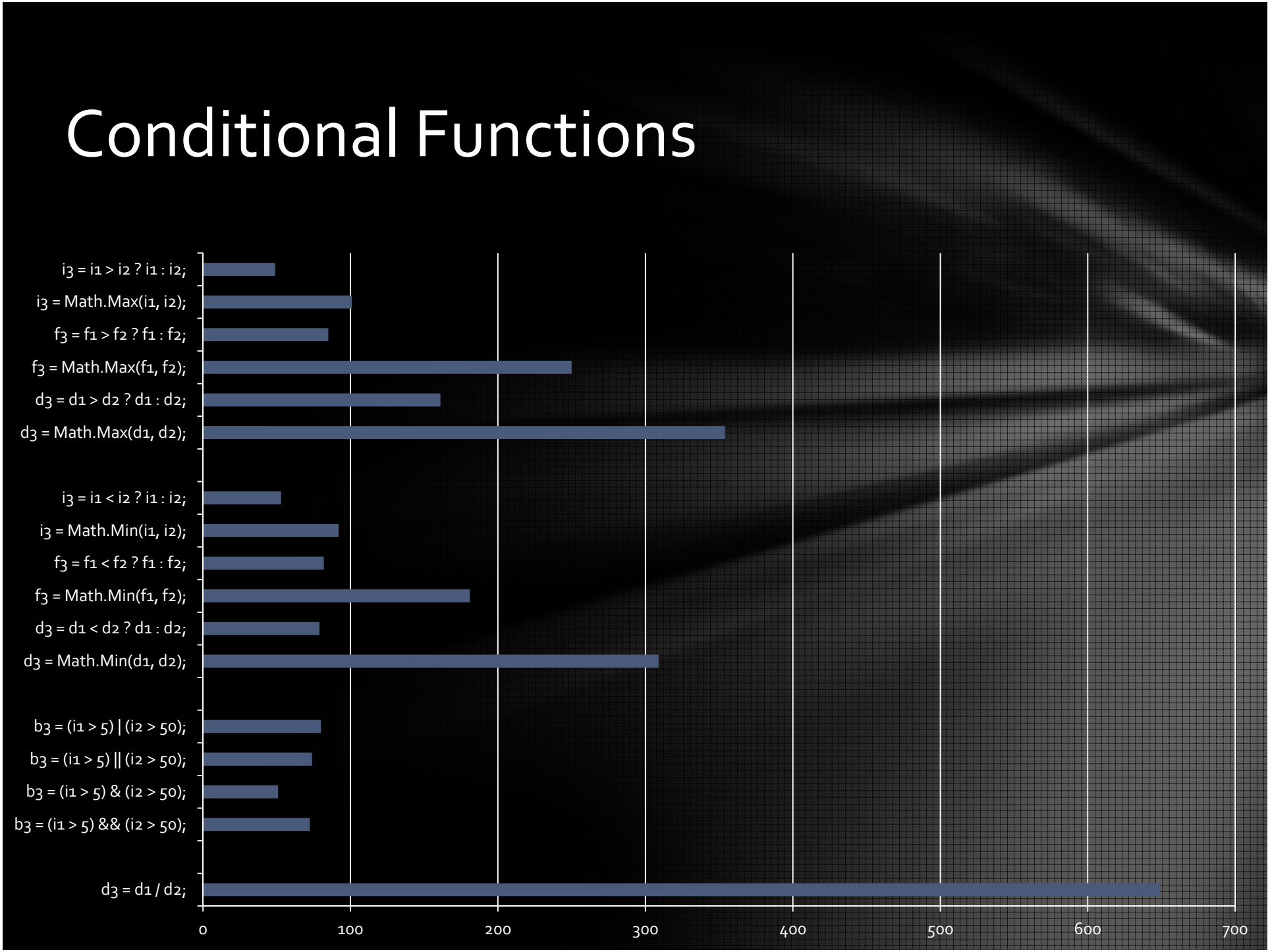
Point/List Construction

String Construction

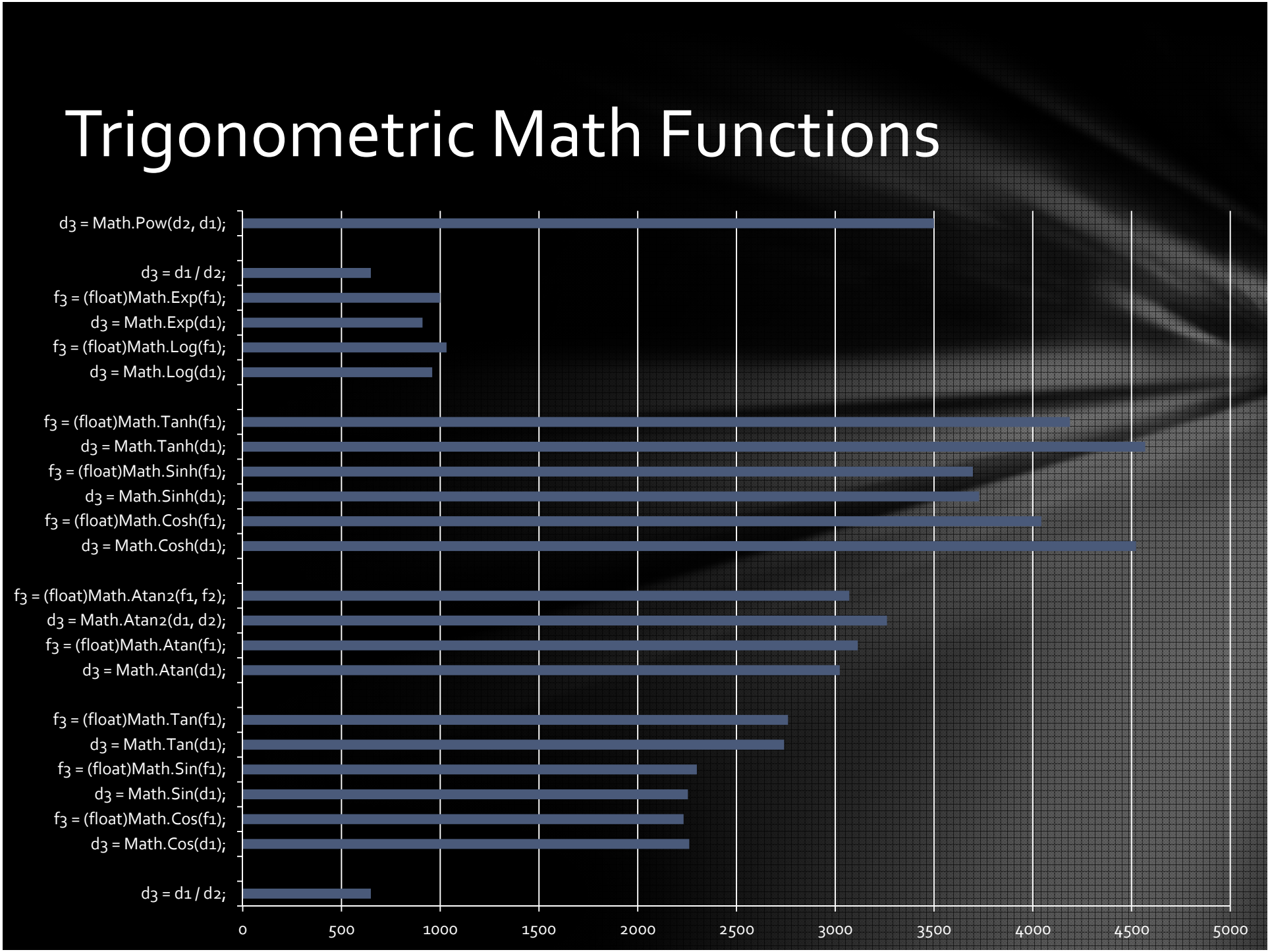
Arithmetic Operations



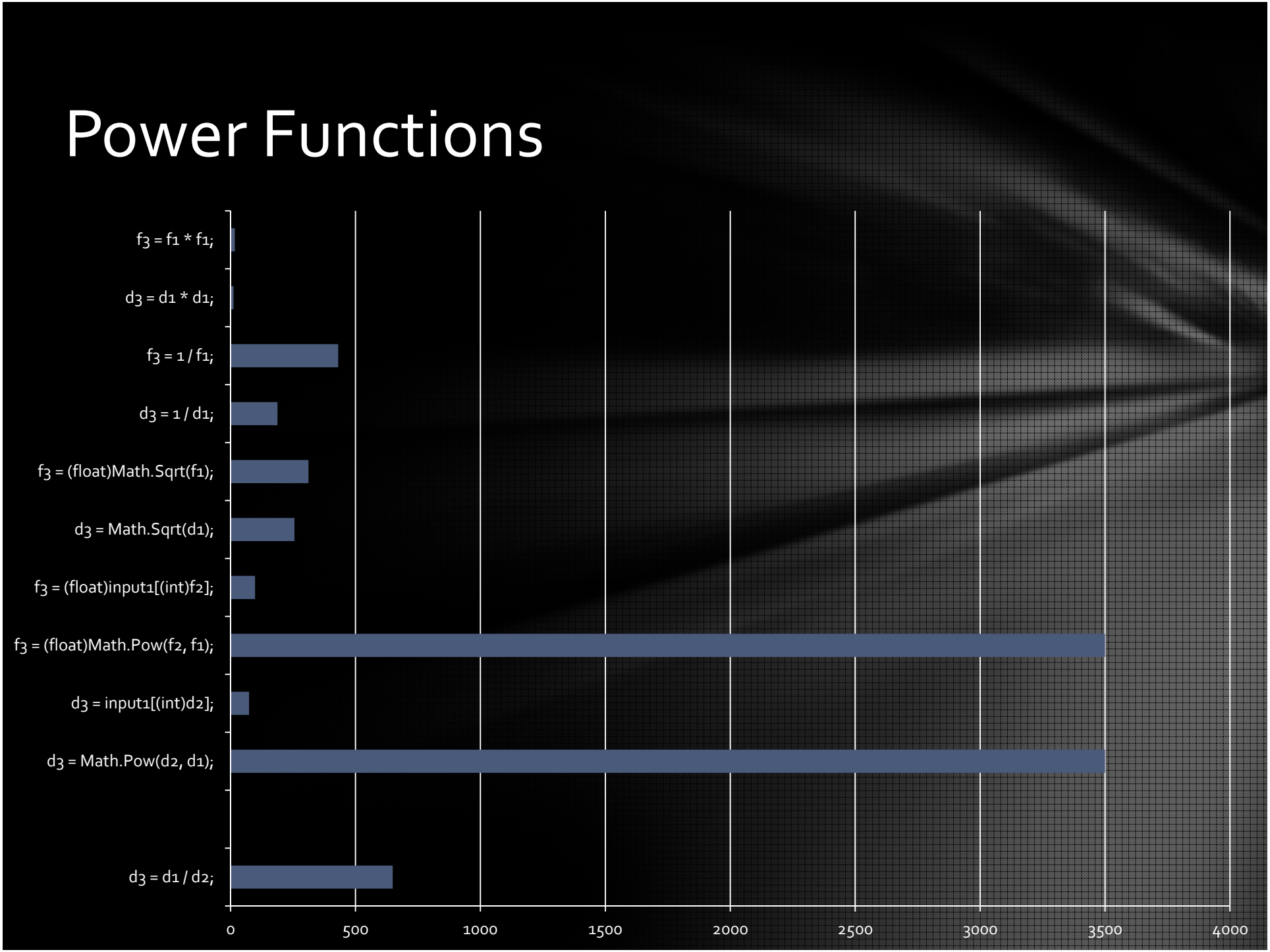
Conditional Functions



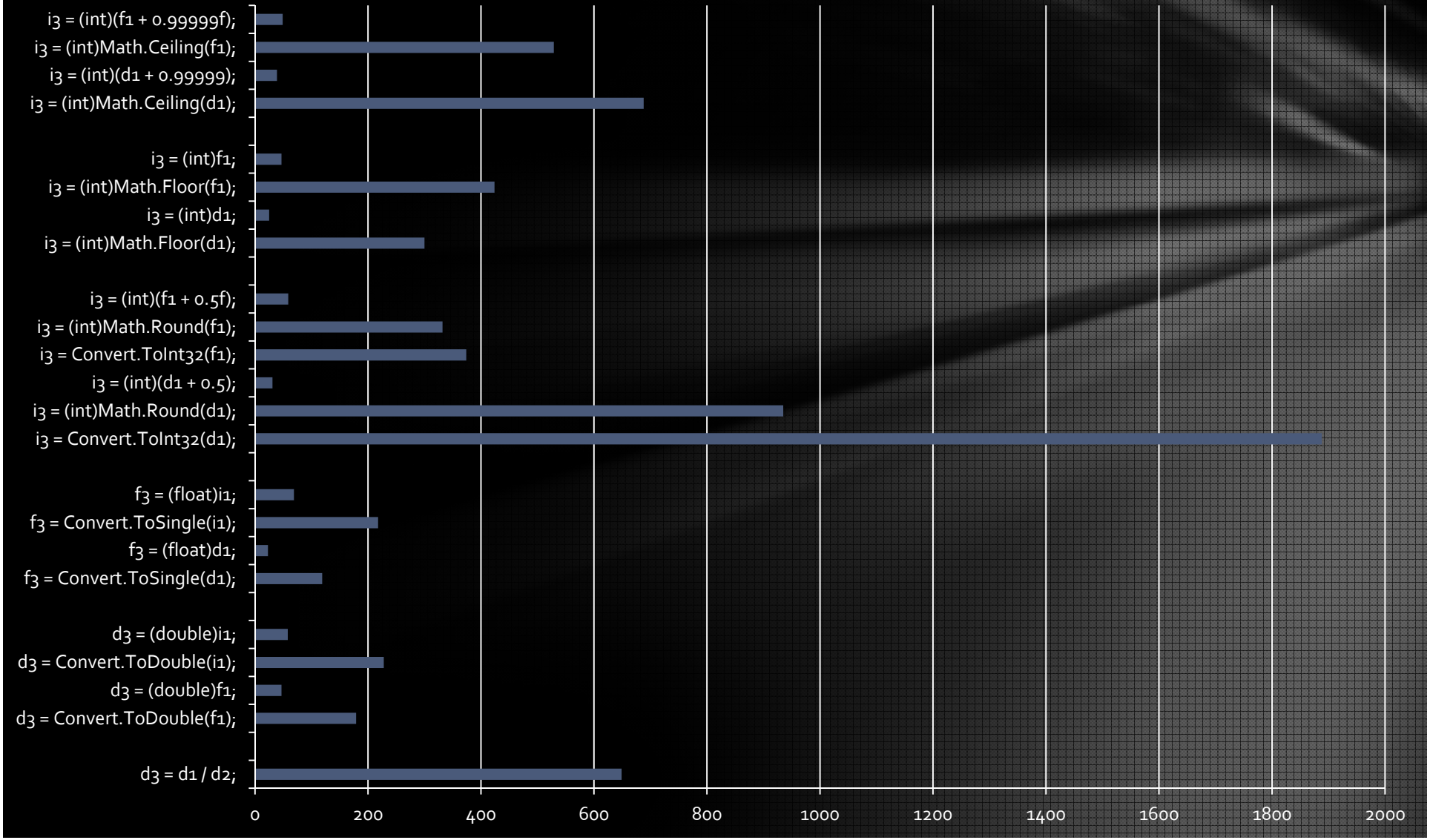
Trigonometric Math Functions



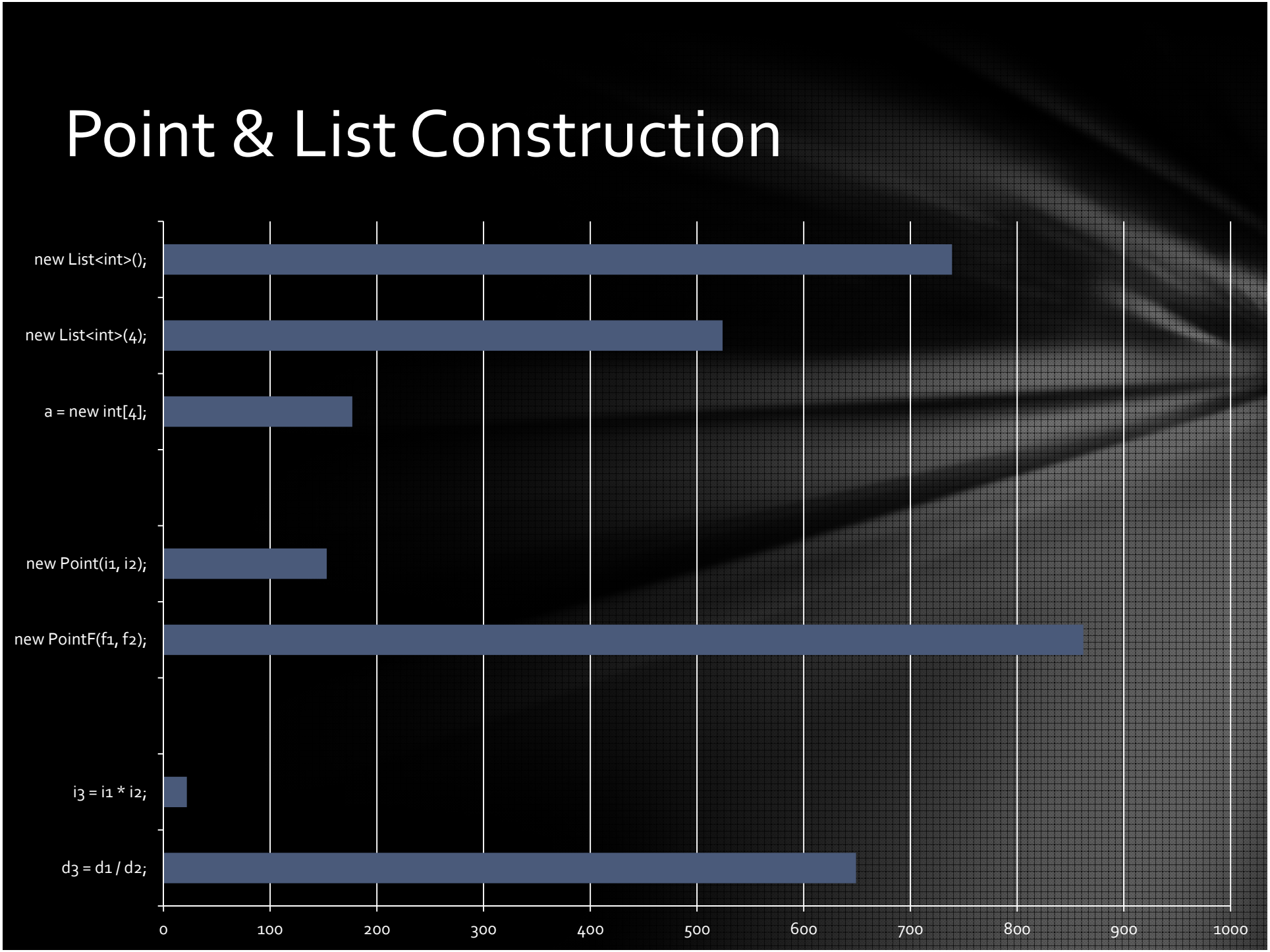
Power Functions



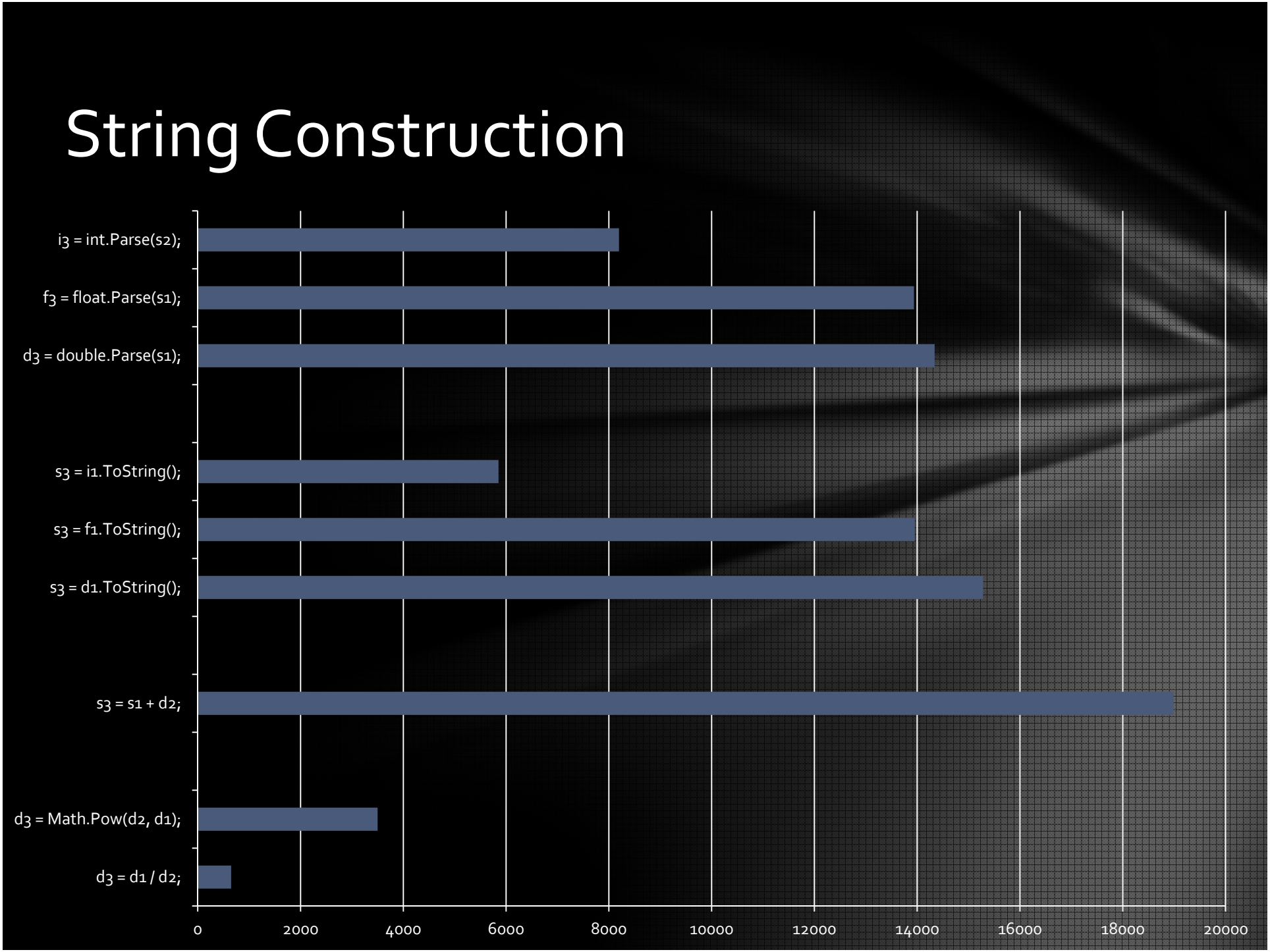
Type Conversion Functions



Point & List Construction



String Construction



CPU Optimization Tips

- Avoid Convert.To*(), , Math.Round(), Math.Ceiling() and Math.Floor()
  - Instead, use (int)(rVariable + C)
- Avoid Math.Pow()
  - Instead, consider using 1/r, r*r, Math.Sqrt(r), or a look-up table.
- Prefer float over double
- Avoid excessive type conversions
- Prefer separate component variables over Point*
- Avoid excessive string operations

Fast Inverse Square Root

Used by id Tech 3 game engine for lighting calculations

¼ time of built-in function

0.175% Error

Fast Inverse Square Root

```
float rsqrt(float x)
{
        float xhalf = 0.5f * x;
        int i = *(int*)&x;
        i = 0x5f3759df - (i >> 1);
        float y = *(float*)&i;
        y = y * (1.5f - xhalf * y * y);
        return y;
}
```

int i = *(int*)&x;

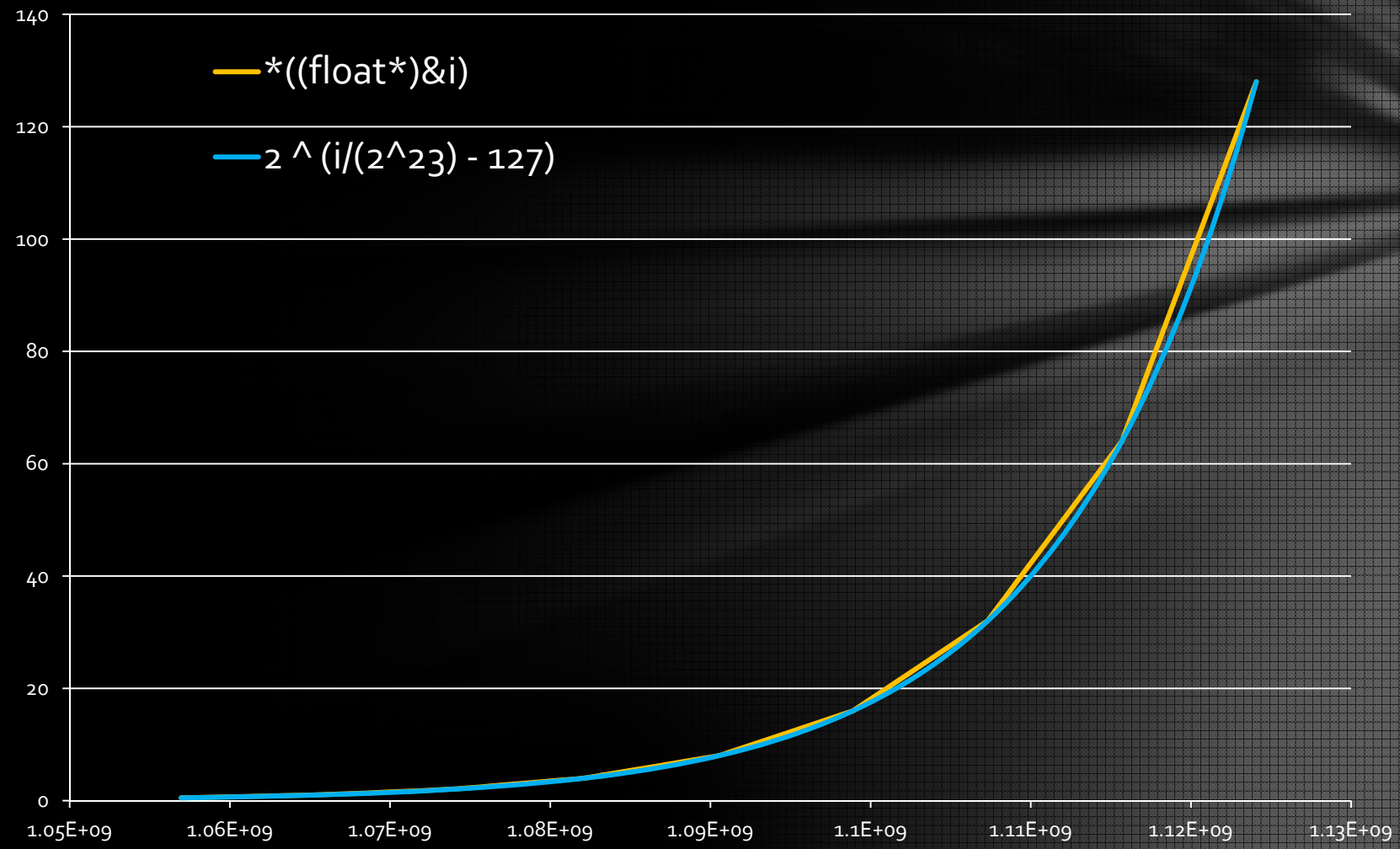
Float aliasing

float = sign * 1.significand * 2 ^ (exponent – 127)

If float is positive and finite:

- float f = *(float *)&i;
  - f ~= b ^ (i / a)
- int i = *(int *)&f;
  - i ~= a * log.b(f)

Float Alias


140
120
100
80
60
40
20
0
((float)&i)
2 ^ (i/(2^23) - 127)
1.05E+09
1.06E+09
1.07E+09
1.08E+09
1.09E+09
1.1E+09
1.11E+09
1.12E+09
1.13E+09

Fast Inverse Square Root

```
float rsqrt(float x)
{
        float xhalf = 0.5f * x;
        int i = *(int*)&x;                      // ~= a * log.b(x)
        i = 0x5f3759df - (i >> 1);
        float y = *(float*)&i;
        y = y * (1.5f - xhalf * y * y);
        return y;
}
```

i = 0x5f3759df - (i >> 1);

Have: mantissa * 2 ^ (exponent – 127)

Want: above ^ -1/2

- ~= mantissa * 2 ^ (63.5 - exponent / 2)

i = (i >> 1); // i = i / 2

- ~= (mantissa / 2) * 2 ^ (exponent/2 – 127)
- ~= mantissa * 2 ^ (exponent/2 - 128)

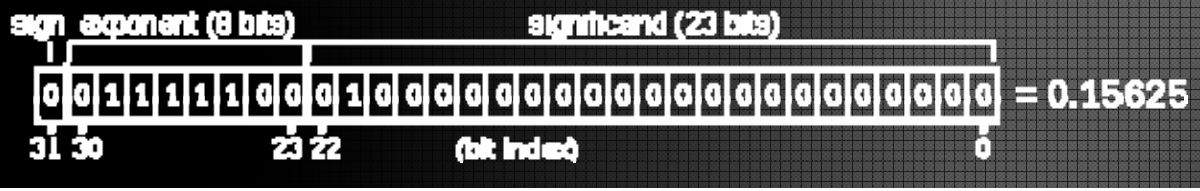
i = -(i >> 1); // i = -i / 2;

- ~= mantissa * 2 ^ (128 - exponent/2)

I = 0x5f800000 - (i >> 1); // In the ballpark

- *(float *)(0x5f800000) == 2 ^ 64
- ~= mantissa * 2 ^ (128 – (exponent/2 + 64))
- ~= mantissa * 2 ^ (64 – exponent/2 + 64)

Experimentally test magic numbers from 2^63 and 2^65 and select one with minimum error.



Fast Inverse Square Root

```
float rsqrt(float x)
{
        float xhalf = 0.5f * x;
        int i = *(int*)&x;                  // ~= a * log.b(x)
        i = 0x5f3759df - (i >> 1);          // ~= -1/2 * a * log.b(x)
        float y = *(float*)&i;
        y = y * (1.5f - xhalf * y * y);
        return y;
}
```

float y = *(float*)&i;

Float aliasing

float = sign * 1.significand * 2 ^ (exponent – 127)

If float is positive and finite:

- float f = *(float *)&i;
  - f ~= b ^ (i / a)
- int i = *(int *)&f;
  - i ~= a * log.b(f)

Fast Inverse Square Root

```
float rsqrt(float x)
{
        float xhalf = 0.5f * x;
        int i = *(int*)&x;                  // ~= a * log.b(x)
        i = 0x5f3759df - (i >> 1);          // ~= -1/2 * a * log.b(x)
        float y = *(float*)&i;              // ~= x ^ -1/2
        y = y * (1.5f - xhalf * y * y);
        return y;
}
```

y = y * (1.5f - xhalf * y * y);

Employs Newton's method to improve our accuracy

- Find x where f(x) = 0.
- x_0 = initial guess for x.
- x_{n+1} = improved guess for $x = x_n - f(x_n) / f'(x_n)$

$y = x^{-1/2} \rightarrow y^{-2} - x = 0$

Root: $f(y) = y^{-2} - x = 0$, x is actually constant

Derivative: $f'(y) = -2y^{-3}$

$y_1 = y_0 - (y_0^{-2} - x) / (-2y_0^{-3})$

...

$y_1 = y_0(1.5 - 0.5xy_0^2)$

Fast Inverse Square Root

```
float rsqrt(float x)
{
        float xhalf = 0.5f * x;
        int i = *(int*)&x;                  // ~= a * log.b(x)
        i = 0x5f3759df - (i >> 1);          // ~= -1/2 * a * log.b(x)
        float y = *(float*)&i;              // ~= x ^ -1/2
        y = y * (1.5f - xhalf * y * y);     // Newton's Method iteration
        return y;
}
```

Branching Optimizations

- If Statements
  - Order if-else-if statements from most frequently true to least.
- Loop unrolling
  - Useful in unmanaged code, not so much in .NET
- Method/delegate calls
  - For per-pixel operations, avoid calling methods

If-Statements

Consider this code snippet:

```
for (int i=0;i< a.Length; i++)  {
        bool justWonLottery = a[i] == 42;
        if (justWonLottery) {
                throw new PartyException();
        } else {
          a[i] += 1;
        }
}
```

If-Statements

Put most common condition first.

```
for (int i=0;i< a.Length; i++) {
        bool justWonLottery = a[i] == 42;
        if (!justWonLottery) {
            a[i] += 1;
        } else {
            throw new PartyException();
        }
}
```

If-Statements

Better yet, take the conditional branch outside the loop:

```
bool justWonLottery = false;
for (int i=0;i< a.Length; i++)  {
                justWonLottery |= a[i] == 42;
                a[i] += 1;
}
if (justWonLottery)  {
                throw new PartyException();
}
```

Loop Unrolling

Instead of:

```
for (int i=0;i<a.Length;i++) {
        a[i] *= 2;
}
```

Try this:

```
for (int i=0;i<a.Length;) {
        a[i++] *= 2;
        a[i++] *= 2;
        a[i++] *= 2;
        a[i++] *= 2;
}
```

Ensure the array length will always be divisible by 4

Memory Optimizations

Locality

- Ask the CPU for a byte of memory, it will get 64
- Access elements in arrays/lists sequentially
- Avoid linked lists and heaps

Allocation

- Recycle unused instances of slow-to-construct types
- Object Pool Design Pattern
- Avoid excessive string manipulation

CPU Memory Cache Hierarchy

Registers

- 16 x 64-bit
- 1 cycle

L1 Cache

- 64 KB / core, 64-byte lines
- 4 cycles

L2 Cache

- 256 KB / core, 64-byte lines
- 10 cycles

L3 Cache

- 8 MB shared by all cores
- 40-75 cycles

RAM

- 60-100 ns

HDD

- 10,000,000 ns

Exercise: Optimize this

```
for (int x = 0; x < width; x++) {
        for (int y = 0; y < height ; y++) {
                output[y][x] = input[y][x] * 2;
        }
}
```

Exercise: Optimize this

```
// x-loop put on inside, for better performance
for (int y = 0; y < height ; y++) {
        for (int x = 0; x < width; x++) {
                output[y][x] = input[y][x] * 2;
        }
}
```

Exercise

```
// row arrays cached in local variables.
for (int y = 0; y < height ; y++) {
        var outputRow = output[y];
        var inputRow = input[y];
        for (int x = 0; x < width; x++) {
                outputRow[x] = inputRow[x] * 2;
        }
}
```

Memory Tips

- Arrange nested loops to optimize sequential memory accesses
- Prefer arrays over linked lists

Disk I/O Optimizations

- Ask the disk for a byte, it will get a sector
  - 4KB - 64KB
- Strive for sequential reads
- Minimize # files to open

Exercise:

When a route generation is requested, Ground Guidance will load map tiles occupied by the route end-points and progressively load adjacent tiles as the search spaces expand. If the next tile to load is adjacent to the previously loaded tile, there is a 75% chance of the load requiring a random disk access.

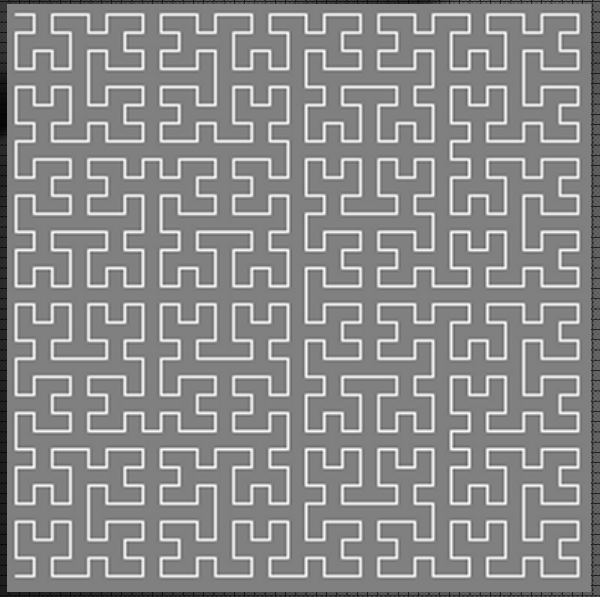
What are some ways to potentially improve this?

Exercise:

When a route generation is requested, Ground Guidance will load map tiles occupied by the route end-points and progressively load adjacent tiles as the search spaces expand. If the next tile to load is adjacent to the previously loaded tile, there is a 75% chance of the load requiring a random disk access.

What are some ways to potentially improve this?

- Batch multiple tiles into one load request
- Order tiles in GGT using a Hilbert Curve
  - Reduces jump distance
  - Doesn't reduce sequential miss rate



References

http://msdn.microsoft.com/en-us/library/ms973852.aspx

http://arstechnica.com/gadgets/2002/07/caching/3/

http://www.tomshardware.com/reviews/Intel-i7-nehalem-cpu,2041-10.html

http://www.gdcvault.com/play/1014645/-SPONSORED-Hotspots-FLOPS-and

http://en.wikipedia.org/wiki/Hilbert_curve

Questions?