Optimization & Vectorization

J. Bikker - Sep-Nov 2018 - Lecture 13: “Snippets”

Welcome!
Today's Agenda:

- Self-modifying code
- Blind stupidity
- Of square roots and sorting
- Madness
Fast Polygons on Limited Hardware

Typical span rendering code:

```c
for ( int i = 0; i < len; i++ )
{
    *a++ = texture[u,v];
    u += du;
    v += dv;
}
```

How do we make this faster?
Every cycle counts...

- **Loop unrolling**
- Two pixels at a time

Self-modifying
Self-modifying

Fast Polygons on Limited Hardware

How about...

```c
switch (len) {
  case 8: *a++ = tex[u,v]; u+=du; v+=dv;
  case 7: *a++ = tex[u,v]; u+=du; v+=dv;
  case 6: *a++ = tex[u,v]; u+=du; v+=dv;
  case 5: *a++ = tex[u,v]; u+=du; v+=dv;
  case 4: *a++ = tex[u,v]; u+=du; v+=dv;
  case 3: *a++ = tex[u,v]; u+=du; v+=dv;
  case 2: *a++ = tex[u,v]; u+=du; v+=dv;
  case 1: *a++ = tex[u,v]; u+=du; v+=dv;
}
```

INFOMOV – Lecture 13 – “Snippets”
Fast Polygons on Limited Hardware

What if a massive unroll isn’t an option, but we have only 4 registers?

```c
for( int i = 0; i < len; i++ )
{
  *a++ = texture[u,v];
  u += du, v += dv;
}
```

Registers: \{ i, a, u, v, du, dv, len \}.

Idea: just before entering the loop,
- replace ‘len’ by the correct constant \textit{in the code};
- replace du and dv by the correct constant.

Our code is now \textit{self-modifying}. 
Self-modifying

Self-modifying Code

Good reasons for **not** writing SMC:

- the CPU pipeline (mind every potential (future) target)
- L1 instruction cache (handles reads only)
- code readability

Good reasons for writing SMC:

- code readability
- genetic code optimization
Self-modifying

Hardware Evolution*

Experiment:

- take 100 FPGA's, load them with random 'programs', max 100 logic gates
- test each chip's ability to differentiate between two audio tones
- use the best candidates to produce the next generation.

Outcome (generation 4000): one chip capable of the intended task.

Observations:

1. The chip used only 37 logic gates, of which 5 disconnected from the rest.
2. The 5 disconnected gates were vital to the function of the chip.
3. The program could not be transferred to another chip.

**: Evolved antenna, Wikipedia.
Self-modifying

Compiler Flags*

Experiment:

“...we propose a **genetic algorithm** to determine the combination of flags, that could be used, to generate efficient executable in terms of time. The **input population** to the genetic algorithm is the **set of compiler flags** that can be used to compile a program and the best chromosome corresponding to the **best combination of flags** is **derived over generations**, based on the time taken to compile and execute, as the fitness function.”

Today's Agenda:

- Self-modifying code
- Blind stupidity
- Of square roots and sorting
- Madness
Application

DotCloud

Application breakdown, Tick:

t.reset(); Transform(); elapsed1 = t.elapsed();
t.reset(); Sort(); elapsed2 = t.elapsed();
t.reset(); Render(); elapsed3 = t.elapsed();

Transform:

for ( int i = 0; i < DOTS; i++ )
    m_Rotated[i] = m.Transform( m_Points[i] );

Sort:

for ( int i = 0; i < DOTS; i++ )
    for ( int j = 0; j < (DOTS - 1); j++ )
        if (m_Rotated[j].z > m_Rotated[j + 1].z) Swap( j, j + 1 );

Render:

for ( int i = 0; i < DOTS; i++ )
    m_Dot->DrawScaled( (sx - size / 2), (sy - size / 2), size, size, screen );
Application breakdown:

- Tick
- Sort
- Transform
- Render
- DrawScaled
Analysis

Performance Analysis & Scalability

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ms per frame</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transform</td>
<td>0.004</td>
<td>0.015</td>
<td>0.052</td>
<td>0.186</td>
</tr>
<tr>
<td>Sort</td>
<td>0.149</td>
<td>1.184</td>
<td>26.57</td>
<td>530.4</td>
</tr>
<tr>
<td>Render</td>
<td>0.519</td>
<td>1.226</td>
<td>4.351</td>
<td>17.10</td>
</tr>
<tr>
<td>Clear</td>
<td>0.131</td>
<td>0.161</td>
<td>0.127</td>
<td>0.130</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ms per dot</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transform</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Sort</td>
<td>0.0006</td>
<td>0.0012</td>
<td>0.0065</td>
<td>0.0324</td>
</tr>
<tr>
<td>Render</td>
<td>0.0020</td>
<td>0.0012</td>
<td>0.0011</td>
<td>0.0010</td>
</tr>
<tr>
<td>Clear</td>
<td>0.0005</td>
<td>0.0002</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
Solving the Sort Problem

Current Sort: bubblesort ($O(N^2)$).
Alternatives*:

<table>
<thead>
<tr>
<th>Quicksort</th>
<th>Shell sort</th>
<th>Pigeonhole sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsort</td>
<td>Binary tree sort</td>
<td>Bucket sort</td>
</tr>
<tr>
<td>Mergesort</td>
<td>Library sort</td>
<td>Spread sort</td>
</tr>
<tr>
<td>Radixsort</td>
<td>Smoothsort</td>
<td>Burstsort</td>
</tr>
<tr>
<td>Insertionsort</td>
<td>Strand sort</td>
<td>Flashsort</td>
</tr>
<tr>
<td>Selectionsort</td>
<td>Cocktail sort</td>
<td>Postman sort</td>
</tr>
<tr>
<td>Monkeysort</td>
<td>Comb sort</td>
<td>Bread sort</td>
</tr>
<tr>
<td>Countingsort</td>
<td>Block sort</td>
<td>Bitonic sort</td>
</tr>
<tr>
<td>Introsort</td>
<td>Odd-even sort</td>
<td>Stooge sort</td>
</tr>
</tbody>
</table>

* See e.g.: http://www.sorting-algorithms.com
Solving the Sort Problem

Current Sort: bubblesort ( $O(N^2)$ ).
Best case: $O(N)$.

Which case do we have here? Factors:

- Size of set
- Already sorted / almost sorted?
- Distributed (even / uneven)
- Type of data (just key / full records)
- Key type (float / int / string)
- ...

How much effort should we spend on this?

- For small sets, sorting takes far less time than rendering
- Anything that is not $O(N^2)$ will probably be fine.
- Would be nice if we can find something that fits well in the current code (save time for other optimizations).
Solving the Sort Problem

Current Sort: bubblesort (\(O(N^2)\)). Alternative: QuickSort (\(O(N \log N)\)).

```c
void Swap( float3& a, float3& b ) { float3 t = a; a = b; b = t; }

int Pivot( float3 a[], int first, int last )
{
    int p = first;
    float3 e = a[first];
    for( int i = first + 1; i <= last; i++ )
        if (a[i].z <= e.z) Swap( a[i], a[++p] );
    Swap( a[p], a[first] );
    return p;
}

void QuickSort( float3 a[], int first, int last )
{
    int pivotElement;
    if (first >= last) return;
    pivotElement = Pivot( a, first, last );
    QuickSort( a, first, pivotElement - 1 );
    QuickSort( a, pivotElement + 1, last );
}
```
### Repeated Profiling

<table>
<thead>
<tr>
<th></th>
<th>bubblesort</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform</td>
<td>0.004</td>
<td>0.015</td>
<td>0.052</td>
<td>0.186</td>
<td></td>
</tr>
<tr>
<td>Sort (bubble)</td>
<td>0.149</td>
<td>1.184</td>
<td>26.57</td>
<td>530.4</td>
<td></td>
</tr>
<tr>
<td>Sort (quick)</td>
<td>0.013</td>
<td>0.058</td>
<td>0.319</td>
<td>0.955</td>
<td></td>
</tr>
<tr>
<td>Render</td>
<td>0.519</td>
<td>1.226</td>
<td>4.351</td>
<td>17.10</td>
<td></td>
</tr>
<tr>
<td>Clear</td>
<td>0.131</td>
<td>0.161</td>
<td>0.127</td>
<td>0.130</td>
<td></td>
</tr>
</tbody>
</table>

Note: Clear is implemented using a loop; a `memset` is faster for clearing to zero (~0.43).
Repeated Profiling
Low-level Optimization of DrawScaled

```c
void Sprite::DrawScaled( float a_X, float a_Y, float a_Width, float a_Height, Surface* a_Target )
{
    Pixel* dest = a_Target->GetBuffer() + (int)a_X + (int)a_Y * a_Target->GetPitch();
    Pixel* src = GetBuffer() + m_CurrentFrame * m_Width;
    for ( int y = 0; y < (int)a_Height; y++ ) for ( int x = 0; x < (int)a_Width; x++ )
    {
        int v = (int)((y * m_Height) / a_Height);
        int u = (int)((x * m_Pitch) / a_Width);
        if (src[u + v * m_Pitch] & 0xffffff) *(dest + x + y * a_Target->GetWidth()) = src[u + v * m_Pitch];
    }
}
```

Functionality:
- for every pixel of the rectangular target image,
- find the corresponding source pixel,
- using interpolation,
- plot if it’s not black.
Low Level Optimization of DrawScaled

A few basic optimizations:

```c
void Sprite::DrawScaled( float a_X, float a_Y, float a_Width, float a_Height, Surface* a_Target )
{
    Pixel* dest = a_Target->GetBuffer() + (int)a_X + (int)a_Y * a_Target->GetPitch();
    Pixel* src = GetBuffer() + m_CurrentFrame * m_Width;
    for ( int y = 0; y < (int)a_Height; y++ )
    {
        int v = (int)((y * m_Height) / a_Height);
        for ( int x = 0; x < (int)a_Width; x++ )
        {
            int u = (int)((x * m_Pitch) / a_Width);
            Pixel color = src[u + v * m_Pitch] & 0xffffff;
            if (color) *(dest + x + y * a_Target->GetWidth()) = color;
        }
    }
}
```

- Loop hoisting (variable v is constant inside x loop)
- Reading source pixel only once
Low-level Optimization of DrawScaled

More basic optimizations:

```c
void Sprite::DrawScaled( float a_X, float a_Y, float a_Width, float a_Height, Surface* a_Target )
{
    Pixel* dest = a_Target->GetBuffer() + (int)a_X + (int)a_Y * a_Target->GetPitch();
    Pixel* src = GetBuffer() + m_CurrentFrame * m_Width;
    float rw = (float)m_Width / a_Width;
    float rh = (float)m_Height / a_Height;
    int iw = (int)a_Width, ih = (int)a_Height;
    for ( int y = 0; y < ih; y++ )
    {
        int v = (int)(y * rh);
        Pixel* line = dest + y * a_Target->GetWidth();
        for ( int x = 0; x < iw; x++ )
        {
            int u = (int)(x * rw);
            Pixel color = src[u + v * m_Width] & 0xffffff;
            if (color) line[x] = color;
        }
    }
}
```

- Precalculate m_Height / a_Height, m_Width / a_Width
- Calculate target address once per line; index using x
Low Level Optimization of DrawScaled

Fixed point optimization:

```c++
void Sprite::DrawScaled( int a_X, int a_Y, int a_Width, int a_Height, Surface* a_Target )
{
    const int rh = (m_Height << 10) / a_Height,
    rw = (m_Width << 10) / a_Width;
    Pixel* line = a_Target->GetBuffer() + a_X + a_Y * a_Target->GetPitch();
    for ( int y = 0; y < a_Height; y++, line += a_Target->GetPitch() )
    {
        const int v = (y * rh) >> 10;
        for ( int x = 0; x < a_Width; x++ )
        {
            const int u = (x * rw) >> 10;
            const Pixel color = GetBuffer()[u + v * m_Pitch];
            if (color & 0xffffff) line[x] = color;
        }
    }
}
```

- Fixed point works really well here... but doesn't improve performance.
- Seems we reached the end here...
Blind Stupidity

Low Level Optimization of DrawScaled

Now what?

- Plot multiple pixels at a time?
- ...

How many different ball sizes do we encounter?

...Why don’t we simply precalculate those frames?
“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.” (W.A. Wulff)
High Level Optimization of DrawScaled

```cpp
Sprite* scaled[64];
void Game::Init()
{
    ...
    for( int i = 0; i < 64; i++ )
    {
        int size = i + 1;
        scaled[i] = new Sprite( new Surface( size, size ), 1 );
        scaled[i]->GetSurface()->Clear( 0 );
        m_Dot->DrawScaled( 0, 0, size, size, scaled[i]->GetSurface() );
    }

    scaled[size]->Draw( (sx - size / 2), (sy - size / 2), m_Surface );
}
```
Profile

Repeated Profiling

<table>
<thead>
<tr>
<th>bubblesort</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform</td>
<td>0.004</td>
<td>0.015</td>
<td>0.052</td>
<td>0.186</td>
</tr>
<tr>
<td>Sort</td>
<td>0.013</td>
<td>0.058</td>
<td>0.319</td>
<td>0.955</td>
</tr>
<tr>
<td>Render (old)</td>
<td>0.519</td>
<td>1.226</td>
<td>4.351</td>
<td>17.10</td>
</tr>
<tr>
<td>Render (new)</td>
<td>0.281</td>
<td>0.591</td>
<td>1.746</td>
<td>5.520</td>
</tr>
</tbody>
</table>
Optimization of Dense Clouds

Observation: beyond a certain dot count, a large number of particles is occluded.

Specifically, we won't be able to see the back half.

```cpp
if (m_Rotated[i].z > -0.2f)
    scaled[size]->Draw( (sx - size / 2), (sy - size / 2), screen );
```

(perhaps we could also limit rendering to the outer shell of the cloud?)

Rendering is now significantly faster, and sorting is significant again:

At 65536 dots, we get 5ms for sorting, 17ms for rendering.
High-level

Sorting in $O(1)$

For this specific situation, we can sort in $O(1)$, e.g., independent of particle count.

Observation: dots do not move independently.

Intuition: why rotate 64k dots if you can rotate a single camera?
High-level

Sorting in O(1)
High-level

Sorting in $O(1)$
High-level

Sorting in O(1)
High-level

Sorting in $O(1)$
High-level

Sorting in $O(1)$

```
// some code
```

```
// more code
```
High-level

Sorting in $O(1)$

For each split:
- Process nearest half first
- Then farthest half
- Recurse

Where ‘nearest’ is the side that the ‘camera’ is on.
High-level

Optimizing Sphere Cloud - TakeAway

- Profile
- High level: bang for the buck
- Mind ‘blind stupidity’
- The fastest solution is sometimes not doing anything at all
Today's Agenda:

- Self-modifying code
- Blind stupidity
- Of square roots and sorting
- Madness
Accurate reciprocal square roots

1.0f / sqrtf( 7 ):

\[ \frac{1.0}{\sqrt{7}} : 0.377964497 \]

1.0 / sqrt( 7 ):

\[ \frac{1.0}{\sqrt{7}} : 0.37796447300922720 \]

_mm_rsqrt_ps( _mm_set1_ps( 7.0f ) ):

\[ 0.377929688 \]

Newton iteration:

\[ y_{n+1} = \frac{y_n(3-xy_n^2)}{2} \]

In SSE:

\[
\begin{align*}
\_m128 \ y &= \_mm_rsqrt_ps( \ x ); \\
\_m128 \ m &= \_mm_mul_ps( \_mm_mul_ps( \ x, \ y ), \ y ); \\
\quad & return \_mm_mul_ps( \_mm_mul_ps( \ half, \ y ), \_mm_sub_ps( \ three, \ m ) );
\end{align*}
\]

\[ 0.377964467 \]
Sorting four key/item pairs

Sorting three numbers a, b, and c:

if (a > b) swap(a, b);
if (b > c) swap(b, c);
if (a > b) swap(a, b);

Sorting four numbers a, b, c, and d:

if (a > c) swap(a, c); if (b > d) swap(b, d);
if (a > b) swap(a, b); if (c > d) swap(c, d);
if (b > c) swap(b, c);

```c
void sse_sort(__m128& v0) {
    __m128 v1, v2, v3, t;
    v0 = _mm_or_ps(_mm_and_ps(v0, idxmask4), idxadd4);
    v1 = _mm_movelh_ps(v1, v0), t = v0;
    v0 = _mm_min_ps(v0, v1), v1 = _mm_max_ps(v1, t);
    v0 = _mm_movelh_ps(v0, v1), t = v0;
    v0 = _mm_min_ps(v0, v1), v1 = _mm_max_ps(v1, t);
    v0 = _mm_movelh_ps(v0, v1), t = v0;
    v0 = _mm_min_ps(v0, v1), v1 = _mm_max_ps(v1, t);
    v0 = _mm_shuffle_ps(_mm_shuffle_ps(v0, v1), 0x0888), t = v0;
    v0 = _mm_min_ps(v0, v1), v1 = _mm_max_ps(v1, t);
}
```

```c
idxmask4 = _mm_set1_ps(0xffffffffc);
idxadd4 = _mm_set_ps(0, 1, 2, 3);
```
Today's Agenda:

- Self-modifying code
- Blind stupidity
- Of square roots and sorting
- Madness
Low Level Optimization of Sprite::Draw

Pre-scaled sprites are faster than on-the-fly scaling.

But, we still have loops, and if-statements, and look-ups. I wonder...
Low Level Optimization of Sprite::Draw

Extreme Optimization:

- We simply generate a function that plots every pixel, without the need for a loop.

Side effect:

L1 data cache is now used for the screen buffer; L1 instruction cache is used for the sprite data.

```cpp
void Sprite::DrawBall( int x, int y, int size, Surface* target )
{
    uint* a = target->GetBuffer() + x + y * SCRWIDTH;
    switch( size )
    {
    case 1:
        break;
    case 2:
         a[1]=1052688;
         a[800]=1052688;
         a[801]=15724527;
        break;
    case 3:
        a[801]=9737364;
        a[802]=8684676;
        a[1601]=8684676;
        a[1602]=8092539;
        break;
    case 4:
        a[2]=1052688;
        a[801]=6513507;
        a[802]=9737364;
        a[803]=4868682;
        a[1600]=1052688;
        a[1601]=9737364;
        a[1602]=15724527;
        a[1603]=7566195;
        a[2401]=4868682;
        a[2402]=7566195;
        a[2403]=3223857;
        break;
    ...
    
    // Further code...
}
```
Low Level Optimization of Sprite::Draw

Extreme Optimization:

- We simply generate a function that plots every pixel, without the need for a loop.

```cpp
FILE* f = fopen( "drawfunc.h", "w" );
fprintf( f, "void Sprite::DrawBall( int x, int y, int size, Surface* target )\n" );
fprintf( f, "\{\nuint* a = target->GetBuffer() + x + y * SCRWIDTH;\nswitch( size )\n\{
    ... 
    case %i: a = target->GetBuffer()[x + y * size] & 0xffffff)
    fprintf( f, "a[%i]=%i;\n", a, scaled[i] - target->GetBuffer()[x + y * size] & 0xffffff );
for( int i = 0; i < 64; i++ )
{ 
    ...
    fprintf( f, "case %i:\n", size );
    for( int y = 0; y < size; y++ ) for( int x = 0; x < size; x++ )
    {
        int a = y * SCRWIDTH + x;
        if (scaled[i] - target->GetBuffer()[x + y * size] & 0xffffff)
            fprintf( f, "a[%i]=%i;\n", a, scaled[i] - target->GetBuffer()[x + y * size] & 0xffffff );
    }
}
}```
"Madness!? This is INFOMOV'18!"
/INFOMOV/

END of “Snippets”

next lecture: “Grand Recap”