Practical Low-level Optimization

Handout for practicum 2 of INFOMOV
- newly created for the 2019/2020 edition.

Introduction

In this tutorial we apply low-level optimization to a simple graphical application. The application shows a spinning orb consisting of coloured particles. The particles are depth-sorted and scale with distance to create perspective. The code runs purely on the CPU, on a single thread. Our aim is to increase the number of particles in the orb while maintaining a frame rate of 60Hz.

Profiling

The application consists of three distinct modules:

- Transform, which applies a rotation matrix to a set of 3D coordinates.
- Sort, which sorts the rotated 3D points by z-coordinate.
- Render, which draws the particles.

In Game::Tick these (and the screen clear) are timed individually using a high resolution timer. The unmodified application shows these timings on my desktop:

```
TRANSFORM:   0.0100
SORT:        0.2090
RENDER:      2.6740
CLEAR:       2.0630
```

Times are in milliseconds, and in Debug mode. In Release mode we get:

```
TRANSFORM:   0.0020
SORT:        0.0770
RENDER:      0.4000
CLEAR:       0.2130
```

Based on these timings our first optimization target should be the render code. However, our goal was to increase the particle count, so let's see what we get with ten times as many particles:

```
TRANSFORM:   0.0250
SORT:        3.5440
RENDER:      1.5280
CLEAR:       0.2210
```

Clearly the first assumption was wrong: for any decent number of particles, our first target should be the sort code.

Optimized Sorting

A quick inspection of the sorting code reveals that the current implementation uses Bubblesort. This is the best choice for a set of 2 or 3 numbers, but for 1280 particles its O(N^2) complexity is pretty terrible. Luckily, there are many alternatives. But which one should we pick?

There are many factors that determine which sorting algorithm is optimal for a particular task:

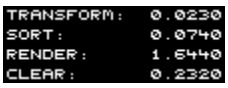
- The size of the set we are sorting.
- The data we are trying to sort: just numbers or records, evenly spaced or concentrated in certain ranges, already mostly sorted or completely random.
- The implementation effort: if we can paste a brief sorting code snippet, we save time for other optimizations.

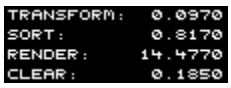
In this case we are sorting vectors (three floats: x, y, z), based on a single key (z). The depths are uniformly distributed in a specific range. The numbers are not sorted at all when we enter the sort function.

The choice is quickly made when we look at the implementation effort. A basic Quicksort is simple to add, and much better than Bubblesort. So let's start with that:

```
01  void Swap( vec3& a, vec3& b ) { vec3 t = a; a = b; b = t; }
02  int Pivot( vec3 a[], int first, int last )
03  {
04     int p = first;
05     vec3 e = a[first];
06     for (int i = first + 1; i <= last; i++) if (a[i].z <= e.z) Swap( a[i], a[++p] );
07     Swap( a[p], a[first] );
08     return p;
09  }
10  void QuickSort( vec3 a[], int first, int last )
11  {
12     int pivotElement;
13     if (first >= last) return;
14     pivotElement = Pivot( a, first, last );
15     QuickSort( a, first, pivotElement - 1 );
16     QuickSort( a, pivotElement + 1, last );
17  }
```

The replaced sorting code yields the following performance figures for 1280 and 12800 particles:





The sorting now scales almost linearly, and rendering is clearly the bottleneck.

Optimized Rendering

The rendering code is pretty brief. It calculates the 2D screen position of a particle, selects the right sprite image, and calls Sprite::DrawScaled. The problem must lie in DrawScaled:

```
01  void Sprite::DrawScaled(int a_X,int a_Y,int a_Width,int a_Height,Surface* a_Target)
02  {
03     Pixel* src = GetBuffer() + m_CurrentFrame * m_Width;
04     for ( int x = 0; x < a_Width; x++ ) for ( int y = 0; y < a_Height; y++ )
05     {
06        int u = (int)((float)x * ((float)m_Width / (float)a_Width));
07        int v = (int)((float)y * ((float)m_Height / (float)a_Height));
08        Pixel color = src[u + v * m_Pitch];
09        if (color & 0xffffff) a_Target->GetBuffer()[…] = color;
10     }
11  }
```

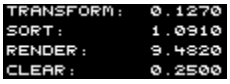
Now that is a function that we can apply the Rules of Engagement to (see OptmzdSummary #0). Several problems are visible right away:

- The expensive divisions always yield the same answer and should be removed.
- Type conversions are abundant, but not as easy to get rid of.
- The calculation of `int u` does not depend on y and is thus constant in the inner loop.

Let's make a few initial improvements to see where we can get:

```
01  Pixel* src = GetBuffer() + m_CurrentFrame * m_Width;
02  float v0 = (float)m_Height / a_Height, v1 = (float)m_Width / a_Width;
03  for( int y = 0; y < a_Height; y++) for( int v = (float)y*v0, x = 0; x < a_Width; x++)
04  {
05     int u = (int)((float)x * v1);
06     Pixel color = src[u + v * m_Pitch];
07     if (color & 0xffffff) a_Target->GetBuffer()[…] = color;
08  }
```

Well it works, and it got faster, by quite a bit in fact:

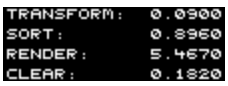


Also, this is *fun* right? We should keep doing this until we squeezed out the last cycle! But wait… Let's take a step back and think about what we are doing. We are drawing scaled sprites. Their size is specified as an integer width and height, and in our application, the aspect ratio is constant. How many sizes are there? Not many. So, why don't we *precalculate all scaled sprites?*

This illustrates a danger of low-level optimization: we may be optimizing something that should have been replaced by something more clever. The time we just spent is wasted when we replace the scaling functionality by something better.

Precalculating Scaling

Once we precalculate all the scaled particles (I'll leave the specifics as an exercise for the reader) we get a performance level that is impossible to achieve with an optimized scaling function:



A few things are clear:

- Rendering remains a bottleneck, so it was a good call not to spend too much time on sorting.
- The screen is pretty full now, so adding more particles is useless. We are done.

But still… If we had an excuse to continue, e.g. to fill a 4k screen, what more could we do?

Let's explore some options on the last page.

Student & Future Work

We have used this assignment (in a slightly different form) in several courses over the past years, and students have come up with all kinds of clever ideas to make the code faster. How about these:

- If we use a z-buffer, we don't have to sort. This could be faster.
- Once the screen is rather full, drawing the backside of the sphere doesn't change any pixels.
- In fact, we can skip the interior of the sphere and just draw the hull. The extra speed can be used to add more particles, which leads to a thinner hull, and so on until we reach infinite particles.

At some point, sorting becomes a bottleneck again. When that happens, Radix sort appears to be a fast alternative for Quicksort. But we can also render the scene *without any sorting.* How this can be achieved is a secret.

And then there is one more option. There is a way to significantly improve upon the raw rendering speed of the precalculated scales, without tricks like a z-buffer or not drawing things that do not change pixels. The method is so horrible though that I won't write it down here.

The End

Questions and comments:

bikker.j@gmail.com or room 4.24.

INFOMOV 2019