Real-time software rendering on x86

6.2.2017

Jari Korkala
jv.korkala@gmail.com

1. Introduction

I originally wanted to learn 3d graphics because I wanted to make a space combat simulator game similar to I-War 2. Back in 2002 internet wasn't the answer machine it is today, so studying physics and mathematics at university level seemed the only way to go.

Somewhere along the way, I got increasingly interested on the low level functionality of processors and GPUs. After reading ”Ramblings in Quake time” by Michael Abrash in 2009, my attention turned to software rendering. It also had that prestigious feeling to it: if you got it to work at playable framerates, you would undeniably be a guru. Never mind that by those days GPUs could outperform software by a factor of 100, and thus the whole endeavor would be pointless besides bragging rights.

I first made a renderer that mimicked Quake 1 engine in multiple aspects: map stored as a BSP-tree, front-to-back rendering of the map with span rasterization to eliminate overdraw. I did, however, use modern SIMD instruction sets where able. This renderer had the obvious problem of being Quake 1 -like, whereas Direct3D 9 and OpenGL already had shaders, multiple textures and customizable vertex formats. By late 2010, I set out to add those features with the goal of using as much parallelism as possible.

2. Rendering pipeline

2.1 Vertex shader

Vertex shader is the first pipeline stage. Its primary purpose is to transform vertex positions into homogeneous space for clipping. The position of a vertex is given by

$$\bar{x}' = PR_c T_c T_o R_o \bar{x} \quad ,$$

where the projection matrix P is given by

$$P = \begin{bmatrix} D & 0 & 0 & 0 \\ 0 & Da & 0 & 0 \\ 0 & 0 & \frac{-z_{far}}{z_{far} - z_{near}} & -z_{near}\frac{z_{far}}{z_{far} - z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix} ,$$

where $D = 1/\tan(\alpha/2)$,α = horizontal field of view, $a = w/h$ is the aspect ratio, and z_{near}, z_{far} define the near and far planes in the depth direction.

The rotation matrices R_c, R_o are for the camera and object, respectively, and describe an arbitrary rotation in 3d space. Since quaternions

$$q = w + \hat{\boldsymbol{i}}\, x + \hat{\boldsymbol{j}}\, y + \hat{\boldsymbol{k}}\, z \quad ,$$

are used to store the rotation of objects, it is practical to use the axis-angle rotation matrix defined in terms of the quaternion:

$$R=\begin{bmatrix} 2x^2+k & 2xy-mz & 2xz+my & 0 \\ 2xy+mz & 2y^2+k & 2zy-mx & 0 \\ 2xz-my & 2yz+mx & 2z^2+k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $k=w^2-x^2-y^2-z^2$ and $m=2w$.

The translation matrices T_c, T_o are simple:

$$T=\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The position vectors these matrices operate on are of the form

$$\bar{x}=\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

The other purpose of the vertex shader is to forward texture coordinates and other per-vertex parameters to the pixel shader.

2.2 Clipper

The clipper ensures that only the visible parts of a triangle are rasterized. The clipping conditions are w>ε, where ε is a small non-zero value; -1 < x < 1; -1 < y < 1; z > 0 (Direct3D) or -1 < z < 1 (OpenGL). Original triangle is split into multiple triangles if neccessary.

2.3 Rasterization

For triangle rasterization, I chose the half-space rasterization algorithm, as described here: http://forum.devmaster.net/t/advanced-rasterization/6145. The algorithm also has the advantage of being screen-space subdivisible, doesn't require division instructions, and only uses fixed point math after initialization. This makes multi-threading simple to implement, and the relatively cheap setup cost helps with small triangles which are dominant.

The half-space rasterizer differs from the scanline algorithm in that it resolves triangle coverage per pixel instead of per scanline. This can be generalized to test if an arbitrary square is completely inside or outside the triangle.

2.3.1 Triangle setup

The first step is to convert a triangle given as three vertices into the rasterization data structures. Clip space coordinates are transformed to display coordinates. Vertex positions are converted to 24.8 fixed point format.

The divisor for parameter interpolation steps is calculated, and used to do backface culling if enabled. Since half-space rasterization required a counter-clockwise vertex order, this is corrected if the backface is visible, or if input data is in clockwise order. Scissor test is done if enabled. For multi-threading purposes, the triangle is added to those screen buckets it overlaps.

Half-space values for top-left corner of the triangle's bounding box are generated. Values used to trivially accept or reject a tile are generated. Values used to scan a tile are generated.

Finally, perspective correct interpolation parameters (p/w) are generated, where w is the true depth (z is mapped to [0,1] range and used for depth buffer tests), and p is a vertex parameters such as texture coordinate or color. Since triangles are planar, it is possible to use a bilinear equation to calculate p/w at any screen coordinate relative to a position with known values (ie. vertex position). It is more efficient to move the base to screen space origin (0,0) because this simplifies the equation.

2.3.2 Coverage test

A 8x8 tile is used for this coarse testing. For a partially covered tile, each pixel is tested individually. SIMD instructions can test four pixels at a time. For purposes that become apparent when describing the pixel shader, it is beneficial to use a 2x2 pixel quad as the test pattern here. Finally, a 64-bit coverage mask is produced for the entire tile.

2.3.3 Depth-stencil test

This test is done after the coverage test. SIMD instructions can be used to provide an efficient branch-free implementation. The number of possible configurations is staggering, however, and the only way to provide all of them in branch-free form is to dynamically patch together suitable pieces of machine code.

Since the coverage test is done for a 8x8 pixel tile in 2x2 quads, it makes sense for the depth and stencil buffers to use this layout whenever possible.

The test results are combined with the coverage mask before performing stencil actions. After this, we know which pixels are actually visible and need to be shaded.

2.3.4 Layout transformation

In order to feed the pixel shader in an optimal manner, the coverage masks of 8x8 tiles are converted into horizontal spans of 2x2 quads. Spans that have only fully visible quads are run-length compressed, whereas spans with partially visible quads come with a bitmask to determine which pixels to shade.

2.4 Pixel shader

The pixel shader operates on a horizontal span of quads described in the previous section. A z-order curve is used to walk through the span. This way, only the parameters at the top-left pixel of the first quad have to be evaluated using the full bilinear step equation. Afterwards only two step variables

$$a = \frac{\Delta\left(\frac{p}{w}\right)}{\Delta x} \quad ,$$

$$b = -\frac{\Delta\left(\frac{p}{w}\right)}{\Delta x} + \frac{\Delta\left(\frac{p}{w}\right)}{\Delta y} \quad ,$$

are required to walk the entire span by incrementing by a,b,a,-b.

Before pixel shader function, each parameter is divided by the interpolated 1/w for that pixel. Fortunately SIMD division does this for four pixels at the price of one, and it has to be done only once because afterwards multiplication can be used. It is also possible to use the approximated 1/x instruction, which provides adequate quality at all but closest inspection.

2.4.1 Shader function

Because data is processed as 2x2 pixel quads that fit nicely into the 4 floats per SIMD register, vertical data layout is used in the shader function. Ie. There is a 4-component vector with the blue color for each pixel, another for the green and so on, instead of an RGBA vector per pixel.

The vertical layout is significantly faster for the following common shader operations: dot product, cross product, swizzles, per-component operations. It also has no wasted SIMD register slots when a 3-component vector is used. Branch predictor performance is also improved by doing four pixels per iteration.

A C++ abstraction is currently used to write shader code. It adds some overhead in form of virtual function calls, but was more practical for development than first developing a shader language compiler and assembler.

2.4.2 Texture sampling

Texture sampling is the most complex action done during pixel shading. It is also the only piece of code that can crash the application if address calculations are out of bounds. Currently nearest neighbor and linear interpolation are supported, with mipmaps enabled or disabled.

2.4.2.1 Mipmap level calculation

One of the advantages of working on 2x2 pixel quads is the possibility to evaluate derivatives in screen space. This in turn makes it possible to calculate mip-map level per quad, since the mipmap level is given by

$$\rho = max\left(\sqrt{\left(\frac{\Delta u}{\Delta x}\right)^2 + \left(\frac{\Delta v}{\Delta x}\right)^2}, \sqrt{\left(\frac{\Delta u}{\Delta y}\right)^2 + \left(\frac{\Delta v}{\Delta y}\right)^2}\right) \quad ,$$

$$\lambda = \log_2(\rho) \quad ,$$

where λ is the mipmap level. The properties of logarithm allow this to be modified to

$$\lambda = \frac{1}{2}\log_2 max\left(\left(\frac{\Delta u}{\Delta x}\right)^2 + \left(\frac{\Delta v}{\Delta x}\right)^2, \left(\frac{\Delta u}{\Delta y}\right)^2 + \left(\frac{\Delta v}{\Delta y}\right)^2\right) \equiv \frac{1}{2}\log_2 \hat{\rho}$$

The properties of IEEE floating point numbers can be used to quickly extract the integer part of the logarithm. The result of the max-function is a non-negative integer, so we can extract the exponent of the floating point number by interpreting it as an integer:

$$\lambda = \left(\left(\hat{\rho} \gg 23\right) - 127\right) \gg 1 \quad ,$$

where >> is the logical right shift operator. This is adequate for nearest mipmaps, but trilinear filtering requires an interpolation scheme between λ and λ+1 based on mantissa bits.

2.4.2.2 Texture filtering

Valid texture coordinates for addressing are in the [0,1] range. It is, however possible to use different values as inputs, when, for example, tiling is used. The first step is thus to extract the fractional part

$$u_{frac} = u - \lfloor u \rfloor \quad ,$$

where $\lfloor x \rfloor$ is the floor function that returns the nearest integer smaller than its argument. Note that for negative integers the result is towards $-\infty$ instead of zero.

The result is needed in fixed point, so it's preferable to convert the coordinate into 16.16 fixed point, after which extracting the fractional bits can be done with a bitwise AND. The coordinates are then scaled by texture dimensions

$$\hat{u} = w\hat{u}_{frac} \quad ,$$

$$\hat{v} = h\hat{v}_{frac} \quad ,$$

where quantities in hats are 16.16 fixed point numbers. The nearest neighbor fetch coordinates are thus

$$u_{\text{int}} = \lfloor \hat{u} \rfloor \quad ,$$

$$v_{\text{int}} = \lfloor \hat{v} \rfloor \quad .$$

In the case of linear interpolation we also fetch $\left(u_{\text{int}}+1, v_{\text{int}}\right), \left(u_{\text{int}}, v_{\text{int}}+1\right), \left(u_{\text{int}}+1, v_{\text{int}}+1\right)$. Depending on overflow mode, these are then either clamped or wrapped around.

The colors read from these locations are c_{ij} , where i=j=0 is the nearest neighbor texel, and i=j=1 is the bottom-right texel. The interpolation weights are

$$w_{00} = 1 + \hat{u}_{frac}\hat{v}_{frac} - \hat{u}_{frac} - \hat{v}_{frac} \quad ,$$

$$w_{10} = \hat{u}_{frac} - \hat{u}_{frac}\hat{v}_{frac} \quad ,$$

$$w_{01} = \hat{v}_{frac} - \hat{u}_{frac}\hat{v}_{frac} \quad ,$$

$$w_{11} = \hat{u}_{frac}\hat{v}_{frac} \quad ,$$

and the final result is given by

$$\bar{c}' = \sum_{ij} w_{ij} \bar{c}_{ij} \quad .$$

If the input texture is 8 bits per channel, the weights can be stored as 0.16 fixed point numbers, and the PMULHUW instruction, shifts and adds can be used to quickly produce the result.

2.4.2.3 Address math

The memory address of a texel is given by

$$\text{address} = \text{base} + b(yw + x) \quad ,$$

where base = address of the (0,0) texel, b = bytes per texel, w = width of the texture. This can be quickly calculated for four fetches using PMADDWD instruction and then adding the base address. In 64-bit mode things are a bit more complicated since PMADDWD produces four 32-bit signed immediate results, which must be expanded to 64 bits before adding the base address.

Simplifications can be made if the texture is known to have power-of-two dimensions. In this case a combination of shuffles, shifts and adds produces the result much faster.

2.4.2.4 Tiled textures

Tiling the texture improves cache performance when texture walk direction differs from the scanlined memory layout. Since tile dimensions are exclusively a power of two, the changes to the address math come without marginal added cost. 4x4 tiling is used in the final version, although something closer to L1 cache size followed by 4x4 sub-tiling would probably provide better performance.

2.4.3 Render target writes

Since the color data exits the pixel shader in the vertical layout, the render target stores 2x2 quads linearly in horizontal spans of quads. This order is swapped into the correct output format just before displaying the frame to minimize the performance penalty.

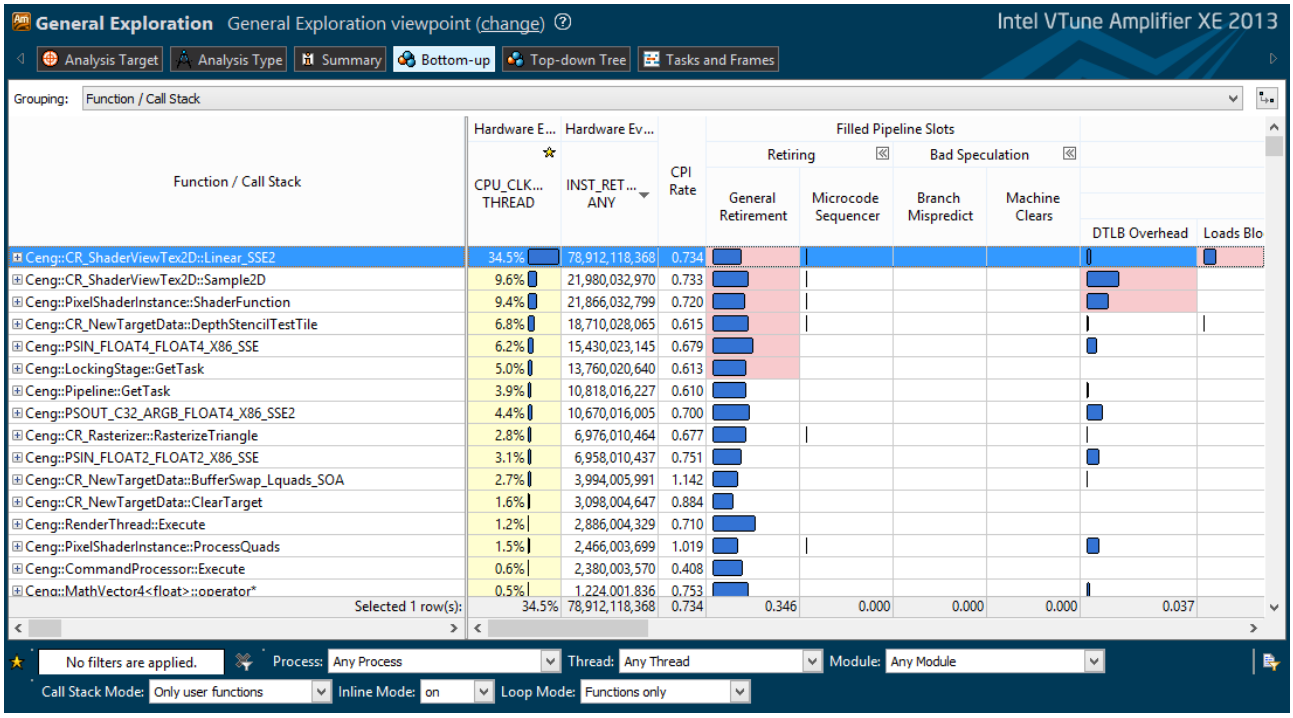
3. Multithreading

There was an initial choice of architecture between dedicated threads and a task based system. In the dedicated system there is are specialized threads for every pipeline stage, and the operating system scheduler ensures fair execution. In the other model there are only *n* threads which are fed by a priority queue. I chose the task based system due to its versatility for future applications.

The system must have a strong ordering so that what comes out matches the order in which draw calls were issued. This is fortunately simplified by the pipelined nature of the rendering process. We can have a separate queue per pipeline stage and only have to ensure that tasks enter the next stage in the same order they entered the current stage. This can be done with a combination of the sliding window protocol and futures. When a task enters a pipeline stage, its correct position in the next stage's queue is determined, and the result is written there once available. The next stage then waits until the task with the smallest serial number is ready and begins to process that, while moving the

sliding window forward by 1.

I first converted all pipeline stages into task form that were processed in a single thread. However, profiling by V-Tune (image 1) revealed that 77% of execution time was consumed by rasterizer and pixel shader stages. I therefore concentrated parallelization efforts to these parts, as I predicted it would give most of the performance increase.

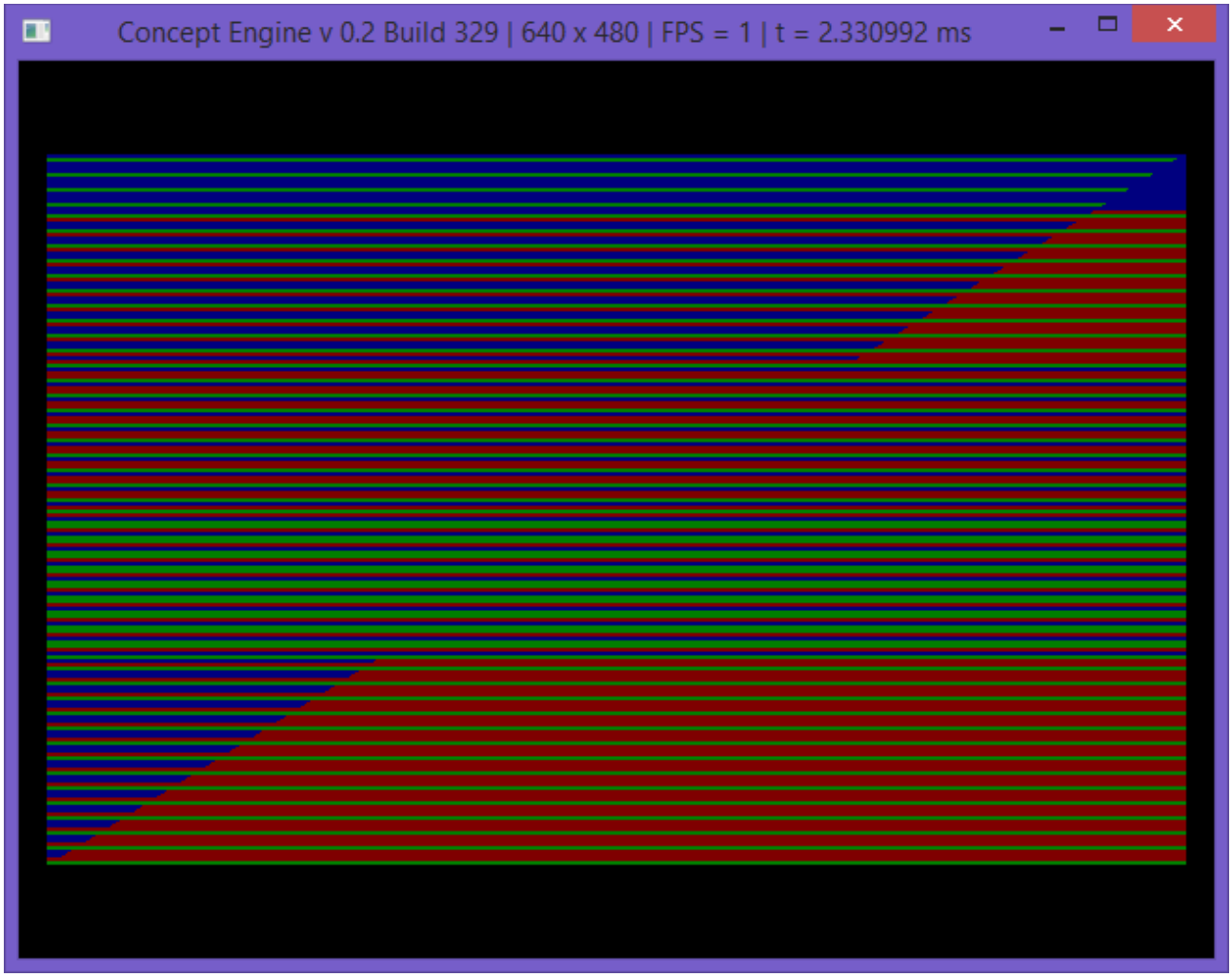


Kuva 1: V-Tune profile. Rasterizer and pixel shader related functions Linear_SSE2, Sample2D, ShaderFunction, DepthStencilTestTile, PSIN_FLOAT4_FLOAT4_X86_SSE, PSOUT_C32_ARGB_FLOAT4_X86_SSE2, RasterizeTriangle, PSIN_FLOAT2_FLOAT2_X86_SSE2 consume 77% of execution time.

These two stages are also significantly easier to parallelize. First, the screen is subdivided into horizontal buckets as can be seen from the debug render in image 2. There are much more buckets than threads to provide load balancing.

In order to avoid false sharing of cache lines, the render buffer is allocated so that each row of 2x2 quads is padded to a full cache line. Additionally, each bucket is an integer multiple of such rows, in this case 3 rows.

Rendering to a bucket is guarded by a mutex. This means we don't have to worry about messing the API ordering of draw calls. However, image 1 shows that the current mutex-protected priority queue consumes 9% of execution time, and could probably be improved by using a separate task manager thread, which would reduce the need for locks.



Kuva 2: Division of work between threads. Each primary color corresponds to a different thread.

4. Conclusions

There is much room for improvement in the design, but I realized by early 2015 that my energy would be much better spent in other endeavors. Moving to OpenGL 3.3 and game engine development were among them.

Still, I learned a lot about vectorization, optimization, data layouts, access patterns, and branch avoidance. I also learned how to write concurrent code, and even some lock-free containers.

Since a shader compiler would have been necessary to tap the full potential of the renderer, I studied assembler design and some compiler theory, but eventually (and wisely) decided that they weren't worth the effort on such a dead end project. The most important remnant of that side project is my current string class which is aware of variable length encodings (utf-8) and introduced me to template metaprogramming.