Fast Non-Uniform 3D Frame

Jarkko Lempiäinen

March, 2022

Abstract

In this paper we present a new 3D frame representation with rotations and non-uniform scaling. The representation has the same memory footprint as the equivalent quaternion representation but with better GPU performance for 3x3 matrix conversions. Unlike quaternions, the new representation also supports extracting z-axis of the frame for free to better support use-cases that doesn't require full frame matrix extraction. In the appendix we provide drop-in HLSL implementations.

1 Quaternion Basics

Quaternions are often used for 3D rotations due to their compact representation and efficient operations that are useful for rotations. Quaternions representing 3D rotations have unit length, and can thus be defined with 3 values instead of 4, and the 4th component calculated as:

$$q_w = \sqrt{1 - q_x^2 - q_y^2 - q_z^2} \tag{1}$$

q_w has two roots $\pm q_w$ and because both $-q$ and $+q$ result in the same 3D rotation, q must be negated if q_w is negative, prior to discarding q_w for the proper recovery of q with the above equation. Vectors can be rotated directly with a unit quaternion using the following equation:

$$\begin{gathered} v' = q * v * q^* = 2q_v \times (q_v \times v + q_w v) + v \\ q_v = [q_x, q_y, q_z] \end{gathered} \tag{2}$$

From the performance point of view, this is fine for rotating couple of vectors, but for more vectors it's more efficient to first convert the quaternion to orthonormal 3x3 matrix and transform vectors with the matrix instead:

$$\begin{aligned} \hat{x} &= [1 - 2q_y^2 - 2q_z^2, 2q_x q_y + 2q_z q_w, 2q_x q_z - 2q_y q_w] \\ \hat{y} &= [2q_x q_y - 2q_z q_w, 1 - 2q_x^2 - 2q_z^2, 2q_y q_z + 2q_x q_w] \\ \hat{z} &= [2q_x q_z + 2q_y q_w, 2q_y q_z - 2q_x q_w, 1 - 2q_x^2 - 2q_y^2] \end{aligned} \tag{3}$$

The terms shown below can be shared to reduce the computation cost of the matrix:

$$1 - 2q_z^2, 2q_x q_y, 2q_x q_z, 2q_y q_z$$

There are also other shared terms ($2q_x^2, 2q_y^2, 2q_x q_w, 2q_y q_w, 2q_z q_w$), but on GPUs supporting Fused Multiply-Add (FMA[1]) and free "multiply by two" modifier, an attempt to share those have a negative performance impact. For example, calculating $\hat{x}_y$ the term $2q_z q_w$ is effectively free with FMA and calculating it separately would just increase the cost. A reminder why simply "counting muls and adds" isn't necessarily a good way to assess the real performance on GPUs.

[1] FMA - https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation

2 PERFORMANCE EVALUATION

For performance evaluation we use Radeon GPU Analyzer[2] by AMD targeting RDNA2 architecture to validate the generated ISA assembly of simple test shaders written in HLSL. However, FMA and "multiply by two"-modifier have been supported by all GPUs for a long time, thus the generated ISA assembly should be quite similar regardless of the target GPU architecture. NVIDIA is unfortunately opaque about the compiled ISA assembly and doesn't allow algorithm performance validation nor optimization for their architectures, so we assume that the generated ISA assembly on NVIDIA GPUs closely matches RDNA2.

Furthermore, we assess the performance based on the utilized full-rate (FR) and quarter-rate (QR) instructions of the compiled ISA assembly. QR instructions that are used for "Transcendental Functions" (e.g. square root and division) are assumed to be substantially more expensive than FR instructions, thus we try to minimize their use. For widely used GCN architecture, QR instructions are 4 times more expensive than FR instructions, while for RDNA it's a bit more complicated[3].

There are various other factors influencing performance, like the GPU architecture and the surrounding code and how Instruction Level Parallelism (ILP) can be utilized in the context, or how code uses VGPRs and influences parallelism and occupancy. We consider these as "factors out of our control" and thus focus purely on FR and QR instruction count with the assumption that lowering the number of the instructions generally improves the overall performance. Explicitly, we don't profile the code as a microbenchmark because these factors can have substantial impact on the performance with misleading results for a code snippet supposed to be embedded in a large shader code body.

As a baseline for the quaternion solution, the basis vectors in Equation (3) can be evaluated with 17 FR instructions. In contrast, transforming a vector directly with a quaternion in Equation (2) requires 18 FR instructions. In addition, calculation of q_w with the Equation (1) followed by matrix conversion with Equation (3) uses 20 FR + 1 QR instructions. A 3D vector multiplication by a 3x3 matrix uses 9 FR instructions. Because unit quaternions represent only 3D rotations, axis scaling requires additional float per scale factor (e.g. 1 float for uniform scaling, or 3 floats for fully non-uniform scaling) and 3 FR instructions per axis, so 3D rotation with fully non-uniform scaling using quaternion representation requires 6 floats and 29 FR + 1 QR instructions to extract the full scaled frame.

[2] Radeon GPU Analyzer - https://gpuopen.com/rga

[3] RDNA Architecture - https://gpuopen.com/wp-content/uploads/2019/08/RDNA_Architecture_public.pdf

3 MOTIVATION

For some use-cases only the z-axis of the 3D frame is needed. For example, to check if a 3D point is within the cone of a spotlight, and if so, only then extract the full frame for shadow map calculation. For this check, it would require 13 FR + 1 QR instruction to extract only the unit $\hat{z}$-axis from the quaternion representation, and if needed, additional 3 FR instructions for the scaled z-axis. Ideally, we would like to have a representation which enables us to get the z-axis for free while having the same storage overhead and computation cost of the quaternion representation.

4 THE REPRESENTATION

To be able to efficiently extract the scaled z-axis of the frame from a representation, we would like to store the axis directly as part of the representation and thus be able to get the axis for free. For the scaled xy-axes of the frame we would like to provide additional data using the same total memory footprint as the quaternion representation that can be evaluated at similar performance cost.

First, we unambiguously define the normalized 3D frame for given z-axis by calculating quaternion rotation from [0, 0, 1] to the z-direction:

$$\begin{aligned} h &= \hat{z} + [0, \varepsilon, 1] \\ q_{zrot} &= [[0,0,1] \times \hat{h}, \hat{h}_z] = [-\hat{h}_y, \hat{h}_x, 0, \hat{h}_z] \end{aligned} \tag{4}$$

ε is a small number to avoid zero-vector thus a singularity when $\hat{z} = [0, 0, -1]$ and small enough to minimize errors for the basis vectors. We use $\varepsilon = 0.000001$, which seems to work well.

Next, we rotate the xy-frame about the z-axis for the desired frame orientation, for which we define the rotated frame xy-axes and transform them with q_{zrot} using the Equation (2):

$$\begin{aligned} x_r &= [\cos\alpha\,, \sin\alpha\,, 0] \\ y_r &= [-\sin\alpha\,, \cos\alpha\,, 0] \\ x &= 2\hat{h} * \left(-\hat{h}_x * \cos\alpha - \hat{h}_y * \sin\alpha\right) + [\cos\alpha\,, \sin\alpha\,, 0] \\ y &= 2\hat{h} * \left(\hat{h}_x * \sin\alpha - \hat{h}_y * \cos\alpha\right) + [-\sin\alpha\,, \cos\alpha\,, 0] \end{aligned} \tag{5}$$

Because the length of h is squared in x and y evaluation (due to multiplying $\hat{h}$ with $\hat{h}_x$ and $\hat{h}_y$), this enables us to discard the normalization of h and multiply the axes by the inverse squared length instead:

$$\begin{aligned} a &= \frac{2}{h \cdot h} \\ x &= h * a * \left(-h_x * \cos\alpha - h_y * \sin\alpha\right) + [\cos\alpha\,, \sin\alpha\,, 0] \\ y &= h * a * \left(h_x * \sin\alpha - h_y * \cos\alpha\right) + [-\sin\alpha\,, \cos\alpha\,, 0] \end{aligned} \tag{6}$$

To introduce non-uniform scaling to these axes we multiply all terms with the scaling factors:

$$\begin{aligned} x &= h * a * scale_x * \left(-h_x * \cos\alpha - h_y * \sin\alpha\right) + scale_x * [\cos\alpha\,, \sin\alpha\,, 0] \\ y &= h * a * scale_y * \left(h_x * \sin\alpha - h_y * \cos\alpha\right) + scale_y * [-\sin\alpha\,, \cos\alpha\,, 0] \end{aligned} \tag{7}$$

Instead of using α, $scale_x$ and $scale_y$, which requires QR transcendental sin and cos instructions, we can pass the scaled rotation coefficients:

$$\begin{gathered} c_x = a * scale_x * \cos\alpha \\ s_x = a * scale_x * \sin\alpha \\ c_y = a * scale_y * \cos\alpha \\ s_y = a * scale_y * \sin\alpha \\ x = h * \left(-h_x * c_x - h_y * s_x\right) + \frac{[c_x, s_x, 0]}{a} \\ y = h * \left(h_x * s_y - h_y * c_y\right) + \frac{[-s_y, c_y, 0]}{a} \end{gathered} \tag{8}$$

The inverse of a in the equation enables us to eliminate the division (QR instruction):

$$\begin{gathered} b = \frac{1}{a} = 0.5 * h \cdot h \\ x = h * \left(-h_x * c_x - h_y * s_x\right) + b * [c_x, s_x, 0] \\ y = h * \left(h_x * s_y - h_y * c_y\right) + b * [-s_y, c_y, 0] \end{gathered} \tag{9}$$

This requires us to pass 4 coefficients c_x, s_x, c_y, s_y to the functions, but we can only afford 3 scalars to achieve the same memory footprint as the quaternion representation. However, we can eliminate one scalar with a little added computation:

$$\begin{gathered} Aspect = \frac{c_y}{c_x} = \frac{s_y}{s_x} = \frac{Scale_y}{Scale_x} \\ x = h * \left(-h_x * c_x - h_y * s_x\right) + b * [c_x, s_x, 0] \\ y = h * Aspect * \left(h_x * s_x - h_y * c_x\right) + b * Aspect * [-s_x, c_x, 0] \end{gathered} \tag{10}$$

Furthermore, in the calculation of h in Equation (4) we use normalized vector $\hat{z}$, but the representation contains scaled z-axis of the frame, which would require vector normalization. Instead of normalizing the z-axis we can calculate h as follows:

$$h = [z_x, z_y + \varepsilon, z_z + |z|] \tag{11}$$

While the evaluation of h still requires QR sqrt instruction, there is no need to multiply the vector with the inverse length, thus saving instructions. If z is already normalized by prior code, the calculation can be eliminated or if prior code calculates $|z|^2$ the result can be used to reduce the cost. The ε can be also applied only upon to rotation coefficient calculation to the scaled z-axis thus eliminating the need to apply it again during the frame extraction.

We also tried to calculate y-axis as the cross-product of x-axis and z-axis and instead of $Aspect$ multiplying the resulting cross-product with $\frac{Scale_y}{Scale_x * Scale_z}$, but this resulted the same number of instructions for the frame extraction.

5 CALCULATING THE ROTATION COEFFICIENTS

The rotation coefficients can be calculated by first extracting the x-axis of the q_{zrot} using the equation (3):

$$x_{zrot} = -a * h_x * h + [1,0,0] \tag{12}$$

The frame extraction with Equation (10) expects the rotation coefficients c_x and s_x to be multiplied by $a * Scale_x$:

$$\begin{aligned} as_x &= a * Scale_x \\ x'_{zrot} &= -a * as_x * h_x * h + [as_x, 0,0] \end{aligned} \tag{13}$$

From this vector we can get the rotation coefficients:

$$\begin{aligned} c_x &= x'_{zrot} \cdot \hat{x} \\ s_x &= -x'_{zrot} \cdot \hat{y} \end{aligned} \tag{14}$$

6 PARTIALLY NON-UNIFORM SCALING

We derived a representation for 3D frame supporting independent non-uniform scaling of all the 3 axes of the frame. However, sometimes it's sufficient to have only a partially non-uniform scaling. Below we describe couple of modified versions with more restricted scaling cases while retaining the good properties of the representation.

A trivial modification is to assume $Aspect = 1$ for uniform xy-scaling with a separate z-scaling, which reduces the memory footprint to 5 values, frame extraction cost to 17 FR + 1 QR instructions and rotation coefficient calculation cost to 19 FR + 1 QR instructions.

Another modification is to have separate scaling for x-axis and y-axis and keep the $\hat{z}$-axis length 1. To retain the 5-value memory footprint the $Aspect$ can be encoded into z-axis length by multiplying the length with $\frac{1}{Aspect} = \frac{Scale_x}{Scale_y}$. This representation requires 26 FR + 1 QR instructions for the frame extraction and 18 FR + 1 QR instruction for calculating the rotation coefficients. The added cost in the frame extraction is due to the required normalization of the scaled z-axis.

7 RESULTS

With the new 3D frame representation, we were able to reduce the non-uniformly scaled and rotated 3D frame computation to 21 FR + 1 QR instructions, which is 8 FR instructions less than the equivalent quaternion representation with the same memory footprint. We can also extract the z-axis for free from the representation for use-cases that doesn't require the full 3D frame. The cost can be further reduced if the z-axis is normalized or if squared length of the z-axis is calculated in a prior use of the representation.

The calculation of the rotation coefficients including the scaling aspect ratio is also quite efficient using only 20 FR + 2 QR instructions making it quite a viable option for some applications. This is quite significantly less than hundreds of lines of branchy ISA assembly we got with the matrix-to-

quaternion code in the Appendix, though the implementation we used can most definitely be improved for GPUs.

Regarding the error, we generated 1 billion random unit quaternions, calculated the 3x3 matrix from them using Equation 3) and performed the conversion back and forth from the representation with scale factors set to 1. For each matrix we calculated the max angular error of the basis vectors of the original and the converted matrix. With $\varepsilon = 0.000001$ and 32-bit floats we got the max error of 0.1°.

8 APPENDIX

```
// epsilon
static const float s_eps=0.000001f;

// get rotation coefficients from orthonormal matrix and scale vector
// scale_.xyz = independent scaling for xyz-axes
float3 get_rot_coeffs_x_y_z(out float3 zdir_, float3x3 rot_, float3 scale_)
{
  float3 h=rot_[2];
  h*=scale_.z;
  h.y+=s_eps;
  zdir_=h;
  h.z+=scale_.z;
  float a=2.0f/dot(h, h);
  float asx=a*scale_.x;
  float3 xdir=h*(-asx*h.x*a);
  xdir.x+=asx;
  return float3(dot(xdir, rot_[0]), -dot(xdir, rot_[1]), scale_.y/scale_.x);
}

// get rotation coefficients from orthonormal matrix and scale vector
// scale_.x = xy-axes scaling, scale_.y = z-axis scaling
float2 get_rot_coeffs_xy_z(out float3 zdir_, float3x3 rot_, float2 scale_)
{
  float3 h=rot_[2];
  h*=scale_.y;
  h.y+=s_eps;
  zdir_=h;
  h.z+=scale_.y;
  float a=2.0f/dot(h, h);
  float asx=a*scale_.x;
  float3 xdir=h*(-asx*h.x*a);
  xdir.x+=asx;
  return float2(dot(xdir, rot_[0]), -dot(xdir, rot_[1]));
}

// get rotation coefficients from orthonormal matrix and frame scale vector
// scale_.x = x-axis scaling, scale_.y = y-axis scaling
float2 get_rot_coeffs_x_y(out float3 zdir_, float3x3 rot_, float2 scale_)
{
  float3 h=rot_[2];
  h.y+=s_eps;
  zdir_=h*(scale_.x/scale_.y);
  h.z+=1.0f;
  float a=2.0f/dot(h, h);
  float asx=a*scale_.x;
  float3 xdir=h*(-asx*h.x*a);
  xdir.x+=asx;
  return float2(dot(xdir, rot_[0]), -dot(xdir, rot_[1]));
}

// extract scaled frame from z-axis and rotation coefficients
// independent scaling for xyz-axes
void extract_frame_x_y_z(out float3 x_, out float3 y_, out float3 z_, float3 zdir_, float3 rc_)
{
  z_=zdir_;
  zdir_.z+=length(zdir_);
```

```
  float sx=(-zdir_.x*rc_.x-zdir_.y*rc_.y);
  float b=dot(zdir_, zdir_)/2.0f;
  x_=zdir_*sx+float3(rc_.x*b, rc_.y*b, 0.0f);
  float sy=(zdir_.x*rc_.y-zdir_.y*rc_.x)*rc_.z;
  b*=rc_.z;
  y_=zdir_*sy+float3(-rc_.y*b, rc_.x*b, 0.0f);
}

// extract scaled frame from z-axis and rotation coefficients
// shared scaling for xy-axes and a separate scaling for z-axis
void extract_frame_xy_z(out float3 x_, out float3 y_, out float3 z_, float3 zdir_, float2 rc_)
{
  z_=zdir_;
  zdir_.z+=length(zdir_);
  float sx=(-zdir_.x*rc_.x-zdir_.y*rc_.y);
  float sy=(zdir_.x*rc_.y-zdir_.y*rc_.x);
  rc_*=dot(zdir_, zdir_)/2.0f;
  x_=zdir_*sx+float3(rc_.x, rc_.y, 0.0f);
  y_=zdir_*sy+float3(-rc_.y, rc_.x, 0.0f);
}

// extract scaled frame from z-axis and rotation coefficients
// separate scaling for x- and y-axes, z-axis is unit length
void extract_frame_x_y(out float3 x_, out float3 y_, out float3 z_, float3 zdir_, float2 rc_)
{
  float rcp_rn=rsqrt(dot(zdir_, zdir_));
  zdir_*=rcp_rn;
  z_=zdir_;
  zdir_.z+=1.0f;
  float sx=(-zdir_.x*rc_.x-zdir_.y*rc_.y);
  float b=dot(zdir_, zdir_)/2.0f;
  x_=zdir_*sx+float3(rc_.x*b, rc_.y*b, 0.0f);
  float sy=(zdir_.x*rc_.y-zdir_.y*rc_.x)*rcp_rn;
  b*=rcp_rn;
  y_=zdir_*sy+float3(-rc_.y*b, rc_.x*b, 0.0f);
}

float4 convert_to_quat(float3x3 m_)
{
  // check for positive matrix trace
  float tr=m_[0].x+m_[1].y+m_[2].z;
  float4 q;
  if(tr>0)
  {
    float s=sqrt(tr+1.0f);
    q.w=s*0.5f;
    s=0.5f/s;
    q.x=(m_[1].z-m_[2].y)*s;
    q.y=(m_[2].x-m_[0].z)*s;
    q.z=(m_[0].y-m_[1].x)*s;
    return q;
  }

  // find largest diagonal value and setup element indices
  uint i=m_[1].y>m_[0].x?1:0;
  if(m_[2].z>m_[i][i])
    i=2;
  const uint next[3]={1, 2, 0};
  uint j=next[i], k=next[j];

  // convert the matrix
  float s=sqrt(m_[i][i]-m_[j][j]-m_[k][k]+1.0f);
  q[i]=s*0.5f;
  s=0.5f/s;
  q.w=(m_[j][k]-m_[k][j])*s;
  q[j]=(m_[i][j]+m_[j][i])*s;
  q[k]=(m_[i][k]+m_[k][i])*s;
  return q;
}
```