Vector length and normalization difficulties

Mike Day, Insomniac Games
mday@insomniacgames.com

Here's a short account of an easily overlooked difficulty with vector length and vector normalization functions, together with one way of solving the problem. We'll use 3-component vectors by way of illustration, but the idea is easily extended to longer or shorter vectors, quaternions, etc. Single-precision floating point is assumed.

The typical implementation of a vector normalize function might look like this, assuming a suitable definition of the '/' operator for vectors:

```
inline Vec3 VecNormalize(const Vec3& v)
{
  return v / VecLength(v);
}
```

Depending on the definition of VecLength(), there will be some problems here with the way small and large vectors are handled. The usual approach to computing the length involves squaring and adding the components, and taking the square root:

```
inline float VecLength(const Vec3& v)
{
  return Sqrtf((v.x * v.x) + (v.y * v.y) + (v.z * v.z));
}
```

This is of course completely standard. But, since components must be squared before summing, small numbers can lead to underflow. For example if we pass the vector (3.0e-25f, 4.0e-25f, 0.0f), despite the fact that the magnitude is 5.0e-25f and easily within the range of representable floats, the first two components underflow to zero when squared and the returned length is zero.

In general, the value returned is only sensible for vectors down to a length of about 10^{-19} or 10^{-22}, depending on whether denormal numbers are supported and on whether the particular implementation of Sqrtf() is able to handle denormals.

Of course the corresponding problem arises for large vectors – squared components can overflow before (or during) summation, and only lengths up to about 10^{19} can be accommodated. The net result is that VecLength() only works for about half the exponent range of the input vector, which is a pity, and could easily catch a user by surprise.

The range of supported inputs can be extended to cover the entire floating point range if we first rescale the vector so that its components lie within the 'good' half of the exponent range, and bearing in mind that the result will need scaling back to the proper range afterwards. One easy way to achieve this is to first divide the vector through by the absolute magnitude of its largest component:

```
inline Vec3 VecAbs(const Vec3& v)
{
  Vec3 r;
  r.x = Absf(v.x);
  r.y = Absf(v.y);
  r.z = Absf(v.z);
  return r;
}


inline float VecMaxAbsComponent(const Vec3& v)
{
  Vec3 a = VecAbs(v);
  return Maxf(a.x, Maxf(a.y, a.z));
}


inline float VecLengthFast(const Vec3& v)
{
  return Sqrtf((v.x * v.x) + (v.y * v.y) + (v.z * v.z));
}


inline float VecLength(const Vec3& v)
{
  float m = VecMaxAbsComponent(v);
  return SelectGreaterZerof(m, m * VecLengthFast(v/m), 0.0f);
}
```

The length function does a bit more work, but all vectors containing finite float values are now supported. Note that the original naïve length function still exists in this implementation under the name VecLengthFast(), which the user can call if there's prior knowledge that the range of inputs is limited to the 'good' values. The zero vector has been singled out as a special case because it would otherwise cause trouble – division of zero by zero produces a NaN which would propagate through to the return value.

The vector normalization function can benefit from a similar treatment:

```
inline Vec3 VecNormalizeFast(const Vec3& v)
{
  return v / VecLengthFast(v);
}


inline Vec3 VecNormalize(const Vec3& v)
{
  float m = VecMaxAbsComponent(v);
  return (m > 0) ? VecNormalizeFast(v/m) : v;
}
```

This implementation returns the zero vector upon encountering a zero input vector. Depending on taste, it may be preferable to return a fixed unit vector under these circumstances, (1,0,0) say. Also, it may be possible to hint to the compiler that we expect the ‘m>0’ condition to be true, so that it can improve branch prediction.