qstring

- Chris Larsen - larsencr@gmail.com

String manipulation utility for analytically multiplying quaternions.

Euler's Rotation Theorem

- If two consecutive rotations are performed there is a single rotation which provides the same net effect.
- Improve code efficiency by collapsing multiple, consecutive rotations into a single rotation.

Procedure

- Convert both rotations to quaternions
  - θ is the angle of rotation,
  - $\bar{v}$ is the rotation vector, normalized if necessary.
- Then the quaternion

  $$\cos(\theta/2) + \bar{v}_x\sin(\theta/2)\boldsymbol{i} + \bar{v}_y\sin(\theta/2)\boldsymbol{j} + \bar{v}_z\sin(\theta/2)\boldsymbol{k}$$

  represents the rotation.
- Code samples below use C++ with quaternion class definition.

Example

- Start with the code

```
GFX_rotate(-90.0f, 1.0f, 0.0f, 0.0f);
GFX_rotate(-rotz,  0.0f, 0.0f, 1.0f);
```

- Replace with a version of `GFX_rotate()` which uses quaternions:

```
const float beta(-90.0f * DEG_TO_RAD_DIV_2);
const float cosBeta(cosf(beta)),
            sinBeta(sinf(beta));
const quaternion q0(cosBeta,
                    sinBeta, 0.0f, 0.0f);
float alpha(-rotz * DEG_TO_RAD_DIV_2);
float cosAlpha(cosf(alpha)),
      sinAlpha(sinf(alpha));
quaternion q1(cosAlpha, 0.0f, 0.0f, sinAlpha);
GFX_rotate(q0);
GFX_rotate(q1);
```

Example (cont'd)

- Reduce to a single rotation:

```
const float beta(-90.0f * DEG_TO_RAD_DIV_2);
const float cosBeta(cosf(beta)),
            sinBeta(sinf(beta));
const quaternion q0(cosBeta,
                    sinBeta, 0.0f, 0.0f);
float alpha(-rotz * DEG_TO_RAD_DIV_2);
float cosAlpha(cosf(alpha)),
      sinAlpha(sinf(alpha));
quaternion q1(cosAlpha, 0.0f, 0.0f, sinAlpha);
GFX_rotate(q0*q1);
```

How to Simplify?

- The first quaternion is
  - a constant, and
  - the sine and cosine of its half angle can be calculated analytically
- θ is -90° so θ/2 is -45°
- cosine(-45°) is 1/sqrt(2), and sin(-45°) is -1/sqrt(2). C/C++ provides the math constant `M_SQRT1_2` so q0 is

```
const quaternion q0(M_SQRT1_2,
                   -M_SQRT1_2, 0.0f, 0.0f);
```

How to Simplify? (cont'd)

- Code becomes

```
const quaternion q0(M_SQRT1_2,
                   -M_SQRT1_2, 0.0f, 0.0f);
float alpha(-rotz * DEG_TO_RAD_DIV_2);
float cosAlpha(cosf(alpha)),
      sinAlpha(sinf(alpha));
quaternion q1(cosAlpha, 0.0f, 0.0f, sinAlpha);
GFX_rotate(q0*q1);
```

- Eliminated two calls to transcendental functions (sine & cosine), and these are generally expensive.
- But can still do better ...

Quaternion Multiplication

- In the general case, quaternion multiplication requires
  - 16 floating point multiplications, and
  - 12 floating point additions/subtractions
- Each of the quaternions has two zero components so many of the multiplications don't need to be done.
- Since many of the intermediate products are zero, many of the additions aren't needed either.

Quaternion Multiplication (cont'd)

- Computing the product analytically the code becomes

```
float alpha(-rotz * DEG_TO_RAD_DIV_2);
float cosAlpha(cosf(alpha)),
      sinAlpha(sinf(alpha));
GFX_rotate(quaternion( M_SQRT1_2*sinAlpha,
                      -M_SQRT1_2*sinAlpha,
                       M_SQRT1_2*sinBeta,
                       M_SQRT1_2*sinBeta);
```

- This reduces the quaternion multiplication to
  - 4 floating point multiplications, and
  - 0 additions/subtractions

But that was only two ...

- This was a simple case, only two quaternions (with a lot of zeroes) needed to be multiplied out by hand
- What about when things get more complicated?

Three Quaternions

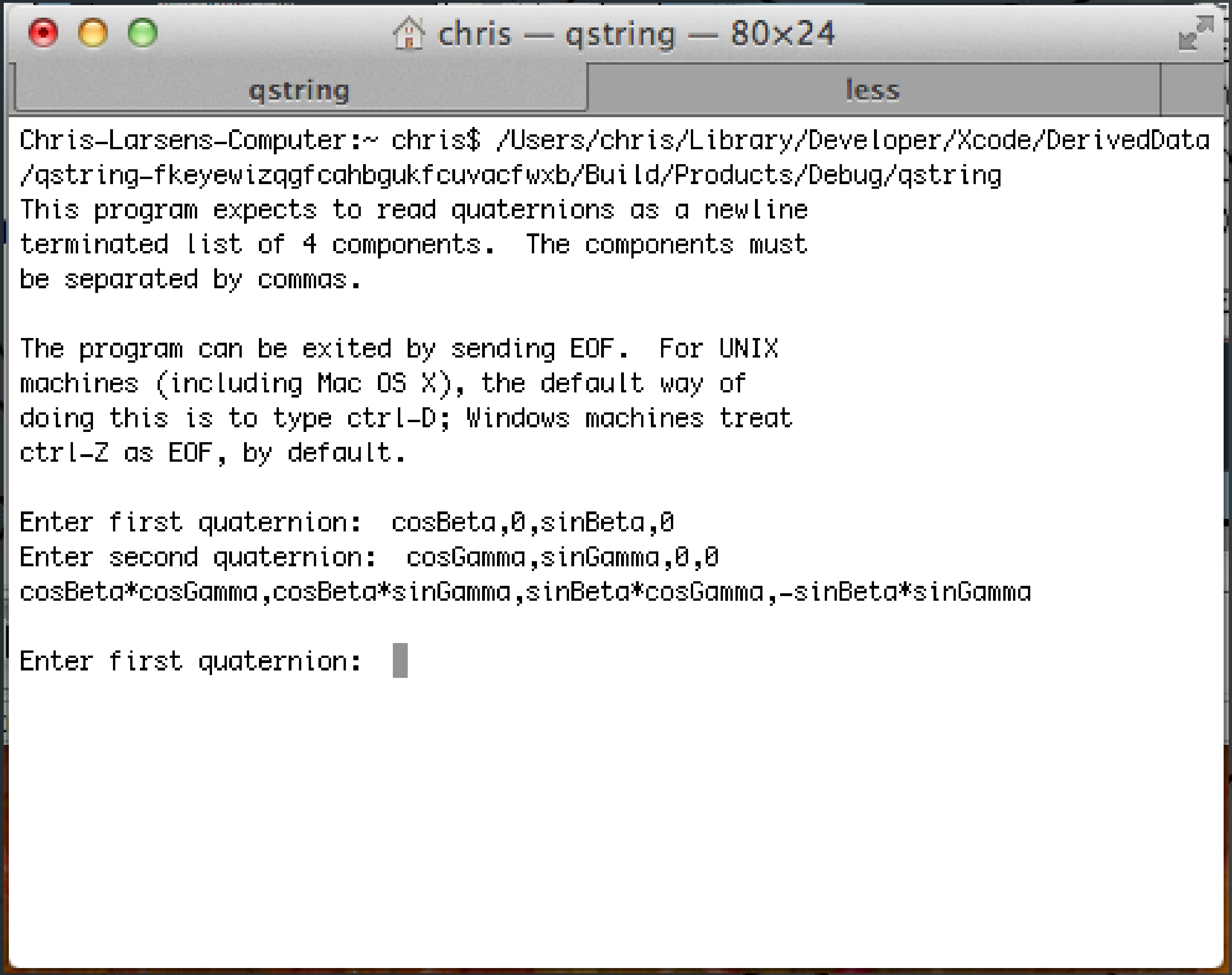
- I don't really want to calculate `q0*q1*q2` manually:

```
float alpha(rotz * DEG_TO_RAD_DIV_2);
float cosAlpha(cosf(alpha)),
      sinAlpha(sinf(alpha));
quaternion q0(cosAlpha, 0.0f, 0.0f, sinAlpha);
float beta(roty * DEG_TO_RAD_DIV_2);
float cosBeta(cosf(beta)), sinBeta(sinf(beta));
quaternion q1(cosBeta, 0.0f, sinBeta, 0.0f);
float gamma(rotx * DEG_TO_RAD_DIV_2);
float cosGamma(cosf(gamma)),
      sinGamma(sinf(gamma));
quaternion q2(cosGamma, sinGamma, 0.0f, 0.0f);
GFX_rotate(q0*q1*q2);
```

Automation!

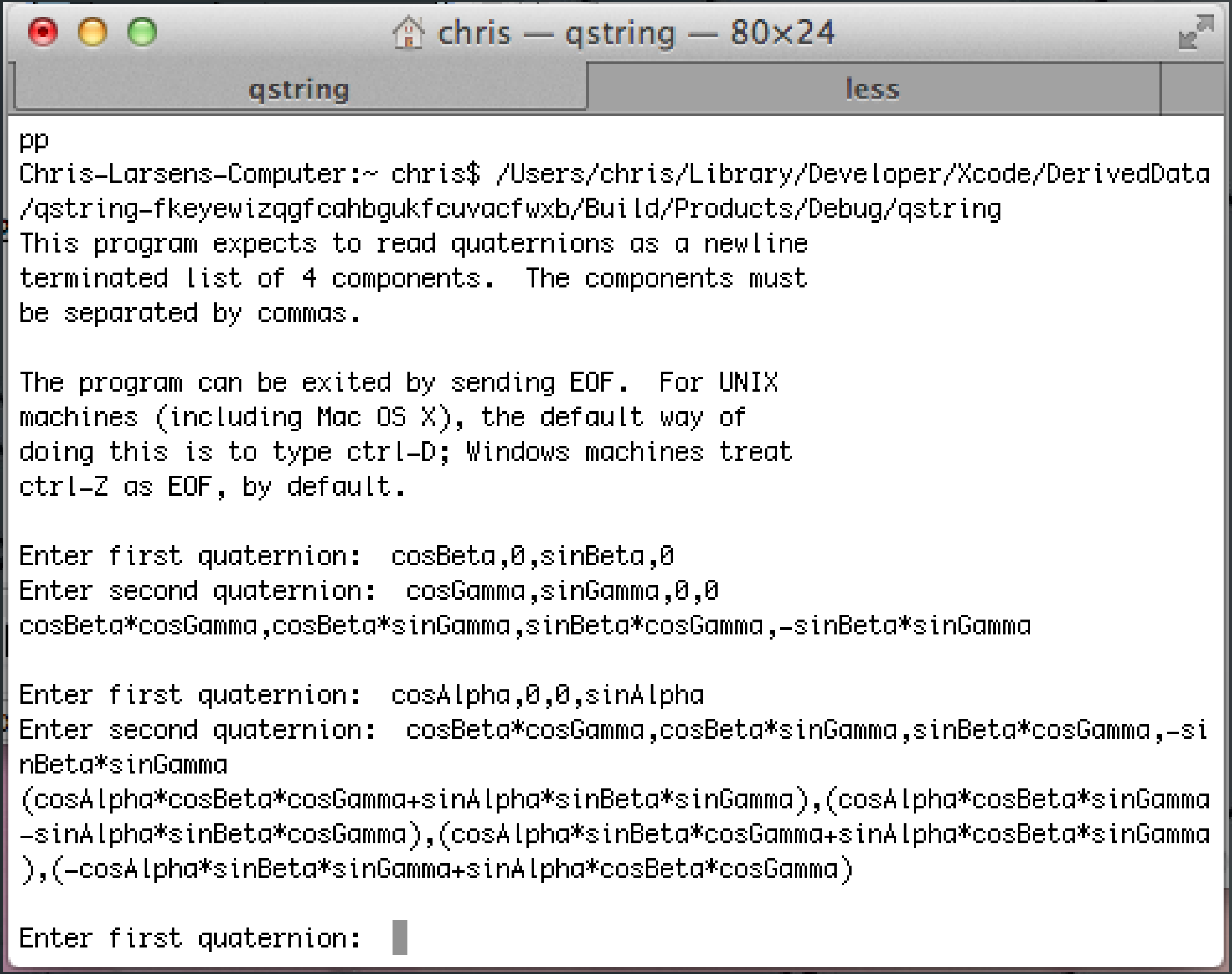
- Wrote a program called qstring
  - Reads two quaternions,
  - Calculates product analytically,
  - Deals with special product cases when either multiplier or multiplicand is zero or one.

Using qstring - 1st Product



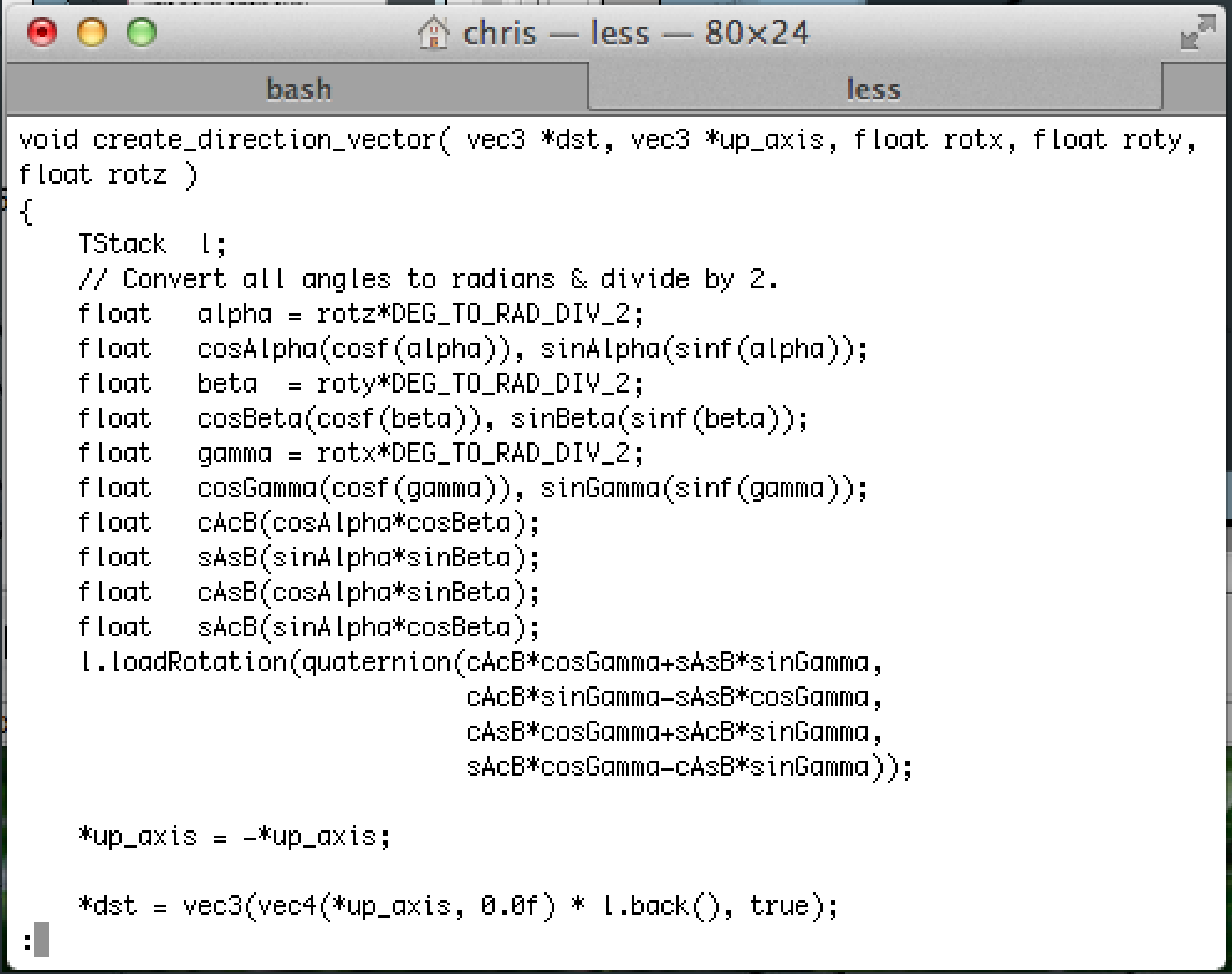
```
Chris-Larsens-Computer:~ chris$ /Users/chris/Library/Developer/Xcode/DerivedData
/qstring-fkeyewizqgfcahbgukfcuvacfwxb/Build/Products/Debug/qstring
This program expects to read quaternions as a newline
terminated list of 4 components.  The components must
be separated by commas.

The program can be exited by sending EOF.  For UNIX
machines (including Mac OS X), the default way of
doing this is to type ctrl-D; Windows machines treat
ctrl-Z as EOF, by default.

Enter first quaternion:  cosBeta,0,sinBeta,0
Enter second quaternion:  cosGamma,sinGamma,0,0
cosBeta*cosGamma,cosBeta*sinGamma,sinBeta*cosGamma,-sinBeta*sinGamma

Enter first quaternion:
```

Using qstring - 2nd Product



```
pp
Chris-Larsens-Computer:~ chris$ /Users/chris/Library/Developer/Xcode/DerivedData
/qstring-fkeyewizqgfcahbgukfcuvacfwxb/Build/Products/Debug/qstring
This program expects to read quaternions as a newline
terminated list of 4 components.  The components must
be separated by commas.

The program can be exited by sending EOF.  For UNIX
machines (including Mac OS X), the default way of
doing this is to type ctrl-D; Windows machines treat
ctrl-Z as EOF, by default.

Enter first quaternion:  cosBeta,0,sinBeta,0
Enter second quaternion:  cosGamma,sinGamma,0,0
cosBeta*cosGamma,cosBeta*sinGamma,sinBeta*cosGamma,-sinBeta*sinGamma

Enter first quaternion:  cosAlpha,0,0,sinAlpha
Enter second quaternion:  cosBeta*cosGamma,cosBeta*sinGamma,sinBeta*cosGamma,-si
nBeta*sinGamma
(cosAlpha*cosBeta*cosGamma+sinAlpha*sinBeta*sinGamma),(cosAlpha*cosBeta*sinGamma
-sinAlpha*sinBeta*cosGamma),(cosAlpha*sinBeta*cosGamma+sinAlpha*cosBeta*sinGamma
),(-cosAlpha*sinBeta*sinGamma+sinAlpha*cosBeta*cosGamma)

Enter first quaternion:  
```

Actual Code Using This Example



```
void create_direction_vector( vec3 *dst, vec3 *up_axis, float rotx, float roty,
float rotz )
{
    TStack  l;
    // Convert all angles to radians & divide by 2.
    float   alpha = rotz*DEG_TO_RAD_DIV_2;
    float   cosAlpha(cosf(alpha)), sinAlpha(sinf(alpha));
    float   beta  = roty*DEG_TO_RAD_DIV_2;
    float   cosBeta(cosf(beta)), sinBeta(sinf(beta));
    float   gamma = rotx*DEG_TO_RAD_DIV_2;
    float   cosGamma(cosf(gamma)), sinGamma(sinf(gamma));
    float   cAcB(cosAlpha*cosBeta);
    float   sAsB(sinAlpha*sinBeta);
    float   cAsB(cosAlpha*sinBeta);
    float   sAcB(sinAlpha*cosBeta);
    l.loadRotation(quaternion(cAcB*cosGamma+sAsB*sinGamma,
                              cAcB*sinGamma-sAsB*cosGamma,
                              cAsB*cosGamma+sAcB*sinGamma,
                              sAcB*cosGamma-cAsB*sinGamma));

    *up_axis = -*up_axis;

    *dst = vec3(vec4(*up_axis, 0.0f) * l.back(), true);
:
```

Cost Calculation

- General case quaternion multiplication:
  - 32 floating point multiplies
  - 24 floating point additions
- Special case multiplication with 50% zeroes:
  - 12 floating point multiplies
  - 4 floating point additions

GitHub Repository

http://github.com/crlarsen/qstring