Getting Started with GaussianLib

Lukas Hermanns

August 7, 2015

Introduction

The GaussianLib is a simple C++ library for 2D and 3D applications. It provides only basic linear algebra functionality for Vectors, Matrices, and Quaternions.

Compilation

In the following we consider to have a single C++ file named "Example.cpp". More over `%GaussianLibPath%` denotes your GaussianLib installation directory.

GNU/C++

The GaussianLib requires `g++` version 4.8.1 or higher, with C++11 feature set enabled. To compile your application with GNU/C++ (or MinGW on Windows), type this into a command line:

```
g++ -I %GaussianLibPath%/include -std=c++11 Example.cpp -o ExampleOutput
```

If everything worked properly, your executable is named "ExampleOutput".

VisualC++

The GaussianLib requires VisualC++ 2013 (12.0) or higher, to support the C++11 features, which are used in the library.

Vectors

In the GaussianLib vectors are considered to be **column vectors** per default, as it is common in mathematics. I.e. if you want a vector y as a result of a multiplication with a matrix M and a vector x, write: $y = M * x$. To use **row vectors** instead, define the macro `GS_ROW_VECTORS` before you include the library. There is the generalized base `Vector<T, N>` class, where `T` specifies the data type of the vector components and `N` specifies the number of components. There are three specialized template classes for vectors: `Vector<T, 2>`, `Vector<T, 3>`, and `Vector<T, 4>`. There are also pre-defined type aliases (*N* is either 2, 3, or 4):

- `VectorN` Is a type alias to `VectorNT<Real>`, where `Real` is either from type `float` or `double`.
- `VectorNf` Is a type alias to `VectorNT<float>`.
- `VectorNd` Is a type alias to `VectorNT<double>`.
- `VectorNi` Is a type alias to `VectorNT<int>`.
- `VectorNui` Is a type alias to `VectorNT<unsigned int>`.
- `VectorNb` Is a type alias to `VectorNT<char>`.
- `VectorNub` Is a type alias to `VectorNT<unsigned char>`.

```cpp
#include <Gauss/Gauss.h>
#include <iostream>

int main()
{
    Gs::Vector3 a(1, 2, 3), b(4, 5, 6);

    std::cout << "a = " << a << std::endl;
    std::cout << "b = " << b << std::endl;
    std::cout << "a * b = " << a*b << std::endl;                // Per-component multiplication
    std::cout << "a . b = " << Dot(a, b) << std::endl;          // Dot product (or scalar product)
    std::cout << "a X b = " << Cross(a, b) << std::endl;        // Cross product (or vector product)
    std::cout << "a V b = " << Angle(a, b) << std::endl;        // Vector angle (in radians)
    std::cout << "|a| = " << a.Length() << std::endl;           // Vector length (or norm of the vector)
    std::cout << "|a|^2 = " << a.LengthSq() << std::endl;       // Squared vector length
    std::cout << "a / |a| = " << a.Normalize() << std::endl; // Normalized vector (unit length of 1)

    return 0;
}
```

For the specialized Vector templates, there are public members available: `x`, `y`, `z`, and `w`. I.e. you are not restricted to the bracket operator `[]` to access vector components:

```cpp
a.x = 2;
a.z = 3;
a[0] += 2; // equivalent to a.x += 2;
```

Matrices

There is only a single general-purpose class for matrices (except AffineMatrix3T and AffineMatrix4T, see section Affine Matrices): `Matrix<T, Rows, Cols>`, where `T` specifies the template typename T, `Rows` specifies the number of rows of the matrix, and `Cols` specifies the number of columns of the matrix.

```cpp
#include <Gauss/Gauss.h>
#include <iostream>

int main()
{
    Gs::Matrix4 A;
    A << 1,  0, 2, 0,
         0, -2, 0, 1,
         4,  0, 5, 6,
         0,  1, 0, 1;

    Gs::Matrix<float, 3, 4> B;
    B << 4, 2,  9, -1,
         0, 1,  6,  2,
         3, 7, -2,  0;

    Gs::Matrix<float, 4, 3> C;
    C = B.Transposed();

    Gs::Matrix<float, 3, 3> D;
    D = B*C;

    Gs::Matrix<float, 4, 4> E;
    E = C*B;

    std::cout << "A = " << std::endl << A << std::endl;
    std::cout << "B = " << std::endl << B << std::endl;
    std::cout << "C = " << std::endl << C << std::endl;
    std::cout << "B*C = " << std::endl << D << std::endl;
    std::cout << "C*B = " << std::endl << E << std::endl;

    std::cout << "A^-1 = " << std::endl << A.Inverse() << std::endl;
    std::cout << "A*A^-1 = " << std::endl << A*A.Inverse() << std::endl;
    std::cout << "Trace(A) = " << std::endl << A.Trace() << std::endl;
    std::cout << "Determinant(A) = " << std::endl << A.Determinant() << std::endl;

    return 0;
}
```

Affine Matrices

In 3D applications a 4x4 matrix is frequently used for affine transformations of 3D models, i.e. translation, rotation, scaling, and sometimes shearing. However, with many 3D models, such transformations require a lot of memory. Moreover, the 4th row of these 4x4 affine matrices is always $(0,0,0,1)$ — except that row vectors are used, where the 4th column is always $(0,0,0,1)$.

To reduce the memory footprint (and some computations), the GaussianLib provides the `AffineMatrix4T<T>` class, where the 4th row (or column for row-vectors) is implicit, and the `AffineMatrix3T<T>` class, where the 3rd row (or column for row-vectors) is implicit:

```
#include <Gauss/Gauss.h>

int main()
{
    // Affine matrices are always initialized to their identity matrix
    Gs::AffineMatrix4 m;

    m.Translate(Gs::Vector3(0, 4, -2));
    m.RotateX(M_PI*0.5);
    m.RotateFree(Gs::Vector3(1, 1, 1), M_PI*1.5);
    m.Scale(Gs::Vector3(1, 0.5, 2));
    m.MakeInverse();

    Gs::Vector3 v(0, 0, 1);
    auto a = Gs::TransformVector(m, v); // Rotate and Translate (with implicit v.w = 1)
    auto b = Gs::RotateVector(m, v); // Only rotate

    return 0;
}
```

Quaternions

Quaternions have the four components `x`, `y`, `z`, and `w` just like Vector4. In contrast to vectors, quaterions can only have floating-point components.

```
#include <Gauss/Gauss.h>

int main()
{
    Gs::Quaternion q0, q1; // Equivalent to Gs::QuaternionT<Gs::Real>
    Gs::Quaternionf qFloat;
    Gs::QuaternionT<double> qDouble;

    // Spherical Linear intERPolation (SLERP) between q0 and q1
    auto q2 = Gs::Slerp(q0, q1, 0.5);

    // Convert to 3x3 matrix
    Gs::Matrix3 rotation = q2.ToMatrix3();

    // Store rotation of quaterion in the left-upper 3x3 matrix of the sparse 4x4 matrix 'transform'
    Gs::AffineMatrix4 transform;
    Gs::QuaternionToMatrix(transform, q2);

    return 0;
}
```

Swizzle Operator

For the three vector classes, there is support for the *swizzle operator* (like in shading languages):

```
// Enable 'swizzle operator'
#define GS_ENABLE_SWIZZLE_OPERATOR

#include <Gauss/Gauss.h>

int main()
{
    Gs::Vector4 a, b;
    Gs::Vector3 c, d
```

```
    Gs::Vector2 e, f;

    e = a.xy();
    f = a.zz();
    c = a.xxz() + e.yxy()*2.0f;
    b = a.xyxy();

    // References can not be used (pointless operation).
    //a.yz() += e;
    //a.zx() *= 2;

    a = e.xxyy();
}
```

Every combination is possible!

Shading Languages

There are two extra header files, which can be included optionally:

```
#define GS_ENABLE_SWIZZLE_OPERATOR
#include <Gauss/Gauss.h>

// Includes all type aliases with name conventions of the DirectX High Level Shading Language HLSL.
#include <Gauss/HLSLTypes.h>

// Includes all type aliases with name conventions of the OpenGL Shading Language (GLSL).
#include <Gauss/GLSLTypes.h>

int main()
{
    // HLSL types
    float4x4 m0;
    double2x3 m1;
    int3 v0;

    // GLSL types
    mat4 m2 = m0;
    ivec2 v1 = v0.yz();
    ivec3 v2 = v0.xxy();

    return 0;
}
```

Fine Tuning

By default, all vectors, quaternions, and matrices are initialized. To increase performance by not automatically initialize this data, add the following to your compiler pre-defined macros:

`GS_DISABLE_AUTO_INIT`

If you don't want to disable the automatic initialization overall, you can explicitly construct a data type who is uninitialized. This can be done with `UninitializeTag`:

```
Gs::Matrix4 m(Gs::UninitializeTag{});
```

`UninitializeTag` is an empty `struct`, so no memory will be allocated. It's just a hint to the compiler, to call another constructor, which does no initialization. Note, that uninitialized data should always be explicitly marked as such!