Getting Started with GaussianLib

Lukas Hermanns

July 27, 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 consideres to be **column vectors**, 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: $y = M * x$. There are three base classes for vectors: `Vector2T<T>`, `Vector3T<T>`, and `Vector4T<T>`, where `<T>` specifies the template typename T. 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>`.

```
#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;
    std::cout << "a . b = " << Dot(a, b) << std::endl;
    std::cout << "a X b = " << Cross(a, b) << std::endl;
    std::cout << "|a| = " << a.Length() << std::endl;
    std::cout << "a / |a| = " << a.Normalize() << std::endl;

    return 0;
}
```

The vector classes have not been generlaized as much as the matrix class. This is due to support the public members `x`, `y`, `z`, and `w`. I.e. you are not restricted to the bracket operator `[]` to access vector components:

```
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 SparseMatrix4T, see section Sparse 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.

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

int main()
{
    Gs::Matrix4 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;
    std::cout << "a . b = " << Dot(a, b) << std::endl;
    std::cout << "a X b = " << Cross(a, b) << std::endl;
    std::cout << "|a| = " << a.Length() << std::endl;
    std::cout << "a / |a| = " << a.Normalize() << std::endl;

    return 0;
}
```

Sparse Matrices

In 3D applications a 4x4 matrix is frequently used for transformations of 3D models. However, with many 3D models, such transformations require a lot of memory. Moreover, the 4th row of these 4x4 matrices is always $(0, 0, 0, 1)$ — assumed that the transformation only consists of translations, rotations, and scaling.

To reduce the memory footprint (and some computations) the GaussianLib provides the `SparseMatrix4T<T>` class, where the 4th row is implicit:

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

int main()
{
    Gs::SparseMatrix4 m = Gs::SparseMatrix4::Identity();

    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, 0);
    auto a = m.Transform(v); // Rotate and Translate (with implicit v.w = 1)
    auto b = m.Rotate(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 = 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::SparseMatrix4 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();
    b = a.xyxy();

    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 the `UninitializeTag` tag:

```
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!