Getting Started with GaussianLib

Lukas Hermanns

July 25, 2015

Introduction

The GAUSSIANLIB is a simple C++ library for 2D and 3D applications. It provides only basic linear algebra functionality such as 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++

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".

Vectors

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):

- `Vector`*N* Is a type alias to `Vector`*N*`T<Real>`, where `Real` is either from type `float` or `double`.

- `Vector`*N*`f` Is a type alias to `Vector`*N*`T<float>`.

- `Vector`*N*`d` Is a type alias to `Vector`*N*`T<double>`.

- `Vector`*N*`i` Is a type alias to `Vector`*N*`T<int>`.

- `Vector`*N*`ui` Is a type alias to `Vector`*N*`T<unsigned int>`.

- `Vector`*N*`b` Is a type alias to `Vector`*N*`T<char>`.

- `Vector`*N*`ub` Is a type alias to `Vector`*N*`T<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;
    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 SparesMatrix4T, 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.

```cpp
#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:

```cpp
#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

. . .

Swizzle Operator

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

```cpp
// 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:

```cpp
#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>

using namespace Gs;

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:

```cpp
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!