Eigen Tutorial

CS2240 **Advanced** Computer Graphics

Introduction

Eigen is an open-source linear algebra library implemented in C++. It's fast and well-suited for a wide range of tasks, from heavy numerical computation, to simple vector arithmetic. The goal of this tutorial is to introduce the features of Eigen required for implementing graphics applications, to readers possessing basic knowledge of C++, linear algebra, and computer graphics.

Goals

After reading this tutorial, the reader should be able to

1. Install Eigen on computers running Linux, Mac OS, and Windows.
2. Create and initialize matrices and vectors of any size with Eigen in C++.
3. Use Eigen for basic algebraic operations on matrices and vectors. The reader should be able to perform addition, multiplication, scalar multiplication, and matrix inversion and transposition.
4. Use Eigen's built-in functions to create 4x4 transformation matrices.

Installing Eigen

We'll use Git to download the source files for Eigen into the user's home directory. You can, of course, use any directory you prefer - just replace the tilde `~` in the following commands with your desired download path.

```
$ cd ~
$ git clone https://github.com/eigenteam/eigen-git-mirror
```

If you're like me and the idea of building libraries from source gives you the jitters, don't worry - we don't need to build anything here. Eigen uses *pure header libraries*, which means all its source code is included in header files. There are no separate **.CPP** files for function and class definitions - everything goes into the **.H** files. This makes using Eigen quite simple. To begin using it, simply `#include` these header files at the beginning of your own code. You can find these headers in `~/eigen-git-mirror/Eigen/src`.

Of course, the compiler needs to be told the location on disk of any header files you include. You can copy the `~/eigen-git-mirror/Eigen` folder to each new project's directory. Or, you could add the folder once to the compiler's default include path. You can do this by running the following command on machines with Linux or Mac OS:

```
$ sudo ln -s /usr/local/include ~/eigen-git-mirror/Eigen
```

If you're using a lab machine, add the following line to the end of `~/.bash_profile`

```
export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH$:~/eigen-git-mirror
```

If you're using QTCreator, add the following line to your project (**.PROJ**) file:

```
QMAKE_CXXFLAGS = -I "~/eigen-git-mirror"
```

Good day, Universe!

Let's test our installation by writing a simple program. If you've followed the steps above, you should be able to compile the following piece of code without any additional configuration.

```
#include <Eigen/Core>
#include <iostream>

using namespace std;
using namespace Eigen;

int main() {
  cout << "Eigen version: " << EIGEN_MAJOR_VERSION << "."
                            << EIGEN_MINOR_VERSION << endl;
  return 0;
}
```

Matrices

Creating matrices with Eigen is simple:

```
Matrix3f A;
Matrix4d B;
```

Eigen uses a naming convention for its datatypes that is quite similar to OpenGL. The actual datatype name is followed by suffixes that describe its size, and the type of its member elements respectively. In the example above, the variable `A` is of type `Matrix3f` — that is, a 3x3 matrix of floats. The variable `B` is a 4x4 matrix of doubles. Keep in mind, though, that not all combinations of size and type are valid — `Matrix2i` works, but `Matrix5s` throws an error. That is not to say you can't create 5x5 matrices of type `short`, or that you can only create square matrices. Doing so merely requires a slightly different syntax:

```
// 5x5 matrix of type short
Matrix<short, 5, 5> M1;

// 20x75 matrix of type float
Matrix<float, 20, 75>  M2;
```

In fact, this is how *all* matrices in Eigen are created under the hood. The datatype names described above are only provided as a convenience for commonly used sizes and types. What's more, Eigen even allows us to create matrices whose size is not known at compile time by using an `X` in place of the size suffix (`MatrixXf`, `MatrixXd`).

However, as the majority of graphics operations only require us to work with 3x3 and 4x4 single or double precision matrices, we will restrict our attention in this tutorial to the datatypes `Matrix3f`, `Matrix4f`, `Matrix3d`, and `Matrix4d`.

Initialization

Now that we've created our matrix variables, let's assign some values to them:

```
// Initialize A
A << 1.0f, 0.0f, 0.0f,
     0.0f, 1.0f, 0.0f,
     0.0f, 0.0f, 1.0f;

// Initialize B by accessing individual elements
for i = 1:4 {
  for j = 1:4 {
    B(j, i) = 0.0;
  }
}
```

The first method shown above uses the comma initializer syntax: the programmer specifies the coefficients *row by row*. Note that all coefficients need to be provided — if the number of coefficients does not match the size of the matrix, the compiler will throw an error. Thus, you can imagine this syntax becoming quite cumbersome for large matrices. [1]

The second method accesses the individual coefficients of the matrix B using the overloaded parantheses operators. Indexing starts at zero, with the first index specifying the row number, and the second the column number. As you have probably guessed, you can use the same syntax to query the coefficients. Can you guess whether the following code prints a zero or a one?

```
cout << A(1, 2);
```

Notice that we have structured our loops such that the outer loop runs over the columns, and the inner loop iterates over the rows. Doing it the other way around would have worked too, but would have been less efficient. This is because Eigen stores matrices in *column-major order* by default. This can be a bit confusing since coefficients in the comma initialization format are specified in row-major order. Another potential point of confusion is that unlike C/C++, or Python arrays, Eigen uses parantheses rather than square brackets to access matrix elements.

In addition to the above two methods, Eigen provides utility functions to initialize matrices to predefined values:

```
// Set each coefficient to a uniform random value in the range
   [-1, 1]
A = Matrix3f::Random();

// Set B to the identity matrix
B = Matrix4d::Identity();

// Set all elements to zero
A = Matrix3f::Zero();

// Set all elements to ones
A = Matrix3f::Ones();
```

[1] When using comma initialization, the inputs can even be other matrices and vectors. You can learn more about this feature at `https://eigen.tuxfamily.org/dox/group__TutorialAdvancedInitialization.html`.

```
// Set all elements to a constant value
B = Matrix4d::Constant(4.5);
```

Matrix Operations

Common arithmetic operators are overloaded to work with matrices:

```
Matrix4f M1 = Matrix4f::Random();
Matrix4f M2 = Matrix4f::Constant(2.2);

// Addition
// The size and the coefficient-types of the matrices must match
cout << M1 + M2 << endl;

// Matrix multiplication
// The inner dimensions and the coefficient-types must match
cout << M1 * M2 << endl;

// Scalar multiplication, and subtraction
// What do you expect the output to be?
cout << M2 - Matrix4f::Ones() * 2.2 << endl;
```

Equality (`==`) and inequality (`!=`) are the only relational operators that work with matrices. Two matrices are considered equal if all corresponding coefficients are equal.

```
cout << (M2 - Matrix4f::Ones() * 2.2 == Matrix4f::Zero())
     << endl;
```

Common matrix operations are provided as methods of the matrix class:

```
// Transposition
cout << M1.transpose() << endl;

// Inversion ( #include <Eigen/Dense> )
// Generates NaNs if the matrix is not invertible
cout << M1.inverse() << endl;
```

Sometimes, we may prefer to apply an operation to a matrix element-wise. This can be done by asking Eigen to treat the matrix as a general array by invoking the `array()` method:

```
// Square each element of the matrix
cout << M1.array().square() << endl;

// Multiply two matrices element-wise
cout << M1.array() * Matrix4f::Identity().array() << endl;

// All relational operators can be applied element-wise
cout << M1.array() <= M2.array() << endl << endl;
cout << M1.array() > M2.array() << endl;
```

Note that these operations do not work in-place. That is, calling `M1.array().sqrt()` returns a new matrix, with `M1` retaining its original value.

Vectors

A vector in Eigen is nothing more than a matrix with a single column:

```
typedef Matrix<float, 3, 1> Vector3f;
typedef Matrix<double, 4, 1> Vector4d;
```

Consequently, many of the operators and functions we discussed above for matrices also work with vectors. The naming convention is also similar (in this case, the size suffix defines the one-dimensional length, rather than the square dimensions).

```
// Comma initialization
v << 1.0f, 2.0f, 3.0f;

// Coefficient access
cout << v(2) << endl;

// Vectors of length up to four can be initialized in the
   constructor
Vector3f w(1.0f, 2.0f, 3.0f);

// Utility functions
Vector3f v1 = Vector3f::Ones();
Vector3f v2 = Vector3f::Zero();
Vector4d v3 = Vector4d::Random();
Vector4d v4 = Vector4d::Constant(1.8);

// Arithmetic operations
cout << v1 + v2 << endl << endl;
cout << v4 - v3 << endl;

// Scalar multiplication
cout << v4 * 2 << endl;

// Equality
// Again, equality and inequality are the only relational
// operators that work with vectors
cout << (Vector2f::Ones() * 3 == Vector2f::Constant(3)) << endl;
```

Since vectors are just one-dimensional matrices, matrix-vector multiplication works as long as the inner dimensions and coefficient-types of the operands agree:

```
Vector4f v5 = Vector4f(1.0f, 2.0f, 3.0f, 4.0f);

// 4x4 * 4x1 - Works!
cout << Matrix4f::Random() * v5 << endl;

// 4x1 * 4x4 - Compiler Error!
cout << v5 * Matrix4f::Random() << endl;
```

As you would expect, matrix multiplication doesn't work with two vectors as the inner dimensions of two nx1 matrices don't match. However, transposing one of the vectors fixes this:

```
// Transposition converts the column vector to a row vector
// This makes the inner dimensions match, allowing matrix
   multiplication
v1 = Vector3f::Random();
v2 = Vector3f::Random();
cout << v1 * v2.transpose() << endl;
```

The linear algebra savvy amongst us probably recognized the last operation as nothing other than the dot product. In fact, vectors have built-in functions for the dot product, and other common operations:

```
cout << v1.dot(v2) << endl << endl;
cout << v1.normalized() << endl << endl;
cout << v1.cross(v2) << endl;

// Convert a vector to and from homogenous coordinates
Vector3f s = Vector3f::Random();
Vector4f q = s.homogeneous();
cout << (s == q.hnormalized()) << endl;
```

And, finally, element-wise operations can be performed by asking Eigen to treat the vector as a general array:

```
cout << v1.array() * v2.array() << endl << endl;
cout << v1.array().sin() << endl;
```

Example: Vertex Transformation

Now that we have a basic understanding of Eigen, let's use it to perform a very common operation in graphics: vertex transformation.

```
#include <Eigen/Core>
#include <Eigen/Geometry>
#include <iostream>

using namespace std;
using namespace Eigen;

int main() {

float arrVertices[] = { -1.0, -1.0, -1.0,
                         1.0, -1.0, -1.0,
                         1.0,  1.0, -1.0,
                        -1.0,  1.0, -1.0,
                        -1.0, -1.0,  1.0,
                         1.0, -1.0,  1.0,
                         1.0,  1.0,  1.0,
                        -1.0,  1.0,  1.0};

MatrixXf mVertices = Map< Matrix<float, 3, 8> > (arrVertices);

Transform<float, 3, Affine> t = Transform<float, 3, Affine>::
   Identity();

t.scale( 0.8f );
t.rotate( AngleAxisf(0.25f * M_PI, Vector3f::UnitX() ) );
t.translate( Vector3f(1.5, 10.2, -5.1) );

cout << t * mVertices.colwise().homogeneous() << endl;
}
```

This example introduces several new Eigen constructs.

To start with, we are using a C++ array to initialize a `MatrixXf` object. The array holds the x, y, and z coordinates of the vertices of a cube. [2] Eigen's `Map` class allows us to perform this initialization.

Next, we create a transformation. You may recall from linear algebra that a transformation is nothing more than a matrix. This is also true in Eigen — the `Transform` class just makes it easier to deal with the underlying matrix representation. You can access the underlying matrix by calling `t.matrix()`.

`Transform<float, 3, Affine> t` creates a 3-dimensional affine transformation with single-precision floating point coefficients. The next three lines apply a uniform scaling, rotation, and translation to the created transform object. In matrix form, this may be written as

$$U = TRSI$$

Where I is the identity matrix.

The rotation is specified as a combination of angle and rotation-axis by using the `AngleAxisf` class. The axis should be normalized, and in most cases we can simply use the convenience functions `Vector3f::UnitX()`, `Vector3f::UnitY`, and `Vector3f::UnitZ()` which represent unit vectors in the x, y, and z directions, respectively.

[2] We could have loaded the array into an array of eight `Vector3f` objects. However, storing it as a matrix is more efficient, and leads to cleaner code.

Finally, we apply the transformation to the vertices of our cube. See how storing the vertices as columns of a matrix allows us to transform all vertices with a single matrix multiplication. To do this, however, the inner dimensions of the transformation matrix, and the vertex matrix have to match. `t` uses homogeneous coordinates, and so a 3-dimensional transformation is represented as a 4x4 matrix. To match these dimensions, we homogenize each column of our vertex matrix by using the `colwise()` method.

Each column of the output represents a transformed vertex:

```
    0.4       2       2     0.4     0.4       2       2     0.4
8.65499 8.65499 9.78636 9.78636 7.52362 7.52362 8.65499 8.65499
1.75362 1.75362   2.885   2.885   2.885   2.885 4.01637 4.01637
```

Further Reading

The Eigen Quick Reference Guide provides a handy reference to most matrix and vector operations: `https://eigen.tuxfamily.org/dox/group__QuickRefPage.html`