CS11 Advanced C++ Lab 2 - Raytracer Kickoff

Ray Tracer!

This term we will go through the process of writing a simple ray tracer. Ray tracers happen to be quite well-suited to implementing in C++, and we will have plenty of opportunities to leverage many of C++'s more interesting features. Also, a larger project like this will give us opportunities to apply other widely used techniques, such as automating the build process and doc-generation, and using a version control system to manage our source code.

This week's lab is pretty simple - you will build a few basic classes that will be central to your ray tracer's operation. Getting these classes nailed down now will allow you to concentrate on the higher level tasks you will be implementing in following weeks. So, it is essential to build good, clean abstractions from the beginning.

One of the nicer features of C++ is that we can specify meanings for the various arithmetic operators we might want to use with our own classes. Although a simplistic approach is to implement operator overloads as member functions on our classes, this isn't always the best idea, and in fact it can be *very* limiting in certain situations. This lab will also give you an opportunity to practice the operator-overloading best practices discussed in class.

Vectors

The vast majority of operations within a ray tracer rely on 3D vectors. The direction of a ray, the surface normal at a particular location on an object, the location of a light in the scene, all of these things and more are represented as vectors. So, one of the first things we will need for our ray tracer is a complete, fully featured implementation of a vector data type.

Vectors are a pretty simple data type, and the math operations that can be performed on vectors are also pretty simple to implement. Since we *know*that our vectors are all going to be 3D, we don't need to dynamically allocate memory for the element values; we can just use a statically sized array of three values. Dynamic memory management in such a critical piece of functionality would destroy performance, so static sizing is the way to go.

Vector elements need to be floating-point values of some sort. It would probably be best to use `float`, since this will be both faster and more space-efficient, but if you want to go all out, you can represent coordinates as `double` values.

Create a class to represent 3D vectors, as outlined above. Make sure to provide a 3-argument constructor for initializing all the elements to specific values, as well as a default constructor that initializes all elements to 0. Of course, copy-construction and copy assignment are both good operations to have too, but C++ will provide them for you automatically. Since your array is fixed-size and not dynamically allocated, the compiler will generate correct versions of these by default.

Coding Style

Since this is going to be a central component of your ray tracer, you should make extra efforts to ensure that the code follows good style, is well commented, and that it uses assertions (defined in the `<cassert>` header) everywhere that inputs or outputs need to be validated. Make sure to do this diligently, so that you can avoid having to do any rework on this lab. This is a requirement for passing this assignment.

Make sure that all classes, every data-member and member-function of each class, and every non-member function are commented. You don't have to give tons of details; just state clearly what each variable or operation is for. Keep in mind that these comments will be used for auto-generated API documentation in a future assignment.

Other Operations to Implement

Here are other operations that your vector class should support. Make sure to follow the operator-overload guidelines discussed in class.

- **Compound assignment operators += and -=.** These should be defined to require a vector on both sides of the operator.
- **Simple arithmetic operators + and -.** Once you have the compound assignment operators defined, these should be simple.
- **Compound assignment operators *= and /= that take a scalar value on the right-hand side.** Since it doesn't make sense to write `scalar *= vector`, you only have to support `vector *= scalar`, etc. Also, you only need to support scalars that are the same data-type as your vector's elements. For example, if your vector uses `float` values, you should only support multiplication or division with `float` scalar values.

  (It also doesn't make sense to divide by zero, so make sure to catch that situation with an assertion.)
- **Simple arithmetic operators * and / for vector/scalar combinations.** Implement the * operator to support "vector * scalar" and "scalar * vector" operations. Only "vector / scalar" makes sense, so you only have to implement that version for division. Of course, since multiplication is commutative, and since you just finished *= and /=, you should be able to implement these very quickly and easily.
- **The unary minus operator.** This operator is exactly like the simple arithmetic operators, except that there is only one argument. It

should also return a const-object. For vectors, this operator should effectively multiply the vector by -1. *(Hint hint...)*

- **You should also provide support for getting and setting the individual elements of your vector.** You can do this by implementing either the `[]` (brackets) operator or the `()` (parentheses) operator, but don't do both; that will just be confusing. Make sure to implement one version for use in RHS expressions, and another version for use as the target of an assignment. (If you only implement a read/write version, you can't use `const` modifiers in your program, so you really need to provide both versions.)
- **Implement the stream-output operator << for your vector class.** Choose a form that is clean and simple to parse, such as **`(x, y, z)`**. In a future lab, you will also implement the stream-imput operator >>, and you will want to support the same format that you produce for stream-output. So make your life easier!

Finally, you need to provide several other operations, but these won't be implemented as operator overloads. The rationale for this is given below.

- **You need to implement dot product and cross product functions.** The result of the dot product is going to be a scalar value, which should be pretty simple to compute. The result of the cross product will be another vector, so that function will operate in a way very similar to the simple arithmetic operators + and -.

  (You can look on MathWorld if you need a reminder of how the dot product works. See equation (8). There is also a page on the cross product. Equation (1) or (2).)

  It's a bit ambigous, whether you should overload (vector * vector) to operator to compute dot products or cross products, since either one *kinda* makes sense. If you want to stick with functions or member-functions, that is fine. You might try assigning * to dot-product and % (the remainder operator) to cross-product, but *document what you choose!* Ray-tracing arithmetic mostly uses dot-products, so implementing * as dot-product can definitely make your life easier.
- **You need to provide functions to report a vector's magnitude.** These should be member functions. Write one function that produces the magnitude-squared, and a second function that produces the magnitude. Many times you simply need the magnitude-squared, so you don't want to incur the performance hit of that square root operation.

  The square-root functions are in the `cmath` header. If you are using doubles, you can use the `sqrt()` function, which takes and returns doubles. Or, if you are using floats, you can use the `sqrtf()` function, which takes and returns floats.

  (You might also want to leverage the fact that the dot-product of a vector with itself is equal to its magnitude-squared...)
- **You should provide a member function to normalize a vector.** That function should be very easy to write, given everything else here.

There will be more vector math operations to implement in following weeks, but this should be sufficient for now.

Colors

Your ray tracer will also need a class to represent colors, since the whole point of ray tracing is to determine the color of each pixel in the image. **Create another class to represent colors in the RGB color-space.** That is, each color will have a red component, a green component, and a blue component. Each color value should be represented by a floating-point number. (Again, we recommend `float`, but you can use `double` if you don't care as much about efficiency.) Normally these values will be between 0 and 1, but in some cases they may go outside that range when particular values are being computed.

For data members, you might want to create an array of three floats or doubles, like before, but you might find it clearer to create three separate data-members named "red", "green" and "blue". Do what is clearest to you, but document what you do. For example, if you use an array of values, document what color component is at each array index.

Here are the operations that your color class should support:

- **Provide accessors and mutators for each component of your color.** You can use the `[]` or `()` operators if you really want to, but in this case it would be better to indicate what color each component is, in the accessor and mutator names. For example, `get_red()` and `set_red(float val)`, etc.
- **Compound assignment operators +=, -=, and *=, for two colors.** Adding, subtracting, or multiplying two colors is simply a component-wise operation; red + red, green + green, and so forth. (Adding color objects is equivalent to combining colors together. Multiplying color objects is equivalent to filtering one color by another color.)

  Also provide corresponding simple arithmetic operators for these operations, where there is a color on both the LHS and RHS.

  We won't need division of one color by another. :-)
- **Compound assignment operators *= and /= for multiplying and dividing a color by a scalar.** The type of the scalar should be the same as the type of your color elements. Each component of the color is multiplied or divided by the scalar.

  Provide corresponding simple arithmetic operators for this operation as well. Make sure to support (color * scalar) and (scalar * color). As before, you only need to support (color / scalar), since (scalar / color) makes no sense.

- **Implement the stream-output operator << for your color class.** Choose a form that is clean and simple to parse, such as **`(r, g, b)`**. In a future lab, you will also implement the stream-imput operator >>, and you will want to support the same format that you produce for stream output.

Testing Your Code

It will take a while before we have enough components to build a fully functional ray tracer, but in the meantime we want to be sure our code is actually correct. To this end, we will employ *unit testing* to exercise the individual operations of our classes. There are several C++ unit-testing frameworks available, but we will use the Google C++ Testing Framework this term. You can download the version 1.5.0 source code from the googletest page, and unpack the archive. Then you can get ready to test your code!

Expand the gtest archive into a subdirectory of where you are writing your raytracer code; you should end up with a `gtest-1.5.0` directory. Then, go into the `gtest-1.5.0/make` subdirectory, and type `make` to build the googletest library. (We will cover `make` in a subsequent lecture.) If everything goes fine, you should end up with a few files, but most importantly, `gtest-all.o` should now be in this directory.

Next, you can go back to the directory containing your raytracer classes, and begin writing unit tests for your code. You will want to create one testing source file for each class you created, to hold the tests for that class. Generally you will keep the tests for different classes separate. The testing source-file will follow this template:

```
#include "myclass.hh"
#include <gtest/gtest.h>

// This is the namespace that the Google C++ Testing Framework uses.
using namespace testing;


TEST(TestMyClass, FeatureOne) {
    ... // Code to exercise the feature
}


TEST(TestMyClass, FeatureTwo) {
    ... // Code to exercise the feature
}


int main(int argc, char **argv) {
    InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The first name in the `TEST()` macro arguments is the test case name, and the second name is the individual test name. (The Google testing framework doesn't follow the more widely used naming convention; Google's "test case" is really a test suite, and Google's "test" is a test case.) The names shouldn't have any spaces.

Within each test, you can write whatever C++ code you want to set up for your tests, and you can also use various testing macros to check different conditions. There are `ASSERT_xxxx()` operations which will test various conditions; if the condition is not true then the entire suite is halted with a failure. There are also `EXPECT_xxxx()` operations, which will report a failure but will continue running the remaining tests. You are encouraged to use the `EXPECT_xxxx()` versions, so that all bugs can be identified in one pass.

Probably the two most important kinds of tests for your classes will be `EXPECT_EQ(a, b)` for *non-floating-point* values, and `EXPECT_FLOAT_EQ(a, b)` (or `EXPECT_DOUBLE_EQ(a, b)`) for *floating-point* values. Since floating-point computations are often inexact, the floating-point comparisons will ensure that the actual value and expected value are within some epsilon of each other.

You can also write tests with custom failure messages, like this:

```
TEST(TestMyClass, FeatureOne) {
    Vector3F v = ... ;
    ASSERT_FLOAT_EQ(v.length(), 3) << "length is wrong!";
}
```

In other words, if the test fails, it will return an object that supports stream-output.

Building your Tests

You should shoot to have your unit tests cover every major function of your classes; you should definitely hit all complicated operations, such as cross-products, dot-products, clamping colors, operators, etc. The more complete your test code is, the more confidence you will have moving forward on subsequent tasks.

When you are finished creating your unit-testing code, you can build it with a command like this:

```
g++ -Wall -I gtest-1.5.0/include gtest-1.5.0/make/gtest-all.o test_myclass.cc -o test_myclass
```

The `-I` argument tells `g++` to look for header files in the specified path, so that it can resolve the `#include <gtest/gtest.h>` statement. The `gtest-1.5.0/make/gtest-all.o` file must be included so that the googletest code can be linked into your unit-testing code.

At this point, the process is pretty clunky, but we will refine it as we progress through the project.

Additional Information

There are many other things you can do with the Google C++ Testing Framework! To learn more, you can read the online documentation. For example, there are many other kinds of testing functions you can use in your tests.

Extra Credit

Here are some additional things you can do if you want to make your data types even cooler:

- Instead of using a class, make your vector datatype a class-template, parameterized on both the dimension and the element-type of the vector. Then, provide `typedef`s for commonly used configurations of the template - perhaps `Vector3D` for a 3D vector of doubles, `Vector3F` for a 3D vector of floats, etc. Make sure that all of your math operations also work properly on these templates.

- Implement a class (or a class-template) for square matrices of fixed dimension (i.e. don't dynamically allocate memory!). Provide all of the standard useful matrix operations, such as matrix multiplication, multiplication by a scalar, etc. To make it really useful, you should also provide a matrix inversion operation for orthogonal matrices (but you should assert that the matrix really is orthogonal!) - note that the inverse of an orthogonal matrix is simply its transpose. Finally, don't forget to provide operations to transform a vector with a matrix; you could overload the multiplication operator for this, for example.
