

# Assignment 2: Algebra/Camtrans

Linear Algebra Helpsession: Thursday, September 24, 7:00 PM

Helpsession: Tuesday, September 29, 7:00 PM

Algo Due: Tuesday, September 29, 5:00 PM

Asgn Due: Wednesday, October 7, 11:59 PM

## 1 Introduction

Having completed the Shapes assignment, you are now intimately familiar with how three-dimensional objects are created and displayed on the screen. However, you may have noticed that the last assignment overlooked something important. . . the program just displayed a shape in the center of the screen. And the shapes were always of the same size. And the rotation was a hack thrown in by the TAs to aid you in completing the assignment correctly. Face it — you need some way of distorting and moving (*transforming*) your shapes so that you can actually *use* them for something other than decorating your screen. In this assignment *you* will be writing the tools necessary to move, rotate and scale your shapes. These tools will manifest themselves as a fully-functional linear algebra package, capable of operations on matrices, vectors, and points, as well as providing functions for creating matrices that possess special properties.

Yes, this handout is long. Yes, this assignment is actually two assignments in disguise. But fear not, for you will have enough time for both! We have combined them to make your life easier and your planning better. We want to be clear up front, however, that this assignment has the potential of being quite rough, so take it seriously and start **early!**

## 2 Project Overview

Unlike other assignments in CS123, there are two separate source folders for this assignment which correspond to Algebra (`/course/cs123/asgn/algebra`) and Camtrans (`/course/cs123/asgn/camera`) respectively. Since your Camtrans (as well as future assignments) makes heavy use of your linear algebra package, it is important to start off working on Algebra first, and only continue on to Camtrans once your Algebra package is really solid. Note: it is important that you keep the `algebra` and `camera` support code in separate directories.

### 2.1 Algebra

There is no demo for Algebra, but we have provided you with a “thorough” and verbose tester. To use the tester, simply type `make` as usual and then run the executable, `./tester`. When you copy over the linear algebra folder from `/course/cs123/asgn/algebra`, it will already contain a compilable skeleton for all of the matrix/vector/point functions you will need to write. Essentially, the tester will include these implementations and compare their results to the TAs implementation on a wide variety of test cases. The tester firsts checks your `operator==` and `operator!=` functions because it relies on them working correctly in later tests. Realize, however, that floating point discrepancies might cause the tester to report a failure even when your result is “correct.”

## 2.2 Templates

Templates are C++'s version of Java's generics and allow you to write generic container/algorithm code once and have it work with a variety of different types of objects by not hard-coding those types a priori. If you've used the STL before, you may have noticed that it makes heavy use of templates. Specifically, you will be implementing a CS123Matrix class which is templated on both its dimensions ( $M \times N$ ) and the type of the elements it contains. You will also be implementing a CS123Vector class which will similarly be templated on the length of the Vector ( $N$ ) and the type of the elements it contains. Note: don't confuse the Vectors you will be implementing in this assignment (CS123Vector) with STL vectors (`std::vector`), as they are completely unrelated.

C++ Template syntax is a bit awkward, even for experienced C++ programmers, but you really don't have much to worry about. We've taken care of all of the template syntax and skeletons for functions which use templates (including examples), and all you have to do is fill in those functions in the generic case (ie. for an  $N$ -length CS123Vector instead of hardcoding a 4-length CS123Vector). For instance, the difference between implementing Vector addition with and without templates would be just replacing a constant with the letter  $N$  everywhere you're talking about the length of the Vector. The difference, however, comes in that with templates at the end of the day, you'll have a high quality linear algebra package which has been abstracted such that you can reuse your implementation for matrices and vectors of arbitrary sizes.

We cannot stress enough that a good Algebra package will be crucial too all of the ensuing assignments in CS123. *Make sure that you pass all of the test cases with the provided tester before continuing on to Camtrans.*

## 2.3 Camtrans

In Camtrans, you will be writing a Camera that will build off of your (already rock solid) linear algebra package to generate all of the transformation matrices that go into the normalization transformation (see Viewing II and Viewing III lectures) necessary for telling OpenGL explicitly how to project points (and hence lines, and hence triangles) from the camera's view of world space to the 2D film plane. These transformations are parameterized by the camera specifications which you'll be able to edit and play with to test out your camera in the accompanying GUI. See the Camtrans sections below for more information.

## 3 Algebra : Requirements

Your task for this assignment will be to write a complete linear algebra package. You will be using what you write in this assignment for most of the remaining assignments, so it will be important that you really put your program through the paces and work out as many bugs as you can.

Your linear algebra package will need to have/support the following:

- A templated CS123Vector class (CS123Vector< $N$ ,  $T$ >)
- A templated CS123Matrix class (CS123Matrix< $M$ ,  $N$ ,  $T$ >)
- Addition:
  - Vector+Vector
  - Matrix+Matrix
- Subtraction:
  - Vector–Vector
  - Matrix–Matrix
- Multiplication (including mirrored versions):

- Vector\*scalar
- Matrix\*scalar
- Matrix\*Vector
- Matrix\*Matrix
- Dot and Cross products of vectors
- Negation of vectors
- Normalization of vectors
- Homogenization of points
- Magnitude (length) of vectors
- Matrix transpose, determinant, and inverse
- Special matrix creation:
  - Scale matrix
  - Translation matrix
  - Rotation about  $X$  axis
  - Rotation about  $Y$  axis
  - Rotation about  $Z$  axis
  - Rotation about axis formed by arbitrary vector and point
- Inverse matrices for the above six special matrices

Also, to make the library easier to use, you will be utilizing operator overloading, one of the most useful (as well as most dangerous) capabilities of C++. Operator overloading allows you to take a standard operator (`==`, `+`, `-`, `*`, etc.), and give it special meaning when used with your classes. Using operator overloading, you can say something like

```
tempVector2 = tempMatrix*tempVector1;
```

when you wish to multiply a vector by a matrix. Otherwise, you would be forced to write something like

```
tempVector2.multiply(tempMatrix, tempVector1);
```

Obviously, the former is much more readable.

We will be providing you with skeleton code files for these methods/classes. Trust us, using operator overloading and the provided skeletons along with Templates will make your life much easier in the long run.

## 4 Algebra : Support Code

Support code for the Algebra part of this assignment is located in `/course/cs123/asgn/algebra`. You should copy this folder over locally and keep it separate from the Camtrans part of this assignment. Within this folder, you will find several source (`.cpp`) and inline (`.inl`) files with a large number of function skeletons for you to fill in. Some of the less important functions have already been filled in for you as examples of how to work with the classes and templates. Every function which you must modify is denoted by a `/// @TODO` comment so you can search for that string to find what is left. You will need to edit all of the `.cpp` and `.inl` files given.

At first, the layout of the provided support code may seem confusing; you might even think that there are some files missing from `/course/cs123/asgn/algebra`, but we assure you that you have all of the files you will need to complete your linear algebra package, and the files that are there have been very carefully chosen. Specifically, you will be given the following files:

- `CS123Vector.inl`  
An "inline" file which is included directly from `CS123Vector.h` and includes all of the templated methods with respect to the `CS123Vector` class. The reason for not having a `.cpp` file containing these definitions is unfortunately rather complicated (and not really important). The gist of it is that whenever you use a C++ template in another class after including `CS123Vector.h`, for instance, the implementation in addition to the prototype/definition must be known at compile-time. This is a special case for C++ templates, and no really good way to get around it. It is very common to put all of the template implementations in addition to their definitions into the header file directly, but we have decided to go with a somewhat cleaner approach and separate out the definitions and implementation into `.h` and `.inl` files respectively, and instead just include the `.inl` file in the `.h` header directly. If this doesn't make complete sense, don't worry about it; it's not at all necessary or important to understand how this C++ quirk works to complete this assignment.
- `CS123Matrix.inl`  
Contains skeletons for all of the templated function of the `CS123Matrix` class.
- `CS123Matrix.cpp`  
Contains skeletons for the global functions which create special-purpose transformation matrices and their inverses (`getTransMat`, `getScaleMat`, `getInvTransMat`, etc.).
- `CS123Point.cpp`  
Contains one function, `homogenize` which takes a point in 3-dimensional homogeneous space ( $x, y, z, w$ ) and normalizes it such that the resulting point will have  $w=1$ . Note that this function is not templated so it appears in a `.cpp` file.

All of the corresponding header files are located in `/course/cs123/include/algebra`. Before you start filling in these provided stencils, make sure you have a thorough look through the header files to get an idea of what's going on overall. `/course/cs123/include/algebra` also includes a convenience header file, `CS123Algebra.h`, which `#includes` all of the header files in the `CS123Algebra` library (`CS123Vector.h`, `CS123Matrix.h`, and `CS123Point.h`) so you can just include this one header file as opposed to having to always include the vector, matrix, and point header files explicitly.

You'll notice that some of the definitions talk about a `Point3` type, but we haven't mentioned `Points` anywhere in the handout so far. Having separate `Point` and `Vector` representations can be very useful when working with code to differentiate semantically between when you're talking about points in space versus vectors/directions, but at the end of the day, they have pretty much the same functionality. Therefore, in your linear algebra package, **Points are typedefed as `CS123Vectors`**. Specifically, this means that a `Point3` is just another name for a `CS123Vector<4, double>`. Hold on a minute you say... why is a `Point3` equivalent to a `CS123Vector<4, double>`? Shouldn't it be a `CS123Vector<3, double>`? The answer is that we're talking about a point in 3-dimensional, homogeneous coordinates, where there is an implicit fourth coordinate,  $w$ . To make it clear that we're talking about a `Point` in 3-space, we've decided to call it a `Point3`, even though it contains 4 elements. Make sure you understand the definition of `Point3` in `CS123Point.h`, and remember that it has the same functionality as a `CS123Vector` of length 4 ( $x, y, z, w$ ).

## 5 Algebra: Notes and Examples

In this section, we point out several important notes that will be critical to understanding this assignment:

- `CS123Vector` stores its internal data as a contiguous array of  $N$  elements of type `T`, where  $N$  and `T` are template parameters. Just like how function parameters can have default values in C++, template parameters can also have sensible defaults set so the user doesn't have to specify them explicitly. For `CS123Vector`, its length,  $N$ , defaults to `4`, and its element type, `T`, defaults to type `double`. Take the following example:

```
CS123Vector< 4 > myVector;
```

The template parameter `T` defaults to type `double` here, so `myVector` is a `CS123Vector` containing 4 doubles.

- `CS123Matrix` stores its internal data as an `M`-length array of `CS123Vector<N, T>`s. This declaration looks like:

```
CS123Vector< N, T > rows[M];
```

This tells us two things. First, there are a total of  $M * N$  elements of type `T` in a `CS123Matrix<M, N, T>`, where `M` denotes the number of rows and `N` denotes the number of columns. Second, the data is stored in row-major format. This is generally more intuitive than column-major format, especially in graphics where 99% of images are stored in row-major format. Having our internal data stored in row-major format also allows us to override the “`[]`” operator for `CS123Matrix` to return a reference to the `CS123Vector` at the specified row. For example:

```
CS123Matrix< 4, 4, int > myMatrix;  
myMatrix[1] = CS123Vector< 4, int >(1, 2, 3, 4);
```

In the first line, we’re declaring a 4x4 `CS123Matrix` comprised of `ints` called `myMatrix`. In the second line, we’re setting the second row of `myMatrix` to the `CS123Vector` specified.

- You’ll see references to `Matrix4x4`, `Vector4`, and `Point3` around the stencil (and in the second part of the assignment, `Camtrans`). These didn’t come out of nowhere – they are just `typedefs` of the most common matrices/vectors that you’ll be using in `CS123`. Specifically, they have the following definitions:

```
typedef CS123Matrix< 4, 4, double > Matrix4x4; // (CS123Matrix.h)  
typedef CS123Vector< 4, double > Vector4; // (CS123Vector.h)  
typedef CS123Vector< 4, double > Point3; // (CS123Point.h)
```

These are provided for convenience purposes and to (optionally) shield you from having to work with templates in future assignments, by just using these `typedefs` instead. Recall that a `typedef` is simply an alias, ie, another name for the same type. So anywhere you see `Matrix4x4` in your implementation or in future assignments, you could substitute `CS123Matrix<4, 4, double>` and it would be completely equivalent.

**Important:** The `tester` will only test `Matrix4x4`, `Vector4`, and `Point3`. That is, it will not check your implementation in all of its templated generic glory. These three concrete special cases of the more generic templated `CS123Matrix` and `CS123Vector` classes will be the most important to future assignments (including `Camtrans`), so feel free to tailor your templated implementation of certain methods in `CS123Matrix` and `Vector4` towards the homogeneous 3-space case (where  $M = N = 4$ ). Along these lines, though we encourage you to keep your implementations as generic as possible with respect to the template parameters, **you are not required** to implement the general case (arbitrary `M` and `N`) for `CS123Matrix::getDeterminant()` and `CS123Matrix::getInverse()`.

You will not lose any points for hardcoding the 4x4 case for these functions and can safely assume that you won’t ever see any other `M` or `N`. All other functions which don’t explicitly say so in the stencil, however, expect you to implement `CS123Matrix` and `CS123Vector` generically with respect to `M` and `N`.

- Some of the `CS123Matrix` functionality only makes sense for square matrices (`getDeterminant`, `getInverse`). In your implementation, you can assume that these functions will only be called with square matrices. You may also assume that `Matrix * Matrix` multiplication will only be called with square matrices.

- Both CS123Vector and CS123Matrix (also Point3 because, again, a Point3 is a CS123Vector), all have an overloaded “<<” which is already implemented for you in the stencil code. Unlike C which mainly uses `printf` to print out debugging information to the shell, in C++ it’s much more common to use `std::ostream` which you’ve probably already seen. Basically, they allow series of builtin and user-defined types to be printed out to the shell; for example:

```
cout << myMatrix << endl;
```

This example assumes that we’ve already included the `ostream` header and have already put a `using namespace std;` in our code so the compiler will find that `cout` is actually referring to `std::cout` and respectively for `endl` and `std::endl`. This example will print out the `myMatrix` variable (which we previously defined) to standard out (most likely the shell).

- OpenGL uses column-major matrices by default with respect to its transformation matrices, so in the Camtrans part of this assignment, when you have to take one of your CS123Matrices and tell OpenGL to use it as the projection matrix, you will have to transpose your row-major matrix first to turn it into the column-major format that OpenGL will be expecting.

## 6 Algebra : Language Issues

What you are writing for this assignment is meant to run very fast and be intuitive to use. As such, you will be using many features of C++ that you may not be completely familiar with. These features include `const` functions/parameters, passing by reference, operator overloading, and, as previously mentioned, templates.

### 6.1 Constant functions/parameters

Constant functions and parameters are not only useful for writing clean code, but can also take advantage of by the compiler so that *faster* code can be generated. Here are some guidelines for using `const`:

- A method of a class should be declared `const` if possible, because if you have a `const` instance of a class, only `const` methods may be called.
- A parameter to *any* function should be declared `const` if possible, as you may wish to pass a `const` variable or class instance to that function at some point.

In summary, use `const` whenever it is possible to do so.

### 6.2 Passing by reference

Passing a parameter *by reference* is a very simple technique which impacts both efficiency and functionality. To specify that you wish to pass a parameter by reference, simply place an ampersand (&) after the parameter type. For example, `int foo(int bar)` passes `bar` by value, whereas `int foo(int& bar)` passes `bar` by reference.

Passing a parameter by reference has the following effects:

- Normally, if you modify a parameter passed to a function, that change is local to that function. However, if you pass a parameter by reference, modifying the parameter *modifies the variable passed to the function*, so the effects are visible outside of the function.
- If you pass to a function an instance of a structure/class by value, a bitwise copy is made of the structure (unless a copy-constructor has been defined). This has two drawbacks:
  - 1. Unless a proper copy-constructor has been defined, when the function exits and the destructor is called on the structure/class, data may be lost. (i.e., by freeing a pointer in the duplicate structure that was needed in the original)

- 2.Regardless of whether or not a copy constructor has been defined, passing an instance often enough takes a non-trivial amount of time.

When you pass by reference, these problems are avoided, since a copy of what is being passed in is not being created.

It is also possible to return a reference to an object.

There may be times when you want the advantages of passing by reference, but don't want the disadvantages (i.e., you want to make sure you don't accidentally modify the parameter). In this case, you can use a `const` reference.

### 6.3 Operator Overloading

Operator overloading allows you to use the standard operators to perform specific actions that you define. You can overload an operator several times for use with different data types. Operator overloading is wonderful for making code simple and easy to read and write. One must be cautious as it is also very easy to go too far, creating code that may be very terse, yet is unreadable because the meanings of the operators have been obfuscated.

As you saw above, overloading an operator is as simple as declaring a function whose name is `operator $symbol$ ()`. The arguments to the function are the arguments to the operator. For example, the function `MyClass& operator+(const MyClass& a, const MyClass& b)` will define the `+` operator for the following situation:

```
MyClass foo, bar;  
.  
.  
MyClass quux = foo+bar;
```

You will often wish to use `const` references when overloading operators, as you generally are dealing with classes.

### 6.4 Floating Point Equality

When writing the `==` and `!=` methods, it is very important to remember that when using floating point values, they are hardly ever exactly the same. Sometimes, however, we would like to consider them equal if they are very close. Therefore, when comparing points or vectors, we must allow a small amount of error. Think about how `fabs()` might be useful for this, along with `EPSILON`, which is defined in `CS123Common.h` ... To convince yourself of the importance of using a tolerance, try the following code:

```
#include <stdio.h>  
#include <iostream.h>  
  
int main(int argc, char ** argv) {  
    float a=10000;  
    float b=1;  
    b=b/3.0;  
    a=a/3.0;  
    a=a/10000.0;  
    cerr << "a: " << a << " b: " << b << endl;  
    if (a==b)  
        cerr << "equal!" << endl;  
    else  
        cerr << "not equal!!!" << endl;  
}
```

## 7 Algebra : Extra Credit

Extra Credit will be given for using quaternions to make your code faster. Arbitrary rotation matrices can be constructed particularly quickly by understanding the mapping between quaternions and rotations. Do some research to figure out how you can use quaternions to construct rotation matrices quickly.

There are many ways to optimize the general matrix inversion function past Gauss Jordan Elimination. However, future assignments will use this function infrequently so there is no real need to go crazy with optimization on general invert. Implementing matrix inversion and determinants in the generic  $M \times N$  case instead of hardcoding them for the  $4 \times 4$  case will also earn you extra credit (note that the provided tester will only test your  $4 \times 4$  implementation).

`CS123Vector` contains two functions which you can take a stab at for extra credit, namely `reflectVector`, and `refractVector`. These are fairly difficult and have very special edge cases; in particular, if their function signatures and arguments don't make sense to you a priori, you might be better off shying away from them, at least for the time being. They will, however, come in handy for your Raytracer down the road.

When you write your raytracer, all of these operations will be called thousands and thousands of times, so the faster the better!

## 8 Camtrans : Introduction

When rendering a three-dimensional scene, you need to go through several stages. One of the first steps is to take the objects you have in the scene and break them into triangles. You did that in assignment 1. The next step is to place those triangles in their proper position in the scene. Not all objects will be at the "standard" object position, and you need some way of resizing, moving, and orienting them so that they are where they belong. You are starting to handle that problem with the linear algebra package you wrote. There remains but one important step: to define how the triangulated objects in the three-dimensional scene are displayed on the two-dimensional screen. This is accomplished through the use of a *camera transformation*, a matrix that you apply to a point in three-dimensional space to find its projection on a two-dimensional plane<sup>1</sup>. While it is possible to position everything in the scene so that all the camera matrix needs to do is flatten the scene, the camera transformation usually incorporates handling where the camera is located and how it is oriented as well.

Now it is entirely possible to simply throw together a camera matrix which you can use as your all-purpose transformation for any three-dimensional scene you may wish to view. All that you would have to do is make sure you position your objects such that they fall within the standard view volume. But this is tedious and inflexible. What happens if you create a scene, and decide that you want to look at it from a slightly different angle or position? You would have to go through and reposition everything to fit your generic camera transformation. Do this often enough and before long you would really wish you had decided to become a sailing instructor instead of coming to Brown and studying computer science.

For this part of the assignment, you will be writing a subclass of `CS123Camera`, which is an object that provides all the methods for almost all the adjustments that one could perform on a camera. `CS123Camera` represents a perspective transformation. It will be your job to implement the functionality behind those methods. Once that has been completed, you will possess all the tools needed to handle displaying three-dimensional objects oriented in any way and viewed from any position.

## 9 Camtrans : Applets

The Brown Exploratories Project has created applets for visualizing the perspective and parallel transformations. They are available at:

<http://www.cs.brown.edu/exploratories/freeSoftware/>

---

<sup>1</sup>Actually, projection is the step following clipping, but as you know from reading the lecture slides, our particular matrix makes both steps very simple. In any case, clipping and projection are handled for you.

Be sure that you have configured your browser to run the appropriate Java plugin (including Java3D support), otherwise, the applets probably won't work (try them first; if they don't work, see the Exploratories website for help).

## 10 Camtrans : Demo

When you start camtrans, you will see the control panel and display. The display will show a vicious monster (courtesy of Marshall Agnew) being rather uncomfortably skewered by a set of axes. The  $X$  axis is green, the  $Y$  axis is blue, and the  $Z$  axis is red. These axes represent the normal or scene's coordinate space.

The Demo uses the following defaults for the camera's initial state. The near clip plane is 1 and the far clip plane is 30. The vertical view angle is 60 degrees and the screen aspect ratio is 1:1. The eye of the camera is at (2,2,2) the camera is looking at the origin, and its up vector is (0,1,0).

The translate and rotate buttons modify the camera's position, though relative to a virtual set of axes (a set of axes different from the ones shown in the scene), which represent the camera's "local" coordinate space. In the camera's coordinate space, the camera is located at (0,0,0), the camera is looking along the negative  $Z$  axis, the  $Y$  axis is pointing upwards, and the  $X$  axis is pointing to the right. To emphasize the difference between the normal coordinate space and the camera's "local" coordinate space, these axes are usually referred to as  $W$ ,  $V$ , and  $U$ , respectively. When you move the camera, this set of axes move with it. The four buttons at the bottom of the control box, on the other hand, set the camera's position and orientation relative to the visible axes (the big red, green, and blue ones). The " $FOO$ -axis" buttons position the camera so that it is located two units along the  $FOO$  axis, pointing towards the origin. The "Axonometric" button positions the camera such that it is located at the point (2,2,2) and is pointing towards the origin.

The last two parameters, "Height angle" and "Aspect ratio," control the shape of the viewing *frustum*. The "Height angle" slider interactively adjusts the height angle of the camera; similarly for the "Aspect ratio" slider (recall that width = height\*Aspect ratio). However, the two controls are interdependent: certain combinations of Height angle and Aspect ratio are illegal in the sense that they result in angles larger than 180°. In these cases, the GUI disallows these combinations; to see this, try different combinations of the sliders and see how the display responds.

## 11 Camtrans : Requirements

For this assignment you must implement a fully functional subclass of `CS123Camera`. This involves the following:

- —Maintaining a world-to-film matrix that implements the perspective transformation.
- —Setting the camera's absolute position/orientation given an eye point, look vector, and up vector.
- —Setting the camera's height angle and aspect ratio.
- —Translating the camera **in world space**.
- —Rotating the camera about one of the axes **in its own virtual coordinate system**.
- —Setting the near and far clipping planes.
- —And having the ability to, at any point, spit out the current eye point, look vector, up vector, height angle, aspect ratio, or world to film matrix.

## 12 Camtrans : Support Code

This is the last assignment where you will receive more than the minimal support code, so, starting with the next assignment, everything will be done the way you want it.

Camtrans requires that you use your code from Algebra. It is important to note that for all the assignments to follow you are expected to build on or use code from previous assignments. Make sure your Algebra works before you start on this section.

Here is the class declaration for `CS123Camera`:

```
class CS123Camera {

public:

    CS123Camera() {};
    virtual ~CS123Camera {};

    virtual Point3 getPosition() const = 0;
    virtual Vector4 getLook() const = 0;
    virtual Vector4 getUp() const = 0;
    virtual double getHeightAngle() const = 0;
    virtual double getAspectRatio() const = 0;

    virtual void putWorldToFilm()=0;

    virtual void orientLook(const Point3& eye,
        const Vector4& look,
        const Vector4& up)=0;

    virtual void setHeightAngle(float degrees)=0;
    virtual void setAspectRatio(float aspect)=0;
    virtual void translate(const Vector4 &v)=0;
    virtual void rotateX(const float degrees)=0;
    virtual void rotateY(const float degrees)=0;
    virtual void rotateZ(const float degrees)=0;
    virtual void setClip(float near, float far)=0;
};
```

Many of these methods are very self explanatory. `getPosition()`, `getLook()`, `getUp()`, `getHeightAngle()`, and `getAspectRatio()` return the data described by their name (`getHeightAngle()` should return its answer in degrees). `orientLook` orients the camera such that it is located in the specified position, the up vector is as specified, and the camera is pointed along `look`. `setHeightAngle()` and `setAspectRatio()` do exactly that, and the `translate` and `rotatefoo` methods translate and rotate the camera in its own coordinate space, respectively. `setClip()` sets the positions of the near and far clipping planes. The `putWorldToFilm()` method will require you to write a few lines of OpenGL code to set the Projection Matrix, so it has its own section below.

You will need to modify the Makefile and the file `main.cpp` so that it uses your subclass of `CS123Camera` instead of the filler version that we give you (`CS123DummyCamera`, which does a pretty good job of making the poor beast look pretty bad).

## 13 Camtrans : Projection Matrix

You will need to insert the World to Film matrix yourselves using a few OpenGL commands. OpenGL requires two separate transformation matrices to place objects in their correct locations. The first, the ModelView matrix, positions and orients your camera relative to the scene, or if you prefer, orients the world relative to your camera. In any case, it is taken care of by support code in this assignment. The second, the Projection matrix, is responsible for projecting the world onto the film plane so it can be displayed on your screen. In Camtrans, this is your responsibility. More specifically, in your `putWorldToFilm` function in the Camera, you must make the appropriate calls to configure the Projection Matrix.

The method `putWorldToFilm()` is called frequently by the support code, so you should cache the matrix rather than recomputing it needlessly every time it is called. Here is some sample code to help you out...

```
// tell GL that future matrix operations should be applied to the projection matrix
glMatrixMode(GL_PROJECTION);

// give a pointer to an array of doubles containing the matrix to load
// the matrix pointed to will become the new projection matrix
glLoadMatrixd(pointer_to_first_elt_of_double[16]_array);
```

For more information `man glMatrixMode` and `man glLoadMatrix` in a shell. Please do not use any other library camera functions, we want to see your camera and math functions in action. Remember, `GL_PROJECTION` is only for the camera transformation matrix. Other parts of an object's transformation do not belong here.

## 14 Camtrans : Extra Credit

For extra credit, try implementing an orthographic camera (pretty simple), or an oblique camera (a bit harder). Of course, both of these will require some additional work in terms of user interface...

## 15 Algebra/Camtrans: Handing In

**Important** To hand in your assignment, type `make handin twice`: once from within the directory containing your linear algebra package, and once again from within the directory containing your camtrans implementation.

**Linear Algebra Helpsession: Thursday, September 24, 7:00 PM**  
**Helpsession: Tuesday, September 29, 7:00 PM**  
**Algo Due: Tuesday, September 29, 5:00 PM**  
**Asgn Due: Wednesday, October 7, 11:59 PM**