Homework Guidelines

CS 200

Fall 2015

The code that you turn in for your homework assignments is graded according to several criteria. These criteria may be broken up roughly into three categories: quality, correctness, and efficiency. These are described in more detail below.

All code must be your own. You may verbally *discuss* the homework assignment with others. However, you may not write code for others, and you may not copy the work of others. In particular, you may not make your code available to others to copy or alter. Instructors become especially incensed when they come across code submissions that, aside from variable names and comments, are identical! Receiving a poor grade on an assignment because you did not have time to finish it is better than getting a zero for the course!

Code quality

- Every file that you submit must have a heading that contains (1) your name, (2) the assignment number, (3) the course name (CS 200), and (4) the term (Fall 2015).
- Code that you submit should compile without warnings or errors. Make sure that you test your code out on more than one compiler. Here are some common issues.
  - The function names `sqrtf`, `sinf`, and `cosf` are **not** standard C++ function names. You should include the header file `cmath` and use the (overloaded) functions `sqrt`, `sin`, and `cos` instead.
  - There are several versions (overloads) of the `abs` function. The version that takes an *integer* argument is declared in the header

file `cstdlib`. Versions that take a floating point argument are declared in `cmath`.

- Files should be named *exactly* as stated in the assignment handout. Pay particular attention to capitalization. E.g., for the first assignment you will turn in a file named `Affine.cpp`. This is *not* the same as `affine.cpp`.

- Code should be reasonably organized and commented. Many functions that you will implement are straightforward and require only a few lines of code. You do not need to provide comments in this case. However, if the code that you write involves multiple steps, you should provide some comments so that the reader can immediately see what you are doing.

Code correctness

- I will usually provide a simple test driver. Even if your code works correctly with this driver, do not automatically assume that it works in all cases. You should write your own test drivers to test your code more extensively.

- Make sure that your code runs correctly in debug mode; i.e., when the macro `NDEBUG` is not defined. If your code works in non–debug mode but causes an assertion failure in debug mode, it is wrong!

Code efficiency

- Avoid redundant computations. A loop that computes a quantity that is the same for each iteration of the loop is inefficient. For example, in the loop

```
for (int i=0; i < N; ++i) {
  Affine A = Scale(5.4f) * Trans(Vector(-3,5));
  Point P = A * vertex[i];
  ...
```

the affine transformation `A` is the same with each loop iteration. To avoid the redundant computation, you should compute its value outside of the loop:

```
Affine A = Scale(5.4f) * Trans(Vector(-3,5));
for (int i=0; i < N; ++i) {
  Point P = A * vertex[i];
  ...
```

- Organize conditionals to reduce the number of comparisons needed to reach a block of code. For instance, suppose we have the point (x, y) in the plane, and we want to determine which quadrant it lies in. If we code this as

```
if (x >= 0 && y >= 0)
  Q = 1;
else if (x < 0 && y >= 0)
  Q = 2;
else if (x < 0 && y < 0)
  Q = 3;
else if (x >= 0 && y < 0)
  Q = 4;
```

then to reach the statement '`Q = 4`', six comparisons need to be made (with short–circuit evaluation). However, we if we use

```
if (x >= 0) {
  if (y >= 0)
    Q = 1;
  else
    Q = 4;
  }
else {
  if (y >= 0)
    Q = 2;
  else
    Q = 3;
}
```

then only two comparisons need to be made to reach *any* of the assignment statements.

- Unless directed otherwise, do not rely on compiler or platform dependent optimizations. Code that is simple is preferred over code that overly complicated but *might* run slightly faster on some machines.