eecs 487

INTERACTIVE COMPUTER GRAPHICS

EECS 487 PA1: Rasterization

This assignment is due on Saturday, Sep. 27th, 2014 at 10:00 pm

Plain Page for Printing

Overview

Review the grading policy page on the course website. Remember that to incorporate publicly available code in your solution is considered cheating in this course. To pass off the implementation of an algorithm as that of another is also considered cheating. For example, if the assignment asks you to implement sort using heap sort and you turn in a working program that uses insertion sort in place of the heap sort, it will be considered cheating. If you can not implement a required algorithm, you *must* inform the teaching staff when turning in your assignment by documenting it in your writeup.

In this assignment you will implement the midpoint line drawing algorithm to rasterize line segments, scan conversion of triangles, line clipping and line anti-aliasing. You will use barycentric coordinates to interpolate colors and provide anti-aliasing for lines and triangles.

Graded Tasks (100 points total)

1. Implement the midpoint algorithm for line segments. 5 pts
2. Implement triangle rasterization. 5 pts
3. Implement line clipping. 25 pts
4. Implement anti-aliased rendering/filtering for lines. 25 pts
5. Implement anti-aliased rendering/filtering for triangles. 25 pts
6. Create a fun and interesting scene. 10 pts
7. Writeup. 5pts

Support Code: Vcanvas

Download and extract the support code gzipped tarball. The support code should compile under Linux, Mac OS X, and Windows. We provide a `Makefile` to build the application on the command line. We do not provide any IDE (Eclipse, Visual Studio, or XCode) project files. If you're going to use an IDE, please take a look at the `Makefile` to find out which source files, header files, and libraries must be included in your project. For this assignment, the `Makefile` builds two applications. The first one, called `vcanvas`, depends **only** on the files `vcanvas.cpp`, `rasterizer.cpp`, `rasterizer.h`, and `xvec.h`. The second one, called `draw`, requires all the `.cpp` and `.h` files provided, **except** `vcanvas.cpp`. To use an IDE you must create two separate projects, one to build `vcanvas`, the other to build `draw`. If you include both `draw.cpp` and `vcanvas.cpp` in either project, your IDE will complain and you won't be able to build either program. It may also be helpful to read the details on setting up IDE projects in the *Building OpenGL/GLUT Programs* course note.

Once you've built `vcanvas` and run it, you should see a window similar to the one from Labs 1 and 2. The program displays a coarse grid of virtual pixels and allows you to draw one line or one triangle at a time. Two clicks on different virtual pixels give you a line. A third click on a pixel different from the first two gives you a triangle. The program is very simple and doesn't check for degenerate, e.g., colinear, triangles. Either of the following actions clears the grid, ready for another line or triangle to be drawn:

- Clicking on either end points of a line
- Clicking anywhere on the grid after a triangle is drawn
- Pressing the SPACEBAR or ESC key on the keyboard

In addition, the program offers the following functionalities, listed with their keyboard shortcuts:

- a Toggles anti-aliasing on and off.
- c Toggles clipping on and off. When clipping is on, a clipping window is drawn on the grid. Only line clipping is currently supported; triangle clipping is not supported.
- t Turns the first (red) vertex of the line or triangle transparent, cycling through alpha values of 0.75, 0.5, 0.25, 0.0, and back to 1.0 on each press of the keyboard shortcut. (The size of alpha decrement is controlled by the `ALPHA_DEC` macro in `vcanvas.cpp`.)
- q Quits the program.

Your Tasks

Search `rasterizer.cpp` for /*YOUR CODE HERE*/ to find where you can put your implementation. All your modifications must be *within* `rasterizer.cpp` (and `rasterizer.h` if necessary). Do NOT use any OpenGL state or call any OpenGL function. You are writing a software rasterizer; to use OpenGL's rasterizer would be considered cheating. Besides, the support code will malfunction if you use OpenGL in your code.

The vector class XVec is used heavily throughout this course. Be comfortably familiar with it.

1. Midpoint Algorithm for Line Segments

The assignment uses two graphical primitives. The first is the line. In `rasterizer.cpp` implement the following function using the midpoint line rasterization algorithm:

```
void Line::drawInRect(XVec4i& clipWin);
```

The argument `clipWin` is described in the Line Clipping section below. The lines drawn should be 1-pixel thick, that means every row (or every column, or both) contains at most one pixel. You can draw a solid red line first to test your line drawing code. You may, and are expected to, simply adapt your Lab1 code for this task (which is why it is not worth that many points).

Once your line drawing code is correctly implemented, incorporate color interpolation across the line using the parametric equation of a line segment; do not forget to interpolate the alpha values also.

To set pixels use the function:

```
void drawPoint(XVec2f& p, XVec4f& color, XVec4f& clipWin);
```

To test your line drawing code, build and run the `vcanvas` application. Click in the virtual canvas to set the first end point of the line. Then move the mouse and click on another virtual pixel to set the second end point, release the mouse button and click again (double click) on the same virtual pixel to draw a line. Moving the mouse before that third click produces a triangle.

2. Triangle rasterization

The second graphical primitive is the triangle. In `rasterizer.cpp` implement the function:

```
bool Triangle::containsPoint(XVec2f& p, XVec4f& pointColor);
```

It returns a boolean denoting whether the point `p` is within the triangle; if it is, the function returns `true` and sets `pointColor` to be the interpolated color of the vertices' colors, using barycentric coordinates. You may, and are expected to, adapt your Lab2 code for this task. To test your triangle drawing code, you may want to return only the color blue initially. Once your triangle drawing code is correctly implemented, incorporate color interpolation using barycentric coordinates; again, do not forget to interpolate the alpha values.

Next implement the function:

```
void Triangle::drawInRect(XVec4f& clipWin);
```

It should draw a triangle using scan conversion; be sure to have it use the function `containsPoint()` above! Build and run the `vcanvas` application to test your triangle rasterization code.

3. Line Clipping

The argument `clipWin` of function `Line::drawInRect(XVec4f& clipWin)` is a 4-vector containing, in order, the *x-coordinate of the lower left corner, the y-coordinate of the lower left corner, the width, and the height* of the clipping window.

You are to implement the Cohen-Sutherland trivial accept/reject and Cyrus-Beck line clipping alogrithms. The goal is to discard any line segment that lies entirely outside the clipping window, and trim line segments that lie partially outside the clipping window, leaving only parts that are inside the window. Thus line clipping must be done at the start of the `drawInRect()` function before drawing. When clipping, do *not* modify the vertices of the given line segment, instead use local variables to store the clipped endpoints. If you have written your line rasterization code above to use the vertices of the given line segment, you would need to modify it to use the clipped endpoints instead. When a line is clipped, its color interpolation must be clipped accordingly. For example, if one vertex is red and the other is green and the line is clipped three quarters of the way towards green, the clipped line should show only gradations of green.

Note: the triangle clipping provided in `draw.cpp` is a brute force rejection test of every pixel. Do *not* use the brute force method for your line clipping. You are *required* to implement both the Cohen-Sutherland and the Cyrus-Beck algorithm. Use Cohen-Sutherland to trivially accept/reject a line. Then clip those lines that Cohen-Sutherland cannot trivially accept/reject using the Cyrus-Beck algorithm. If you cannot implement either algorithm, you must say so in your writeup. To pass off another algorithm as the one required is considered cheating.

To test your line clipping code, first draw a clipping window by pressing 'c', then draw a line with one or both end points outside the clipping window. Pressing 'c' repeatedly toggles the clipping window on and off. The clipping window is of fixed size and location. The lower left corner of the clipping window is currently set at (4,4) in virtual pixel coordinates (controlled by macros `CMINX` and `CMINY` in `vcanvas.cpp`). The size of the clipping window is `(screen_width/2, screen_height/2)`. Note that a virtual pixel at coordinate `(x, y)` is centered at `(x+0.5, y+0.5)`. So when drawing pixels be aware when you need to use `rintf()`, `floorf()`, or `ceilf()` to convert from `float` to `int`. When clipping is on, portion(s) of the displayed line outside the clipping window is shown in shades of grey and the portion inside the clipping window is shown in color. Your clipping code should not change the rasterization of a line (which pixels are turned on to draw the line). However, due to rounding error, you may find some pixels inside the clipping window near its boundaries to be grayed out. This is ok.

What to Test

Here are some of the cases we will test for:

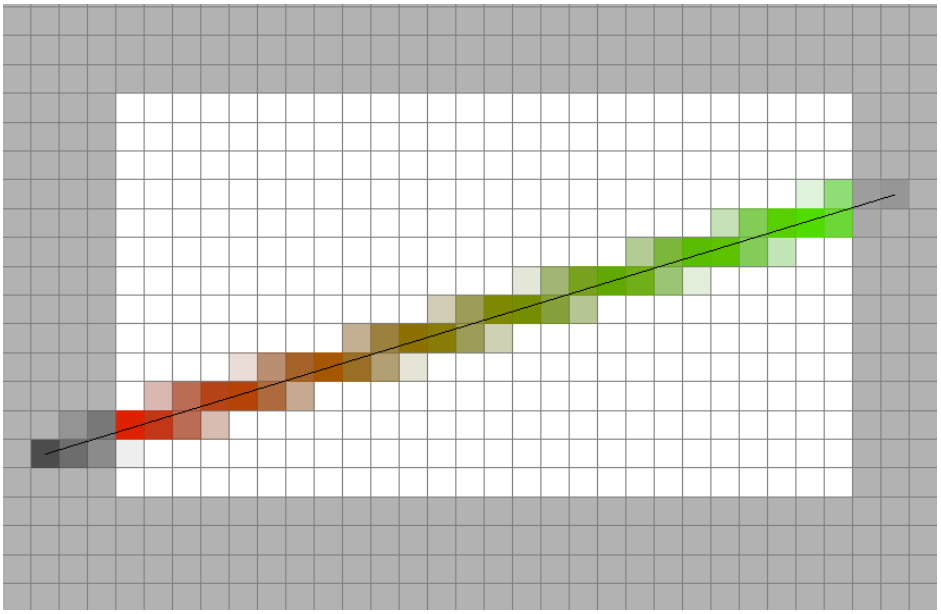
- Lines entering the clip window from 8 directions are properly clipped.
- Lines leaving the clip window to 8 directions are properly clipped.
- Lines straddling the clip window are properly clipped.
- Horizontal and vertical lines are properly treated.
- Cases where Cohen-Sutherland trivial accept/reject test fails are properly clipped.
- Clipped lines should lie exactly on the original unclipped lines.
- Color is properly clipped; lines completely outside the clipping window should be grayed out. Some pixels inside the clipping window near it boundary may be grayed out due to round-off error. This is ok.

4. Anti-Aliasing Lines with Area-sampling

The `Line` object has a member variable denoting whether it should be drawn anti-aliased. If this variable is set to true, draw your line anti-aliased using an area-sampling based algorithm briefly described as follows.

Considering only lines with slope `0 ≤ m ≤ 1`, the midpoint algorithm begins by drawing a point, then it moves right, and perhaps up, and repeats. Every time the algorithm draws the point `(x,y)`, it also draws the point above or below the point. Doing this will create a two-pixel thick line. Now just set the alpha values of those two pixels "appropriately" and the line will appear smoother. It is part of the assignment for you to determine how to set the alphas appropriately. (Hint: between what values is `fmid/dx` in the midpoint algorithm?) Further, if the alphas of the line's two endpoints are not the same, for each point on the line, the alphas of the two pixels should add up to the interpolated alpha at that point. This is a form of anti-aliasing by area sampling, also known as pre-sample filtering. See Lecture 8 for further discussion on the algorithm.

Press the '`a`' key to toggle anti-aliasing on and off. The figure below shows an anti-aliased line clipped to the clipping window. (The color of the anti-aliased line becomes darker when clipping is turned on. This is due to our drawing a greyed-out unclipped line under the clipped line. The color of each pixel is blended with the greyed-out pixel, darkening the color. This is okay as long as your pixels do not turn completely grey.)

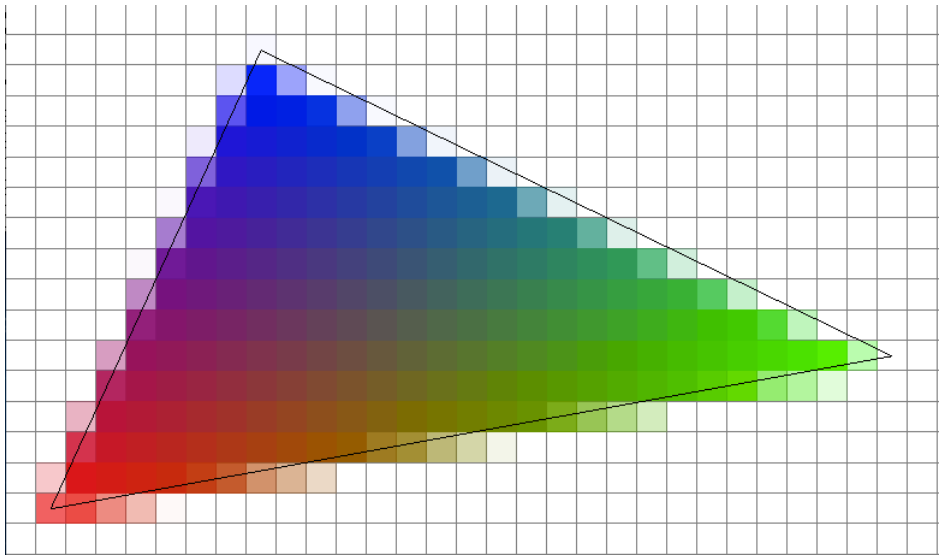


5. Anti-Aliasing Triangles with Multi-sampling

The `Triangle` object has a member variable denoting whether it should be drawn anti-aliased. If this variable is set, draw the triangle anti-aliased using multi-sampling with at least four samples. That is, for every pixel, compute and average the colors of at least four subpixels. Recall that we differentiate multi-sampling from super-sampling. Whereas super-sampling draws into a large buffer and then average down and re-draw to a smaller buffer, multi-sampling averages the color of multiple sub-pixels of a pixel in a single buffer. It does *not* use multiple buffers of different sizes. You can treat a given pair of pixel coordinates as specifying the lower left corner of the unit area covered by a pixel.

Press the '`a`' key to toggle anti-aliasing on and off. Remember that if you cannot implement an algorithm, you must disclose it in your writeup. This applies to the anti-aliasing algorithms also.

The figure below shows an anti-aliased triangle:



6. Creative Scene

Draw an interesting scene using the provided drawing application (described in the next section). Do not simply throw down 42 triangles and call it a Picasso. Put some time into it and play! Try to convey lighting or make a happy clown. Graphics is part programming, part math, and part art, so use your imagination! See samples from previous terms in the Image Gallery. Before having fun with this part, read the following section on how to use the drawing application.

Draw Application

The support code comes with a simple drawing application that utilizes your rasterizer. To build the drawing program, type "`make draw`" and run the `draw` executable binary file. You should see the application running and showing its window. By default, the application is set up to use your sofware renderer. Until you have your software renderer implemented, you would want to toggle the application to use the hardware renderer, as described below, to see how it is supposed to behave.

How to Use

It is highly recommended that you skim through the code to understand how this application works (it actually has a lot less code than you might imagine). The application provides two panels, the *canvas* on the left, the *picker* on the right. The picker is fixed size and contains a few "sliders" and "buttons"---even if it ducks back to a 1980s UI (such is life with GLUT).

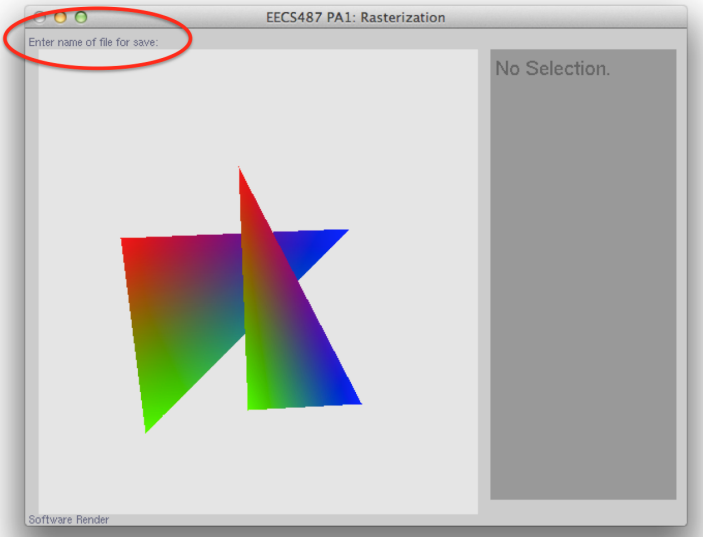
To draw a triangle in the canvas, click on the canvas with the left mouse button and do not release. Drag the mouse and then release. This sets the first two vertices of the triangle. Then move the mouse and watch a triangle grow. Click again to set the third vertex to complete the triangle. Repeat to create multiple triangles. To draw a line, do not move the mouse after specifying the second vertex, instead click it at the same location one more time.

Clicking on an existing triangle selects it. Selection of triangles uses the function `containsPoint()`. Make sure to test this function with "hardware rendering" turned on. When a line or triangle is selected, its vertices have boxes drawn on them and a thin edge surrounds it. One vertex will have a yellow box; this is the selected vertex. The picker on the right can be used to modify the color of the selected vertex (which is always displayed at the top of the picker). Try playing with the sliders (the diamond-shape outlines on the "Opacity", "Hue", and "Saturation+Value" boxes) to change the hue, saturation, value (color spaces such as this, HSV, will be discussed later in the course), and alpha (opacity) of the vertex. User can select any already drawn line or triangle or its vertices and change the color accordingly. The picker will not function if no vertex is selected.

The above explanation is enough to create wonderful scenes. The application does offer a little more functionality for ease of use. Right-click in the application window and a popup

menu appears. Each menu item is listed with its keyboard shortcut.

g	Toggles a grid on and off to help alignment of vertices.
x	Toggles snapping on and off. With it enabled, vertices will automatically snap to the grid (even if it is not visible).
backspace	Deletes the currently selected object.
a	Toggles anti-aliasing of the currently selected object. Polygon anti-aliasing in the hardware renderer is not implemented, so do not expect to compare your results against those produced by the hardware renderer.
h	Toggles between hardware and software rendering. Toggle on hardware rendering to see how the application is supposed to behave in terms of color interpolation and clipping using OpenGL instead of your code. Note that without your `Triangle::containsPoint()`, triangle selection doesn't work.
c	Enables the user to draw a clipping window on the screen: press 'c', then press the mouse button to mark one corner of the clipping window, without releasing the mouse button, drag the mouse to the opposing corner of the clipping window and release the mouse button.
CTRL+c	Toggles clipping on and off toggle clipping off and on (when drawing a clipping window, clipping is automatically toggled on). When clipping is toggled on, lines and triangles will be clipped to the clipping window; visually, the portions inside the clipping window will be shown in color, while those outside will be shown in grayscale. Toggle on "hardware rendering" to see how clipping is supposed to work. You only need to implement the line clipping function, the draw program provides the triangle clipping function.
f	Brings the currently selected object forward. There may be no visual change if the object "above" it does not overlap. The user may have to tap 'f' a few times to move an object forward enough.
CTRL+f	Brings the currently selected object to the front.
b	Sends the currently selected object backward. There may be no visual change if the object "below" it does not overlap. The user may have to tap 'b' a few times to move an object backward enough.
CTRL+b	Sends the currently selected object to the back.
CTRL+n	Clears the canvas and creates a new scene. It does not save the current canvas or re-confirm canvas clearing, so be careful!
CTRL+o	Type in the name of a file to open in the input area above the canvas.
CTRL+s	Saves the scene. It also saves a tga image of the scene. If this is the first time you try to save and the file is untitled, CTRL+s behaves the same as CTRL+SHIFT+s below. In Mac OS X, choosing "Save" under the "File" menu or hitting CMD+S will store a screen shot of the application as a tiff file instead (which is **not** what you should turn in).
CTRL+SHIFT+s	Type in the input area above the canvas the name of a file for saving. It also saves a tga image of the scene under the same name, appended with a ".tga " extension, as shown in the screen shot below. Note that in Mac OS X, choosing "Save As" under the "File" menu or hitting CMD+SHIFT+s will create a screen shot of the application as a tiff file instead.



ESC Cancel the current operation. It applies to line drawing, triangle drawing, clip area drawing, and file name input.

The application does not feature undo/redo nor cut/copy/paste. When the grid is on, it is visible; when snapping is on a magnet is shown in the upper left corner; whether rendering is currently done in hardware or software is displayed in the lower right corner. Play, fidget, and have fun!

WARNING: The program reads and writes a simple file format. Modify the files created at your own peril.

Final Thoughts

After completing this project take a moment to understand why hardware and software rendering differ so greatly in speed. Think about how the graphical components on screen interact; this is UI stuff but still heavily graphics-related. Imagine some simple additions to the application and how they may be accomplished. Hopefully those who use Photoshop or Gimp now appreciate their true power.

Submission Requirements

Test your compilation! Your submission must compile without **errors** and **warnings** (on Visual Studio, warnings of "PBO file not found" is ok). Code that does not compile will be heavily penalized.

Create a writeup in **text format** that discusses:

1. Your platform - Linux, Mac OS X, or Windows, and which version and flavor of each.
2. Anything about your implementation that is noteworthy. Be sure to explain how you determine the "appropriate" alpha values in line anti-aliasing and how corner cases were implemented.
3. The name of the scene you created and a brief description.
4. Feedback on the assignment.
5. Name the file writeup-*uniqname*.txt.
   For example, the person with uniqname *tarukmakto* would create *writeup-tarukmakto.txt*.

Save an image file from the `draw` application (using CTRL+s, **not** by capturing a screen shot!) and post the image (one or more) to the course's Image Gallery by logging in (using `ssh`) to

an ITD/CAEN Linux machine and run the following:

```
% cd <to where your image.tga file is located>
% /afs/umich.edu/class/eecs487/scripts/postimg <image.tga> [<image2.tga ...>]
```

On Windows, you can use MobaXterm Personal/Home Edition to login to an ITD/CAEN Linux machine.

Do not include the rendered image in your *PA1 files* below.

Your *PA1 files* then consists of your `writeup-uniqname.txt`, `rasterizer.cpp`, and, if modified, `rasterizer.h`. Your code must not require other external libraries or include files other than the ones listed in the `Makefile`.

To turn in your PA1, upload your *PA1 files* to your `pa1` folder on MFiles:
`https://mfile.umich.edu/?path=/afs/umich.edu/class/eecs487/f14/FOLDERS/<uniqname>/pa1/`.
Replace "<uniqname>" with your uniqname. Please report any problems to ITCS.

You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline. We archive a SHA1 of each submission by the due date and on each late day, so please do not modify your submitted files past the due date unless you want to turn in your work late.

Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Your code must not require other external libraries or include files other than the ones listed in the `Makefile`. For this assignment, you should turn in only your `writeup-uniqname.txt`, `rasterizer.cpp`, and, if modified, `rasterizer.h`. Do not turn in any binary files (object, executable, or image) with your assignment. Post your image(s) to the course's Image Gallery using the `postimg` script as explained above.

Do remove all `printf()`'s or `cout`'s and `cerr`'s you've added for debugging purposes.

General

It is part of the Honor Code of this course that the overall design and final details and implementation of your programming assignments must be your own. If you're stuck in either the design, implementation, or debugging of the assignment, you're allowed and encouraged to consult with your classmates. However, the original design and final implementation details must all be your own. So you cannot come up with the original design *together* with your classmates. You can consult your classmates *only after* you've come up with your own design but ran into some specific problems. Similarly for the implementation, you cannot consult your classmates prior to writing your own implementation. And in all cases, you're not allowed to look at any of your classmates' source code, not even in order to help them to debug. The same applies to design and implementation from previous terms.

Coding style

Use a reasonable organization for your overall program:

> Design a fairly reasonable class structure. On the one hand, don't stick everything into one class/struct. On the other hand, don't be bureaucratic and require the reader to follow one class definition after another to find a single line of code wrapped in n layers of methods, with each method doing nothing but calling the next one. If the way you design your code feels sloppy to you, it probably is. Utilize multiple files in a way that is consistent with the general use of C/C++. Don't use more files than necessary, you don't have to put each class/struct in a separate file of its own.

Don't use literals!

> Use either `const`, `enum`, or `#define` to give your literals meaningful names. We do deduct points for *each occurrence of literals*, even if it is the same one. The only exceptions will be for loop counter, command-line options, NULL(0) and TRUE/FALSE(1/0) testing/setting, and help and error messages printed out to user.

Use reasonable comments:

Explain what each class does and what each data member is used for. A one or two line description of most member functions is also desirable. Where you use non-standard coding techniques, document them. **List your name and the date last modified for each file.**

Remember that a useless comment is worse than no comment at all.
`int temp; // declare temp. variable`
would be an example of a useless comment which just makes code harder to read!

Use reasonable formatting:

From indentation alone, it should be obvious where a given code block ends. Avoid lines that wrap in an 80 column display wherever possible. Your code should be tight, compact, and visually tidy. Don't let bits and pieces fly off every which way. Your code is not abstract painting.

Variable names:

Use reasonable and informative variable names, but limit name size to a reasonable length. A 40-character name better has a very good reason to exist. Variable names like 'i' and 'j' can be reasonable, but you should not use such variables to store meaningful long-term data. Other than LCV (loop control variables) you should use descriptive names for your variables, functions, classes, methods, structures, etc.

Reduce, Reuse, and Recycle your code, algorithms, and structures:

Try using inheritance, templating, polymorphism (virtual function), or similar methods to reduce the size of your code. Do not unnecessarily duplicate code. Less code leads to less debugging. If you find yourself rewriting basically the same code more than once, stop and try to see if you can somehow reuse the code by making it a function call or implementing a polymorphic function.

Unreadable code can cost you up to 10 points!

Empirical efficiency

We will check for empirical efficiency both by measuring the memory usage and running time of your code and by reading the code. We will focus on whether you use unnecessary temporary variables, whether you copy data when a simply reference to it will do, whether you use an *O*(*n*) algorithm or an *O*(*n*^2) algorithm, but **not** whether you use `printf`'s or `fprintf`'s or `cout`. Nor whether your ADTs have the cleanest interfaces. In general, if the tradeoff is between illegible and fast code vs. pleasant to read code that is unnoticeably less efficient, we will prefer the latter. (Of course pleasant to read code that is also efficient would be best.) However, take heed what you put in your code. You should be able to account for every class, method, function, statement, down to every character you put in your code. Why is it there? Is it necessary to be there? Can you do without? Perfection is reached not when there is nothing more to add, but when there is nothing more that can be taken away, someone once said. Okay, that may be a bit extreme, but do try to mind how you express yourself in code.

Hints and advice

- Design your data structures and work through algorithms on paper first. Draw pictures. Consider different possibilities *before* you start coding. If you're having problems at the design stage, come to office hours. After you have done some design and have a general understanding of the assignment, re-read this document. Consult it often during your assignment's development to ensure that all of your code is in compliance with the specification.
- Always think through your data structures and algorithms before you code them. It is important that you use efficient algorithms in this programming assignment and in this course, and coding before thinking often results in inefficient algorithms.
- Make sure you don't clutter `stdout` with unnecessary output. Use `gdb` to debug.
- You shouldn't print to `stderr` unless there is an error.
- The teaching staff will be happy to help you track down bugs, but you have to fix them yourself once they are found. We will not help you track down a bug unless you can show us *in gdb* where you suspect the bug to be. That is, you need to show us that you

have tried your best to track down the bug, and that you have used gdb.

To encourage early start on the assignment we will stop helping you to debug 48 hours before the due date. For a Wednesday 10:00 pm deadline, we stop helping you at 10:00 pm on the Monday prior.

Testing Your Code

We will be grading for correctness primarily by running your program on a number of test cases. If you have a single silly bug that causes most of the test cases to fail, you will get a very low score on that part of the programming assignment *even though you completed 95% of the work*. Most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. Each testcase should test only one particular feature of your program.

If any part of this document is unclear, ambiguous, contradictory, or just plain wrong, please let one of the teaching staff know. Have fun coding!