CSC 418/2504 Computer Graphics, Fall 2010

Assignment 1 (15% of course grade)

Part A (written): Due at beginning of tutorial (6:09pm) on Wednesday, October 6, 2010.
Part B (programming): Due 11:59pm on Friday, October 15, 2010.

Part A [50 marks in total]

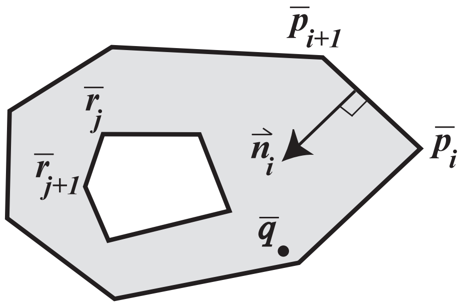
Below are 4 exercises covering different topics from the first weeks of class. They require thought, so you are advised to consult the relevant sections of the textbook, the online lecture notes and slides, and your notes from class well in advance of the due date. Your proofs and derivations should be clearly written, mathematically correct, and concise.

All questions require showing the steps toward the solution, and marks will be subtracted if this is not the case. Even if you cannot answer a question completely, it is very important that you show your (partial) answers and your reasoning. Otherwise your TA will ***not*** *be able to award you partial marks.*

You can either give your handwritten solutions to Part A to your TA or you can submit them online in PDF format by the due date/time (e.g., by scanning your handwritten solution or by using LaTeX/word to typeset it).

1. **Points & Polygons** [15 marks]

   Suppose the vertices of a convex polygon are $\bar{p}_1, \ldots, \bar{p}_N$, given in counter-clockwise order. Let the coordinates of vertex $\bar{p}_i$ be (x_i, y_i).



   (a) [3 marks] Express the coordinates of the inward-facing normal $\vec{n}_i$ at $\frac{\bar{p}_i+\bar{p}_{i+1}}{2}$ in terms of the coordinates of vertices $\bar{p}_i$ and $\bar{p}_{i+1}$.

   (b) [3 marks] Let $\bar{q} = (x, y)$ be an arbitrary point on the 2D plane, let l_i be the line containing $\bar{p}_i$ and $\bar{p}_{i+1}$, and let $\vec{n}_i$ be the normal in (a). Give a test that determines whether point $\bar{q}$ is on the "same (inward-facing) side" of l_i as $\vec{n}_i$.

   (c) [9 marks] Describe an algorithm that can tell whether a 2D point $\bar{q}$ is inside the gray-shaded region in the figure. You should assume that the polygons defined by vertices $\bar{p}_1, \ldots, \bar{p}_N$ and $\bar{r}_1, \ldots, \bar{r}_M$, respectively, are both convex. *Hint:* Each edge is contained in an infinite line. Each infinite line divides the 2D plane into two half-planes, the left half-plane and the right half-plane (here left/right means left/right with respect to a counter-clockwise direction of traversal of the vertices). The key insight you should use is that the interior of a convex polygon is the intersection of the left half-planes of all the edges of the polygon.

2. **Transformations & Commutativity** [12 marks]

   We say that the 2D transformations $f()$ and $g()$ *commute* if and only if $f(g(\bar{p})) = g(f(\bar{p}))$ for all points $\bar{p} \in \mathbb{R}^2$. For each of the four cases below, where $f()$ and $g()$ are homographies, prove whether or not they commute:

   (a) [3 marks] Both $f()$ and $g()$ are arbitrary homographies.

   (b) [3 marks] One is an arbitrary rotation and the other an arbitrary translation.

   (c) [3 marks] One is an arbitrary translation and the other is a non-uniform scaling.

   (d) [3 marks] One is an arbitrary rotation and the other is a reflection.

   In each case, your solution can either be a derivation that proves/disproves commutativity or, if $f()$ and $g()$ do not commute, a specific counter-example.

3. **Points, Triangles & Homographies** [15 marks]

   (a) [5 marks] Derive the homography that maps points $(1,\ 0), (3,\ 0), (2,\ 1), (0,\ 1)$ to points $(1,\ 0), (2,\ 1), (2,\ 2), (0,\ 2)$, respectively. Does this homography represent an affine transformation? Explain why it does or does not.

   (b) [10 marks] Let $f()$ be an arbitrary homography, and let $\bar{p}_1, \bar{p}_2, \bar{p}_3$ be the vertices of a triangle. Prove that if that triangle contains a point $\bar{q}$, the homography-transformed triangle defined by $f(\bar{p}_1), f(\bar{p}_2)$, and $f(\bar{p}_3)$ may *not* contain point $f(\bar{q})$. In other words, the inclusion relationship between points and triangles is not preserved under arbitrary homographies.

4. **Curves & Trajectories** [8 marks]

   Consider a computer game where all celestial bodies are represented by points on the 2D screen.

   (a) [4 marks] A comet is traveling towards Earth on a parabolic trajectory that can be expressed parametrically as follows:

$$\begin{aligned} x(t) &= t \\ y(t) &= at^2 + b\ , \end{aligned}$$

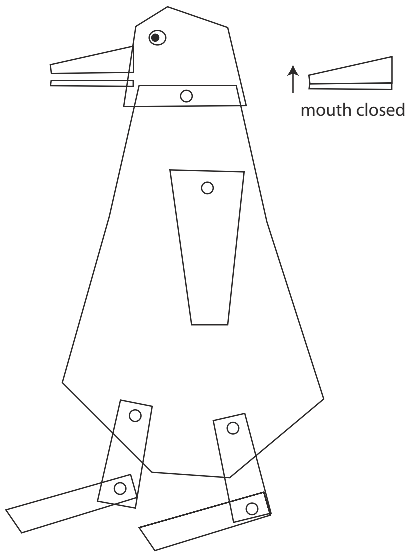
   where a and b are constants and $-\infty < t < \infty$ is the free parameter corresponding to time. Compute the comet's acceleration vector and unit normal vector as a function of time t.

   (b) [4 marks] Suppose that $a = 4,\ b = -45$ and we know that impact happened at $t = 10$. What was the Earth's position at the time of impact with the comet? With what velocity vector did the comet strike?

[Continues on next page $\longrightarrow$]

Part B: Animating a Hierarchical 2D Object [50 marks in total]

The figure below shows an articulated nine-part, seven-degree-of-freedom (DOF) planar robot penguin:



It has six rotational joints (depicted by circles), each with one rotational degree of freedom. The beak has a single translational degree of freedom: the beak can only move up or down (in the coordinate frame of the head). Your task will be to render and animate such a robot penguin using OpenGL.

Hierarchical objects like this are often defined by specifying each part in a natural, part-based coordinate frame, along with transformations that specify the relative position and orientation of one part with respect to another. These transformations are often organized into a kinematic tree (e.g., with the torso as the root, and the jaw as a leaf). In addition to the kinematic tree, one must also specify the transformation from the root (e.g., the torso) to the world coordinate frame. Then, for example, to draw the torso you transform the points that define the torso from the torso's coordinate frame to the world coordinate frame, and then from the world coordinate frame into device coordinates. Then to draw an arm, you must transform the points that define the arm in the arm coordinate frame to the torso's coordinate frame, and then from the torso's coordinate frame to the world coordinate frame, and then into device coordinates. And so on down the tree.

Rendering articulated objects is easiest if a current part-to-device mapping is accumulated as you traverse the object/part hierarchy. You maintain a stack of coordinate transformations that represents a sequence of transformations from the current part coordinates up through the part hierarchy to world coordinates, and finally to device coordinates. For efficiency, we do not apply each of the transformations on the stack in succession. Rather, the top of the stack always represents the composition of the preceding transformations. OpenGL provides mechanisms to help maintain and apply these transformations.

Your Programming Task

Your task is to design and render the articulated robot penguin using OpenGL. When the program is run, the robot should move (i.e., animate) in order to help test that the rendering is done correctly.

To accomplish this, you must perform the following basic tasks:

(a) [5 marks] Design the parts in terms of suitable generic shapes and deformations, and draw them using OpenGL.

(b) [10 marks] Design and implement suitable transformations that map each part's local coordinate frame to the coordinate frame of its predecessor in the kinematic tree. Extend the GUI with additional spinners to control each of the 9 degrees of freedom (DOFs). *H*int: The interactive DOF controls will be useful in debugging the transform hierarchy you build.

(c) [10 marks] Design and implement a set of functions that will control the animation, i.e., will control the state of each joint in each frame. You can use simple functions such as sinusoids to define the way in which parts move with respect to one another. Or, if you wish, you could specify a sequence of specific joint angles that the rendering will loop through. You can also use key-framing to specify a few key poses for the penguin (in terms of the joint angles) and linearly interpolate between them for smooth animation. *H*int: You can use the GUI build for (b) to choose the set of key-frames or help you specify the values for the joint angles that produce the desired animation.

(d) [15 marks] Put it all together to generate your animation, by drawing each part in turn as you descend the kinematic tree (once per frame). Use the OpenGL transformation stack to control relative transformations between parts, the world and the display device. It is not necessary to write code that could be used to render arbitrary articulated objects, thereby requiring that your code can traverse any kinematic tree. To keep things simple, you may hardcode the sequence of parts that are drawn.

(e) [2 marks] Be sure to also draw the small circles which depict the locations of the rotary joints.

(f) [8 marks] Explain everything you did in a written report (see below).

Helper Code

To get started, we have created a simple demo for you. This will show you how to use the very basic commands of OpenGL to open a window and draw some basic shapes. It will also provide you with a template Makefile for compilation and linking of your programs.

This simple demo program opens a window, and animates a two squares connected by a hinge. To unpack, compile and run this demo on CDF, download the file *a1.tgz* and use the following commands

```
tar xvfz a1.tgz
cd a1/penguin
make
penguin
```

Compilation

To compile programs easily we have set up a simple makefile for you, that builds the executable using the *make* command you issued above. *Make* searches the current directory for the file called *Makefile* which contains instructions for compilation and linking with the appropriate libraries.

You will find this Makefile useful when compiling programs for your later assignments. For example, if you wish to compile a program with a different name, change all occurrences of *penguin* to the name of the file you wish to compile. You can also change the Makefile to include code from several files by listing the C++ source files (i.e., the files ending in *.cpp*) on the line *CPPSRCS=*. The name of the executable file is determined by the name you use in the Makefile on the line *PROGRAM = penguin*.

When you run the demo program a graphics window will appear. The size of the window is determined by parameters (xmax and ymax) to the initialization routine. Each pixel is indexed by an integer pair denoting the (x, y) pixel coordinates with (0,0) in the top left hand corner and (xmax,ymax) in the bottom right.

Turning in your Solution to Part B

All your code should remain in the directory a1/penguin. In addition to your code, you must complete the files *CHECKLIST* and *REPORT* contained in that directory. *Failure to complete these files will result in zero marks on your assignment*.

The *REPORT* file should be a well-structured written (or diagramatic) explanation of your design, your part descriptions, and your transformations. The description should be a clear and concise guide to the concepts, not a simple documentation of the code. In addition to correctness, you will also be marked on the clarity and quality of your writing. We expect a well-written report explaining your design of parts and transformations.

Note that this file should *not* be thought of as a substitute for putting detailed comments in your code. Your code should be well commented if you want to receive full (or even partial) credit for it.

To pack and submit your solution, execute the following commands from the directory containing your code (i.e., *a1/penguin*):

```
cd ../../
tar cvfz a1-solution.tgz a1
submit -c csc418h -a A1b a1-solution.tgz   (if registered for CSC418)
submit -c csc2504h -a A1b a1-solution.tgz  (if registered for CSC2504)
```

Compatibility

All of your assignments **must** run on CDF Linux. You are welcome, however, to develop on other platforms (and then port to CDF for the final submission). This sample code is designed to work on Linux and Mac OS X machines, but should be portable to other Unix platforms (including Windows with Visual C++). If you are running your own machine, it almost certainly has OpenGL installed; if not, you can search for an rpm or go to http://www.opengl.org/ (see their getting started FAQ).

If you are planning to develop for windows with Visual C++, we have included some starter code to help you out. Unpack the starter code and place the unpacked files and the *include* directory in your *a1/penguin* directory. You will then need to add the skeleton code to your Visual C++ project.

If you develop on a different platform, be sure you know how to compile and test your code on CDF, well before the deadline. Your assignment must run on the CDF Linux configuration. Several marks will be deducted if your code does not compile and/or run without modification. If the marker cannot easily figure out how to compile and execute the code, it will most likely receive zero marks. If you choose to develop the assignment in MacOSX or Windows, test porting your code to Linux long before the deadline, perhaps even before you have finished the assignment. Often bugs that are "hidden" when compiling on one platform, make their presence known by crashing the application on a different platform. We will not be sympathetic to porting problems that you have at the last minute.