CPSC 40 Final Exam

Spring 2011

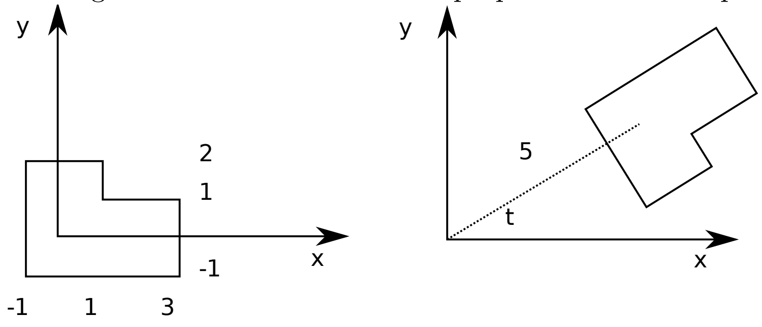
Due Thursday May 12 at 5pm
Open book exam

NAME:______________________________

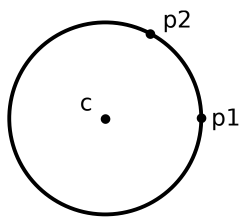
- You should work on this exam by yourself.
- You may consult you notes, textbooks, and code you wrote for labs.
- You may use the OpenGL red book documentation online, but otherwise, no other online resources are allowed.
- You may describe your solution in words and figures. You do not need to write C++, openGL, GLSL, or CUDA code unless explicitly stated.
- If you cannot solve a problem as specified, attempt a partial solution or describe a similar problem that you can solve. Partial credit will be assigned based on how close your problem matches the scope of the original problem.
- submit your solutions in hardcopy or via email to me before the deadline Thursday May 12th at 5pm.

Problem 1: Internally, OpenGL represents both points and vectors in homogeneous coordinates. Briefly explain what homogeneous coordinates are and list at least three reasons why homogeneous coordinates are useful. You may want to cite specific examples or discuss how you would accomplish certain tasks without homogeneous coordinates to support your reason.

Problem 2: Consider the two images shown below, the parameter t indicates a counter clockwise rotation by t degrees. The figure may not be to scale. Write the sequence of openGL matrix operations (glTranslate*, etc.) that should be applied to the geometry on the **left** to create the arrangement on the right. Be sure to indicate the proper order of the operations.



Problem 3: Suppose we are given a circle defined by three points: a center c, and two points p_1 and p_2 on the perimeter of the circle. Assume that the front face of the circle is determined to be the side in which one moves counterclockwise from p_1 to p_2 (see figure below.) Given a ray $\vec{r} = s + \vec{v}t, t \geq 0$, derive an algorithm that determines (a) if the ray hits the circle and (2) if so, computes the intersection point. Hint: this is more like the ray/triangle intersection test than the ray/sphere intersection test.



Problem 4: For this problem, you will need to consult the images in
`http://web.cs.swarthmore.edu/ adanner/cs40/s11/final/`
The image `correct.png` shows a correctly ray-traced image of the scene described in `input.txt`. Effects include ambient, diffuse, and specular lighting from multiple sources, as well as shadows. In each of the `bug.png` images, a single bug was introduced to make the image incorrect. For each of the eight bugs, describe the problem and suggest a possible component of the ray tracer that is incorrect (e.g., ambient lighting is computed incorrectly for triangles). Also describe a possible fix (this may be clear from describing the problem). There may be multiple ways of creating the same buggy effect. You only need to describe one. Note the bugs are usually only in one or two lines of code and not some elaborate obfuscated bug-beast spanning multiple functions/files. Feel free to modify your raytracer code to re-create some of the bugs.

Problem 5: When introducing bump-mapping (see your class/w10-bumpnoise folder), we implemented bump-mapping using shaders. More specifically, most of the effect was computed in the fragment shader. Briefly explain bump-mapping and describe why using a fragment shader might be a good way to implement this effect. Could this be implemented using a simple fragment shader but a more complex vertex shader? Could the effect be implemented in the OpenGL fixed pipeline? Explain any difficulties or advantages of using an alternate approach.

Problem 6: For this problem, you will be writing a little CUDA code and testing some properties of CUDA kernels. The code is available in your final folder if you run update40. First compile and run `maxval.cu`. It may give some warnings about static, throw, something, something, but this is OK. This program is supposed to compute the maximum of an array of floats. Initially a CPU only version has been provided for you. I provided various timing code to time the GPU and CPU versions of this max function. Your first step is to write a simple CUDA kernel that works with only one block and one thread. Because a global CUDA kernel can only have a void return type, the variable `result` can be used to hold a GPU buffer that can store one or more results. Call your kernel `max_gpu_single`, and have it store the max value in the buffer `result[0]`. The code in `main` will copy this buffer into a `partial_results` buffer and do some post processing. Call your kernel in main with one block and one thread, and note the time. Check that your GPU result matches the CPU result before proceeding.

Next, change the size of `N` near the top of the code from 32 to 32*1024*1024. Comment out the line in main which prints out the values of `a[i]` using `cout`, so you do not see 32 million items print. Run your code and note the time for the GPU and CPU versions. If your GPU version is significantly slower, that is OK at this point. Next, make the following changes and run some experiments.

1. Write a kernel called `max_gpu_block` that can be called on multiple blocks, each containing one thread. Call your kernel with the following number of blocks: 32, 64, 256, 512, 2048, and record the time. Note you will need to recompile. Note your kernel only needs to have each block compute the max of all the elements that block checks. Each block can store its maximum in `results[blockIdx.x]`. A small amount of post-processing by the GPU can then find the maximum over all the blocks.
2. Write a kernel called `max_gpu_thread` that can be called on a single block containing multiple threads. Call your kernel with the following number of threads and record the time: 32, 64, 256, 512. You may need to change the variable `partial_size` in main to max sure the results buffer is the appropriate size. Each thread will write to one slot in this buffer which is again post-processed by the CPU in main.
3. Finally, write a kernel called `max_gpu_combined` that can be called on a arbitrary number of blocks, each with multiple threads. Try various block and thread counts when calling your kernel, reporting at least three experiments and highlighting the parameters that result in the shortest run time.

Use `add_loop*` in your `class/w12-cuda` folder and `dot.cu` in your `class/w13-cuda_pt2` for suggested starting points. Once the first kernel is complete, running the experiments for the other kernels should be pretty quick. Summarize your results and any conclusions you may have in your final exam write-up. Submit your code via handin. Your code does not need to run all the experiment in one run, but should have all the kernels in one file.