Program 1

Parallel Hardware Ray Tracing

Due: 11:59:59 PM, September 5, 2011

Before you start, make sure you fully read the TRaX programming guidelines and information about setting up the compiler, linked on the class web site. This will save you lots of time and bugs!

Implement classes for `Vector`, `Ray`, and `Color` as discussed in class. You should implement all of the legal operators including arithmetic operators, dot product, cross product, vector length and vector normalize.

You will also implement a `Sphere` class. This will be replaced with a more powerful version in a future assignment, but will be used here for the purpose of testing the above infrastructure. The sphere should record the center and radius and have a single method called "intersects" that returns whether a ray hits the sphere.

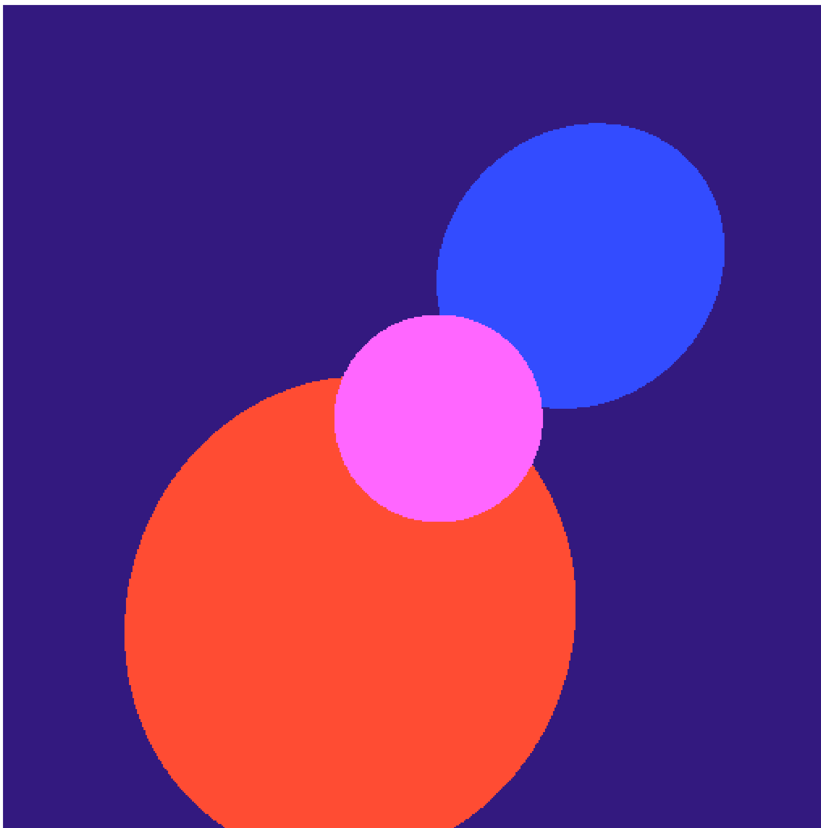
You may also want to implement an `Image` class, which will handle writing pixels to the frame buffer.

This assignment is worth 100 points.

1. **Required Image(s)** (80 points): Your ray tracer should reproduce the image shown here containing three spheres (circles without shading). The image is 512x512 resolution. If the image matches, you will get all 80 points. If you want partial credit you must provide your source code.

   The image is generated by creating three spheres:

   - Sphere 1: Center: [0.0f, 0.2f, 1.1f] Radius: 1.0f
   - Sphere 2: Center: [1.4f, 1.5f, 1.2f] Radius: 1.3f
   - Sphere 3: Center: [-1.5f, -0.5f, 1.0f] Radius: 1.9f



The following code will generate the rays to test against the three spheres.

```
trax_setup();
int xres = loadi( 0, 1 );
int yres = loadi( 0, 4 );
int start_fb = loadi( 0, 7 );
Image image(xres, yres, start_fb);

for(int pix = atomicinc(0); pix < xres*yres; pix = atomicinc(0))
  {
    int i = pix / xres;
    int j = pix % xres;

    Color result;
    float x = 2.f * (j - xres/2.f + 0.5f)/xres;
    float y = 2.f * (i - yres/2.f + 0.5f)/yres;
    Ray ray(Vector(0.f,0.f,-3.f), Vector(x, y, 1.f));
    if(sphere1.intersects(ray))
       result = Color(1.f, .4f, 1.f);
    else if(sphere2.intersects(ray))
       result = Color(.2f, .3f, 1.f);
    else if(sphere3.intersects(ray))
       result = Color(1.f, .3f, .2f);
    else
       result = Color(.2f, .1f, .5f);
    image.set(i, j, result);
  }
trax_cleanup();
```

Notice a few things about the code above:

- The file trax.hpp, is essentially an API for programming the TRaX architecture, and you will need to include it in your code. It is designed so that you can compile the exact same program for CPU and TRaX at the same time. When writing TRaX code, you must use this API if you want your program to work in the simulator as well as on CPU machines.
- `trax_setup` is defined in trax.hpp, which sets up an array of memory for the CPU compiled version of your code, exactly like the TRaX simulator would use. This is so that the same API calls, (such as storing to the frame buffer, loading the resolution, etc...) will work on the CPU.
- "xres" and "yres" are loaded from TRaX memory from addresses 1 and 4, respectively. Alternatively, you can use `GetXRes` and `GetYRes` defined in trax.hpp.
- "start_fb" is a pointer to the frame buffer's location in memory. The same API call (`loadi (0, 7)`) will return the correct frame buffer location no matter which architecture the code is compiled for (CPU or TRaX). Alternatively, use `GetFrameBuffer`.
- `atomicinc` is an intrinsic instruction for the TRaX architecture which atomically returns a unique consecutive integer, starting at 0. This is the mechanism for assigning work to threads. The CPU version of your code will be single-threaded, but we must use atomicinc for compatibility with TRaX.
- Notice that my code uses an `Image` class, with a method `set(int i, int j, Color result)`. This simply encapsulates calls to `storef` to write colors to the frame buffer. See "examples/gradient.cpp" for an example of storing data to the frame buffer. An `Image` class is recommended, but not required.
- Your code can not use the `double` data type at all. This includes immediate values. Notice that all of my immediate values (even integers that would be cast to float), are specified as being float, such as `2.f`.
- `trax_cleanup` writes the content of the frame buffer to a file called "out.png".
- Debugging: trax.hpp includes simple printing methods: `trax_printi` and `trax_printf`, for printing an integer value and a floating point value, respectively. Unfortunately, it does not support printing strings.
- It is recommended that you put all of your code in a subdirectory of the LLVM_Trax directory. The example Makefile is set up to work this way. Take a look at the tutorial on setting up and using the compiler for an example of how compile a TRaX program.

2. **Code listing** (20 points): Link your source code to your web page (preferably .tar). All of the code that you use should be included. You will not be graded on the quality of your code or comments, only on the presence of your source. We will verify that the code you hand in will produce the image(s) turned in.

3. **Creative Image(s)** (none): There will not be a creative image on this assignment. You can't get that creative just drawing circles.

What to turn in:

This assignment is just an introduction to TRaX programming and a very simple introduction to ray tracing. You are not required to use the TRaX simulator for this assignment, but you must use the TRaX programming guidelines. By midnight on the due date, you should send e-mail to teach-cs6965@list.eng.utah.edu with the following information:

1. **URL:** A pointer to a web page containing the following information:
   (a) Required image
   (b) Link to source code
2. **Time required:** How many hours did it take you to complete this assignment?
3. **Difficulty of assignment:** Was the assignment difficult or not? Feel free to expound or to be brief.

You will not be graded on these last two items. They will be used to help improve the class in future assignments and in future years.