# Program 2

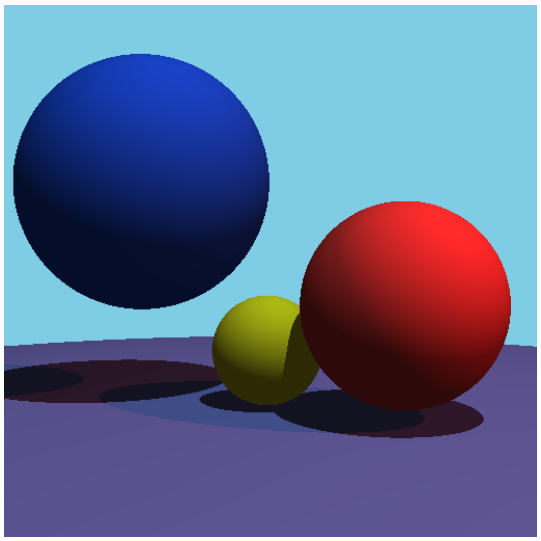
## Hardware Ray Tracing

Due: 11:59:59 PM, February 8, 2014

**Before you begin, make sure to update the repository with `svn up` and recompile the simulator.**

1. **Lambertian shading** (50 points) Implement Lambertian shading and cameras in your ray tracer. You must support a `PinholeCamera` and `PointLight` as described in class. Upgrade your `Sphere` class to support an object ID and material ID, and to support nearest hit information (recommend a `HitRecord` class). Implement a shading model that supports Lambertian shading with shadows for an arbitrary number of spheres and point lights in the scene. It is recommended you also implement a `Scene` class. See the lecture 6 and lecture 7 slides for recommendations on these classes.

   **Required Image** Your ray tracer should reproduce the image shown here containing four spheres and two point light sources.



The details of the scene are as follows:

- Resolution: 512x512
- Eye point: [-24.f, -2.f, 5.2f]
- Gaze direction: [0.986794f, 0.118415f, -0.110521f] OR Lookat point: [1.f, 1.f, 2.4f]
  - Gaze direction = normalize(Lookat point - Eye point)
- Up: [0.f, 0.f, 1.f]
- u_length: 0.194f (22° field of view)
- 4 spheres:
  - Sphere 1: Center: [1.5f, 3.5f, 4.f], Radius: 2.4f, Material 0
  - Sphere 2: Center: [-0.5f, -1.5f, 2.f], Radius: 1.8f, Material 1
  - Sphere 3: Center: [0.5f, 1.0f, 1.f], Radius: 1.f, Material 2
  - Sphere 4: Center: [0.5f, 1.0f, -100.f], Radius: 100.f, Material 3
- 4 materials. Each material is a simple Lambertian material with Kd = 0.7 and Ka = 0.3
  - Material 0: Color: [0.1f, 0.3f, 0.9f]
  - Material 1: Color: [1.f, 0.2f, 0.2f]
  - Material 2: Color: [1.f, 0.9f, 0.1f]
  - Material 3: Color: [.4f, .4f, .7f]
- 2 point light sources:
  - Light 1: Position: [-30.f, -20.f, 80.f], Color: [.7f, .9f, .9f]
  - Light 2: Position: [-20.f, -50.f, 40.f], Color: [.6f, .1f, .1f]
- Ambient light: Color: [.6f, .6f, .6f]
- Constant background: Color: [0.5f, 0.8f, 0.9f]

It is recommend you create a `Scene` class to keep the details of the scene in one place. See lecture 7 for a class skeleton. For example, in `main` to create the scene your code might look something like this:

```
Color materials[4];
Sphere spheres[4];
PointLight lights[2];

materials[0] = Color(0.1f, 0.3f, 0.9f);
// . . . initialize other materials

// Last 2 parameters are object ID and material ID
spheres[0] = Sphere(Vector(1.5f, 3.5f, 4.f), 2.4f, 0, 0);
spheres[1] = Sphere(Vector(-0.5f, -1.5f, 2.f), 1.8f, 1, 1);
// . . . initialize other spheres

lights[0] = PointLight(Vector(-30.f, -20.f, 80.f), Color(.7f,.9f,.9f));
lights[1] = PointLight(Vector(-20.f, -50.f, 40.f), Color(.6f,.1f,.1f));

Color background(0.5f, 0.8f, 0.9f);
Color ambient(0.6f, 0.6f, 0.6f);

// 4 spheres, 4 materials, 2 lights
Scene scene(spheres, 4, materials, 4, lights, 2, background, ambient);
```

It is also recommended you create a `shade` function. An example of calling `shade` from `main` might look something like this:

```
// . . . for each pixel, generate a ray using the camera . . .

HitRecord hit(100000.f); // "infinite" max_t
// intersect ray with each sphere
for(int k=0; k < scene.numSpheres; k++)
{
   scene.spheres[k].intersect(hit, ray);
}
Vector hitPoint = ray.origin() + (hit.minT() * ray.direction());
result = shade(hit, ray, hitPoint, scene);
image.set(i, j, result);
```

2. **Analysis** (30 points)

   **Before using the real simulator, make sure your program is generating the correct image by using the functional simulator (`run_rt`).**

   **For all of the experiments below, use a resolution of 128x128 (default), and a single TRaX thread using `tiny.config`.**

   Generate an analysis document. This can either be a separate document (pdf preferred), or simply inline on your assignment web page. Your document should address the following:

   (a) Profiling

   For this assignment we will address data dependence stalls, previously ignored in assignment 1. For this assignment, you can safely ignore icache and resource conflicts.

   **See the lecture 8 slides for information about data dependencies and stalls.**

   Run your program with the basic single-thread TRaX machine, adding the simulator argument `--profile`:

   ```
   --num-thread-procs 1
   --num-icaches 1
   --num-icache-banks 1
   --config-file ../samples/configs/tiny.config
   --l1-off
   --l2-off
   --disable-usimm
   --no-scene
   --load-assembly <path to your ray tracer rt-llvm.s>
   --profile
   ```

   `--profile` will cause simtrax to generate a file `profile.out` containing the input assembly file with performance data on each instruction. The line number and instruction from the input assembly file appears first, followed by two numbers: the number of times that instruction was executed, and the number of data-dependence stalls that instruction caused. The assembly instruction and profile data are separated by tabs.

   i. The profile output is best viewed in a spreadsheet by copy/pasting the data in to tab-separated columns. Sort the data by the "Data Stall Cycles" column. Which instruction causes the most data dependence stalls? For simplicity, consider only the following instruction types: `FPDIV`, `DIV`, `FPINV`, `FPINVSQRT`, `FPSQRT`.

   ii. Open the original assembly file and go to the line number of the instruction identified in the previous part. Identify the region of code involved in the dependency and list it here (copy/paste the block of instructions starting from the one that causes the stall, to the first instruction requiring its result). Judging by the latency of that instruction defined in `tiny.config`, how many data dependence stall cycles do you expect this sequence of instructions to generate? Explain your reasoning. If you are unsure which functional unit an instruction is handled by (and thus its latency), see: `https://code.google.com/p/simtrax/wiki/ISA`

iii. Does your expected latency per execution of the assembly instruction from the previous part agree with the profile information? Explain.

(b) Code reordering

Data dependence stalls are a function of the latency of an instruction, and the amount of time passed before its result is required. If an instruction takes N cycles, but the result is not needed until N cycles have passed, no stalls will occur. The compiler can attempt to issue unrelated instructions during those N cycles, but is not always successful.

i. Starting with the assembly instruction identified as causing the most data stalls in the previous section, try to reorder the nearby instructions to reduce the number of stalls. Note that this may also require altering which registers certain instructions read/write, in order to avoid dependencies. Consult `https://code.google.com/p/simtrax/wiki/ISA` for a complete list of every instruction's dependencies (inputs and outputs).
**Note that it may not be possible to correctly reorder any particular region of code such that stalls are reduced.**
See the lecture 8 slides for an example of code reordering.

ii. Were you able to reduce the number of data stalls with code reordering?

- If yes, show the relevant instructions from the original code and the reordered code. How much did performance increase?
- If not, show the relevant instructions and explain the restrictions preventing any effective code reordering.

3. **TRaX Output** (10 points)

   Provide the simtrax output running your program with the simtrax options specified above. If your code reordering was successful, also provide the modified assembly file, and the simtrax output using this modified file.

4. **Code listing** (10 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.

5. **Creative Image** (10 points): Using spheres and point-light sources, produce an image of your choice that shows off your ray tracer and your creative skills.

# What to turn in:

By midnight on the due date, you should send e-mail to teach-cs6958@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
   (c) Analysis (see above)
   (d) TRaX output (see above)
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.