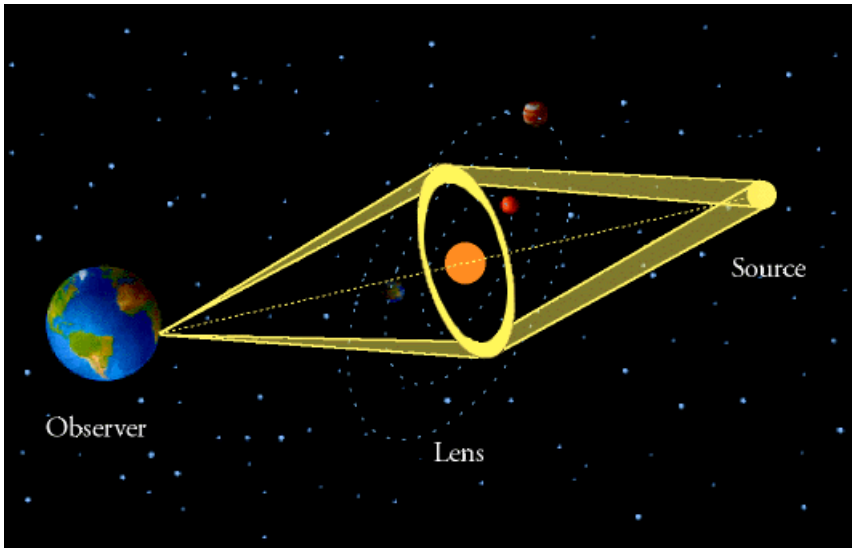
159.735 Assignment 4 - Additional Notes

The diagram below shows the geometry of gravitational lensing. The "lens" is a star system with one or more stars. The "source" is a background star that gets lensed by the lens system.



Imagine a plane located at the lens, the "lens plane", and another plane located at the source, the "source plane". Both the lens and the source plane are perpendicular to the line-of-sight. We are going to backwards trace light rays from the lens plane back to the source plane. This is governed by the "lens equation" which, for a given lens configuration, maps a position (x_l, y_l) on the lens plane to a position (x_s, y_s) on the source plane.

The positions of the lens stars are fixed throughout and are stored on the arrays **`xlens`** and **`ylens`**. Also needed are the "mass fractions" of the lens components—these are stored in the array **`eps`**.

In the startup code, the lens plane has boundaries from -2 to +2 in both x and y axes. This plane is divided into 801×801 pixels derived from the lens scale given in the code (you make bigger images by changing the lens scale to a smaller value). In completing the sequential code, you are asked to fill in the body of the double loop over these pixels.

```
for (int iy = 0; iy < npixy; ++iy)
for (int ix = 0; ix < npixx; ++ix) {
...
}
```

For each pixel index number, we need to scale it to a physical value in the range -2 to 2. This is not complex maths, just scaling and proportionality which we encounter in everyday life. So it need not be confusing, i.e.

```
yl = YL1 + iy * lens_scale;
xl = XL1 + ix * lens_scale;
```

Given the position on the lens plane, we use the lens equation to find the corresponding position on the source plane. This is all encoded in the function **shoot()** and you can treat it as a black box. Note that the source plane positions are passed by reference:

```
shoot(xs, ys, xl, yl, xlens, ylens, eps, nlenses);
```

Now that we have the position of the light ray on the source plane, we now test to see if it lands within the source star—the location of which is fixed in the code. If it does land within the source star, then the lens plane with pixel indices **ix** and **iy** form part of the lens image of the source star. So we colour this pixel in, using the limb darkening formula, ie

```
xd = xs - xsrc;
yd = ys - ysrc;
sep2 = xd * xd + yd * yd;
if (sep2 < rsrc2) {
  mu = sqrt(1 - sep2 / rsrc2);
  lensim(iy, ix) = 1.0 - ldc * (1 - mu);
}
```

If the ray lands outside the source star radius, then we leave the lens image blank at that pixel.

NOTE

- For the GPU version of this, you will need to implement a version of the `shoot()` function that runs on the device. Remember, you cannot call a host function from the device.
- The C++ array class implemented in `array.hxx` cannot be implemented on the device. You will need to store the lens image on the device as a one dimensional array that is accessed via a `float*` pointer. However, with careful use of pointers, you can copy from device to an instance of the `Array<>` class on the host.
- Remember, at the simplest level, CUDA will organize its threads into blocks with a maximum of 1024 threads per block (for the cards in the lab workstations). Each block has a unique ID given by `blockIdx.x` and each thread within within a block has an ID `threadIdx.x` which is unique within a block. To get a thread ID that is unique over the entire grid of threads, you can do something like:

  ```
  int n = blockDim.x * blockIdx.x + threadIdx.x;
  ```

  This thread ID can then map to a particular pixel location on your lens image.