Canny Edge Detection:

The Canny Edge Detector is one of the most commonly used image processing tools, detecting edges in a very robust manner. It is a multi-step process, which can be implemented on the GPU as a sequence of filters. Many of these intermediate filters are those I have already asked you to implement for other parts of this assignment! The following steps are also outlined in various online articles (e.g., Wikipedia (http://en.wikipedia.org/wiki/Canny_edge_detector)) if you prefer a different treatment.

If you would like to compare your results to a C implementation (or double check that you understand the algorithm), you can download an implementation here: ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src

Step 0: Convert to Grayscale

There is no reason why you could not do Canny edge detection on a color image, but I encourage you to first convert the image to grayscale using some sort of RGB→grayscale or RGB→luminance conversion (such as the one I assigned for this assignment).

Step 1: Noise Reduction

Usually noise reduction implies some sort of blurring operation.... Most people use a Gaussian filter to do this. One suggestion is to use the following 5×5 filter:

$$\frac{1}{159} * \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Note that this 5×5 filter is roughly equivalent to the Gaussian filter I asked you to implement for this assignment, using a $\sigma \approx 1.4$.

Step 2: Compute Gradient Magnitude and Angle

Compute the derivatives ($D_x(x, y)$ and $D_y(x, y)$) of the image in the x and y directions. You can do this exactly the same way I suggested when performing the Sobel filter, i.e., use central differencing using the following 3×3 kernels:

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

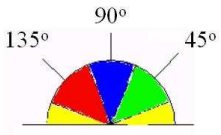
Then compute the gradient magnitude:

$$D = \sqrt{D_x^2(x, y) + D_y^2(x, y)} \tag{1}$$

and the angle of the gradient:

$$\theta = \arctan\left(\frac{D_x(x, y)}{D_y(x, y)}\right) \tag{2}$$

Compute θ' by rounding the angle θ to one of four directions $0°$, $45°$, $90°$, or $135°$. Obviously for edges, $180° = 0°$, $225° = 45°$, etc. This means θ in the ranges $[-22.5°...22.5°]$ and $[157.5°...202.5°]$ would "round" to $\theta' = 0°$. For a pictoral representation, each edge take on one of for colors: Here, the colors would repeat on the lower half of the circle



(green around $225°$, blue around $270°$, and red around $315°$). This image taken from: http://www.pages.drexel.edu/ weg22/can_tut.html

Step 3: Non-Maximum Surpression

As you might have noticed when using the Sobel filter, the edges it finds can be either very thick or very narrow depending on the intensity across the edge and how much the image was blurred first. One would like to have edges that are only one pixel wide. The "non-maximal surpression" step keeps only those pixels on an edge with the highest gradient magnitude. These maximal magnitudes should occur right at the edge boundary, and the gradient magnitude should fall off with distance from the edge.

So, three pixels in a 3×3 around pixel (x, y) are examined:

- If $\theta'(x, y) = 0°$, then the pixels $(x + 1, y)$, (x, y), and $(x - 1, y)$ are examined.
- If $\theta'(x, y) = 90°$, then the pixels $(x, y + 1)$, (x, y), and $(x, y - 1)$ are examined.
- If $\theta'(x, y) = 45°$, then the pixels $(x + 1, y + 1)$, (x, y), and $(x - 1, y - 1)$ are examined.
- If $\theta'(x, y) = 135°$, then the pixels $(x + 1, y - 1)$, (x, y), and $(x - 1, y + 1)$ are examined.

If pixel (x, y) has the highest gradient magnitude of the three pixels examined, it is kept as an edge. If one of the other two pixels has a higher gradient magnitude, then pixel (x, y) is not on the "center" of the edge and should not be classified as an edge pixel.

Step 4: Hysteresis Thresholding

Some of the edges detected by Steps 1–3 will not actually be valid, but will just be noise. We would like to filter this noise out. Eliminating pixels whose gradient magnitude D falls below some threshold removes the worst of this problem, but it introduces a new problem.

A simple threshold may actually remove valid parts of a connected edge, leaving a disconnected final edge image. This happens in regions where the edge's gradient magnitude fluctuates between just above and just below the threshold. *Hysteresis* is one way of solving this problem. Instead of choosing a single threshold, two thresholds t_{high} and t_{low} are used. Pixels with a gradient magnitude $D < t_{low}$ are discarded immediately. However, pixels with $t_{low} \leq D < t_{high}$ are only kept if they form a continuous edge line with pixels with high gradient magnitude (i.e., above t_{high}).

This is actually the tricky part to implement on a GPU – to do it completely correctly is difficult (and it might actually be an open research problem, if you're looking for a possible research project). However, for this assignment you can implement a partially correct version:

- If pixel (x, y) has gradient magnitude less than t_{low} discard the edge (write out black).
- If pixel (x, y) has gradient magnitude greater than t_{high} keep the edge (write out white).
- If pixel (x, y) has gradient magnitude between t_{low} and t_{high} and any of its neighbors in a 3×3 region around it have gradient magnitudes greater than t_{high}, keep the edge (write out white).
- If none of pixel (x, y)'s neighbors have high gradient magnitudes but at least one falls between t_{low} and t_{high}, search the 5×5 region to see if any of these pixels have a magnitude greater than t_{high}. If so, keep the edge (write out white).
- Else, discard the edge (write out black).

If you want to do this step "correctly," come talk to me to get some ideas.