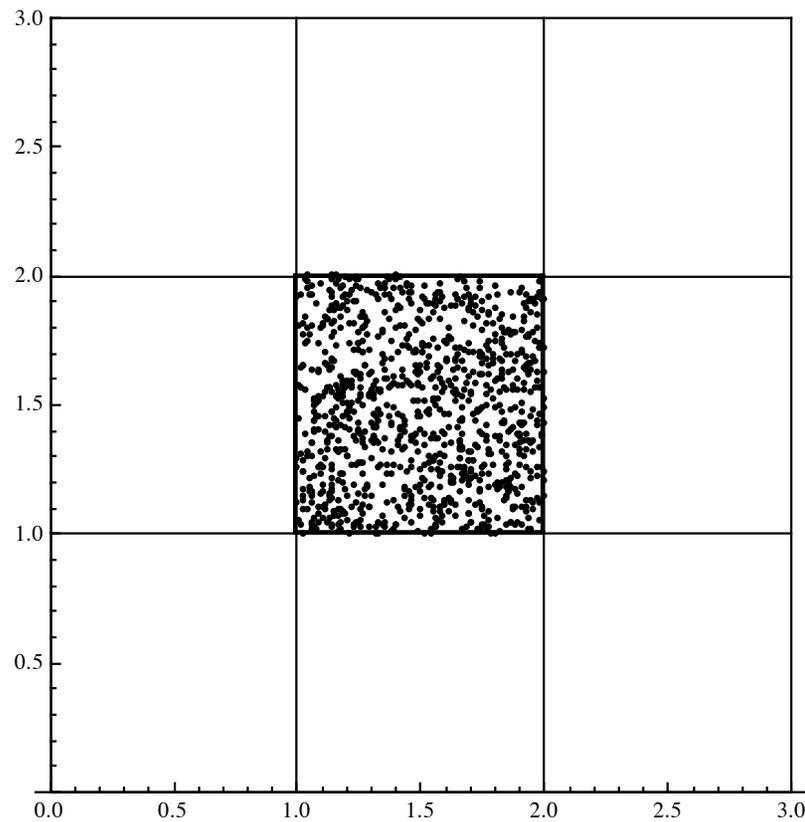


Repixellization, with an analytic solution for pixel overlap

(Still not brain surgery)

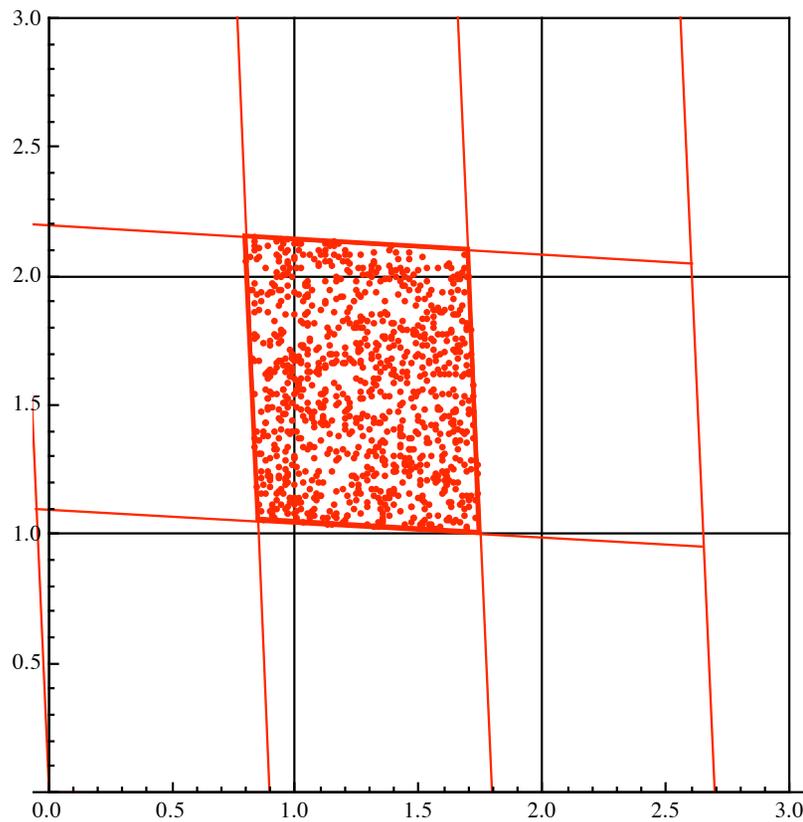
Paul Stankus, Apr. 2, 2008

Updated Jun. 11, 2008



First we look at just one pixel in some original grid in some rectangular sky coordinates.

We only have one number for the total amount of light in this pixel, so if we don't make any assumptions about how the brightness varies within the pixel then we must approximate it as uniform. We can represent this as a (large) set of points chosen randomly within the pixel's box.

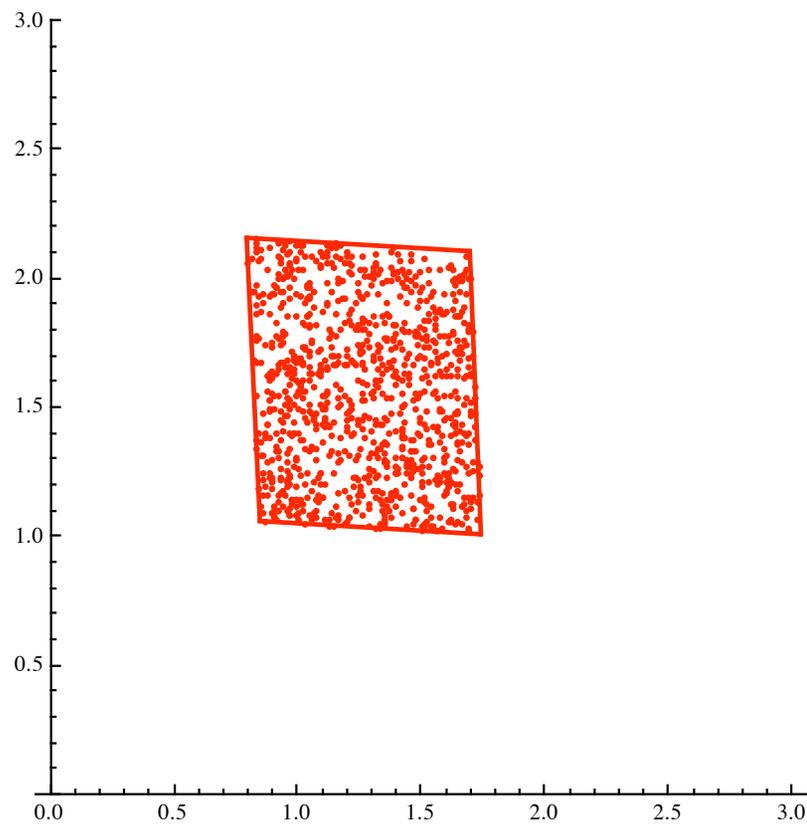


To apply a shear we just multiply the (x,y) vector of each point by a shear matrix; here

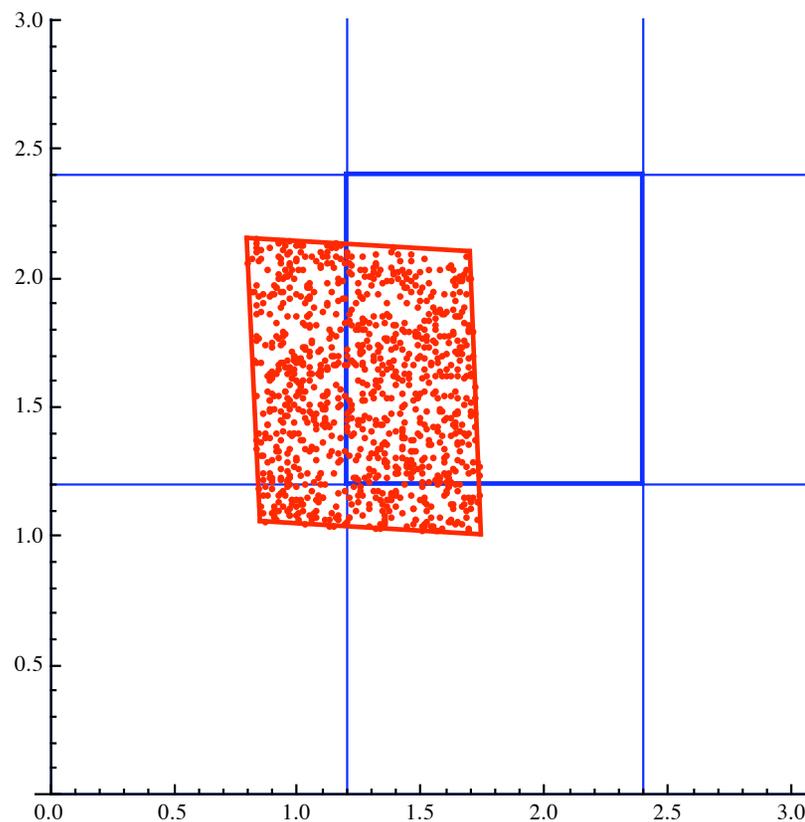
$$\begin{bmatrix} 0.90 & -0.05 \\ -0.05 & 1.1 \end{bmatrix}$$

(Q: what is the magnitude of the shear generated by this matrix?)

The **red points** are the new locations of the original black points, and the red grid is the transformation of the original black pixel grid.

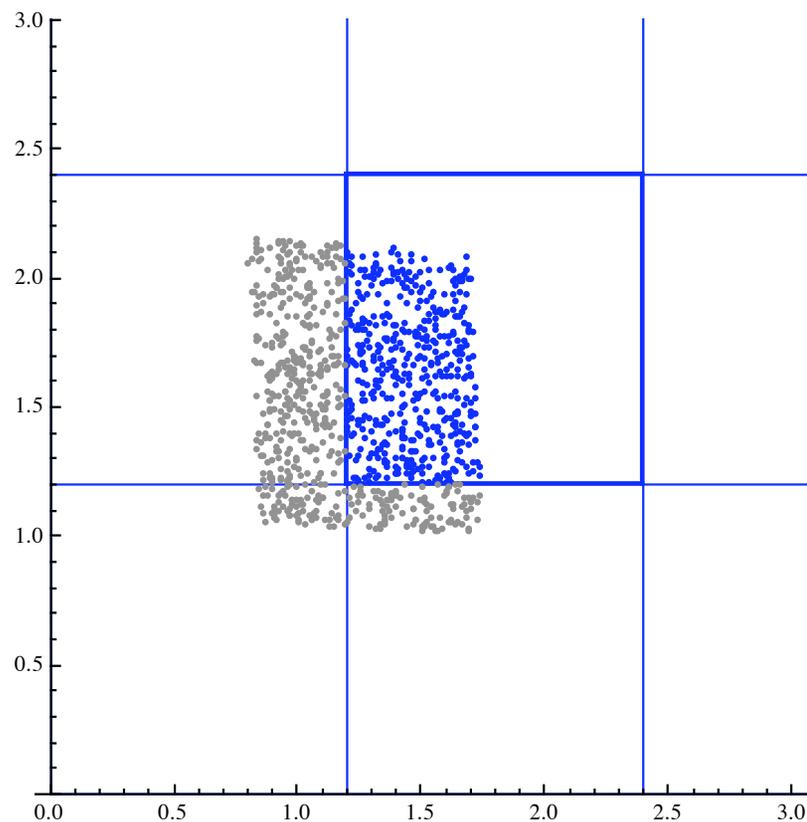


Forgetting about the original pixel grid, the **red points** and their surrounding **red parallelogram** are now our best approximation to what the sheared image of the original pixel would look like.



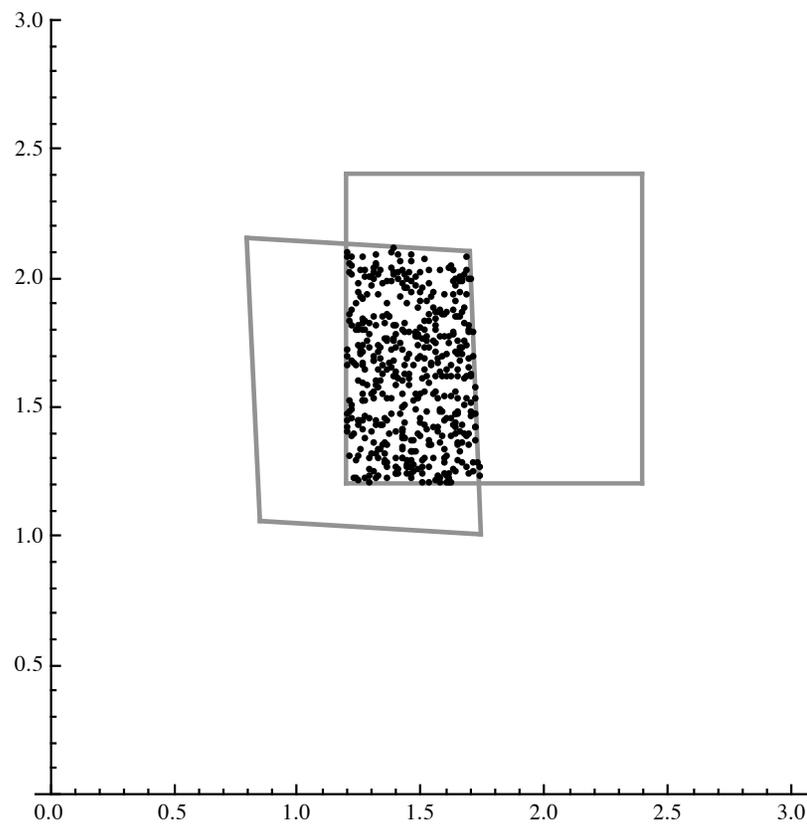
Now suppose we want to re-pixelate the **sheared image** onto a **new grid**, with somewhat lower resolution/bigger pixels, as shown in **blue**. (Here the new grid is chosen to be aligned with the same sky coordinate system as the original pixel grid; this is not strictly necessary, but the new grid should at least be rectangular.)

The question is, how do we distribute the brightness of the **original pixel**, now sheared, into the **new larger pixels**?



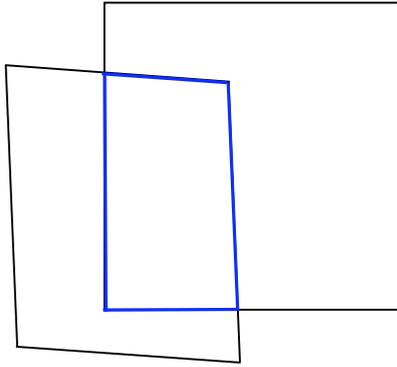
For a particular pixel in the new grid, like the one [highlighted here](#), the obvious choice is to fill it with a number proportional to how many of the sheared points fall into the new pixel's box.

In essence, we transfer to the new pixel a *fraction* of the original pixel's brightness, that fraction being what proportion of the area of the sheared original box is also within the new box (basically the ratio $\text{blue}/(\text{blue}+\text{gray})$ in the color scheme shown here).



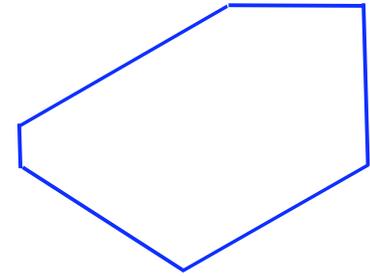
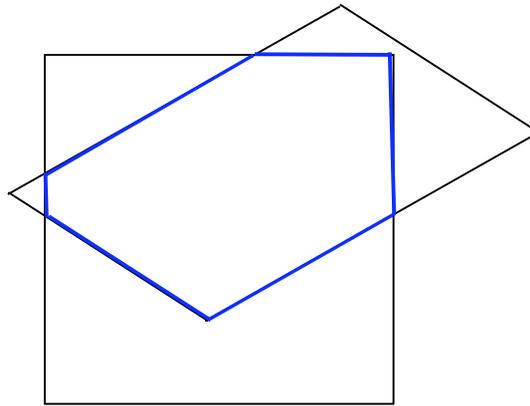
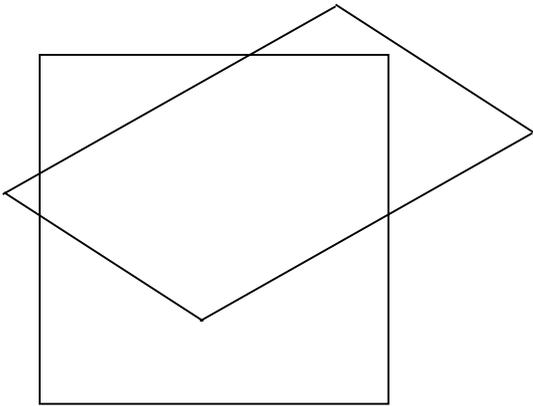
Determining the area of the overlap between two polygons by counting random points which land in both is reliable (it's the Metropolis algorithm for integration), but slow and kind of dumb; plus, you have to worry about exactly how many points you need so as not to introduce a significant statistical sampling error.

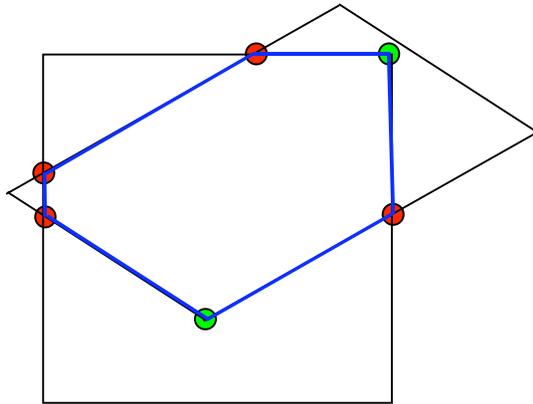
Much faster would be a general formula to calculate the overlap area between two polygons; see next slides of one approach.



A simple analytical approach is to (1) Find the polygon defined by the overlap of the two pixel boundaries, then (2) Calculate its area, and (3) Use the ratio of that area to the area of the original pixel as the fractional weighting to re-distribute the original pixel's intensity to the new pixel.

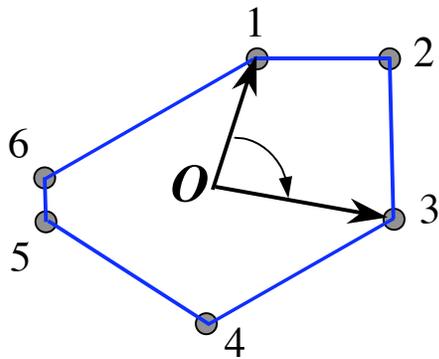
Though the overlap polygon could be somewhat complicated in shape, we can be assured that it is *convex* as long as the two original boundary polygons are both convex.



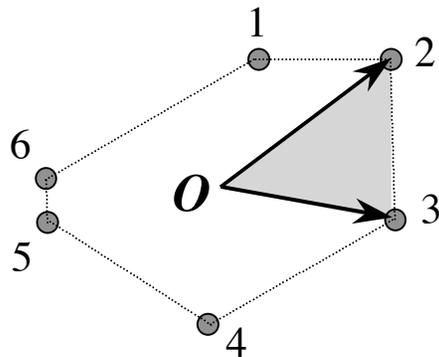


Here's a simple, reasonably fast algorithm for finding the overlap polygon between two convex polygons and then calculating its area:

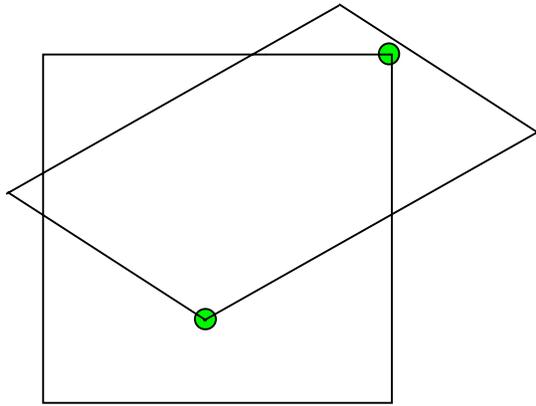
1. All the vertices of the overlap polygon are either (i) Intersections between side segments of the two parent polygons (**red**), or (ii) Vertices of the original polygons which are inside the other parent (**green**). If there are no such points of either type then the parent polygons do not overlap and the overlap area is (of course) zero.



2. Compile a list of all points of the two types, and these will be the vertices of the overlap polygon. Pick a point O which is known to be within the overlap polygon -- the 2-D average of all the vertices should be fine -- and then sort & order the vertices according to their opening angle relative to O and to one point chosen as the first vertex.

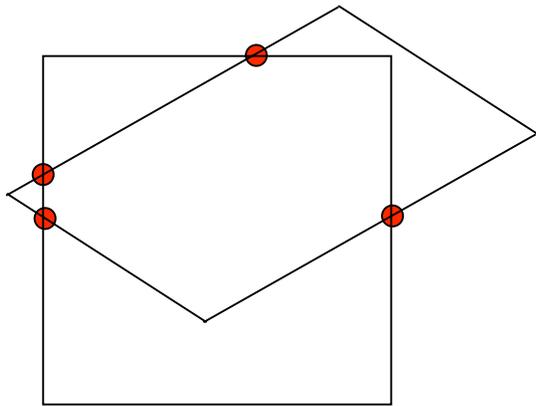


3. The area of the polygon is then the sum of the areas of all the triangles $(O, vertex_n, vertex_{n+1})$, which are fast/easy to calculate using the ordinary vector cross product. (Note that for this calculation the origin/anchor point O does not have to be within the polygon, though if it is taken to be outside the polygon then one has to be careful about accounting for triangles with negative area.)



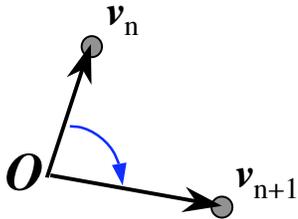
This algorithm may arguably be overkill for the specific case of the overlap between a rectangle and a parallelogram, but it is simple and reasonably fast. To implement it we need at least these three utility subroutines:

1. Given a polygon and a point, decide whether the point is or is not within the polygon, with a specific default if the point lies on an edge or is one of the vertices.



2. Given two line segments, determine if they intersect along their lengths, and if they do then return the intersection point. Again, we need a specific default for the cases that one of the segments' endpoints lies on the other segment, or if the two segments are identical and/or contained within the same line.

3. Given an origin point O and an ordered pair of vertices v_n and v_{n+1} , determine the opening angle between the two vectors (O, v_n) and (O, v_{n+1}) within a $0-2\pi$ continuous range.



These should be pretty straightforward if anyone's up for a coding exercise. Note that if you want to get fancy, the general subject of polygon clipping in vector graphics is very highly developed; see <http://www.cs.fit.edu/~wds/classes/graphics/Clip/clip/clip.html> for just one example.