Winding Number Algorithm

The winding number algorithm was the original algorithm tested for loops. Where P is the point we are testing, C is the continuous closed curve, and n is the number of points on the boundry of C. We compute the winding number like:

$$\forall V_i \in C, i \leq n-1$$

$$wn(P,C) = \frac{1}{2\pi}\sum_{i=1}^{n-1} arccos(\frac{(V_i - P) * (V_{i+1} - P)}{|V_i - P||V_{i+1} - P|})$$

Code

```
def winding_num(p,domain,total, min_max):

    x_max = min_max[0]
    y_max = min_max[1]
    x_min = min_max[2]
    y_min = min_max[3]

    if p[0] < x_min or p[1] < y_min:
        return 0

    if p[0] > x_max or p[1] > y_max:
        return 0

    for pos in range(0,len(domain)):
        if pos < len(domain)-1:
            point_1 = domain[pos]
            point_2 = domain[pos+1]
            vector_diff_1 = (p[0] - point_1[0], p[1]- point_1[1])
            vector_diff_2 = (p[0] - point_2[0], p[1]- point_2[1])
            dot_prod = vector_diff_1[0]*vector_diff_2[0] + vector_diff_1[1]*vector_diff_2[1
            vector_length_1 = math.sqrt(vector_diff_1[0]**2 + vector_diff_1[1]**2)
            vector_length_2 = math.sqrt(vector_diff_2[0]**2 + vector_diff_2[1]**2)
            denom = vector_length_1*vector_length_2
            value = float(dot_prod/denom)
            calculation = np.arccos(value)
            total += calculation
    return (1/(2*np.pi))*total
```

If a point is inside the domain C then the winding number has to be greater than or equal to 1.

Ray Casting Algorithm

This algorithm is the fastest algorithm out of the ones tested so far. Where P is the point we are testing, C is the continuous closed curve, and n is the number of points on the boundry of C. Then, $\forall V_i \in C, i <= n-1$ If we are going to assume V_i has the bigger y value. These are the cases where an intersection would occur. $P_x < \min V_i.xV_{i+1}.x$ If the statement before is false. Then the variable α is used to be a configurable small value and we look at, $|(V_i.x - V_{i+1}.x)| > \alpha$. If that's true we set $\mathrm{A} = \frac{(V_{i+1}.y - V_i.y)}{(V_{i+1}.x - V_i.x)}$. If the statement prior is false then, $\mathrm{A} = \beta$ where β is an abirtrarly large number. The next step is to define B, if $|V_i.x - P.x| > \alpha$ then $\mathrm{B} = \frac{P.y - V_i.y}{P.x - V_i.x}$. If $|V_i.x - P.x| < \alpha$ then, $\mathrm{B} = \beta$. After we define A and B, if B >= A, then an intersection occurs.

We take the steps above for each set of points within the domain C. Let I represent the total amount of intersections. If $I \bmod 2 = 1$, that tells us the point is within the circle. Otherwise, the point is outside the circle.

Code

```
def ray_casting_alg(domain, p, prior_intersections, min_max):
    global _eps
    global _huge
    global _tiny

    intersect = prior_intersections
    x_max = min_max[0]
    y_max = min_max[1]
    x_min = min_max[2]
    y_min = min_max[3]

    if p[0] < x_min or p[1] < y_min:
        return 0

    if p[0] > x_max or p[1] > y_max:
        return 0

    for pos in range(0, len(domain)):
        if pos < len(domain)-1:
            p_1 = domain[pos]
            p_2 = domain[pos+1]
            if p_1[1] > p_2[1]:
                p_1 = p_2
                p_2 = p_1
            if p[1] == p_1[1] or p[1] == p_2[1]:
                p = (p[0], p[1] + _eps)
```

```
        if (p[1] > p_1[1] or p[1] <  p_2[1]) or (p[0] > max(p_1[0], p_2[0])):
            pass

        if p[0] < min(p_1[0], p_2[0]):
            intersect += 1
        else:
            if abs(p_1[0] - p_2[0]) > _tiny:
                m_red = (p_2[1] - p_1[1]) / (float(p_2[0] - p_1[0]))
            else:
                m_red = _huge
            if abs(p_1[0] - p[0]) > _tiny:
                m_blue = (p[1] - p_1[1]) / (float(p[0] - p_1[0]))
            else:
                m_blue = _huge
            if m_blue >= m_red:
                intersect += 1
return intersect
```