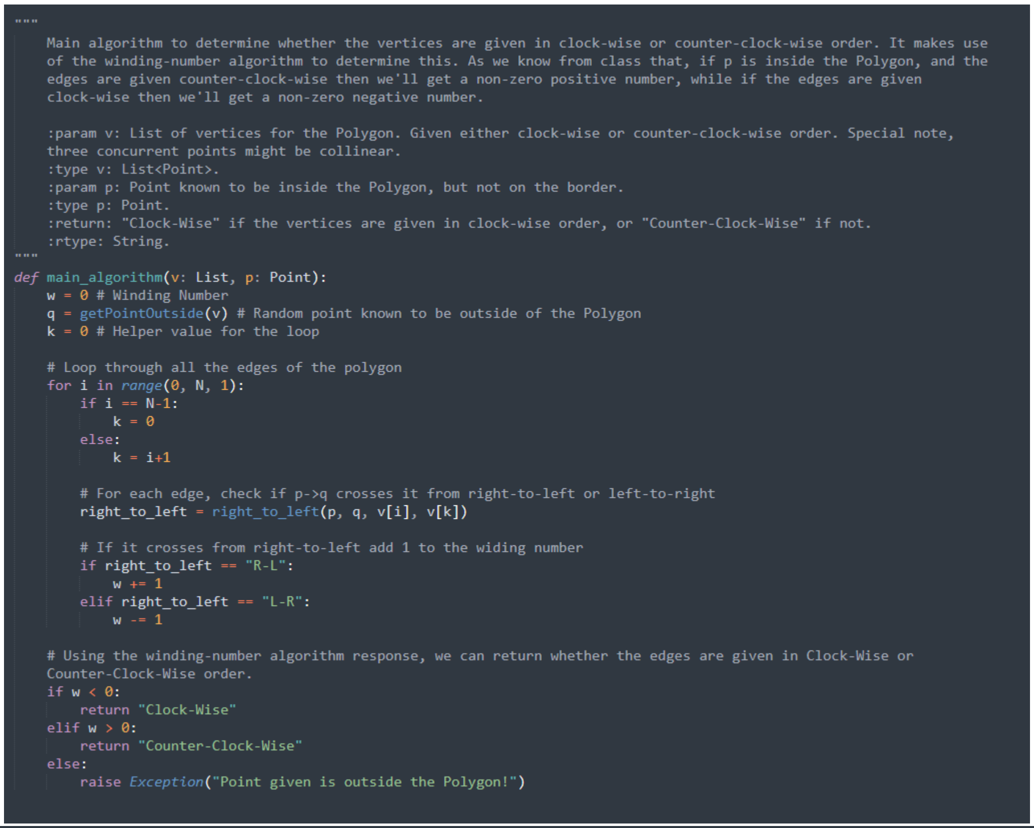
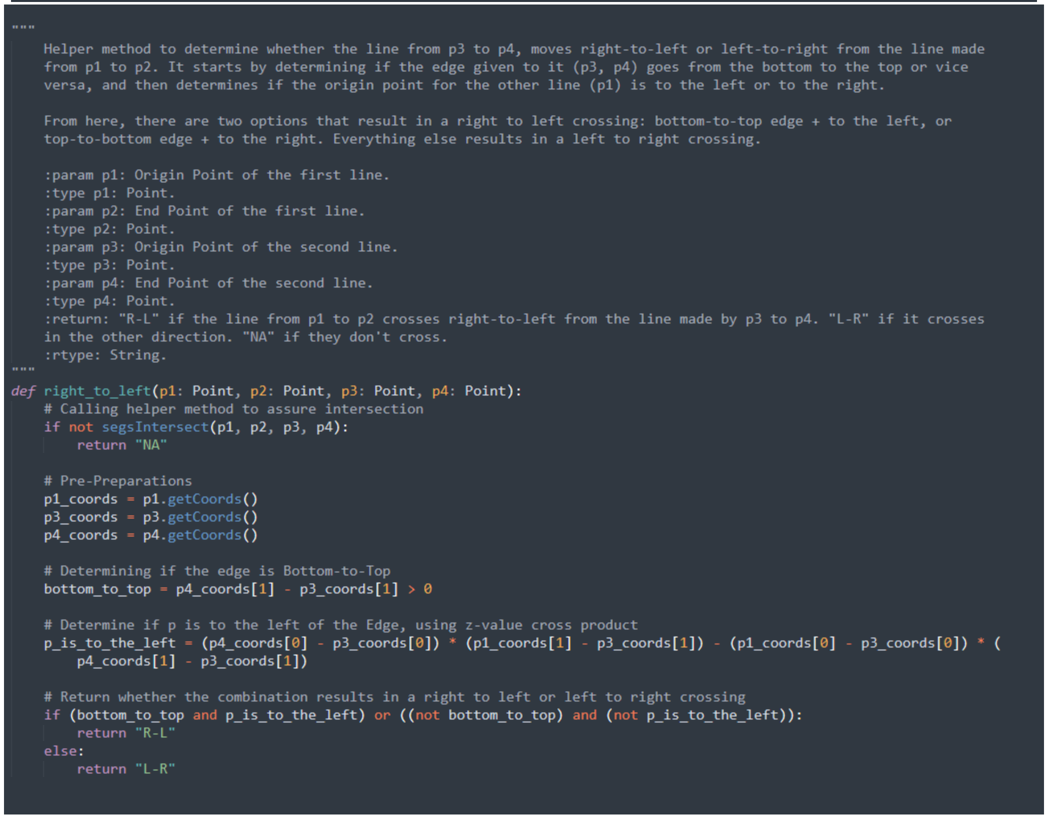
CS 480 – Fall 2021, PSet 2

Answer 1

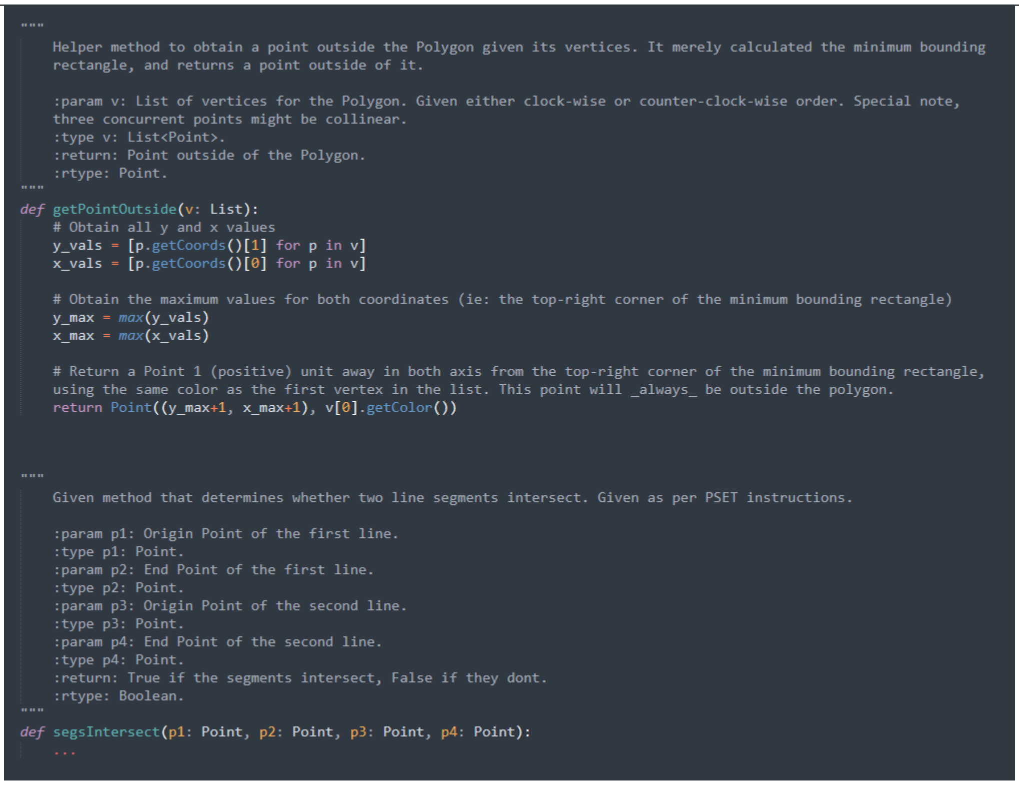
(a)



```python
"""
    Main algorithm to determine whether the vertices are given in clock-wise or counter-clock-wise order. It makes use
    of the winding-number algorithm to determine this. As we know from class that, if p is inside the Polygon, and the
    edges are given counter-clock-wise then we'll get a non-zero positive number, while if the edges are given
    clock-wise then we'll get a non-zero negative number.

    :param v: List of vertices for the Polygon. Given either clock-wise or counter-clock-wise order. Special note,
    three concurrent points might be collinear.
    :type v: List<Point>.
    :param p: Point known to be inside the Polygon, but not on the border.
    :type p: Point.
    :return: "Clock-Wise" if the vertices are given in clock-wise order, or "Counter-Clock-Wise" if not.
    :rtype: String.
"""
def main_algorithm(v: List, p: Point):
    w = 0 # Winding Number
    q = getPointOutside(v) # Random point known to be outside of the Polygon
    k = 0 # Helper value for the loop

    # Loop through all the edges of the polygon
    for i in range(0, N, 1):
        if i == N-1:
            k = 0
        else:
            k = i+1

        # For each edge, check if p->q crosses it from right-to-left or left-to-right
        right_to_left = right_to_left(p, q, v[i], v[k])

        # If it crosses from right-to-left add 1 to the widing number
        if right_to_left == "R-L":
            w += 1
        elif right_to_left == "L-R":
            w -= 1

    # Using the winding-number algorithm response, we can return whether the edges are given in Clock-Wise or
    Counter-Clock-Wise order.
    if w < 0:
        return "Clock-Wise"
    elif w > 0:
        return "Counter-Clock-Wise"
    else:
        raise Exception("Point given is outside the Polygon!")
```



```python
"""
    Helper method to determine whether the line from p3 to p4, moves right-to-left or left-to-right from the line made
    from p1 to p2. It starts by determining if the edge given to it (p3, p4) goes from the bottom to the top or vice
    versa, and then determines if the origin point for the other line (p1) is to the left or to the right.

    From here, there are two options that result in a right to left crossing: bottom-to-top edge + to the left, or
    top-to-bottom edge + to the right. Everything else results in a left to right crossing.

    :param p1: Origin Point of the first line.
    :type p1: Point.
    :param p2: End Point of the first line.
    :type p2: Point.
    :param p3: Origin Point of the second line.
    :type p3: Point.
    :param p4: End Point of the second line.
    :type p4: Point.
    :return: "R-L" if the line from p1 to p2 crosses right-to-left from the line made by p3 to p4. "L-R" if it crosses
    in the other direction. "NA" if they don't cross.
    :rtype: String.
"""
def right_to_left(p1: Point, p2: Point, p3: Point, p4: Point):
    # Calling helper method to assure intersection
    if not segsIntersect(p1, p2, p3, p4):
        return "NA"

    # Pre-Preparations
    p1_coords = p1.getCoords()
    p3_coords = p3.getCoords()
    p4_coords = p4.getCoords()

    # Determining if the edge is Bottom-to-Top
    bottom_to_top = p4_coords[1] - p3_coords[1] > 0

    # Determine if p is to the left of the Edge, using z-value cross product
    p_is_to_the_left = (p4_coords[0] - p3_coords[0]) * (p1_coords[1] - p3_coords[1]) - (p1_coords[0] - p3_coords[0]) * (
        p4_coords[1] - p3_coords[1])

    # Return whether the combination results in a right to left or left to right crossing
    if (bottom_to_top and p_is_to_the_left) or ((not bottom_to_top) and (not p_is_to_the_left)):
        return "R-L"
    else:
        return "L-R"
```


```
"""
    Helper method to obtain a point outside the Polygon given its vertices. It merely calculated the minimum bounding
    rectangle, and returns a point outside of it.

    :param v: List of vertices for the Polygon. Given either clock-wise or counter-clock-wise order. Special note,
    three concurrent points might be collinear.
    :type v: List<Point>.
    :return: Point outside of the Polygon.
    :rtype: Point.
"""
def getPointOutside(v: List):
    # Obtain all y and x values
    y_vals = [p.getCoords()[1] for p in v]
    x_vals = [p.getCoords()[0] for p in v]

    # Obtain the maximum values for both coordinates (ie: the top-right corner of the minimum bounding rectangle)
    y_max = max(y_vals)
    x_max = max(x_vals)

    # Return a Point 1 (positive) unit away in both axis from the top-right corner of the minimum bounding rectangle,
    using the same color as the first vertex in the list. This point will _always_ be outside the polygon.
    return Point((y_max+1, x_max+1), v[0].getColor())



"""
    Given method that determines whether two line segments intersect. Given as per PSET instructions.

    :param p1: Origin Point of the first line.
    :type p1: Point.
    :param p2: End Point of the first line.
    :type p2: Point.
    :param p3: Origin Point of the second line.
    :type p3: Point.
    :param p4: End Point of the second line.
    :type p4: Point.
    :return: True if the segments intersect, False if they dont.
    :rtype: Boolean.
"""
def segsIntersect(p1: Point, p2: Point, p3: Point, p4: Point):
    ...
```

Answer 2

(a)

$$M = \begin{bmatrix} -\frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} & a \\ 0 & -2 & 0 & b \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) The transformations that occur when applying M to a 3D point are: Scaling, Transposing, and Rotation along the y Axis. Each use the matrices shown below:

$$T = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y\theta_y = \begin{bmatrix} -\frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These are multiplied in the order as shown above: Translation * Scaling * Rotation. So they equal the homogeneous matrix M.

Answer 3

The matrix used in the shearing of the letter 'h' in the image shown follows this formula:

$$M_{var} = \begin{bmatrix} 1 & \frac{\alpha}{y_{mult}} & -\alpha \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If we replace these variables we get:

$$M_{final} = \begin{bmatrix} 1 & \frac{2}{4} & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We can confirm this by applying the matrix to some points:

$$p_1 = \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix}$$

$$p_1' = \begin{bmatrix} 1 & \frac{2}{4} & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix}$$

$$p_2 = \begin{bmatrix} 3 \\ 8 \\ 1 \end{bmatrix}$$

$$p_2' = \begin{bmatrix} 1 & \frac{2}{4} & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 8 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 1 \end{bmatrix}$$

$$p_3 = \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix}$$

$$p_3' = \begin{bmatrix} 1 & \frac{2}{4} & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \\ 1 \end{bmatrix}$$

As you see, all points land where they are expected to land on the new sheared 'h'.

Answer 4

(a) Assuming the following formulas with rotation angle θ and axis $v = (x, y, z)$:

$q = (q_0, q_1, q_2, q_3)$

$q_0 = cos(\frac{\theta}{2})$

$q_1 = x \times sin(\frac{\theta}{2})$

$q_2 = y \times sin(\frac{\theta}{2})$

$q_3 = z \times sin(\frac{\theta}{2})$

We can get that:

$cos(\frac{\theta}{2}) = 0 \equiv \theta = 180 \vee \theta = 540$

$x \times sin(\frac{\theta}{2}) = 0 \equiv x = 0 \vee sin(\frac{\theta}{2}) = 0 \equiv x = 0 \vee \theta = 0 \vee \theta = 360$

$y \times sin(\frac{\theta}{2}) = \frac{1}{\sqrt{2}} \equiv (y = 1 \wedge \theta = 90) \vee (y = 1 \wedge \theta = 180) \vee (y = \frac{1}{\sqrt{2}} \wedge \theta = 180)$

$z \times sin(\frac{\theta}{2}) = \frac{1}{\sqrt{2}} \equiv (z = 1 \wedge \theta = 90) \vee (z = 1 \wedge \theta = 180) \vee (z = \frac{1}{\sqrt{2}} \wedge \theta = 180)$

As we can see, there is an option where all three values match up, when $\theta = 180$ and $v = (0, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$. This also checks out with regards to v being a unit vector, as $|v| = \sqrt{0^2 + \frac{1}{\sqrt{2}}^2 + \frac{1}{\sqrt{2}}^2} = 1$.

(b) Assuming the same formulas as above, we get that:

$cos(\frac{\theta}{2}) = \frac{1}{\sqrt{2}} \equiv \theta = 90 \vee \theta = 630$

Since we are closed in by this first angle, lets assume that $\theta = 90$. Calculating the rest we get:
$x \times sin(\frac{\theta}{2}) = 0 \equiv x \times sin(\frac{90}{2}) = 0 \equiv x \times sin(45) \equiv x \times \frac{1}{\sqrt{2}} = 0 \equiv x = 0$

$y \times sin(\frac{\theta}{2}) = -\frac{1}{2} \equiv y \times sin(\frac{90}{2}) = -\frac{1}{2} \equiv y \times sin(45) = -\frac{1}{2} \equiv y \times \frac{1}{\sqrt{2}} = -\frac{1}{2} \equiv y = \frac{-\frac{1}{2}}{\frac{1}{\sqrt{2}}} = -\frac{\sqrt{2}}{2}$

$z \times sin(\frac{\theta}{2}) = -\frac{1}{2} \equiv z \times sin(\frac{90}{2}) = -\frac{1}{2} \equiv z \times sin(45) = -\frac{1}{2} \equiv z \times \frac{1}{\sqrt{2}} = -\frac{1}{2} \equiv z = \frac{-\frac{1}{2}}{\frac{1}{\sqrt{2}}} = -\frac{\sqrt{2}}{2}$

With that, we get: $\theta = 90$ and $v = (0, -\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2})$, we can again, confirm that v is a unit vector: $|v| = \sqrt{0^2 + (-\frac{\sqrt{2}}{2})^2 + (-\frac{\sqrt{2}}{2})^2} = 1$

(c) Yes! The unit vectors are actually collinear, except one points in one direction, and the other in the opposite direction. But for all intents and purposes this means that they represent the same axis, which means that the only difference between the two, is the angle of rotation.

Answer 5

There are 5 steps to this transformation. Firstly, translation from C to the Origin (0,0,0). Secondly, rotation to align the local coordinate basis with the world coordinate basis. Then we can finally scale the Polygon along the x axis. After this we repeat the previous steps backwards. We re-rotate to align the local coordinate basis back to what it was, and finally re-translate from the Origin to C.

In order to more easily manipulate the Matrices during rotation, let's call the local coordinate basis vectors u, v, w each representing the x, y, z directions.

With that said, we are left with the following individual Homogeneous matrices:

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_{in} = \begin{bmatrix} - & u^T & - & 0 \\ - & v^T & - & 0 \\ - & w^T & - & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_{out} = \begin{bmatrix} | & | & | & 0 \\ u^T & v^T & w^T & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2 = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Since order matters in Rotations, we must proceed to multiply these matrices in their correct order: $T_1 \times R_{in} \times S \times R_{out} \times T_2$, if so we get:

$$T_1 \times R_{in} = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & -c_x \\ v_x & v_y & v_z & -c_y \\ w_x & w_y & w_z & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(T_1 \times R_{in}) \times S = \begin{bmatrix} u_x & u_y & u_z & -c_x \\ v_x & v_y & v_z & -c_y \\ w_x & w_y & w_z & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x \times u_x & u_y & u_z & -c_x \\ S_x \times v_x & v_y & v_z & -c_y \\ S_x \times w_x & w_y & w_z & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here I'll stop to define three new vectors u_s, v_s, w_s, all of which share the same coordinates as their non-s counterparts with the exception that their x component is scaled by S_x.

$$(T_1 \times R_{in} \times S) \times R_{out} = \begin{bmatrix} u_{sx} & u_{sy} & u_{sz} & -c_x \\ v_{sx} & v_{sy} & v_{sz} & -c_y \\ w_{sx} & w_{sy} & w_{sz} & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_x & v_x & w_z & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_s^T \cdot u & u_s^T \cdot v & u_s^T \cdot w & -c_x \\ v_s^T \cdot u & v_s^T \cdot v & v_s^T \cdot w & -c_y \\ w_s^T \cdot u & w_s^T \cdot v & w_s^T \cdot w & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(T_1 \times R_{in} \times S \times R_{out}) \times T_2 = \begin{bmatrix} u_s^T \cdot u & u_s^T \cdot v & u_s^T \cdot w & -c_x \\ v_s^T \cdot u & v_s^T \cdot v & v_s^T \cdot w & -c_y \\ w_s^T \cdot u & w_s^T \cdot v & w_s^T \cdot w & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} u_s^T \cdot u & u_s^T \cdot v & u_s^T \cdot w & c_x \times (u_s^T \cdot u) + c_y \times (u_s^T \cdot v) + c_z \times (u_s^T \cdot w) - c_x \\ v_s^T \cdot u & v_s^T \cdot v & v_s^T \cdot w & c_x \times (v_s^T \cdot u) + c_y \times (v_s^T \cdot v) + c_z \times (v_s^T \cdot w) - c_y \\ w_s^T \cdot u & w_s^T \cdot v & w_s^T \cdot w & c_x \times (w_s^T \cdot u) + c_y \times (w_s^T \cdot v) + c_z \times (w_s^T \cdot w) - c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With all that done, the Homogeneous Matrix above is the total combined Transformation Matrix for Scaling the Polygon along the u axis, by a factor of S_u