CS 4620 Final Exam

Wednesday 9, December 2009—2$\frac{1}{2}$ hours
Prof. Doug James

Explain your reasoning for full credit.
You are permitted a double-sided sheet of notes.
Calculators are allowed but unnecessary.

Problem 1: Continuity (8 pts)

You have learned about parametric and geometric continuity. For each 2D curve, answer the continuity query as correctly as possible, and provide a *brief* explanation:

(a) Is a circle C^0 continuous?

- **Answer:** Parametric "C" continuity refers to the continuity of the parameterization used. However, since no parameterization has been specified, the question is ambiguous—circle parameterizations may or may not be C^0.

(b) Is a circle G^0 continuous?

- **Answer:** Geometric "G" continuity refers to the continuity of the geometric shape. Yes the circle is G^0 continuous because the curve is connected (unbroken) everywhere.

(c) Is a circle C^∞ continuous?

- **Answer:** Again it is unclear, since no parameterization has been specified. Circle parameterizations may or may not be C^0.

(d) Is a circle G^∞ continuous?

- **Answer:** Yes, because the circle is infinitely smooth. It can also be parameterized by $(x(t), y(t)) = (sin(t), cos(t))$ which is infinitely differentiable.

(e) Is a square C^0 continuous?

- **Answer:** Unclear, since no parameterization has been specified. Parameterizations of the square may or may not be C^0.

(f) Is a square G^0 continuous?

- **Answer:** Yes, because the curve is continuous/unbroken everywhere.

(g) Is a square C^1 continuous?

- **Answer:** Unclear, since no parameterization has been specified. Parameterizations may or may not be C^1, even though there are corners.

(h) Is a square G^1 continuous?

- **Answer:** No, because tangents to the geometric curve have direction discontinuities at the corners.

Problem 2: View Frustum Culling (10 pts)

"View frustum culling" is a technique to avoid drawing (or cull) geometry which is outside the view frustum. To assist with culling, assume that *each object has a bounding sphere* with object-frame center position, $\mathbf{c}_o = (c_x, c_y, c_z, 1)^T$, and radius R_o. Imagine that you know you have an $[l, r] \times [b, t] \times [f, n]$ orthographic viewing volume, and you know each of the matrices ($\mathbf{M}_{vp}$, $\mathbf{M}_{orth}$, $\mathbf{M}_{cam}$, $\mathbf{M}_m$) used to construct the orthographic view transformation which maps points from *object space* to *screen space*:

$$\mathbf{p}_s = \begin{pmatrix} x_s \\ y_s \\ z_c \\ 1 \end{pmatrix} = \mathbf{M}_{vp}\, \mathbf{M}_{orth}\, \mathbf{M}_{cam}\, \mathbf{M}_m\, \mathbf{p}_o = \mathbf{M} \begin{pmatrix} x_o \\ y_o \\ z_o \\ 1 \end{pmatrix}.$$

Derive a simple mathematical test to determine if an object is safely "off screen."

- **Answer:** It suffices to transform the bounding sphere position to the camera frame, and consider whether or not the sphere intersects the view volume. You can assume that the modeling transformation $\mathbf{M}_m$ is a rigid-body transformation, and thus that $\mathbf{M}_{cam}\mathbf{M}_m$ is a rigid-body transformation that maps spheres to spheres. Given the sphere center in the camera frame,

$$\mathbf{c}_{cam} = \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \mathbf{M}_{cam}\mathbf{M}_m\mathbf{c}_o, \qquad (1)$$

we can not cull the geometry if it is within the $R = R_o$-inflated view volume, $[l - R, r + R] \times [b - R, t + R] \times [f - R, n + R]$. Therefore we must draw the geometry only if the following expression is true:

$$(l - R \le X \le r + R) \text{ AND } (b - R \le Y \le t + R) \text{ AND } (f - R \le Z \le n + R). \qquad (2)$$

Extra: If you did not assume that $\mathbf{M}_m$ was a rigid-body transformation, then it suffices to consider the comparison with but replacing the R by sR, where s is the two-norm of the 3-by-3 rotation-like block of $\mathbf{M}_m$.

Problem 3: Rasterizing Curves (15 pts)

In this question you will extend Bresenham's midpoint algorithm for line rasterization to build a DDA-based rasterizer for a *quadratic Bézier curve*. For simplicity you may assume that the curve is parameterized in the form

$$y(x) = y_0 B_0(x) + y_1 B_1(x) + y_2 B_2(x),$$

where

$$B_i(x) = \binom{2}{i} x^i (1-x)^{2-i}$$

are the quadratic Bernstein polynomials. You may even assume that the slope of the curve satisfies $0 \le y'(x) \le 1$.

(a) First, derive the equations needed to use **forward differencing** to evaluate the Bézier curve at unit $\Delta x = 1$ spacings without unnecessary multiplication. (Hint: First convert $y(x)$ to monomial form.)

- **Answer:** First convert to monomial form as suggested:

$$\begin{aligned} y(x) &= y_0 B_0(x) + y_1 B_1(x) + y_2 B_2(x) && (3)\\ &= y_0(1-x)^2 + y_1 2x(1-x) + y_2 x^2 && (4)\\ &= y_0(1-2x+x^2) + y_1(2x-2x^2) + y_2 x^2 && (5)\\ &= (y_0 - 2y_1 + y_2)x^2 + 2(y_1 - y_0)x + y_0 && (6)\\ &\equiv ax^2 + bx + c && (7)\end{aligned}$$

  Forward differences are easier than for the cubic case done in class. The first difference is

$$\begin{aligned} \Delta y(x) &= y(x+1) - y(x) && (8)\\ &= a[(x+1)^2 - x^2] + b[(x+1) - x] && (9)\\ &= a[2x+1] + b && (10)\\ &= 2ax + (a+b), && (11)\end{aligned}$$

  and the second difference is

$$\begin{aligned} \Delta^2 y(x) &= \Delta y(x+1) - \Delta y(x) && (12)\\ &= 2a(x+1) - 2ax && (13)\\ &= 2a, && (14)\end{aligned}$$

  with all higher differences zero. In summary, we must update Δy each increment by just adding $2a$ to it,

$$\Delta y(x+1) = \Delta y(x) + 2a, \qquad (15)$$

  and we update y by adding $\Delta y(x)$,

$$y(x+1) = y(x) + \Delta y(x). \qquad (16)$$

(b) Second, provide pseudocode for a simple DDA rasterizer from $x = x_0$ to $x = x_1$. You need not consider shading, attribute interpolation, or antialiasing—you only need to "turn on" pixels using appropriate calls to `output(x,y)`.

- **Answer:** The pseudocode is identical to the original midpoint code, except with the line-based DDA update for y replaced by the forward-difference update for the Bézier curve. Recall that for line rasterization, each time we incremented `x` or `y` by 1, we updated the residual variable, `d = mx + b - y`, by `m` or 1, respectively:

```
x = ceil(x0)
y = round(m*x + b)
d = m*(x + 1) + b - y
while (x < floor(x1))
     if (d > 0.5)
          y += 1
          d -= 1
     x += 1
     d += m
     output(x, y)
```

- The appropriate `d` variable for our problem is

$$\mathtt{d} = \mathtt{d}(\mathtt{x}, \mathtt{y}) = y(\mathtt{x}) - \mathtt{y}. \tag{17}$$

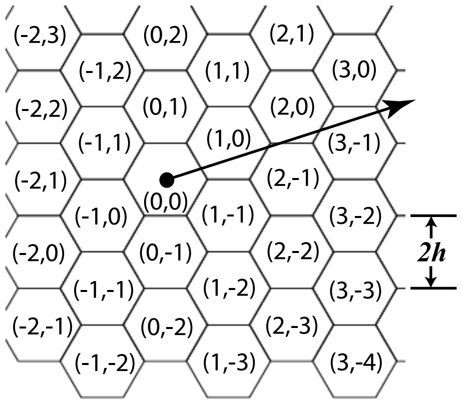
Incrementing `y` will still decrement `d` by 1, however now incrementing `x` will use the forward-difference update, $\Delta y(\mathtt{x})$. The pseudocode is as follows:

```
// INIT:
a2  = 2*a;
x   = ceil(x0)
f   = a*x*x + b*x + c;   /// y(x)
Df  = 2*a*x + (a+b);     /// Delta y(x)
y   = round( f )

// Evaluate d at x+1, first computing y(x+1) and Delta y(x+1):
f  += Df;
Df += a2;
d   = f - y;  /// = y(x+1) - y

while (x < floor(x1))
     if (d > 0.5)
          y += 1
          d -= 1
     x  += 1
     d  += Df;
     Df += a2;
     output(x, y)
```

Problem 4: Tracing rays through hexagonal subdivisions (12 pts)



You have seen how to trace a ray through a square grid in 2D, and even a voxel grid in 3D. In this question you will consider 2D hexagonal grids. Analogous to rectangular grids, assume that the hexagonal cells have an (i, j) indexing as shown in the figure. Assume that each hexagon's parallel edges are $2h$ apart (see figure).

Propose an efficient pseudocode implementation to trace the ray through an infinite hexagonal subdivision, making calls to `output(i,j)` indices of hexagons traversed. For simplicity, assume that the ray $\boldsymbol{r}(t) = \boldsymbol{e} + t\boldsymbol{v}$, $t \geq 0$, starts at the *center* of cell $(i, j) = (0, 0)$ as shown in the figure. Ignore boundaries.

- **Answer:** There are three families of parallel lines associated with the normal directions

$$\begin{aligned} \boldsymbol{n}_0 &= (\sin(30^o), \cos(30^o))^T && (18)\\ \boldsymbol{n}_1 &= (\sin(90^o), \cos(90^o))^T = (1, 0) && (19)\\ \boldsymbol{n}_2 &= (\sin(150^o), \cos(150^o))^T. && (20)\end{aligned}$$

The signed time Δt to travel a distance h in each of these directions is given by

$$\begin{aligned} \tau_0 &= h/\boldsymbol{v}^T\boldsymbol{n}_0 && (21)\\ \tau_1 &= h/\boldsymbol{v}^T\boldsymbol{n}_1 && (22)\\ \tau_2 &= h/\boldsymbol{v}^T\boldsymbol{n}_2. && (23)\end{aligned}$$

Observe that crossing edges along each of the directions results in an (i, j) cell index update, $(i, j)+\delta$, given by

$$\begin{aligned} \delta_0 &= \text{sgn}(\tau_0)(1, 0) && (24)\\ \delta_1 &= \text{sgn}(\tau_1)(0, 1) && (25)\\ \delta_2 &= \text{sgn}(\tau_2)(-1, 1). && (26)\end{aligned}$$

From our starting point at the center of cell $(0, 0)$, we can initialize t-distances to the nearest edges as

$$\begin{aligned} \Delta t_0 &= |\tau_0| && (27)\\ \Delta t_1 &= |\tau_1| && (28)\\ \Delta t_2 &= |\tau_2|. && (29)\end{aligned}$$

We will first cross an edge in the direction i given by

$$i = \arg\min_k \Delta t_k,$$

at which time we update Δt_i to be $2|\tau_i|$, but the two other directions must have their times updated too. By observation, the update rules (for $j \neq i$) are

$$\Delta t_j \leftarrow (\Delta t_j + |\tau_j|) \pmod{2|\tau_j|}.$$

Pseudocode to implement this HEX-WALK is as follows (multiplies by 2 can be avoided using additional precomputed variables):

```
// INITIALIZE
n_0 = ...;
n_1 = ...;
n_2 = ...;
tau_0 = ...;
tau_1 = ...;
tau_2 = ...;
delta_0 = ...;
delta_1 = ...;
delta_2 = ...;
Dt_0 = ...;
Dt_1 = ...;
Dt_2 = ...;

(I,J) = (0,0);
output(I,J);

while (continue walk)
{
   // Find crossed edge:
   i = argmin_k  Dt_k;

   // Update cell indices:
   (I,J) += delta_i;
   output(I,J);

   // Update Dt values:
   for(j=0..2) {
       if(j==i)
         Dt_i = 2|tau_i|;
       else
         Dt_j = (Dt_j + |tau_j|) (mod 2|tau_j|);
     }
   }
}
```

Problem 5: Phong Tesselation (20 pts)

Recall that *Phong Shading* interpolates vertex normals across a triangle for smooth shading on low-resolution meshes, i.e., the unnormalized surface normal at barycentric coordinate (u, v, w) (where $w = 1 - u - v$) is approximated by barycentrically interpolated vertex normals,

$$\boldsymbol{n}'(u, v) = u\boldsymbol{n}_i + v\boldsymbol{n}_j + w\boldsymbol{n}_k,$$

where the unit vertex normals are $\boldsymbol{n}_i$, $\boldsymbol{n}_j$ and $\boldsymbol{n}_k$. Of course, since each triangle is still planar,

$$\boldsymbol{p}(u, v) = u\boldsymbol{p}_i + v\boldsymbol{p}_j + w\boldsymbol{p}_k, \tag{30}$$

the piecewise planar shape is still apparent at silhouettes.

Recently, Boubekeur and Alexa [SIGGRAPH Asia 2008] introduced *Phong Tesselation* as a simple way to use vertex normals to deform a triangle mesh to have smoother silhouettes (see Figures 1 and 2). In the following, you will derive their formula for a curved triangle patch, $\boldsymbol{p}^*(u, v)$, and analyze surface continuity.

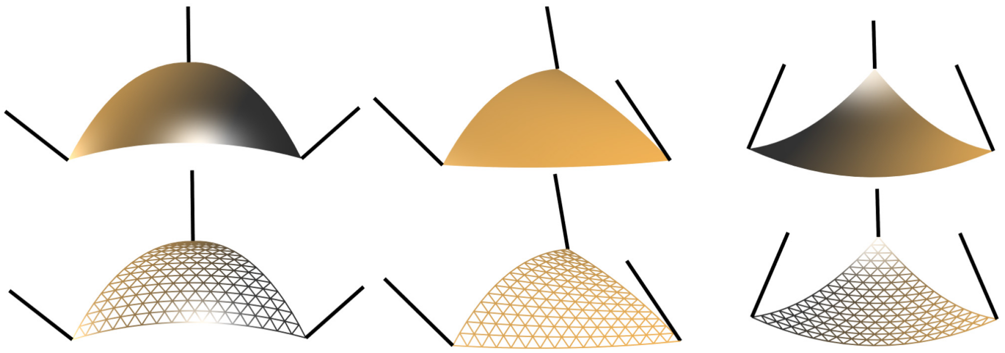


Figure 1: Phong Tesselation Examples: A triangle deformed with different vertex normals.

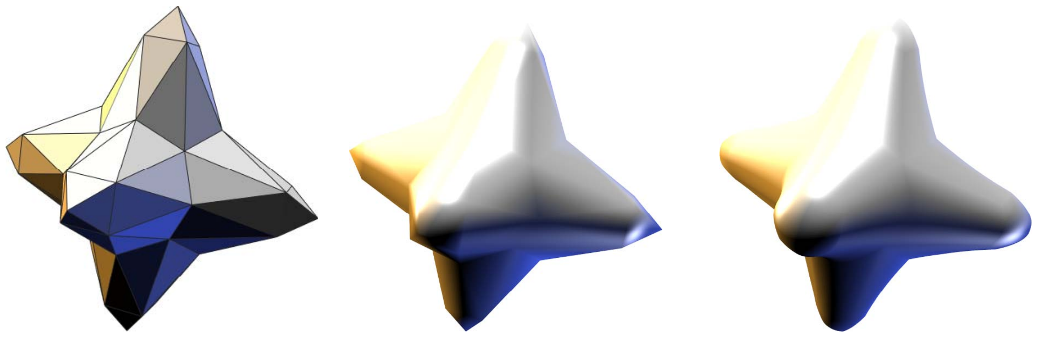


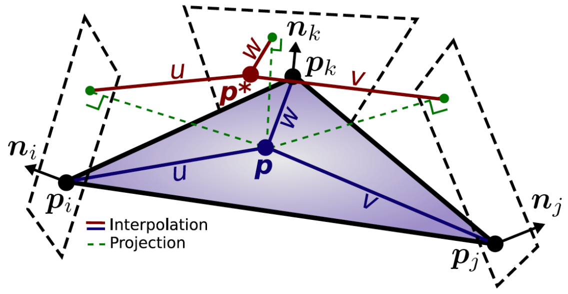
Figure 2: Phong Tesselation

Answer the following four questions:

(a) Consider the plane passing though vertex i's position, $\boldsymbol{p}_i$, and sharing the same normal, $\boldsymbol{n}_i$. Give an expression for the orthogonal projection of a point $\boldsymbol{p}$ onto vertex i's plane, hereafter denoted by $\boldsymbol{\pi}_i(\boldsymbol{p})$.

- **Answer:** The component of $\boldsymbol{v} = (\boldsymbol{p} - \boldsymbol{p}_i)$ along the normal is $\boldsymbol{v}^T \boldsymbol{n}_i$. Therefore the component along the surface is $\boldsymbol{v} - \boldsymbol{n}_i \boldsymbol{n}_i^T \boldsymbol{v}$, and so

$$\begin{aligned} \boldsymbol{\pi}_i(\boldsymbol{p}) &= \boldsymbol{p}_i + \boldsymbol{v} - \boldsymbol{n}_i \boldsymbol{n}_i^T \boldsymbol{v} && (31) \\ &= \boldsymbol{p} - \boldsymbol{n}_i \boldsymbol{n}_i^T (\boldsymbol{p} - \boldsymbol{p}_i) && (32) \end{aligned}$$



(b) The deformed position $\boldsymbol{p}^*(u, v)$ is simply the barycentrically interpolated projections of the undeformed point $\boldsymbol{p}(u, v)$ onto the three vertex planes, i.e., the barycentric interpolation of $\boldsymbol{\pi}_i(\boldsymbol{p}(u, v))$, $\boldsymbol{\pi}_j(\boldsymbol{p}(u, v))$, and $\boldsymbol{\pi}_k(\boldsymbol{p}(u, v))$. Derive a polynomial expression for $\boldsymbol{p}^*(u, v)$ in terms of u, v and w—you can also write it only in terms of u and v but it is messier. (Hint: Express your answer in terms of projected-vertex positions, such as $\boldsymbol{\pi}_i(\boldsymbol{p}_j)$.)

- **Answer:** The provided definition says that

$$\begin{aligned} \boldsymbol{p}^*(u, v) &= u\, \boldsymbol{\pi}_i(\boldsymbol{p}(u, v)) + v\, \boldsymbol{\pi}_j(\boldsymbol{p}(u, v)) + w\, \boldsymbol{\pi}_k(\boldsymbol{p}(u, v)) && (33) \\ &= u\, \boldsymbol{\pi}_i\left(u\boldsymbol{p}_i + v\boldsymbol{p}_j + w\boldsymbol{p}_k\right) + && (34) \\ &\quad\; v\, \boldsymbol{\pi}_j\left(u\boldsymbol{p}_i + v\boldsymbol{p}_j + w\boldsymbol{p}_k\right) + && (35) \\ &\quad\; w\, \boldsymbol{\pi}_k\left(u\boldsymbol{p}_i + v\boldsymbol{p}_j + w\boldsymbol{p}_k\right) && (36) \\ &= u^2\, \boldsymbol{\pi}_i(\boldsymbol{p}_i) + uv\, \boldsymbol{\pi}_i(\boldsymbol{p}_j) + uw\, \boldsymbol{\pi}_i(\boldsymbol{p}_k) + && (37) \\ &\quad\; uv\, \boldsymbol{\pi}_j(\boldsymbol{p}_i) + v^2\, \boldsymbol{\pi}_j(\boldsymbol{p}_j) + vw\, \boldsymbol{\pi}_j(\boldsymbol{p}_k) + && (38) \\ &\quad\; uw\, \boldsymbol{\pi}_k(\boldsymbol{p}_i) + vw\, \boldsymbol{\pi}_k(\boldsymbol{p}_j) + w^2\, \boldsymbol{\pi}_k(\boldsymbol{p}_k) && (39) \\ &= u^2\boldsymbol{p}_i + v^2\boldsymbol{p}_j + w^2\boldsymbol{p}_k + && (40) \\ &\quad\; uv(\boldsymbol{\pi}_i(\boldsymbol{p}_j) + \boldsymbol{\pi}_j(\boldsymbol{p}_i)) + && (41) \\ &\quad\; vw(\boldsymbol{\pi}_j(\boldsymbol{p}_k) + \boldsymbol{\pi}_k(\boldsymbol{p}_j)) + && (42) \\ &\quad\; uw(\boldsymbol{\pi}_k(\boldsymbol{p}_i) + \boldsymbol{\pi}_i(\boldsymbol{p}_k)). && (43) \end{aligned}$$

(c) What degree is this triangular bivariate polynomial patch, $\boldsymbol{p}^*(u, v)$?

- **Answer:** From our derived formulae, it is clear that $\boldsymbol{p}^*(u, v)$ is a quadratic (or degree-2) polynomial patch.

  (Note: It is incorrect to state that "it is a quadratic patch since it has 3 control points" supposedly in analogy with the 1D curve setting. Note that a planar triangle patch (30) also has 3 control points, but is degree one.)

(d) Given a triangle mesh that is converted to these polynomial patches, consider the parametric continuity of the resulting spline surface:

(i) Show that the surface is G^0 continuous.

- **Answer:** Clearly the quadratic patch is G^0 continuous within the triangle domain, so it remains to show that it is continuous across patch boundaries. First, observe that the patch interpolates vertices. Second, the fact that there are no cracks across edges follows from the fact that for (u, v) coordinates on the edge the function $\boldsymbol{p}^*(u, v)$ only depends on the position and normal values at the end-points of the edge. For example, on the edge with $w = 0$ we have

$$\boldsymbol{p}^*(u, v) = u^2\boldsymbol{p}_i + v^2\boldsymbol{p}_j + uv(\boldsymbol{\pi}_i(\boldsymbol{p}_j) + \boldsymbol{\pi}_j(\boldsymbol{p}_i)) \tag{44}$$

where $\boldsymbol{\pi}_i$ only depends on the position and normal of vertex i. Therefore patches on either side of the edge will share the same interface curve values, and the surface is G^0 continuous.

(ii) Show that the surface is not G^1 continuous.

- **Answer:** Clearly the quadratic patch is G^1 continuous within the triangle domain, so it suffices to consider the patch boundaries. We already know that edge-adjacent patches will share the same interface curve, and therefore they will have the same directional derivative along the patch edge. Therefore, we can try to show that the normals along the edges may differ by considering derivatives along directions not parallel to the edge—since the normal is a cross product of such partial derivatives. If this derivative depends on the opposite vertex data, then the normals will not (in general) be continuous across the edge.

  Another way to proceed is to consider the patch normal at a vertex, to see that it depends on more than just the position/normal at that vertex, and therefore can not be the same as normals of the adjacent patches. This would require taking two derivatives and a cross product.

  For simplicity, let's consider the edge case, and use the $w = 0$ edge. It suffices to show that the $\frac{\partial \boldsymbol{p}^*}{\partial w}$ value on the $w = 0$ edge depends on the values at the opposite vertex, k. Differentiating (43) we find that (for brevity, let $\partial_w \equiv \frac{\partial}{\partial w}$)

$$\begin{aligned}
\frac{\partial}{\partial w}p^*(u,v) &= \frac{\partial u^2}{\partial w}\boldsymbol{p}_i + \frac{\partial v^2}{\partial w}\boldsymbol{p}_j + \frac{\partial w^2}{\partial w}\boldsymbol{p}_k + (v\frac{\partial u}{\partial w} + u\frac{\partial v}{\partial w})(\boldsymbol{\pi}_i(\boldsymbol{p}_j) + \boldsymbol{\pi}_j(\boldsymbol{p}_i)) + \\
&\quad (w\frac{\partial v}{\partial w} + v\frac{\partial w}{\partial w})(\boldsymbol{\pi}_j(\boldsymbol{p}_k) + \boldsymbol{\pi}_k(\boldsymbol{p}_j)) + (w\frac{\partial u}{\partial w} + u\frac{\partial w}{\partial w})(\boldsymbol{\pi}_k(\boldsymbol{p}_i) + \boldsymbol{\pi}_i(\boldsymbol{p}_k)) \\
&= -2u\boldsymbol{p}_i - 2v\boldsymbol{p}_j + 2w\boldsymbol{p}_k + (-v-u)(\boldsymbol{\pi}_i(\boldsymbol{p}_j) + \boldsymbol{\pi}_j(\boldsymbol{p}_i)) + \\
&\quad (-w+v)(\boldsymbol{\pi}_j(\boldsymbol{p}_k) + \boldsymbol{\pi}_k(\boldsymbol{p}_j)) + (-w+u)(\boldsymbol{\pi}_k(\boldsymbol{p}_i) + \boldsymbol{\pi}_i(\boldsymbol{p}_k))
\end{aligned}$$

  Which on the edge $w = 0$ becomes

$$\left.\frac{\partial \boldsymbol{p}^*}{\partial w}\right|_{w=0} = -2u\boldsymbol{p}_i - 2v\boldsymbol{p}_j - (u+v)(\boldsymbol{\pi}_i(\boldsymbol{p}_j) + \boldsymbol{\pi}_j(\boldsymbol{p}_i)) + \tag{45}$$

$$v(\boldsymbol{\pi}_j(\boldsymbol{p}_k) + \boldsymbol{\pi}_k(\boldsymbol{p}_j)) + u(\boldsymbol{\pi}_k(\boldsymbol{p}_i) + \boldsymbol{\pi}_i(\boldsymbol{p}_k)), \tag{46}$$

  and since it depends on the position and normal of vertex k, the surface can not be G^1 continuous.