CS 4620 Final Exam (Prof. James)

Thursday 9 December 2010—2.5 hours

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

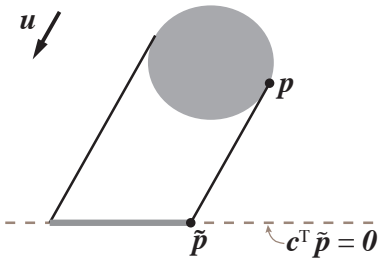
Problem 1: Shading (10 pts)

Given a triangle mesh, briefly explain the key differences in how *triangle normals* are used to evaluate (i) flat shading, (ii) Gouraud shading, and (iii) Phong shading.

- **Answer:** Given a triangle mesh we can compute each triangle normal using a cross product. The question asks you how these normals are used to compute shading, i.e., the color of reflected light. Note that the shading methods all use some reflectance function, such as the Blinn-Phong BRDF, that uses position/normal at a point. These shading functions all use normals differently: (i) Flat shading is evaluated using the normal of the triangle associated with the sample point. (ii) Gouraud shading uses normals associated with the triangle's vertices (which may be computed by averaging triangle normals around the vertex), then evaluates the color of each vertex using the reflectance model (e.g., Blinn-Phong BRDF) and linearly interpolates those colors across the triangle. In contrast, (iii) Phong shading interpolates vertex normals across the triangle (e.g., via linear interpolation followed by normalization), then evaluates the shaded color by applying the reflectance model at a point on a triangle with the interpolated normal. Consequently, flat shading produces constant colors over triangle facets, Gouraud shading shows piecewise linear color variations but is incapable of producing specular highlights within a triangle—something Phong shading can do quite well.

Problem 2: Planar Shadows (15 pts)

Consider a directional light source (unit direction, $\boldsymbol{u}$), and the planar shadow created by literally projecting each mesh vertex position, $\boldsymbol{p}$, to its shadow point $\tilde{\boldsymbol{p}}$ on the 3D plane specified (in homogeneous coordinates) by $\boldsymbol{c}^T\tilde{\boldsymbol{p}} = 0$ where $\boldsymbol{c} \in \mathbb{R}^4$. Derive a formula for the 4x4 projection matrix, $\mathbf{A}$, that maps a homogeneous object point, $\boldsymbol{p} = (x, y, z, 1)^T$, to its shadow point, $\tilde{\boldsymbol{p}} = \mathbf{A}\boldsymbol{p}$. *(Hint: Consider the ray $\boldsymbol{p} + t\boldsymbol{u}$.)*



- **Answer:** Following the hint, let the point on the ray that hits the plane be given by

$$\tilde{\boldsymbol{p}} = \boldsymbol{p} + t\boldsymbol{u},$$

then, given that the projected point must lie on the plane, t must satisfy

$$0 = \boldsymbol{c}^T\tilde{\boldsymbol{p}} = \boldsymbol{c}^T(\boldsymbol{p} + t\boldsymbol{u}) = \boldsymbol{c}^T\boldsymbol{p} + t\boldsymbol{c}^T\boldsymbol{u}t$$

so that

$$t = -\frac{\boldsymbol{c}^T\boldsymbol{p}}{\boldsymbol{c}^T\boldsymbol{u}}$$

(unless $\boldsymbol{c}^T\boldsymbol{u} \neq 0$ in which case the ray is parallel to the plane and never intersects). Note that one possibility is that this t is negative, in which case there is no shadow, be we can ignore this case. Substituting this t back into the ray equation we get

$$\begin{aligned}
\tilde{\boldsymbol{p}} &= \boldsymbol{p} + \boldsymbol{u}t && (1)\\
&= \boldsymbol{p} - \boldsymbol{u}\frac{\boldsymbol{c}^T\boldsymbol{p}}{\boldsymbol{c}^T\boldsymbol{u}} && (2)\\
&= \boldsymbol{p} - \frac{1}{\boldsymbol{c}^T\boldsymbol{u}}\boldsymbol{u}\boldsymbol{c}^T\boldsymbol{p} && (3)\\
&= \left(\boldsymbol{I} - \frac{1}{\boldsymbol{c}^T\boldsymbol{u}}\boldsymbol{u}\boldsymbol{c}^T\right)\boldsymbol{p} && (4)\\
&= \mathbf{A}\boldsymbol{p} && (5)\\
& && (6)
\end{aligned}$$

Problem 3: Splines (30 pts)

Consider the cubic spline segment, $\mathbf{p}(t) = \mathbf{a}\,t^3 + \mathbf{b}\,t^2 + \mathbf{c}\,t + \mathbf{d}$, on the unit interval $t \in [0, 1]$. Recall that Hermite splines use $\mathbf{p}$ and $\mathbf{p}'$ controls at each end point, i.e., $\mathbf{p}_0 = \mathbf{p}(0)$, $\mathbf{p}'_0 = \mathbf{p}'(0)$, $\mathbf{p}_1 = \mathbf{p}(1)$, $\mathbf{p}'_1 = \mathbf{p}'(1)$. In this question you will derive a spline which instead uses the following controls: at $t = 0$, use position ($\mathbf{p}_0$), velocity ($\mathbf{p}'_0$) and acceleration ($\mathbf{p}''_0$); at $t = 1$, use position ($\mathbf{p}_1$).

(a) What is this spline's 4x4 geometry matrix $\mathbf{G}$ such that $\mathbf{p}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \mathbf{G} \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}'_0 \\ \mathbf{p}''_0 \\ \mathbf{p}_1 \end{bmatrix}$?

- **Answer:** Taking derivatives of $\mathbf{p}(t)$ we have

$$\begin{aligned}
\mathbf{p}(t) &= \mathbf{a}\,t^3 + \mathbf{b}\,t^2 + \mathbf{c}\,t + \mathbf{d} && (7)\\
\mathbf{p}'(t) &= 3\mathbf{a}\,t^2 + 2\mathbf{b}\,t + \mathbf{c} && (8)\\
\mathbf{p}''(t) &= 6\mathbf{a}\,t + 2\mathbf{b} && (9)
\end{aligned}$$

so that

$$\begin{aligned}
\mathbf{p}(1) &= \mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{d} && (10)\\
\mathbf{p}(0) &= \mathbf{d} && (11)\\
\mathbf{p}'(0) &= \mathbf{c} && (12)\\
\mathbf{p}''(0) &= 2\mathbf{b}. && (13)
\end{aligned}$$

It follows that

$$\begin{aligned}
\mathbf{d} &= \mathbf{p}_0 && (14)\\
\mathbf{c} &= \mathbf{p}_0' && (15)\\
\mathbf{b} &= 1/2\mathbf{p}_0'' && (16)\\
\mathbf{a} &= \mathbf{p}_1 - (\mathbf{b}+\mathbf{c}+\mathbf{d}) && (17)\\
&= \mathbf{p}_1 - 1/2\mathbf{p}_0'' - \mathbf{p}_0' - \mathbf{p}_0 && (18)\\
&= -\mathbf{p}_0 - \mathbf{p}_0' - 1/2\mathbf{p}_0'' + \mathbf{p}_1 && (19)
\end{aligned}$$

Therefore the curve and geometry matrix are given by

$$\begin{aligned}
\mathbf{p}(t) &= \mathbf{a}\,t^3 + \mathbf{b}\,t^2 + \mathbf{c}\,t + \mathbf{d} && (20)\\
&= \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -0.5 & 1 \\ 0 & 0 & +0.5 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_0' \\ \mathbf{p}_0'' \\ \mathbf{p}_1 \end{bmatrix}. && (21)
\end{aligned}$$

(b) Given a cubic spline contructed from these segments, is the resulting curve (i) C^0 continuous? (ii) C^1 continuous? (iii) G^0 continuous? (iv) G^1 continuous? Briefly explain your reasoning in each case.

- **Answer:** The question asks you to consider continuity at the endpoints of a spline segment. You can assume that we have a spline that has uniformly spaced knots, with each knot specified by triples of values ($\mathbf{p}_0$, $\mathbf{p}_0'$, $\mathbf{p}_0''$) at $t = 0$, then ($\mathbf{p}_1$, $\mathbf{p}_1'$, $\mathbf{p}_1''$) at $t = 1$, then ($\mathbf{p}_2$, $\mathbf{p}_2'$, $\mathbf{p}_2''$) at $t = 2$, etc. Assume that we have two spline segments, $\mathbf{p}(t)$ for $t \in [0, 1]$ and a separate $\mathbf{q}(t)$ for $t \in [1, 2]$ (which could be mapped to our [0,1] derivation by a translation, $\tau = t - 1$), and that we will consider what happens at $t = 1$. Since the cubic spline constructed by joining these segments is geometrically connected at $t = 1$ (it must share the same endpoint $\mathbf{p}_1$) it must have G^0 geometric continuity. Furthermore, since the coordinate functions are cubic polynomials (which are continuous to all orders), then the curve must also have the same limits at $t = 1$, i.e., $\lim_{t\to 1_-} \mathbf{p}(t) = \lim_{t\to 1_+} \mathbf{q}(t) = \mathbf{p}_1$, and therefore it must have C^0 parametric continuity. However, unlike the Hermite cubic spline, there is no reason that the curve's tangents should be equal at the endpoints; the values of $\mathbf{p}'(1)$ and $\mathbf{q}'(1)$ depend on different control parameters, ($\mathbf{p}_0$, $\mathbf{p}_0'$, $\mathbf{p}_0''$, $\mathbf{p}_1$) and ($\mathbf{p}_1$, $\mathbf{p}_1'$, $\mathbf{p}_1''$, $\mathbf{p}_2$), respectively, and therefore we won't have C^1 parametric continuity. Furthermore, since the endpoint tangents needn't even point in the same direction, we can't have G^1 geometric continuity either.

Problem 4: Rotations (15 pts)

In this question, you will use quaternion multiplication to show that two rotations don't necessarily commute, i.e., the order in which they are performed matters. In particular, let the quaternions $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ represent rotations about the x and y axes, by angles θ_x and θ_y, respectively. Use quaternion multiplication to show that $\boldsymbol{q}_x\boldsymbol{q}_y \neq \boldsymbol{q}_y\boldsymbol{q}_x$.

- **Answer:** Recall the unit quaternion corresponding to a rotation about axis $\boldsymbol{u}$ by angle θ is $\boldsymbol{q} =$

$\left\langle \cos\frac{\theta}{2}; \sin\frac{\theta}{2}\boldsymbol{u}\right\rangle$. Therefore the two unit quaternions are

$$\boldsymbol{q}_x = \left\langle \cos\frac{\theta_x}{2}; \sin\frac{\theta_x}{2}, 0, 0\right\rangle \tag{22}$$

$$\boldsymbol{q}_y = \left\langle \cos\frac{\theta_y}{2}; 0, \sin\frac{\theta_y}{2}, 0\right\rangle \tag{23}$$

Recall the formula for quaternion multiplication,

$$< d; \boldsymbol{u} >< d'; \boldsymbol{u}' >=< dd' - \boldsymbol{u}\cdot\boldsymbol{u}';\ d\boldsymbol{u}' + d'\boldsymbol{u} + \boldsymbol{u}\times\boldsymbol{u}' > . \tag{24}$$

Applying this formula (and using $\hat{\boldsymbol{x}}\times\hat{\boldsymbol{y}} = \hat{\boldsymbol{z}}$ and $\hat{\boldsymbol{y}}\times\hat{\boldsymbol{x}} = -\hat{\boldsymbol{z}}$), we have

$$\boldsymbol{q}_x\boldsymbol{q}_y = \left\langle \cos\frac{\theta_x}{2}\cos\frac{\theta_y}{2};\ \cos\frac{\theta_y}{2}\sin\frac{\theta_x}{2},\ \cos\frac{\theta_x}{2}\sin\frac{\theta_y}{2},\ \sin\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\right\rangle \tag{25}$$

$$\boldsymbol{q}_y\boldsymbol{q}_x = \left\langle \cos\frac{\theta_x}{2}\cos\frac{\theta_y}{2};\ \cos\frac{\theta_y}{2}\sin\frac{\theta_x}{2},\ \cos\frac{\theta_x}{2}\sin\frac{\theta_y}{2},\ -\sin\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\right\rangle \tag{26}$$

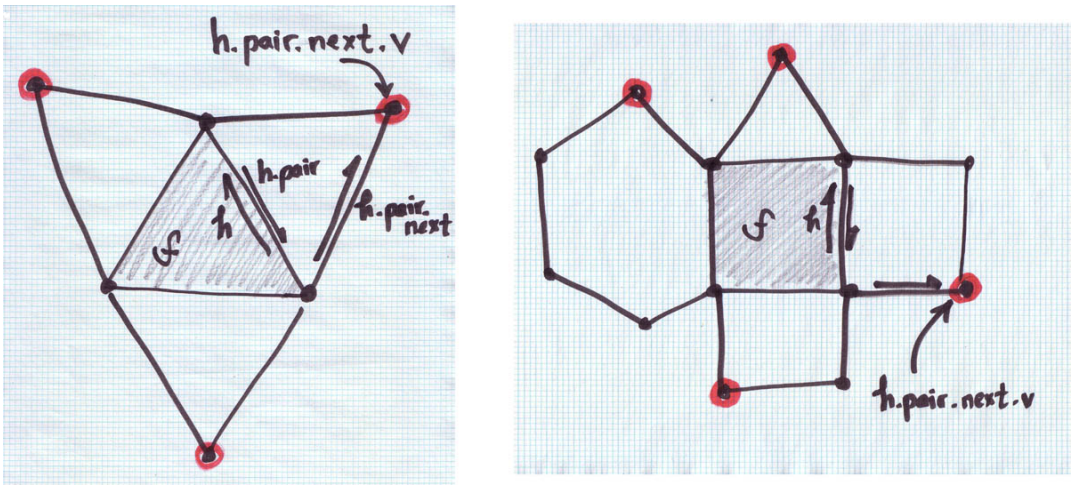
The resulting two quaternions are different due to the sign change on the last component, and thus the original x/y rotations do not commute.

Problem 5: Meshes (10 pts)

Recall the half-edge data structure. What does the following pseudocode accumulate in the set `S`? *(Hint: Drawing a picture may help.)*

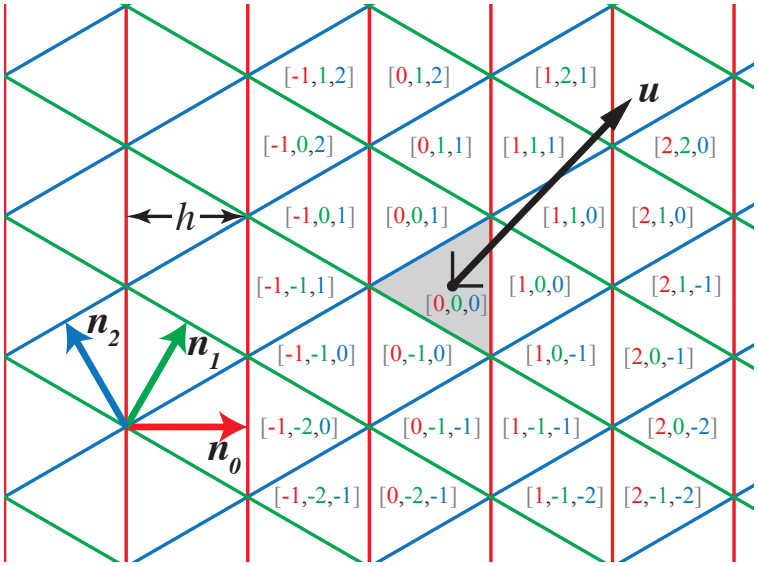
```
ProcessFace(f, S) {
  h = f.h;
  do {
    h = h.next;
    S.add(h.pair.next.v);
  } while (h != f.h);
}
```

- **Answer:** Observe that the loop walks around the edges of the face until it reaches the starting edge `f.h`. Note that the code works for general polygons, and the faces need not be triangles. Given a face edge `h` at any step, `h.pair` extracts it's opposite edge pair, so that `h.pair.next` is the next edge on the edge-opposite face as shown. Finally, the edge's vertex `h.pair.next.v` is added to the set. In the case of triangle meshes the accumulated `S` vertices form a larger triangle (shown highlighted in red). Since there is nothing about the problem that requires triangle meshes, a more general case is also drawn.



Problem 6: Ray Tracing (20 pts)

Uniform subdivisions are commonly used for ray tracing, wherein a ray is intersected with cell boundaries to "walk" through the subdivision's cells. In this problem, you will generate pseudocode to walk through a uniform triangular subdivision (see figure). It can be viewed as the union of three 1D uniform subdivisions (color-coded for convenience), each associated with a normalized direction vector ($\boldsymbol{n}_0$, $\boldsymbol{n}_1$ or $\boldsymbol{n}_2$). Each 1D cell is of width, h. Denote each triangular cell by a 3-index label $[i_0, i_1, i_2]$ (RGB color coded) based on the cell's location in each of the 1D subdivisions. Assume that our ray $\boldsymbol{r}(t)$ originates from the origin, $\mathbf{0}$, so that $\boldsymbol{r}(t) = \mathbf{0} + t\boldsymbol{u}$, $t > 0$. Assume that the origin lies at the *centroid* of the triangular cell indexed by [0, 0, 0].



Write pseudocode to efficiently generate the sequence of traversed cells when given the ray direction, $\boldsymbol{u}$ and these particular cell direction vectors (assume normalized). For example, given the $\boldsymbol{u}$ vector shown in the figure, your code would output the infinite sequence:

$$[0, 0, 0], [1, 0, 0], [1, 1, 0], [1, 1, 1], [1, 2, 1], \ldots$$

Output a traversed cell with the function "`output` $[i_0, i_1, i_2]$."

(Hint: Consider each direction's next t intersection value.)

- **Answer:** This is basically another ray-plane intersection problem. The rate at which the ray $\boldsymbol{r}(t)$ travels along the unit cell directions $\boldsymbol{n}_i$ is given by the component of $\boldsymbol{r}' = \boldsymbol{u}$ along each $\boldsymbol{n}_i$ direction. In otherwords, the rate of position change in direction i is given by $\boldsymbol{u}^T\boldsymbol{n}_i$, and since each direction must travel h far between line crossings, the change in t between crossings for direction i is given by

$$\Delta t_i = \frac{h}{\boldsymbol{u}^T\boldsymbol{n}_i} \qquad (\text{assuming } \boldsymbol{u}^T\boldsymbol{n}_i \neq 0), \tag{27}$$

  which may be negative. The remaining question is how far each direction must travel before reaching the first crossing, and this is related to the position of the origin at the centroid of the [0,0,0] triangle. A simple geometric fact (that you can easily show) is that for an equilateral triangle placed flat against the ground, of height h, the centroid occurs at height $h/3$. Therefore for directions $\boldsymbol{n}_0$ and $\boldsymbol{n}_2$, if their $\Delta t_i > 0$ they must travel a distance $h/3$ to their first crossing, but for $\Delta t_i < 0$ its a distance of $2h/3$. Unfortunately $\boldsymbol{n}_1$ is somewhat backwards: for positive Δt_1 it must travel $2h/3$, but for negative Δt_1 it must travel $h/3$. These spatial distances can be converted to t distances as with Δt_i in (27).

  The algorithm then proceeds as follows. For each direction i, we compute Δt_i, and initialize a variable t_i^{next} for the next crossing time. The cell indices are set to [0,0,0]. At each step, we find the k associated with the smallest t_k^{next}, increment/decrement i_k by 1 (depending on the sign of Δt_k), increase t_k^{next} by $|\Delta t_k|$, and `output` $[i_0, i_1, i_2]$. Pseudocode to do this is given below.

```
// INITIALIZE:
[i0, i1, i2] = [0, 0, 0]
Δt0 = h/u^T n0
Δt1 = h/u^T n1
Δt2 = h/u^T n2
t0^next = ( Δt0 > 0 ? 1 : 2 ) |Δt0|/3
t1^next = ( Δt1 > 0 ? 2 : 1 ) |Δt1|/3
t2^next = ( Δt2 > 0 ? 1 : 2 ) |Δt2|/3

// WALK THROUGH CELLS:
forever {
    k    = arg min_i t_i^next
    t_k^next += |Δt_k|
    i_k  += sign(Δt_k)
    output [i0, i1, i2]
}
```

  For speed we could precompute/cache the $|\Delta t_k|$ and $\texttt{sign}(\Delta t_k)$ constants used in the forever loop.