Name Solution

GPU Programming
EE 4702-1
Practice Midterm Examination

This practice exam has been made a little longer than the actual midterm is expected to be. Also, some questions are expected to be answered using references, whereas on the actual midterm all needed information will be provided.

Alias Practice Exam Total _____ (100 pts)

Good Luck!

Problem 1: A scene contains a light at position L and a vertex at position V. The scene contains a plane which includes point P_0 and has normal n. The scene also contains a circle of radius r with the center at C, and with normal m.

(a) Let P be the point on the plane where the vertex casts a shadow. Find an expression for P.

This is a simple line/plane intercept problem.

To find the shadow consider a line passing through L and V. Let P be the point where the line passes through the plane (assuming it does). P will be the shadow location if V is between L and P.

Let $\vec{LV}$ be the vector from L to V. The parametric equation of the line through L and V is then $P = V + t\vec{LV}$, where t is the parameter. If P is on the plane then $\vec{PP_0}$ is orthogonal to the plane normal, n and so $\vec{PP_0} \cdot n = 0$. Substituting for P yields

$$\begin{aligned}(V + t\vec{LV})P_0 \cdot n &= 0\\ (P_0 - \vec{V} - t\vec{LV}) \cdot n &= 0\\ \vec{VP_0} \cdot n - t\vec{LV} \cdot n &= 0\\ t &= \frac{\vec{VP_0} \cdot n}{\vec{LV} \cdot n}\end{aligned}$$

Substituting t in the line equation gives the possible shadow location:

$$P = V + \frac{\vec{VP_0} \cdot n}{\vec{LV} \cdot n}\vec{LV}.$$

(b) For what value of r will the shadow cast by V be exactly on the circle perimeter. *Hint: Simple once the previous part solved.*

This is another line plane intercept problem. Find the point at which the line defined by the vertex and light intercepts the plane containing the circle, call that point A. Then r is just the distance between A and C.

Problem 2: A scene contains a moving sphere of radius r with center at $S(t) = S_0 + tV$, where t is the time, S_0 is the position at $t = 0$, and D is the velocity.

The scene also contains a cube with which the sphere may collide. The cube center is at C, a side of the cube has length a, and the face normals are $\pm\hat{n}_1$, $\pm\hat{n}_2$, and $\pm\hat{n}_3$. *Hint: From that information we can determine that a point on one of the faces is* $C + \frac{a}{2}\hat{n}_1$*, a point on one of the edges is* $C + \frac{a}{2}\hat{n}_1 + \frac{a}{2}\hat{n}_2$ *and one of the corners is* $C + \frac{a}{2}\hat{n}_1 + \frac{a}{2}\hat{n}_2 + \frac{a}{2}\hat{n}_3$.

(a) Describe the three ways the sphere can collide with the cube. Each will require solving a different kind of geometric problem.

The ball can collide with a face (sphere/plane collision), an edge (sphere/line collision), or a corner (sphere/point collision). The brute-force way to determine collision time is to find the collision time with each of the six faces, 12 edges, and four corners, and then use the smallest such time.

(b) Write expressions for collision time with each type of element (assuming they do collide). *Hint: One of these is a solution to a prior problem. Note: This problem is harder than I would ask on a real exam. On a real exam I might ask for expressions indicating how close the sphere is to each type of element at a particular time (with a distance of zero or perhaps less than* r *indicating contact).*

We need expressions for when the sphere will hit a plane (cube face), line (cube edge), and point (cube corner). The path of the center of the sphere forms the line $S(t) = S_0 + tD$.

Let Q be a point on a corner of the cube, $Q = C + \frac{a}{2}\hat{n}_1 + \frac{a}{2}\hat{n}_2 + \frac{a}{2}\hat{n}_3$. The plane collision will be shown for the face containing Q and normal n_1. For the line collision it will be the edge touching Q and in direction n_1, and for the corner collision the point Q.

At time t the sphere center will be at $S(t)$, at the time of impact the sphere will be distance r from the plane, so we will solve the plane/line intercept problem for a plane moved distance r from the cube, that moves Q to $Q_p = Q + r\hat{n}_1$; the plane normal is still n_1.

Using the solution to the line/plane intercept from the previous problem we find the collision time

$$t = \frac{S_0\vec{Q}_p \cdot n_1}{D \cdot n_1}$$

.

For the corner collision we need to find the time the sphere is at distance r from a corner. This is equivalent to asking when the center of the sphere intercepts a sphere of radius r centered on Q.

When $S(t)$ is a distance r from Q then $\|Q\vec{S}(t)\| = r$, or $Q\vec{S}(t) \cdot Q\vec{S}(t) = r^2$. Substituting for $S(t)$:

$$\begin{aligned} Q\vec{S}(t) \cdot Q\vec{S}(t) &= r^2 \\ (S_0 + tD - Q) \cdot (S_0 + tD - Q) &= r^2 \\ (Q\vec{S}_0 + tD) \cdot (Q\vec{S}_0 + tD) &= r^2 \\ Q\vec{S}_0 \cdot Q\vec{S}_0 + 2Q\vec{S}_0 \cdot tD + t^2 D \cdot D &= r^2 \end{aligned}$$

Solving for t:

$$t = \frac{-2Q\vec{S}_0 \cdot D \pm \sqrt{(2Q\vec{S}_0 \cdot D)^2 - 4(D \cdot D)(Q\vec{S}_0 \cdot Q\vec{S}_0 - r^2)}}{2D \cdot D} \tag{1}$$

.

The sphere will collide with the corner if the discriminant is non-negative, that is $(2Q\vec{S}_0 \cdot D)^2 - 4(D \cdot D)(Q\vec{S}_0 \cdot Q\vec{S}_0 - r^2) \geq 0$. If it's positive there are two solutions, the one to use is either the smaller one, or the smaller non-negative positive one.

For an edge collision we need to find the time the sphere is at a distance r from the edge. The edge is on the line $Q(u) = Q + un_1$. Suppose $D \cdot n_1 = 0$. That would mean the closest point on the line to the sphere is fixed, so we could just use the line/sphere intercept time solution, (1). Find a new sphere velocity D' in which the velocity component in the direction of n_1 is removed: $D' = D - (D \cdot n_1)n_1$. It should be easy to verify that $D' \cdot n_1 = 0$. Since only velocity in the direction of n_1 was removed from D' spheres following $S(t) = S_0 + tD$ and $S'(t) = S_0 + tD'$ should be at distance r at the same time. So just solve (1) substituting D' for D.

(c) It's possible the sphere never collides with the cube. Show a computationally inexpensive way to test for this possibility.

For this find the time of closest approach to either the center of the cube or a corner. If the time is based on the center then collision is impossible if the sphere center never comes closer than $\sqrt{3/4}a + r$. To test for that inexpensively evaluate argument to the square root in (1) but replace r with $\sqrt{3/4}a + r$ and Q with C:

$$(2C\vec{S}_0)^2 - 4(D \cdot D)(C\vec{S}_0 \cdot C\vec{S}_0 - (\sqrt{3/4}a + r)^2) \tag{2}$$

If this quantity is negative the sphere cannot touch the cube, if its zero or positive it might hit the cube and so specific collisions must be tested for. Note that $(\sqrt{3/4}a + r)^2$ can be precomputed, though quantities like S_0 are might be changed elsewhere in the simulation.

Problem 3: A scene contains a tray upon which are cups. On the next page is code that renders the scene using a Tray class and a Cup class. A tray object can hold multiple cups.

Each object provides an array of vertices that describes its appearance (but not contained objects). So, for example, `tray->vtx_array_pointer` returns vertices that form a tray, but not the cups upon it. The vertices are in the object's own coordinate space. The `location` member gives the location of an object in its parent's coordinate space. So `cup->location` is a coordinate in the tray's coordinate space. See the code on the next page.

(*a*) The code sets the correct modelview matrix for rendering the tray, but code for setting the modelview matrix for the cup is not finished. Finish it.

The solution appears below. Note that the `mtray` transformation matrix is multiplied.

```
pMatrix mcup =  mtray * pTranslate(-cup->location) * cup->orientation;
```

(*b*) Changing a transformation is relatively time consuming, especially when it's only done for a few vertices, as might be the case with the cup at the center of the loop nest.

Show how to compute cup vertex coordinates for which the modelview matrix set for the tray is sufficient. That is, find a transformation matrix for the code below that will make the `glLoadTransposeMatrixf(mcup)` call unnecessary (but the `glLoadTransposeMatrixf(mtray)` will still be needed).

```
 pMatrix matrix = pTranslate(-cup->location) * cup->orientation; // SOLUTION
\endsol
\beginlit
 for( int i=0; i<cup->vtx_cnt; i++ )
  cup->vtx_array_pointer_new[i] = matrix * cup->vtx_array_pointer[i];
```

(*c*) If the loop from the previous part were executed each time `show_tray` was called, execution would probably be slower despite the fact that the transformation matrix is not changed as often. Explain why.

The GPU can perform the matrix multiplication faster than the CPU, so running the loop above might take more time than is saved by not having to change transformation matrices.

Problem 3, continued: The code for rendering tray and cups:

```
void show_tray()
{
  // Location of tray in world coordinate space.
  pCoor tray_location = tray->location();

  // Matrix that rotates tray coordinates (in tray space) to correct
  // position in world coordinate space.
  pMatrix tray_orientation = tray->orientation();

  // Note: modelview is the current modelview matrix, which maps
  // world coordinates to eye coordinates.

  glMatrixMode(GL_MODELVIEW);
  pMatrix mtray = modelview * pTranslate(-tray_location) * tray_orientation;
  glLoadTransposeMatrixf(mtray);
  glVertexPointer(3,GL_FLOAT,0,tray->vtx_array_pointer);
  glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArrays(GL_TRIANGLES,0,tray->vtx_cnt());

  while ( Cup* cup = tray->contents_iterate() )
    {
      // Render one cup that's on the tray.

      // Location of cup in tray coordinate space.
      pCoor location = cup->location;

      // A rotation matrix that will rotate the cup into the correct
      // position in tray coordinate space.
      pMatrix orientation = cup->orientation;

      glMatrixMode(GL_MODELVIEW);

      pMatrix mcup =                              ;   // SOLUTION HERE.



      glLoadTransposeMatrixf(mcup);

      glVertexPointer(3,GL_FLOAT,0,cup->vtx_array_pointer);
      glEnableClientState(GL_VERTEX_ARRAY);
      glDrawArrays(GL_TRIANGLES,0,cup->vtx_cnt());
    }
}
```

Problem 4: Write OpenGL code that renders a simple cup (a hollow cylinder with one end covered).

The cup center is at the origin. The radius of the cup is 11 (in world space coordinates). The cup is upright (up is $(0, 1, 0)$), and has a height of 22.

- Include vertices and normals.
- Ignore colors and lighting.
- The code should be reasonably efficient.

Solution appears below.

```
const int sides = 10;
const double delta_theta = 2 * M_PI / sides;

// Cylinder
glBegin(GL_QUAD_STRIP);
for ( double theta = 0; theta <= two_pi; theta += delta_theta )
  {
    const double cos_th = cos(theta);
    const double rcos_th = r * cos_th;
    const double sin_th = sin(theta);
    const double rsin_th = r * sin_th;
    pVect normal( cos_th, 0, sin_th );
    glNormalf( cos_th, 0, sin_th );
    glVertex3f( rcos_th, 0,      rsin_th );
    glVertex3f( rcos_th, height, rsin_th );
  }
glEnd();

// Circle
glBegin(GL_TRIANGLE_FAN);
glNormalf(0,1,0);
glVertex3f(0,0,0);
for ( double theta = 0; theta <= two_pi; theta += delta_theta )
  {
    const double cos_th = cos(theta);
    const double rcos_th = r * cos_th;
    const double sin_th = sin(theta);
    const double rsin_th = r * sin_th;
    glVertex3f( rcos_th, 0,      rsin_th );
  }
glEnd();
```

Problem 5: A scene contains a wall that has letters (forming some text) cut out of it. The wall itself is dark gray and behind the wall are light-colored objects, which you can see through the holes forming the letters.

You are given a texture with the text (say, the EE 4702 syllabus) where the letters (the ink) are black and opaque ($\alpha = 1$) and the background is transparent (any color, $\alpha = 0$). The command `glBindTexture(GL_TEXTURE_2D,texid_text);` will bind this texture to a texture unit.

Describe how to achieve this effect using a texture, the stencil buffer, and an alpha test. See section 4.1 of the OpenGL 3.0 specification for details on how the alpha and stencil test relate. *If this weren't a practice exam more details would be given.*

Assume the code `wall->render()` will send wall vertices to OpenGL and `other->render()` will send vertices of other parts of scene to OpenGL.

The solution appears below. In the first pass the stencil buffer is written with a 1 wherever there is ink. Ink is "detected" using the alpha test. If the fragment's alpha value is not 1 it fails the alpha test (and so the stencil buffer isn't written). The stencil test in pass 1 always fails (if its reached). The glStencilOp tells OpenGL to replace the existing stencil value with our own when the stencil test fails (the other two arguments are for depth test fail and pass). The glTexEnvi call tells OpenGL not to blend the texture with the wall, so we don't have to worry about the texel alpha values changing.

In pass 2 the stencil test is set up to reject fragments if the stencil buffer is 1 (and so there will be no wall where there was ink). The glAlphaFunc command turns off the alpha test (which is normal) and glTexEnvi specifies a typical texture blending function. Also a different texture is used, perhaps one that looks like concrete.

```
// Pass 1:
// Write a 1 in stencil buffer at "ink" locations,
// ink is present if texel alpha value is 1.
//
glAlphaFunc(GL_EQUAL,1.0);  // Reject if alpha != 1.
glStencilFunc(GL_NEVER,1,-1);  // Stencil func fails, so pixel not written..
glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP); // ..but replace stencil val w. 1.
glBindTexture(GL_TEXTURE_2D,texid_text); // Use texture with text.
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,GL_REPLACE); // No lighting.
glEnable(GL_TEXTURE_2D);
wall->render();

// Pass 2:
// Render wall wherever stencil value is not 1.
// Use real wall texture, not the text.
glAlphaFunc(GL_ALWAYS,1.0);  // Alpha test always passes. (Normal setting.)
glStencilFunc(GL_NOTEQUAL,1,-1);  // Pass if stencil is not 1.
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP); // Don't change stencil buffer.
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,GL_MODELVIEW); // Use lighting.
glBindTexture(GL_TEXTURE_2D, some_other_texture_id); // Wall's real texture.
wall->render();
```

Problem 6: Determine the best lighting type (emissive, ambient, diffuse, specular) to use for each of the following situations, and explain your reason.

(*a*) A triangle out in space illuminated only by a nearby star.

Diffuse lighting, with only the quadratic attenuation set to a non-zero value.

(*b*) A computer monitor screen (turned on, working properly).

Emissive.

(*c*) A glossy sphere.

Specular.

(*d*) A triangle in a room with white walls and many light sources.

Ambient.

Problem 7: Answer each question below.

(*a*) OpenGL defines several ways to combine texels with the lighted color to obtain the final color to write. Why does one need to combine a texel with a lighted color, why not just write the texel?

Combining with the lighted color provides realistic lighting, for example, a darker appearance with distance or angle from light.

(*b*) What is the advantage of using `TRIANGLE_STRIPS` over `TRIANGLES`?

With triangle strips a vertex is shared by up to three triangles, so one avoids having to waste time duplicating calculations.