Name: ______________________ **Student ID:** ______________________

This is just a practice for the second midterm. Were this a real exam, you would not be allowed any aids (books, notes, etc.), there would be twelve questions, and you would have to write your answers in the spaces provided.

1) Given a point with pixel coordinates (p, q) in an $m \times n$ image, construct the camera space ray through the pixel for a perspective projection specified in the usual way (left, right, bottom, top, near, far).

The indicated point on the near plane has camera space coordinates $\vec{x}_0 = \left(l + \frac{p}{m}(r-l), b + \frac{q}{n}(t-b), -n\right)$. This is the ray's origin, and the direction is $\vec{d} = \frac{\vec{x}_0}{\|\vec{x}_0\|}$.

2) Give pseudo-code for checking if a ray with origin $\vec{x}_0$ and direction $\vec{d}$ intersects a plane containing point $\vec{p}$ with normal $\hat{n}$.

The parametric equation of the ray is $\vec{x}(s) = \vec{x}_0 + s\vec{d}$, and an implicit equation for the plane is $(\vec{x} - \vec{p}) \cdot \hat{n} = 0$. Solving these together gives $(\vec{x}_0 - \vec{p}) \cdot \hat{n} + s\vec{d} \cdot \hat{n} = 0$. Therefore:

```
 if $\vec{d} \cdot \hat{n} = 0$:  return false (no intersection)
$s = -(\vec{x}_0 - \vec{p}) \cdot \hat{n} / \vec{d} \cdot \hat{n}$
return $s > 0$
```

3) How can you compute the barycentric coordinates of the intersection of a ray with a triangle?

We can use the `orient` function (or signed volume, etc.). If the ray goes from point $\vec{x}_0$ to point $\vec{x}_1$ and the triangle has vertices $\vec{x}_2$, $\vec{x}_3$, $\vec{x}_4$ then:

$$\begin{aligned} A &= \text{orient}(\vec{x}_0, \vec{x}_1, \vec{x}_3, \vec{x}_4) \\ B &= \text{orient}(\vec{x}_0, \vec{x}_1, \vec{x}_4, \vec{x}_2) \\ C &= \text{orient}(\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3) \\ S &= A + B + C \end{aligned}$$

$$\alpha = \frac{A}{S} \qquad \beta = \frac{B}{S} \qquad \gamma = \frac{C}{S}$$

4) Give recursive pseudo-code for efficiently intersecting a ray with a large set of primitive shapes organized in a BVH.

Call `intersect()` with the ray and the root of the BVH:

```
intersect(ray, node) =
  if ray doesn't intersect node's bounding volume:
    return false
  loop over any primitive shapes stored in this node:
    if ray intersects the shape, return true
  loop over any children of this node:
    if intersect(ray, child), return true
  otherwise, return false
```

5) In what sense can raycasting have better asymptotic performance than Z-buffer rasterization?

For an image with N pixels and depth complexity D, raycasting with a good acceleration structure can take $O(N \log D)$ time, but Z-buffer will take $O(ND)$ time. As the depth complexity increases, raycasting will eventually be faster.

Name: ____________________ **Student ID:** ____________________

6) Give recursive pseudo-code for efficiently determining if any of a large set of primitive shapes organized in a BVH intersect each other.

We want to avoid testing every primitive shape against every other shape, as that would be very slow—we want to use the BVH to cull away any tests from two branches of the tree that don't have overlapping bounding volumes. We'll use two functions to do this, `self-intersect` which checks a node against itself and `intersect` which checks two distinct nodes against each other. We'll assume for simplicity that shapes are only stored at leaf nodes.

Start by calling `self-intersect()` with the root of the BVH:

```
self-intersect(node) =
  if this is a leaf node:
    loop over all pairs of shapes stored here:
      if they intersect return true
  else:
    for every pair of children nodes:
      if the bounding boxes of child1 and child2 overlap:
        if intersect(child1, child2) return true
    for every child node:
      if self-intersect(child) return true
  return false

intersect(node1, node2) =
  if node2 has any children:
    for every child of node2:
      if node1's and child's bounding volume overlap:
        if intersect(node1, child) return true
  else if node1 has any children:
    for every child of node1:
      if node2's and child's bounding volume overlap:
        if intersect(child, node2) return true
  else:
    for every pair of primitive shapes from node1 and node2:
      if they intersect return true
  return false
```