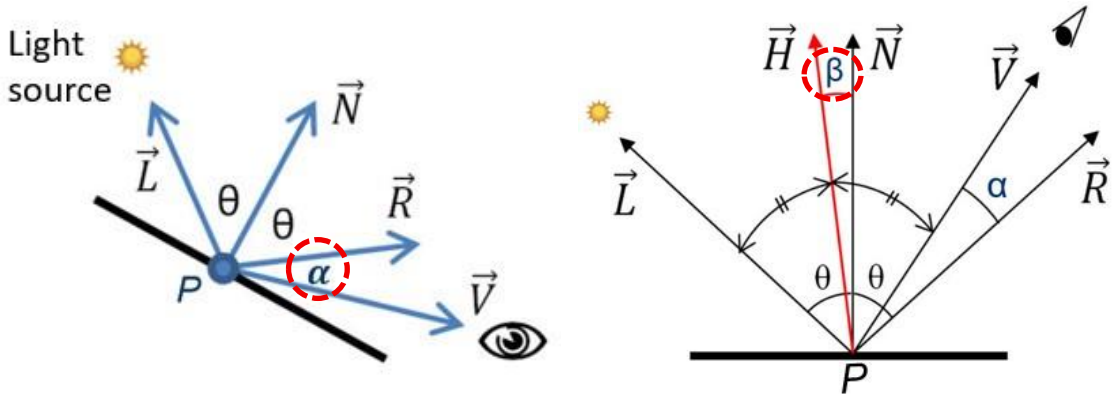
CS535 Fall 2025

Final exam Solution Set (100 + 10 extra points), December 18, 2025

Name___sample_____

1. (4 points)

Given the normal *N* of a surface at a given point *P* and an incident ray *L*, we need to compute the specular reflection ray *R* at that point to compute its shade, the so-called Phong model. Actually what we really need is $\cos(\alpha)$ (see the figure on the left for α). Note that *V* is the unit vector from *P* to the view point.

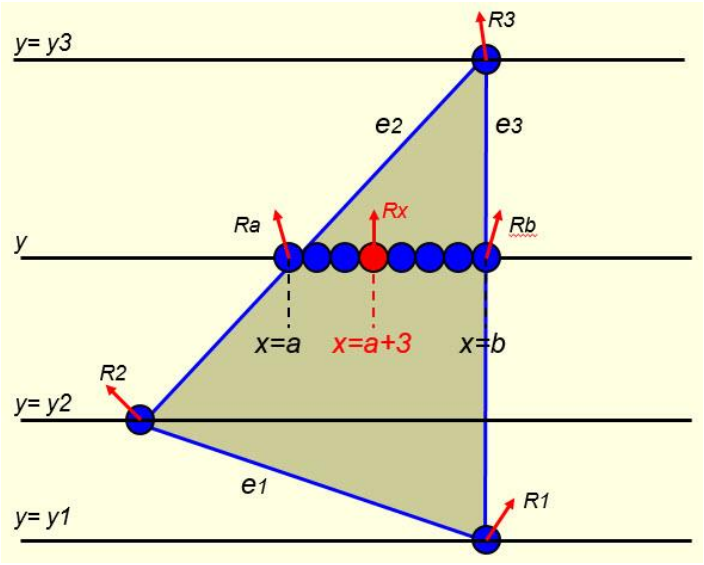


An alternative is to compute the vector *H* (see the figure on the right for *H*) so we can use $\cos(\beta)$ instead of cos(α) in the computation of the specular reflection component of the shade at *P*. This alternative approach is the so-called Blinn-Phong model. The reason to use this alternative approach is because the computation process of *H* is simpler than the computation process of *R* and the shading result computed this way is similar to the result of Phong model. You task here is to show me how is *H* computed and also how is $\cos(\beta)$ computed. Note that all the vectors shown in the above figures are normalized, i.e. each of them is of unit distance.

H = $(L+V)/\|L+V\|$ $\cos(\beta) =$ $N \cdot H$

2. (8 points)

When rendering a 3D polygonal mesh, we usually use an incremental method to estimate the specular reflection vectors for points of each triangle of the mesh if the specular reflection vectors at the vertices of each triangle are known. In the following figure, assume R1, R2 and R3 are the specular reflection vectors of the three vertices of a triangle of a 3D mesh.



If each edge in the Bucket-Sorted Edge Table (ET) is represented as follows

Then for edge 'e2',

Rmin = R2

$\Delta R = \dfrac{(R3 - R2)}{(y3 - y2)}$

For scan line y, once we have Ra and Rb from edge e2 and edge e3 (e2 and e3 are edges in the Active Edge List now), we compute a step size for the specular reflection vector ΔxR as follows:

$\Delta xR = \dfrac{(Rb - Ra)}{(b - a)}$

The specular reflection vector for x=a+1 would be

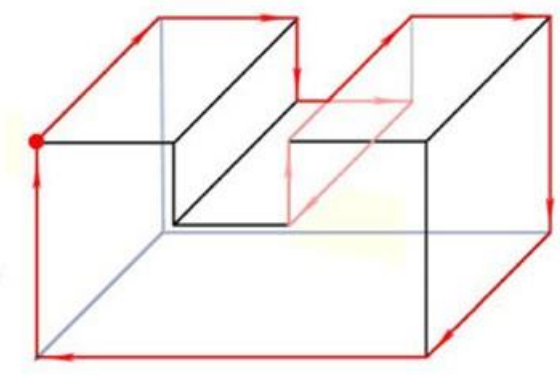
R_{a+1} = Ra + ΔxR

3. Gouraud shading (intensity-interpolation shading) and Phong shading (normal-interpolation shading) can both be used to eliminate intensity discontinuities when rendering a polygonal mesh. However, Gouraud shading would generate the so-called ***Mach band effect*** and Phong shading would not. Can you think of a reason for Gouraud shading to get the Mach band effect? (10 extra points)

4. (10 points)

Three techniques have been introduced to generate shadows: **shadow volume method**, **shadow map method**, and **ray tracing method.**

(a) For the shadow volume method, if the silhouette (outline) of an object with respect to the point light source is marked with dark red line segments as shown in the following figure, then how many shadow polygons should be generated for this object?

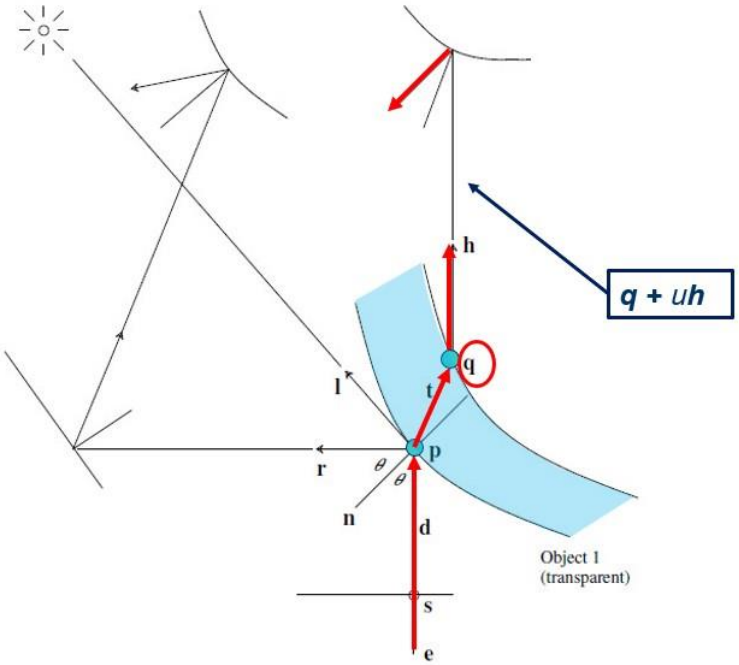


Put your answer in the red box on the right [10]

(b) The shadow map method is performed by applying the Z-buffer method twice, once with respect to the view point and once with respect to the light source. In the first case two arrays, a depth array and a color intensity array, are created. In the second case only a depth array is created. For operation efficiency consideration, should the Z-buffer method be performed with respect to the view point first or the light source first? Mark the appropriate box below.

View point [ ] light source [X]

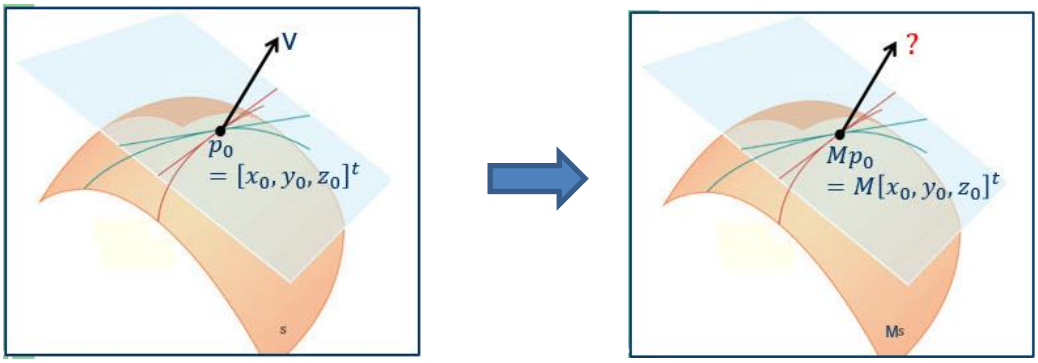
(c) The ray tracing method determines if a point is in shadow by generating a ray from the point to the light source (see the vector 'l' in the following figure) and check if that ray intersects anything before it reaches the light source.



In the case shown in the above figure, does the ray tracing method check if the point 'q' is in shadow? Justify your answer in either case. (4 points)

No if the vector 'h' is inside the object. But usually this is not the case, so the ray tracing process will check if 'q' is in shadow by checking if the vector 'h' would intersect anything before it reaches the light source.

5. (5 points)
Given a point p_0 of a surface *S* with normal vector *V* (see the left figure below). If the surface *S* is transformed by an MV matrix *M* to a new location MS, then what is the normal vector of p_0 at its new location (see the right figure below)?



V is represented as column vector then the new normal is $(M^{-1})^t V$.
Otherwise, it is $(M^{-1})^t V^t$.

6. (10 points)
Gouraud shading (intensity-interpolation shading) and Phong shading (normal-interpolation shading) can both be used to eliminate intensity discontinuities when rendering a polygonal mesh. However, Gouraud shading could generate the so-called ***Mach band effect*** and Phong shading would not. Nevertheless, Phong shading has a disadvantage over Gouraud shading. What is that disadvantage? Is there a way to overcome that disadvantage of Phong shading? (10 points)

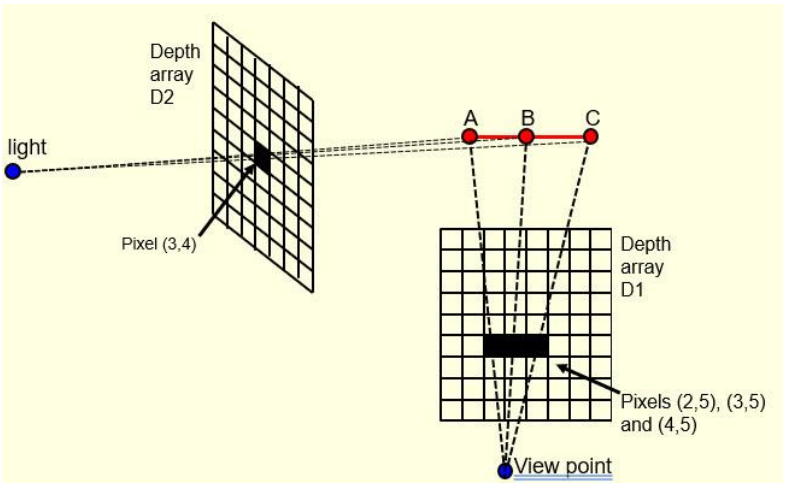
Sol.

Phone shading is not as efficient as Gouaud shading because Phone shading has to compute the specular reflection vector R for each point of the polygon mesh. One option is to use the bisector of the angle between L and V, *H*, to compute the color/intensity, the so called Blinn-Phone shading. This can only improve the performance to certain degree though. Another option is to use an incremental method to estimate the specular reflection vector for each point of the triangle.

7. (10 points)
The **shadow map based** 'shadow generation' algorithm is easy to implement. But it has a potential problem. What is it? What is the reason for getting this potential problem? Is there a way to overcome this potential problem?

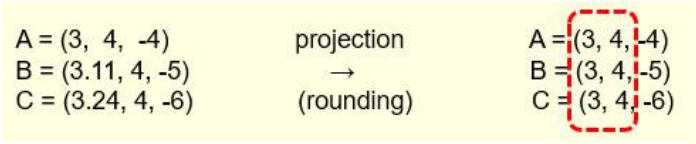
Sol:

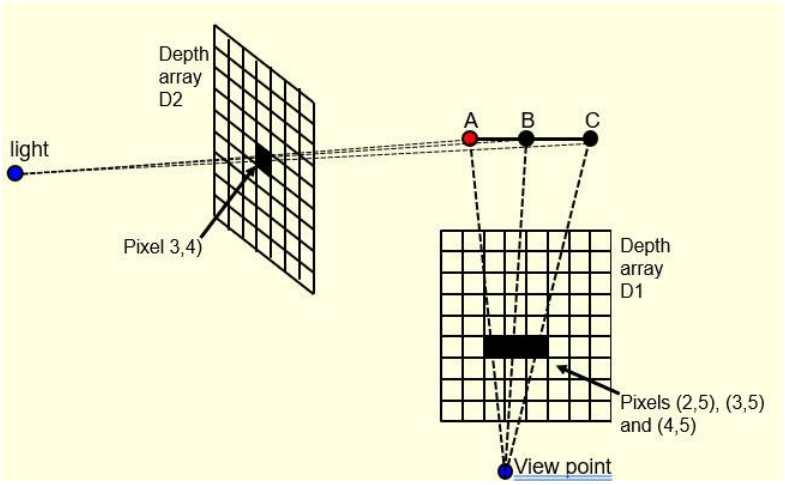
Consider the case shown in the following figure. The line segment AC is visible both to the view point and the light source. Therefore, theoretically, none of the points of the line segment should be in shadow.



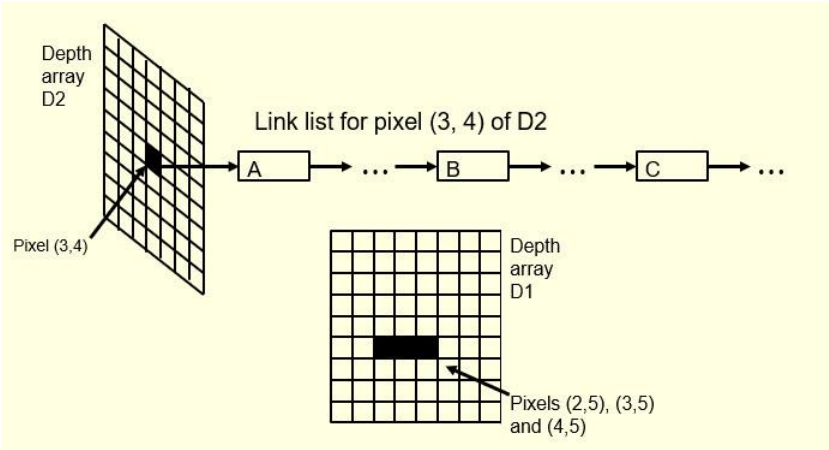
However, if the coordinates of points A, B and C in the light source coordinate system are (3, 4, -4), (3.11, 4, -5) and (3.14, 4, -6), respectively (see the left side of the

following figure), then after projection, they will all have (3, 4) as the x- and y-coordinates (see the right side of the following figure). Since A has the largest depth (z-coordinate) value, entry (3, 4) of the depth array D2 will store the depth of A, -4. The depth value of B is -5 and the depth value of C is -6, both are smaller than the depth of A, content of entry (3, 4) of D2. Therefore, both points (actually the entire line segment except A) will be considered not visible to the light source and consequently will be shown as points in shadow (see the next figure).





To overcome this problem, one way is to increase the resolution of the display surface (this is not really a solution, but a way to reduce the seriousness of the problem). A real solution to this problem is to build a link list for each entry of the depth array D2 (see the following figure).



When the Z-buffer method is performed with respect to the light source, each point projected into pixel (m, n) will not only be comparing its depth (z-coordinate) value with the current content of entry (m, n) of the depth array D2, the point (with coordinates before projection) will also be inserted into a link list for that entry. The link list is sorted by depth. Therefore for entry (3, 4) of D2, we would have A, B, and

C in the link list (in that order; see the above figure).

For each point that is visible to the view point, say point B, we first compute coordinates of B in the light source coordinate system, (3.11, 4, -5). After rounding, we get (3, 4, -5). Then we compare the depth of B, -5, with the content of entry (3, 4) of D2. If -5 equals the content of entry (3, 4), B is obviously not in shadow. If -5 is smaller than the content of entry (3, 4) of D2, we then search through the link list of entry (3, 4) to find B. If none of the points in the list before B blocks B, then B is not in shadow. Otherwise, B is classified as in shadow. Question: how do you tell if a point in the list before B blocks B? (if light source, that point and B are collinear)

8. (5 points)

(a) What is the reason for using a **binary tree** to record the ray tracing process for each pixel of the screen? (3 points)

The ray tracing process generates a specular reflection ray and a refraction ray for the first point hit by a viewing ray. The specular reflection ray will return contribution from neighboring objects and the refraction ray will return contribution of light sources behind the object if the object is transparent. A binary search tree **(BST)** is a natural choice for this process because for each node, we can use the left branch to trace the work of the specular reflection ray and use the right branch to trace the work of the refraction ray.

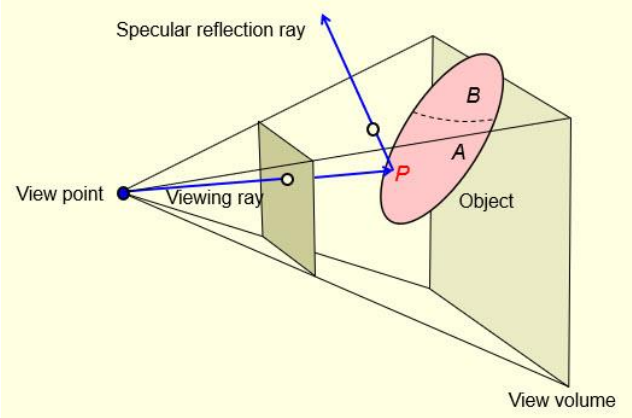
(b) If none of the objects in the 3D scene is transparent, do we still need a binary tree to record the ray tracing process for each pixel of the screen? If not, then what else approach can be used? (5 points)

No. In such a case, a **linear array** would be sufficient.

9. **(10 points)**

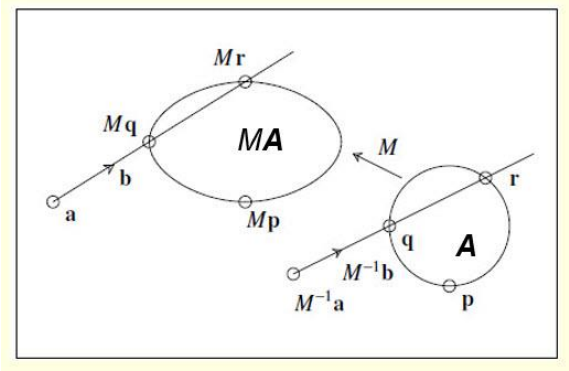
Clipping is not necessary for the ray tracing process? **Why?**

Even if a portion of an object is outside the view volume (like portion B of the object in the following figure), since each viewing ray generated in the ray tracing process is bounded by the four bounding planes of the view volume, the first intersection point returned by the viewing ray will always be a point on the visible portion of an object (the portion of the object that is inside the view volume), such as the point *P* in the figure below. If the specular reflection ray or refraction ray generated for an intersection point hits a bounding plane of the view volume, we simply return the background color for that specular reflection ray or the refraction ray. So there is no need for a clipping process in the ray tracing algorithm.



10. (6 points)
When ray trace an instance of an object transformed by a matrix M, we usually perform the ray tracing process in the space of the original object. What is the advantage of doing the tracing this way?

Doing it this way, we only need to maintain one ray-object intersection point computation procedure for each primitive object, instead of performing a separate ray-object intersection point computation for each instance of a primitive object.
For instance, in the following figure, ***A*** is a primitive object and ***MA*** is an instance of ***A*** through the mapping of the transformation matrix *M*. To find the intersection points of the ray *a+tb* with ***MA*** directly, we have to develop separate code to perform the ray-object intersection point computation for *a+tb* and ***MA***. By transforming the ray *a+tb* into the space of ***A***, we can use the ray-object intersection point computation procedure for ***A*** and the ray $M^{-1}a + t\,(M^{-1}b)$ to find the intersection points and the normals at the intersection points and then transform the intersection points into the space of ***MA*** by *M* and the normals by $(M^{-1})^t$. This approach saves both computation time and software implementation.



11. (10 points)

A modern CPU can have 4 or 8 cores, but a modern GPU can have thousands. These GPU cores can be used for computationally intensive tasks through the use of ***compute shaders***. Compute shaders are programmed in GLSL and run independently. A computer shader can perform parallel computing in the following sense: if a compute shader is required to perform a task on *n* different data sets, one can first creates n copies of the compute shader (invokes the compute shader n times) and then assign each copy of the compute shader a different data set (assign a different task ID (invocationID)). These copies of the compute shader then run in parallel to perform the task on assigned data sets. In the following box, explain how these two things are implemented in a computer shader program, especially the GLSL commands/variables needed for these two steps.

First, one needs to use the following command to define a 1D, 2D or 3D grid of *n* nodes with each node being a work group (core, but simply think of it as a copy of the compute shader program):

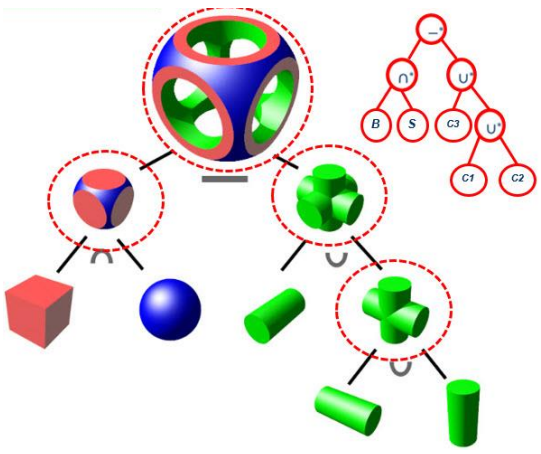
glDispatchCompute( )

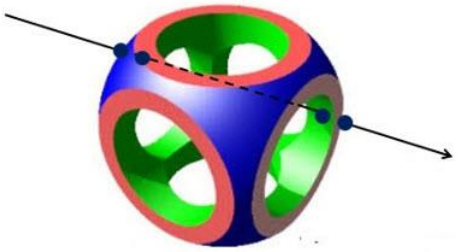
Here we assume the size of each work group to be 1.

Next, in the main() of the compute shader program, each work group will get a task ID assigned by the special variable ***gl_GlobalInvocationID*** and then perform the assigned task independently.

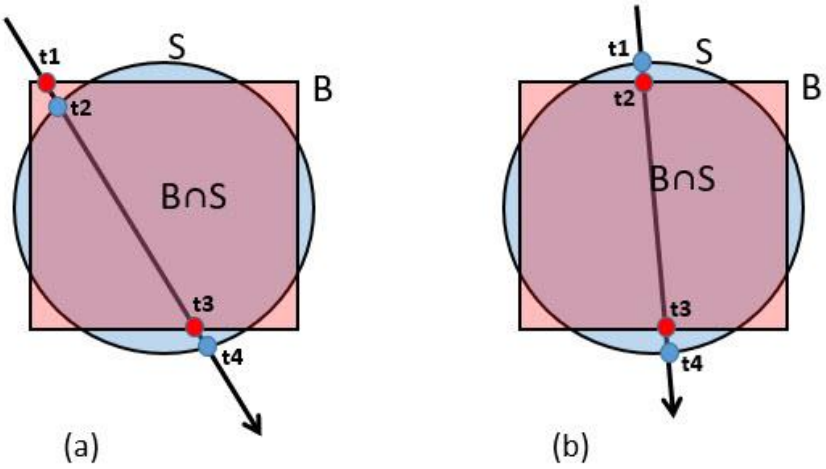
12. (4 points)

Given a virtual object represented as a CSG (Constructive Solid Geometry) tree (see Section 10.10 for the definition of a CSG tree), one can use ray casting or even ray tracing technique to render this virtual object on screen. To use ray casting technique to render a CSG object, we need to find the intersection points of each ray with the object. For instance, for the CSG object given below (left figure), for the given ray, we need to find the parameters of the intersection points of the ray with the object (right figure).





In the following, use the two given cases (intersection of a 2D sphere and a 2D cube, such as the intersection of B and S in the above CSG representation) to explain which two parameters should be reported for each case and why. (10 points)



In case (a), by taking the intersection of intervals [t1, t3] (the portion of the ray that is inside B) and [t2, t4] (the portion of the ray that is inside S), we get the interval [t2, t3] (note that t1 < t2 < t3 < t4). So the intersection points of the ray with B∩S in this case are t2 and t3.

In case (b), by taking the intersection of intervals [t1, t4] (the portion of the ray that is inside S) and [t2, t3] (the portion of the ray that is inside B), we get the interval [t2, t3] (note that t1 < t2 < t3 < t4). So the intersection points of the ray with B∩S in this case are also t2 and t3.

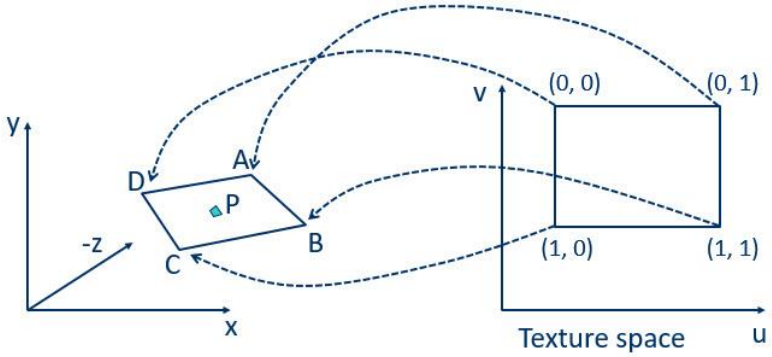
13. (10 points)
Can texture mapping be integrated into the ray tracing process? Justify your answer and be specific. For instance, if your answer is YES, you must tell me how it is done, including how you get the texture space coordinates, how to handle out-of-range problem, and how to handle the do-not-directly-match-a-texel problem. If your answer is NO, then you have to tell me why this is not possible, with all the supporting arguments.

Yes, ray tracing can be used with texture mapping to reproduce surface texture on an otherwise smooth polygon surface.

How is this done? The most efficient way to do this is to perform the following steps after the intersection point to the closest object has been determined so that we don't spend time calculating texture coordinates for hidden objects.

getTexture(point p)
Convert p from world coordinates to local object coordinates (lp)
Convert lp point to texture space coordinates (u,v)
If the (u,v) coordinates are out of range use tiling, mirroring, or another technique to get appropriate coordinates.
If the (u,v) coordinates do not match directly use the nearest neighbor or interpolation to get the correct color
Return the color to the ray tracer

Since we are still just returning a color to the ray tracer, texture mapping can be easily integrated into the tracing process.



14. (8 points)
(a) [Surface Modeling] After performing one Catmull-Clark subdivision, all the faces in the new control mesh of a Catmull-Clark subdivision surface are four-sided. Why? (4 points)

Note that each new face point is connected to each adjacent new edge point and each new vertex point is connected to each adjacent new edge point.
If each new edge formed by a new face point and a new edge point is called an F-E edge and each new edge formed by a new vertex point and a new edge point is called a V-E edge, then it is easy to see that each new face is formed by two F-E edges and two V-E edges. Therefore, each face is a four-sided face.

(b) [Surface Modeling] After performing one Catmull-Clark subdivision, the number of extra-ordinary points in the control mesh of a Catmull-Clark subdivision surface would remain a constant. Why? (4 points)

Note that the degree of a new edge point is always four, the degree of a new vertex point is the same as the degree of the original vertex, and the degree of a new face point is equal to the number of edges of the original face. Since new faces created after the first Catmull-Clark subdivision are all four-sided, the degrees of new faces points created subsequently are always four. Hence, after the first Catmull-Clark subdivision, only those new vertex points corresponding to old extraordinary vertices remain extraordinary. Hence, the number of extra-ordinary vertices remains a constant after the first Catmull-Clark subdivision.