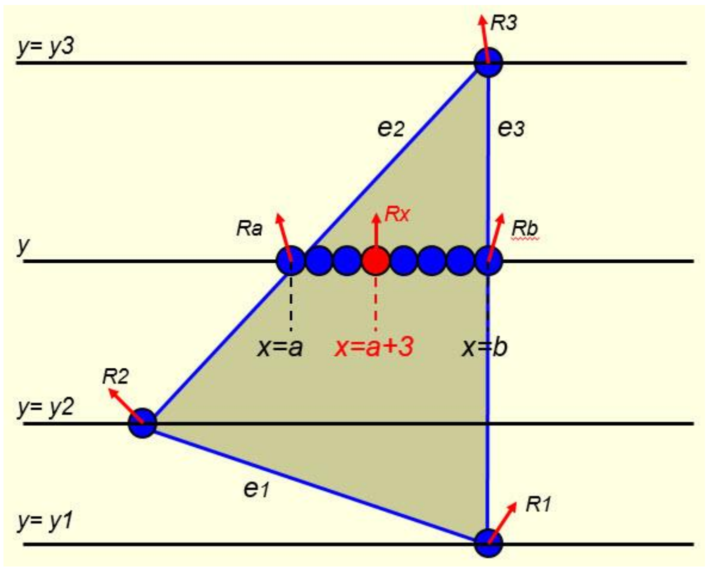
# CS535 Fall 2024

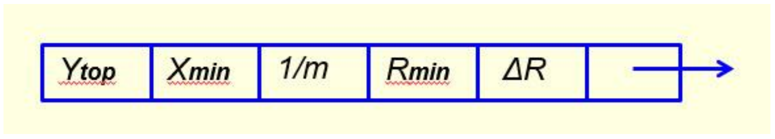
**Final exam Solution Set (100 + 10 extra points), December 17, 2024**

Name____Sample___________

1. Given the normal of a surface at a given point *N* and an incident ray *L*, we need to compute the specular reflection ray *R* at that point to compute its shade. Develop an incremental method to compute that vector for points of a triangle, assuming *R1*, *R2* and *R3* at the three vertices of the triangle are already given. (10 points)



**Sol:**
The normal of a triangle is a constant. Therefore, to compute the intensity/color for a point of the triangle, we only need to know the specular reflection vector of that point. Computing the real specular reflection vector for each point of the triangle is a very expensive process, so one alternative (besides the Phong shading) is to estimate the specular reflection vector for each point of the triangle if we know the specular reflection vectors at the vertices of the triangle. This can be done as follows.
For each edge of a polygon, we use a representation as follows:



***Rmin*** is the specular reflection vector of the lower vertex of the edge. So, for edge e2, this entry would be R2 and for edge e3 this entry would be R1. **ΔR** is the step size for specular reflection vector in y direction. So for edge e2, **ΔR** is computed as follows:

$$\Delta R = (R3 - R2)/(y3 - y2)$$

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 $\Delta_x R$ as follows:

$$\Delta_x R = (Rb - Ra)/(b - a)$$

Then the specular reflection vector for each subsequent pixel in the span [a, b] would simply be the sum of the specular reflection vector of the previous pixel and $\Delta_x R$. So, the specular reflection vector for x=a+1 would be

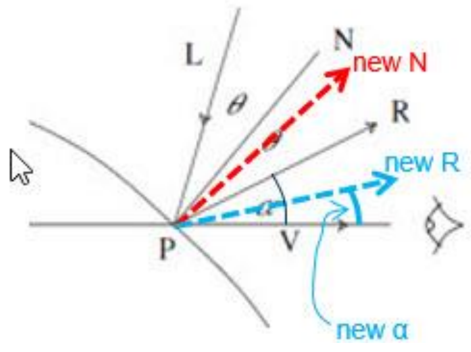
$$R_{a+1} = R_a + \Delta_x R$$

And the specular reflection vector for x=a+2 would be

$$R_{a+2} = R_{a+1} + \Delta_x R.$$

2. Gouraud shading (intensity-interpolation shading) and Phong shading (normal-iterpolation 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. Can you think of a reason for Gouraud shading to get the Mach band effect? (10 extra points)

   **Sol:**
   The Mach band effect has something to do with the way the normal of a vertex $v$, $Nv$, of the polygon-mesh is defined. Defining the normal of a vertex as the average of normals of the adjacent polygons is not the best way to define the normal of a vertex, especially when the normal of one adjacent polygon is quite different from normals of the other adjacent polygons. In such a case, intensities computed for points (pixels) of that particular polygon using "intensity interpolation shading" would be quite different from the original value, especially when specular reflection is considered (see the above figure). Phong shading wouldn't have such a problem b/c Phong shading interpolates normal vectors directly.



3. The ***shadow volume based*** 'shadow generation' algorithm can be integrated with the scan-line hidden surface elimination process so that we can do hidden surface elimination and shadow generation at the same time. How are **shadow polygons** used by the scan-line method to determine if a point (pixel) is in shadow? If necessary, draw a figure to illustrate the process. (10 points)

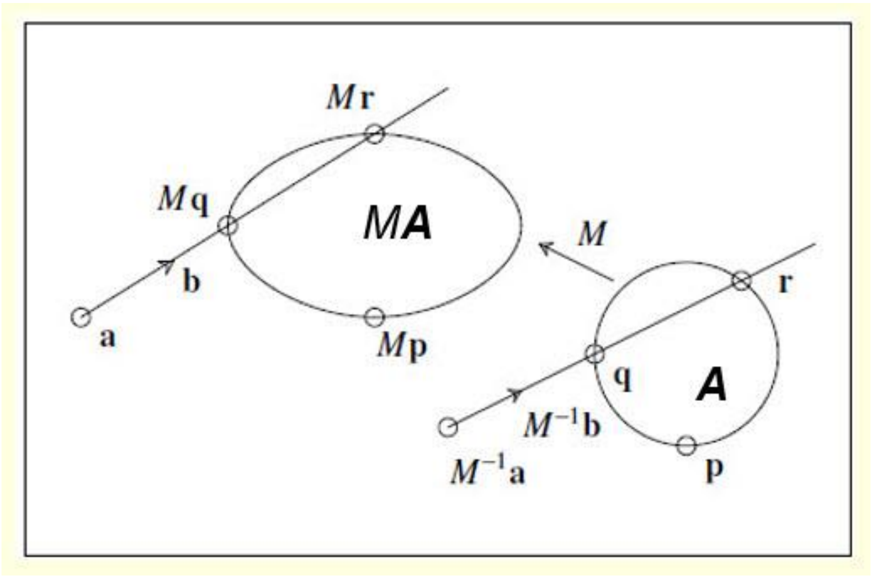
   **Sol:** See slides 6 and 7 of the notes "Lighting and Shadows II".

4. 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? (10 points)

   **Sol:** For the potential problem and the cause of the problem, see slides 16, 17 and 18 of the notes "Lighting and Shadows II". To overcome this potential problem, one way is to increase the resolution of the display surface. However, this is not really a solution, but a way to reduce the seriousness of the problem. For a real solution to this problem, see Question 5 of the solution set for HW5.

5. 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/primitive. What is the advantage of doing the tracing this way? (10 points)

   **Sol:**
   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.



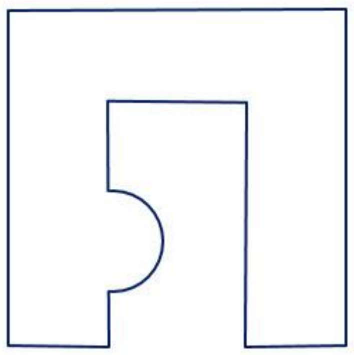
   For instance, in the above figure, ***A*** is a primitive object and ***MA*** is an instance of ***A*** through the mapping of the transformation matrix *M.* To find the intersection points of the ray *a+tb* with ***MA*** directly, we have to develop separate code to perform the ray-object intersection 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 $M^{-1}a + t\,(M^{-1}b)$ , and then transform the intersection points into the space of ***MA*** by *M*. This approach saves both computation time and software implementation.

6. The CSG (Constructive Solid Geometry) tree representation technique introduced

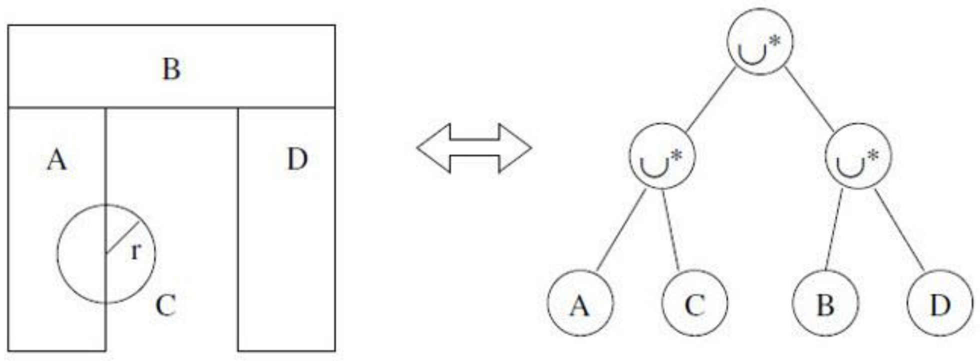
in Section 10.10 is not unique, i.e., there are usually more than one CSG representation for a CSG object. Are there occasions that the CSG representation for a CSG object is unique? Either way, justify your answer. (10 points)

**Sol:**
The answer is NO in general unless the object is a primitive itself. In that case the CSG tree representation of the object has only one node, a primitive node.

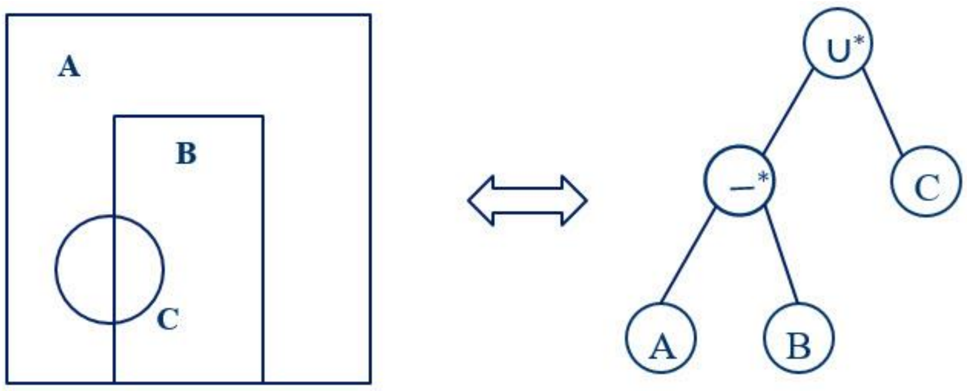
We will use a 2D example to show that in general it is possible to use different construction procedures to build a solid. Consider, for instance, the 2D object shown below.



This object can be constructed using two types of primitives: unit block and unit circle, through three regular union operations as follows:



It can also be constructed using two types of primitives: unit block and unit circle, but through a regular difference operation and a regular union operation, as follows:



So the CSG representation technique is not unique for this object.

However, if an object is itself a primitive, such as a unit cube, then the CSG tree representation of this object has one node only, a primitive node for the unit cube. Here we ignore the possibility of representing the object as the union (or, intersection) of two identical unit cubes.

7. Can ray tracing reproduce texture of a surface? Justify your answer. (10 points)

**Sol.**
Yes, ray tracing can reproduce the texture of a surface. To do this, we need to incorporate UV mapping into the ray tracing process so that a texel (or, a set of texels) in the texture image can be identified and used in the rendering process of the intersected point of the surface, instead of the standard approach.

Specifically, first, we generate a ray and find the first intersection point on the object as we did in a normal ray tracing process. Then we derive the u and v coordinates of the intersection point on the surface. Instead of setting the color of the intersected point using the standard approach, we use the UV texture coordinates to find the corresponding entry (texel) in the texture image and set the initial color of the intersected point to be that entry of the texture image. After that, we can calculate diffuse light, specular light and refraction as we did in normal ray tracing process. With such modification to a normal ray tracing process, proper texture information can be applied to each intersection point.

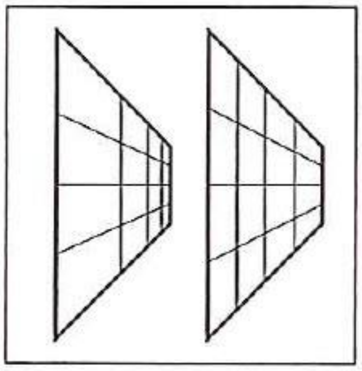
If the projection type is perspective, we also need to apply the perspective correction on the UV mapping.

8. Why perspective correction is necessary when doing texture mapping? How should it be done? Your answer should address two issues here:
   (1) Theoretically, how can it be done?
   (2) Practically, how should it be done efficiently?

   The term "efficiently" means the process is so efficient that it can be integrated with the triangle rasterization process without changing its performance much at all. (10 points)

   **Sol.**

   Perspective correction is necessary when doing texture mapping because when we do scan conversion, the UV coordinates are calculated by linearly interpolating points in the screen space. So the linearly interpolated UV texture coordinates will not give the perspective effect. Consider the following graph:



We get the image on the right if we do interpolation in screen space, which is not correct. The correct one is shown on the left which shows the perspective effect.

To do texture mapping perspective correction, after we derived the incorrect UV texture coordinates, u and v, we need to divide them by the depth value Z, which is also interpolated. So we can use the new u and v coordinates to find the proper texel value from the texture image which will correctly represent the perspective effect when mapped to the object.

At each pixel, instead of interpolating the texture coordinates directly, the coordinates are divided by their depth Z (relative to the viewer), and the reciprocal of the depth value 1/Z is also interpolated. Then, we take the new U and V values, index into the texture map and find the appropriate entry for the screen pixel. A pseudo code for this process is shown below.

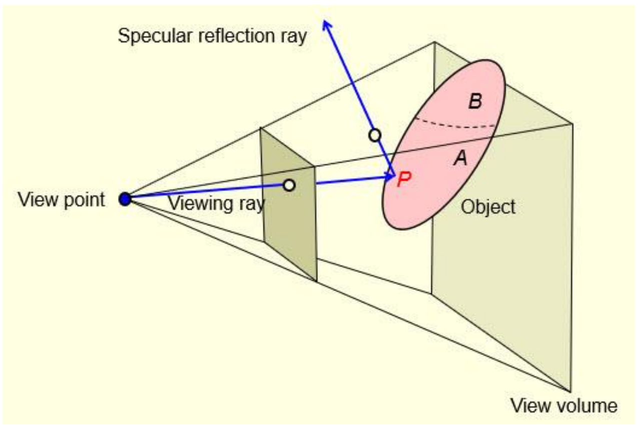
Pseudo-code:

```
    su = Screen-U = U/Z
    sv = Screen-V = V/Z
    sz = Screen-Z = 1/Z
    for x = startx to endx
        u = su/sz
        v = sv/sz
        PutPixel(x, y, texture[v][u])
        su+ = Δsu
        sv+ = Δsv
        sz+ = Δsz
  end
```

Note that in the above loop, u=U and v=V only for the start pixel, not for the remaining pixels.

9. **Clipping** is not necessary for the ray tracing process? **Why?** (10 points)

**Sol:**
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. 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. (10 points)

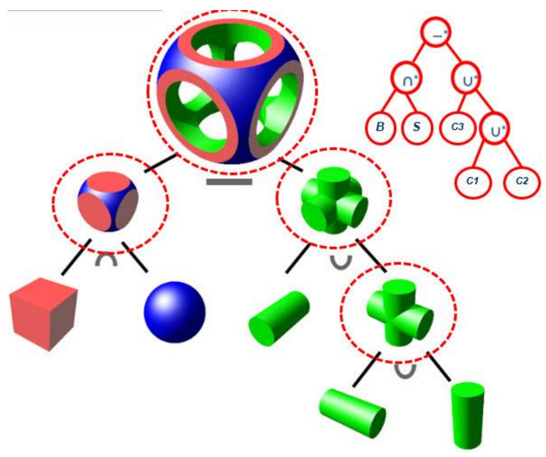
First, one needs to use the following command to define a 1-D, 2D or 3D grid of *n* nodes with each node being a work group (core, but simply think of it as a copy of the compute shader program):

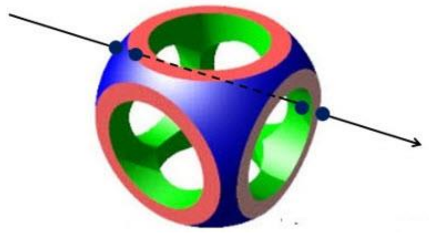
***glDispatchCompute( )***

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

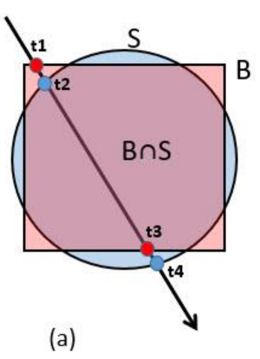
Next, in the main() of the compute shader program, each work group will get a task ID assigned by the special variable ***gl_GlobalInvocationID*** and then perform the assigned task independently.

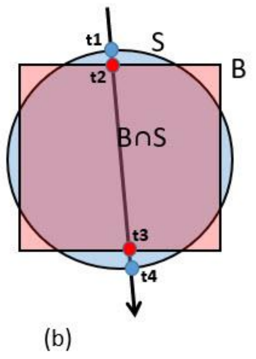
11. Given a virtual object represented as a CSG (Constructive Solid Geometry) tree, one can use ray casting or even ray tracing technique to render this virtual object on screen. To use ray casting technique to render a CSG object, we need to find the intersection points of each ray with the object. 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.