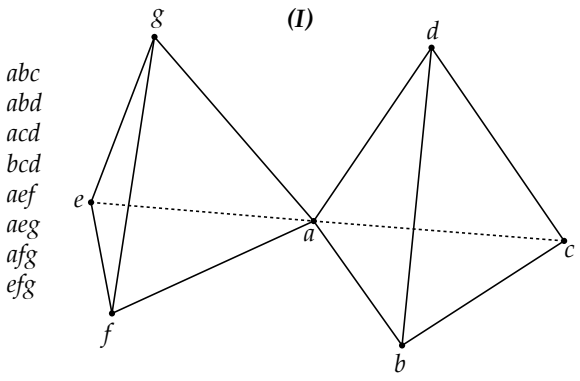
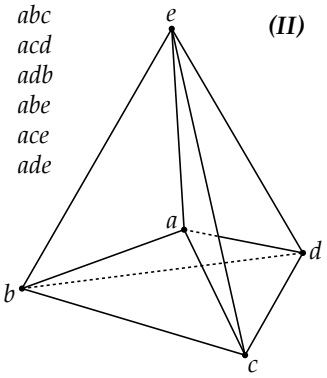
Stanford CS248A: Computer Graphics
Participation Exercise 4

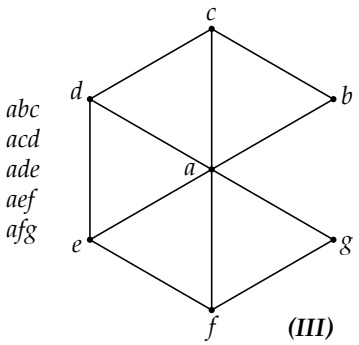
This exercise is graded for CREDIT ONLY. Serious attempts to answer the problems will be given full credit, even if the answers are incorrect.

Problem 1: Meshes and Manifolds

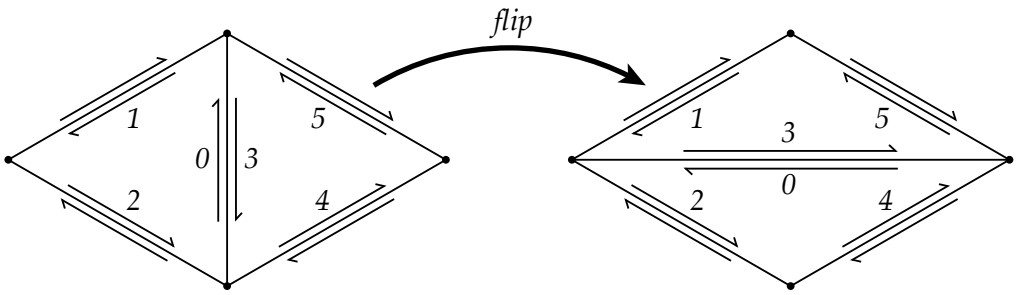
A. A triangle mesh has *boundary* if at least one of its edges is contained in only one triangle. A triangle mesh is a *manifold* if (i) every edge is contained in one or two triangles, and (ii) every vertex is contained in a single loop or fan of triangles. Using these definitions, indicate whether each of the examples below are manifold and/or have boundary. If a mesh is non-manifold, specify one of its non-manifold vertices or edges. If a mesh has boundary, specify one of its boundary edges.





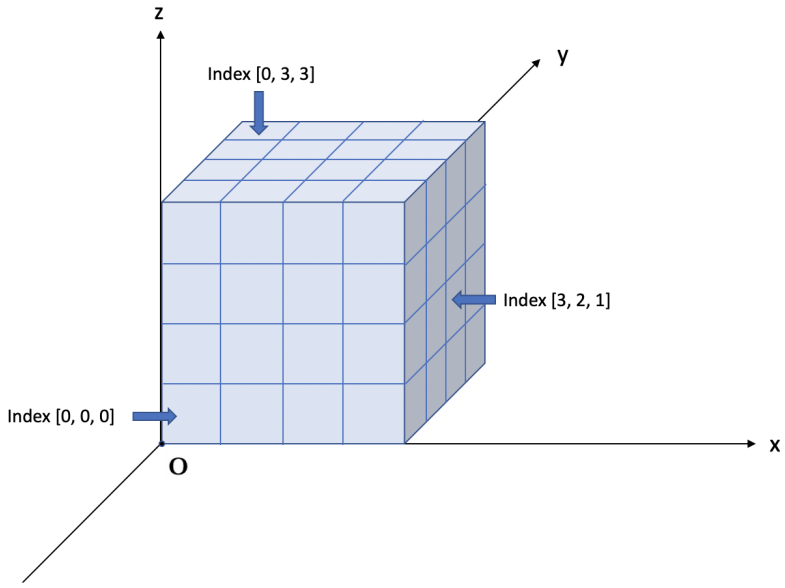


B. Consider a simplified halfedge data structure that does not explicitly encode vertices, edges, or faces—each halfedge keeps track of only its "next" and "twin" halfedges. This data is stored in two fixed-length arrays `N` and `T` of size $2E$, where E is the number of edges in the mesh. In particular, suppose we give each halfedge a unique index in the range 0 to $2E - 1$. Then for any halfedge `k`, `N[k]` is the index of its next halfedge, and `T[k]` is the index of its twin. Update these arrays to reflect the edge flip below, using as few assignments as possible.



Problem 2: Ray-Grid Intersection

We are interested in casting rays onto a uniform 3D grid shown below. The grid is 4×4×4 and each grid cell is a cube with side length 1. The origin of the grid (**O**) is located at the origin of the coordinate system: $\mathbf{O} = (0, 0, 0)$.



Recall that a ray can be represented as $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$. Assume that the origin of the ray is ALWAYS outside of the grid and recall that a plane can be represented as $\mathbf{N}^T\mathbf{x} = c$.

Hint: A plane parallel to the XY plane has $\mathbf{N} = (0, 0, 1)$*, a plane parallel to the XZ plane has* $\mathbf{N} = (0, 1, 0)$ *and a plane parallel to the YZ plane has* $\mathbf{N} = (1, 0, 0)$.

Assume that you have access to the following:
(1) Struct `Plane` that stores normal `N` and offset `c`
(2) Struct `Ray` that stores origin `o` and direction `d`
(3) Function `bool rayPlaneIsect(Plane plane, Ray r, float* t)` that takes as input a plane and a ray and set `t` to be the intersection. A boolean is returned to indicate if an intersection is found.

THE QUESTIONS BEGIN ON THE NEXT PAGE

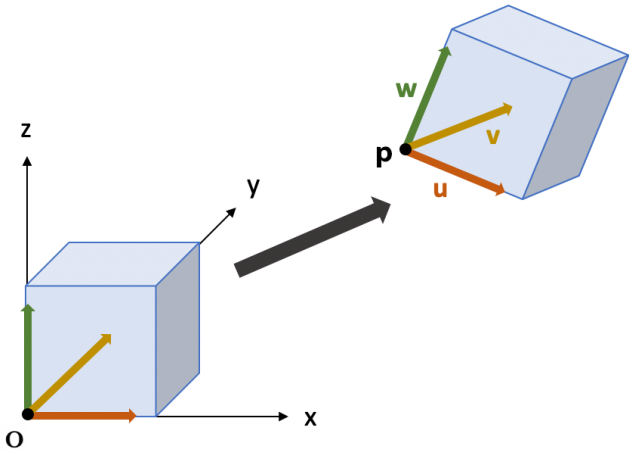
A. Give an algorithm for finding the index of the first cell getting hit by the ray. The index of a cell should be represented as a 3D vector (a, b, c) corresponding to the indices in x, y, and z axis, as shown in the figure above. **The complexity of your algorithm SHOULD NOT scale as the number of cells in the grid increases.** Your solution can be described in words or rough pseudocode, but make sure it's clear what your algorithm is. Be precise about how you initialize the given structs and how you use `rayPlaneIsect()` to determine the ray-grid intersection. Hint: figure out where the ray first hits the volume of the grid, then turn that into an index.

```
struct Plane {
  vec3 N;        // unit normal
  float c;       // offset
};
struct Ray {
  vec3 o;        // ray origin
  vec3 d;        // ray direction
};

bool rayPlaneIsect(Plane plane, Ray r, float* t);
```

B. Now we want to remove some cells from the grid. Given a collection of cell indices that are removed `removedCellIndices`, give an algorithm for finding the index of the first cell hit by the ray. **Now, the complexity of your algorithm CAN scale linearly with the number of grid cells.** Your solution can be described in words.

```
vec3[M] removedCellIndices = [vec3(0,1,0), vec3(1,2,1), ...]
```

C. Now consider the case where the grid is translated and rotated such that the grid origin **O** is located at position **p** and the three sides next to **O** now have unit directional vector **u**, **v** and **w**. An illustration is provided below:



We want to find the index of the first grid cell hit by a ray. Assume that you can access your solution to part B via function `getCellIndex(Ray r)`. Please implement the function:
`getCellIndex(Ray r, vec3 p, vec3 u, vec3 v, vec3 w)`.
Be precise about how to construct arguments to `getCellIndex()`. **Specifically, please be precise about how you construct and use transformation matrices.**

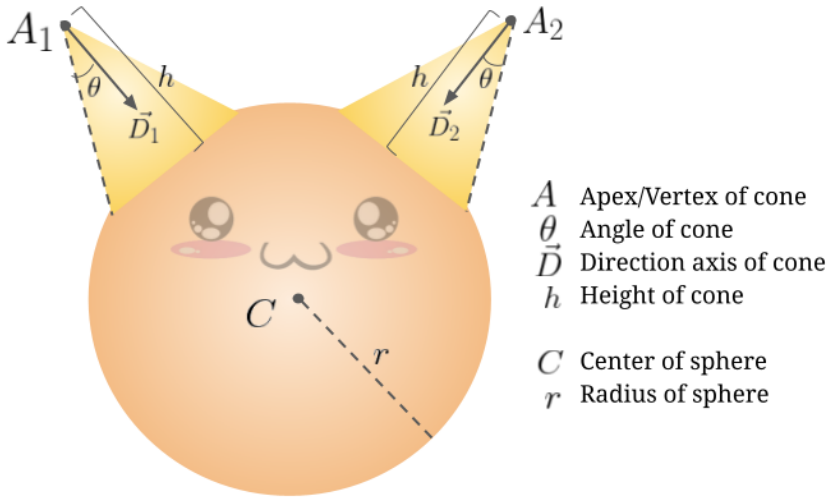
```
vec3 getCellIndex(Ray r, vec3 p, vec3 u, vec3 v, vec3 w) {

// implement your solution here...
```

Solution:

Create a transform:

OPTIONAL PROBLEM 3: Ray-Cat Intersection

One way of modeling a cat head is with a union of a sphere centered at C with radius r and two cones of height h).



Assume you have access to the following:

(1) an `InfiniteCone` struct (a cone with a height of infinity) that stores A, D and θ
(2) a `Sphere` struct that stores sphere center point C and radius r
(3) a `rayInfiniteConeIsect()` function
(4) and a `raySphereIsect()` function

Questions are on the next page...

A. Give an algorithm for finding the closest intersection of a ray with the cat above. Your solution can be described in words, but make sure it's clear what your algorithm is. Be precise how you use `rayInfiniteConeIsect()` to determine ray-cone intersection with a cone of height h.

```
struct InfiniteCone {
  Vec3D A;        // apex position
  Vec3D D;        // direction of axis
  float theta;    // cone angle
};
struct Sphere {
  vec3D C;        // center of sphere
  float r;        // radius
};

// in the functions below t1 is the ''closer hit'': t1 <= t2
void rayInfiniteConeIsect(InfiniteCone b, Ray r, float* t1, float* t2);
void raySphereIsect(Sphere c, Ray r, float* t1, float* t2);
```

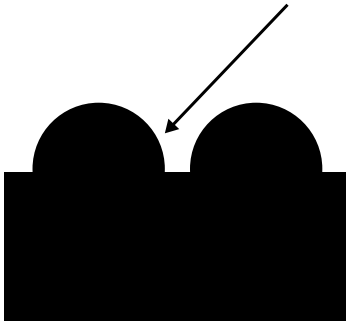
B. Ah, you thought you got off easy without having to solve a ray-primitive intersection problem! Consider a simpler problem where you need to intersect a ray with a cone that has an apex at the origin, is oriented along the z axis (D=(0,0,1)), and has half angle θ. An implicit form of this infinite cone is $x^2 + y^2 = (z \tan \theta)^2$. (Convince yourself this is true!)

Please give an equation for the t value of the intersection point between the infinite cone and a ray with origin $\mathbf{o}$ and ray direction $\mathbf{d}$. You do not need to apply the quadratic formula to solve for t (just give us an equality involving t and components of $\mathbf{o}$ and $\mathbf{d}$).

C. Now imagine you want to compute the intersection of a ray with an infinite cone has apex at point A and is oriented in the direction D (just like in part A). How would you modify the ray's origin and direction to reduce this problem to intersection with the cone from the previous problem that has apex at the origin and is oriented along $(0, 0, 1)$? A description (in words) that involves rotations and translations is fine. But in what direction do you translate, and how do you define the rotation?

OPTIONAL PROBLEM 4: Intersecting Solids

Consider the 2D shape given below, which is made up of the intersection of volumes contained by two circles and a rectangle.

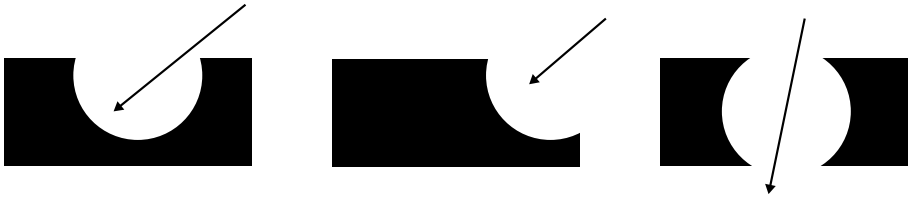


Assume you are given two functions for ray-shape intersection as given below. Both functions return true if there is an intersection (false otherwise), and fill in ray t values for up to two hits. If there is a single intersection point, such as when the ray grazes the shape, just assume that both t values are the same. If there is no intersection, assume the t's are set to a very large number.

```
bool rayBoxIsect(Box b, Ray r, float* t1, float* t2);
bool rayCircleIsect(Circle c, Ray r, float* t1, float* t2);
```

A. Assuming we call the circles c_1 and c_2, and the box b, give an algorithm for finding the closest intersection of a ray with the shape above.

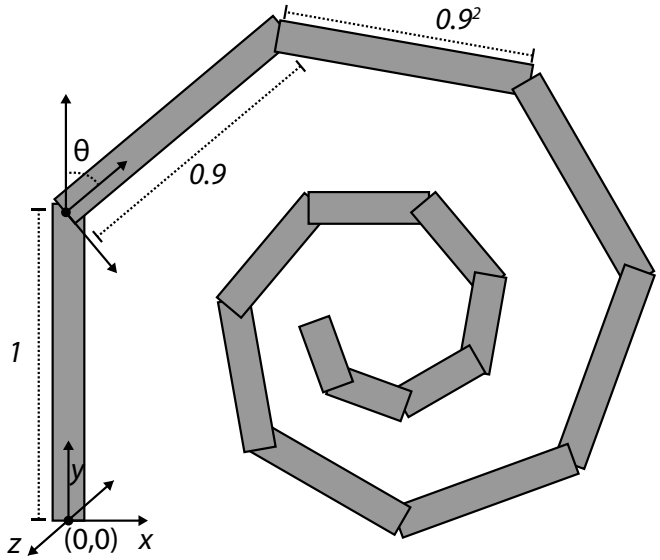
B. Now consider a new kind of shape, which is formed by *subtracting the region inside one circle from the region inside one box. (a few cases are given to help your understanding.)*



Give an algorithm for finding the closest intersection of a ray with this new type of shape. *Note, please make sure you handle the case where the shape is actually two disjoint pieces, and the case where there is no intersection! However, to make things simpler, you can assume the ray origin is outside the area of the box or the circle.*

PRACTICE PROBLEM 5: Ray Tracing A Funky Scene

Consider a scene contain 15 rectangular-boxes as shown below. The scene is formed by starting with a box of length=1.0, with it's center-bottom at the origin, and extending down the Y axis (the width and depth of the box do not matter in this problem). The next box begins at the end of the first box (at point (0,1,0)), but it has a length that is 0.9 times the first box, and rotated with an angle θ relative to the first box. Note that the rotation is about the Z axis, in the *clockwise direction* as viewed when looking down the -Z axis. This pattern of starting the next rectangle at the end of the previous, rotating by θ, and shrinking length by a factor of 0.9 repeats for a total of 15 boxes.



You are given a function `void ray_rect_isect(Ray r, float rect_length)` that computes the first hit of a ray r, with a box aligned with the Y axis of length `rect_length` (see code on next page) For example, the call `ray_rect_isect(r, 1.0)` would compute the intersection of a ray with the first box. Upon return `r.min_t` would reflect the distance to the closest hit. *If the original `r.min_t` of the ray passed into the function is smaller than the t computed from the intersection, then `r.min_t` is unchanged.*

You are also given a function `rotatez` that takes a 3-vector v and rotates it θ degrees about the Z axis in the **counter-clockwise direction.**. Using only these routines, on the next page, please implement the function `isect_funky_scene(Ray& r)`, which will fill in `r.min_t` to contain the closest point on the ray. You should make no assumptions about the origin or direction of the ray in 3D, except that it does not originate from inside any of the boxes.

Hint: be careful about the direction of your rotations. The geometry rotations in the figure are clockwise, and the function `rotatez()` specifies a rotation in the counter-clockwise direction as was typical in class.

```
// students are free to assume that useful constructors, or common ops like
// addition, multiplication on vectors is available..
struct vec3 {
  float x, y, z;
};

struct Ray {
  vec3 o;
  vec3 d;
  float min_t;  // assume this is initialized to INFINITY
};

// rotate counter-clockwise about Z axis (counter clockwise defined when looking down -z)
vec3 rotatez(float theta, vec3 v);

// fills in r.min_t if intersection with rectangular box is closer than current r.min_t
void ray_rect_isect(Ray r, float rect_length);

// assume r.min_t is INFINITY at the start of the call
// result: fill in r.min_t as a result of intersecting r with the 15 segments
void isect_funky_scene(Ray& r) {









}
```