Full Name: _____

Andrew Id: _____

15-462/662, Fall 2016

Midterm Exam

Oct XX, 2016

Instructions:

- This exam is CLOSED BOOK, CLOSED NOTES. If your work gets messy, please clearly indicate your final answer.

Problem	Your Score	Possible Points
1		20
2		20
3		22
4		18
5		20
Total		100

Getting the Party Started: Miscellaneous Short Problems

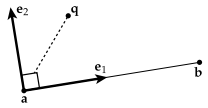
Problem 1. (27 points):

A. (3 pts) In your own words, what is *aliasing?* Give two examples of aliasing that show up either in algorithms or in real life.

B. (4 pts) Do affine transformations map lines to lines? Do they preserve parallel lines? For each question, if yes, explain why, if not, give a counter-example.

Solution: *Yes and yes. To see why, use the parametric equation for a line* $l : \mathbf{p} + t\mathbf{u}$. *We can write the result of the affine transformation on any point as* $\mathbf{A}(\mathbf{p} + t\mathbf{u}) + \mathbf{b}$. *The result is therefore a new line* $l_t : (\mathbf{Ap} + \mathbf{p}) + t\mathbf{Au}$ *(part 1). If two parallel lines* ($\mathbf{u} = k\mathbf{v}$) *are transformed, it is easy to see their transformed directions remain parallel* ($\mathbf{u}_t = k\mathbf{v}_t$).

C. (5 pts) Meshes are made of polygons, which can be split up into triangles, which each have three edges. Suppose we want to compute the distance to a mesh. Get things started by implementing a method that computes the distance (in 2D, *not* 3D) from a query point $\mathbf{q} \in \mathbb{R}^2$ to a segment with endpoints $\mathbf{a}, \mathbf{b} \in \mathbb{R}^2$. You may assume that $\mathbf{a} \neq \mathbf{b}$, and that `Vec2D` has all the usual 2D vector operations (addition, subtraction, normalization, dot product, etc.). *(Hint: compute the components of the vector* $\mathbf{r} := \mathbf{q} - \mathbf{a}$ *in the orthonormal basis* $\mathbf{e}_1, \mathbf{e}_2$ *depicted below. How do each of the two components help you determine the distance to the segment?)*




```
double segmentDist( Vec2D a, Vec2D b, Vector2D q )
{
   Vec2D e1 = ( b-a ).unit();
   Vec2D e2( -e1.y, e1.x );

   double r1 = dot( e1, q-a );
   double r2 = dot( e2, q-a );

   double L = (q-a).norm();

   if( r1 <= 0. ) return (q-a).norm();
   if( r1 >= L  ) return (q-b).norm();

   return abs( r2 );
}
```


D. (4 pts) Now that you can compute the distance to a segment, finding the distance to a triangle should be relatively easy. However, the routine you implemented above always returns a positive value (or at least, it should!), which makes it difficult to determine whether a point is inside or outside a triangle. What's a *tiny* change you can make to your segment routine to make it easy to determine whether a point is inside a triangle? Assuming you've made this change, implement the distance-to-triangle method below. This method should return zero for points inside the triangle.

Solution: *Instead of returning the absolute value of* `r2`*, I would just return its (signed) value. This way, I can just check if a point is inside the triangle by simply checking if all three segment distances are negative. The resulting triangle distance method is implemented below.*

```
double triDist( Vec2D a, Vec2D b, Vector2D c, Vector2D q )
{
   double dab = segmentDist( a, b, q );
   double dbc = segmentDist( b, c, q );
   double dca = segmentDist( c, a, q );

   if( dab <= 0 && dbc <= 0 && dca <= 0 )
   {
      return 0.;
   }

   return min( abs(dab), abs(dbc), ads(dca) );
}
```

E. (4 pts) What is the 2D affine transformation matrix that first translates points by $(2, 1)$, then rotates them by 90 degrees counterclockwise?

Solution:

$$\underbrace{\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{rotate}} \underbrace{\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate}} = \begin{bmatrix} 0 & -1 & -1 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

Return to the Planet of Trees

Problem 2. (20 points):

A. (4 points) In your homework, we asked you to "plant" trees all over a planet by finding *any* rotation that took a tree from one triangle to another. In this case, *which* rotation didn't matter, as long as the new tree still pointed "up" relative to the new triangle. Sometimes, though, the particular choice of rotation *does* matter. Suppose, for instance, that we want to plant sunflowers all over a planet, rather than trees. In this case, we want the flowers to face the "sun," which is specified by some point **p** in space. Assuming that we have a vector that gives us the "up" and "forward" directions for the flower, as well as the normal of the target triangle, explain how you would go about finding a rotation where:

- the new up vector is oriented along the normal, and
- the new forward vector points—*as closely as possible*—toward the target point.

(You do not have to write any equations or code, but you *do* have to give a crystal-clear explanation!)

Solution: I would first construct an orthonormal basis for the new flower, where one of the bases is the normal of the new triangle. For the second basis, I would take the vector from the root of the flower to the target point, and make it orthogonal to the normal via projection. To get the third basis, I would take the cross product of the first two. Finally, to get the rotation I would put write each of the bases as a 3x3 matrix with the bases as column vectors. I would multiply the inverse of the old basis (on the right) with the new basis (on the left). Since these matrices are orthonormal, their composition is a rotation.

B. (3 points) Ok, let's finally plant some trees (or flowers). Suppose you are given a list of vertex positions for the flower (with the root at the origin), a rotation (already computed for you using the method described in the previous part), and the three vertices $\mathbf{q}_1$, $\mathbf{q}_2$, $\mathbf{q}_3$ of the target triangle. Your task is to calculate new vertex positions which "transplant" the tree to the new triangle, replacing the old vertex positions. You should make sure the root ends up at a *reasonable* place (e.g., somewhere inside the triangle). Note that you can get the size of a vector by calling `vector::size()`, and you can access the ith element of a vector `v` via `v[i]`.

```
void transplantTree( vector<Vec3D>& vertices,
                     Matrix3x3 rotation,
                     Vec3D q1, Vec3D q2, Vec3D q3 )
{
   Vec3D c = ( p1 + p2 + p3 ) / 3.; // compute the triangle barycenter

   for( int i = 0; i < vertices.size(); i++ )
   {
      vertices[i] = rotation*vertices[i] + c; // rotate, THEN translate
   }
}
```

C. (4 points) Suppose that you want to ray trace a planet with millions of trees on it, and your tree model contains millions of polygons. Assuming you've already built bounding volume hierarchy (BVH) for the tree, what's a good strategy for doing this kind of "ray-forest intersection?" Describe, at a high level, your algorithm for intersecting a given ray with the scene, assuming that the ray is given to you in world coordinates, and you also want the final hit point in world coordinates (rather than model coordinates). What geometric transformations would you have to apply to which objects, in which order? How do you make it efficient to determine which *tree* was hit (and not just which polygon in the tree was hit)? Assume that you can only compute intersections with *axis-aligned* bounding boxes; also assume you care about both speed *and* storage. *(Hint: a really fast strategy may involve more than one BVH...)*

Solution: To intersect a ray with a particular tree, I would (i) rotate and translate the ray into the coordinate system of the tree, (ii) find the intersection with the tree's BVH, and then (iii) apply the inverse transformation to get the final hit point in world coordinates. To quickly determine *which* tree was hit, I would build a BVH of the bounding boxes of all the transformed trees. Overall, then, I would first intersect with the BVH of trees, then transform, then intersect with the single tree BVH, then invert the transform.

D. (4 points) Ray tracing a planet of trees sounds hard! Can't we just rasterize it instead? Suppose you are given a method `depthCount( Mesh& m )` that tries drawing a mesh and returns the number of fragments that passed the depth test, without actually writing any values to the depth buffer. Using these routines, describe a strategy for reducing the amount of draw that has to happen. Describe how you can further improve performance by drawing the trees in a special order. *(Hint: the bounding box around a mesh can itself be described as a mesh...)*

Solution: Before drawing any tree, I would create a mesh of its bounding box and pass it to `depthCount`. If this method returns zero, I know that everything inside the box will be occluded (so I skip it). To make this scheme particularly efficient, I draw everything in front-to-back order. This way, it is more likely that a far tree will already be occluded by a near tree.

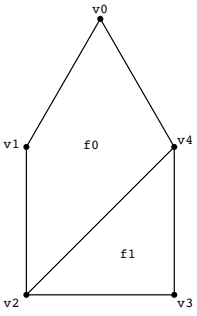
E. (5 points) Suppose we now want to make the leaves of our trees translucent. With ray tracing, dealing with transparency is easy: we just compute all the hits along a ray, and accumulate the values appropriately. What about rasterization? In class, we briefly discussed the *order-independent transparency*: rather than store a single color value per pixel, we temporary store a *list* of color and depth values. Once rasterization is finished, we blend these values together to get the final color values. However, since this scheme is not supported by modern graphics hardware, you will implement it in software below! You may assume that both `samples` and `finalColor` have already been allocated to hold `width*height` elements, and that there is a method `sort(vector<Sample>)` that will sort a list of samples by depth, from *furthest* to *nearest*.

```
struct Sample
{
   double depth;
   double alpha;
   Vec3D color;
};

void computeFinalColor( int width, int height,
                        vector<vector<Sample>> samples,
                        vector<Vec3D> finalColor )
{
   for( int i = 0; i < width; i++ )
   for( int j = 0; j < height; j++ )
   {
      int p = i+j*width;

      sort( samples[p] );

      Vec3D c = samples[0];
      for( int k = 1; k < samples[p].size(); k++ )
      {
         double a = samples[p].alpha;
         c = (1-a)*c + a*samples[p].color;
      }

      finalColor[p] = c;
   }
}
```

Problem 3. (22 points):

After graduating from CMU, you get a terrific-sounding job offer from a hot young tech company in the San Francisco bay area. Unfortunately, as you begin looking at housing options, you discover this offer is actually not so good after all: the average purchase price of a one-bedroom dwelling in San Francisco is $1.2 million, which comes with a monthly mortgage payment of $5,143![1] This situation makes you take a second look at beautiful Pittsburgh, PA, where the sale prices for a *three*-bedroom house average a much more affordable $152k. On the downside, many of these houses are in very poor condition. So, using your mad geometry skills from 15-462/662, you decide to write some software for "flipping" a dilapidated old house in Pittsburgh.

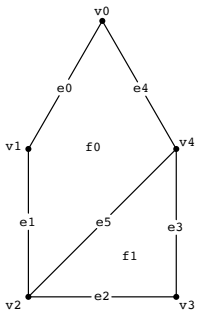
A. (2 points) Your first task is to digitize the blueprints you obtained from the Pittsburgh Department of City Planning. (What a mess!) Just to get some data in the system, you first encode the geometry using the simplest possible data structure: polygon soup. A basic polygon soup data structure is just a list of lists, where each element of the outer list is a polygon, and each element of an inner list is the index of some vertex (for instance, a triangle with vertices 2, 7, and 8 would be $\{2, 7, 8\}$. Encode the figure below using this data structure, giving the vertices of each polygon in counter-clockwise order. *Throughout this entire problem you do* ***not*** *need to worry about the vertex* ***positions*** *(you can assume they are already stored somewhere in memory).*



Solution: $\{\{0, 1, 2, 4\}, \{2, 3, 4\}\}$

[1] Believe it or not, these are all *real prices*, as of October 2015.

B. (4 points) To do any kind of sophisticated repairs on your home, you will need to convert the data to a more sophisticated data structure. Fill out the vertex-edge and edge-face adjacency matrix corresponding to the polygon soup you defined in the previous part, using the edge indices indicated in the figure below.



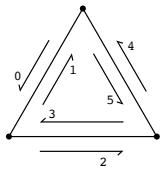
$$A_{VE} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

$$A_{EF} = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

C. (6 points) Using the intuition you built in the previous part, write a method that extracts the vertex-edge adjacency matrix for *any* given polygon soup. You can access the list of lists via a pair of indices (e.g., `soup[i][p]` yields the pth vertex of the ith polygon); the size of any array `a` (whether inner or outer) can be obtained by calling `a.size()`. Throughout, you should assume that *all indices start at zero*. You can also assume that you have a data structure `Set` that has only two methods: `add(i,j)`, which adds an ordered pair to the set, and `contains(i,j)`, which returns true if and only if an ordered pair is included in the set. Finally, you are given a `Matrix` data structure; the entries of a matrix `A` can be set via `A(row,column) = value`. You can assume the matrix will automatically grow to accommodate the largest row/column index.

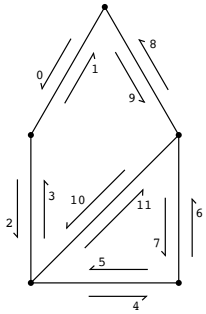
```
void soupToEdgeIncidence( vector<vector<int>> soup, Matrix& VE )
{
   Set edges;
   int nEdges = 0;
   for( int p = 0; p < soup.size(); p++ )
   {
      int degree = soup[p].size();
      for( int v = 0; v < degree; v++ )
      {
         int i = soup[p][v];
         int j = soup[p][(v+1)%degree];

         if( i > j ) swap( i, j );

         if( !edges.contains( i, j ) )
         {
            edges.add( i, j );
            VE(nEdges,i) = 1;
            VE(nEdges,j) = 1;
            nEdges++;
         }
      }
   }
}
```


D. (3 points) In Scotty3D you've been using a sophisticated halfedge data structure that keeps track of vertices, edges, faces, and halfedges. But a much simpler way to encode a halfedge mesh is by just storing the halfedges! In particular, we can just store an array `T` of "twin" indices and an array `N` of "next" indices. For instance, here's what that looks like for a single triangle:




```
int T[6] = { 1, 0, 3, 2, 5, 4 };  // twin
int N[6] = { 2, 5, 4, 1, 0, 3 };  // next
```

We can make this data structure even simpler by always labeling twins in even/odd pairs, as in the figure above: the halfedges of the first edge are 0 and 1, the halfedges of the second edge are 2 and 3, and so on. This way, we can always find the twin by finding the next-largest odd number (for even halfedges) or the next-smallest even number (for odd halfedges). In other words, *the connectivity of a polygon mesh can be encoded purely by an array of "next" indices!* Write the "next array" `N` for the figure below.



k	0	1	2	3	4	5	6	7	8	9	10	11
N[k]												

Solution:

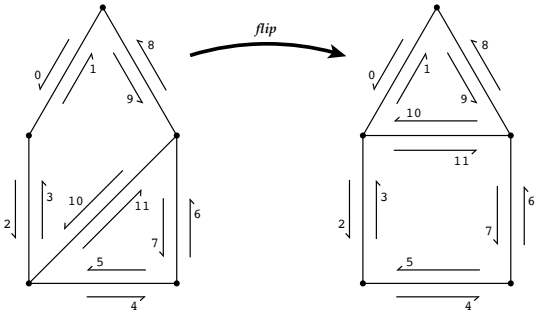
k	0	1	2	3	4	5	6	7	8	9	10	11
N[k]	2	9	4	1	6	11	8	5	0	10	3	7

E. (4 points) Suppose someone hands you a list of integers that is supposed to be the "next" array for a polygon mesh. What are two basic invariants that **must** hold in order for this list to be valid? How can you efficiently check each of these properties?

Solution:

- The number of elements must always be even, which can be checked by simply testing whether the list length modulo two equals zero.
- If n is the list length, then every index between 0 and $n-1$ must appear exactly once. This property can be checked in $O(n \log n)$ time by sorting the list and checking (in linear time) that the ith element is equal to i.

F. (3 points) Ok, finally time to flip this house! You call up Chip and Joanna Gaines and get to work on the roof, which has almost completely collapsed. Your job is to fix it by applying the edge flip depicted below. However, to ensure the list remains valid at all times, the only operation you're allowed to perform is a swap of two elements of the `N` array, which you can write as `swap(N[i],N[j])`. This way, you can't possibly get any of the crashes you probably experienced in your assignment! *(Hint: your solution should not require more than the provided space.)*

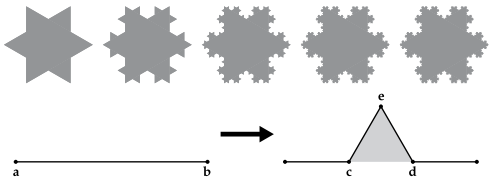


Solution:

```
swap( N[3], N[5] );
swap( N[5], N[10] );
```


Melting Snow

Problem 4. (18 points):



Winter is coming. Fortunately, using the power of computer graphics, you can turn any snow you might encounter into a warm, wet puddle. But first, we have to make the snow! The *Koch snowflake* is a recursive fractal, much like the tree you visualized in Quiz 4. This time, though, rather than repeatedly applying transformations, you will repeatedly split two sides of an equilateral triangle into thirds, placing the base of a new, smaller equilateral triangle along the middle third. Several steps of this process are depicted above. To get a "melted" snowflake, we will then compute the (approximate) distance $\phi(q)$ to this fractal set. By drawing the implicit curve $\phi(q) - c = 0$ for increasing values of c (starting at $c = 0$) we get the animation of a melting snowflake shown below.

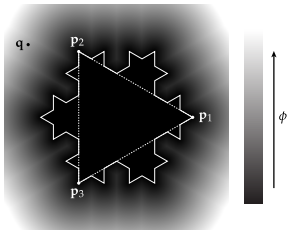
A. (4 pts) Given the endpoints $\mathbf{a}, \mathbf{b} \in \mathbb{R}^2$ of a segment, give expressions for $\mathbf{c}$, $\mathbf{d}$, and $\mathbf{e}$. *These expressions must be valid for **any** given endpoints* $\mathbf{a}, \mathbf{v}$*—not just a horizontal segment as depicted above!* If you like, you may use the imaginary unit $\imath$ to denote a 90-degree counter-clockwise rotation. Alternatively, if you are uncomfortable with complex numbers, you may use Rot_{90} to denote a 90-degree counter-clockwise rotation. *(Hint: what is the height of an equilateral triangle with base length 1? Use the Pythagorean theorem!)*

Note: In the exercises that follow, you may assume that these expressions have been implemented in a method `splitSnowflakeEdge( Vec2D a, Vec2D, Vec2D& c, Vec2D& d, Vec2D& e );` However, *you do **not** have to implement this method yourself!*

Solution: The three points are

$$\begin{aligned} \mathbf{c} &= \mathbf{a} + \tfrac{1}{3}(\mathbf{b} - \mathbf{a}) \\ \mathbf{d} &= \mathbf{a} + \tfrac{2}{3}(\mathbf{b} - \mathbf{a}) \\ \mathbf{e} &= \mathbf{m} + \imath h \mathbf{n}, \end{aligned}$$

where $\mathbf{m} := (\mathbf{a} + \mathbf{b})/2$ is the midpoint of the segment, $h := |\mathbf{d} - \mathbf{c}|\sqrt{3}/2$ is the height of the equilateral triangle, and $\mathbf{n} := \imath(\mathbf{d} - \mathbf{c})$ is the normal vector, orthogonal to the segment from $\mathbf{c}$ to $\mathbf{d}$. The height h of an equilateral triangle with side length ℓ can be derived by dropping a line from the apex of the triangle to the midpoint of its base, forming a pair of right triangles. From the Pythagorean theorem, we then have $h^2 + (\ell/2)^2 = \ell^2$. Solving this equation for h yields $\pm\ell\sqrt{3}/2$, but of course the height is a nonnegative quantity.

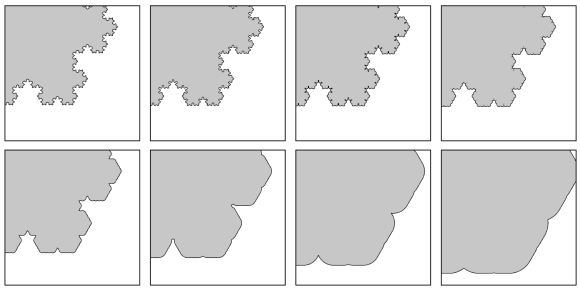


B. (5 pts) Now write a recursive method that, given any query point $\mathbf{q}$, computes the distance to all triangles generated by a single edge of a snowflake (up to a certain depth). The input to this method is a segment, a query point, the target depth, and a pointer to the smallest distance value observed so far. You can assume this method will initially be executed for the three edges $(\mathbf{p}_1, \mathbf{p}_2)$, $(\mathbf{p}_2, \mathbf{p}_3)$, $(\mathbf{p}_3, \mathbf{p}_1)$ depicted above. Your implementation may call the `splitSnowflakeEdge` method from the previous part, and the `triDist` method from the warmup. Points *inside* the snowflake should return a distance value of *zero*; you may assume that all points in triangle $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ will already be assigned a zero distance value.

```
void kochDist( const Vec2D a, const Vec2D b, // input segment
               const Vec2D q,                // query point
               int depth,                    // target depth
               double* distance )            // smallest depth so far
{
   Vec2D c, d, e;
   splitSnowflakeEdge( a, b, c, d, e );

   *distance = min( *distance, triDist( c, d, e, q ) );

   if( depth > 0 )
   {
      kochDist( a, c, q, depth-1, distance );
      kochDist( c, e, q, depth-1, distance );
      kochDist( e, d, q, depth-1, distance );
      kochDist( d, b, q, depth-1, distance );
   }
}
```

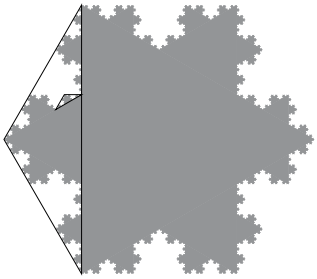

C. (5 pts) The distance function you computed in the previous part is an implicit description: any points with distance value $\phi(q) = 0$ are contained in the snowflake. We can also use this function to compute a "smoothed" version of the snowflake by visualizing sets of points with distance values greater than zero. Rather than drawing a filled region, you will draw a thin curve around a target distance value. You can assume you already have a method `snowflakeDist( Vec2D q )` that returns the distance between a Koch snowflake and the given query point q. The first input to `drawMeltedSnowflake` is the target distance; to draw a curve, you want to turn on any pixel whose distance value equal to this target distance, plus or minus the given range. Pixels in this band should get a value of 0; pixels outside this band should get a value of 1. You can assume that the array `pixels` has already been allocated, and has size `w*h`. (By varying the value of `targetDistance`, you can create a "melting" animation of the kind depicted below.)

```
void drawMeltedSnowflake( double targetDistance, double range,
                          int w, int h, double* pixels )
{
   for( int i = 0; i < w; i++ )
   for( int j = 0; j < h; j++ )
   {
      Vec2D q( i, j );
      double dij = snowflakeDist( q );

      pixels[i+j*w] = 1.

      if( abs(dij-targetDistance) < range )
      {
         pixels[i+j*w] = 0.;
      }
   }
}
```



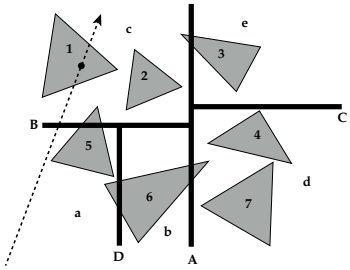
D. (4 pts) Repeatedly evaluating the distance to a fractal structure is quite expensive, especially if you want to produce a lot of detail or many frames of an animation. How might you modify your distance evaluation code to accelerate the process? You do **not** need to write any code. *(Hint: looking at the first figure in this problem, how can you easily get a lower bound on the distance to one "branch" of the snowflake?)*

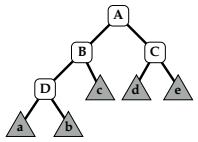
Solution: It is easy to see that all triangles associated with a segment (**a**, **b**) will be contained in the "bounding triangle" (**a**, **b**, **e**) (with **e** defined as above). Therefore, before making a recursive call to `kochDist` we can check if the smallest distance value computed so far is already smaller than the distance to this triangle. If so, we can safely skip the recursive call.



Problem 5. (20 points):

The *k*-d-tree shown below is used to partition the space occupied by seven triangles. The (2D) splitting planes are identified by upper-case letters, and lower-case letters are used to denote the areas of space occupied by each leaf of the *k*-d-tree.





A. (3 pts) Write down the list of triangles associated with each leaf node.

Solution: *a: 5, 6. b: 6. c: 1, 2, 3, 5. d: 4, 6, 7, e: 3*

B. (4 pts) It is obvious to us that the camera ray shot into the scene intersects triangle 1. Nevertheless, a computer program would have to do some work to figure that out. Assuming front-to-back traversal ordering and early termination, list the order in which the nodes of the tree are visited (use the upper and lower case labels shown in the figure on the right).

Solution: *A, B, D, a, b, c*

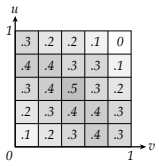
C. (3 pts) Which triangles need to be explicitly checked for intersection against the ray? You can assume that each triangle will be checked at most once.

Solution: *5, 6, 1, 2, 3*

D. (5 pts) Suppose that triangle 1 has vertices $\mathbf{p}_1 = (0,0)$, $\mathbf{p}_2 = (1,0)$ and $\mathbf{p}_3 = (0,2)$, and that a ray shot from *outside* the plane intersects the triangle at $\mathbf{p} = (2/5, 4/5)$. What are its barycentric coordinates?

Solution: *Using cross products to determine the areas of two of the sub-triangles, then dividing by the area of the whole triangle, the barycentric coordinates work out to be 1/5, 2/5 and 2/5.*

E. (5 pts) A different ray now hits the same triangle. Suppose that you repeating the procedure above, and determine that the barycentric coordinates of the new point are 0.3, 0.3, and 0.4. If the texture coordinates of the three triangle vertices are $\mathbf{t}_1 = (0, 0.2)$, $\mathbf{t}_2 = (0.6, 0.8)$, $\mathbf{t}_3 = (1, 0.4)$, what color value will be assigned to the point hit by this ray if we sample from the texture below? You may assume that we are using *nearest neighbor* interpolation.



Solution: *0.5*