Graphics 2009/2010

T2

Endterm exam

Mon, Nov 02, 2009, 08:30–10:30

SKETCHES AND SOLUTIONS

Note that for most problems, there can be more than one correct solution. Also, the following sketches should not be considered as standard solutions but rather as detailed comments including explanations that may go way beyond what was required to achieve the maximum credits for a particular subproblem. In addition, we've been rather generous in the grading if we realized that there have been obvious misunderstandings with the problem description.

No responsibility is taken for the correctness of the provided information.

1

Problem 1: Perspective projection

Subproblem 1.1 [1.5 pt] In order to transform objects in world space coordinates into objects in camera coordinates, we need an orthonormal base $\vec{u}$, $\vec{v}$, $\vec{w}$ with origin at the eye position $\vec{e}$. Explain how we can construct such an orthonormal basis given the eye position $\vec{e}$, the gaze vector $\vec{g}$, and a view-up vector $\vec{t}$. (Note: construct the orthonormal base in a way that we are looking into the negative $\vec{w}$-direction, as we did in the tutorials.)

Solution/comments. *First, we construct the base vector $\vec{w}$. Because we are looking in the negative $\vec{w}$-direction, we can just use the normalized negative gaze vector $\vec{g}$ to construct $\vec{w}$, i.e. we set*

$$\vec{w} = \frac{-\vec{g}}{\|\vec{g}\|}$$

(note: we have to normalize, because we want to create an orthonormal *basis)*

Then, we have to create two unit vectors $\vec{u}$ and $\vec{v}$ that are orthogonal to $\vec{w}$ (and, of course, to each other). We know that the cross product of two nonlinear vectors creates a new vector that is orthogonal to the original ones. Hence, we can use the new vector $\vec{w}$ and the view-up vector $\vec{t}$ (which is nonlinear to $\vec{g}$ and therefore also nonlinear to $\vec{w}$) to create our second base vector $\vec{u}$, i.e. we set

$$\vec{u} = \frac{\vec{t} \times \vec{w}}{\|\vec{t} \times \vec{w}\|}$$

(note again that we have to normalize to make sure to get a unit *vector)*

Finally, we get our third and last base vector $\vec{v}$ by taking the cross product of the first two base vectors $\vec{w}$ and $\vec{u}$, i.e. we set

$$\vec{v} = \frac{\vec{w} \times \vec{u}}{\|\vec{w} \times \vec{u}\|}$$

(Note: if you change the order of the vectors when taking the cross product in the last two equations, you would get a result that is different, but also correct.)

Subproblem 1.2 [0.5 pt] We have seen that we can do perspective projection by matrix multiplication. The precise matrix needed is not relevant here; let's call it M_p. Assume (x, y, z) is a point in $\mathbb{R}^3$. Before the homogeneous divide, we have

$$M_p \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z\frac{n+f}{n} - f \\ \frac{z}{n} \end{pmatrix}$$

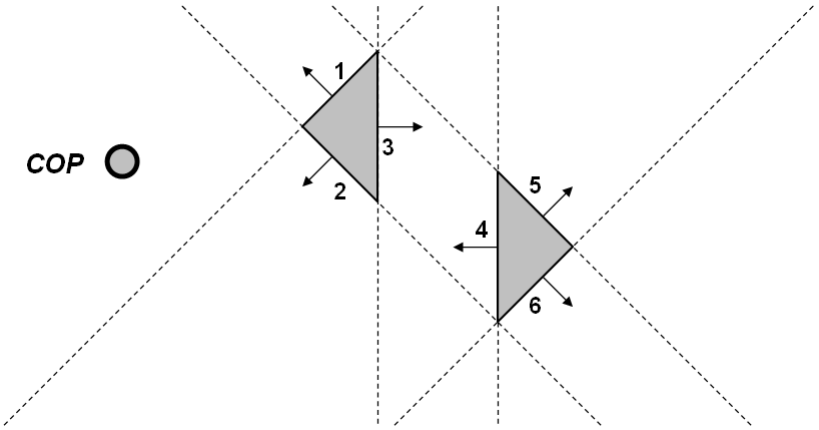
Now assume that we have two random points $\vec{p}_1 = (x_1, y_1, z_1)$ and $\vec{p}_2 = (x_2, y_2, z_2)$ within the view frustum. Show that M_p does not change their order along the z-axis.

Solution/comments. *To prove this, we have to show that the order of the z-coordinates z_1 and z_2 of the points $\vec{p}_1$ and $\vec{p}_2$ after multiplication with M_p is the same as before. By looking at the transformed z-value we see that $z_1\frac{n+f}{n} - f > z_2\frac{n+f}{n} - f$ is obviously true if $\vec{z}_1 > \vec{z}_2$. The same applies for $\vec{z}_1 < \vec{z}_2$ and $\vec{z}_1 = \vec{z}_2$, so the statement is correct.*

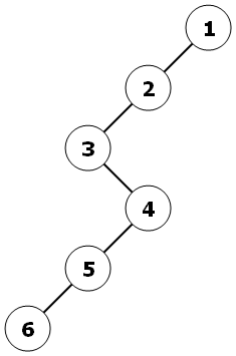
2

Problem 2: Hidden surface elimination

The scene below consists of two triangles that are defined by the line segments 1, 2, 3 and 4, 5, 6, respectively, and a camera view point (i.e. the center of projection *COP*). The arrows indicate the normal vectors of the related segments. The dashed lines are not part of the input, but have been included to illustrate the positions of the objects with respect to each other.



Subproblem 2.1 [1 pt] Create a Binary Space Partitioning tree (BSP tree) for the scene illustrated above. (Note: add the segments in the order $1, 2, \ldots, 6$ to your tree.)



Solution/comments.

Subproblem 2.2 [1 pt] Assume we have a scene that contains only closed polygons that are defined by connected line segments (in 2D) or connected triangles (in 3D). Further assume we have normal vectors for each segment/triangle that point to the outside of the closed polygone. Also, the camera position is not within any of the closed polygones. (Note: the above illustration is a good example for such a scene in 2D) Use this setup to explain what *backface culling* is and how we can easily implement it.

Solution/comments. *If we have a scene with closed polygones and a camera position outside of any of the polygones, we know that we can not see the backside of any of the line segments defining these polygones.*

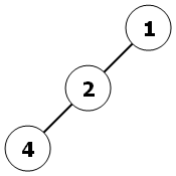
Thus, there is no need to draw segments from which we only see the backside. In backface culling we remove ("cull") such segments ("backfaces") before drawing.

Using the normal vectors, we can easily check if the camera position is on the positive or negative side of the line (or plane in 3D) defined by the related segment and thus identify the backfacing segments.

Hence, we check the camera position against each segment. If the camera is on it's negative side, we remove the segment from our scene. If it's on the positive side, we keep the segment.

Subproblem 2.3 [0.5 pt] Assume you want to get the correct drawing order for the segments in the scene illustrated above by first applying backface culling and then using the BSP tree approach. Create the BSP tree you would get after backface culling (add segments to your tree in the same order as in 2.1).

Solution/comments. *Backface culling removes the faces/segments for which the back is facing the camera. In the scene above, these are segments 3, 5, and 6. The BSP tree that is build with the remaining segments looks like this:*

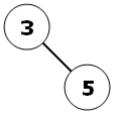


Subproblem 2.4 [0.5 pt] How do the trees created in 2.1 and 2.3 change if the COP is moved to the top-right corner of the scene, i.e. the area defined by the negative side of the lines through 4 and 6 and the positive side of the line through 5?

Solution/comments. *Creating a BSP tree for a general scene is independent from the camera position. Hence, we do not have to create a new tree for 2.1. It stays the same.*

However, the situtaion obviously changes if we apply backface culling first. Because we remove segments based on the camera's position, we end up having a different scene when we modify the camera.

In the concrete example, we would remove segment 1, 2, 4, and 6 with respect to the new camera position. The BSP tree for the remaining segments looks like this:



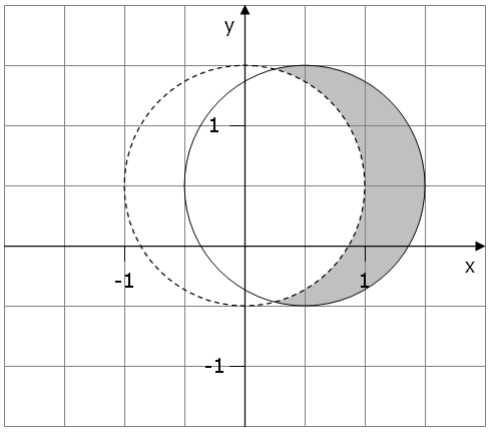
Problem 3: Ray tracing

Subproblem 3.1 [1 pt] Assume $\vec{p}$ is a point in barycentric coordinates, i.e. $\vec{p} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$ with two parameters β, γ and the three vectors $\vec{a}$, $\vec{b}$, $\vec{c}$ that define our barycentric coordinate system. How can we easily check if $\vec{p}$ lies within the triangle defined by $\vec{a}$, $\vec{b}$, $\vec{c}$? (Note: "within" here means including the borders of the triangle.)

Solution/comments. *Because of the definition of barycentric coordinates, $\vec{p}$ lies within the respective triangle if and only if*

$$\beta, \gamma \geq 0 \text{ and } \beta + \gamma \leq 1.$$

Subproblem 3.2 [1 pt] Assume we have two circles C_1 and C_2 in $\mathbb{R}^2$ both with radius 1 centered around $\vec{c}_1 = (0, 0.5)$ and $\vec{c}_2 = (0.5, 0.5)$, respectively (cf. image below).



(a) Explain how we can use these two circles and Constructive Solid Geometry (CSG) to create the image of a waxing moon as illustrated by the gray area in the image above.

(b) Explain how we can calculate the intersection of the ray defined by $y = 0.5$ and the moon image created in (a) with CSG and give the values of the intersection points. (Note: you have to explain how you can get these intersection points. Just writing them down without explanation will not give you any credit!)

Solution/comments. ***(a)*** *CSG creates new (usually complexer) shapes by combining solid (usually simpler) shapes using set operations.*

In this concrete example, when looking at the image, we see that the moon shape matches with the area of circle C_2 that doesn't overlap with circle C_1. Hence, we can use difference (aka exclusion) to create it, i.e. with CSG, the moon image can be expressed as $C_2 - C_1$.

(b) *We get the intersection points of a ray with an object that was created with CSG by calculating the intersection intervals of the ray with the original objects and then applying the same set operations to the resulting intervals that have been used to create the object.*

In the concrete example, we get $I_1 = [(-1, 0.5), (1, 0.5)]$ as intersection interval of the ray with circle C_1 and $I_2 = [(-0.5, 0.5), (1.5, 0.5)]$ as intersection interval of the ray with circle C_2. Applying the set operation that we used to create the moon object to these intervals, i.e. calculating $I_2 - I_1$ gives us $I_{moon} = [(1, 0.5), (1.5, 0.5)]$. The borders of this interval are the intersection points we've been looking for.

Subproblem 3.3 [1.5 pt] Assume we have a 2D image containing many triangles. Explain how we can use BSP trees to separate the space in a way that each of the resulting cells contains only two triangles at the most. (Note: it is required to explain how we split the space and how we create the related BSP tree. It is however *not* required to explain how we traverse this tree when doing ray/object intersection tests and doing so will not give you any extra credit.)

Solution/comments. *To make calculations easier, we generally limit ourselves to only using axis-parallel splitting planes.*

We start by splitting our space in two sub-spaces using a splitting plane that splits the objects into two groups of (more or less) equal size (note: more or less because we can have an odd number of triangles; also, if we use only axis-parallel splitting planes, it might not be possible to find a splitting plane that splits the objects evenly).

This first splitting plane becomes the root of our BSP tree.

If the resulting sub-spaces contain more than two triangles, we go into recursion, i.e. we split a sub-space again in two groups containing (more or less) the same number of objects until all resulting sub-spaces contain at most two triangles.

When doing the recursive splitting, we split along alternating dimensions (i.e. switch between adding horizontal and vertical splitting lines).

Each new splitting line is entered into the BSP tree according to the following rule: we check the new splitting plane against the splitting plane represented by the root. If it is left of it (for horizontal splitting planes) or below it (for vertical splitting planes), we go into the left subtree. Otherwise, we go into the right subtree.

There, we recursively check the new splitting plane against the root of this sub-tree and go left or right depending on the outcome of this check until we reach a leaf. This is where we enter our new splitting plane into the tree.

Problem 4: Texture mapping

Subproblem 4.1 [1.5 pt]

(a) Give a procedure that creates a checker board-like 2D texture consisting of equal sized squares with width = length = π and alternating black and white color.

(b) How do you have to change your procedure from (a) in order to allow users to control the size of the squares with a parameter w?

Solution/comments. ***(a)*** *The following procedure does the trick (it's a simple extension of the "2D strip"-texture we had in the lecture and an easier version of the "3D squared tiles"-texture we had in the tutorials):*

```
stripe( point (x_p, y_p, z_p) ){
   if ( (sin x_p > 0 AND sin y_p > 0) OR (sin x_p < 0 AND sin y_p < 0))
      return black;
   else
      return white;
   }
}
```

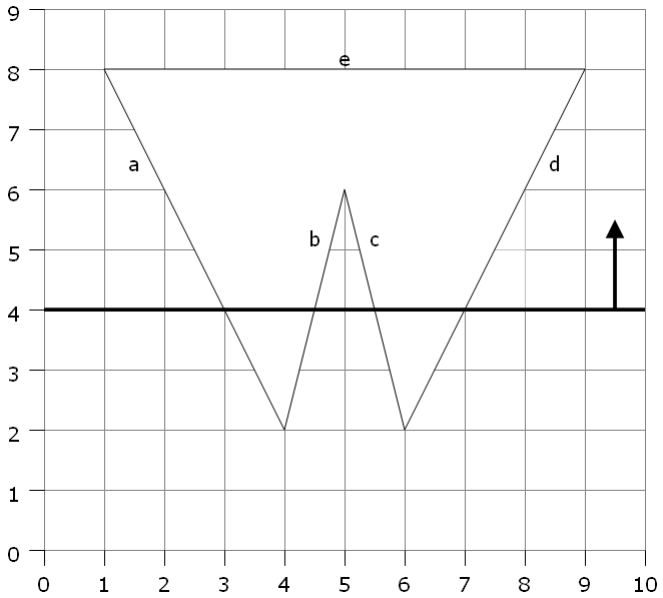
(b) *To get a variable width, we have to modify the* sin *function in a way that its frequence changes from* 2π *to* $2w$*. This is done by replacing* $\sin x_p$ *and* $\sin y_p$ *with* $\sin(\pi x_p/w)$ *and* $\sin(\pi y_p/w)$*, respectively. (Of course, we also have to pass the parameter w to our procedure now in the first line.).*

Subproblem 4.2 [1 pt] Give a short description of environment mapping.

Solution/comments. *The goal of environment mapping is to make objects appear to reflect their surrounding specularly. For this, we first place a cube around the object and project the environment of the object onto the planes of the cube in a preprocessing (texture mapping) step. During rendering, we compute a reflection vector and use that to look up texture values from the cubic texture map.*

Problem 5: Triangle rasterization

Subproblem 5.1 [2 pt] Assume we want to rasterize the polygone with edges a, b, c, d, and e that is illustrated in the image below using the scanline approach (note: we assume a horizontal scanline that moves vertically from the bottom to the top as illustrated by the arrow in the image; also, we assume that the integer coordinates shown in the image represent pixel centers).



For the scanline approach, we use the two data structures *Edge Table* (ET) and *Active Edge Table* (AET).

(a) How and where is edge a, i.e. the line segment between the points $(4,2)$ and $(1,8)$ (cf. image above) represented in the Edge Table? Explain why this is a well-defined representation of this edge/segment. (Note: do not just write down the values but explain their meaning.)

(b) What values are stored in the Active Edge Table at scanline number 4 which is illustrated in the image? Write the values down, explain what they mean, and specify what points we have to rasterize.

Solution/comments. ***(a)*** *The edge starts at y-value 2, so it is stored in the Edge Table at index 2. As it's entry, we store the x-coordinate of the lowest vertex (here: $x_{low} = 4$), the y-coordinate of the highest vertex (here: $y_{high} = 8$) and the Δx-value that indicates the change in the x-coordinate for points on the edge when we move the scanline one step (i.e. for $\Delta y = 1$; here: $\Delta x = -1/2$). Hence, the whole entry becomes*

$$2 : (4, 8, -1/2)$$

This is a well-defined representation of the segment, because it implicitly contains all information to get the two vertices (which have been used as a representation of the segment in the problem description in the first place). The lower vertex is represented by the index of the endge entry ($y_{low} = ET_{index} = 2$) and the first entry of the edge record ($x_{low} = 2$). The $y-$value of the higher vertex is stored in the 2nd entry of the edge table ($y_{high} = 8$). It's $x-$value is implicitly specified by the Δx-value and x_{low} (because we can get it by $x_{high} = x_{low} + (y_{high} - y_{low})\Delta x$). [Note: it was not required to give such a detailed explanation in order to get full credits for this subproblem]

(b) *The Active Edge Table contains the (adapted) edge records from the Edge Table that are intersected by*

the current scanline. From the image we see that these are the segments a, b, c, and d. The concrete values are

$$\textit{scanline number } 4 : (3,8,-1/2),(4.5,6,1/4),(5.5,6,-1/4),(7,8,1/2)$$

The values have the same meaning as in the Edge Table, except for the first one which is the "adapted" x-value, i.e. the x-value of the current intersection point. If we order the entries of the Active Edge Table according to this x-values, we can use them pairwise to identify the pixels we have to rasterize. In the example above these are

(3,4), (4,4), (6,4), and (7,4)

Subproblem 5.2 [1 pt] Assume we have two vertices $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ of a triangle with colors c_1 and c_2. We want to use *Gouraud shading* to color the edge between these two vertices, i.e. we want to linearly interpolate between the two color values. (Note: for simplicity, we assume that a color c_i is not expressed by three values (e.g. RGB) but by a single scalar.) We can easily integrate this into the scanline approach that we are using for rasterization by calculating a Δc-value similarly to the Δx-value needed for rasterization.

Give the concrete formula for calculating this Δc-value and explain how it is used to calculate the colors for each pixel on the edge.

Solution/comments. *When we move the scanline from p_1 to p_2, we are passing $y_2 - y_1$ pixels. While doing this, we want to change the color values from c_1 to c_2. So overall, we want to change it about $c_2 - c_1$. We want to do this linearly, i.e. about the same value in each step. This value is our Δc and it is obviously calculated by*

$$\Delta c = \frac{c_2 - c_1}{y_2 - y_1}$$

Starting with c_1 and continuously adding Δc to the color value from the previous step will give us the desired linear interpolation between c_1 and c_2.

Problem 6: Radiosity and shadows

Subproblem 6.1 [1 pt] Explain the meaning of the following formula, which can be used to calculate the radiosity B_i of a patch A_i:

$$B_i = E_i + \rho_i \sum_j B_j F_{ij}$$

Solution/comments. *The radiosity of a patch is the sum of*

- *the energy E_i emitted by the object itself (e.g. if it is a light source) and*
- *the light that it reflects which in turn is the energy reflected by the other objects and defined by*
  - *the sum of the radiosities emitted by all objects B_j multiplied with the form factor F_{ij} which specifies "how well two objects see each other",*
  - *weighted by the reflective factor ρ_i of patch A_i (which depends on the material of the object's surface and specifies how much light is reflected).*

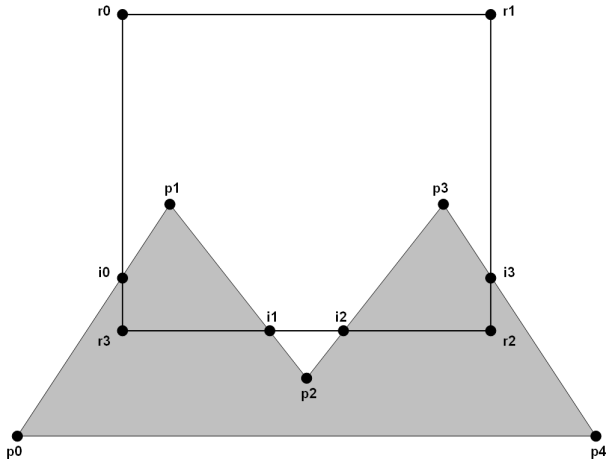
Subproblem 6.2 [1 pt] One approach to integrate shadows into our images is via *shadow volumes*. In practice, they are often implemented using a Stencil buffer which in turn contains a counter for each pixel that we want to rasterize.

Explain what this counter represents and how it is used to decide if we have to draw an object in a shadow or not. (Note: you just have to explain the meaning of the counter and what the related value represents. It is *not* required to explain the Stencil buffer and doing so will not give you any extra credit.)

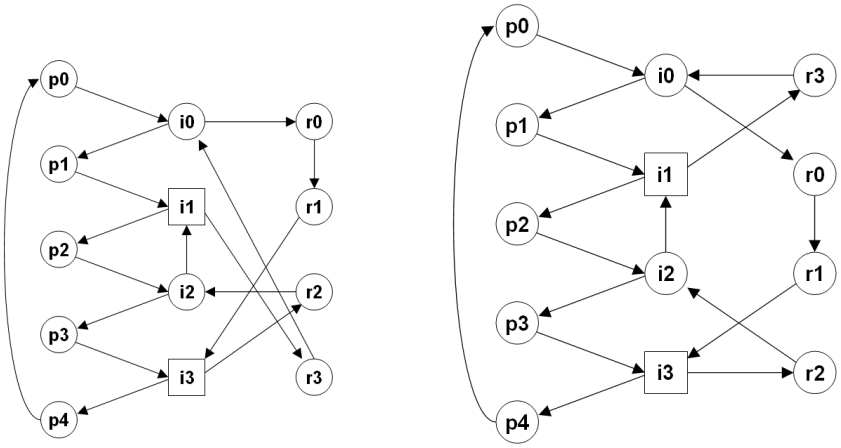
Solution/comments. *Shadows are drawn as volumes that cover the area (or in 3D more precisely the volume) that is in the shadown that is casted by an object. When we look through a pixel towards a scene, we increment the counter everytime we enter one of these shadow volumes and decrement the counter everytime we exit one. We do this until we reach the first object. If the counter is 0, we know that we entered as much shadow volumes as we exited (because for each entry we increase and for each exit we decrease the counter). Hence, we know that the object is not in a shadow volume, so we can draw it in light. If the counter is larger than 0, we know that we entered more shadow volumes as we exited. Hence, we know that the object must be within a shadow volume, so we have to draw a shadow on it.*

Problem 7: Clipping

Subproblem 7.1 [1 pt] We want to use the Weiler-Atherton algorithm to clip the gray polygone illustrated below agains the clipping area represented by the rectangle. Construct the graph that is used by this algorithm for the given rectangle and polygone.



Solution/comments. *Note: intersection-nodes with a square represent outgoing nodes, whereas the ones with circles are incoming nodes. The right graph is the same as the left one but drawn with a clearer arrangement of the nodes. Both are of course correct.*



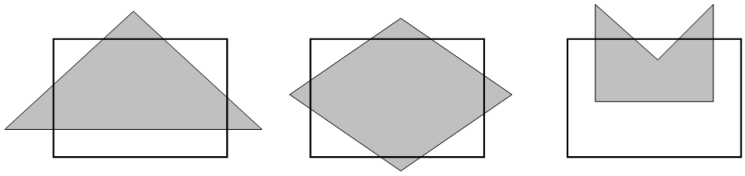
Subproblem 7.2 [1 pt] In the pseudocode for using the graph from the Weiler-Atherton algorithm to draw the clipped polygones, we have the following line:

- *Continue, changing from polygone boundary to clipping region and the other way around at outgoing and incoming intersection vertices, respectively, until we reach the starting vertex.*

When applying this algorithm to the example from the previous subproblem, we only change once, namely from the clipping region to the polygone region. Give an example where we have to do more than one change (note: do this by drawing a rectangular clipping area and an appropriate polygone).

Solution/comments. *In the example above, we start at an outgoing intersection vertex, e.g.* i1 *and walk along the clipping region (reporting each vertex along the way) till we reach the first incoming intersection* i0*. Then we switch to walking along the polygone (again reporting every vertex along the way) till we reach the next intersection vertex. This happens to be our starting vertex* i1 *which means we are done (note that we haven't applied the case mentioned in the problem description). The same situation happens for the second clipped polygone.*

Note: it was not required to explicitly write this down to get full credit. However, it helps finding an example where we actually would need this case (if you didn't remember the one that I gave in the lecture), such as one of the following:



(Basically every polygone that after clipping "touches" at least two different areas of the clipping region will do it.)