CS 130, Final

Solutions

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

Read the entire exam before beginning. **Manage your time carefully.** This exam has 100 points; you need 80 to get full credit. Additional points are extra credit. 100 points → 1.8 min/point. 80 points → 2.25 min/point.

Problem 1 (2 points)

For each set of colors below, what color will be observed if a light of equal intensity of each color is shined on a white sheet of paper? No explanation is required. Write your answer in the table.

#	light colors	observed color
(a)	red ●, green ●	yellow ●
(b)	blue ●, yellow ●	white
(c)	blue ●, green ●	cyan ●
(d)	magenta ●, green ●	white

Problem 2 (3 points)

Fuzzy reflections occur with surfaces that are reflective but rough. How can this effect be achieved with ray tracing?

Cast reflected rays (as for reflective surfaces) but randomly perturb the reflected direction. Multiple rays can be cast for less noisy results.

Problem 3 (3 points)

Concrete sidewalks are generally observed to be very brightly lit by the sun at around noon but only dimly lit when the sun is near the horizon. Why? You may assume that the sun is unobstructed (not shadowed by trees, mountains, or other objects that may extend above the horizon).

This occurs for the same reason as in the Lambertian lighting model. When the sun illuminates the sidewalk at a grazing angle, an incoming cylinder of light rays will be spread over the sidewalk over a much larger area. Since less light strikes a given area of the sidewalk, it appears dimmer.

Problem 4 (2 points)

Which of these features lead to recursion in a ray tracer? No explanation is required, but no partial credit is possible. (1) Texture mapping, (2) transparency shader, (3) Phong shader, (4) antialiasing, (5) reflective shader, (6) bump mapping, (7) area lights.

Only reflective and transparency shaders lead to recursion, since they recursively shade reflected (and possibly also transmitted) rays. Note that shadow rays are not recursive, so there is no recursion associated with lights or Phong shaders. Antialiasing and area lights involve casting more rays, but not recursively. Bump mapping and texture mapping just add image lookups to the shading calculation.

Problem 5 (3 points)

Explain why (a) a flashlight shined on a nearby wall is much brighter than when shined on a far wall but (b) a laser pointer appears about the same brightness in both cases.

(a) A flashlight falls off with distance since the light rays spread out; the same amount of light energy illuminates a larger wall area. (b) Laser pointers do not spread out with distance, so the light does not fall off much. (A very small amount of light energy is lost through interactions with particles in the air.)

Problem 6 (2 points)

During which pipeline stage would texture mapping be implemented? Explain your answer.

This would be implemented in the fragment shader since it must be performed per pixel and the texture is use-specified (even runtime-generated).

Problem 7 (3 points)

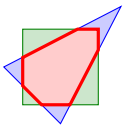
Why do we need to clip before the perspective divide?

Some critical information is lost during the perspective divide. For example, the segment connecting $(2, 0, 0, 1)$ and $(2, 0, 0, -1)$ should be completely discarded (it fails $x < w$). After the perspective divide, these become $(2, 0, 0)$ and $(-2, 0, 0)$, which should be clipped to $(1, 0, 0)$ and $(-1, 0, 0)$, resulting in a segment across the middle of the image.

Problem 8 (3 points)

The portion of a triangle that lies in a square forms a polygon. Our clipping algorithm is essentially triangulating this polygon, but here we care about the polygon itself.
(a) What is the maximum number of edges that this polygon might have in 2D? Justify your answer.
(b) Draw a triangle and a square such that this worst-case bound is achieved.

(a) Seven. In the worst case, a portion of every edge of the square and every edge of the triangle forms an edge of the clipped polygon. This is $4 + 3 = 7$ edges. Every edge of the clipped polygon must be a part of an edge of either the square or the triangle, and no two edges of the clipped polygon may come from the same edge.

(b) 

Problem 9 (3 points)

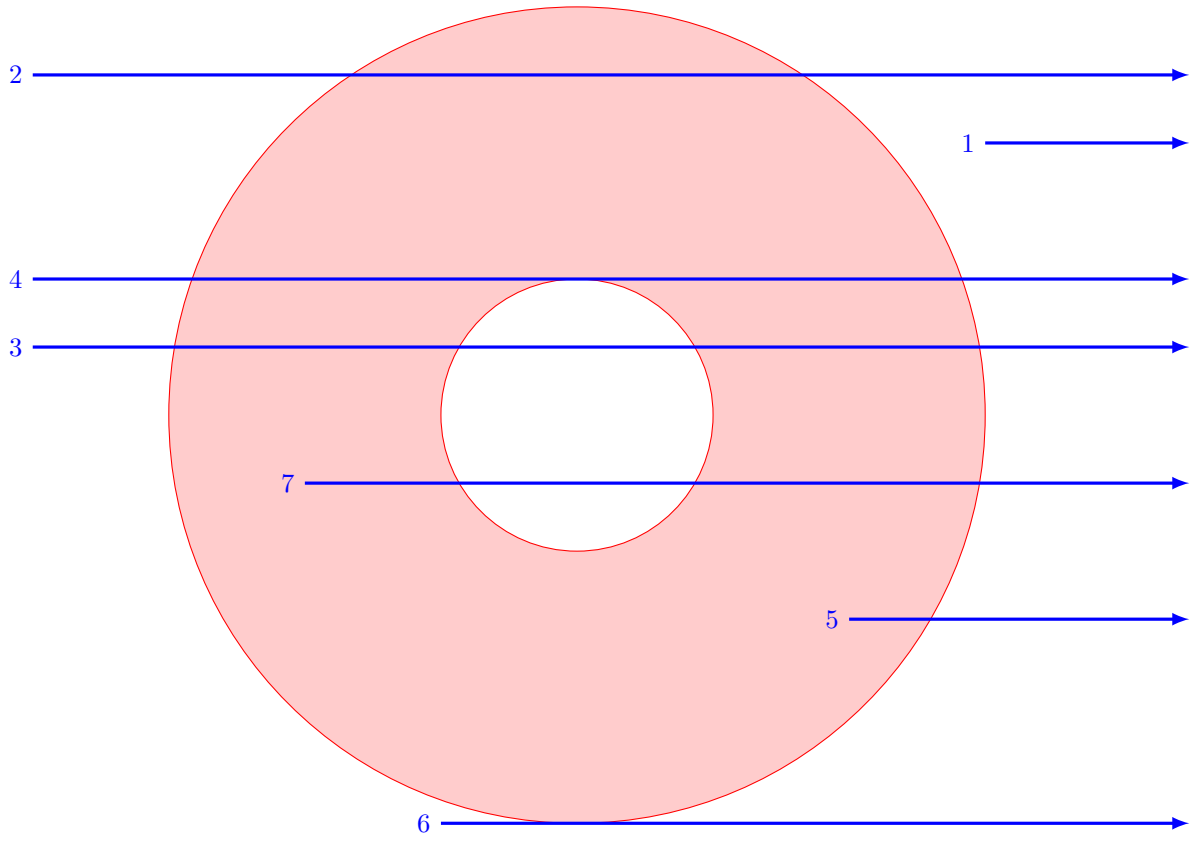
The graphics pipeline contains many stages, including the following: (a) perspective divide (divide by w), (b) geometry shader, (c) vertex shader, (d) rasterization, (e) z-buffering, (f) fragment shader, (g) clipping, and (h) tessellation shader. List these stages in the order that they are performed in the pipeline. If two steps may be performed in either order, you may select an order arbitrarily. No explanation is required.

Order: c, h, b, g, a, d, f, e. Under some circumstances, f and e can be reversed as an optimization.

Problem 10 (7 points)

When raytracing Booleans, we must keep track of a list of intersection distances along the ray. When intersecting with the object below, many different lists are possible. For each of the intersection lists in the table, draw and label (1,2,...) a ray what would produce such an intersection list.

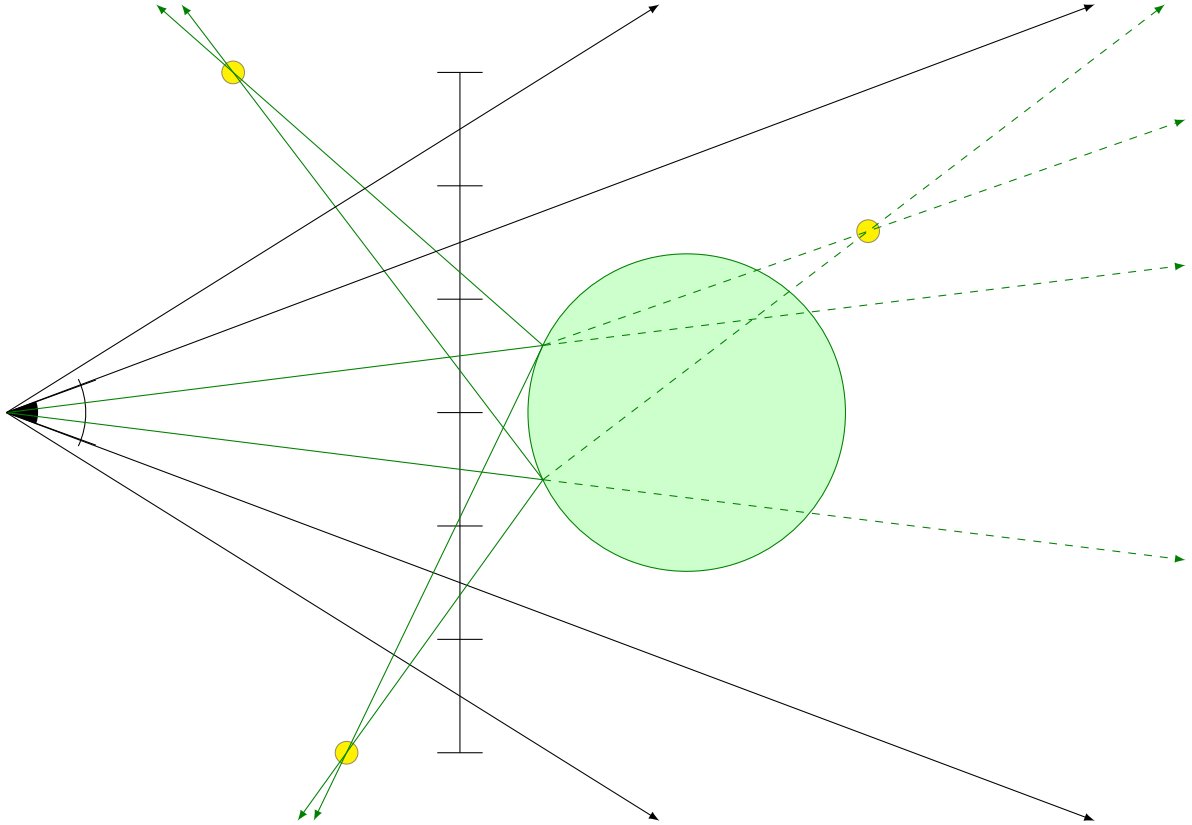
#	list
1	()
2	(a, b)
3	(a, b, c, d)
4	(a, b, b, c)
5	$(0, a)$
6	(a, a)
7	$(0, a, b, c)$



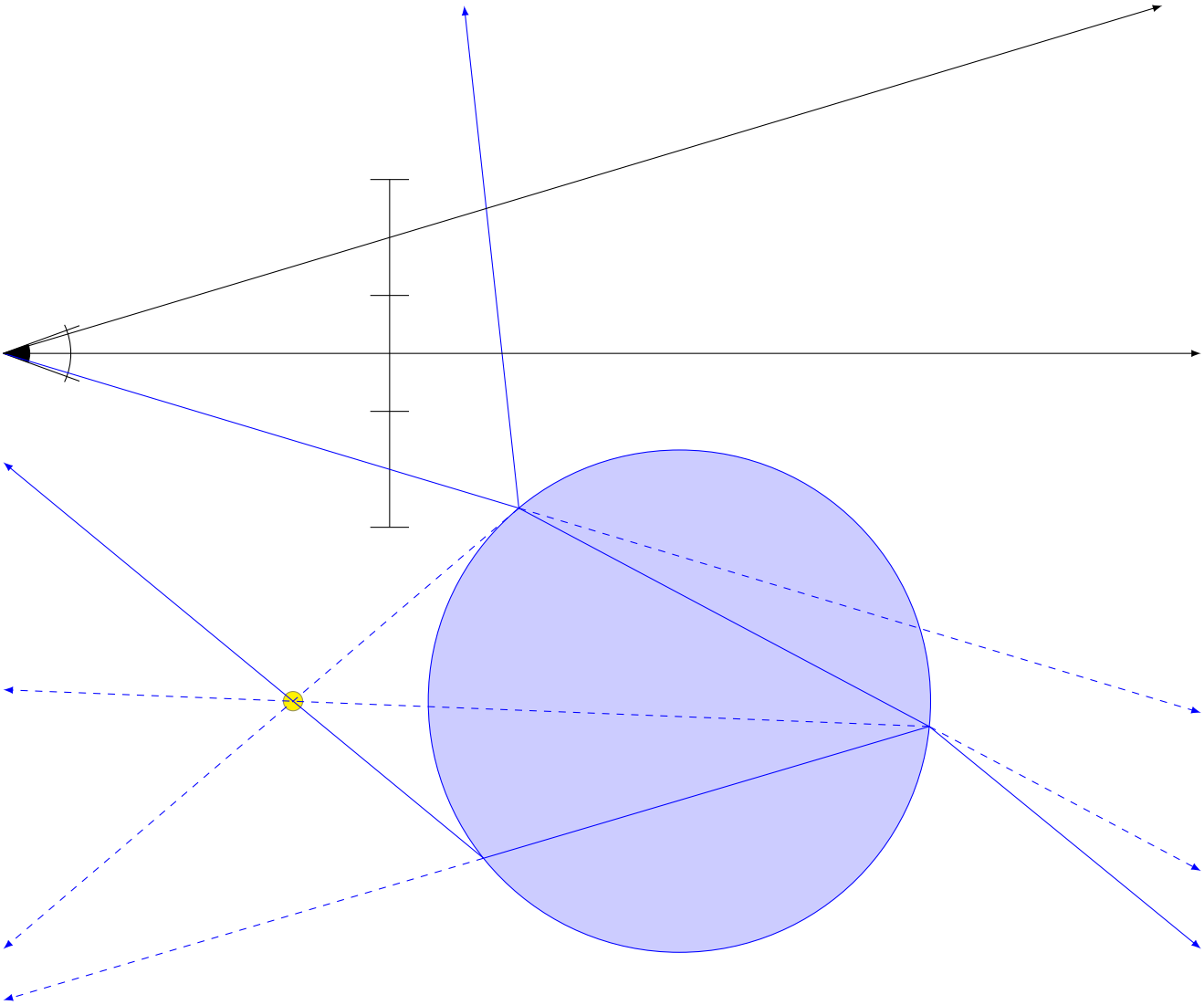
In each of the raytracing problems, green objects are wood, red objects are reflective, and blue objects are transparent. The scenes are in 2D with a 1D image. Each image has three pixels unless otherwise stated. yellow circles are point lights; the ray tracer supports shadows. Draw all of the rays that would be cast while raytracing each scene. Use a maximum recursion depth of 3. (Don't worry about precisely what counts as depth 3 or depth 4; I just care that recursion is being performed correctly when necessary and that important rays are not missing. No problem contains more than 20 rays in the "exact" solution. If you find the need for more rays, you may need to reduce your recursion depth.)

Problem 11 (5 points)

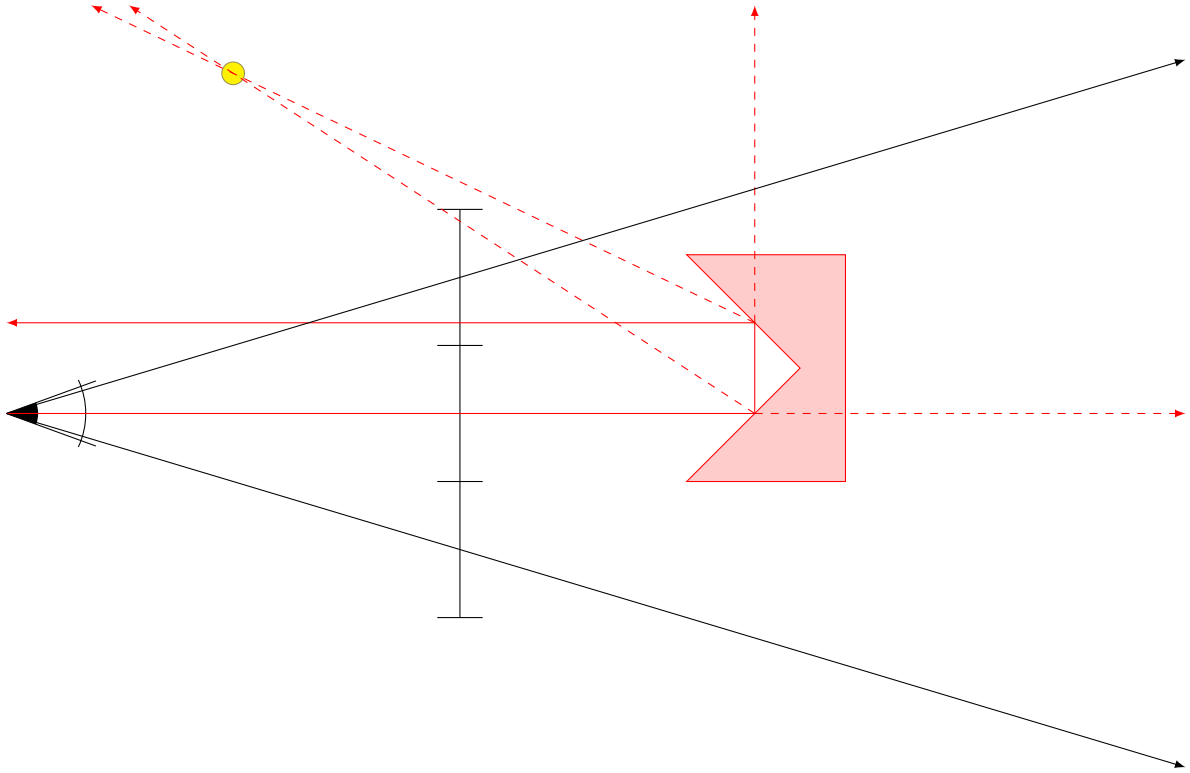
Note that this image has six pixels.



Problem 12 (5 points)

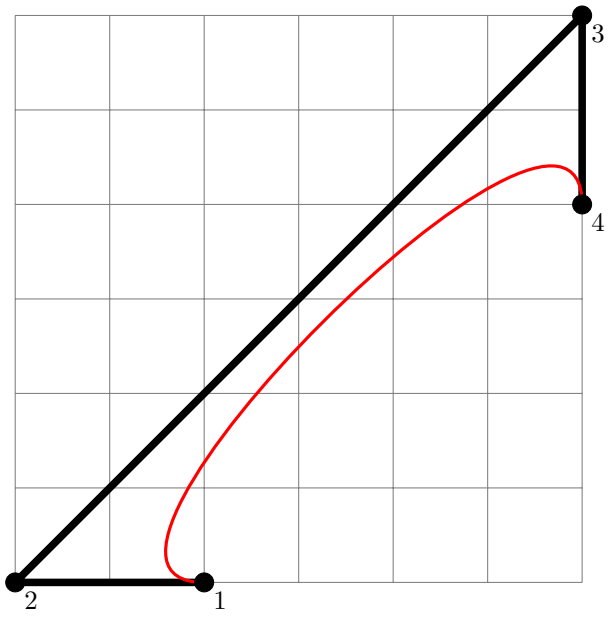


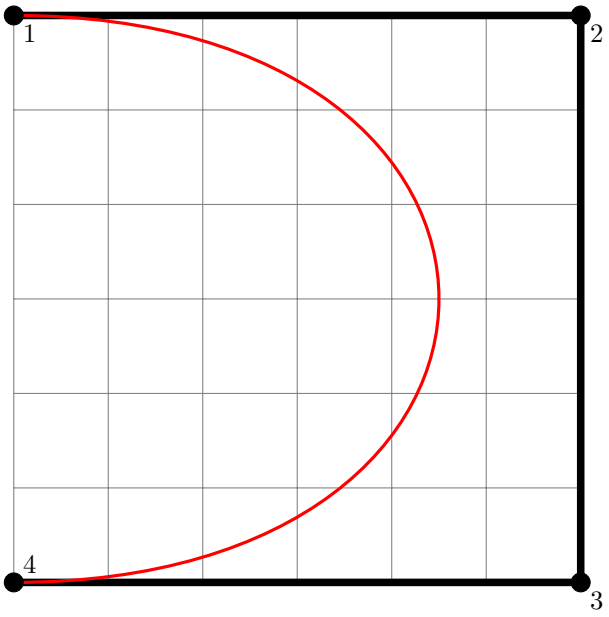
Problem 13 (5 points)

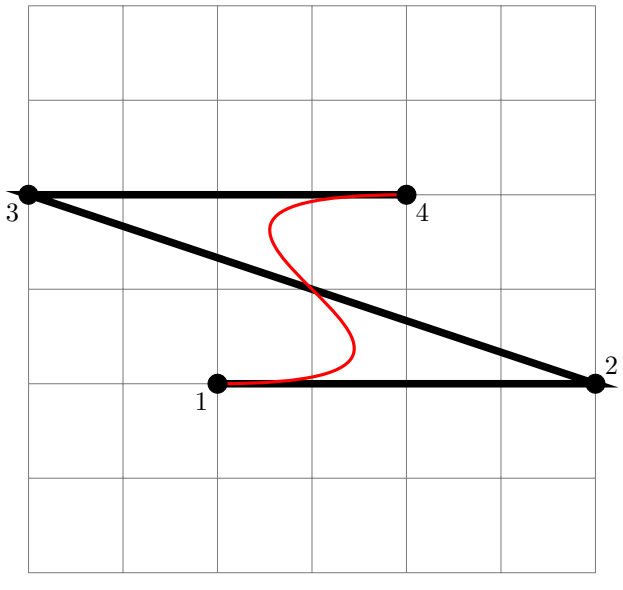


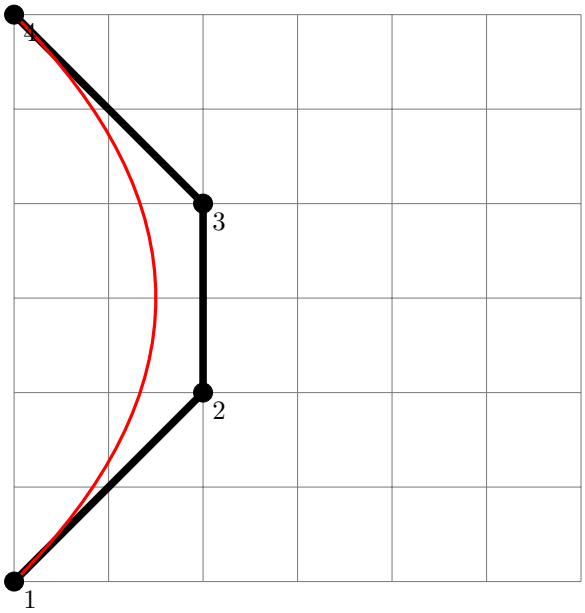
Problem 14 (4 points)

In each of the four examples below, control points are shown for a cubic Bezier curve. Sketch out approximately what these curves will look like.



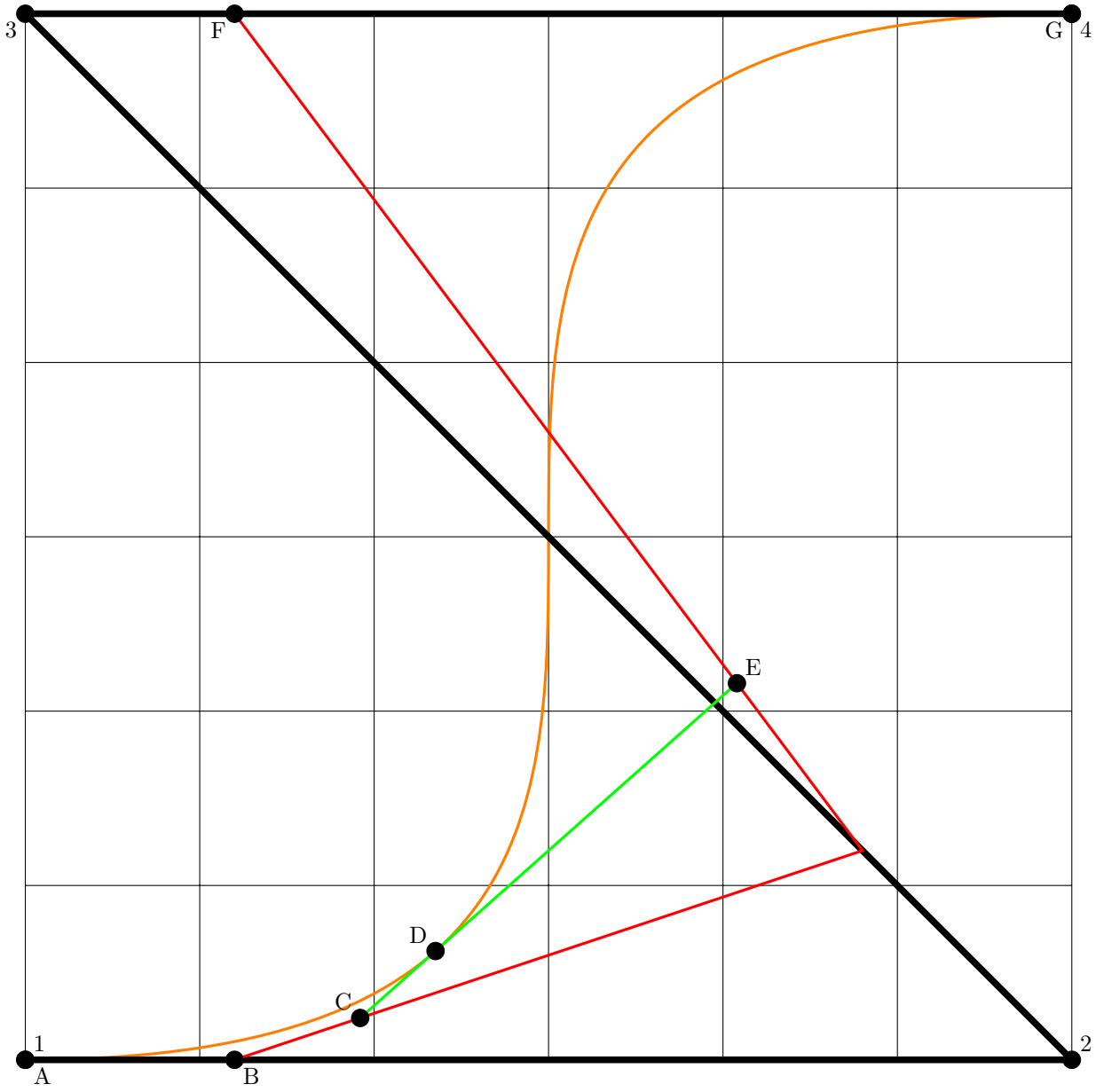






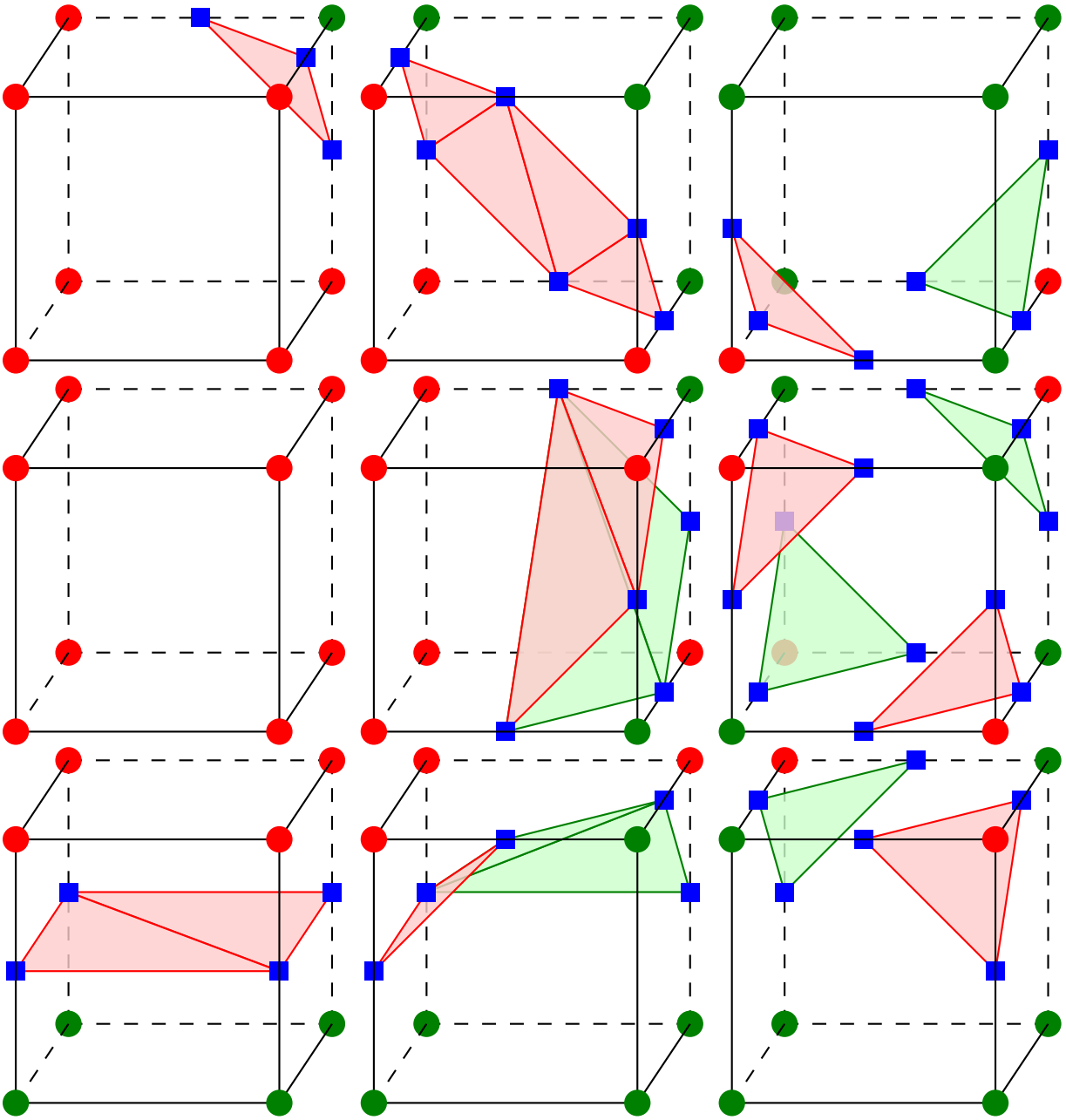
Problem 15 (4 points)

Geometrically subdivide the Bézier curve given by the control points below and at $t = 0.2$ along the curve. Label the new control points A, B, C, ..., G. (The two resulting Bézier curves should share their common point, so only 7 control points are required.)



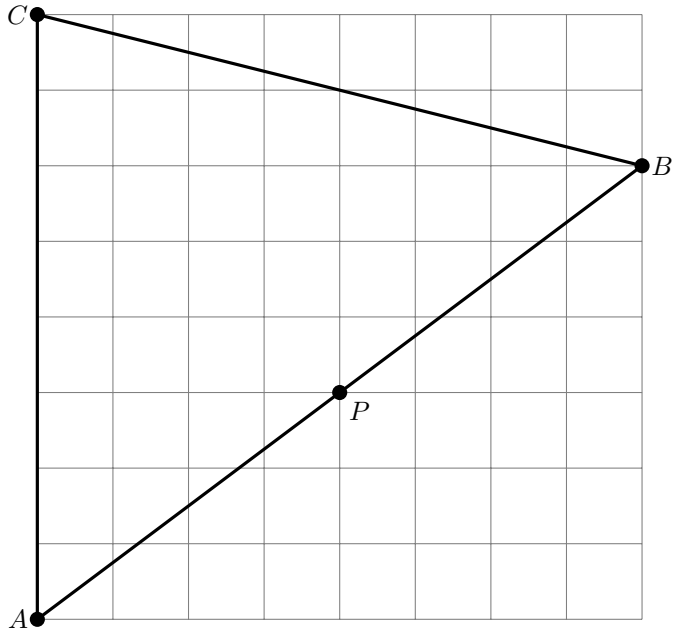
Problem 16 (8 points)

Construct a consistent marching cubes triangulation of the cubes shown below. The cubes should actually be touching, but they have been separated apart for clarity. Draw squares at the vertices of your triangles.



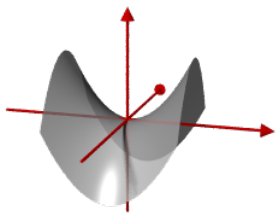
Problem 17 (4 points)

Compute the barycentric weights of the point P.



P is at the midpoint of one of the edges, which makes the barycentric weights easy: $\alpha = \beta = \frac{1}{2}$, $\gamma = 0$.

Problems 18-20 refer to the hyperbolic paraboloid defined by the implicit function $f(x,y,z) = x^2 - y^2 - z$. You may assume that $f(x,y,z) > 0$ corresponds to *outside* of the surface. The surface is illustrated at right. These problems also refer to the *ray* defined by $g(t) = \begin{pmatrix} 2t-1 \\ -t-1 \\ -2t-1 \end{pmatrix}$, where $t \geq 0$. All of these problems are independent (you can solve them in any order, even if you have not solved earlier ones).



Problem 18 (3 points)

What are the direction and endpoint of the ray?

Endpoint is $g(0) = \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}$. Direction is $\begin{pmatrix} \frac{2}{3} \\ -\frac{1}{3} \\ -\frac{2}{3} \end{pmatrix}$. (Don't forget to normalize!)

Problem 19 (3 points)

Compute the closest intersection *location* of the surface with the ray. [Hint: the answer should not contain square roots; if it does not, check your math.]

Plugging $g(t)$ into $f(x,y,z)$ for the cone we get $(2t-1)^2 - (-t-1)^2 - (-2t-1) = 3t^2 - 4t + 1 = (3t-1)(t-1)$. Thus, $t = \frac{1}{3}, 1$. The first is closer, so the closest intersection location is $g(\frac{1}{3}) = (-\frac{1}{3}, -\frac{4}{3}, -\frac{5}{3})$. The farther intersection location is at $g(1) = (1, -2, -3)$.

Problem 20 (4 points)

What is the normal direction at an arbitrary point (x, y, z) lying on the surface? Don't worry about whether the normal points inwards or outwards.

$$f = x^2 - y^2 - z$$
$$\nabla f = \begin{pmatrix} 2x \\ -2y \\ -1 \end{pmatrix}$$
$$\|\nabla f\| = \sqrt{4x^2 + 4y^2 + 1}$$
$$n = \frac{\nabla f}{\|\nabla f\|} = \frac{1}{\sqrt{4x^2 + 4y^2 + 1}} \begin{pmatrix} 2x \\ -2y \\ -1 \end{pmatrix}$$

An alternative strategy is to parameterize the surface, such as with (x, y) and $\mathbf{w} = (x, y, z) = (x, y, x^2 - y^2)$.

$$\mathbf{w} = \begin{pmatrix} x \\ y \\ x^2 - y^2 \end{pmatrix} \qquad \mathbf{w}_x = \begin{pmatrix} 1 \\ 0 \\ 2x \end{pmatrix} \qquad \mathbf{w}_y = \begin{pmatrix} 0 \\ 1 \\ -2y \end{pmatrix} \qquad \mathbf{w}_x \times \mathbf{w}_y = \begin{pmatrix} (0)(-2y) - (2x)(1) \\ (2x)(0) - (1)(-2y) \\ (1)(1) - (0)(0) \end{pmatrix} = \begin{pmatrix} -2x \\ 2y \\ 1 \end{pmatrix}$$

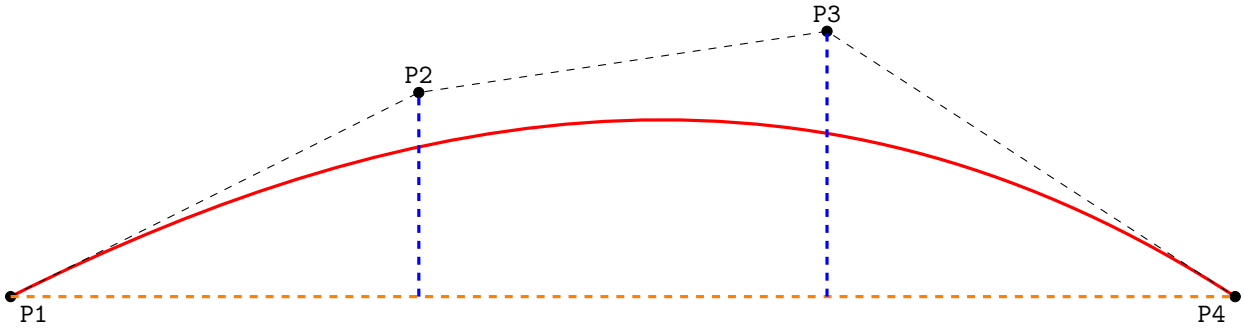
This agrees, up to sign, with what we obtained the original way.

Problem 21 (4 points)

Write a routine that rasterizes a filled circle. That is, it draws pixels that lie on or inside the circle but not pixels that lie outside it. Your routine should be written in C++-like syntax and use the signature `void fill_circle(int x, int y, int r);`, where (x, y) and r are the circle's center and radius. You may call the routine `void draw(int x,int y);` to set the pixel (x, y). You may assume that the circle lies entirely inside the image.

```
void fill_circle(int x, int y, int r)
{
  for(int i=-r;i<=r;i++)
    for(int j=-r;j<=r;j++)
      if(i*i+j*j<=r*r)
        draw(x+i,y+j);
}
```

In **Problems 22-27**, we will write a more robust routine to render Bezier curves based on curve subdivision. These problems will begin with the skeleton code below and will focus in on filling in the missing details. In some of the questions, you will be asked to write bits of code. You may freely mix math notation ($\|\mathbf{u}+\mathbf{v}\|$) and code notation (`(u + v).magnitude()`) as you please. All of these problems are independent. You may solve them in any order, even if you have not solved previous problems. Problems will also refer to the figure below.




```
// This routine is defined elsewhere; it probably
// just sends the segment to OpenGL to be drawn in hardware.
void rasterize_segment(vec2 A, vec2 B);

// Distance from point P to line segment AB.
// We will define this later.
double dist_to_segment(vec2 P, vec2 A, vec2 B);

void rasterize_bezier(vec2 P1, vec2 P2, vec2 P3, vec2 P4)
{
  double tol = 0.5; // half of a pixel
  bool b1 = dist_to_segment(P2, P1, P4) <= tol;
  bool b2 = dist_to_segment(P3, P1, P4) <= tol;

  // If the Bezier curve deviates from the line segment
  // by no more than half a pixel, draw the line segment.
  if(condition)
  {
    rasterize_segment(endpoint1, endpoint2);
    return;
  }

  // Otherwise, we need to subdivide the Bezier curve.
  vec2 A1, A2, A3, A4, B1, B2, B3, B4;

  // Compute subdivision here.

  rasterize_bezier(A1, A2, A3, A4);
  rasterize_bezier(B1, B2, B3, B4);
}
```

Problem 22 (3 points)

When rendering the Bezier curve, we want to ensure that no portion of the curve lies more than half a pixel from the line segments that we actually draw on the screen. Rather than checking the Bezier curve itself to see if any point on the curve exceeds this distance, we use a simpler procedure where we check `P2` and `P3` instead. How would you describe this simpler approach? **You must justify your answer.** (E1) Heuristic; the curve will sometimes exceed our distance estimate, but not generally by much. (E2) Approximate; the curve generally stays within these bounds, but there are pathological cases where they may be exceeded. (E3) Exact; the distance computed by the simplified approach is the same as would be computed from the Bezier curve itself. (E4) Upper bound; the distance computed by the simplified approach is never less than would be computed from the Bezier curve itself, but it might be significantly larger.

E4. The convex hull property of Bezier curves (the curve lies within the convex hull of its control points) ensures that at least one control point is as far or farther from the line segment than any point on the Bezier curve. In fact, it will always be strictly farther, except when the control points are colinear (in which case the Bezier curve is just a line).

Problem 23 (4 points)

Implement the function: double dist_to_segment(vec2 P, vec2 A, vec2 B);. This function computes and returns the distance from point P to line segment AB. This corresponds to the lengths of the dashed blue lines in the figure.

There are many ways to do this. I will write the vector AP as the sum of a vector along AB and a vector orthogonal to AB (the blue part). The distance is just the length of the orthogonal vector.

$$\mathbf{u} = P - A \qquad \mathbf{v} = B - A \qquad \mathbf{u} = t\mathbf{v} + \mathbf{w} \qquad \mathbf{v} \cdot \mathbf{w} = 0$$

$$\mathbf{v} \cdot \mathbf{u} = t\mathbf{v} \cdot \mathbf{v} + \mathbf{v} \cdot \mathbf{w} = t\mathbf{v} \cdot \mathbf{v} \qquad \Longrightarrow \qquad t = \frac{\mathbf{v} \cdot \mathbf{u}}{\mathbf{v} \cdot \mathbf{v}}$$

$$\mathbf{w} = \mathbf{u} - t\mathbf{v} = \mathbf{u} - \frac{\mathbf{v} \cdot \mathbf{u}}{\mathbf{v} \cdot \mathbf{v}}\mathbf{v} \qquad \text{dist} = \|\mathbf{w}\|$$

```
double dist_to_segment(vec2 P, vec2 A, vec2 B);
{
  vec2 u = P - A, v = B - A;
  double t = dot(v, u) / dot(v, v);
  vec2 w = u - t * v;
  return w.magnitude();
}
```

Problem 24 (3 points)

What code should replace `condition` in the code skeleton?

b1 && b2

Problem 25 (3 points)

what code should replace `endpoint1` and `endpoint2` in the code skeleton?

`p1` and `p4`.

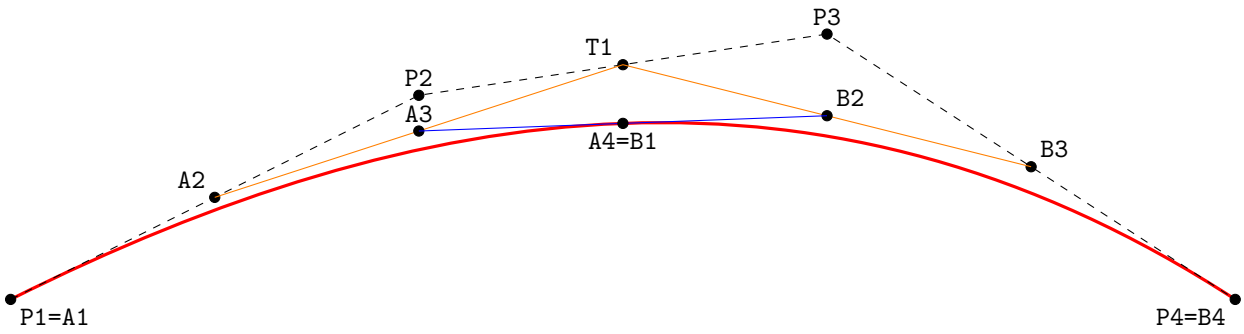
Problem 26 (3 points)

Our dist_to_segment routine cannot be computed using only basic arithmetic (`+, -, *, /`). However, it turns out that the distance d can be written in the form $d = \sqrt{\frac{r}{s}}$, where r and s can be computed using only cheap operations (`+, -, *`). [Hint: if your routine did not use square roots or division, it is likely incorrect. Note that routines like u.magnitude() and u.normalized() calculate square roots internally, and the latter also uses a division.] Show that our tolerance check ($d \le \text{tol}$) can be performed without any using any divisions or square roots. You may use r and s directly, and you may assume that $r \ge 0$ and $s > 0$. Avoiding square roots is half credit, and avoiding both is full credit.

$$\sqrt{\frac{r}{s}} \le \text{tol}$$
$$\frac{r}{s} \le (\text{tol})^2$$
$$r \le s(\text{tol})^2 \qquad \text{Assumes } s > 0$$
r<= tol * tol * s

Problem 27 (4 points)

Write code to perform the Bezier curve subdivision. Subdivide at $t = \frac{1}{2}$. Compute the control points `A1, A2, A3, A4, B1, B2, B3, B4` for the new curves.




```
A1 = P1;
B4 = P4;
A2 = 0.5 * (P1 + P2);
T1 = 0.5 * (P2 + P3);
B3 = 0.5 * (P3 + P4);
A3 = 0.5 * (A2 + T1);
B2 = 0.5 * (T1 + B3);
A4 = B1 = 0.5 * (A3 + B2);
```