2IV60 / 2IV10 Computer Graphics

Examination, January 27 2014, 9:00 – 12:00

This examination consist of **four** questions with in total 16 subquestion. Each subquestion weighs equally. In all cases: **EXPLAIN YOUR ANSWER. Use sketches where needed to clarify your answer. Read first all questions completely.** If an algorithm is asked, then a description in steps or pseudo-code is expected, which is clear enough to be easily transferred to real code. Aim at compactness and clarity. Use additional functions and procedures if desired. Give a short description of input and output for each function and procedure. The use of the book, copies of slides, notes and other material is not allowed.

1 We consider some basic concepts of computer graphics.

a) What are the parameters of a spotlight, and how can they be used to determine if a point Q is illuminated by the spotlight?

A spotlight can be defined by a position P, a light direction V, and an opening angle φ. A point Q is illuminated if it is inside the cone with P as top, direction V, and top angle φ. Specifically, if

$V \bullet (Q - P) / | Q - P | < \cos(\varphi/2)$

Q is illuminated.

b) Give an example where constant-intensity rendering (or flat surface shading) can be used without reduction of the image quality.

If the object to be displayed consists of flat surfaces, and variation of the shading is low (because only diffuse reflection is used, or because viewer and light source are far away), constant-intensity rendering gives good results.

And, also if polygons are small (say, smaller than pixels), constant-intensity rendering can be used.

c) Describe the depth-buffer algorithm.

The version given in the slides:

```
var zbuf: array[N,N] of real;          { z-buffer: 0=near, 1=far }
    fbuf: array[N,N] of color;         { frame-buffer }
For all 1<= i, j <=N do
   zbuf[i,j] := 1.0; col[i,j] := BackgroundColour;
For all polygons do                    { scan conversion }
    For all covered pixels (i,j) do
        Calculate depth z;
        If z < zbuf[i,j] then { closer! }
            zbuf[i,j] := z;
            fbuf[i,j] := surfacecolor(i,j);
```

A more verbal version, mentioning the key-points, can be as follows.

The depth-buffer algorithm is a hidden-surface algorithm. Central is the depth-buffer: a 2D array that stores for all pixels of the image the closest point found so far. Initially, it is set to infinity (or 1, assuming a perspective transformation has been done). Next, all polygons are scan-converted (in arbitrary order). For all pixels covered the depth is calculated. This depth is compared with the

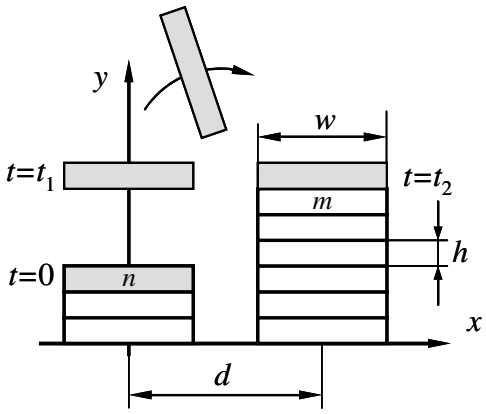
depth stored in the depth-buffer. If the new point is closer, its z-value is stored in the depth-buffer, and its color is stored in the frame-buffer.

d) In 2D texture mapping typically two parameters s and t are specified per vertex. What do they mean?

The parameters s and t denote which point of the texture has to be displayed at the vertex. More specific, the point (s, t) in texture coordinates is mapped to the vertex.

---

2 The figure shows two stacks of coins, viewed from the side. The left stack has n coins (here 3), the right stack m (here 6). It may be assumed that $n < m$. Each coin has thickness h and diameter w; the left stack is centered around the y-axis; the right stack is centered around a line $x=d$. We want to move the (grey) coin on top of the left stack to the top of the right stack via a smooth animation: from $t=0$ to $t = t_1$ the coin is moved up; and from $t = t_1$ to $t = t_2$ the coin is rotated around a point over 180 degrees.



We use a 3×3 time-dependent homogenous transformation matrix $M(t)$ to describe this 2D animation. A position Y in global coordinates is related to a position X in local coordinates via $Y=M(t)X$ for time t. It may be assumed that $T(x,y)$ gives a translation matrix along the vector (x,y); that $R(\varphi)$ gives a rotation matrix of φ degrees around the origin; and that $S(a,b)$ gives a scaling matrix with scale factors a and b in x and y-direction respectively.

To draw the coin for a frame with time t of the animation, we first set $M(t)$ and next we call a routine *DrawSquare*(). This routine draws a unit square with, in local coordinates, its lower-left corner in the origin (0, 0) and its upper-right corner at (1,1). Answer the following questions. Matrices do not have to be given element-wise, results from earlier subquestions may be used.

a) Give an expression for $M(0)$ (start position).
We have to scale and to translate the rectangle. For scaling we use $S(w, h)$ to get the right size. Also, we have to move it to the left to center it around the y-axis, and also, it has to move up $(n-1)h$ units. For this we use $T(-w/2, (n-1)h)$. These transformations are in global coordinates, hence we multiply from the left, which gives

$M(0) = T(-w/2, (n-1)h)\ S(w, h)$.

Alternatively, we can translate first and scale next. In this case we get

$M(0) = S(w, h)\ T(-1/2, n-1)$.

The unit rectangle is translated here, hence a translation over $(-1/2, n-1)$ is used.

b) Give an expression for $M(t)$ for $0 < t \leq t_1$ (vertical motion).
The rectangle has to be positioned first at the start position. We can conveniently use $M(0)$, calculated in the previous subquestion, for this. In the given time interval the rectangle has to move up $(m + 1 - n)h$ in total, in the configuration shown $4h$. The position at a time t can be obtained by linear interpolation, such that when $t = t_1$ the highest position is reached. Hence, the vertical distance translated at time t is equal to $(t / t_1)(m + 1 - n)h$, as can easily be verified by substituting $t = 0$ and $t = t_1$. The division of time t by t_1 gives a standard parameter in the unit interval. Multiplying from the left again (using global transformations) gives

$M(t) = T(0, (t / t_1)(m + 1 - n)h)\ M(0)$.

c) Give an expression for $M(t)$ for $t_1 < t \leq t_2$ (rotation). What center of rotation P to use?
To position the rectangle for the start of this interval, we use $M(t_1)$ from the previous subquestion. The rectangle must be rotated around a point P. This point is located in between the two stacks, and has as height the center of the coin m+1 on top of the right stack. Hence, $P = (d/2, (m+1/2)\text{h})$.

For the rotation we use the standard pattern. First, make P the origin by applying $T(-P)$, rotate over an angle $\alpha(t)$ via $R(\alpha(t))$, finally translate back with $T(P)$. Here $\alpha(t)$ is some linear function of t, such that $\alpha(t_1) = 0$ and $\alpha(t_2) = -\pi$: at the start, no rotation; at the end, a clockwise rotation over 180 degrees has been done. If we set $\alpha(t) = At + B$ and substitute this in the constraints for start and end, we can solve for A and B and thereby fix $\alpha(t)$. Alternatively, we can also define $\alpha(t) = -\pi\, u(t)$, where $u(t)$ ranges from 0 to 1 for t to t_1 to t_2. We can get $u(t)$ by subtracting t_1 from t (to make t_1 the origin) and dividing by $t_2 - t_1$ (to get a unit length interval). Combining this gives:

$M(t) = T(P)\, R(-\pi\,(t - t_1)/(t_2 - t_1))\, T(-P)\, M(t_1)$, with $P = (d/2, (m+1/2)\text{h})$.

d) We want the center of the coin to have a constant velocity v. How to set t_1 and t_2 to achieve this?
Here we use that velocity can be defined as $v = \Delta s / \Delta t$. The distance Δs moved in the first time interval is equal to $(m + 1 - n)h$ (see question b), the time interval Δt has length t_1. Substitution and rearranging gives

$t_1 = (m + 1 - n)\, h / v$.

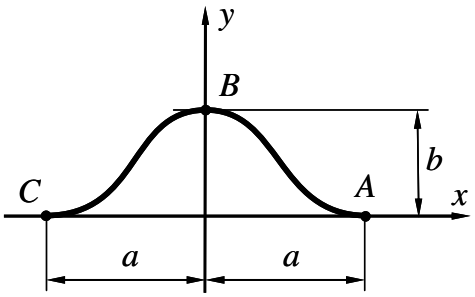
We see that thicker coins, a bigger difference in the height of the stacks, and a lower velocity all give (as you may expect) a longer animation. For the second interval, $\Delta s = \pi d/2$ and $\Delta t = t_2 - t_1$. Subsitution and rearranging gives

$t_2 = t_1 + \pi d / (2v)$.

Here the length of the second interval is determined only by the distance d between the stacks and, again, the given velocity v.

---

3 We want to draw a curve in the shape of a bump, as shown in the figure. The bump has width $2a$ and height b, and must be symmetric around the y-axis. At the points $A = (a, 0)$, $B = (0, b)$, and $C = (-a, 0)$ the tangent line to the curve is parallel with the x-axis. We explore different options to define this curve.



a) Define the curve using a cosine function.
We can define the curve as $y(x) = p\cos(qx + r) + s$. Here p influences the amplitude, q the stretching along the horizontal axis, r translates the curve horizontally, and s translates the curve vertically.
Now, we can set $r = 0$, since the bump is symmetric around the y-axis. The minimum is achieved for $qx = \pm\pi$, for $x = \pm a$.Hence, $q = \pi/a$. Furthermore, we have $y(0) = b$ and $y(a) = 0$..This gives $p + s = b$ and $-p + s = 0$. Solving this gives $s = b/2$ and $p = b/2$. Putting everything together, we get

$y(x) = (b/2)(\cos(\pi x/a) + b/2$, with $-a \leq x \leq a$ or

$y(x) = (b/2)(1 + \cos(\pi x/a))$, with $-a \leq x \leq a$.

Alternatively, we can observe that $P(t) = (t, \cos t)$, with $-\pi \leq t \leq \pi$, is a curve that has already the desired bump shape, but not the right position and size. To this end, we translate the curve by adding (0, 1), and next we scale the height by $b/2$ and the width by a/π. This gives:

$Q(t) = (at/\pi, (1 + \cos t)b/2)$, with $-\pi \leq t \leq \pi$.

If we want to have a parameter u in the standard unit interval, we use $t(u) = pu + q$, with $t(0) = -\pi$ and $t(1) = \pi$. Solving for p and q gives $t = 2\pi u - \pi$, substitution gives

$Q(u) = (2au - a, ((1 + \cos(2\pi u - \pi))b/2)$, with $0 \le u \le 1$.

b) If we use quadratic or cubic Bézier splines for this shape, we need to use multiple segments. Explain why we need three segments for a quadratic Bézier spline and two segments for a cubic Bézier spline.

The bump has two points of inflection, where the curvature is zero. Such a point of inflection cannot be modelled within a quadratic Bézier spline segment, but only at junctions between segments. Hence, we need two junctions or three segments. A cubic Bézier spline segment allows for one point of inflection, hence we need two of these.

Also, we can use the properties of the control-points to show that using two segments for a quadratic Bézier spline and one segment for a cubic Bézier spline would not lead to the desired result. For a cubic, the control-points P_0 and P_3 are fixed (equal to A and C), and to get a curve that is tangent to the x-axis the control-points P_1 and P_2 must lie on the x-axis also. As a result, the segment becomes a straight line segment and does not pass through B. For the quadratic version, we can define one segment for instance by $P_0 = A$ and $P_2 = B$. Now, the point P_1 in between has to be positioned on the x-axis (such that the curve has the right tangent at A) and also on the line $y=1$ (such that the curve has the right tangent at B). As these lines are parallel, we cannot find such a point P_1.

c) Define the curve using two cubic Bézier segments, by giving all possible positions of the control-points of two segments P and Q. How many degrees of freedom are available? Segment P is defined as $P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$, Q is defined similarly.

We first define the right part of the bump, using one segment P. We fix the start and end by setting

$P_0 = A$ and $P_3 = B$.

To make the curve tangent to the x-axis at A, the point P_1 must be located on the x-axis, to the left of A, i.e.,

$P_1 = (p, 0)$, with $p < a$.

To get a horizontal tangent at B, the point P_2 must be located on a horizontal line through B, i.e.,

$P_2 = (q, b)$, with $q > 0$.

For the left part, we find similarly:

$Q_0 = B$, $Q_1 = (-q, b)$, $Q_2 = (-p, 0)$, $Q_3 = C$.

We reuse the parameters p and q here, to get symmetry around the y-axis.
We have two degrees of freedom, the parameters p and q.

d) Define the curve using three quadratic Bézier segments, by giving all possible positions of the control-points of three segments P, Q, and R. How many degrees of freedom are available? Segment P is defined as $P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$; Q and R are defined similarly.

Suppose the segments are from right to left P, Q, and R. The positions of the endpoints of the bump then give:

$P_0 = A$ and $R_2 = C$.

To get the tangents right at these points, we position the points P_1 and R_1 on the x-axis, similarly to c):

$P_1 = (p, 0)$ and $R_1 = (-p, 0)$ and with $p < a$.

To connect the segments, we set

$P_2 = Q_0$ and $R_0 = Q_2$

Remains to define the middle segment Q. We put the middle point Q_1 somewhere at the y-axis above point B, using a parameter q:

$Q_1 = (0, q)$ with $q > b$.

Next, we fix Q_0. Smoothness dictates that the vectors $(Q_1 - Q_0)$ and $(P_2 - P_1)$ must have the same direction. This is satisfied if the point $P_2 = Q_0$ is located on the line segment between P_1 and Q_1. Hence, we can define

$Q_0 = (1 - r)\, P_1 + r\, Q_1$ and $Q_2 = (1 - r)\, R_1 + r\, Q_1$

using a new parameter r. We are constrained in the value to use for r, because we must make sure that $Q(1/2) = B$. Substitution gives $Q_0 + 2\, Q_1 + Q_2 = 4B$. For the x-coordinate this condition is satisfied by construction, for the y-coordinate we get:

$(1 - r)\, P_{1y} + r\, Q_{1y} + 2\, Q_{1y} + (1 - r)\, R_{1y} + r\, Q_{1y} = 4b$ or

$rq + 2q + rq = 4b$ or

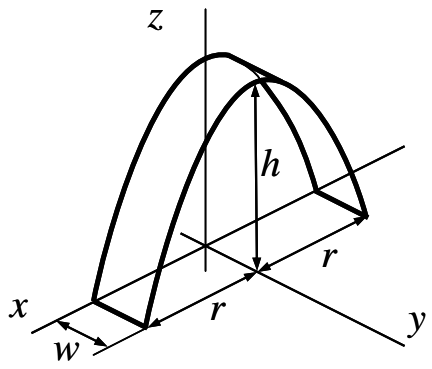
$rq + q = 2b$ hence

$r = (2b - q) / q$.

With this, we have fixed all control-points, using two parameters p and q, giving two degrees of freedom.

---

4 We want to draw an arch as shown in the figure. The arch has the shape of a parabola (and, a parabola in general is given by $z = ax^2 + bx + c$). The maximal spanning distance of the arch, in the plane z=0, is $2r$; the height is h; and the width (or depth) is w.



a) Give a parametric definition of the arch $P(u,v)$, with $-r \le u \le r$ and $0 \le v \le w$.

The arch can be defined as $P(u,v) = (u, v, au^2 + bu + c)$.
We know that $P(0,0) = (0, 0, h)$, $P(-r,0) = (-r, 0, 0)$, and $P(r, 0) = (r, 0, 0)$. This gives (for the z-coordinates): $c = h$, $ar^2 + br + c = 0$, and $ar^2 - br + c = 0$. Subtracting the last two equations gives $b = 0$, and next we find $a = -h / r^2$. Combining everything gives:

$P(u,v) = (u, v, -hu^2 / r^2 + h)$.

b) Give a function *UnderArch*(*x,y,z*) that returns true if the point (*x, y, z*) is located under the arch and above the plane $z = 0$, and false otherwise.

We can define this function by requiring that all coordinates are within certain ranges, where only the z range is special:

UnderArch$(x,y,z) = (0 < z < -hx^2/r^2 + h)$ and $(-r \le x \le r)$ and $(0 \le y \le w)$.

A more compact version is:

UnderArch$(x,y,z) = (0 < z < -hx^2/r^2 + h)$ and $(0 \le y \le w)$.

Also, we can use an implicit definition of the (infinite) arch surface, which is $z = -hx^2/r + h$, or $f(x, y, z) = z + hx^2/r^2 - h$. If $f(x, y, z) > 0$, the point (x, y, z) is above the surface, if $f(x, y, z) < 0$ it is below. This gives, equivalently:

UnderArch$(x,y,z) = (z + hx^2/r^2 - h < 0)$ and $(-r \le x \le r)$ and $(0 \le y \le w)$ and $(z > 0)$.

Finally, note that the test $-r \le x \le r$ is not needed: a point can only survive the tests on z only if x is in this range.

c) Give the normal on the surface for a given point. Choose yourself if this point is given by parameter values u and v, or by coordinates x, y, and z.

Using the parametric definition, we get:

$$N(u, v) = \partial N(u, v) / \partial u \times \partial N(u, v) / \partial v$$
$$= (1, 0, -2hu/r^2) \times (0, 1, 0)$$
$$= (2hu/r^2, 0, 1).$$

Using the implicit definition (see b), we get:

$N(x, y, z) = \nabla f(x, y, z) = (2hx/r^2, 0, 1)$.

d) Give a procedure to draw the arch, assuming a function DT(*A, B, C*) is available to draw a triangle *ABC*. Use a parameter N for the total number of triangles that is used to approximate the arch. You may assume that N is even.

We can use a single loop here, as the surface is not double curved, and use $u=x$ as stepping direction. For instance:

```
drawArch(N);
{
    du = (2*r) / (N/2);          // du is the step in u direction: total width 2r divided by N/2 steps

    for i = 0 to N/2 – 1 do      //  Loop over N/2 steps
    {
        u1 = – r + i*du;         // Get parameter value start of segment
        u2 =   u1 + du;          // Get parameter value end of segment

        A = P(u1, 0);  B = P(u1, w);   // Calculate points ABCD of rectangular segment
        C = P(u2, 0);  D = P(u2, w);

        DT(A, B, C);    // Draw two triangles, taking care that the orientation (order of vertices)
        DR(B, D, C);    // is the same
    }
}
```

As usual, this can also be done in many other ways.