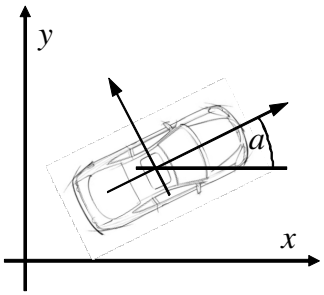
Example Examination 2IV60 - 040712

(translated version 2IV10 4 july 2012, 14:00-17:00)

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 a procedure 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 from each function and procedure a short description of input and output. The use of the book, copies of slides, notes and other material is not allowed.

1 We consider moving a car for a 2D computer game. The car is modeled in local coordinates (see figure). The center of the car is in the origin of the local axis frame, a line from this center to the center of the front of the car coincides with the local positive x-axis. The position and direction of the car in world coordinates is described with a homogenous transformation matrix M, via $Y = MX$, where X is a point in local coordinates and Y a point in world coordinates. Answer the following questions. It may be assumed that $T(x,y)$ and $R(a)$ are functions that return a homogenous transformation matrix, respectively for translation over a vector (x, y) and for counterclockwise rotation around the orign over an angle a in radials. Transformation matrices do not have to be given element-wise.



a) Give a transformation matrix $P(x, y, a)$ that has the effect that the car is moved to point (x, y) and gets direction a.

In local coordinates: Move the car to (x, y) with $T(x,y)$, next rotate the car around the local origine with $R(a)$. Hence $P(x, y, a) = T(x,y)\ R(a)$. In global coordinates: rotate first, next translate, which gives obviously the same effect.

b) It is desired to move the car by multiplying the matrix M with another matrix N, where N represents a move in local coordinates. What is the proper way to multiply: $M' = NM$ or $M' := MN$?

Post-multiplication is the proper way to handle transformations in local coordinates, hence MN is the right answer. In other words: the original point (in local coordinates) is updated first by multiplying with N, and next the current global transformation M is applied.

c) We consider three variations for the move N:

F: Advance the car forward over a distance d;

B_L: Make a turn to the left;

B_R: Make a turn to the right.

For the turns it holds that the car is moved along a circular arc with radius r over an angle α. Give the transformation matrices F, B_L and B_R.

For F we only have to translate the car along the x-axis, hence $F = T(d, 0)$. For B_L we consider how to move the car in global coordinates. We first translate with $T(0, -r)$, such that the center of the arc is in the origin. Next we rotate counterclockwise with $R(\alpha)$ and translate back with $T(0, r)$. Combining this gives $B_L = T(0, r)\ R(\alpha)T(0, -r)$. We can find B_R in a similar way, the difference is that all signs are reversed: $B_L = T(0, -r)\ R(-\alpha)T(0, r)$.

d) We want to make turns with slip. We describe this simply with a sidewards translation per move over a distance s. Give the corresponding transformation matrices B_{SL} en B_{SR} that represent turns with slip.

For B_{SL} we describe the sidewards translation as a translation $T(0, -s)$ in local coordinates: outward from the circular arc. Hence, $B_{SL} = B_L\, T(0, -s)$. Similarly, we get $B_{SR} = B_R\, T(0, s)$, for a rightward turn the car slips in the opposite direction.

---

2 It is desired to draw a mountain landscape. The base plane is a rectangle given by $X_0 \le x \le X_1$ and $Y_0 \le y \le Y_1$, the height z is described with a function $z = f(x,y)$.

a) Give a procedure for drawing this mountain landscape. Make the accuracy controllable. Assume a function *DrawTriangle*(*P,Q,R*) is available for drawing a triangle *PQR*.

We subdivide the initial rectangle in $N \times N$ rectangles, and subdivide each rectangle in two rectangles. We use indices i and j for the x and y direction. A procedure is then (for instance):

```
DrawMountainLandscape(N: integer); // draw mountainlandscape, using N×N squares

    function GP(x, y: float): Point; // return a point of the landscape, given its coordinates (x, y)
    {
        GP = (x, y, f(x, y))
    }
{
    Point P, Q, R, S;  // corners of current rectangle
    float x, y;        //  coordinate of lower left corner rectangle
    float dx, dy;      //  width and height of rectangle

    dx = (X1–X0)/N;  // calculate width dx and height dy of rectangle
    dy = (Y1–Y0)/N;

    for i = 0 to N do
    {
        x = X0 + i*dx;
        for j = 0 to N do
        {
            y = Y0 + j*dy; // x and y now known, let's calculate points

            P = GP(x, y);
            Q = GP(x+dx, y);
            R = GP(x+dx, y+dy);
            S = GP(x, y+dy);

            DrawTriangle(P, Q, R);  // split up the rectangle in two triangles and draw these.
            DrawTriangle(P, R, S);  //  choose order such that both have the same orientation.
        }
    }
}
```

Obviously, this can be accelerated in many ways (for instance by not recalculating points), but here the aim is to get a minimal and readable version.

b) We add illumination. We use diffuse reflection and color triangles uniformly. The viewing point is at a position E, the vector L points towards a light source at infinity with intensity I. Assume a function *SetGray*(g) is available for setting the gray value (between 0 and 1) of triangles to be drawn. Give a procedure *DrawShadedTriangle*(*P,Q,R*), where the grey value is set before drawing the triangle.

We use the standard graphics model based on Lambert's law to calculate the diffuse reflection: the diffuse reflection is proportional to the cosine of the angle of a unit vector towards the light source and the normal on the surface. This gives:

```
DrawShadedTriangle(P,Q,R);
{
    vector N =(Q–P)×(R–P);  // get normal vector on triangle with cross-product.
    N = N / | N |;          // normalize the normal to get a unit-length vector

   // Check if the normal has the right sign. The normal should be outward to the side
   // we are looking at. We check this by looking at the sign of the dot product of the current
   // normal and a vector E–P towards the view point.

    if N.(E–P) < 0 then N = –N;

  // Next, we apply Lambert's law to get the cosine:

   cosa = N.L/|L|;   // Note: The normalization of L can be taken out of the loop

   if cosa < 0 then g = 0          // Light at other side of surface, hence no illumination
               else  g = cosa*I;   // else, multiply with intensity I.

   SetGray(g);                 // Set the gray value, and
   DrawTriangle(P, Q, R); // draw the triangle
}
```

c) It is desired to accentuate steep parts of the mountain landscape by coloring triangles red that are at an angle larger than 60 degrees with the horizontal plane. Assume that a function *SetColor*(*r,g,b*) is available for controlling the color. Describe how this can be implemented.

Let *a* be the angle between the normal of a triangle and a normal to the horizontal plane. If *a* is larger than 60 degrees (and smaller than 120 degrees) then the triangle is considered steep. Of course, we do not calculate the angle explicitly, it suffices to use cos(*a*), which is equal to *N*.(0, 0, 1). If |cos(*a*)| < cos(60 degrees) then the triangle has to be colored red. We implement this by replacing the call to *SetGray* in our previous function by:

```
if abs(N_z) < 0.5          // using cos(60 degrees) = 1/2
then SetColor(g, 0, 0);  // Modulate only the red component if steep
else SetColor(g, g, g);  // else make use g for all components
```

Some possible variations on *SetColor*(*g*, 0, 0) are *SetColor*(1, 0, 0) (ignore illumination, give a fully saturated alarm color) and *SetColor*(*g*, *g*/3, *g*/3) (lower saturation of red triangles).

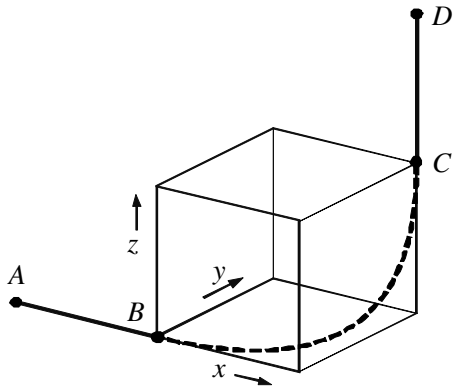
d) Assume that the *z*-buffer algorithm cannot be used, what is then a simple method to make sure that in the end only visible surfaces are displayed?

Here the Painter's or depth-sort algorithm can be conveniently used, because the relative order of triangles is fixed by the grid in the horizontal plane. In other words, just drawing the triangles from back to front (by appropriately tuning the loops for *i* and *j*) will suffice.

3 Given two line segments AB and CD, A=(–1,0,0), B=(0,0,0), C=(1,1,1) and D=(1,1,2). It is desired to connect point B and C with a Bézier quadratic spline, such that we obtain a single smooth curve. A segment of a Bézier quadratic spline with control points P_0, P_1 and P_2 is given by

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 .$$

a) Describe why at least two segments are needed to meet the demands.



Suppose a single segment is used. The point P_0 has to coincide with B, the point P_2 with C to achieve zero-order continuity. So far, so good. However, the point P_1 has to be located on the infinite line through A and B, and also on the infinite line through C and D to achieve smooth transitions at B and C. These constraints cannot be satisfied simultaneously, because there is no such point: the two infinite lines do not intersect.

b) Suppose that two segments P and Q are used, with control points P_0, P_1 and P_2, and Q_0, Q_1 and Q_2. What are the conditions to the control points to make sure that the two segments are connected and that all transitions from segment to segment are smooth?

For zero-order continuity, the control-points should satisfy:
$P_0 = B$;
$P_2 = Q_0$;
$Q_2 = C$.
For first-order continuity, the middle-points should satisfy:
$P_1 = B + u(B-A)$, with $u > 0$;
$Q_1 = C + v(C-D)$, with $v > 0$;
and also, to make sure that the transition between P and Q is smooth:
$Q_1 - Q_0 = w(P_1 - P_2)$ with $w > 0$.

The last constraint can also written as:
$P_2 = (1-t)Q_1 + tP_1$ with $0 < t < 1$
to emphasize that the endpoint of P (and the startpoint of Q) must be located on a line segment inbetween the two middle-points P_1 and Q_1.

c) Give all possible positions for the control points, with as additional constraint that the segments should not be outside the unit cube $0 \le x, y, z \le 1$. How many free parameters are left over?

If all control-points are inside the unit cube, then the segments are certainly inside, thanks to the convex hull property. This holds if $u < 1$ and $v < 1$. If for instance u has a larger value, then the x-coordinate of P_1 will be > 1, and also the line segment from P_1 to Q_1 will be located outside the unit cube (Q_1 has always an x=1, independent of v). This makes it impossible to find a value for t such that P_2 is inside the box, and hence the set of possible positions is given by $0 < u, v, t < 1$, using the notation of the previous sub-question. Hence, we (still) have three free parameters left over.

d) Next, determine the control points such that all distances between succeeding control points (P_0 and P_1, P_1 and P_2, Q_0 and Q_1, and Q_1 and Q_2) are equal.

Simple substitution gives that the length $|P_1P_0|$ is equal to u, and that the length $|Q_1Q_2|$ is equal to v. As these length must be equal, u=v. Point P_2 must be halfway P_1 and Q_1, hence $t = 1/2$. Remains to fix the value of u. We calculate
$|P_2P_1| = |P_1P_0|$, or
$|Q_1P_1|/2 = u$, or

$|(1, 1, 1-u) - (u, 0, 0)| = 2u$, or
$|(1-u, 1, 1-u)| = 2u$.
Taking the square of both sides we get
$2(1-u)^2 + 1 = 4u^2$.
Solving this in the standard way and removing the spurious root, we get
$u = \text{sqrt}(10)/2 - 1 \approx 0.5811$.

---

4 We consider basic techniques and concept for computer graphics.

a) Mention a characteristic difference between *perspective* projection and *parallel* projection:

One of:
- In parallel projection lines that are parallel in world space remain parallel after projection, in contrast to perspective projection;
- In perspective projection, objects that are far away are displayed smaller, whereas in parallel projection the size does not change dependent on distance from the projection plane;
- In perspective projection a division by depth is needed in the projection step, in parallel projection this is not necessary.

b) Mention a characteristic difference between *convex* and *concave* polygons.

One of:
- all interior angles of a convex polygon are smaller than 180 degrees, concave polygons can have larger angles;
- Let A and B be points inside and outside the polygon P. If P is convex, then the number of crossings of the line segment AB with P is 1, for concave polygons this can be higher.
- Let A and C be points inside and at the polygon P. If P is convex, then the line segment AC (exclucing end points) does not intersect P, for concave polygons this can happen.

c) Mention two illumination/shading effects that can be simulated with ray-tracing and not with a simple illumination model.

Two of:
- transparency with multiple layers;
- refraction in transparency;
- mirroring reflection;
- cast shadows.

d) What is the difference between *flat surface rendering* and *Gouraud surface rendering*?

In flat surface rendering the color of triangles is constant, whereas in Gouraud surface rendering the color can vary. More precisely, in Gouraud surface rendering the colors of the vertices are linearly interpolated over the interior.
