Prof. Dr. Leif Kobbelt
J. Born, M. Ibing,
J. Möbius, P. Schmidt

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Basic Techniques in Computer Graphics

Assignment 5

Date Published: November 28th 2017, Date Due: December 5th 2017

- All assignments (programming and text) have to be done in teams of 3–4 students. Teams with less than 3 or more than 4 students will receive no points.
- Hand in **one solution per team per assignment**.
- Every team must work independently. Teams with identical solutions will receive no points.
- Solutions are due 18:00 on December 5th 2017. Late submissions will receive zero points. No exceptions!
- Instructions for **programming assignments**:
  - Download the solution template (a zip archive) through the L^{2}P course room.
  - Unzip the archive and populate the `assignmentXX/MEMBERS.txt` file. Any team member not listed in this file will not receive any points! (Also see the instructions in the file.)
  - Complete the solution.
  - Prepare a new zip archive containing your solution. It must contain exactly those files that you changed. **Only change those files you are explicitly asked to change in the task description.** The directory layout must be the same as in the archive you downloaded. (At the very least it must contain the `assignmentXX/MEMBERS.txt`.)
  - Upload your zip archive through the L^{2}P before the deadline.
  - Your solution must compile and run correctly **on our lab computers** using the exact same `Makefile` provided to you. If it does not, you will receive no points.
- Instructions for **text assignments**:
  - Prepare your solutions on paper.
  - If you write your solution by hand, write neatly! Anything we cannot decipher will receive zero points. No exceptions!
  - If you hand in more than one sheet, staple your sheets together. (No paper clips!)
  - Put the names and matriculation numbers of all team members onto every sheet.
  - Unless explicitly asked otherwise, always justify your answer.
  - Be concise!
  - Put your solution into the designated drop box at our chair before the deadline. (1st floor, E3 building.)

Prof. Dr. Leif Kobbelt
J. Born, M. Ibing,
J. Möbius, P. Schmidt

Exercise 1 The Depth Buffer [3 Points]

As explained in the lecture, the frustum mapping transforms the original z-coordinates (in the eye space of the camera) *non-linearly* from $[-n, -f]$ to $[-1, 1]$. Afterwards, the depth-values are linearly transformed to the range $[0, 1]$ (where 0 corresponds to the near cutting plane) and then discretized to the m bits of the depth buffer (here: by rounding to the next smallest integer value).

For the following, let $n = 200$, $f = 1000$ and $m = 24$.

(a) [1 Point]

Consider the point $p = (0, 0, -500)$ given in the local coordinate system of the camera. Derive the depth values assigned to p in the different stages of the pipeline described above, *i.e.*, derive the depth value of p in normalized device coordinates ($[-1, 1]$), in the range $[0, 1]$ and the final depth buffer value.

(b) [2 Points]

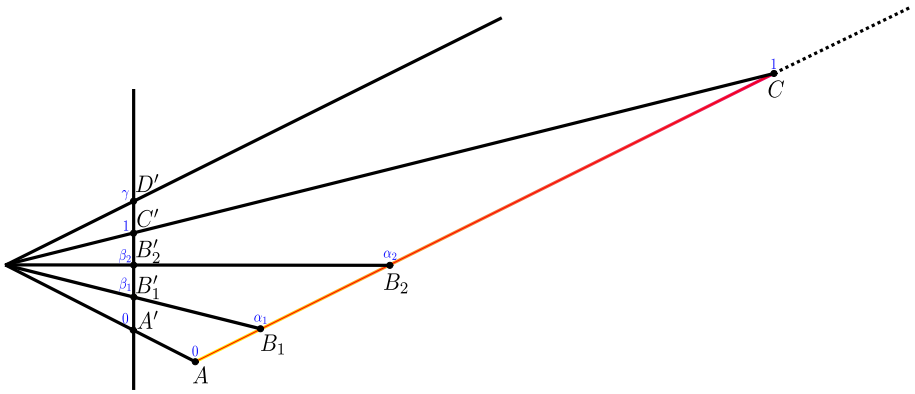
Given two points $q_0 = (0, 0, z_0)$ and $q_1 = (0, 0, z_1)$ with z_0, $z_1 \in [-n, -f]$, let n_{q_0} and n_{q_1} be the depth buffer values of the images of q_0 and q_1. Determine the smallest distance $|z_1 - z_0|$ such that $|n_{q_1} - n_{q_0}| > 1$, independent of the actual choices for z_0 and z_1 (within $[-n, -f]$).

Prof. Dr. Leif Kobbelt
J. Born, M. Ibing,
J. Möbius, P. Schmidt

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Exercise 2 Perspective Projection of Lines [2 Points]

In the lecture we saw that points on a line do not keep their ratios when mapped to the image plane. Here, we want to examine this effect by projecting a colored line segment. We consider its endpoints A and C and two intermediate points B_1 and B_2. All four points correspond to neighboring fragments A', B'_1, B'_2, C' on the screen. D' marks the vanishing point of the line. Although the color of the original line blends *linearly* from yellow to magenta, the color of its projection will change *non-linearly*.



All points except B_2 are given:

$$A = \begin{pmatrix} 6 \\ -3 \end{pmatrix}, B_1 = \begin{pmatrix} 8 \\ -2 \end{pmatrix}, C = \begin{pmatrix} 24 \\ 6 \end{pmatrix}$$

$$A' = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, B'_1 = \begin{pmatrix} 2 \\ -0.5 \end{pmatrix}, B'_2 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, C' = \begin{pmatrix} 2 \\ 0.5 \end{pmatrix}, D' = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

The color of L is defined as an RGB value by

$$c(\alpha) = \alpha \cdot \begin{pmatrix} 0.89 \\ 0 \\ 0.4 \end{pmatrix} + (1-\alpha) \cdot \begin{pmatrix} 1 \\ 0.93 \\ 0 \end{pmatrix}.$$

(a) [1 Point]

Give parametric representations for the original line $L(\alpha)$ as well as for the projected one $L'(\beta)$ such that $L(0) = A, L(1) = C$ and $L'(0) = A', L'(1) = C'$. Then, find the parameters α_1 and β_1 for the point B_1 and its projection B'_1. Notice that these parameters do not match. Look up the color for fragment B'_1 using the correct parameter.

(b) [1 Point]

Now we want to color the second fragment B'_2. However, the parameter α_2 is not given. First, find β_2 as before. Then, compute the corresponding α_2 using the formula we derived in the lecture. Hint: You will need the parameter γ of the vanishing point. Finally, find the color of fragment B'_2.

Prof. Dr. Leif Kobbelt
J. Born, M. Ibing,
J. Möbius, P. Schmidt

Exercise 3 Programming: Rasterization based on Pineda [5 Points]

For this exercise you will only need to add your own code to `assignment04/assignment.cpp`. Modifications to any other file (apart from `assignment04/MEMBERS.txt`) are not permitted.

The goal of this exercise is to implement a simple parallelizable rasterizer based on the algorithm by Pineda. The simplest version of this algorithm iterates for each triangle over all pixels of the screen and decides if it is inside the triangle (pixel will be drawn) or outside (pixel gets discarded). The decision is made by iterating over each edge of the current triangle. The implicit function F defined by each edge is evaluated at the current pixel. The result shows, if the point is left or right of the edge. If the pixel is left of all triangle edges (defined in counterclockwise order), the pixel is inside and will be drawn.

Note: Drawing all triangles by testing all screen pixels can be very slow! You might want to draw less triangles until you implement the bounding box (part b).

(a) [1 Point]

Complete the function **`int`** `evaluateF(...)` which is supposed to implement the evaluation of the implicit function F! F is used to decide whether a point is left or right of an edge. The edge is defined by two points `p1` and `p2`. **`int`** `evaluateF(...)` is passed an additional point `p`, at which F is evaluated. If the point `p` is left of the line going from `p1` to `p2`, the function has to return a positive value, otherwise a negative value.

(b) [1 Point]

Going over all points of the screen for each triangle is very inefficient. Within the function **`void`** `drawTriangle(...)` Change the values of `minX`, `minY`, `maxX`, `maxY` such that only points within the minimal bounding box of the triangle are rasterized. Furthermore, make sure that you do not rasterize points outside the screen ($x, y < 0$ or $x \geq$ width, $y \geq$ height).

(c) [1 Point]

In **`void`** `drawTriangle(...)`, at the marked position decide if a point is outside the triangle using the function **`int`** `evaluateF(...)` function from Exercise (a). If the point is inside the triangle, use the setPixel function to draw it. (At this point of the exercise you should see a rendered bunny.)

(d) [0.5 Points]

In **`void`** `drawScene(...)`, at the marked position implement a rotation of the model around the y axis depending on the time. (You can use the `_runTime` parameter to get the number of seconds passed since the start of the program.) Apply the rotation to the correct matrix in the matrix set which is already defined. The rotation is only used in scene 2 and 3 (activate with `b` and `c` key).

(e) [0.5 Points]

If you implemented the rotation above, you can see that the screen is not cleared after every frame. In **`void`** `drawScene(...)`, at the marked position use the `setPixel(...)` to implement a simple screen clearing by setting all pixels to black.

(f) [1 Point]

The visualization still only shows a silhouette. In order to get a better idea of its contour, compute a very simply diffuse lighting coefficient. In **`void`** `drawTriangle(...)`, at the marked position update the variable `diffuse` (that has previously been initialized with 1.0) with the diffuse lighting coefficient, given as the dot product between the normal and the viewing direction (0,0,1).

Since we are dealing with the special case where the position of the camera and the position of the light source are the same, you can use the diffuse lighting coefficient to perform backface culling as well, which

means that we disregard triangles that do not point to the screen. If the normal of the triangle points into the opposite direction of the viewing direction, we see the back of the triangle. Simply skip the rasterization for backfacing triangles by returning from the function early. (Note that we did not implement a z buffer here. Therefore you will see some artifacts in the image.)

Note: `void drawTriangle(...)` is provided with normalized device coordinates, i.e. the projection has already been applied to these points.

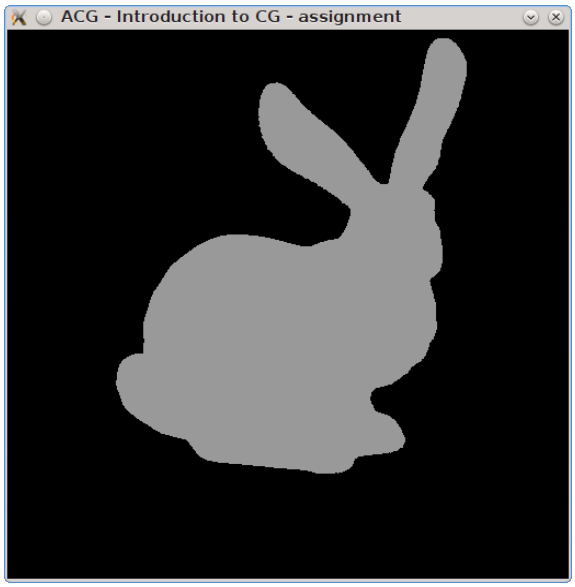


Figure 1: This is what the scene should look like after completing (a)–(e).

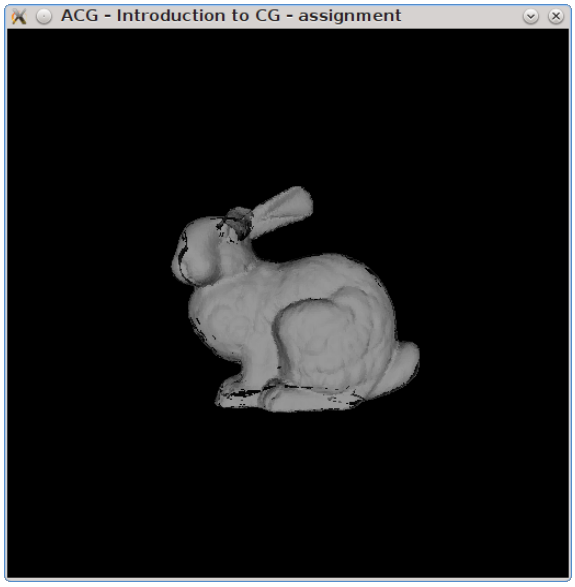


Figure 2: This is what the scene should look like after completing (a)–(f) in rotation mode (activated with the c key). Note the artifacts in the picture due to the lack of a z buffer.