Prof. Dr. Leif Kobbelt
Dr. Jan Möbius
Patrick Schmidt, Moritz Ibing

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Basic Techniques in Computer Graphics

Assignment 5

Date Published: November 12th 2019, Date Due: November 19th 2019

- All assignments (programming and text) have to be completed in teams of 3–4 students. Teams with fewer 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 14:30 on November 19th 2019. Late submissions will receive zero points. No exceptions!
- Instructions for **programming assignments**:
  - Download the solution template (a zip archive) through the Moodle course room.
  - 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.
  - Upload your zip archive through Moodle before the deadline. Use the Moodle group submission feature. Only in the first week (when Moodle groups have not been created yet), list all members of your group in the file `assignmentXX/MEMBERS.txt`. Remember, only one submission per group.
  - Your solution must compile and run correctly **on our lab computers** using the exact same `Makefile` provided to you. Do not include additional libraries and do not change code outside of the specified sections. If it does not compile on our machines, you will receive no points.
- Instructions for **text assignments**:
  - Prepare your solution as a single pdf file per group. Submissions on paper will not be accepted.
  - If you write your solution by hand, write neatly! Anything we cannot decipher will receive zero points. No exceptions!
  - Add the names and student ID numbers of all team members to every pdf.
  - Unless explicitly asked otherwise, always justify your answer.
  - Be concise!
  - Submit your solution via Moodle, together with your coding submission.

Prof. Dr. Leif Kobbelt
Dr. Jan Möbius
Patrick Schmidt, Moritz Ibing

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Exercise 1 Frustum Mapping for Orthogonal Projections [8 Points]

In contrast to perspective projections, in orthogonal projections all viewing rays are orthogonal to the image plane. As a result, the viewing frustum of an orthogonal projection has a different as from the one of a perspective projection. Also, the frustum transformation is much simpler for orthogonal projections. In this exercise you will derive both, the shape of the viewing frustum and the matrix of the frustum transformation.

(a) Shape of the Frustum [2 Points]

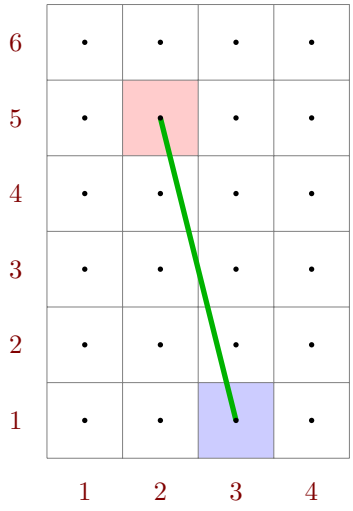
What geometric shape does the viewing frustum of an orthogonal projection have if both, the near and the far plane are parallel to the image plane? Explain your reasoning in a few sentences and name the resulting shape. No formulas needed here.

(b) Transformation Matrix [6 Points, 2 per matrix]

Similarly to a perspective projection, the viewing frustum of an orthogonal projection can be uniquely defined by a far plane, a near plane and a rectangle on the near plane. Let the near, far and image plane be orthogonal to the z-axis at $z_{\text{near}} = -n$, $z_{\text{far}} = -f$ and $z_{\text{image}} = -1$. Let the rectangle on the near plane be given by the top coordinate t, the bottom coordinate b, the left coordinate l and the right coordinate r. Derive the frustum matrix $\mathbf{M} \in \mathbb{R}^{4\times4}$ that maps this frustum into the cube $[-1, 1]^3$ as a combination of a translation $\mathbf{T}$ and a scaling $\mathbf{S}$. Specify the matrices of both transformations as well as the final transformation matrix. Remember that the near plane is mapped to the plane orthogonal to the z-axis at $z = -1$.

Exercise 2 Line Rasterization [8 Points]

In this task you are going to rasterize the line in the figure below using the Bresenham algorithm introduced in the lecture. Keep in mind that the grid represents a set of pixels where the center of each pixel is marked by a dot and has the integer coordinates (x, y).



(a) Case Reduction [2 Points]

Our line segment starts at $x_0 = (3, 1)^\mathsf{T}$ and ends at $x_1 = (2, 5)^\mathsf{T}$. **Specify the map** $(x, y)^\mathsf{T} \to (\dots, \dots)^\mathsf{T}$ transforming the line into a setting with slope $m \in [0, 1]$, as seen in the lecture. **Apply this map** to x_0 and x_1.

(b) Bresenham Algorithm [6 Points]

Compute the **initial value of the decision variable** d, as well as the **updates** Δ_{E} and Δ_{NE} using the formulas derived in the lecture.

Then, execute the Bresenham algorithm by hand and **fill all blanks** in the following table. Make sure to use integer arithmetic only.

Hint: To check your result, you can appy the inverse transformation from part (a) to each point, and complete the above figure.

Step	x	y	d	decision (E or NE)
0				
1				
2				
3				
4				End

Exercise 3 Rasterization of Quadratic Polynomials [8 Points]

The idea of the Bresenham algorithm, *Digital Differential Analysis* (DDA), can be generalized to the stepwise evaluation of arbitrary polynomials. In this task you will adapt this algorithm to the rasterization of quadratic polynomials that have the form

$$y = f(x) = ax^2 + bx + c.$$

For this question, we assume that the polynomial is rasterized for $x \in \{x_0, x_0 + 1, \ldots, x_0 + n\}$, and that $0 \leq f'(x) \leq 1$, for all $x \in [x_0, x_0 + n]$.

We can turn this explicit formulation into an implicit one by checking if a point lies above or below the line:

$$F(x, y) = y - f(x) = y - ax^2 - bx - c.$$

(a) Decision Variable [4 Point]

Suppose that we finished drawing the pixel $(x, y)^{\mathsf{T}}$. For the next pixel $x + 1$, the algorithm will have to choose between either east (E) or north east (NE). **Give a formula** for the decision function $d(x, y)$ and **explain** how $d(x, y)$ can be used to decide whether to go east or north east.

(b) Decision Variable Updates [4 Point]

Instead of computing $d(x, y)$ for each pixel from scratch, we can use DDA to compute $d(x + 1, y)$, or $d(x + 1, y + 1)$ for the next pixel as an increment of $d(x, y)$. **Derive** the increment for both cases, i.e. $\Delta_{\mathrm{E}}(x, y)$ and $\Delta_{\mathrm{NE}}(x, y)$. Note that both expressions are now linear functions in x and y.

You can stop here, i.e. you don't have to compute the constant updates for those linear updates in this task. Also, it is okay that your expressions contain non-integer arithmetic.

Prof. Dr. Leif Kobbelt
Dr. Jan Möbius
Patrick Schmidt, Moritz Ibing

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Exercise 4 Programming: Rasterization based on Pineda [20 Points]

For this exercise you will only need to add your own code to `assignment05/assignment.cpp`. Modifications to any other file 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) [4 Point]

Complete the function **`float`** `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`. **`float`** `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) [4 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) [4 Point]

In **`void`** `drawTriangle(...)`, at the marked position decide if a point is outside the triangle using the function **`float`** `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) [2 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) [2 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) [4 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.

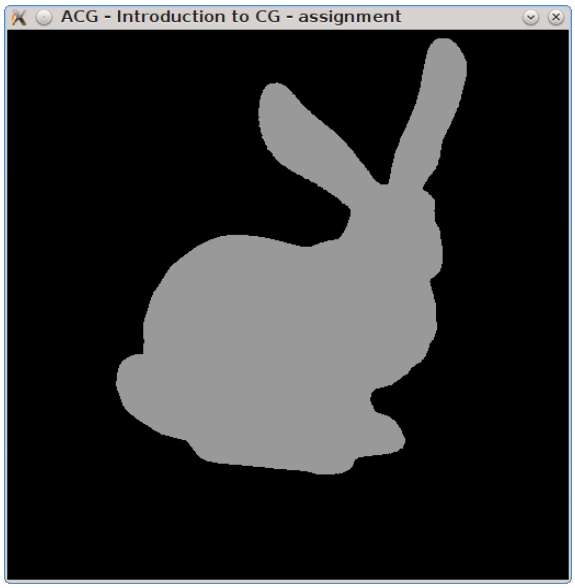


Figure 1: This is what the scene should look like after completing (a)–(e).

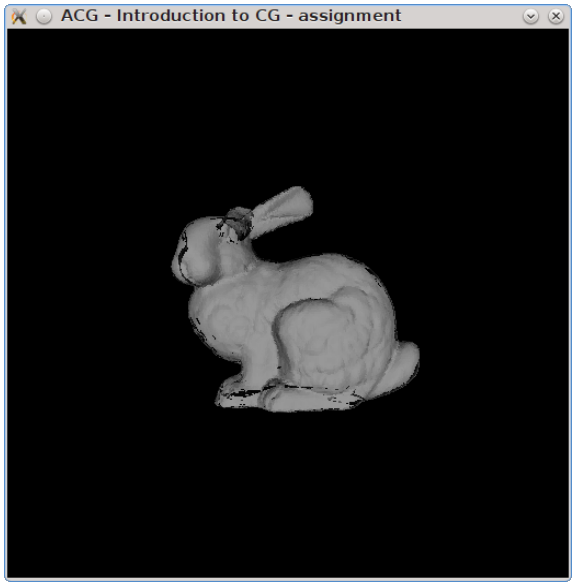


Figure 2: This is what the scene should look like after completing (a)–(f) in rotation mode (activated with the c key). Note the artifacts in the picture due to the lack of a z buffer.