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

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Basic Techniques in Computer Graphics

Assignment 4

Date Published: November 21st 2017, Date Due: November 28th 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 November 28th 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 Frustum Mapping for Orthogonal Projections [2 Points]

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

(a) Shape of the Frustum [0.5 Points]

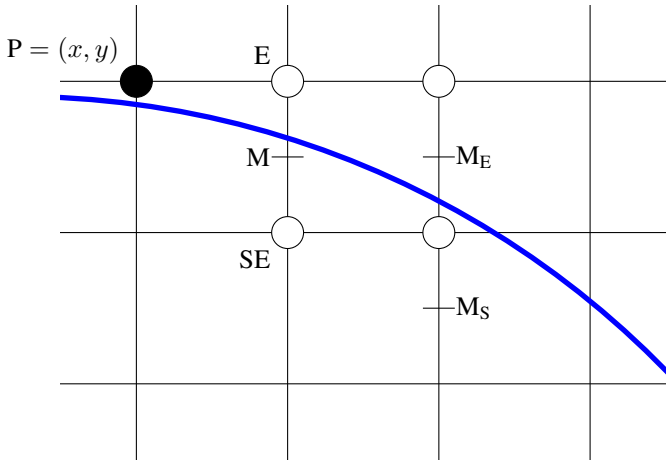
What geometric shape does the viewing frustum of an orthogonal projection have if both, the near and the far plane are parallel to the image plane? Explain your answer!

(b) Transformation Matrices [1.5 Points, 0.5 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 transform that linearly maps this frustum into the cube $[-1, 1]^3$ as a combination of a translation and a scaling transform. 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 $z = -1$.

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

Exercise 2 Circle Rasterization [3 Points]



In the lecture the Bresenham algorithm for line rasterization was presented. It was also indicated that shapes with arbitrary polynomial implicit representations can be rasterized with some modifications to the Bresenham algorithm. In this exercise we would like to derive the Bresenham algorithm for a circle, given that it is centered at the origin and has a radius r.

(a) [0.5 Points]

Because of the symmetry of the circle, the algorithm doesn't need to be applied for the whole circle. Which part of the circle has to be calculated and which parts can be derived out of the symmetry?

(b) [0.5 Points]

Write down the implicit representation of the circle and show how this can be used to decide whether a point lies inside or outside the circle.

(c) [1 Point]

Suppose that we finished drawing the point P and we are moving clockwise to the next pixel. The algorithm will have to choose between either E or SE. Give a formula for the decision variable d and define for which values which point is chosen.

(d) [1 Point]

Give the Bresenham algorithm for the whole circle rasterization as pseudo-code. Don't forget the initial parameters.

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

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Exercise 3 Programming [5 Points]

In this exercise you add a third dimension to last week's 2D racing "game". We already went ahead and replaced the circles by spheres and implemented the game logic and rendering, including simple lighting, for you.

All functions you need to change are implemented in `assignment.cpp`. Only make modifications to that file! Do *not* modify any other file in the folder (besides `MEMBERS.txt`)!

(a) Look-at Transformation [1 Point]

Complete the `lookAt(...)` function. It gets camera position, viewing direction, and up-vector as parameters and should set up and return a view matrix of type `glm::mat4` that performs the corresponding lookAt transformation (see lecture slides) – such that afterwards everything is in a standard projection setting. Remember that you have to normalize the forward, up and right vector before constructing the lookAt view matrix to avoid scaling effects.

Note: You have to set the matrix entries yourself. *Do not* use any of the convenience function that do the work for you! You may define multiple matrices and use the predefined matrix multiplication operator to combine them.

(b) Frustum Transformation [1 Point]

Complete the `buildFrustum(...)` function. Its arguments are the vertical field of view angle ϕ in degree (`phiInDegree`), the screen's aspect ratio `aspectRatio`, and the near and far plane distances `near` and `far`. Complete this function so it sets up and returns a matrix that performs the frustum transformation according to these parameters (see lecture slides).

Note: Just as in Exercise (a), set the matrix entries yourself!

(c) Camera Transformation 1 [1 Point]

The race track revolves around the origin. To get a good look at it, call the `lookAt(...)` function near the beginning of `drawScene(...)` (the comments in the source code indicate where exactly) to create a view matrix that positions the camera at $\begin{pmatrix}0 & -1 & 1\end{pmatrix}$, looking towards the origin. Store this matrix in the global variable `g_ViewMatrix` (which is then used for the track rendering in the routines we already implemented for you).

(d) Using the Frustum Transformation [1 Point]

The `buildFrustum(...)` function needs to be called and the returned matrix has to be stored to the global variable `g_ProjectionMatrix`. This needs to be done not only once, but every time the user resizes the window because that changes the dimensions and the aspect ratio of the frustum. In our framework the function `resizeCallback(...)` gets called whenever the window gets resized (and also once at startup). Hence, this is the perfect place for you to set `g_ProjectionMatrix`. Use a vertical field of view angle of $90°$, use the correct aspect ratio, and set the near and far plane to some suitable values such that the scene (which is centered around the origin and is somewhat smaller than a unit cube) is fully visible—the actual choice is not too important in our case. But remember that using very large z-ranges increases z-fighting problems. After that, you should finally be able to see the track.

(e) Camera Transformation 2 [1 Point]

In addition to being able to view the track from a fixed point of view, we now also want to let the camera drive inside one of the cars, looking into the direction of travel. To this end, inside the function `drawScene(...)`, in scene 5 (`if (scene == 5)`), set `g_ViewMatrix` to a look-at transformation that depends on the values `height` and `angle1`. (Of course, we want you to use the `lookAt(...)` function you programmed earlier.)

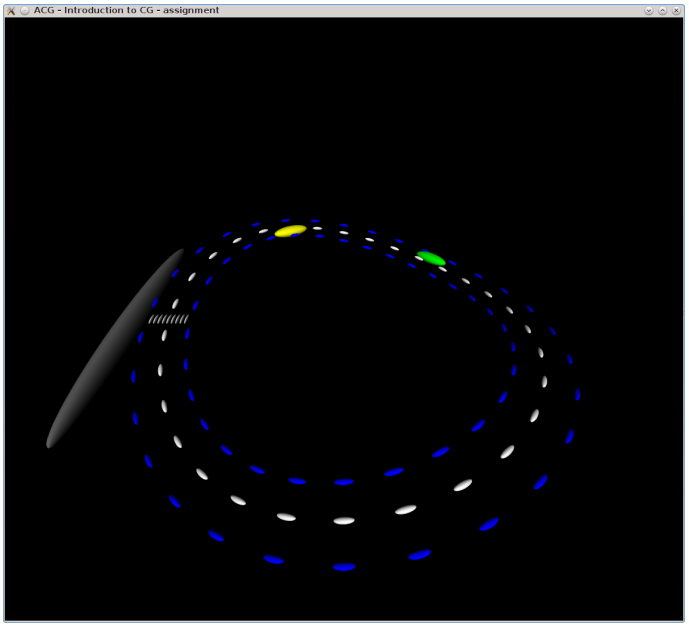


Figure 1: The track seen with a FoV of 90° from $\begin{pmatrix} 0 & -1 & 1 \end{pmatrix}$.

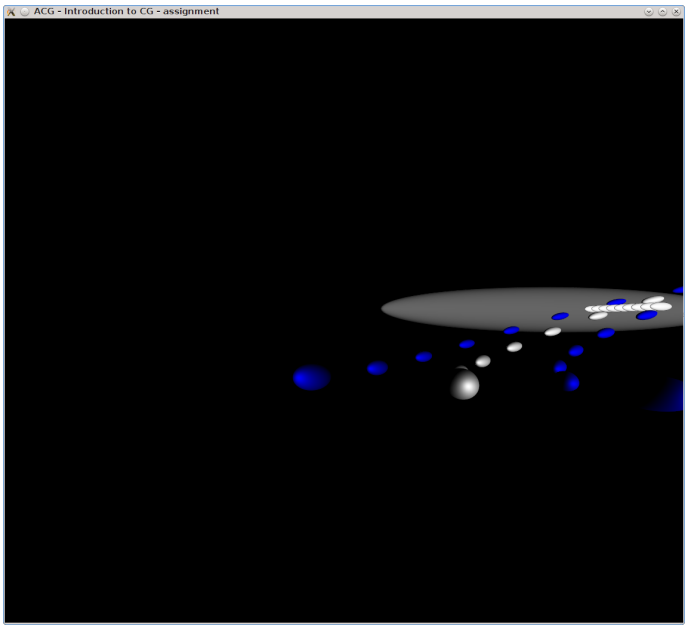


Figure 2: Driving along the track.