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

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Basic Techniques in Computer Graphics

Assignment 4

Date Published: November 05th 2019, Date Due: November 12th 2019

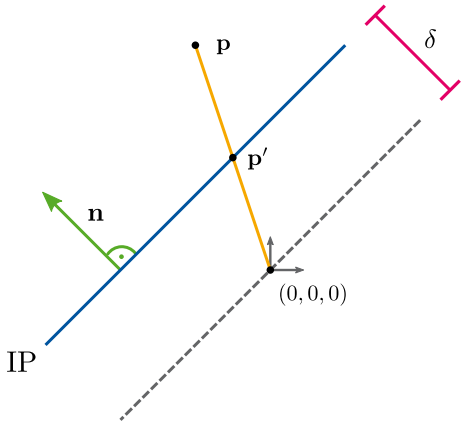
- All assignments (programming and text) have to be completed in teams of 3–4 students. Teams with fewer than 3 or more than 4 students will receive no points.
- Hand in **one solution per team per assignment**.
- Every team must work independently. Teams with identical solutions will receive no points.
- Solutions are due 14:30 on November 12th 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 Projective Geometry [20 Points]

Consider the following sketch of a 3D scene with the camera located at the origin $(0,0,0)^\mathsf{T}$. The image plane (IP) is defined by the normal vector $\mathbf{n}$ with $||\mathbf{n}|| = 1$ and the focal distance δ. The point $\mathbf{p}'$ is the projection of $\mathbf{p}$ onto the image plane.



(a) Projection Matrix [4 Points]

Assume that $\mathbf{n} = (0, -\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2})^\mathsf{T}$. Use the intercept theorem[1] to **derive** the projection matrix $\mathbf{P} \in \mathbb{R}^{4\times4}$ that maps $\mathbf{p}$ to $\mathbf{p}'$ depending on δ. Both points are expressed in homogeneous coordinates, i.e. $\mathbf{p} \in \mathbb{R}^4$ and $\mathbf{p}' \in \mathbb{R}^4$. Don't forget to explain your derivation.

(b) Vanishing Points Derivation [4 Points]

From now on, assume we picked a value for δ such that the projection matrix becomes

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 \end{bmatrix}.$$

Given a line $\mathbf{L}(\lambda) = \mathbf{o} + \lambda\mathbf{d}$, **derive** a formula for its vanishing point under the projection $\mathbf{P}$. Since we did not define a 2D coordinate system on the image plane yet, you can specify the vanishing point in 3D.
Hint: Note that we are not in the standard projection setting.

(c) Existence of Vanishing Points [2 Points]

In which case does a vanishing point exist in this setting? **Write down** a formula for this condition.

(d) Compute Vanishing Points [4 Points]

Consider the triangle spanned by $\mathbf{p}_0 = (-1,2,2)^\mathsf{T}$, $\mathbf{p}_1 = (-2,4,0)^\mathsf{T}$ and $\mathbf{p}_2 = (2,0,-2)^\mathsf{T}$. **Compute** the vanishing points of its edges under $\mathbf{P}$, or argue why they do not exist. Again, you can specify the vanishing points as 3D points on the image plane. The normal is still given as $\mathbf{n} = (0, -\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2})^\mathsf{T}$.

(e) Geometric Construction of Vanishing Points [2 Points]

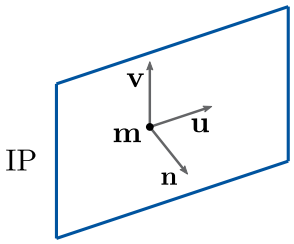
As an alternative to the formula you derived in part (b), briefly **describe** a geometric construction that also gives the vanishing point of the line $\mathbf{L}(\lambda) = \mathbf{o} + \lambda\mathbf{d}$ without using the projection matrix $\mathbf{P}$.

[1] https://en.wikipedia.org/wiki/Intercept_theorem

(f) Local Coordinate Systems [4 Points]

So far, we expressed points on the image plane in 3D. This is because in this exercise the viewing direction is not aligned with one of the standard axes, so we cannot simply drop one coordinate.

We are now given a 3D point $\mathbf{m}$ and two orthogonal 3D vectors $\mathbf{u}$ and $\mathbf{v}$, all lying within the image plane. Using homogeneous coordinates, **derive** the matrix $\mathbf{M} \in \mathbb{R}^{3\times4}$ that expresses a 3D point $(x, y, z, w)^\mathsf{T}$ on the image plane as a 2D point $(\alpha, \beta, w)^\mathsf{T}$ within the local coordinate system with the origin $\mathbf{m}$ and main axes $\mathbf{u}$ and $\mathbf{v}$.



You can assume, that $\mathbf{n}$, $\mathbf{u}$ and $\mathbf{v}$ form an orthonormal frame, i.e. $\mathbf{n}^\mathsf{T}\mathbf{u} = \mathbf{n}^\mathsf{T}\mathbf{v} = \mathbf{u}^\mathsf{T}\mathbf{v} = 0$ and $\mathbf{n}^\mathsf{T}\mathbf{n} = \mathbf{u}^\mathsf{T}\mathbf{u} = \mathbf{v}^\mathsf{T}\mathbf{v} = 1$. Don't forget to explain your derivation.

Exercise 2 Programming [20 Points]

In this task 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.

(a) Look-at Transformation [4 Points]

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 functions that do the work for you! You may define multiple matrices and use the predefined matrix multiplication operator to combine them.

(b) Frustum Transformation [4 Points]

Complete the buildFrustum (...) function. Its arguments are the vertical field of view angle ϕ in degrees (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 [4 Points]

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 [4 Points]

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 [4 Points]

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.)

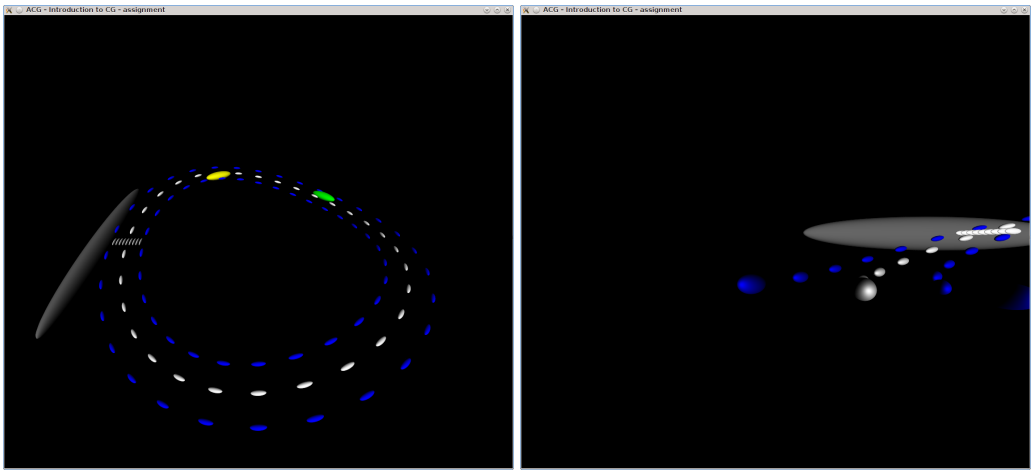


Figure 1: (a) The track seen with a FoV of 90° from the specified camera position. (b) Driving along the track.