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

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Basic Techniques in Computer Graphics

Assignment 8

Date Published: December 3rd 2019, Date Due: December 10th 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 December 10th 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 Shadow Maps [8 Points]

Another method to render shadows, besides using shadow volumes, is the Shadow Maps algorithm presented in the lecture. Its basic idea is to render the scene from the perspective of the light source and store the depth values observed by the light source into a special texture, the so-called shadow map. When rendering the scene from the view of the camera, the shadow map is used to determine which points seen from the camera are also seen from the light source and are therefore lit.

(a) [2 Points]

Let M_c be the view matrix of the camera and let M_l be the view matrix of the light source (with respect to which the shadow map has been rendered). Given an arbitrary point $p \in \mathbb{R}^3$ in the local coordinate system of the camera, specify the formula that transforms it into its representation p' in the coordinate system of the light source.

(b) [2 Points]

Let $f_{\text{SM}} : \mathbb{R}^3 \to \mathbb{R}$ be the function that takes a point q in the coordinate system of the light source and returns the depth value stored in the shadow map at the position onto which q projects on the image plane of the light source. Specify the condition (in terms of M_c, M_l, and f_{SM}) that answers whether a point p (represented in the local coordinate system of the camera) lies in the shadow according to the Shadow Maps technique.

(c) [4 Point]

What are the two types of aliasing that can occur when using shadow maps and which one can be fixed by using Perspective Shadow Maps? Explain your answer in **at most 5 sentences in your own words**.

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

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Exercise 2 Anti-Aliasing [12 Points]

(a) [2 Points per column]

Aliasing in the context of rendering can have the following causes:

- Texture alias: Introduced by point sampling textures which leads to jagged edges in the case of magnification and moirée pattern in the case of minification.
- Geometry alias: Caused by the binary decision for each pixel whether a pixel is rasterized or not. Noticeable on geometry silhouettes.
- Shader alias: Produced by the fragment shader. An example would be a fragment shader that outputs a black and white checkerboard based on the world coordinates.

There exist several methods to reduce the effect of those artifacts such as Mipmapping/linear interpolation, Multisample Anti-Aliasing(MSAA), Post-processing Anti-Aliasing(for example FXAA) and Supersample Anti-Aliasing(SSAA). Indicate which method helps against which type of alias by filling the table below with either check marks(the method reduces the type of alias) or crosses(the method does not help with the type of alias).

	Mipmapping	MSAA	FXAA	SSAA
texture				
geometric				
shader				

(b) [4 Point]

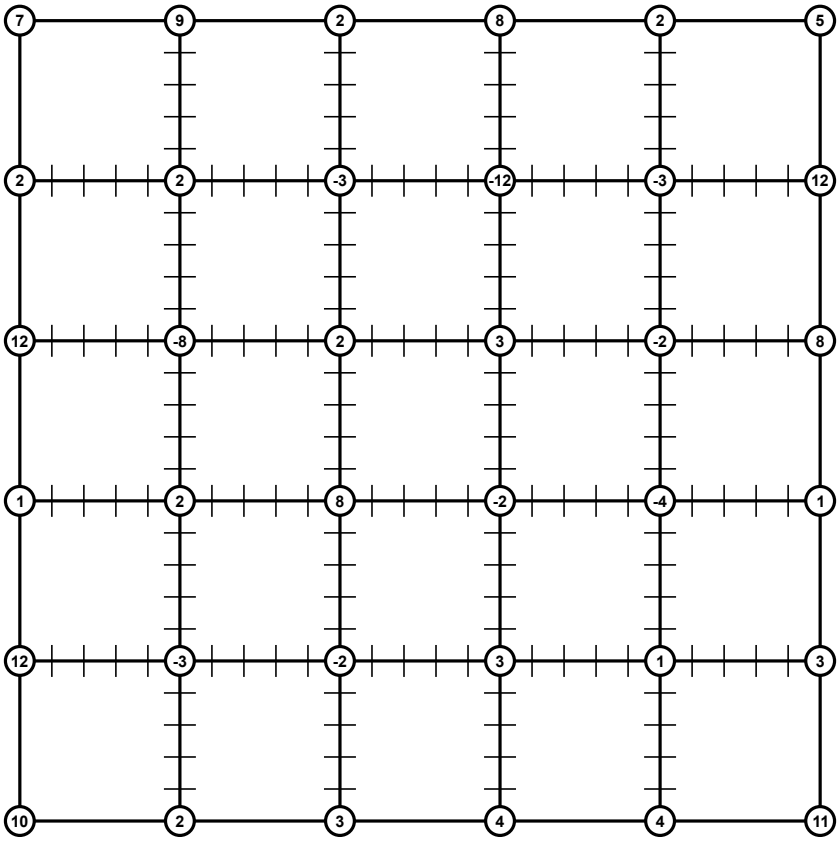
Shortly explain, **in your own words**, what *Multisampling* and *Supersampling* techniques do. Discuss the difference between both techniques.

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

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Exercise 3 Indirect Rendering of Implicit and Volumetric Geometry [12 Points]

Instead of rendering implicit or volumetric functions directly using ray traversal techniques, one can also take an indirect route and first extract a polygonal mesh representation that is then rendered using the standard triangle rasterization pipeline. For this purpose the Marching Cubes (MC) algorithm has been presented. Given an implicit volumetric representation in form of discrete sample values at grid points, it constructs a polygonal representation of the zero-level surface (or any other iso-level).



(a) Surface Extraction [8 Points]

Execute the Marching Squares algorithm (the 2D equivalent of MC) on the discrete implicit data given above to extract a zero-level representation, i.e. draw the resulting set of line segments (instead of polygons in 3D) into the raster. Whenever there is an ambiguity, assume the square center has a positive value. Make sure to get the geometry right, not only the topology, i.e. position the samples at the correctly interpolated positions.

(b) Topological Equivalence [4 Points]

Is the zero level representation which is extracted from the Marching Squares algorithm always topologically equivalent to the original implicitly defined object? If it is, explain your answer. If not, give a counter example.

Exercise 4 Programming [8 Points]

In computer graphics *Environment Mapping* is a common way to efficiently simulate reflections of the environment of an object on its reflective surface. In practice, *Cube-* and *Sphere-Mapping* are the methods of choice for many applications. In both techniques textures of the environment are precomputed and the reflected color is fetched according to the respective viewing ray reflected on the object's surface. Usually, a *Sphere Map* is a photograph of a reflective sphere that mirrors *almost* the entire surrounding environment. In contrast, a *Cube Map* consists of six images. Each image shows the surrounding scene as seen from the center of a cube in the direction of each of the cube's facets. In this week's practical exercise your task is to implement *Sphere-Mapping*, while a *Cube Map* is already implemented for comparison.

In OpenGL, sphere maps are treated as simple 2D textures. In order to get the correct texture coordinate for each fragment, you will have to create a mapping from the reflected viewing direction vector to the two-dimensional texture coordinates by yourself.

In the provided code you will find shaders that implement basic texturing and lighting using the Phong-Blinn model with Phong shading. The meshes are already equipped with lighting that is displayed in the initial scene. This we will refer to as the *original lighting* in the following. The scene also contains a *sky box* which is basically just a cube that encloses the visible scene onto which the respective cube map is projected. This is a very handy visual tool to verify whether the reflections look correct.

The reflection vectors for the texture look-up are in *Global Space*, which is already correct, so you do not have to do any transformations before computing the texture coordinates. Different from the previous assignments, this time the draw callback function has four parameters:

meshNumber (toggled with key 'm') If this variable is `true` the bunny mesh should be displayed in the scene. If `false` the bunny should be replaced by the sphere.

cubeMapping (toggled with key 'c') If this variable is `true` use the cube map to compute the environment map. If the value is `false` use the sphere map.

debugTexture (toggled with key 'x') If this variable is `true` use the debug textures for the currently active environment map. These texture have the appendix "`Debug`" or "`x`" in its file name. If the value is `false` use the normal texture files.

mixEnvMapWithTexture (toggled with key 'e') If this variable is `true` use texture blending to display the object's original lighting blended with 0.1 times the values from the currently active environment map. If `false` only show the environment map.

Furthermore, you can switch between rotation of the object itself and rotating the camera around the object via pressing key '`r`'. The necessary transformations are already implemented in the given code. In this exercise you only have to edit the fragment shader *envmap.fsh*. Now, proceed as follows:

(a) [6 Points]

Implement the sphere mapping technique inside the fragment shader. Verify your results using the debug textures! The mapping of the reflected vectors to texture coordinates has to be done according to what was discussed in the lecture. Note that in the method from the lecture the computed values of the texture coordinates are in the range $[-1, 1]$. In order to comply with OpenGL's texture coordinate format, they will have to be scaled to the range $[0, 1]$.

(b) [2 Point]

Integrate texture blending between the respective mesh's original lighting and the environment map. Multiply a factor of 0.1 to the environment map.

If everything is implemented correctly, your scene should be similar to those shown in Fig. 1 for the different parameters.

Don't worry if the resulting images generated with the two different methods are not *exactly* identical!

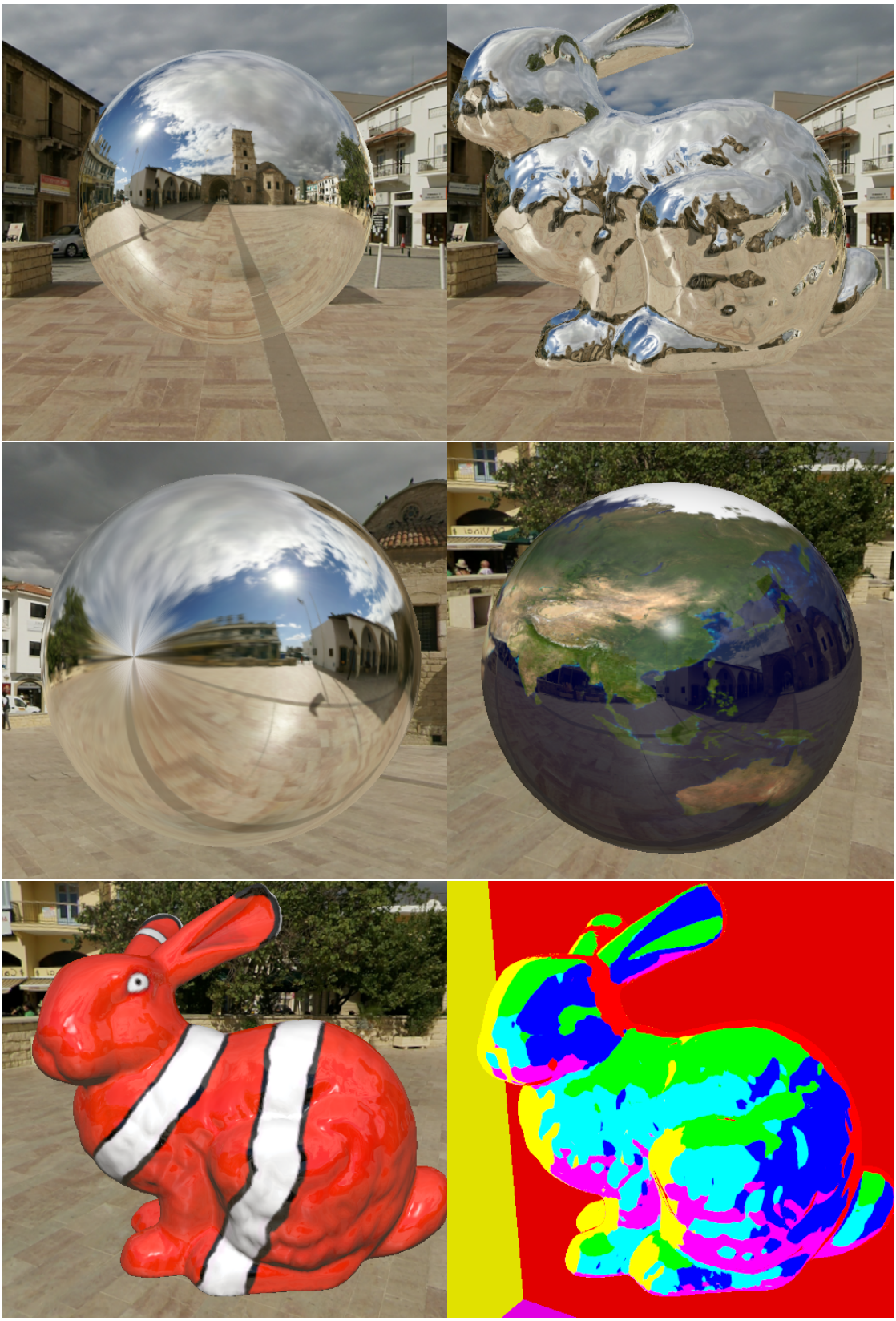


Figure 1: Screenshots showing the scene with different parameter values.