Assignment 8

Basic Techniques in Computer Graphics
WS 2022/2023
December 20, 2022

Frederik Muthers	412831
Tobias Broeckmann	378764
Debabrata Ghosh	441275
Martin Gäbele	380434

Exercise 1 Triangle Texturing

(a) 3D-Barycentric Coordinates P is given in Camera Space as $\mathrm{P} = \begin{pmatrix} 3.8 \\ 3 \\ 2.5 \end{pmatrix}$. Triangle T_1 is defined by $\mathrm{A} = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \mathrm{B} = \begin{pmatrix} 5 \\ 0 \\ 2 \end{pmatrix}$ and $\mathrm{C} = \begin{pmatrix} 3 \\ 7 \\ 4 \end{pmatrix}$ in Camera Space. To compute the barycentric coordinates for our point P relative to T_1, we consider α, β such that $\alpha\mathrm{A} + \beta\mathrm{B} + (1-\alpha-\beta)\mathrm{C} = \mathrm{P}$ (with $\gamma = 1-\alpha-\beta$) which gives us:

$$
\begin{aligned}
\mathrm{P} &= \alpha\mathrm{A} + \beta\mathrm{B} + (1-\alpha-\beta)\mathrm{C} \\
&= \alpha \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix} + \beta \begin{pmatrix} 5 \\ 0 \\ 2 \end{pmatrix} + (1-\alpha-\beta) \begin{pmatrix} 3 \\ 7 \\ 4 \end{pmatrix} \\
&= \begin{pmatrix} \alpha + 5\beta + 3 - 3\alpha - 3\beta \\ 2\alpha + 7 - 7\alpha - 7\beta \\ -\alpha + 2\beta + 4 - 4\alpha - 4\beta \end{pmatrix} \\
&= \begin{pmatrix} -2\alpha + 2\beta + 3 \\ -5\alpha - 7\beta + 7 \\ -5\alpha - 2\beta + 4 \end{pmatrix} \\
&= \begin{pmatrix} 3.8 \\ 3 \\ 2.5 \end{pmatrix}
\end{aligned}
$$

Therefore we have,

$$-2\alpha + 2\beta + 3 = 3.8 \implies -2\alpha + 2\beta = 0.8 \tag{1}$$

$$-5\alpha - 7\beta + 7 = 3 \implies -5\alpha - 7\beta = -4 \tag{2}$$

$$-5\alpha - 2\beta + 4 = 2.5 \implies -5\alpha - 2\beta = -1.5 \tag{3}$$

Subtracting (2) from equation (3) gives us $5\beta = 2.5 \implies \beta = 0.5$. We now have, $-2\alpha + 1 = 0.8 \implies \alpha = 0.1$. So the barycentric coordinates for our point P relative to T_1 is $(0.1, 0.5, 0.4)$.

(b) UV Coordinates Triangle T_2 is defined by A'$= \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix}$, B'$= \begin{pmatrix} 0.8 \\ 0.2 \end{pmatrix}$, C'$= \begin{pmatrix} 0.2 \\ 0.8 \end{pmatrix}$ in UV space. Given T_2, using the barycentric coordinates, we can calculate the corresponding point P' in UV space as:

$$
\begin{aligned}
&\alpha\mathrm{A'} + \beta\mathrm{B'} + \gamma\mathrm{C'} \\
&= 0.1 \times \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix} + 0.5 \times \begin{pmatrix} 0.8 \\ 0.2 \end{pmatrix} + 0.4 \times \begin{pmatrix} 0.2 \\ 0.8 \end{pmatrix}
\end{aligned}
$$

$$= \begin{pmatrix} 0.02 + 0.4 + 0.08 \\ 0.02 + 0.1 + 0.32 \end{pmatrix}$$
$$= \begin{pmatrix} 0.5 \\ 0.44 \end{pmatrix}$$

(c) Distortion The order of operations does matter when computing the UV Coordinate of a point.

The mapping $T : (u, v) \to (x, y, z)$ specifies an affine map from the texture space triangle to the object space triangle. Now, for the reverse mapping, one way we can do this is by projecting the triangle onto the image space to get a 2D triangle and then performing the barycentric interpolation. But this leads to a flaw that stems from the property of projective transformation. We know that projective mapping is invariant in terms of cross-ratio but the aspect ratio doesn't stay the same which results in changed barycentric coordinates than in the image plane. So this method doesn't take the case of perspective foreshortening into account.

On the other hand, if we first perform barycentric interpolation and then project the triangle onto the image plane, we avoid the possibility of changed barycentric coordinates and can interpolate any point from the 3D camera space to UV space using the same values of barycentric coordinates keeping the aspect ratio intact.

Exercise 2 Anti-Aliasing

(a)

	Mipmapping	MSAA	FXAA	SSAA
texture	✓	×	✓	✓
geometric	×	✓	✓	✓
shader	×	×	✓	×

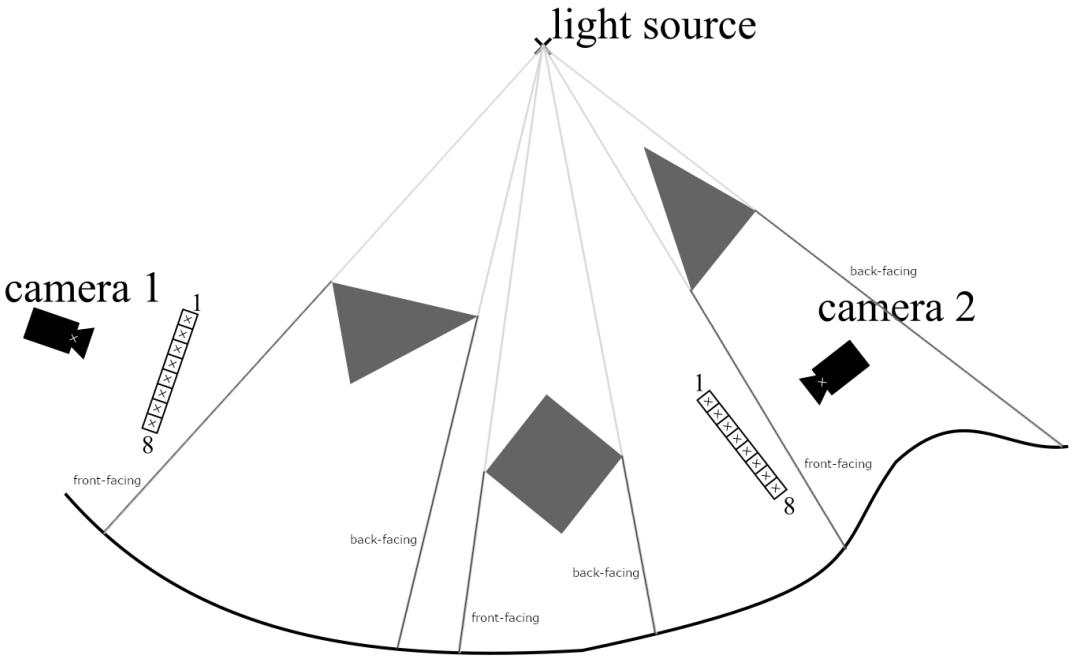
(b) For tackling aliasing effects on the boundary of polygons we need to use geometric anti-aliasing techniques as we can't pre-compute the texture depending on the viewing direction or the scene behind one of the polygon boundaries. Multisampling and Supersampling techniques help us with such geometric anti-aliasing. Depending on how the rasterization process converts a continuous polygon into a discrete set of pixels covered by the projection footprint of the polygon, we can see jagged edges resulting in geometric alias. We can solve this via Supersampling or Multi-sampling in a full-scene anti-aliasing method.

In supersampling we use a higher resolution to render the scene and then when the full scene is rendered, the image is down-sampled back to the normal resolution. This higher resolution is used to get rid of the jagged edges. So, if we render an image with N-times higher resolution we'll have to check N times per pixel whether the position is covered by our polygon to be rendered and N-times more computation of the color and memory needed to store those N color values and depth to compute the local average. Though supersampling is a general method and the same idea can be used for texture anti-aliasing, the computational overhead often makes it infeasible.

Multisampling is the less-computationally expensive version of the same anti-aliasing method. Assuming within a pixel the color of the polygon we are rendering doesn't change too much, we can compute the color only once for each pixel. But if we apply a texture on the polygon, the assumption about the color might not hold. In that scenario, we can use Mipmapping to get color values from the one color we computed. So the memory overhead of N color and depth still remains like supersampling. So the main difference between multisampling and supersampling can be seen in terms of computational cost (of calculating the color value).

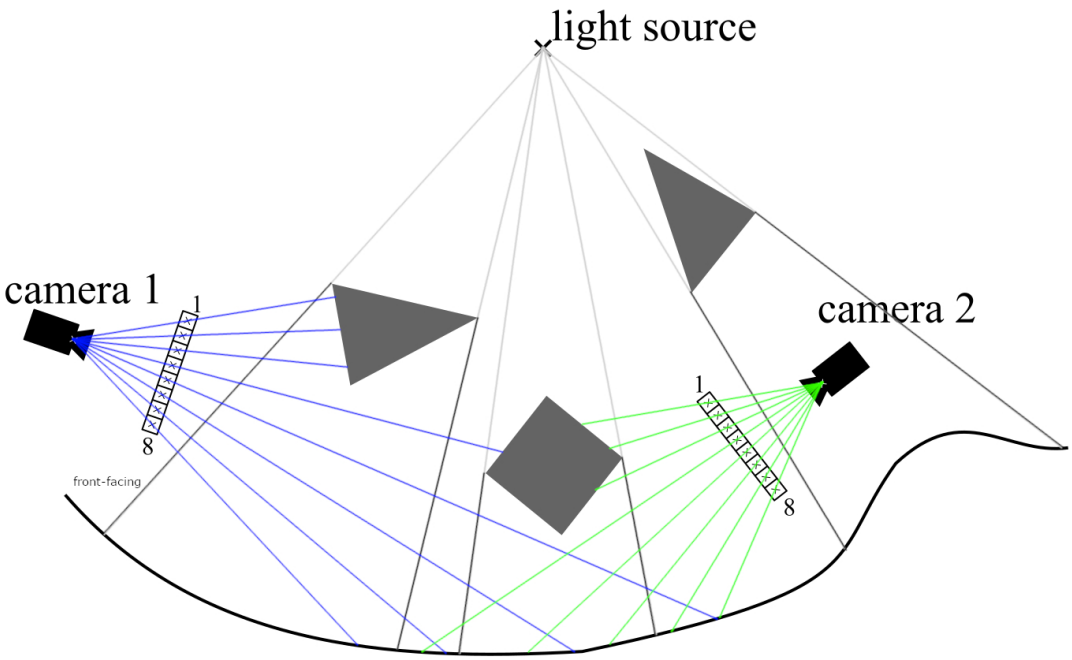
Exercise 3 Shadow Volumes

(a) Note: The outer edges of the occluders with the drawn traces are included in the shadow volumes (for example the quadratic shaped occluders volume is included in the shadow volume generated by it)



(b) A ray-casting based shadow volume algorithm casts a ray from the camera to the point that the color should be calculated of. Then it increments (/decrements) a variable for each time the ray intersects a front-facing (/back-facing) edge of the shadow volume. If that variable is 0 then the point is illuminated from the light-source. A Stencil-Buffer-based shadow volume algorithm computes the same value by first rendering the scene, thus creating a z-buffer. Then each shadow volume is rendered with the goal of creating a stencil-buffer, which holds the value of the same variable the ray-tracing based algorithms calculate, for each pixel. To do so each shadow polygon first runs the z-buffer test to determine whether or not the polygon is visible and if it is visible we increment the stencil buffer in case we have a front-facing polygon and decrement it if we have a back-facing polygon. This effectively computes the same value for each pixel that the shadow-polygon is affecting as the ray-tracing based algorithm.

(c)



Pixel	Stencil-Buffer Camera 1	Stencil-Buffer Camera 2
1	1	-1
2	1	-1
3	1	0
4	0	-1
5	0	0
6	1	0
7	0	-1
8	1	-1

(d) The stencil buffer for each pixel should be incremented by one. This is caused by the camera being placed inside one of the shadow volumes. We can fix this problem by evaluating the variable based on the intersections of the ray and shadow volumes "behind" the point of interest. That means that instead of "counting" the intersections of the ray with the shadow volumes between the camera and point of interest we "count" the intersections of the ray between the point of interest and inifinity (we can crop this so we don't actually need to compute anything in regards to infinity).

Exercise 4 Shadow Maps

(a) We know that $M_c p_w = p$ and $M_l p_w = p'$. Thus we can conclude

$$M_l M_c^{-1} p = p'$$

Thus the matrix that transforms p to p' is the matrix $M_l M_c^{-1}$.

(b) Let p be the point of interesting with respect to the camera coordinates. We first need to transform this point into the point $p' = \begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix}$ as in (a). Then we evaluate $f_{SM}(p')$. If the "depth" of p' is bigger than the stored value, p' lies in the shadow. So in total we evaluate

$$v = f_{SM}(M_l M_c^{-1} p) - (M_l M_c^{-1} p)_z$$

If $v \geq 0$ the point p does not lie in the shadow, otherwise it does.

(c) The two types of aliasing that can occur when using shadow maps are perspective aliasing and projective aliasing. Projective aliasing stems from the angle in which the lightsource hits the scene (specifically the angle between the normal vector and the light-direction). The bigger the angle between the normal-vector of the scene and the light-source direction gets, the worse the aliasing becomes. Perspective aliasing stems from the ratio of the distances from the camera to the scene and the light-source to the scene, specifically it gets worse the closer the camera is to the scene and the further the light source is located from the scene. Perspective Shadow Maps can be used to fix the perspective aliasing. Perspective Shadow Maps first frustum-transform the image and then create the shadow map of the transformed (/distorted) scene. This ensures that objects closer to the camera correspond to more pixels in the resolution of the shadow-map thus counter-acting the perspective aliasing. By taking the frustum-transform we essentially move the projection center (camera) further away from the scene, which also explains that effect.