UU Graphics

academic year 2013/14 – 4th period

Theoretical Assignment #4: Rasterization – Homogeneous Coordinates & Perspective

- Solutions -

June 03 2014

Assignment #1: *Homogenous coordinates – the basics, and some cool apps*

CORE
core topics
important

a) Consider the following matrix and vector; they operate on / represent 3D points embedded in the 4D projective space (using homogenous coordinates). Perform the matrix-vector multiplication manually to see what happens!

$$\begin{bmatrix} \lambda_1 & 0 & 0 & t_x \\ 0 & \lambda_2 & 0 & t_y \\ 0 & 0 & \lambda_3 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Solution

$$\begin{bmatrix} \lambda_1 & 0 & 0 & t_x \\ 0 & \lambda_2 & 0 & t_y \\ 0 & 0 & \lambda_3 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \lambda_1 x + w t_x \\ \lambda_2 y + w t_y \\ \lambda_3 z + w t_z \\ w \end{bmatrix}$$

It scales the vector by factors $\lambda_1, \dots, \lambda_3$ in x,y,z-direction and adds $w\left(t_x, t_y, t_z\right)^T$. If we input $w = 1$, as usual, this performs a scaling and translation by $\left(t_x, t_y, t_z\right)^T$. We will need this for the following assignments.

b) Now for something more complicated. Find a matrix in homogenous coordinates that zooms into a 2D image (i.e., this is $\mathbb{R}^2$ and the homogenous coordinates live in $\mathbb{R}^3$) with zoom factor $\lambda \in \mathbb{R}$ and origin $\mathbf{p} \in \mathbb{R}^2$.

Solution:

We move the origin to **p**, then scale, and move the origin back:

$$\begin{bmatrix} 1 & 0 & p_1 \\ 0 & 1 & p_2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -p_1 \\ 0 & 1 & -p_2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \lambda & 0 & p_1 - \lambda p_1 \\ 0 & \lambda & p_2 - \lambda p_2 \\ 0 & 0 & 1 \end{bmatrix}$$

c) Imagine we want to implement a 2D vector drawing program, such as Powerpoint or Inkscape. We have a world coordinate system, in which have located an A4 page (with its upper left corner as the origin of the coordinate system, the positive x-axis points to the right, and the y-axis down, all units are measured in cm, and an A4 page is 21×29.7 cm in size, width × height. You also have a drawing app, running on a 1920×1080 pixel display (full-screen, landscape mode, i.e., taller side is left-right, square pixels); the origin of the coordinate system is also in the upper left corner.

Determine a (homogenous) transformation matrix that maps the A4 page to the screen, such that the height of the paper fills the whole display height, and the virtual paper on the screen is centered on the screen.

Solution:

We need to scale the paper coordinates such that 29.7 units $\widehat{=}$ 1080 units. This means, the scaling matrix should be:

$$\begin{pmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ with } \lambda = \frac{1080}{29.7} = 36.\overline{36}$$

We now need to determine the translation that is required. In y-direction, this is easy: The point $y_{paper} = 0$ should map to $y_{screen} = 0$, so no translation is required. In x-direction, we want that the middle of the paper (10.5 units) is mapped to the middle of the screen (960 units). Hence, we substract 10.5 units before scaling, and add back 960 after scaling (to shift the origin, as in assignment (b) above). We get:

$$\mathbf{V} = \begin{bmatrix} 1 & 0 & 960 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{1080}{29.7} & 0 & 0 \\ 0 & \frac{1080}{29.7} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -10.5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

d) Now our drawing program needs an additional feature! Implement a zoom bar – that means, compute an additional homogeneous matrix, to be multiplied with the first one, such that you can zoom in and out of the document. Important: keep the center of the virtual document fixed to the center of the screen! The factor λ = 100% = 1.0 should yield the result of (c).

Solution:

This is easy – we can just recycle the result from (b). We can choose between post-multiplying or pre-multiplying the scaling matrix with the result from (c), i.e., perform the zoom in world coordinates, multiplying from the left, i.e., from the input side, or performing the zoom in screen coordinates, multiplying from the right.

First, this is the zooming matrix in screen coordinates:

$$\mathbf{Z}_{lambda} = \begin{bmatrix} 1 & 0 & \frac{1920}{2} \\ 0 & 1 & \frac{1080}{2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -\frac{1920}{2} \\ 0 & 1 & -\frac{1080}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

We then form $\mathbf{Z}_{lambda} \cdot \mathbf{V}$. If we operate in world space, we have to use paper units (29.7/2, 21/2) and multiply the matrix with $\mathbf{V}$ from the right-hand side.

e) Then, you have to adapt your software – a new, totally cool mobile phone came out, and everybody needs to port their apps. The new phone has a 4K display (3840 × 2160, portrait mode, i.e., the taller side is up). However, due to a misunderstanding in the engineering department of the phone company, the API for the display was screwed up and it has the origin of the coordinate system in the lower left corner, with the positive x- and y-axis pointing right and upwards, respectively. What do you need to change to your solution in (c) to support the cool new phone?

Solution:

We take the solution from (c) and change the scale factors. In addition, we flip around the y-axis to account for the different orientation of the coordinate system; this is done by using a negative scale factor in the centered scaling (marked in red below):

$$\mathbf{V}_{4\mathrm{K}} = \begin{bmatrix} 1 & 0 & \frac{3840}{2} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{2160}{29.7} & 0 & 0 \\ 0 & -\frac{2160}{29.7} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -10.5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

f) To make the story complete – the new phone permits users to turn around the screen from portrait to landscape format. The way the user holds the phone is indicated by a flag; the screen coordinate system does not change. What needs to be adapted to get the same results in landscape mode?

Solution:

We can just keep the matrix $\mathbf{V}_{4K}$ above and post-multiply (multiply from the left-hand-side) with a flipping matrix, if needed. This matrix has to rotate by 90°, and change the scale factor from $\frac{2160}{29.7}$ to $\frac{3840}{29.7}$, to make again the paper fit to screen. In other words, we have to scale all coordiantes by

$$\frac{3840}{29.7} : \frac{2160}{29.7} = \frac{3840}{2160}$$

The result that we get is:

$$\mathbf{R}_{90°} = \begin{pmatrix} 0 & \frac{3840}{2160} & 0 \\ -\frac{3840}{2160} & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \text{ and}$$

$$\mathbf{R}_{270^\circ} = \begin{pmatrix} 0 & -\dfrac{3840}{2160} & 0 \\ \dfrac{3840}{2160} & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

respectively, for rotating by 90° or -90° in clockwise direction. The result is then $\mathbf{R}_{90^\circ} \cdot \mathbf{V}_{4K}$, or $\mathbf{R}_{270^\circ} \cdot \mathbf{V}_{4K}$.

Assignment #2: *Perspective*

CORE
core topics
important

a) Recap: Derive the perspective projection matrix (slides 47/48 of the lecture). Explain the main steps of the derivation.

Solution: See slides 47/48

b) We are given the following specification from the art director (the art department does not bother with orthogonal frames & stuff):

- The camera should be placed at point $\mathbf{p} = (1,2,3)$!
- It is looking at the origin! ($\mathbf{lookAt} = (0,0,0)$)
- The y-axis should be depicted as upward! ($\mathbf{up} = (0,1,0)$)

Compute an orthogonal camera coordinate frame (with the z-axis being the viewing direction)!

Solution:

The normalized view vector (w-axis of the camera coordinate system $\widehat{=}$ z-axis of the original perspective matrix) is given by

$$\mathbf{w} = \frac{\mathbf{lookAt} - \mathbf{p}}{\|\mathbf{lookAt} - \mathbf{p}\|}.$$

Next, we have to create an v-vector (y-coordinate of the camera coordinate system) that is orthogonal to **w**. This can be done using Gram-Schmidt, but it is easier to first compute the u-vector (x-coordinate) using a cross product, and then creating the v-vector using a second cross product:

$$\mathbf{u} = \mathbf{up} \times \mathbf{w}$$

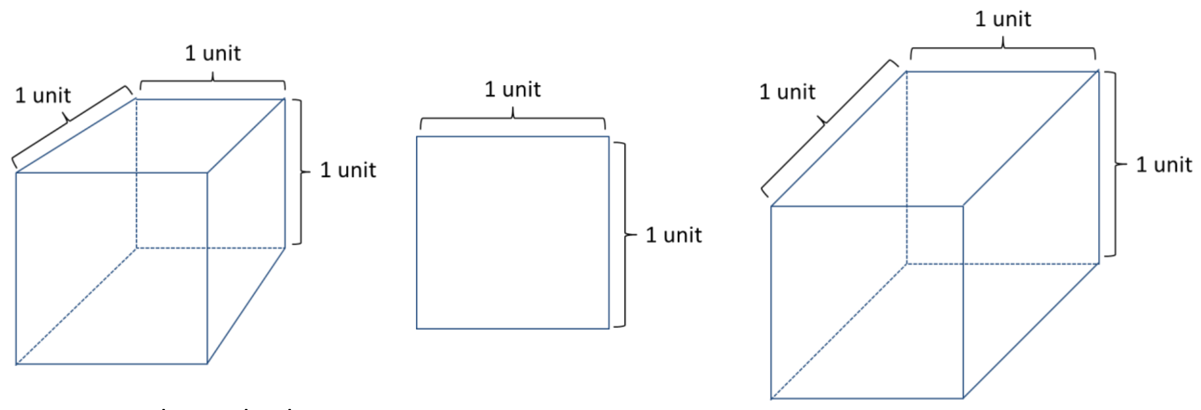
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

c) For CAD, it is often useful not to use perspective projection but orthogonal or parallel projection. Create homogenous projection matrices that do the following:

- Project the scene orthogonally in x-direction. This means, only the y- and z-axis of the world coordinates are maintained (as x- and y-axis of the screen, respectively), and the x-coordinate of the world coordinate system is just ignored (this yields a projection in z-direction without perspective, called orthogonal projection).

- Project the scene using parallel projection. Here, x- and y-axis are maintained unchanged, and the changes in the z-axis translate points diagonally. Setup a homogenous transformation matrix for this case, too.

The image below shows the different cases:



perspective projection, as discussed in the lecture

orthogonal projection

parallel projection

Solution

Orthogonal projection in x-direction: We just output y,z as x,y-coordinates, ignoring the rest:

$$\mathbf{P}_{ortho\_x} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

My matrix outputs the x-coordinate as z-value of the screen space; this might be useful for visibility determination, but this is optional in this formulation of the assignment.

Parallel projection in z-direction: We just add the z-coordinates to x,y and output this as x,y-coordinates:

$$\mathbf{P}_{parallel\_z} = \begin{pmatrix} 1 & 0 & \frac{1}{2}\sqrt{2} & 0 \\ 0 & 1 & \frac{1}{2}\sqrt{2} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Assignment #3: *Raytracing vs. rasterization – the complexity battle*

CORE

core topics
important

Consider the standard z-buffer algorithm and the naïve raytracing algorithm (neither of them would use any smart data structures for speedup; raytracing performs brute-force search of the randomly ordered triangles). The scene consists of *n* triangles, which cover *k* fragments (i.e., we count potentially occluded pixels again and again, if covered repeatedly, in order to determine *k*). The display overall consists of *m* pixels.

What is the asymptotic complexity of z-buffer rendering vs. raytracing, in big-O notation, dependent on these parameters?

Solution

As discussed in the lecture, complexity of z-Buffering is $O(n+k+m)$ for transforming n triangles, creating the k fragments of all of the triangles together, and clearing the color and z-buffers once.

Raytracing in its naïve form needs to test all n triangles for intersection for all of the pixels, leading to a complexity of $O(m \cdot n)$. z-Buffering has the same complexity in the worst case, if all triangles cover the complete screen (then, $k = mn$).