Camera matrix from basic principles

a reason to drag 4D space into 3D graphics

Anders Leino
`http://aleino.net`

October 5, 2019

1 Scope and intended audience

The intention is that this document should be readable by someone who knows a bit of linear algebra, but may not have much practice in working with projective space. We derive a slightly generalized *pinhole camera matrix* [1]. The end formulation is general in the sense that the sides of the frustum may have different slopes. This is general enough to be used for either eye in stereo rendering, for example. Finally, we also compute the inverse camera matrix.

Both the forward and inverse camera transformations we end up with are represented by 4×4 matrices. This is the canonical way to represent camera transformations in 3D graphics. Since 4×4 matrices can represent any transformation that 3×3 matrices can represent, and more, they've become the canonical form for all transformations in some circles. Representing 3-dimensional transformations using 4×4 matrices is an unfamiliar thing to do, so we'll show exactly how that works. Furthermore, we'll also provide the motivation for it – we'll show that we can't express the camera transformation as a 3×3 matrix, which will motivate the introduction of a new scheme for representing transformations.

A bit more concretely, we'll be expressing 3-dimensional transformation f_A using a 4×4 matrix, A. We'll choose exactly what f_A is later on. We

[1] This is sometimes called a "projection matrix". The only problem with this name is that the matrix in question does not represent a projection in any sense of the word.

will show that our choice or representation will have the property that "f_A followed by f_B" is the same as f_{BA}, where the BA is a regular matrix product. This property is important because it means that we can iterate towards our camera transformation in small steps – in the end we'll be able to obtain the composite just by multiplying together the matrices of each step in reverse order. Similarly, to help us derive the inverse f_A^{-1}, we show that this is just equal to $f_{A^{-1}}$ where A^{-1} is the regular matrix inverse. These two facts will let us forget about f_A and just deal with A for the most part, as is the practice in 3D graphics.

2 The problem

We have vertices in 3-dimensional camera space, and we want to transform them, in the way a pinhole camera would, into another 3-dimensional space called NDC space [2].

NDC space acts as an interface with the rasterization stage, because 3D graphics APIs define a certain set called the *clipping volume* in NDC space. For the sake of concreteness, we'll follow Direct3D and define the clipping volume as the following box

$$\begin{aligned} -1 \leq\, & x \leq 1 \\ -1 \leq\, & y \leq 1 \\ 0 \leq\, & z \leq 1 \end{aligned}$$

The clipping volume is shown in figure 2. Since it has the shape of a box, the clipping volume transforms via a simple scaling exactly onto the entire screen viewport and depth buffer, for purposes of rasterization. The corners of this box will transform to the corners of the screen viewport, and the ends of the depth buffer. So exactly the stuff we should be able to see must get transformed into the clipping volume in NDC space. In camera space, the stuff we can see is inside a shape called the *camera frustum*, shown in figure 1.

Whatever transformation we come up with, it must transform the camera frustum shown in figure 1 onto the clipping volume shown in figure 2. In the next section, we'll show that such a transform can't be represented by 3×3 matrices.

[2]NDC stands for "Normalized Device Coordinates".

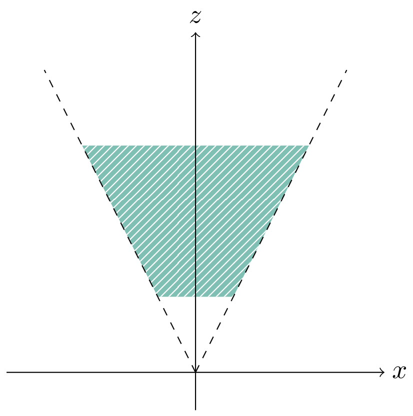


Figure 1: Camera frustum, camera space

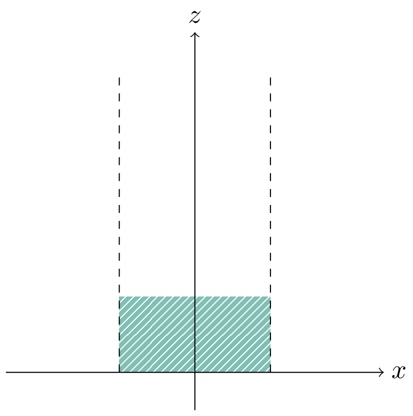


Figure 2: Clipping volume, NDC space

3 3×3 matrices are out

A basic fact about matrices is that they must transform planes trough the origin onto (in general different) planes trough the origin. Now take a look at the planes (dashed lines) in figure 1. They go trough the origin, but we want them to transform to the corresponding planes in figure 2, and those planes *do not go trough the origin*.

So our camera transformation can't be done with 3×3 matrices. But if not 3×3 matrices, then what? In the next section, I'll make a proposal for a different way to represent transformations of 3-dimensional space, which will let us represent our camera transformation.

4 A different way to represent 3D transformations

So we know that we can't use 3×3 matrices at this point. Let's look for another simple representation. Preferably we want a representation of a strictly larger set of transformations. Note, however, that we don't want to go too general – we want something that will work well on a computer.

First I'll define two helper functions

$$\pi\left(\begin{bmatrix}x\\y\\z\\w\end{bmatrix}\right) = \frac{1}{w}\begin{bmatrix}x\\y\\z\end{bmatrix}$$

$$\varphi\left(\begin{bmatrix}x\\y\\z\end{bmatrix}\right) = \begin{bmatrix}x\\y\\z\\1\end{bmatrix}$$

Using these, we can write our proposed transformation as [3] [4]

$$f_A(v) = \pi(A\varphi(v))$$

In the above, A is any 4×4 matrix that defines the transformation f_A. What happens if we compose two such transformations in sequence? That is, first we first transform a vector by f_A and transform the result by f_B. It's an

[3]If you're interested, read up on *projective space* and *projective transformations*. The general subject is *projective geometry*.

[4]If we want to be super careful, it should be pointed out that $\pi(v)$ is only defined when $v_w \neq 0$, which means that $f_A(v)$ is only defined when $(A\varphi(v))_w \neq 0$. In the rest of the text we'd have to add a lot of warnings like this about all uses of f_A. If you wish, you can add in those warnings yourself. However, this is not some theoretical consideration – nothing is stopping a vertex shader from returning a vertex with a w-coordinate that's zero. There is actually a right way to handle this corner case. It becomes apparent if you skip dividing by $w = 0$ and instead re-interpret your vector. You keep the regular xyz components, but now they mean something different – they now indicate an infinitely long vector pointing out into some direction. If one of your triangles has a vertex which happens to be this kind of corner case, your triangle in screen space should be rasterized like an infinitely long paralellogram (a slanted rectangle) rather than a triangle. This is completely doable after clipping. The infinite sides of the paralellogram are parallel to the xyz-components of your corner-case vertex. You can think trough the cases, where you have two or three corner-case vertices in a triangle, for yourself. We will skip thinking about this – at any rate these are cases for the 3D graphics API to handle, not the application. It's actually a corner case that can be handled beautifully – sadly, our chosen representation f_A cannot represent these exotic cases. That's a deliberate trade-off so that we'll be able to think about regular 3D space without infinitely long vectors. For our use-case, we only need to think about transforming points inside the camera frustum, and it'll be easy to check later on that we don't get in trouble there.

exercise for the reader to check that $\pi(B\varphi(\pi(v))) = \pi(Bv)$ for any v with $v_w \neq 0$. Using this we obtain

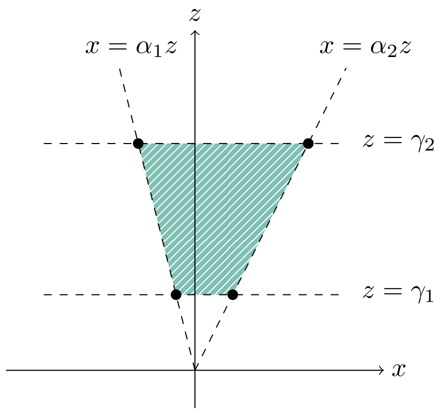
$$f_B(f_A(v)) = \pi(B\varphi(\pi(A\varphi(v)))) = \pi(BA\varphi(v)) = f_{BA}(v)$$

In other words, f_A followed by f_B is the same as f_{BA}. This formula is important, because it means we will can break our problem into steps. When we're done, we can obtain a compact formulation by multiplying together the matrices corresponding to the steps, in reverse order. In the end we will have a single 4×4 matrix that defines our transformation!

Now we have a candidate to replace 3×3 matrix transformations, and we know how to compose them. However, we still haven't shown that this candidate can do the job that 3×3 matrices could not. We'll do that in the next section by solving the problem using the kinds of transformations introduced in this section.

5 Deriving the camera transformation

We're really supposed to transform figure 1. However, let's generalize it a bit so that both sides of the frustum may have different slopes, as seen in the image below. As said in the first section, this is the situation in stereo rendering, for example. So we begin with the frustum in camera space shown in the image below.



Next, we'll iteratively deform the camera frustum shape into the clipping volume in NDC space, keeping a record of the sequence of matrices we've used so that we can obtain the final matrix in the end.

5.1 Bend the frustum into a box

First, let's bend the frustum into a box shape, since that's the shape in the final image. To do this, we apply f_A where [5]

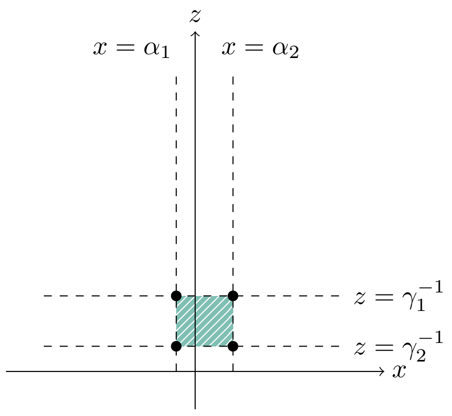
$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Let's see what our transformation is:

$$f_A\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} x/z \\ y/z \\ 1/z \end{bmatrix}$$

If you plug in the equation for the left plane, $x = \alpha_1 z$, you'll see that it transforms to the plane $x = \alpha_1$. Similarly $x = \alpha_2 z$ transforms to $x = \alpha_2$ and the planes $z = \gamma_1$ and $z = \gamma_2$ will transform to $z = \gamma_1^{-1}$ and $z = \gamma_2^{-2}$, respectively.

So f_A will transform our frustum shape to a box shape, as in the following image.



5.2 Translation

Now we want to translate our box so as to center it along the x and y direction, and also to line up its bottom with $z = 0$. This is another thing

[5]This is the one place we have to be a bit careful – f_A is not defined on the $z = 0$ plane. As said before, it's not actually such an ugly case. In any event, our camera frustum never intersects this plane, so let's ignore this issue.

you can't do with a 3×3 matrix. Again we can prove this assertion by using a basic fact about matrices – they must transform the zero vector into the zero vector, which means you can't do translations.

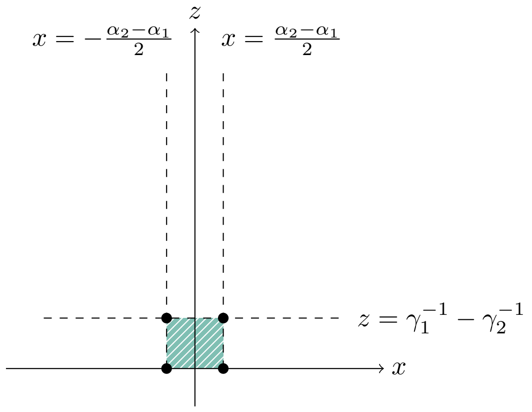
We define

$$B = \begin{pmatrix} 1 & 0 & 0 & -\frac{\alpha_1+\alpha_2}{2} \\ 0 & 1 & 0 & -\frac{\beta_1+\beta_2}{2} \\ 0 & 0 & 1 & -\gamma_2^{-1} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Our transformation will do the following

$$f_B\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} x - \frac{\alpha_1+\alpha_2}{2} \\ y - \frac{\beta_1+\beta_2}{2} \\ z - \gamma_2^{-1} \end{bmatrix}$$

So, for instance, the $x = \alpha_1$ plane transformed into $-\frac{\alpha_2-\alpha_1}{2}$ as desired. The effect of the transformation on the other planes can be checked similarly. We end up with a picture like the following.


z
$x = -\frac{\alpha_2-\alpha_1}{2}$
$x = \frac{\alpha_2-\alpha_1}{2}$
$z = \gamma_1^{-1} - \gamma_2^{-1}$
x

5.3 Scaling

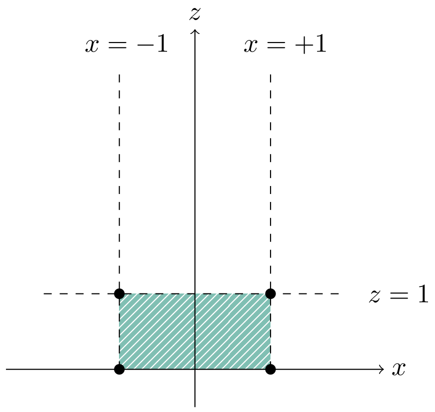
Our last image looks about right, except our region does not have the right size – so we'll scale it. This operation we could have done with 3×3 matrices, but now we need to do it in the proposed way so that it will compose nicely with our other transformations. We define

$$C = \begin{pmatrix} \frac{2}{\alpha_2-\alpha_1} & 0 & 0 & 0 \\ 0 & \frac{2}{\beta_2-\beta_1} & 0 & 0 \\ 0 & 0 & \frac{1}{\gamma_1^{-1}-\gamma_2^{-1}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Our transformation becomes

$$f_C\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} \frac{2x}{\alpha_2-\alpha_1} \\ \frac{2y}{\beta_2-\beta_1} \\ \frac{z}{\gamma_1^{-1}-\gamma_2^{-1}} \end{bmatrix}$$

So the $z = \gamma_1^{-1} - \gamma_2^{-1}$ becomes $z = 1$ when transformed, as desired if we want to fit it to the clipping volume. If you check the other planes in a similar way, you'll agree that the image ends up looking as follows.



This region is exactly the same as the clipping region in NDC space, which is what we were iterating towards.

5.4 Obtaining the final matrix

We now have three matrices A, B and C which we have chosen such that f_A followed by f_B followed by f_C implements the transformation we want. We know that this composition equals simply f_{CBA}, so let's calculate CBA:

$$CBA = \begin{pmatrix} \frac{2}{\alpha_2-\alpha_1} & 0 & 0 & 0 \\ 0 & \frac{2}{\beta_2-\beta_1} & 0 & 0 \\ 0 & 0 & \frac{1}{\gamma_1^{-1}-\gamma_2^{-1}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{\alpha_1+\alpha_2}{2} \\ 0 & 1 & 0 & -\frac{\beta_1+\beta_2}{2} \\ 0 & 0 & 1 & -\gamma_2^{-1} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{2}{\alpha_2-\alpha_1} & 0 & \frac{\alpha_1+\alpha_2}{\alpha_1-\alpha_2} & 0 \\ 0 & \frac{2}{\beta_2-\beta_1} & \frac{\beta_1+\beta_2}{\beta_1-\beta_2} & 0 \\ 0 & 0 & \frac{\gamma_2^{-1}}{\gamma_2^{-1}-\gamma_1^{-1}} & \frac{1}{\gamma_1^{-1}-\gamma_2^{-1}} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Now we have a single matrix CBA that represents a transformation f_{CBA} which will transform our frustum shape into the clipping volume shape. Transforming between those two shapes is a necessary requirement, but does the fact that we fulfill this one requirement mean we're done?

6 Are we done?

To answer this question, let's obtain the actual formulas for how a vector is transformed by f_{CBA}:

$$f_{CBA}\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} \frac{2}{\alpha_2-\alpha_1}\frac{x}{z} + \frac{\alpha_1+\alpha_2}{\alpha_1-\alpha_2} \\ \frac{2}{\beta_2-\beta_1}\frac{y}{z} + \frac{\beta_1+\beta_2}{\beta_1-\beta_2} \\ \frac{z^{-1}-\gamma_2^{-1}}{\gamma_1^{-1}-\gamma_2^{-1}} \end{bmatrix}$$

With these explicit formulas, you can do a bunch of checks to convince yourself that this transform is good. For one thing, the formulas have perspective foreshortening along the x and y directions. Note that the two constant terms in the x and y components would disappear for a symmetric frustum where $\alpha_1 = -\alpha_2$ and $\beta_1 = -\beta_2$.

So that all looks good. However, take a look at the third component, which is called the *depth* in the terminology of 3D graphics APIs. It is

$$d(z) = \frac{z^{-1} - \gamma_2^{-1}}{\gamma_1^{-1} - \gamma_2^{-1}}$$

This value transforms via a simple scaling into the depth buffer. $d(z)$ has the shape of z^{-1}, except it has been translated vertically by $-\gamma_2^{-1}$ and then scaled vertically by $\frac{1}{\gamma_1^{-1}-\gamma_2^{-1}}$, which is a positive number. This means that smaller z values get larger depth values and vice versa.

Is that a problem? No! In all modern 3D graphics APIs you can change the comparison function to pass a new fragment if their depth is *larger* than the existing depth, rather than *smaller* which is the default.

The actual important feature of $d(z)$ is that it's monotonic – not whether it increases or decreases. In fact, not only is it not an issue – it's often an advantage because the shape of d roughly cancels out the non-uniform distribution of floating point numbers so that the distribution of depth values becomes roughly uniform, as explained in "Depth Precision Visualized" by Nathan Reed [6].

Since d is monotonic, and $d(\gamma_1) = 1$ and $d(\gamma_2) = 0$, we conclude that the entire range of our camera frustum transforms exactly into the entire range of the clip volume in NDC space. We specifically enforced this in the derivation, but it's a good sanity check.

7 Who does what?

If you're familiar with 3D graphics APIs, π should be familiar to you. If your vertex shader returns v, then your 3D graphics API specifies that $\pi(v)$ will be calculated for you automatically. In fact π transforms your vertex shader output positions into NDC space! So your vertex shader only has to calculate $A\varphi(v)$. If you set things up so that your vertex shader gets $\varphi(v)$ as input, that might mean that the only thing your vertex shader does is to apply A.

8 The inverse

Can we find f_{CBA}^{-1}? This is the inverse transform, which should take you from NDC-space back to camera space.

8.1 A general formula

Let's see what we can say about f_A^{-1} in general. We'll assume A is invertible, which is in fact the case in our specific application. You can also start from a pixel coordinate and depth buffer value, and un-quantize that to get to NDC-space. [7]

f_A is a composition of three functions: first do φ, then apply A and finally do π. Suppose we start with a point (x, y, z) in NDC space. We obtain the transformation of this point under f_A^{-1} by applying π^{-1}, A^{-1} and finally φ^{-1} in that order.

[6]`https://developer.nvidia.com/content/depth-precision-visualized`

[7]You can't really "un-quantize", of course. Quantizing means throwing away information. I just mean that you'll have an approximation of the value in NDC-space, after "un-quantizing".

8.2 Applying π^{-1}

π is not an invertible function – that is to say: if we have (x, y, z), we don't know which of the points (xw, yw, zw, w) (with $w \neq 0$) transformed via π to (x, y, z). What's the solution? We continue with all of these points at once!

It turns out that we'll be able to determine w later on. So let's proceed with a whole set of candidates of the form (xw, yw, zw, w), where all we know about w is that $w \neq 0$.

8.3 Applying A^{-1}

The next step is to apply the inverse of A to each of our candidates. So now we have

$$A^{-1}\begin{bmatrix} xw \\ yw \\ zw \\ w \end{bmatrix}$$

where $w \neq 0$.

8.4 Applying φ^{-1}

Now we apply φ^{-1}. However, note that φ only outputs points with w-coordinate equal to 1. This in turn means that the only candidate we can actually send trough φ^{-1} is the one with w-coordinate equal to 1.

In other words, we have a constraint

$$\left(A^{-1}\begin{bmatrix} xw \\ yw \\ zw \\ w \end{bmatrix}\right)_w = 1$$

This constraint determines w:

$$w = \frac{1}{\left(A^{-1}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\right)_w}$$

Now that we know that w can only take the above value – we can substitute that to reduce our set of candidates to a single one:

$$\frac{A^{-1}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}{\left(A^{-1}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\right)_w}$$

Sending this candidate trough φ^{-1}, we obtain

$$\frac{\left(A^{-1}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\right)_{xyz}}{\left(A^{-1}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\right)_w}$$

That's cumbersome notation... I wonder if we have any helper functions laying around... Hey, I know, let's write the above in terms of π and φ! We get

$$\pi\left(A^{-1}\varphi\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right)\right)$$

But that's just the definition for f_A, except that we have A^{-1} in place of A. In other words, our final inverse is just

$$f_{A^{-1}}\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right)$$

So we have figured out that $f_A^{-1} = f_{A^{-1}}$. Therefore, in order to go backwards, we just need to obtain $(CBA)^{-1}$.

8.5 Obtaining $(CBA)^{-1}$

Have you ever tried to invert a 4×4 matrix by hand? There are formulas for it, but it's messy. Since we're doing some math, let's take a page from Gus Van Sant's book (or short film, rather) and apply *The Discipline of Do Easy*.

We can get the inverse in an easier way because we know more than just the final camera matrix – we know how to express it as a product of three simple matrices: CBA. A general fact about matrices is that if each of A, B and C are invertible, then so is CBA and we have a nice formula: $(CBA)^{-1} = A^{-1}B^{-1}C^{-1}$. (This generalizes to any length of product). You can easily see that $A^{-1} = A$. B is just a translation, so that's easy to invert by making a similar matrix that translates by the negative of the offset by which B translates. Finally C is a scaling, which is also easy to invert. So we calculate

$$\begin{aligned}
(CBA)^{-1} &= A^{-1}B^{-1}C^{-1} \\
&= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 & 0 & -\frac{\alpha_1+\alpha_2}{2} \\ 0 & 1 & 0 & -\frac{\beta_1+\beta_2}{2} \\ 0 & 0 & 1 & -\gamma_2^{-1} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} \frac{2}{\alpha_2-\alpha_1} & 0 & 0 & 0 \\ 0 & \frac{2}{\beta_2-\beta_1} & 0 & 0 \\ 0 & 0 & \frac{1}{\gamma_1^{-1}-\gamma_2^{-1}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \\
&= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & \frac{\alpha_1+\alpha_2}{2} \\ 0 & 1 & 0 & \frac{\beta_1+\beta_2}{2} \\ 0 & 0 & 1 & \gamma_2^{-1} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{\alpha_2-\alpha_1}{2} & 0 & 0 & 0 \\ 0 & \frac{\beta_2-\beta_1}{2} & 0 & 0 \\ 0 & 0 & \gamma_1^{-1} - \gamma_2^{-1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{\alpha_2-\alpha_1}{2} & 0 & 0 & \frac{\alpha_1+\alpha_2}{2} \\ 0 & \frac{\beta_2-\beta_1}{2} & 0 & \frac{\beta_1+\beta_2}{2} \\ 0 & 0 & \gamma_1^{-1} - \gamma_2^{-1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \frac{\alpha_2-\alpha_1}{2} & 0 & 0 & \frac{\alpha_1+\alpha_2}{2} \\ 0 & \frac{\beta_2-\beta_1}{2} & 0 & \frac{\beta_1+\beta_2}{2} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \gamma_1^{-1} - \gamma_2^{-1} & 0 \end{pmatrix}
\end{aligned}$$

So now if someone gives you a depth buffer value and a pixel position, you can convert that to NDC space with a simple scaling and then apply $f_{CBA}^{-1} = f_{(CBA)^{-1}}$, which will transform back to camera space.

9 Acknowledgements

Some people found mistakes and gave suggestions. Thanks Sami Väisänen and Konsta Hölttä!