CAPITULO 2
OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

CAPITULO 4
ANIMACIÓN POR COMPUTADOR

2.3 Transformaciones Geométricas en 3D
2.4 Visualización en 3D
2.5 Proyecciones, eliminación de superficies ocultas

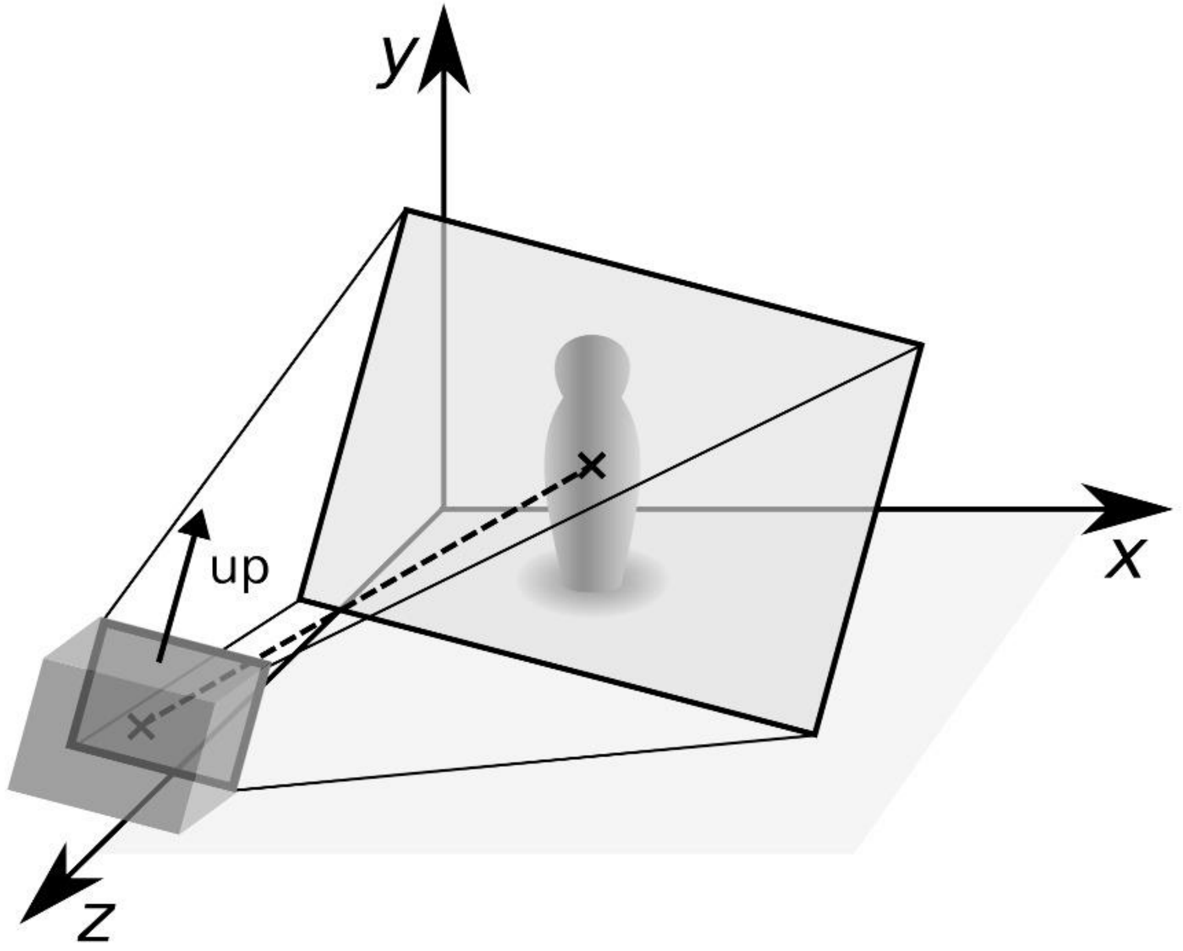
4.1 Cinemática Directa e Inversa
4.4 Realidad Virtual y Técnicas Avanzadas

VP3D – Camera

How we can set up a camera in OpenGL?

OpenGL by itself is **not familiar with the concept of a camera**, but we can try to simulate one by **moving all objects in the scene in the reverse direction**, giving the illusion that we are moving.

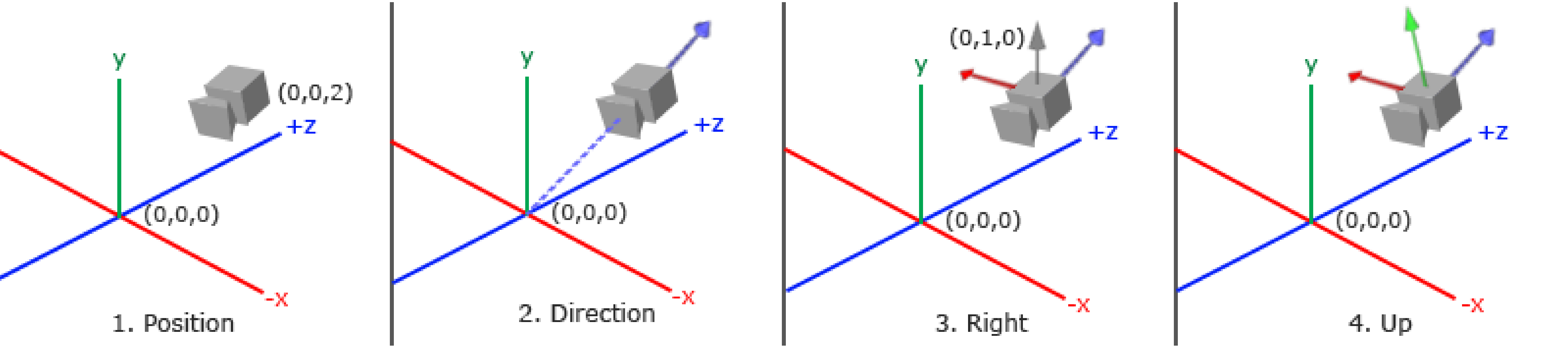
- We will discuss a **fly style camera** that allows you to freely move around in a 3D scene.
- We'll also discuss keyboard and mouse input and finish with a custom **camera class**



VP3D – Camera - View space

Camera / View Space means that all the vertex coordinates as seen from the camera's perspective as the origin of the scene.

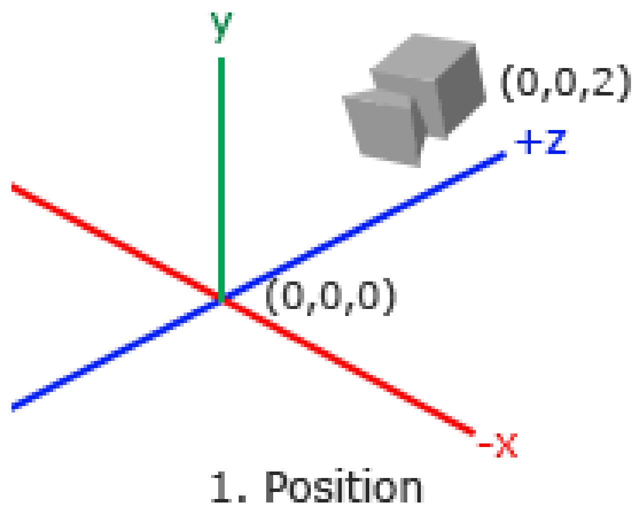
The view matrix transforms all the world coordinates into **view coordinates that are relative to the camera's position and direction**. To define a camera we need its position in world space, the direction it's looking at, a vector pointing to the right and a vector pointing upwards from the camera. A careful reader might notice that we're actually going to **create a coordinate system with 3 perpendicular unit axes with the camera's position as the origin.**



VP3D – Camera - View space

1. Camera Position

Camera / View Space means that all the vertex coordinates as seen from the camera's perspective as the origin of the scene.

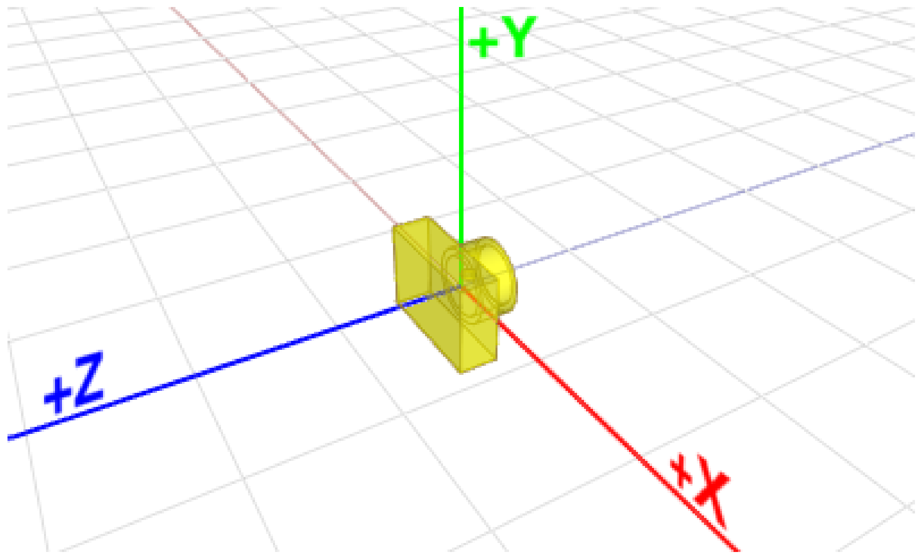


1. Position

- The camera position is a **vector** in world space that points to the **camera's position.**

- We set the camera at the same position we've set the camera in the previous chapter:

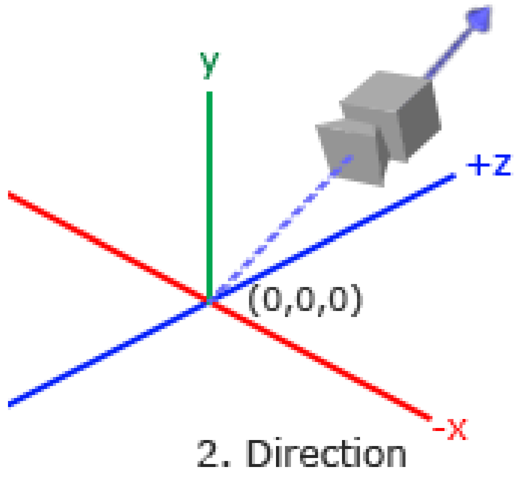
```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```



Don't forget that the positive z-axis is going through your screen towards you so if we want the camera to move backwards, we move along the positive z-axis.

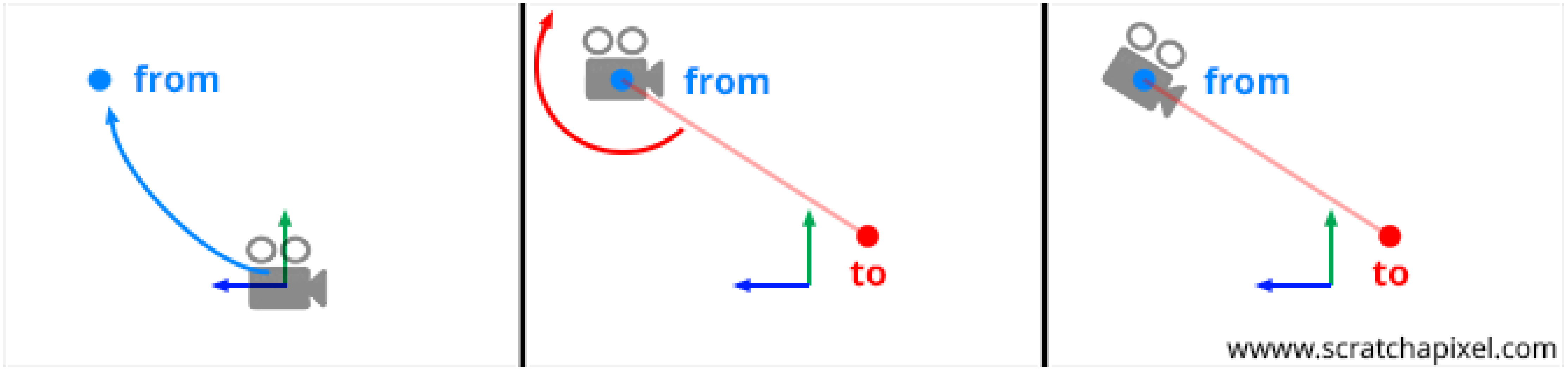
VP3D – Camera - View space

2. Camera Direction



- The next vector required is the camera's direction e.g. at what direction it is pointing at.
- For now we let the camera point to the origin of our scene: **(0,0,0).**
- Remember that if we subtract two vectors from each other we get a vector that's the difference of these two vectors? **Subtracting the camera position vector from the scene's origin vector** thus results in the **direction vector** we want.

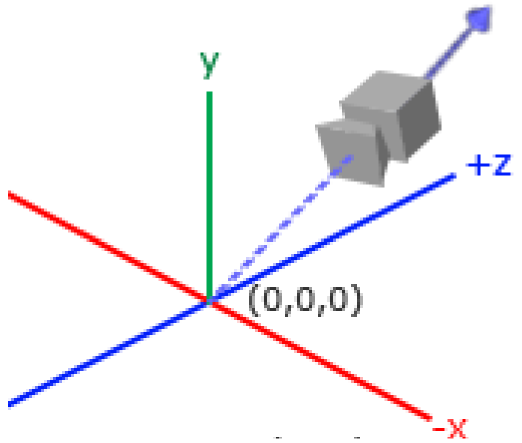
```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```



VP3D – Camera - View space

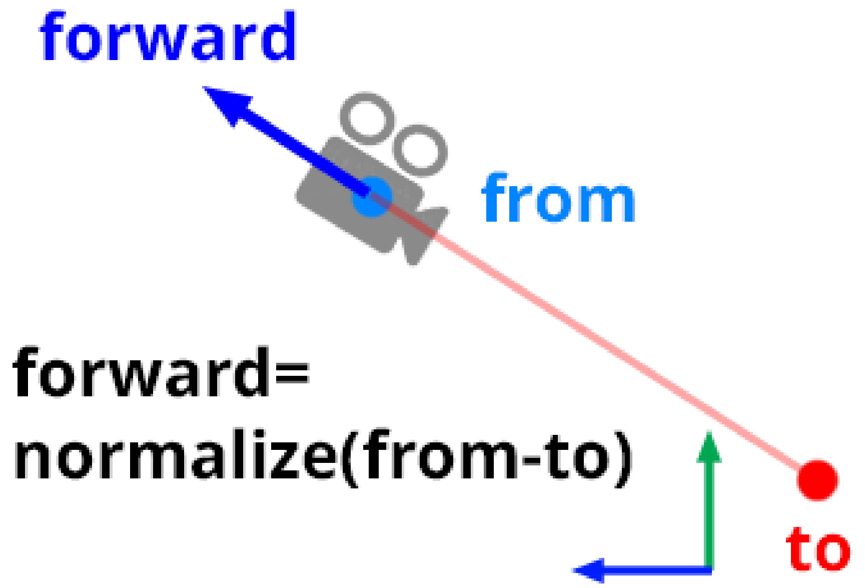
2. Camera Direction

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```



- For the view matrix's coordinate system we want its **z-axis to be positive** and because by convention (in OpenGL) the camera points towards the negative z-axis we want to negate the direction vector.

- If we switch the subtraction order around we now get a vector pointing towards the camera's positive z-axis



wwww.scratchapixel.com

The name direction vector is not the best chosen name, since it is actually pointing in the reverse direction of what it is targeting.

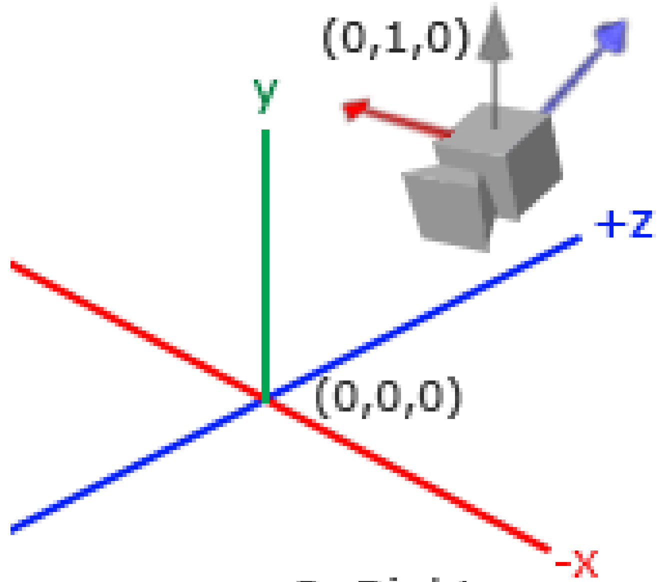
http://www.songho.ca/opengl/gl_camera.html

VP3D – Camera - View space

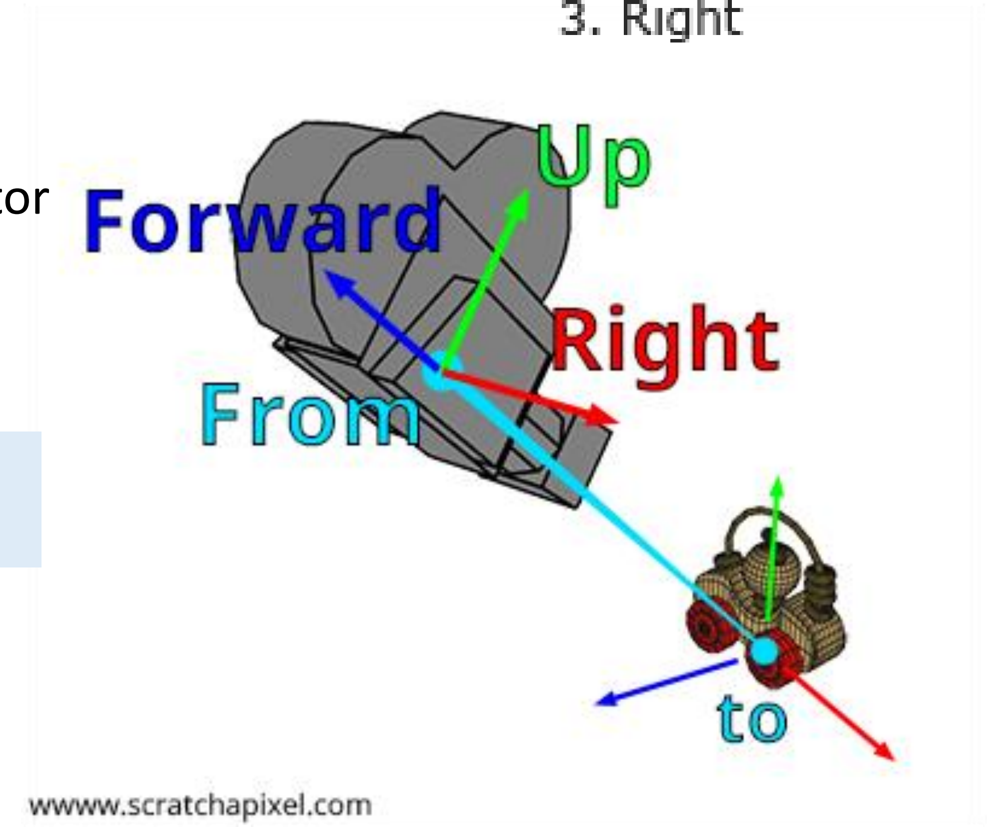
3. Right Axis

- Represents the positive x-axis of the camera space.
- To get the right vector we use a little trick by first specifying an **up vector** that points upwards (in world space). Then we do a **cross product** on the up vector and the direction vector from step 2.
- Since the result of a **cross product is a vector perpendicular to both vectors**, we will get a vector that **points in the positive x-axis's direction** (if we would switch the cross product order we'd get a vector that points in the negative x-axis):

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

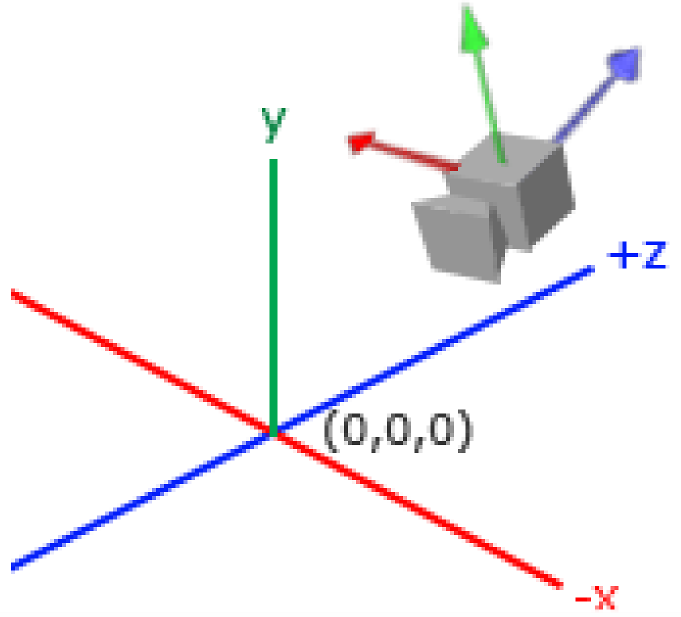


3. Right



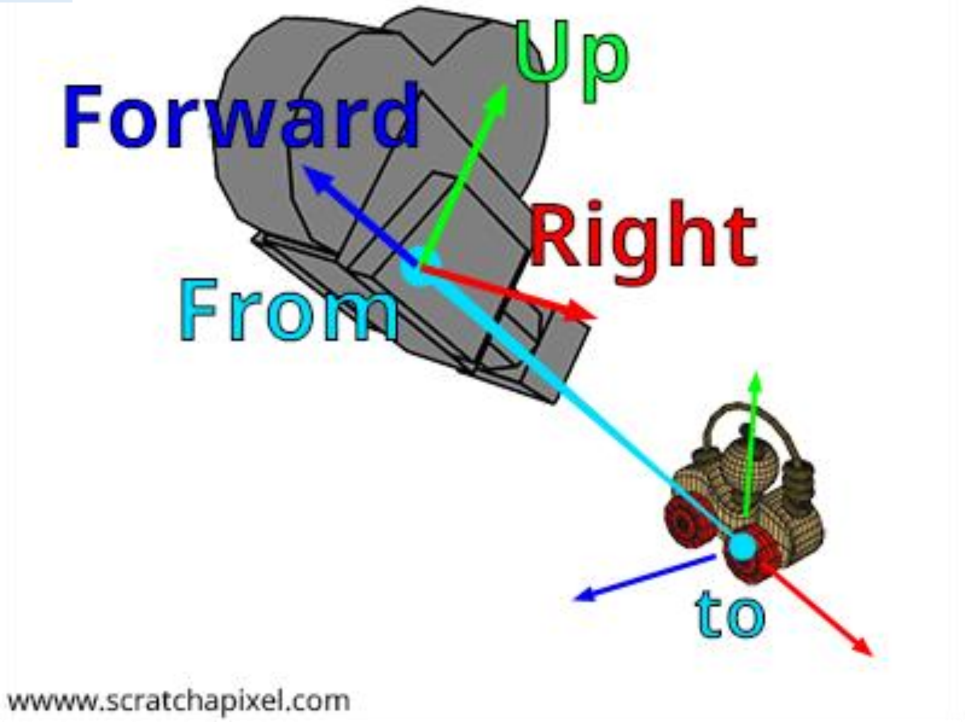
VP3D – Camera - View space

3. Up Axis



Now that we have both the x-axis vector and the z-axis vector, retrieving the vector that points to the camera's positive y-axis is relatively easy: we take the cross product of the right and direction vector

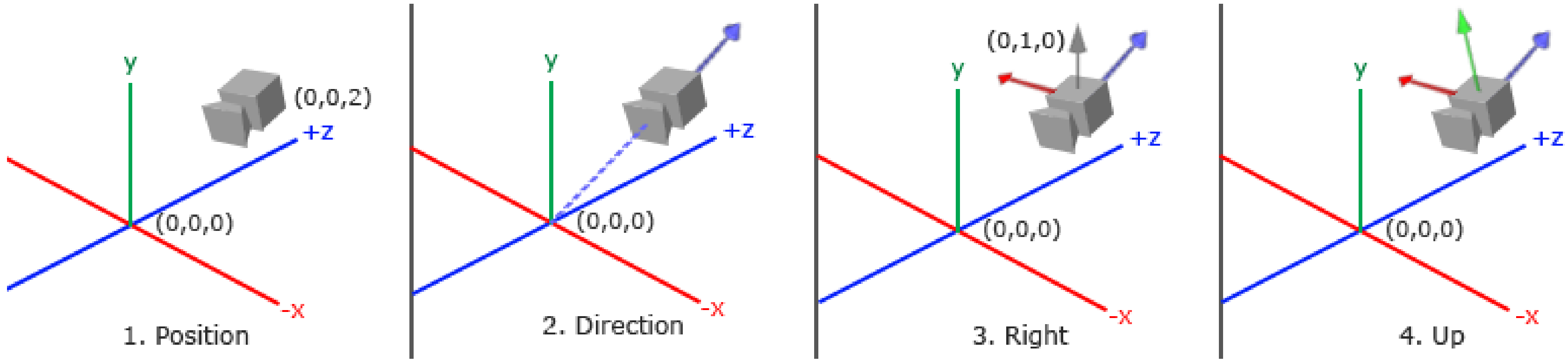
```
glm::vec3 cameraUp = glm::normalize( glm::cross(cameraDirection, cameraRight));
```



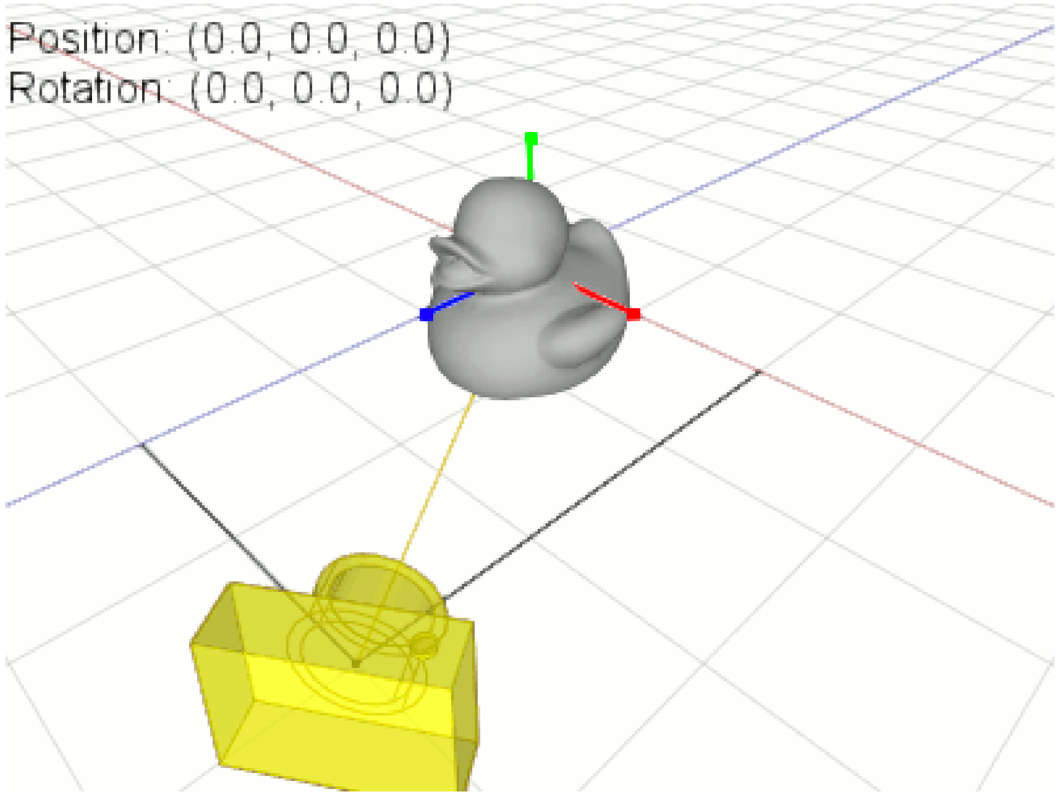
This process is known as the **Gram-Schmidt process** in linear algebra. Using these camera vectors we can now create a LookAt matrix that proves very useful for creating a camera

http://www.songho.ca/opengl/gl_camera.html

VP3D – Camera - View space



This process is known as the **Gram-Schmidt process** in linear algebra. Using these camera vectors we can now create a **LookAt matrix** that proves very useful for creating a camera



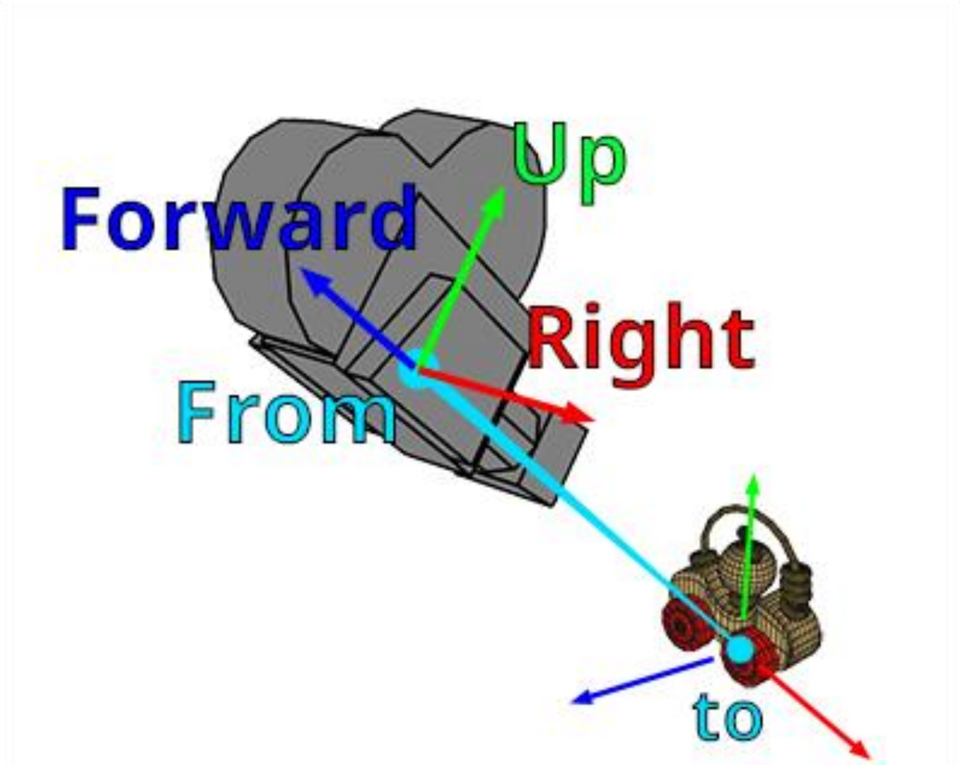
http://www.songho.ca/opengl/gl_camera.html

VP3D – Camera – Look At

If you define a coordinate space using **3 perpendicular (or non-linear) axes** you can create a matrix with those 3 axes plus a translation vector and you can **transform any vector to that coordinate space** by multiplying it with this matrix.

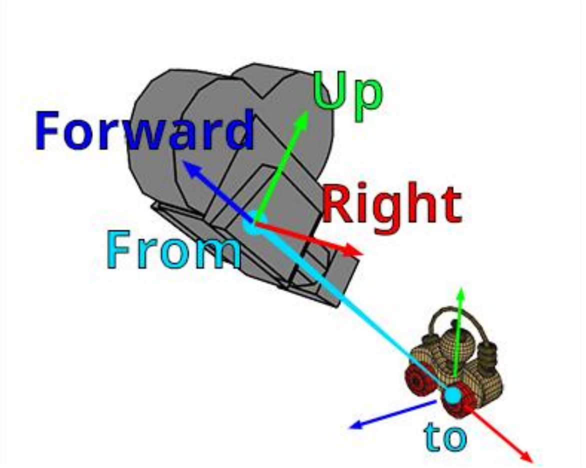
- This is exactly what the **LookAt matrix does** and now that we have **3 perpendicular axes and a position vector** to define the camera space we can create our own LookAt matrix:

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



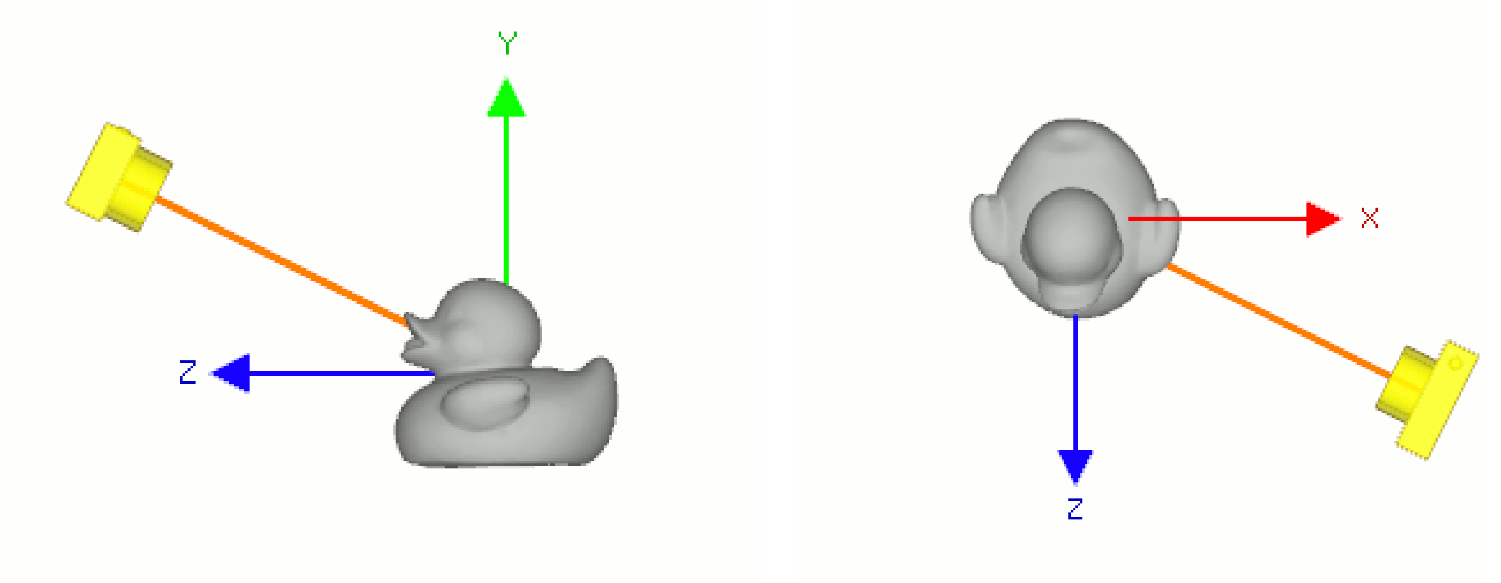
VP3D – Camera – Look At

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Where **R** is the right vector, **U** is the up vector, **D** is the direction vector and **P** is the camera's position vector.

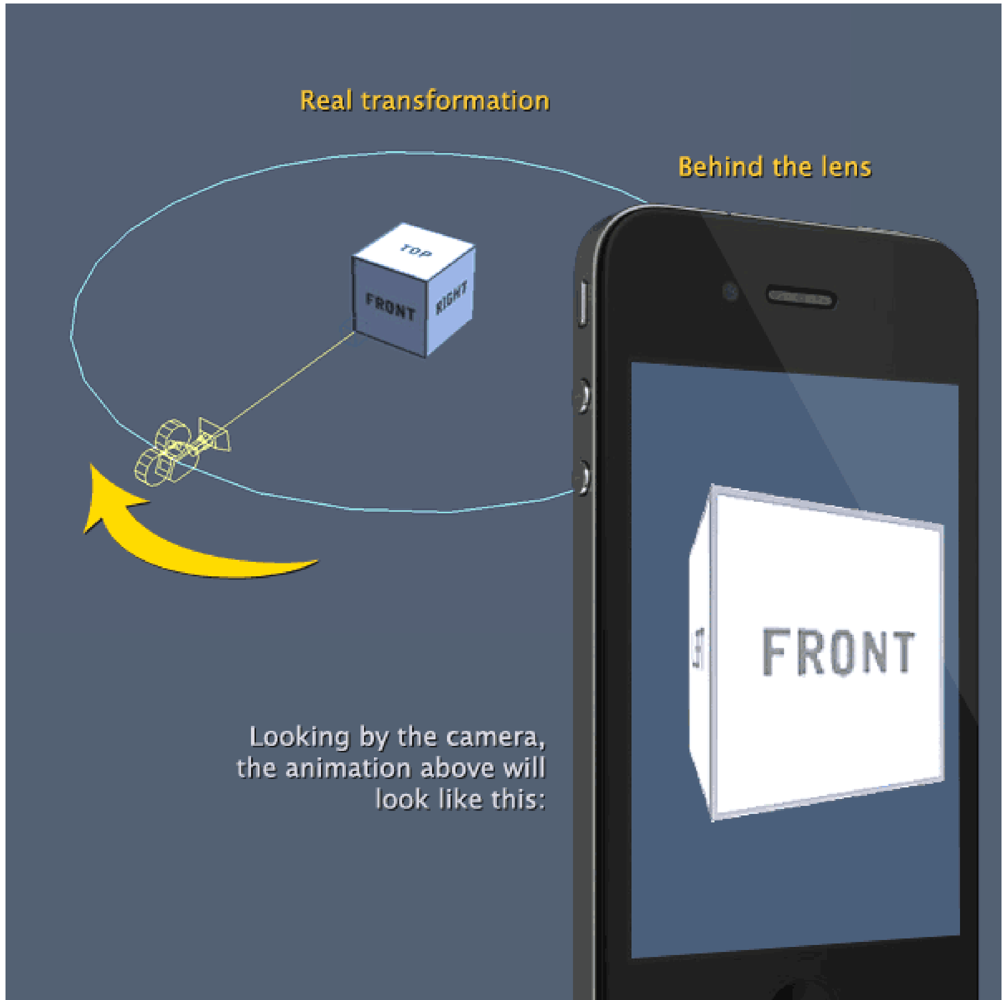
- Note that the **rotation (left matrix)** and **translation (right matrix)** parts are inverted (transposed and negated respectively) since we want to rotate and translate the world in the opposite direction of where we want the camera to move.

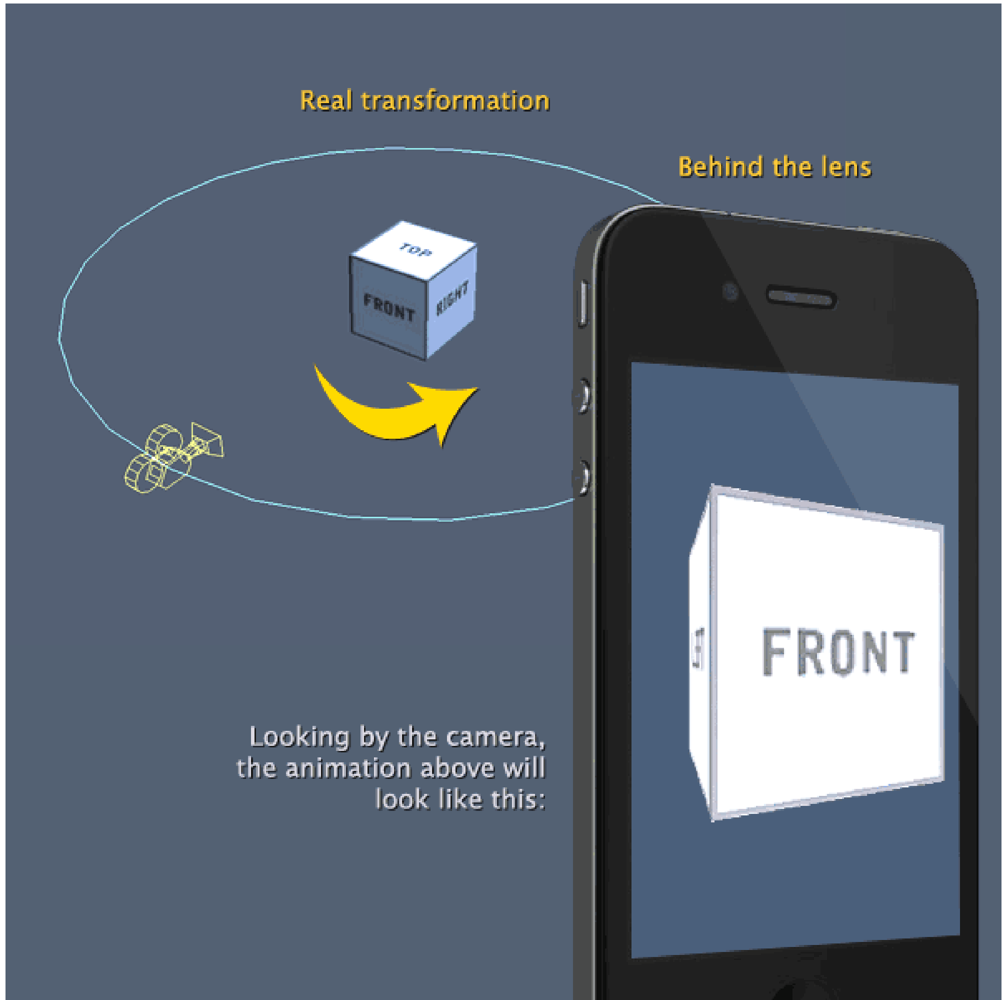


Using this LookAt matrix as our view matrix effectively transforms all the world coordinates to the view space we just defined. The LookAt matrix then does exactly what it says: it creates a view matrix that looks at a given target.

VP3D – Camera – Look At

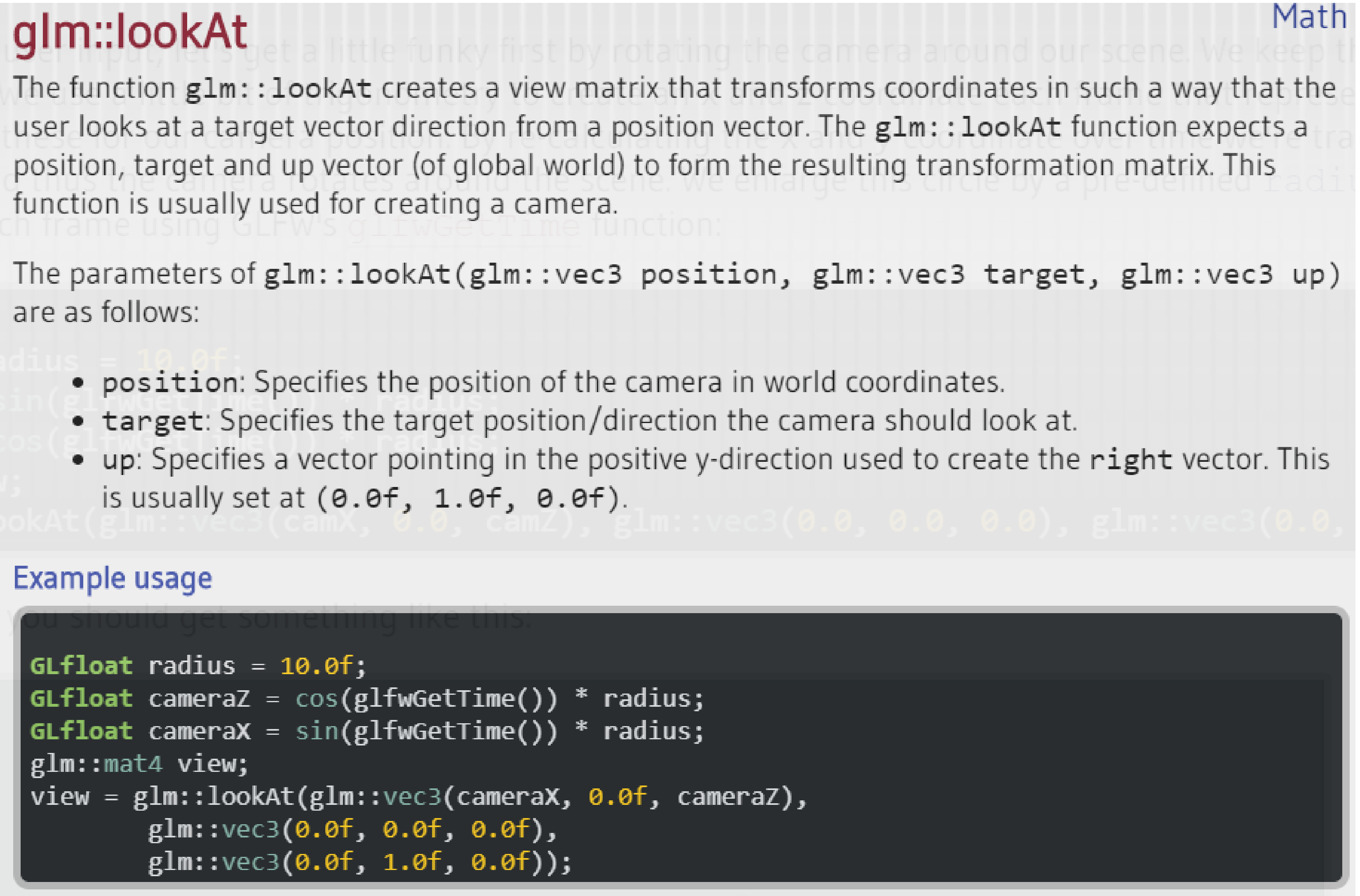
- Note that the **rotation (left matrix)** and **translation (right matrix)** parts are inverted (transposed and negated respectively) since we want to rotate and translate the world in the opposite direction of where we want the camera to move.


Real transformation
Behind the lens
TOP
FRONT
RIGHT
Looking by the camera, the animation above will look like this:
FRONT


Real transformation
Behind the lens
TOP
FRONT
RIGHT
Looking by the camera, the animation above will look like this:
FRONT

VP3D – Camera – Look At

GLM already does all this work for us. We only have to specify a camera position, a target position and a vector that represents the up vector in world space (the up vector we used for calculating the right vector). GLM then creates the **LookAt** matrix that we can use as our view matrix:



Math

glm::lookAt

The function `glm::lookAt` creates a view matrix that transforms coordinates in such a way that the user looks at a target vector direction from a position vector. The `glm::lookAt` function expects a position, target and up vector (of global world) to form the resulting transformation matrix. This function is usually used for creating a camera.

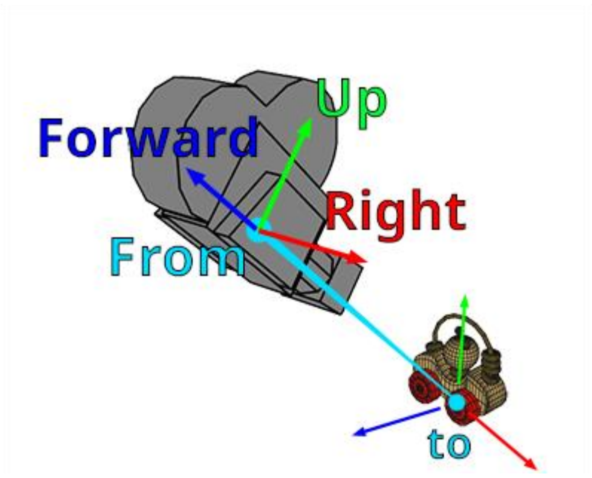
The parameters of `glm::lookAt(glm::vec3 position, glm::vec3 target, glm::vec3 up)` are as follows:

- `position`: Specifies the position of the camera in world coordinates.
- `target`: Specifies the target position/direction the camera should look at.
- `up`: Specifies a vector pointing in the positive y-direction used to create the `right` vector. This is usually set at `(0.0f, 1.0f, 0.0f)`.

Example usage

```
GLfloat radius = 10.0f;
GLfloat cameraZ = cos(glfwGetTime()) * radius;
GLfloat cameraX = sin(glfwGetTime()) * radius;
glm::mat4 view;
view = glm::lookAt(glm::vec3(cameraX, 0.0f, cameraZ),
        glm::vec3(0.0f, 0.0f, 0.0f),
        glm::vec3(0.0f, 1.0f, 0.0f));
```

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

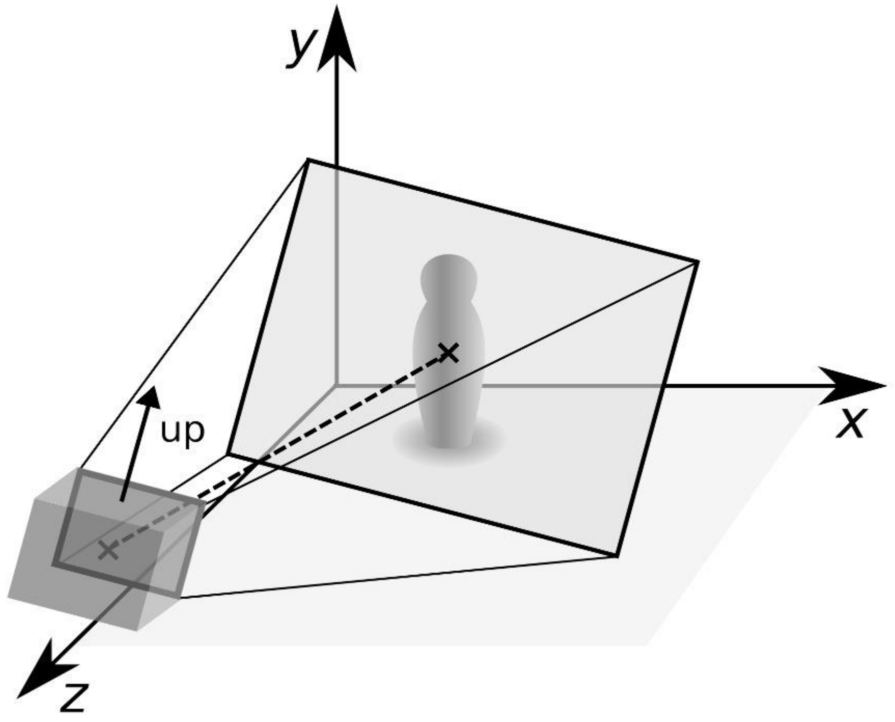


VP3D – Camera – Look At

GLM then creates the **LookAt** matrix that we can use as our view matrix:

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
                   glm::vec3(0.0f, 0.0f, 0.0f),
                   glm::vec3(0.0f, 1.0f, 0.0f));
```



- The **glm::LookAt** function requires a **position**, **target** and **up vector** respectively.
- This example creates a view matrix that is the same as the one we created in the previous exercise.