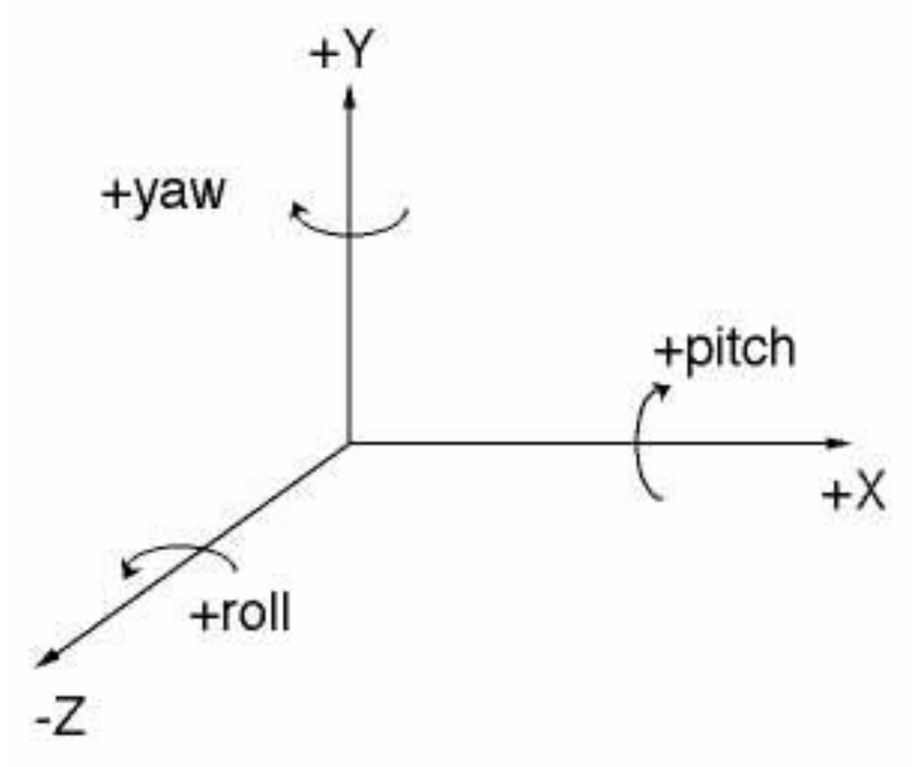
CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

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

T3D – Coordinate Systems

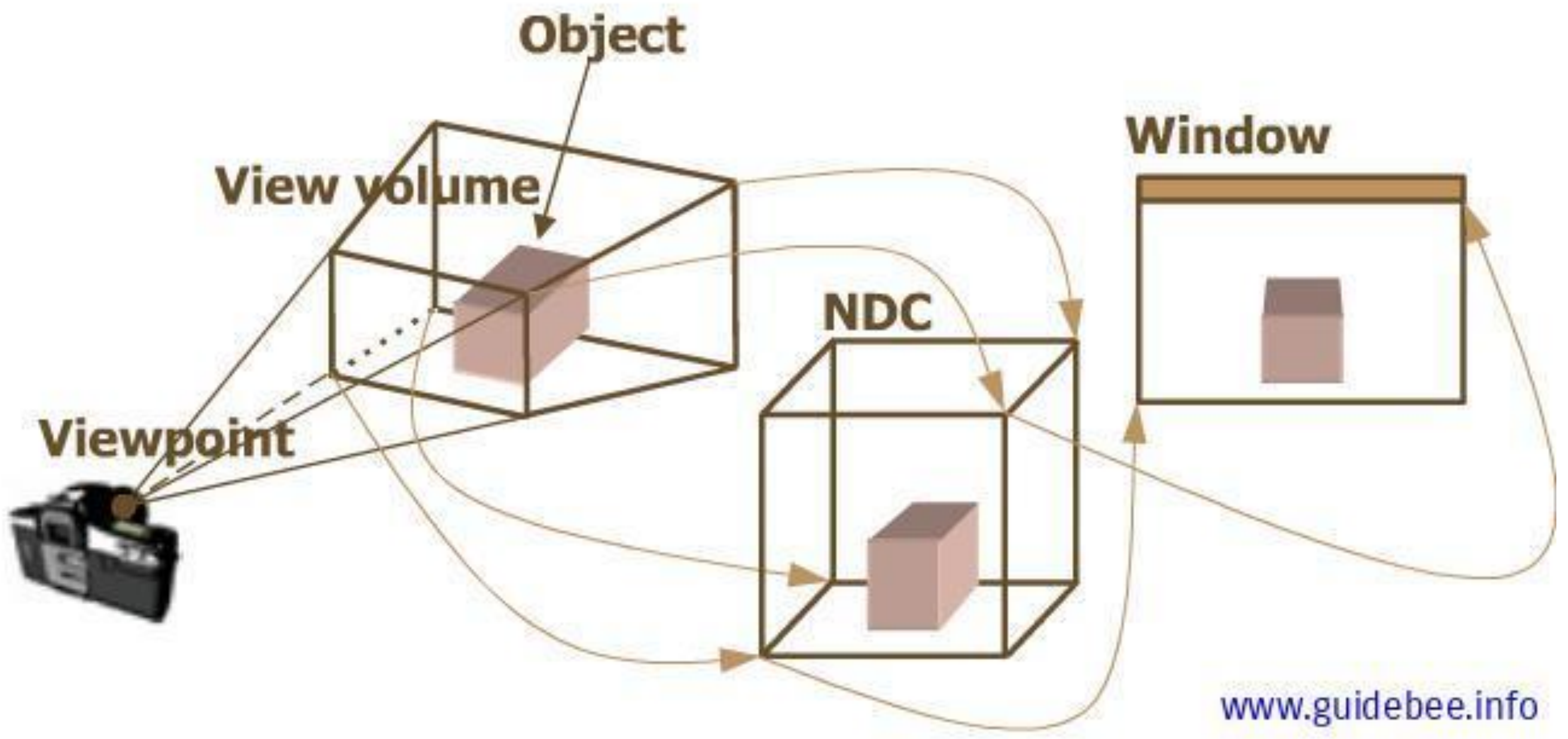
- OpenGL expects all the vertices, that we want to become visible, to be in normalized device coordinates after each vertex shader run.
- That is, the **x, y and z coordinates of each vertex should be between -1.0 and 1.0**; coordinates outside this range will not be visible.



T3D – Coordinate Systems

- What we usually do, is specify the coordinates in a range (or space) we determine ourselves and in the **vertex shader** transform these coordinates to **normalized device coordinates** (**NDC**).

- These **NDC** are then given to the rasterizer to transform them to 2D coordinates/pixels on your screen

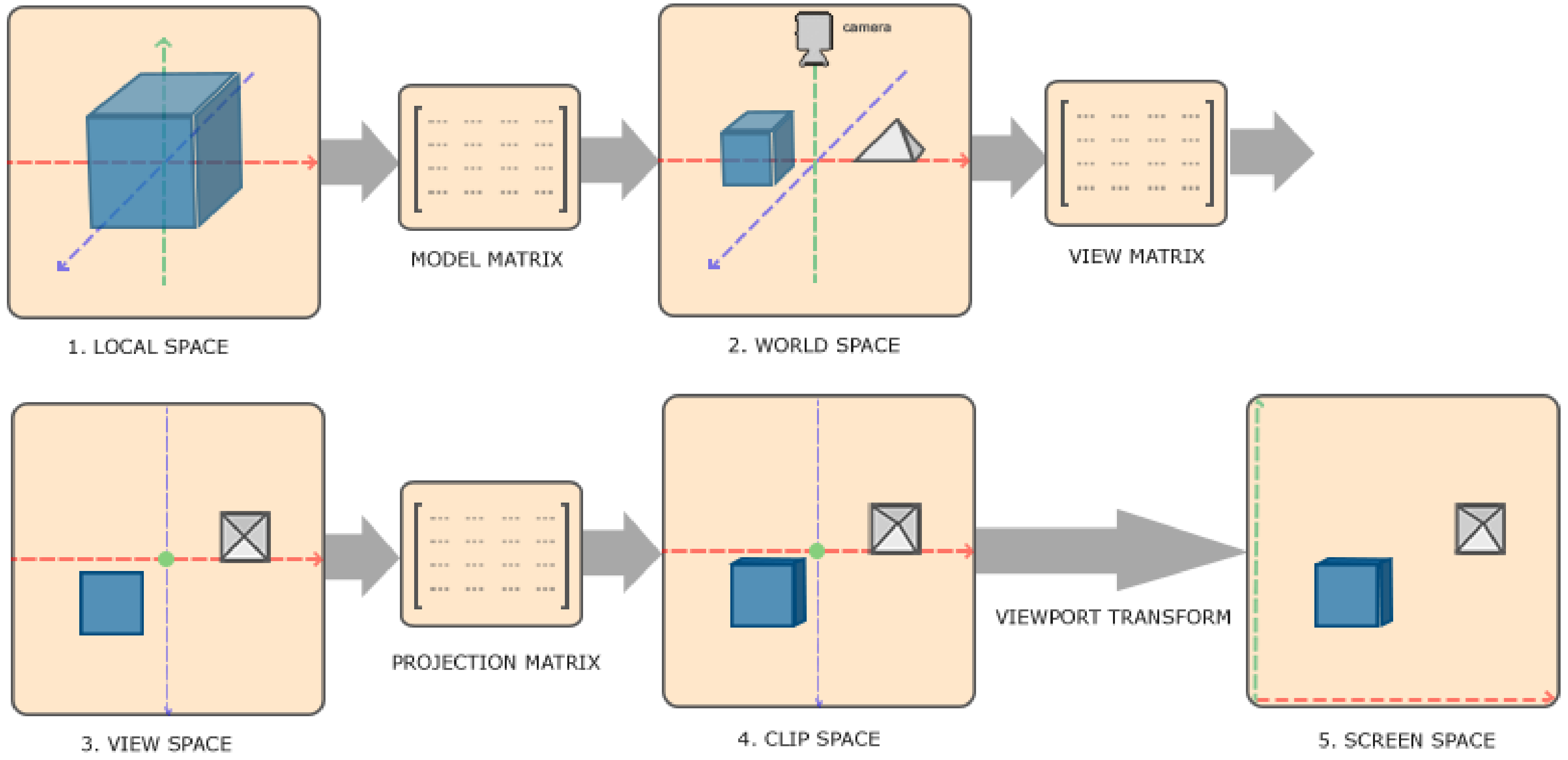


T3D – Coordinate Systems

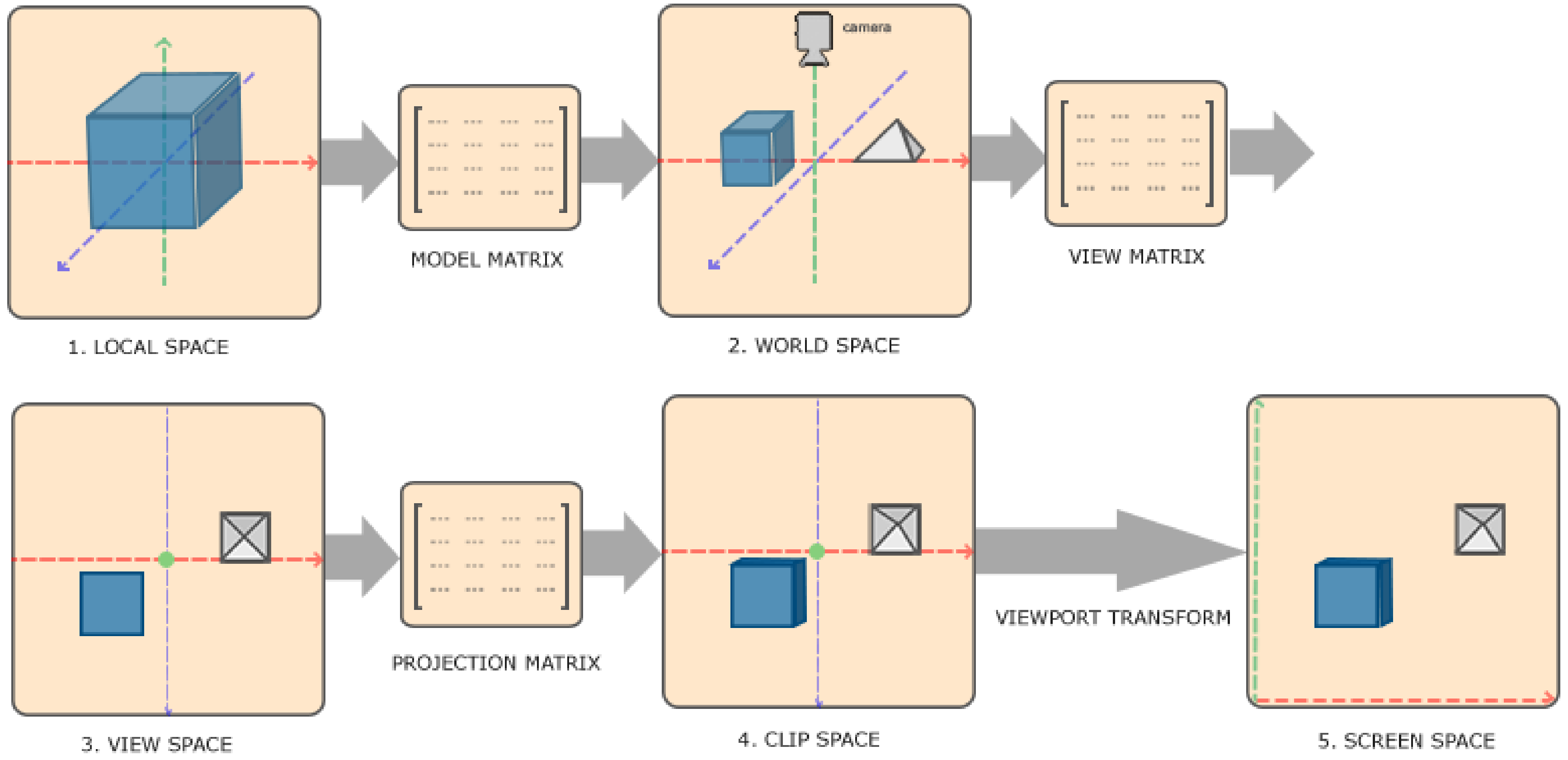
- Transforming **coordinates to NDC** is usually accomplished in a step-by-step fashion where we transform an object's vertices to several coordinate systems before finally transforming them to NDC.
- Please visit: https://jsantell.com/model-view-projection/
- The advantage of transforming them to several intermediate coordinate systems is that some operations/calculations are easier in certain coordinate systems as will soon become apparent.

There are a total of **5 different coordinate systems** that are of importance to us:

- Local space (or Object space)
- World space
- View space (or Eye space)
- Clip space
- Screen space

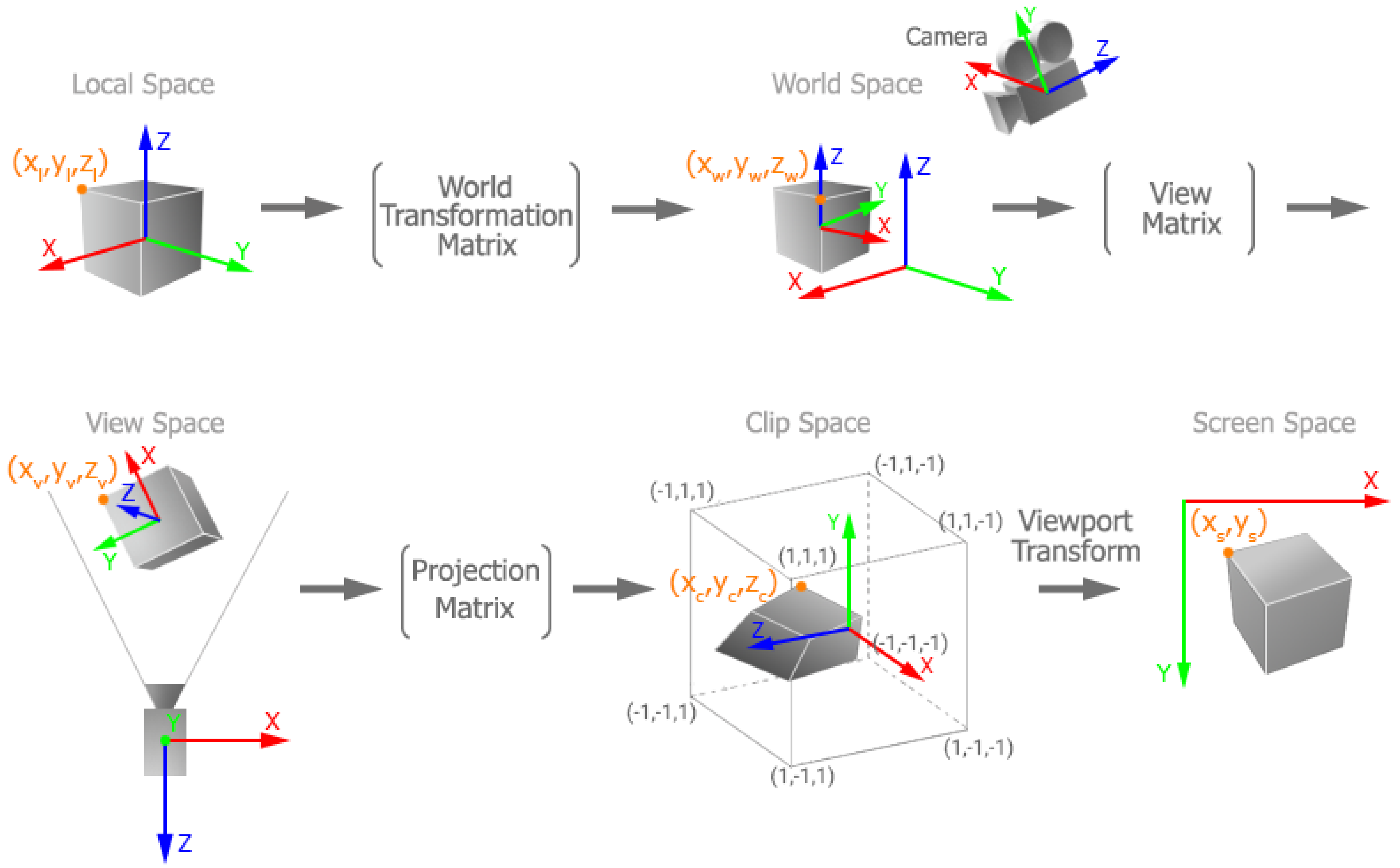


T3D – Coordinate Systems – The global picture



To transform the coordinates from one space to the next coordinate space we'll use **several transformation matrices** of which the most important are the model, view and projection matrix. Our vertex coordinates first start in **local space** as local coordinates and are then further processed to **world coordinates**, **view coordinates, clip coordinates** and eventually end up as **screen coordinates**.

T3D – Coordinate Systems – The global picture

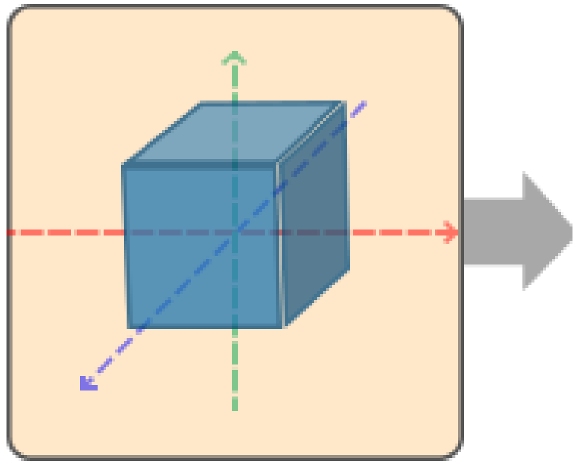


To transform the coordinates from one space to the next coordinate space we'll use **several transformation matrices** of which the most important are the model, view and projection matrix. Our vertex coordinates first start in **local space** as local coordinates and are then further processed to **world coordinates**, **view coordinates, clip coordinates** and eventually end up as **screen coordinates**.

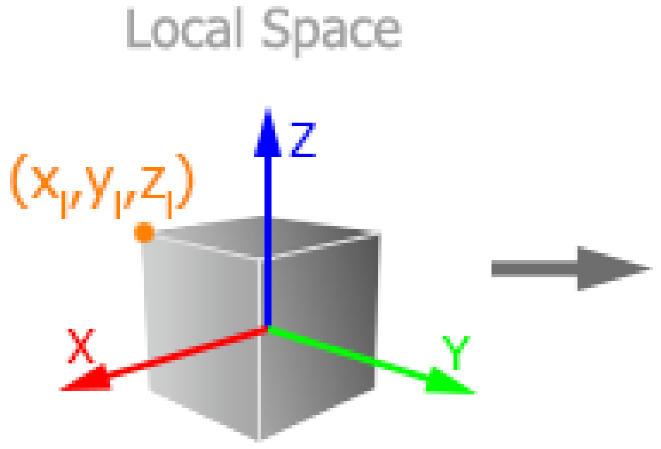
T3D – Coordinate Systems – The global picture

1. Local space

- **Local space** is the coordinate space that is local to your object, i.e. where your object begins in. Imagine that you've created your cube in a modeling software package (like Blender).
- The origin of your cube is probably at **(0,0,0)** even though your cube might end up at a different location in your final application.
- Probably all the models you've created all have **(0,0,0)** as their initial position.



1. LOCAL SPACE

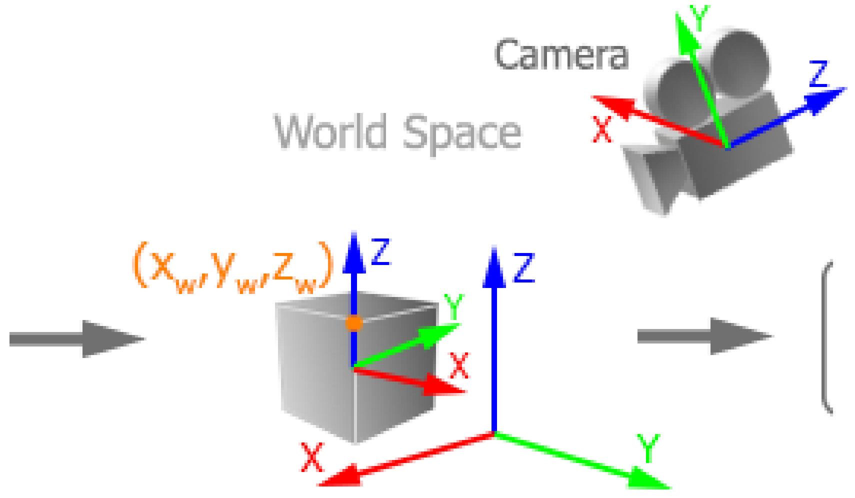


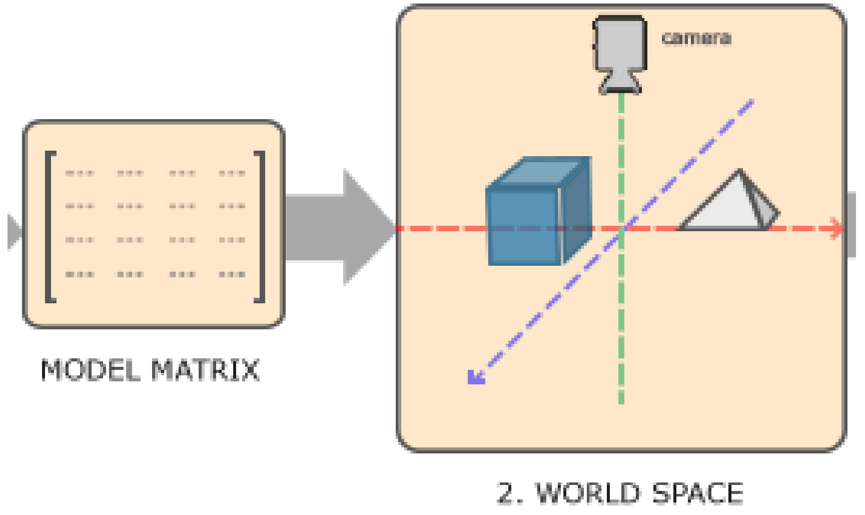
- All the vertices of your model are therefore in local space: **they are all local to your object.**
- The vertices of the container we've been using were specified as coordinates between -0.5 and 0.5 with 0.0 as its origin. These are **local coordinates.**

T3D – Coordinate Systems – The global picture

2. World Space

- If we would import all our objects directly in the application they would probably all be somewhere positioned inside each other at the world's origin of (0,0,0) which is not what we want.
- We want to define **a position for each object to position them inside a larger world.**
- The coordinates of your object are transformed from local to world space; this is accomplished with the model matrix.



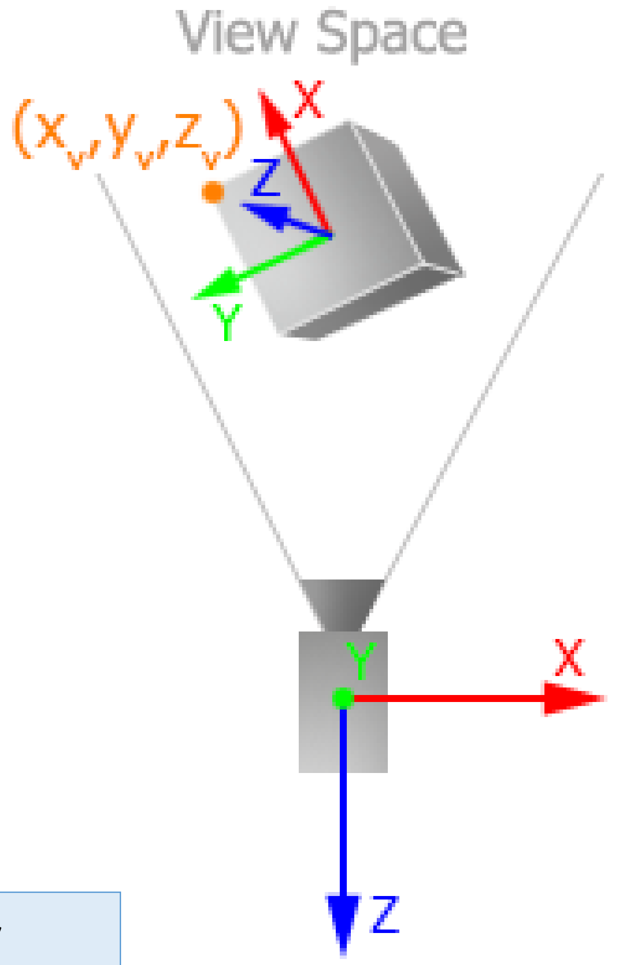


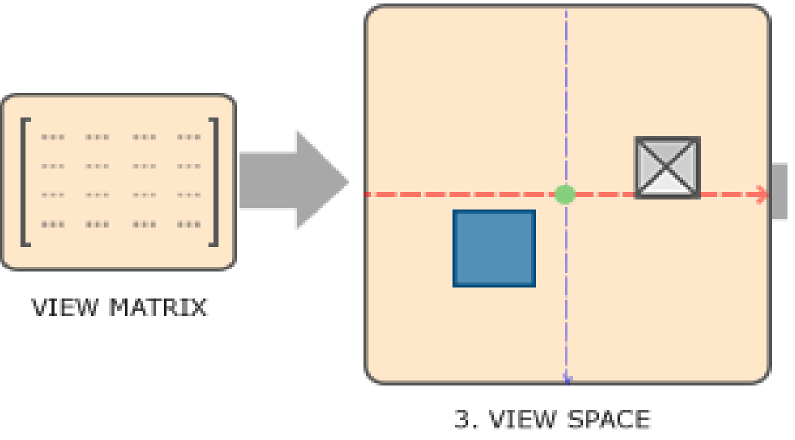
- The model matrix is a transformation matrix that translates, scales and/or rotates your **object to place it in the world at a location/orientation they belong to**.

T3D – Coordinate Systems – The global picture

3. View Space

- The view space is what people usually refer to as the **camera** of OpenGL (it is sometimes also known as **camera space or eye space**).
- The view space is the result of transforming your world-space coordinates to **coordinates that are in front of the user's view**.
- The view space is thus the space as seen from the **camera's point of view**.
- This is usually accomplished with a combination of translations and rotations to translate/rotate the scene so that certain items are transformed to the **front of the camera**.



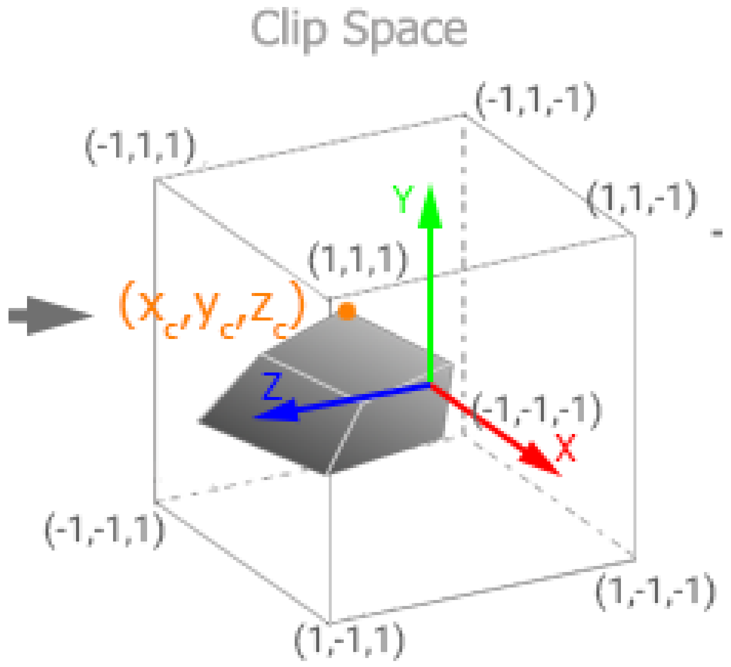


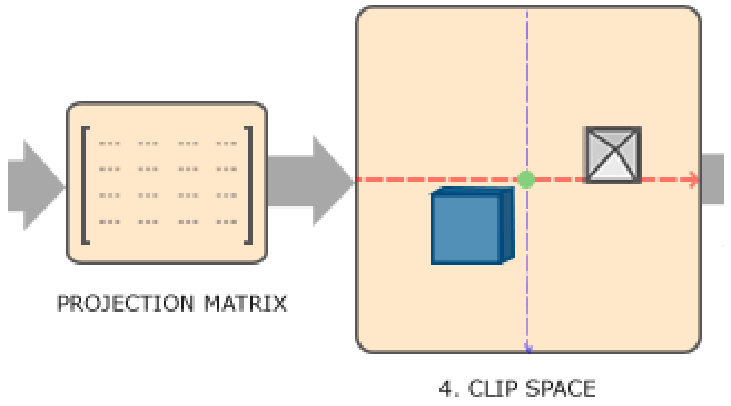
These combined transformations are generally stored inside a **view matrix** that transforms world coordinates to view space.

T3D – Coordinate Systems – The global picture

4. Clip Space

- At the end of each vertex shader run, OpenGL expects the coordinates to be **within a specific range** and any coordinate that falls **outside this range** is **clipped**.
- Coordinates that are clipped are **discarded**, so the remaining coordinates will end up as fragments visible on your screen.
- This is also where **clip space** gets its name from.



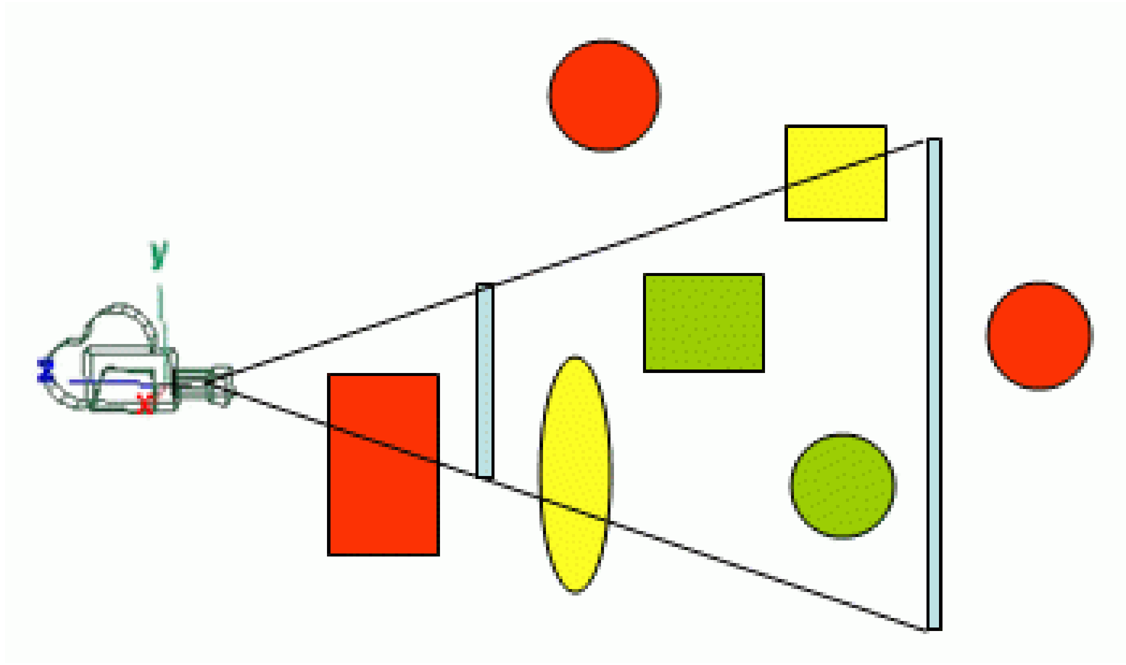


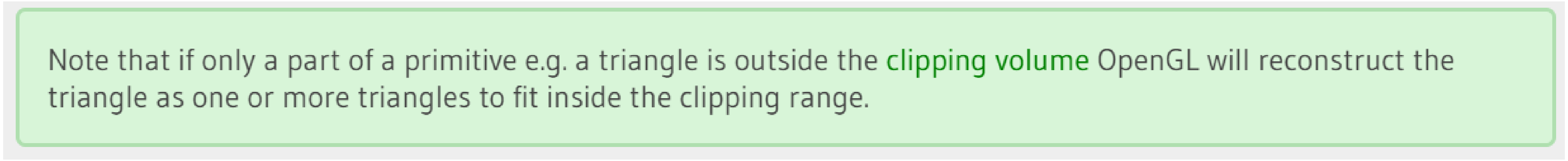
- To transform vertex coordinates from view to clip-space we define a so called **projection matrix** that specifies a range of coordinates e.g. -1000 and 1000 in each dimension.
- The **projection matrix** then **transforms coordinates** within this specified range to **normalized device coordinates (-1.0, 1.0)**.

T3D – Coordinate Systems – The global picture

4. Clip Space

- All coordinates outside this range will not be mapped between -1.0 and 1.0 and therefore be clipped.
- With this range we specified in the projection matrix, a coordinate of **(1250, 500, 750)** would **not be visible**, since the x coordinate is out of range and thus gets converted to a coordinate higher than 1.0 in NDC and is therefore **clipped**.



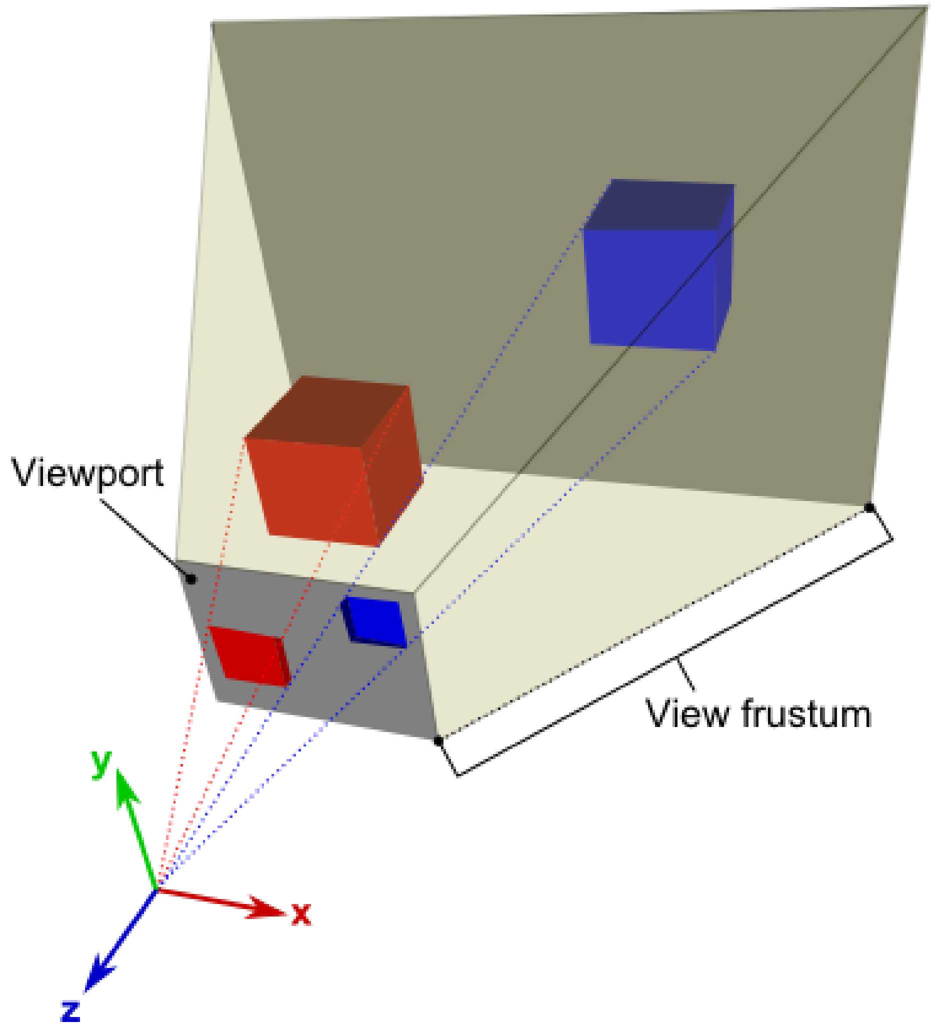

Note that if only a part of a primitive e.g. a triangle is outside the clipping volume OpenGL will reconstruct the triangle as one or more triangles to fit inside the clipping range.

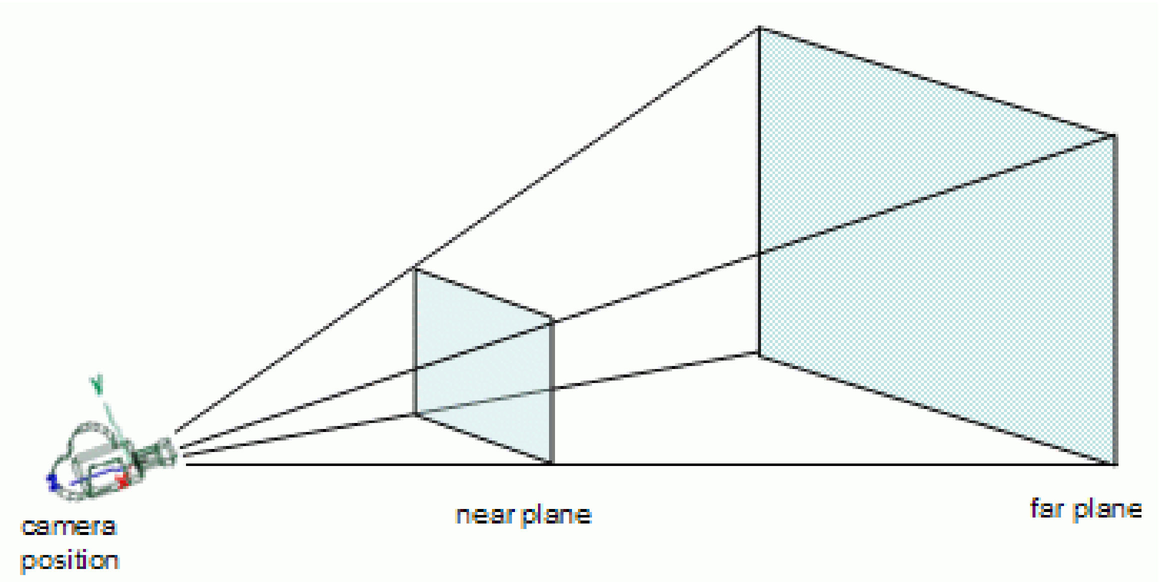
T3D – Coordinate Systems – The global picture

4. Clip Space - Proyection

- This viewing box a projection matrix creates is called a **frustum** and each coordinate that ends up inside this frustum will end up on the user's screen.

- The total process to convert coordinates within a specified range to NDC that can easily be mapped to 2D view-space coordinates is called **projection** since the projection matrix **projects 3D coordinates to the easy-to-map-to-2D normalized device coordinates**.





This step is performed **automatically** at the end of the **vertex shader** step.

T3D – Coordinate Systems – The global picture

4. Clip Space – Perspective Division

Once all the vertices are transformed to clip space a final operation called **perspective division** is performed where we divide the x, y and z components of the **position vectors** by the vector's homogeneous **w component**; perspective division is what transforms the **4D clip space coordinates to 3D normalized device coordinates**.

- It is after this stage where the resulting coordinates are mapped to screen coordinates (using the settings of **glViewport**) and turned into fragments.

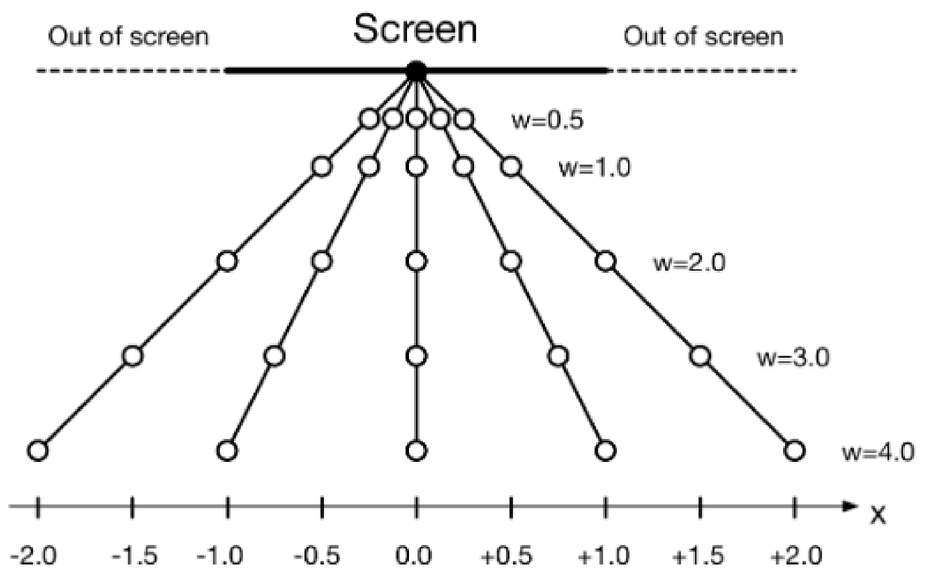
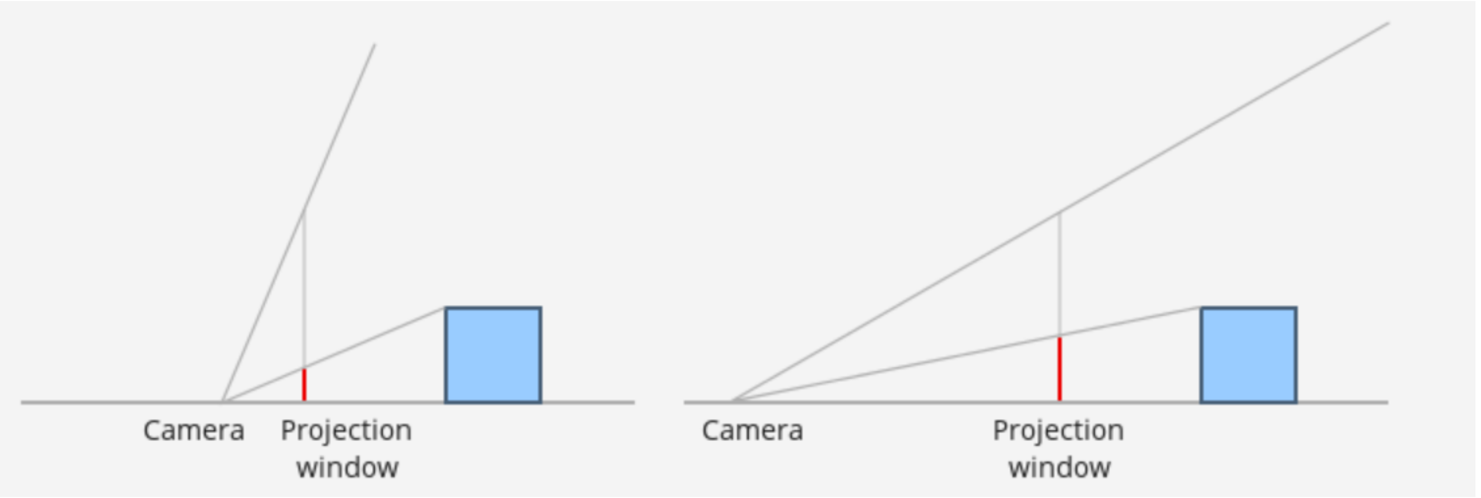


Figure 4.1

5 sets of homogeneous coordinates, each of them corresponding to the set of unidimensional Cartesian coordinates [-1.0, -0.5, 0.0, +0.5, +1.0].

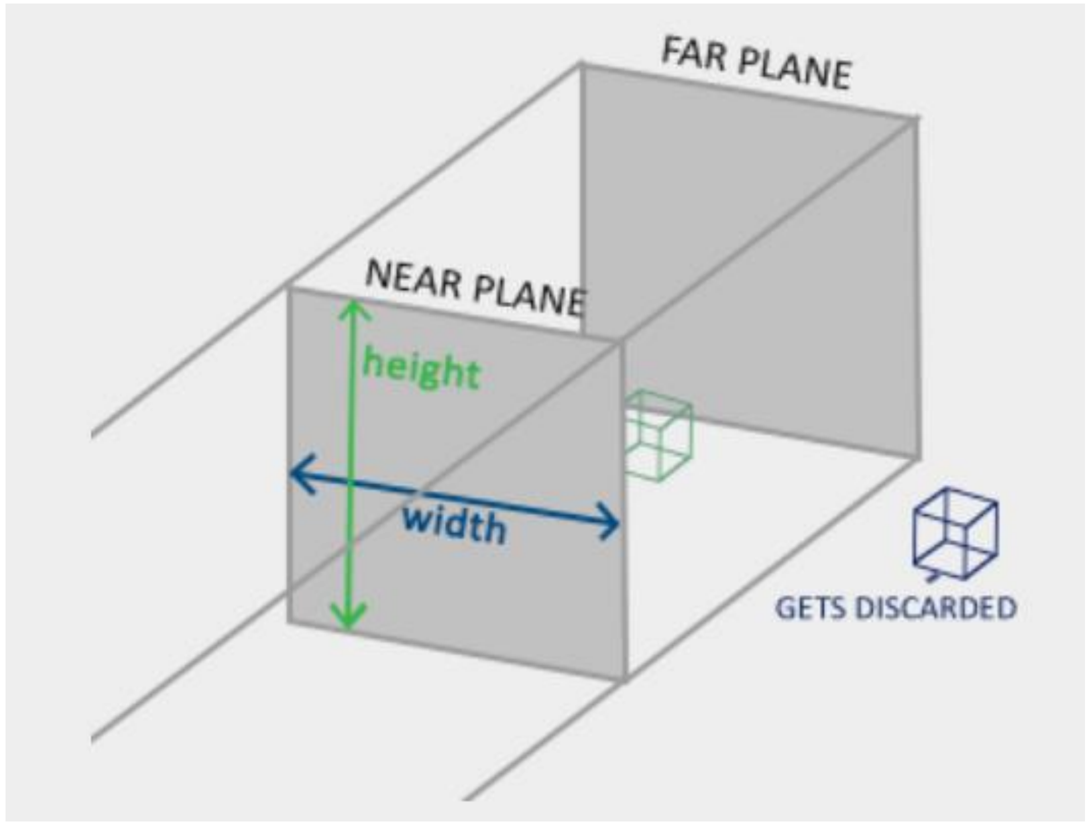


- The projection matrix to transform view coordinates to clip coordinates usually takes two different forms, where each form defines its own unique frustum. We can either create an **orthographic projection** matrix or a **perspective projection** matrix

T3D – Coordinate Systems – The global picture

4. Clip Space – Orthographic projection

- An orthographic projection matrix defines **a cube-like frustum** box that defines the clipping space where each vertex **outside this box is clipped**.
- When creating an orthographic projection matrix we specify the **width, height and length** of the **visible frustum**.



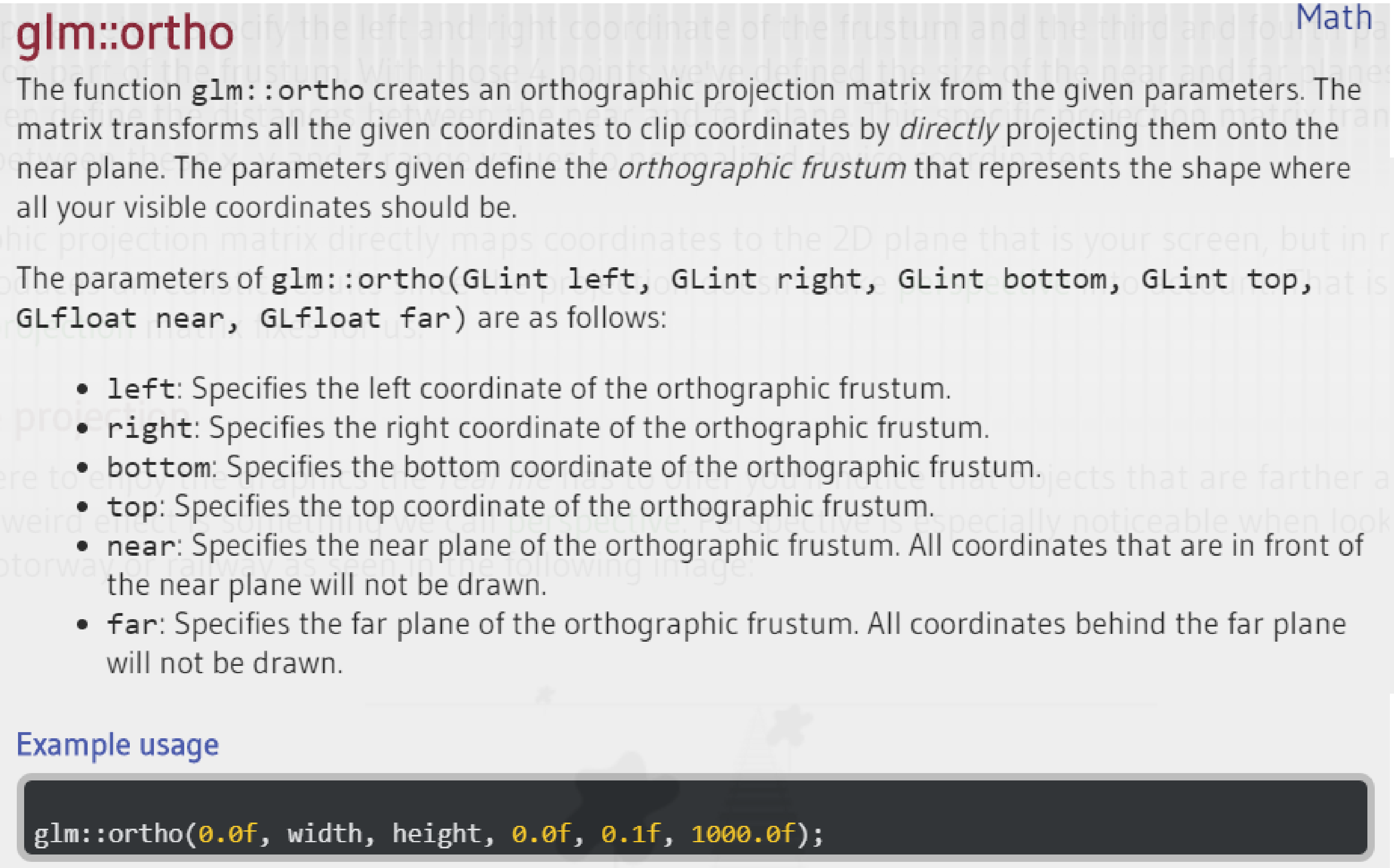
All the coordinates inside this frustum will end up within **the NDC range** after transformed by its matrix and thus won't be clipped.

T3D – Coordinate Systems – The global picture

4. Clip Space – Orthographic projection

To create an orthographic projection matrix we make use of GLM's built-in function **glm::ortho**:

glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);



Math

glm::ortho

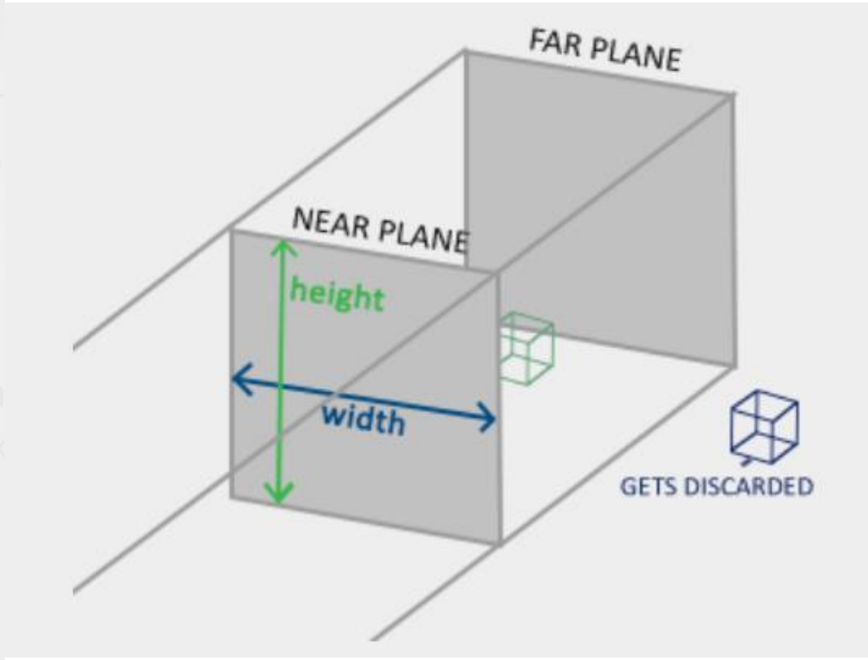
The function `glm::ortho` creates an orthographic projection matrix from the given parameters. The matrix transforms all the given coordinates to clip coordinates by *directly* projecting them onto the near plane. The parameters given define the *orthographic frustum* that represents the shape where all your visible coordinates should be.

The parameters of `glm::ortho(GLint left, GLint right, GLint bottom, GLint top, GLfloat near, GLfloat far)` are as follows:

- `left`: Specifies the left coordinate of the orthographic frustum.
- `right`: Specifies the right coordinate of the orthographic frustum.
- `bottom`: Specifies the bottom coordinate of the orthographic frustum.
- `top`: Specifies the top coordinate of the orthographic frustum.
- `near`: Specifies the near plane of the orthographic frustum. All coordinates that are in front of the near plane will not be drawn.
- `far`: Specifies the far plane of the orthographic frustum. All coordinates behind the far plane will not be drawn.

Example usage

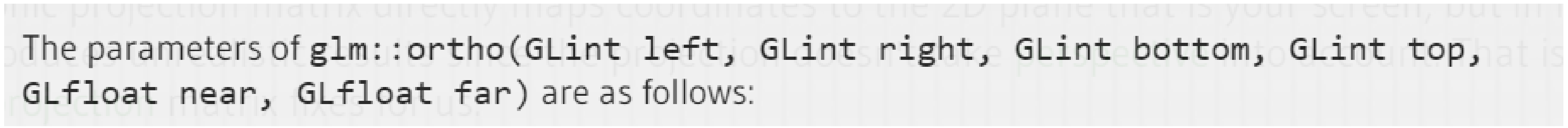
```
glm::ortho(0.0f, width, height, 0.0f, 0.1f, 1000.0f);
```



T3D – Coordinate Systems – The global picture

4. Clip Space – Orthographic projection

To create an orthographic projection matrix we make use of GLM's built-in function **glm::ortho**:

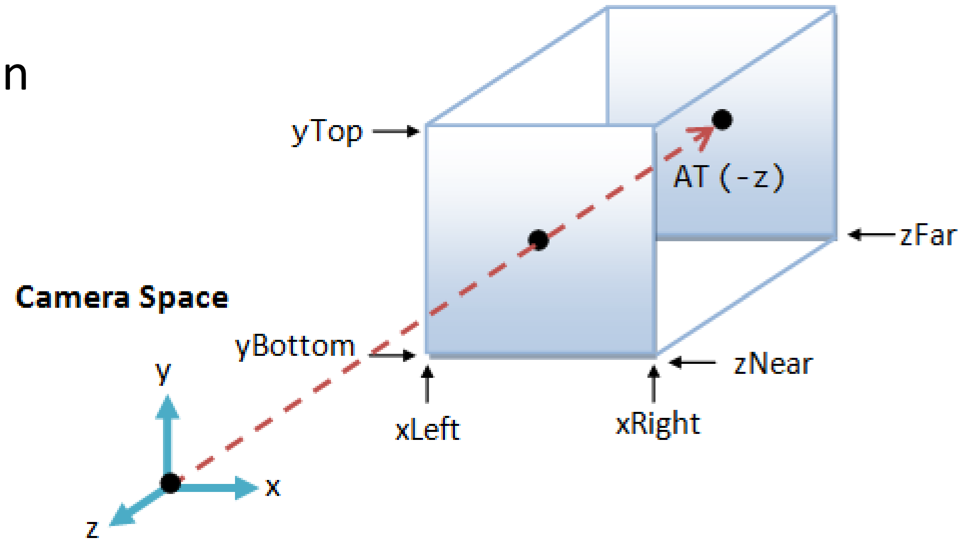


The parameters of `glm::ortho(GLint left, GLint right, GLint bottom, GLint top, GLfloat near, GLfloat far)` are as follows:

- This specific projection matrix transforms all coordinates between these **x, y and z range** values to **normalized device coordinates**.

- An orthographic projection matrix directly maps coordinates to the 2D plane that is your screen, but in reality **a direct projection produces unrealistic results since the projection doesn't take perspective into account**.

That is something the **perspective projection matrix** fixes for us.



Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

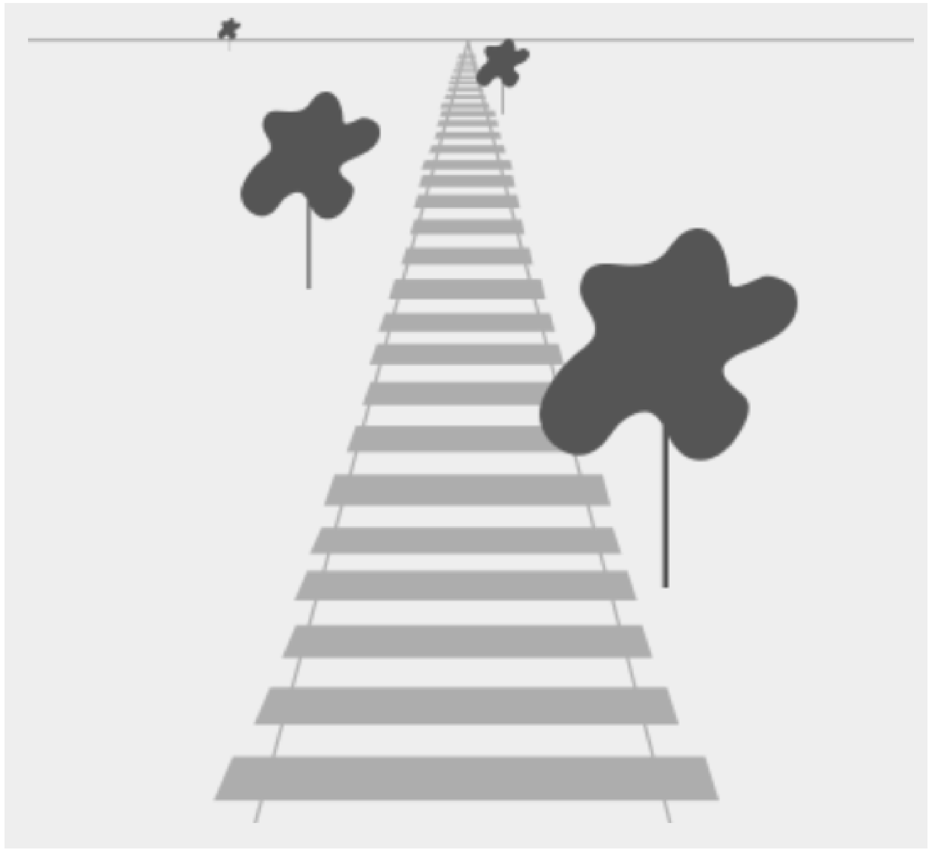
T3D – Coordinate Systems – The global picture

4. Clip Space – Perspective projection

The graphics the real life has to offer you'll notice that objects that are **farther away appear much smaller**. This weird effect is something we call **perspective**.

- As you can see, due to perspective the lines seem to coincide at a far enough distance.
- This is exactly the effect perspective projection tries to mimic and it does so using a **perspective projection matrix**

The **projection matrix** maps a given frustum range to clip space, but also **manipulates the w value** of each vertex coordinate in such a way that the further away a vertex coordinate is from the viewer, the higher this w component becomes.

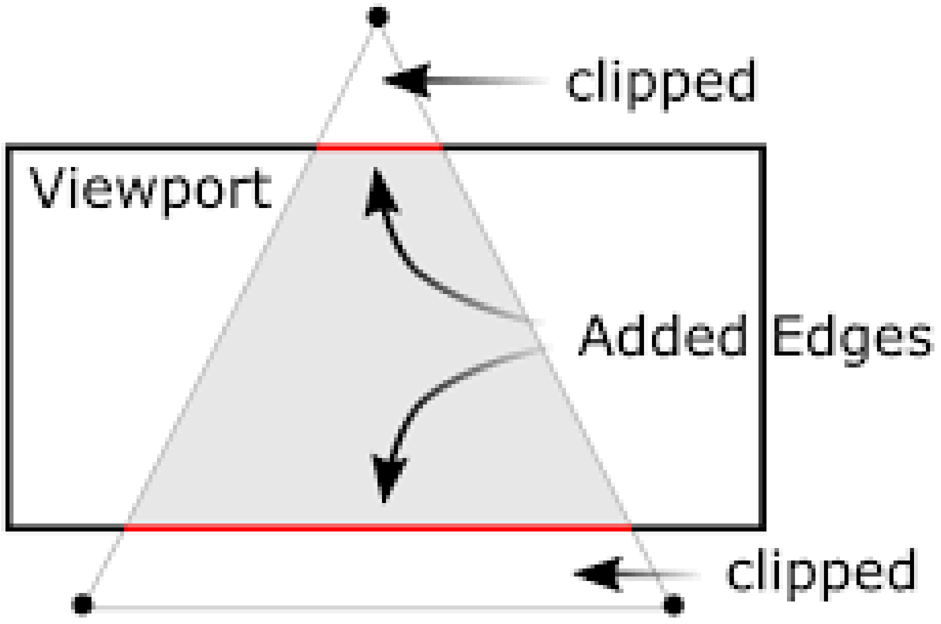


T3D – Coordinate Systems – The global picture

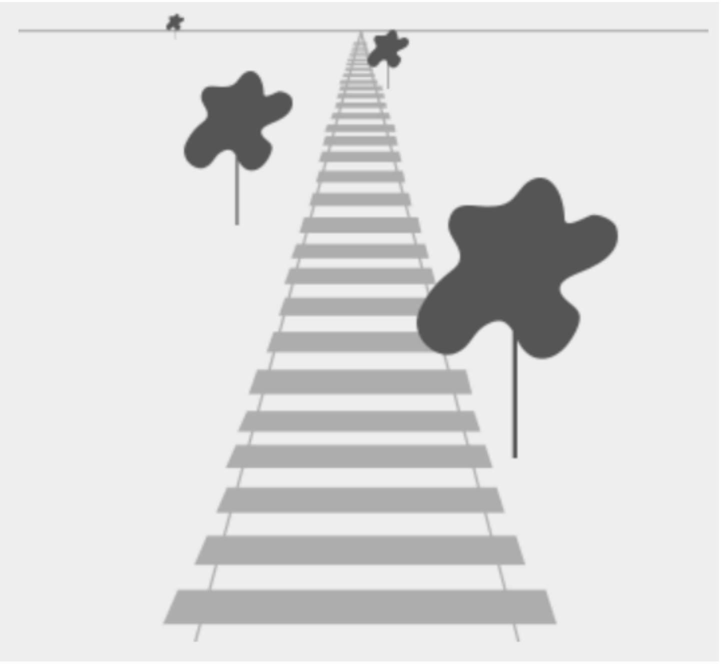
4. Clip Space – Perspective projection

- Once the coordinates are transformed to clip space they are in the range **-w to w** (anything outside this range is clipped).

- OpenGL requires that the visible coordinates fall between the range **-1.0 and 1.0** as the **final vertex shader output**, thus once the coordinates are in clip space, perspective division is applied to the clip space coordinates:



$$out = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$



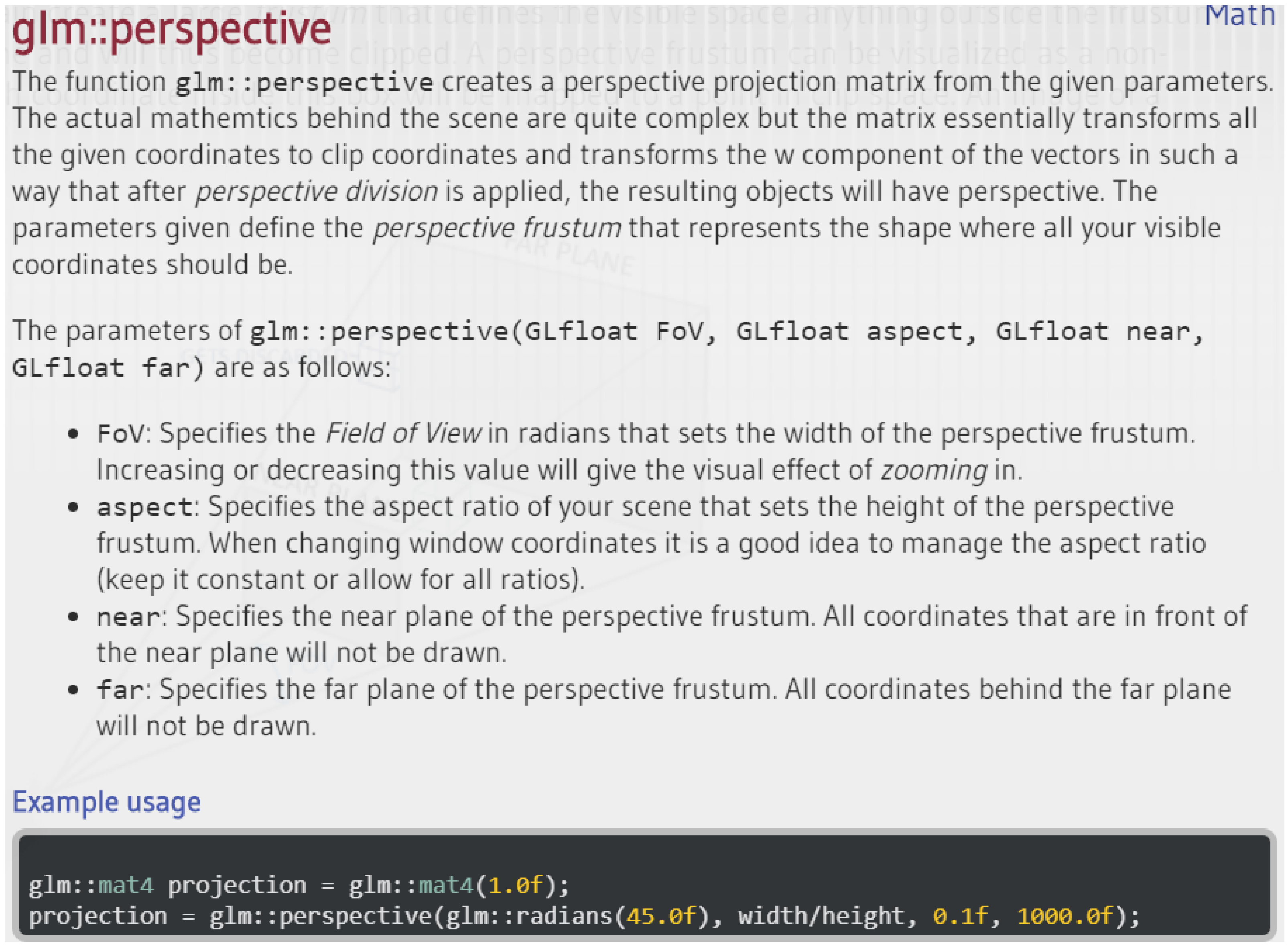
Each component of the vertex coordinate is **divided by its w component** giving smaller vertex coordinates the further away a vertex is from the viewer. This is another reason why the w component is important, since it helps us with **perspective projection**.

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

T3D – Coordinate Systems – The global picture

4. Clip Space – Perspective projection

A perspective projection matrix can be created in GLM as follows:



Math

glm::perspective

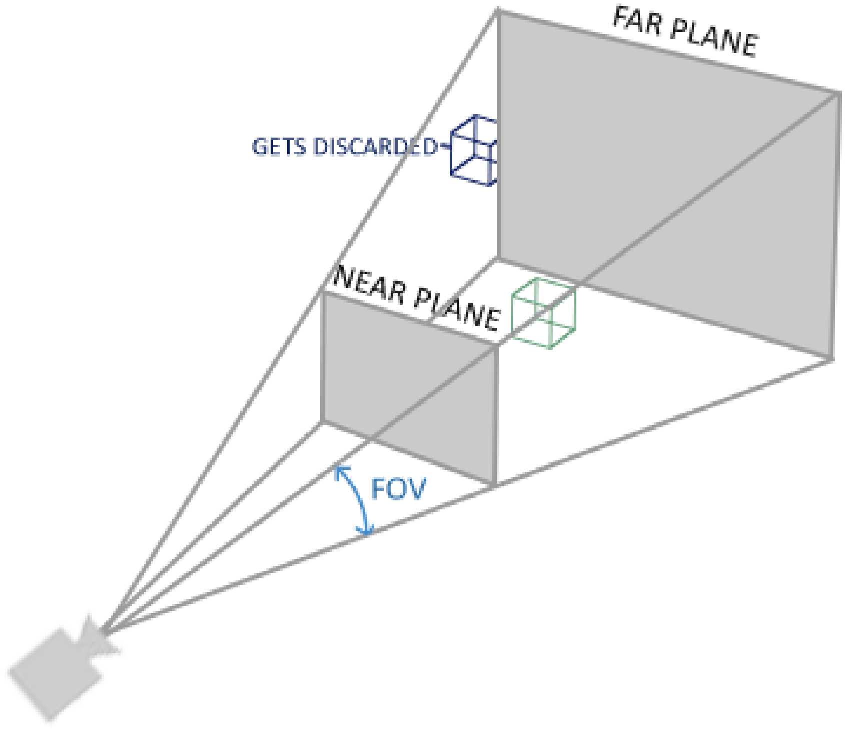
The function `glm::perspective` creates a perspective projection matrix from the given parameters. The actual mathemtics behind the scene are quite complex but the matrix essentially transforms all the given coordinates to clip coordinates and transforms the w component of the vectors in such a way that after *perspective division* is applied, the resulting objects will have perspective. The parameters given define the *perspective frustum* that represents the shape where all your visible coordinates should be.

The parameters of `glm::perspective(GLfloat FoV, GLfloat aspect, GLfloat near, GLfloat far)` are as follows:

- `FoV`: Specifies the *Field of View* in radians that sets the width of the perspective frustum. Increasing or decreasing this value will give the visual effect of *zooming* in.
- `aspect`: Specifies the aspect ratio of your scene that sets the height of the perspective frustum. When changing window coordinates it is a good idea to manage the aspect ratio (keep it constant or allow for all ratios).
- `near`: Specifies the near plane of the perspective frustum. All coordinates that are in front of the near plane will not be drawn.
- `far`: Specifies the far plane of the perspective frustum. All coordinates behind the far plane will not be drawn.

Example usage

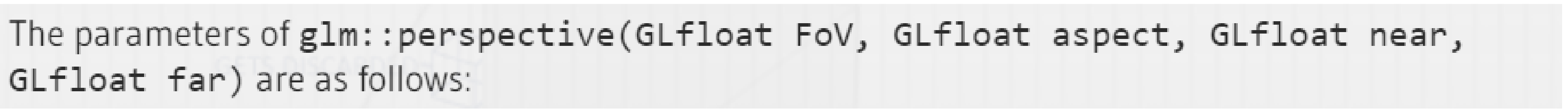
```
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(45.0f), width/height, 0.1f, 1000.0f);
```

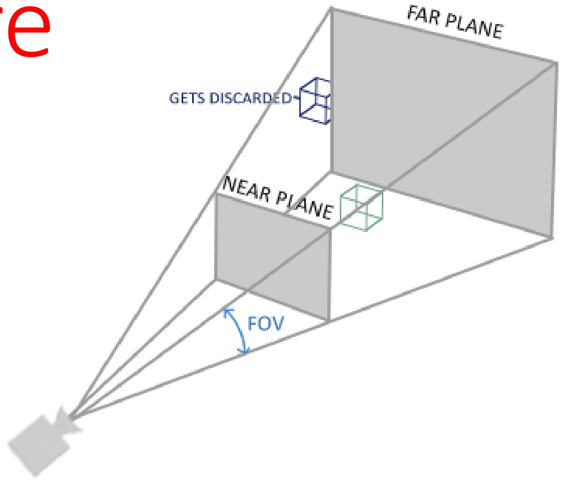


T3D – Coordinate Systems – The global picture

4. Clip Space – Perspective projection

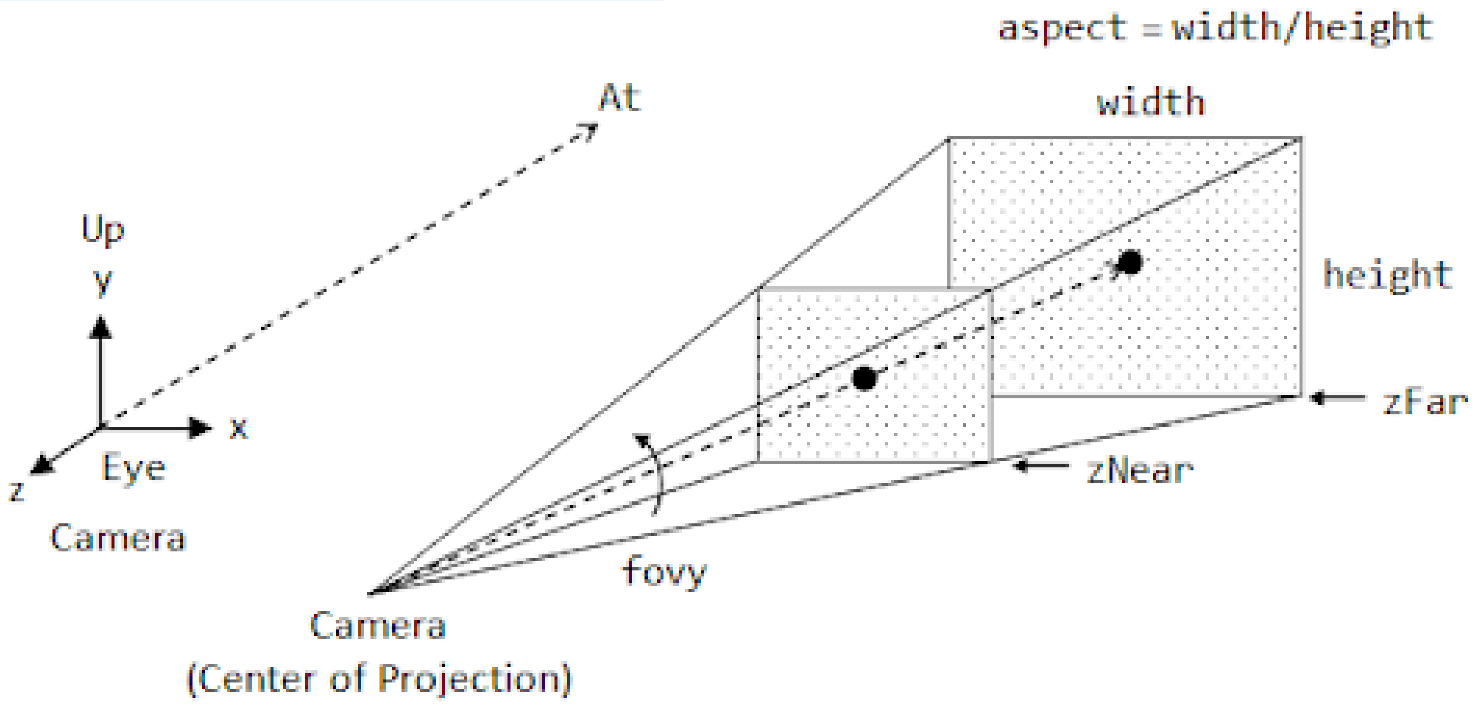
A perspective projection matrix can be created in GLM as follows:


The parameters of `glm::perspective(GLfloat FoV, GLfloat aspect, GLfloat near, GLfloat far)` are as follows:



```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
```

- **FoV →** a realistic view it is usually set to 45 degrees.
- **Aspect →** dividing the viewport's width by its height.
- **Near and Far →** usually set the near distance to 0.1 and the far distance to 100.0.

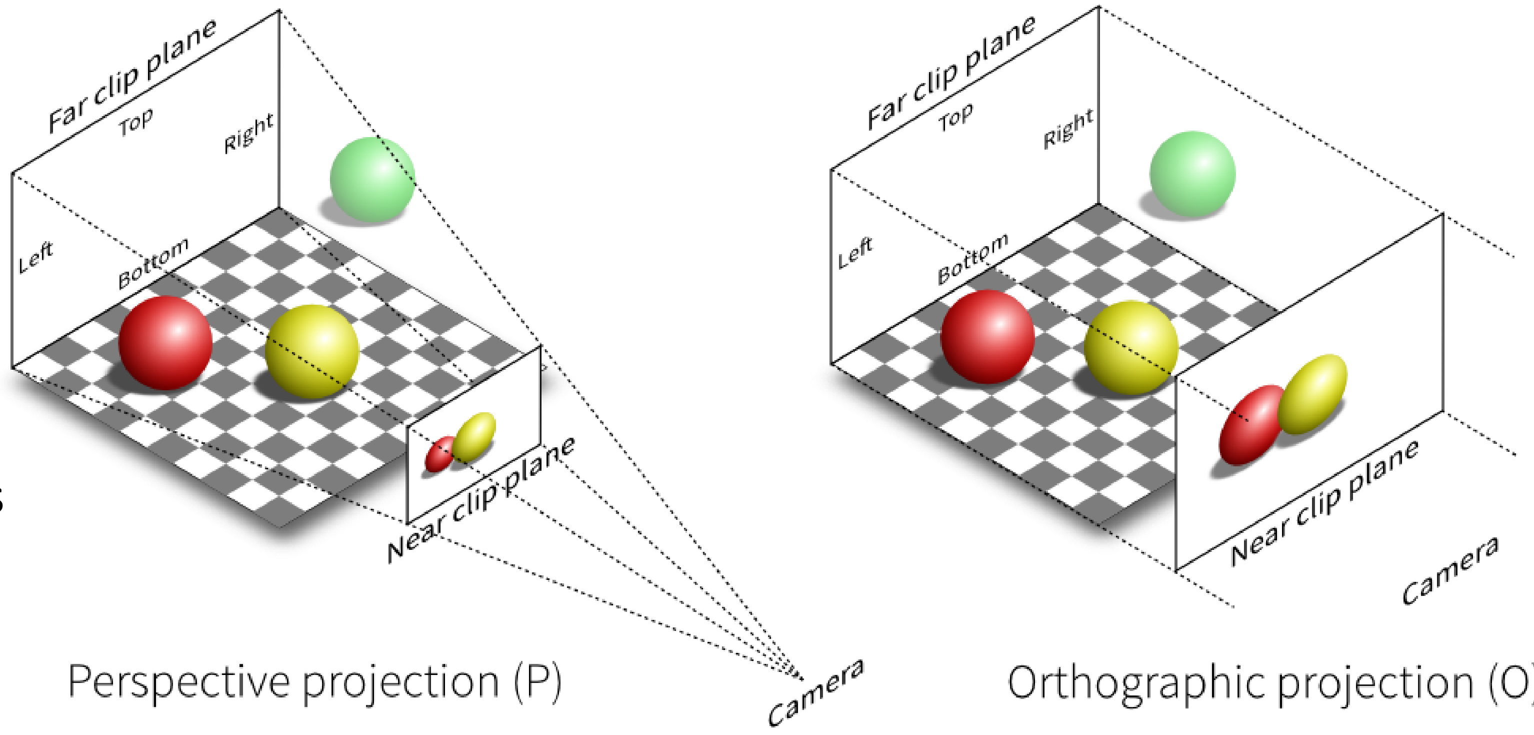


T3D – Coordinate Systems – The global picture

4. Clip Space – Ortographic vs. Perspective

- When using orthographic projection, the w component is not manipulated (it stays 1) and thus has no effect).

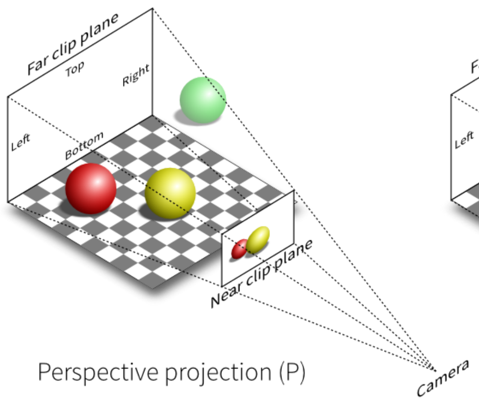
- Because the orthographic projection doesn't use perspective projection, objects farther away do not seem smaller, which produces a weird visual output.



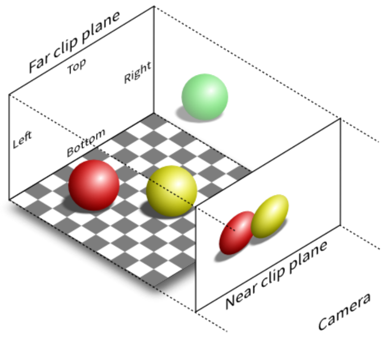
The **orthographic projection** is mainly used for 2D renderings and for some **architectural or engineering applications** where we'd rather **not have vertices distorted by perspective**.

T3D – Coordinate Systems – The global picture

4. Clip Space – Ortographic vs. Perspective



Perspective projection (P)



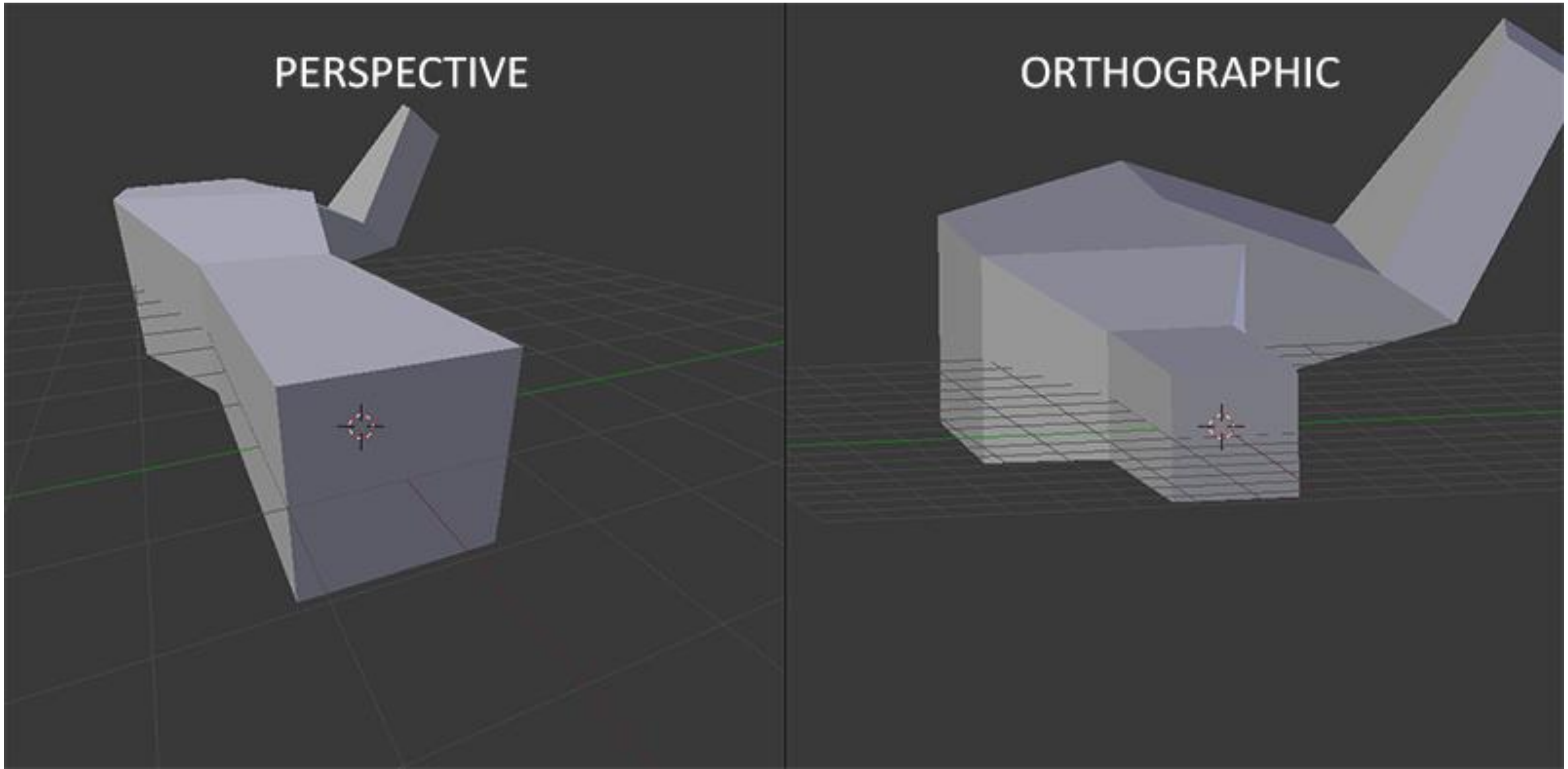
Orthographic projection (O)

Applications like **Blender** that are used for 3D modelling sometimes use **orthographic projection** for modelling, because it more accurately depicts each object's dimensions.

T3D – Coordinate Systems – The global picture

4. Clip Space – Ortographic vs. Perspective

You can see that with **perspective projection**, the vertices farther away appear much smaller, while in **orthographic projection** each vertex has the same distance to the user



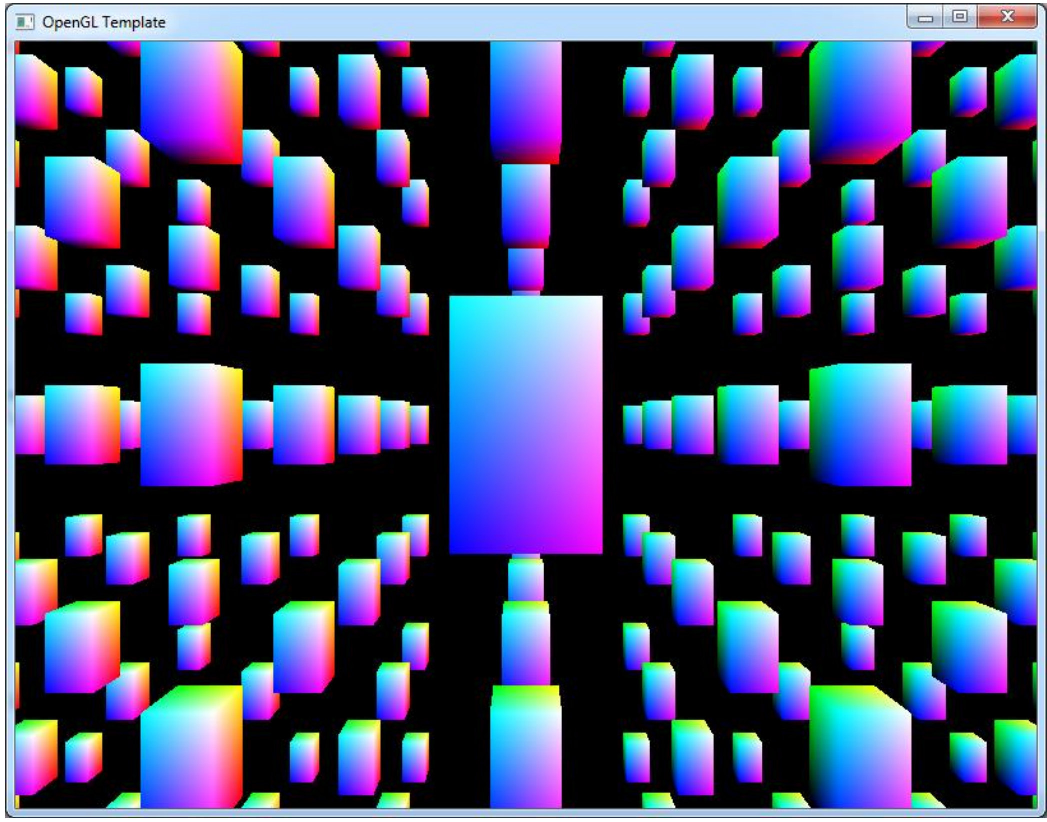
T3D – Coordinate Systems – The global picture

Putting it all together

We create a transformation matrix for each of the aforementioned steps: **model, view and projection matrix**. A vertex coordinate is then transformed to clip coordinates as follows:

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

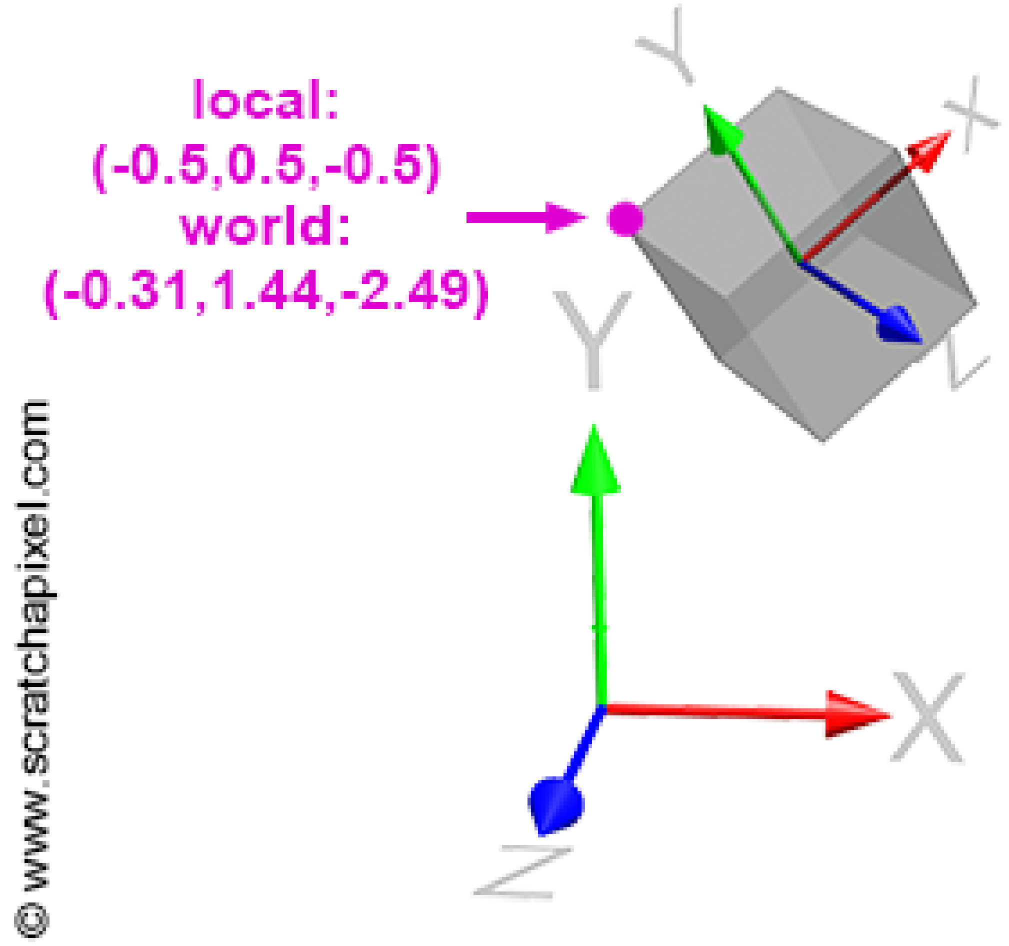
- Note that the order of matrix multiplication is reversed (remember that we need to read matrix multiplication from right to left).
- The resulting vertex should then be assigned to **gl_Position** in the **vertex shader** and OpenGL will then automatically perform perspective division and clipping.



And then?
The output of the vertex shader requires the coordinates to be in clip-space which is what we just did with the transformation matrices. OpenGL then performs *perspective division* on the *clip-space coordinates* to transform them to *normalized-device coordinates*. OpenGL then uses the parameters from `glViewPort` to map the normalized-device coordinates to *screen coordinates* where each coordinate corresponds to a point on your screen (in our case a 800x600 screen). This process is called the *viewport transform*.

T3D – Going 3D – Model Matrix

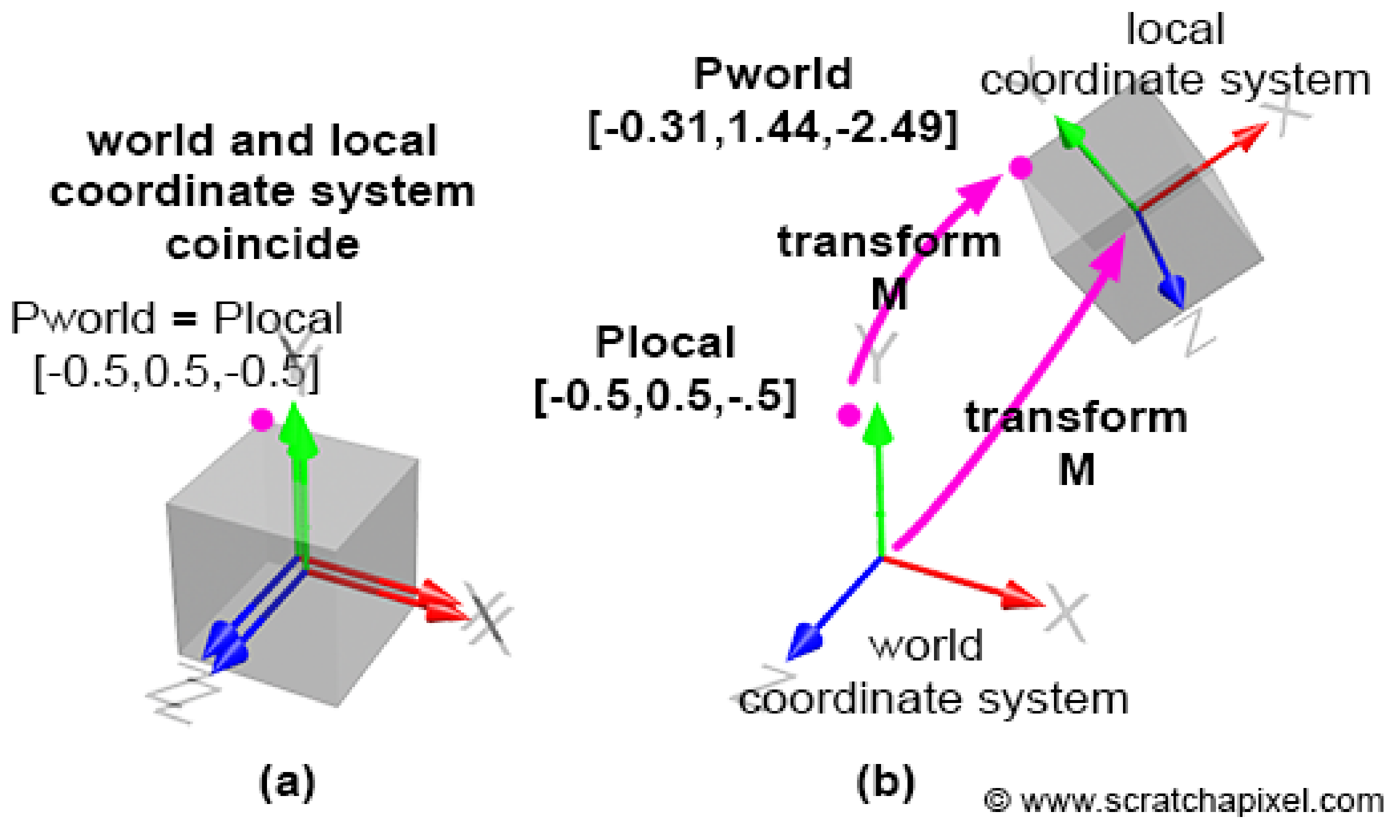
- To start drawing in 3D we'll first create a **model matrix**.
- The model matrix consists of **translations, scaling and/or rotations** we'd like to apply to transform **all object's vertices to the global world space**.
- Let's transform our plane a bit by rotating it on the x-axis so it looks like it's laying on the floor. The model matrix then looks like this:



https://www.scratchapixel.com/lessons/3d-basic-rendering/computing-pixel-coordinates-of-3d-point/mathematics-computing-2d-coordinates-of-3d-points

T3D – Going 3D – Model Matrix

- To start drawing in 3D we'll first create a **model matrix**.
- The model matrix consists of **translations, scaling and/or rotations** we'd like to apply to transform **all object's vertices to the global world space**.
- Let's transform our plane a bit by rotating it on the x-axis so it looks like it's laying on the floor. The model matrix then looks like this:



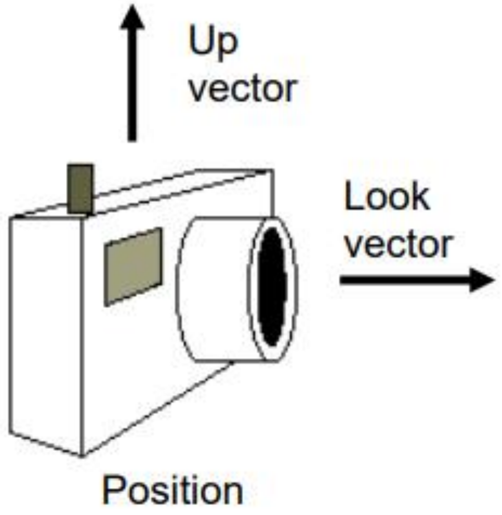
```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

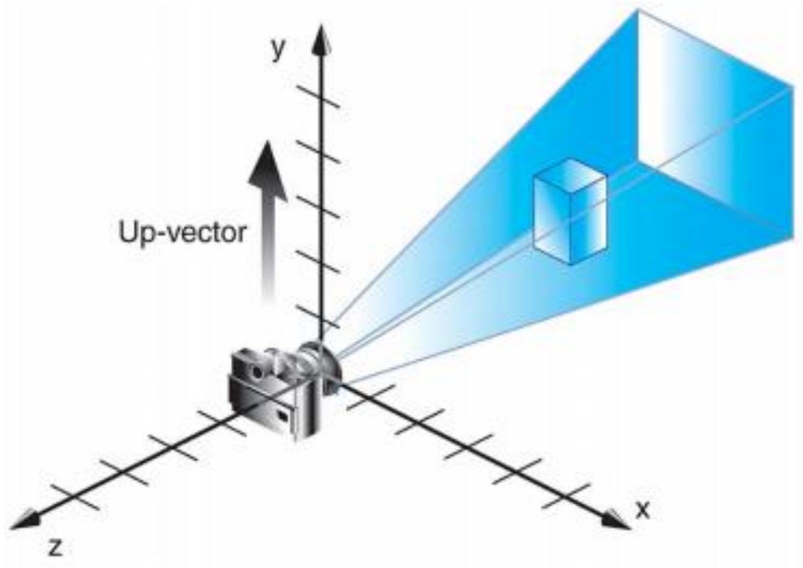
By **multiplying** the vertex coordinates with this model matrix we're transforming the vertex coordinates to **world coordinates**.
Our plane that is slightly on the floor thus represents the plane in the global world.

T3D – Going 3D – View Matrix

Next we need to create a **view matrix**. We want to move slightly backwards in the scene so the object becomes visible (when in world space we're located at the origin (0,0,0))

- To move around the scene, think about the following: **To move a camera backwards, is the same as moving the entire scene forward.**

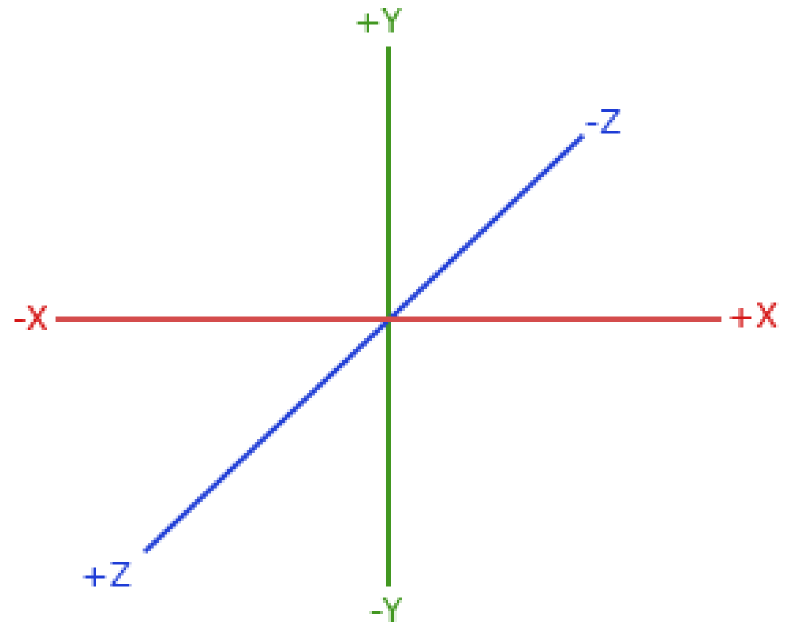


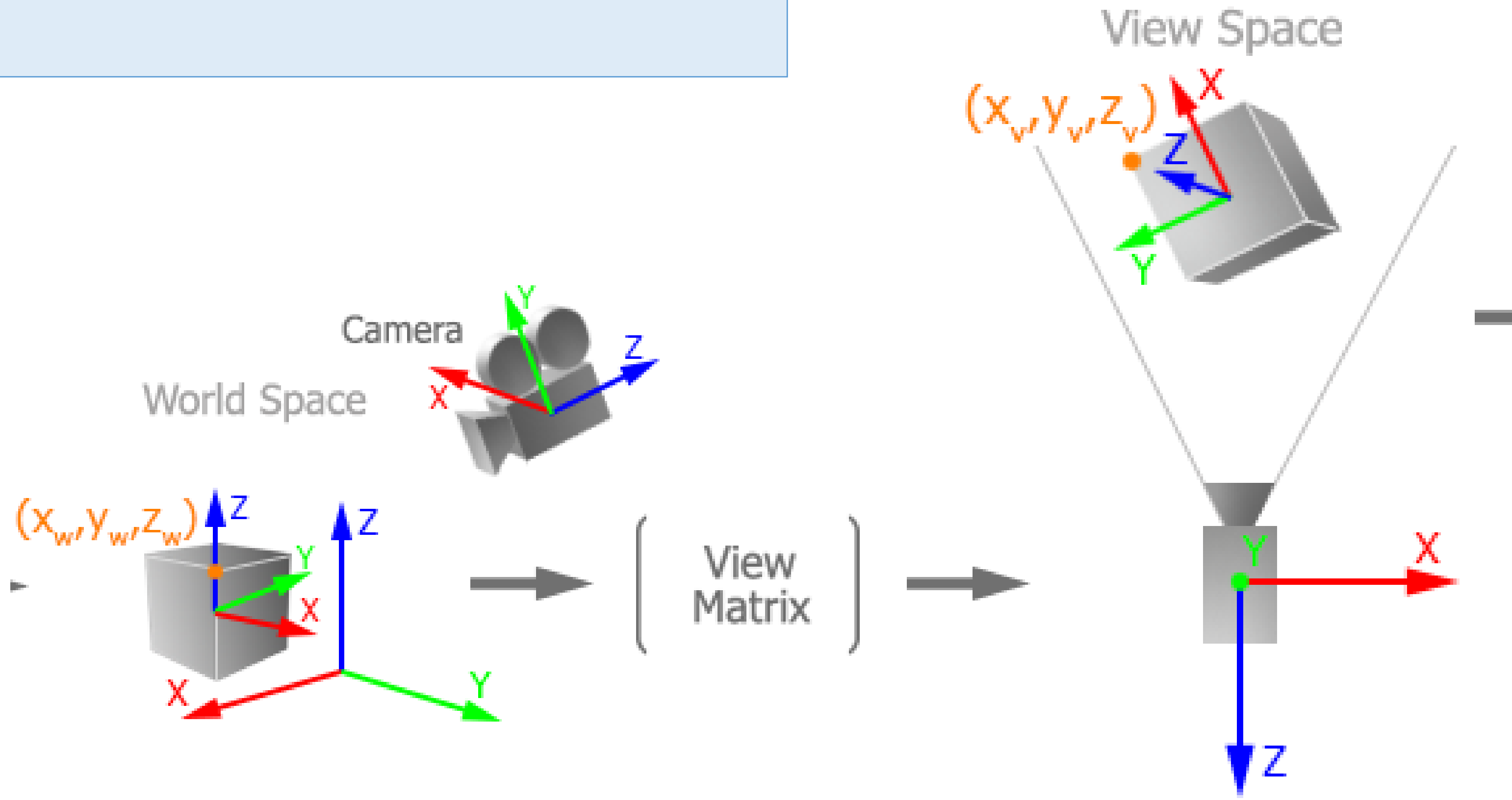


T3D – Going 3D – View Matrix

That is exactly what a view matrix does, we move the entire scene around inversed to where we want the camera to move.

Because we want to move backwards and since **OpenGL is a right-handed system** we have to **move in the positive z-axis**. We do this by translating the scene towards the negative z-axis. This gives the impression that we are moving backwards.

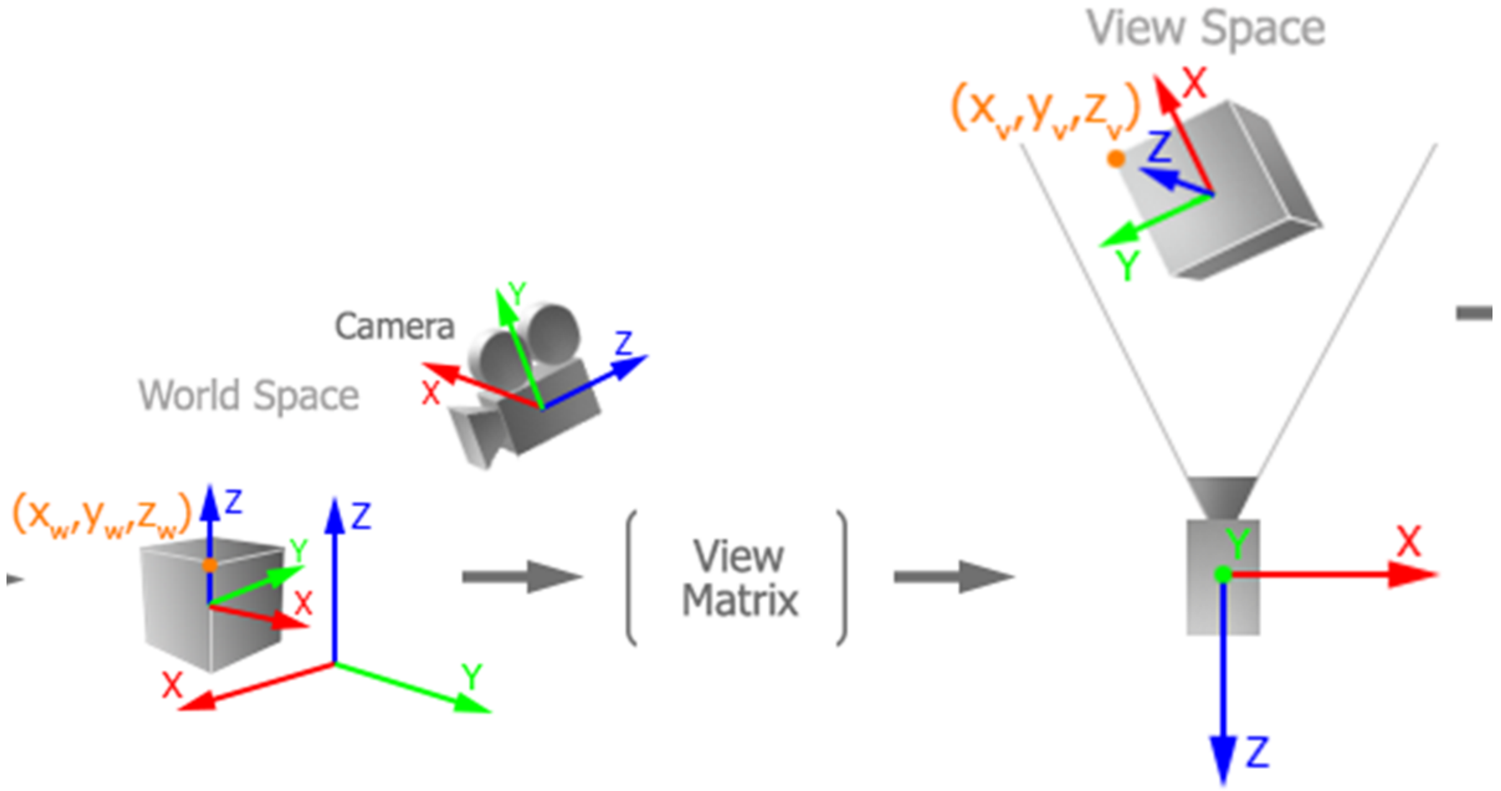




T3D – Going 3D – View Matrix

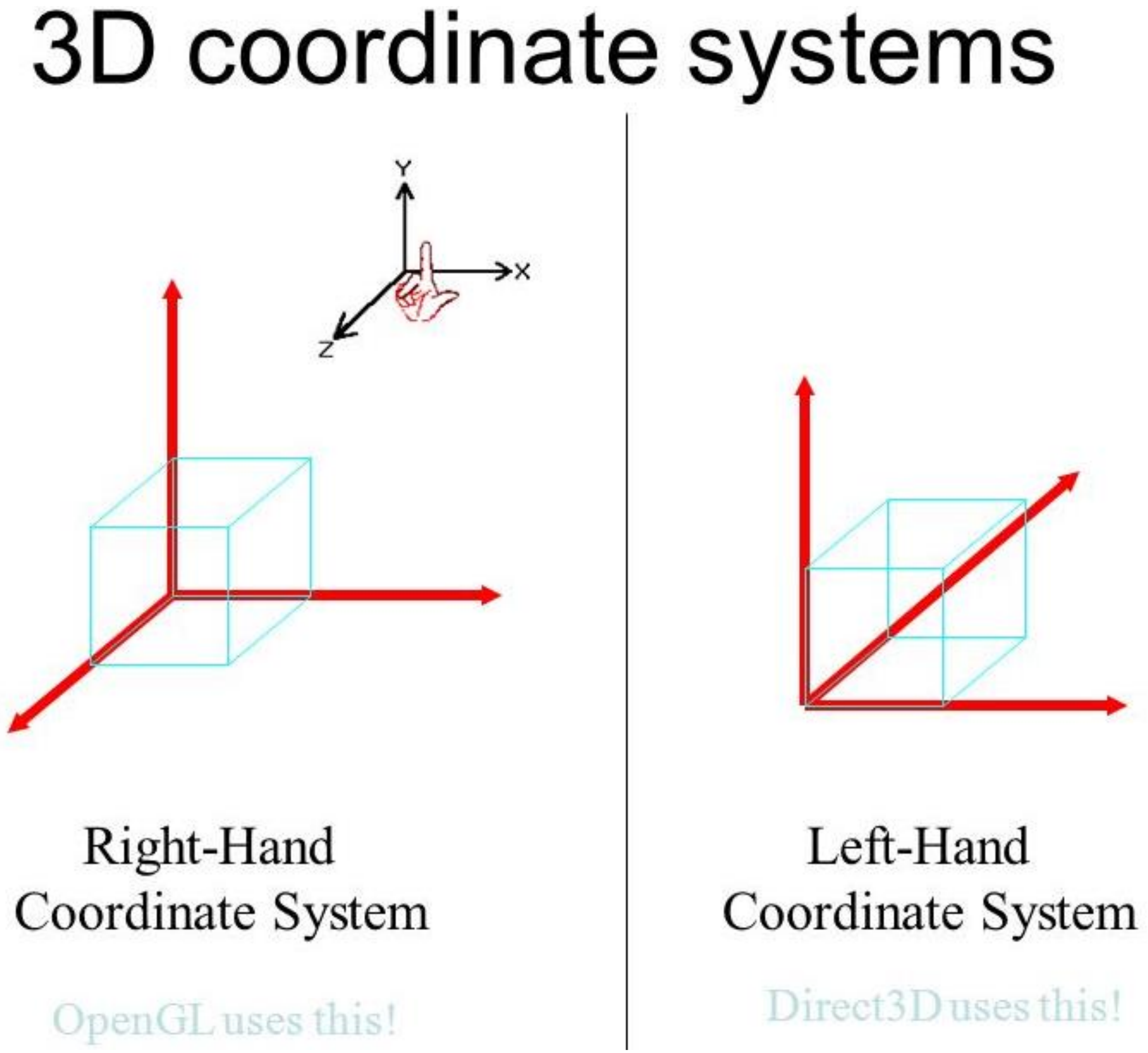
The view matrix looks like this:

```
glm::mat4 view = glm::mat4(1.0f);
// note that we're translating the scene in the reverse direction of where we want to move
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```



T3D – Going 3D – Coordinate Systems

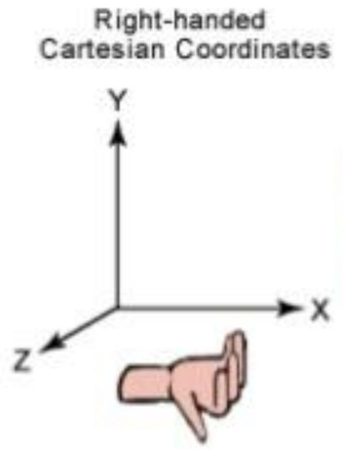
- By convention, **OpenGL is a right-handed system**.



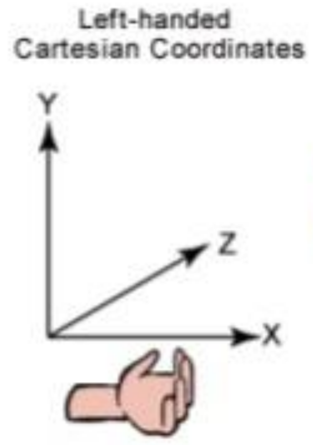
T3D – Going 3D – Coordinate Systems

Conventional OpenGL Handedness

- Right-handed
  - Object space
  - Eye space
- Left-handed
  - Clip space
  - Normalized Device Coordinate (NDC) space
  - Window space



In eye space, eye is "looking down" the negative Z axis

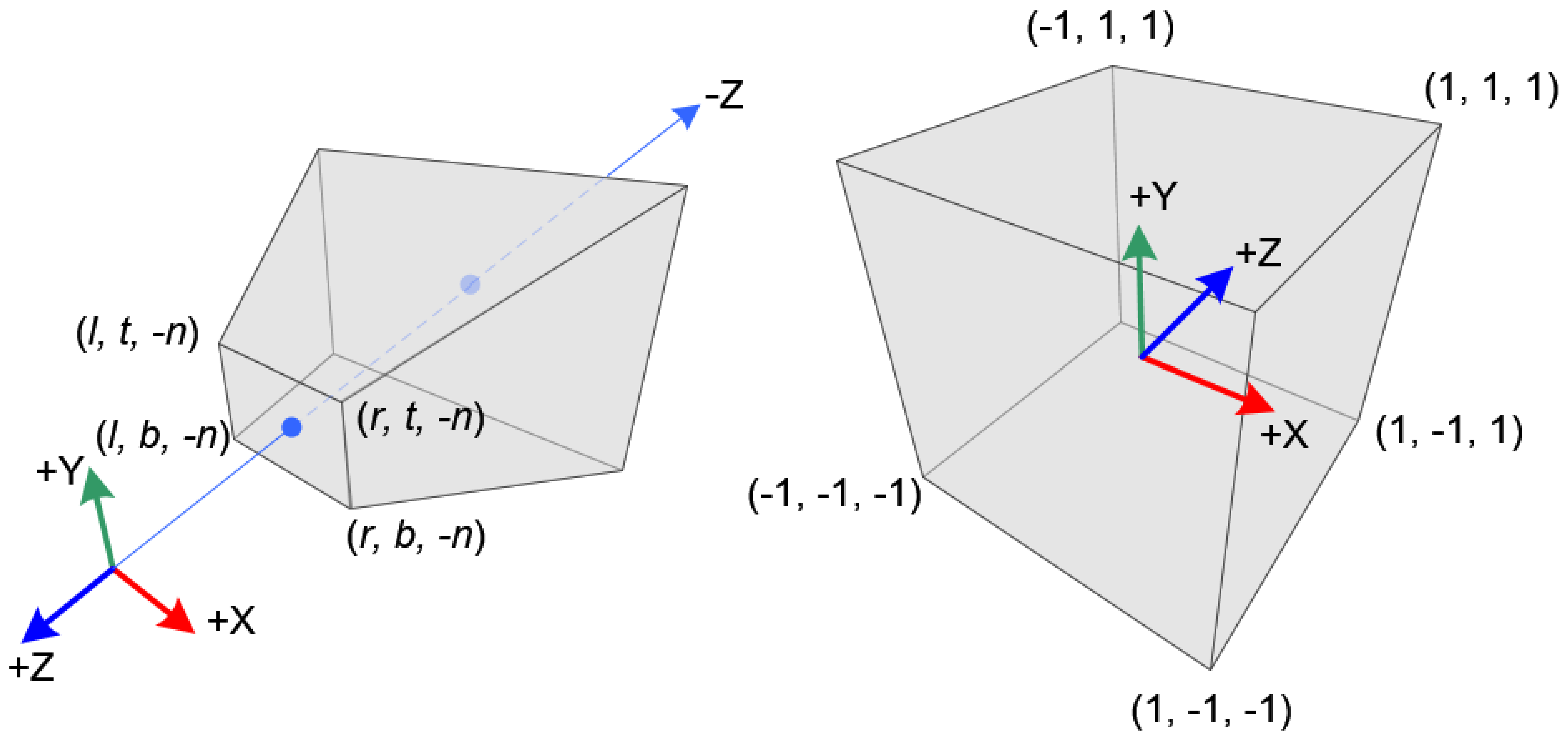


Positive depth is further from viewer

Note that in **normalized device coordinates** OpenGL actually uses **a left-handed system** (the projection matrix switches the handedness).

T3D – Going 3D – Coordinate Systems

Note that in **normalized device coordinates** OpenGL actually uses **a left-handed system** (the projection matrix switches the handedness).

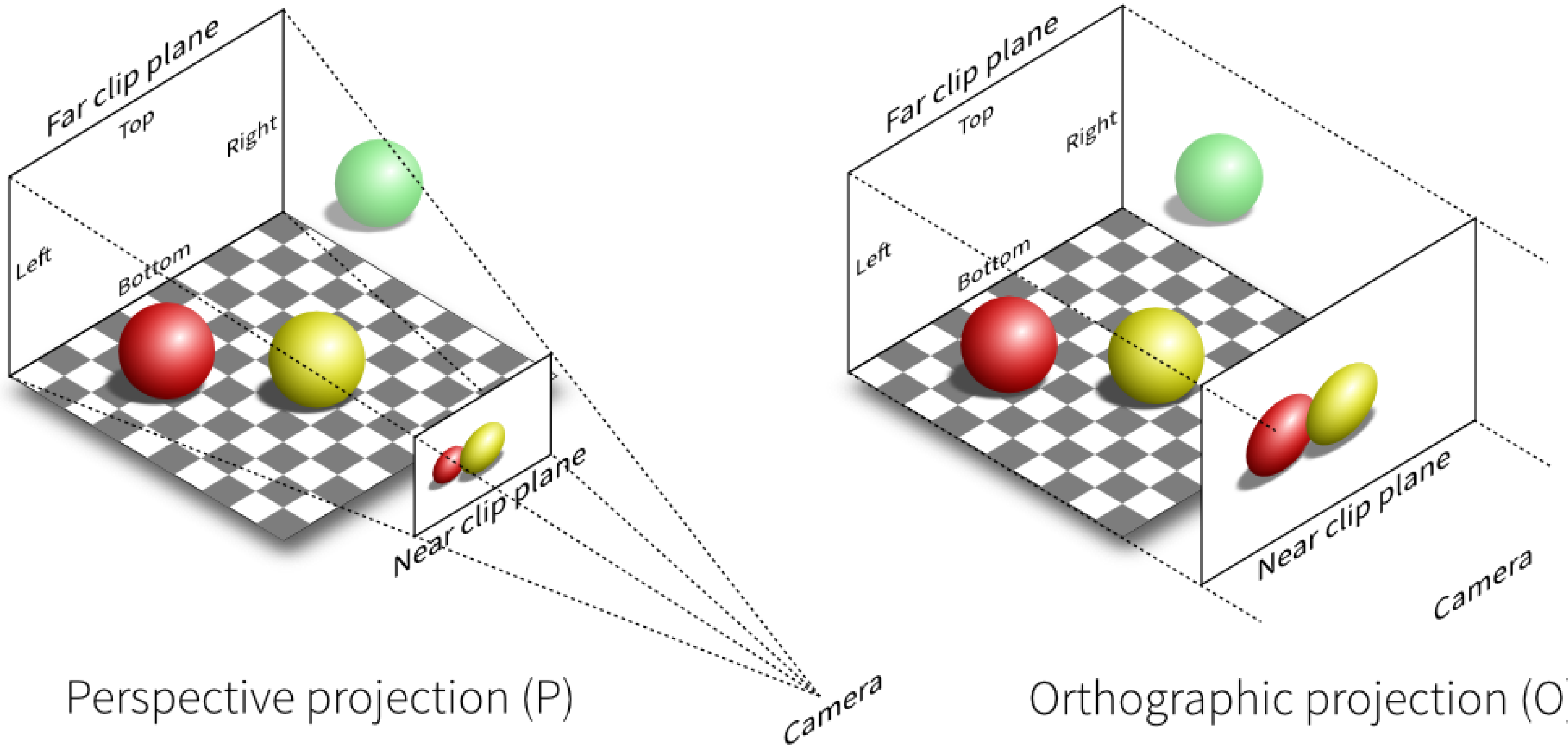


Perspective Frustum and Normalized Device Coordinates (NDC)

T3D – Going 3D – Projection Matrix

The last thing we need to define is the projection matrix. We want to use **perspective projection** for our scene so we'll declare the projection matrix like this:

```
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
```



Perspective projection (P)

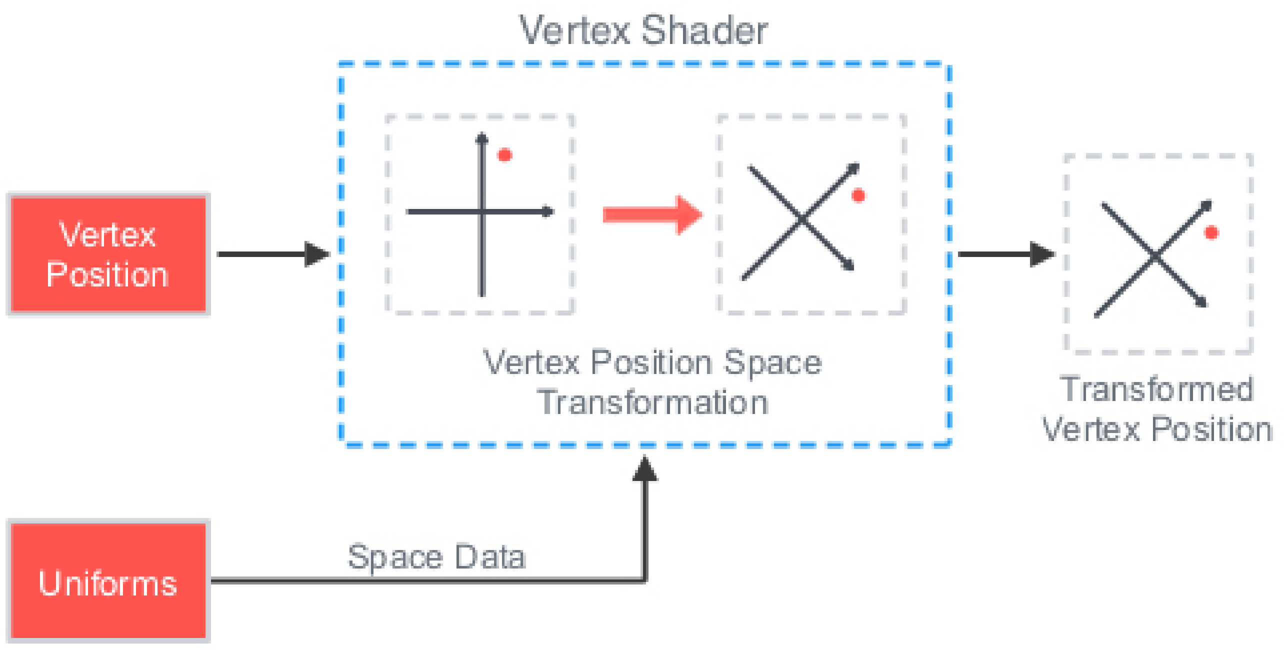
Orthographic projection (O)

T3D – Going 3D – Shaders

Now that we created the transformation matrices we should pass them to our **shaders**.

- First let's declare the **transformation matrices** as **uniforms** in the vertex shader and multiply them with the vertex coordinates:

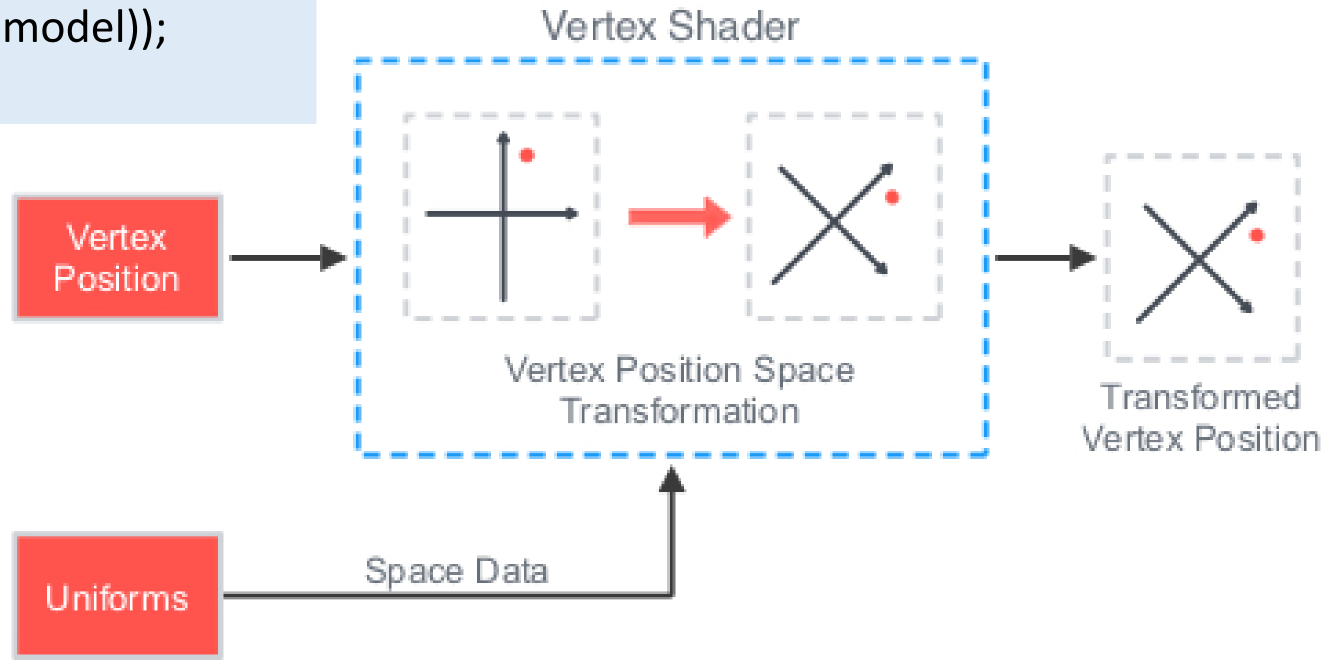
```
#version 330 core
layout (location = 0) in vec3 aPos;
...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
   // note that we read the multiplication from right to left
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   ...
}
```


Vertex Shader
Vertex Position
Vertex Position Space Transformation
Transformed Vertex Position
Uniforms
Space Data

T3D – Going 3D – Shaders

We should also send the matrices to the shader (this is usually done each frame since transformation matrices tend to change a lot):

```
int modelLoc = glGetUniformLocation(ourShader.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
... // same for View Matrix and Projection Matrix
```



Vertex Shader
Vertex Position
Vertex Position Space Transformation
Transformed Vertex Position
Uniforms
Space Data

Now that our vertex coordinates are transformed via the model, view and projection matrix the final object should be:

- Tilted backwards to the floor.
- A bit farther away from us.
- Be displayed with perspective (it should get smaller, the further its vertices are).