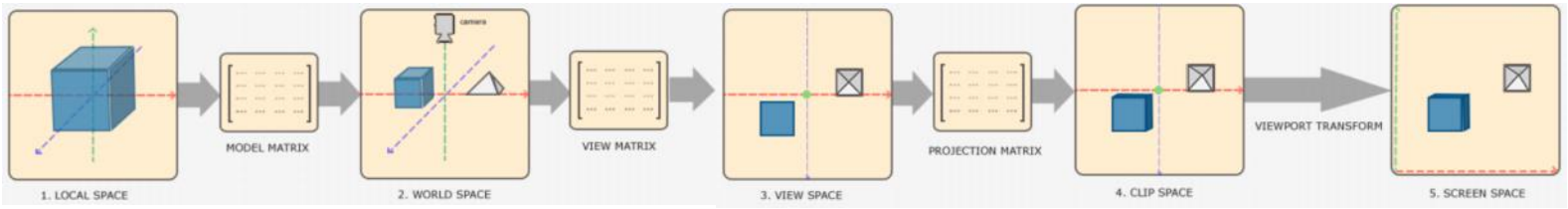
COORDINATE SYSTEMS

The x, y, and z coordinates of each vertex should be in the form of Normalized Device Coordinates (NDC), meaning they should be between -1.0 to 1.0, otherwise they will not be visible.
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 NDC. These NDC are then given to the rasterizer to transform them to 2D coordinates/pixels on the screen.

Transforming coordinates to NDC is usually done step-by-step, where we transform an object's vertices to several coordinate systems before finally transforming them to NDC.
- Some operations/calculations are easier to use or make more sense in certain coordinate systems.
  - Example: When modifying an object, it makes the most sense to do so in local coordinates, while calculating certain operations on the object with respect to the position of other objects makes more sense in world coordinates.
- There are 5 different coordinate systems that are of importance:
  1. Local space (or Object space)
  2. World space
  3. View space (or Camera/Eye space)
  4. Clip space
  5. Screen space

The Global Picture
To transform coordinates from one space to the next, we use several transformation matrices.
- The most important transformation matrices are the model, view, and projection matrix.



1. **Local Coordinates** are the coordinates of an object relative to its local origin; they're the coordinates the object begins in.
2. **World-space Coordinates** are the coordinates of an object with respect to a larger world. These coordinates are relative to some global origin of the world, together with many other objects placed relative to the world's origin.
3. **View-space Coordinates** are the coordinates as seen from the camera or viewer's point of view.
4. **Clip-space Coordinates** are processed to NDC and determine which vertices will end up on the screen. Projection to clip-space coordinates can add perspective if using perspective projection.
5. **Screen-space Coordinates** are transformed to using a process called **viewport transform** that transforms the coordinates from `-1.0` and `1.0` to the coordinate range defined by `glViewport`. The resulting coordinates are then sent to the rasterizer to turn them into fragments.

Local Space
Local space is the coordinate space that is local to your object, i.e. where your object begins in.
- Example: When you create a model in blender, its origin probably centered at `(0, 0, 0)`, even though the model may end up somewhere else in the world in your application. Therefore, all the vertices of your model are in local space, because they are relative to the model itself.

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.

World Space
We want to define a position for each object to position them inside a larger world, otherwise all the objects would just be overlayed on each other at the world's origin.
- This is the coordinate space where you want your objects transformed to in such a way that they form your actual (game) world.
- The coordinates of your object are transformed from local coordinates to world coordinates using the model matrix.

The model matrix is a transformation matrix that scales, rotates, and/or translates your object to place it in the world at the location/orientation it should be.
You could think of the transformation matrix used in the previous chapter, which was used to move the container object all over the screen, as a sort of model matrix; we transformed the local coordinates of the container to some different place in the scene/world.

View Space
The view space is what people usually refer to as the camera of OpenGL.
- The view space is the space from the camera's point of view.
- This is usually accomplished with a combination of translations and rotations to transform the scene so that certain items are in front of the camera.
  - These transformations are stored inside the view matrix that transforms from world coordinates to view coordinates.

Clip Space
At the end of each vertex shader run, OpenGL expects the coordinates to be within a specific range, known as a **clipping volume**. Any coordinate not in that range is **clipped**, discarding it. Hence, *clip* space.
- We specify our own coordinate set to work in and convert those back to NDC as OpenGL expects them.
- The projection matrix transforms vertex coordinates from view space to clip space.
  - The projection matrix *projects* 3D coordinates to the 2D NDC.
  - It does so by specifying a range (e.g. -1000 to 1000) and then transforming coordinates in that range to NDC. Any vertices outside this range will be clipped.
  - **NOTE:** If only 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 volume.

The viewing box a projection matrix creates is called a **frustum**.
Once all the vertices are transformed to clip space, **perspective division** is performed; we divide the x, y, and z components of the position vectors by the vector's homogeneous w component, transforming the 4D clip space coordinates to 3D NDC.
- This is performed automatically at the end of the vertex shader step.

After this stage, the resulting vertices are transformed (automatically) to screen space coordinates, using the settings of `glViewport`, and turned into fragments.

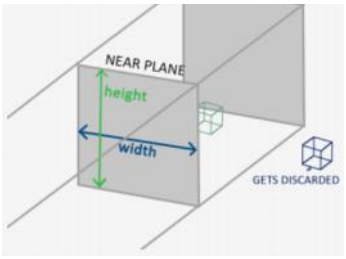
The projection matrix can take on two different forms: orthographic or perspective.

Orthographic Projection
An **orthographic projection matrix** defines a cube-like frustum box.
- When creating an orthographic projection matrix, we specify the width, height, and length (view distance) of the visible frustum.
- Produces unrealistic, but stylized, results since the projection doesn't take perspective into account.

Orthographic Projection

An **orthographic projection matrix** defines a cube-like frustum box.

- When creating an orthographic projection matrix, we specify the width, height, and length (view distance) of the visible frustum.
- Produces unrealistic, but stylized, results since the projection doesn't take perspective into account.



The frustum defines the visible coordinates and is specified with a width and height, and a near and far clipping plane.

- The orthographic frustum directly maps all coordinates inside the frustum to NDC without any side effects because it doesn't touch the w component of the transformed vector.

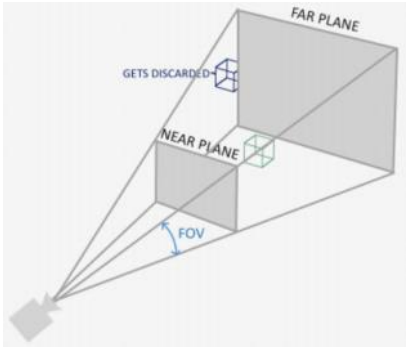
Create an orthographic projection matrix for the window.

- Creating an Orthographic Projection Matrix

Perspective Projection

Perspective is the idea that objects that are farther away appear much smaller. A perspective projection matrix tries to mimic this.

- The perspective 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.
- Once the coordinates are in clip space, perspective division is applied to the clip space coordinates, and then the resulting coordinates are in NDC.



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

Create a perspective projection matrix for the window.

- Creating a Perspective Projection Matrix

More info at: http://www.songho.ca/opengl/gl_projectionmatrix.html

Putting It All Together

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

- $V_{clip} = M_{projection} * M_{view} * M_{model} * V_{local}$
  - **NOTE:** The order of matrix multiplication is reversed. You know this.

Going 3D

Create a model matrix that rotates the container object so it appears to be tilting back.

- Model Matrix

Create a view matrix that zooms the camera out a little bit.

- Moving a camera backwards is the same as moving the entire scene forward.
- **NOTE:** OpenGL is a **right-handed system**, meaning the positive z direction is towards you and the negative z direction is away from you.
  - **ALSO NOTE:** In NDC, OpenGL actually uses a left-handed system.
- View Matrix

Create a projection matrix that uses perspective projection.

- Projection Matrix

Integrate the above transformation matrices into the vertex shader so the container object is properly transformed.

- Model/View/Projection Integration

Make the container object a cube and rotate it over time on an angle.

- Save yourself some time and stress and get the vertices from here: Cube Vertices
- Cube Rotation

OpenGL gives no guarantee on the order of triangles rendered, so faces of 3D objects may be overwritten by other faces of the same object.

Z-Buffer

OpenGL stores depth information in a buffer called the z-buffer, or **depth buffer**, which allows OpenGL to decide when to draw over a pixel and when not to.

- The depth is stored within each fragment as the fragment's z value, and whenever the fragment wants to output its color, OpenGL compares its depth values with the z-buffer.
- If the current fragment is behind the other fragment, it is discarded, otherwise it is overwritten. This is known as **depth testing**.
- GLFW automatically creates a z-buffer, however, depth testing is not enabled in OpenGL by default.

Enable depth testing.

- Enabling Depth Testing

More Cubes!

If we want to add more cubes to the world, we need only modify the model matrix with each cube's position, rotation, and/or scale.

Add 9 more cubes to the world in varying positions with varying rotations.

- Take a look at the world coords for each new cube:
- Adding More Objects to the World

EXERCISES

1. Try experimenting with the FoV and aspect-ratio parameters of GLM's projection function. See if you can figure out how those affect the perspective frustum.
2. Play with the view matrix by translating in several directions and see how the scene changes. Think of the view matrix as a camera object.
3. Try to make every 3rd container (include the 1st) rotate over time, while leaving the other containers static using just the model matrix.

Creating an Orthographic Projection Matrix

Wednesday, March 23, 2022 8:04 PM

```
// Creates the orthographic projection matrix
glm::mat4 orthoProjMat = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

Creating a Perspective Projection Matrix

Wednesday, March 23, 2022 8:28 PM

```
// Creates the perspective projection matrix
glm::mat4 perspProjMat = glm::perspective(glm::radians(90.0f), (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);
```

Model Matrix

Wednesday, March 23, 2022 8:47 PM

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

View Matrix

Wednesday, March 23, 2022 8:48 PM

```
// Creates the view matrix
glm::mat4 view = glm::mat4(1.0f);
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

Projection Matrix

Wednesday, March 23, 2022 9:05 PM

```
// Creates the projection matrix using perspective projection
glm::mat4 projection = glm::perspective(glm::radians(90.0f), (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);
```

NOTE: I created constants for the width and height of the window/viewport.

Model/View/Projection Integration

Wednesday, March 23, 2022 9:08 PM

Vertex Shader

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   ...
}
```

Render Loop

Sending the transformation matrices to the vertex shader is usually done each frame since transformation matrices tend to change a lot.

```
// Sends the model, view, and projection matrices to the vertex shader
glUniformMatrix4fv(glGetUniformLocation(shaderProgram.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(glGetUniformLocation(shaderProgram.id, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(shaderProgram.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
```

Cube Rotation

Wednesday, March 23, 2022 10:27 PM

Cube Vertices

Yes, it would be better to use an EBO for duplicate vertices, but this is just for demonstration purposes.

```
// Vertices of a cube
float vertices[] = {
//  Positions            Colors                        Texture Coords
   -0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,
    0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
    0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
    0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
   -0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
   -0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,

   -0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,
    0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
    0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
    0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
   -0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
   -0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,

   -0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
   -0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
   -0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
   -0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
   -0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,
   -0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,

    0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
    0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
    0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
    0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
    0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,
    0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,

   -0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
    0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
    0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
    0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
   -0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,
   -0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,

   -0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f,
    0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 1.0f,
    0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
    0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       1.0f, 0.0f,
   -0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 0.0f,
   -0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 1.0f, 1.0f,       0.0f, 1.0f
};
```

Render Loop

```
// Creates the model matrix
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, (float)glfwGetTime() * glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));

...

glDrawArrays(GL_TRIANGLES, 0, 36);
```

Enabling Depth Testing

Wednesday, March 23, 2022 10:37 PM

Remember, you must enable the depth buffer bit, otherwise OpenGL won't display the faces of 3D objects on the screen correctly.

```
glEnable(GL_DEPTH_TEST);
```

Render Loop

Since we're using a depth buffer, we also want to clear the depth buffer before each render iteration.
Otherwise, the depth buffer information of the previous frame stays in the buffer.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Adding More Objects to the World

Wednesday, March 23, 2022 10:59 PM

```
// Positions of cube objects in world space
glm::vec3 cubePositions[] = {
    glm::vec3( 0.0f,  0.0f,  0.0f),
    glm::vec3( 2.0f,  5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3( 2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f,  3.0f, -7.5f),
    glm::vec3( 1.3f, -2.0f, -2.5f),
    glm::vec3( 1.5f,  2.0f, -2.5f),
    glm::vec3( 1.5f,  0.2f, -1.5f),
    glm::vec3(-1.3f,  1.0f, -1.5f)
};
```

Render Loop

Since we want to draw more than one cube, we put the draw function in a for loop and manipulate the model matrix before each draw call.

```
// Draws a cube with a specified position and rotation 10 times
for (int i = 0; i < 10; ++i) {
   glm::mat4 model = glm::mat4(1.0f);
   model = glm::translate(model, cubePositions[i]);
   model = glm::rotate(model, glm::radians(20.0f * i), glm::vec3(1.0f, 0.3f, 0.5f));
   glUniformMatrix4fv(glGetUniformLocation(shaderProgram.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glDrawArrays(GL_TRIANGLES, 0, 36);
}
```