Computer Graphics

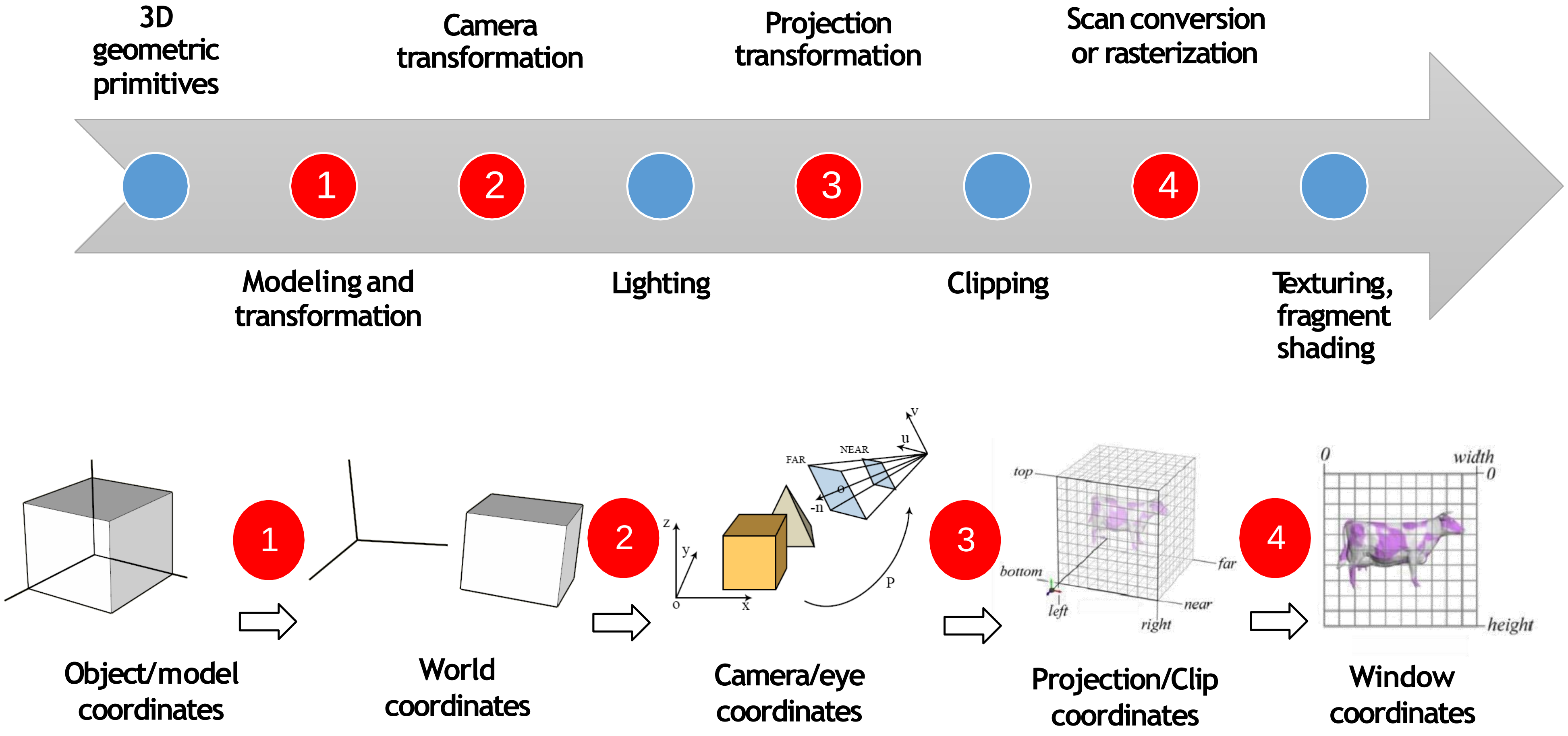
3.02 Object World View Projection

Dr Joan Llobera – joanllobera@enti.cat

Outline

1. Transform operations
    1. Translate
    2. Scale
    3. Rotation
    4. Composing operations
2. Model, View and Projection Matrices

Reminder. Transformation operations



1. Transform operations

Matrix multiplication, in practice:

In C++, with GLM:

```
glm::mat4 myMatrix;
glm::vec4 myVector;
// fill myMatrix and myVector somehow
glm::vec4 transformedVector = myMatrix * myVector; // Again, in this order ! this is important
```

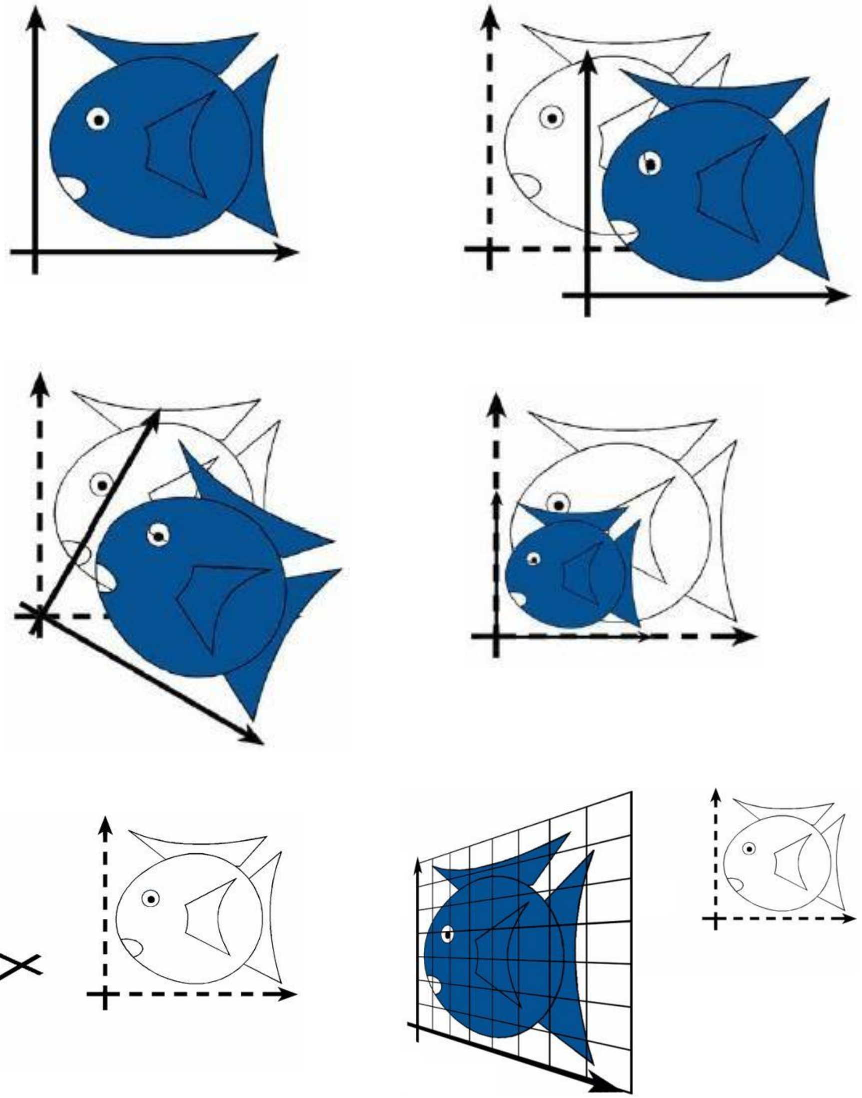
In GLSL :

```
mat4 myMatrix;
vec4 myVector;
// fill myMatrix and myVector somehow
vec4 transformedVector = myMatrix * myVector; // Yeah, it's pretty much the same than GLM
```

1. Transformation operations

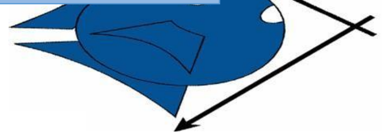
Identity matrix

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$



In C++, with GLM:

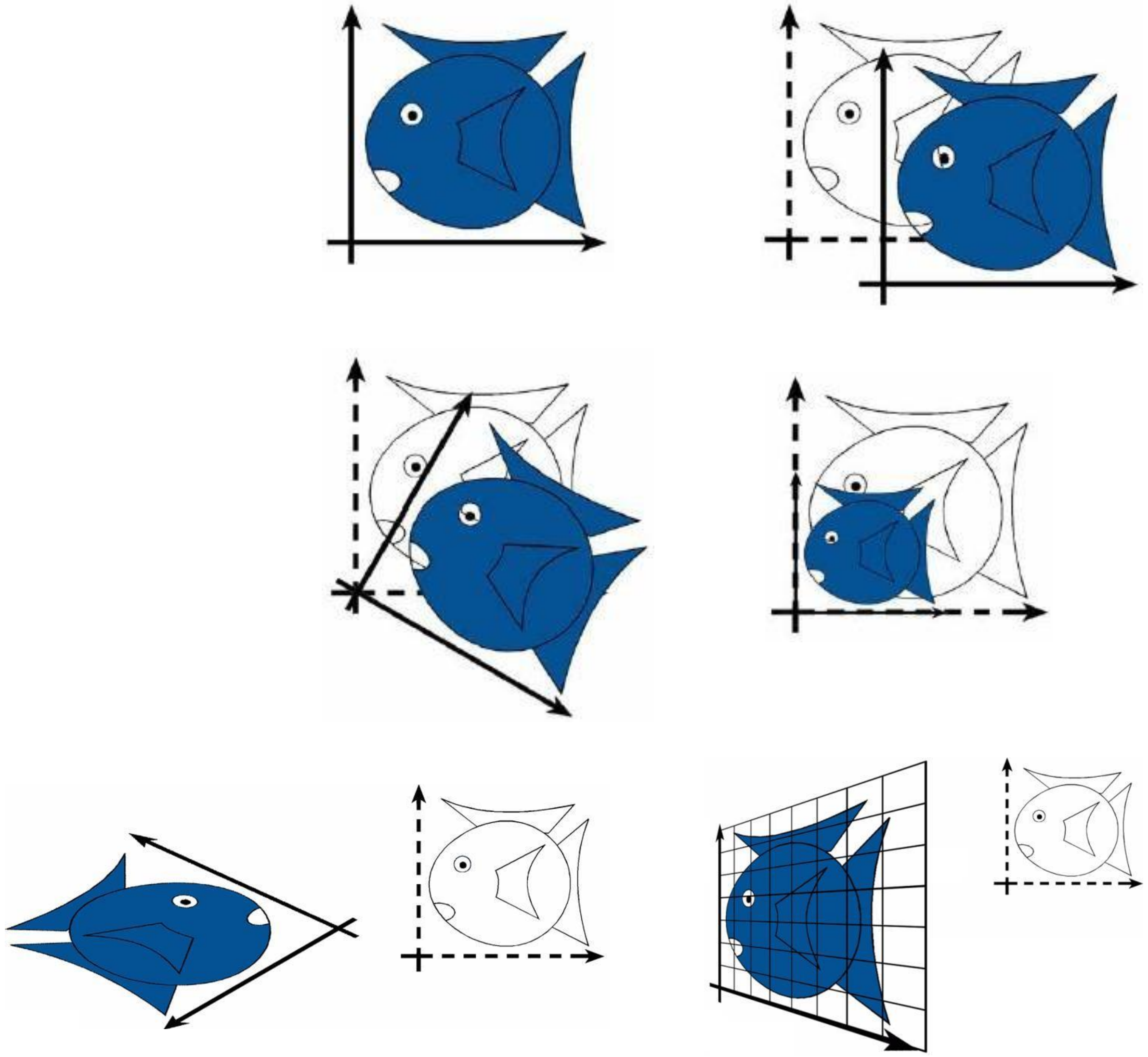
```
glm::mat4 myIdentityMatrix = glm::mat4(1.0f);
glm::vec4 transformedVector = myMatrix * myVector;
```



1. Transformation operations

Translation matrix

$$
\begin{matrix}
1 & 0 & 0 & X \\
0 & 1 & 0 & Y \\
0 & 0 & 1 & Z \\
0 & 0 & 0 & 1
\end{matrix}
$$



1. Transform operations

Translate

In C++, with GLM:

```
glm::mat4 myMatrix = glm::translate(glm::mat4(), glm::vec3(X, Y, Z));
glm::vec4 transformedVector = myMatrix * myVector; // guess the result
```

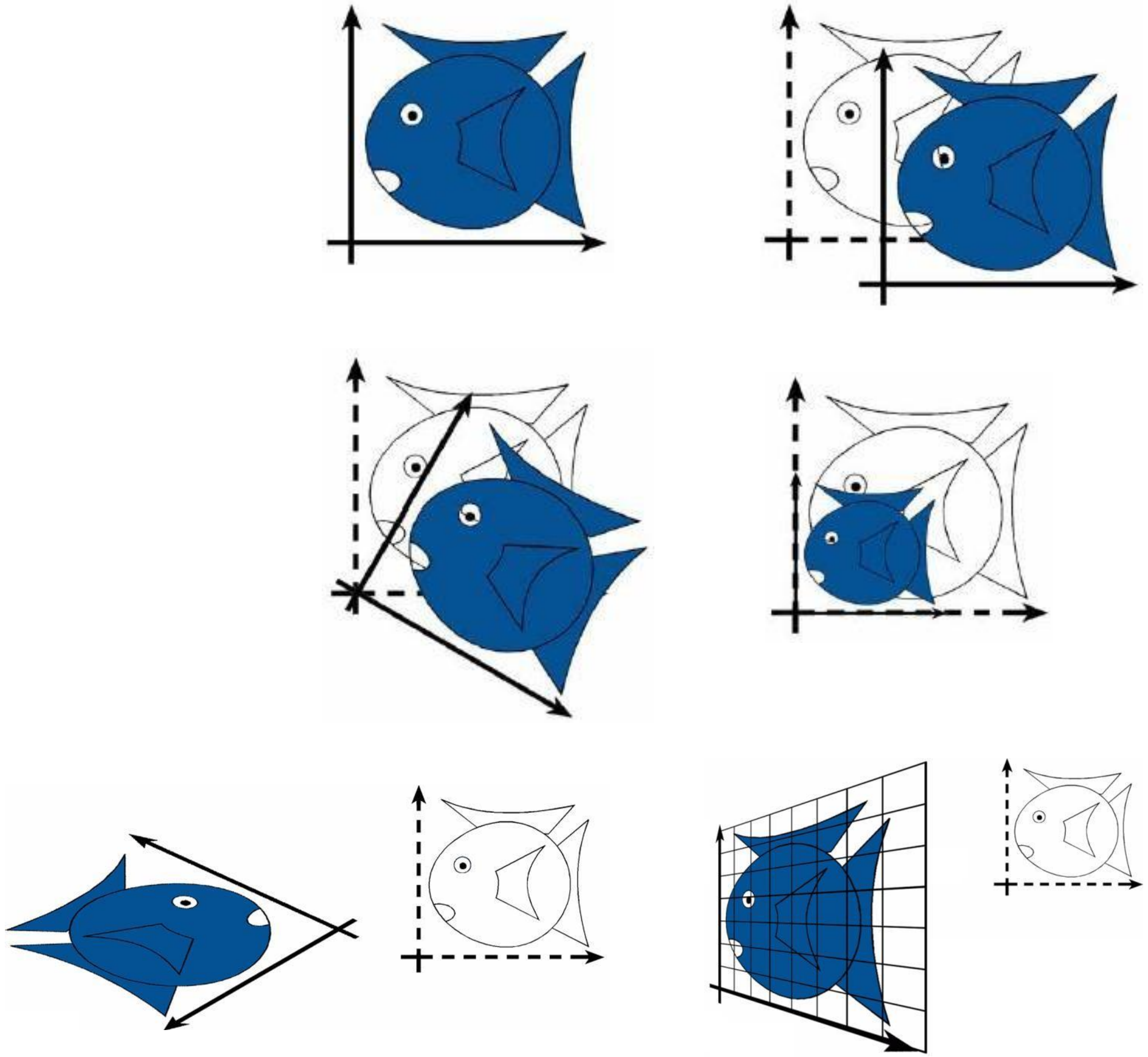
In GLSL :

```
vec4 transformedVector = myMatrix * myVector; // Yeah, it's pretty much the same than GLM
```

1. Transformation operations

Scaling matrix

$$\begin{matrix} X & 0 & 0 & 0 \\ 0 & Y & 0 & 0 \\ 0 & 0 & Z & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$



1. Transform operations

Scale

In C++, with GLM:

```
glm::mat4 myScalingMatrix = glm::scale(glm::mat4(), glm::vec3(X, Y, Z));
glm::vec4 transformedVector = myMatrix * myVector;
```

In GLSL :

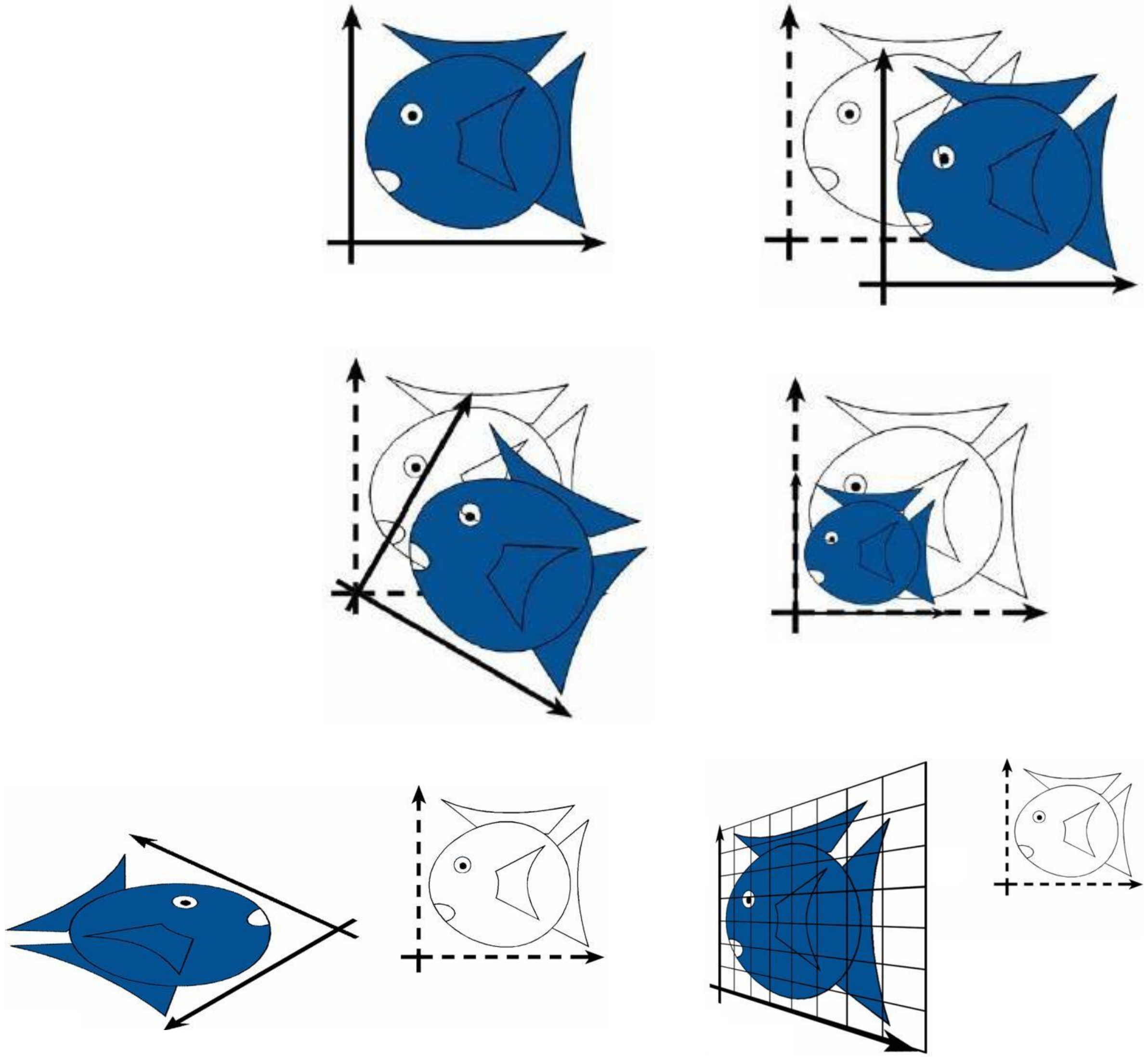
```
vec4 transformedVector = myMatrix * myVector; // Yeah, it's pretty much the same than GLM
```

1. Transformation operations

Rotation matrix

(x axis)

$$Rx= \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos & \sin & 0 \\ 0 & -\sin & \cos & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

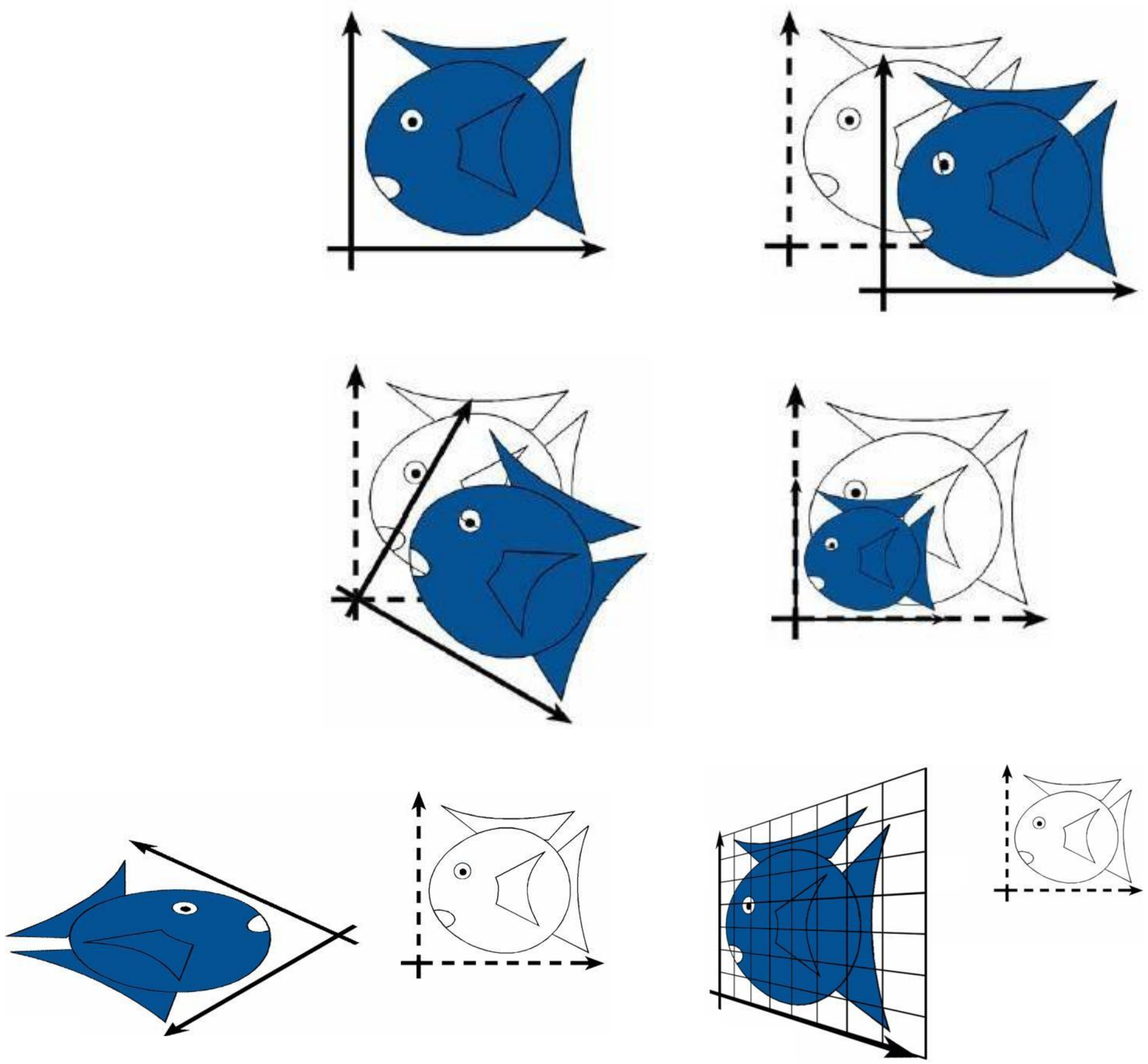


1. Transformation operations

Rotation matrix

(y axis)

$$
Ry= \begin{matrix} \cos & 0 & -\sin & 0 \\ 0 & 1 & 0 & 0 \\ \sin & 0 & \cos & 0 \\ 0 & 0 & 0 & 1 \end{matrix}
$$

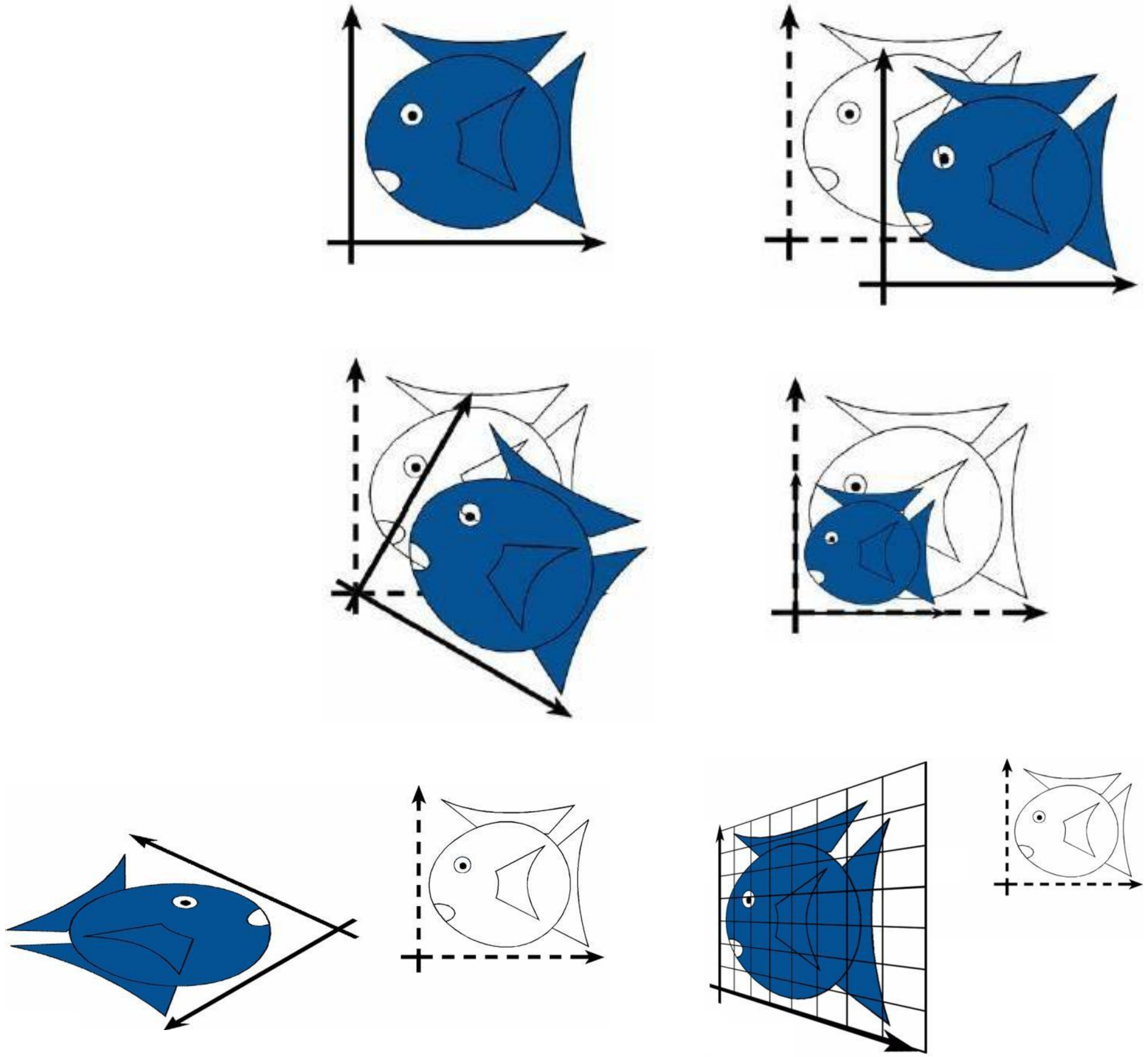


1. Transformation operations

Rotation matrix

(z axis)

$$
Rz=\begin{array}{cccc}
\cos & -\sin & 0 & 0 \\
\sin & \cos & 0 & 0 \\
 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{array}
$$



1. Transformation operations

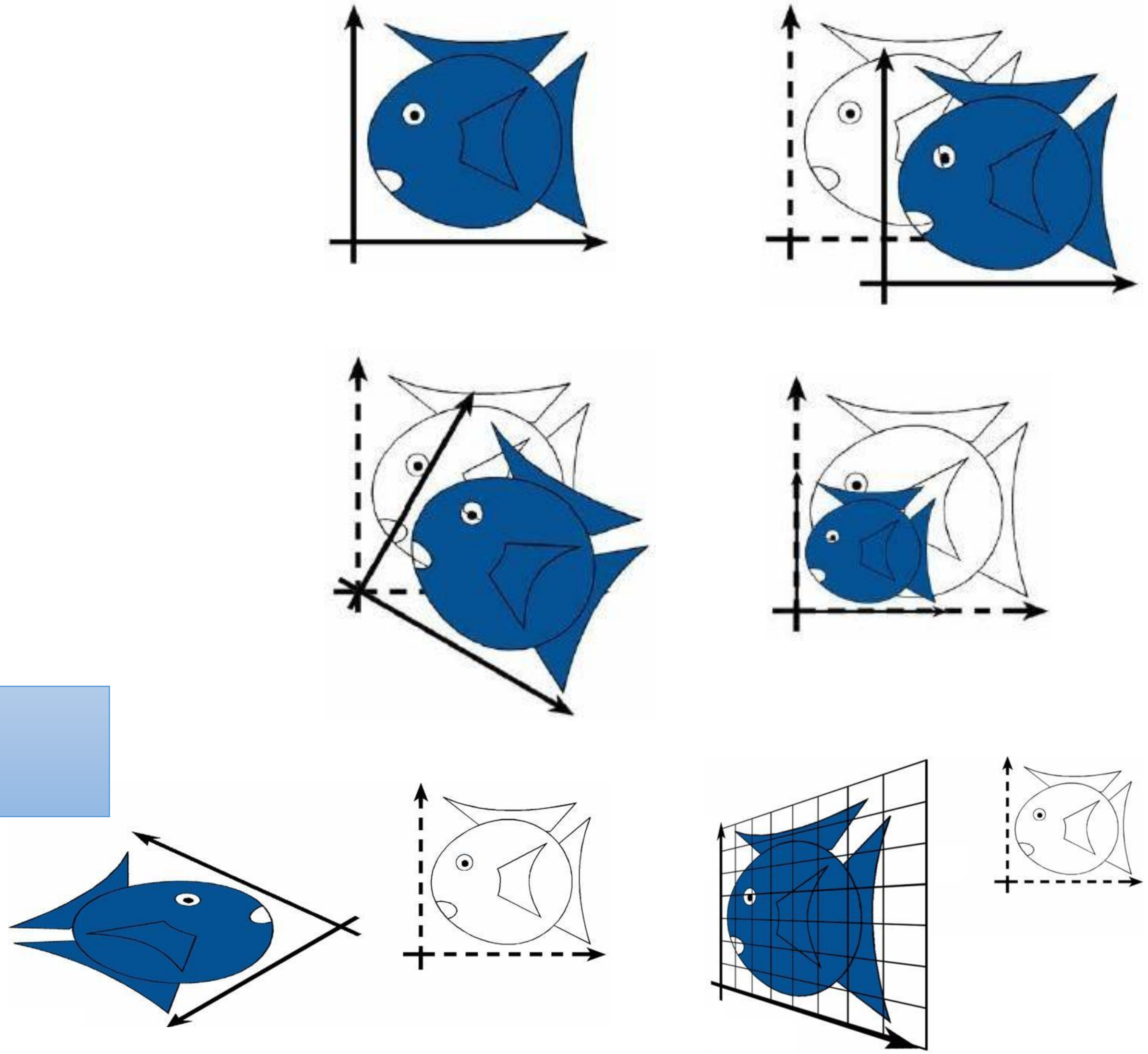
Rotation matrices can be multiplied:

$Rz(\gamma)Ry(\beta)Rx(\alpha)$

Beware! The order matters

An arbitrary axis, also possible:

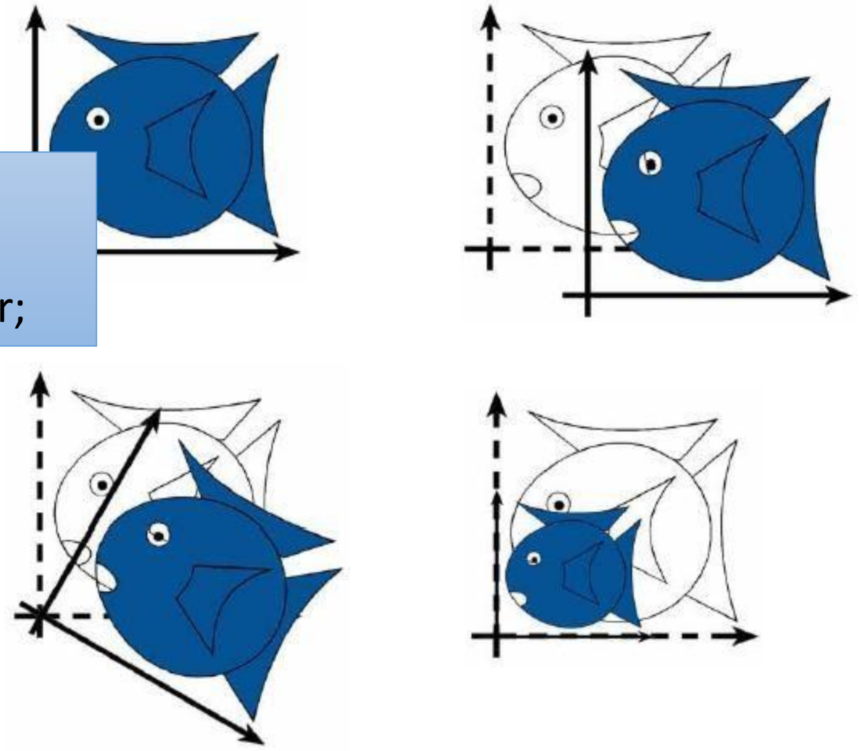
```
glm::vec3 myRotationAxis(X,Y,Z);
 glm::rotate( angle_in_degrees, myRotationAxis );
```



1. Transformation operations

Composing Operations

```
glm::mat4 myModelMatrix =
myTranslationMatrix * myRotationMatrix * myScaleMatrix;
glm::vec4 myTransformedVector = myModelMatrix * myOriginalVector;
```

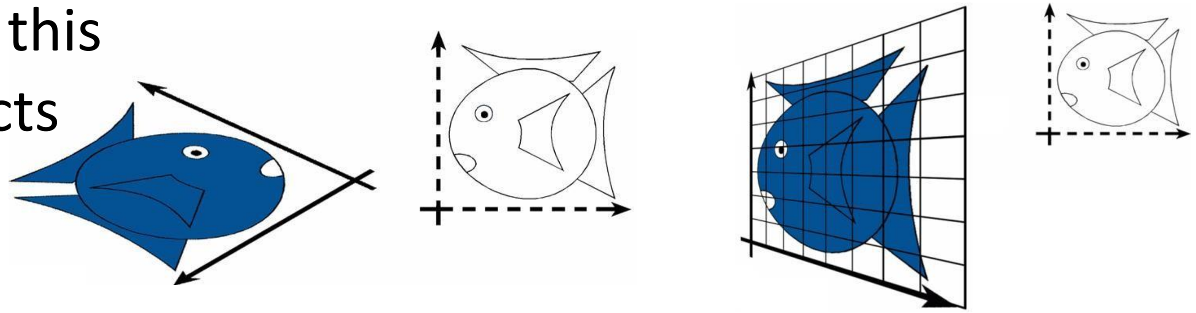


Be careful!

- The scaling is done FIRST,
- The rotation SECOND
- The translation third

Why?

- Because matrix multiplication works like this
- Because the order makes sense for objects



1. Transformation operations

Composing Operations

Why have we defined this order of operations?

- As a matter of fact, the order above is what you will usually need for game characters and other items : Scale it first if needed; then set its direction, then translate it. For instance, given a ship model (rotations have been removed for simplification) :
- The wrong way :
  - You translate the ship by (10,0,0). Its center is now at 10 units of the origin.
  - You scale your ship by 2. Every coordinate is multiplied by 2 *relative to the origin*, which is far away… So you end up with a big ship, but centered at 2*10 = 20. Which you don't want.
- The right way :
  - You scale your ship by 2. You get a big ship, centered on the origin.
  - You translate your ship. It's still the same size, and at the right distance.

1. Transform operations

Exercises.

1. Draw two cubes.
2. Make the second cube change colour
   - Red or black following sin(currentTime)
3. Make the second cube move along the y axis
   - When up it should be red, when down it should be black
4. Make the second cube scale to the double of the original size and go back
   - When up it should be double size, when down it should be simple size
5. Make the second cube rotate along the y axis
6. Make the second cube rotate along the first cube

Ex1.

```
void draw2Cubes(double currentTime) {
glEnable(GL_PRIMITIVE_RESTART);
glBindVertexArray(cubeVao);
glUseProgram(cubeProgram);
glm::mat4 t = glm::translate(glm::mat4(), glm::vec3(-1.0f, 2.0f, 3.0f));
objMat = t;
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "objMat"), 1, GL_FALSE, glm::value_ptr(objMat));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "mv_Mat"), 1, GL_FALSE, glm::value_ptr(RenderVars::_modelView));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "mvpMat"), 1, GL_FALSE, glm::value_ptr(RenderVars::_MVP));
glUniform4f(glGetUniformLocation(cubeProgram, "color"), 0.1f, 1.f, 1.f, 0.f);
glDrawElements(GL_TRIANGLE_STRIP, numVerts, GL_UNSIGNED_BYTE, 0);
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.0f, 3.0f));
objMat = t;
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "objMat"), 1, GL_FALSE, glm::value_ptr(objMat));
glDrawElements(GL_TRIANGLE_STRIP, numVerts, GL_UNSIGNED_BYTE, 0);
glUseProgram(0);
glBindVertexArray(0);
glDisable(GL_PRIMITIVE_RESTART);
}
```

Ex2.

```
void draw2Cubes(double currentTime) {
glEnable(GL_PRIMITIVE_RESTART);
glBindVertexArray(cubeVao);
glUseProgram(cubeProgram);
glm::mat4 t = glm::translate(glm::mat4(), glm::vec3(-1.0f, 2.0f, 3.0f));
objMat = t;
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "objMat"), 1, GL_FALSE, glm::value_ptr(objMat));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "mv_Mat"), 1, GL_FALSE, glm::value_ptr(RenderVars::_modelView));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "mvpMat"), 1, GL_FALSE, glm::value_ptr(RenderVars::_MVP));
glUniform4f(glGetUniformLocation(cubeProgram, "color"), 0.1f, 1.f, 1.f, 0.f);
glDrawElements(GL_TRIANGLE_STRIP, numVerts, GL_UNSIGNED_BYTE, 0);
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.0f, 3.0f));
objMat = t;
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "objMat"), 1, GL_FALSE, glm::value_ptr(objMat));
float red = 0.5f + 0.5f*sin(3.f*currentTime);
glUniform4f(glGetUniformLocation(cubeProgram, "color"),red, 0.f, 0.f, 0.f);
glDrawElements(GL_TRIANGLE_STRIP, numVerts, GL_UNSIGNED_BYTE, 0);
glUseProgram(0);
glBindVertexArray(0);
glDisable(GL_PRIMITIVE_RESTART);
}
```

Ex3.

Change:

```
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.0f, 3.0f));
objMat = t;
```

With:

```
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.5f + 2.f*sin(3.f*currentTime), 3.0f));
objMat = t;
```

Ex4.

Change:

```
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.5f + 2.f*sin(3.f*currentTime), 3.0f));
objMat = t;
```

With:

```
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.5f + 2.f*sin(3.f*currentTime), 3.0f));
float size = 1.5f + 0.5f*sin(3.f*currentTime);
glm::mat4 s = glm::scale(glm::mat4(), glm::vec3(size, size, size));
objMat = t*s;
```

Ex5.

Change:

```
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.5f + 2.f*sin(3.f*currentTime), 3.0f));
float size = 1.5f + 0.5f*sin(3.f*currentTime);
glm::mat4 s = glm::scale(glm::mat4(), glm::vec3(size, size, size));
objMat = t*s;
```

With:

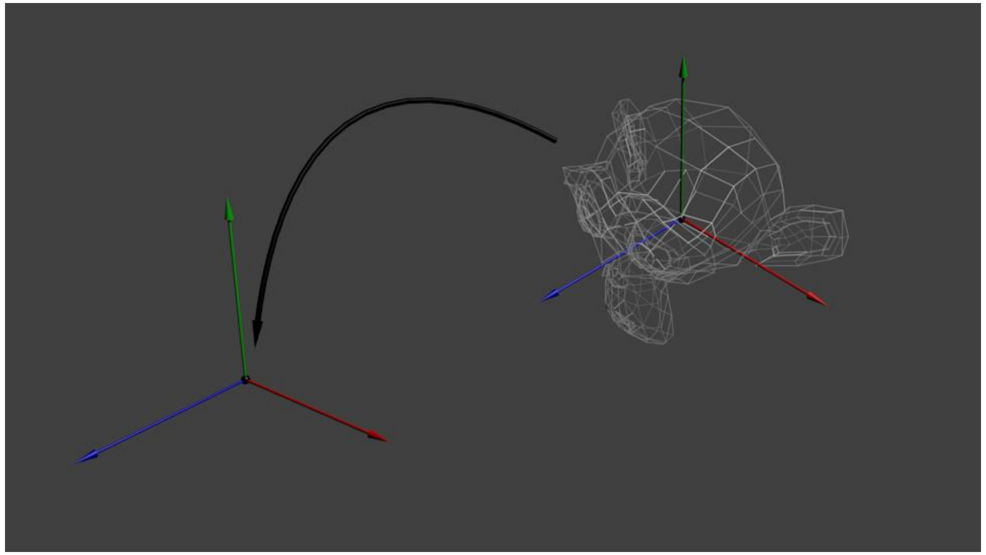
```
t = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.5f + 2.f*sin(3.f*currentTime), 3.0f));
float size = 1.5f + 0.5f*sin(3.f*currentTime);
glm::mat4 s = glm::scale(glm::mat4(), glm::vec3(size, size, size));
glm::mat4 r = glm::rotate(glm::mat4(), 1.f*(float)sin(3.f*currentTime), glm::vec3(0.0f, 1.0f,
0.0f));
objMat = t*r*s;
```

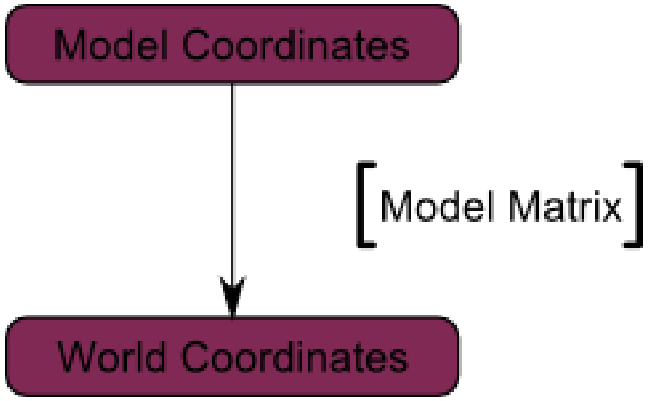
Ex6.

```
glm::mat4 t = glm::translate(glm::mat4(), glm::vec3(-3.0f, 2.0f, 3.0f));
objMat = t;
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "objMat"), 1, GL_FALSE, glm::value_ptr(objMat));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "mv_Mat"), 1, GL_FALSE,
glm::value_ptr(RenderVars::_modelView));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram, "mvpMat"), 1, GL_FALSE,
glm::value_ptr(RenderVars::_MVP));
glUniform4f(glGetUniformLocation(cubeProgram, "color"), 0.1f, 1.f, 1.f, 0.f);
glDrawElements(GL_TRIANGLE_STRIP, numVerts, GL_UNSIGNED_BYTE, 0);


float red = 0.5f +  0.5f*sin(3.f*currentTime);

//glm::mat4 t2 = glm::translate(glm::mat4(), glm::vec3(1.0f, 2.5f + 2.f*sin(3.f*currentTime), 3.0f));
glm::mat4 t2 = glm::translate(glm::mat4(), glm::vec3(1.0f, 0.f , 3.0f));
float size = 1.0f;// 1.5f + 0.5f*sin(3.f*currentTime);
glm::mat4 s = glm::scale(glm::mat4(), glm::vec3(size, size, size));

glm::mat4 r = glm::rotate(glm::mat4(), 2.f*(float)sin(3.f*currentTime), glm::vec3(0.0f, 1.0f, 0.0f));

objMat = t*r*(t2)*s;
```

2. Model, View, Projection

Model and World Space

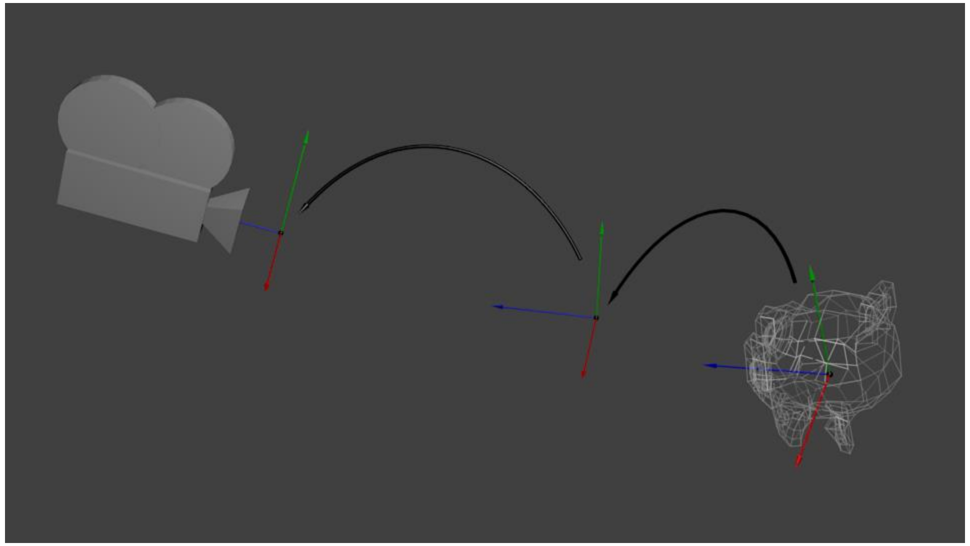


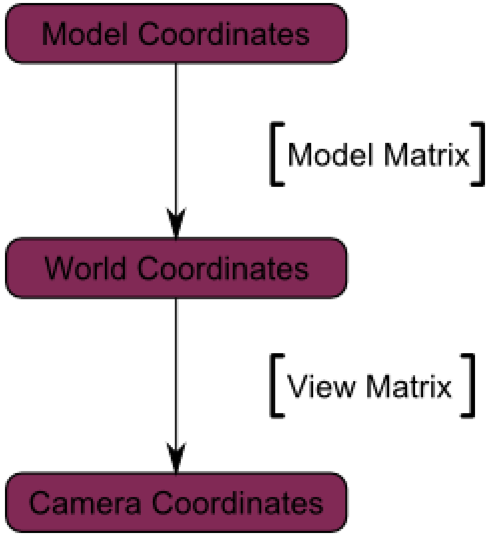



```
glm::mat4 myModelMatrix =
myTranslationMatrix * myRotationMatrix * myScaleMatrix;
glm::vec4 myTransformedVector = myModelMatrix * myOriginalVector;
```

2. Model, View, Projection

World and Camera Space

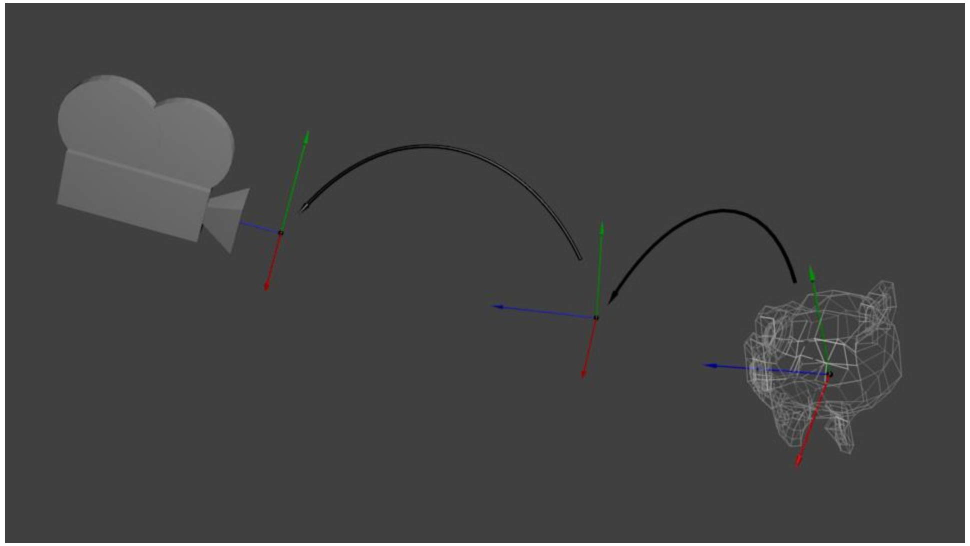


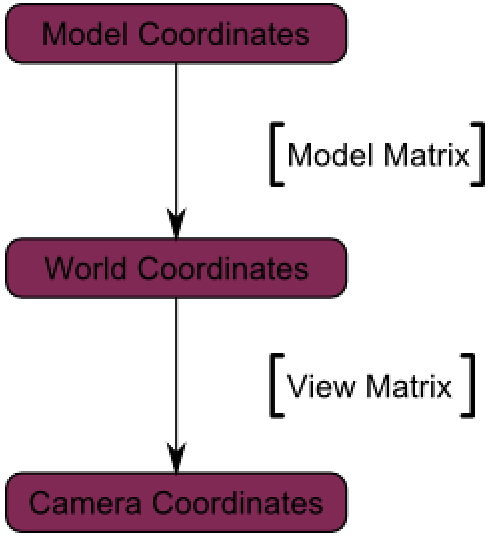


glm::mat4 CameraMatrix = glm::lookAt(
cameraPosition, *// the position of your camera, in world space*
cameraTarget, *// where you want to look at, in world space*
upVector *// probably glm::vec3(0,1,0), but (0,-1,0) would make you looking upside-down, which can be great too*);

2. Model, View, Projection

World and Camera Space



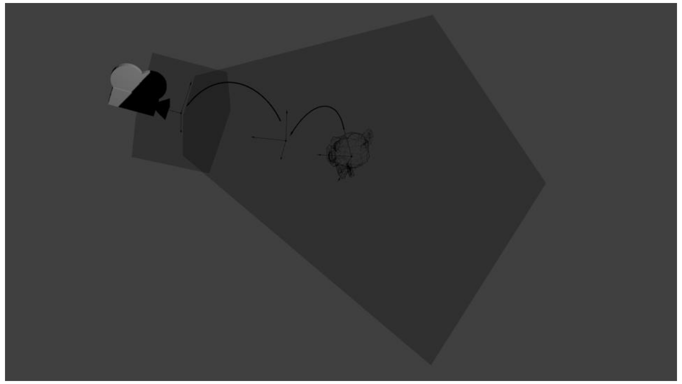


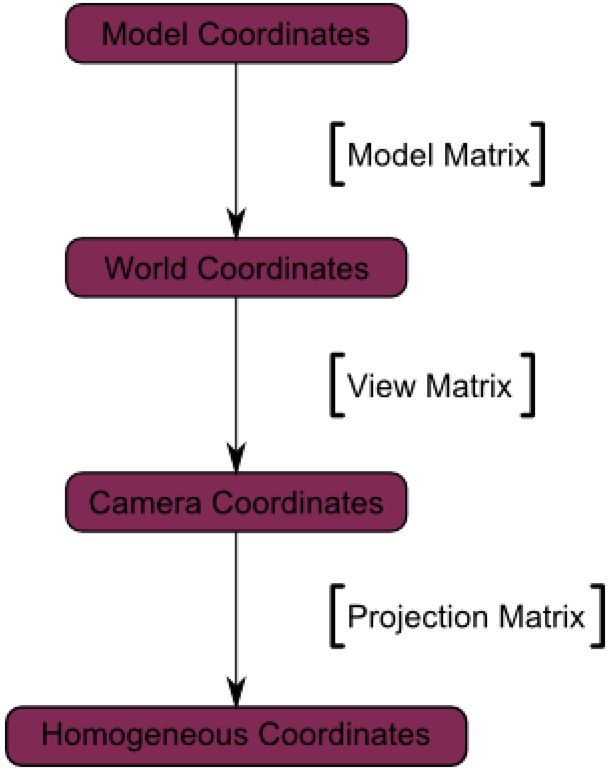

```
glm::mat4 CameraMatrix = glm::lookAt(
cameraPosition, // the position of your camera, in world space
cameraTarget, // where you want to look at, in world space
upVector // probably glm::vec3(0,1,0), but (0,-1,0) would make you looking upside-down, which can be great too );
```

2. Model, View, Projection

Projection Space

Note: this is NOT
an affine transform.
No parallel lines preserved

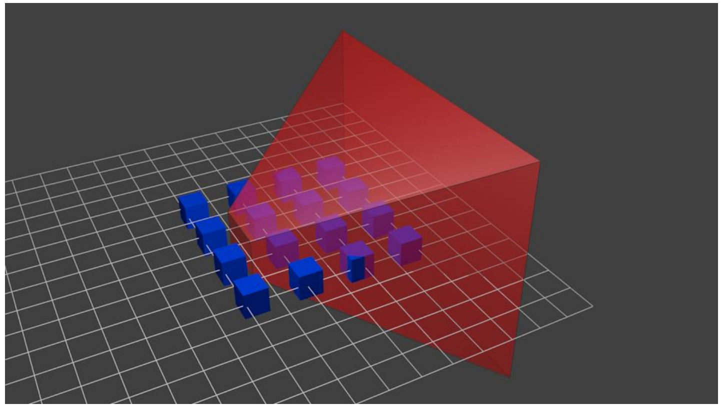


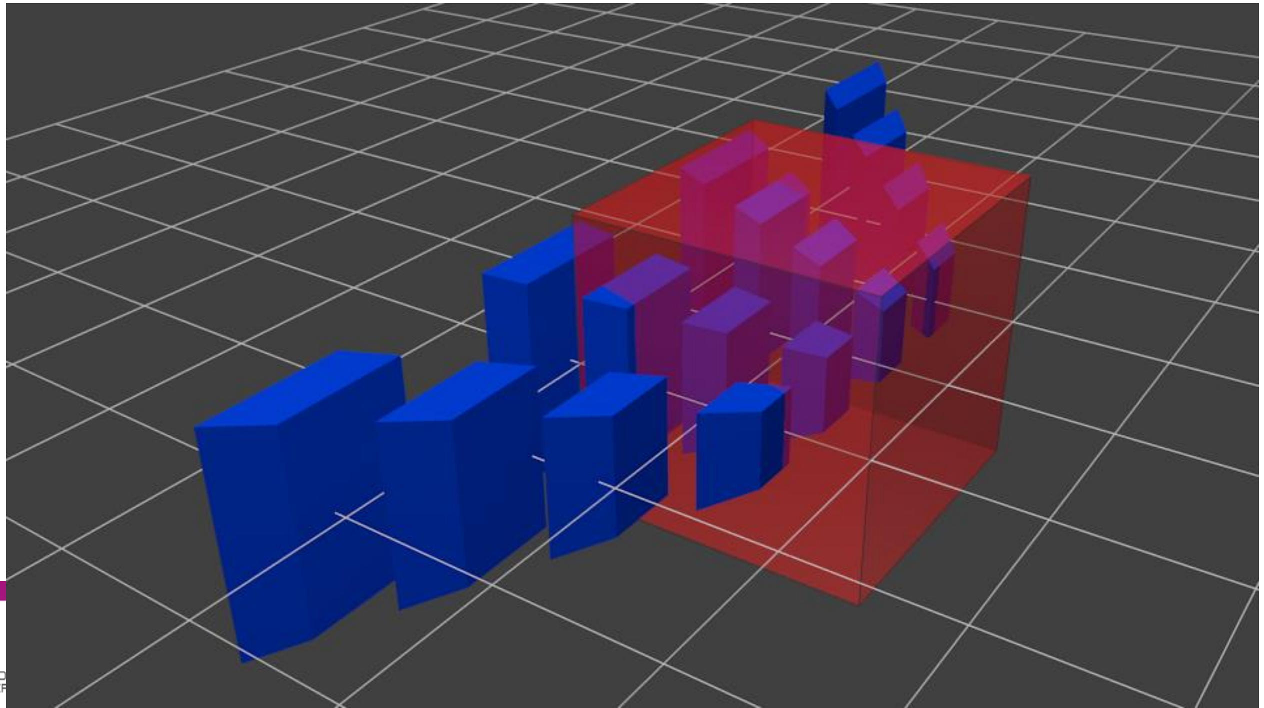


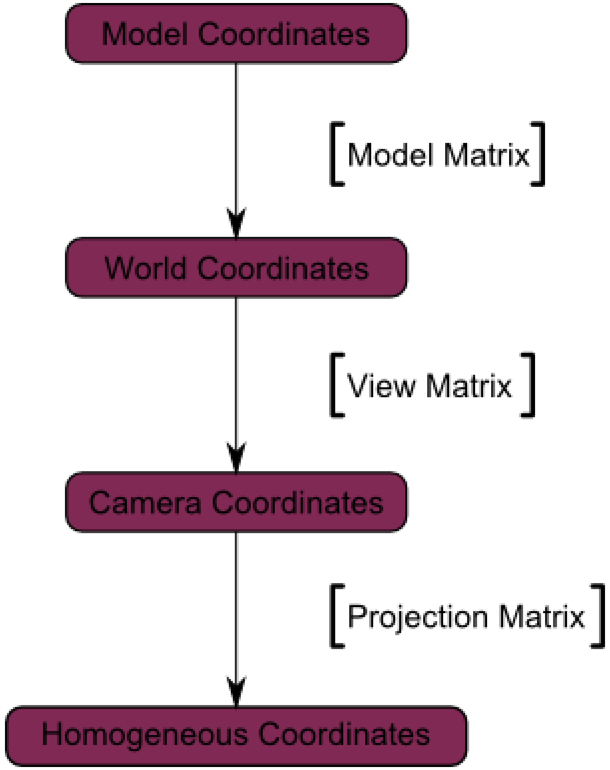
glm::mat4 projectionMatrix = glm::perspective(
glm::radians(FoV), *// The vertical Field of View, in radians: the amount of "zoom". Think "camera lens". Usually between 90° (extra wide) and 30° (quite zoomed in)*
4.0f **/** 3.0f, *// Aspect Ratio. Depends on the size of your window. Notice that 4/3 == 800/600 == 1280/960, sounds familiar ?*
0.1f, *// Near clipping plane. Keep as big as possible, or you'll get precision issues.*
100.0f *// Far clipping plane. Keep as little as possible.*
);

2. Model, View, Projection

Projection Space

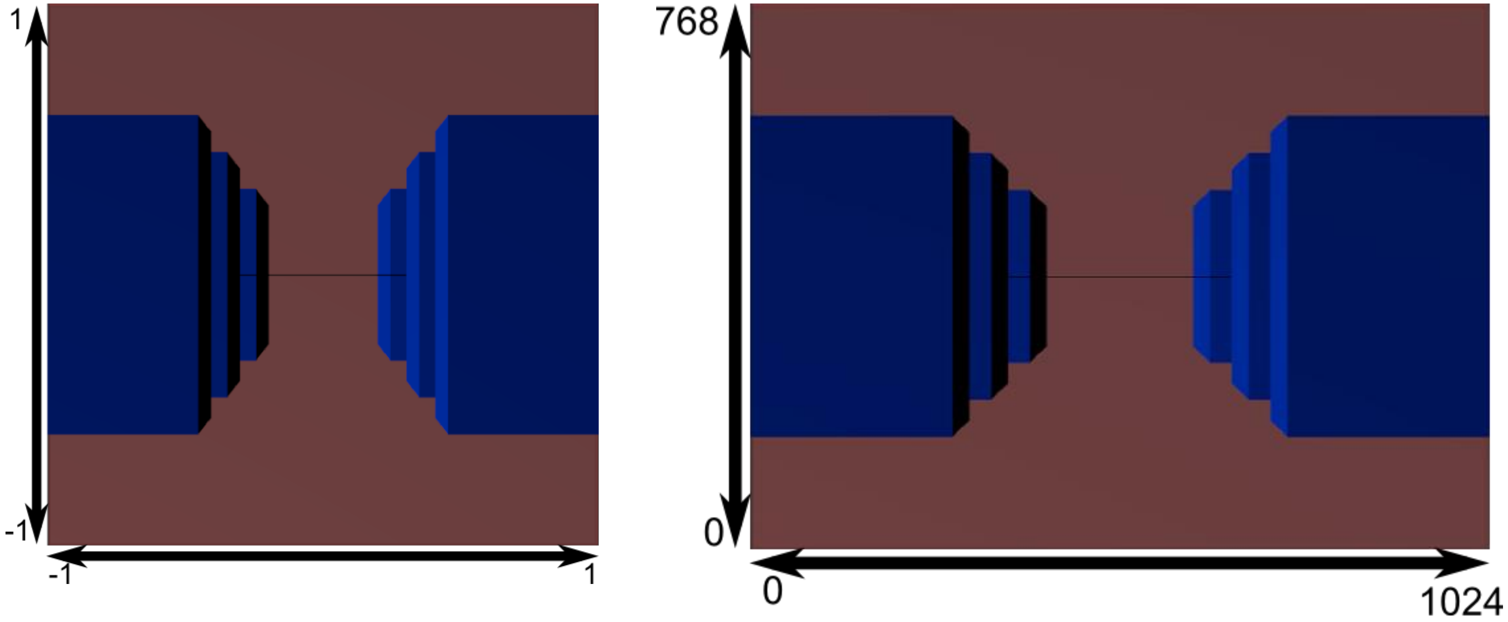






2. Model, View, Projection

Projection Space to Windows space (automatic)



2. Model, View, Projection

In practice (1):

In C++, with GLM:

```
// C++ : compute the matrix
glm::mat4 MVPmatrix = projection * view * model; // Remember : inverted !
```

In GLSL :

```
// GLSL : apply it
transformed_vertex = MVP * in_vertex;
```

2. Model, View, Projection

In practice (2):

```
// C++
RV::_modelView = glm::mat4(1.f);
RV::_modelView = glm::translate(RV::_modelView,
glm::vec3(RV::panv[0], RV::panv[1],
RV::panv[2]));
RV::_modelView = glm::rotate(RV::_modelView,
RV::rota[1], glm::vec3(1.f, 0.f, 0.f));
RV::_modelView = glm::rotate(RV::_modelView,
RV::rota[0], glm::vec3(0.f, 1.f, 0.f));

RV::_MVP = RV::_projection * RV::_modelView;
```

```
// GLSL
const char* cube_vertShader =
"#version 330\n\
in vec3 in_Position;\n\
in vec3 in_Normal;\n\
out vec4 vert_Normal;\n\
uniform mat4 objMat;\n\
uniform mat4 mv_Mat;\n\
uniform mat4 mvpMat;\n\
void main() {\n\
gl_Position = mvpMat * objMat *
       vec4(in_Position, 1.0);\n\
vert_Normal = mv_Mat * objMat *
       vec4(in_Normal, 0.0);\n\
}";
```

2. Model, View, Projection

In practice (2 - continued):

```
// GLSL Glue
glUseProgram(cubeProgram);
glUniformMatrix4fv(glGetUniformLocation(cubeProgram,
"objMat"), 1, GL_FALSE, glm::value_ptr(objMat));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram,
"mv_Mat"), 1, GL_FALSE,
glm::value_ptr(RenderVars::_modelView));
glUniformMatrix4fv(glGetUniformLocation(cubeProgram,
"mvpMat"), 1, GL_FALSE, glm::value_ptr(RenderVars::_MVP));
```

2. Model, View, Projection

Exercises.

1. Try changing the glm::perspective
2. Instead of using a perspective projection, use an orthographic projection (glm::ortho)
3. Modify the Model Matrix to translate, rotate, then scale the cube
4. Do the same thing, but in different orders. What do you notice? What is the “best” order that you would want to use for a character?
5. Modify the View matrix to translate and rotate the camera. Compare with the result from exercise 3, using the same values. What do you notice?

Resources

- [opengl-tutorial2018] unknown authors (last retrieved 02/2018) Learn OpenGL. Tutorial 3http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
- [Sellers2016] Graham Sellers, Richard S. Writght, Jr. Nicholas Haemel (2016) *OpenGL SuperBible*, 6th Edition. Pearson education (chapter 4)
- [Akenine2008] Tomas Akenine-Möller,, Eric Haines, Naty Hoffman (2008) Real-Time Rendering. 3rd Edition CRC Press (sections 2.3 and 4.6)