Advanced Computer Graphics
Lab 3

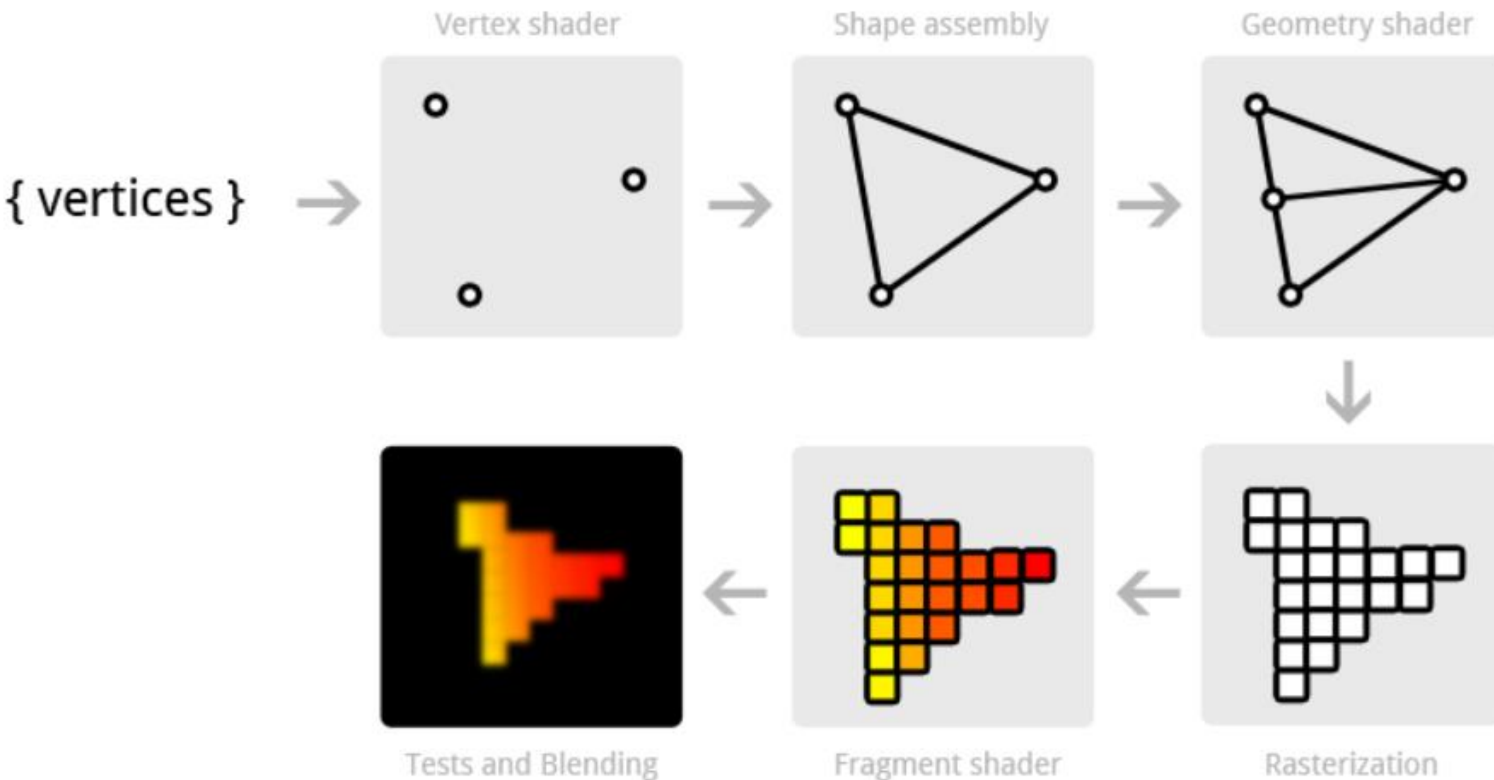
Today's roadmap

- Shaders
- Transformations in 2D:
  - Translation
  - Scaling
  - Rotation

Shaders – recap

- Modern OpenGL requires that we at least set up a **vertex and fragment shader** if we want to do some rendering
- Written in a language called **glsl** (similar to C)
- **Vertex shader**
  - used to manipulate the attributes of vertices
- **Fragment shader**
  - Also called **"pixel shader"**
  - Takes care of how the pixels between the vertices look (interpolated pixels)

Graphics pipeline – recap



Shaders

- Programs which are run for each specific section of the **graphics pipeline.**
- In a basic sense, shaders are nothing more than programs **transforming inputs to outputs.**
- The main communication they have is via their **inputs** and **outputs**
- **Uniforms** are another way to pass data from **our application (opengl code) to the shader.** A uniform variable is **unique** per shader program object, and can be accessed from any shader at any stage in the shader program. Whatever you set the uniform values to, they will **keep their values** until they're either reset or updated.

Shaders

```
#version version_number
in type in_variable_name;
in type in_variable_name;
out type out_variable_name;
uniform type uniform_name;
void main() {
// process input(s) and do some weird
graphics stuff ...
// output processed stuff to output variable
out_variable_name = weird_stuff_we_processed;
}
```

Vectors

- **Vector: A vector in GLSL** has a maximum size of 4 and each of its values can be retrieved via **vec.x, vec.y, vec.z and vec.w** respectively where each of them represents a coordinate in space. Note that the vec.w component is not used as a position in space (we're dealing with 3D, not 4D) but is used for **homogenous coordinates**.
- Examples:
  - vec2 a = vec2(1.0, 2.0);
  - vec3 b = vec3(-1.0, 0.0, 0.0);
  - vec4 c = vec4(0.0, 0.0, 0.0, 1.0);
  - vec4 a = vec4(0.0); // = vec4(0.0, 0.0, 0.0, 0.0)

- Obs: We can access the components of the vector with: a.x, a.y or a.xy, a.xyz etc.

Tasks – part 1

1) The objects' color is set to red in the vertex shader. Change the implementation so that it is set to green in the **fragment shader.** Remove any unneeded code from the two shaders.

Homogenous coordinates

- A point (x, y) can be re-written in **homogeneous coordinates** as (x_h, y_h, h)
- We can then write any point (x, y) as (hx, hy, h), hx = x / h, hy = y/h.
- We can conveniently choose h = 1 so that (x, y) becomes (x, y, 1)
- **Why?** To be able to represent transforms as 3x3 matrices => simplify the calculations

2D Transform Matrix – Recap

Translation

$$x_{new} = x_{old} + dx$$
$$y_{new} = y_{old} + dy$$

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{old} \\ y_{old} \\ 1 \end{bmatrix} = \begin{bmatrix} 1*x_{old} + 0*y_{old} + dx*1 \\ 0*x_{old} + 1*y_{old} + dy*1 \\ 0*x_{old} + 0*y_{old} + 1*1 \end{bmatrix} = \begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix}$$

Scaling

$$x_{new} = Sx \times x_{old}$$
$$y_{new} = Sy \times y_{old}$$

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{old} \\ y_{old} \\ 1 \end{bmatrix} = \begin{bmatrix} s_x \times x_{old} \\ s_y \times y_{old} \\ 1 \end{bmatrix} = \begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix}$$

Rotation

$$x_{new} = x_{old} \times \cos\theta - y_{old} \times \sin\theta$$
$$y_{new} = x_{old} \times \sin\theta + y_{old} \times \cos\theta$$

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{old} \\ y_{old} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \times x_{old} - \sin\theta \times y_{old} \\ \sin\theta \times x_{old} + \cos\theta \times y_{old} \\ 1 \end{bmatrix} = \begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix}$$

Combining transforms

- Allowed by the fact that we use **homogenous coordinates**
- Let's say we want to apply two transforms **(A and B)** to the vector p:

$$p' = A \times p$$
$$p'' = B \times p' \quad \Rightarrow \quad p'' = (B \times A) \times p$$

- We can note M = B x A (a new matrix)

Transformations – in OpenGL

- **GLM Library** (Open**GL** **M**athematics)
- Declare and initialize a vector:
  - glm::vec3 vec(1.0f, 0.0f, 1.0f); //2D
  - glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f); //3D
- Declare and initialize a matrix (identity)
  - glm::mat3 mat;
  - glm::mat3 mat(1.0f); //2D
  - glm::mat4 mat(1.0f); // 3D
- Declare and initialize a matrix with custom values:
  - glm::mat2 mat(1.0f, 2.0f, 3.0f, 4.0f);
- Access an element:
  - Vector: vec.x OR vec[0];
  - Matrix: mat[0][0];

Transformations – in OpenGL

- **Translate** using translation vector (Tx,Ty, Tz) – *What should be the size of mat?*
  - glm::translate(mat, glm::vec3(Tx, Ty, Tz));
- **Scaling** using scaling factors (sx, sy, sz)
  - glm::scale(mat, glm::vec3(sx, sy, sz));
- **Rotation** – angle in degrees and axis – *Around which axis here? Can we have a different one?*
  - glm::rotate(mat, 90.0f, glm::vec3(0.0, 0.0, 1.0));
- !!! All transforms return a glm::mat result! => the transformations matrix after applying the transform on the "mat" sent as a parameter.

- **OBS!** Some versions of glm require the angle in **radians,** so we can use **glm::radians(angle_degrees).**
- Obs: We can change the rotation angle based on time, using the **glfwGetTime()** function *(= returns the time spent since we initialized glfw, in seconds)*.

Transformations – Steps

- Create a transform matrix:

```
glm::mat4 trans(1.0f);
trans = glm::scale(trans, glm::vec3(sx, sy, sz));
```

- Send uniform variables to the shader:
  - unsigned int transformLoc = glGetUniformLocation(programID, "transform");
  - glUniformMatrix4fv(transformLoc 1, GL_FALSE, glm::value_ptr(trans));

- Obs:
  - **glGetUniformLocation** uses:
    - Shader program id
    - **Name** of the uniform variable (THE SAME that we use in the shader)
  - **glUniformMatrix4fv** uses:
    - Uniform location (obtained above)
    - Number of matrices
    - If we should transpose the matrix
    - Pointer to the matrix data

- Other uniforms: glUniform3i, glUniform3fv etc. Full doc here: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml
- In the vertex shader, we add the uniform variable and multiply the transform with the vertex position.

Tasks – part 2

2) Observe what changes we made in order to scale the object in the Lab3 example.

Achieve the same result from **Task 1**, but this time send the green color from the main program.

3) Create a transform in order to rotate the square with 45 degrees.

4) Modify the rotation transform in order to make the square rotate continuously. **Hint**: Use **glfwGetTime()** to change the rotation angle ☺**.**

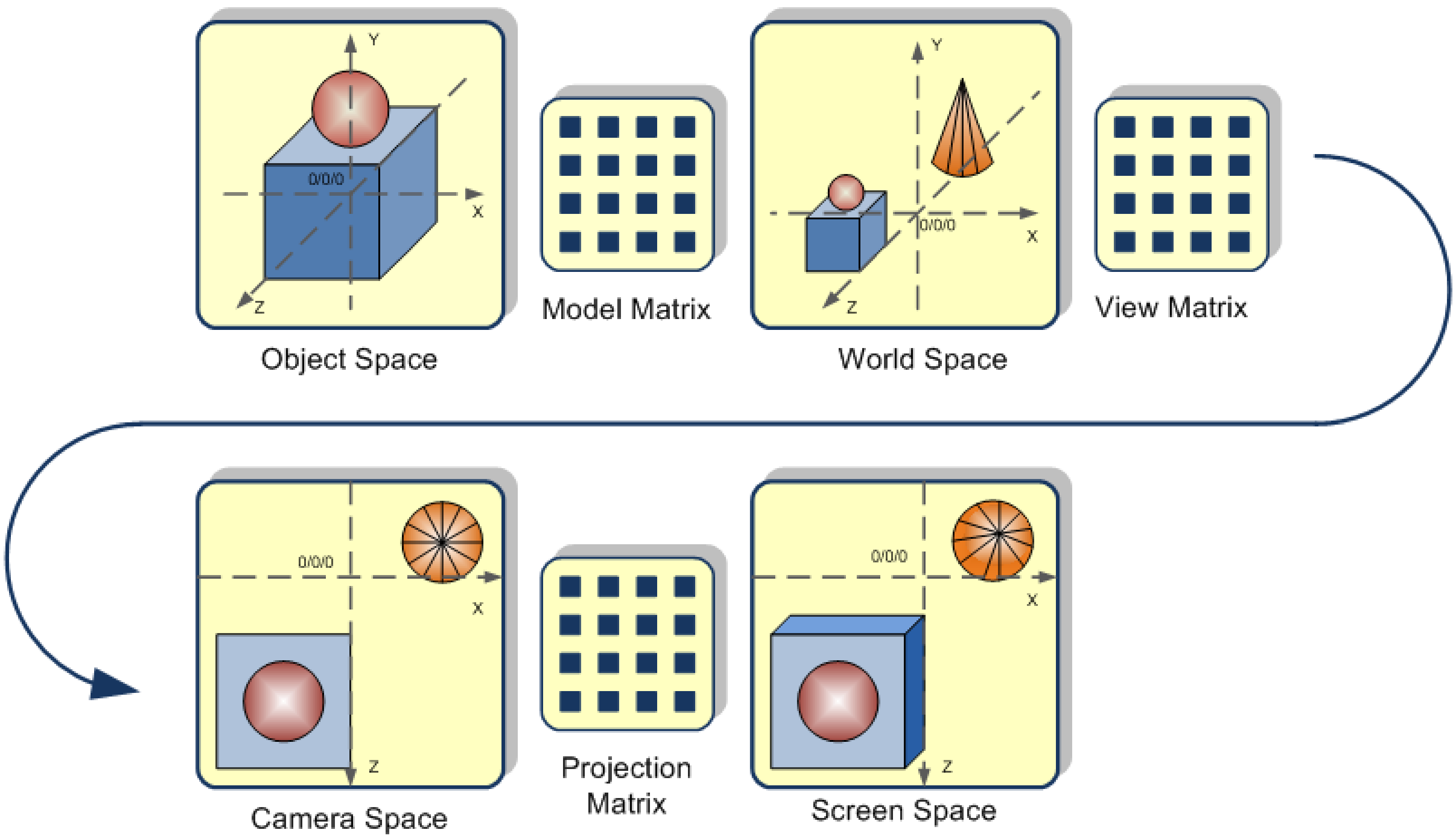
Tasks – part 3

5) Use the following data to draw 10 squares translated in these positions. Try to draw them in different colors too.

```
glm::vec3 positions[] = {
glm::vec3(0.0f,  0.0f,  0.0f),
glm::vec3(0.2f,  0.5f, -0.15f),
glm::vec3(-0.15f, -0.22f, -0.25f),
glm::vec3(-0.38f, -0.2f, -0.123f),
glm::vec3(0.24f, -0.4f, -0.35f),
glm::vec3(-0.17f,  0.3f, -0.75f),
glm::vec3(0.93f, -0.2f, -0.75f),
glm::vec3(0.15f,  0.2f, -0.25f),
glm::vec3(0.15f,  0.7f, -0.55f),
glm::vec3(-0.13f,  0.1f, -0.15f)
};
```

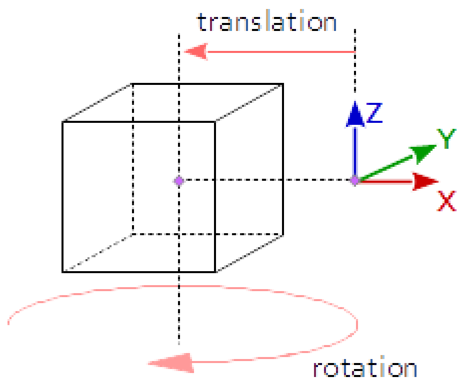
6) Make the squares from Ex 5) rotate continuously. Change the order between the translation and the rotation transforms. Are there any changes?

Tasks – part 3 (explanation)

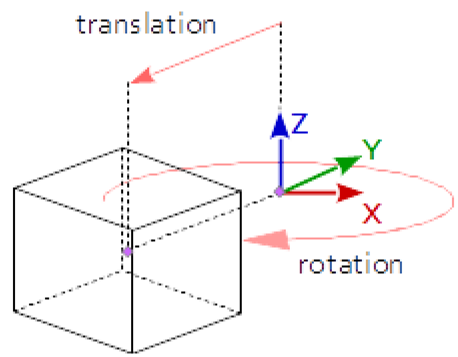


Tasks – part 3 (explanation)

Correct: Rotate (around Oz), then translate



Weird:
First translate => our object is not in the origin anymore;
Then rotate => rotate around the world axis



Task – bonus (if you want to practice more)

Take the vertices and indices from Lab 2 and use them to draw 10 cubes in different positions (similar to task 5). Make all the changes needed in your program.

First person that sends me the screenshot of it working & an explanation on the private chat will get 2 extra points at this lab ☺