UNIVERSITÉ
Concordia
UNIVERSITY

COMP 371
Computer Graphics

Lab 02 – Graphics Pipeline
Transformations

Prepared by Nicolas Bergeron

This Week

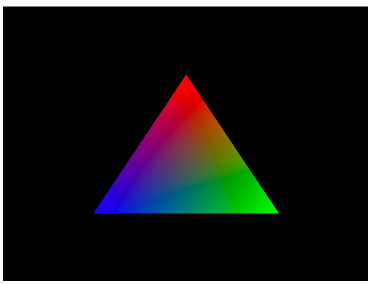
Tutorial:
World, View, Projection Transforms
Animation over time
GLM matrix API

Exercises
Move simple virtual camera with ASDW

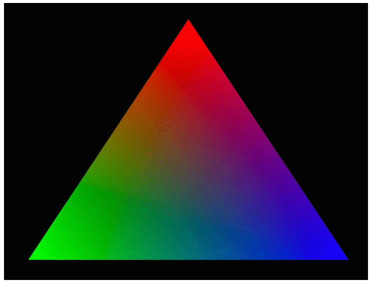
Expected results

World

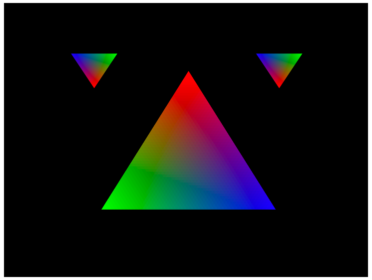
Initial



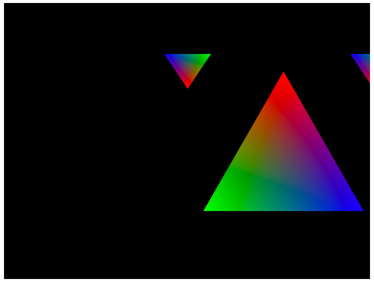
Animation



Instances

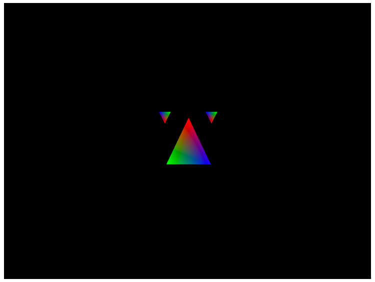


View



Projection

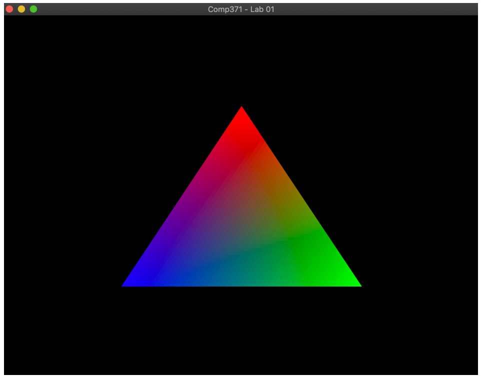
Orthographic



Perspective

Getting Started

- Download Lab01.zip from Moodle
- Download Lab02.zip and add lab02.cpp to the project (Visual Studio or Xcode)
- After compiling and running the application, you should see the solution for lab01 (tutorial)



Outline for Lab02
Implement transforms in OpenGL

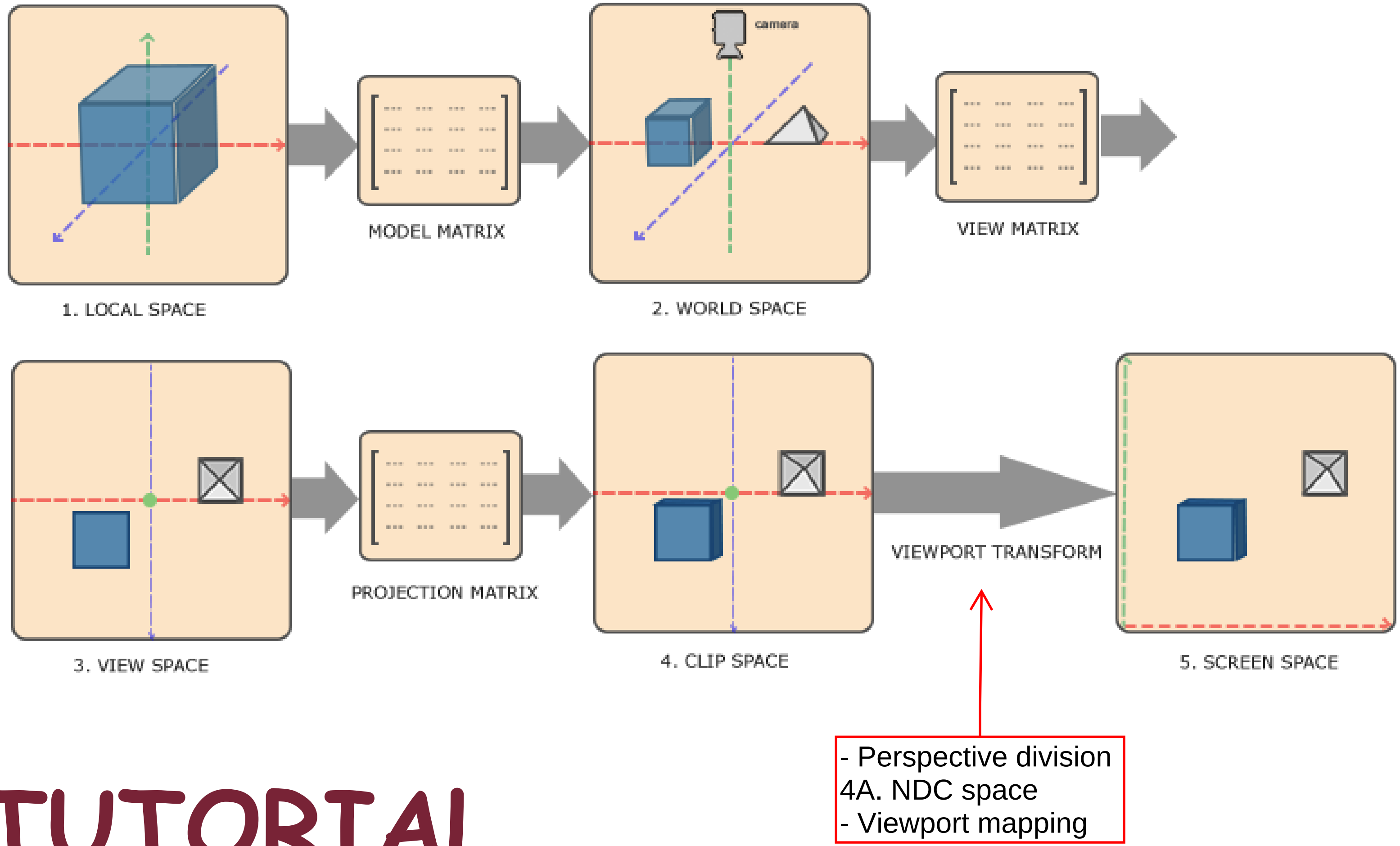
- Understand World Transform
  - Instantiate multiple models
  - Animate models over time
- Understand View Transform
  - See scene from different angles
- Understand Projection Transform
  - Use different camera parameters
  - Perspective vs orthographic camera
- Misc
  - Backface culling
  - Uniform Variables in Shaders

Graphics Pipeline Transformations (Vertex position from model to pixel)

- Positions in each space can be computed as follow:
  - $P^W = M^{World} * P^M$
  - $P^V = M^{View} P^W$
  - $\tilde{P} = M^{Proj} * P^V$
  - $P^N = \tilde{P} / \tilde{w}$
  - $P^D = M^{Viewport} * P^N$

Focus for this lab

- Matrices can be concatenated to skip step
  - $\tilde{P} = M^{Proj} M^{View} M^{World} * P^M$



TUTORIAL

TRANSFORM RAINBOW TRIANGLE

World Transform (aka Model Transform)

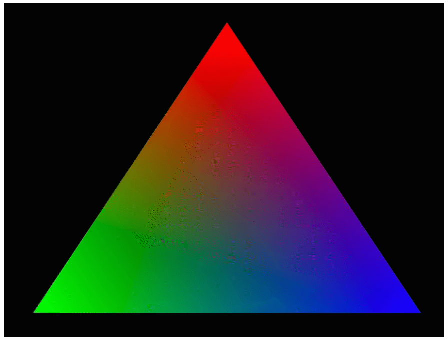
- Generally the first transform in graphics pipeline
- Allows positioning, scaling and orienting models in a common 3D world.
- Allows multiple instances of the same model by reusing the data pre-loaded on the GPU.
- Computed for each model, for each frame, and sent to Vertex Shader as a uniform variable.
- The world transform must be recomputed every frame based on the Scaling, Rotation and Translation (Order is important)

World Transform
Exposing matrix in Vertex Shader

- World matrix will be exposed as a uniform variable, as each vertex on a 3D model will be transformed the same way.
- We need to transform the vertex position by the world transform in the vertex program.

```
"#version 330 core\n"
"layout (location = 0) in vec3 aPos;"
"layout (location = 1) in vec3 aColor;"
""
"uniform mat4 worldMatrix;"
""
"out vec3 vertexColor;"
"void main()"
"{"
"   vertexColor = aColor;"
"   gl_Position = worldMatrix * vec4(aPos.x, aPos.y, aPos.z, 1.0);"
"}";
```

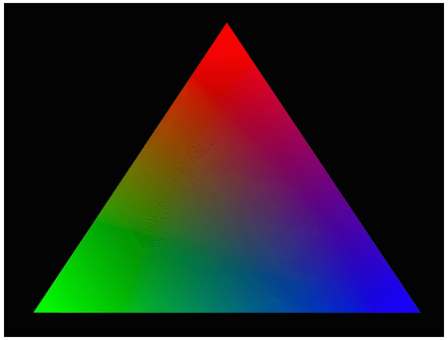
Rotate model about Y-Axis (1/2)



Steps:

- Determine the time step (frame duration) in order to control the rotation speed
- Increase rotation angle according to rotation speed and time step
- Create a rotation matrix around the y-axis
- Bind the rotation matrix to the Vertex Shader as the World transform for our model
- Draw the model

Rotate model about Y-Axis (2/2)



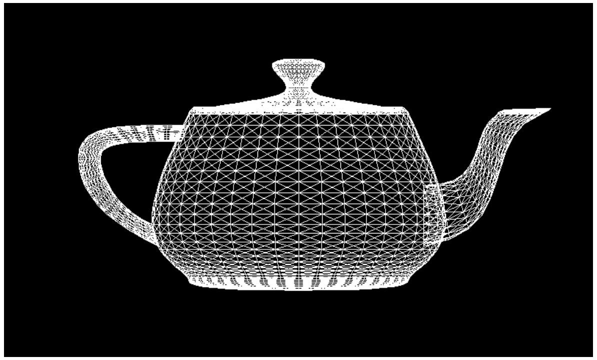
```
// Draw geometry
glUseProgram(shaderProgram);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

float dt = glfwGetTime() - lastFrameTime;
lastFrameTime += dt;

angle = (angle + rotationSpeed * dt); // angles in degrees, but glm expects radians (conversion below)
glm::mat4 rotationMatrix = glm::rotate(glm::mat4(1.0f), glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));

GLuint worldMatrixLocation = glGetUniformLocation(shaderProgram, "worldMatrix");
glUniformMatrix4fv(worldMatrixLocation, 1, GL_FALSE, &rotationMatrix[0][0]);

glDrawArrays(GL_TRIANGLES, 0, 3); // 3 vertices, starting at index 0
```

More instances with same VBO

- All we need to do is change the vertex shader world matrix and draw again.

```
glUniformMatrix4fv(worldMatrixLocation, 1, GL_FALSE, &rotationMatrix[0][0]);

glDrawArrays(GL_TRIANGLES, 0, 3); // 3 vertices, starting at index 0

// Top right triangle - translate by (0.5, 0.5, 0.5)
// Scaling model by 0.25, notice negative value to flip Y axis
glm::mat4 scalingMatrix = glm::scale(glm::mat4(1.0f), glm::vec3(0.25f, -0.250f, 0.25f));
glm::mat4 translationMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.5f, 0.5f, 0.0f));

glm::mat4 worldMatrix = translationMatrix * scalingMatrix;

glUniformMatrix4fv(worldMatrixLocation, 1, GL_FALSE, &worldMatrix[0][0]);
glDrawArrays(GL_TRIANGLES, 0, 3);

// Top left triangle - translate by (-0.5, 0.5, 0.5)
translationMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-0.5f, 0.5f, 0.0f));
worldMatrix = translationMatrix * scalingMatrix;

glUniformMatrix4fv(worldMatrixLocation, 1, GL_FALSE, &worldMatrix[0][0]);
glDrawArrays(GL_TRIANGLES, 0, 3);

// End Frame
glfwSwapBuffers(window);
```

Backface Culling



- Pipeline optimization to reject polygons facing away from camera for watertight 3D models. By default:
  - Keep polygons with vertices in Counter Clockwise (CCW) order
  - Reject polygons with vertices in CW order
  - Typically reduces the fragments to process by 50%

```
// Variables to be used later in tutorial
float angle = 0.0f;
int rotationSpeed = 180.0f;  // 180 degrees per second
float lastFrameTime = glfwGetTime();

// Enable Backface culling
glEnable(GL_CULL_FACE);

// Entering Main Loop
while(!glfwWindowShouldClose(window))
{
    // Each frame, reset color of each pixel to glClearColor
```

View Transform

- To see the world from different points of view (virtual camera), we can add the view transform
- It defines the position and orientation of the camera in the world

Steps:

- Expose view matrix to vertex shader
- Generate the view matrix
- Bind the matrix to Vertex Shader

(we create two view matrices, map them to keys [1] and [2]

View Matrix with GLM

- glm::lookAt creates a view transform, parameters are
  - eye: position of virtual camera in the world
  - center: focus camera towards that position
  - up: vector describing upwards, generally Y-Axis

Université Concordia University

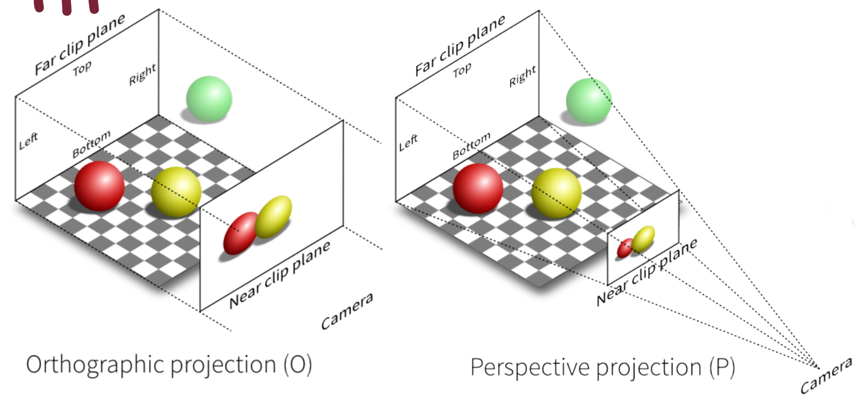
View Transform in Vertex Shader

```
"#version 330 core\n"
"layout (location = 0) in vec3 aPos;"
"layout (location = 1) in vec3 aColor;"
""
"uniform mat4 worldMatrix;"
"uniform mat4 viewMatrix = mat4(1.0);"  // default value for view matrix (identity)
""
"out vec3 vertexColor;"
"void main()"
"{"
"   vertexColor = aColor;"
"   gl_Position = viewMatrix * worldMatrix * vec4(aPos.x, aPos.y, aPos.z, 1.0);"
"}";
```

Let's add 2 camera point of views

```
// Handle inputs
if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    glfwSetWindowShouldClose(window, true);

// by default, camera is centered at the origin and look towards negative z-axis
if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS)
{
    glm::mat4 viewMatrix = glm::mat4(1.0f);

    GLuint viewMatrixLocation = glGetUniformLocation(shaderProgram, "viewMatrix");
    glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, &viewMatrix[0][0]);
}

// shift camera to the left
if (glfwGetKey(window, GLFW_KEY_2) == GLFW_PRESS) // shift camera to the left
{
    glm::mat4 viewMatrix = glm::lookAt(glm::vec3(-0.5f, 0.0f, 0.0f), // eye
                                       glm::vec3(-0.5f,0.0f,-1.0f),  // center
                                       glm::vec3(0.0f, 1.0f, 0.0f) );// up

    GLuint viewMatrixLocation = glGetUniformLocation(shaderProgram, "viewMatrix");
    glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, &viewMatrix[0][0]);
}
```

Projection Transform



- We can think of this one as the lens of the camera
- It defines the field of view, the near and far planes. In other words, the visible region of the world, called view frustum.
- Orthographic projections preserve parallel lines
- Perspective projections add parallax effects due to field of view

Steps:

- Again, expose it to vertex shader
- Bind the matrix from application
  - glm::ortho creates orthographic projections
  - glm::perspective creates perspective projections

Projection Transform in Vertex Shader

```
"#version 330 core\n"
"layout (location = 0) in vec3 aPos;"
"layout (location = 1) in vec3 aColor;"
""
"uniform mat4 worldMatrix;"
"uniform mat4 viewMatrix = mat4(1.0);"  // default value for view matrix (identity)
"uniform mat4 projectionMatrix = mat4(1.0);"
""
"out vec3 vertexColor;"
"void main()"
"{"
"   vertexColor = aColor;"
"   mat4 modelViewProjection = projectionMatrix * viewMatrix * worldMatrix;"
"   gl_Position = modelViewProjection * vec4(aPos.x, aPos.y, aPos.z, 1.0);"
"}";
```

Let's bind a perspective and an orthographic camera to keys 3, 4

```
if (glfwGetKey(window, GLFW_KEY_3) == GLFW_PRESS)
{
    glm::mat4 projectionMatrix = glm::perspective(70.0f,            // field of view in degrees
                                                  800.0f / 600.0f,  // aspect ratio
                                                  0.01f, 100.0f);   // near and far (near > 0)

    GLuint projectionMatrixLocation = glGetUniformLocation(shaderProgram, "projectionMatrix");
    glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, &projectionMatrix[0][0]);
}

if (glfwGetKey(window, GLFW_KEY_4) == GLFW_PRESS)
{
    glm::mat4 projectionMatrix = glm::ortho(-4.0f, 4.0f,      // left/right
                                            -3.0f, 3.0f,      // bottom/top
                                            -100.0f, 100.0f);  // near/far (near == 0 is ok for ortho)

    GLuint projectionMatrixLocation = glGetUniformLocation(shaderProgram, "projectionMatrix");
    glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, &projectionMatrix[0][0]);
}
```

EXERCISE

Control the position of the camera with keyboard

- A: move camera left
- S: move camera back
- D: move camera right
- W: move camera forward

- Hold shift to increase the speed