Computação Gráfica

Computer Graphics

Engenharia Informática (11569) – 3º ano, 2º semestre

Chap. 3 – Geometric Transformations

Outline

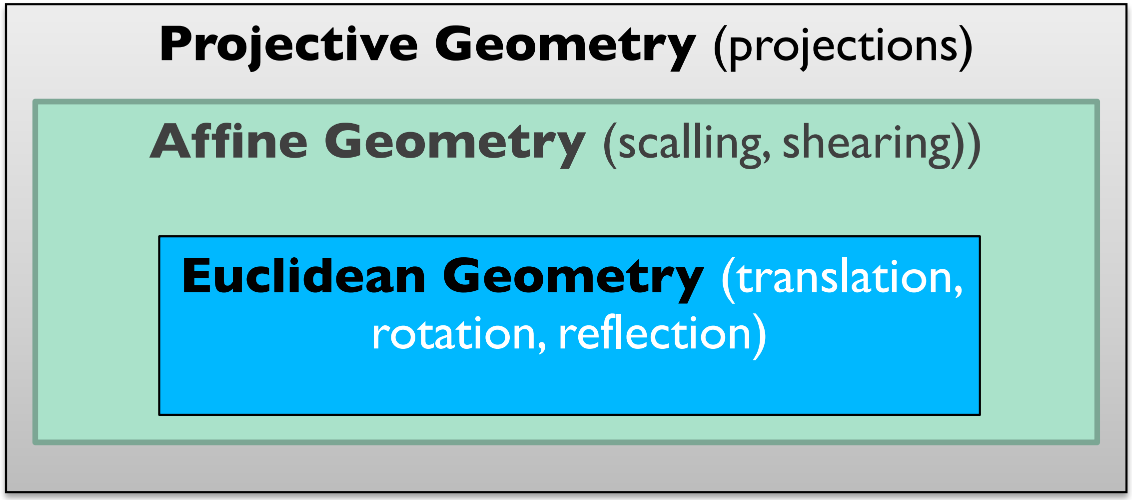
…:

- Motivation
- Euclidean transformations: translation and rotation
- Euclidean geometry
- Homogeneous coordinates
- Affine transformations: translation, rotation, and shearing
- Matrix representation of affine transformations
- Composition of 2D and 3D transformations
- Geometric transformations in OpenGL/GLM
- Matrix operations in OpenGL/GLM and arbitrary transformations
- Example in OpenGL/GLM

Geometric transformations

Classification:

- Translation, Rotation, Reflection
- Scaling, shearing
- Orthogonal projection, perspective projection



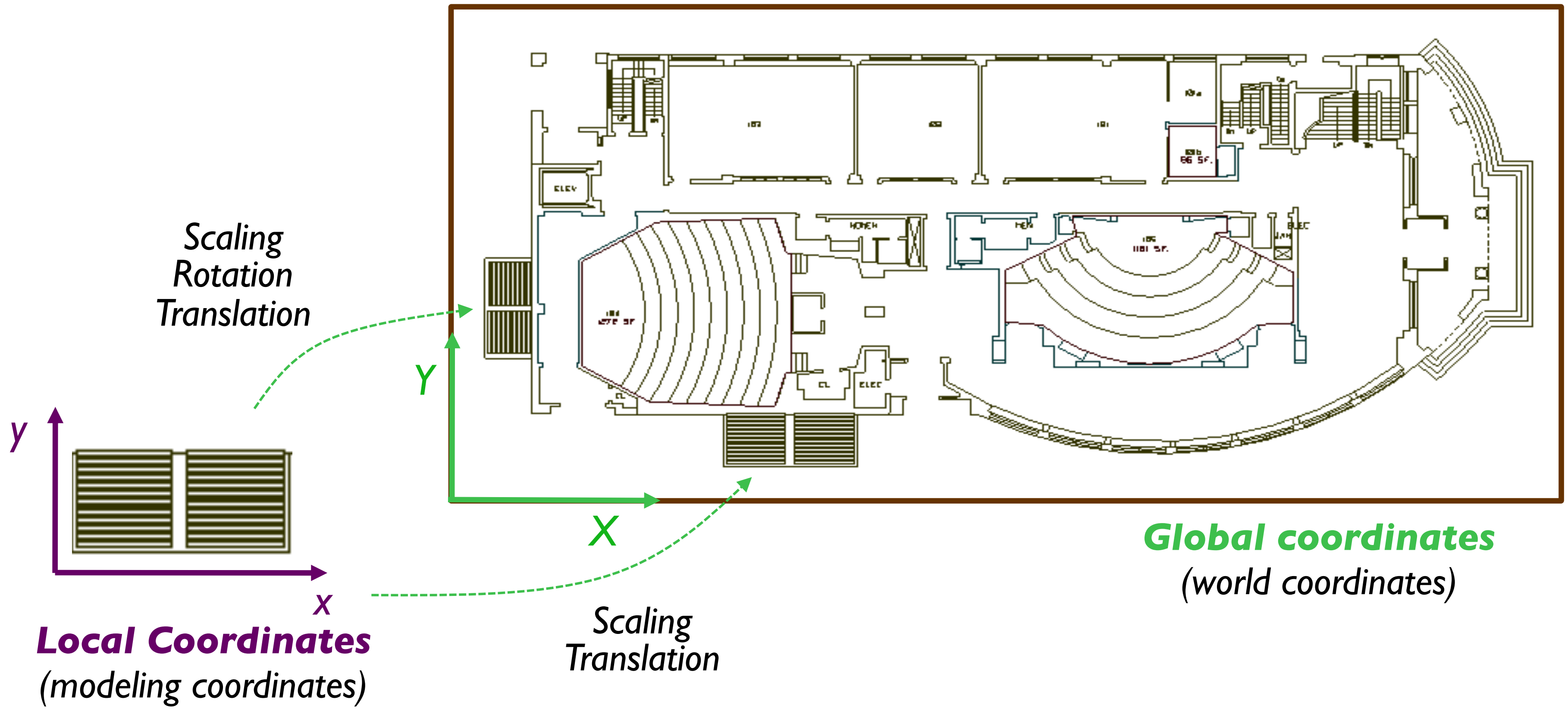
REMARK: the graphics pipeline is an implementation of the projective geometry!

Motivation

Why are geometric transformations important?

- Modeling operations of objects in 2D/3D
  - They serve to model geometric objects in 2D/3D.
  - They allow us to define na object in its own local coordinate system (modeling coordinates)
  - They allow us to instance na object several times in a global coordinate system (world coordinates)
- Positioning operations of objects in 2D/3D
  - They allow us to position and move objects in 2D/3D.
- Viewing operations in 2D/3D
  - They allow us to observe objects and scenes from distinct viewpoints. This requires the definition of the viewer, projection plane, and the 3D scene itself.

Example: modeling and positioning of objects

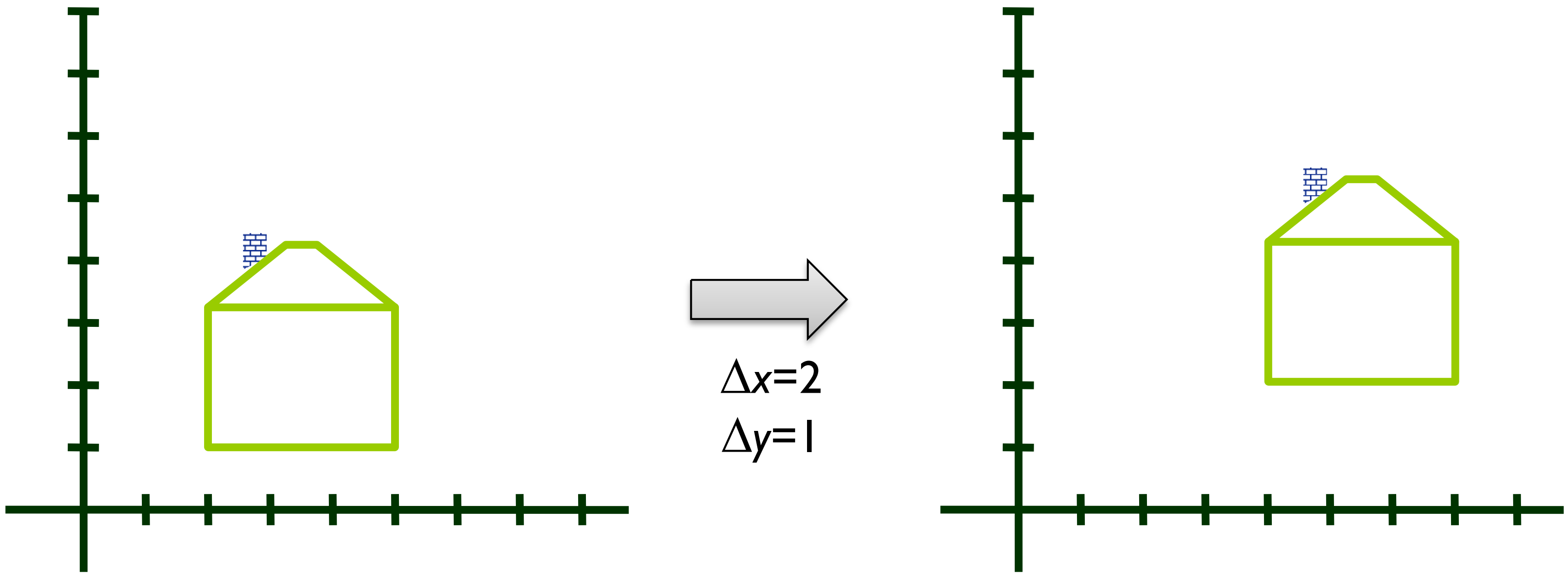


Translation in 2D

$$\begin{cases} x' = x + \Delta x \\ y' = y + \Delta y \end{cases}$$

Key idea:

- Translating one point (x, y) means to move it into a new location by an amount of linear movement $(\Delta x, \Delta y)$.



Translation in 2D: matrix form

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

Issue:

- x' is not a linear combination of x and y
- y' is not a linear combination of x and y

A linear combination of x and y is an expression ax+by, where a,b are constants.

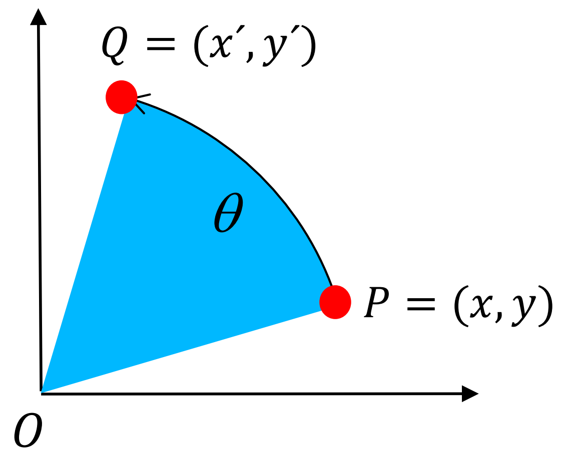
http://encyclopedia.laborlawtalk.com/Linear_combination

Rotation

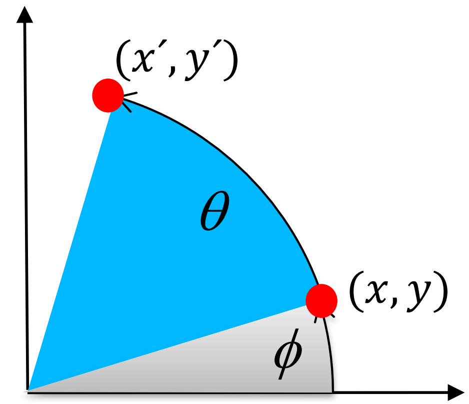
$$\begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$$

Key idea:

- Rotating a point $P = (x, y)$ by an angle θ about the origin means to find another point $Q = (x', y')$ on the same circumference centered at the origin that contains P, with $\theta = \measuredangle POQ$.



2D rotation: matrix form



$$\begin{cases} x' = r\cos(\phi + \theta) \\ y' = r\sin(\phi + \theta) \end{cases}$$

Algebraic manipulation:

– Given $\begin{cases} x = r\cos\phi \\ y = r\sin\phi \end{cases}$, the expressions of x' and y' can be rewritten as follows:

$$\begin{cases} x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ y' = r\cos\phi\sin\theta + r\sin\phi\cos\theta \end{cases} \Leftrightarrow \begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$$

Matrix form:

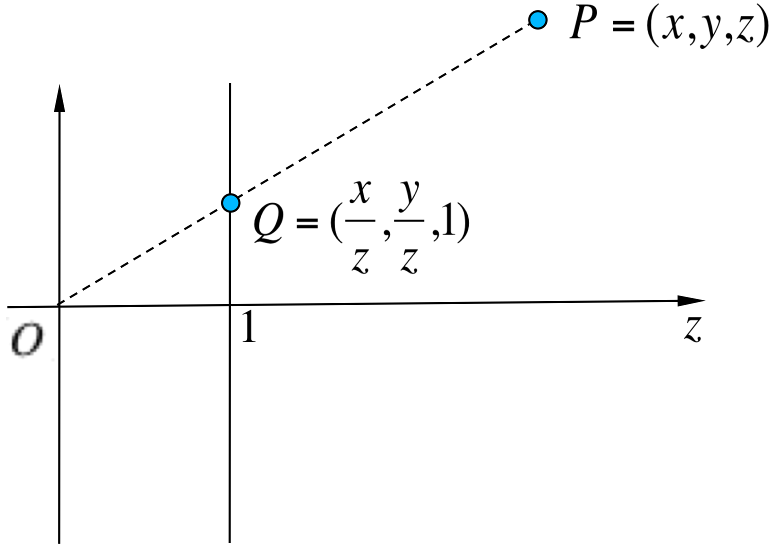
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

– x' is a linear combination of x and y

– y' is a linear combination of x and y

– though $\sin(\theta)$ and $\cos(\theta)$ are not linear functions of θ

Homogeneous coordinates



http://www.geometer.org/mathcircles/cghomogen.pdf

Why are they necessary?

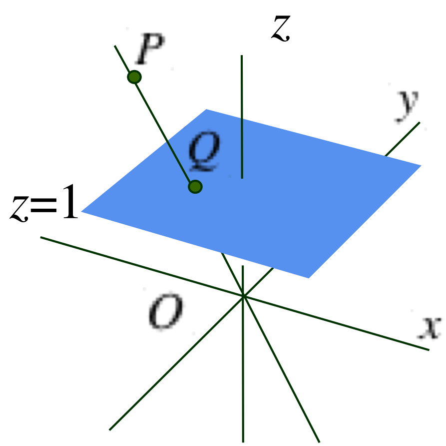
– The rotation can be formulated in terms of linear combinations, but the translation cannot!

– Consequently, we cannot use the matrix product to combine a series of translations and rotations.

Solution:

– Homogeneous coordinates!

How do they work?: $(x, y) \rightarrow (x, y, z)$, with $z = 1$.



– A single point can be represented by many sets of homogeneous coordinates.

– Thus, (x, y, z) and (x', y', z') represent the same point iff there exists a scalar α such that $x' = \alpha x$, $y' = \alpha y$ and $z' = \alpha z$.

2D translation and rotation in homogeneous coordinates

Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Now, we can combine translations and rotations using matrix product.

Euclidean geometry in 2D

Definition:

- Informally, the set of isometries (translations and rotations) in $\mathbb{R}^2$.
  - 2D Euclidean geometry: $(\mathbb{R}^2, \mathbf{GI}(2))$
  - $\mathbf{GI}(n)=(\mathbf{I}(n),\circ)$ is a group, where $\mathbf{I}(n)$ is the set of isometries (translations and rotations) and $\circ$ denotes the concatenation operator.

Metric invariant:

- Distance between points

Other metric invariants:

- Angles, lenghts, areas, and volumes

Notion of group: a refresher

Definition:

– A set C and an operation $\circ$ form a group $(C,\circ)$ if:

- *Closure Axiom*. $\forall c_1, c_2 \in C,\ c_1 \circ c_2 \in C.$
- *Identity Axiom*. $\exists i \in C$ such that $c \circ i = i \circ c,\ \forall c \in C.$
- *Inverse Element Axiom*. $\forall c \in C,\ \exists c^{-1} \in C$ such that
$$c \circ c^{-1} = i = c^{-1} \circ c$$
- *Associativity Axiom*. $\forall c_1, c_2, c_3 \in C,$
$$c_1 \circ (c_2 \circ c_3) = (c_1 \circ c_2) \circ c_3$$

Affine geometry

Definition:

- Informally, the set of affine transformations (or affinities): translation, rotation, scaling, shearing
  - It is a generalization of the Euclidean geometry
  - 2D affine geometry: $(\mathbb{R}^2, \mathbf{GA}(2))$
  - $\mathbf{GI}(n)=(\mathbf{A}(n),\circ)$ is a group, where $\mathbf{A}(n)$ is the set of affinities and $\circ$ denotes the concatenation operator.

Invariant: parallelism.

Other invariants: colinearity; distance ratio between any 3 points of a line

Examples:

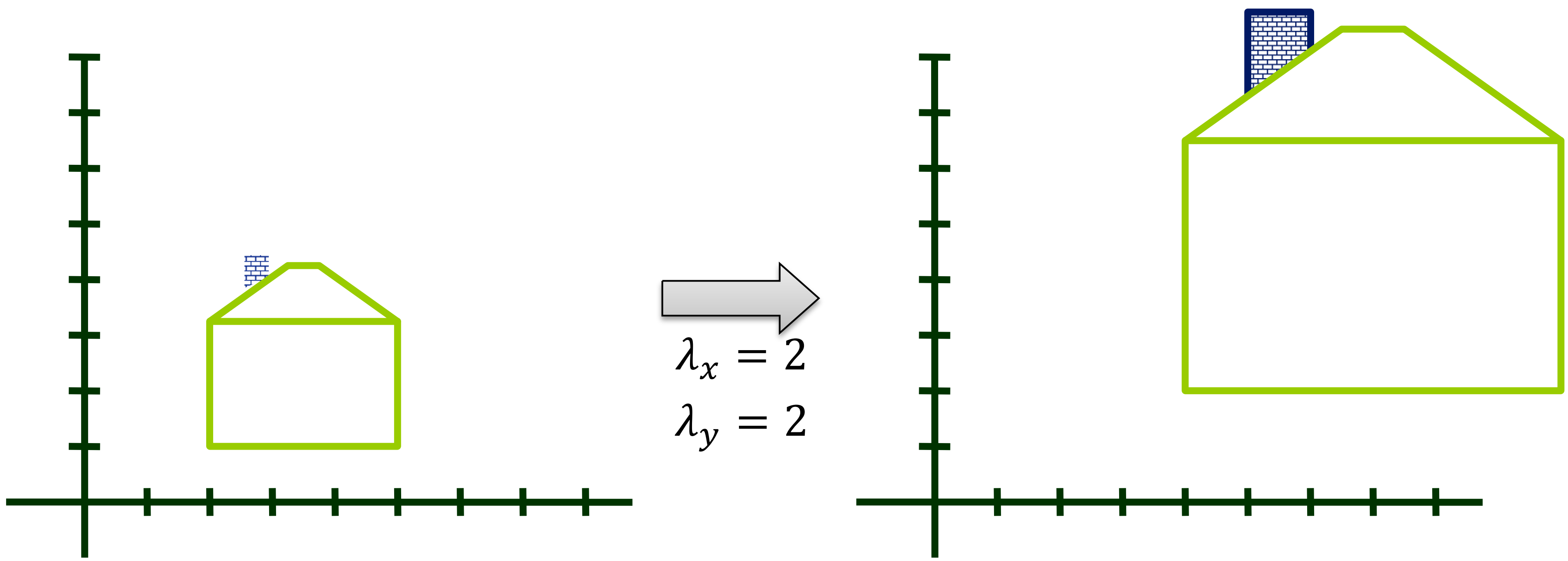
- A square can be transformed into a rectangle
- A circle can be transformed into an ellipse

Scaling

$$\begin{cases} x' = \lambda_x x \\ y' = \lambda_y y \end{cases}$$

Key idea:

– Scaling an object consists in multiplying each component of its points (x, y) by a scalar.

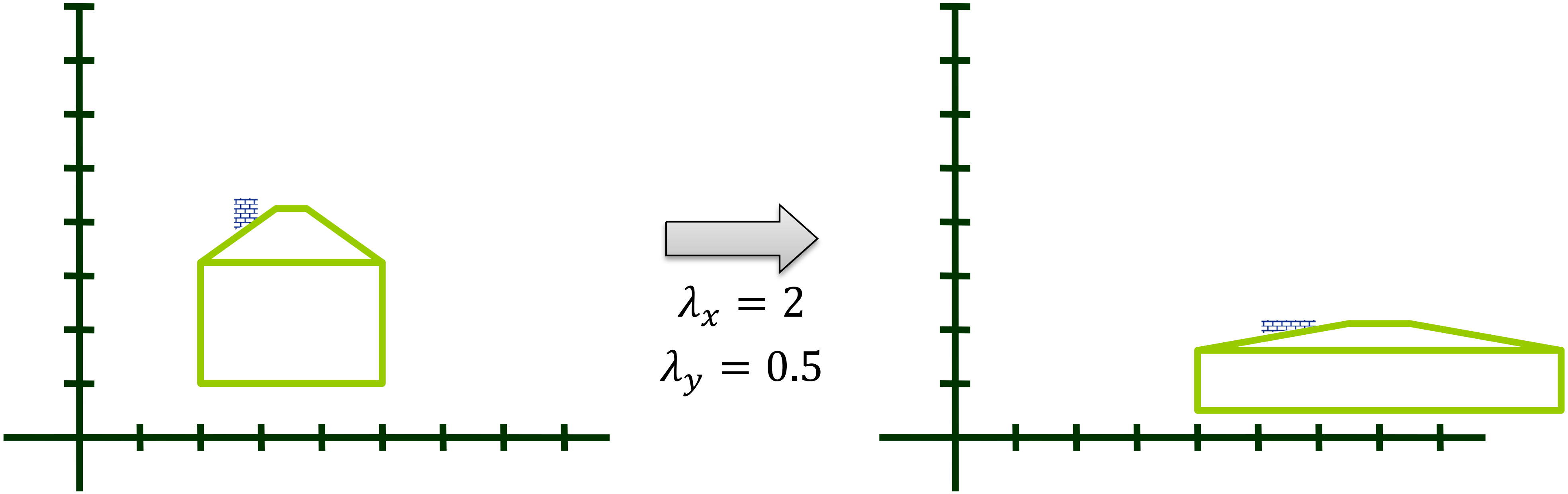


Non-uniform scaling

$$\begin{cases} x' = \lambda_x x \\ y' = \lambda_y y \end{cases} \qquad \text{with} \quad \lambda_x \neq \lambda_y$$

Key idea:

– Scaling an object consists in multiplying each component of its points (x, y) by a scalar, but the scalars are not necessarily identical.

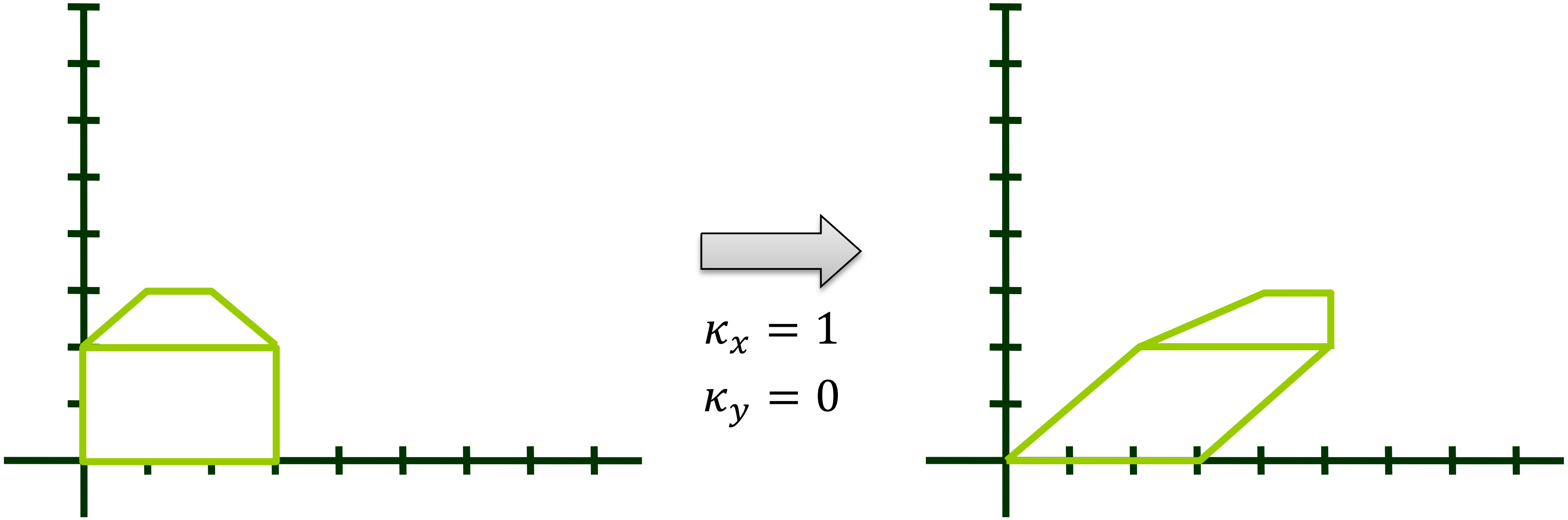


Shearing

$$\begin{cases} x' = x + \kappa_x y \\ y' = y + \kappa_y x \end{cases}$$

Key idea:

- Shearing an object consists in linearly deforming it in the direction of the x-axis or y-axis.



In short: matrix form of 2D affinities

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda_x & 0 & 0 \\ 0 & \lambda_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \kappa_x & 0 \\ \kappa_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Shearing

Composition of 2D affinities

Operator:

– The composition operator is the matrix product.

Remarks:

– The order of the composition of affinities matters.

– The matrix product is not commutative.

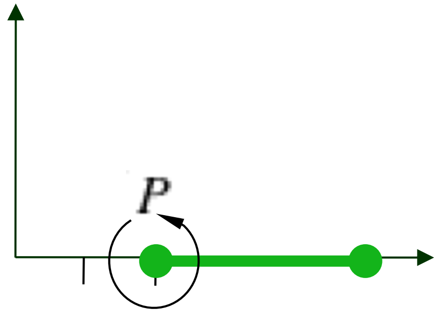
– The affine geometry does not satisfy the commutativity axiom.

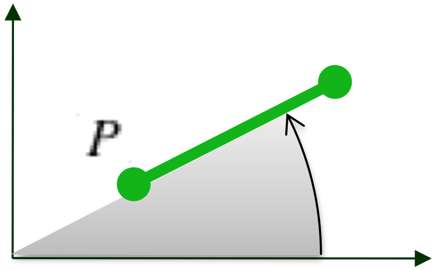
Example:

– If we change the order of the following matrices, the resulting matrix is not the same.

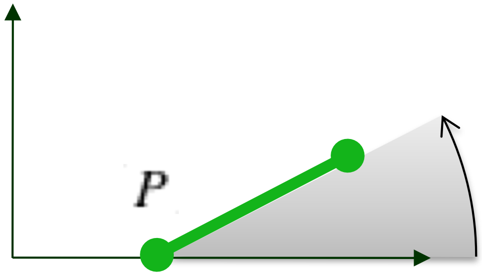
$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \lambda_x & 0 & 0 \\ 0 & \lambda_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Example: rotation of θ=30º of a straight line segment PQ about the point P(2,0)



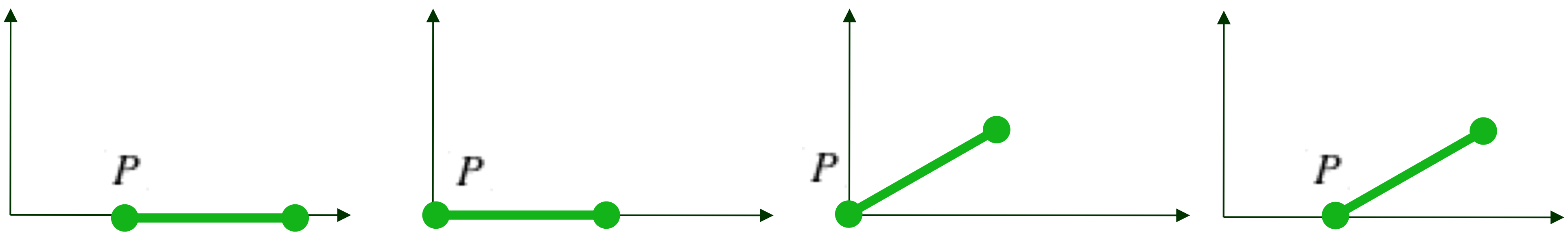


Incorrecto
Rot(30)



Correcto
Tr(-2,0) Rot(30) Tr(2,0)

Example: rotation of θ=30° of a straight line segment PQ about the point P(2,0)



$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 30 & -\sin 30 & 0 \\ \sin 30 & \cos 30 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D Affinities

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Identity

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda_x & 0 & 0 & 0 \\ 0 & \lambda_y & 0 & 0 \\ 0 & 0 & \lambda_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Reflection relative to YZ plane

Other 2D affinities

Rotation about z-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotation about y-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotation about x-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Affinities in GLM & GLSL (OpenGL Mathematics & OpenGL Shading Language)

http://www.c-jump.com/bcc/common/Talk3/Math/GLM/GLM.html

$$\begin{bmatrix} a & b & c & d \\ e & f & h & i \\ j & k & l & m \\ n & o & p & q \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + hz + iw \\ jx + ky + lz + mw \\ nx + oy + pz + qw \end{bmatrix}$$

myMatrix myVector transformedVector

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
```

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

Translation in GLM & GLSL

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

myMatrix myVector transformedVector

In C++, with GLM:

```
#include <glm/gtx/transform.hpp> // after <glm/glm.hpp>

glm::mat4 myMatrix = glm::translate(glm::mat4(), glm::vec3(10.0f, 0.0f, 0.0f));
glm::vec4 myVector(10.0f, 10.0f, 10.0f, 1.0f);
glm::vec4 transformedVector = myMatrix * myVector; // guess the result
```

In GLSL :

```
vec4 transformedVector = myMatrix * myVector;
```

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

Rotation and scaling in GLM

In C++ :

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::mat4 myScalingMatrix = glm::scale(2.0f, 2.0f ,2.0f);
```

In C++ :

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::vec3 myRotationAxis( ??, ??, ??);
glm::rotate( angle_in_degrees, myRotationAxis );
```

Example in OpenGL

P02, Example: Graphics application to draw a moving house

```
// Include standard headers
#include <stdio.h>
#include <stdlib.h>

// Include GLEW
#include <GL/glew.h>

// Include GLFW
#include <GLFW/glfw3.h>
GLFWwindow* window;

// GLM header file
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
using namespace glm;

// shaders header file
#include <common/shader.hpp>

// Vertex array object (VAO)
GLuint VertexArrayID;

// Vertex buffer object (VBO)
GLuint vertexbuffer;

// color buffer object (CBO)
GLuint colorbuffer;

// GLSL program from the shaders
GLuint programID;

GLint WindowWidth = 800;
GLint WindowHeight = 800;
float delta = 0.0;
```

```
// function prototypes
void transferDataToGPUMemory(void);
void cleanupDataFromGPU();
void draw();
```

P02, Example: Graphics application to draw a moving house

```
int main( void )
{
	// Initialise GLFW
	glfwInit();
	// Setting up OpenGL version and the like
	glfwWindowHint(GLFW_SAMPLES, 4);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make MacOS happy; should not be needed
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
	// Open a window
	window = glfwCreateWindow( WindowWidth, WindowHeight, "Moving House in 2D ", NULL, NULL);
	// Create window context
	glfwMakeContextCurrent(window);
	// Initialize GLEW
	glewExperimental = true; // Needed for core profile
	glewInit();
	// Ensure we can capture the escape key being pressed below
	glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
	// Dark blue background
	glClearColor(1.0f, 1.0f, 1.4f, 0.0f);
	// transfer my data (vertices, colors, and shaders) to GPU side
	transferDataToGPUMemory();
	// render scene for each frame
	do{
		draw();                      // drawing callback
		glfwSwapBuffers(window);     // Swap buffers
		glfwPollEvents();            // looking for input events
		if (delta < 10 )             // shift for the house
			delta += 0.1;
	} while (glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS && glfwWindowShouldClose(window) == 0 );
	// Cleanup VAO, VBOs, and shaders from GPU
	cleanupDataFromGPU();
	// Close OpenGL window and terminate GLFW
	glfwTerminate();
	return 0;
}
```

P02, Example: Graphics application to draw a moving house

```
void transferDataToGPUMemory(void)
{
        // VAO
        glGenVertexArrays(1, &VertexArrayID);
        glBindVertexArray(VertexArrayID);

        // Create and compile our GLSL program from the shaders
        programID = LoadShaders( "SimpleVertexShader.vertexshader", "SimpleFragmentShader.fragmentshader" );
        // vertices for 2 triangles
        static const GLfloat g_vertex_buffer_data[] = {
                0.0f,  0.0f,  0.0f,        20.0f, 0.0f,  0.0f,        20.0f, 20.0f, 0.0f,
                0.0f,  0.0f,  0.0f,        20.0f, 20.0f, 0.0f,        0.0f,  20.0f, 0.0f,
                0.0f,  20.0f, 0.0f,        20.0f, 20.0f, 0.0f,        10.0f, 30.0f, 0.0f,
        };

        // One color for each vertex
        static const GLfloat g_color_buffer_data[] = {
                1.0f,  0.0f,  0.0f,        1.0f,  0.0f,  0.0f,        1.0f,  0.0f,  0.0f,
                1.0f,  0.0f,  0.0f,        1.0f,  0.0f,  0.0f,        1.0f,  0.0f,  0.0f,
                0.0f,  1.0f,  0.0f,        0.0f,  1.0f,  0.0f,        0.0f,  1.0f,  0.0f,
        };

        // Move vertex data to video memory; specifically to VBO called vertexbuffer
        glGenBuffers(1, &vertexbuffer);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
        glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);
        // Move color data to video memory; specifically to CBO called colorbuffer
        glGenBuffers(1, &colorbuffer);
        glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
        glBufferData(GL_ARRAY_BUFFER, sizeof(g_color_buffer_data), g_color_buffer_data, GL_STATIC_DRAW);

}
```

P02, Example: Graphics application to draw a moving house

```
void cleanupDataFromGPU()
{
	glDeleteBuffers(1, &vertexbuffer);
	glDeleteBuffers(1, &colorbuffer);
	glDeleteVertexArrays(1, &VertexArrayID);
	glDeleteProgram(programID);
}
```

P02, Example:
Graphics application to draw a moving house

```
void draw (void)
{
        // Clear the screen
        glClear( GL_COLOR_BUFFER_BIT );
        // Use our shader
        glUseProgram(programID);

        // create the scene domain
        glm::mat4 mvp = glm::ortho(-40.0f, 40.0f, -40.0f, 40.0f);

        // retrieve the matrix uniform locations
         unsigned int matrix = glGetUniformLocation(programID, "mvp");
        glUniformMatrix4fv(matrix, 1, GL_FALSE, &mvp[0][0]);

        glm::mat4 trans;
        trans = glm::translate(trans, glm::vec3(delta, delta, 0.0f));
        unsigned int m = glGetUniformLocation(programID, "trans");
        glUniformMatrix4fv(m, 1, GL_FALSE, &trans[0][0]);

        // 1rst attribute buffer : vertices
        glEnableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
        glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0 );

        // 2nd attribute buffer : colors
        glEnableVertexAttribArray(1);
        glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
        glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, 0, (void*)0 );

        // Draw the 3 triangles !
        glDrawArrays(GL_TRIANGLES, 0, 9); // 9 indices starting at 0
        // Disable arrays of attributes for vertices
        glDisableVertexAttribArray(0);
        glDisableVertexAttribArray(1);
}
```

P02, Example:
Graphics application to draw a moving house

vertexshader.vs

```
#version 330 core

// Input vertex data and color data
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColor;

// Values that stay constant for the whole mesh.
uniform mat4 mvp;
uniform mat4 trans;

// Output fragment data
out vec3 fragmentColor;

void main()
{
	// position of each vertex in homogeneous coordinates
	gl_Position = mvp * trans * vec4(vertexPosition, 1.0);

	// the vertex shader just passes the color to fragment shader
	fragmentColor = vertexColor;
}
```

P02, Example:
Graphics application to draw a moving house

fragmentshader.fs

```
#version 330 core

// Interpolated values from the vertex shaders
in vec3 fragmentColor;

// Ouput data
out vec3 color;

void main()
{
	color = fragmentColor;
}
```

Summary:

. . . :

- Motivation
- Euclidean transformations: translation and rotation
- Euclidean geometry
- Homogeneous coordinates
- Affine transformations: translation, rotation, and shearing
- Matrix representation of affine transformations
- Composition of 2D and 3D transformations
- Geometric transformations in OpenGL/GLM
- Matrix operations in OpenGL/GLM and arbitrary transformations
- Example in OpenGL/GLM