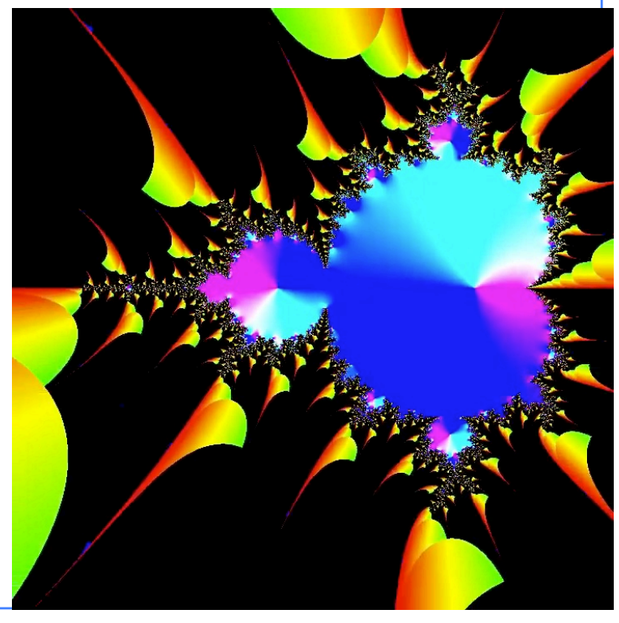
TSBK07/TSBK11/ETE378

Computer Graphics

Ingemar Ragnemalm, ISY



Lecture 2

2D transformations

Introduction to OpenGL

Lab situation

We will fit in three labs: Asgård, Olympen, Egypten

No booking needed. Work in pairs if possible.

If you work with Vulkan, use Egypten.

Duggas

5 duggas

Same for TSBK07 and TSBK11, but TSBK11 has lower demands.

Week 10 will not have a dugga (one less lecture)

Week 11 will be a retake for all duggas!

Take the duggas seriously!

Lecture questions

1. Why are we using 4x4 matrices for 3D operations?

2. Why do we use matrix multiplication for something as simple as scaling?

3. Why does the order matter between transformations, e.g a translation and a scaling?

4. How can you rotate around an arbitrary point?

5. What is a callback function and when are they practical in graphics?

6. What do the two shader stages do?

Fundamental vector operations

2D vector: **a** = (a_x, a_y) but can also be written as a column vector $\begin{bmatrix} a_x \\ a_y \end{bmatrix}$

A vector is *positional* or *directional*.

Vector addition: **a** + **b** = (a_x+b_x, a_y+b_y, a_z+b_z)

Multiplication with scalar: s * **a** = ($s*a_x$, $s*a_y$, $s*a_z$)

Magnitude: $|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}$ ← (a.k.a. length, norm)

(actually ||a|| but |a| is shorter)

Normalize: $\hat{\mathbf{a}} = |\mathbf{a}|^{-1} * \mathbf{a}$

Dot product

aka inner product

$$\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}| * |\mathbf{b}| * \cos\theta = a_x b_x + a_y b_y + a_z b_z$$

Properties of dot products:

Scalar value!

$\mathbf{a}\cdot\mathbf{b} = 0$ if $\mathbf{a}$ and $\mathbf{b}$ are orthogonal

$\mathbf{a}\cdot\mathbf{b} = \mathbf{b}\cdot\mathbf{a}$ Commutative

$\mathbf{a}\cdot(\mathbf{b} + \mathbf{c}) = \mathbf{a}\cdot\mathbf{b} + \mathbf{a}\cdot\mathbf{c}$ Distributive

Cross product

$$\mathbf{a} \times \mathbf{b} = \hat{\mathbf{n}} * |\mathbf{a}| * |\mathbf{b}| * \sin\theta =$$

$$(a_y b_z - a_z b_y,\ a_z b_x - a_x b_z,\ a_x b_y - a_y b_x)$$

Properties of cross products:

Vector!

Orthogonal to both **a** and **b**

a x **b** = (0,0,0) if **a** and **b** are parallel

a x **b** = - **b** x **a** Non-Commutative

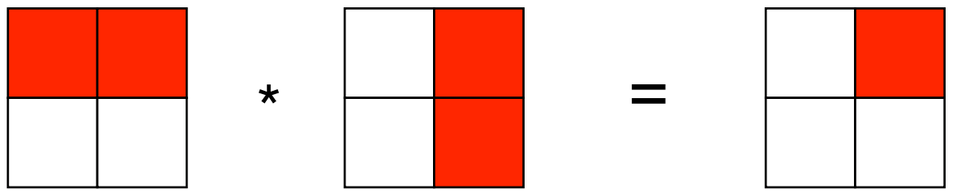
a x (**b** + **c**) = **a** x **b** + **a** x **c** Distributive

Matrix-vector multiplication

$$M * \mathbf{a} = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} * \begin{bmatrix} a_x \\ a_y \end{bmatrix} = \begin{bmatrix} M_{11}a_x + M_{12}a_y \\ M_{21}a_x + M_{22}a_y \end{bmatrix}$$

Matrix-matrix multiplication

$$M * N = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} * \begin{bmatrix} N_{11} & N_{12} \\ N_{21} & N_{22} \end{bmatrix} = \begin{bmatrix} M_{11}N_{11} + M_{12}N_{21} & M_{11}N_{12} + M_{12}N_{22} \\ M_{21}N_{11} + M_{22}N_{21} & M_{21}N_{12} + M_{22}N_{22} \end{bmatrix}$$



Properties of matrix multiplications:

Associative:

A*B*C = (A*B)*C = A*(B*C)

Non-commutative:

A*B and B*A not guaranteed to be equal!

Identity matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Zeroes everywhere except on diagonal, which is 1

$IA = A$

The ”one” of matrices

Matrices may have any size, 2x2, 3x3, 4x1, 100x2...

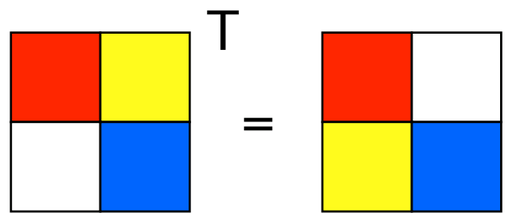
A vector is also a single-column matrix!

We mainly care about 3x3 and 4x4 matrices in this course!

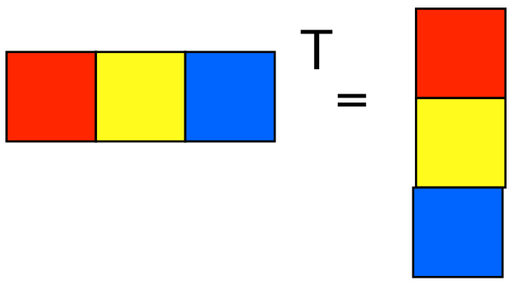
Some operations on matrices:

Inverse $AA^{-1} = I$

$$ABB^{-1} = A$$



Transpose A^T, flip over diagonal



Dot product = matrix multiplication of a row and a column matrix! Thus:

$a \cdot b = a^T b$

OpenGL

where it fits

what it contains

how you work with it

OpenGL

The cross-platform graphics library!

Open = Open specification

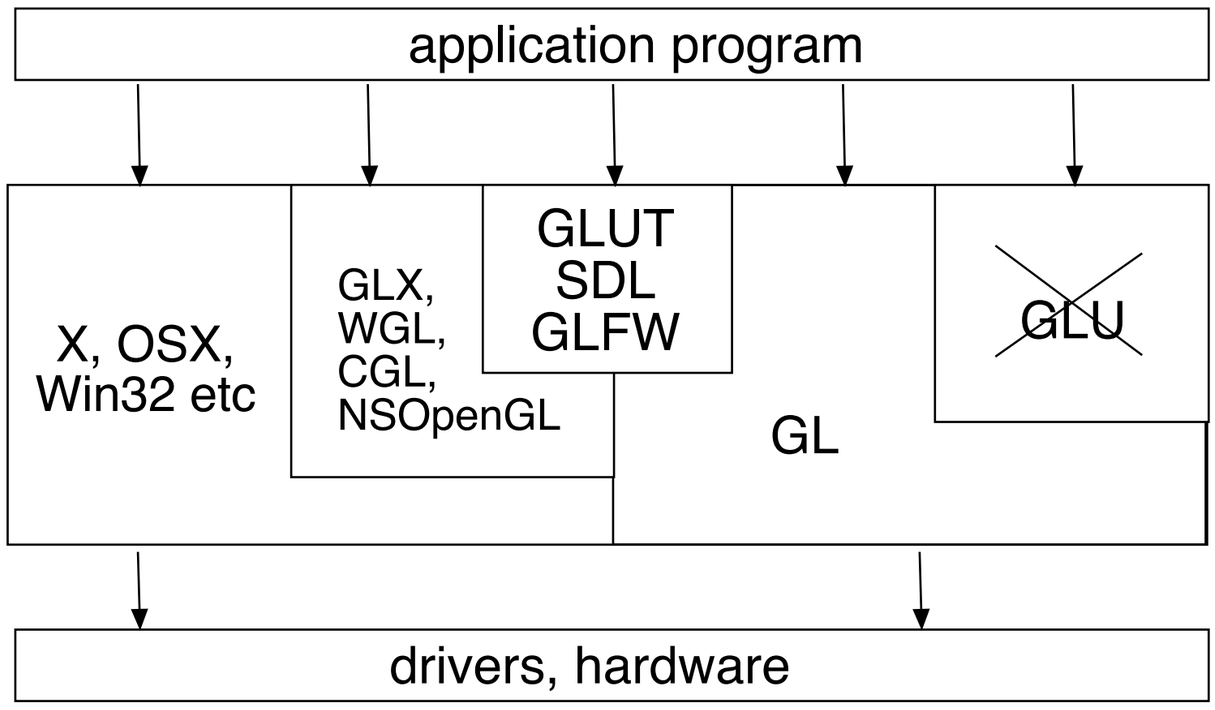
Runs everywhere - Linux, Mac, Windows...

Any language

Three branches:

OpenGL - ordinary computers
OpenGL ES - phones and tablets
WebGL - web browsers

OpenGL and Related APIs



OpenGL parts:

GL = Graphics Library (core lib)

GLU = GL Utilities (no longer supported)

GLX, WGL, CGL, NSOpenGL = System dependent libraries

GLUT = GL Utility Toolkit (optional)

FreeGLUT, MicroGlut

Also note: SDL (Simple Directmedia Layer)

GLFW (similar to GLUT)

OpenGL versions

OpenGL 1-2: Old OpenGL. Avoid!

Big leap here!

OpenGL 3.2 and up: Modern OpenGL! Latest version 4.6.

Even hotter (but harder): Vulkan (released 2016)

Simple OpenGL example

A single triangle

• upload to the GPU once

• draw any time you want

but also

• upload shader to specify transformations and colors

Vital concepts

- VAO, VBO
- Vertex shader
- Fragment shader

VAO and VBO

VBO = Vertex Buffer Object

A reference to an array of vertices on the GPU

VAO = Vertex Array Object

A reference to a set of buffers

What is a shader?

Small program kernel that runs on the GPU!

Gives you great freedom to specify certain operations.

Run in parallel on multiple cores (hundreds or even thousands!) in the GPU! Extremely efficient!

Executed for each vertex and each fragment every frame!

Vertex shader

Specifies transformations on each vertex

Translations, rotations...

Short program with data sent from the main program

In the example: Pass-through

Fragment shader

Specifies color of each pixel

Short program with data sent from the main program or vertex shader.

In the example: Set-to-white

Sample main program (in C)

```
void main( int argc, char** argv )
{
   glutInit(argc, argv));
   glutInitContextVersion(3, 2);
   glutCreateWindow(”GL3 white triangle example”);
   glutDisplayFunc( display );
   init();
   glutMainLoop();
}
```

Mostly MicroGlut calls here. Looks different for different ”wrapper” libraries

Sample shaders

Minimal, doing pretty much nothing

Vertex shader:
Specifying positioning
(pass-through)

```
#version 150

in  vec3 in_Position;

void main(void)
{
  gl_Position = vec4(in_Position, 1.0);
}
```

Fragment shader:
Specifying pixel colors
(set-to-white)

```
#version 150

out vec4 out_Color;

void main(void)
{
  out_Color = vec4(1.0);
}
```

Name conventions

All calls prefixed by gl

All constants prefixed by GL

Some calls suffixed by number, f,d,v

Example

glUniform3fv(index, count, v)

Number of components

2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)

Data type

b - byte
ub - unsigned byte
s - short integer
us - unsigned short
i - int (integer)
ui - unsigned int
f - float
d - double

Vector

If present: arguments are array

If omitted: arguments as separate scalars:

glUniform1f(loc, time)

OpenGL Initialization

Set up whatever state you're going to use

```
void init( void )
{
   glClearColor( 0.0, 0.0, 0.0, 1.0 );
   glClearDepth( 1.0 );
   glEnable( GL_DEPTH_TEST );
   glEnable( GL_CULL_FACE );
}
```

May also include uploading of geometry, loading shaders etc.

OpenGL type names

Standard types redefined prefixed GL
(for compatibility reasons)

From gl.h:

```
typedef unsigned long GLenum;
typedef unsigned char GLboolean;
typedef unsigned long GLbitfield;
typedef signed char GLbyte;
typedef short GLshort;
typedef long GLint;
typedef long GLsizei;
typedef unsigned char GLubyte;
typedef unsigned short GLushort;
typedef unsigned long GLuint;
typedef float GLfloat;
typedef float GLclampf;
typedef double GLdouble;
typedef double GLclampd;
typedef void GLvoid;
```

MicroGLUT Callback Functions

Routines for MicroGlut (same as GLUT and FreeGLUT) to call when something happens

• window resize or redraw
• user input (mouse, keyboard)

“Register” callbacks with MicroGlut

Common ones:

```
glutDisplayFunc( display );
glutMouseFunc( mouse );
glutKeyboardFunc( keyboard );
```

The MicroGlut API is based on GLUT but *not* identical.

Rendering Callback. Do all of our drawing here!

```
glutDisplayFunc( display );


void display(void)
{
    // clear the screen
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Select VAO and draw
    glBindVertexArray(vertexArrayObjID);
    glDrawBuffer(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, NULL);

    glutSwapBuffers();
}
```

Built-in timer

glutRepeatingTimer. Triggers repeated calls to the update function.

Use for animation and continuous update

Simple Example, geometry

A shape can be hard-coded in an array

```
GLfloat vertices[] = {  -0.5f,-0.5f,0.0f,
                        -0.5f,0.5f,0.0f,
                         0.5f,-0.5f,0.0f };
GLfloat colors[] = { 1.0f, 0.0f, 0.0f,
                     0.0f, 1.0f, 0.0f,
                     0.0f, 0.0f, 1.0f };
```

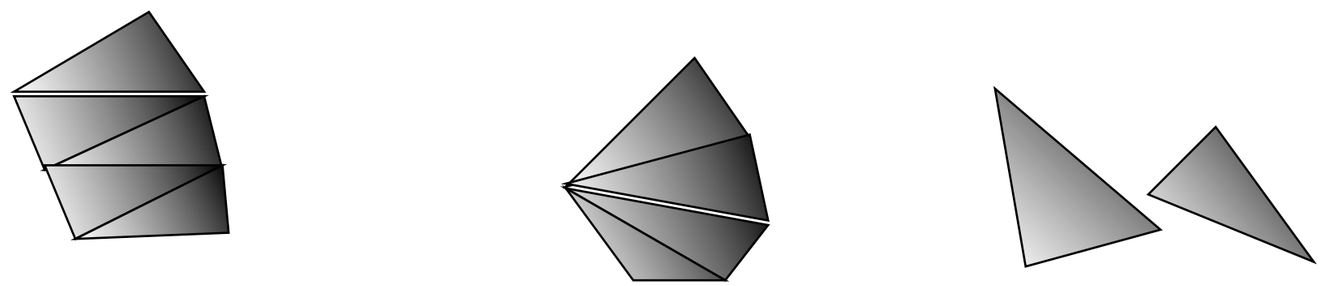
But usually you read data from files.

Third method: Procedural shapes.

OpenGL Geometric Primitives

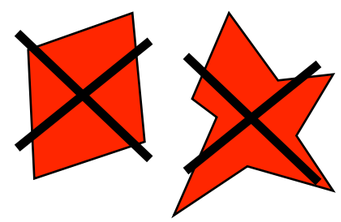
All geometric primitives are specified by vertices:

GL_POINTS GL_LINES GL_LINE_STRIP GL_LINE_LOOP



GL_TRIANGLE_STRIP GL_TRIANGLE_FAN GL_TRIANGLES

Why not QUAD or POLYGON?



Uploading to the GPU is a bit hairy.

```
// Allocate and activate Vertex Array Object
  glGenVertexArrays(1, &vertexArrayObjID);
  glBindVertexArray(vertexArrayObjID);

  // VBO for vertex data
  glGenBuffers(1, &vertexBufferObjID);
  glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObjID);

  // Data to VBO
  glBufferData(GL_ARRAY_BUFFER, 9*sizeof(GLfloat), vertices,
GL_STATIC_DRAW);
  glVertexAttribPointer(glGetAttribLocation(program, "in_Position"), 3,
                GL_FLOAT, GL_FALSE, 0, 0);
  glEnableVertexAttribArray(glGetAttribLocation(program,
"in_Position"));
```

Drawing

The typical code sequence for drawing shapes follow this list:

- Activate the appropriate shaders
- Upload variables (matrices) to the shaders
- Activate the VAO (buffers) that you want to draw
- Draw with glDrawBuffers (easiest) or glDrawElements (best)

OpenGL color

RGBA or indexed color

We primarily use RGBA

- R = red
- G = green
- B = blue
- A = alpha

Values are specified from 0.0 to 1.0

Alpha = transparency!

Affine transformations in OpenGL

Rotation, translation, scaling and more. Rotation matrix:

```
GLfloat rotationMatrix[] = { cos(a), -sin(a), 0.0f, 0.0f,
                             sin(a), cos(a), 0.0f, 0.0f,
                               0.0f, 0.0f, 1.0f, 0.0f,
                             0.0f, 0.0f, 0.0f, 1.0f };
```

Multiply your matrices and upload to the vertex shader!

```
glUniformMatrix4fv(glGetUniformLocation(program, "myMatrix"), 1,
                                        GL_FALSE, rotationMatrix);
```

DEMO TIME: Let's see apply this to the triangle.

In practice: Hide common code

Doesn't this look slightly more practical? Package creation of common matrices.

```
void SetRotationMatrix(GLfloat a, GLfloat *m)
{
  m[0] = cos(a); m[1] = -sin(a); m[2] = 0; m[3] = 0.0;
  m[4] = sin(a); m[5] = cos(a); m[6] = 0; m[7] = 0.0;
  m[8] = 0; m[9] = 0; m[10] = 1.0; m[11] = 0.0;
  m[12] = 0.0; m[13] = 0.0; m[14] = 0.0; m[15] = 1.0;
}
```

VectorUtils3 (or 4) is used in the lab for this. The function above is there called Rz().
It also includes Rx() and Ry().

Multiple transformations

You can multiply in the vertex shader...

```
in  vec3 inPosition;
uniform mat4 mdlMatrix;
uniform mat4 camMatrix;
uniform mat4 projMatrix;

void main(void)
{
  gl_Position = projMatrix * camMatrix * mdlMatrix *
vec4(inPosition, 1.0);
}
```

but global transformations are better done on CPU. (Why?)

Multiple transformation example

Vertex shader still only multiplies one matrix

Uses our vector/matrix library ”VectorUtils4”

```
total = Rz(Pi/4, rot) * T(0, -1, 0, trans);

glUniformMatrix4fv(glGetUniformLocation(program,
        "myMatrix"),  1, GL_TRUE, total);
```

Beware that the order of transformations matter!

Note 1: This can be optimized. How?

Note 2: The upload is simplified in VectorUtils4 as uploadMat4ToShader().

Demo

Spaceship. We use transformations to move it around.

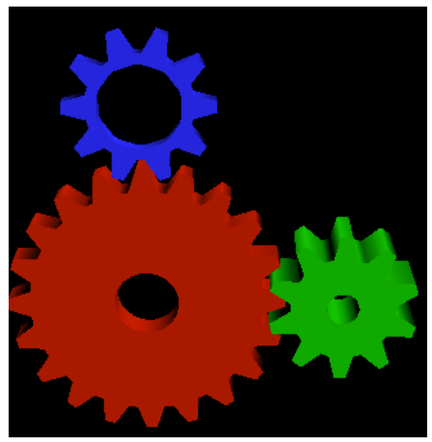
Beware of old OpenGL demos!

Old OpenGL was easy to use but fooled people to low-performing methods. Avoid demos using

glTranslate, glRotate, glScale
glPushMatrix, glPopMatrix
glBegin, glVertex, glNormal, glEnd

etc

But we *can* modernize them!



Summary:

Goal of this introduction

To provide you with some basic insights in how to use OpenGL to try simple graphics operations.

To see how the graphics operations (transformations) I just presented can appear in working code.

We do not - yet - cover high-performance methods with large models and large scenes. But we will.

Next lecture:

Adding the third dimension!

All this and more. Transformations in 3D. Camera placement and projection.