GeoGebra

Computer Graphics Seminar

MTAT.03.305

Spring 2018

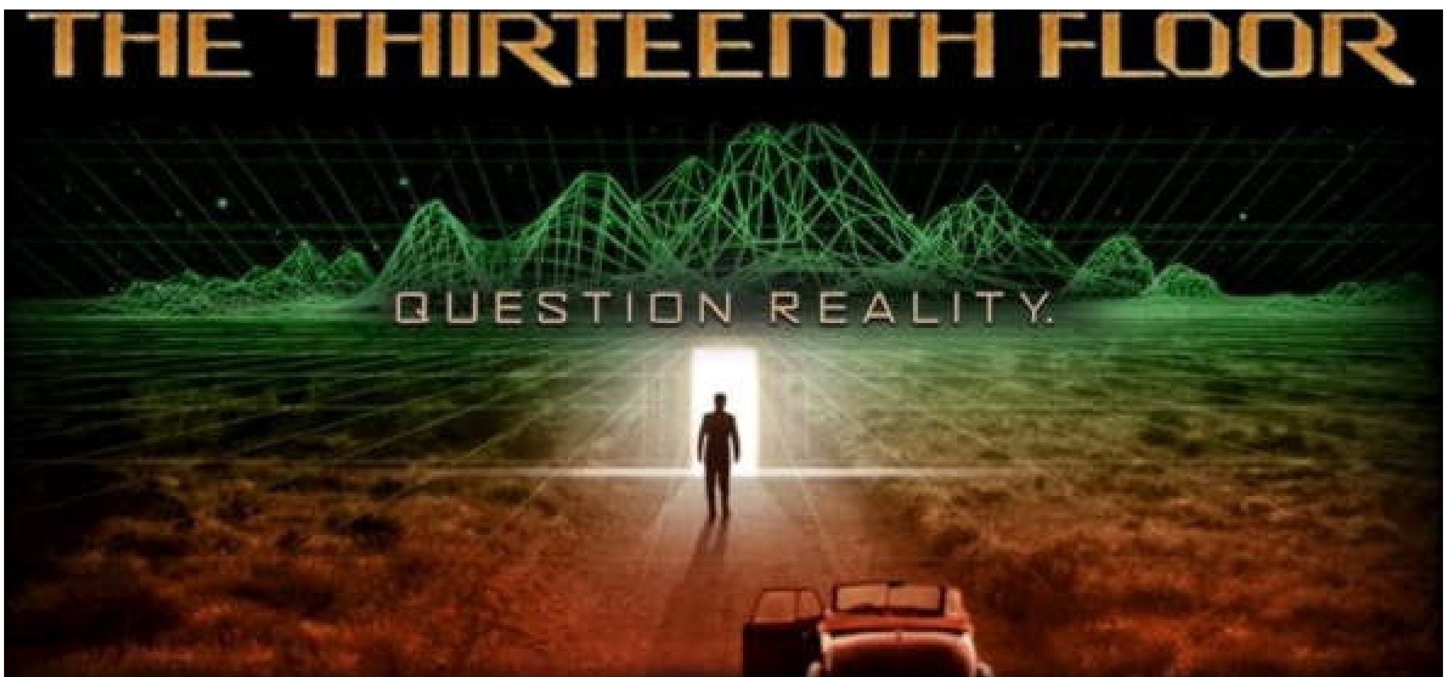
Raimond Tunnel

TARTU ÜLIKOOL · UNIVERSITAS TARTUENSIS · 1632

IT Akadeemia toetab Skype™

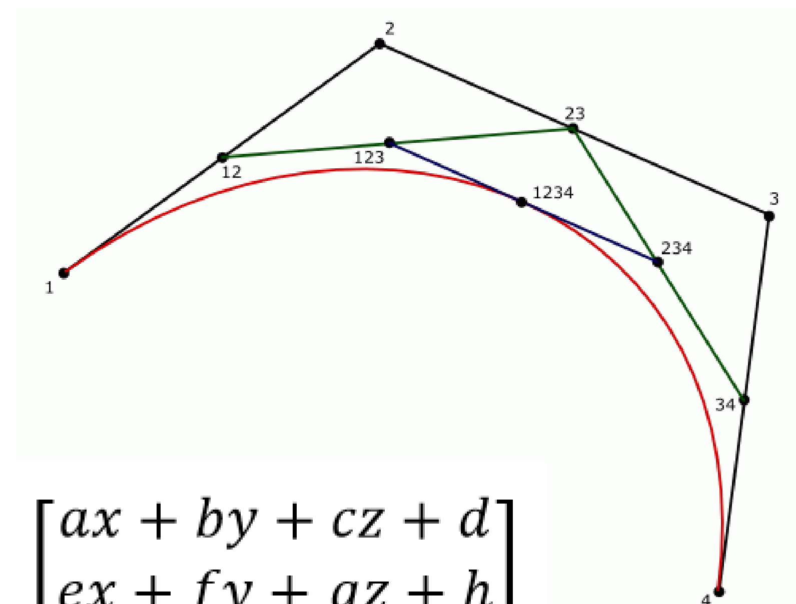
Computer Graphics

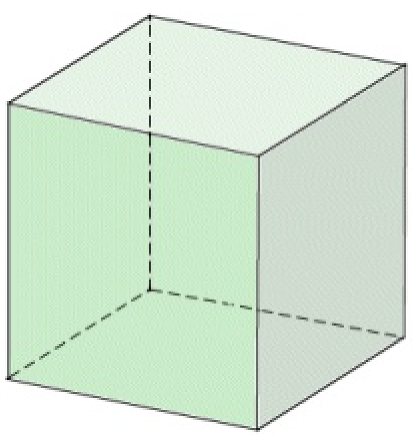
- Graphical illusion via the computer
- Displaying something meaningful (incl art)



Math

- Computers are good at... computing.
- To do computer graphics, we need math for the computer to compute.
- Geometry, algebra, calculus.

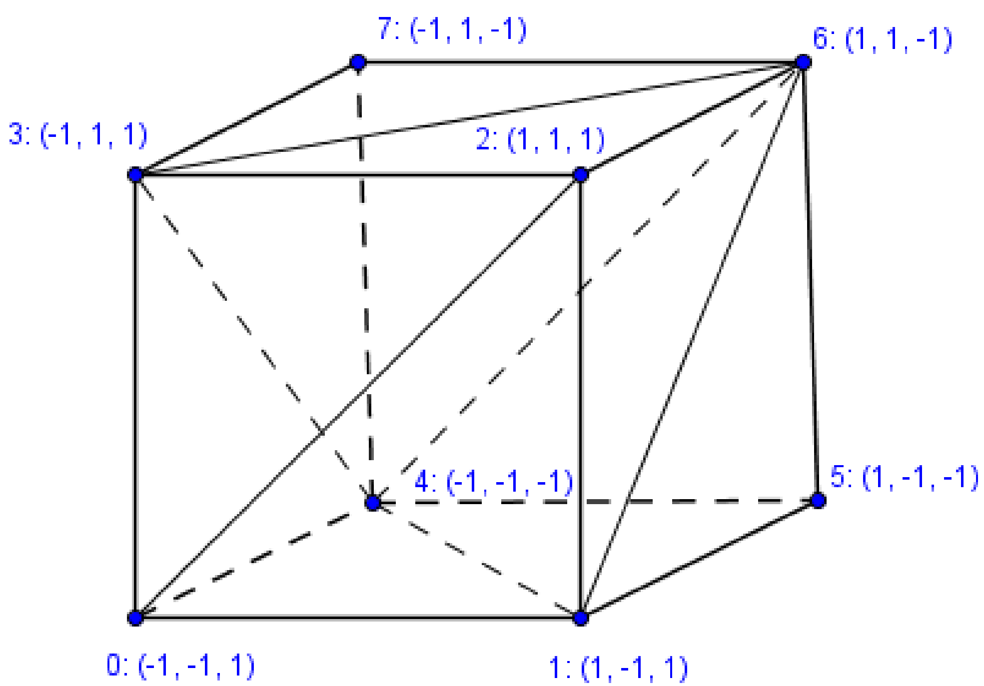




$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix}$$

Math

- For actually creating and manipulating objects in 3D we need:
  - **Analytic geometry** – math about coordinate systems
  - **Linear algebra** – math about vectors and spaces

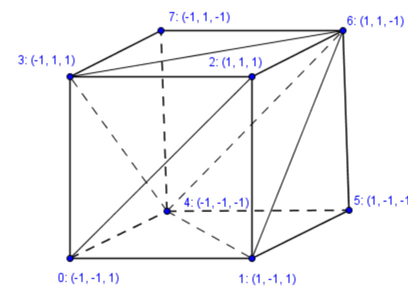


Skills for Computer Graphics

- Mathematical understanding

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

- Geometrical (spatial) thinking



- Programming

```
GLuint vaoHandle;
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(vaoHandle)
```

- Visual creativity & aesthetics

Point

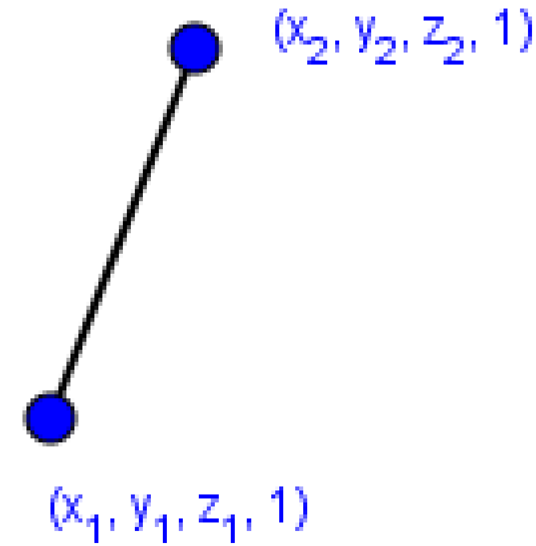
- Simplest geometry primitive
- In homogeneous coordinates:

$$(x, y, z, w),\ w \neq 0$$

- Represents a point $(x/w, y/w, z/w)$
- Usually you can put **w = 1 for points**
- Actual division will be done by GPU later

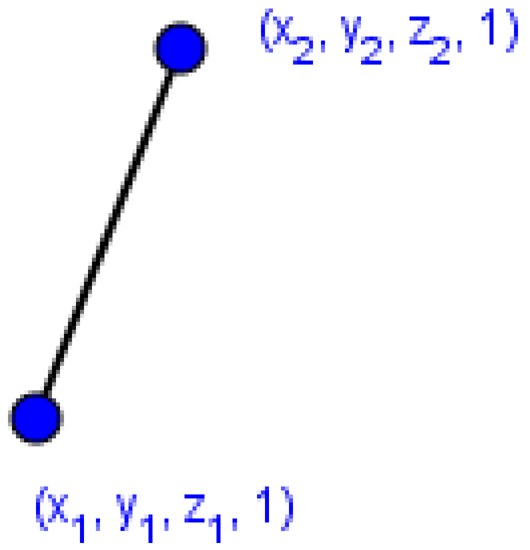
Line (segment)

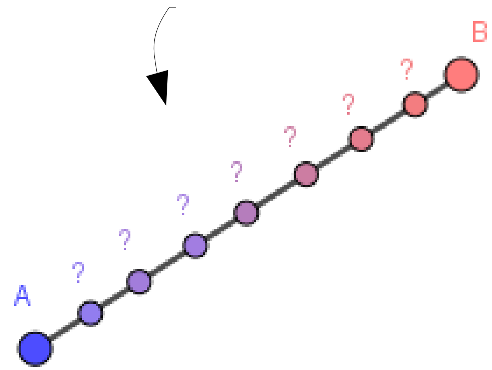
- Consists of:
  - 2 endpoints
  - *Infinite* number of points between
- Defined by the endpoints
- Interpolated and rasterized in the GPU



Line (segment)

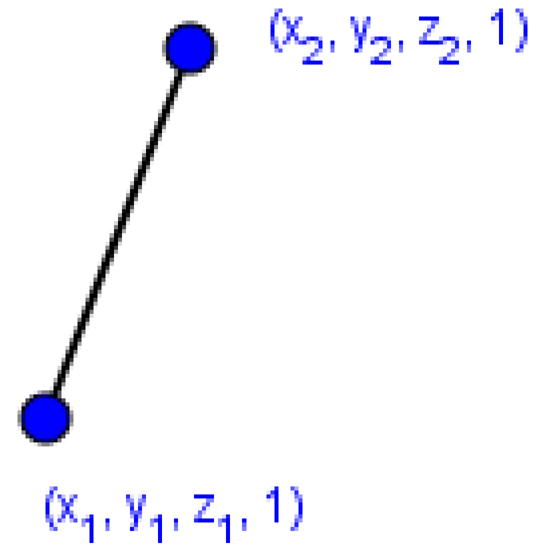
- Consists of:
  - 2 endpoints
  - *Infinite* number of points between
- Defined by the endpoints
- Interpolated and rasterized in the GPU

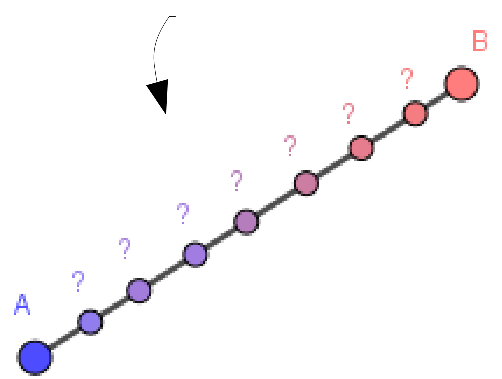


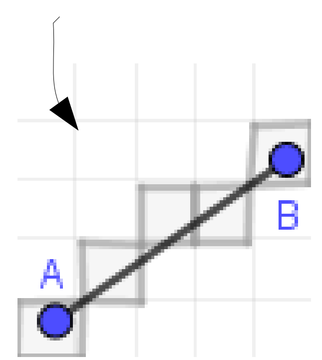


Line (segment)

- Consists of:
  - 2 endpoints
  - *Infinite* number of points between
- Defined by the endpoints
- Interpolated and rasterized in the GPU

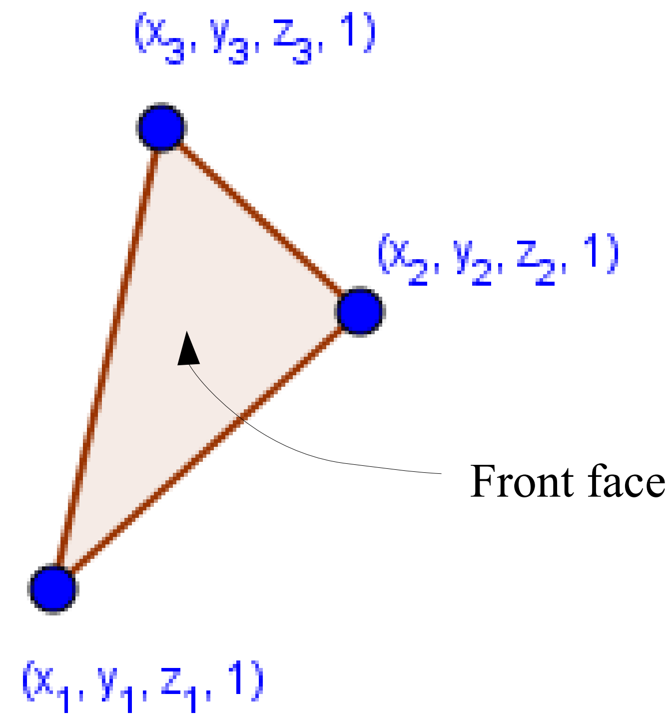






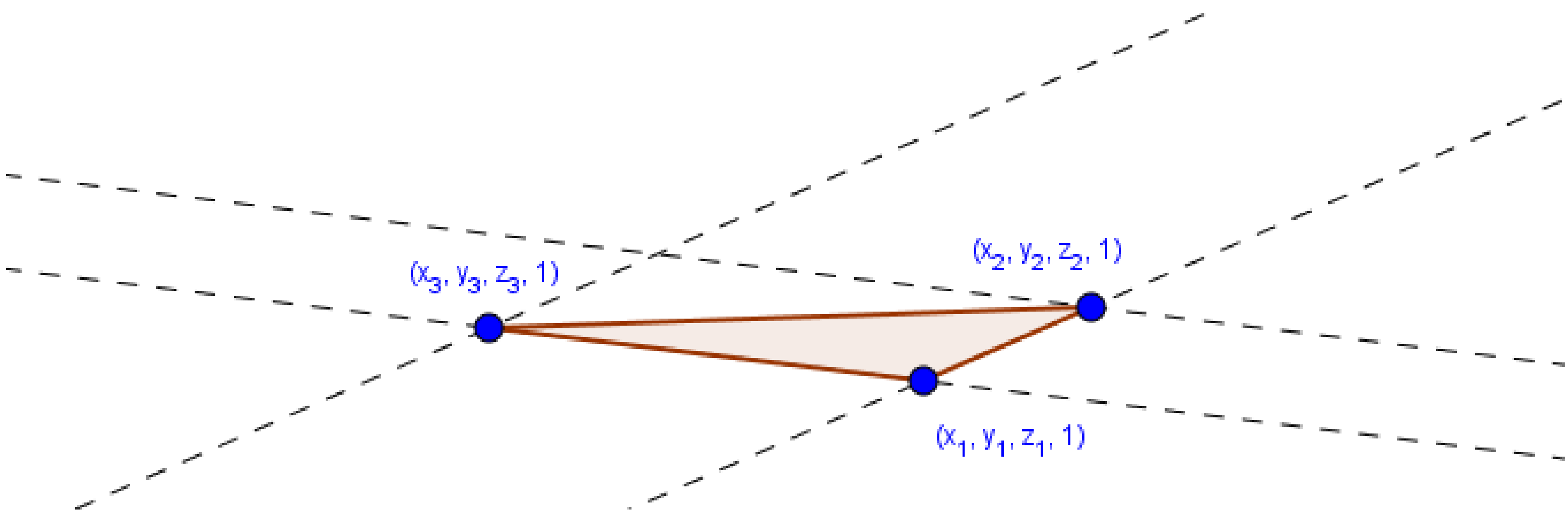
Triangle

- Consists of:
  - 3 points called vertices
  - 3 lines called edges
  - 1 face
- Defined by 3 vertices
- Face interpolated and rasterized in the GPU
- Counter-clockwise order defines front face



Why triangles?

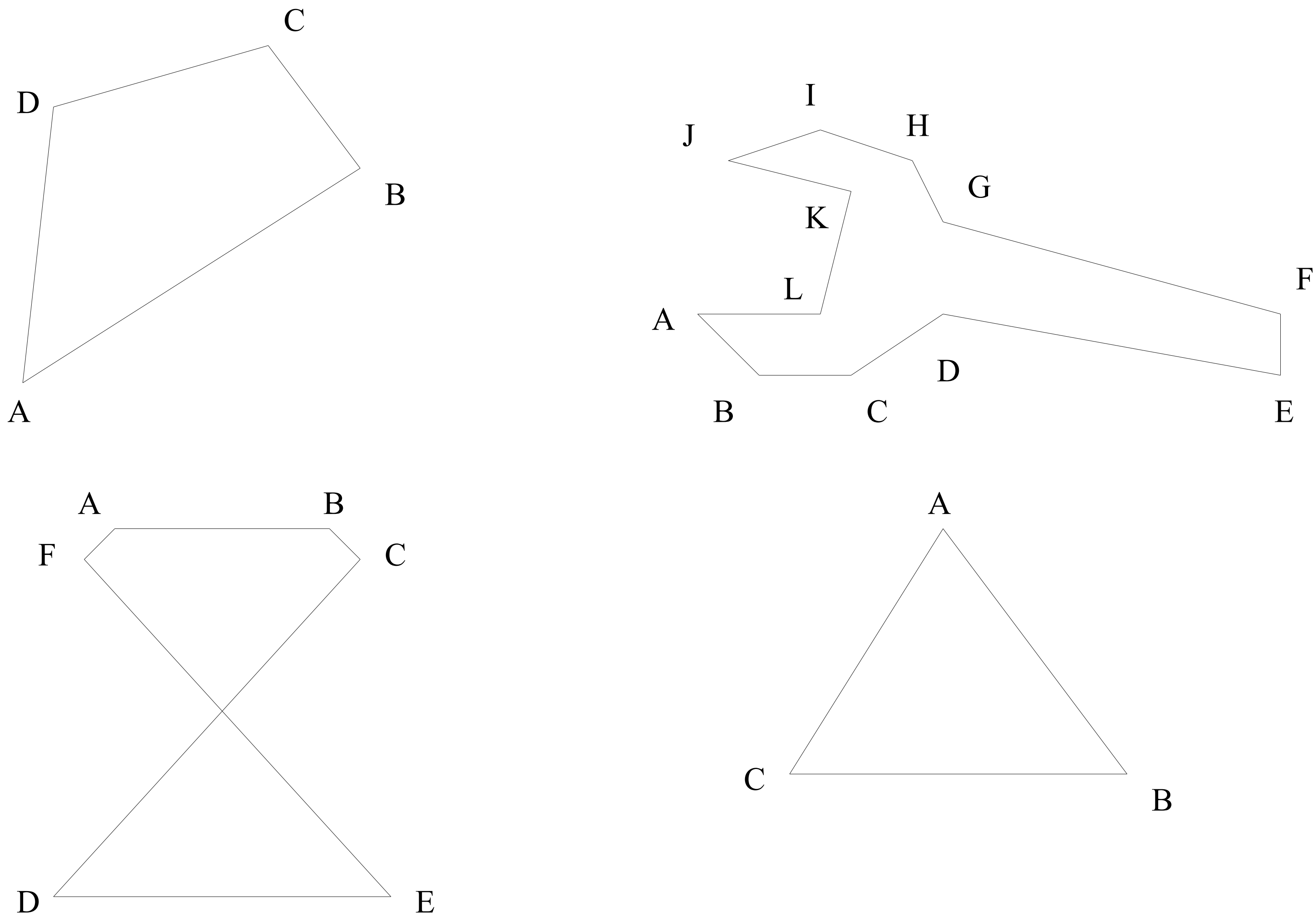
- They are in many ways the simplest polygons
  - 3 different points always form a plane
  - Easy to rasterize (fill the face with pixels)
  - Every other polygon can be converted to triangles



Why triangles?

- They are in many ways the simplest polygons
  - 3 different points always form a plane
  - Easy to rasterize (fill the face with pixels)
  - Every other polygon can be converted to triangles
- OpenGL used to support other polygons too
  - Must have been:
    - **Simple** – No edges intersect each other
    - **Convex** – All points between any two points are inner points

Examples of polygons



OpenGL < 3.1 primitives

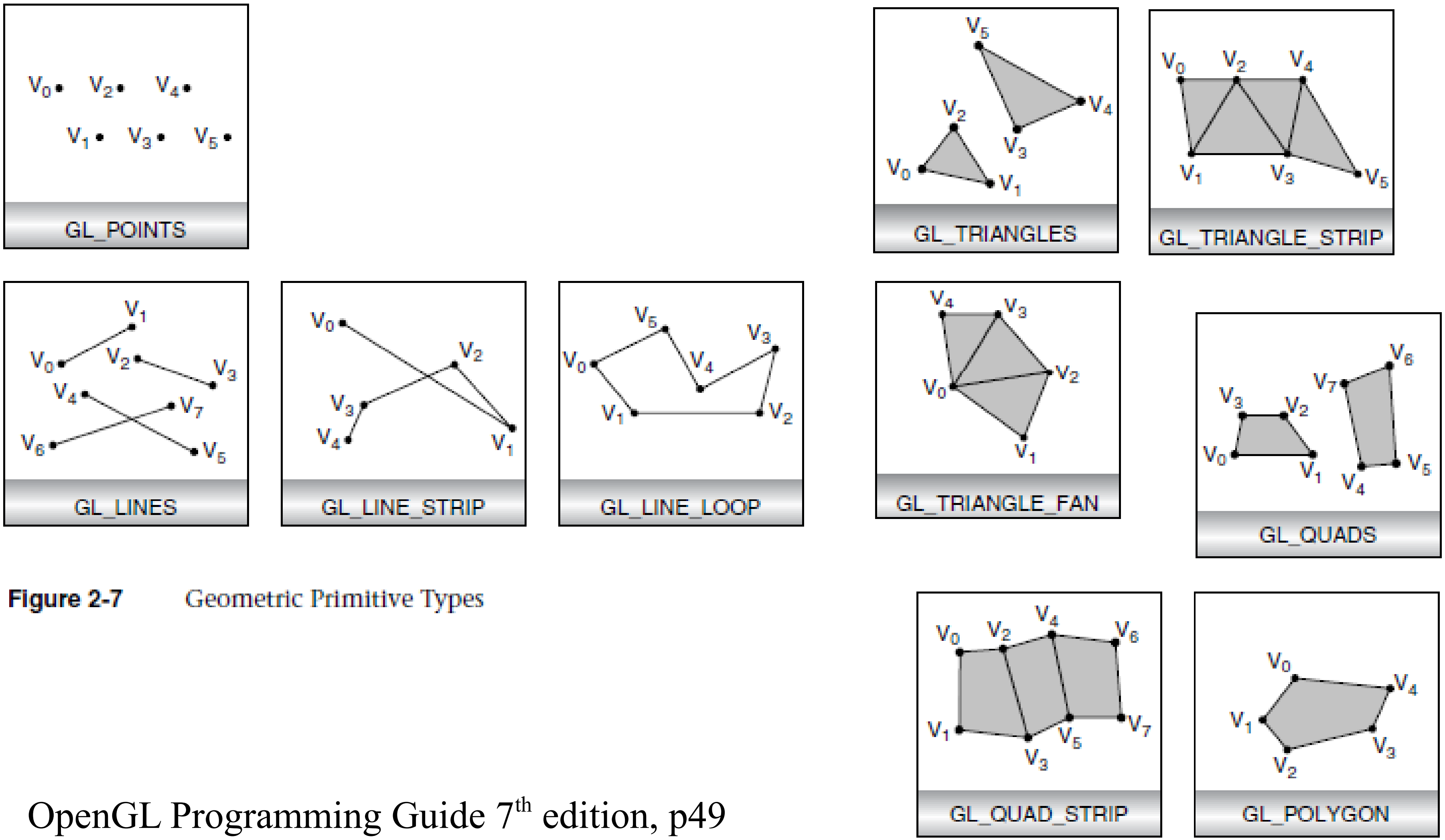


Figure 2-7 Geometric Primitive Types

OpenGL Programming Guide 7th edition, p49

After OpenGL 3.1

Table 3.1 OpenGL Primitive Mode Tokens

Primitive Type	**OpenGL Token**
Points	GL_POINTS
Lines	GL_LINES
Line Strips	GL_LINE_STRIP
Line Loops	GL_LINE_LOOP
Independent Triangles	GL_TRIANGLES
Triangle Strips	GL_TRIANGLE_STRIP
Triangle Fans	GL_TRIANGLE_FAN

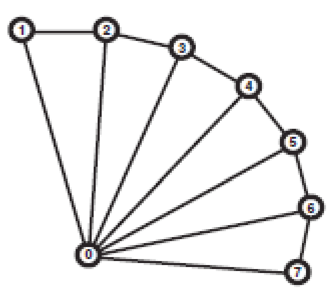


Figure 3.2 Vertex layout for a triangle fan

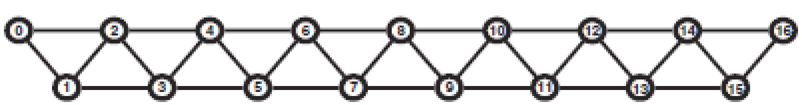
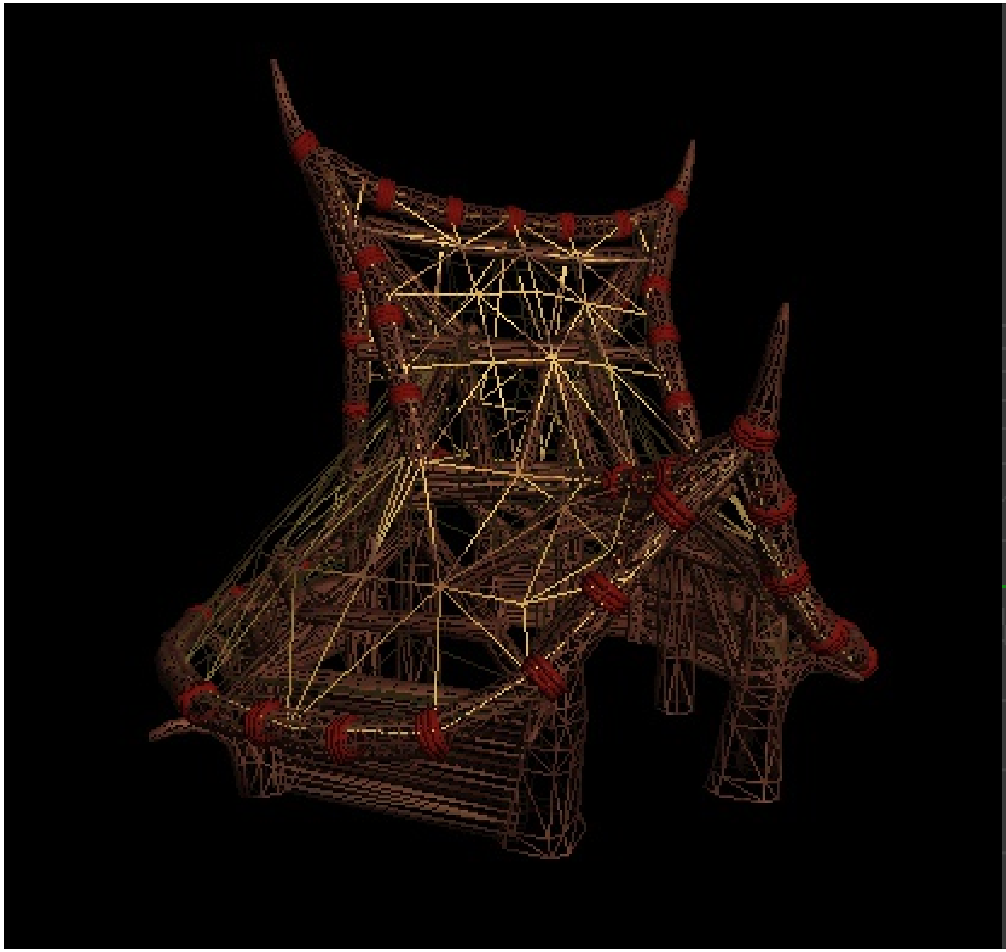


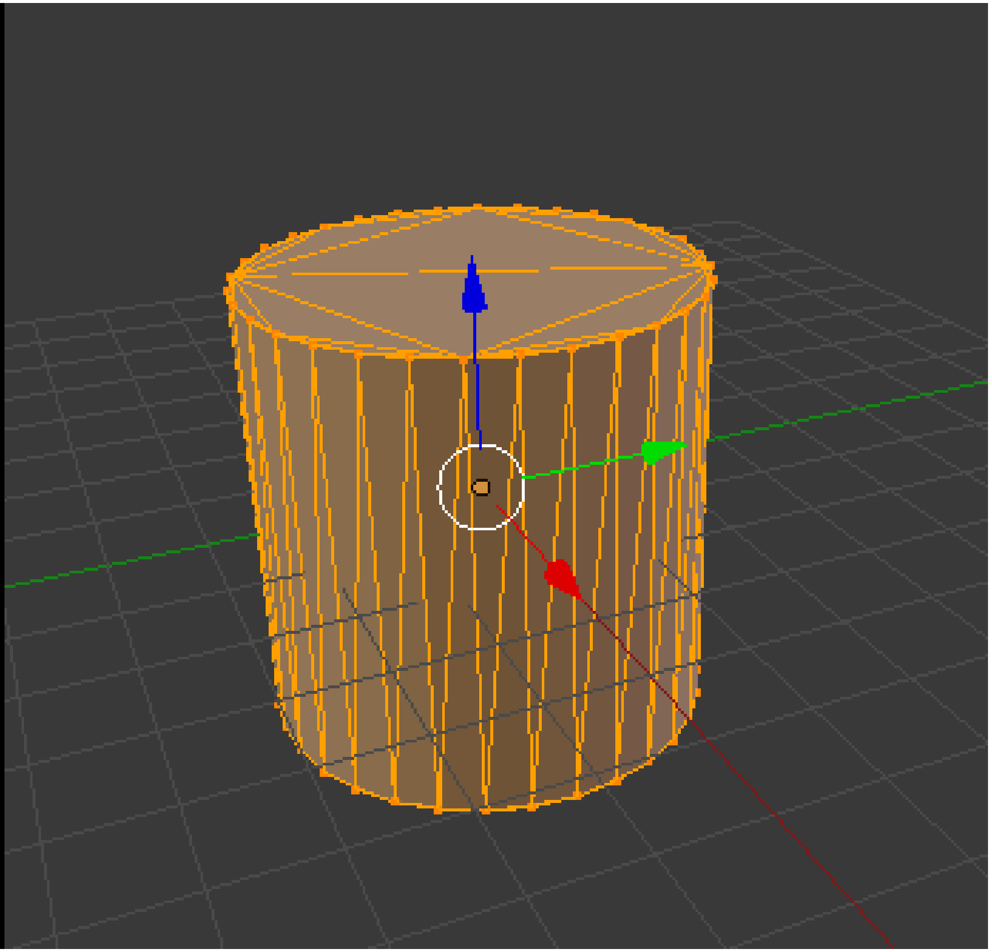
Figure 3.1 Vertex layout for a triangle strip

OpenGL Programming Guide 8th edition, p89-90

In the beginning there were points

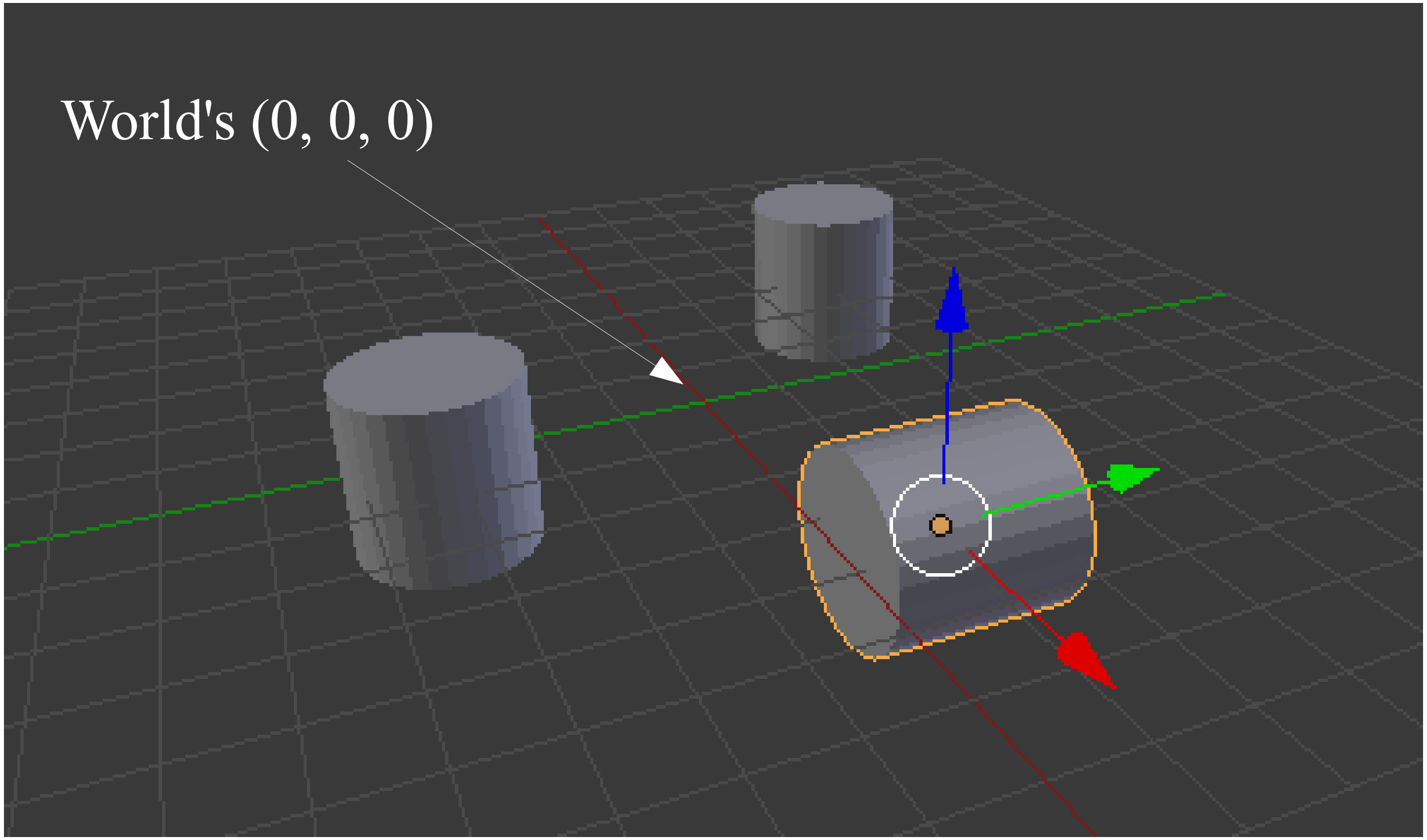
- We can now define our geometric objects!





In the beginning there were points

- We can now define our geometric objects!
- We want to move our objects!



Transformations

- Homogeneous coordinates allow easy:
  - Linear transformations
    - Scaling, reflection
    - Rotation
    - Shearing
  - Affine transformations
    - Translation (moving / shifting)
  - Projection transformations
    - Perspective
    - Orthographic

Actually these we could do without homogeneous coordinates...

This too...

Transformations

- Every transformation is a function
- As you remember from Algebra, linear functions can be represented as matrices

$$f(v) = \begin{pmatrix} 2 \cdot x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Linear function, which increases the first coordinate two times.

Same function as a matrix

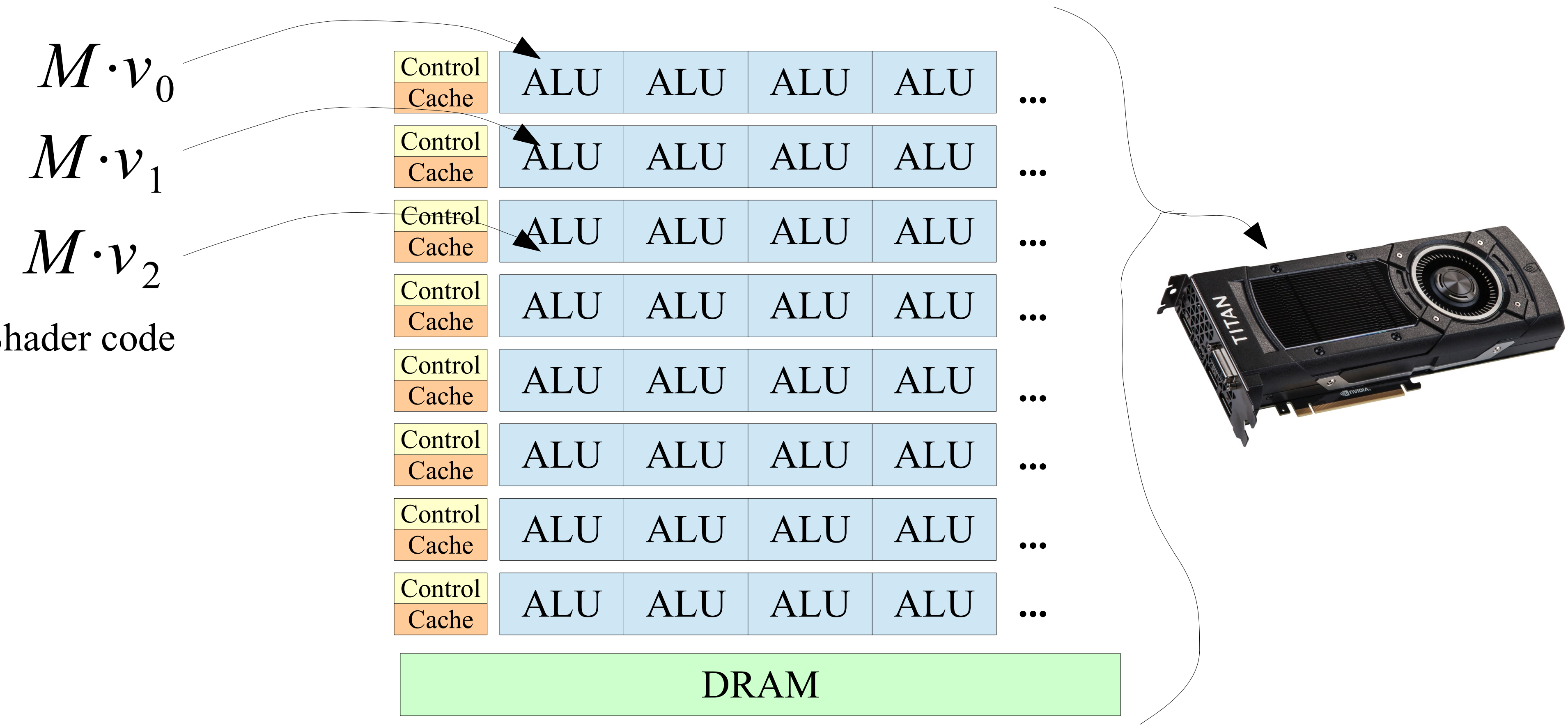
$$v \in R^3$$

$$v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Column-major format

Transformations

- GPU-s are built for doing transformations with matrices on points (vertices).



Transformations

- GPU-s are built for doing transformations with matrices on points (vertices).

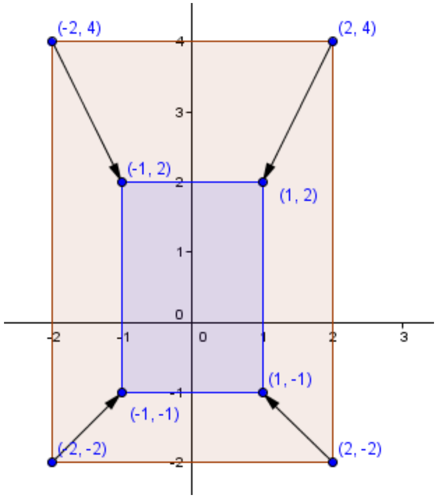
- Linear transformations satisfy:

$$f(a_1 x_1 + \ldots + a_n x_n) = a_1 f(x_1) + \ldots + a_n f(x_n)$$

We don't use homogeneous coordinates at the moment, but they will be back...

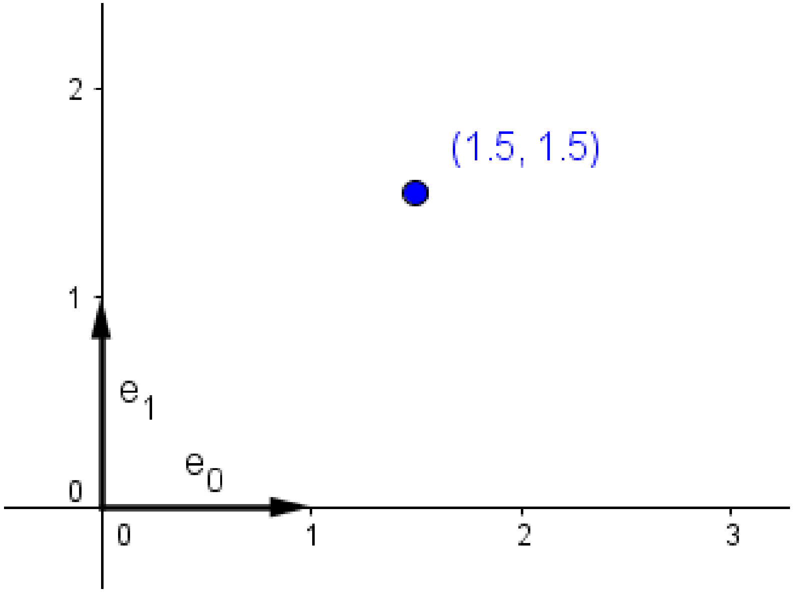
Linear Transformation

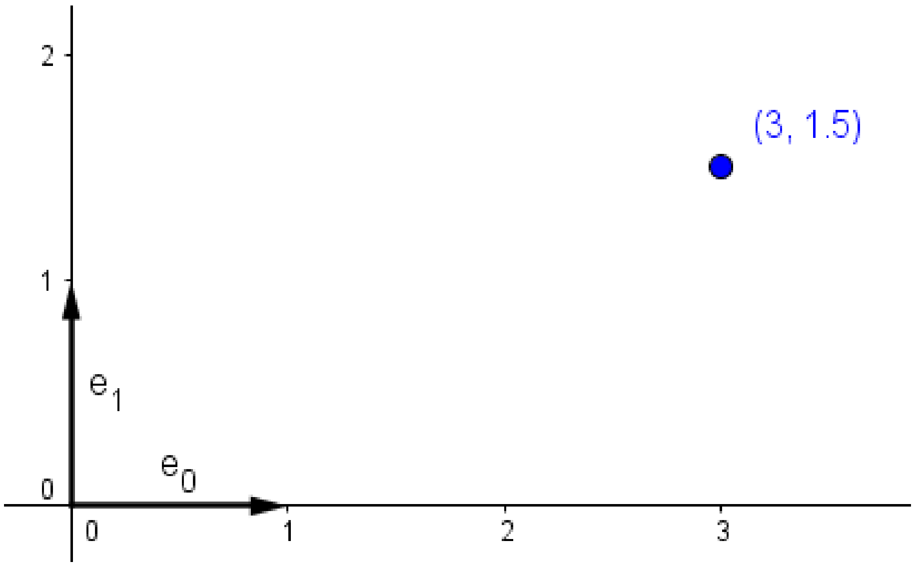
Scale



Scaling

- Multiplies the coordinates by a scalar factor.



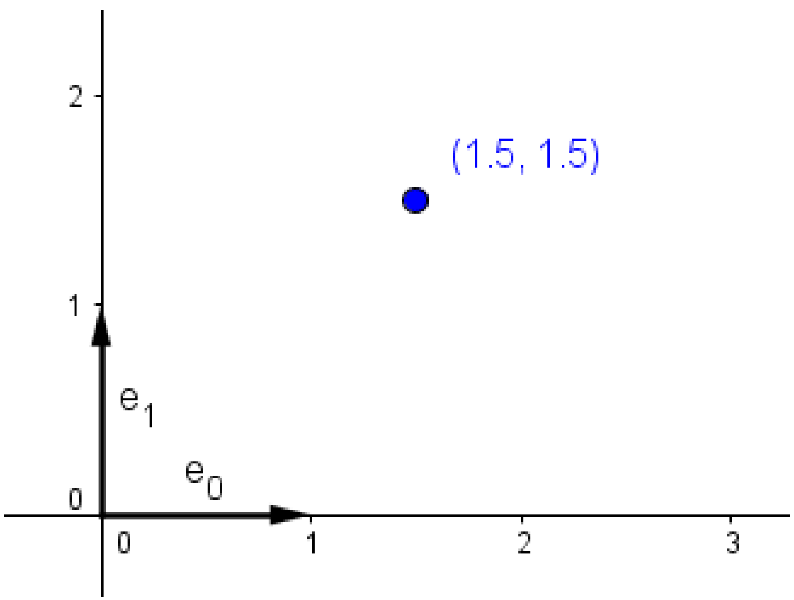


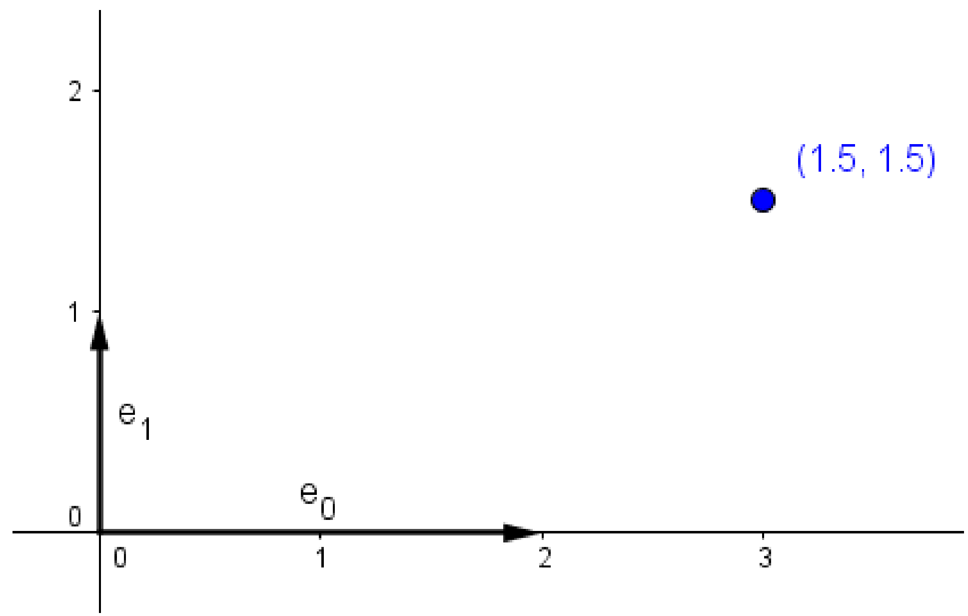
$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} = \begin{pmatrix} 3 \\ 1.5 \end{pmatrix}$$

Scaling

- Multiplies the coordinates by a scalar factor.
- Scales the standard basis vectors / axes.





$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix} = e_0$$

$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = e_1$$

Scaling

- In general we could scale each axis

$$\begin{pmatrix} a_x & 0 & 0 \\ 0 & a_y & 0 \\ 0 & 0 & a_z \end{pmatrix}$$

a_x – x-axis scale factor

a_y – y-axis scale factor

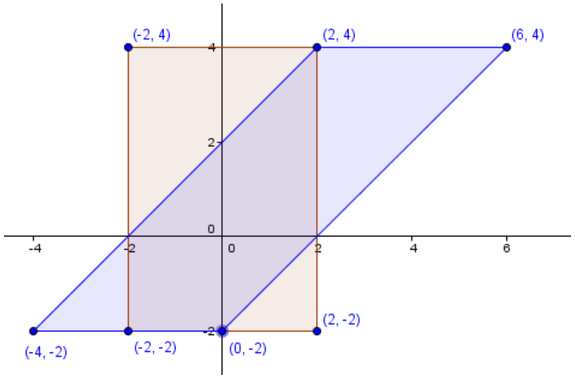
a_{x} – z-axis scale factor

- If some factor is negative, this matrix will reflect the points from that axis. Thus we get reflection.

What happens to out triangles when an odd number of factors are negative?

Linear Transformation

Shear



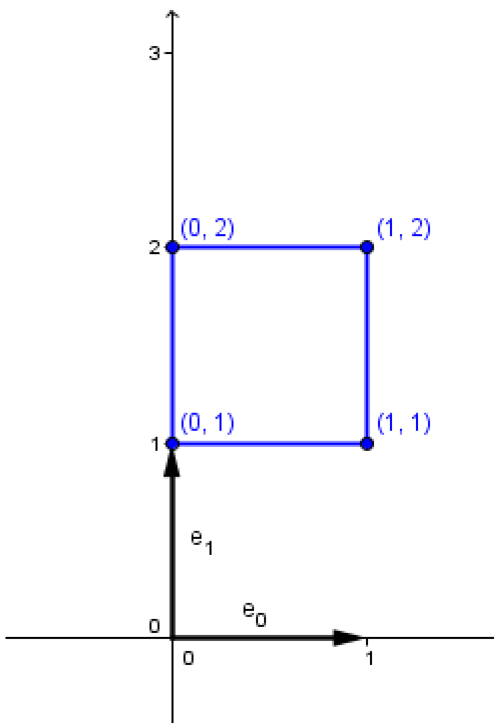
Shearing

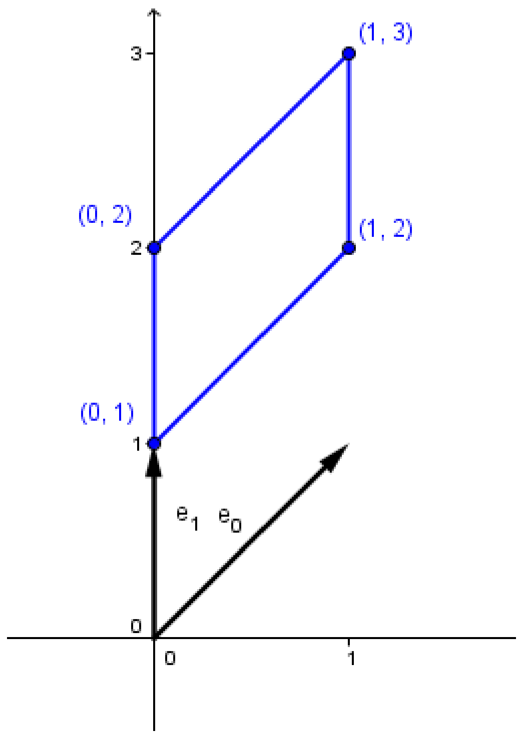
- Not much used by itself, but remember it for translations later.
- Tilts only one axis.
- Squares become parallelograms.

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

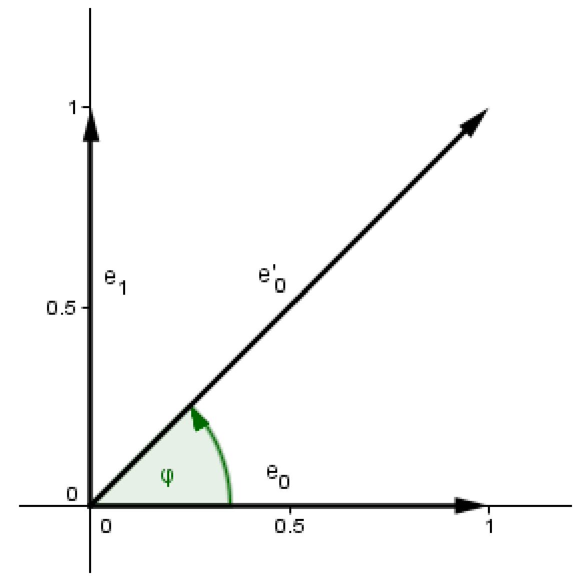




Shearing

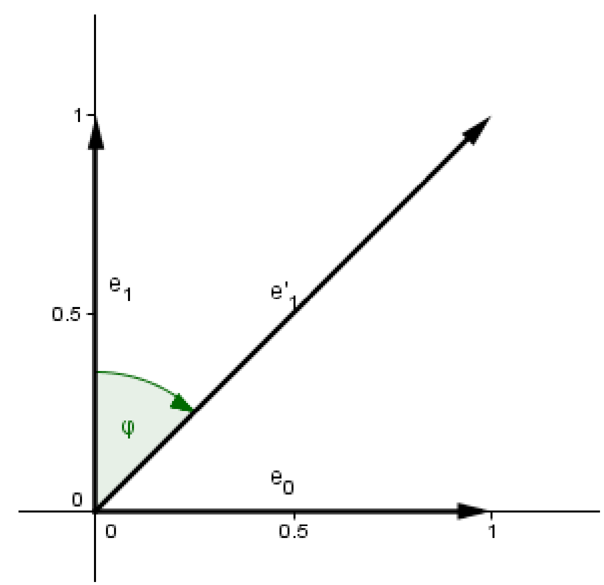
- **Shear-y**, we tilt parallel to y axis by angle φ counter-clockwise

$$\begin{pmatrix} 1 & 0 \\ \tan(\varphi) & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y + \tan(\varphi) \cdot x \end{pmatrix}$$



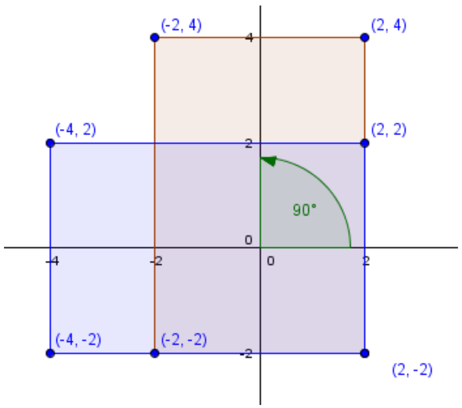
- **Shear-x**, we tilt parallel to x axis by angle φ clockwise

$$\begin{pmatrix} 1 & \tan(\varphi) \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + \tan(\varphi) \cdot y \\ y \end{pmatrix}$$



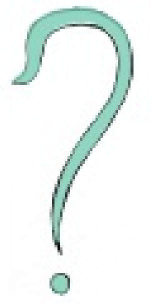
Linear Transformation

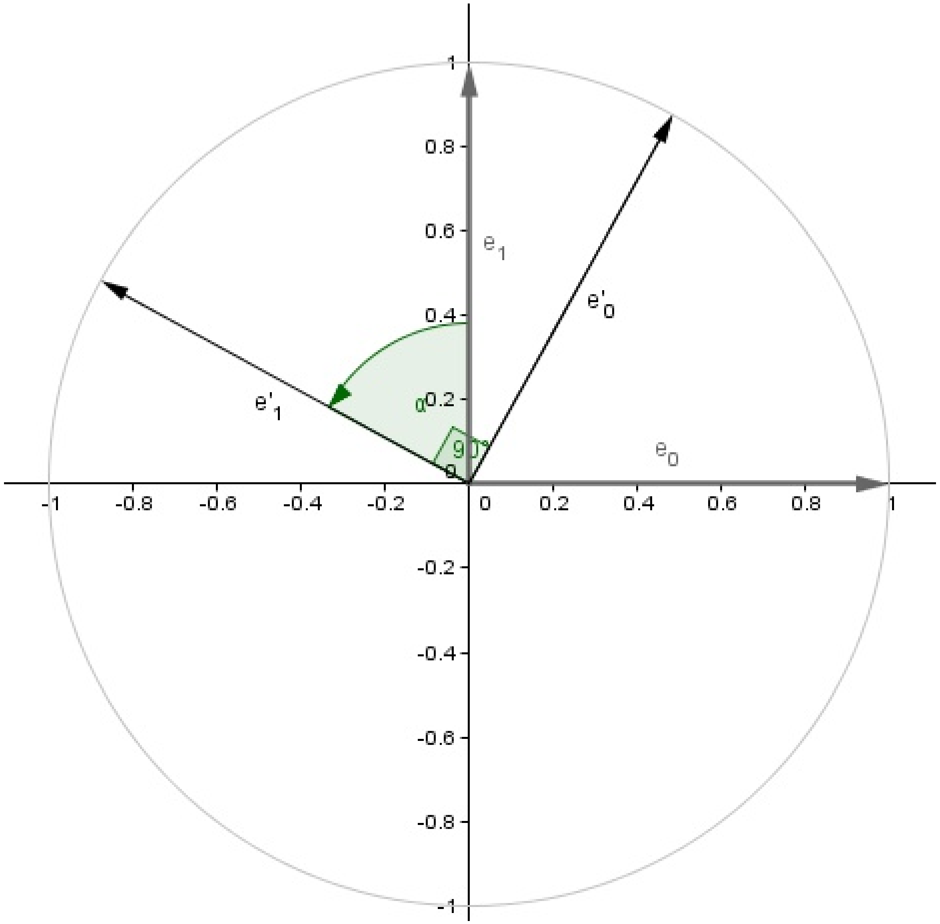
Rotation



Rotation

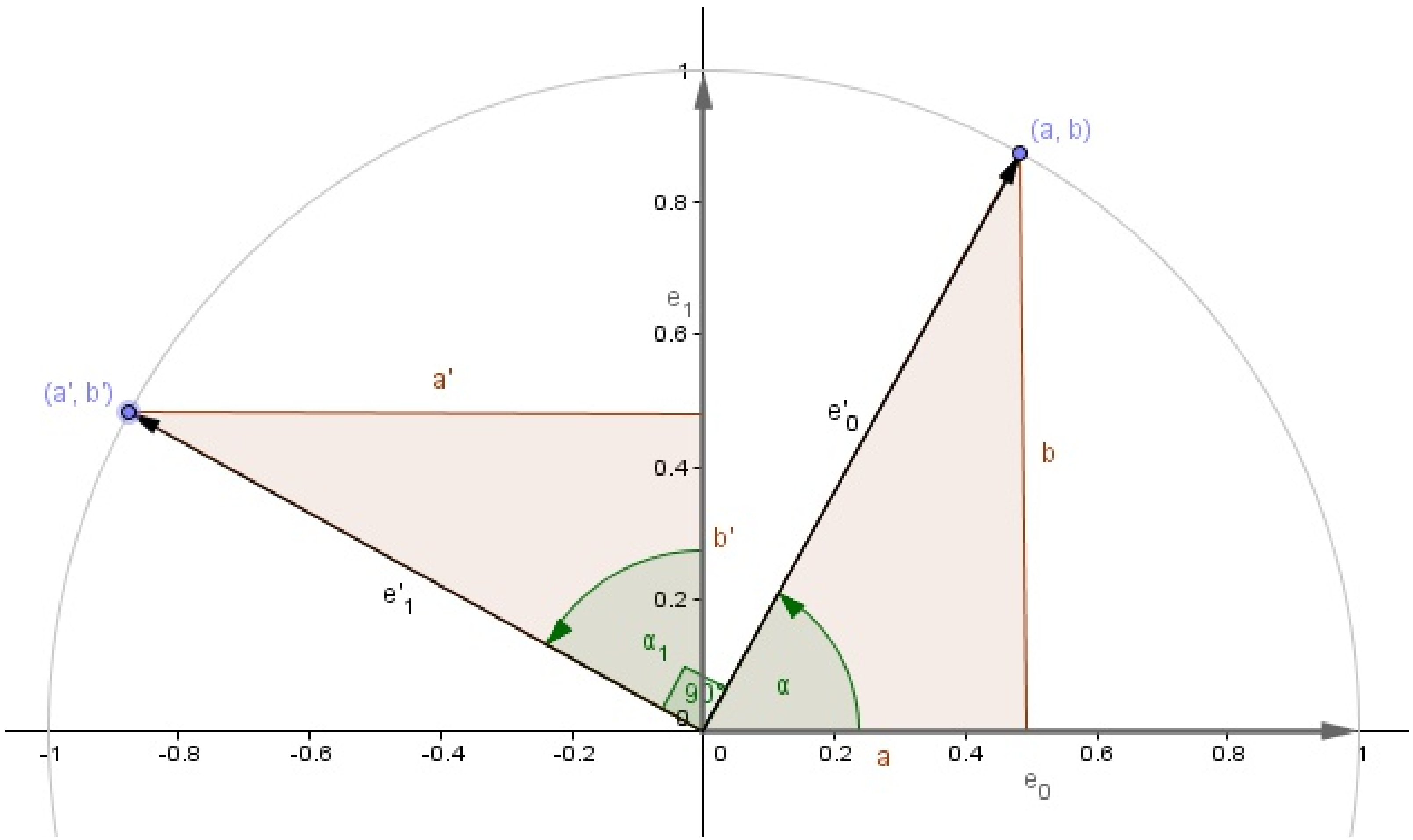
- Shearing moved only one axis
- Also changed the size of the basis vector
- Can we do better?





Did you notice that the columns of the transformation matrix show the coordinates of the new basis vectors?

Rotation



$$e'_0 = (|a|, |b|) = (\cos(\alpha), \sin(\alpha))$$
$$e'_1 = (|a'|, |b'|) = (-\sin(\alpha), \cos(\alpha))$$

$$\cos(\alpha) = \frac{|a|}{|e'_0|} = \frac{|a|}{1} = |a|$$

Rotation

- So if we rotate by α in counter-clockwise order in 2D, the transformation matrix is:

$$\begin{array}{c} \begin{array}{cc} e'_0 \qquad\qquad & \qquad\qquad e'_1 \end{array} \\ \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \end{array}$$

- In 3D we can do rotations in each plane (xy, xz, yz), so there can be 3 different matrices.

Rotation

- To do a rotation around an arbitrary axis, we can:
  - Rotate that axis to be the x-axis
  - Rotate around the new x-axis
  - Invert the first rotations (move the old x-axis back)

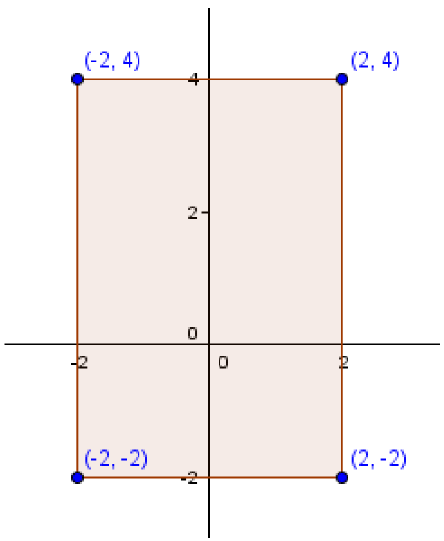
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

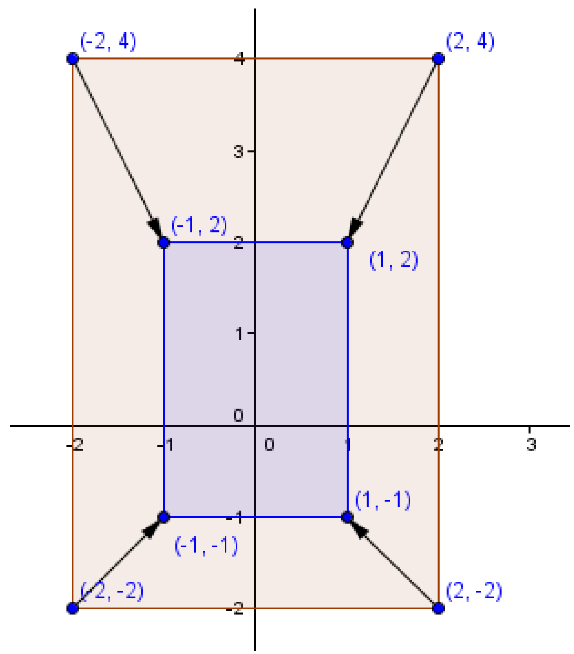
- OpenGL provides a command for rotating around a given axis.
- Often quaternions are used for rotations.

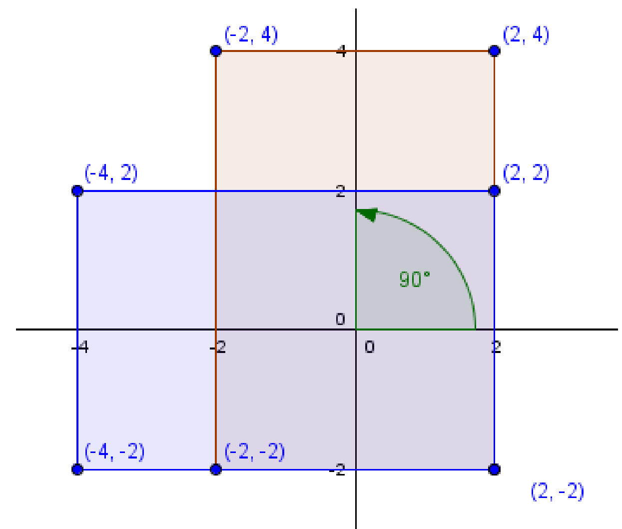
Quaternions are elements of a number system that extend the complex numbers...

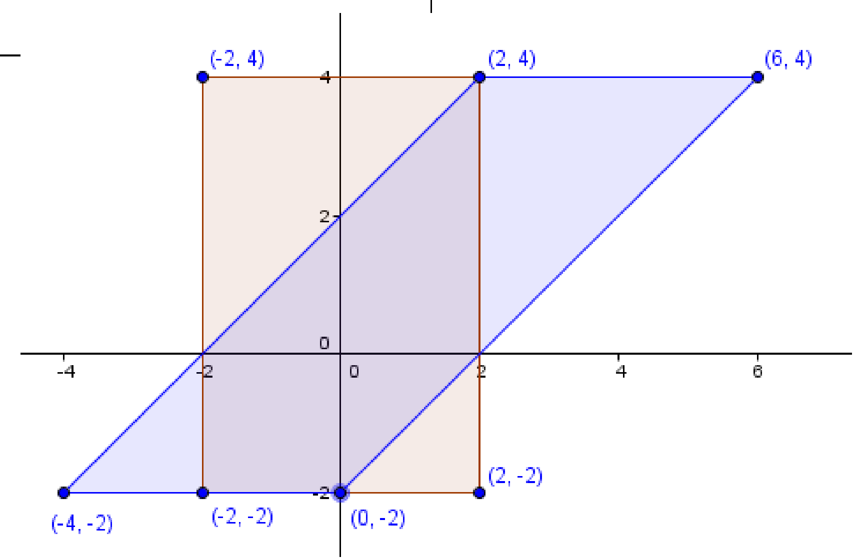
Do we have everything now?

- We can scale, share and rotate our geometry around the origin...





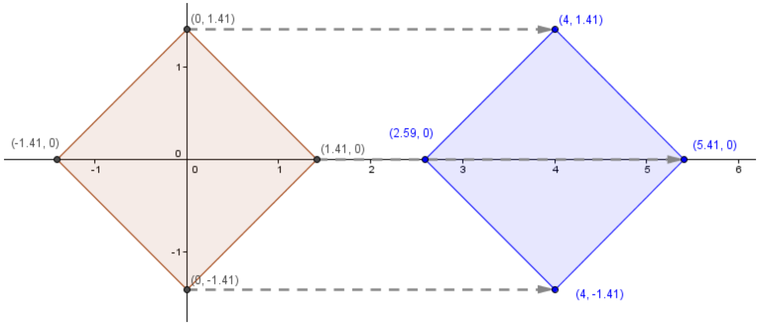




What if we have an object not centered in the origin?

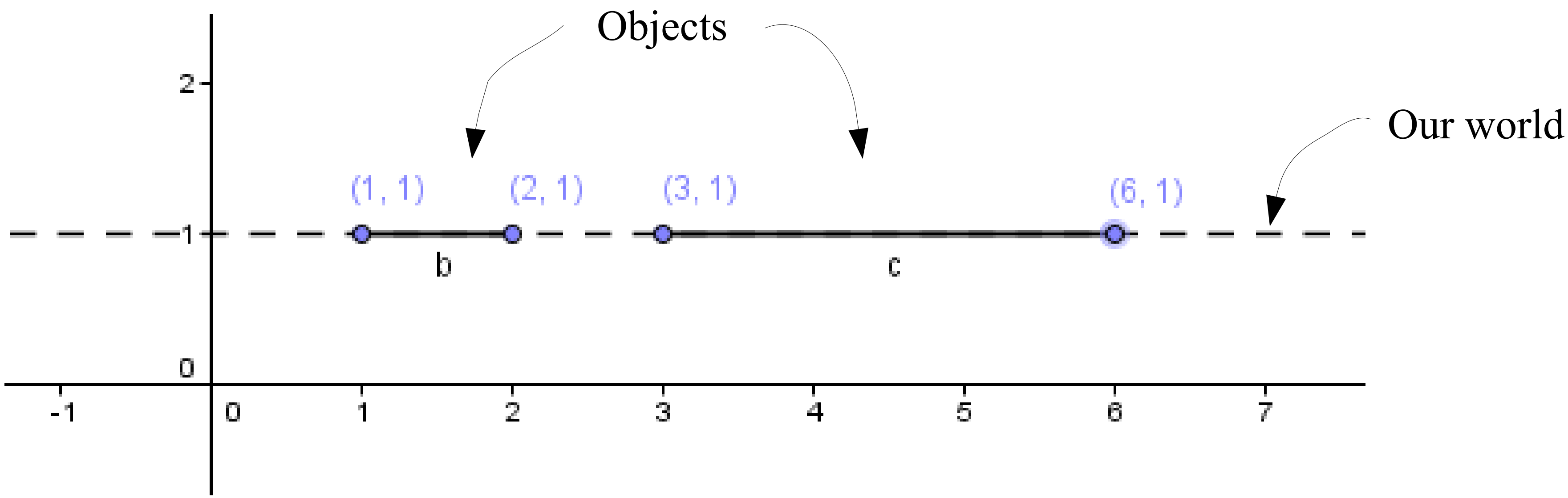
Affine Transformation

Translation



Translation

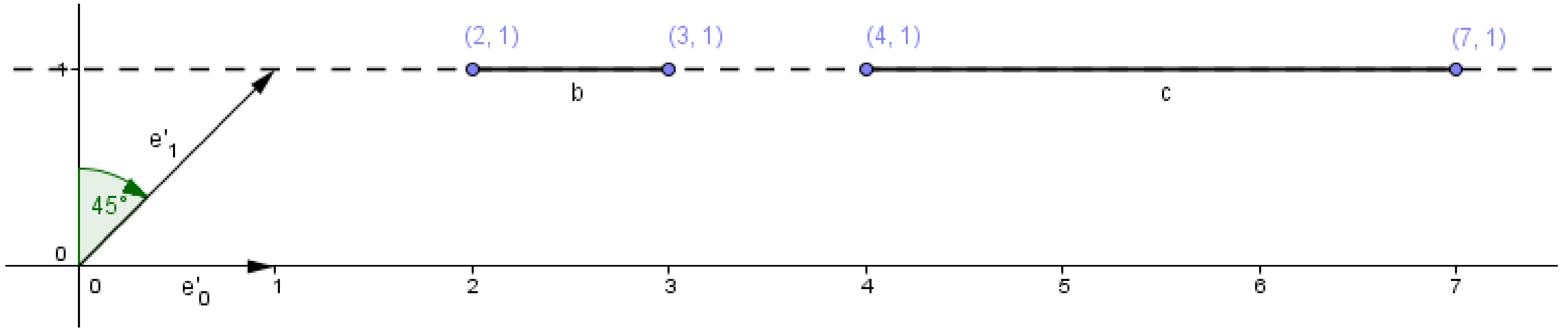
- Imagine that our 1D world is located at y=1 line in 2D space.



- Notice that all the points are in the form: (x, 1)

Translation

- What happens if we do shear-x(45°) operation on the 2D world?



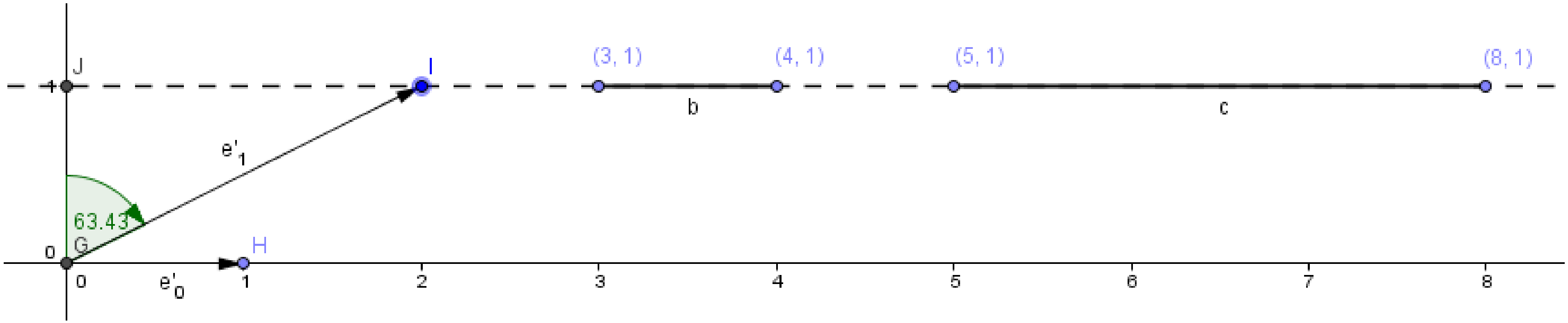
- Everything in our world has moved magically one x-coordinate to the right...

$\tan(45°) = 1$

Translation

- What if we do shear-x(63.4°)?

$\tan(63.4°) = 2$



- Everything has now moved 2 x-coordinates to the right from the original position
- We can do translation (movement)!

Translation

- When we represent our points in one dimension higher space, where the extra coordinate is 1, we get to the **homogeneous** space.

$$\begin{pmatrix} 1 & x_t \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_t \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \\ z + z_t \\ 1 \end{pmatrix}$$

Transformations

- This together gives us a very good toolset to transform our geometry as we wish.

Linear transformations

Translation column

Affine trasnformation

$$\begin{pmatrix} a & b & c & x_t \\ d & e & f & y_t \\ g & h & i & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + cz + x_t \\ dx + ey + fz + y_t \\ gx + hy + iz + z_t \\ 1 \end{pmatrix}$$

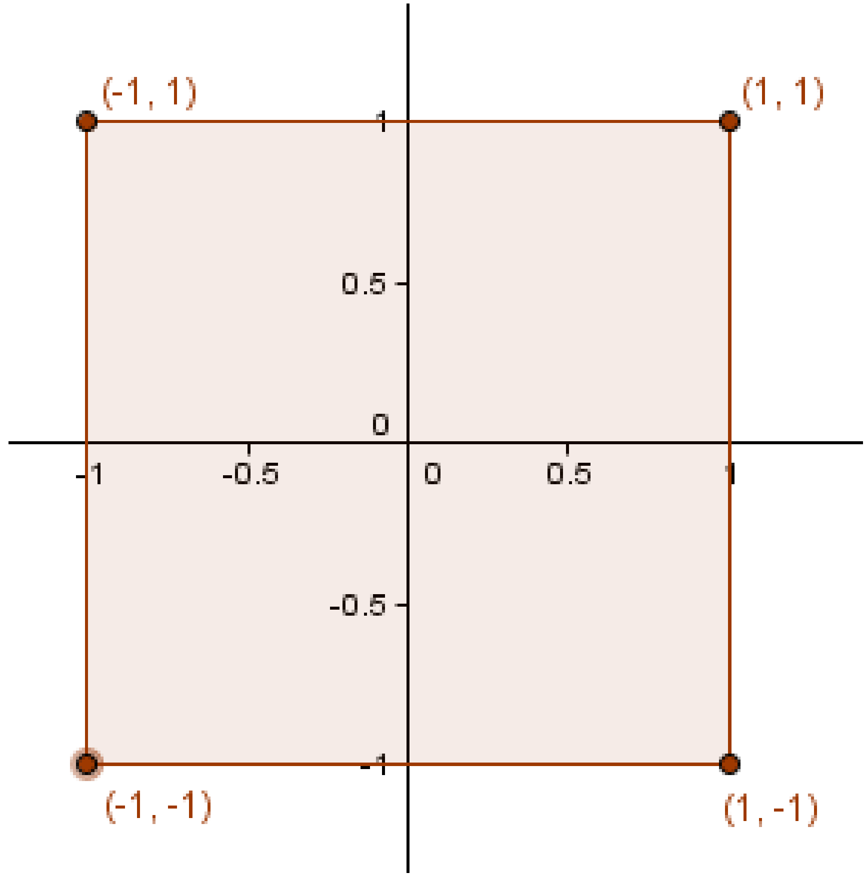
Used for perspective projection...

Multiple transformations

- Everything starts from the origin!
- To apply multiple transformations, just multiply matrices.

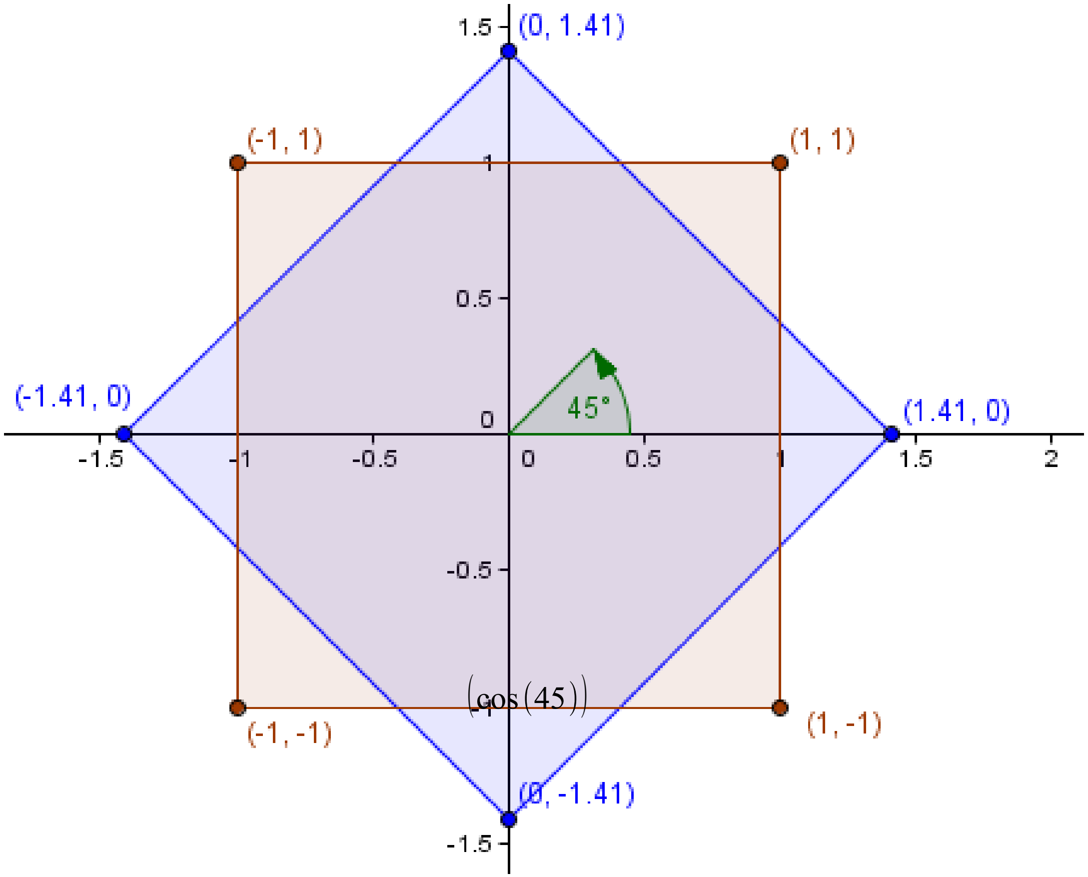


Multiple transformations



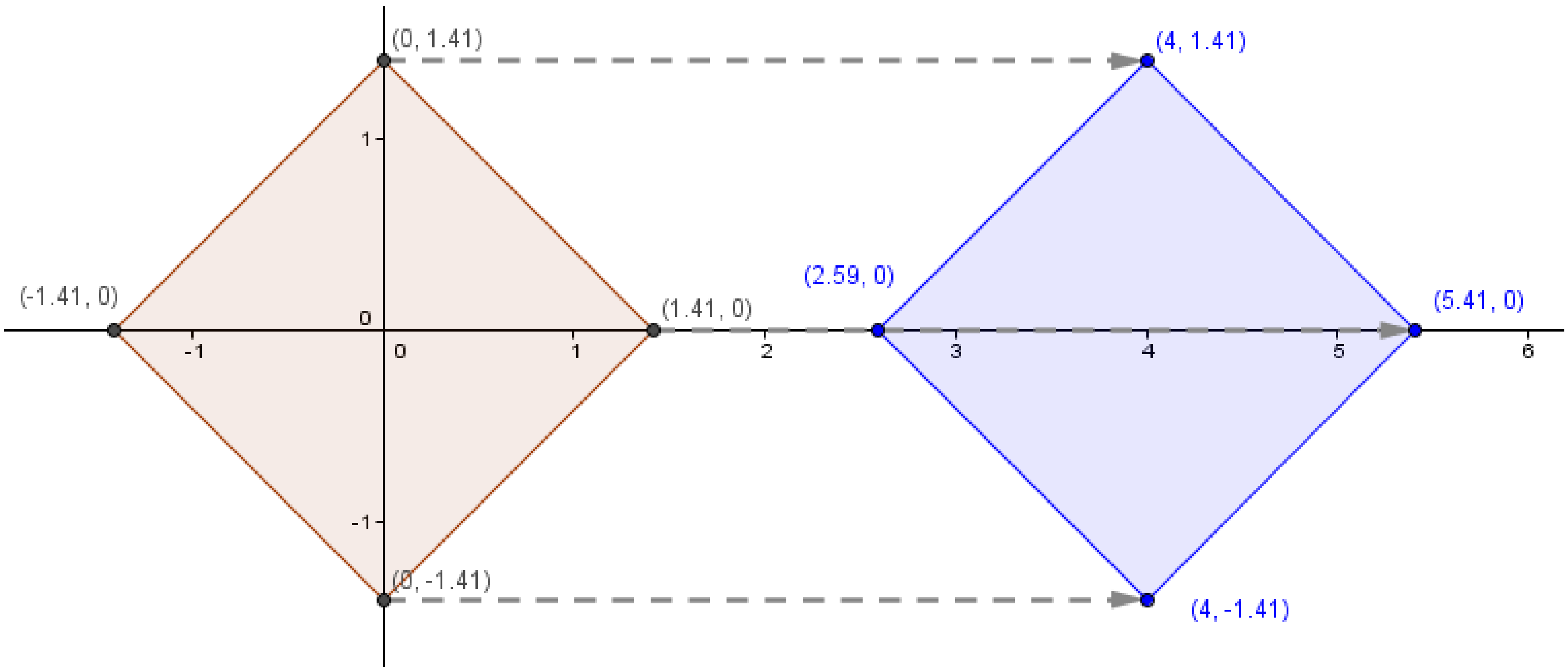
Our initial geometry defined by vertices: (-1, -1), (1, -1), (1, 1), (-1, 1)

Multiple transformations



$$\begin{pmatrix} \cos(45°) & -\sin(45°) & 0 \\ \sin(45°) & \cos(45°) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Multiple transformations



$$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Multiple transformations

- We can combine the transformations to a single matrix.

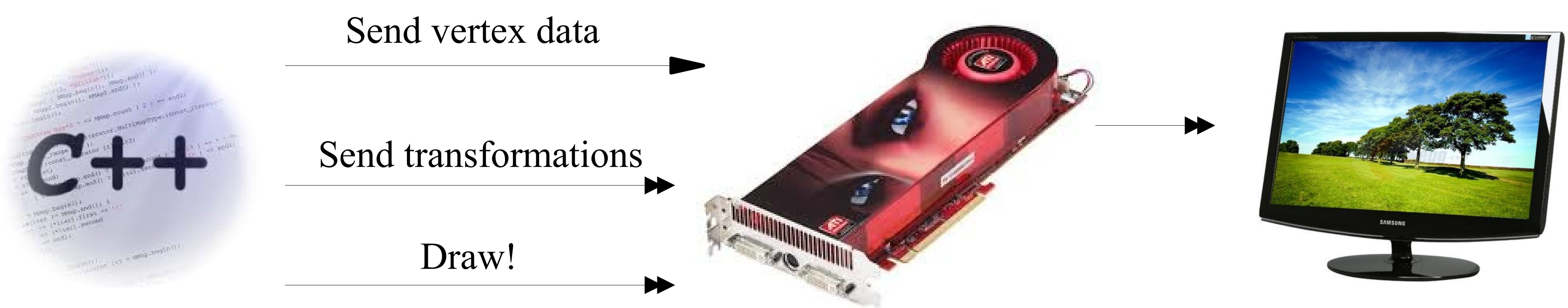
$$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(45°) & -\sin(45°) & 0 \\ \sin(45°) & \cos(45°) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(45°) & -\sin(45°) & 4 \\ \sin(45°) & \cos(45°) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- This works for combining different affine transformations, but the result is hard to read...
- Order of transformations / matrices is important!
- http://cgdemos.tume-maailm.pri.ee

This in Practice?

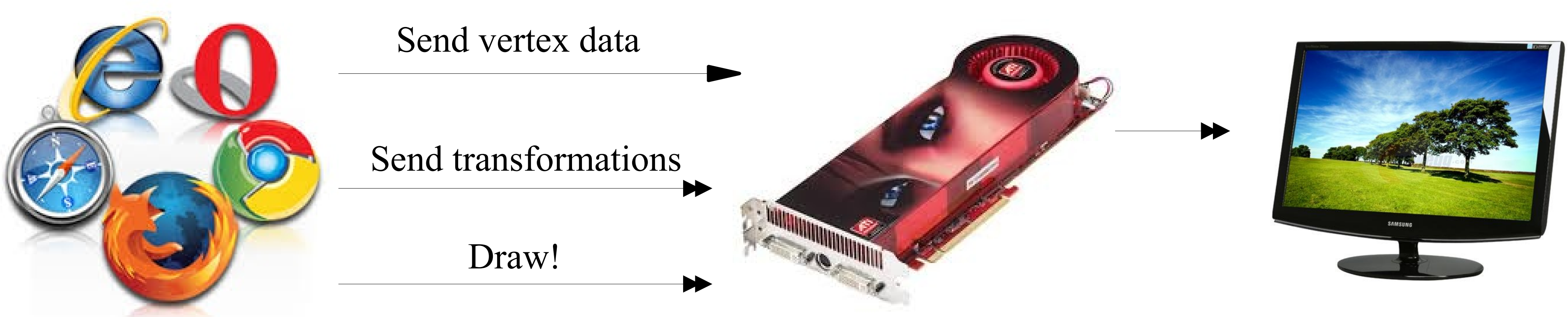
OpenGL

- GPU API / middleware
- Set of commands that a program can give to GPU
- Supported in many languages

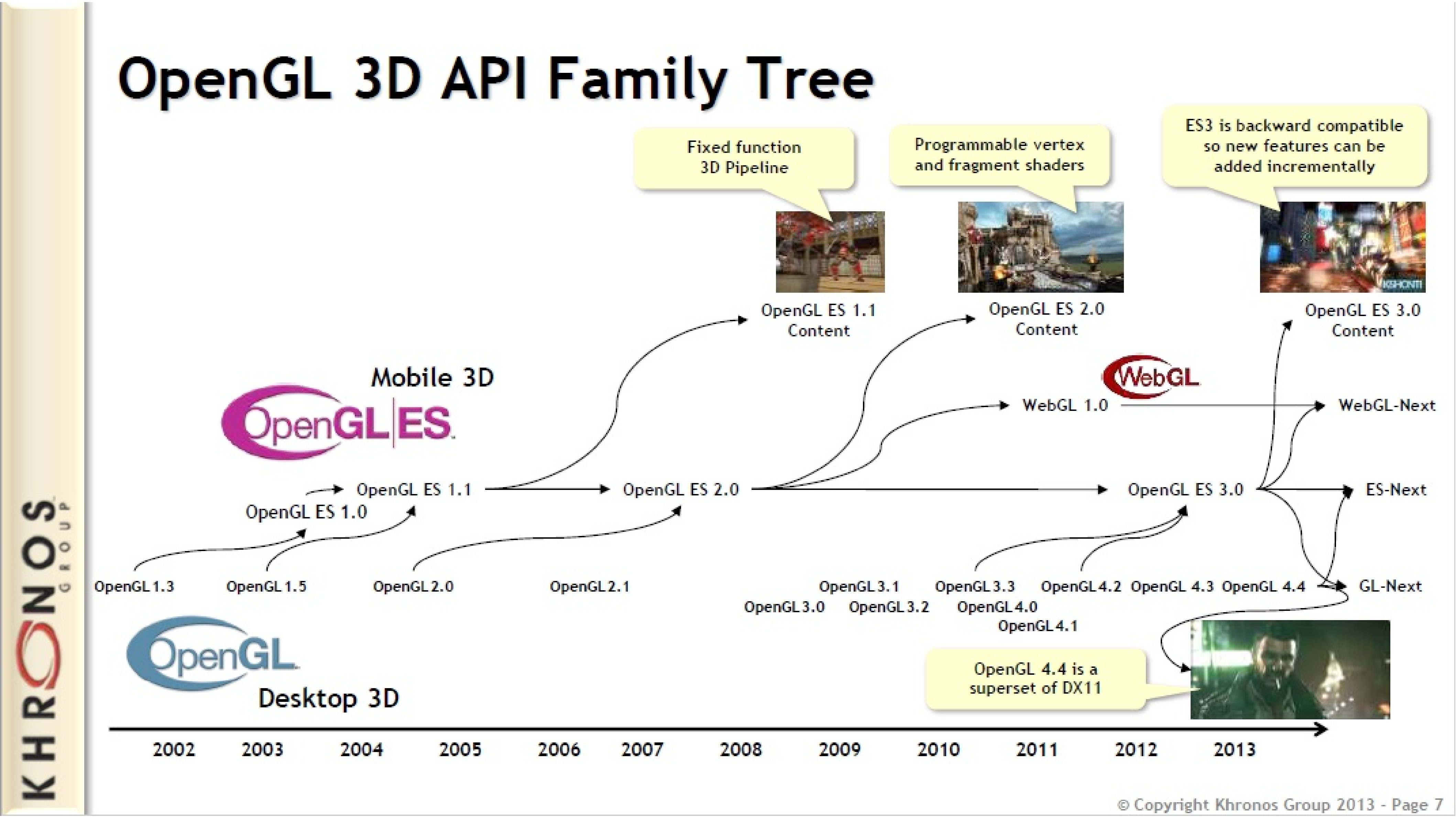


WebGL

- GPU API in JavaScript
- Supported by major browsers
- THREE.js – Higher level library to ease your coding: http://threejs.org/docs/



OpenGL and WebGL



https://www.slideshare.net/Khronos_Group/open-standardsforgaming-nov13

The Setup

Sending vertex data to the GPU

Vertex Array Object (VAO)

- An object to contain data arrays for vertices

```
GLuint vaoHandle;
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(vaoHandle);

// Generate and bind vertex buffers - VBO-s
// set data for position, color etc


glBindVertexArray(0);
```

Vertex Buffer Object (VBO)

- Buffer for hold one vertex data array

```
GLuint vaoHandle;
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(vaoHandle);

GLuint vboHandle;
glGenBuffers(1, &vboHandle);
glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
glBufferData(GL_ARRAY_BUFFER,
        sizeof(GLfloat) * vertexCount, vertexDataArray, GL_STATIC_DRAW);

...
```

Vertex Attribute

- Variable in the shader, which points to the data

```
GLuint vboHandle;
glGenBuffers(1, &vboHandle);
glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
glBufferData(GL_ARRAY_BUFFER,
        sizeof(GLfloat) * vertexCount, vertexDataArray, GL_STATIC_DRAW);

GLuint loc = glGetAttribLocation(shaderProgram, name);
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, elementsPerVertex, GL_FLOAT, GL_FALSE, 0, 0);

...
```

Each vertex gets their own values for the same attribute variable.

VBO: Element Array Buffer

- Buffer for indices to map the vertices → faces

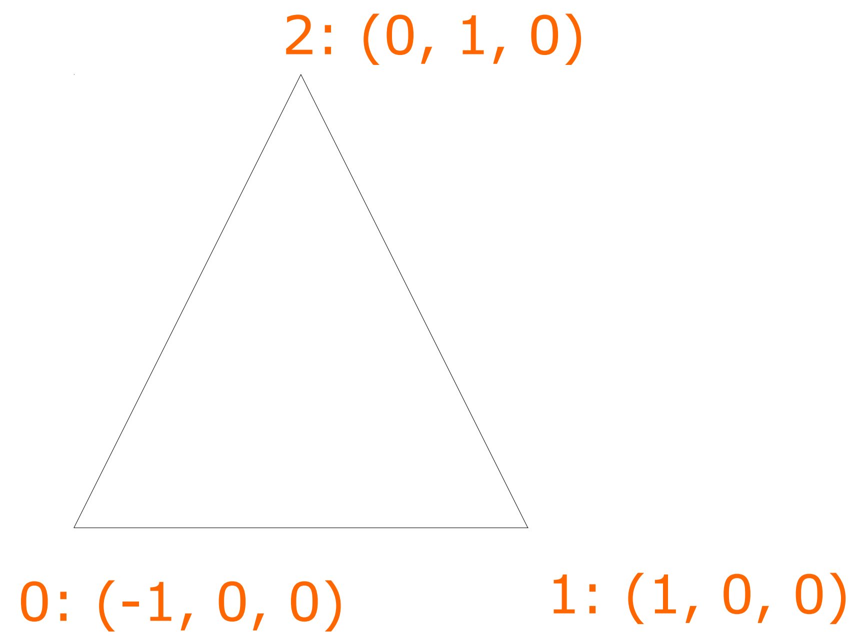
```
GLuint loc = glGetAttribLocation(shaderProgram, name);
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, 0);

glGenBuffers(1, &vboHandle);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboHandle);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(GLfloat)*indexCount, indices, GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Example

triangleVAO

positionVBO: [-1, 0, 0, 1, 0, 0, 0, 1, 0]

indicesVBO: [0, 1, 2]

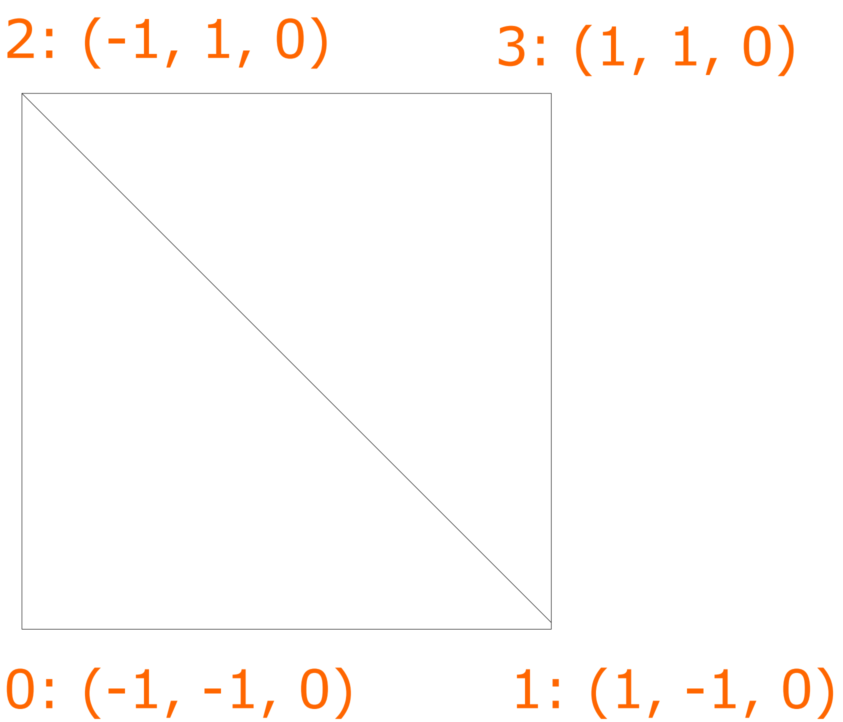


Example

squareVAO

positionVBO: [-1, -1, 0, 1, -1, 0, -1, 1, 0, 1, 1, 0]

indicesVBO: [0, 1, 2, 1, 3, 2]



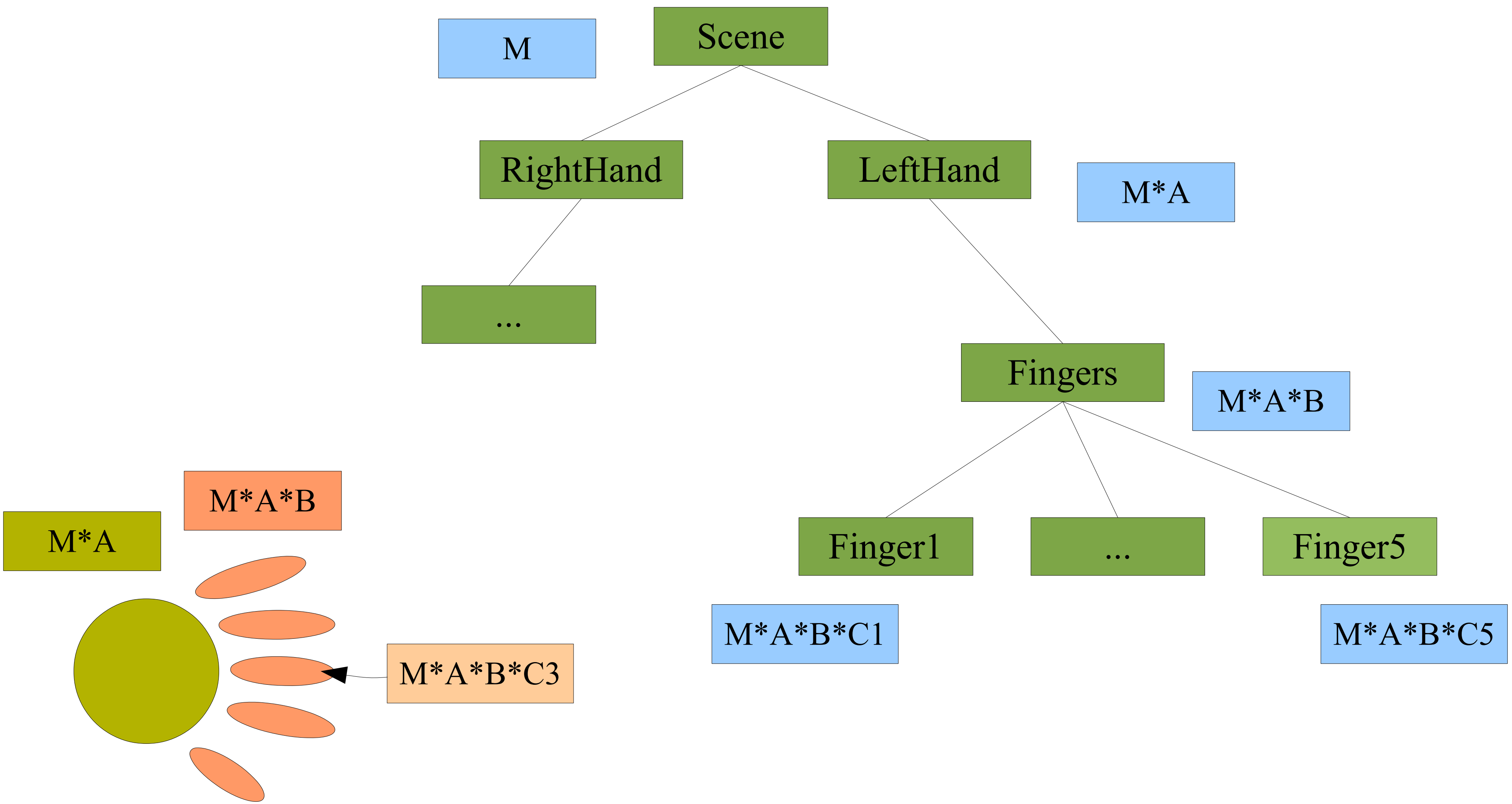
Drawing

- **Each frame** we tell the GPU to draw

```
std::stack<glm::mat4> ms;
ms.push(glm::mat4(1.0));

ms.push(ms.top());
    ms.top() = glm::rotate(ms.top(), ...);
    ms.top() = glm::translate(ms.top(), ...);
    GLint loc = glGetUniformLocation(prog, matrixName);
    glUniformMatrix4fv(loc, 1, GL_FALSE, glm::value_ptr(ms.top()));

    glBindVertexArray(vaoHandle);
    glDrawElements(GL_TRIANGLES, vertexCount, GL_UNSIGNED_BYTE, 0);
ms.pop();
```

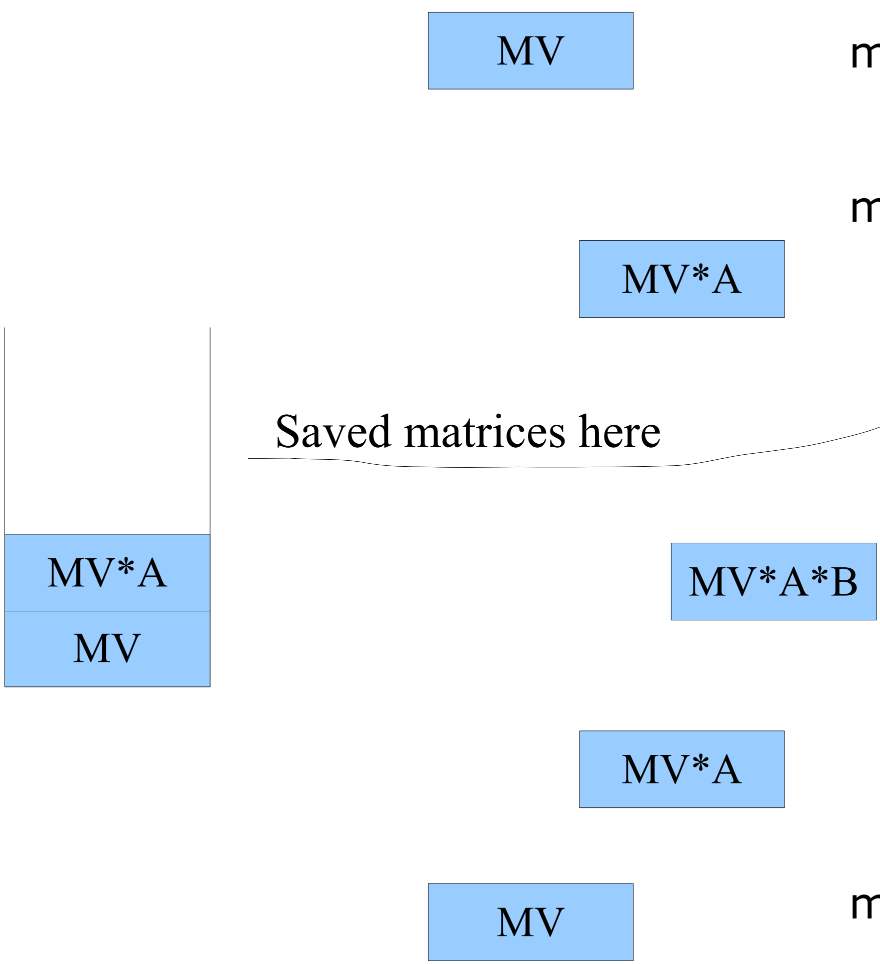
The Scene

- Useful to think of the scene as a tree



Use of the Matrix Stack

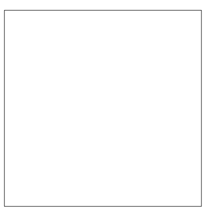
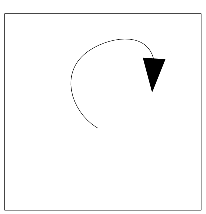
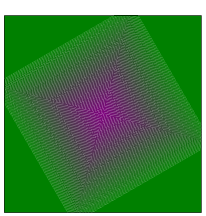
- More complex geometry for a single object




```
std::stack<glm::mat4> ms;
ms.push(glm::mat4(1.0));

ms.push(ms.top());
        DrawPalm...
        (transform top, send matrix, draw)

        ms.push(ms.top());
                DrawFingers...
                (transform top, send matrix, draw)

        ms.pop();
ms.pop();
```

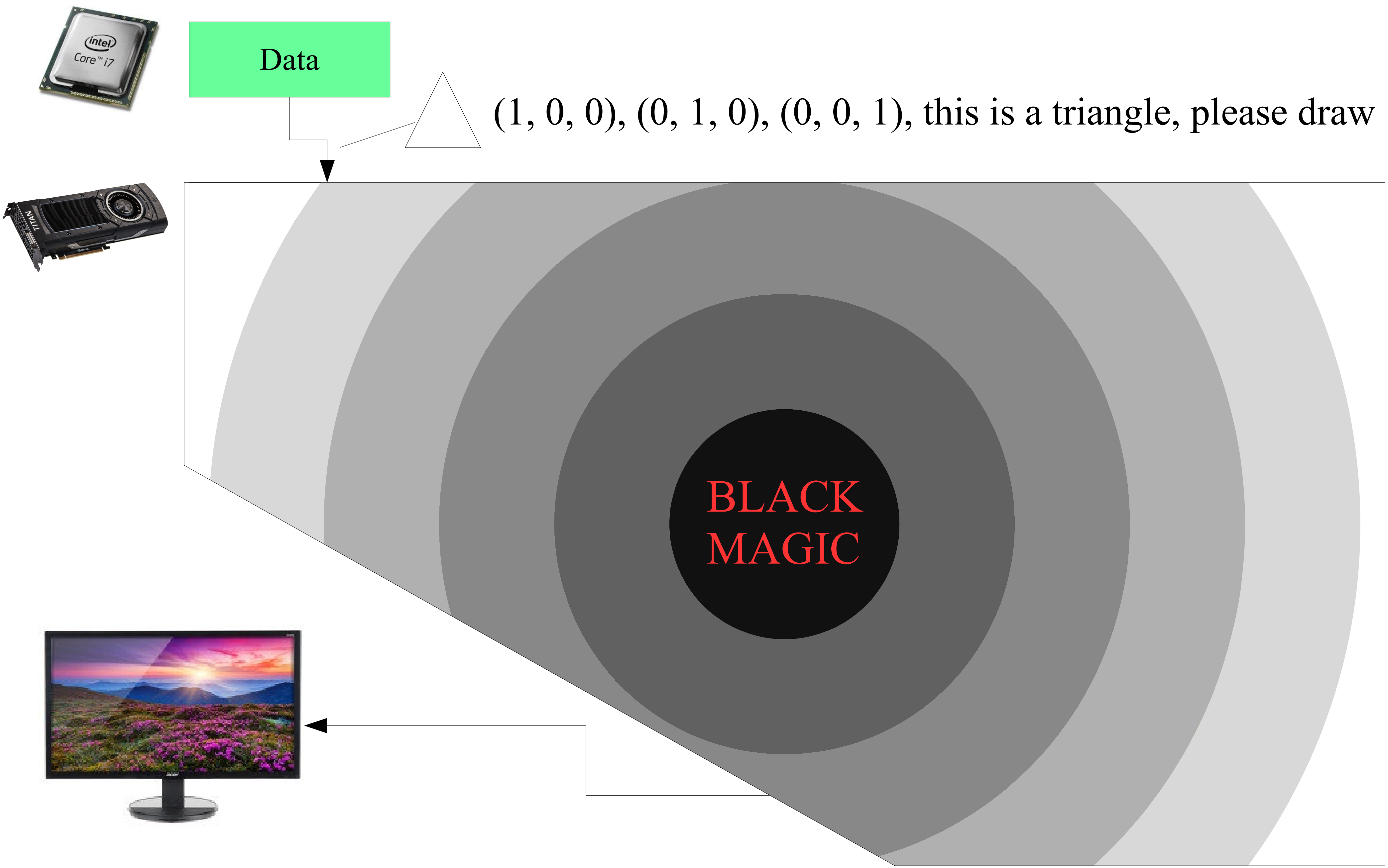
Old and new OpenGL?

- It used to be different before OpenGL 3.
- Everything old still works in compatibility mode.

	Prior to OpenGL 3	**OpenGL 3+**
	• glBegin(...) • glVertex(...) • glEnd(...)	• Vertex Array Object (VAO) • Vertex Buffer Object (VBO)
	• glTranslate(...) • glRotate(...) • glScale(...)	• Use other Matrix library (eg GLM) • Send your matrices to shaders
	• glMaterial(...)	• Vertex Buffer Object (VBO)

Good tutorials: http://antongerdelan.net/opengl/

Now You Know



Next time...

- The **graphics pipeline** in more detail
- How to define **color** for our geometry?
- Vertex and fragment **shaders**