Computer Graphics
– Transformations

J.-Prof. Dr. habil. Kai Lawonn

Introduction

- We know how to create objects, color them and/or give them a detailed appearance using textures
- But they are static objects
- Could change their vertices and re-configuring their buffers each frame to move them → cumbersome and costs quite some processing power
- Better: transform an object by using (multiple) matrix objects

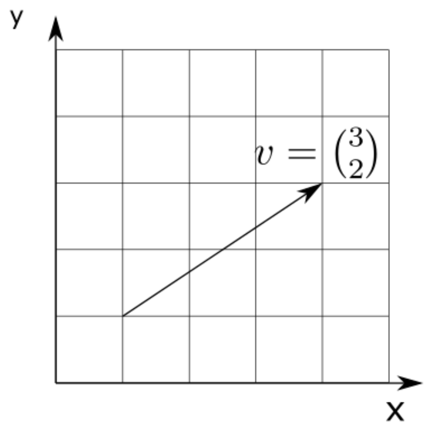
Introduction

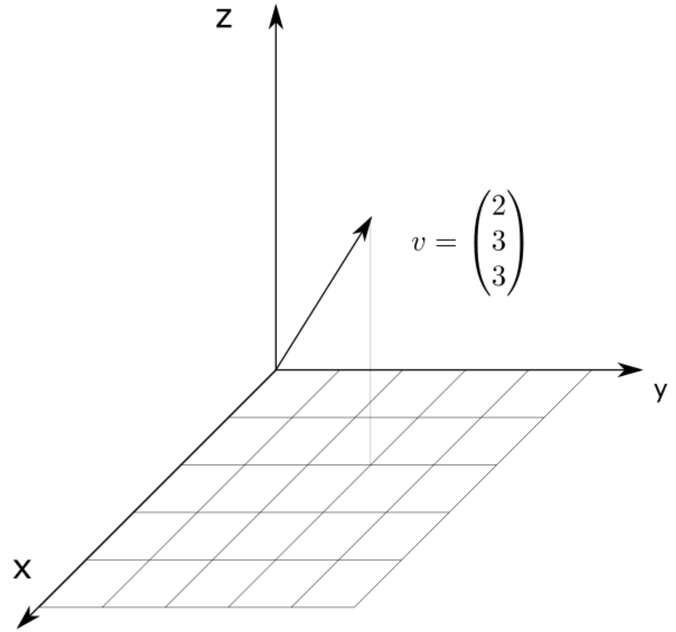
- Matrices are very powerful mathematical constructs that are very important in computer graphics
- This lecture is about a small introduction to vectors and matrices

Vectors

Introduction

- A vector has a direction and a magnitude (tuple of numbers)
- Vectors can have any dimension, but we usually work with dimensions of 2 to 4





Operations

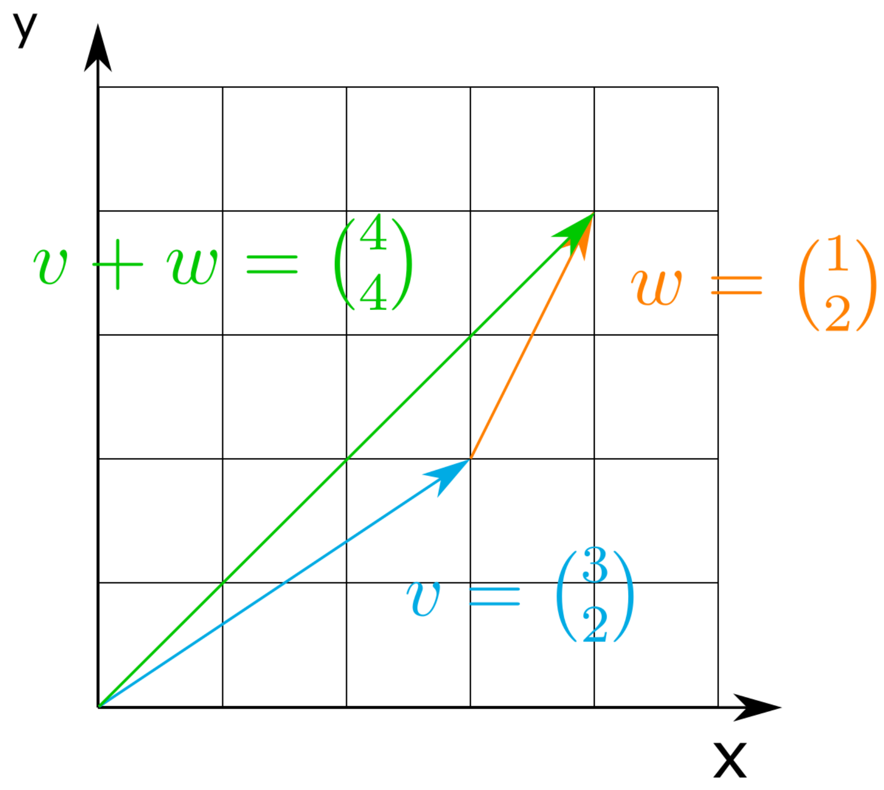
- In GLSL vectors can be defined by vec2, vec3, and vec4
- GLSL offers simple scalar operations:

```
vec2 p = vec2(2,3);
p = p + 2; //p = (4,5)
p-= 3; //p = (1,2)
vec2 q = -p; //q = (-1,-2)
```

Operations

- Addition of two vectors is defined as a component-wise addition:

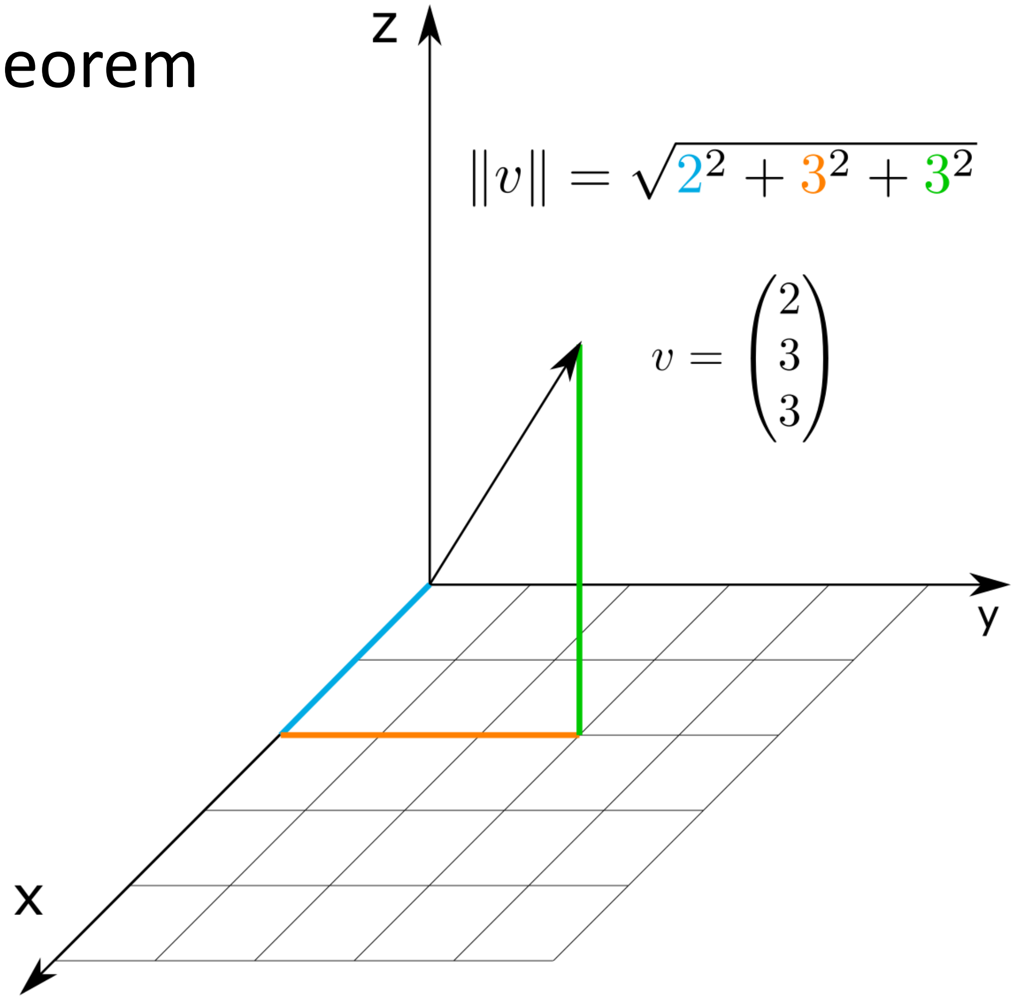
```
vec3 p = vec3(1,2,3);
vec3 q = vec3(4,5,6);
vec3 res = p + q; //res = (5,7,9)
```



Operations

- Length of a vector → Pythagoras theorem

```
vec3 v = vec3(2,3,3);
float L = length(v); //L = sqrt(22)
```



Operations

- Normalizing of a vector → divide by length

```
vec3 v = vec3(2,3,3);
vec3 n = normalize(v); //n = v/|v|
```

$$n = \frac{v}{\|v\|}$$

Operations

- Dot product: component-wise multiplication and addition afterwards
- In this case three dimensional vectors

```
vec3 v = vec3(1,2,3);
vec3 w = vec3(4,5,6);
float d = dot(v,w); //d=4+10+18=32
```

$$v \cdot w = \langle v, w \rangle = v_1 w_1 + v_2 w_2 + v_3 w_3$$

Operations

- Dot product with the vector itself is the length squared

$$v \cdot v = v_1^2 + v_2^2 + v_3^2 = \|v\|^2$$

Operations

- Dot product: also the lengths multiplied with cos of the angle between both vector

```
vec3 v = vec3(1,2,3);
vec3 w = vec3(4,5,6);
float d = dot(v,w); //d=4+10+18=32
float len_v = length(v);
float len_w = length(w);
float a = acos(d/(len_v*len_w));
```

$$v \cdot w = \|v\| \cdot \|w\| \cdot \cos(\alpha)$$

$$\alpha = \operatorname{acos} \frac{v \cdot w}{\|v\| \cdot \|w\|}$$

Operations

Multiplying two vectors results in a component-wise multiplication:

```
vec3 v = vec3(1,2,3);
vec3 w = vec3(4,5,6);
vec3 u = v*w; // = (4,10,18)
```

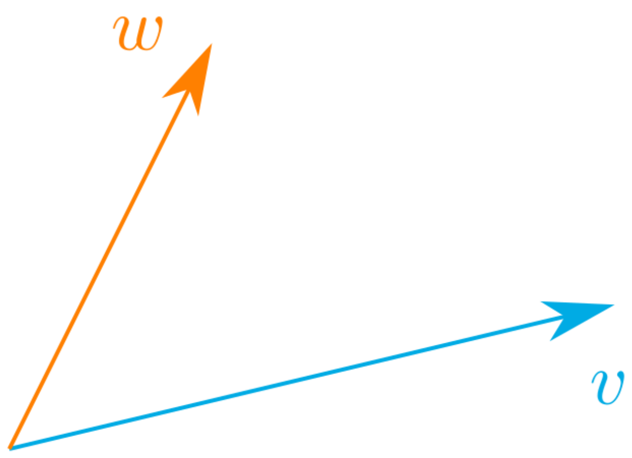
Operations

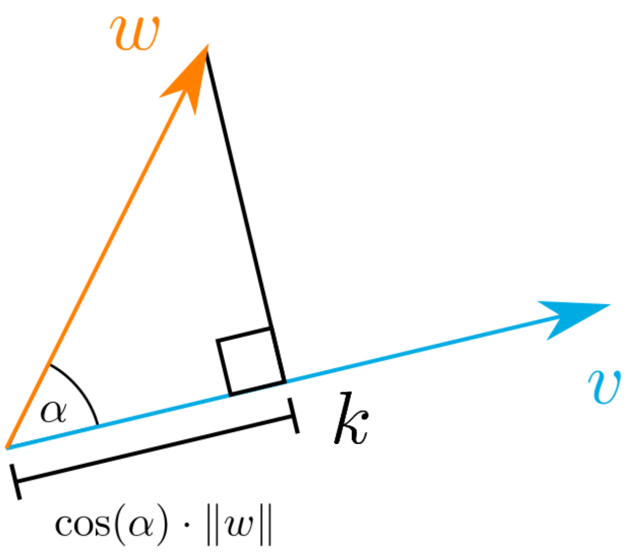
- Useful conclusion:

$$v \cdot w = \|v\| \cdot \|w\| \cdot \cos(\alpha)$$

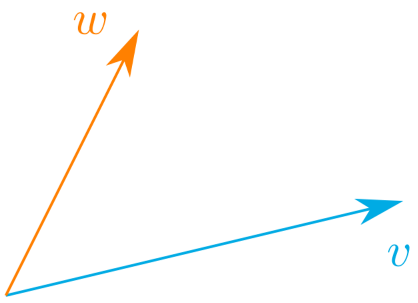
$$v \cdot w = \|v\| \cdot k$$

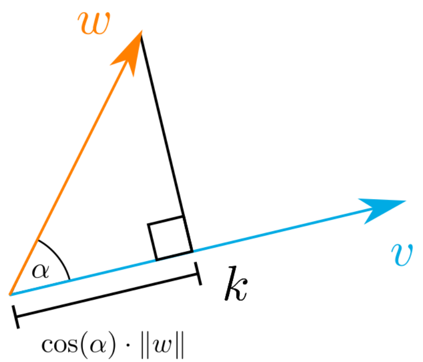
$$k = \frac{v \cdot w}{\|v\|}$$





Operations





- Relation:

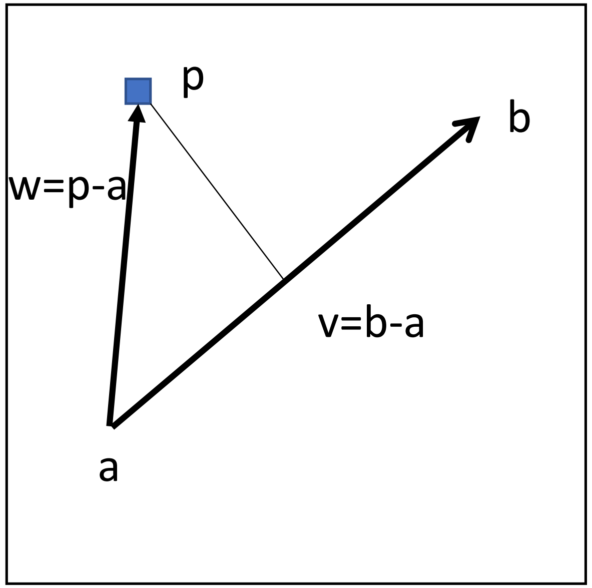
$$\frac{k}{\|v\|} = \frac{v \cdot w}{\|v\|^2}$$
$$= \frac{v \cdot w}{v \cdot v}$$

Operations

- Why is this useful?
- Remember when we determined the orthogonal projection:

$$\frac{k}{\|v\|} = \frac{v \cdot w}{v \cdot v}$$

$$\begin{aligned} 0 &= \langle p - x, v \rangle \\ &= \langle p - (a + \lambda v), v \rangle \\ &= \langle p - a, v \rangle - \langle \lambda v, v \rangle \\ \lambda \langle v, v \rangle &= \langle p - a, v \rangle \\ \lambda &= \frac{\langle v, w \rangle}{\langle v, v \rangle} \end{aligned}$$

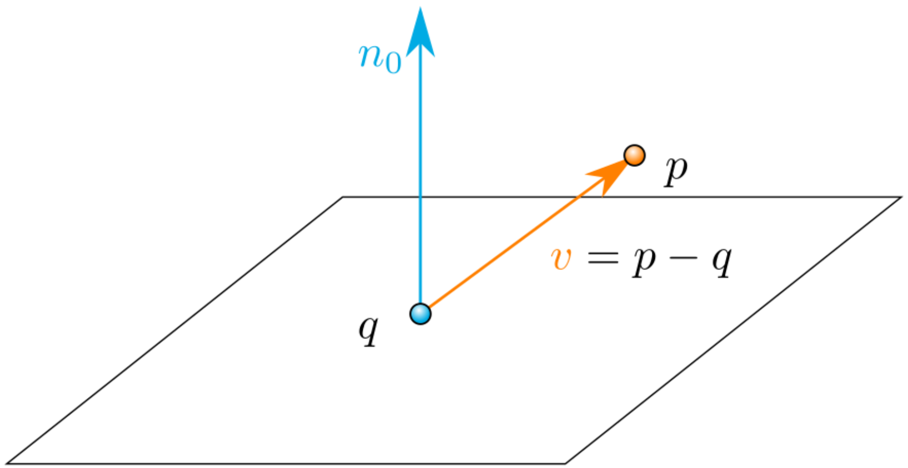


Operations

- Calculate the orthogonal projection of a point onto the plane

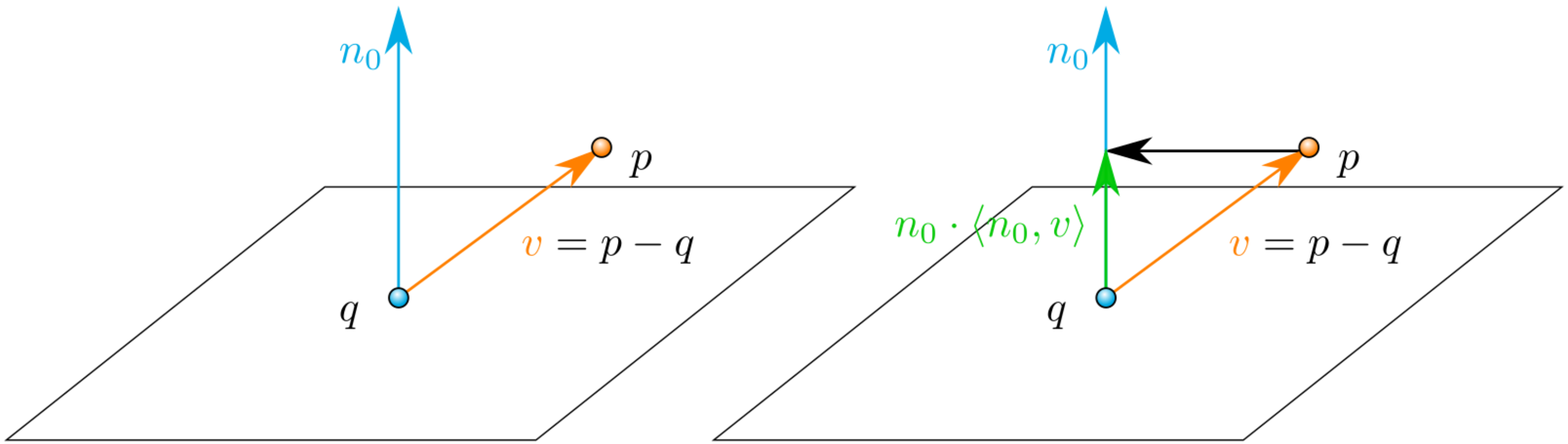
Operations

- Calculate the orthogonal projection of a point onto the plane
- Given is a point q on the plane and a normalized normal n_0
- Task: project p orthogonally on the plane



Operations

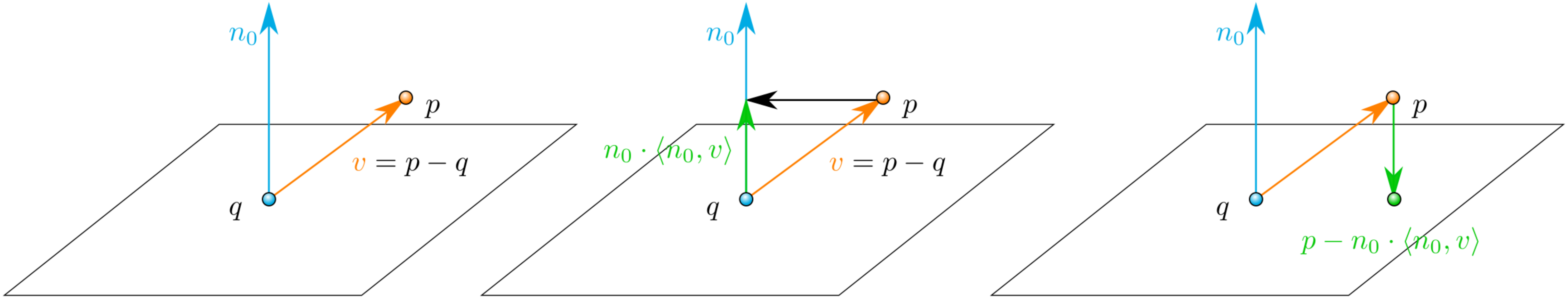
- Calculate the orthogonal projection of a point onto the plane
- Given is a point q on the plane and a normalized normal n_0
- Task: project p orthogonally on the plane



Operations

- Calculate the orthogonal projection of a point onto the plane
- Given is a point q on the plane and a normalized normal n_0
- Task: project p orthogonally on the plane

$$\boxed{p - n_0 \cdot \langle n_0, v \rangle}$$



Operations

- Example:

$$q = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \ n_0 = \begin{pmatrix} 1/3 \\ 2/3 \\ 2/3 \end{pmatrix}$$

$$p = \begin{pmatrix} 3 \\ 3 \\ 6 \end{pmatrix}$$

$$\boxed{p - n_0 \cdot \langle n_0, v \rangle}$$

Operations

- Example:

$$q = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \; n_0 = \begin{pmatrix} 1/3 \\ 2/3 \\ 2/3 \end{pmatrix}$$

$$p = \begin{pmatrix} 4 \\ 5 \\ 9 \end{pmatrix}$$

$$\begin{aligned} OP_p &= p - n_0 \cdot \langle n_0, p - q \rangle \\ &= p - n_0 \cdot (1 + 2 + 4) \\ &= \begin{pmatrix} 4 \\ 5 \\ 9 \end{pmatrix} - \begin{pmatrix} 7/3 \\ 14/3 \\ 14/3 \end{pmatrix} \\ &= \frac{1}{3} \begin{pmatrix} 5 \\ 1 \\ 13 \end{pmatrix} \end{aligned}$$

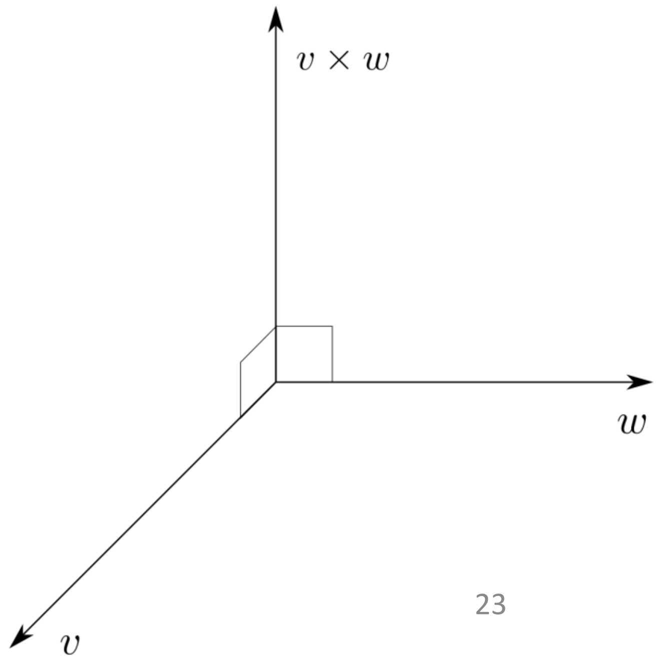
Operations

- Cross product of two vectors results in an orthogonal vectors

$$v \times w = \begin{pmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{pmatrix}$$

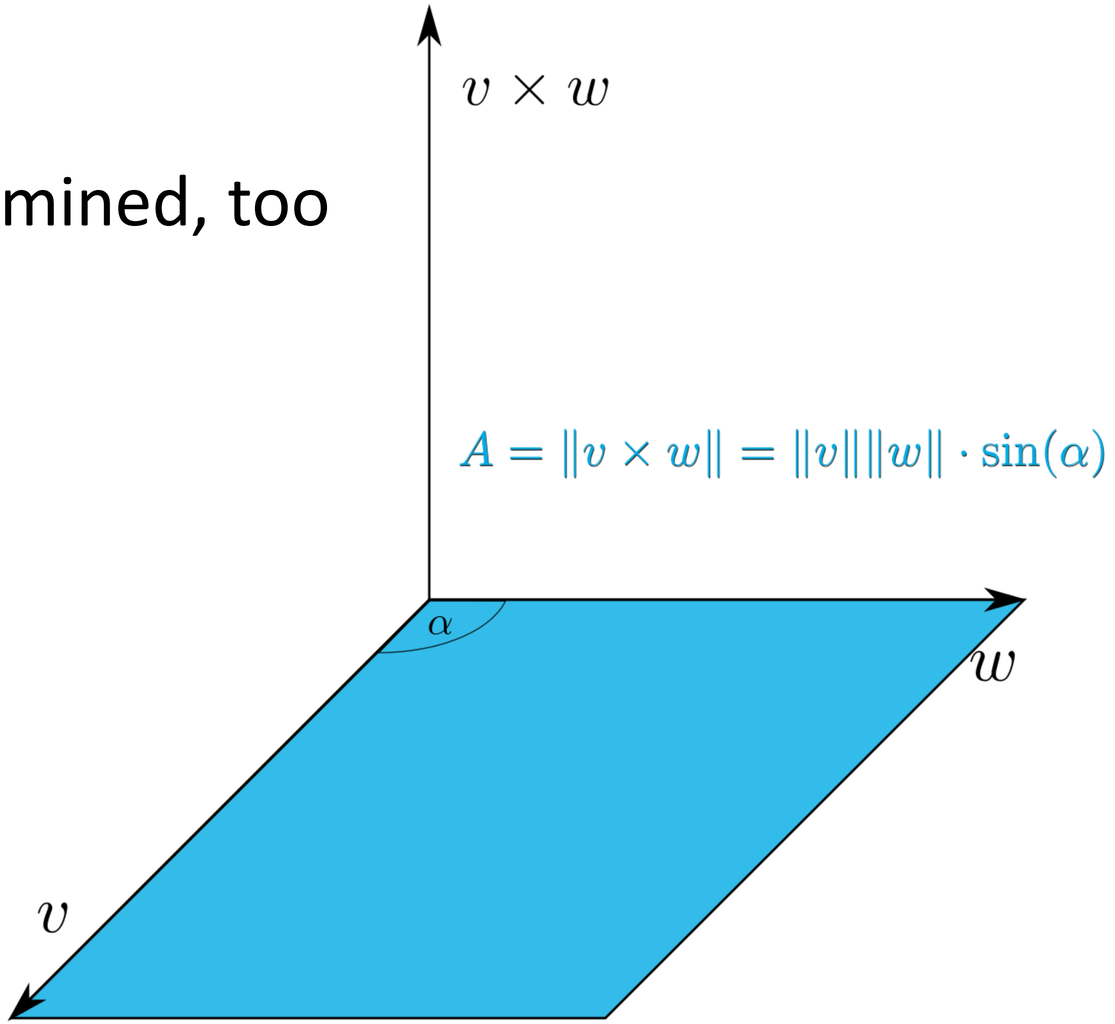
```
vec3 v = vec3(1,2,3);
vec3 w = vec3(4,5,6);

vec3 u = cross(v,w); //u = (-3,6,-3)
```



Operations

- The enclosing area can be determined, too



Matrices

Introduction

- A matrix is basically a rectangular array of numbers, symbols and/or expressions
- Each individual item in a matrix is called an element of the matrix
- Example of a 2x3 matrix:

```
mat3x2 M = mat3x2(1,4,2,5,3,6); //column wise
//matnxm n,m ∈ {2,3,4}, m ≠ n
//matn, n ∈ {2,3,4}
```

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Introduction

- Indexing
- Constructions

```
vec3 col0 = vec3(1, 2, 3);
vec3 col1 = vec3(4, 5, 6);
vec3 col2 = vec3(7, 8, 9);
mat3 M = mat3(col0, col1, col2); //set columns
float M20 = M[2][0]; // = 7 (col2[0])
float M11 = M[1].y;  // = 5 (col1.y)
```

$$M = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

Indexing

Matrices are indexed by:

- **(Math)**

  (i,j) where i is the row and j is the column

  i x j matrix (row,column)

- **(GLSL)**

  [i][j] where i is the column and j is the row

  i x j matrix (column,row)

Operations

- GLSL allows scalar operations:

```
mat2 M = mat2(1,2,3,4);

mat2 N = M+2;

mat2 O = M*2;
```

$$M = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

$$N = \begin{pmatrix} 3 & 5 \\ 4 & 6 \end{pmatrix}$$

$$O = \begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix}$$

Operations

- Of course standard operations:

```
mat2 M = mat2(1,2,3,4);

mat2 N = mat2(1,3,4,6);

mat2 O = M+N;
```

$$M = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

$$N = \begin{pmatrix} 1 & 4 \\ 3 & 6 \end{pmatrix}$$

$$O = \begin{pmatrix} 2 & 7 \\ 5 & 10 \end{pmatrix}$$

Operations

- Matrix-matrix multiplication

```
mat2 M = mat2(1,2,3,4);


mat2 N = mat2(1,3,4,6);


mat2 O = M*N;
```

$$M = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

$$N = \begin{pmatrix} 1 & 4 \\ 3 & 6 \end{pmatrix}$$

$$O = \begin{pmatrix} \boxed{1 \cdot 1 + 3 \cdot 3} & 1 \cdot 4 + 3 \cdot 6 \\ 2 \cdot 1 + 4 \cdot 6 & \boxed{2 \cdot 4 + 4 \cdot 6} \end{pmatrix}$$

$$= \begin{pmatrix} 10 & 22 \\ 26 & 32 \end{pmatrix}$$

Operations

- Matrix-vector multiplication

```
mat2 M = mat2(1,2,3,4);

vec2 v = vec2(1,3);

vec2 o = M*v;
```

$$M = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

$$v = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

$$o = \begin{pmatrix} 1 \cdot 1 + 3 \cdot 3 \\ 2 \cdot 1 + 4 \cdot 6 \end{pmatrix}$$

$$= \begin{pmatrix} 10 \\ 26 \end{pmatrix}$$

Operations

- Identity matrix

```
mat3 Id = mat3(1);

vec2 v = vec2(1,2,3);

vec2 o = Id*v;
```

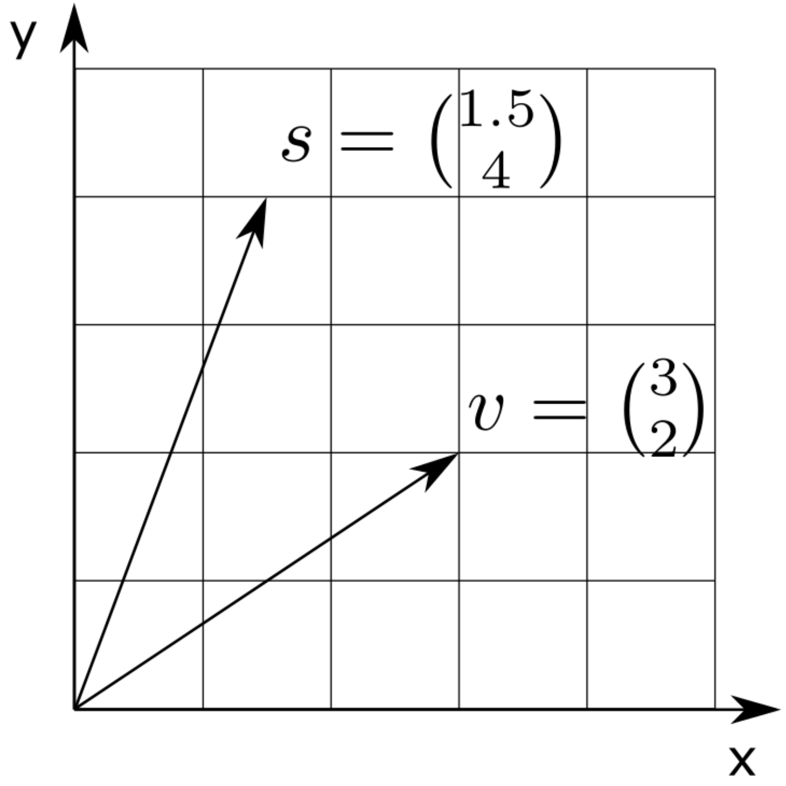
$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$v = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

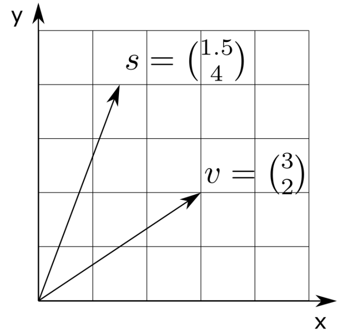
$$o = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Operations

- Scaling the vector $v = (3,2)$ along the x-axis by 0.5 and along the y-axis by 2:



Operations



- The scaling operation is a non-uniform scale, because the scaling factor is not the same for each axis
- If the scalar would be equal on all axes it would be called a uniform scale

Operations

- Constructing a transformation matrix that does the scaling
- Identity matrix multiplied 1 with the corresponding vector element → change the 1s in the identity matrix to the scaling factor
- Represent the scaling variables as $S = (s_1, s_2, s_3)$, then define scaling matrix:

$$\begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_1 x \\ s_2 y \\ s_3 z \\ 1 \end{pmatrix}$$

- For now ignore the last component (1)

Operations

- Additionally, we want to translate the vector after scaling
- The translating variables are represented as $\mathrm{T} = (t_1, t_2, t_3)$, then define the matrix:

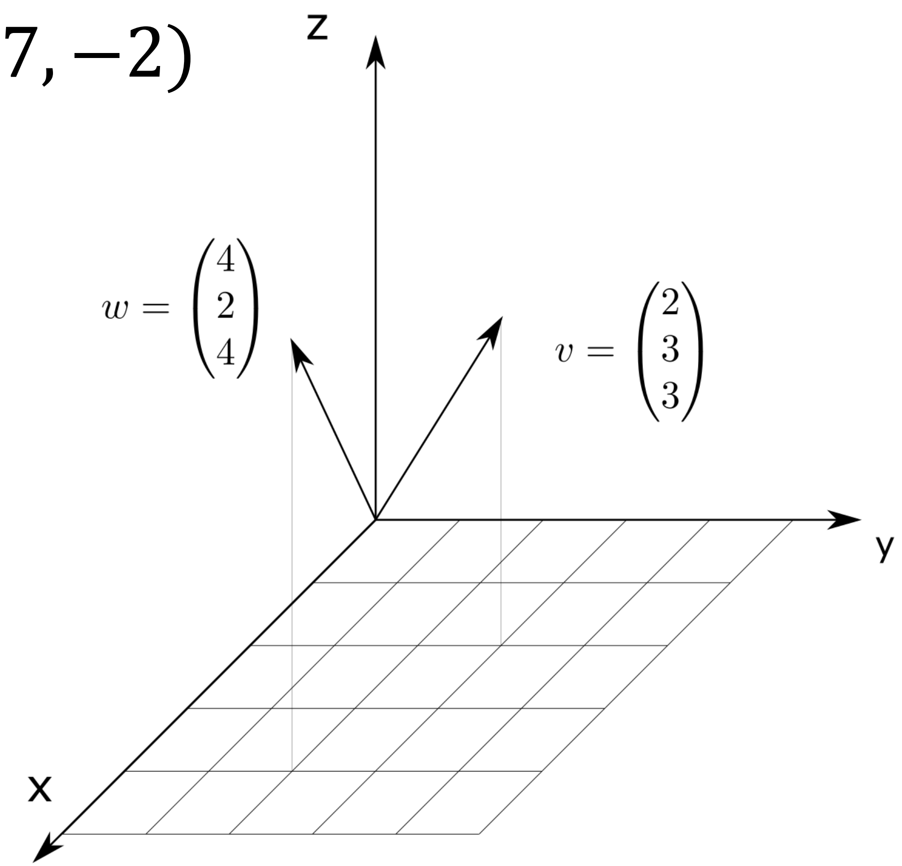
$$\begin{pmatrix} s_1 & 0 & 0 & t_1 \\ 0 & s_2 & 0 & t_2 \\ 0 & 0 & s_3 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_1 x + t_1 \\ s_2 y + t_2 \\ s_3 z + t_3 \\ 1 \end{pmatrix}$$

Operations

- Example: $v = (2,3,3), S = (2,3,2), T = (0,-7,-2)$

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & -7 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \cdot 2 + 0 \\ 3 \cdot 3 - 7 \\ 2 \cdot 3 - 2 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 4 \\ 2 \\ 4 \\ 1 \end{pmatrix}$$



Homogeneous Coordinates

- The w component of a vector is also known as a homogeneous coordinate
- 3D vector from a homogeneous vector → divide x, y, z by w
- (Did not notice this because w component was 1.0)
- Advantages: allows to do translations on 3D vectors (without a w or 0 can't translate)

Operations

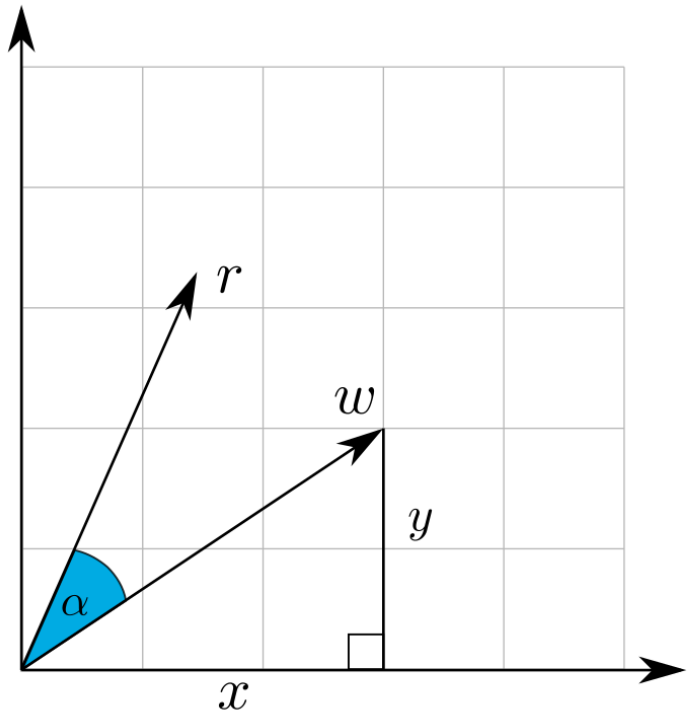
- Next step: rotations
- A rotation in 2D or 3D is represented with an angle
- Angles could be in degrees or radians (whole circle has 360° or 2π in radians)

$$angle\,in\,degrees = angle\,in\,radians \cdot \frac{180}{\pi}$$

$$angle\,in\,radians = angle\,in\,degrees \cdot \frac{\pi}{180}$$

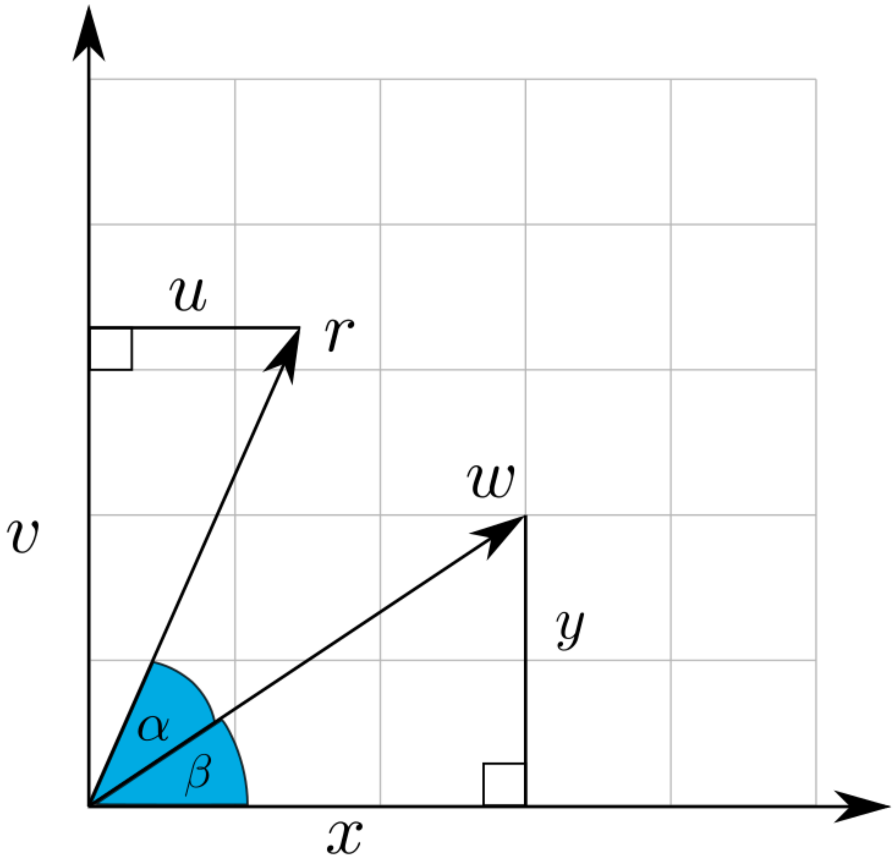
Rotations

- Rotation in 2D requires an angle and a direction (clock-wise (cw) / counter-clock-wise (ccw))
- Suppose we want to ccw rotate a vector $w = (x, y)$ around an angle α



Rotations

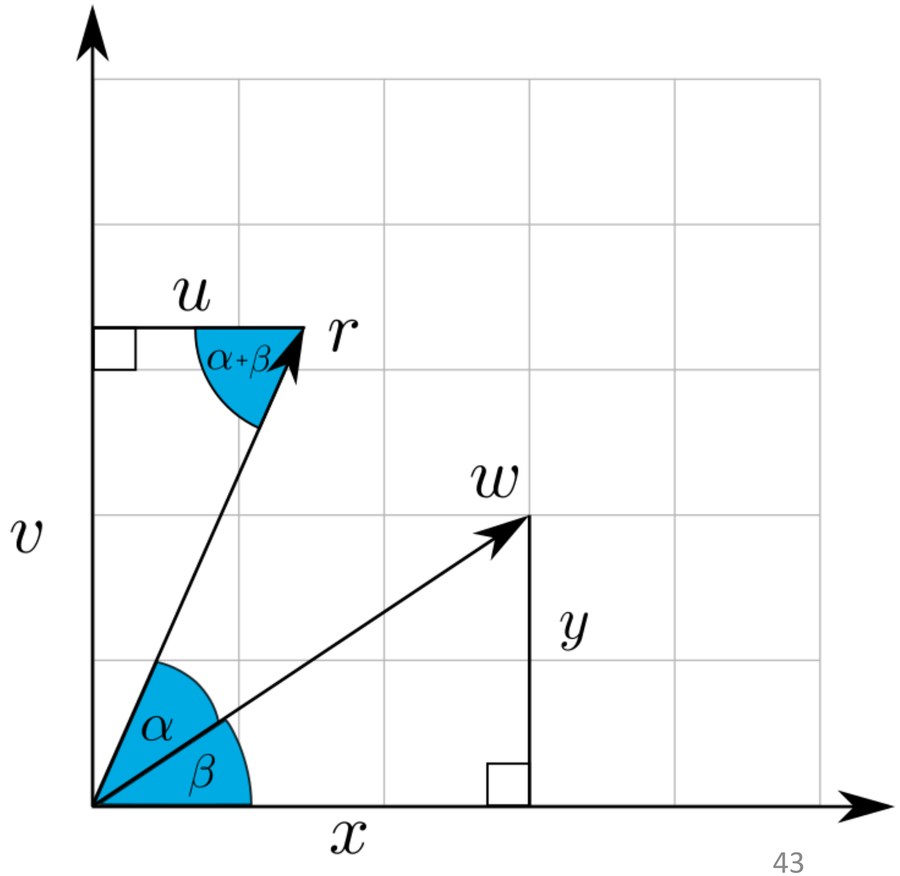
- First, compute v



Rotations

- First, compute v

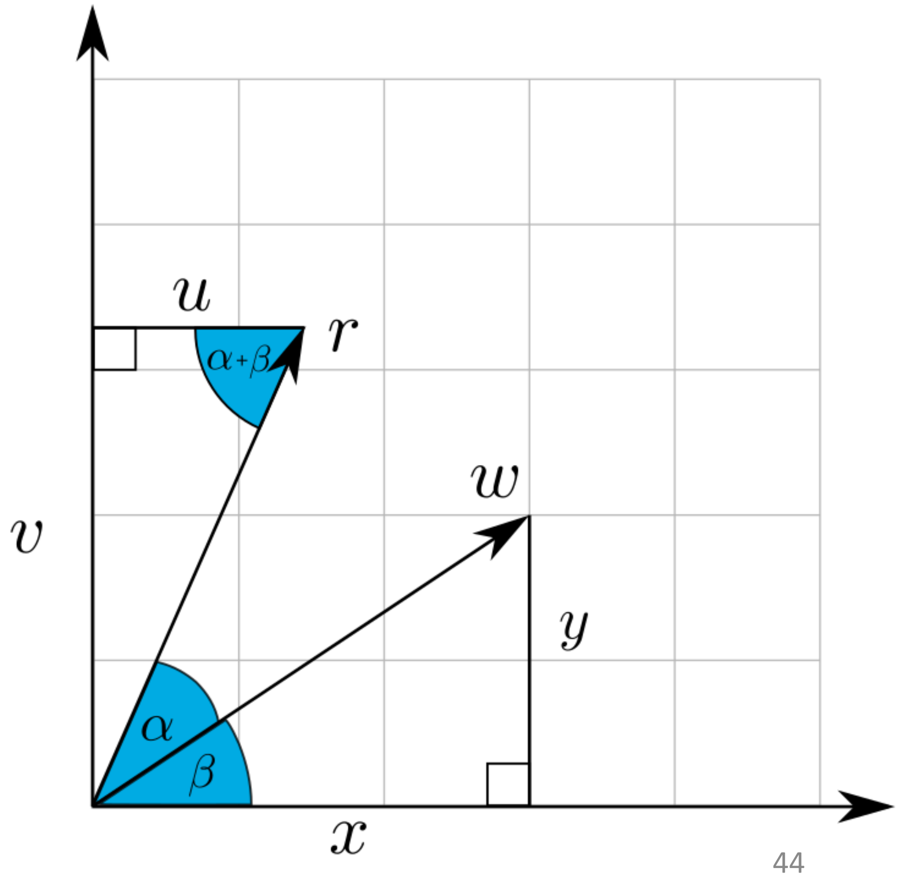
$$
\begin{aligned}
v &= \|w\| \cdot \sin(\alpha + \beta) \\
&= \|w\| \cdot (\sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)) \\
&= \|w\| \cdot (\sin(\alpha)\frac{x}{\|w\|} + \cos(\alpha)\frac{y}{\|w\|}) \\
&= x\sin(\alpha) + y\cos(\alpha)
\end{aligned}
$$



Rotations

- Then, compute u

$$
\begin{aligned}
u &= \|w\| \cdot \cos(\alpha + \beta) \\
&= \|w\| \cdot (\cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)) \\
&= \|w\| \cdot (\cos(\alpha)\frac{x}{\|w\|} - \sin(\alpha)\frac{y}{\|w\|}) \\
&= x\cos(\alpha) - y\sin(\alpha)
\end{aligned}
$$



Rotations

- All together:

$$u = x\cos(\alpha) - y\sin(\alpha)$$
$$v = x\sin(\alpha) + y\cos(\alpha)$$

Rotations

- All together:

$$\begin{aligned} u &= x\cos(\alpha) - y\sin(\alpha) \\ v &= x\sin(\alpha) + y\cos(\alpha) \end{aligned} \implies \begin{pmatrix} u \\ v \end{pmatrix} = \underbrace{\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}}_{R_\alpha} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

$$R_\alpha = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

Rotations

- Rotations in 3D are specified with an angle and a rotation axis
- The 2D rotation helps us to define 3D rotations:

$$R_\alpha^x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \quad R_\alpha^y = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix} \quad R_\alpha^z = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Signs are different to ensure the ccw rotation

Rotations

- Using the rotation matrices → position vectors can be rotate around one of the three unit axes
- Also possible to combine them (e.g., first rotate around the x-axis, then around the y-axis)
- This quickly introduces a problem called Gimbal lock → normally we have three degrees of freedom, after rotating it may happen that two axes coincide such that we loose one degree of freedom

Rotations

- Better solution is to rotate around an arbitrary unit vector n
- Instead of combining the rotation matrices

$$R_{\hat{n}}(\alpha) = \begin{pmatrix} n_1^2\left(1-\cos\alpha\right)+\cos\alpha & n_1 n_2\left(1-\cos\alpha\right)-n_3\sin\alpha & n_1 n_3\left(1-\cos\alpha\right)+n_2\sin\alpha \\ n_2 n_1\left(1-\cos\alpha\right)+n_3\sin\alpha & n_2^2\left(1-\cos\alpha\right)+\cos\alpha & n_2 n_3\left(1-\cos\alpha\right)-n_1\sin\alpha \\ n_3 n_1\left(1-\cos\alpha\right)-n_2\sin\alpha & n_3 n_2\left(1-\cos\alpha\right)+n_1\sin\alpha & n_3^2\left(1-\cos\alpha\right)+\cos\alpha \end{pmatrix}$$

- Even this matrix does not completely prevent gimbal lock (but it gets a lot harder)
- To truly prevent Gimbal locks, need quaternions (safer and computationally friendly)

Combining Matrices

- True power from using matrices for transformations is the combination of multiple transformations in a single matrix
- Say we have a vector (x, y, z) and we want to scale it by 2 and then translate it by $(1,2,3)$
- → Need a translation and a scaling matrix:

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

Combining Matrices

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

- Note, first a translation and then a scale transformation
- Matrix multiplication is not commutative (order is important!)
- Right-most matrix is first multiplied with the vector → read the multiplications from right to left
- When combining matrices it is advised to do:
  - 1. scaling
  - 2. rotations
  - 3. Translations
- E.g., if you would first do a translation and then scale, the translation vector would also scale!

Combining Matrices

- Running the final transformation matrix on our vector results in the following vector:

$$\begin{pmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 2x+1 \\ 2y+2 \\ 2z+2 \\ 1 \end{pmatrix}$$

GLM

Introduction

- Now time to use transformations
- OpenGL does not have any form of matrix or vector knowledge built in
- But, there is an easy-to-use and tailored-for-OpenGL mathematics library called GLM

Introduction

- GLM stands for Open**GL** **M**athematics (header-only library → only include no linking and compiling)
- GLM can be downloaded: https://glm.g-truc.net

GLM 0.9.9.8

Groovounet released this on 13 Apr · 21 commits to master since this release

Features:

- Added GLM_EXT_vector_intX* and GLM_EXT_vector_uintX* extensions
- Added GLM_EXT_matrix_intX* and GLM_EXT_matrix_uintX* extensions

Improvements:

- Added clamp, repeat, mirrorClamp and mirrorRepeat function to GLM_EXT_scalar_commond and GLM_EXT_vector_commond extensions with tests

Fixes:

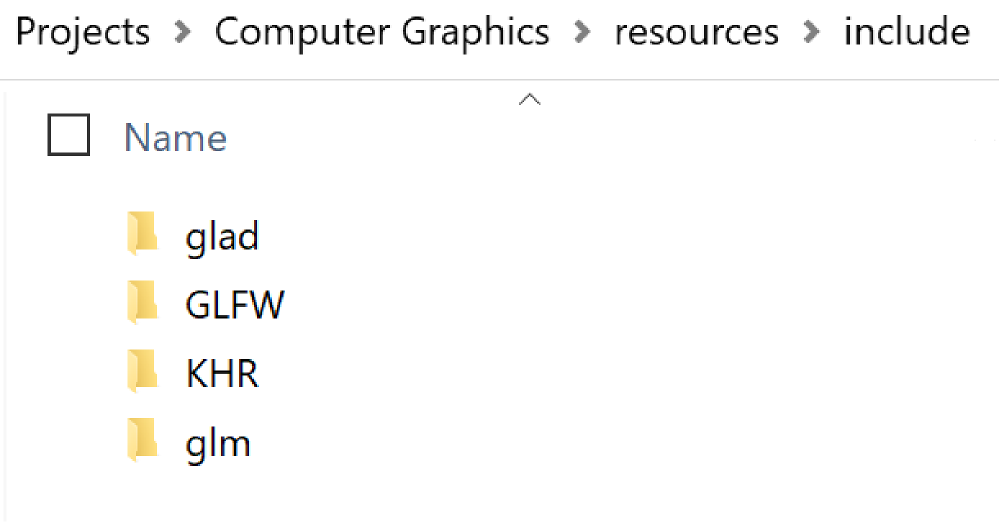
- Fixed unnecessary warnings from matrix_projection.inl #995
- Fixed quaternion slerp overload which interpolates with extra spins #996
- Fixed for glm::length using arch64 #992
- Fixed singularity check for quatLookAt #770

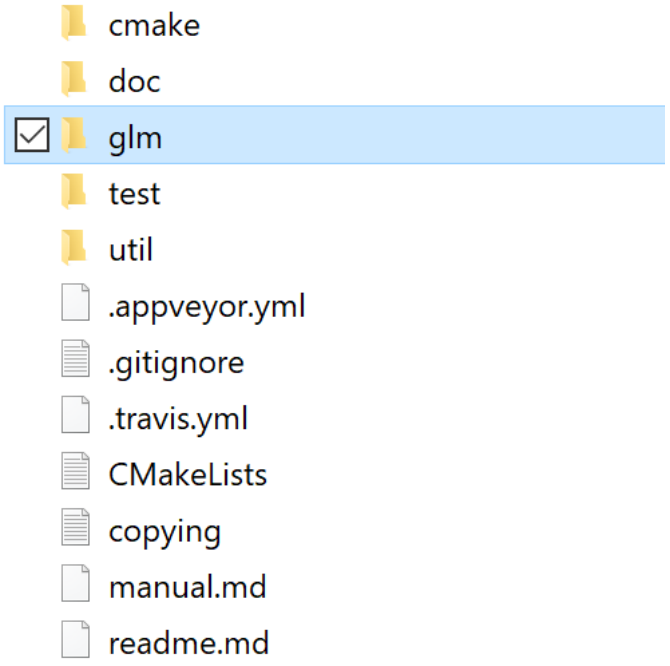
Assets 4

glm-0.9.9.8.7z	3.27 MB
glm-0.9.9.8.zip	5.41 MB
Source code (zip)	
Source code (tar.gz)	

Introduction

- Copy the root directory (glm) of the header files into your includes folder





Introduction

- Most of GLM’s functionality can be found in only 3 headers files:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

Introduction

- First, translate a vector of (1,0,0) by (1,1,0)
- (Note that we define it as a glm::vec4 with its homogenous coordinate set to 1.0):

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0); // not an identity matrix per default

trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

!!!Note: `glm::mat4 trans = glm::mat4(1.0);`
this is different than in the book!!!

Introduction

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0); // not an identity matrix per default

trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

- First define a vector named *vec* using GLM's built-in vector class
- Next define a mat4 named *trans* (it is set as the identity matrix)
- Next create a transformation matrix by passing *trans* to the glm::translate function, together with a translation vector (given matrix is multiplied with a translation matrix and the resulting matrix is returned)
- Then, multiply *vec* by the transformation matrix and output the result

Introduction

- Now, translate, scale, and rotate the textured wall from last lecture
- First, rotate the wall by 90 degrees counter-clockwise
- Then, scale it by 0.5, thus making it twice as small
- Finally, translate it:

```
glm::mat4 trans = glm::mat4(1.0);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
```

Introduction

```
glm::mat4 trans = glm::mat4(1.0);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
```

- GLM expects its angles in radians → convert the degrees to radians using glm::radians
- Note 1: Textured rectangle is on the XY plane → rotate around the Z-axis
- GLM automatically multiplies the matrices together (resulting in one transformation matrix)
- Note 2: Read the transformations from bottom to top!!!

Introduction

- Transformation matrix should be passed to the shader
- So, use a mat4 uniform and multiply the position vector by the matrix

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
uniform mat4 transform;

out vec2 TexCoord;
void main()
{
gl_Position = transform*vec4(aPos, 1.0);
TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

Introduction

GLSL also has mat2 and mat3 types that allow for swizzling-like operations just like vectors.

```
mat3 Matrix;
Matrix[1].yzx = vec3(3.0, 1.0, 2.0);
```

Introduction

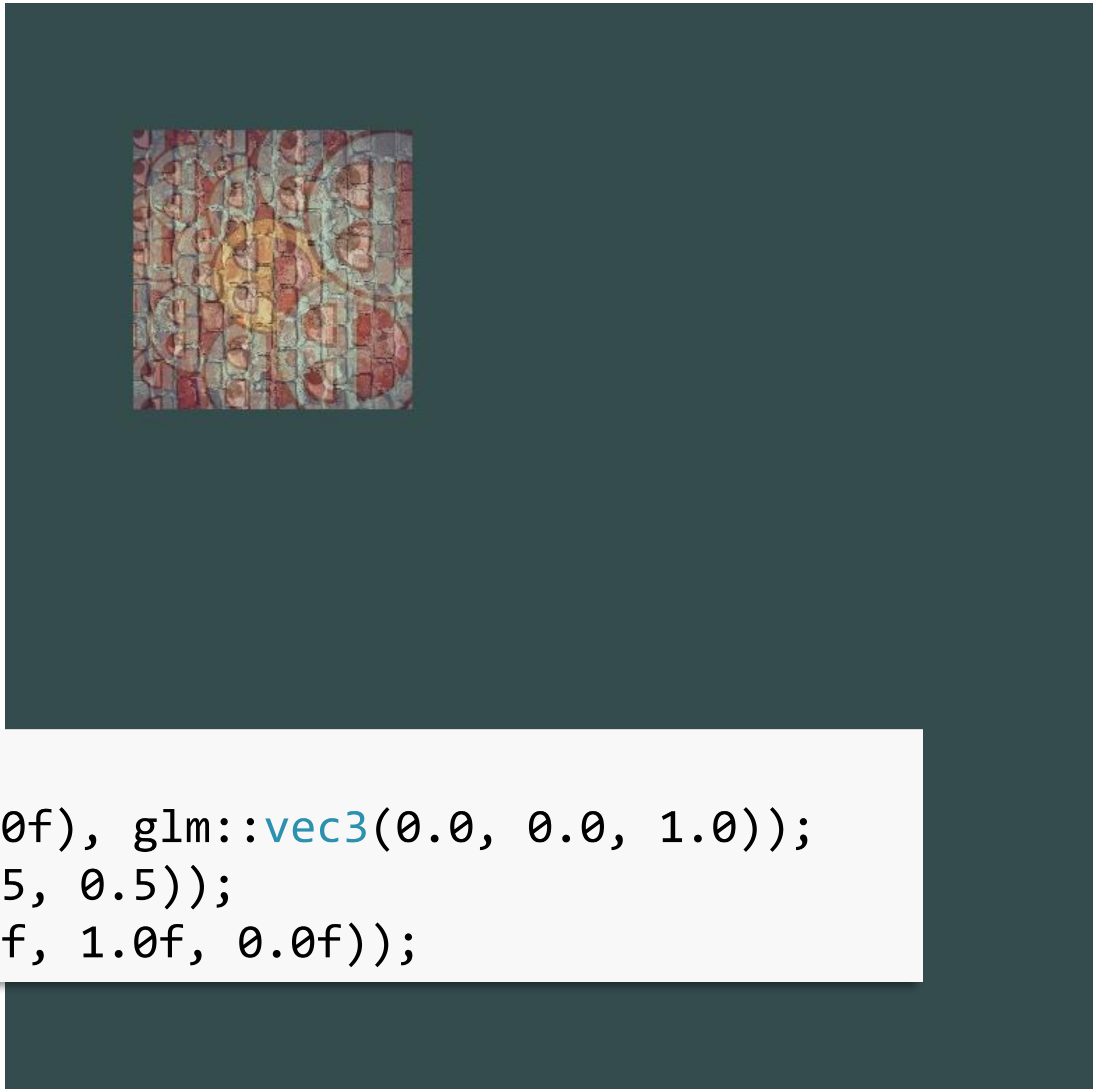
- Still need to pass the transformation matrix to the shader though:

```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

- 1. Is the uniform’s location
- 2. tells OpenGL how many matrices are send = 1
- 3. asks if the matrix should be transposed (swap the columns and rows, no as GLM gives the right matrix)
- 4. is the actual matrix data, but GLM stores their matrices not in the exact way that OpenGL likes to receive them so transform them with GLM’s built-in function value_ptr

F5...

...nice!



```
glm::mat4 trans = glm::mat4(1.0);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
```

F5…

…nice, too!

But be careful with the order!

```
glm::mat4 trans = glm::mat4(1.0);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

Rotation

- To rotate the wall over time use this code in the game loop
- (Needs to update the matrix each render iteration):

```
glm::mat4 trans = glm::mat4(1.0);
trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0, 0.0, 1.0));
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

F5…

… rotating beauty!

Complex numbers*

Introduction

- The complex numbers extend the range of real numbers such that the equation

$$x^2 + 1 = 0$$

has a solution:

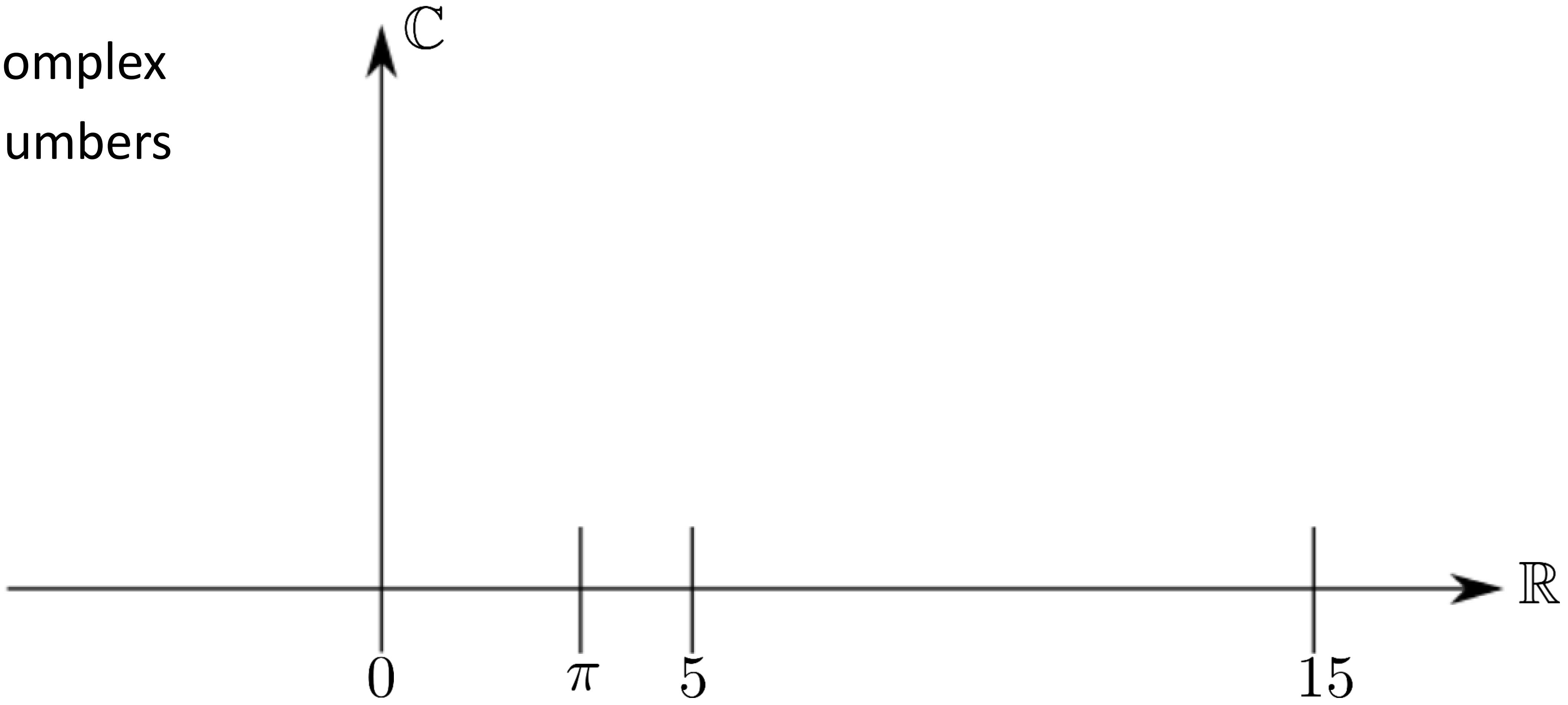
$$x_0 = i$$

Complex Numbers

- Real numbers

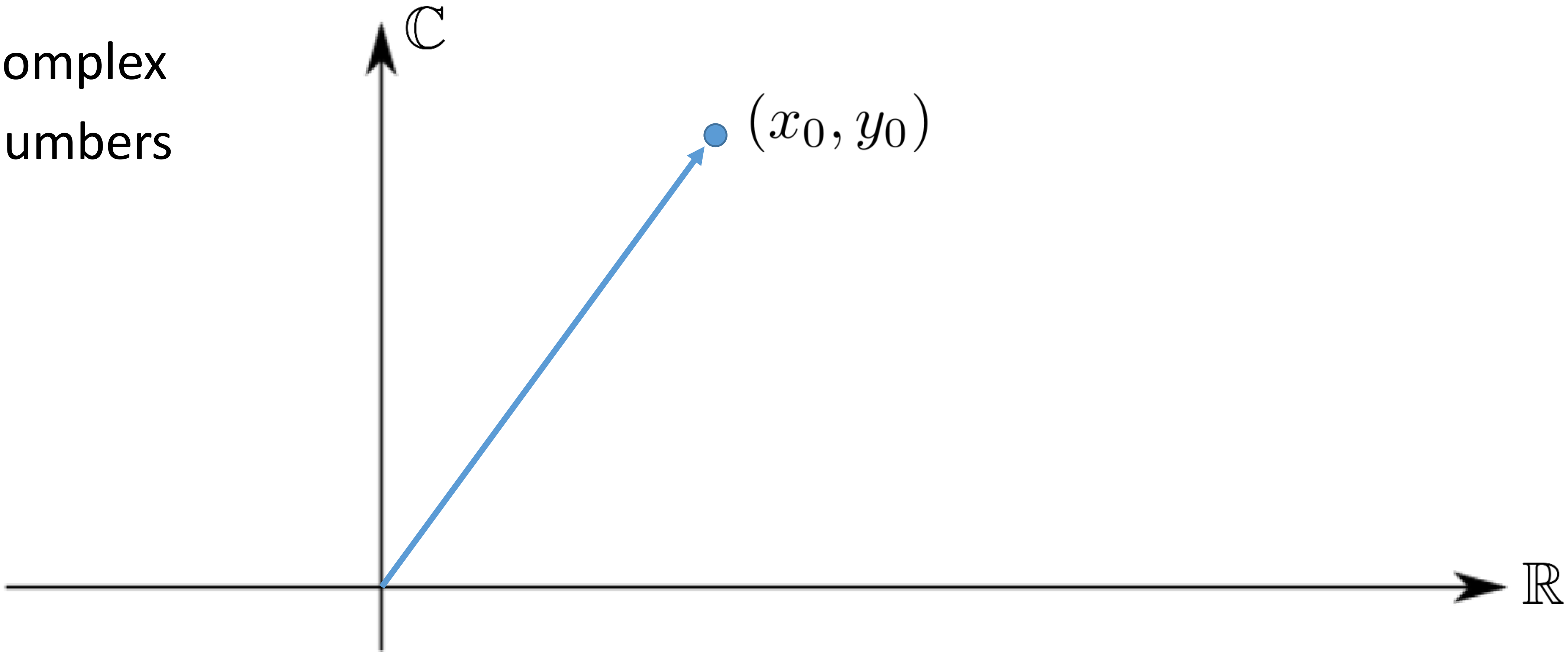
Complex Numbers

- Complex numbers



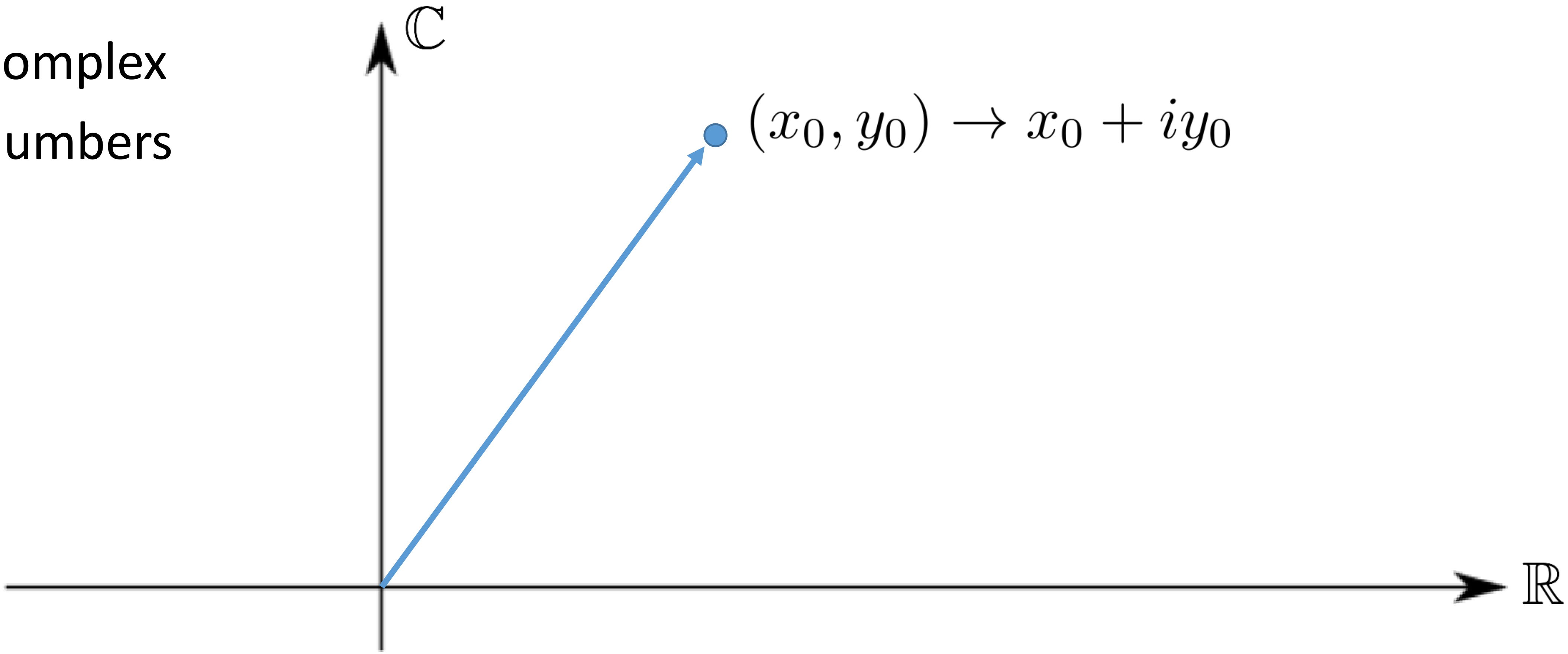
Complex Numbers

- Complex numbers



Complex Numbers

- Complex numbers



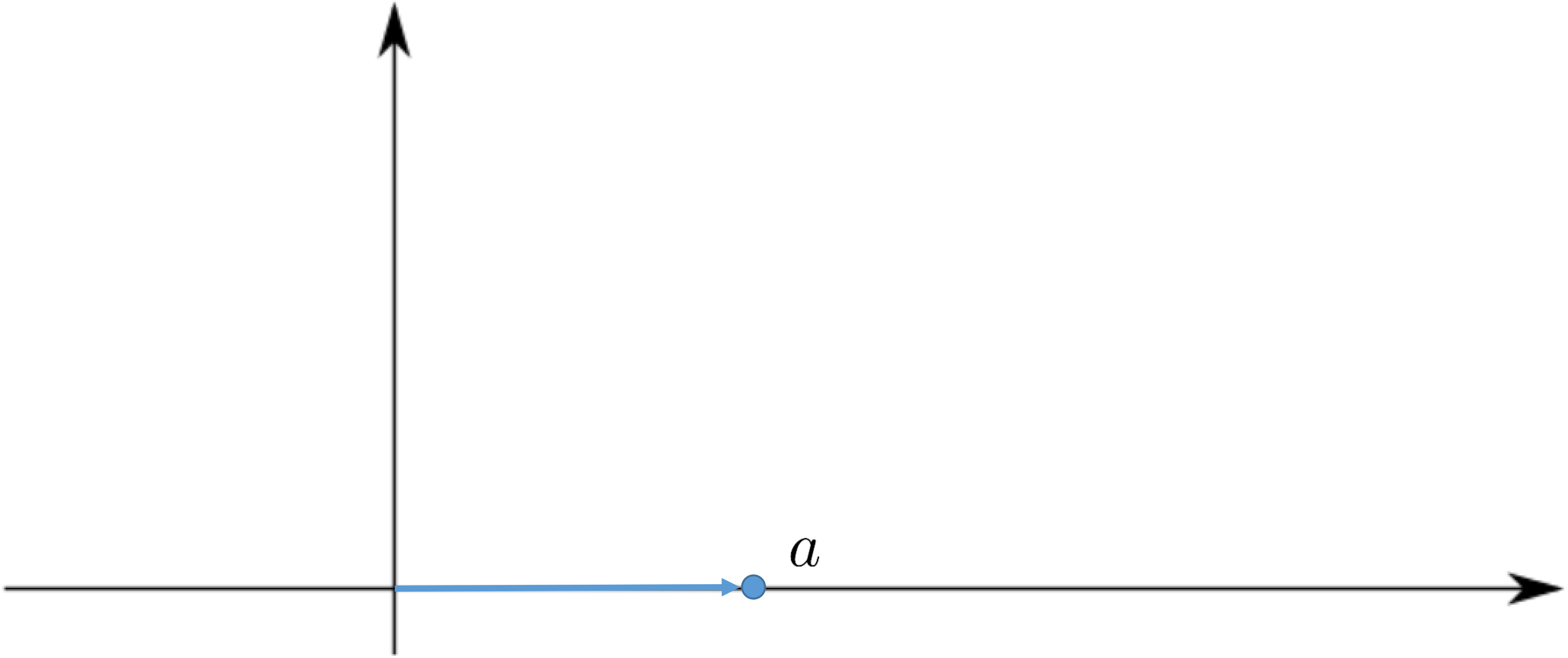
Complex Numbers

- What is ‘i’?

$$i^2 = -1$$

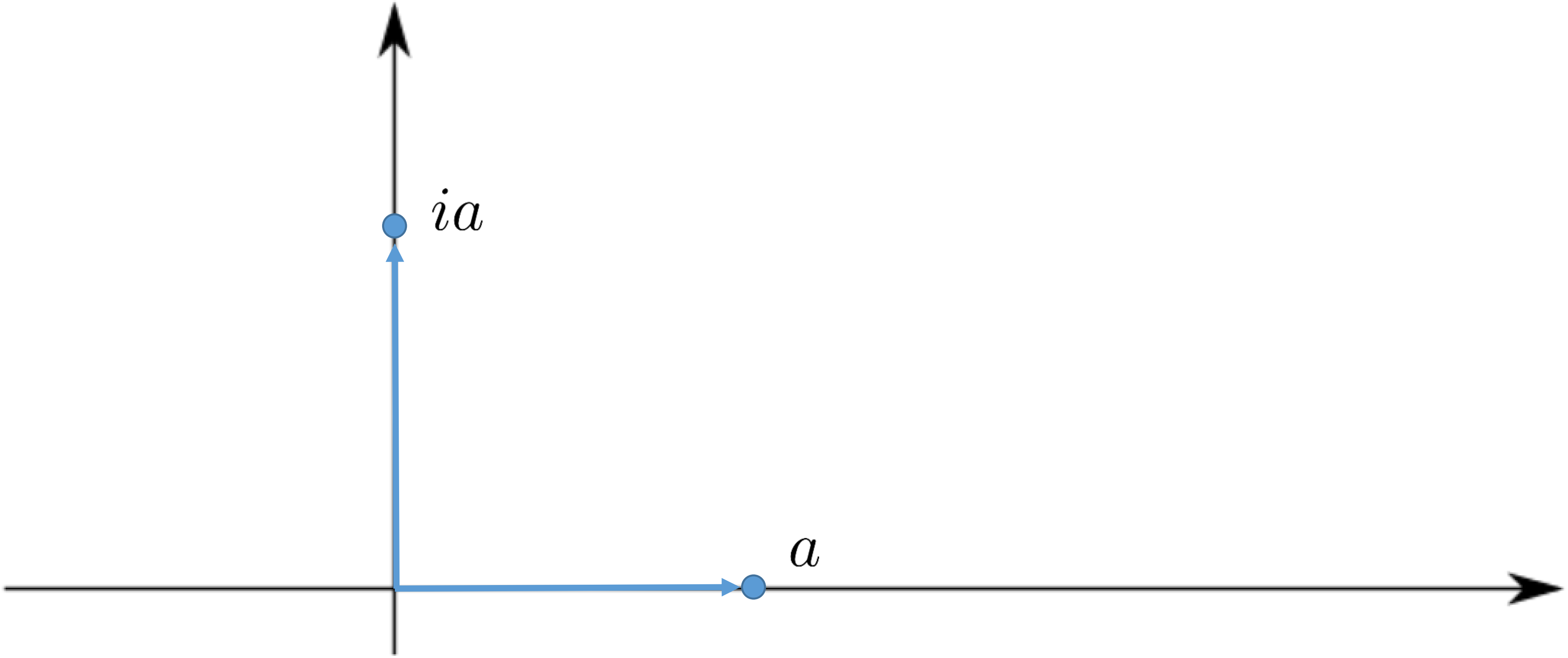
Complex Numbers

- Geometrically it is a counterclockwise rotatiaon of 90°



Complex Numbers

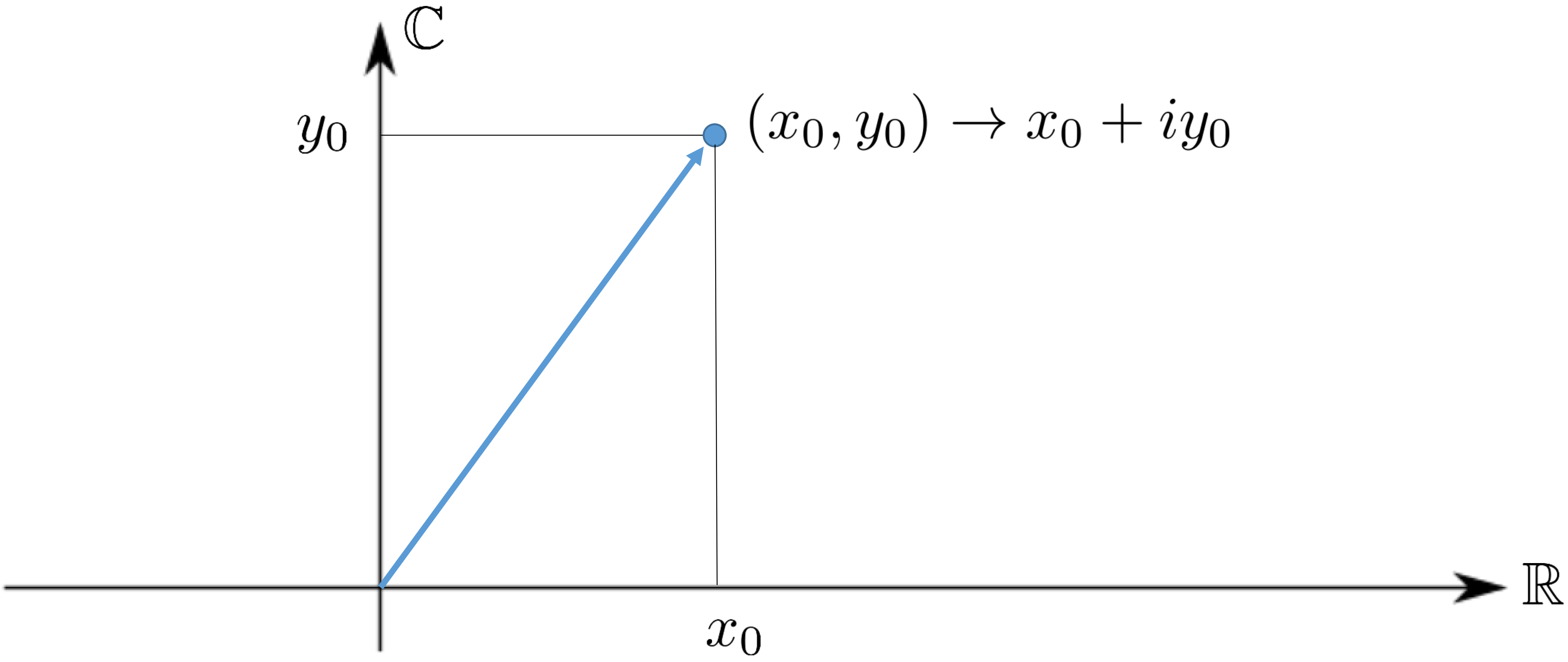
- Geometrically it is a counterclockwise rotation of 90°



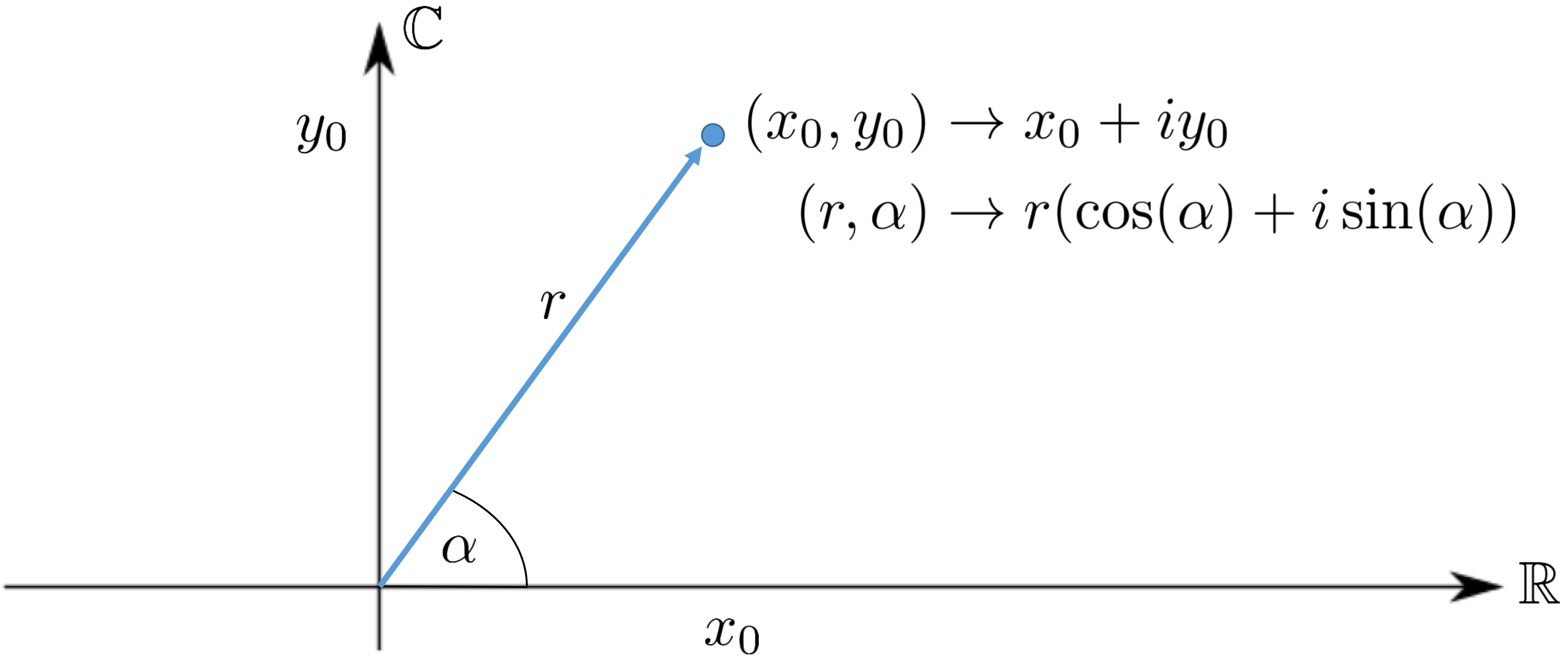
Complex Numbers

- Different ways to express a complex number

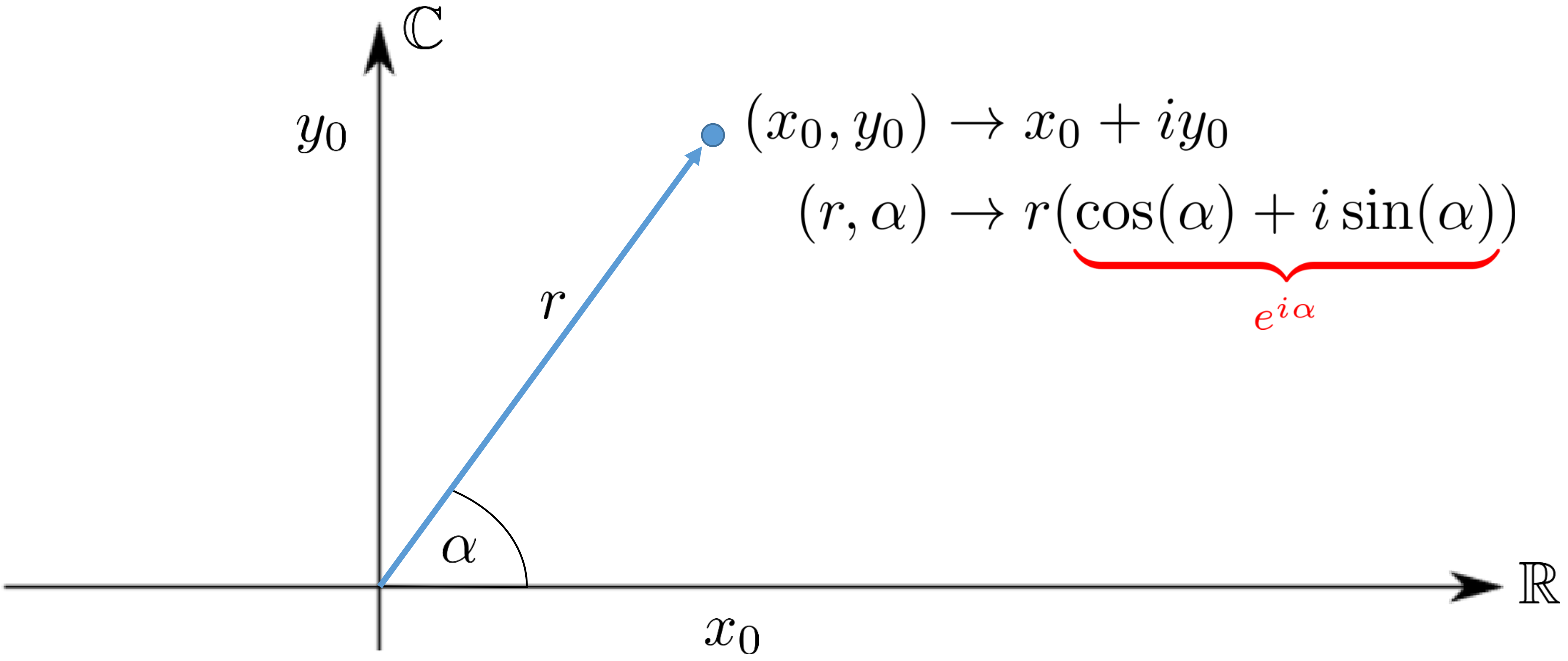
Complex Numbers



Complex Numbers



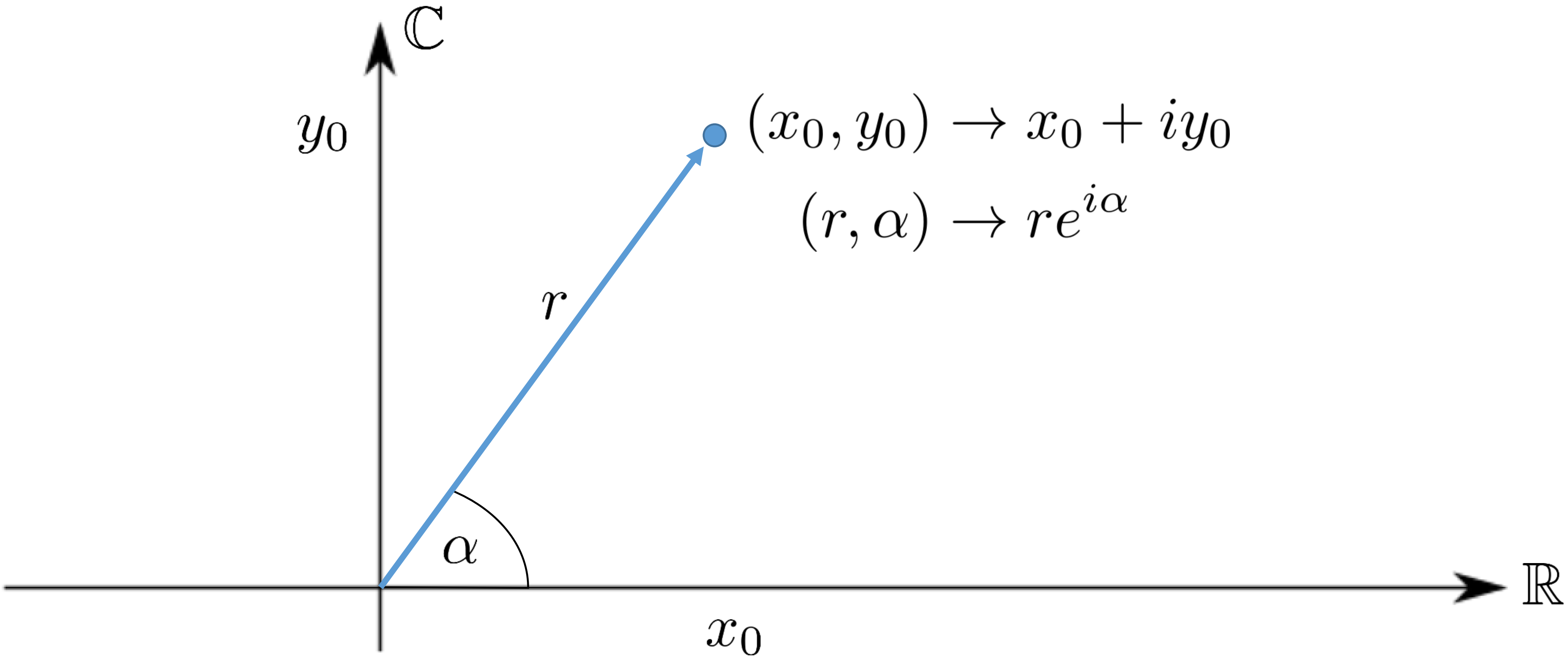
Complex Numbers



$$(x_0, y_0) \rightarrow x_0 + iy_0$$

$$(r, \alpha) \rightarrow r\underbrace{(\cos(\alpha) + i\sin(\alpha))}_{e^{i\alpha}}$$

Complex Numbers



Rules

- Let $z_1 = a + b\mathrm{i}$ and $z_2 = c + d\mathrm{i}$ be complex numbers with $a, b, c, d \in \mathbb{R}$

$$z_1 + z_2 = (a + bi) + (c + di) = (a + c) + (b + d)i$$

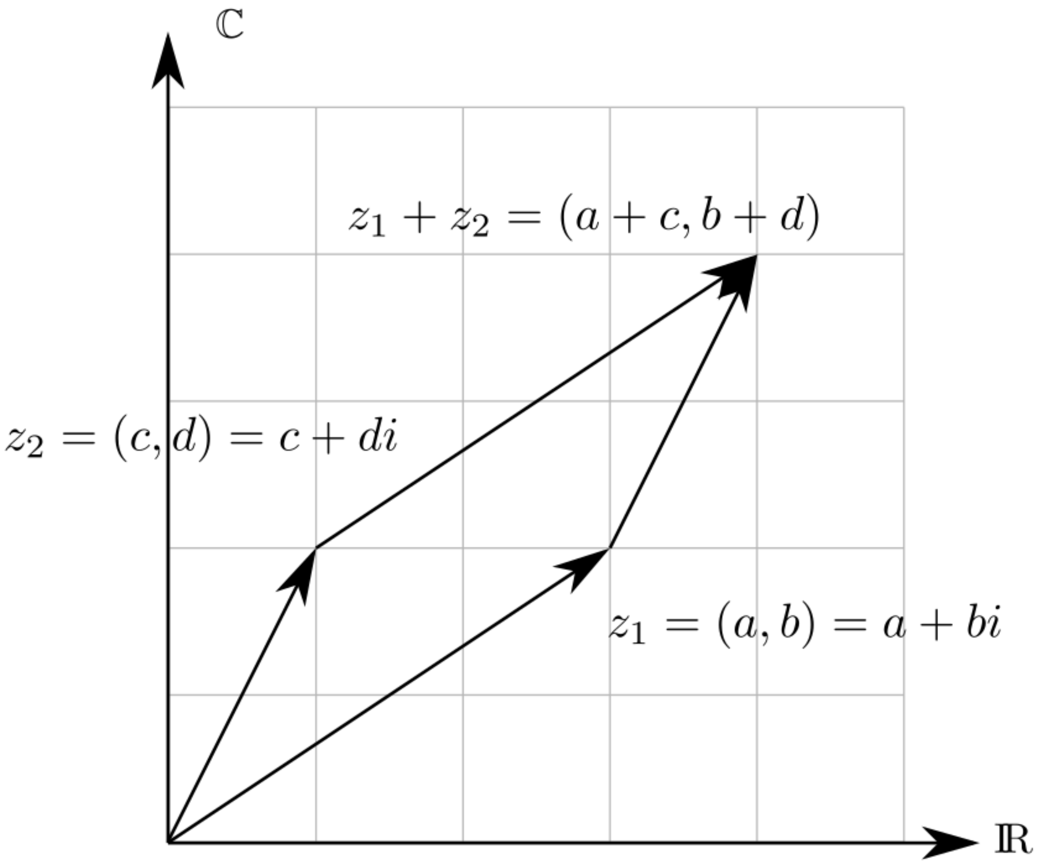
$$z_1 - z_2 = (a + bi) - (c + di) = (a - c) + (b - d)i$$

$$z_1 \cdot z_2 = (a + bi) \cdot (c + di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i$$

$$\frac{z_1}{z_2} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

Rules

- Addition/subtraction is a simple vector addition



Rotation

- To rotate a vector $\mathrm{z} = (a, b)$ CCW around an angle α, we can multiply it with the rotation matrix:

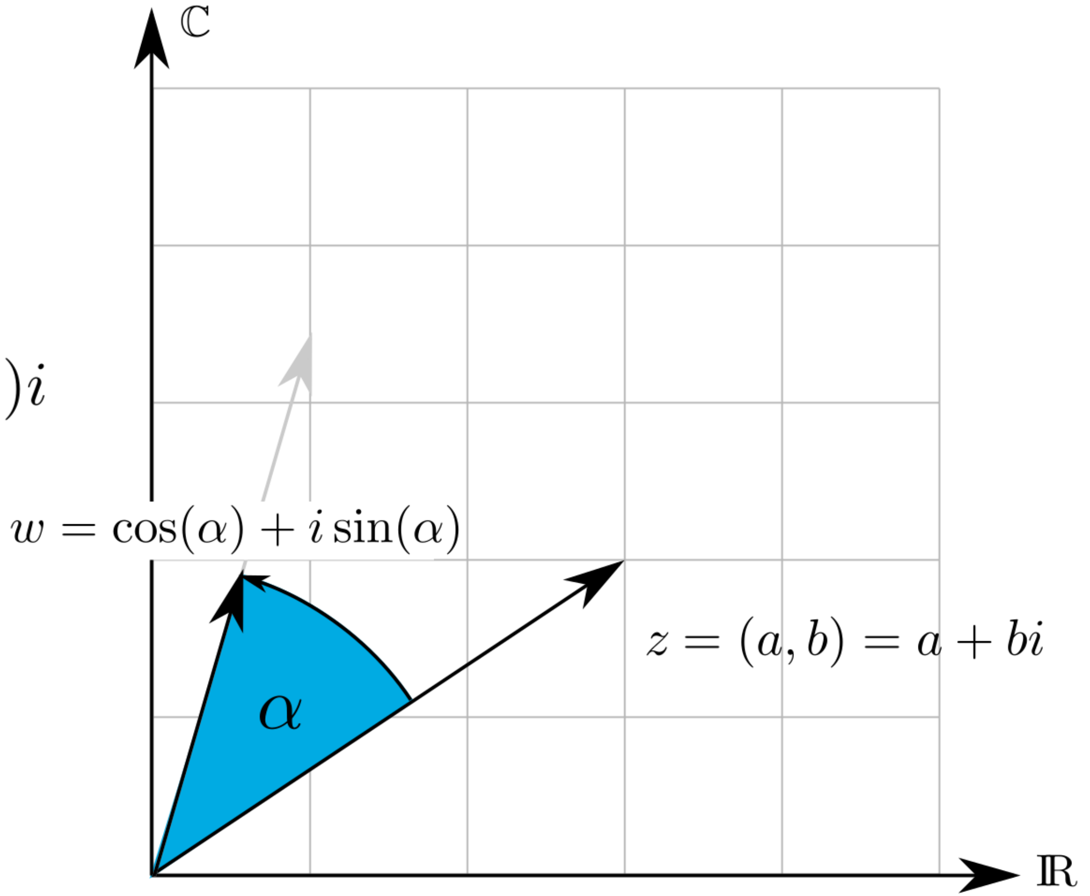
$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot z = \begin{pmatrix} a\cos(\alpha) - b\sin(\alpha) \\ a\sin(\alpha) + b\cos(\alpha) \end{pmatrix}$$

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot z = \begin{pmatrix} a\cos(\alpha) - b\sin(\alpha) \\ a\sin(\alpha) + b\cos(\alpha) \end{pmatrix}$$

Rotation

- Or, we multiply it with a complex number:

$$\begin{aligned} z \cdot w &= (a + bi) \cdot (\cos(\alpha) + i\sin(\alpha)) \\ &= (a\cos(\alpha) - b\sin(\alpha)) + (a\sin(\alpha) + b\cos(\alpha))i \end{aligned}$$



Rotation

- 2D rotations can be achieved with a rotation matrix or with the multiplication of complex numbers of the form $\cos(\alpha) + i\sin(\alpha)$

Quaternions*

Quaternions

- Maybe, we need to somehow extend the complex numbers such that we use a further dimension:

$$z = a + ib + jc$$

Quaternions

- Maybe, we need to somehow extend the complex numbers such that we use a further dimension:

$$z = a + ib + jc$$

- That’s what people thought in the past, but it does not work

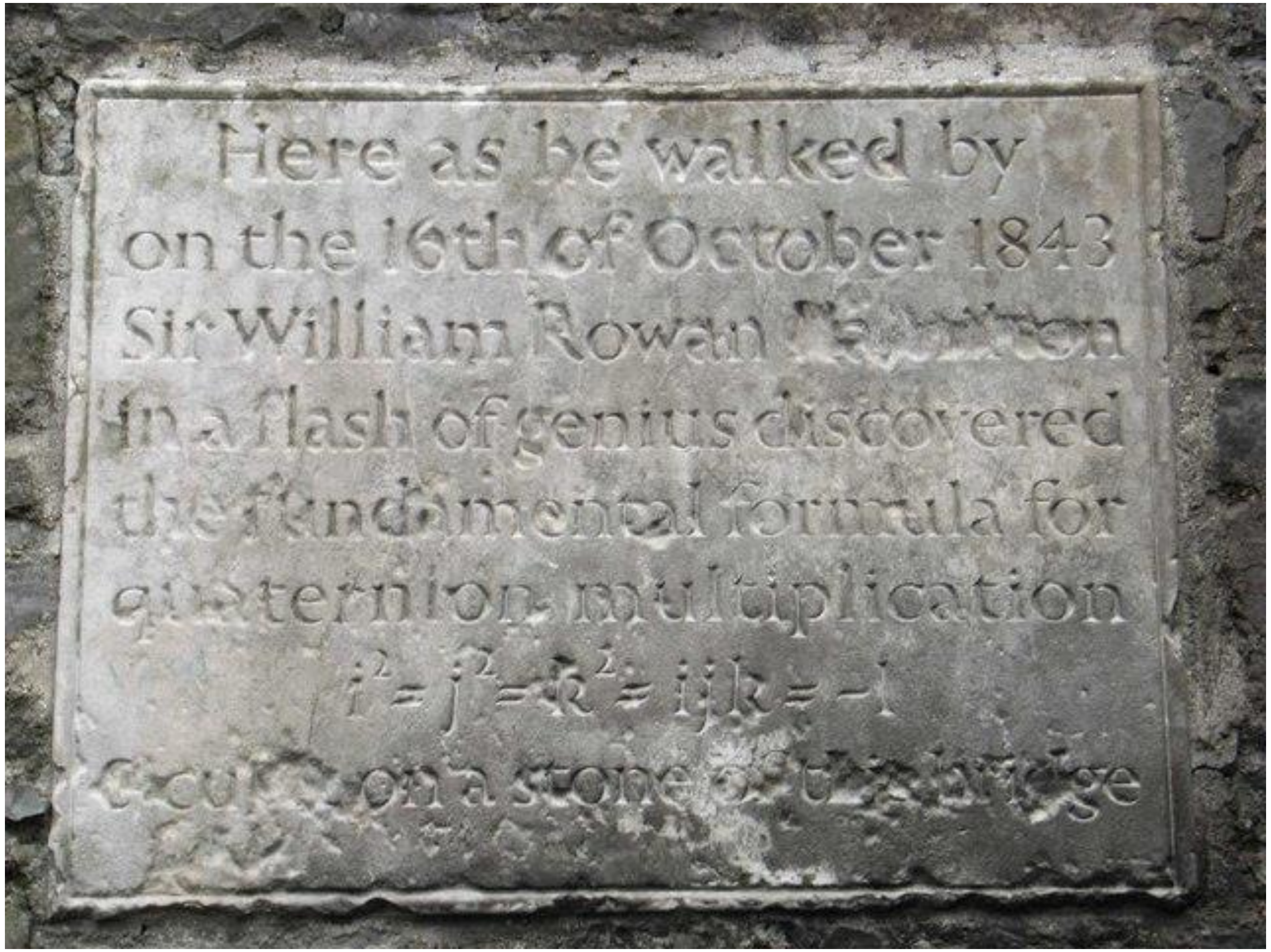
Quaternions

- Actually, we need four dimensions to rotate in 3D!

$$z = a + ib + jc + kd$$

Quaternions

"Here as he walked by
on the 16th of October 1843
Sir William Rowan Hamilton
in a flash of genius discovered
the fundamental formula for
quaternion multiplication
$i^2 = j^2 = k^2 = ijk = -1$
& cut it on a stone of this bridge"



Quaternions

- Complex number:

$$z = a + ib,\ i^2 = -1$$

Quaternions

- Complex number:

$$z = a + ib, \;\; i^2 = -1$$

- Quaternion:

$$w = a + ib + jc + kd$$

$$i^2 = j^2 = k^2 = -1$$

$$ijk = -1$$

Quaternions

- Quaternion multiplication
- Not commutative

x	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

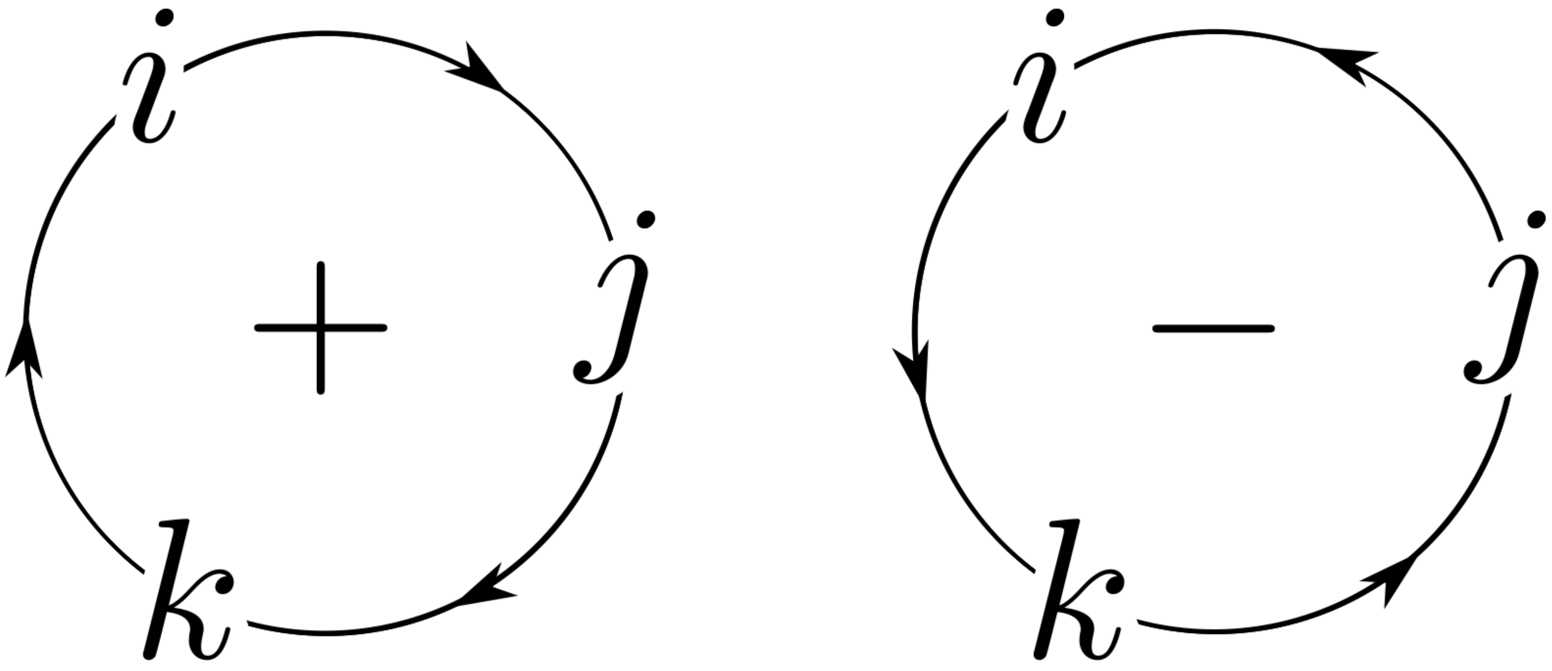
b

ab

a

Quaternions

- Quaternion multiplication
- Not commutative



Quaternions

- Quaternion multiplication

$$
\begin{aligned}
w_1 \cdot w_2 &= (a_1 + ib_1 + jc_1 + kd_1)(a_2 + ib_2 + jc_2 + kd_2) \\
&= (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) \\
&+ i(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2) \\
&+ j(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2) \\
&+ k(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)
\end{aligned}
$$

(From: Berthold K. P. Horn, Closed-form solution of absolute orientation using unit quaternions)

Quaternions

$$\begin{aligned} w_1 \cdot w_2 = {} & (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) \\ & + i(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2) \\ & + j(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2) \\ & + k(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2) \end{aligned}$$

- Product can also be written as a matrix, $w_1 = (a, b, c, d)$

$$w_1 \cdot w_2 = \underbrace{\begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}}_{\mathbf{W}_1} \cdot w_2$$

Quaternions

$$\begin{aligned} w_1 \cdot w_2 = &(a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) \\ &+ i(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2) \\ &+ j(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2) \\ &+ k(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2) \end{aligned}$$

- Product can also be written as a matrix, $w_1 = (a, b, c, d)$

$$w_1 \cdot w_2 = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix} \cdot w_2 = \mathbf{W}_1 \cdot w_2$$

$$w_2 \cdot w_1 = \begin{pmatrix} a & -b & -c & -d \\ b & a & d & -c \\ c & -d & a & b \\ d & c & -b & a \end{pmatrix} \cdot w_2 = \bar{\mathbf{W}}_{\mathbf{1}} \cdot w_2$$

Quaternions

$$\begin{aligned} w_1 \cdot w_2 = (&a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) \\ &+ i(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2) \\ &+ j(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2) \\ &+ k(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2) \end{aligned}$$

- Product can also be written as a matrix, $w_1 = (a, b, c, d)$

$$w_1 \cdot w_2 = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix} \cdot w_2 = \mathbf{W}_1 \cdot w_2$$

$$w_2 \cdot w_1 = \begin{pmatrix} a & -b & -c & -d \\ b & a & d & -c \\ c & -d & a & b \\ d & c & -b & a \end{pmatrix} \cdot w_2 = \bar{\mathbf{W}}_{\mathbf{1}} \cdot w_2$$

Quaternions

$$\mathbf{W} = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

- If $w_1 = (a, b, c, d)$ has unit length, the matrix is orthogonal

$$\mathbf{W}\mathbf{W}^T = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix} \begin{pmatrix} a & b & c & d \\ -b & a & d & -c \\ -c & -d & a & b \\ -d & c & -b & a \end{pmatrix}$$

$$= \begin{pmatrix} r^2 & 0 & 0 & 0 \\ 0 & r^2 & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \end{pmatrix}, \quad r^2 = a^2 + b^2 + c^2 + d^2$$

Quaternions

$$\bar{\mathbf{W}} = \begin{pmatrix} a & -b & -c & -d \\ b & a & d & -c \\ c & -d & a & b \\ d & c & -b & a \end{pmatrix}$$

- If $w_1 = (a, b, c, d)$ has unit length, the matrix is orthogonal

$$\bar{\mathbf{W}}\bar{\mathbf{W}}^T = \begin{pmatrix} a & -b & -c & -d \\ b & a & d & -c \\ c & -d & a & b \\ d & c & -b & a \end{pmatrix} \begin{pmatrix} a & b & c & d \\ -b & a & -d & c \\ -c & d & a & -b \\ -d & -c & b & a \end{pmatrix}$$

$$= \begin{pmatrix} r^2 & 0 & 0 & 0 \\ 0 & r^2 & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \end{pmatrix}, \quad r^2 = a^2 + b^2 + c^2 + d^2$$

Quaternions

- Dot product of two quaternions:

$$
\begin{aligned}
w_1 \circ w_2 &= (a_1 + ib_1 + jc_1 + kd_1) \circ (a_2 + ib_2 + jc_2 + kd_2) \\
&= a_1 a_2 + b_1 b_2 + c_1 c_2 + d_1 d_2
\end{aligned}
$$

- Complex conjugate:

$$
\begin{aligned}
w &= a + ib + jc + kd \\
w^* &= a - ib - jc - kd
\end{aligned}
$$

$$
\begin{aligned}
w_1 \cdot w_2 &= (a_1 + ib_1 + jc_1 + kd_1)(a_2 + ib_2 + jc_2 + kd_2) \\
&= (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) \\
&+ i(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2) \\
&+ j(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2) \\
&+ k(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)
\end{aligned}
$$

Quaternions

- Quaternion multiplication

$$
\begin{aligned}
w_1 \cdot w_1^* &= (a_1 + ib_1 + jc_1 + kd_1)(a_1 - ib_1 - jc_1 - kd_1) \\
&= (a_1a_1 + b_1b_1 + c_1c_1 + d_1d_1) \\
&+ i(-a_1b_1 + b_1a_1 - c_1d_1 + d_1c_1) \\
&+ j(-a_1c_1 + b_1d_1 + c_1a_1 - d_1b_1) \\
&+ k(-a_1d_1 - b_1c_1 + c_1b_1 + d_1a_1)
\end{aligned}
$$

(From: Berthold K. P. Horn, Closed-form solution of absolute orientation using unit quaternions)

$$
\begin{aligned}
w_1 \cdot w_2 &= (a_1 + ib_1 + jc_1 + kd_1)(a_2 + ib_2 + jc_2 + kd_2) \\
&= (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) \\
&+ i(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2) \\
&+ j(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2) \\
&+ k(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)
\end{aligned}
$$

Quaternions

- Quaternion multiplication

$$
\begin{aligned}
w_1 \cdot w_1^* &= (a_1 + ib_1 + jc_1 + kd_1)(a_1 - ib_1 - jc_1 - kd_1) \\
&= (a_1a_1 + b_1b_1 + c_1c_1 + d_1d_1) \\
&+ i(-a_1b_1 + b_1a_1 - c_1d_1 + d_1c_1) \\
&+ j(-a_1c_1 + b_1d_1 + c_1a_1 - d_1b_1) \\
&+ k(-a_1d_1 - b_1c_1 + c_1b_1 + d_1a_1) \\
&= (a_1a_1 + b_1b_1 + c_1c_1 + d_1d_1)
\end{aligned}
$$

(From: Berthold K. P. Horn, Closed-form solution of absolute orientation using unit quaternions)

Quaternions

- Dot product of two quaternions:

$$
\begin{aligned}
w_1 \circ w_2 &= (a_1 + ib_1 + jc_1 + kd_1) \circ (a_2 + ib_2 + jc_2 + kd_2) \\
&= a_1 a_2 + b_1 b_2 + c_1 c_2 + d_1 d_2
\end{aligned}
$$

- Complex conjugate:

$$
\begin{aligned}
w &= a + ib + jc + kd \\
w^* &= a - ib - jc - kd
\end{aligned}
$$

$$\boxed{ww^* = w \circ w}$$

Quaternions

$$\mathbf{W} = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

- Non-commutative rule

$$w_1 \cdot w_2 = \mathbf{W}_1 \cdot w_2$$
$$w_2 \cdot w_1 = \bar{\mathbf{W}}_1 \cdot w_2$$

$$w_1^* \cdot w_2 =$$

Quaternions

$$\mathbf{W} = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

- Non-commutative rule

$$w_1 \cdot w_2 = \mathbf{W}_1 \cdot w_2$$
$$w_2 \cdot w_1 = \bar{\mathbf{W}}_1 \cdot w_2$$

$$w_1^* \cdot w_2 = \mathbf{W}_1^T \cdot w_2$$

Quaternions

$$\mathbf{W} = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

- Non-commutative rule

$$w_1 \cdot w_2 = \mathbf{W}_1 \cdot w_2$$
$$w_2 \cdot w_1 = \bar{\mathbf{W}}_1 \cdot w_2$$

$$w_1^* \cdot w_2 = \mathbf{W}_1^T \cdot w_2$$
$$w_2 \cdot w_1^* = \bar{\mathbf{W}}_1^T \cdot w_2$$

$$w_1^* \cdot w_2 = \mathbf{W}_1^T \cdot w_2$$
$$w_2 \cdot w_1^* = \bar{\mathbf{W}}_1^T \cdot w_2$$

Quaternions

- The composite product:

$$\begin{aligned} r' &= qrq^* \\ &= \\ &= \\ &= \end{aligned}$$

$$w_1^* \cdot w_2 = \mathbf{W}_1^T \cdot w_2$$

$$w_2 \cdot w_1^* = \bar{\mathbf{W}}_1^T \cdot w_2$$

Quaternions

- The composite product:

$$\begin{aligned} r' &= qrq^* \\ &= (\mathbf{Q}r)q^* \\ &= \\ &= \end{aligned}$$

Quaternions

$$w_1^* \cdot w_2 = \mathbf{W}_1^T \cdot w_2$$
$$w_2 \cdot w_1^* = \bar{\mathbf{W}}_1^T \cdot w_2$$

- The composite product:

$$\begin{aligned} r' &= qrq^* \\ &= (\mathbf{Q}r)q^* \\ &= \bar{\mathbf{Q}}^T(\mathbf{Q}r) \\ &= \end{aligned}$$

$$w_1^* \cdot w_2 = \mathbf{W}_1^T \cdot w_2$$
$$w_2 \cdot w_1^* = \bar{\mathbf{W}}_1^T \cdot w_2$$

Quaternions

- The composite product:

$$\begin{aligned} r' &= qrq^* \\ &= (\mathbf{Q}r)q^* \\ &= \bar{\mathbf{Q}}^T(\mathbf{Q}r) \\ &= (\bar{\mathbf{Q}}^T\mathbf{Q})r \end{aligned}$$

Quaternions

- The composite product:

$$r' = (\bar{\mathbf{Q}}^T \mathbf{Q}) r$$

- Let's assume

$$qq^* = 1$$

then the matrix $\boldsymbol{Q}$ is orthogonal

Quaternions

- The composite product:

$$r' = (\bar{\mathbf{Q}}^T \mathbf{Q}) r$$

- If the matrix $\boldsymbol{Q}$ is orthogonal then $\bar{\boldsymbol{Q}}^T \boldsymbol{Q}$ is orthogonal, too:

$$\begin{aligned}
(\bar{\mathbf{Q}}^T \mathbf{Q})(\bar{\mathbf{Q}}^T \mathbf{Q})^T &= (\bar{\mathbf{Q}}^T \mathbf{Q})(\mathbf{Q}^T \bar{\mathbf{Q}}) \\
&= \bar{\mathbf{Q}}^T \underbrace{\mathbf{Q}\mathbf{Q}^T}_{I} \bar{\mathbf{Q}} \\
&= \bar{\mathbf{Q}}^T \bar{\mathbf{Q}} \\
&= I
\end{aligned}$$

Quaternions

- Multiplication with a quaternion and the complex conjugate quaternion is equivalent with a rotation:

$$r' = qrq^* = (\bar{\mathbf{Q}}^T \mathbf{Q}) r$$

$$\bar{\mathbf{Q}}^T \mathbf{Q} = \begin{pmatrix} qq^* & 0 & 0 & 0 \\ 0 & & & \\ 0 & & \underbrace{\mathbf{R}}_{\text{Rotation}} & \\ 0 & & & \end{pmatrix}$$

(From: Berthold K. P. Horn, Closed-form solution of absolute orientation using unit quaternions)

Quaternions

- Applying a second rotation:

$$
\begin{aligned}
r'' &= pr'p^* \\
&= p(qrq^*)p^* \\
&= (pq)r(q^*p^*) \\
&= (pq)r(pq)^*
\end{aligned}
$$

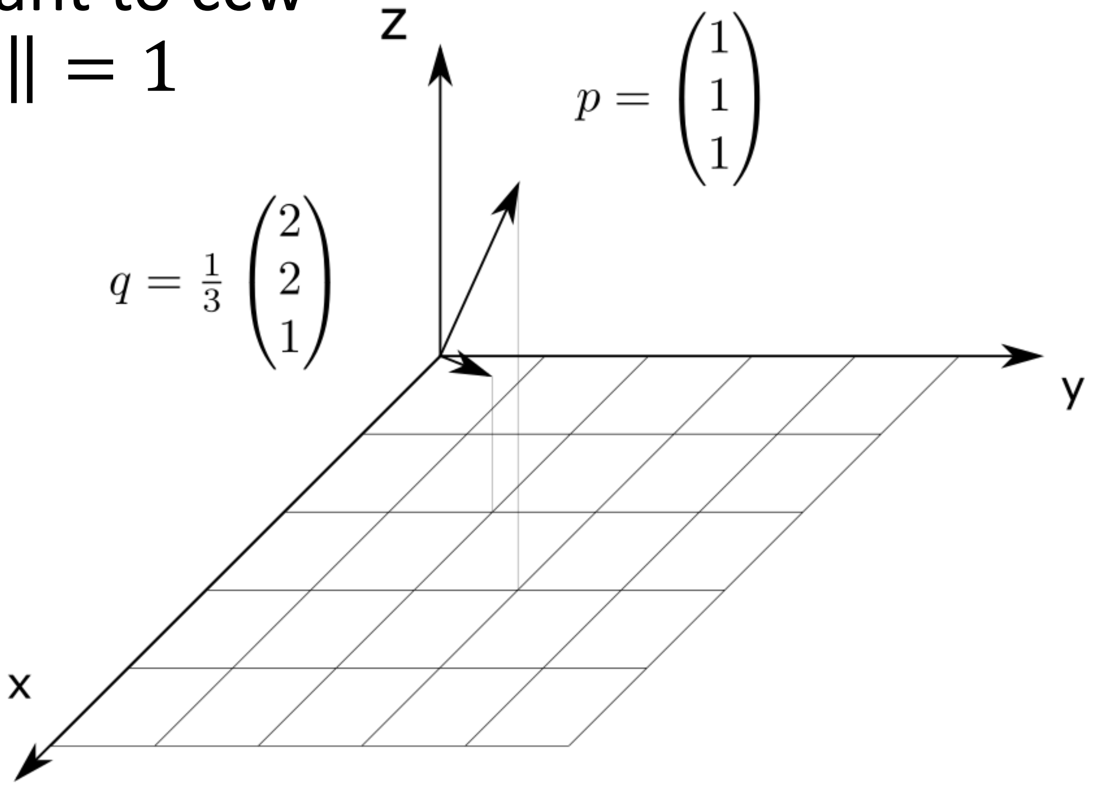
(From: Berthold K. P. Horn, Closed-form solution of absolute orientation using unit quaternions)

Rotations

- So again, instead of rotating a 3D point by defining rotation matrices, it can be done with a multiplication of a unit length quaternion

Rotations

- Assume, we have a point p and we want to ccw rotate them around an axis q with $\|q\| = 1$ about the angle α



Rotations

- First, rewrite q and p as a quaternion:

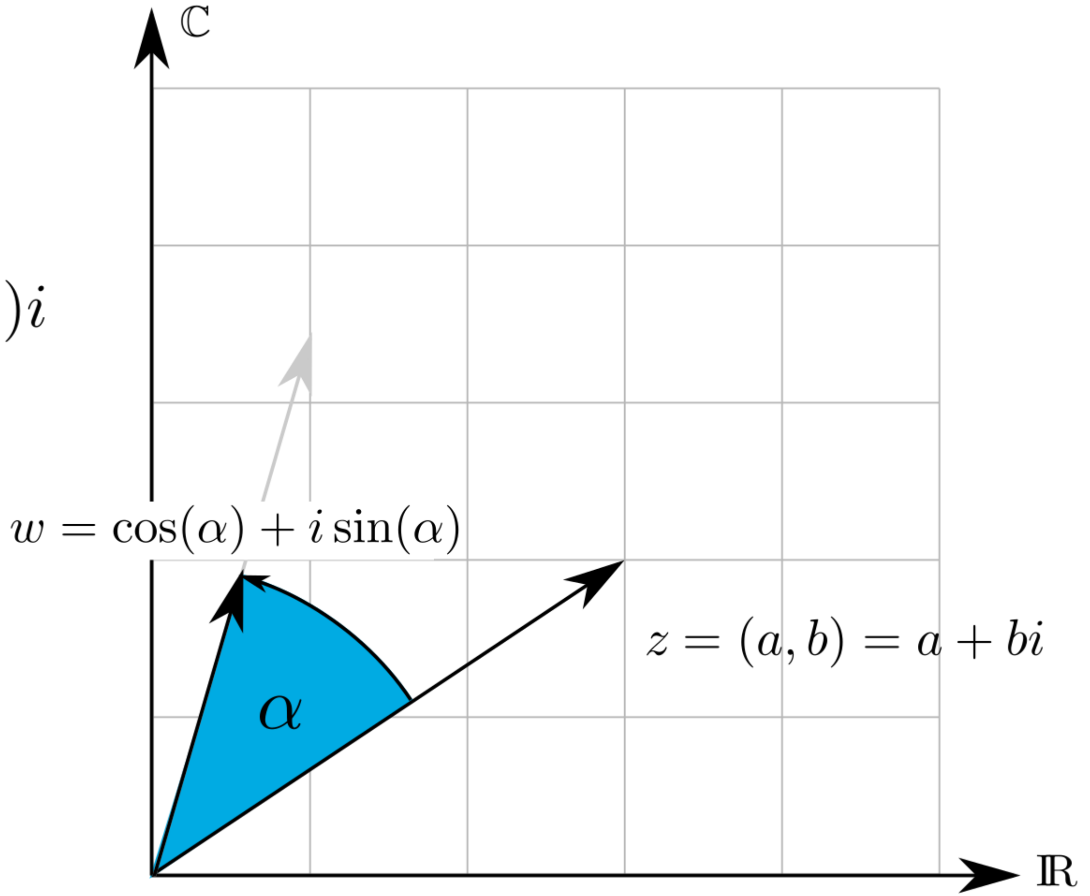
$$p = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \Rightarrow 1i + 1j + 1k$$

$$q = \frac{1}{3}\begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} \Rightarrow \frac{2}{3}i + \frac{2}{3}j + \frac{1}{3}k$$

Rotation

- Remember complex numbers:

$$
\begin{aligned}
z \cdot w &= (a + bi) \cdot (\cos(\alpha) + i\sin(\alpha)) \\
&= (a\cos(\alpha) - b\sin(\alpha)) + (a\sin(\alpha) + b\cos(\alpha))i
\end{aligned}
$$



$$p = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \Rightarrow 1i + 1j + 1k$$

$$q = \frac{1}{3} \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} \Rightarrow \frac{2}{3}i + \frac{2}{3}j + \frac{1}{3}k$$

Rotations

- It is slightly different
- First, we assign:

$$q \leftarrow \cos(\alpha/2) + \sin(\alpha/2) \cdot q$$

- Then, we determine

$$rot = q \cdot p \cdot q^*$$

$$rot = q \cdot p \cdot q^*$$

Rotations

- And we are done
- The complex parts of rot yield the coordinates

$$p = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \Rightarrow 1i + 1j + 1k$$

$$q = \frac{1}{3} \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} \Rightarrow \frac{2}{3}i + \frac{2}{3}j + \frac{1}{3}k$$

Rotations

- Assume $\alpha = 60°$

$$q \leftarrow \cos(\alpha/2) + \sin(\alpha/2) \cdot q$$

$$q = \frac{\sqrt{3}}{2} + \frac{1}{3}i + \frac{1}{3}j + \frac{1}{6}k$$

$$p = 1i + 1j + 1k$$

$$q^* = \frac{\sqrt{3}}{2} - \frac{1}{3}i - \frac{1}{3}j - \frac{1}{6}k$$

Rotations

$$q = \frac{\sqrt{3}}{2} + \frac{1}{3}i + \frac{1}{3}j + \frac{1}{6}k$$

$$p = 1i + 1j + 1k$$

$$q^* = \frac{\sqrt{3}}{2} - \frac{1}{3}i - \frac{1}{3}j - \frac{1}{6}k$$

- Assume $\alpha = 60°$

$$p \cdot q^* = \frac{5}{6} + (\frac{1}{6} + \frac{\sqrt{3}}{2})i + (-\frac{1}{6} + \frac{\sqrt{3}}{2})j + \frac{\sqrt{3}}{2}k$$

$$q \cdot p \cdot q^* = 0 + \frac{1}{18}(19 + 3\sqrt{3})i + \frac{1}{18}(19 - 3\sqrt{3})j + \frac{7}{9}k$$

$$rot = \frac{1}{18}\begin{pmatrix} 19 + 3\sqrt{3} \\ 19 - 3\sqrt{3} \\ 14 \end{pmatrix}$$

Rotations

- WHY!?!
- Why is this complicated computation necessary?

Rotations

- Imagine you rotate the objects continually (for example during exploration)
- This means the current rotation matrix is multiplied with another rotation matrix and so on:

$$Q_{cur} = Q_1 \cdot Q_2 \cdot \ldots \cdot Q_n$$

- Due to numerical issues the rotation matrix may be not orthogonal at the end, resulting in a weird behavior

Rotations

- What could you do?
- Probably fix the matrix, but how?
- Normalizing the columns may not result in an orthogonal matrix
- At the end it is not trivial to fix the matrix

Rotations

- Another application might be to interpolate between two rotation matrices
- Linear interpolation of two rotation matrices is mostly not a rotation matrix anymore

Rotations

- Using quaternions makes it easy to fix these problems
- It is easy to fix a quaternion such that it is a proper rotation again
- Two quaternions can be linearly interpolated after normalization, the interpolated rotation is good enough

Quaternions

- Composition of rotations corresponds to multiplication of quaternions
- Product of many orthogonal matrices may no longer be orthogonal, just as the product of many unit quaternions may no longer be a unit quaternion (limitations in precisions)
- Trivial to find the nearest unit quaternion, whereas it is quite difficult to find the nearest orthogonal matrix

(From: Berthold K. P. Horn, Closed-form solution of absolute orientation using unit quaternions)

Quaternions

- Finally some code…

```
#include <glm/gtc/quaternion.hpp>
…

glm::quat rot = glm::angleAxis(glm::radians(45.f), glm::vec3(0.f, 0.f, 1.f));
…
trans=glm::mat4_cast(rot);
```

- Define a quaternion with an angle and a rotation axis
- Perform calculations
- Cast it back to a 4x4 matrix that can be used for our purposes

Questions???