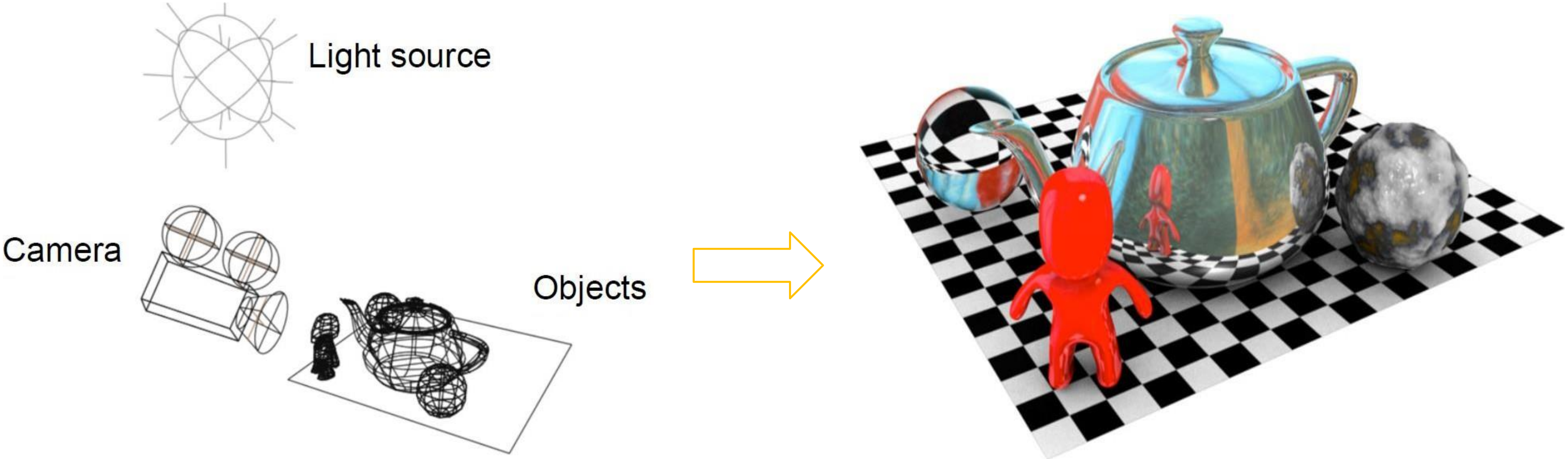
Computer Graphics

2. Introduction to OpenGL

Dr Joan Llobera – joanllobera@enti.cat

Spring 2019


Light source
Camera
Objects

Outline

1. What is OpenGL? History
2. What is OpenGL? Common Features
3. Programmable stages in OpenGL
4. GLSL: the OpenGL Shading Language
5. Transformation Operations

1. What is OpenGL? History

- A specification (Open Graphics Library)
- Cross-language, multi-platform Application Programming Interface (API)
- Used to interact with a Graphical Processing Unit
- Designed to be implemented mostly in hardware (although it can be implemented entirely in software)

Benefits

- Easy to use
- Close to hardware
  → Performance
- Focus on rendering
- No windows or input
  → No system dependencies

1. What is OpenGL? History

Stakeholders in the OpenGL evolution

Silicon Graphic (1982)

- **Silicon Graphics** (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware
- GL (“**Graphics library**”) allowed programmers to program three dimensional interactive applications in relatively simple way

OpenGL Architecture Review Board (1992)

- OpenGL **became a standard f**or controlling graphics accelerators in PCs
- GLQuake and 3dfx's Voodoo graphics accelerators pushed 3d accelerators into the **mainstream**
- **OpenGL ARB** is in charge of maintain and expand the speciffication (3Dlabs, Apple, ATI, Dell, IBM, Intel, Microsoft, NVIDIA, SGI and SunMicrosystems)

The Khronos group (2006)

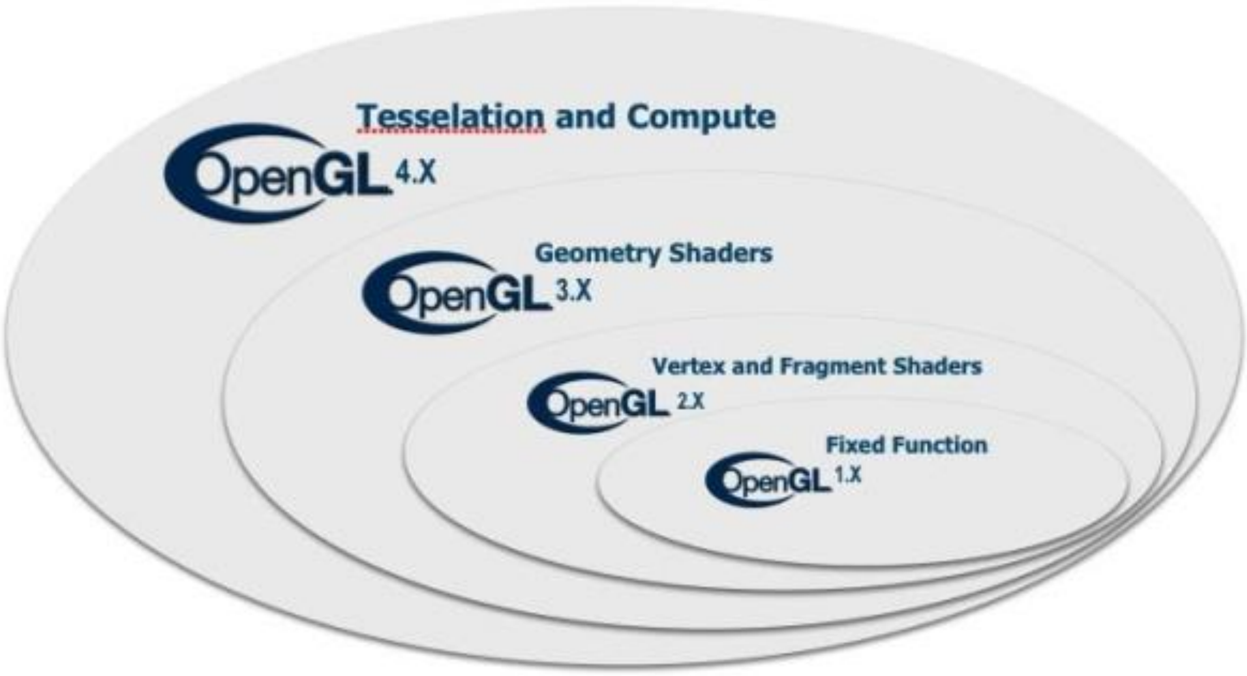
- It is a **not for profit industry consortium** creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on any platform and device
- It taken stewardship of the OpenGL standard, updating it to support the features of **modern programmable GPUs**, pushing it into the **mobile** and online domains with **OpenGL ES** and **WebGL**

1. What is OpenGL? History

OpenGL 1 and 2 focused on "surface realism"

OpenGL 3 and 4 focus more on "shape realism"

OpenGL 4 brings many optimization functionality



1. What is OpenGL? History

OpenGL 1.0 (1992/01)

- 1.1 (1997/03)
- 1.2 (1998/03)
- 1.3 (2001/10)
- 1.4 (2002/07)
- 1.5 (2003/07)

- 3DLabs conceived this version to evolve a stagnating OpenGL
- Programmable pipeline is faced thanks to the OpenGL Shading Language (GLSL)

OpenGL 2.0 (2004/09)

- 2.1 (2006/07)

- OpenGL is almost totally redefined
- Introduces a deprecation mechanism to promote the programmable pipeline instead of the fixed pipeline
- Totally shader-based

OpenGL 3.0 (2008/08)

- 3.1 (2009/03)
- 3.2 (2009/08)
- 3.3 (2010/03)

OpenGL 4.0 (2010/03)

- 4.1 (2010/07)
- 4.2 (2011/08)
- 4.3 (2012/08)
- 4.4 (2013/07)
- 4.5 (2014/11)

- target hardware capable of dealing with Direct3D 11 (part of DirectX)
- Add tessellation, compute shaders

1. What is OpenGL? History

Other kinds of OpenGL: OpenGL ES (Embedded Systems)

- Designed for mobile devices
- Subset of OpenGL

OpenGL ES 2.0

- spec in 2007
- subset of OpenGL 2.0
- but no backward compatibility with OpenGL 1.X
- Basis of WebGL 1.0 (see next slide)

- OpenGL ES 3.0
  - Specification in 2012
  - Backward compatible with OpenGL ES 2.0
  - Fully compatible with OpenGL 4.3
  - Basis of WebGL 2.0 (see next slide)
- OpenGL ES 3.1
  - Specification in 2014
  - Compute shaders
  - Independent vertex and frag. Shaders
  - Indirect draw commands

KHRONOS GROUP

1. What is OpenGL? History

Other kinds of OpenGL: WebGL

WebGL 1.0
- Specification in 2011
- Derived from early canvas 3D
- Based on OpenGL ES 2.0
- It's a Javascript API
- Automatic memory management

WebGL 2.0
- spec finished in 2017
- Based on OpenGL ES 3.0
- Already some browsers implement it

It is increasingly supported, but:
- Dependencies on browser support
- Dependencies on hardware available

Other kinds of API for graphics hardware:

DirectX

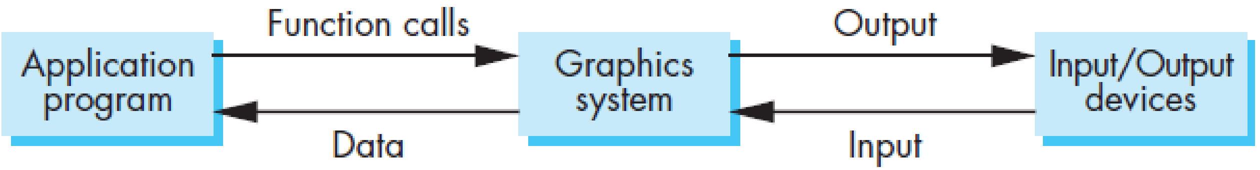
2. What is OpenGL? Common features

OpenGL is a **state machine**

It interacts with the application program and the input/output devices

The information in OpenGL can be used in **two different ways**

- **Primitive generating**
  - Can cause output if primitive is visible
  - How vertices are processed and appearance of primitive are controlled by the state
- **State changing**
  - Some variable values specify what and how the information is drawn
  - Variable values remain unchanged until we explicitly change them through functions that alter the state
  - For example, once we set a color, that color remains the current color until it is changed through a color-altering function



2. What is OpenGL? Common features

Different kinds of functions:

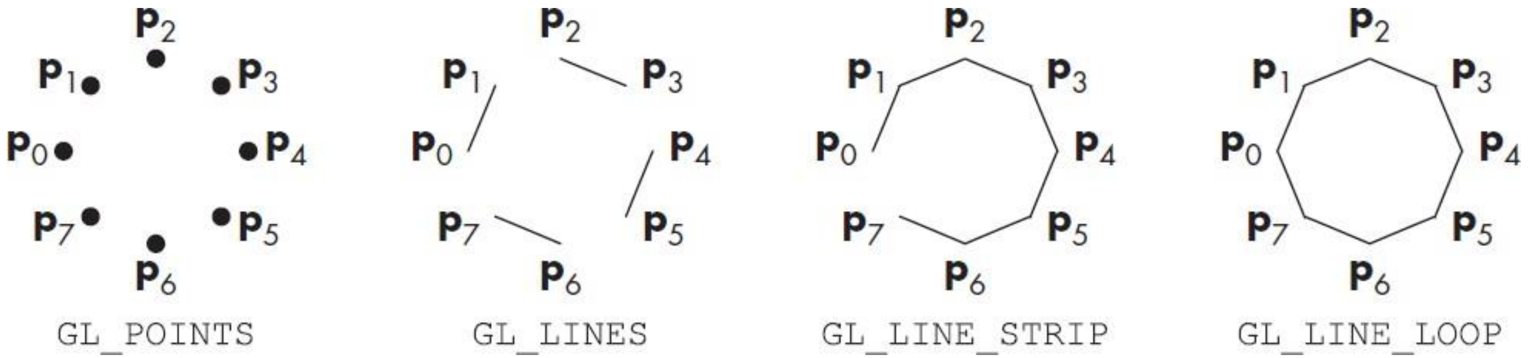
- The **primitive functions** define the atomic entities to display (points, line segments, polygons, ….)
- The **viewing functions** manage the camera's position and orientation
- The **transformations functions** allow us to rotate, translate, and scale objects
- The **attribute functions** define how the primitives are displayed (color, texture, depth)
- The **input functions** deal with keyboard, mouse, …
- The **control functions** enable us to communicate with the window system, to initialize our programs, and to deal with any errors that take place during the execution of our programs
- The **query functions** get information from devices

2. What is OpenGL? Common features

Primitives

- OpenGL supports three classes of geometric primitives: **points, line segments, and closed polygons** (in practice, mostly triangles)
  - Each **vertex** is associated with its **attributes** such as the position, color, normal, depth and texture
  - **Regular polyhedrons, quadrics, Bezier curves** and **surfaces** need libraries that simulate them based on the supported primitives

Circles, curves, surfaces and solids are non simple objects, and not supported by hardware (should they? Less portability of code, but simpler API)



2. What is OpenGL? Common features

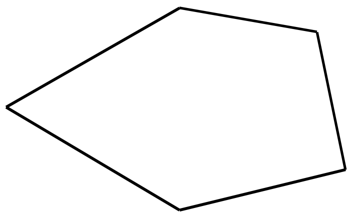
The name **polygon** is reserved for an object that has a border that canbe

described by a line loop but also has a **well-defined interior**

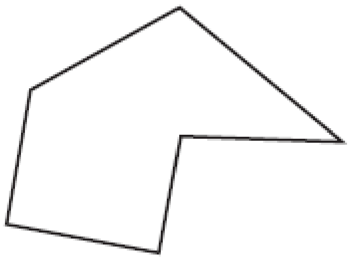
- **Well defined interior** means that the polygon is **convex** and it simplifies the whole rendering process because system can easily manipulate and fill that polygon

If the **polygon is not convex the final result will be unexpected** due to graphic systems assumes that all polygons defined by the user are convex

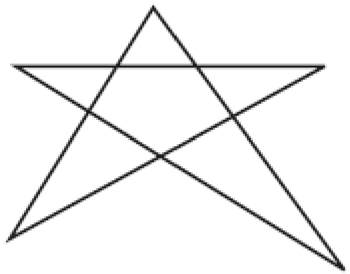
- **Non convex polygons** can be simulated using **tessellation** process
- **Tessellation** divides a polygon into a polygonal mesh, not all of which need be triangles. General tessellation algorithms are complex, especially when the initial polygon may contain holes



Convex polygon

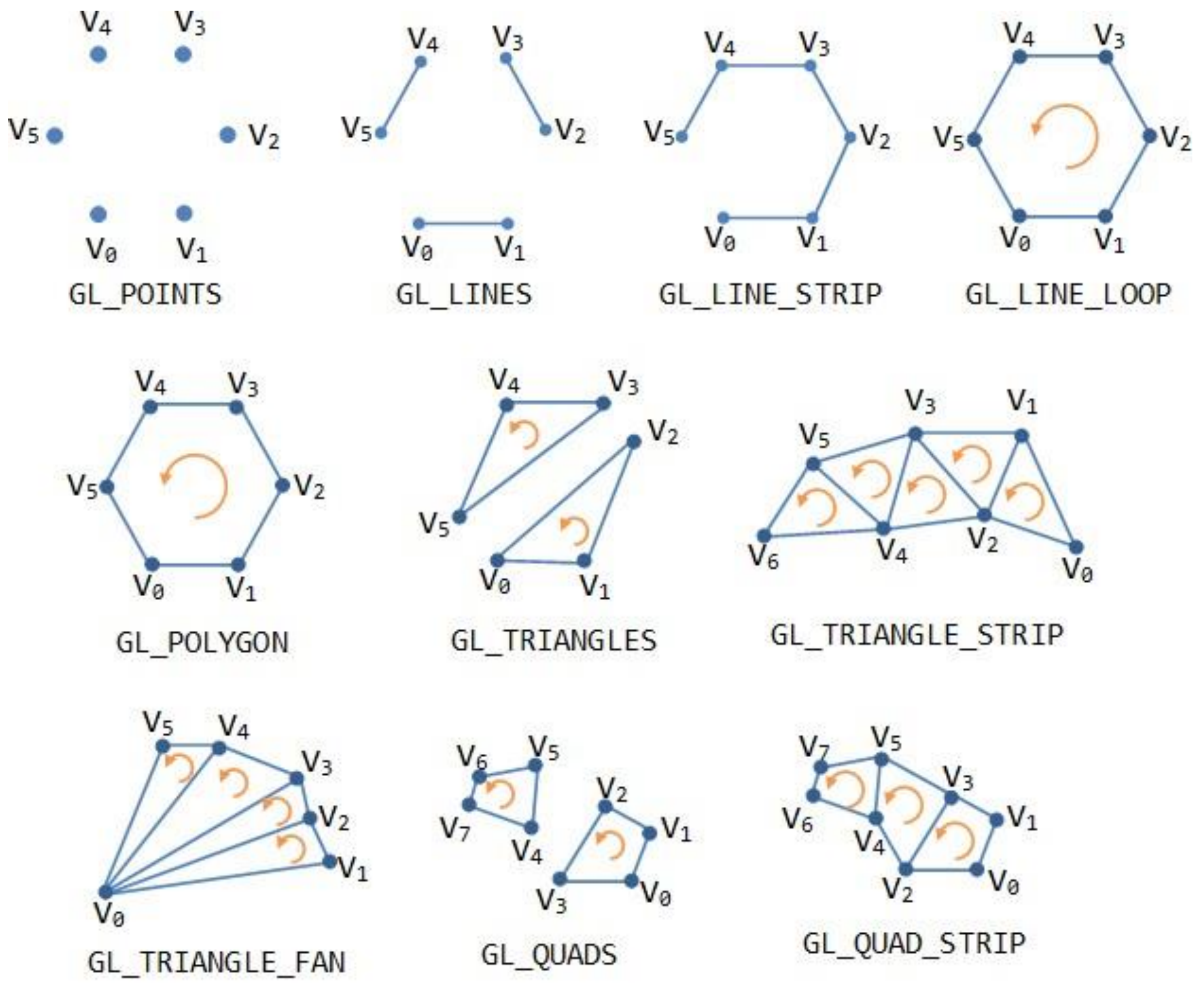


Non convex polygon



Non convex polygon composed by convex polygons

2. What is OpenGL? Common features

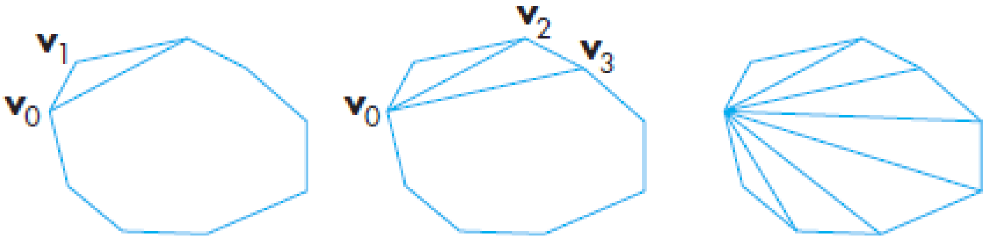


OpenGL Primitives

2. What is OpenGL? Common features

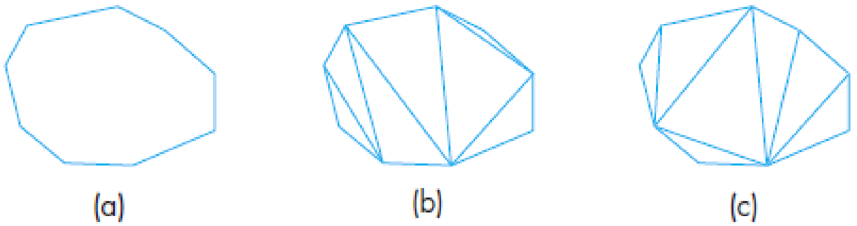
Triangulation is the process that generates a set of triangles consistent with the polygon defined

- Triangles are **easy to render** due to they are simple, flat and convex
- Triangles are the **only fillable geometric** entity that OpenGL recognizes

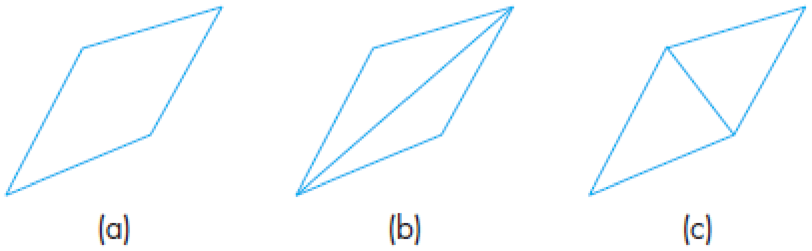


Recursive triangulation of a convex polygon.

Although the **result of the triangulation can be different**, it represent the same polygon



(a) Two-dimensional polygon. (b) A triangulation. (c) Another triangulation.



(a) Quadrilateral. (b) A triangulation. (c) Another triangulation.

2. What is OpenGL? Common features

And Text?

- In computer graphics is problematic
- OpenGL does not have a text primitive
- But stroke and bitmap characters can be created from other primitives

There exist two approaches to text:

Stroke text

- It uses vertices to specify line segments or curves that outline each character.
- **Advantage**: It can be defined to have all the detail of any other object
- **Drawback**: Its creation can be complex and the font will take up memory and CPU

Raster text

- It defines characters as rectangles of bits that can be placed in the frame buffer
- **Advantage**: It is simple and fast
- **Drawback**: Transformations are limited

No text, no input, no windows… We need support libraries!

Source: https://vector-conversions.com/vectorizing/raster_vs_vector.html

2. What is OpenGL?

Utility libraries used:

SDL (simple Directmedia Library)

- Lightweight C library working as a wrapper for input and window functionality.

GLEW (OpenGL Extension Wrangler Library)

- It is a cross-platform library helps in querying and loading OpenGL Extensions
- In modern OpenGL, the API functions are determined at *run time*, not *compile time*
- GLEW will handle the run time loading of the OpenGL API

GLM (OpenGL Mathematics)

- It is a mathematics library that mainly handles vectors and matrices
- In modern OpenGL, we must do all of the math ourselves

2. What is OpenGL?

There are other support libraries commonly used

GLU (OpenGL Utility Library)

- Utility functions mainly focus on primitive rendering and mapping between screen- and world-coordinates (projection and camera)

Instead of SDL, we often use:

GLUT (OpenGL Utility Toolkit)

- A library of utilities for OpenGL, which primarily focuses on window definition, window control and monitoring of keyboard and mouse input

GLFW

- A lightweight open source and multi-platform library for creating windows with OpenGL contexts and receiving input and events

3. Programmable stages in OpenGL

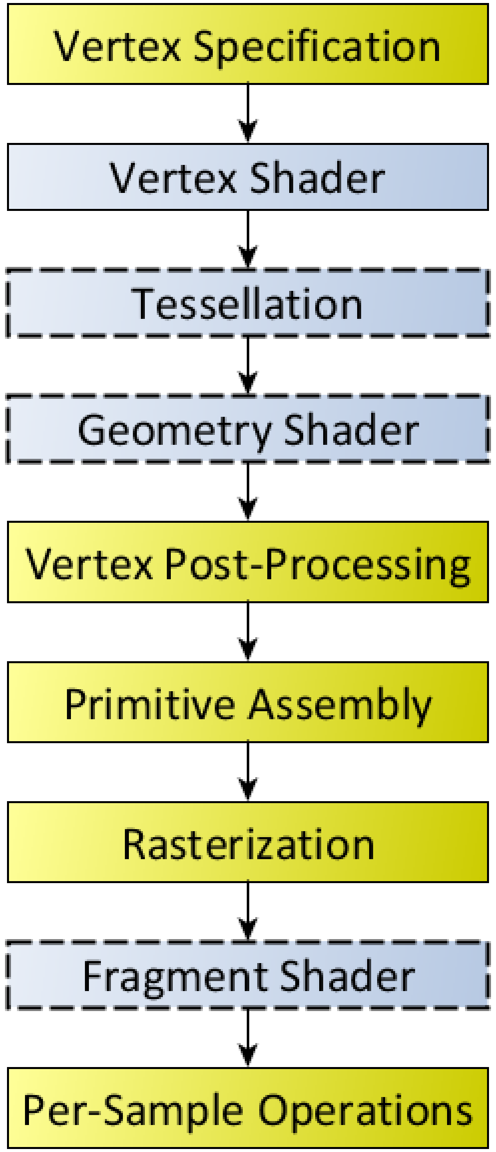
The host program fills OpenGL-managed memory buffers with arrays of vertices; these vertices are

1. projected into screen space,
2. assembled into triangles, and
3. rasterized into pixel-sized fragments;

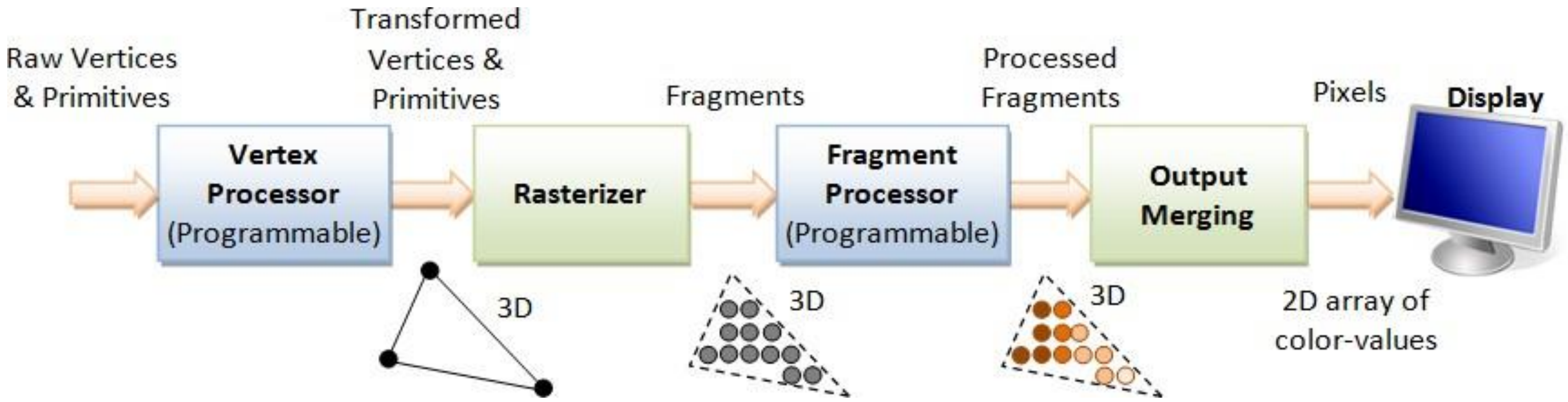
finally, the fragments are:

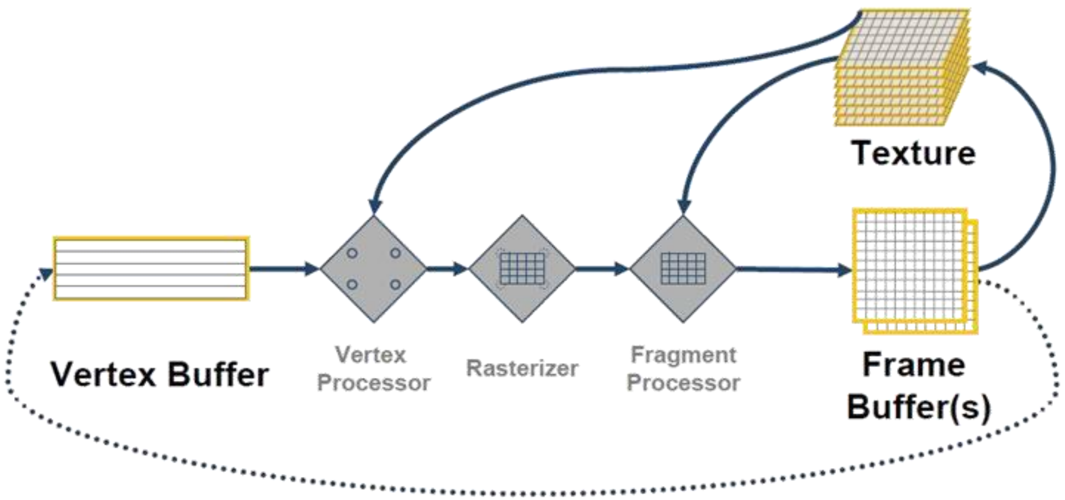
1. assigned color values and
2. drawn to the framebuffer.

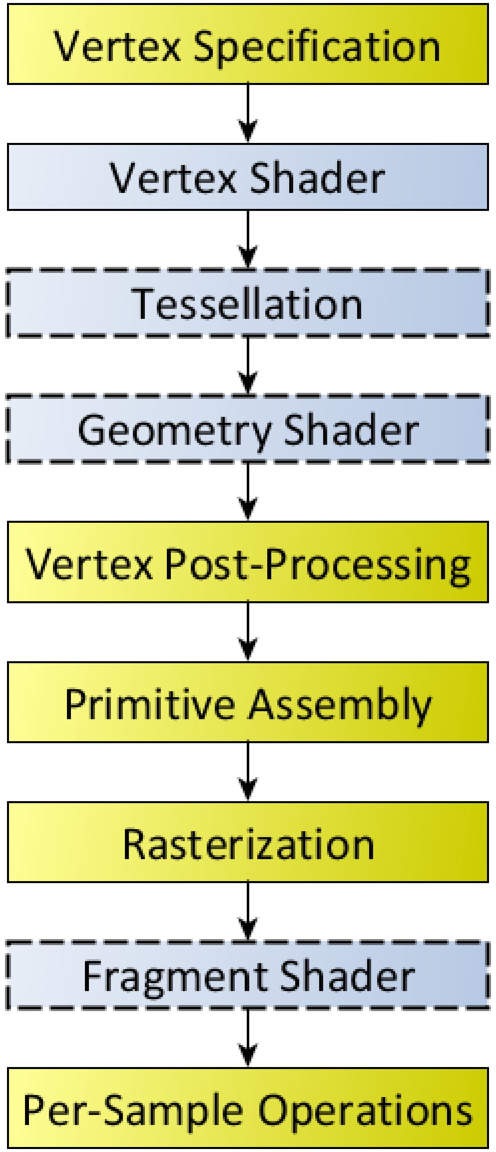
Modern GPUs get their flexibility by delegating the "project into screen space" and "assign color values" stages to uploadable programs called **shaders**



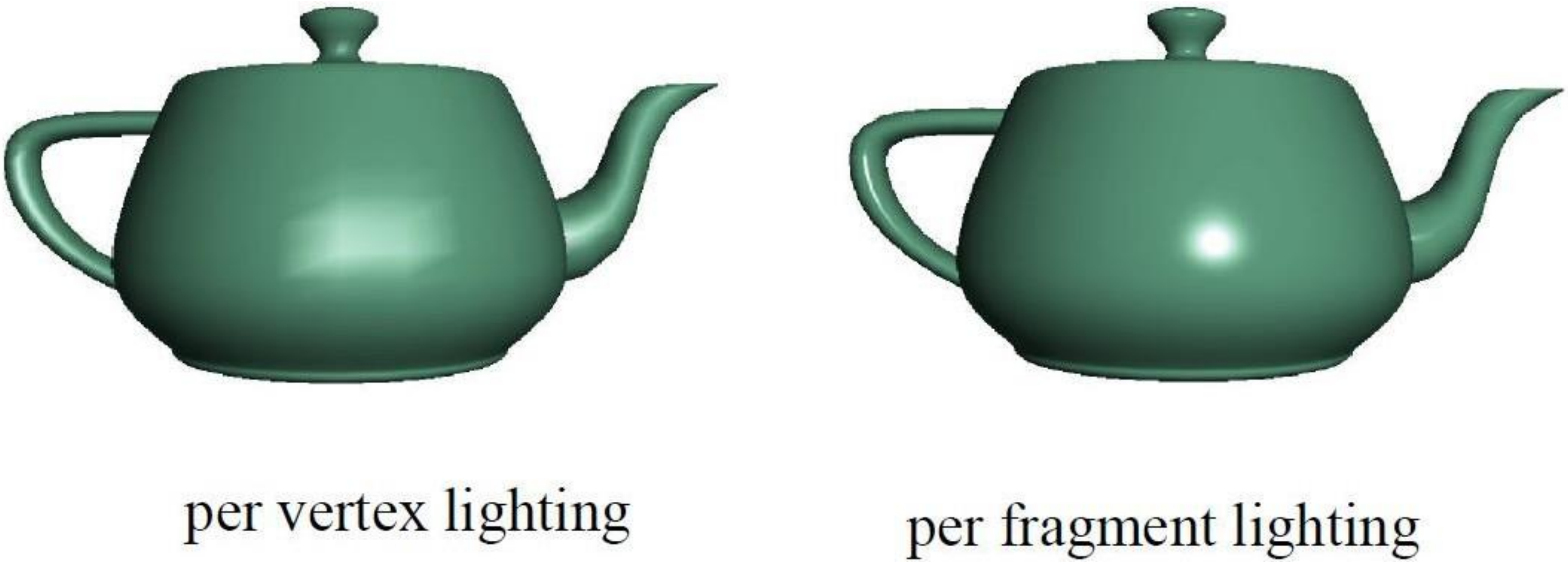
3. Programmable stages in OpenGL







3. Programmable stages in OpenGL



4. GLSL: the OpenGL Shading Language

OpenGL Shading Language (abbreviated: GLSL or GLslang), is a high-level shading language based on the syntax of the C programming language

It was **created by the OpenGL ARB** (OpenGL Architecture Review Board) to give developers more direct control of the graphics pipeline without having to use ARB assembly language or hardware-specific languages

Main **benefits**:

- **Cross-platform compatibility** on multiple operating systems, including GNU/Linux, Mac OS X and Windows
- The ability to write shaders that can be used on **any hardware vendor's**
- graphics card **that supports the OpenGL Shading Language**
- **Each hardware vendor** includes the GLSL compiler in their driver, thus allowing each vendor to create **code optimized** for their particular graphics card's architecture

4. GLSL: the OpenGL Shading Language

What do I need to know to program a shader?

GLSL shaders are not stand-alone applications

- They require an application that utilizes the OpenGL API, which is available on many different platforms
- There are language bindings for C, C++, C#, Java…

GLSL shaders themselves are simply a set of strings that are passed to the hardware vendor's driver

- They are compiled from within an application using the OpenGL API's entry points
- Shaders can be created on the fly from within an application, or read-in as text files, but must be sent to the driver in the form of a string

GLSL is similar to the C programming language

- It supports loops and branching (if-else, for, do-while, break, continue, …)
- User-defined functions are supported
- A wide variety of commonly used functions are provided built-in as well (exp(), abs()….)
- Others are specific to graphics programming, such as smoothstep() and texture()
- Recursion is forbidden
- Pointers are NOT allowed

4. GLSL: the OpenGL Shading Language

What do I need to know to program a shader?

```
#version version_number
in type in_variable_name;
in type in_variable_name;
out type out_variable_name;
uniform type uniform_name;
void main() {
   // Process input(s) and do graphics calculations
   ...
   // Output whatever was processed to an output
   variable
   out_variable_name = weird_stuff_we_processed;
```

4. GLSL: the OpenGL Shading Language

What do I need to know to program a shader?

All shaders have the same structure

- **#version** refers to the GLSL version used
- **in** refers to the input variables that come from the previous stage
  - From the OpenGL program (vertex attribute array and buffer objects)
  - From a previous shader
- **out** refers to the output variables that are sent to the next stage
- **uniform** refers to the variables managed from the OpenGL
- **void main()** is the starting point of the computes

4. GLSL: the OpenGL Shading Language

GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59 [1]	2.0	April 2004	#version 110
1.20.8[2]	2.1	September 2006	#version 120
1.30.10[3]	3.0	August 2008	#version 130
1.40.08[4]	3.1	March 2009	#version 140
1.50.11[5]	3.2	August 2009	#version 150
3.30.6[6]	3.3	February 2010	#version 330
4.00.9[7]	4.0	March 2010	#version 400
4.10.6[8]	4.1	July 2010	#version 410
4.20.11[9]	4.2	August 2011	#version 420
4.30.8[10]	4.3	August 2012	#version 430
4.40[11]	4.4	July 2013	#version 440
4.50[12]	4.5	August 2014	#version 450

4. GLSL: the OpenGL Shading Language

GLSL **has most of the default basic types** we know from languages like C:

- int,
- float,
- double,
- bool

GLSL also features three special container types:

- **vectors,**
- **matrices** and
- **samplers**

GLSL data type	C data type	Description
bool	int	Conditional type, taking on values of **true** or **false**.
int	int	Signed integer.
float	float	Single floating-point scalar.
vec2	float [2]	Two component floating-point vector.
vect3	float [3]	Three component floating-point vector.
vec4	float [4]	Four component floating-point vector.
bvec2	int [2]	Two component Boolean vector.
bvec3	int [3]	Three component Boolean vector.
bvec4	int [4]	Four component Boolean vector.
ivec2	int [2]	Two component signed integer vector.
ivec3	int [3]	Three component signed integer vector.
ivec4	int [4]	Four component signed integer vector.
mat2	float [4]	2×2 floating-point matrix.
mat3	float [9]	3×3 floating-point matrix.
mat4	float [16]	4×4 floating-point matrix.
sampler1D	int	Handle for accessing a 1D texture.
sampler2D	int	Handle for accessing a 2D texture.
sampler3D	int	Handle for accessing a 3D texture.
samplerCube	int	Handle for accessing a cubemap texture.
sampler1DShadow	int	Handle for accessing a 1D depth texture with comparison.
sampler2DShadow	int	Handle for accessing a 2D depth texture with comparison.

4. GLSL: the OpenGL Shading Language

Vertex shader directly sends the input information to the following stage

- The input data are the vertices that user has written in VBO and specified in VAO

Fragment shader always assigns the white color to any fragment

- Color is sent to the following stage through the finalColor output variable

Vertex shader

```
#version 150

in vec3 vert;

void main() {
    // does not alter the vertices at all
    gl_Position = vec4(vert, 1);
}
```

```
#version 150

out vec4 finalColor;

void main() {
    //set every drawn pixel to white
    finalColor = vec4(1.0, 1.0, 1.0, 1.0)
}
```

Fragment shader

4. GLSL: the OpenGL Shading Language

Uniform variables are variables that are constant for an entire primitive

- They an be changed in application and sent to shaders
- They cannot be changed in shader
- Examples of use:
  - To pass information to shader such as the time
  - A bounding box of a primitive

```
// in application
vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation( program, ”color" );
gl.uniform4f( colorLoc, color);
….
```

```
// in fragment shader (similar in vertex shader)
uniform vec4 color;
void main() {
        gl_FragColor = color;
}
```

4. GLSL: the OpenGL Shading Language

There are no pointers in GLSL

- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.
  - mat3 func(mat3 a)
- variables passed by copying

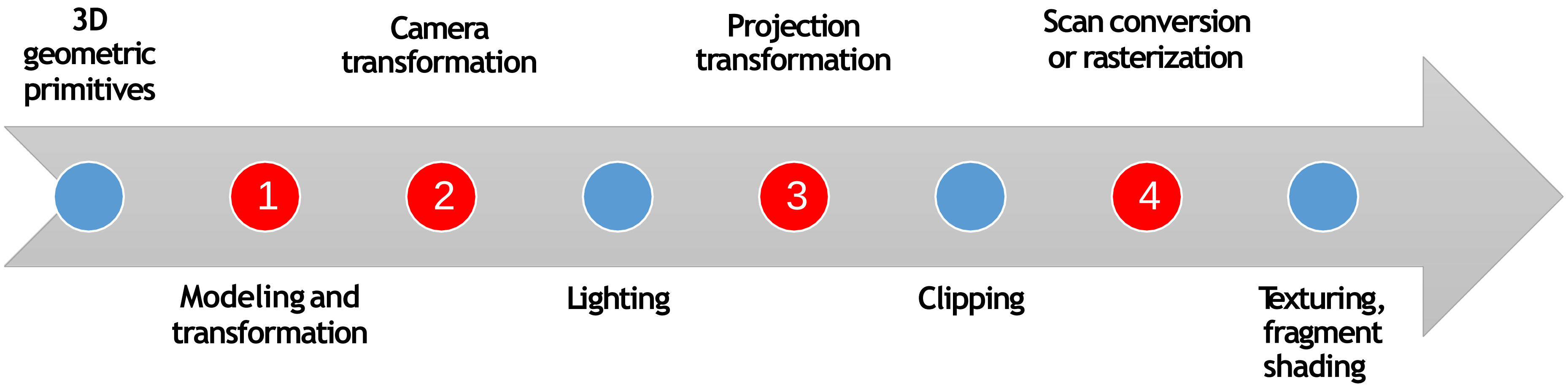
We Can refer to array elements by element using [] or selection (.) operator with

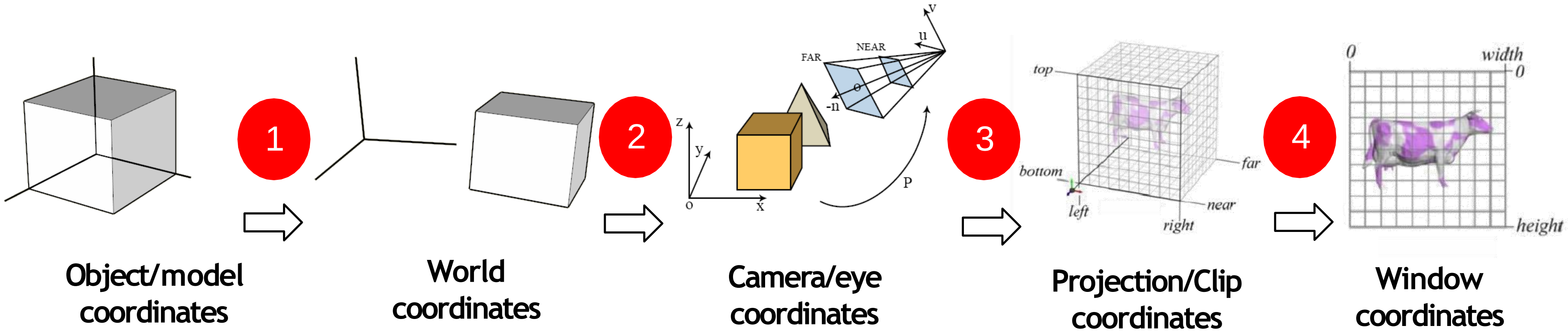
- x, y, z, w
- r, g, b, a
- s, t, p, q
- **a[2], a.b, a.z, a.p** are different ways of doing the same: Access to the 2nd element

Swizzling operator lets us manipulate components. Examples:

- vec4 a, b, c;
- c = vec4(1.0, 2.0, 3.0, 4.0)
- a.yz = vec2(1.0, 2.0, 3.0, 4.0); ⇒ a=(2.0, 3.0)
- b = c.yxzw; ⇒ b=(2.0,1.0, 3.0, 4.0)

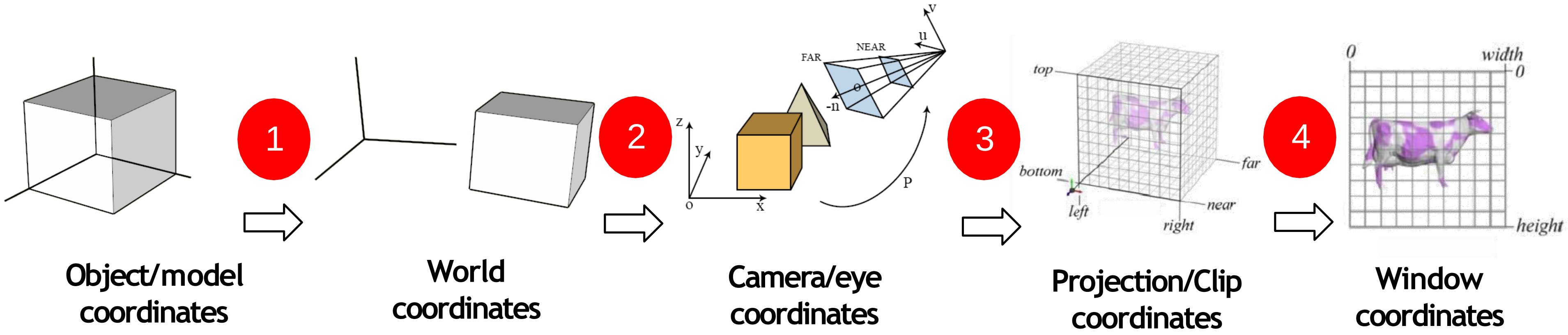
5. Transformation operations





5. Transformation operations

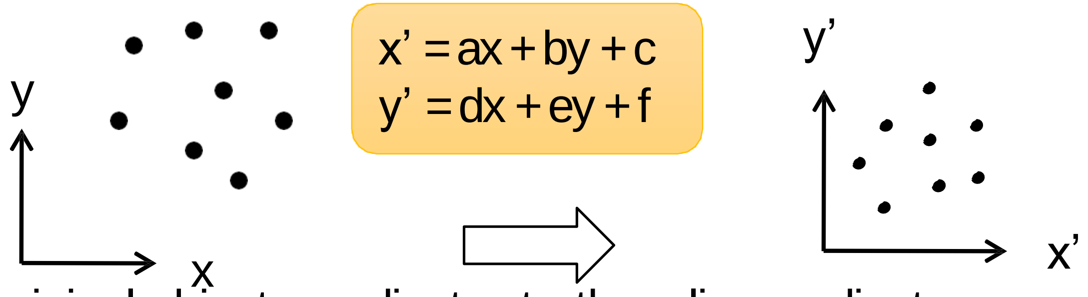
1. **Model transformation.** It transforms a position in a model to the position in the world
2. **View transformation.** The camera in OpenGL cannot move and is defined to be located at (0,0,0) facing the negative Z direction. That means that instead of moving and rotating the camera, the world is moved and rotated around the camera to construct the appropriate view
3. **Projection transformation.** It projects the information to clip coordinates that are in normalized devices coordinates
4. **Viewport transformation.** It adapts the image to the window resolution



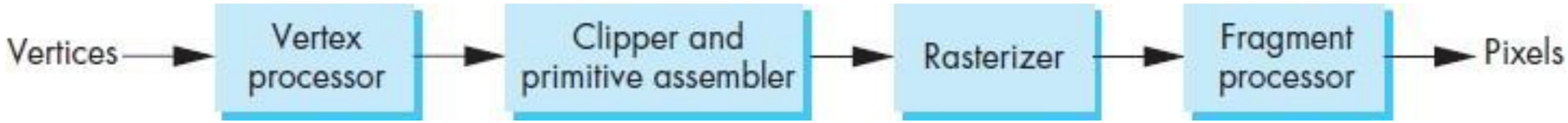
5. Transformation operations

What is a transformation?

A **transformation** is an operation that converts a set of points relative to a coordinate system to another coordinate system to achieve a targeted effect

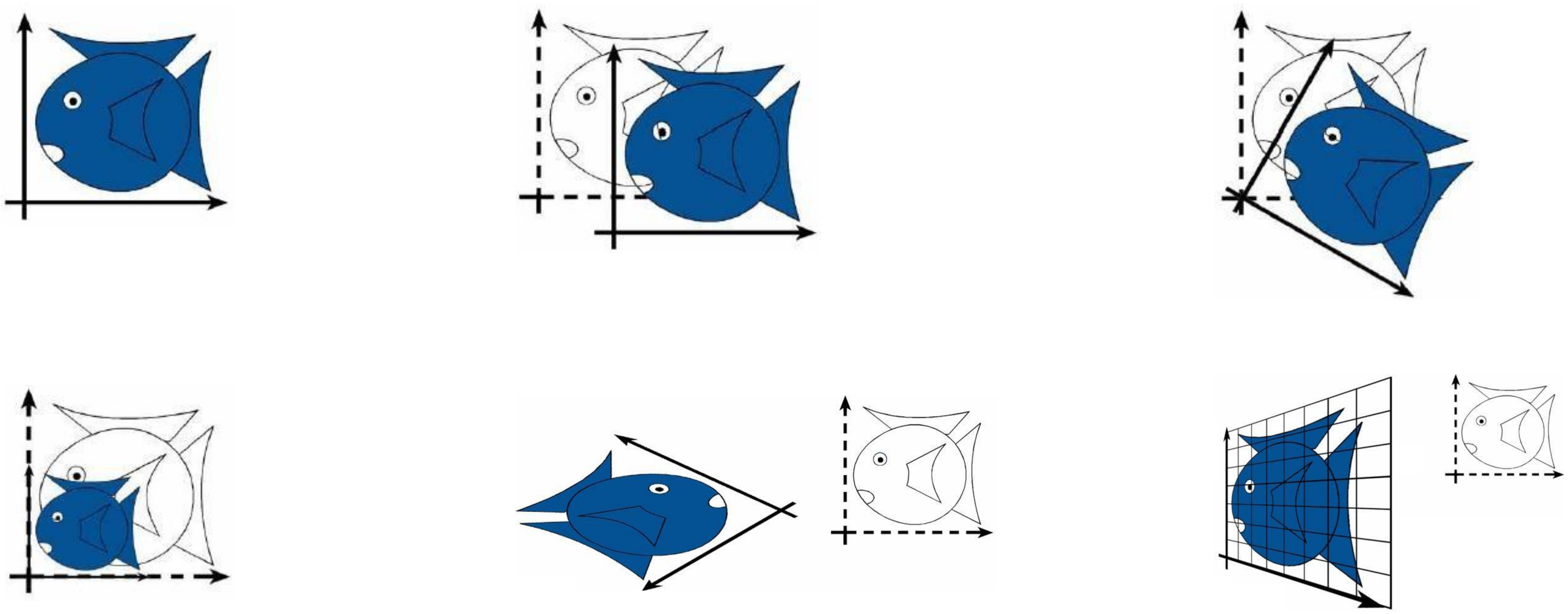


- **Transformations** are crucial for converting the original object coordinates to the clip coordinates
- All the transformation operations are manipulating vertices so, they 'play' in the **vertex processor**



ESCOLA DE NOVES TECNOLOGIES INTERACTIVES
UNIVERSITAT DE BARCELONA

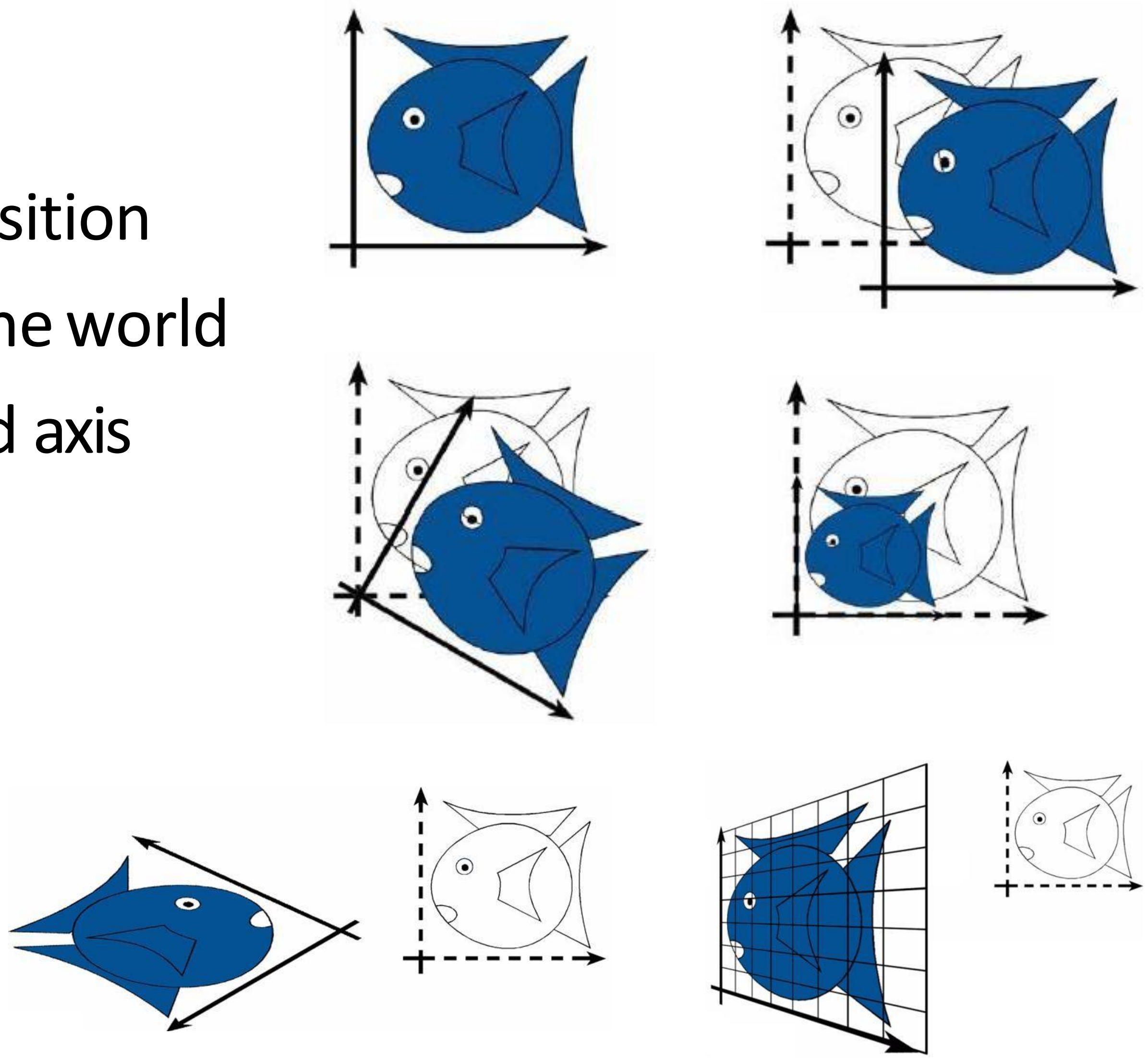
5. Transformation operations



5. Transformation operations

Examples of transformations

1. Do not alter the original object position
2. Placing an object at a position of the world
3. Rotating an object along a specified axis
4. Changing the shape of an object
5. Shearing the shape of an object
6. Project the object into the screen



5. Transformation operations

The Cartesian system

A transformation changes a point from a coordinate system to another

For example, a 2D transformation can be represented as:

$$x' = ax + by + c$$

$$y' = dx + ey + f$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{bmatrix} c \\ f \end{bmatrix}$$

$$p' = Mp + t$$

This operations is NOT optimal because a CPU/GPU needs to compute **two matrices operations**

5. Transformation operations

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

$$p' = \color{red}{M}\, p$$

The Homogeneous system

- It's an **extension of the Cartesian coordinate system** that allows to reduce the cost of applying transformations
- Basically, it introduces an additional dimension often called 'w'
  - If w=0, the element is representing a direction vector
  - If w=1, the element is representing a point. If w>1 it means that the element has been scaled.

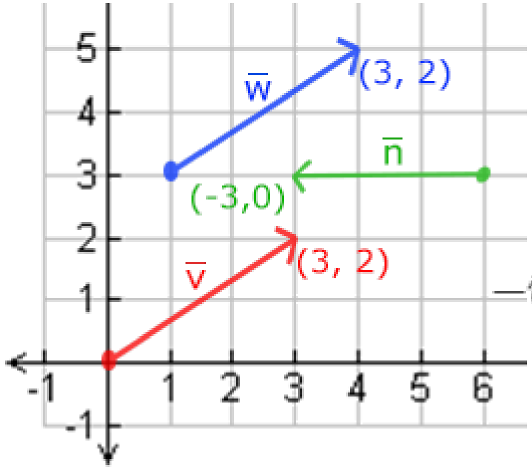
Examples:

- Point **P** is **(x, y, z) in** cartesian and **(x, y, z, 1)** in homogeneous Point **P** is **(x, y, z, w) in** homogeneous and **(x/w, y/w, z/w)** in Cartesian
- Vector **v is (a, b, c)** in cartesian and **(a, b, c, 0)** in homogeneous

- Thus, the transformation operation only requires **one matrix operation**

5. Transformation operations

Vector operations

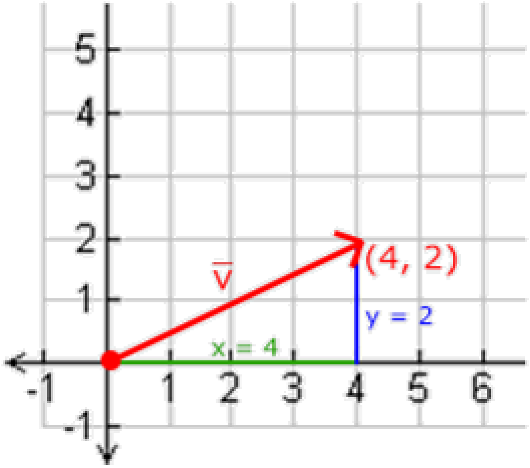


$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x = \begin{pmatrix} 1+x \\ 2+x \\ 3+x \end{pmatrix}$$

Operators combining scalar and vector

$$-\bar{v} = -\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} -v_x \\ -v_y \\ -v_z \end{pmatrix}$$

Vector negation

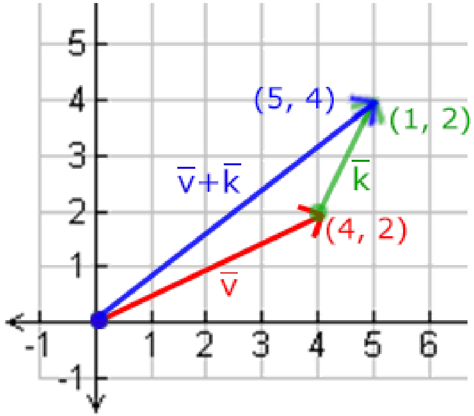


Length

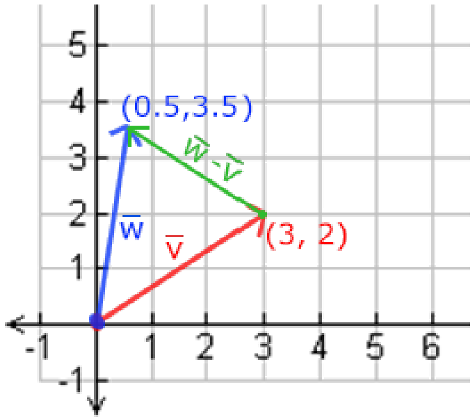
$$||\bar{v}|| = \sqrt{x^2 + y^2}$$

$$||\bar{v}|| = \sqrt{4^2 + 2^2} = \sqrt{16 + 4} = \sqrt{20} = 4.47$$

5. Transformation operations



$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + \bar{k} = \begin{pmatrix} 1+4 \\ 2+5 \\ 3+6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$



$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + -\bar{k} = \begin{pmatrix} 1+(-4) \\ 2+(-5) \\ 3+(-6) \end{pmatrix} = \begin{pmatrix} -3 \\ -3 \\ -3 \end{pmatrix}$$

5. Transformation operations

The **dot product** of two vectors is equal to the scalar product of their lengths times the cosine of the angle between them.

$$\bar{v} \cdot \bar{k} = ||\bar{v}|| \cdot ||\bar{k}|| \cdot cos\theta$$

If v and k are **unitary vectors**, so, the formula is reduced to:

$$\bar{v} \cdot \bar{k} = 1 \cdot 1 \cdot cos\theta = cos\theta$$

This allows us to easily test if the two vectors are **orthogonal** or **parallel** to each other using the dot product. You might remember that the cosine or cos function becomes:

- 0 when the angle is 90 degrees, or
- 1 when the angle is 0

The dot product proves very useful when doing **lighting calculations**

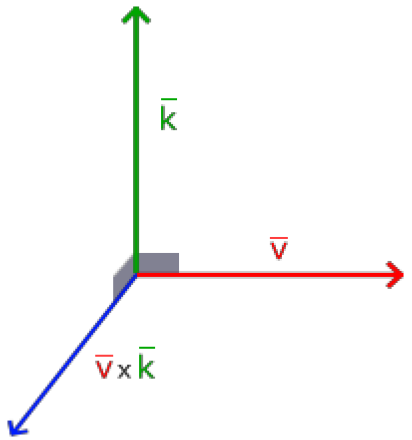
The dot product is a **component-wise multiplication where we add the results** together. To calculate the **degree** between both these unit vectors we use the inverse of the cosine function **cos−1**

$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8$$

5. Transformation operations

The cross product is only defined in 3D space and takes two non-parallel vectors as input and produces a third vector that is orthogonal to both the input vectors

If both the input vectors are orthogonal to each other as well, a cross product would result in 3 orthogonal vectors



$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$

5. Transformation operations

Matrix operations.

A matrix is basically a rectangular array of numbers, where each individual item in a matrix is called an element of the matrix. The number of rows and columns define the dimensions of the matrix

The addition or subtraction of two matrices requires that both have the same dimension

The matrix-matrix multiplication requires that the number of columns of the 1st matrix is equal to the number of rows of the 2nd matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix} \qquad 2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

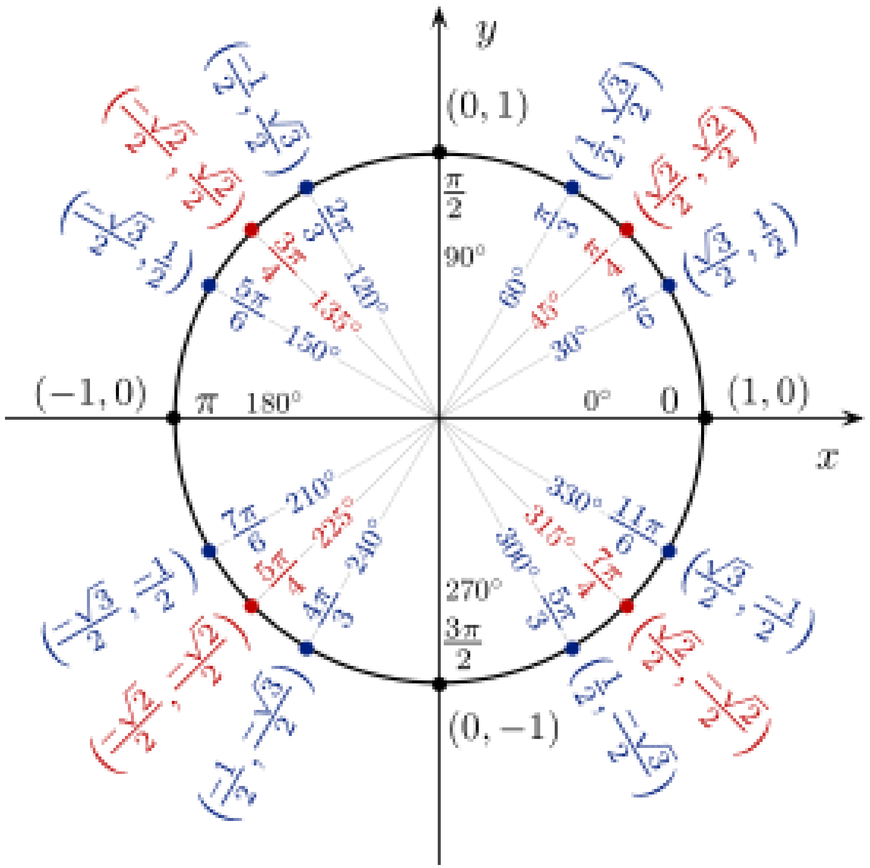
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} \boxed{1 \cdot 5 + 2 \cdot 7} & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & \boxed{3 \cdot 6 + 4 \cdot 8} \end{bmatrix} = \begin{bmatrix} \boxed{19} & 22 \\ 43 & \boxed{50} \end{bmatrix}$$

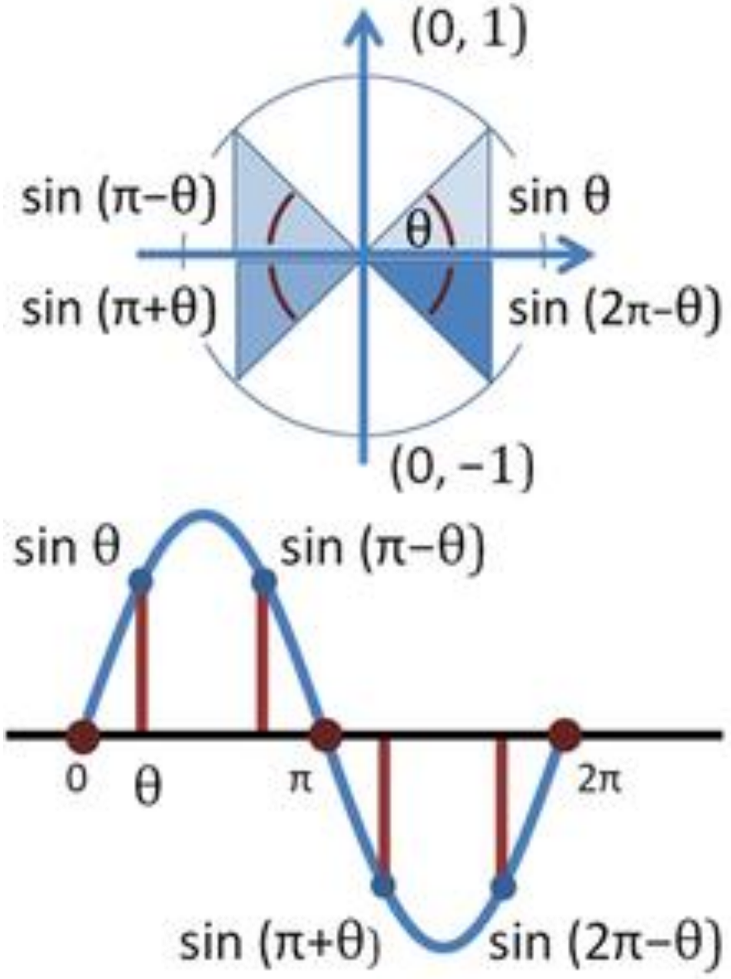
$$\begin{bmatrix} 4 & 2 & 0 \\ 0 & 8 & 1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 & 1 \\ 2 & 0 & 4 \\ 9 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 4 \cdot 4 + 2 \cdot 2 + 0 \cdot 9 & 4 \cdot 2 + 2 \cdot 0 + 0 \cdot 4 & 4 \cdot 1 + 2 \cdot 4 + 0 \cdot 2 \\ 0 \cdot 4 + 8 \cdot 2 + 1 \cdot 9 & 0 \cdot 2 + 8 \cdot 0 + 1 \cdot 4 & 0 \cdot 1 + 8 \cdot 4 + 1 \cdot 2 \\ 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 9 & 0 \cdot 2 + 1 \cdot 0 + 0 \cdot 4 & 0 \cdot 1 + 1 \cdot 4 + 0 \cdot 2 \end{bmatrix}$$

$$= \begin{bmatrix} 20 & 8 & 12 \\ 25 & 4 & 34 \\ 2 & 0 & 4 \end{bmatrix}$$

5. Transformation operations

Trigonometric relations





Source: https://en.wikipedia.org/wiki/Unit_circle

Resources

- [Angel2011] Edward Angel, Dave Shreiner (2011) *Interactive Computer Graphics: A Top-down Approach Using Opengl*, 6th Edition. Pearson education
- [GLSL, 2018] OpenGL Shading Language specification

https://www.khronos.org/registry/OpenGL/index_gl.php , (02/2018)

- [Khronos,2018] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview, (02/2018)
- [GLM, 2018] OpenGL Mathetmatics, http://glm.g-truc.net/, (02/2018)