

CS4052 Computer Graphics

01 Introduction	5
02a Graphics Programming	6
OpenGL (Open Graphics Library)	6
Graphics API Architecture	7
State Machine	7
Processing	7
Event-Driven Model	8
Primitives	8
Conventions	9
Drawing Process	9
Clearing the Window	9
Specifying a Colour	9
Complete Drawing the Scene	9
GLUT (GL Utility Toolkit)	10
GLUT Initialisation	11
GLUT Callback Registration	11
Sample Program - Triangles	12
Programmable Pipeline	14
Shaders	15
GLSL (OpenGL Shading Language)	15
Operators	15
Components	16
Qualifiers	16
Making a Shader Program	18
Compiling Shaders	18
Creating / Linking / Using Shaders	19
Vertex Buffer Objects	21
Working with Buffers	21
Loading the Buffer with Data	21
Link to Shader	23
Index Buffers	24
VAOs (Vertex Array Objects)	25
Uniforms	25
02b Graphics Pipeline	26
Programmable Pipeline	27
Application Stage	27
Geometry Stage	27
1) Model & View Transformation	28
2) Vertex Shading	28

CS4052 Computer Graphics

Rachel McDonnell

3) Projection	28
4) Clipping	29
5) Screen Mapping	29
Rasteriser Stage	30
1) Triangle Setup	30
2) Triangle Traversal	30
3) Pixel Shading	31
4) Merging	31
Z-Buffer	31
Double Buffering	32
OpenGL Coordinates	32
03a Geometric Transformations	33
Coordinate Systems	33
Conventions	34
Row vs. Column Formats	34
Vectors & Points	35
Computer Graphics Problems	35
Matrices	36
Geometric Transformations	37
Homogeneous Coordinates	37
Translation	38
Scaling	39
Rotation	40
Rotation About The z-Axis	41
Combining Rotation, Translation & Scaling	42
Homogeneous Coordinates	42
Affine Transformations	42
Transformation Composition I	43
Rotation About a Point	43
Transformation Composition II	43
Euler Angles	44
Rotational DOF (Degrees of Freedom)	44
OpenGL Uniforms	44
General Rotation	45
Rotation About An Arbitrary Axis	45
Rotation About An Axis	46
Aligning With The z-Axis	46
03b Linear Algebra for Graphics	47
Vector Addition	47
Negative Vectors	47
Vector Subtraction	47

CS4052 Computer Graphics
Rachel McDonnell

The Parallelogram Law	48
Vector Multiplication	48
Linear Combinations	49
Vector Magnitude	50
Normalised Vectors	50
Dot Product	51
Dot Product in Computer Graphics	53
Normals	54
Cross Product	55
Cross Product in Computer Graphics	56
Coordinate Systems	56
Change of Basis	57
Transformation	58
Affine Spaces	58
Normals & Polygons	59
04 Viewing	60
Transformation Pipeline	60
Camera Modelling	60
Model Matrix	61
3D to 2D Projection	61
Orthogonal Projection	62
Orthogonal Projection Matrix	64
Perspective Projection	64
Homogeneous Coordinates	65
Perspective Projections	65
Frustum	66
Perspective	67
Lens Configurations	68
Positioning The Camera	68
View Matrix	69
Positioning The Camera	69
Up Vector	70
LookAt	70
Viewport	70
Viewport to Window Transformation	71
Aspect Ratio	72
Multiple Projections	72
05 Hierarchical Modelling	73
Character Animation	73
Relative Motion	74
Degrees of Freedom	74

CS4052 Computer Graphics

Rachel McDonnell

Model Transformations	74
Hierarchical Transformations	75
OpenGL Implementation	76

01 Introduction

Email: Rachel.McDonnell@cs.tcd.ie

Website: <https://www.scss.tcd.ie/Rachel.McDonnell/IntroCompGraphics.shtml>

We will be using **OpenGL 3 & 4**:

- Don't read tutorials for older versions as they are no longer relevant.
- Can be written in C, but the object-oriented programming of C++ may be useful.

The coursework-exam split is 20:80%.

Assignments:

- Will be **demonstrated** and **graded** in labs.
- A **report** must be submitted, but will not count towards the grade.

There are three aspects of graphics:

1. **Modelling**: Representing a real object.
2. **Rendering**: Creating an image from a model on a display.
3. **Animation**: Changing the rendering over time.

Rasterisation is the process of displaying an image in pixels.

Modelling tools:

- [3ds Max](#) has a free version for academic use (recommended).
- [Blender](#) is open-source and has a large community.

This course follows very closely to [Learn OpenGL](#).

02a Graphics Programming

A **vertex** is a 3D point (x, y, z).

Can use (x, y, z, w) which allows vertices and points to be used together.

A triangle:

- Consists of **3 vertices**.
- Has a **normal**.

A **normal** is a vector perpendicular to the triangle which indicates the direction in which it faces. Vertices *can* have normals.

Rendering is the process by which a computer creates **images** from **models**.

Representing 3D models:

1. Modelling programs
2. Laser scanning
3. Procedural modelling / generation

A **graphics system** consists of:

1. Input devices
2. CPU (Central Processing Unit)
3. GPU (Graphics Processing Unit)
4. Memory
5. Frame buffer
6. Output devices

OpenGL (Open Graphics Library)

OpenGL:

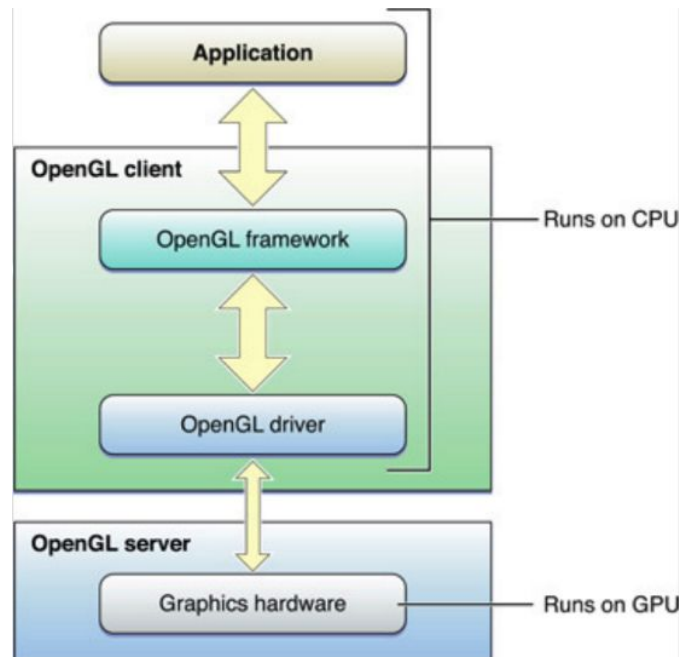
- Used for accessing and controlling the **graphics subsystem** of a device
- Device-independent
- Cross-platform
- Only used for **3D** graphics
- Has no platform specifics i.e. windowing, inputs
- C/C++ library

OpenGL uses a **client-server** model:

1. The **application** is the client.
2. The OpenGL on the **graphics card** is the server.

OpenGL behaves like a **state machine**.

Graphics API Architecture



The CPU:

1. Runs **setup** and the **rendering loop**.
2. Copies **mesh data** to **buffers** in the graphics hardware memory.
3. Writes **shaders** to draw on the GPU.
4. Queues drawing on the GPU with a particular shader and mesh data.

The CPU and GPU then run **asynchronously**.

State Machine

Various aspects of the state machine are set via the API such as:

- Colour
- Lighting
- Blending

When rendering, the **current state** of the state machine is used.

Most parameters are **persistent**, and must be explicitly changed.

Processing

Commands are always processed in the order in which they are received.

Each primitive is drawn **completely** before any subsequent command takes effect.

Event-Driven Model

OpenGL is **interactive** and **dynamic**, therefore we must handle interaction from the user.

The **GL library** is the core OpenGL system:

- Modelling
- Viewing
- Lighting
- Clipping

The **GLUT** (GL Utility Toolkit) provides the interface with the **windowing system**:

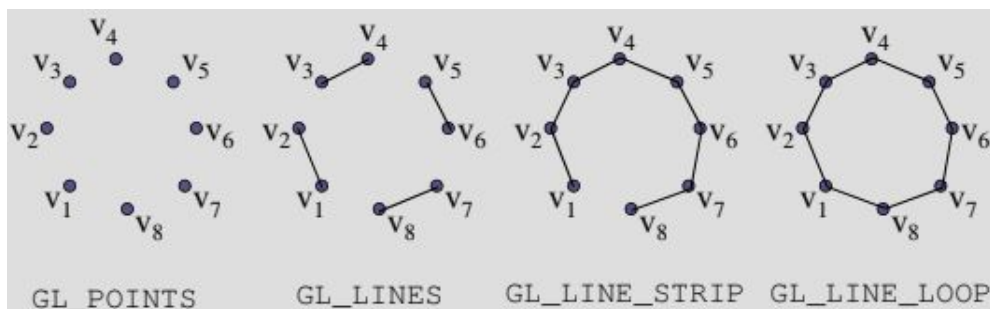
- Window management
- Menus
- Mouse interaction

Primitives

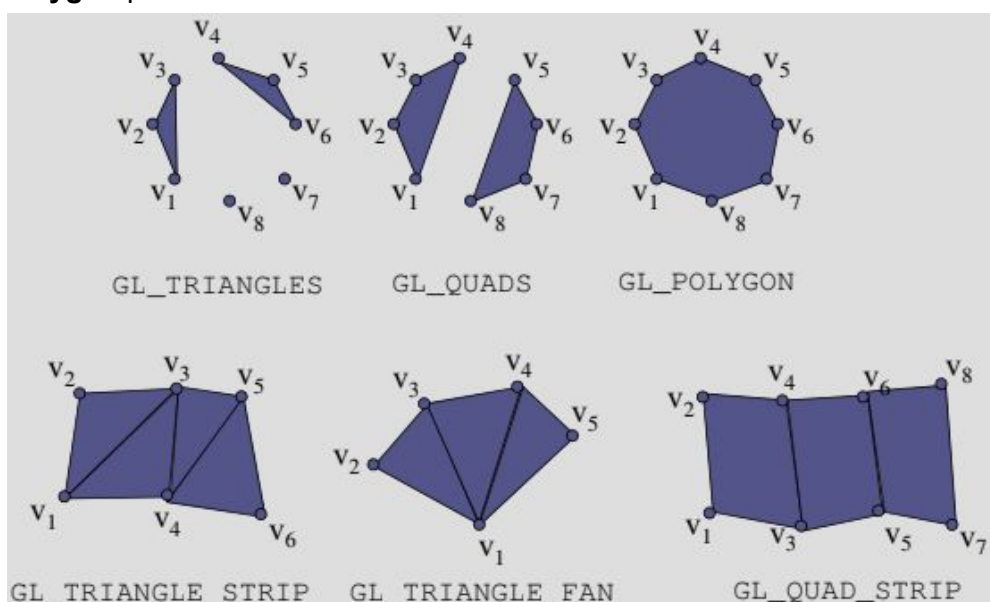
GPUs are optimised for points, lines and triangles.

All geometric objects in OpenGL are created from a set of **basic primitives**:

1. Line-based primitives:



2. Polygon primitives:



Conventions

Function names begin with `gl` or `glut`.

Constants begin with `GL_`, `GLU_`, or `GLUT_`.

Function names can encode **parameters** e.g. `glVertex2i(1, 3)`.

Drawing Process

Typical process:

1. Clear screen
2. Draw scene
3. Complete drawing
4. Swap buffers

In animation, there are usually **two buffers**:

1. Drawing occurs on the **background buffer**.
2. When complete, the background buffer is **swapped** with the front.
This gives a smooth animation without the user seeing the drawing.

The technique used to swap the buffers depends on which **windowing library** you use.

Clearing the Window

```
glClearColor(0.0, 0.0, 0.0, 0.0);  
glClear(GL_COLOR_BUFFER_BIT);
```

Typically you will clear the **color and depth buffers**:

```
glClearColor(0.0, 0.0, 0.0, 0.0);  
glClearDepth(0.0);  
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

You can also clear the **accumulation** and **stencil buffers** with `GL_ACCUM_BUFFER_BIT` and `GL_STENCIL_BUFFER_BIT`.

Specifying a Colour

Colours are specified in RGBA form (red, green, blue, alpha), in a range from 0 - 1.

Complete Drawing the Scene

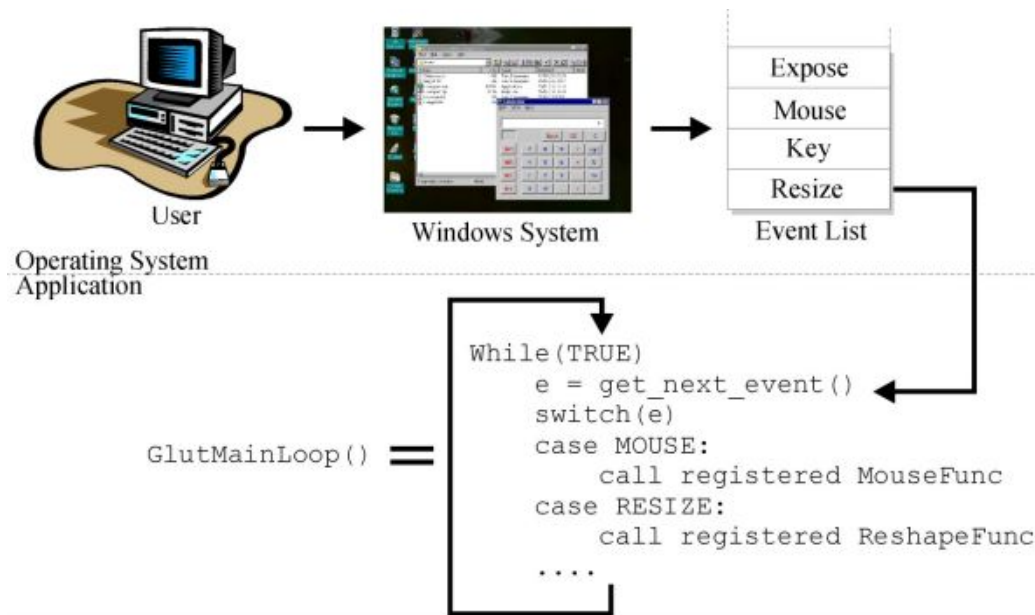
Either `glFinish()` or `glFlush()` can be used in OpenGL.

GLUT (GL Utility Toolkit)

Interaction with the user is handled through an **event loop**.

The application registers **handlers** / **callbacks** to be associated with particular events.

GLUT provides a **wrapper** on the X-Windows or Win32 core event loop, which manages event creation and passing.



To add handlers for events, we call a **callback-registering function** such as:

```
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));
```

This takes a **function** (callback) as a parameter.

Handlers must conform to the specification defined e.g.:

```
void key_handler(unsigned char key, int x, int y);  
glutKeyboardFunc(key_handler);
```

In this case, **key** is the ASCII code of the pressed key, and **(x,y)** is the mouse position within the window when the key was hit.

GLUT Initialisation

```
// Pass command-line arguments to GLUT system.
glutInit(&argc, argv);

// RGB colour, depth testing and double buffering.
glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);

// Set window title.
glutCreateWindow("RGSquare Application");

// Request for window size (not necessarily accepted).
glutReshapeWindow(400, 400);
```

GLUT Callback Registration

```
// Callback functions
void reshape(int w, int h) {...}
void keyhit(unsigned char c, int x, int y) {...}
void idle(void) {...}
void motion(int x, int y) {...}

// button = GLUT_UP or GLUT_DOWN.
// state = GLUT_LEFT_BUTTON or GLUT_MIDDLE_BUTTON or GLUT_RIGHT_BUTTON.
void mouse(int button, int state, int x, int y) {...}

// state = GLUT_NOT_VISIBLE or GLUT_VISIBLE
void visibility(int state) {...}

// value = timer ID
void timer(int value) {...}

// Register callbacks.
glutReshapeFunc(reshape); // On window resize.
glutKeyboardFunc(keyhit); // On key press.
glutIdleFunc(idle); // On system idle.
glutDisplayFunc(draw); // On paint window.
glutMotionFunc(motion); // On mouse move.
glutMouseFunc(mouse); // On mouse press.
glutVisibilityFunc(visibility); // On window deiconify.
glutTimerFunc(timer); // On timer elapse.
```

Sample Program - Triangles

```
int main(int argc, char** argv) {  
    // Create a window using GLUT.  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_RGBA);  
    glutInitWindowSize(512, 512);  
    glutInitContextVersion(4, 3);  
    glutInitContextProfile(GLUT_CORE_PROFILE);  
  
    // Create window with its name.  
    glutCreateWindow(argv[0]);  
  
    if (glewInit()) {  
        cerr << "Unable to initialise GLEW... exiting" << endl;  
        exit(EXIT_FAILURE);  
    }  
  
    init();  
  
    glutDisplayFunc(display);  
  
    // Start event loop.  
    glutMainLoop();  
}
```

```
void init(void) {
    // Setup object's initial position.
    glGenVertexArrays(NumVAOs, VAOs);
    glBindVertexArray(VAOs[Triangles]);

    GLfloat vertices[NumVertices][2] = {
        // Triangle 1.
        { -0.90, -0.90 },
        {  0.85, -0.90 },
        { -0.90,  0.85 },

        // Triangle 2.
        {  0.90, -0.85 },
        {  0.90,  0.90 },
        { -0.85,  0.90 },
    };

    glGenBuffers(NumBuffers, Buffers);
    glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
    glBufferData(
        GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW
    );

    // Specify shaders.
    ShaderInfo shaders[] = {
        { GL_VERTEX_SHADER, "triangles.vert" },
        { GL_FRAGMENT_SHADER, "triangles.frag" },
        { GL_NONE, NULL }
    };

    GLuint program = loadShaders(shaders);
    glUseProgram(program);

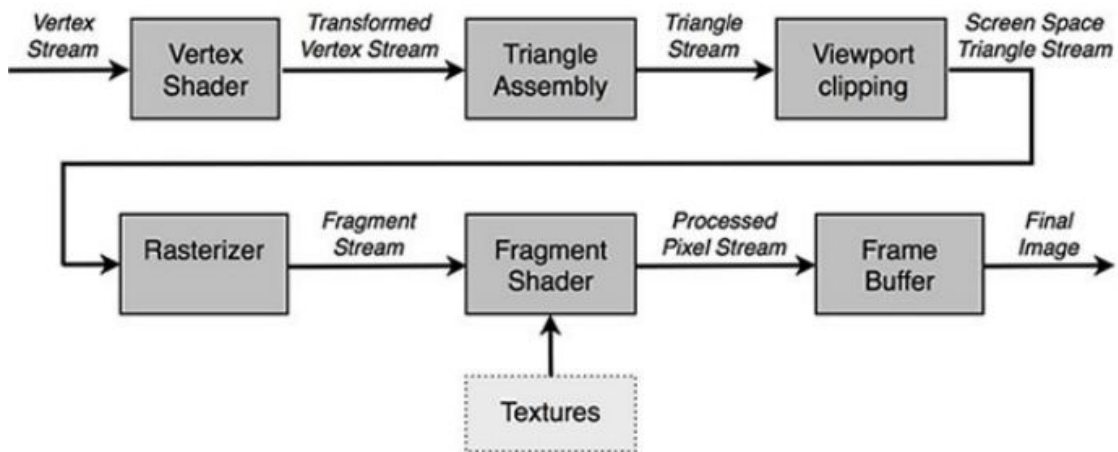
    glVertexAttribPointer(
        vPosition, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0)
    );
    glEnableVertexAttribArray(vPosition);
}
```

```
// Does the actual screen drawing.
void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);

    // Pick current vertex array.
    glBindVertexArray(VAO[Triangles]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);

    // Request image to be presented on screen.
    glFlush();
}
```

Programmable Pipeline



Shaders

OpenGL was fixed, but is now shader-based.

A **shader**:

- A program with `main` as its entry point.
- Source code is a **text file**.
- Written in Cg, HLSL and GLSL.

GLSL (OpenGL Shading Language)

GLSL:

- C-like language.
- Compiled into a program.
- We get back IDs which are `ints`.

Data types:

- Scalar types:
`float`, `int`, `bool`
- Vectors types:
`vec2`, `vec3`, `vec4`
- Matrix types:
`mat2`, `mat3`, `mat4`

Texture sampling:

`sampler1D`, `sampler2D`, `sampler3D`, `samplerCube`

C++ style constructors:

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

Operators

GLSL uses standard C/C++ arithmetic and logic operators.

These are overloaded for **matrix** and **vector** operations.

```
mat4 m;  
vec4 a, b, c;  
  
b = a * m;  
c = m * a;
```

Components

Vector components can be accessed using either:

1. Array indexing: `[]`
2. Named components: `xyzw`, `rgba` or `stpq`.

```
vec3 v;
```

`v[1]`, `v.y`, `v.g` and `v.t` all refer to the same element.

Qualifiers

Vertex attributes and other variable info can be copied in and out of shaders.

```
in vec2 textureCoord;  
out vec4 color;
```

The `uniform` keyword declares a shader **constant** variable.

```
uniform float time;  
uniform vec4 rotation;
```


There are two primary types of shader:

1. **Vertex** shader:
 - Changes the **position** of a vertex (trans / rot / skew).
 - *May* determine colour of the vertex.
2. **Fragment** shader:
 - Determines the **colour** of a **pixel**.
 - Uses lighting, materials, normals etc.

Vertex shaders:

- Process vertices.
- Must always output a **vertex location**.
- Usually transform vertices into **homogeneous clip space**.

The data describing what triangles are formed is **not** available to the vertex shader.

Vertex shader outputs become **pixel shader inputs**.

Fragment / pixel shaders:

- Can be passed a total of **16 vectors** from the vertex shader.
A **flag** could be used to determine which side of the triangle is visible.
- Can only influence the fragment handed to it:
 - ✗ Cannot send its results directly to neighbouring pixels.
 - ✗ Uses the data **interpolated** from the vertices along with stored constants and texture data.

There are other ways to affect neighbouring pixels.

[Shadertoy](#) is a useful WebGL tool for experimenting with shaders on the fly.

Making a Shader Program

Steps:

1. Compile a **vertex shader** \Rightarrow get ID.
2. Compile a **fragment shader** \Rightarrow get ID.
3. **Link** those two shaders together \Rightarrow get ID.

The final ID should be used before rendering triangles.

You can use **separate shaders for each model**.

Examples:

```
in vec4 vPosition;

void main() {
    // The value of vPosition should be between -1.0 and 1.0.
    gl_Position = vPosition;
}
```

```
out vec4 fColor;

void main() {
    // Colour the pixel red.
    fColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Compiling Shaders

`<GLuint> glCreateShader(<type>)`

- **Creates an ID** (a GLuint) of a shader.
- e.g. GLuint ID = `glCreateShader(GL_VERTEX_SHADER);`

`glShaderSource(<id>, <count>, <src code>, <lengths>)`

- **Binds the source code** to the shader.
- Occurs **before compilation**.

`glCompileShader(<id>)`

- Used to make the shader program.

Creating / Linking / Using Shaders

<GLuint> `glCreateProgram()`

- Returns an ID that should be stored for the life of the program.

`glAttachShader(<prog ID>, <shader ID>)`

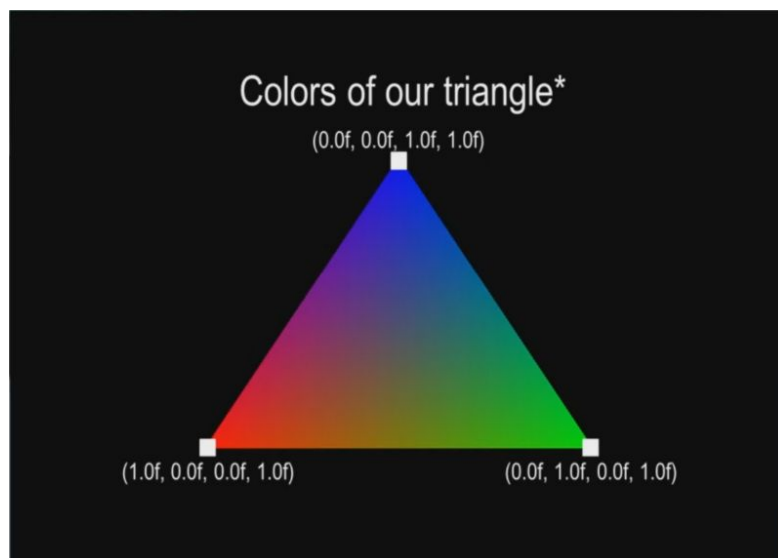
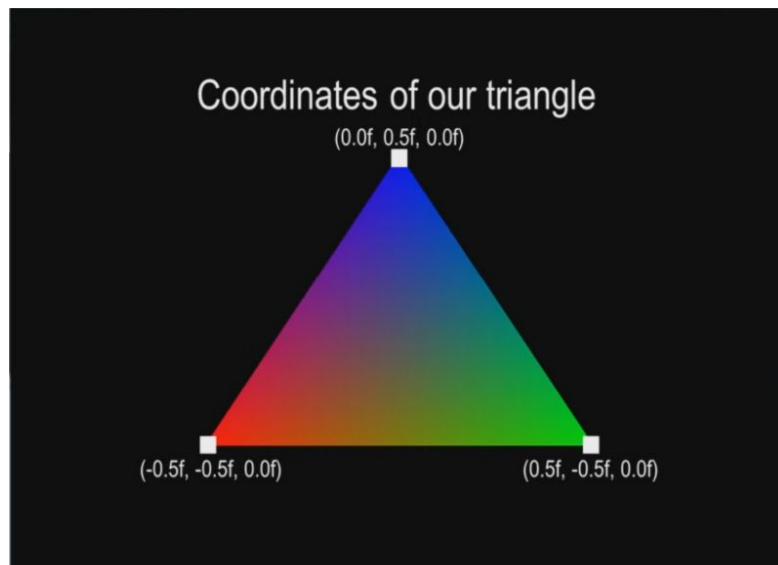
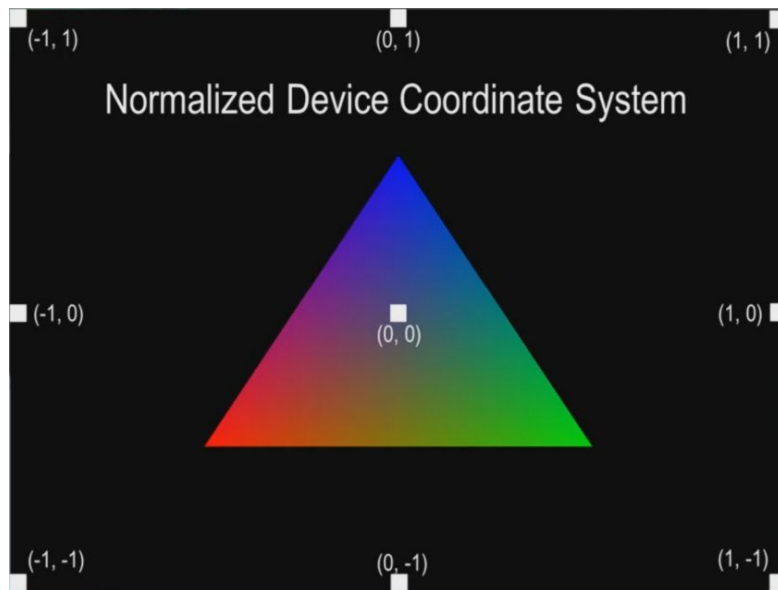
- Do this for both the **vertex** and **fragment shaders**.

`glLinkProgram(<prog ID>)`

- Actually makes the shader program.

`glUseProgram(<prog ID>)`

- Use this shader when you're about to draw triangles.



Vertex Buffer Objects

How do we get the geometry & colour onto the GPU?

Typically we also need a **normal** and **texture coordinate** for each vertex.

We ask the OpenGL driver to create a **buffer object**:

- Chunk of **memory** e.g. array.
- Located **on the GPU**.

Working with Buffers

To create a **buffer ID**:

```
// This will be the ID of the buffer.  
GLuint buffer;  
  
// Ask OpenGL to generate exactly 1 unique ID.  
glGenBuffers(1, &buffer);
```

To set the buffer as the **active** one and specify which buffer we're referring to:

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);
```

Notes:

- The buffer is now **bound** and **active**.
- Any **drawing** will come from that buffer.
- Any **loading** will go into that buffer.

Loading the Buffer with Data

We want to store an array data in the GPU:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
```

GL_<freq>_<mode>:

- <freq> = STREAM / STATIC / DYNAMIC (How frequently the data will change)
- <mode> = DRAW / READ / COPY

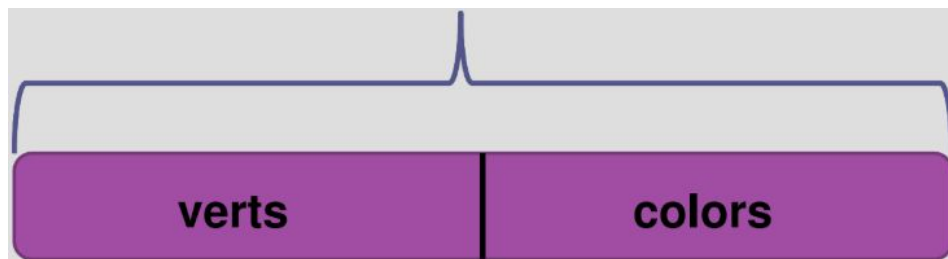
Process:

1. **Create the buffer** and pass **no data**.
2. Load the **geometry**.
3. Load the **colours**.
4. Load the **normals**, **texture coordinates**, ...

Buffers can be organised in any way.

```
// vertices = (x,y,x), colors = (r,g,b,a)
generateObjectBuffer(GLfloat vertices[], GLfloat colors[]) {
    // Vertex buffer object.
    GLuint VBO;

    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(
        GL_ARRAY_BUFFER,
        numVertices * 7 * sizeof(GLfloat), // 7 = verts + colors
        NULL,
        GL_STATIC_DRAW
    );
    glBufferSubData(
        GL_ARRAY_BUFFER,
        0,
        numVertices * 3 * sizeof(GLfloat),
        vertices
    );
    glBufferSubData(
        GL_ARRAY_BUFFER,
        numVertices * 3 * sizeof(GLfloat),
        numVertices * 4 * sizeof(GLfloat),
        colors
    );
}
```



At the moment we have a buffer which:

- Has an ID.
- Lives on the graphics card.
- Is full of vertex position / colour data.

Buffer information needs to be passed to our **shader**.

Link to Shader

```
in vec4 vPosition;  
in vec4 vColor;  
out vec4 color;  
  
void main() {  
    gl_Position = s_vPosition;  
    Color = vColor;  
}
```

We need to **query the shader program** for its variables:

```
// Get vPosition ID.  
GLuint vpos = glGetAttribLocation(programID, "vPosition");
```

In OpenGL, we must **enable attributes**:

```
// Turn on vPosition.  
glEnableVertexAttribArray(vpos);
```

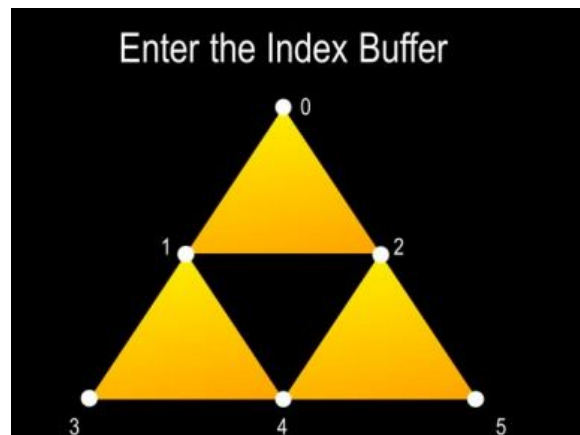
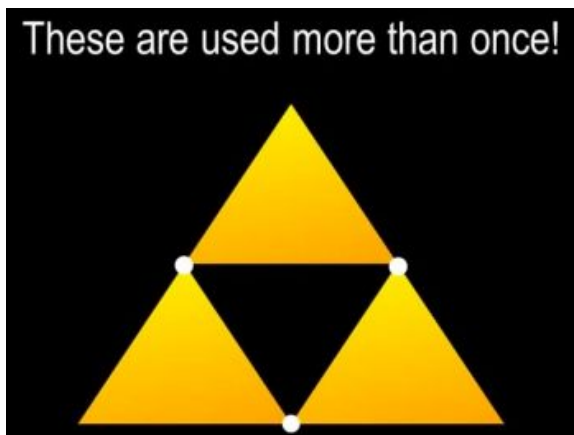
Tell variables where to find their info in the currently-bound buffer:

```
void glVertexAttribPointer(  
    GLuint index,  
    GLint size,  
    GLenum type,  
    GLboolean normalized,  
    GLsizei stride,  
    const GLvoid* offset  
);  
  
// Bind the variable to a spot in the buffer.  
glVertexAttribPointer(  
    vpos,  
    3,  
    GL_FLOAT,  
    GL_FALSE,  
    0,  
    0 // Where to find variable in buffer.  
);
```

```
cpos = glGetAttribLocation(programID, "vColor");
glEnableVertexAttribArray(cpos);
glVertexAttribPointer(
    cpos,
    4,
    GL_FLOAT,
    GL_FALSE,
    0,
    BUFFER_OFFSET(numVertices * 3 * sizeof(Glfloat))
);
```

Index Buffers

An index buffer allows you to create an object which reuses the same vertices.



```
GLuint ebo;
glGenBuffers(1, &ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, size, indices, GL_STATIC_DRAW);
```

To render the triangles, instead of using `glDrawArrays`, use:

```
glDrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_INT, NULL);
```


VAOs (Vertex Array Objects)

- Pure State: It remembers almost everything about buffers.
- Set it up once, then call it before drawing.
- `glVertexAttribPointer()`;
- Does **not** bind the VBO.

Creating a vertex array object:

```
// This will be the ID of the VAO.
GLuint vao;

// Ask the driver for exactly 1 unique ID.
glGenVertexArrays(1, &vao);

// Everything after this will be a part of the VAO.
glBindVertexArray(vao);
```

Uniforms

Uniform:

Data that is passed into a shader that **stays the same** e.g. a transformation matrix.

Get data directly from the application to shaders.

Two approaches:

1. Declare in default block.
2. Store in a buffer object.

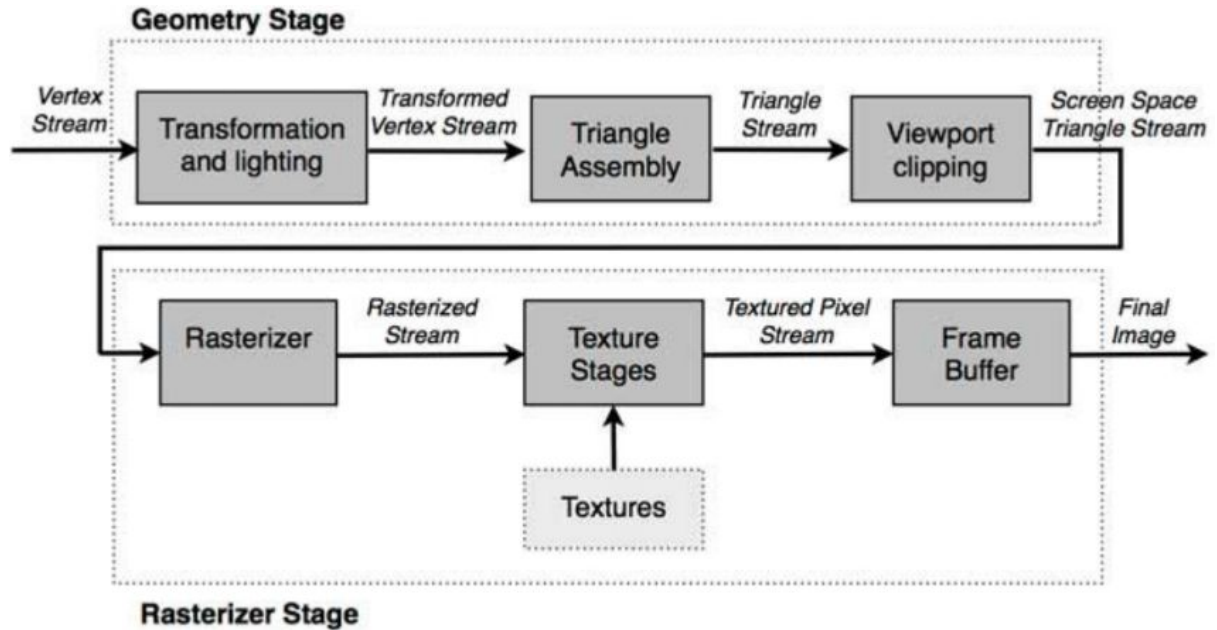
Simply place the keyword `uniform` at the beginning of the **variable definition**.

```
in vec4 vPosition; // The vertex in local coordinate system.
uniform mat4 mM; // The matrix for the pose of the model.
uniform mat4 mV; // The matrix for the pose of the camera.
uniform mat4 mP; // The projection matrix (perspective).

void main() {
    gl_Position = mP * mV * mM * vPosition;
}
```

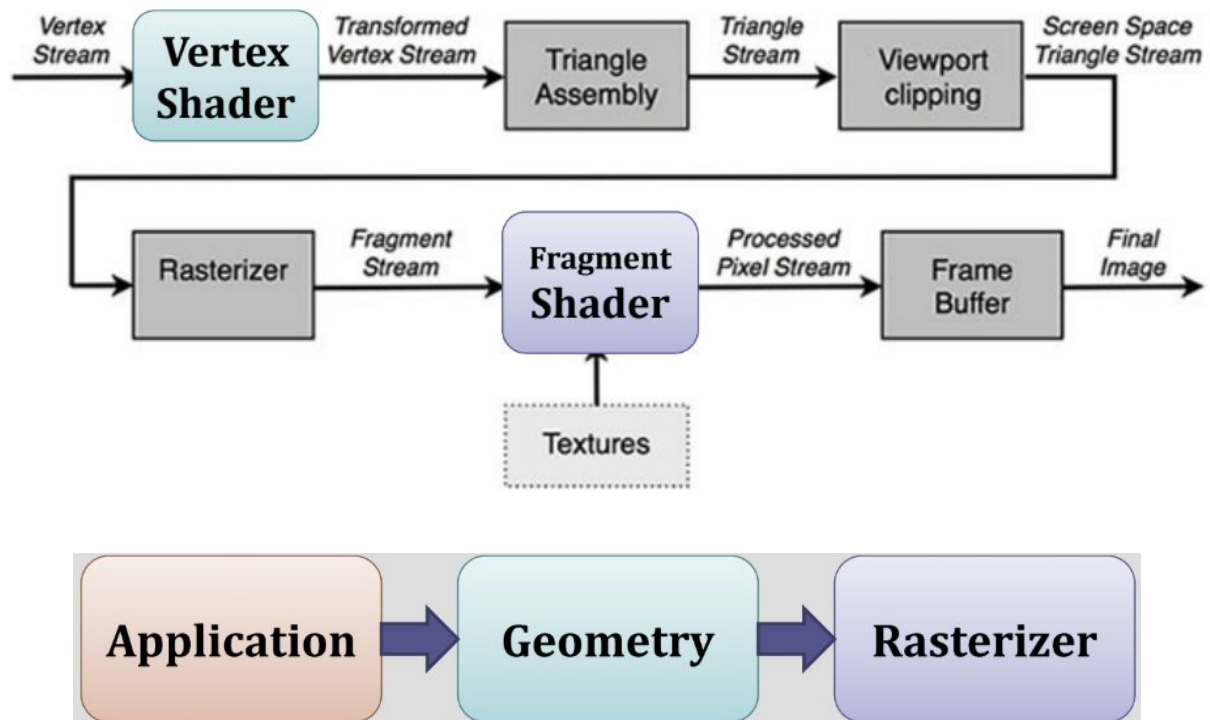
02b Graphics Pipeline

Old versions of OpenGL used a **fixed-function** pipeline:



However, in the newer versions of OpenGL, a **programmable** pipeline is used.

Programmable Pipeline



Each stage of the pipeline is a pipeline in itself.

The **slowest pipeline stage** determines the rendering speed (FPS).

Application Stage

The developer has **full control** of the application stage.

It is executed on the **CPU**.

At the end of the application stage, the rendering **primitives** are fed to the **geometry stage**.

Geometry Stage

This stage is responsible for the **per-polygon** and **per-vertex** operations.



The first three stages are implemented in the **vertex shader**:

1. Model & view transformation
2. Vertex shading
3. Projection

1) Model & View Transformation

Models are transformed into several spaces or **coordinate systems**.

Models initially reside in the **model space**.

Model transformation positions the object in the **world coordinates** or world space.

The **view transform** ???

2) Vertex Shading

Shading determines the effect of a **light** on a material.

A variety of **material data** can be stored at each vertex:

- Point location
- Normal
- Colour

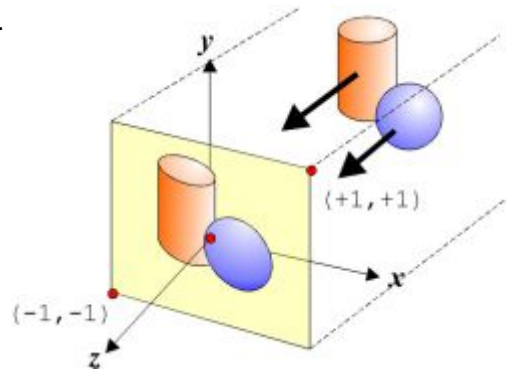
Vertex shading results (colours, vectors, texture coordinates) are sent to the **rasterisation stage** to be **interpolated**.

3) Projection

After shading, rendering systems perform **projection**.

Models are projected from three dimensions to two.

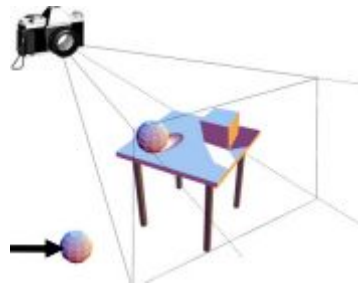
This uses **perspective** or **orthographic** viewing.



4) Clipping

Memory may contain models, textures and shader data for **all** objects in a scene.

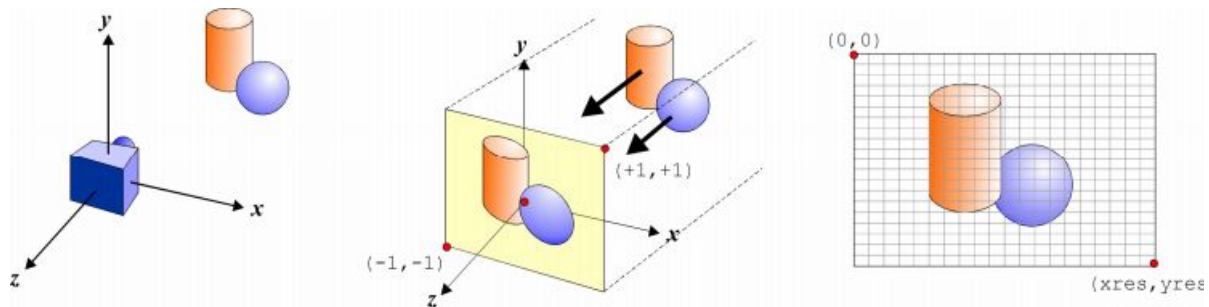
The virtual camera viewing the scene only sees the objects **within the field of view**.
The computer does not need to transform, texture or shade the objects outside of this.



A **clipping algorithm** skips these objects making rendering more efficient.

5) Screen Mapping

Only the **clipped** primitives inside the view volume are passed to this stage.

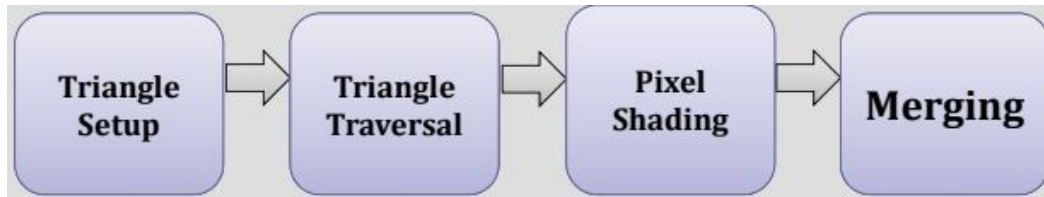


Coordinates are in **3D**.

The x and y coordinates of each primitive are transformed to for **screen coordinates**.

Rasteriser Stage

Given the **transformed** and **projected** vertices with their associated **shading data** from the geometry stage.



The goal of the rasteriser stage is to **compute** and **set colours** for the pixels covered by the object.

Rasterisation converts from 3D vertices in screen-space to **pixels** on the screen.

1) Triangle Setup

Vertices are collected and **converted into triangles**.

Information is generated that will allow later stages to accurately **generate the attributes** of every pixel associated with the triangle.

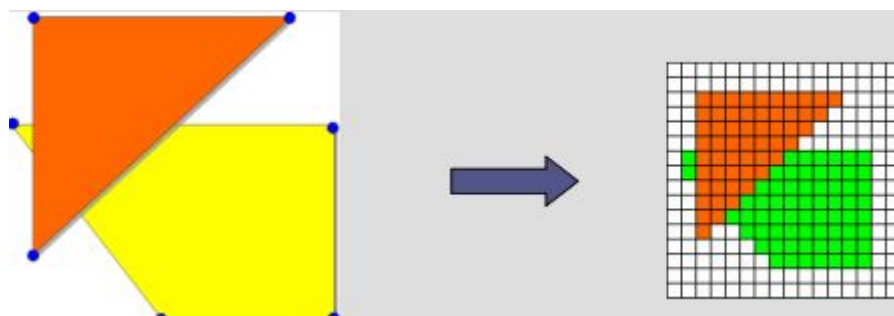
2) Triangle Traversal

Decides which pixels are inside a triangle.

Each pixel that has its **center covered** by the triangle is checked.

A **fragment** is generated for the part of the pixel that overlaps the triangle.

Triangle vertices are **interpolated**.



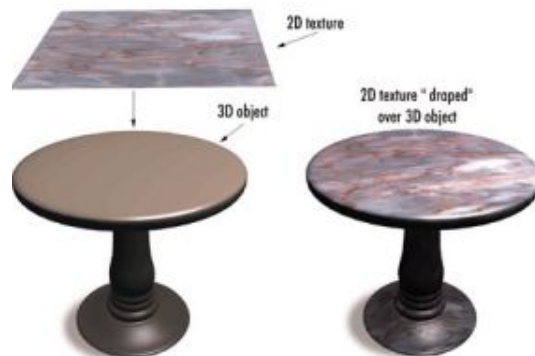
3) Pixel Shading

Performs **per-pixel shading** computations.

The end result is one or more colours to be passed to the next stage.

Executed by **programmable GPU cores**.

Texturing is employed here.



4) Merging

Information for each pixel is stored in the **colour buffer**, which is a rectangular array of colours.

The stage combines the **fragment colour** produced by the shading stage with the **colour currently stored** in the buffer.

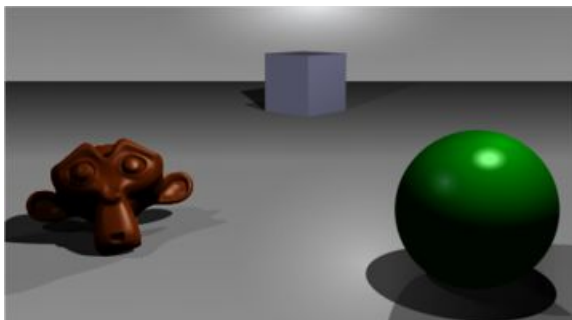
This stage is also responsible for resolving **visibility** with the **z-buffer**.

Z-Buffer

The z-buffer is arranged as a 2D array with one element for each screen pixel. It stores the *z* value from the camera to the currently closest primitive.

If another object of the scene must be rendered in the **same pixel**, the method **compares their depths** and chooses the closest one to the observer.

The chosen depth is then saved to the z-buffer, replacing the old one.



Double Buffering

The screen displays the contents of the colour buffer.

To avoid perception of primitives being rasterised, double buffering is used.

Rendering takes place off screen in a **back buffer**.

Once complete, the contents are swapped with the **front buffer**.

OpenGL Coordinates

OpenGL uses a **4-component vector** to represent a vertex.

This is known as a **homogeneous coordinate system**.

Usually $w = 1$.

In 2D space, $z = 0$.

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

03a Geometric Transformations

Linear algebra is the cornerstone of computer graphics.

We need to be able to manipulate:

1. Points
2. Vectors

These form the basics of all geometric objects and operations.

Geometric operations (scale, rotate, translate) are defined using **matrix transformations**.

Optical effects (reflect, refract) are defined using **vector algebra**.

Coordinate Systems

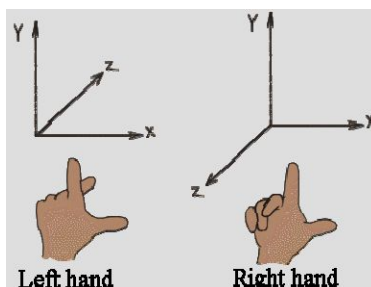
By convention, we usually employ a **Cartesian basis**:

- Basis vectors are **mutually orthogonal** and **unit length**.
- Basis vectors are named x , y and z .

We need to define the relationship between these three vectors.

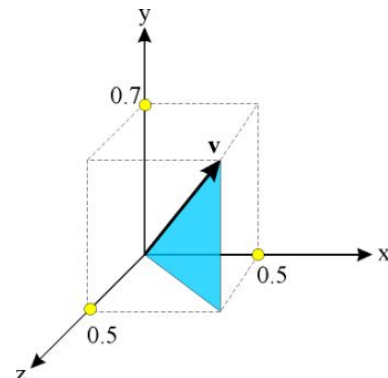
There are two possibilities:

1. **Right-handed systems**: [OpenGL]
 z comes **out** of the page
2. **Left-handed systems**:
 z goes **in** to the page



$$\mathbf{v} = \begin{bmatrix} 0.5 \\ 0.7 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.7 \\ -0.5 \end{bmatrix}$$

RHS LHS



Conventions

Vector quantities are denoted as \mathbf{v} or \vec{v} .

Each vector is defined with respect to a set of **basis vectors**.

We will use **column-format vectors**:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \neq [v_1 \ v_2 \ v_3] \quad (= [v_1 \ v_2 \ v_3]^T)$$

Row vs. Column Formats

The two formats are fundamentally different:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} u & v & w \end{bmatrix} \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

$\mathbf{M}\mathbf{v} = \mathbf{v}^T \mathbf{M}^T$

$$\begin{bmatrix} ? \\ ? \\ ? \\ ? \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$$

In GLSL:

```
mat3 m;
vec3 v;

mat3 u = m * v;
```

Vectors & Points

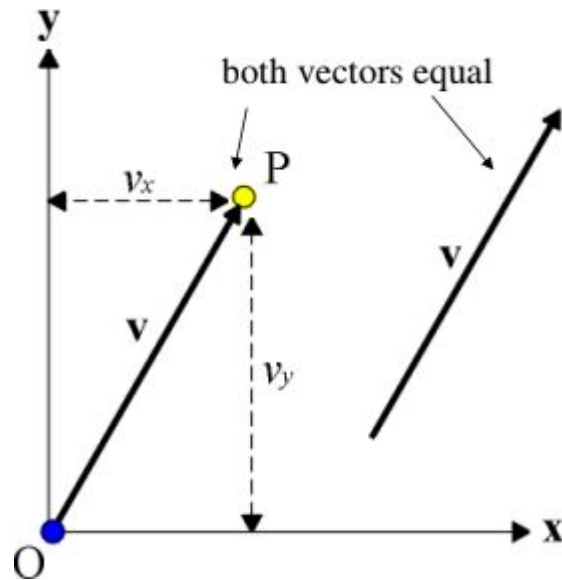
Although vectors and points are often used interchangeably in graphics texts, it is important to distinguish them:

1. **Vectors** represent **directions**.
2. **Points** represent **positions**.

Both are meaningless without reference to a **coordinate system**:

1. Vectors require a set of **basis vectors**.
2. Points require an **origin** and a **vector space**.

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$



Computer Graphics Problems

Much of graphics concerns itself with the problem of displaying **3D objects** in a **2D screen**.

We want to be able to do the following to our objects:

- **Rotate, translate & scale.**
- View them from arbitrary **points of view**.
- View them in **perspective**.

We want to display objects in coordinate systems that are convenient for us, and to be able to **reuse** object descriptions.

Matrices

If you want to rotate many vertices about some axis by multiplying each vertex by several matrices, you can instead **pre-multiply** the matrices.

You can then multiply each vertex by the resulting matrix.

Addition:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} + \begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} a+e & c+g \\ b+f & d+h \end{bmatrix}$$

Multiplication:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} \times \begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} ae+cf & ag+ch \\ be+df & bg+dh \end{bmatrix}$$

Matrix multiplication is **not commutative** i.e. $AB \neq BA$.

If $AB = AC$, it does **not** follow that $B = C$.

Matrix multiplication is **associative** and **distributive**:

1. $(AB)C = A(BC)$ *(associativity)*
2. $A(B + C) = AB + AC$ *(distributivity)*
3. $(A + B)C = AC + BC$ *(distributivity)*

The **transpose** A^T of a matrix A is one whose **rows** are switched with its **columns**.

Geometric Transformations

Many geometric transformations are **linear** and can be represented as a **matrix multiplication**.

A function f is linear *iff*:

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

To transform a line, we transform the **endpoints**.

Points between are **affine combinations** of the transformed endpoints.

Given a line between points P and Q , points along the transformed line are affine combinations of transformed P' and Q' .

$$L(t) = P + t(Q - P)$$

$$L'(t) = P' + t(Q' - P')$$

Homogeneous Coordinates

The basis of the homogeneous coordinate system is the set of n **basis vectors** and the **origin position**:

$$v_1, v_2, \dots, v_n \text{ and } P_0$$

All points and vectors are therefore compactly represented using their coordinates:

$$\begin{bmatrix} a_1 \\ \vdots \\ a_n \\ a_o \end{bmatrix} \quad \text{or more usually} \quad \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Vectors have **no positional information** and are represented using $a_0 = 0$.

Points are represented with $a_0 = 1$.

$$\vec{v} = a_1 v_1 + \dots + a_n v_n + 0$$

$$P = a_1 v_1 + \dots + a_n v_n + P_0$$

$\begin{bmatrix} 0.2 \\ 1.3 \\ 2.2 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 1 \end{bmatrix}$		$\begin{bmatrix} 0.2 \\ 1.3 \\ 2.2 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 0 \end{bmatrix}$
Points			Associated vectors	

Using this scheme, every rotation, translation and scaling operation can be represented by a **matrix multiplication**. Any **combination** of the operations corresponds to the **products** of the corresponding matrices.

Using homogeneous coordinates allows us to treat translation in the **same way** as rotation and scaling.

Translation

A **translation** is a displacement in a particular direction.

They are defined by specifying the displacements a , b and c .

$$x' = x + a$$

$$y' = y + b$$

$$z' = z + c$$

Translations only apply to **points**, never vectors.

$$\begin{array}{l} x' = x + a \\ y' = y + b \\ z' = z + c \end{array} \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + a \\ y + b \\ z + c \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Translations are independent, and can be performed in **any order** (including all at once).

Scaling

Scaling allows us to make models **larger** or **smaller**.

If we multiply all of our coordinates by $\frac{1}{3}$, then we get a model that is $\frac{1}{3}$ of the size.

However, the **coordinates** are scaled as well.

In order to prevent this, we:

1. **Translate** to origin.
2. **Scale**.
3. **Translate** back.

Generally we centre the scaling at the **origin**:

$$x' = s_x x$$

$$y' = s_y y$$

$$z' = s_z z$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \mathbf{v}' = \mathbf{S}\mathbf{v}$$

We would also like to scale points, thus we need a **homogeneous transformation** for consistency:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ w \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad \mathbf{S}^{-1} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

A **rotation** turns about a point (a, b) through an angle θ .
Generally, rotations are **anti-clockwise** about the **origin**.

For example, using the z -axis as the axis of rotation the equations are:

$$x' = x \cos\theta - y \sin\theta$$

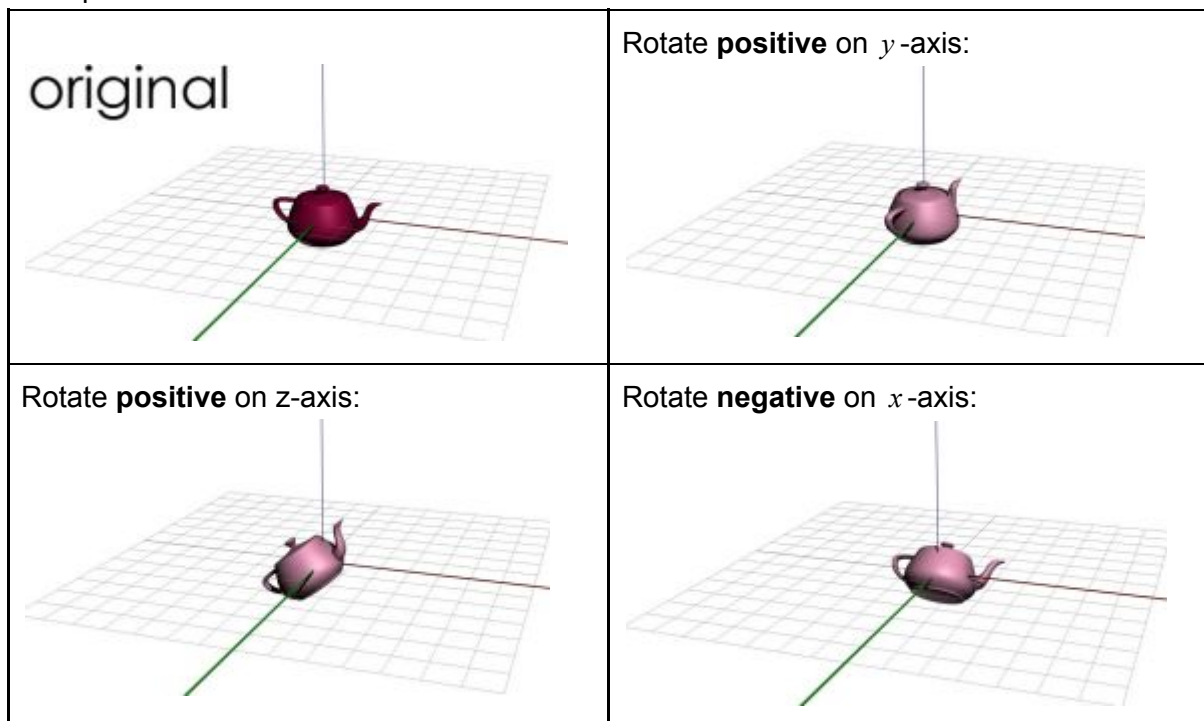
$$y' = x \sin\theta + y \cos\theta$$

$$z' = z$$

To rotate about an axis:

1. Put your eye on that axis in the **positive direction**, and **look towards the origin**.
2. A **positive rotation** corresponds to a **counter-clockwise** rotation.

Examples:



Rotation About The z -Axis

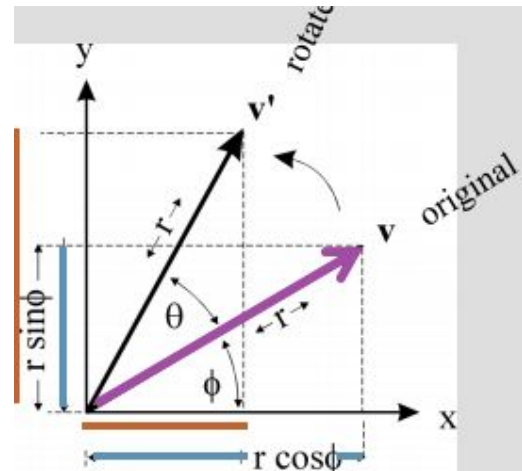
$$\mathbf{v} = \begin{bmatrix} r \cos \phi \\ r \sin \phi \end{bmatrix}$$

$$\begin{aligned} x' &= r \cos(\Phi + \theta) = r \cos(\Phi) \cos(\theta) - r \sin(\Phi) \sin(\theta) \\ y' &= r \sin(\Phi + \theta) = r \cos(\Phi) \sin(\theta) + r \sin(\Phi) \cos(\theta) \end{aligned}$$

Since $x = r \cos(\Phi)$ and $y = r \sin(\Phi)$:

$$x' = x \cos(\theta) - y \sin(\theta)$$

$$y' = x \sin(\theta) + y \cos(\theta)$$



Rotation in the clockwise direction is the **inverse** of rotation in the counterclockwise direction, and vice-versa.

2D rotation of θ about origin:

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

3D homogeneous rotations:

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_y = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note:

There is a difference for the rotation about y due to the RHS.

Note:

$$\begin{aligned} \cos(-\theta) &= \cos \theta \\ \sin(-\theta) &= -\sin \theta \end{aligned} \Rightarrow \mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta) = \mathbf{R}^T(\theta)$$

Combining Rotation, Translation & Scaling

It is often advantageous to **combine** various transformations to form a more complex transformation.

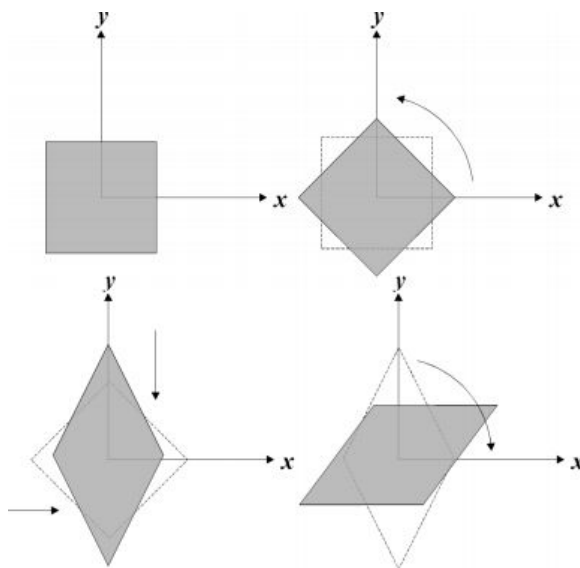
Things get complicated if we use algebra, so instead we use **matrices**.

Homogeneous Coordinates

Using this scheme allows us to represent every rotation, translation and scaling operation by a **matrix multiplication**.

Any **combination** of the operations corresponds to the **products** of the corresponding matrices.

Affine Transformations



All **affine transformations** are combinations of rotations, scaling and translation operations.

Transformation Composition I

It is common for graphics programs to apply **more than one** transformation to an object.

Example:

Take a vector v_1 , scale (S) it and then rotate (R) it.

1. $v_2 = Sv_1$
2. $v_3 = Rv_2$

Since matrix multiplication is **associative**, we can instead say $v_3 = (RS)v_1$.

We can therefore recreate the effect of the two matrices by **multiplying them together** first.

More complex transformations can be created by **concatenating / composing** individual transformations together:

$$M = T \circ R \circ S \circ T = TRST$$

$$v' = T[R[S[Tv]]] = Mv$$

Matrix multiplication is **noncommutative**, therefore order is vital.

Rotation About a Point

We can create an **affine transformation** representing rotation about a point P_R :

$$M = T(P_R)R(\theta)T(-P_R)$$

In other words (we order the operations from **right** \rightarrow **left**):

1. $T(-P_R)$ Translate to origin
2. $R(\theta)$ Rotate about origin
3. $T(P_R)$ Translate back to original location

Transformation Composition II

To rotate about a point P by q degrees in the xy plane:

$$M = T(P)R(\theta)T(-P)$$

$$= \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos \theta & -\sin \theta & 0 & P_x - P_x \cos \theta + P_y \sin \theta \\ \sin \theta & \cos \theta & 0 & P_y - P_x \sin \theta - P_y \cos \theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Euler Angles

Euler angles represent the angles of rotation about the coordinate axes required to achieve a **given orientation** $(\theta_x, \theta_y, \theta_z)$.

The resulting matrix is:

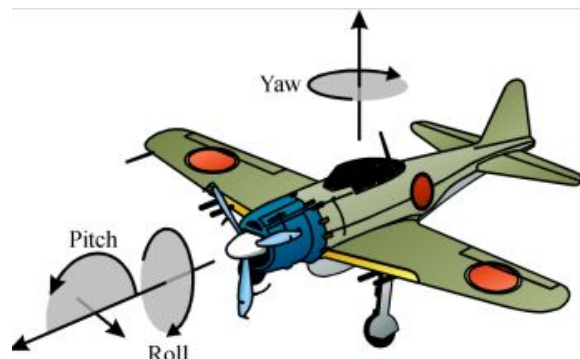
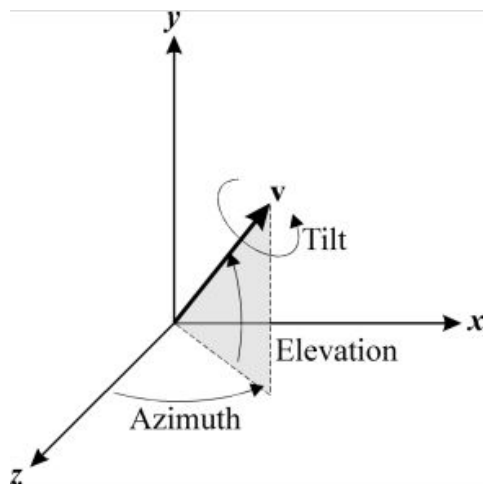
$$M = R(\theta_x)R(\theta_y)R(\theta_z)$$

Any required rotation may be described as a **composition of three rotations** about the **coordinate axes**.

NB: Rotation does **not** commute. Order is important.

Rotational DOF (Degrees of Freedom)

These are sometimes referred to as **roll**, **pitch** and **yaw**:



OpenGL Uniforms

We use **uniforms** to declare shader input data that **stays the same** e.g. a transformation matrix.

```
in vec4 vPosition; // Vertex in the local coordinate system.
uniform mat4 mM; // Matrix for the pose of the model.
uniform mat4 mV; // Matrix for the pose of the camera.
uniform mat4 mP; // Projection matrix (perspective).

void main() {
    // vPosition = old position, gl_Position = new position.
    gl_Position = mP * mV * mM * vPosition;
}
```

General Rotation

What do we do if we want to rotate about an axis **other** than the three principal axes?

1. Perform one or two rotations about the principal axes in order to align with the z -axis.
2. Rotate about the z -axis.
3. Undo the rotations in *step 1*.

Rotation About An Arbitrary Axis

A frequent requirement is to determine the **matrix** for rotation about a given axis.

Such rotations have **3 degrees of freedom**:

- 2 for spherical angles specifying **axis orientation**.
- 1 for **twist** about the **rotation axis**.

We assume that the axis is defined by points P and Q .

The **pivot point** is P .

The **rotation vector** is $v = \frac{P-Q}{|P-Q|}$.

Steps:

1. Translate the **pivot point** of the axis to the **origin**.
 $T(-P)$
2. Determine a **series of rotations** to achieve the required rotation about the desired vector.
3. Rotate the axis and object so that the axis **lines up** with z .
e.g. $R(-\theta_x)R(-\theta_y)$
4. Rotate about z by the required angle θ :
 $R(\theta)$
5. **Undo** the first two rotations to bring us back to the **original rotation**.
 $R(-\theta_x)R(-\theta_y)$
6. Translate back to the **original position**:
 $T(P)$
7. The final **rotation matrix** is:
 $M = T(P) R(-\theta_y) R(-\theta_x) R(\theta) R(\theta_x) R(\theta_y) T(-P)$

Rotation About An Axis

We need the **Euler angles** θ_x and θ_y which will orient the rotation axis along the z -axis. We determine these using simple **trigonometry**.

Aligning With The z -Axis

We rotate the line segment into the plane $y = 0$.

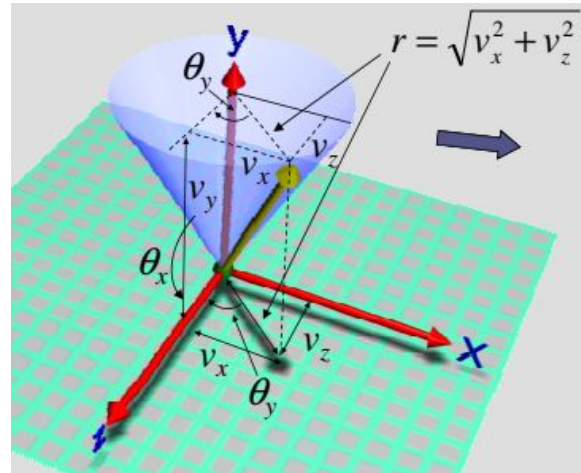
$$r = \sqrt{v_x^2 + v_z^2}$$

$$\cos(\theta_x) = \frac{v_z}{r}$$

$$\sin(\theta_x) = \frac{v_x}{r}$$

$$\cos(\theta_y) = \frac{v_z}{r}$$

$$\sin(\theta_y) = \frac{v_x}{r}$$



The rotation about the x -axis is anti-clockwise, but the y -axis rotation is clockwise.

Therefore the required y -axis rotation is $-\theta_y$:

$$\mathbf{R}(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & r & -v_y & 0 \\ 0 & v_y & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow \mathbf{R}(-\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & r & v_y & 0 \\ 0 & -v_y & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}(\theta_y) = \begin{bmatrix} v_z/r & 0 & v_x/r & 0 \\ 0 & 1 & 0 & 0 \\ -v_x/r & 0 & v_z/r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow \mathbf{R}(-\theta_y) = \begin{bmatrix} v_z/r & 0 & -v_x/r & 0 \\ 0 & 1 & 0 & 0 \\ v_x/r & 0 & v_z/r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M = T(P) R(\theta_y) R(-\theta_x) R(\theta) R(\theta_x) R(-\theta_y) T(-P)$$

03b Linear Algebra for Graphics

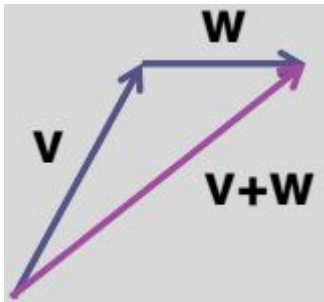
Vectors are **equivalent** if they both have the same:

1. Length
2. Direction

Equivalent vectors are regarded as equal even if they are located in **different positions**.

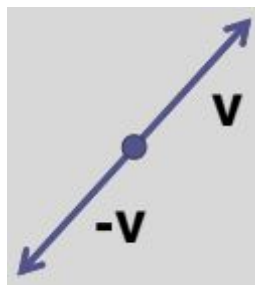
Vector Addition

Head-to-tail rule:



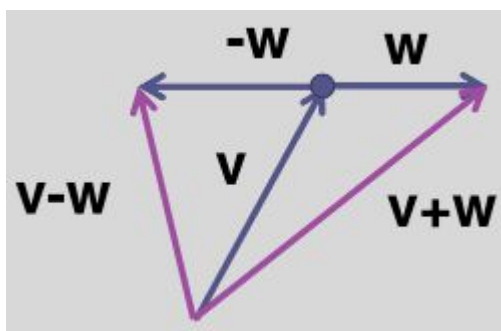
Negative Vectors

The negative of a vector v has the same magnitude but opposite direction.

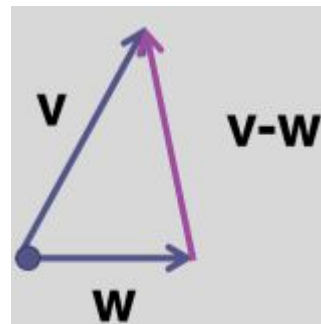


Vector Subtraction

$$v - w = v + (-w)$$



Tail-to-tail rule:



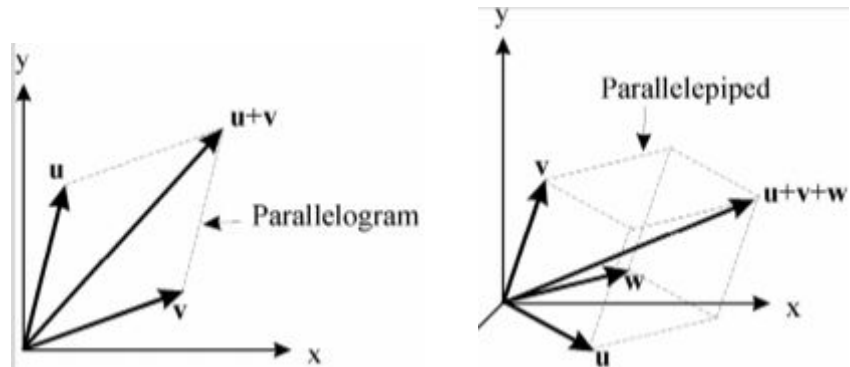
The Parallelogram Law

Addition and subtraction of vectors follows the **parallelogram law** in 2D.
In higher dimensions, they follow the **parallelepiped law**.

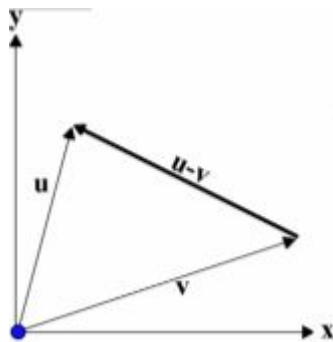
Addition:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 9 \end{bmatrix}$$

$\mathbf{u} \quad \mathbf{v} \quad \mathbf{u+v}$

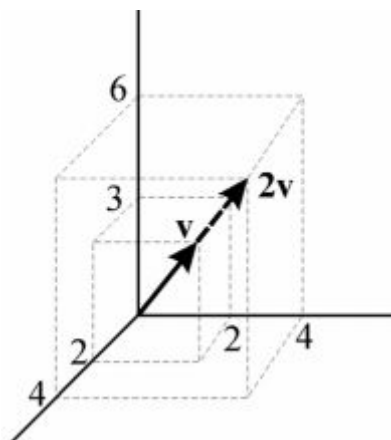


Subtraction:



Vector Multiplication

Multiplying a vector by a **scalar** scales the vector's **length** appropriately.
It does **not** affect its direction.



Vectors that are scalar multiples of each other are **parallel**.

Linear Combinations

The **linear combination** of a set of vectors is the **sum of scalar multiples** of those vectors:

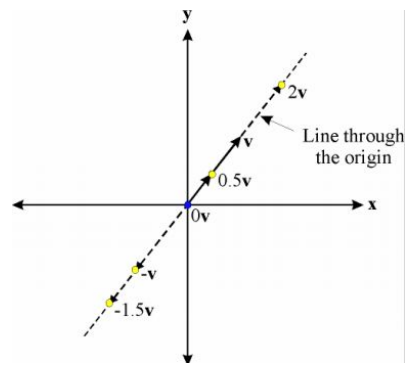
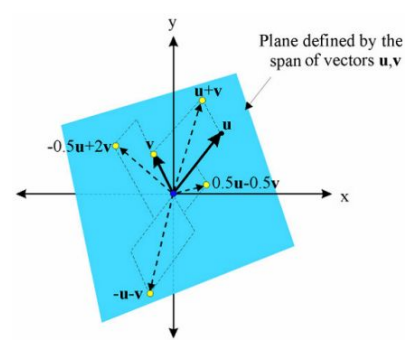
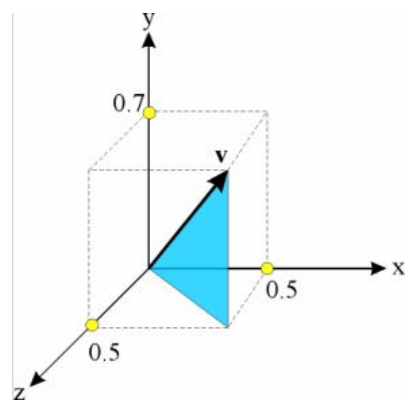
$$u = a_1v_1 + a_2v_2 + \dots + a_nv_n$$

Fixing vectors v_i yields an infinite number of u depending on the scalars a_i where:

- u is the **span** of the vectors v_i .
- v_i are **basis vectors** for the space.

If none of the v_i can be created as a linear combination of the others, then the vectors v_i are said to be **linearly independent**.

All linear combinations contain the **zero vector**.

	<p>Linear combinations of one vector \Rightarrow an infinite line.</p>
	<p>Linear combinations of two vectors \Rightarrow a plane.</p>
	<p>Linear combinations of three vectors \Rightarrow a 3D volume.</p> <div style="background-color: #f0f0f0; padding: 10px; margin: 10px 0;"> $\mathbf{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{z} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ </div> <p>v is linear combination of the basis vectors x, y and z:</p>

$$\mathbf{v} = \begin{bmatrix} 0.5 \\ 0.7 \\ 0.5 \end{bmatrix} = 0.5 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 0.7 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + 0.5 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Vector Magnitude

The **magnitude** or **norm** of a vector \mathbf{v} of dimension n is given by the standard **Euclidean distance metric**:

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

For example:

$$\left\| \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \right\| = \sqrt{1^2 + 3^2 + 1^2} = \sqrt{11}$$

Vectors of **length one** (unit vectors) are often termed **normal** or **normalised vectors**.

Normalised Vectors

When describing **direction** we use **normalised vectors**.

We normalise a vector by dividing by its **magnitude**:

$$\mathbf{v}' = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}$$

Examples:

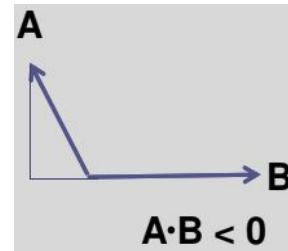
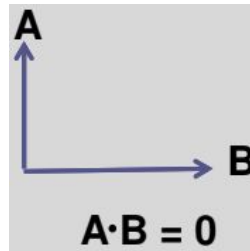
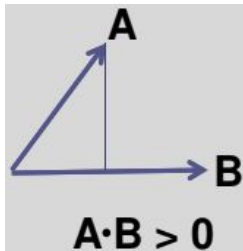
Let $\mathbf{u} = [2, -2, 3]$, $\mathbf{v} = [1, -3, 4]$ and $\mathbf{w} = [3, 6, -4]$.

$$\begin{aligned} \|\mathbf{u} + \mathbf{v}\| &= \sqrt{83} \\ \|\mathbf{u}\| + \|\mathbf{v}\| &= \sqrt{17} + \sqrt{26} \\ \|\mathbf{w} - 2\mathbf{u}\| + 2\|\mathbf{u}\| &= 4\sqrt{17} \end{aligned}$$

Dot Product

A dot product of two vectors returns a **scalar**.
It calculates **angles**.

"It returns the length of the projection A onto B."



The dot product (inner product) is defined as:

$$u \cdot v = \sum_i u_i v_i$$

$$u \cdot v = u^T v$$

$$= \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= u_1 v_1 + u_2 v_2 + u_3 v_3$$

Note:

$$u \cdot u = u_1^2 + u_2^2 + u_3^2 = \|u\|^2$$

Therefore, we can also define **magnitude** in terms of the dot product operator:

$$\|u\| = \sqrt{u \cdot u}$$

The dot product operator is **commutative**.

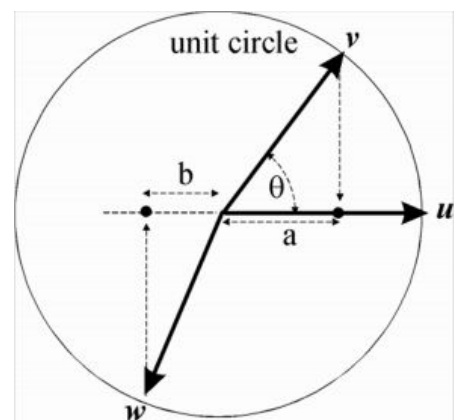
If **both** vectors are **normalised**, then the dot product defines the **cosine** of the angle between vectors:

$$u \cdot v = \cos \theta$$

In general:

$$u \cdot v = \|u\| \|v\| \cos \theta$$

$$\Rightarrow \theta = \cos^{-1} \frac{u \cdot v}{\|u\| \|v\|}$$



If **one** of the vectors is normalised, then the dot product defines the **projection of the other** onto it (perpendicularly).

In this example, a is positive and b is negative.

$$a = u \cdot v$$

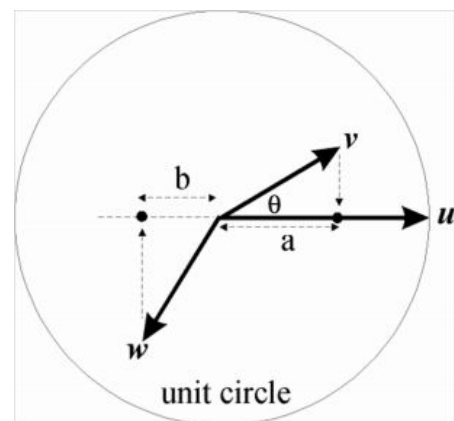
$$b = u \cdot w$$

Note that if both vectors are pointing in the **same direction** then the dot product is **positive**.

$$u \cdot v = \|u\| \|v\| \cos \theta$$

$$\Rightarrow a = \|v\| \cos \theta$$

$$\Rightarrow \cos \theta = \frac{a}{\|v\|}$$

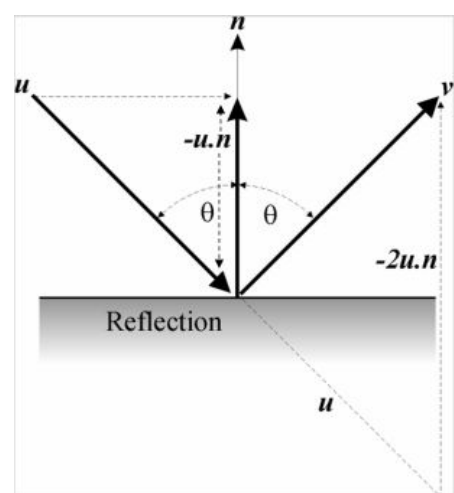


If $\theta = 90^\circ$ then the dot product $= 0$.

The projection of one onto the other has zero length

\Rightarrow The vectors are **orthogonal**.

Similarly, if $\theta < 90^\circ$ then the dot product < 0 .



Dot Product in Computer Graphics

The dot product can be used to find:

- Is an angle between two vectors acute, a right angle or obtuse?
- Is a polygon facing towards or away from the camera?

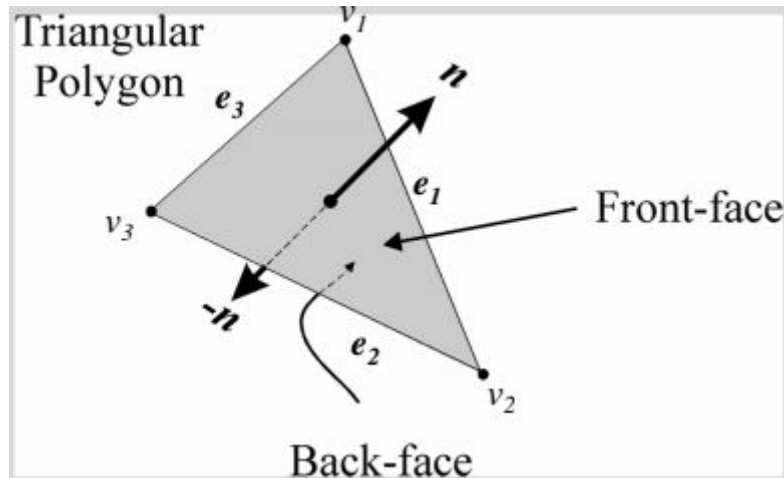
A problem with the dot product is that arccosine always returns a **positive number**.

The dot product is **directionless**, giving you the same result for $A \cdot B$ as $B \cdot A$.

Normals

Polygons are planar regions bounded by n edges connecting n points / vertices.

For lighting and viewing calculations, we need to define the **normal** to a polygon. The normal distinguishes the **front face** from the back face of a polygon.



The plane of the polygon divides 3D space into two **half-spaces**:

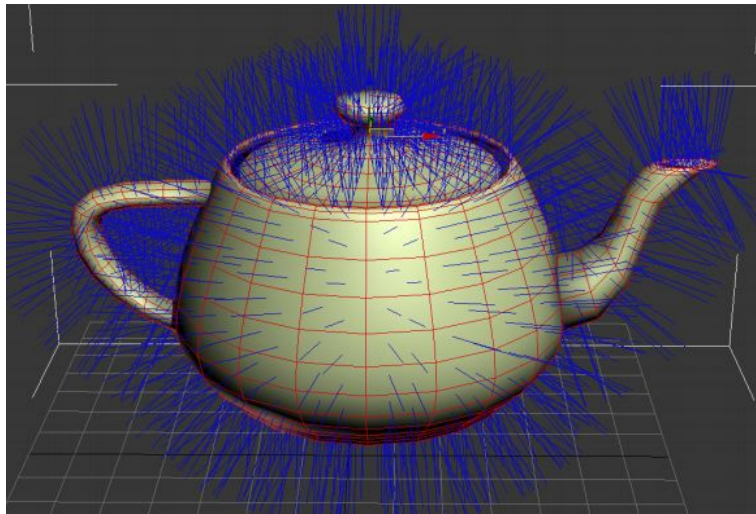
1. **In front** of the polygon.
2. **Behind** the polygon.

To determine the side that a polygon lies on:

$$d = n \cdot (P - v_i)$$

The values of d correspond as follows:

1. $d < 0 \Rightarrow P$ lies **behind** the polygon.
2. $d = 0 \Rightarrow P$ lies **on** the polygon.
3. $d > 0 \Rightarrow P$ lies **in front of** the polygon.



Cross Product

The cross product of two vectors returns a **vector**.
It calculates **direction**.

The cross product returns a vector that is **orthogonal** (perpendicular) to the plane formed by the two input vectors.

The cross product is used for:

1. Defining **orientation**
2. Constructing **coordinate axes**

It is defined as:

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{bmatrix}$$

$$\mathbf{A} = [a, b, c] \quad \mathbf{B} = [d, e, f]$$

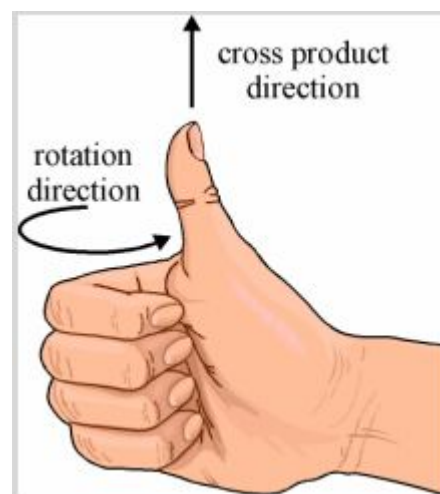
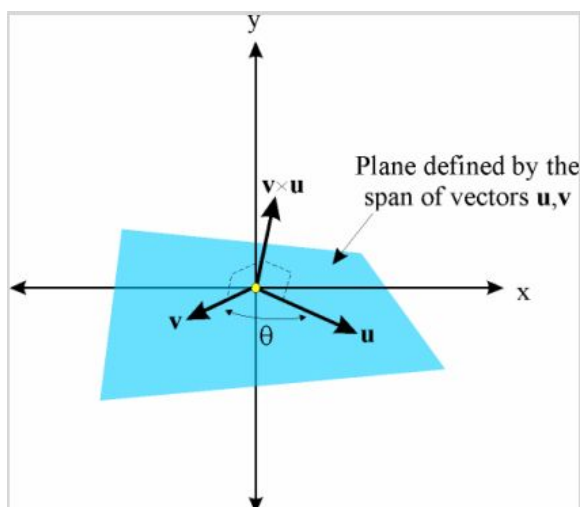
$$\begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ a & b & c \\ d & e & f \end{vmatrix}$$

$$\mathbf{A} \times \mathbf{B} = [(bf - ce), (cd - af), (ae - bd)]$$

$$\mathbf{A} \times \mathbf{B} \neq \mathbf{B} \times \mathbf{A}$$

If $\mathbf{w} = \mathbf{u} \times \mathbf{v}$:

1. $\|\mathbf{u} \times \mathbf{v}\| = \|\mathbf{u}\| \|\mathbf{v}\| \sin \theta$
2. $\mathbf{u} \times \mathbf{v} = \mathbf{w} \|\mathbf{u}\| \|\mathbf{v}\| \sin \theta$



Right-Handed Coordinate System

The cross product is **anticommutative**:

$$u \times v = -(v \times u)$$

The cross product is **not associative**:

$$u \times (v \times w) \neq (u \times v) \times w$$

Therefore the direction of the resulting vector is defined by **operand order**.

Cross Product in Computer Graphics

The classic use of the cross product is figuring out the **normal vector** of a polygon. This is fundamental to calculating which polygons are facing the camera.

This process determines which polygons are drawn and ignored.

Dot Product	Cross Product
<ul style="list-style-type: none">• Returns a scalar.• Calculates angles. $A \cdot B = B \cdot A$	<ul style="list-style-type: none">• Returns a vector.• Calculates direction. $A \times B \neq B \times A$

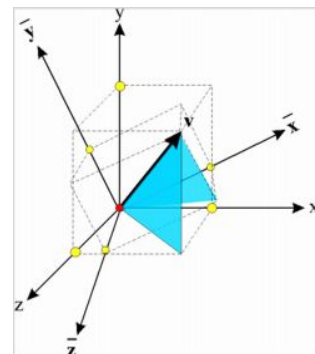
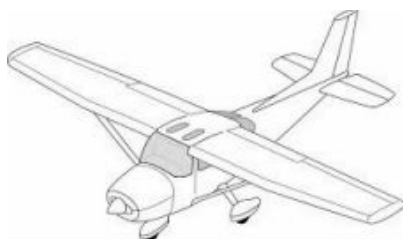
Coordinate Systems

The [Cartesian coordinate system](#) is one of infinitely many possible orthonormal bases.

The **global coordinate system** in graphics is the **canonical coordinate system**. It is special because x , y , z and the origin are never explicitly stored.

If we want to use **another** coordinate system with origin p and orthonormal basis vectors u , v and w , then we **do** store these vectors explicitly.

In this example, the coordinate system associated with the plane is the **local coordinate system**.



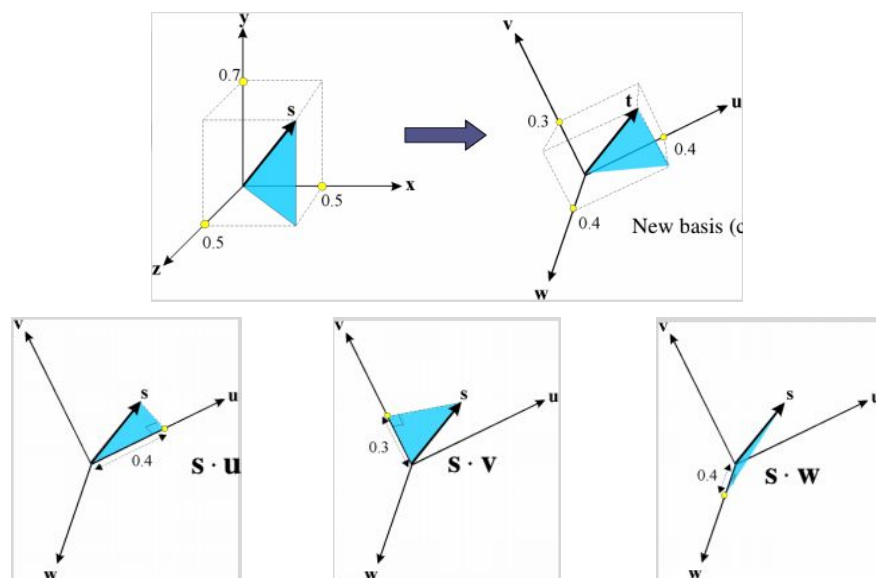
Change of Basis

If we know s that is defined w.r.t. basis xyz , then we can determine t which is the same vector defined w.r.t. basis uvw :

- t_u is the projected distance of s onto u .
- t_v is the projected distance of s onto v .
- t_w is the projected distance of s onto w .

$$\mathbf{t} = \begin{bmatrix} \mathbf{s} \cdot \mathbf{u} \\ \mathbf{s} \cdot \mathbf{v} \\ \mathbf{s} \cdot \mathbf{w} \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} = \mathbf{M}\mathbf{s} \quad \begin{cases} t_u = u_x s_x + u_y s_y + u_z s_z = \mathbf{u} \cdot \mathbf{s} \\ t_v = v_x s_x + v_y s_y + v_z s_z = \mathbf{v} \cdot \mathbf{s} \\ t_w = w_x s_x + w_y s_y + w_z s_z = \mathbf{w} \cdot \mathbf{s} \end{cases}$$

The **transformation matrix** M allows us to transform a vector from **one basis to another**. Many common geometric operations can be expressed as a transformation matrix.



Normally the vectors forming the basis of a coordinate system are:

1. Unit length.
2. Mutually orthogonal.

A basis is said to be **orthonormal** $\Rightarrow M = M^{-1}$:

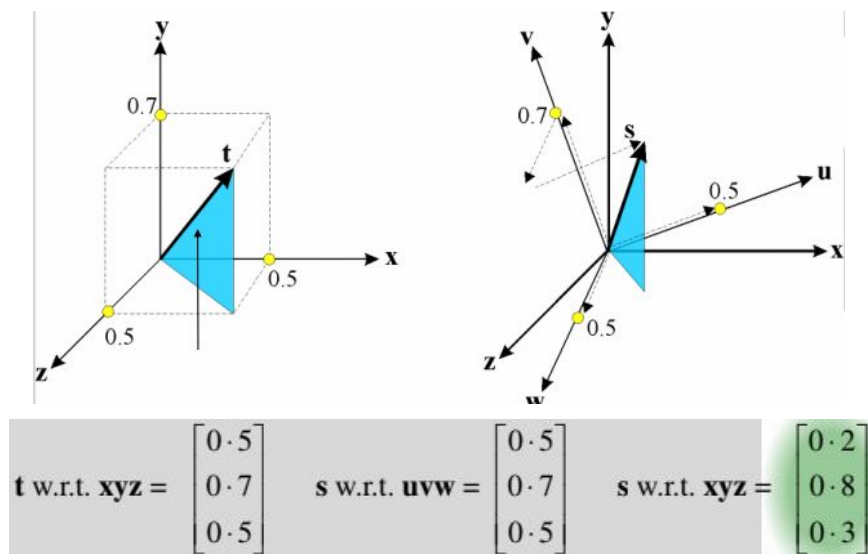
- True for all **rotation matrices**.
- Not true for **scaling transformations**.

Therefore if we have a vector t defined w.r.t. basis uvw , then the vector s w.r.t. basis xyz is given by:

$$\mathbf{s} = t_u \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} + t_v \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + t_w \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \begin{bmatrix} t_u \\ t_v \\ t_w \end{bmatrix} = \mathbf{M}^{-1} \mathbf{t} = \mathbf{M}^T \mathbf{t}$$

Transformation

Changing basis is geometrically equivalent to **transformation**.



Affine Spaces

Vectors define **direction** and **magnitude** only.

To encode position we need to **fix** the origin.

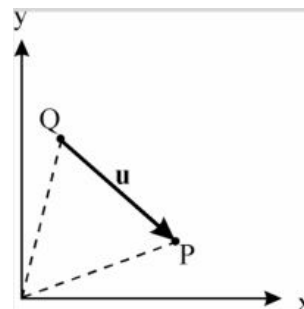
An **affine space** is:

1. A **set of points**
2. With an associated **vector space**
3. With the **operations**:
 - a. Difference
 - b. Translate

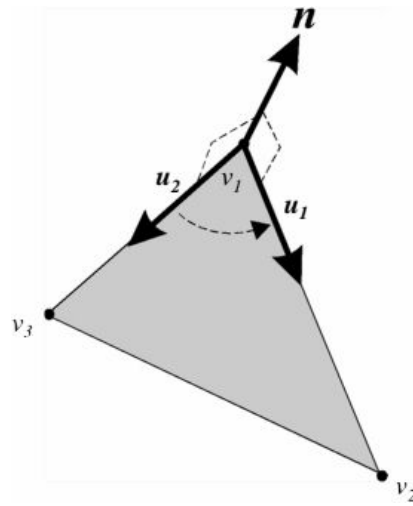
Points are related by vectors:

$$u = P - Q$$

$$P = Q + u$$



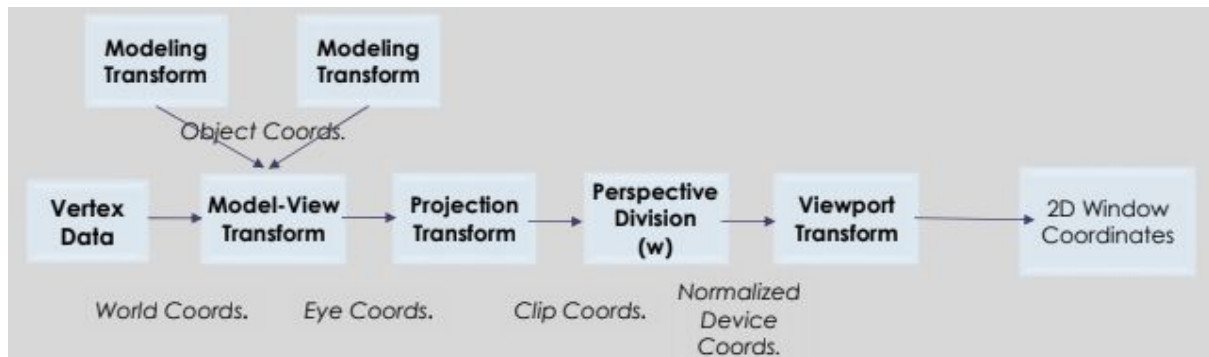
Normals & Polygons



04 Viewing

Transformation Pipeline

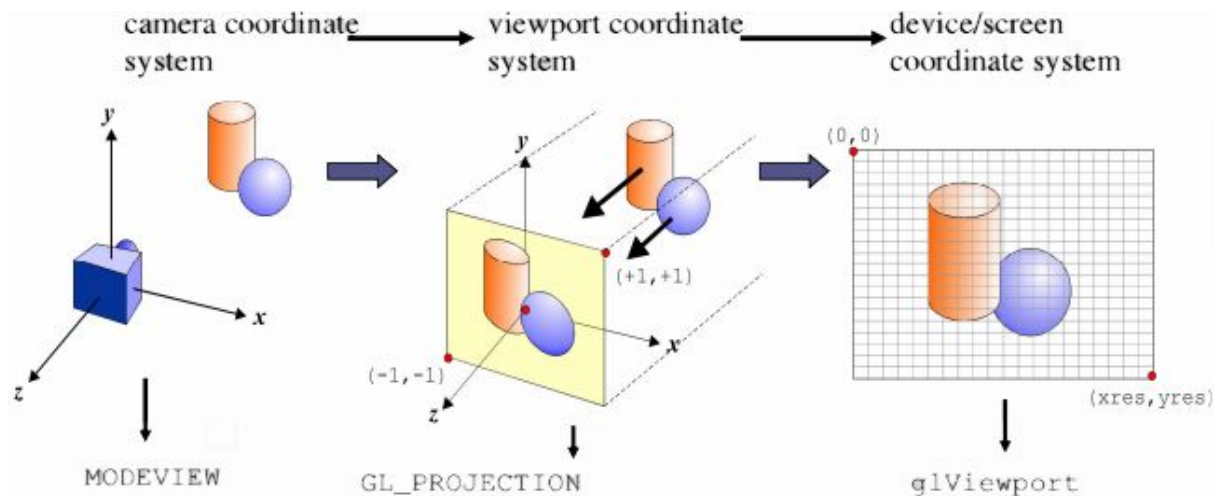
Transformations take us from one **space** to another.
All of our transforms are 4×4 matrices.



Camera Modelling

Types of transformation:

- **Projection** transformations adjust the **lens** of the camera.
- **Viewing** transformations define the **position** and **orientation** of the viewing volume in the world.
- **Modelling** transformations move the **model**.
- **Viewport** transformations enlarge / reduce the physical photograph.



Model Matrix

When creating a triangle or loading a mesh from a file, it has some $(0, 0, 0)$ **origin** which is local to that mesh.

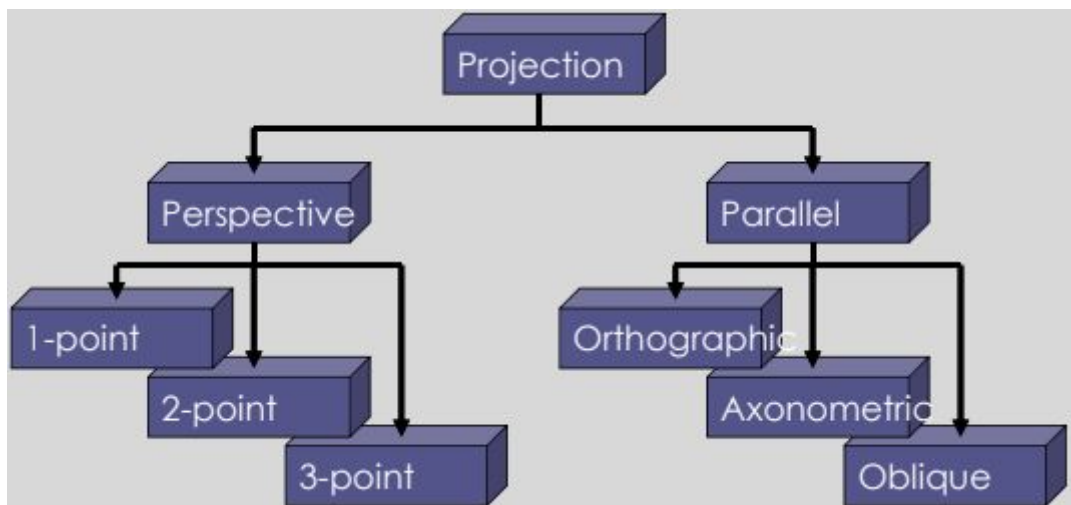
In order to translate, rotate or scale to position the model in a virtual world, we multiply its points with a **model matrix** (world matrix).

```
mat4 M = T * R * S; // translation * rotation * scale  
vec4 pos_wor = M * vec4(pos_loc, 1.0);
```

3D to 2D Projection

The type of projection depends on a number of factors:

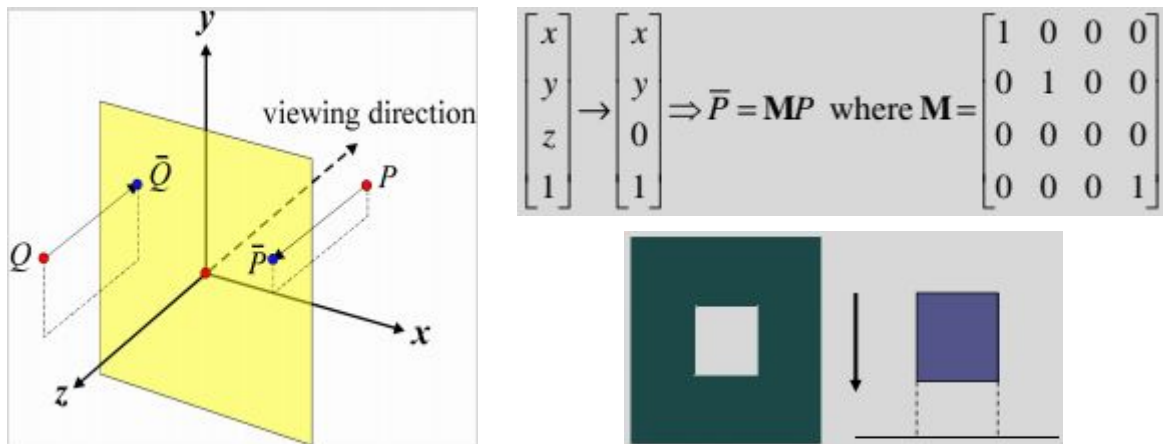
1. **Location** and **orientation** of the viewing plane / **viewport**.
2. **Direction of projection** which is described by a vector.
3. Projection type.



Orthogonal Projection

Orthogonal projections are **parallel** projections onto the view plane.

Usually the view plane is **axis-aligned** (often $z = 0$):



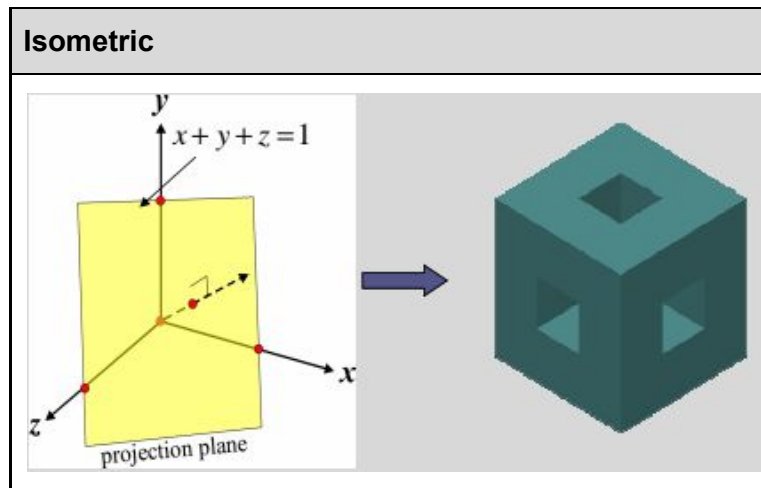
The projection is:

1. **Orthographic** if the object is **axis-aligned**.
2. **Axonometric** otherwise.

Orthographic	Axonometric

An **axonometric projection** is a type of orthographic projection used to create a **pictorial drawing** of an object. The object is **rotated** along one (or more) of its axes relative to the plane of projection.

If the projection plane intersects the principle axes at the **same distance** from the origin, then the projection is **isometric**.

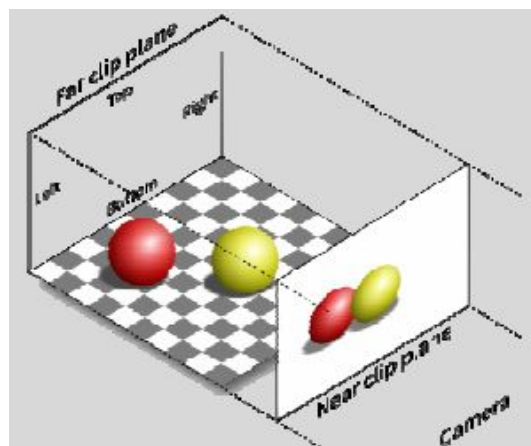


In OpenGL, the default projection matrix is an **identity matrix**, or equivalently:

```
// ortho(left, right, bottom, top, near, far)
mat4 N = ortho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

This is a **canonical view volume**:

- Points within the cube are mapped to the same cube.
- Points outside the cube remain outside \Rightarrow **clipped**.



Note:

We always view in the $-z$ direction. We need to transform the world in order to view in other arbitrary directions.

Orthogonal Projection Matrix

The projection **linearly** maps view-space coordinates into clip-space coordinates.

We transform this canonical view volume to the cube centered at the origin, with sides of length 2.

We need to **translate to the origin** and **scale the sides** to have a length of 2:

$$N = \begin{bmatrix} 2 / \text{right} - \text{left} & 0 & 0 & -(\text{left} + \text{right} / \text{right} - \text{left}) \\ 0 & 2 / \text{top} - \text{bottom} & 0 & -(\text{top} + \text{bottom} / \text{top} - \text{bottom}) \\ 0 & 0 & -2 / \text{far} - \text{near} & -(\text{far} + \text{near} / \text{far} - \text{near}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection

Parallel / Orthogonal Projection	Perspective Projection
<ul style="list-style-type: none"> Keeps parallel lines parallel. Preserves size and shape of planar objects. Unrealistic. Used in architecture. Represents less-natural images. Simple. 	<ul style="list-style-type: none"> Objects further away appear smaller. More realistic.

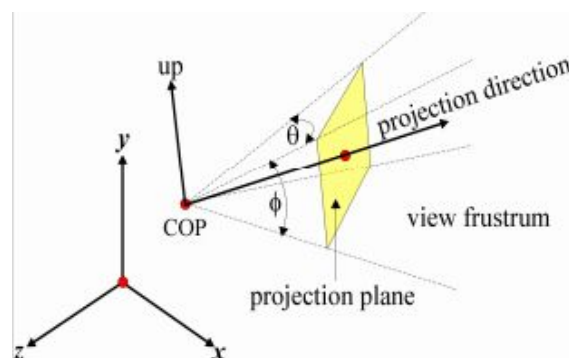
Perspective projections are **more complex** and exhibit **foreshortening**.

Foreshortening:

Parallel lines appear to **converge** at points.

Parameters:

1. **Center** of projection (COP)
2. **Field of view** (FOV) (θ, ϕ)
3. **Projection** direction
4. **Up** direction

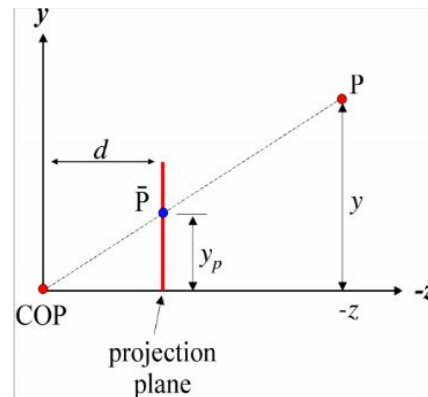


Consider a perspective projection with:

- The **viewpoint** at the **origin**.
- A **viewing direction** oriented along the positive z -axis.
- The **view plane** located at $z = -d$.

$$\frac{y}{z} = \frac{y_p}{d}$$

$$\Rightarrow y_p = \frac{y}{z/d} \quad (\text{Non-uniform shortening})$$



A similar construction for x_p :

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ z/d \\ y \\ z/d \\ -d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ -z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

↑
Transformation Matrix

Homogeneous Coordinates

Consider the matrix:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

It transforms the point:

$$\begin{bmatrix} x \\ y \\ -z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ -z \\ z/d \end{bmatrix}$$

Divide by w to return to the original 3D:

$$\begin{bmatrix} x \\ z/d \\ y \\ z/d \\ -d \\ 1 \end{bmatrix}$$

Perspective Projections

Although perspective transformations preserve lines, they are not **affine**.

$$(x, y, z) \rightarrow (x_p, y_p, z_p)$$

The transformation is **irreversible**:

- All points along a projection project onto the **same point**.
- We **cannot** recover a point from its projection.

Depending on the application, we can use different mechanisms to specify a perspective view:

1. `lookAt()`
2. `frustum()`
3. `perspective()`

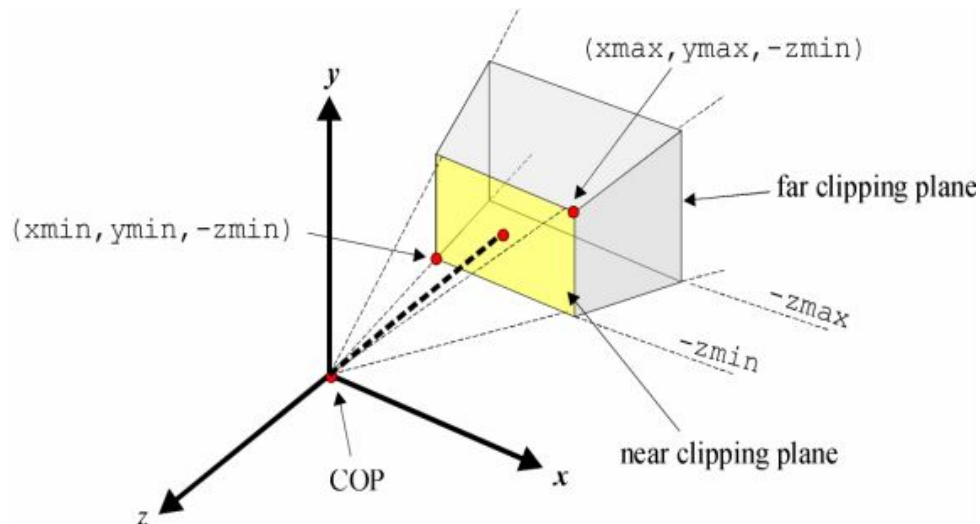
The **FOV angle** may be derived if the distance to the **viewing plane** is known.

The **viewing direction** may be obtained if a **point** in the scene is identified that we wish to look at.

Frustum

```
mat4 frustum(xmin, xmax, ymin, ymax, zmin, zmax);
```

`zmin` and `zmax` are specified as positive distances along $-z$.



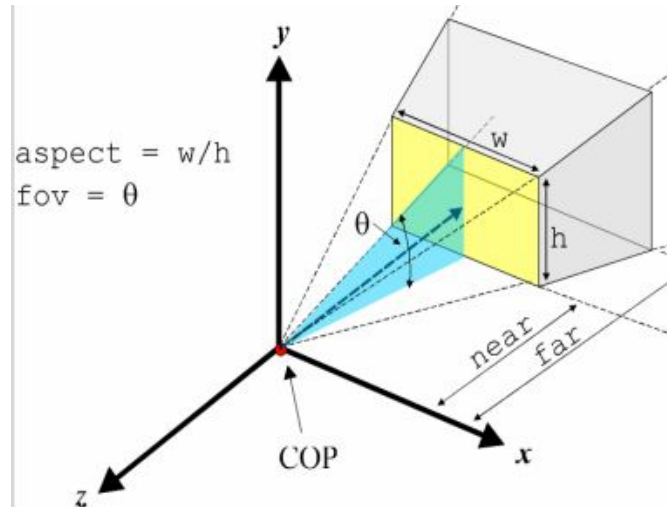
It is not necessary to have a **symmetric frustum** like:

```
frustum(-1.0, 1.0, -1.0, 1.0, 5.0, 50.0);
```

Asymmetric frustums introduce **obliqueness** into the projection.

Perspective

```
mat4 perspective(fov, aspect, near, far);
```



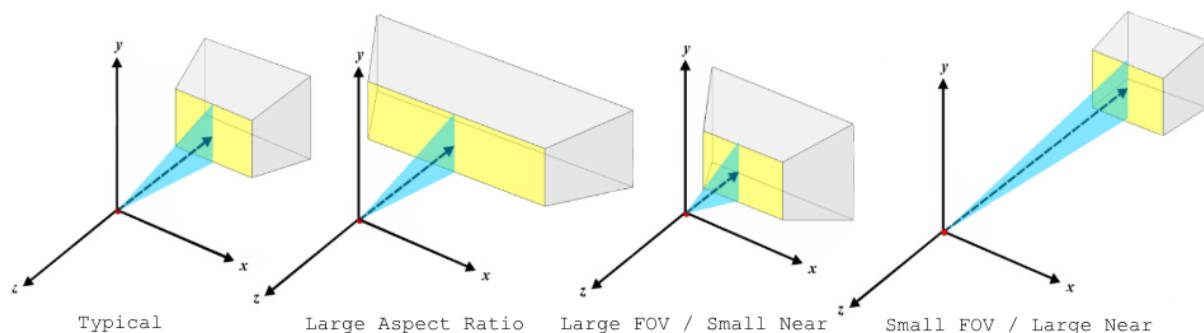
$$\frac{h/2}{\text{near}} = \tan \frac{\theta}{2}$$
$$\Rightarrow h = 2 \text{ near } \tan \frac{\theta}{2}$$

This function simplifies the specification of perspective views. It only allows the creation of **symmetric frustums**.

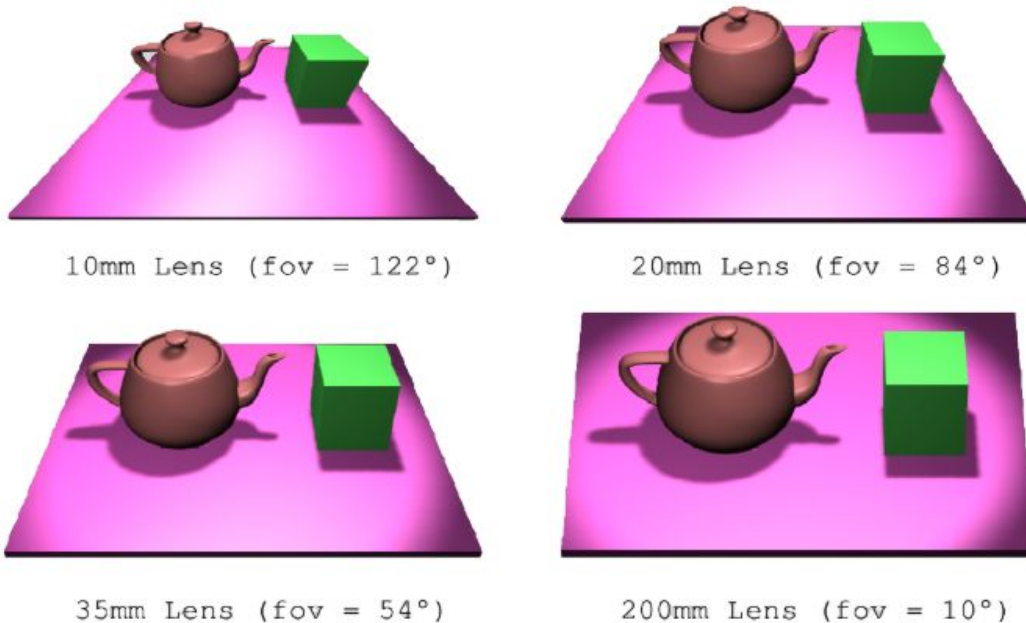
Properties:

- Viewpoint is at the **origin**.
- Viewing direction is the $-z$ -axis.
- Field of view angle must be in the range $0-180^\circ$.

The **aspect** parameter **eliminates distortion** by matching the viewport width and height.



Lens Configurations



Positioning The Camera

In previous projections we had a **fixed** origin and projection direction.

To obtain arbitrary camera orientations and positions, we manipulate the **view matrix**. This positions the camera w.r.t. the model.

If we wish to position the camera at (10, 2, 10) w.r.t. the world, we have two possibilities:

1. **Transform the world** prior to the creation of objects using translation and rotation matrices:
`translate(-10, -2, -10);`
2. Use the **lookAt function** to position the camera with respect to the world coordinate system:
`lookAt(10, 2, 10, ...);`

Both of these are equivalent.

View Matrix

Objects are positioned in a scene / world with a (0,0,0) origin.

We want to show the view from a camera, moving through the virtual world.

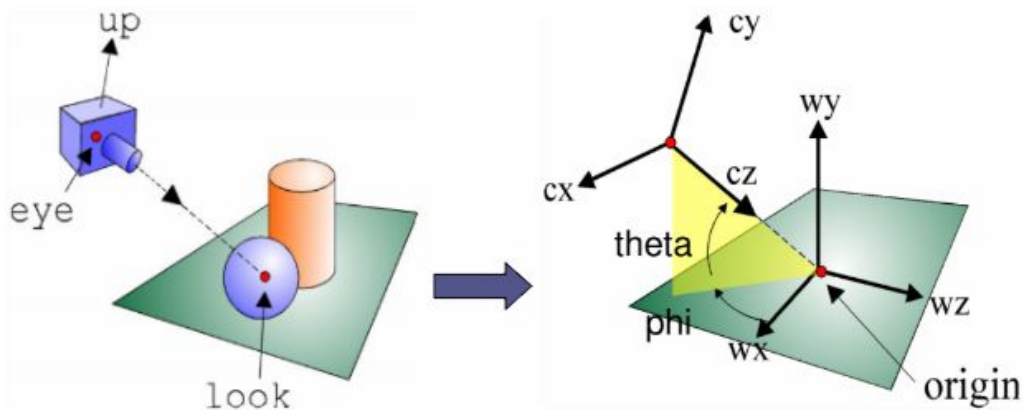
We multiply the **world-space** points by a **view** matrix to get to **eye-space**.

```
mat4 V = R * T; // 1. Inverse of camera position and angle.  
mat4 V = lookAt(pos, target, up); // Or 2.
```

```
vec4 pos_eye = V * pos_world;
```

Positioning The Camera

```
lookAt(eyeX, eyeY, lookX, lookY, lookZ, upX, upY, upZ);
```

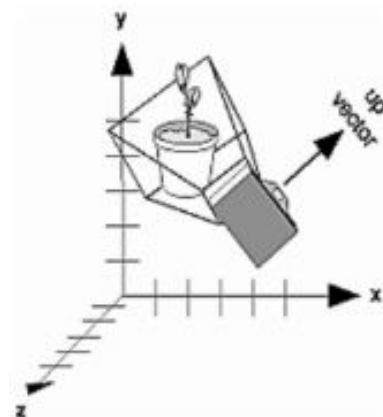
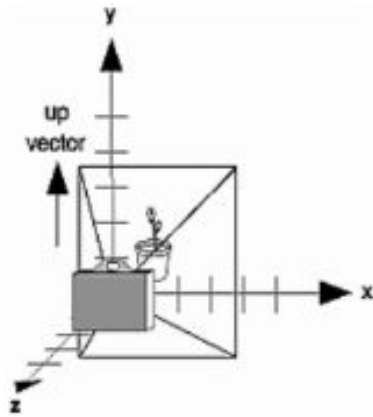


This is equivalent to:

```
translate(-eyeX, -eyeY, -eyeZ);  
rotate(theta, 1.0, 0.0, 0.0);  
rotate(phi, 0.0, 1.0, 0.0);
```

Up Vector

The **up vector** tells us which direction is up.
It must be **perpendicular** to the line of sight.



LookAt

The lookAt function is useful when **panning** across a scene.

Viewport

The **projection matrix** defines the mapping from a 3D world coordinate → 2D viewport coordinate.

The viewport extents are defined as a parameter of the projection

`frustum(left, right, bottom, top, near, far)`



`perspective(fov, aspectRatio, near, far)`

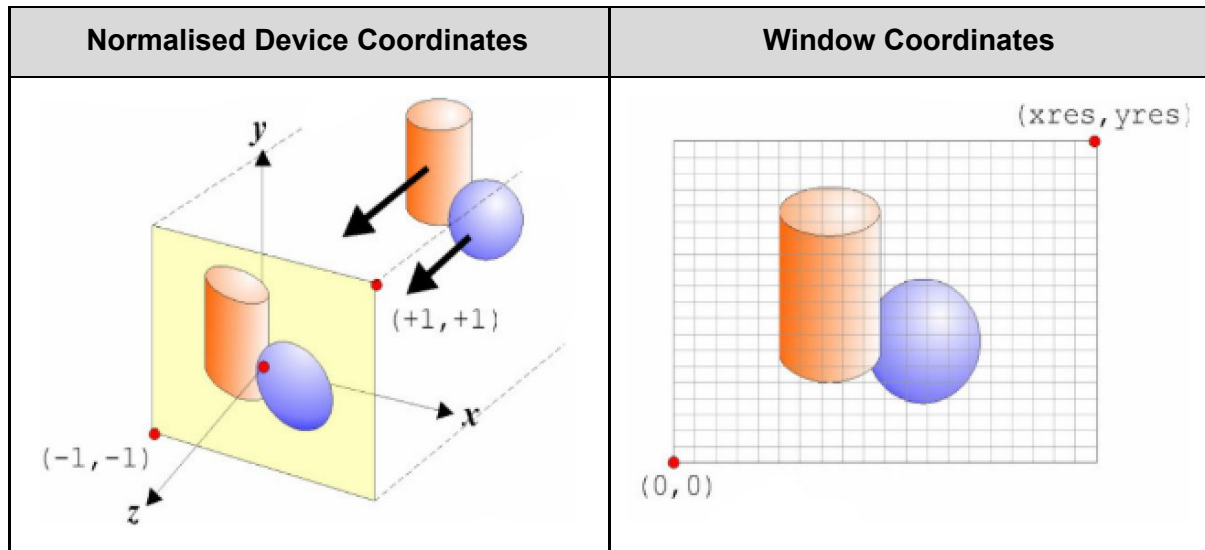
$$height = near \tan \frac{fov}{2}$$

$$width = height \times aspectRatio$$



Viewport to Window Transformation

We need to associate the 2D viewport coordinate system with the window coordinate system to determine the **correct pixel** associated with each vertex.



We use an **affine planar transformation**.

After the projection to the view-plane, all points $([-1, +1], [-1, +1])$ are transformed to normalised device coordinates:

$$x_n = 2 \left(\frac{x_p - x_{min}}{x_{max} - x_{min}} \right) - 1$$

$$y_n = 2 \left(\frac{y_p - y_{min}}{y_{max} - y_{min}} \right) - 1$$

We use `glViewport` to relate coordinate systems:

```
void glViewport(int x, int y, int width, int height);
```

Width and height are the dimensions of the viewport:

$$x_w = (x_n + 1) \left(\frac{width}{2} \right) + x$$

$$y_w = (y_n + 1) \left(\frac{height}{2} \right) + y$$

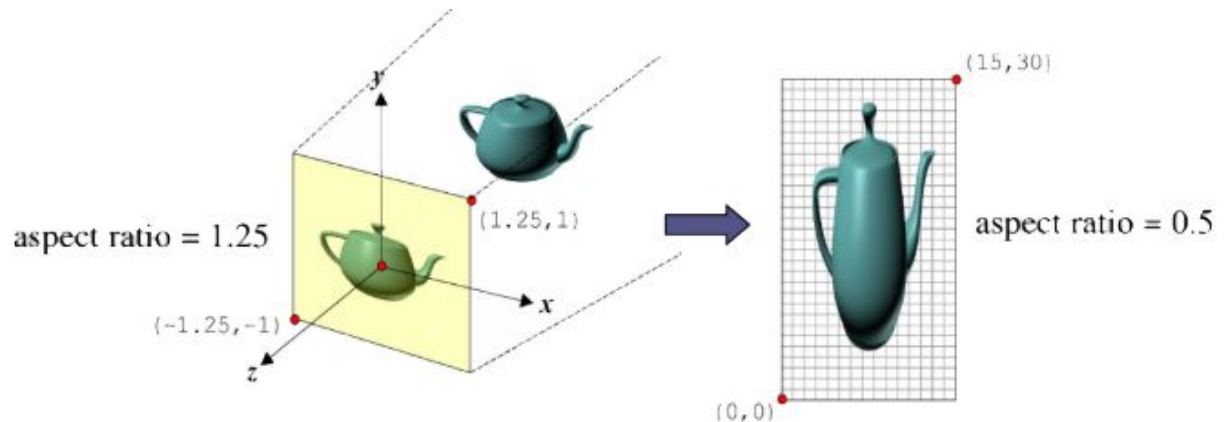
Aspect Ratio

The aspect ratio defines the relationship between the **width** and **height** of an image.

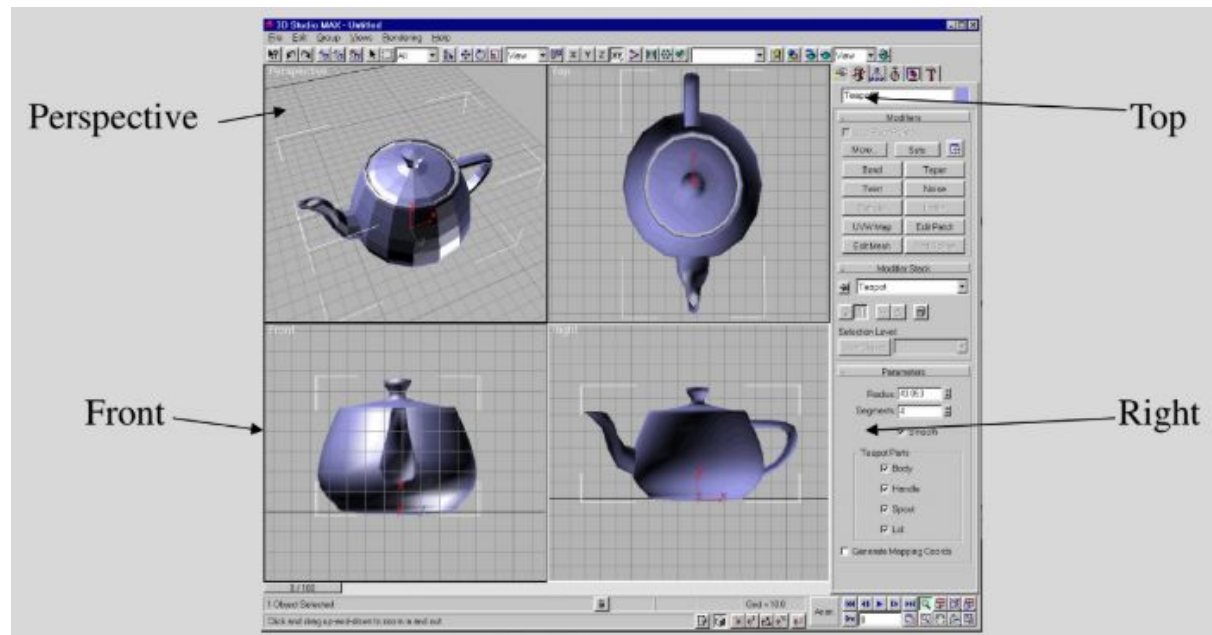
Using a perspective matrix, a viewport aspect ratio can be provided.

Otherwise, the aspect ratio is a function of the supplied **viewport width** and **height**.

Unmatched window and viewport aspect ratios will cause **affine distortion**:



Multiple Projections

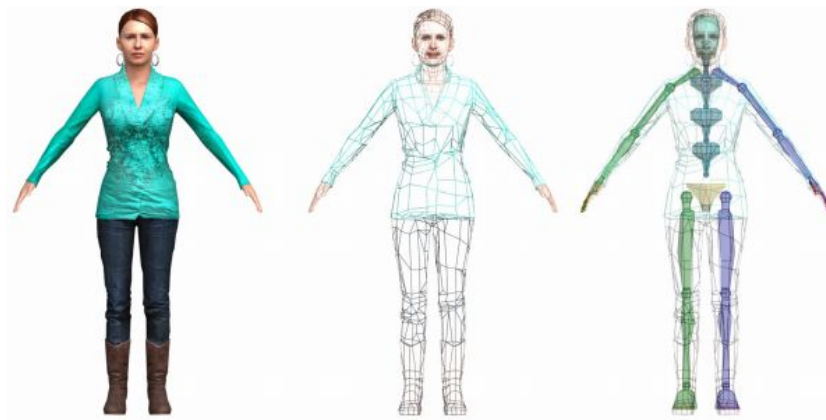


05 Hierarchical Modelling

Character Animation

Animating a character model described as a polygon mesh by **moving each vertex** is **impractical**.

Instead, we specify the motion of characters through the movement of an internal articulated **skeleton**.



Rigid Body Limitations

When human joints bend, the body shape bends as well. There are **no distinct parts**.

We cannot represent human joints with **rigid bodies**, else the pieces would **separate**.



Relative Motion

We are interested in animating objects whose motion is **relative** to another.
Such a sequence is called a **motion hierarchy**.

Components of a hierarchy represent objects that are **physically connected** / linked.

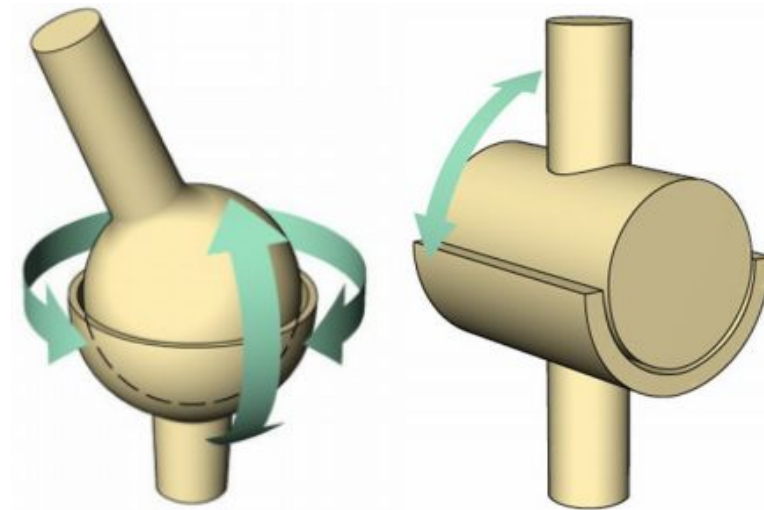
In some cases, motion can be restricted:

- Reduced dimensionality.
- Hierarchy enforcing constraints.

There are two approaches for **animating figures** defined by hierarchies:

1. **Forward** kinematics
2. **Inverse** kinematics

Degrees of Freedom



Model Transformations

A **local frame view** is usually adopted as it extends naturally to the specification of **hierarchical model frames**.

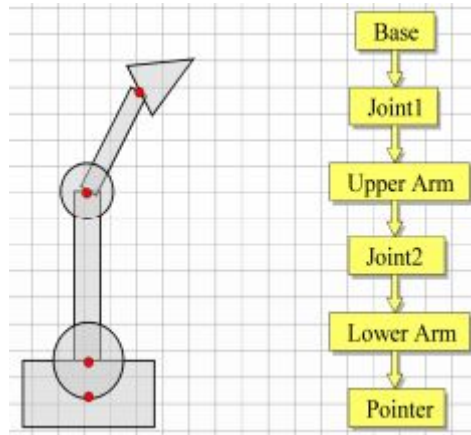
This allows the creation of **jointed assemblies**: articulated figures (e.g. animals, robots etc).

In a hierarchical model, each sub-component has its own **local frame**.
Changes to the **parent frame** are propagated down to the **child frames**.

This simplifies the specification of animation.

Hierarchical Transformations

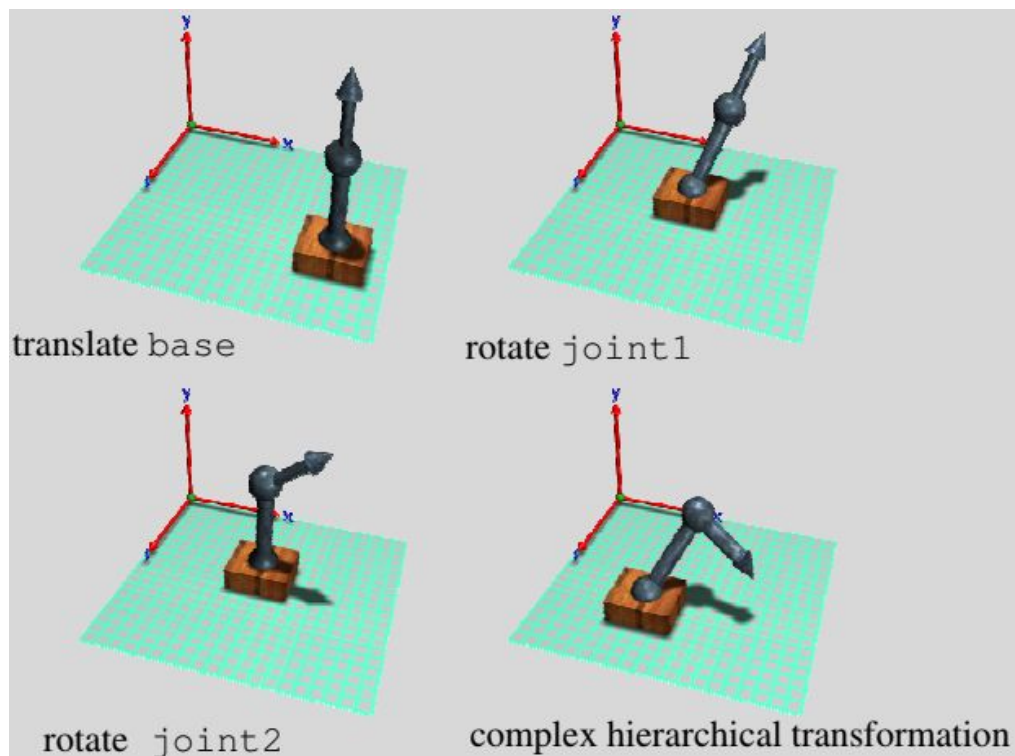
For geometries with an implicit hierarchy, we wish to associate **local frames** with **sub-objects** in the assembly.



Parent-child frames are related via a **transformation**.
Transformation linkage is described as a **tree**.

Each node has its own **local coordinate system**.

Hierarchical transformation allows **independent control** over subparts of an assembly:



OpenGL Implementation

```
local1 = identity_mat4();
local1 = rotate(base_orientation) * local1;
local1 = translate(bx, by, bz) * local1;
global1 = local1;

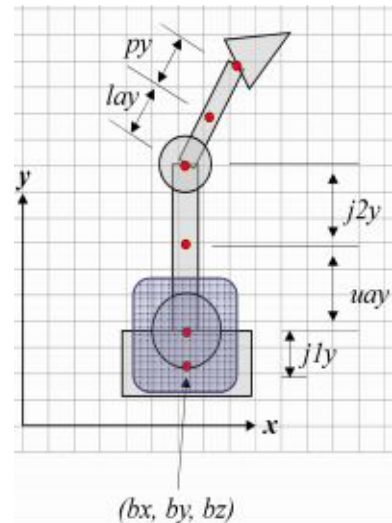
updateUniformVariables();
drawBase();

local2 = identity_mat4();
local2 = rotate(joint1_orientation) * local2;
local2 = translate(0, j1y, 0) * local2;
global2 = local1 * local2;

updateUniformVariables();
drawJoint1();

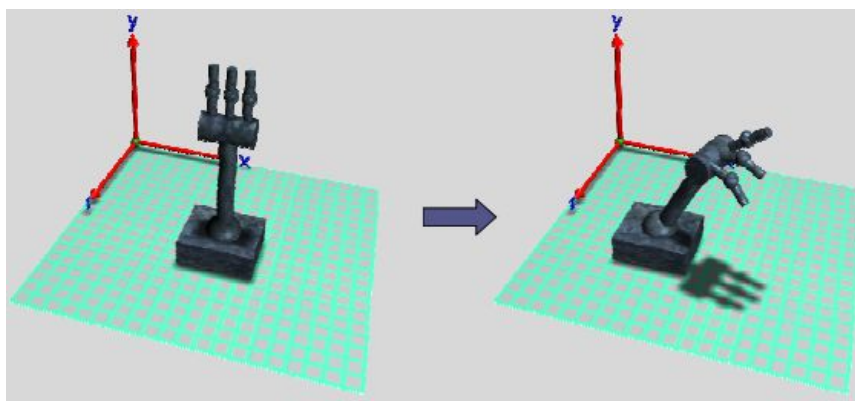
local3 = identity_mat4();
local3 = rotate(upperArm_orientation) * local3;
local3 = translate(0, uay, 0) * local3;
global3 = local1 * local2 * local3;

updateUniformVariables();
drawUpperArm();
```

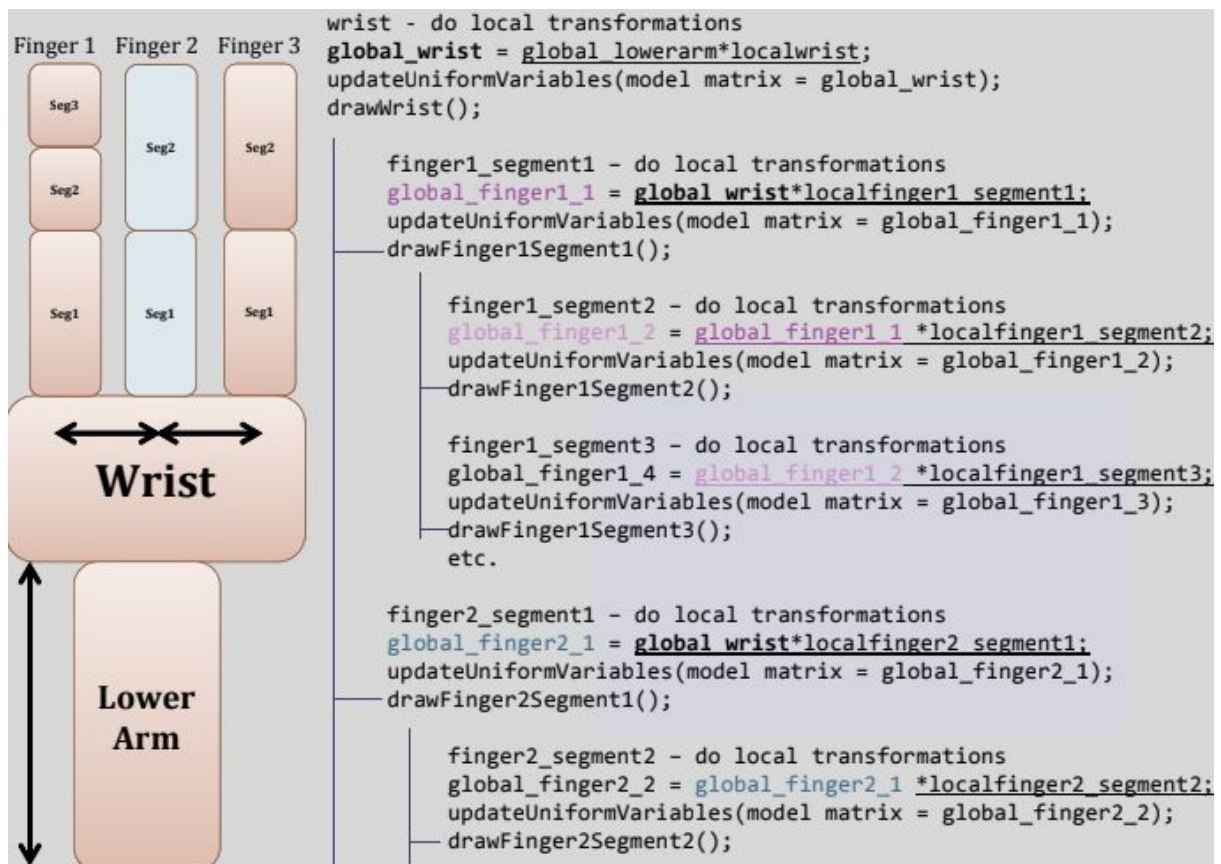
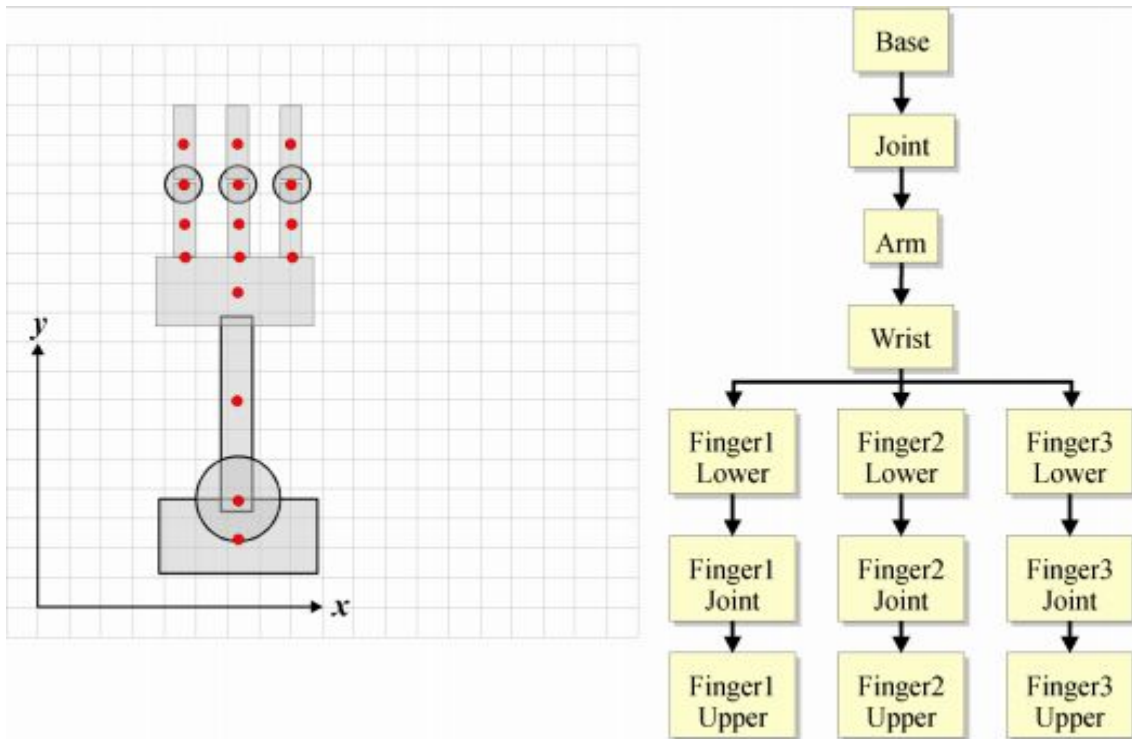


This example has simple one-to-one parent-child linkages.
In general there may be **many** child frames derived from a single parent frame.

We need to remember the parent frame and return to it when creating new children. We store a global transformation as we go.



Each finger is a **child** of the wrist. This gives **independent control** over the orientation of the fingers relative to the wrist.



Link to [Part Two](#)