# COMPUTER GRAPHICS

by Yannick Giovanakis

July 23, 2018

# Contents


**1 Graphic Adapters** **5**

1.1 Vector Graphic Adapters . . . . . 5

1.2 Raster Graphic Adapters . . . . . 5

1.3 Accelerated Graphic Adapters . . . . . 6

1.4 Color vision . . . . . 7

1.4.1 Human Vision . . . . . 7

1.4.2 Color reproduction . . . . . 8

1.5 Image Resolution . . . . . 9

**2 2D Graphics** **10**

2.1 Point primitives . . . . . 10

2.1.1 Linear Interpolation . . . . . 11

2.2 Line primitives . . . . . 12

2.2.1 Interpolation Algorithm . . . . . 13

2.2.2 Bresenham Algorithm . . . . . 14

2.3 Triangle Primitives . . . . . 17

2.3.1 Triangles with edge // to x-axis . . . . . 17

2.3.2 Triangle splitting . . . . . 18

2.4 Normalized coordinates . . . . . 19

**3 3D Graphics** **20**

3.1 Affine Transformations . . . . . 20

3.1.1 Translation . . . . . 21

3.1.2 Scaling . . . . . 22

3.1.3 Rotation . . . . . 24

3.2 Shear . . . . . 27

**4 3D Transform** **28**

4.1 Inversion of transformations . . . . . 29

4.2 Composition . . . . . 30

4.2.1 Properties of composition of transformations . . . . . 30

4.3 Transformations around an arbitrary axis or center . . . . . 32

**5 Projections** **34**
5.1 Parallel Projections . . . 36
5.1.1 Aspect Ratio . . . 39
5.1.2 Projection matrices and aspect ratio . . . 40
5.2 Perspective Projections . . . 40
5.2.1 Perspective matrix on screen . . . 44

**6 View and World Transformations** **45**
6.1 View Matrix . . . 45
6.1.1 Look-in-direction matrix . . . 46
6.1.2 Look-at matrix . . . 48
6.2 Local coordinates and World Matrix . . . 50
6.2.1 World Matrix : scaling . . . 50
6.2.2 World Matrix : rotating . . . 51
6.2.3 World Matrix : positioning . . . 52
6.2.4 Final World Matrix . . . 53
6.3 Gimbal Lock . . . 53

**7 A complete projection example** **55**
7.1 World-View-Projection Matrices . . . 55
7.2 The example . . . 57

**8 Meshes and Clipping** **61**
8.1 Meshes . . . 61
8.1.1 Mesh encoding . . . 63
8.1.2 Indexed Primitives . . . 64
8.2 Clipping . . . 66
8.2.1 Clipping triangles . . . 68

**9 Hidden Surfaces** **71**
9.1 Back-face culling . . . 72
9.2 Z-Buffering . . . 77
9.2.1 Z-Buffering issues . . . 77
9.2.2 Stencil buffer . . . 79

**10 Lights and rendering** **81**
10.1 The Bidirectional Reflectance Distribution Function . . . . . 81
10.2 Rendering equation . . . . . 83
10.2.1 Scan-line : considering colors . . . . . 84
10.3 Light models . . . . . 85
10.3.1 Direct light models . . . . . 85
10.3.2 Point Lights . . . . . 86
10.3.3 Spot lights . . . . . 88
10.4 BRDF Models . . . . . 89
10.4.1 Diffuse: Lambert Model . . . . . 89
10.4.2 Specular: Phong Model and Blinn Model . . . . . 91

**11 Smooth Shading** **96**
11.1 Gouraud shading . . . . . 98
11.2 Phong shading . . . . . 99
11.3 Transformations of normal vectors . . . . . 99

**12 Ambient Light Emissions** **100**

**13 Textures** **102**
13.0.1 Perspective interpolation . . . . . 105

**14 UV Mapping** **109**
14.1 Magnification Filtering . . . . . 110
14.1.1 Nearest Pixel . . . . . 110
14.1.2 (Bi)Linear Interpolation . . . . . 111
14.2 Minification Filtering . . . . . 112
14.2.1 MultiploInParving-Mapping . . . . . 112
14.3 UV Intervals . . . . . 113
14.3.1 Clamp . . . . . 113
14.3.2 Repeat . . . . . 114
14.3.3 Mirror . . . . . 114
14.3.4 Constant . . . . . 115

**15 Animations** **117**
15.1 Quaternions . . . 117
15.2 Bezier Curves . . . 119
15.3 3D Animation . . . 121

**16 WegGL & Shaders** **124**
16.1 Intro . . . 124
16.2 GLSL . . . 124
16.3 WebGL - Client . . . 125
16.3.1 Canvas . . . 125
16.3.2 Context . . . 125
16.3.3 Drawing with 2D Context . . . 126
16.3.4 Drawing with webgl Context . . . 127
16.3.5 Drawing with webgl context : 3D . . . 130
16.4 GLSL - Server . . . 132
16.4.1 Vertex Shader . . . 133
16.4.2 Fragment Shader . . . 133
16.4.3 Data Types . . . 134
16.4.4 GLSL Program Language . . . 135
16.4.5 GLSL Compiling . . . 138
16.4.6 Workflow Recap . . . 139
16.5 Shader spaces . . . 139
16.5.1 World space . . . 140
16.5.2 Camera space . . . 141
16.5.3 Object space . . . 142
16.5.4 Static space . . . 143
16.5.5 Space recap . . . 143
16.6 GLSL Textures . . . 144
16.6.1 Using texture : steps . . . 145

# 1 Graphic Adapters

$$\text{Graphic Adapters Overview} = \begin{cases} Vector \\ Raster \\ Accelerated \end{cases}$$

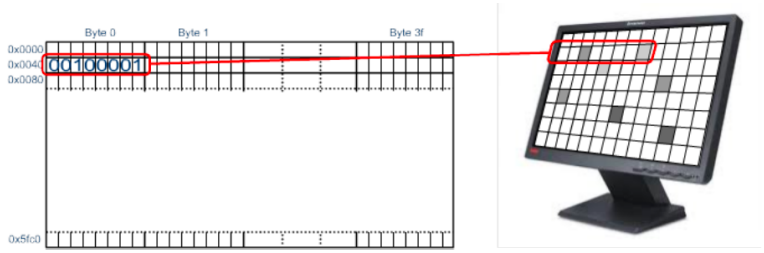
## 1.1 Vector Graphic Adapters

This type of adapters are old-fashioned and not used any more. The technology used is similar to the one in oscilloscopes : a moving , turned on/off **beam** in a CRT used to draw objects on a screen.
Used mainly in '70s in high-end visualisation tools , later in arcade gaming machines ( ex: Atari Battlezone ) and even used in the Vectrex , a home entrainment system.
Graphics are drawn as a **set of commands** sent from software to hardware (adapter). The commands are used to generate **3 analog** signals that control horizontal & vertical positions and the beam intensity.


## 1.2 Raster Graphic Adapters

Raster graphic adapters divide the screen into a **matrix** of individually assignable elements called **pixels** which are assigned **colors.** If the color comes from a spatial sampling of an image the adapter can reproduce it on the monitor.
Raster adapters have a special memory called Video Memory (**VRAM**) made of cells that contain information about the color of each pixel on the screen. A

component on the video card ( RAMDAC for analog displays ) converts the information to the signal required to transfer the image to the monitor.
Images on the screen can be written by setting specific values in the VRAM ( `writeScreen()` function). Still in used but slowly dismissed due to reduction of




hardware costs of better technologies. Initially the memory was just enough to store a single screenshot.

## 1.3 Accelerated Graphic Adapters

Are a special kind of raster graphic adapters that have **much more memory** than the one required to store a single screenshot. Instead of writing **directly on the screen buffer** ,images are stored in different areas of the VRAM.
Commands that can be interpreted by the adapter include :

- Draw points,lines and other figs
- Write text
- Transfer raster images from VRAM to screen buffer
- 3D projections
- Deform + effects on images

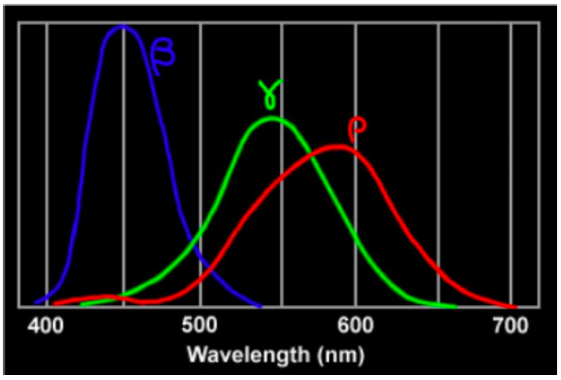
Used today , these adapters can perform complex tasks ( multi - display screening, stereoscopic images ..)

## 1.4 Color vision

How is color on-screen encoded in bits? Commonly a system called **RGB** is used. In the following sections we'll find why.

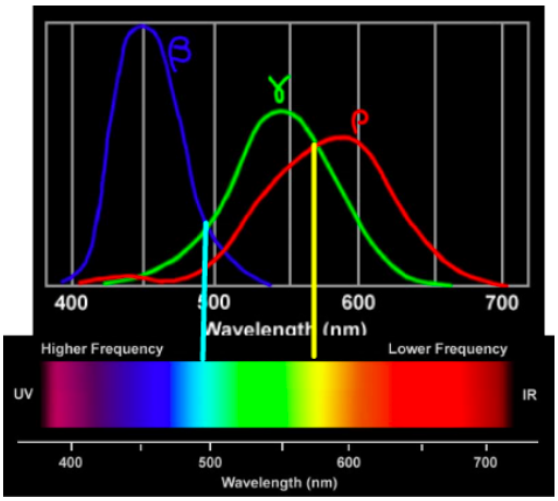
### 1.4.1 Human Vision

The color of the light is determined by the **wavelength** of the photons that transmit it. Visible light ranges from 400-700 nm wavelength.
Depending on the **light source** , lots of photons of **different wavelengths** are emitted. The photons then interact with the environment where objects, depending on their composition , **reflect or absorb** the various wavelengths at different intensities. The reflected photons are then focused on the retina where **rods** (sensible to light intensity) and **cones** (sensible to light color) transmit information to the brain through **nerves**.
There are 3 types of cones $\rho, \beta, \gamma$ each sensible to a different portion of light spectrum.
By combing the stimuli of different cones the brain allows the vision of a given color.



### 1.4.2 Color reproduction

Color reproduction uses the inverse procedure of the human vision :
it associates a different **emitter** for each color the human cones can capture. Since the main wavelengths perceived by the cones are **red, green & blu** , different hues are constructed by mixing light of these three.
Mixing two of three primary colors **cyan, magenta and yellow** are obtained.
Mixing the three in different proportions **all** possible hues can be obtained.



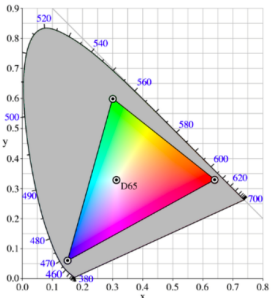

**Color range**

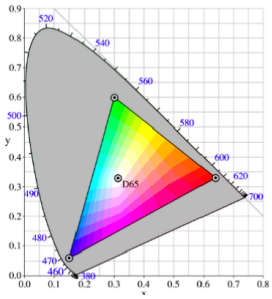
The **spectrum** that a monitor can produce is just a small portion of the entire spectrum the eye can see (grey area).
Color range of a monitor corresponds to a cube where the three colors are placed on the 3 axis.

**Color synthesis**

The levels of red green and blue are translated into three **electrical signals** whose intensity controls the light emitted by the screen for each of the primary colors.

As the system is digital **DACs** are used to **quantize** the signals. Quantisation further **reduces** the number of visible colors : quantisation levels are usually $2^{d/3}$ with **d** being the number of bits per pixel




## 1.5 Image Resolution

The resolution of an image defines the **density of pixels** that compose it. When dealing with **raster graphics on screen** the density is relative to the **monitor size** : the resolution defines thus the number of pixels displayed on the screen on the horizontal (**width** ) and vertical (**height**
Pixels are not always **square shaped** so the horizontal resolution $\neq$ vertical resolution.

### Memory of Images

Raster images require lots of memory : FullHD up to 6MB

$$\text{Memory} = w * h * d_{bits}$$

A first way to reduce image size was to reduce the **colors** using a **color palette** : a predefined set of colors that contains a **limited** number of entries.
The palette is encoded as an array of **RGB values** that are used to define the possible colors that can be used in an image.
If the **color depth d** (number of bits per pixel) is $\leq 8$ a color palette is used ( the palette contains $2^d$ colors).
If a user defined palette is used, the size of the palette must be added to the image size. With p = bits to encode a palette entry

$$\text{Memory} = w * h * d + p * 2^d_{bits}$$

# 2 2D Graphics

2D Graphics primitives are procedures that draw simple geometric shapes based on a **2D coordinates system** , a set of integer with unit the **pixel** → *pixel coordinates*.
The coordinate system is **Cartesian** , with the origin on the **left-top** ranging from

$$0 \leq x \leq s_w - 1$$

$$0 \leq y \leq s_h - 1$$

A **clipping** procedure avoids that coordinates go out of boundaries ,causing **wrap-arounds** or writing of **un-allocated memory space**


Figure 1: Axis disposition

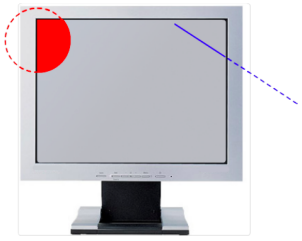


Figure 2: clipping

## 2.1 Point primitives

The **point** is the simplest 2D primitive which consists in setting a **pixel** in given **position & color**.
Generally the graphic primitive that draws the point is called `plot()` but the **actual** way in which a point is drawn is **hardware dependent** : every adapter has its own plot() algorithm.

### 2.1.1 Linear Interpolation

A very simple numerical way to compute **intermediate points** giving **two** known points is called **interpolation** :

$$I(x_0, x, x_1, y_0, y_1) = y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

where $(x_0, y_0), (x_1, y_1)$ are the known values.
Interpolation can be used to find N-1 intermediate points, equally spaced among two points $(x_0, y_0), (x_N, y_N)$ :

$$I(0, i, N, y_0, y_N) = y_i = y_0 + \frac{y_N - y_0}{N}i$$

.
Alternatively $y_i$ can be found **recursively** starting from $y_{i-1}$ :

$$y_i = y_0 + \frac{y_N - y_0}{N}i \text{ where } dy = \frac{y_N - y_0}{N} \rightarrow y_1 = y_{i-1} + dy$$

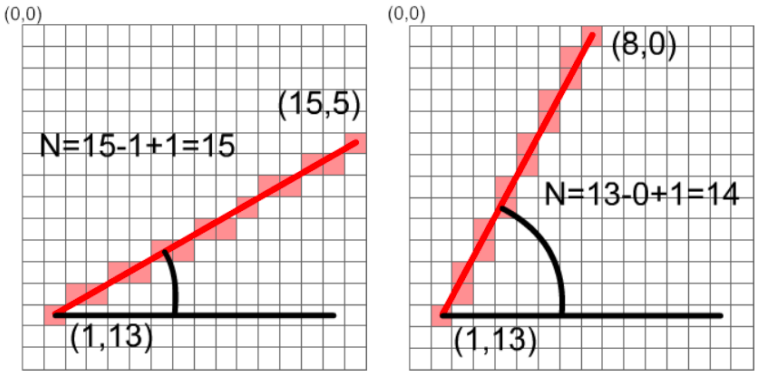
## 2.2 Line primitives

The **line primitives** connect **two points** $(x_0, y_0), (x_1, y_1)$ on screen with a straight segment. Each pixel that composes the line (except for $p_0, p_1$ must touch another pixel to keep the line continuous. The **number of pixels** involved depends on the **angle** between the line and the x-axis :

$$N = max(|x_1 - x_0|, |y_1 - y_0|) + 1$$

which corresponds to

$$\text{Number of pixels} = \begin{cases} \theta < 45 & |x_1 - x_0| + 1 \\ \theta > 45 & |y_1 - y_0| + 1 \end{cases}$$




After finding the number of required pixels, different algorithms perform the line drawing task. Two main algorithms are :

- Interpolation algorithm : floating points operations (good on modern hardware)
- Bresenham algorithm : integer operations (good on old or embedded/ special purpose hardware)

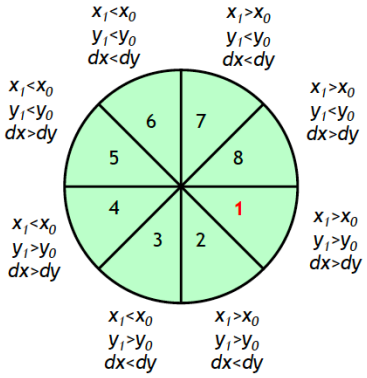
### 2.2.1 Interpolation Algorithm

A popular line drawing algorithm is the Interpolation algorithm.

```
{
   if( |x1-x0| >= |y1-y0|){ //Angle >45 o <45?
      if(x0 > x1){          // to get smallest value as index in for loop
         swap(x0,y0,x1,y1);
      }
      y=y0;
      dy = (y1-y0)/(x1-x0); //interpolation increment (can be negative!!)
      for(x=x0;x<=x1;x++){
         plot(x,round(y),c); //rounding to nearest int
         y += dy;
      }
   }else{
         if(y0 > y1){
         swap(x0,y0,x1,y1);
      }
      x=x0;
      dx = (x1-x0)/(y1-y0);
      for(y=y0;y<=y1;y++){
         plot(round(x),y,c);
         x += dx;
      }
}
```

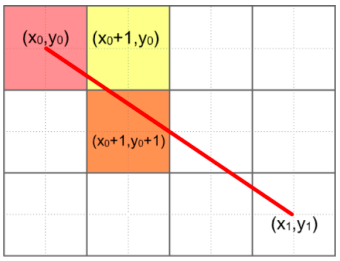
### 2.2.2 Bresenham Algorithm

The algorithm has 8 different implements depending on which **octant** the points lie :




Using octant 1 a example :

1. Step. : Find the number of pixels $N = max|x_1 - x_0|, |y_1 - y_0| + 1$
2. Step. : First pixel drawn in its position
3. Step. : Depending on the slope select the feasible pixels




4. Step. : Select the pixels whose center is closer to the line ( orange pixel )
5. Step. : The process is repeated from step 2 until the end is reached. At each iteration y (in this case) **remains constant** or **increases** by one depending on whether the distance from the previous pixel is greater than 0.5. If the distance is greater y is increased and the distance is reset by one.

```
{
   dy = (y1-y0)/(x1-x0);
   dist = 0 ;
   y=y0;
   plot(x,y,c);
   for(x=x0+1;x<=x1;x++){
      dist += dy;
      if(dist > 0.5){
         y++;
         dist = dist-1;
      }
      plot(x,y,c);
   }
}
```

The algorithm above used **floats** fro computation which is not what we wanted. The integer version of the algorithm is obtained by **multiplying** all terms considering the distance times **2(x1-x0)**:

```
{
   dy = 2(y1-y0);
   dx = x1-x0;
   idist = 0 ;
   y=y0;
   plot(x,y,c);
   for(x=x0+1;x<=x1;x++){
      idist += dy;
      if(idist > dx){
         y++;
         idist -= 2dx;
      }
      plot(x,y,c);
   }
}
```

Obviously the algorithm changes slightly for the other octants.

**1**

```
dy = 2*(y1 - y0);
dx = x1 - x0;
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(x = x0+1; x <= x1; x++) {
    idist += dy;
    if(idist > dx) {
        y++;
        idist -= 2 * dx;
    }
    plot(x, y, c);
}
```

**8**

```
dy = 2*(y0 - y1);
dx = x1 - x0;
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(x = x0+1; x <= x1; x++) {
    idist += dy;
    if(idist > dx) {
        y--;
        idist -= 2 * dx;
    }
    plot(x, y, c);
}
```

**4**

```
dy = 2*(y1 - y0);
dx = x0 - x1;
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(x = x0-1; x >= x1; x--) {
    idist += dy;
    if(idist > dx) {
        y++;
        idist -= 2 * dx;
    }
    plot(x, y, c);
}
```

**5**

```
dy = 2*(y0 - y1);
dx = x0 - x1;
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(x = x0-1; x >= x1; x--) {
    idist += dy;
    if(idist > dx) {
        y--;
        idist -= 2 * dx;
    }
    plot(x, y, c);
}
```

**2**

```
dy = y1 - y0;
dx = 2*(x1 - x0);
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(y = y0+1; y <= y1; y++) {
    idist += dx;
    if(idist > dy) {
        x++;
        idist -= 2 * dy;
    }
    plot(x, y, c);
}
```

**3**

```
dy = y1 - y0;
dx = 2*(x0 - x1);
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(y = y0+1; y <= y1; y++) {
    idist += dx;
    if(idist > dy) {
        x--;
        idist -= 2 * dy;
    }
    plot(x, y, c);
}
```

**7**

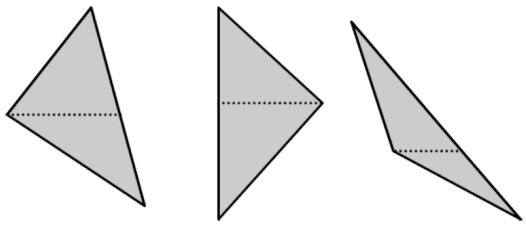
```
dy = y0 - y1;
dx = 2*(x1 - x0);
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(y = y0-1; y >= y1; y--) {
    idist += dx;
    if(idist > dy) {
        x++;
        idist -= 2 * dy;
    }
    plot(x, y, c);
}
```

**6**

```
dy = y0 - y1;
dx = 2*(x0 - x1);
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(y = y0-1; y >= y1; y--) {
    idist += dx;
    if(idist > dy) {
        x--;
        idist -= 2 * dy;
    }
    plot(x, y, c);
}
```

## 2.3 Triangle Primitives

Triangles are very important as they're the basis for **3D** computer graphics. Triangles having one **edge parallel to the horizontal** axis is the **easiest** to draw. Other triangles can be split in 2 to obtain easy to draw triangles.

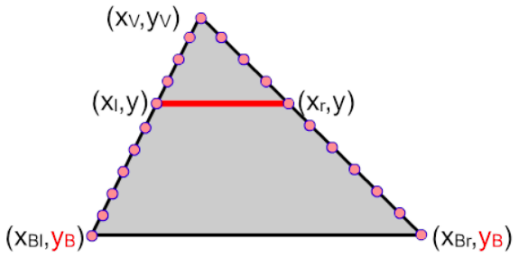


### 2.3.1 Triangles with edge // to x-axis

Triangles of this type are characterized by 5 values:

- $x_v$ ,$y_v$ coordinates of the vertex
- $y_B$ vertical coordinates of the base
- $x_{Bl}, x_{Br}$ horizontal coordinates of the base

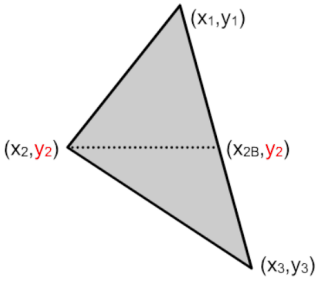
The two edges not parallel to x-axis can be considered as two lines.// The triangles is **filled** by drawing horizontal lines that connect pixels over the angled edges. The $x_l, x_r$ coordinates can be found using **interpolation**.



```
{  /*since the two lines have diff. slope , store in
   dxr ,dxl the increments in the x direction */
   dxl = (xBl - xv)/(yB - yv);
   dxr = (XBr - xv)/(yB - yv);
   xl = xr = xv; // starting from vertex
   /* assumption yv < yB , otherwise change loop direction */
   for ( y = yv; y <=yB ; y++) {
      for(x=round(xl);x<=round(xr); x++){
         plot(x,y,c)
      }
      xl += dxl;
      xr += dxr;
   }
}
```

### 2.3.2 Triangle splitting

More complex triangles can be split in order to obtain two triangles with edges parallel to x-axis.




An easy way to find the middle point (x2,y2) is to take the three points and sort them through the y-axis. The one in the middle is the middle point.
To find the corresponding point on the opposing edge :

- **y-coordinate** is the same as in point (x2,y2)

- **x-coordinate** is obtained via **interpolation**:

$$x_{2B} = I(y_1, y_2, y_3, x_1, x_3)$$

## 2.4 Normalized coordinates

Current displays are available in different **resolutions** and **sizes**. Moreover in windowed operating systems applications must be **confined** only in a portion of the screen. When changing the resolutions/window size the applications still want to show the **same image** exploiting all the features of the display. A special coordinates system called **Normalized Screen Coordinates** is normally used to address points on screen in a device in an independent way.
NSC are Cartesian coordinate system where x and y range between to **canonical values** [ OpenGL -1 $\rightarrow$ 1 ]. If the window/memory area resolution is known in $s_w, s_h$ pixels, the coordinates system $(x_s, y_s)$ can be derived from the normalized screen coordinates $(x_n, y_n)$ :

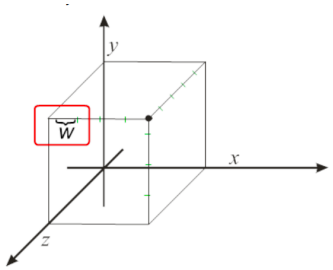
$$x_s = (s_w - 1) * (x_n + 1)/2$$

$$y_s = (s_h - 1) * (1 - y_n)/2$$

# 3 3D Graphics

To define a point in a 3D-space , 3 coordinates are typically used . However in computer graphics **4** coordinates $x, y, z, w$ called **homogeneous coordinates** are used :

- $x, y, z$ are used to define the **point in the 3D space**
- $w$ defines a **scale**, the unit of measure used by the coordinates




Consequence of using 4 coordinates $\rightarrow$ **infinite** number of coordinates define the same point , in particular all tuples of four values that are **linearly dependent** represent the same point in 3D space :

$$(2, 2, 2.5, 0.5), (4, 4, 5, 1) : (4, 4, 5, 1) = 2(2, 2, 2.5, 0.5)$$

The **real position** of a point in 3D spcae is defined by $w = 1$. To obtain the real position a simple division of **w** is sufficient.

$$(x, y, z, w) \rightarrow (x', y', z') = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$$

## 3.1 Affine Transformations

The process of varying the coordinates of the points of an object in the space is called **transformation**.
Transformations can be very **complex** since all points should be repositioned in a 3D space .
A large **set of transformations** with a mathematical concept called **affine transformations**
Objects in 3D space are defined by the coordinates of their points.By applying affine transformations to the coordinates 4 different transformations can be done :

- **Translation**
- **Scaling**
- **Rotation**
- **Shear**

The new object is drawn using the new points and the corresponding **primitives**.

$$p = (x, y, z, w) \rightarrow p' = (x', y', z', w')$$

To express transformations $4x4$ matrices M can be used. So the new point p' can be obtained by multiplying p by the **transformation matrix** :

$$p' = M \cdot p^T$$

or

$$p' = p \cdot M^T$$

depending on the **convention**

### 3.1.1 Translation

Moves the points of the object while maintaining its **size & orientation**. Translation can be performed along the three axis : $dx, dy, dz$ are the quantities that define how much the object is being moved :

$$x' = x + dx$$

$$y' = y + dy$$

$$z' = z + dz$$

By using the 4th coordinate the **translation matrix** can be obtained:

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ z + dz \\ 1 \end{bmatrix}$$

### 3.1.2 Scaling

Scaling modifies the **size of an object** while maintaining constant **position and orientation**.It can have different effects :

- **Enlarge**
- **Shrink**
- **Deform** ex:Sphere → rugby ball
- **Mirroring** ex: Object on the right → symmetrical on the left

Scale transformations have **a center** : a point that is **not moved** during the transformation. The center of transformation can be anywhere on the 3D space ( also outside the object!).
Now we assume that the center corresponds to the **origin**.

- **Proportional scaling**
  Enlarges or shrinks the object of the same amount **s** in all the directions : this leaves the proportions intact.

$$x' = s \cdot x$$

$$y' = s \cdot y$$

$$z' = s \cdot z$$

  If $s > 1 \rightarrow$ **enlarge**
  Else $0 < s < 1 \rightarrow$ **shrink**

- **Non proportional scaling**
  Deforms an object by using different scaling factors $s_x, s_y, s_z$ that allows shrinking or enlarging in different directions.

$$x' = s_x \cdot x$$

$$y' = s_y \cdot y$$

$$z' = s_z \cdot z$$

If $s_i > 1 \rightarrow$ **enlarge**
Else $0 < s_i < 1 \rightarrow$ **shrink**
A **scaling matrix** can be used to represent transformations :

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Mirroring**
  By using **negative scaling factors** mirroring can be obtained. Three different types exist in 3D space:

  1. **Planar**
     Creates a symmetric object with respect to a **plane** by assigning **-1** scaling factor to the axis **perpendicular to the plane**

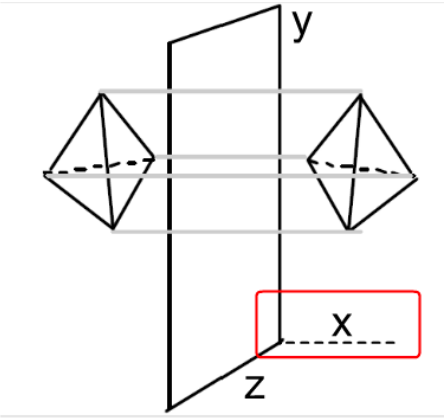



Figure 3: $s_x = -1, s_y = 1, s_z = 1$

2. **Axial**
   Creates a symmetric object with respect to a **axis** by assigning **-1** to all scaling factors **except** the one of the axis.

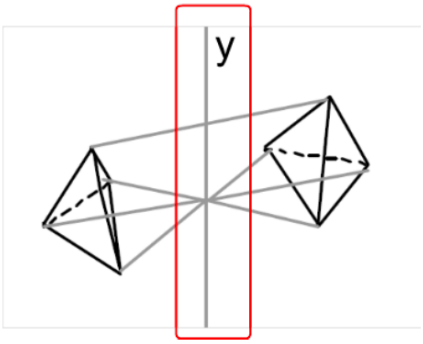



Figure 4: $s_x = -1, s_y = 1, s_z = -1$

3. **Central**
   Creates a symmetric object with respect to the **origin**.It is obtained by assigning **-1** to all scaling factors.

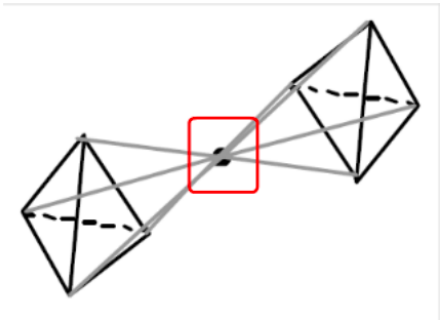


Figure 5: $s_x = -1, s_y = -1, s_z = -1$

Notice that if a scaling factor of **0** is chosen it **flattens** the image along that axis. This makes the transformation matrix **not invertible**

### 3.1.3 Rotation

Varies the objects **orientation** leaving unchanged its **position and size**. Rotation happens along a chosen axis , a line where points are **unaffected** by the

transformation.
Rotation can occur also on non conventional axis but we will mainly consider rotations along x,y,z axis passing through the origin.
A rotation of angle $\alpha$ about the z-axis :

$$x' = x \cdot cos\alpha - y \cdot sin\alpha$$

$$y' = x \cdot sin\alpha + y \cdot cos\alpha$$

$$z' = z$$

As the z-axis is the **axis of rotation** its points remain unchanged. A rotation of angle $\alpha$ about the y-axis :

$$x' = x \cdot cos\alpha + z \cdot sin\alpha$$

$$y' = y$$

$$z' = -x \cdot sen\alpha + z \cdot cos\alpha$$

A rotation of angle $\alpha$ about the x-axis :

$$x' = x$$

$$y' = y \cdot cos\alpha - z \cdot sin\alpha$$

$$z' = y \cdot sin\alpha + z \cdot cos\alpha$$

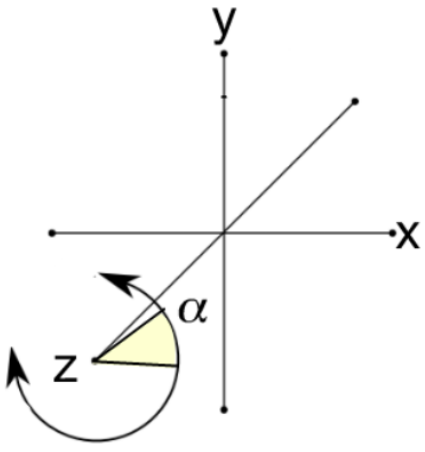


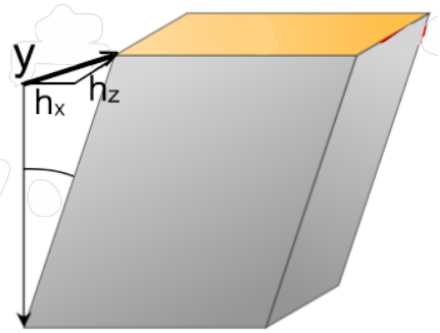

Figure 6: Z-Axis rotation

Again matrices can be used to express rotation :

$$R_z = \begin{bmatrix} cos\alpha & -sin\alpha & 0 & 0 \\ sin\alpha & cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} cos\alpha & 0 & sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sin\alpha & 0 & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha & 0 \\ 0 & sin\alpha & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## 3.2 Shear

Shear bends the objects in one direction. It has an **axis** and **center**. Considering axis y and the origin as center




as the values of y increase the object is bent following the direction of a vector defined by two values ( $h_x, h_z$ in this case ) :

$$x' = x + y \cdot h_x$$
$$y' = y$$
$$z' = z + y \cdot h_z$$

Along the **x-axis**:

$$x' = x$$
$$y' = y + x \cdot h_y$$
$$z' = z + x \cdot h_z$$

Along the **z-axis**:

$$x' = x + z \cdot h_x$$
$$y' = y + z \cdot h_y$$
$$z' = z$$

Again matrices can be used to express shear :

$$H_x(h_y, h_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ h_y & 1 & 0 & 0 \\ h_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad H_y(h_x, h_z) = \begin{bmatrix} 1 & h_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h_z & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad H_z(h_x, h_y) = \begin{bmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

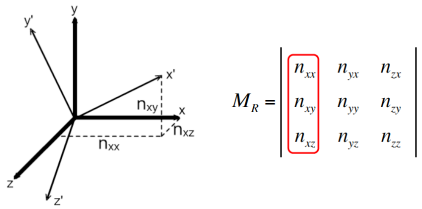
# 4 3D Transform

A general matrix representation can be derived starting from all the transformations found so far :

$$M = \left|\begin{array}{ccc:c} n_{xx} & n_{yx} & n_{zx} & d_x \\ n_{xy} & n_{yy} & n_{zy} & d_y \\ n_{xz} & n_{yz} & n_{zz} & d_z \\ \hdashline 0 & 0 & 0 & 1 \end{array}\right| = \left|\begin{array}{c:c} M_R & \mathbf{d}^T \\ \hdashline \mathbf{0} & 1 \end{array}\right|$$

- **Mr** : sub-matrix representing **rotation,scaling & shear**
- **dt** : translation
- **1** : to ensure that the w coordinate remains **unchanged**

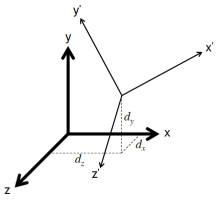
The columns of $M_R$ represent **directions & sizes** of the new axes in the old reference system
**Rotations** maintain the size of the axis constant but change their direction.
**Scalings** maintain the direction of the axis constant , changing the size.




$$M_R = \left|\begin{array}{ccc} n_{xx} & n_{yx} & n_{zx} \\ n_{xy} & n_{yy} & n_{zy} \\ n_{xz} & n_{yz} & n_{zz} \end{array}\right|$$

Vector $d^t$ represents the position of the origin of the new coordinate system in the old one

$$M = \left|\begin{array}{c:c} M_R & \begin{array}{c} d_x \\ d_y \\ d_z \end{array} \\ \hdashline \mathbf{0} & 1 \end{array}\right|$$



## 4.1 Inversion of transformations

To return an object to its **origina** state transformation can be reversed. **Matrix inversion** can be applied when using the matrices representation of transformations.

$$p' = (x', y', z', 1) \rightarrow p = (x, y, z, 1)$$

$$p = M^{-1}p$$

Matrix $M^{-1}$ is **invertible** if its submatrix $M_R$ is invertible. Generally $M^{-1}$ is always invertible except when dealing axis degeneration ( zero factor scaling for example).
Another method of inverting transformations is by using a reverse matrix :

$$\begin{vmatrix} 1 & 0 & 0 & -d_x \\ 0 & 1 & 0 & -d_y \\ 0 & 0 & 1 & -d_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Figure 7: Translation

$$\begin{vmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

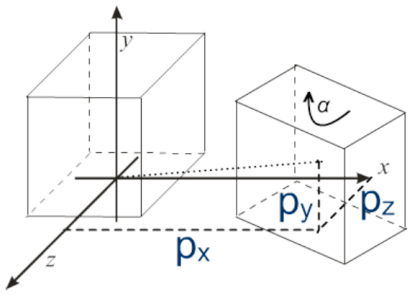
Figure 8: Scaling

$$R_x(-\alpha) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad R_y(-\alpha) = \begin{vmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad R_z(-\alpha) = \begin{vmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Figure 9: Reverse rotation

## 4.2 Composition

During the creation of scene an object is subject to **several** transformations. Applying a **sequence of transformations** is called **composition**. An example is the movement of a cube , sides parallel to x,y,z axis and with center in the origin.
- Translation of center to position $p_x, p_y, p_z$
- Rotation of angle $\alpha$ around y




- **Rotation** around y of $\alpha \rightarrow p' = R_y(\alpha) \cdot p$

$$R_y(\alpha) = \begin{bmatrix} cos\alpha & 0 & sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sin\alpha & 0 & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Translation** in position $p'' = T(p_x, p_y, p_z) \cdot p'$

$$T(p_x, p_y, p_z) = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boxed{p'' = T(p_x, p_y, p_z) \cdot R_y(\alpha) \cdot p}$$

Matrices appear in **reverse** order wrt to the transformations they represent.

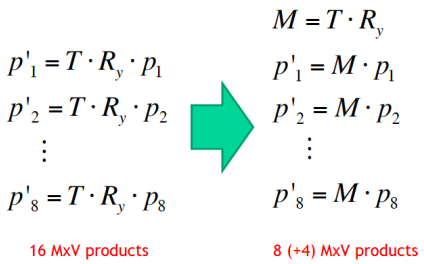
### 4.2.1 Properties of composition of transformations

Matrix-Matrix and Matrix-Vector products are **associative**:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

$$A \cdot (B \cdot p) = (A \cdot B) \cdot p$$

Using the associative property we can obtain a **single matrix** corresponding to the product of **all the transformations**.
This is useful because usually the multiplication is done for many points ($10^4 \sim 10^6$ , so having one matrix that sums up all transformation **improves performances**.




As in the figure instead of having 16 matrix-vector products we only have 12 ( 4 are to create the matrix M ).
**Inversion** can be handled by considering :

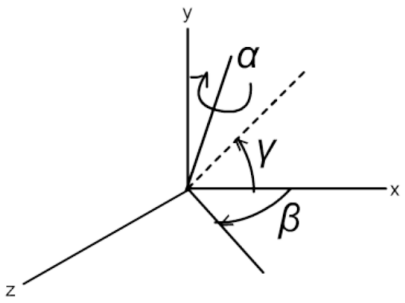
$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

Example : $M = R_y(30°) \cdot T(1,2,3) \rightarrow M^{-1} = T(1,2,3)^{-1} \cdot R_y(30°)^{-1}$ Matrix products are **not commutative** : the order of the transformations is important , and transformations cannot be swapped without obtaining a **different** result.

## 4.3 Transformations around an arbitrary axis or center

**Case : Rotation**

Instead of rotating an object around the x,y or z axis we consider now a rotation of angle $\alpha$ around an arbitrary axis that passes through the origin. Depending on where the considered axis is it forms **two angles** with the the other axis.




In this case the two angles are $\gamma, \beta$ :
- $\gamma \rightarrow$ how much the axis rises on the xz plane
- $\beta \rightarrow$ how much it rises on the xy plane

The angles and planes chosen are arbitrary, other angles and planes can be used to describe the same transformations.

1. $R_y(-\beta)$
   Considering a rotation of $-\beta$ along the y-axis ,the arbitrary axis now lies on the xy plane.
2. $R_z(-\gamma)$
   Then considering a rotation of $-\gamma$ along the z-axis ,the arbitrary axis now corresponds to the x-axis.
3. $R_x(\alpha)$
   As our chosen axis corresponds to the x-axis , the rotation of the object of angle $\alpha$ can be done around the x-axis.
4. $R_z(\gamma)$ and $R_y(\beta)$
   To restore the original axis position.

Final transformation composition :

$$\boxed{p' = R_y(\beta)R_z(\gamma)R_x(\alpha)R_Z(-\gamma)R_y(-\beta) \cdot p}$$

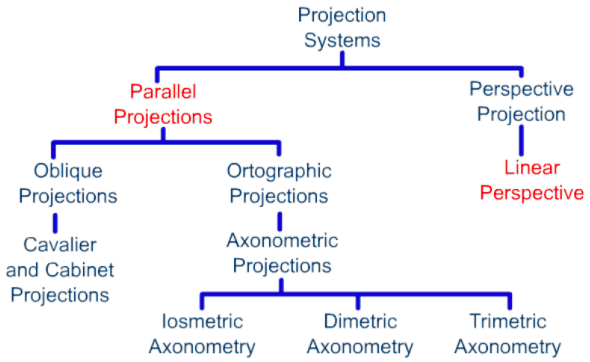
If the axis does not pass through the **origin** , a **translation T** must be applied of a point known through which the axis passes:

$$\boxed{p' = T(p_x, p_y, p_z)R_y(\beta)R_z(\gamma)R_x(\alpha)R_Z(-\gamma)R_y(-\beta)T(-p_x, -p_y, -p_z) \cdot p}$$

Similar procedures can be applied to :

- **Scaling**
- **Shear**

# 5 Projections



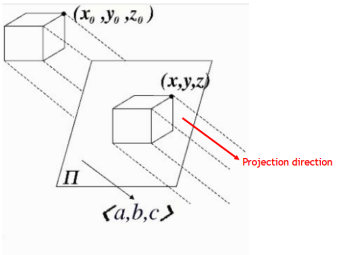

In 3D computer graphics the goal is to represent a three dimensional space on a screen with 2 dimensions:

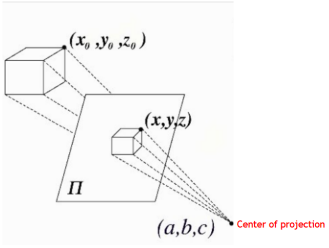
- The 3D graphics uses geometrical primitives defined in 3 dimensions
- 3D graphics produces a 2D representation of the scene to show on screen.

The second step is performed using **projections**. Key features :

- Projections of linear segments **remain** linear segments
- Projected segments connect the projections of the segment's end points

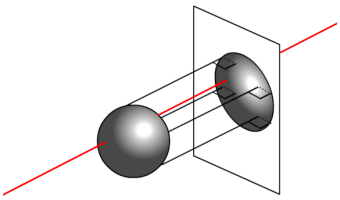
So to create a 2D projection of a 3D polyhedron it is sufficient to **project its vertices** and connect them.
In parallel projections all the rays are **parallel to the same direction**.
In perspective projections all the rays pass **through a point** called

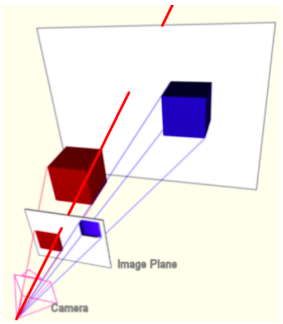



Doing a projection , we loose one coordinate so a point on screen corresponds to an **infinite** number of coordinates ( consequence of moving from 3D $\rightarrow$ 2D) :

in both parallel & perspective projections any point on screen corresponds to **a line of points** in 3D. In parallel projections all points that pass through a line parallel to projections ray are mapped to the **same pixel**.
In perspective projections all points aligned with both projected pixel and the center of projection are mapped to the **same pixel**.

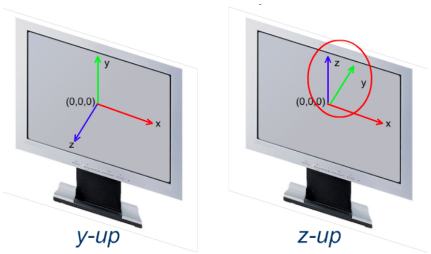





In 3D computer graphics the concept of projection becomes the **conversion** of 3D coordinates from one reference system to another.
**World coordinates→ 3D Normalized Screen Coordinates**

**World coordinates**

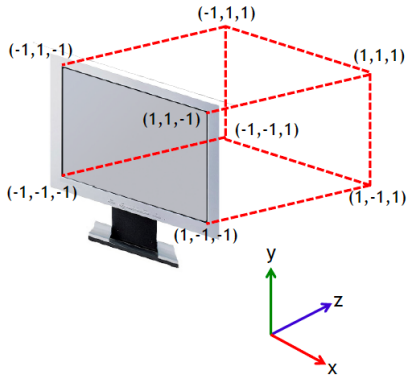
Coordinate system that describes the objects in the 3D space. It is a right-handed Cartesian coordinate system with the **origin** in the **center of the screen.** Some applications invert the z and y axis , with the y axis point inside the screen.




**3D Normalized Screen Coordinates**

Allow to specify the positions of points on screen (or window) in a device-independent way. 3D images must be characterized by a **distance** to allow ordering the surfaces and prevent the construction of unrealistic images.

3D Normalized coordinates have a **third** component ranging form the same extents (ex : -1,1). This way coordinates with a smaller z-value will be considered to be **closer** to the viewer.



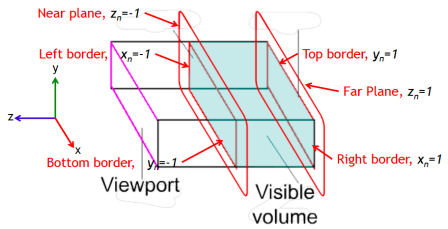

## 5.1 Parallel Projections

**Orthogonal projections** are projections where the plane is either xy,xz or yz and the **projections rays** are **perpendicular to it**.

**Projection plane parallel to xy-plane**

The projections are **perpendicular** to the **z-axis**. Limiting the range of a scene is important to avoid showing objects **behind the observer** or **too far away** :

- The plane with the **minimum z component** $\rightarrow$ **near plane**
- The plane with the **maximum z component** $\rightarrow$ **far plane**




Usually distance from viewport to near plane is very small. Things before the near plane and behind the far plane are **not shown** in the scene: only

the visible volume will be seen.

**Orthogonal projections** can be implemented by **normalizing** the x,y,z coordinates of the projection box in the (-1,1) range. Then a **projection matrix** can be computed to find the normalized 3D coordinates :

$$p_N = P_{ort} \cdot p_W$$

How to find $P_{ort}$?
Coordinates l,r are the **x-coordinates** in the 3D space that will be displayed on the left and right borders of the screen. Everything on the left of l or right of r will be **cut**.
Similarly t,b are the **y-coordinates** of the top and bottom borders of the screen. Finally we call -n ,-f the **z-coordinates** of the near and far planes . Since the z-axis is oriented in the opposite direction the positive distance is used over the negative one. Also using this annotation means that $n > f$ even if n is closer than f! Bottom left front point will have coordinates (-1,-1,-1) while the top right back point has coordinates (1,1,1).
To create the $P_{ort}$ matrix:

1. Move the center of the box to correspond to the center of the space
   The center of the box will have coordinates :

   $$c = (\frac{l+r}{2}, \frac{t+b}{2}, \frac{f+n}{2})$$

   So to align the center we must **inverse translate**

   $$c' = T^{-1}(\frac{l+r}{2}, \frac{t+b}{2}, \frac{f+n}{2}) \cdot c$$

   so that the center now corresponds to the origin.

$$T_{ort} = \begin{vmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{-f+(-n)}{2} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

2. Then normalise the coordinates

$$S_{ort} = \begin{vmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

3. Z goes to the viewer : points closer should have inverse Z coordinate
   Changing the sign of Z is done by mirroring :

$$M_{ort} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Which can be resumed by using a single combined matrix:

$$P_{ort} = M_{ort} \cdot S_{ort} \cdot T_{ort} = \begin{vmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Once the **normalized** screen coordinates are obtained for example for the vertices of a triangle we can apply the the usual drawing primitive procedure to fill up the whole triangle. By repeating this step for **every triangle** that composes the image we can build up a **2D view of a 3D object**.

### 5.1.1 Aspect Ratio

Normalized coordinates are able to support **non-square pixels** it wasn't able to deal with the proportion of the window where objects are drawn . The **aspect ratio** $a = \frac{D_x}{D_y}$ must be considered where $D_x, D_y$ are the **horizontal, vertical** dimensions. Aspect ratios are usually 4:3 or 16:9.
Having a resolution of 2000 x 1000 with **rectangular pixels**. In this case is $a = \frac{2000}{1000} = 2$ ? No because the **aspect ratio** is defined using **metrical units**.
Summing up:

- **Square pixels**
  The **aspect ratio** can be computed by dividing the the number of pixels on the horizontal and vertical directions.

- **Rectangular pixels**
  The **aspect ratio** must be computed by using the actual **physical** dimensions must be used.

Normalized screen coordinates does **not** take care of the aspect ratio : the **projection matrix** must take care of this adding a **scaling factor**.
Considering the viewport the **width** is r-l and the height is t-b so the ratio of the window is $\frac{r-l}{t-b}$ if :

$$\frac{r-l}{t-b} = a = \frac{D_x}{D_y}$$

then the image will not be **distorted**.

### 5.1.2 Projection matrices and aspect ratio

Usually the **projection box** is centred vertically and horizontally in the world. Using the half-width **w** from the center to the left/right border we can use less information to computed the matrix. The vertical equivalent of the w is computed using the **aspect ratio a** : $\frac{a}{w}$. This way we can compute the position of left,right, bottom and top only knowing the half-width and the aspect ratio.

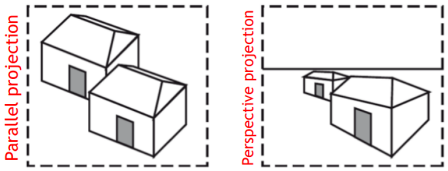
$$P_{ort} = \begin{vmatrix} \frac{1}{w} & 0 & 0 & 0 \\ 0 & \frac{a}{w} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Note that the element in position (1,3)=(1,4)= 0 because the projection box is **already** centred in the origin.

## 5.2 Perspective Projections

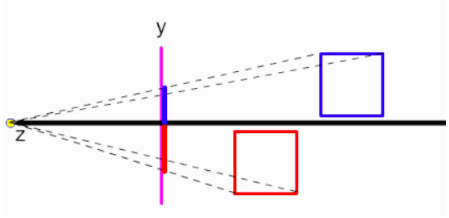
Parallel projections do **not** change the apparent size of an object with the distance from the observer. It is used mainly for drawings and is not that suitable for 3D computer graphics.

**Prospective projection** on the other hand represent an object with a different size depending on its **distance** from the **projection plane** : this makes it suitable for **immersive visualisations**. This is the result of all the projections rays **passing through the same point**.

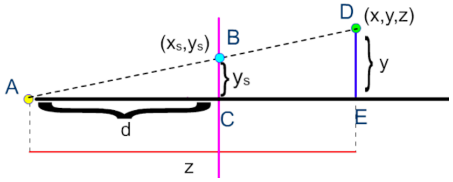



Rays intersect the **projection plane** at different points depending on the

distance of the object : if two objects have the same size but are at different distances than the ones **closer** to the plane have a **larger** projection.




As for the parallel projections also perspective projections make us of **normalized screen coordinates** to project objects on the projection plane. Given the space coordinates $(x, y, z) \rightarrow (x_s, y_s, z_s)$. Now we will focus only on $x_s, y_s$.




- $y_s$

  To simplify the computation the **center of projection** (yellow dot) corresponds to the origin $(0, 0, 0)$. The projection plane is located at distance d on the z-axis from the center of projection.

  Tracing the projection ray from point $(x, y, z)$ to the center of projection we obtain two **similar** triangles $ABC, ADE$. It is easy to see that $y_s$ is the height of ABC while $y$ the height of ADE $\rightarrow y_s : d = y : z$ which leads to

$$\boxed{y_s = \frac{d \cdot y}{z}}$$

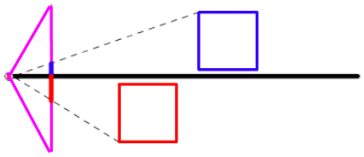
- $x_s$

  The same computation as for $y_s$ occur :

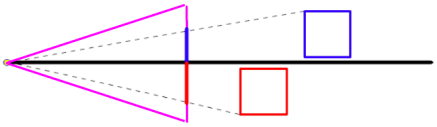
$$\boxed{x_s = \frac{d \cdot x}{z}}$$

The distance **d** from center of projection to projection plane plays an important role .It acts like the **camera lens** so changing parameter d has the effect of performing a **zoom**:

- **Short d**
  Like a wide lens, emphasizes the distances of objects from the plane. Allows to capture a larger number of objects producing smaller images



- **Long d**
  Like tele-lens, reducing the differences in size for objects at different distances. It reduces the number of objects visible in the scene producing **enlarged** objects.



- $D \to \infty$
  If distance d tends to infinity we obtain **parallel projections**

As for the parallel projections , also **perspective projections** can be obtained with a **matrix-vector product**.As the world coordinate system is oriented in the **opposite** direction of the z-axis , the z-coordinates are **negative** :

$$\boxed{x_s = \frac{d \cdot x}{-z}}$$

$$\boxed{y_s = \frac{d \cdot y}{-z}}$$

The projection matrix for perspective with **center in the origin** and projection plane at distance **d** on the z-axis:

$$P_{persp} = \begin{vmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{vmatrix}$$

The last row is no longer |0001| as per usual : the result is a vector which no longer has component **w =1**:

$$\boxed{[d \cdot x, d \cdot y, d \cdot z, -z]}$$

To obtain the equivalent Cartesian coordinates we must divide by the w component ( -z ) :

$$[x_s, y_s, -d, 1]$$

The z-coordinate is always **-d** regardless of the what the z-coordinate is, which means that all information about **distance** is lost → no proper 3D normalized screen coordinates can be defined.
The solution to not flat the z component is to **add** an element = 1 in the third row of the fourth column:

$$P_{persp} = \begin{vmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & \boxed{1} \\ 0 & 0 & -1 & 0 \end{vmatrix}$$

which leads to normalized screen coordinates :

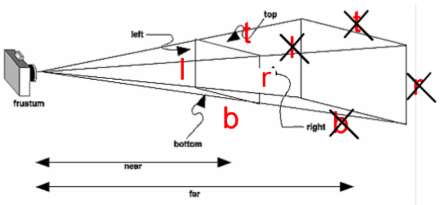
$$|x_s, y_s, -d - \frac{1}{z}, 1|$$

Now we have a third component depending on d but also on **z**. Depending on the distance:

- Negative but smaller when closer to the viewer.
- Negative but larger when farther away from the viewer.
- Tends to $-d$ as the distance goes to $\infty$.

### 5.2.1 Perspective matrix on screen

Now that we have basic tools for creating (after normalization) proper normalized screen coordinates that respect the distance of objects we need to combine these new tools with transformations to be able to show correctly on screen the desired coordinates.
In the case of perspective projections instead of a view box like in parallel projections we have a **frustum**:



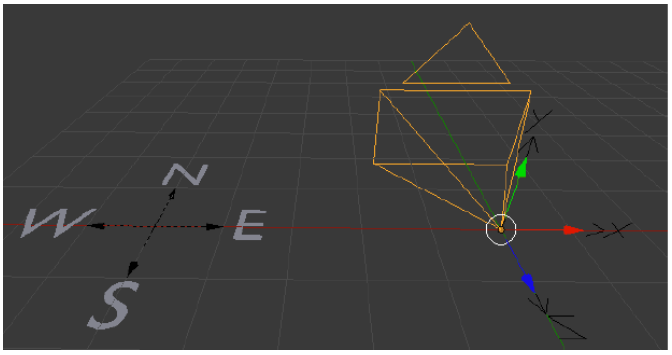

The frustum is defined by it's **near** and **far** plane and **top**,**bottom** ,**left** and **right** coordinates. The coordinates are **not constant** any more: now they depend on the **distance**.
By default t,b,l,r are defined on the **near plane** so the distance **d** corresponds to the value **n** of the near plane. The resulting projection matrix is:

$$U_{persp} = \begin{vmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 1 \\ 0 & 0 & -1 & 0 \end{vmatrix}$$

# 6 View and World Transformations

Last chapter was about what is required to find the screen coordinates of on object in space . This chapter focuses on how to position and view objects from **different angles** in 3D ( **motion** of the object and camera) .
Assumption :

- **negative** z-axis → **North**
- **positive** x-axis → **East**




3D world coordinates and their transformations were specified in a **map** with a center in the origin and the x-axis ranging from west to east, the z coordinate ranging from north to south and the y-axis orthogonal to the plane.
The goal is to find a mapping between a geographical map and the world coordinates in the 3D space.

## 6.1 View Matrix

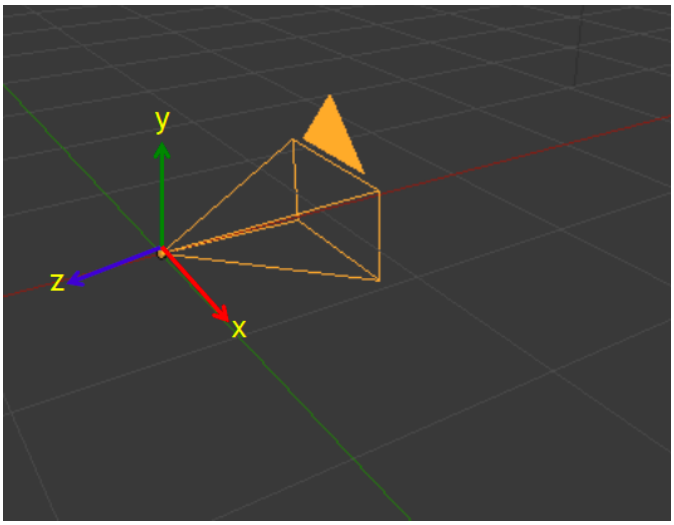
The view matrix assumes that the **projection plane** (= the screen) is the xy axis.

- Parallel projection : projection ray is parallel to **z-axis**
- Perspective projection : center of projection is the **origin**

Changing the way in which the world is seen (position of object or direction in which we are looking) can be achieved by adding some **transformations** that are

perform **before** the projections.
We can think of the projections matrix as a **virtual camera** that looks at the screen from the center of projection. It has:

- **position** ( initially in origin)
- **direction** it is aiming towards (initially along negative z-axis)




The camera can be moved by applying a transformation matrix $M_c$ (**camera matrix)** that moves the camera object to its position and direction. Now that the camera is in its new position **all** objects are moved by applying the **inverse** matrix $M_c^{-1}$ so that the new projection plane is parallel to the xy-plane and the center of projection is in the origin.
The matrix $M_c^{-1}$ is called **View Matrix** $M_v = M_c^{-1}$.
To create the view matrix in a user-friendly way (two most popular):

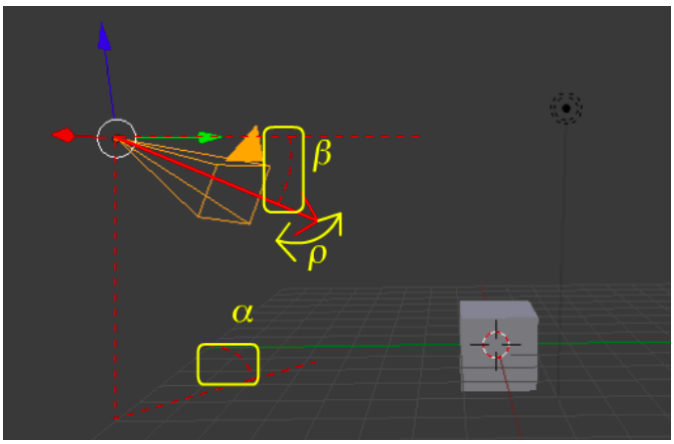
- **look-in direction matrix**
- **look-at matrix**

### 6.1.1 Look-in-direction matrix

It's the one used in first person games in which you want to see the world in a fixed point and change the direction where you're looking at and the position from

which you look the world at.
In this kind of model we have (these are just conventions):

- $(c_x, c_y, c_z)$ position of the center of the camera in world coordinates
- $\alpha$ horizontal angle = where you look at.
  - $\alpha = 0° \rightarrow$ look north
  - $\alpha = 90° \rightarrow$ look west
  - $\alpha = -90° \rightarrow$ look east
  - $\alpha = +/- 180° \rightarrow$ look south
- $\beta$ vertical angle = look up and down
  - $\beta > 0° \rightarrow$ look up
  - $\beta < 0° \rightarrow$ look down
- $\rho$ roll over the viewing angle = tilting (not often used)
  - $\rho > 0° \rightarrow$ turn counter clockwise
  - $\rho < 0° \rightarrow$ turn clockwise




The **View Matrix** is the inverse of this camera matrix so :

$$M_c = T(c_x, c_y, c_z) \cdot R_y(\alpha) \cdot R_x(\beta) \cdot R_z(\rho)$$

$$M_v = M_c^{-1} = R_z(-\rho) \cdot R_x(-\beta) \cdot R_y(-\alpha) \cdot T(-c_x, -c_y, -c_z)$$

So to obtain the normalized coordinates of a point A in space seen from a certain point of view :

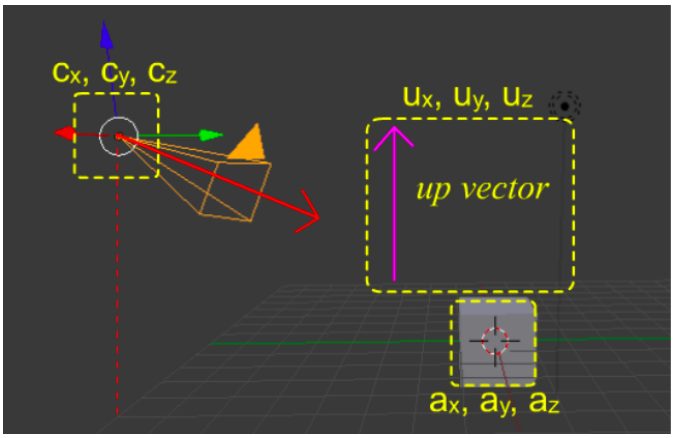
$$A' = P_{proj} \cdot M_v \cdot A$$

### 6.1.2 Look-at matrix

This model has :

- $(c_x, c_y, c_z)$ position of center of the camera
- $(a_x, a_y, a_z)$ position of the object aimed at
- $(u_x, u_y, u_z)$ up vector (usually u= (0,1,0) in the y direction)

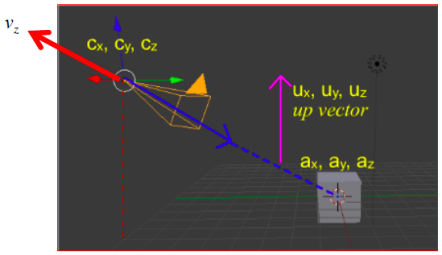
These two points are not enough as they do not decide the **roll** of the camera. To specify the orientation of the camera the **up vector** is added. It is the inverse of gravity and shows which way is up in the world. The camera bottom axis should always be **perpendicular** to the up vector : this way when you roll the camera is always aligned with the ground.



The view matrix again is determined starting from the camera matrix.

$$v_z = \frac{c - a}{|c - a|}$$

$$v_z = \frac{(c_x - a_x, c_y - a_y, c_z - a_z)}{\sqrt{(c_x - a_x)^2 + (c_y - a_y)^2 + (c_z - a_z)^2}}$$




The x-axis is perpendicular to both the new z-axis and the up vector

$$v_x = \frac{u \times v_z}{|u \times v_z|}$$

$$u \times v_z = |u_x, u_y, u_z| \times |v_x, v_y, v_z| = |u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x|$$

The y-axis is perpendicular to both the new z-axis and the x-axis :

$$v_y = v_z \times v_x$$

Which leads to

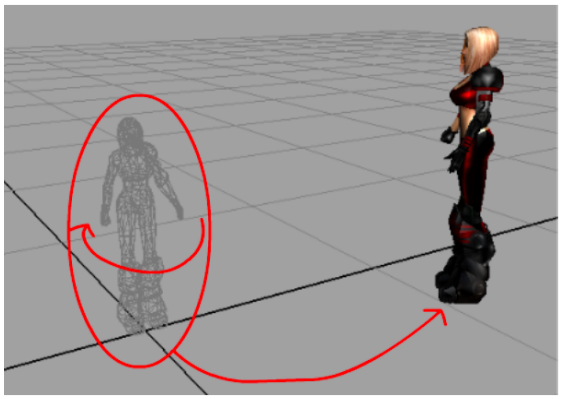
$$M_c = \begin{array}{ccc|c} v_x & v_y & v_z & c \\ 0 & 0 & 0 & 1 \end{array} = \begin{array}{c|c} R_c & c \\ 0 & 1 \end{array}$$

Where R is the rotation matrix So the **View Matrix** is:

$$M_v = M_c^{-1} = \begin{array}{c|c} R_c^T & -(R_c)^T \cdot c \\ 0 & 1 \end{array}$$

## 6.2 Local coordinates and World Matrix

One of the main features of 3D graphics is showing **moving objects**. Moving is achieved with a **World Matrix**.
Every object is characterized by a set of **local coordinates**: the positions of the object's points in the space where it was created. When a scene is composed the position of the objects is moved from where it was modelled to where is must be shown . This transformation assigns to the objects new coordinates : the **global/world coordinates**.



The world matrix $M_w$ transforms (translations,rotation,scaling,shear) the local coordinates into the corresponding world coordinates.
There are many definitions of transformation order applied to objects with one dominating :

1. **Scale/Mirror** the object
2. **Rotate** the object
3. **Position** the object

### 6.2.1 World Matrix : scaling

Must be performed first: if the object is scaled or mirrored of $s_x, s_y, s_z$ any rotation must be applied after otherwise it will cause scaling along an arbitrary axis.
Scaling makes the objects larger/smaller. If the scaling is proportional is happens

equally along all axis. No proportional scaling happens on a specific axis. Applying scaling $s_x, s_y, s_z$ to unitary local coordinates translates into $s_x, s_y, s_z$ global units in the global coordinate system.

### 6.2.2 World Matrix : rotating

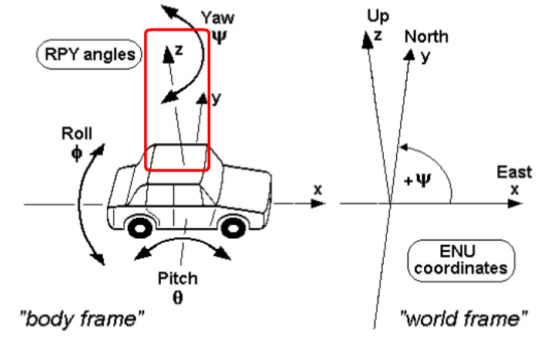
Must be performed between scaling and positioning.
To define a specific orientation in 3D space a combination of rotations along the 3-axis must be performed. In which order must these rotation be performed?
Several ways exist to compute the rotation of the object. A consistent way to specify the parameters required by the user to define orientation in 3D space of an object consists in the **Euler Angles** :

- **Roll (x-axis)**
  The roll $\phi$ identifies the rotation of the object along the facing axis.A **positive** roll turn an object**clockwise** in the direction it is facing.

- **Pitch (y-axis)**
  The pitch $\theta$ defines the elevation of the object and corresponds to a rotation around its side axis. A **positive** pitch turns the head of the object **facing down**.

- **Yaw (z-axis)**
  The yaw $\psi$ defines the direction of the object and corresponds to a rotation along the vertical axis. $\psi = 0° \rightarrow$ East

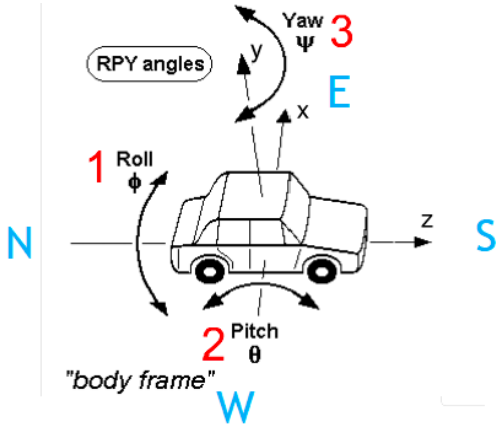
Euler angles are defined for a z-up coordinates system.



Object are modelled so that they face the positive x-axis, the side aligned with the y-axis and vertically along the z-axis.
With the above conventions transformations are performed in the alphabetical order:

1. x-axis → Roll
2. y-axis → Pitch
3. z-axis → Yaw

If the axis conventions are different the order must always be Roll-Pitch-Yaw




For example in the 'Look-in-direction' camera model a y-up Euler angle orientation system was used. The camera is however oriented in the negative z-axis. For this reason the Roll $\phi$ and Pitch $\theta$ work in the opposite way as $\rho$ and $\beta$ and direction $\alpha = 0$ corresponds to the camera looking North instead of South.Rotations are still performed in the same order.

### 6.2.3 World Matrix : positioning

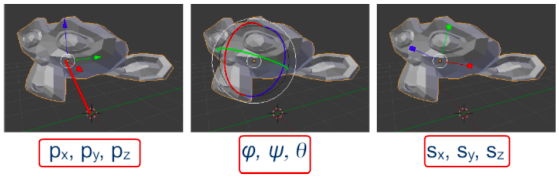
Must be performed last otherwise the coordinates would be changed during the other transformation.
When positioning the object the user wants to specify the coordinates where it should be placed in the 3D place.These coordinates should be independent of the size and orientation of the object.
Positioning at $p = (p_x, p_y, p_z)$ is performed by applying a **transformation** $T(p_x, p_y, p_z)$.

So $(p_x, p_y, p_z)$ are the coordinates of the origin of the object after the transformations ( initially is the origin in local coordinates was $(0, 0, 0)$ )

### 6.2.4 Final World Matrix

With this y-up convention (object facing the positive z-direction , x is the side direction )an object in space can be positioned in a 3D space using 9 parameters:

- position $p_x, p_y, p_z$
- rotation $\phi, \psi, \theta,$
- scaling $s_x, s_y, s_z$

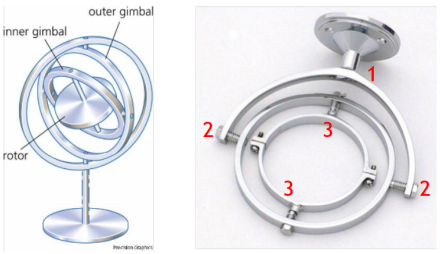



The final world matrix is :

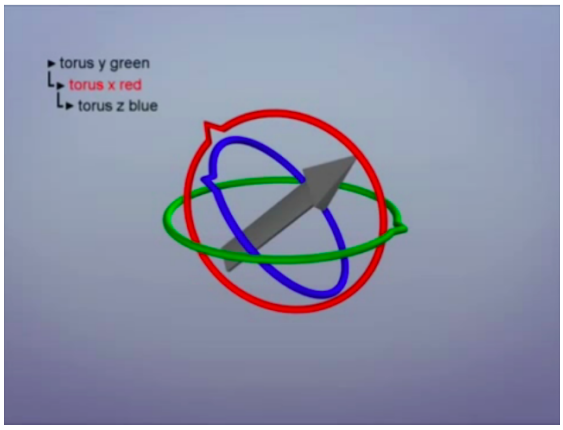
$$\boxed{M_w = T(p_x, p_y, p_z) \cdot R_y(\psi) \cdot R_x(\theta) \cdot R_z(\phi) \cdot S(s_x, s_y, s_z)}$$

## 6.3 Gimbal Lock

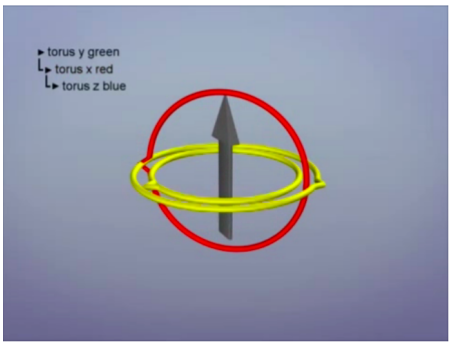
A rotation defined by the Euler Angles is perfect for **planar** movements ( good for driving games or FPS).Euler Angles are a problem for applications such as flight simulators as they suffer from a problem known as **gimbal lock**. A **gimbal** is a ring that can spin around its diameter.



A physical system that allows freely orienting an object in the space has **at least three** gimbals connected to each other (one for each rotation roll-pitch-yaw). The problem is that the rotations are connected : each ring corresponds to a rotation along x,y,z -axis. Consider an order of y-x-z as in figure below




Moving the yaw (green outermost ring) a movement of the other two rings can be observed. Moving the the pitch (red middle ring) the roll (blue inner ring) moves too.
If the pitch (x-axis) rotates 90 degrees , the z and y axis are **aligned**




So we **loose** a degree of freedom. When a **gimbal lock** occurs some movements are no longer performable: such movements must be performed by doing complex **combinations** of these movements. In our case a common solution used to express the rotation of an object is to use a mathematical device called **quaternion** instead of Euler Angles.

# 7 A complete projection example

To obtain the position of the pixels on screen from the local coordinates that define the 3D model five steps should be performed:

1. World Transform
2. View Transform
3. Projection
4. Normalization
5. Screen Transform

Each steps performs a coordinate transformation from on 3D space to another. The first three can be done with a **matrix-vector** product. The screen transform can be done possibly also this way. Normalization instead requires a different procedure.

## 7.1 World-View-Projection Matrices

**1. Model**

Firstly a 3D model is created in local coordinates $p_M$. Local coordinates are usually 3D Cartesian coordinates and are firs transformed into **homogeneous coordinates** $p_L$ by adding a **forth** component equal to 1.

$$p_M = |p_{Mx} \quad p_{My} \quad p_{Mz}|$$
$$p_L == |p_{Mx} \quad p_{My} \quad p_{Mz} \quad 1|$$

**2. World Matrix**

The **World Transform** converts the coordinates from local space to global space by multiplying them by the **World Matrix** :

$$p_W = M_w \cdot p_L$$

**3.View Matrix**

The view transform allow to see the 3D world from a given point in space.It

transforms the global space coordinates into **camera space coordinates** by using the **View Matrix**

$$p_V = M_V \cdot p_W$$

.

**4.Projection Matrix**

The projection transformation prepares the coordinates to be shown on screen by performing either a **parallel** or **perspective** projection.
For parallel projections the transformations is performed using a parallel projection matrix $M_{P-ort}$ and it converts the camera space coordinates into **Normalized Screen Coordinates**
For perspective projections the transformations is done using a perspective projection matrix $M_{P-pers}$ and it converts the camera space coordinates into **Clipping Coordinates** (! not **normalized**!)

$$p_C = M_p \cdot p_V$$

These matrices can be compressed in a single matrix (**World View Projection Matrix**) :

$$p_C = M_p \cdot M_V \cdot M_W \cdot p_L = M_{WVP} \cdot p_L$$

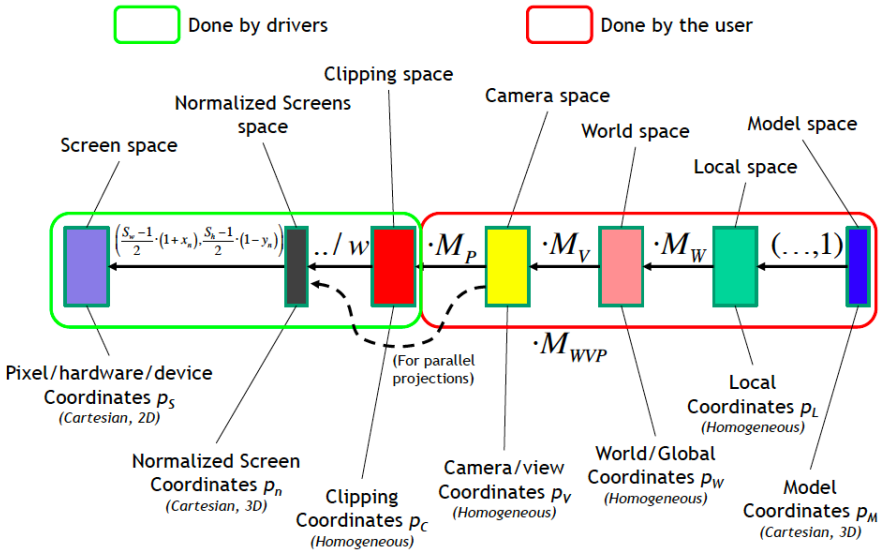
The **Normalization** step is require in case of perspective projections where the transformations produces **clipping coordinates**. As opposed to other transformations this step is done by normalizing the homogeneous coordinates that describe the points in the clipping space. Every component is divided by the fourth component and the last component is then discarded :

$$|x_C \quad y_C \quad z_C \quad w_C| \rightarrow \left| \frac{x_C}{w_C} \quad \frac{y_C}{w_C} \quad \frac{z_C}{w_C} \quad 1 \right| \rightarrow (x_N, y_N, z_N)$$

This must be done only for perspective projections as the parallel ones already result in normalized coordinates where it is sufficient to just drop the last component.
This normalization step is performed by the video cart adapter and is transparent to the user : it first transforms the clipping coordinates in normalized screen

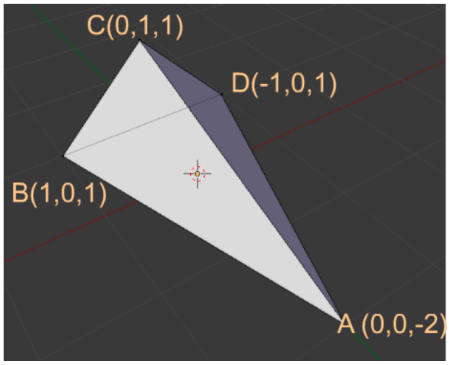
coordinates and the into pixel coordinates to show objects

$$(x_S, y_S) = \left( \frac{S_W - 1}{2} \cdot (1 + x_n), \frac{S_h - 1}{2} \cdot (1 - y_n) \right)$$



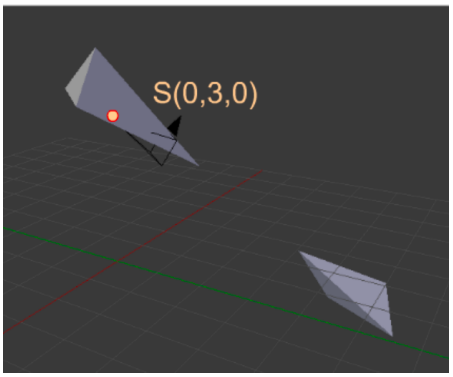

## 7.2 The example

The starship is models with a tetrahedron and has the following local coordinates.




It is facing the **negative z-axis**.

In a moment of the game the player is in position $(0, 3, 0)$ with Pitch $-30°$, Roll $0°$, Yaw $-45°$
The enemy ship is in position $E(3, -1, -5)$ with Pitch $45°$, Roll $0°$, Yaw $120°$

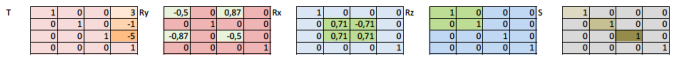



What are the pixel coordinates of the vertex of the tetrahedron seen on a 960x540 pixel screen (5:4 aspect ration with non square pixels)?
Additional informations :

- Field of View : $90°$ so quite wide-angle.
- Near plane $= 0.5$ (you see windscreen of starship), far plane $= 9.5$ (see nothing beyond)
- Scaling $S = (1, 1, 1)$

**World Matrix of enemy ship**

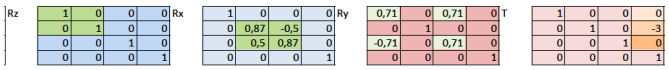
- Position $(p_x, p_y, p_z) = (3, -1, -5)$
- Rotation Yaw,Pitch,Roll$= (120°, 45°, 0°)$
- Scaling $S = (1, 1, 1)$



| Mw | | | | |
|---|---|---|---|---|
| | -0,5 | 0,61 | 0,61 | 3 |
| | 0 | 0,71 | -0,71 | -1 |
| | -0,87 | -0,35 | -0,35 | -5 |
| | 0 | 0 | 0 | 1 |

### View Matrix

- Center position $(c_x, c_y, c_z) = (0, 3, 0)$
- Angles $(\alpha, \beta, \rho) = (-45°, -30°, 0°)$




Mv

| | | | |
|---|---|---|---|
| 0,71 | 0 | 0,71 | 0 |
| 0,35 | 0,87 | -0,35 | -2,6 |
| -0,61 | 0,5 | 0,61 | -1,5 |
| 0 | 0 | 0 | 1 |

### Projection matrix

- (FoV,aspect ratio) $= (90, 1.25)$
- (n,f) $= (0.5, 9.5)$

Pp

| | | | |
|---|---|---|---|
| 0,8 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | -1,11 | -1,06 |
| 0 | 0 | -1 | 0 |

### WVP Matrix

| | | | |
|---|---|---|---|
| -0,7727 | 0,15 | 0,15 | -1,13 |
| 0,1294 | 0,95 | -0,27 | -0,64 |
| 0,249 | 0,26 | 1,05 | 6,61 |
| 0,2241 | 0,24 | 0,95 | 6,9 |

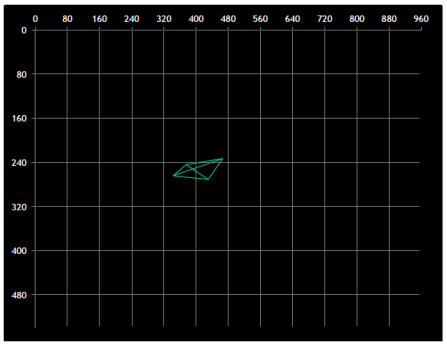
Then we multiply the points of the vertexes (to which a fourth component equal to 1 is added) with the WVP Matrix.

| A | B | C | D |
|---|---|---|---|
| 0 | 1 | 0 | -1 |
| 0 | 0 | 1 | 0 |
| -2 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

We obtain the **clipping coordinates**:

| | | | |
|---|---|---|---|
| -1,4242 | -1,7577 | -0,84 | -0,21 |
| -0,0939 | -0,7771 | 0,05 | -1,04 |
| 4,5098 | 7,9091 | 7,92 | 7,41 |
| 5,0089 | 8,0682 | 8,08 | 7,62 |

These coordinates are then normalized (in 3D) and then transformed into 2D pixel coordinates $(343, 264), (375, 244), (430, 271), (466, 233)$



# 8 Meshes and Clipping

We want to represent objects of different nature ( curved, glossy ,bumpy...).A suitable encoding must be found to represent objects in a virtual environment.Using just a set of points is not enough to represent a solid object. Even using many vertices ( 10000) can make the object look empty and more points would be computationally expensive.
Every solid object is stored by its boundary: what is **inside** vs what is **outside**.
Object geometries are encoded following mathematical models that represent **surfaces** through a set of parameters. Many models have been defined in literature :

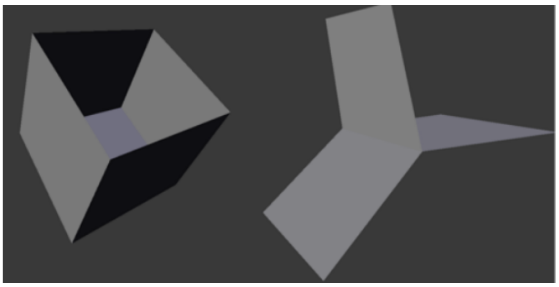
- **Meshes** (polygonal surfaces)
- **Hermite surfaces**
- **NURBS** (non uniform ration b-splines)
- **HSS** ( hierarchical subdivision surfaces) **Metaballs**

All models are converted into **meshes** : its the only type of encoding that a low level rendering engine is capable of doing.

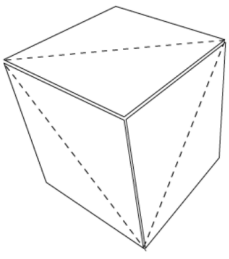
## 8.1 Meshes

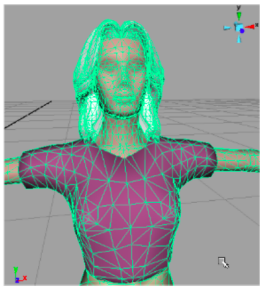
A **mesh** is a polygonal surface that can be described by a set of contiguous polygons (cube, pyramid , prism are meshes ; cones,cylinders,spheres are not meshes but can be rendered as one using tricks shown later).
A polygon that describes a planar surface portion of a surface is called a **surface**.
Sides of the polygons are called **edges** and correspond commonly to **intersection** of surfaces.

If every edge is **adjacent** to exactly **two** faces then the surfaces has a special topology called **2-manifold**. Non 2-manifold surfaces usually represent non-physical objects and require special care (see below examples : solid with holes and lamina-faces)



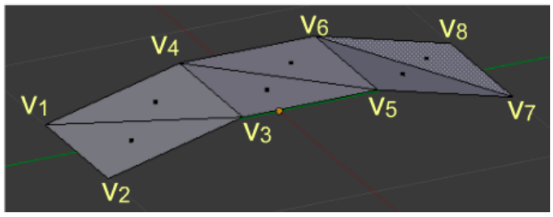
Sometimes non-2- manifold objects can be used to model very thin objects (pages...) or to obtain a special effect (opne box, magazine...). In these kind of situations the later presented **back-face culling** algorithm will **NOT** work.
Each polygon can be reduced to a set of **triangles** that shares some edges. A set of **adjacent triangles** is called a **mesh**.
Triangles are used because three points define a **planar surface** : otherwise if we had more points we could end up with different planes ( and thus different interpretations).
A polygonal surface (planar or not) is first **converted** into triangles known as **tessellation**. Polygon tessellation is not unique : several might be defined and not all are equivalent (some are better some are not).
A mesh representation of an object stores its surfaces with the set of polygons that delimit its boundary. Then the boundary polygons are in turn encoded as a set of contiguous triangles that share some edges.



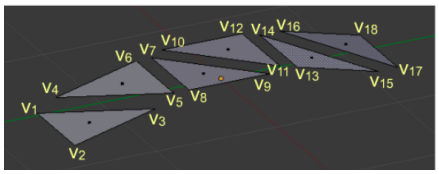


### 8.1.1 Mesh encoding

Meshes are usually encoded as a **set of vertices**. The rendering engine uses such vertices to determine the end points of the triangles that compose the mesh




In the figure we have 3 surfaces divided into 6 triangles with and 8 vertices. Each triangle has 3 vertices but instead of having 6x3 = 18 vertices we use 8 because 10 are shared

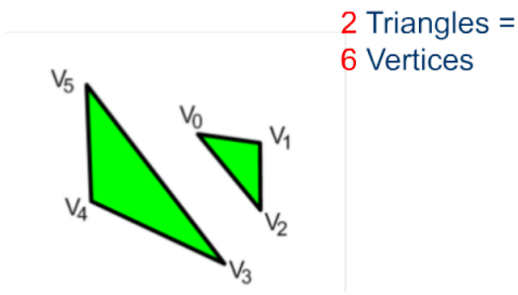



Three main types of mesh-encoding are:

- **Triangle Lists**
- **Triangle Strips**
- **Triangles Fans**

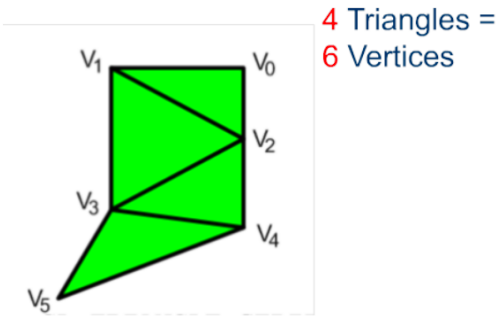
**Triangle lists**

Triangle lists do **not** exploit any **sharing vertices** and encode each triangle **separately**. They are used to encode **unconnected** triangles



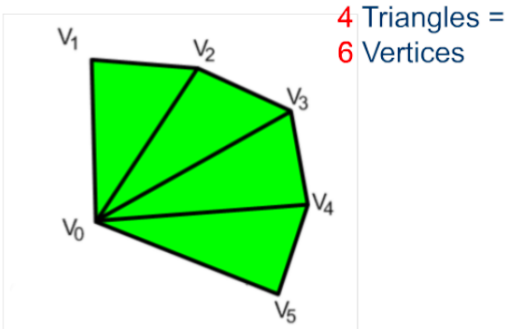
**Triangle strips**

Triangle strips encode a **set of adjacent** triangles that define a band-like surface.The encoding begins by considering the first two vertices. Then each new vertex is connected to the previous two.




**Triangle fans**

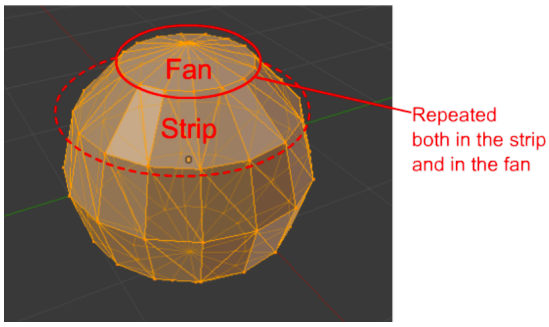
Triangle fans encode polygons where all the triangles share a **vertex**.The first two vertexes are specified independently.Then each new vertex connects both the previous one and the first of the list.




### 8.1.2 Indexed Primitives

Triangle lists and strips can **save** some memory most of the times. Sometimes though they are not an advantage even if the topology would seem to be appropriate. This is due to the fact that often vertexes are **repeated** .Another reason is that vertexes are encoded by more parameters than just their coordinates (i.e. normal vector and texture mapping) : you can save memory only if shared vertexes are **identical** with respect to all parameters , otherwise different encodings

for that vertex are required.
Many primitives cannot be encoded with a single triangle strip/fan so many vertexes can still be shared between different strips/fans. **Indexed primitives** allow reducing the cost of replicating the same vertex between different lists/strips/fans.




In this case the cap of the sphere is encoded using a fan, and the rings using strips. The points connecting fan and strip are shared , so they are encoded two times. This is also true for the first ring and second ring , the second ring and the third ring and finally the third ring and the lower cap.A solution to this are **Indexed primitives**.They are defined by two arrays:

- **Vertex Array**:
  Contains the definitions/positions of the different vertexes.For example a cube will have a list of 8 (x,y,z) elements.

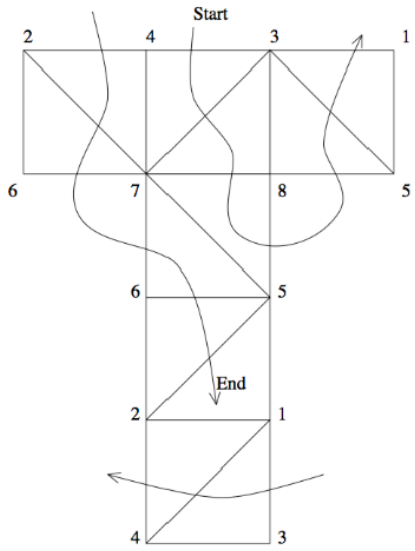
- **Index Array**:
  Are used to **indirectly** specify the triangles. For example a cube has 6 faces so we have 12 triangles. So in the index array we have 12 elements (a,b,c) where a,b,c are the indexes of the vertex array composing that triangle.

**Example**

Consider a cube encoded in strip lists. We have 6 faces, 12 triangles each has 3 vertexes using three coordinates each is a float (4B) so :

$$6 * 2 * 3 * 3 * 4 = 432 Bytes$$

If the cube is encoded in triangle strips only 14 vertexes are required




$$14 * 3 * 4 = 168$$

Using indexed primitives we need 8 vertexes (the ones of the cube) and 36 indices ($\approx$ 1B):

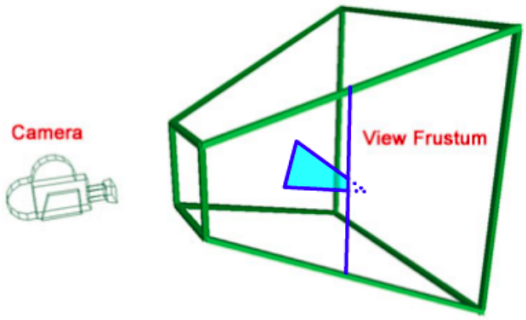
$$8_{vert} * 3_{xyz} * 4_{float} + 6_{faces} * 2_{tri} * 3_{vert} * 1_{byte} = 132Bytes$$

By using indexed primitives with triangle strips we can reduce the size even more:

$$96 + 14 = 110Bytes$$

## 8.2 Clipping

The triangles of a mesh can intersect the boundary of the screen and can be partially shown. The clipping process is performed **after** the projection transform but **before** the normalization step (in other word it is performed on the **clipping** coordinates)
In 3D space clipping is performed against the viewing frustum :




The equation of a plane is

$$n_x x + n_y y + n_z z + d = 0$$

where $n_x, n_y, n_z$ represent the **normal** to the plane. The constant term d defines the distance from the origin (0 if the plane passes through the center of axis). The equation divides the 3D space into two regions called **half space**:

$$n_x x + n_y y + n_z z + d > 0$$

$$n_x x + n_y y + n_z z + d < 0$$

The frustum is convex solid an can be determined by 6 half spaces. For each of these 6 half-spaces the above equations are used to determine if a points belong the correct half space.This can be simplified by a scalar product of two vector:

- $n = (n_x, n_y, n_z, d)$ identifies the plane
- $p = (x, y, z, 1)$ for the point

$$n_x x + n_y y + n_z z + d =\rightarrow n \cdot p = 0$$

The six normal vectors can be computed together with the projection matrix.However if clipping is performed into clipping space, since coordinates are meant to be inside the frustum if between -1 and 1, the six vector become very simple:

$$\frac{x}{w} > -1, \quad x > -w, \quad x + w > 0, \quad n_l = |1 \quad 0 \quad 0 \quad 1|$$

$$\frac{x}{w} < 1, \quad x < w, \quad -x + w > 0, \quad n_r = |-1 \quad 0 \quad 0 \quad 1|$$

$$\frac{y}{w} > -1, \quad y > -w, \quad y + w > 0, \quad n_b = |0 \quad 1 \quad 0 \quad 1|$$

$$\frac{y}{w} < 1, \quad y < w, \quad -y + w > 0, \quad n_t = |0 \quad -1 \quad 0 \quad 1|$$

$$\frac{z}{w} > -1, \quad z > -w, \quad z + w > 0, \quad n_n = |0 \quad 0 \quad 1 \quad 1|$$

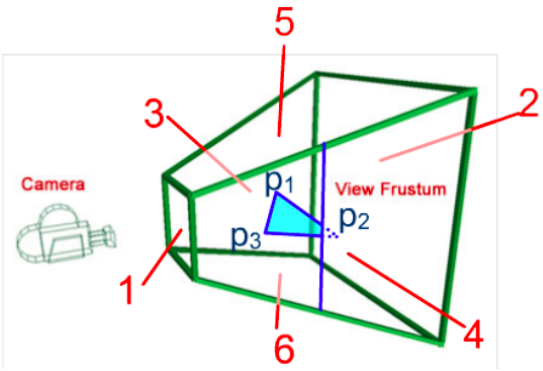
$$\frac{z}{w} < 1, \quad z < w, \quad -z + w > 0, \quad n_f = |0 \quad 0 \quad -1 \quad 1|$$

So points will be inside the frustum if :

$$\boxed{p \cdot n_v > 0 \quad \forall v \in \{l, r, t, b, n, f\}}$$

### 8.2.1 Clipping triangles

Clipping points is easily done given the normal vectors and the points in space with the above formula. In triangles the same principle is applied for all its vertex. But how is a triangles rendered inside the frustum when clipping is performed on one or more vertexes?



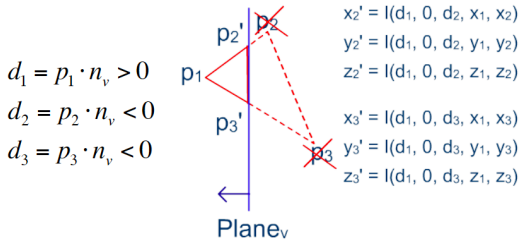

As shown the triangle and the frustum face are intersected . Computationally this is done by the **Sutherland-Hodgman Algorithm**.
Focusing one on the figure above we have a face defined by its vector $n_v$ and the three vertexes of the triangle $p1, p2, p3$ :

$$d_1 = p_1 n_v$$

$$d_2 = p_2 n_v$$

$$d_3 = p_3 n_v$$

Trivial cases all points are inside/outside ($d_i > 0$ / $d_i < 0$). If all points are outside the algorithms stops.If the they are all positive the algorithms moves to the next face.
On the other hand if two points are on the outside ($p_2, p_3$) and one the inside ($p_1$) the two **intersections** $p_2', p_3'$ are computed using **interpolation**.



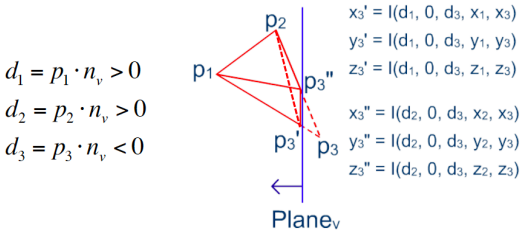

The distance from plane $d_2, d_3$ are used as interpolation factors:

$$p_2' = (x_2', y_2', z_2')$$

$$x_2' = I(d_1, 0, d_2, x_1, x_2)$$
$$y_2' = I(d_1, 0, d_2, y_1, y_2)$$
$$z_2' = I(d_1, 0, d_2, z_1, z_2)$$

$$p_3' = (x_3', y_3', z_3')$$
$$x_3' = I(d_1, 0, d_3, x_1, x_3)$$
$$y_3' = I(d_1, 0, d_3, y_1, y_3)$$
$$z_3' = I(d_1, 0, d_3, z_1, z_3)$$

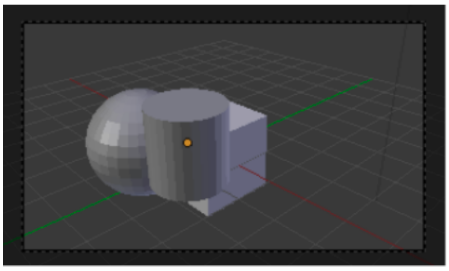
In clipping space also the fourth coordinate w must be interpolated! f two points are on the inside ($p_2, p_3$) and one the outside ($p_1$) we have a more complex situation : two triangles are formed.



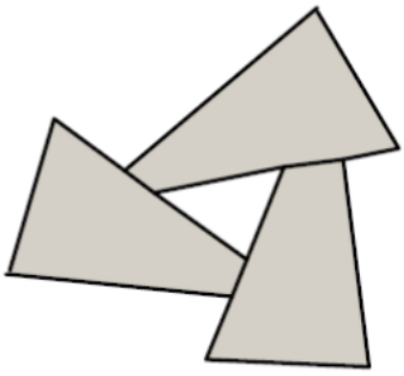

The algorithm terminates when all faces have been checked for clipping or all three points are outside the frustum.
The algorithms is simple but can produce many triangles since it can potentially double at every check

# 9 Hidden Surfaces

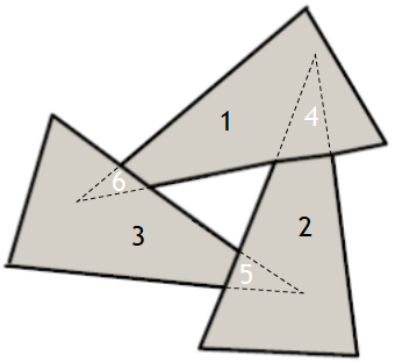
In a complex scene where many objects can overlap it is important that polygons closer to the viewer **cover** the objects behind them. If faces are not drawn in the correct order unrealistic figures will appear.Respecting the proper order of primitives visualization is called **Hidden Surfaces Elimination**.



When dealing with non-transparent object a technique called **Painter Algorithm** is applied : primitives are drawn in reverse order with respect to the distance from the projection plane : in this way,objects closer to the the view cover the oner further away.
In this figure above the drawing order is cube - sphere - cylinder. There are cases in which the painter algorithm cannot be applied.



In this case the primitives must be split so that is possible to find a proper order of the considered pieces.




Three main algorithms are used for hidden surface elimination:

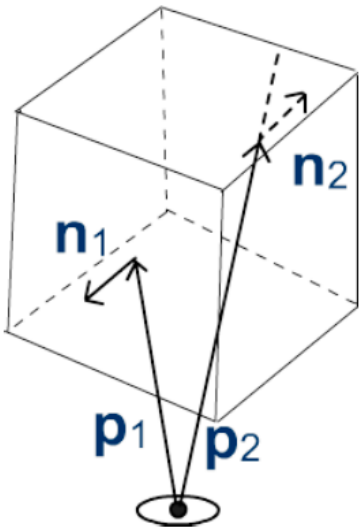
- **Back-face culling**
  Allows identifying and excluding objects that belong to the back of an object
- **Occlusion culling**
  Excludes objects that fall completely behind others ( not covered in the course)
- **Z-Buffering**
  Allows to implement the painter algorithms without sorting the polygons by distance. It is a per pixel based algorithm.

Z-Buffering alone is **enough** for hidden surface performance. However Occlusion and Back-face culling can increase the **performance**.

## 9.1 Back-face culling

This techniques allows to exclude objects that belong to the backside of a mesh simply considering:

- **Normal vectors**:
  Can be stored either with the faces or computed on the fly if the vertices are ordered in a specific direction.
- **Projection vectors**:
  Depend on the projection type



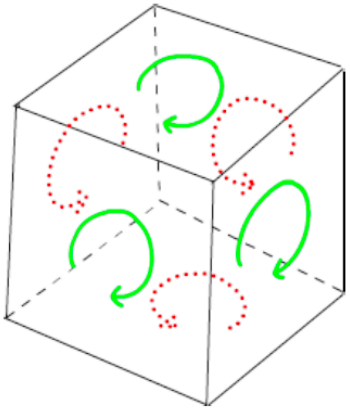

A cube is encoded as a set of 12 triangles. All the triangles are obviously **planar** surfaces. Each triangle surface has its **normal** vector n ,directed outside. The projection rays are directed from the viewer to the object. By performing a **scalar product** $p \cdot n$ we obtain:

- $p \cdot n < 0$ then the object belong to the **front**
- $p \cdot n > 0$ then the object belong to the **back** and is occluded by the front faces.
- $p \cdot n = 0$ then face is perfectly aligned with the projection rays and so it is seen at most as one single pixel

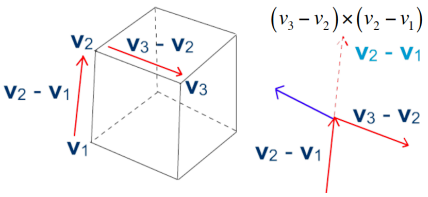
The normal vector :

- can be stored together with the face. This is a simple and fast solution but required more memory. Changing the position of the face must also result in a change in the normal vector.

- can be easily computed at run time from the vertices of the object if they are stored using a consistent order (eg.: clockwise).

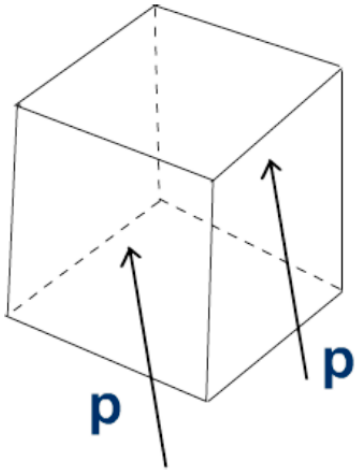


The cross-product of two vectors is a vector that is perpendicular to the plane (direction is determined by the right hand rule). In triangles the normal vector can be computed starting from the difference of two vectors (identified by the vertexes) and then the cross product of the two differences is computed.

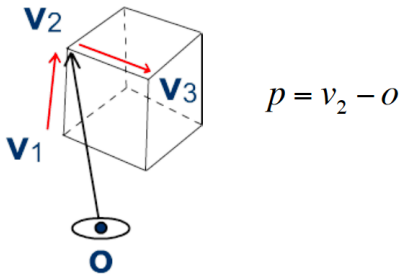



The projections rays:

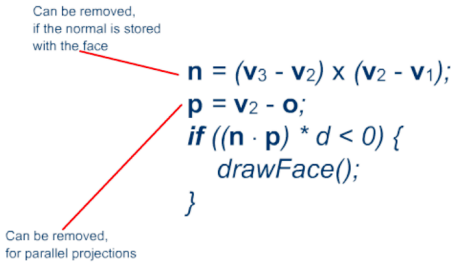
- in **parallel** projection the vector p is **fixed** and corresponds to the direction of the projection ray.




- in **perspective** projection the vector p must be computed relative to one of the vertices ( planar faces means that any vertex is equivalent) and the center of projection O : $p = v - O$




We must also take into account that scaling with **negative** values (central or planar mirroring) may invert the direction of the vertices. The sign to accept/reject a face must change accordingly.
To summarize :

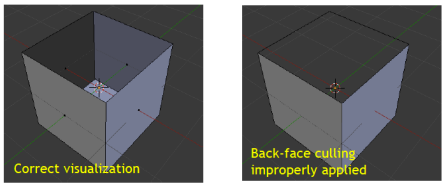


Where d is a constant of +1 if we want to accept faces ordered clockwise or -1 if we want to accept faces ordered counter clockwise.
Back-face culling can be applied :

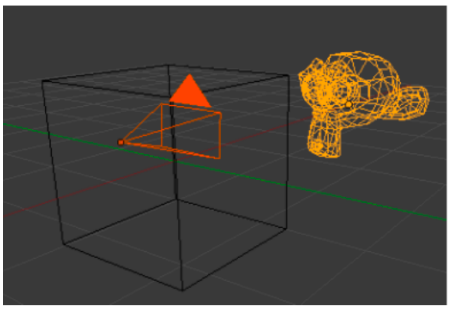
- **before** the world and view transformations.In this case the center of projection or the projection rays have to be mapped in the local space of the object by **inverting** the view and world transformations.
- **after** the world and view transformations. In this case if the vector has been stored with the face then also the normal vector needs to be transformed.

Finally back-face culling cannot always be applied:

- in **non-2 manifold** objects ( where we can have holes or lamina faces).
- in **transparent** objects where the back face must be visible from the front face



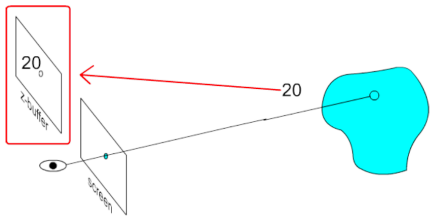

Usually engines separate non-2 manifolds from 2-manifolds for this reason.
Another special situation is when the camera is **inside** an object ( for example inside a room/box). In this case the normal is oriented in the wrong direction. When designing a scene this must be taken into account if back-face culling is applied : implementing this kind of situation without special adjustments means that the back-face culling algorithm hides the surface ( basically the camera shows objects behind a solid wall which he should not really see).



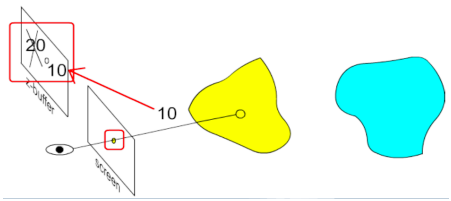
A solution to this is to create a room/skybox with the normal vectors pointing towards the **inside.**

## 9.2 Z-Buffering

Applies the painter algorithm to the **pixel** resulting from complete projection sequence rather than the faces of the objects drawn. This results in an extra memory area ( **z-buffer** ) that stores additional information for every pixel on the screen ( for an HD screen almost 2 million values so around 8MB!).
The algorithms draws all primitives on the screen that have passed **clipping** and **back-face culling** and tests whether to draw their corresponding pixels on screen. For each pixel both the color and the **distance** from the observer are computed.The z-buffer stores the **distance** (i.e. the **z-coordinate**) for each pixel on the screen.




Then when a new objects is tested the algorithms checks ,pixel per pixel, if it is already in the buffer.If it is and the new distance is lower then it is **updated** in the z-buffer.




### 9.2.1 Z-Buffering issues

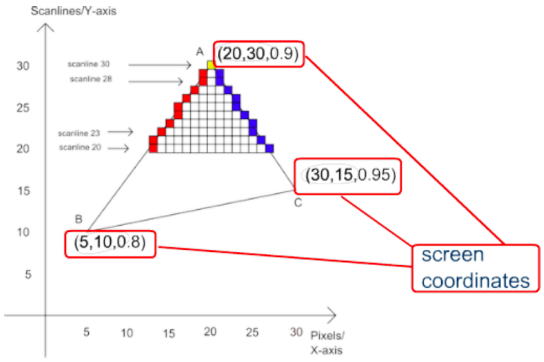
**Issue 1**

This technique is very simple and solves all the hidden surface problems

but it also computationally expensive. It not only requires to save the z-buffer area but also to compute all the **pixels** of the primitives. This is way back-face culling or occlusion culling **improve** the performance.

**Issue 2**

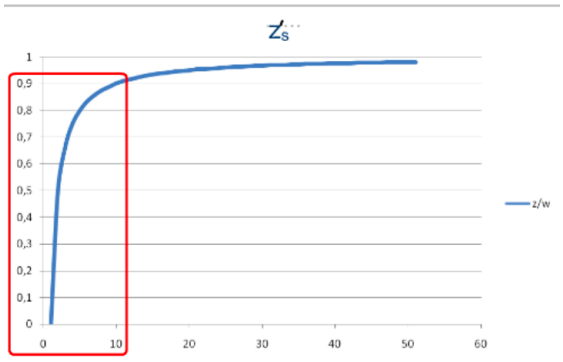
When dealing with triangles and their pixels also the z-coordinate must be interpolated : this is not always easy. **Normal screen coordinates** of x,y,z are in the range -1,1 and **pixel coordinates** are even more compressed for z in the range $z_n \in [0, 1]$.



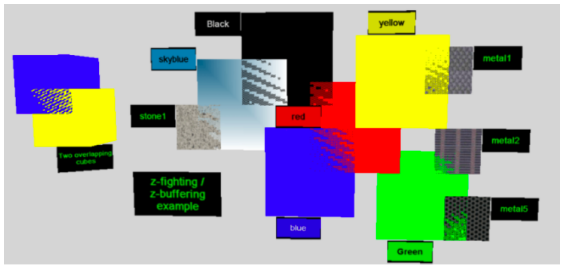

With these coordinates the vertices can be stored and from there via **interpolation** the corresponding pixels can be found. However the z-coordinate on screen has a **non-linear** behaviour : the distance between the lines becomes smaller as we move away from the plane and cannot be obtained by using interpolation in local/world/camera or clipping coordinates. The solution is to use the component $z_s$ of the **normalized screen coordinates** where $z_s = \frac{z_c}{w_c}$ is obtained from the clipping coordinates (see **perspective correct interpolation** with proof later on).

**Issue 3**

Another issue is **numerical precision** : the largest part of the $[0, 1]$ range of the $z_s$ coordinates is used for the points that are very **close** to the projection plane (and not much used in the real scene)



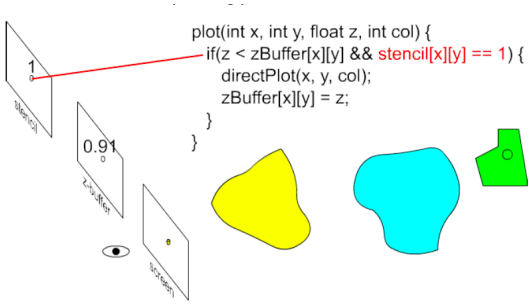

Since values a **discretized** we need more precision to store values that are further away otherwise a problem called **Z-fighting** occurs: when two almost co-planar figures are rendered the final **color** is determined by the round off error.




Since the $z_S$ coordinate is normalized with respect to the **near** and **far** planes these two parameters cannot be set to be arbitrarily small/large but must always be appropriate for the scene.

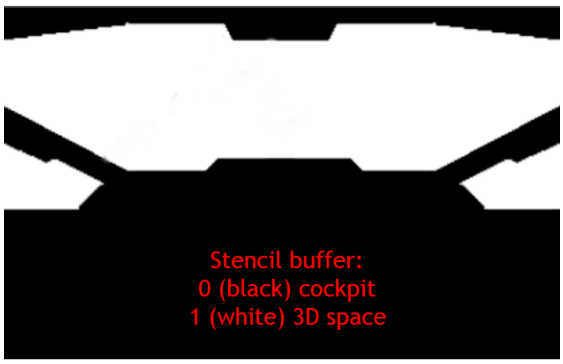
### 9.2.2 Stencil buffer

It is a technique similar to Z-buffer ,adopted to prevent an application to draw in some region of the screen.Again it is per pixel so an extra memory area (**stencil buffer**) is used to store data about each pixel. This kind of buffer is used to render for example head up displays in games like Wing Commander. More complex applications use the stencil buffer to draw **shadows** and **reflections** in multi-pass rendering techniques.




The buffer is a memory area that stores an integer information for each pixel (encoded at bit level). Usually binary values are stored :

- 1 the pixel must be drawn
- 0 the pixel can be skipped



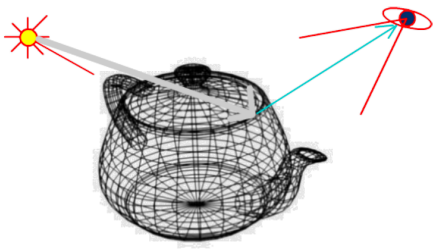

But more complex encodings/scenarios can be used with the stencil buffer.

# 10 Lights and rendering

To obtain realistic images with filled 3D primitives, **light reflection** should be correctly emulated.Some definitions:

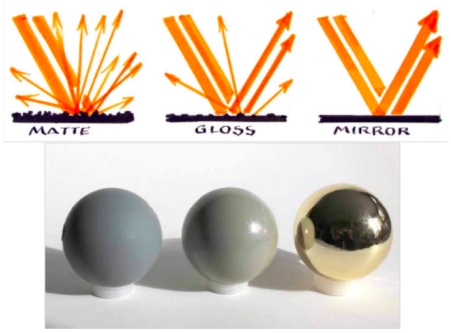
- **Rendering** reproduces the effects of the illumination by defining light sources and surface properties.
- **Lights** are the sources of illumination ,used to makes objects visible
- **Materials** set the parameters that determine how light is reflected.

The algorithms used to compute the colors of a surface starting from lights and material are called **Shaders**.
As seen in Chap.1 light sources emit different frequencies of the spectrum and objects **reflect** part of them. The photons emitted by the lights source bounce of the objects and some of them reach the viewpoint (camera) : rendering computes the **quantity** and the **color** of such photons.

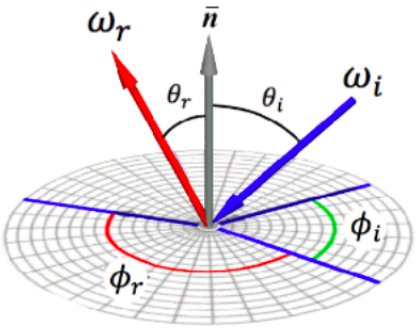


## 10.1 The Bidirectional Reflectance Distribution Function

Rendering must be able to define the **radiance** (quantity of light) received in each point of the projection plane ( each pixel) according to the direction of the corresponding projection ray.The **direction** and **intensity** of reflection is determined by the material that compose the surface of an object : the direction bounce depends on the microscopic structure of the surface




These surface properties can be encoded by the **bidirectional reflectance distribution function (BRDF)**




The inputs of the function are $\omega_i$ (direction of incoming light) and $\omega_r$ (direction that tells us if the objects reflects or not light in that direction). The output is the probability of reflecting in direction $\omega_r$ coming from $\omega_i$. In the case of a **mirror** surface, the probability will be equal to 0 for all $\omega_r$ except the one corresponding to angle $\theta_r = \theta_i$.

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\omega_i, \omega_r)$$

This function allows considering different ways in which the incoming radiance can be reflected at different angles,depending on the material. The BRDF function is expressed as sum of two terms:

- the **diffuse reflection**
  Represents the main color of the object

- the **specular reflection**
  Shiny objects tend to reflect the incoming light in a particular angle called the specular direction

$$f_r(x, \vec{lx}, \omega_r) = f_{diffuse}(x, \vec{lx}, \omega_r) + f_{specular}(x, \vec{lx}, \omega_r)$$

## 10.2 Rendering equation

The BRDF allows relating together the irradiance in all the directions for all points in a scene in the so called **rendering equation**:

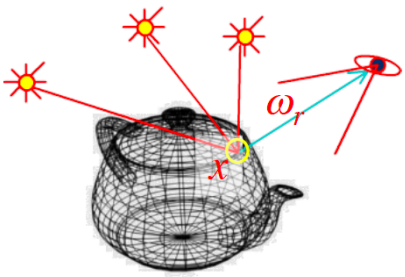
$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \vec{yx}) f_r(x, \vec{yx}, \omega_r) G(x, y) V(x, y) dy$$

This is a very complex equation but a simplified equation is available : the **scan-line rendering equation**. The approximation that is done is that only a specific **finite** set of lights sources illuminate the scene (sum over l light sources). The contributions of each light source l to the final color of the pixel are added together. Each term is determined as the product of :

- the **light model** : quantity and direction of the light source. Vector $\vec{lx}$ connect the light l to the point x.
- the **BRDF** of the surface that reflects the light

$$\boxed{L(x, \omega_r) = \sum_l L(l, \vec{lx}) f_r(x, \vec{lx}, \omega_r)}$$

This equation tells the total radiance of point **x** of an object in a direction $\omega_r$ of the space.

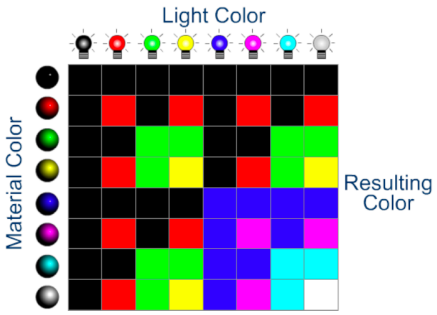



### 10.2.1 Scan-line : considering colors

The scan-line equation should be repeated for **every wavelength** of the light (the three different RGB channels):

$$L(x, \omega_r, \lambda) = \sum_l L(l, \vec{lx}, \lambda) f_r(x, \vec{lx}, \omega_r, \lambda)$$

The lights have associated RGB values that accounts for the photons emitted for each of the three main frequencies.Objects are characterized by a different BRDF for each of the three main frequencies : three separate images are produced independently and summed up to produce the final image.
Due to the separation of color components , combinations of lights and material colors can lead to unexpected results :




For example purple light illuminating a yellow objects results in a red color. This is because in the scan-line equation one of the terms is equal to zero .
In the scan-line rendering equation all terms are $\in [0,1]$ range. It can happen that the sum of the terms $\sum_l L(l,\vec{lx}) f_r(x,\vec{lx},\omega_r) > 1$. To solve this **clamping** is applied :

$$\boxed{L(x,\omega_r) = \text{clamp}\left(\sum_l L(l,\vec{lx}) f_r(x,\vec{lx},\omega_r)\right)}$$

$$clamp = \begin{cases} 0 & y < 0 \\ y & y \in [0,1] \\ 1 & y > 1 \end{cases}$$

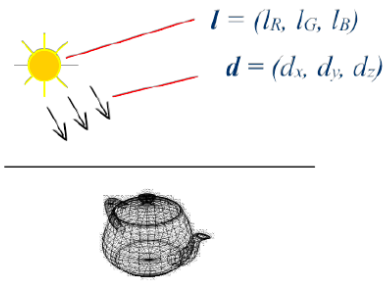
This effect can be seen in overexposed images : areas that contain RGB values greater than 1 will be clamped to be 1. So for example $(7.3, 2.5, 1.6)$ after clamping becomes $(1,1,1)$ which translates to the **white color** , characteristic of overexposed photos.

## 10.3 Light models

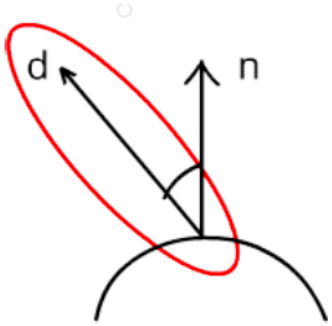
Here are considered three different light models that describe how light is emitted in different **directions** of the space and at which **intensity**.

### 10.3.1 Direct light models

Direction or direct lights are sources of light that are **far away** from the objects (for example the sun in a scene). Due to the distance of the source the light rays are **parallel** to each other in all the positions of the space.




- $d = (d_x, d_y, d_z)$ specifies the **light direction** that is independent of the position of the objects in the scene ($\vec{lx} = d$).Conventionally the direction vector d is unitary ($|d| = 1$) and is pointing **from** the object **to** the light




- $l = (l_R, l_G, l_B)$ is the RGB component vector that defines the light color.Since the quantity of light emitted per direction is the same, all parts of the scene receive the same amount of red, green and blue.This means the $L(l, \vec{lx}) = l$

So the scan-line equation becomes :

$$\boxed{L(x, \omega_r) = l * f_r(x, d, \omega_r)}$$

The star * indicates the element wise product of the terms ( l is a vector of colors , and the BRDF returns also a vector) since we are not considering the single color frequencies $\lambda$.

### 10.3.2 Point Lights

Point lights emit light from a **fixed** position in space characterized by their position. Rotating the light while maintaining the position does not alter the scene

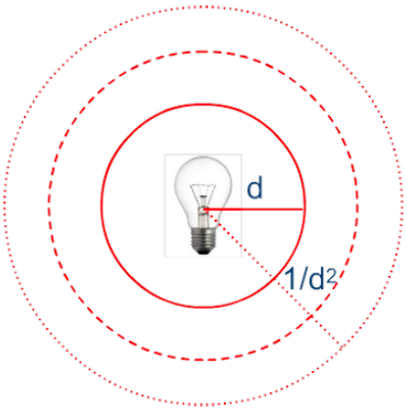
(like a lamp in a room) : they emit light in all the directions, starting from a specific position in space.

- $p = (p_x, p_y, p_z)$ position of point light
- $l = (l_R, l_G, l_B)$ color vector
- the direction (required by BRDF) varies according to the point **x** of the object that is illuminated

$$\vec{lx} = \frac{p - x}{|p - x|}$$

The vector is normalized to make the vector unitary.

To reproduce the physical properties of light sources, point lights are characterized by a **decay factor** : the intensity of a point light reduces at a rate that is proportional to the **inverse of the square of the distance**




Implementing this results in images that often are very dark as the light does not bounce of the other objects resulting in a more illuminated area. This is where the **decay factor** $\beta$ comes in handy. Another parameter that increases physical behaviour is the **scaling factor g**.

$$\boxed{L(l, \vec{lx}) = l \left( \frac{g}{|p - x|} \right)^{\beta}}$$

- $\beta = 0 \rightarrow$ constant light without decay.
- $\beta = 1 \rightarrow$ inverse linear decay

- $\beta = 2 \rightarrow$ inverse squared decay (most like in reality)

The scan-line rendering equation becomes (with decay) :

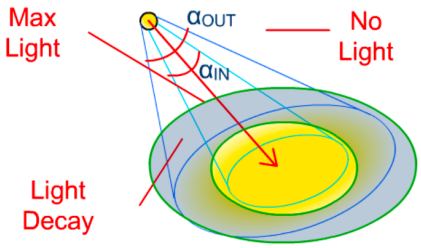
$$\boxed{L(x, \omega_r) = l \left( \frac{g}{|p - x|} \right)^{\beta} * f_r \left( x, \frac{p - x}{|p - x|}, \omega_r \right)}$$

The scan-line rendering equation becomes (without decay) :

$$\boxed{L(x, \omega_r) = l * f_r \left( x, \frac{p - x}{|p - x|}, \omega_r \right)}$$

### 10.3.3 Spot lights

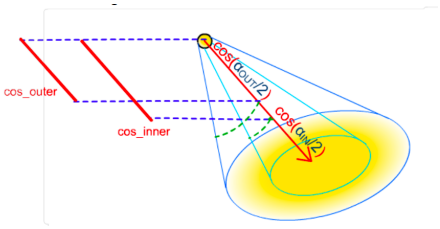
Spot lights are light sources characterized by a **position p** from where they emit in a **distance d**.




Spotlights divide the scene into three area :

- where direct light (max light) is received ( $< \alpha_{IN}$ )
- where decaying light is received ($> \alpha_{IN}, < \alpha_{OUT}$)
- where no light is received ($> \alpha_{OUT}$)

The closer the two angles are together the clearer is the cut . The more they are fare away the more the light is faded.
Since the spotlight is an extension of the point light it is characterized by the same color vector **l** and the **decay factor** $\beta$. For the implementation of the spotlight usually the **cosine** of the half-angles are used ( $c_{IN} > c_{OUT}$ )




Computing

$$cos\alpha_l = \vec{lx} \cdot d$$

where $\alpha_l$ is the angle formed by the vector from the center of the light to the point x. By applying clamping the **cone** of light is computed as:

$$\text{clamp}\left(\frac{cos\alpha_l - c_{out}}{c_{IN} - c_{OUT}}\right)$$

The scan-line equation becomes (with decay):

$$\boxed{L(x,\omega_r) = l\left(\frac{g}{|p-x|}\right)^{\beta} \cdot \text{clamp}\left(\frac{\frac{p-x}{|p-x|}\cdot d - c_{out}}{c_{IN} - c_{OUT}}\right) * f_r\left(x, \frac{p-x}{|p-x|}, \omega_r\right)}$$

The scan-line equation becomes (without decay):

$$\boxed{L(x,\omega_r) = l \cdot \text{clamp}\left(\frac{\frac{p-x}{|p-x|}\cdot d - c_{out}}{c_{IN} - c_{OUT}}\right) * f_r\left(x, \frac{p-x}{|p-x|}, \omega_r\right)}$$

## 10.4 BRDF Models

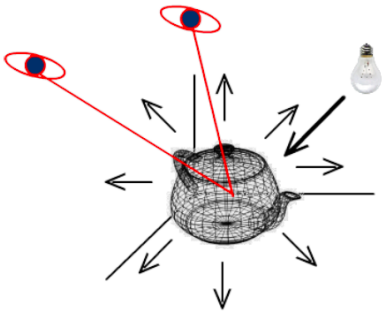
Here are considered three different BRDF models that describe how much light is reflected off the surface of an object. As already seen the BRDF is composed of two terms, the **diffuse** and the **specular** term. The **Lambert Model** refers to the diffuse part and the **Phong** and **Blinn** refer to the specular part.

### 10.4.1 Diffuse: Lambert Model

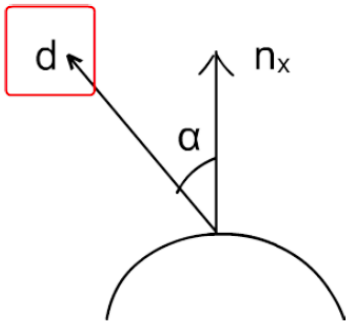
The Lambert Model would be enough to render the color of an object but it would appear **matt** (no **specular** part). The **Lambert reflection law** states that each point that is hit by a light ray,reflect it with **uniform probability** in **all**

directions. The reflection is **independent** of the viewing angle and corresponds to a **constant** BRDF :

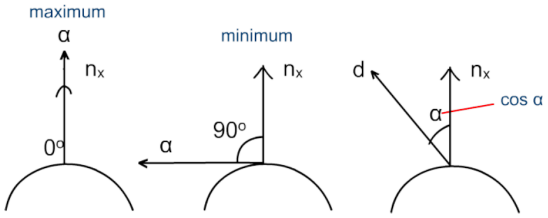
$$f_r(x, \omega_i, \omega_r) = \rho_x$$



However the **quantity** of light received by an object, depends on the **angle** between the ray of light and the reflecting surface. Consider $n_x$ being the normal (normalized) vector to the surface, $d$ the direction of the ray of light of unitary length directed from the object to the source of light ( convention) :




This way the amount of received light can be described :




As seen in the figure Lambert found out that the amount of light is proportional to the cosine of the angle between the direction of the light and the normal to the

surface:

$$\vec{d} \cdot \vec{n_x} = cos\alpha$$

Next the behaviour of the BRDF function with respect to a specific color must be analysed . The vector

$$m_D = (m_R, m_G, m_B)$$

(**diffuse color**) expresses the capability of a material to perform the Lambert reflection for each of the three primary color frequencies RGB (so $\rho_x = m_D$ , the constant assigned to the BRDF). This tell how each frequency is reflected by the object. The diffuse part of the BRDF becomes :

$$\boxed{f_r(x, \vec{lx}, \omega_r) = f_{diffuse} = m_D \cdot \text{clamp}(\vec{lx} \cdot n_x)}$$

The clamp function limits the scalar product in range [0,1] instead of [-1,1]. Normally if the cosine of the angle is -1 then we're referring to the backface of the object which will be excluded by back-face culling. It is still good practice to set the received light to 0.
Point light (no decay )model example :

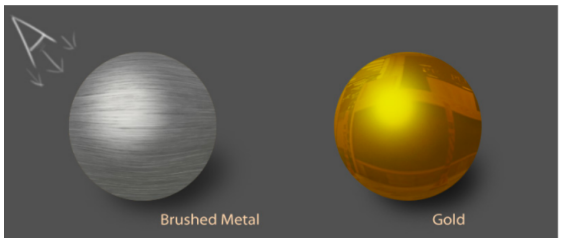
$$L(x, \omega_r) = l * m_D \cdot \text{clamp}\left(\frac{p - x}{|p - x|} \cdot n_x\right)$$

Directional light model example :

$$L(x, \omega_r) = l * m_D \cdot \text{clamp}\left(d \cdot n_x\right)$$

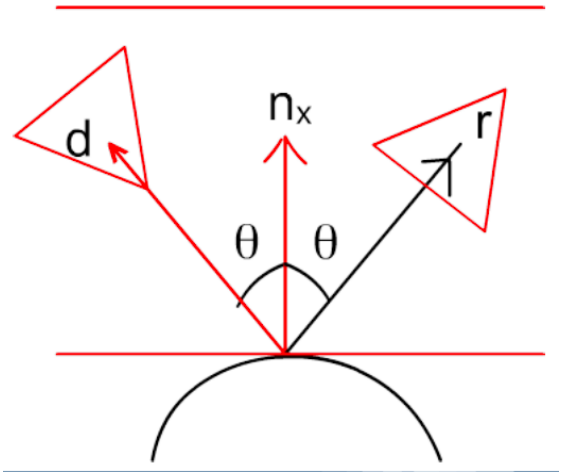
### 10.4.2 Specular: Phong Model and Blinn Model

Most of the materials reflect more than the incoming light in the specular direction.This part is encoded in the $f_{specular}$ term of the BRDF. As for the diffuse case, this component is characterized by a color $m_s = (m_{Rs}, m_{Gs}, m_{Bs})$ that defines how the RGB components of the incoming light are reflected. Most materials have a **white** specular color so $m_s = (1, 1, 1)$ but some metallic materials like copper have a specular color identical to their diffuse




Phong Model

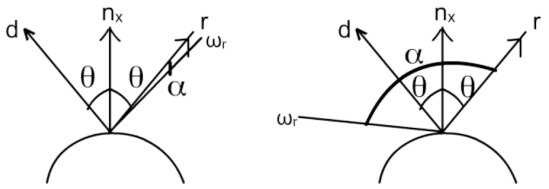
In the Phong model the specular reflection has the **same** angle $\theta$ as the incoming ray with respect to the **normal** vector, but oriented in the **opposite** direction, positioned on the same plane as the light and the normal vectors.




Vector $\omega_r$ points in the direction from which the object is being observed in the BRDF. So depending on the type of projection :

- **parallel** projections : rays are all parallel to $\omega_r$ so its a **constant**
- **perspective** projections : $\omega_r = \frac{c-x}{|c-x|}$ , where c is the center of projection and x the point on the surface.

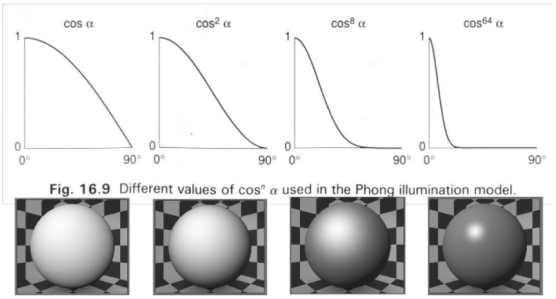
The Phong model computes first the angle $\alpha$ between the reflecting direction r and the direction $\omega_r$




The greater angle $\alpha$ is the less the reflection will be seen : again it is proportional to $cos\alpha$.
To create more contained highlight regions , the term $cos\alpha$ is raised to the power of $\gamma$ :

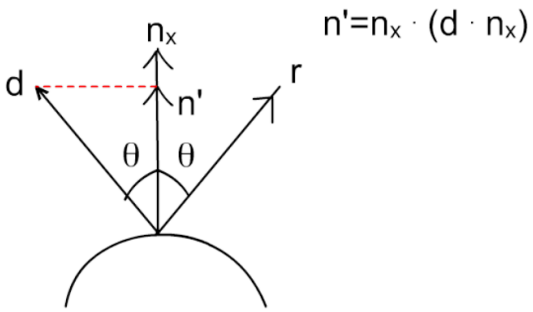
$$cos^{\gamma}\alpha$$

The greater is $\gamma$ the **smaller** is the highlight and the more **shiny** the object appears



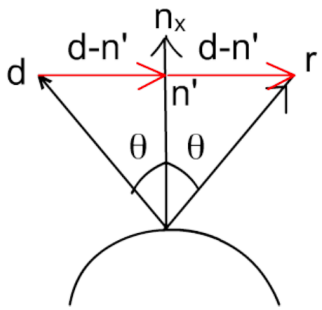

To compute the **direction** of the **reflected ray** :

1. compute $n' = n_x \cdot (d \cdot n_x)$




2. compute $d - n'$ to obtain the perpendicular from d to n

3. subtract $(d - n')$ **two times** to obtain the reflected vector r




$$\boxed{r = d - 2(d - n_x \cdot (d \cdot n_x)) = 2n_x \cdot (d \cdot n_x) - d}$$

Or using the **rendering notation**:

$$\boxed{r_{l,x} = 2n_x \cdot (\overrightarrow{\mathrm{xl}} \cdot n_x) - \overrightarrow{\mathrm{xl}}}$$

When using **shaders** there is a build-in function called `reflect(d,n)` that automatically computes the reflection ray given the normal and the light direction.
The **intensity** of the specular reflection can be computed as:

$$\boxed{COS^{\lambda}\alpha = clamp(\omega_r \cdot \mathbf{r})^{\lambda}}$$

where **r** is the reflected ray and $\omega_r$ is the direction from which the observer is looking raised at the power of $\gamma$ (the greater $\gamma$ the smaller the reflection highlight ( as seen already above).
In summary we have :

- $r_{l,x} = 2n_x \cdot (\overrightarrow{\mathrm{lx}} \cdot n_x) - \overrightarrow{\mathrm{lx}}$
- $f_{\text{specular}}(x, \overrightarrow{\mathrm{lx}}, \omega_r) = m_s \cdot \text{clamp}(\omega_r \cdot r_{l,x})^{\gamma}$ where $m_s$ is the specular component color.

Blinn Model

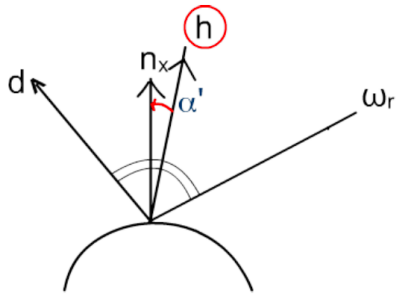
The Phong model is quite **complex** to compute: to obtain the reflection ray a lot of steps are required. The Blinn model obtains this reflection ray in an alternative way by trying to **approximate** it.

The Blinn model uses **half-vector h** , which is the vector in the middle of $d$ and $\omega_r$. The angle $\alpha'$ between $n_x$ and $h$ is the **approximation** of the angle $\alpha$ between the **observer** and the **reflected ray**.
Half vector h can be computed as

$$h_{l,x} = \frac{\overrightarrow{\mathrm{lx} + \omega_r}}{|\overrightarrow{\mathrm{lx}} + \omega_r|}$$

which leads to following color formula :

$$\boxed{f_{\text{specular}}(x, \overrightarrow{\mathrm{lx}}, \omega_r) = \mathbf{m}_S \cdot \mathbf{clamp}(n_x \cdot h_{l,x})^\gamma}$$




Blinn vs Phong

- Phong , in the end, is **less** complex than Blinn as Blinn requires a **normalization** which is more expensive than a **reflection**.
- Blinn is better if $d$ and $\omega_r$ are constant ( like in parallel projections with directional lights and fixed camera) : this way **h** is also constant in the entire scene making this model more efficient than the Phong one.

The two techniques are different and lead to different results. This is why almost always **both** methods are implemented and the final model is chosen depending on the scene that must be reproduced.

# 11 Smooth Shading

To compute the color of each pixel the **rendering equation** is used. Objects are made of **meshes** , several faces but to solve the rendering equation different approaches can be used:

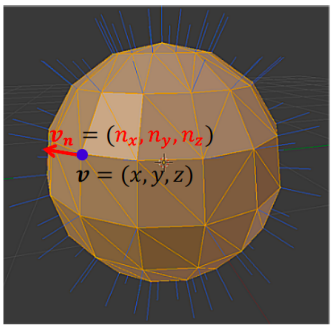
- **Per-face computation**
- **Per-vertex computation**
- **Per-pixel computation**

The smaller the per-object computation the more computational expensive it is. For a Full-HD screen the per-pixel computation requires solving the rendering equation 2 million times but the final result is more visually appealing than the one of the other methods. To guarantee good real-time results rendering should be performed using modern software and hardware techniques (i.e. shading languages executed on GPUs). With current technology there is no gain in using per-face or per-vertex and since per-vertex gets the better final result the **per-face** computation is **no longer used**. The per-vertex and per-pixel computations allow to obtain **smooth shadings**.
Polygonal surfaces can be encoded in many ways (already seen) and often result in **sharp edges**. These edges can be rendered as **continuous surfaces** using **smooth shading** algorithms , with the main 2 being:

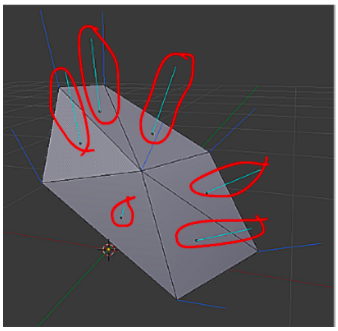
- Gouraud shading (**per-vertex**)
- Phong shading ( **per-pixel**)

Both techniques require extending the **encoding** of the **vertex** ( 3 additional normal vector components) :

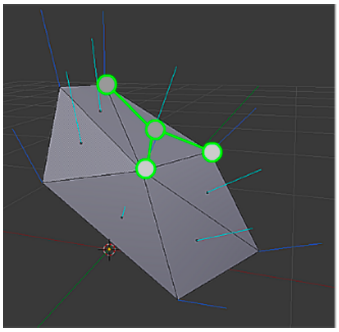
$$v = (x, y, z, n_x, n_y, n_z)$$



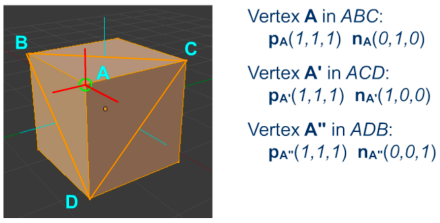

The normal is used to compute the color in the rendering equation.
As seen meshes are composed of **triangles** , which has three vertexes . Each triangle has a **normal** to its surface (identical in all points of the triangle).



Adjacent surfaces can have very different normal vectors and this abrupt change can cause abrupt changes in the **color**. The result is the effect of a **not smooth** color transition.
With **shading techniques** this does not happen : they store for each vertex the Cartesian coordinates and the direction of the normal vector to the **original surface**. So instead of using for each pixel on the face the normal (equal for all points of the surface) the corresponding normal is computed via **interpolation** starting from the **3 vertexes** that make up the triangle.



As already seen different vertexes can belong to different triangles. Depending on the triangle a vertex belongs to, its **normal vector changes**.




If the norm for such shared vertexes is the **same** for all surfaces then the result is a **smooth surface**. So depending on the final effect , the normal vectors should be the **same** if a **smooth** effect is desired (like in a sphere) or should be **different** if a **sharp-edge** effect is desired (like in a cube).

## 11.1 Gouraud shading

The Gouraud shading uses the **light** and **reflection** models to compute the color of each vertex. The colors of the **inner points** are computed using **interpolation** .

Vertex colors for static objects can be **pre-computed** and stored with the vertex geometry if:

- the objects is illuminated by **static lights**
- the objects's material BRDF does not depend on the direction of the observer ( i.e. just **diffuse** component using the Lambert Model)

## 11.2 Phong shading

(Not be confused with Phong reflection model!)
This technique compute the color for **each pixel** separately. In this case **vertex normal vectors** are interpolated to **approximate** the normal vector to the actual surface (the Gouraud techniques directly interpolates the color).This can lead to inner normal vectors the are **longer** the the unit. To avoid this at each step a **normalization** must be performed.
The last step is to compute the illumination model for every pixel using the **interpolated** normal vectors.
The **normalization** step together with solving the rendering equation for **each pixel** makes the Phong shading technique very expensive but nonetheless it achieves very good results. On the other hand the Gouraud is less complex but can sometimes result in **less appealing** visual effects because the techniques **misses** some important illumination / shading details. Generally speaking if the number of vertex is very high ( so each triangle is only a few pixels big) then the performances are **almost equivalent**.

## 11.3 Transformations of normal vectors

Since the normal vector are now stored with the objects the must behave correctly in the case of **view** and **world transformations** .Normal vectors however don't follow the same computations: they are **3D directions** composed of three components and **not homogeneous coordinates** (with 4 components).The **transform** matrix for the **normal** vectors can be obtained starting from the 4x4 matrix that transforms homogeneous coordinates by computing the **inverse transpose** of its 3x3 **upper-left sub-matrix**:

# 12 Ambient Light Emissions

Using only direct light sources (directional,point or spotlight ) can lead to **dark** images. Realistic lightning techniques try to incorporate also **indirect lighting** : illumination caused by lights that **bounce** from object to object.The simplest approximation of such indirect lights is **Ambient Lightning**.
The **ambient light emission factor** ( constant for the entire scene) accounts for all the light reflected by all the objects in all the directions.The **ambient light emission** is specified by an RGB color value :

$$l_A = (l_{AR}, l_{AG}, l_{AB})$$

Each **material** has an **ambient light reflection color** ( usually same as diffuse color):

$$m_A = (m_{AR}, m_{AG}, m_{AB})$$

The contribution to the color of the object is computed by multiplying the ambient light emission with the ambient light reflection

$$l_A \cdot m_A$$

For example in the Phong specular model with single directional light :

$$r_x = 2n_x \cdot (d \cdot n_x) - d$$

$$L(x, \omega_r) = \text{clamp}(l_D \cdot (m_D \cdot \text{clamp}(d \cdot n_x) + m_s \cdot \text{clamp}(\omega_r \cdot r_x)^\gamma) + l_A \cdot m_A)$$

The **emission** term of a material accounts for small amounts of light emitted directly by an object ( for example modem lights, television button lights...). Usually such light is **equal** in all directions.

$$m_E = (m_{ER}, m_{EG}, m_{EB})$$

$$L(x, \omega_r) = \text{clamp}(l_D \cdot (m_D \cdot \text{clamp}(d \cdot n_x) + m_s \cdot \text{clamp}(\omega_r \cdot r_x)^\gamma) + l_A \cdot m_A + m_E)$$

As seen in the last equations the ambient lights terms are considered **once** while the BRDF terms are considered for **each light**.
Both ambient light and emission are **independent** of the **direction** of the **light** and **observer**. Considering these two **without** other direct light sources result in

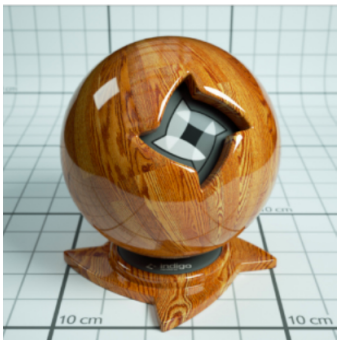
**flat images**.
Finally , the ambient light term of the material $m_A$ is **scaled** by the ambient color of the scheme $l_A$ which allows controlling the color of the objects in a scene-dependent way, adapting the illumination to dark or lit scenes.

# 13 Textures

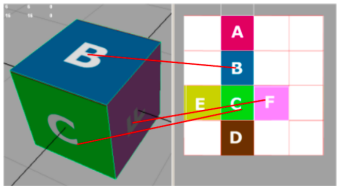
To make 3D images realistic great detail is required. However assigning **different** materials to large number of **very small** triangles is **not feasible** due to memory requirements. The common approach uses **tables** to assign different values to the parameters (color of the diffuse material,color of ambient light, color of specular term ecc..)of the shaders for the internal points of the surface. Most commonly such tables contain the **diffuse color** of the objects, acquired from an image. The considered tables ( or **images** ) are called **textures** (or maps).
Textures can be :

- 1D
- 2D $\rightarrow$ in this course
- 3D

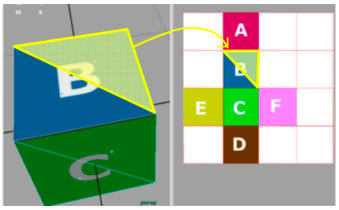
Texture images that define the surface of an object are **planar** . However the objects in a scene often have **complex non-planar** topology.



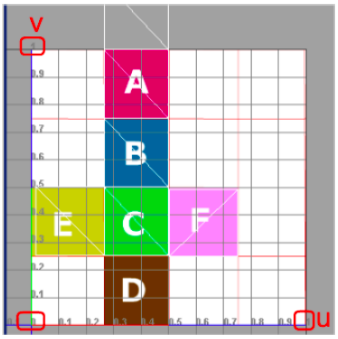

2D textures are applied to 3D objects using a **mapping relation** that associates each point on the surface with a point on the texture. The mapping creates a correspondence between pixels of the textures (**texels**) and points of the object.



As seen in the figure above this often leads to unused areas of the texture ( see white spaces).
In case of **polygonal** objects the mapping create a correspondence between **triangles of the mesh** and **triangles over the texture**

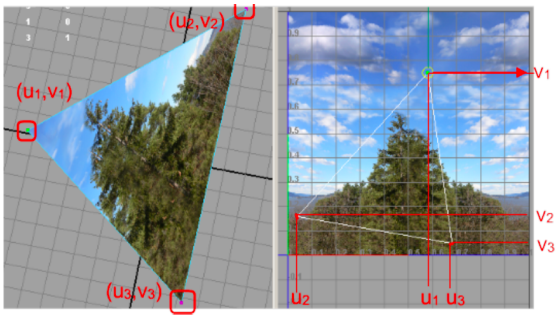



Point of the 2D textures are addressed using **UV Coordinates** ( Cartesian coordinates with axes u and v)

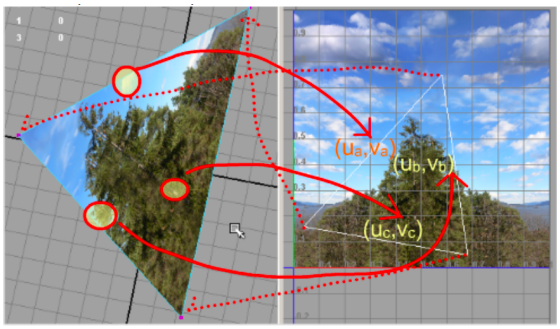



The coordinates range between 0 and 1 which implies that all texture images, regardless of their aspect ratio are **stretched** to fit into a square. UV coordinates are **only** assigned to the **vertexes** of the triangles.
Vertex now have 8 components :

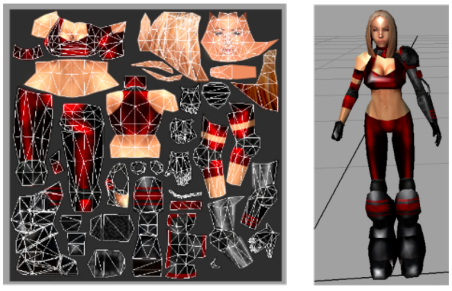
- positional values $x, y, z$ to support projection
- normal values $n_x, n_y, n_z$ to support smooth shading
- UV Coordinates $u, v$ to support textures




Internal points are then interpolated



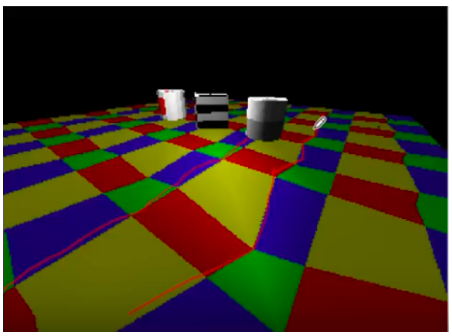

The more the shape of the triangles defined in the objects and the triangles of the texture are similar the **less distorted** will be the applied texture. This matching process is very long and tedious.



As seen above mapping is not a **bijection** : the same part of the texture can be shared by several triangles of the 3D object. This feature can be exploited to reduce memory usage.

### 13.0.1 Perspective interpolation

Conventional interpolation often leads to **wobbling** images if **perspective** is used.



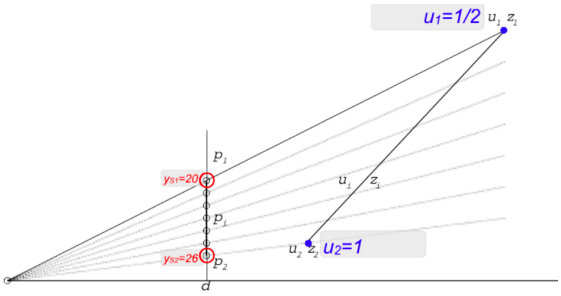
This problem is not only restricted to the textures parameters but also affects parameters like the normal vectors or colors. However for those other parameters the perspective effect is **less perceivable** and can be easily corrected with linear approximations.
An example is provided to explain why perspective interpolation is a problem for textures. Two points in a 3D space are given with **u-coordinate** and **z-coordinate** (distance from projection plane):

- **P1** has $u_1 = \frac{1}{2}$ , $z_1 = 0.3$
- **P2** has $u_2 = 1$ , $z_1 = 0.1$

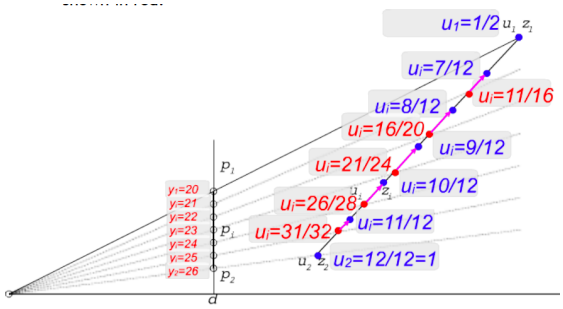
The two points are projected on the projection plane at coordinates

$$y_{S1} = 20$$

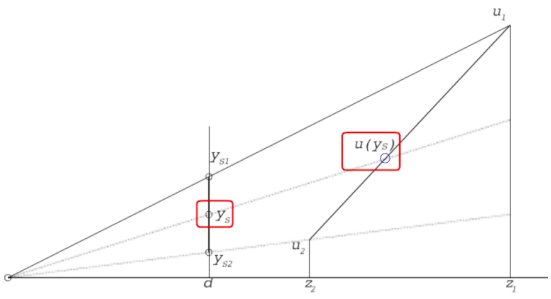
$$y_{S2} = 26$$



So the pixels at coordinate $y_{S1} = 20$ and $y_{S2} = 26$ will have **u-coordinate** ,respectively, $u_1 = \frac{1}{2}$ and $u_2 = 1$. In a 3D space the u-coordinates vary **linearly**. Since there are 5 points between the two y coordinates , spacing the u-coordinates linearly results in the following:




In blue we can see the equally-spaced points that match the linear case. Unfortunately the actual values ( seen in red ) are found in a different manner because the inverse projection of 2D coordinates in 3D space is **non-linear**. To solve this issue **perspective correct interpolation** is used.
Consider parameter u which assumes values $u_1$ at $z_1$ and $u_2$ at $z_2$. The aim is to find the perspective correct value $u(y_S)$ of the parameter $u$ in 3D space, corresponding to a pixel $y_S$ on screen .




Recalling that

$$y_{S1} = \frac{y_1}{\frac{z_1}{d}}$$

$$y_{S2} = \frac{y_2}{\frac{z_1}{d}}$$

$$\dots$$

$$y_{Si} = \frac{y_i}{\frac{z_i}{d}}$$

Since $u$ is **linear in 3D space** it can be expressed as a linear combination of its value at the two extremes

$$u(y_S) = \beta u_1 + (1-\beta)u_2$$

$$x = \beta x_1 + (1-\beta)x_2$$

$$y = \beta y_1 + (1-\beta)y_2$$

$$z = \beta z_1 + (1-\beta)z_2$$

Also the pixel position on screen is **linear combination**:

$$y_S = y_{S1}\alpha + y_{S2}(1-\alpha)$$

The goal is to find a relationship $\alpha \to \beta$ :

$$u(y_{S1}\alpha + y_{S2}(1-\alpha)) = u_1 \cdot \beta(\alpha) + u_2 \cdot (1-\beta(\alpha))$$

1. $y_S = y_{S1}\alpha + y_{S2}(1-\alpha) \to \alpha = \frac{y_S - y_{S2}}{y_{S1} - y_{S2}}$
2. Since $y_{Si} = \frac{y_i}{\frac{z_i}{d}} \to y_S = \alpha\left(\frac{y_1}{\frac{z_1}{d}}\right) + (1-\alpha)\left(\frac{y_2}{\frac{z_2}{d}}\right)$
3. Since $y_S$ can be also found by interpolating separately y and z coordinate using factor $\beta$ and then **projection** the result : $y_S = \frac{\beta y_1 + (1-\beta)y_2}{\frac{\beta z_1 + (1-\beta)z_2}{d}}$
4. $\alpha\left(\frac{y_1}{\frac{z_1}{d}}\right) + (1-\alpha)\left(\frac{y_2}{\frac{z_2}{d}}\right) = \frac{\beta y_1 + (1-\beta)y_2}{\frac{\beta z_1 + (1-\beta)z_2}{d}}$
5. After some simplifications and computations :

$$\boxed{\beta = \frac{\frac{\alpha}{z_1}}{\frac{\alpha}{z_1} + \frac{(1-\alpha)}{z_2}}}$$

Since $u(\alpha) = u_1\beta(\alpha) + u_2(1-\beta(\alpha))$ :

$$\boxed{u(\alpha) = \frac{\alpha\frac{u_1}{z_1} + (1-\alpha)\frac{u_2}{z_2}}{\frac{\alpha}{z_1} + \frac{(1-\alpha)}{z_2}}}$$

Generally an internal point of a triangle on screen can be considered a linear combination of its three vertices where the coefficients sum up to 1 :

$$(x_s, y_s) = (x_1, y_1) \cdot \alpha_1 + (x_2, y_2) \cdot \alpha_2 + (x_3, y_3) \cdot \alpha_3$$

$$\alpha_1 + \alpha_2 + \alpha_3 = 1$$

So the value $u_S$ of $(x_S, y_S)$ is :

$$\boxed{u_S = u(\alpha_1, \alpha_2, \alpha_3) = \frac{\alpha_1 \frac{u_1}{z_1} + \alpha_2 \frac{u_2}{z_2} + \alpha_3 \frac{u_3}{z_3}}{\frac{\alpha_1}{z_1} + \frac{\alpha_2}{z_2} + \frac{\alpha_3}{z_3}}}$$

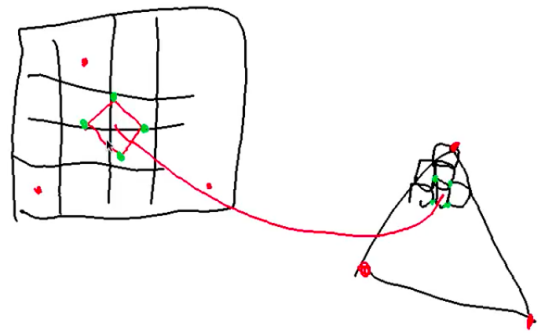
This process of correct interpolation should be done for all parameters considered **after the perspective projection**. As seen this method requires the distance of the point from the center of projection along the negative z-axis which corresponds to the **fourth component** of the **homogeneous coordinate vector changed of sign**

$$P_{persp} \cdot v = \begin{vmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} = \begin{vmatrix} x_c \\ y_c \\ z_c \\ -z \end{vmatrix}$$

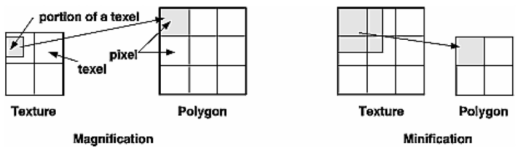
# 14 UV Mapping

Textures are images that are mapped over triangles on screen. However there are some issues that occur during mapping :

- UV coordinates are **floating point** numbers , while **texels** are indexed by integer numbers
- The pixel shape on screen might correspond to several texels on the texture ( shape mismatch)



When the texel is **larger** than the corresponding pixel a **magnification filtering** problem occurs. Vice-versa if the pixel corresponds to **several** texels a **minification filtering** problem occurs.




Minification is the **tougher** of the two filtering problems : it requires fusing together pixels which can be very tricky as you can easily loose detail ( for example if the texture has a text on it the letters could become unrecognisable ) .

The two filter problems can be solved also taking care of the floating to integer conversion.

## 14.1 Magnification Filtering

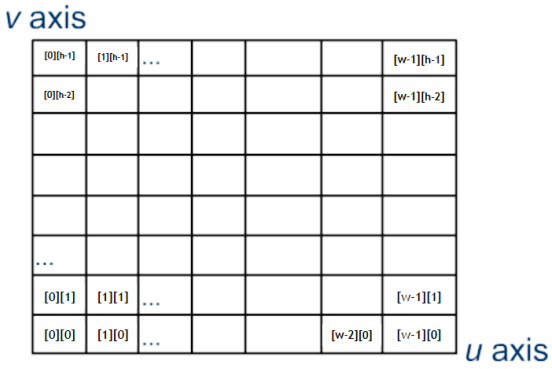
Two magnification filter are usually defined

- **Nearest pixel**
- **(bi)linear interpolation**

A texture of size $wxh$ is considered. Texels over :

- u-axis $\rightarrow$ indexed from $0-(w-1)$
- v-axis $\rightarrow$ indexed from $0-(h-1)$

So texel $P[0][0]$ is on the bottom left:




### 14.1.1 Nearest Pixel

With this technique the look-up procedure :

1. Transform the UV coordinates wrt the texture size :

$$x = u \cdot w$$

$$y = v \cdot h$$

where x,y are still **floating point** numbers ( as x,y $\in [0,1]$ and w $\in [0, w-1]$ ,h $\in [0, h-1]$

2. Return the texel $p[i][j]$ that consider only the **integer part** of the scaled coordinates

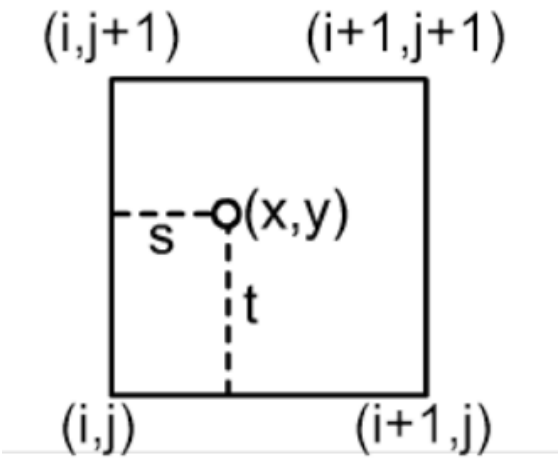
$$i = \lfloor x \rfloor$$

$$j = \lfloor y \rfloor$$

It is very fast and requires reading **one texel per pixel** but results in **blocky** images : this because close u-coordinates for example $u_1 = .01, u_2 = .02...$ are multiplied by the width , for example 32 resulting in $x_1 = .32 x_2 = .64...$ which results in $i_1 = 0, i_2 = 0, i_3 = 0$.

### 14.1.2 (Bi)Linear Interpolation

Linear interpolation interpolates the color of the pixel from the values of its closest neighbours.




1.
$$x = u \cdot w$$
$$y = v \cdot h$$

2.
$$i = \lfloor x - 0.5 \rfloor$$
$$j = \lfloor y - 0.5 \rfloor$$

This is done because the pure color is considered at the **center** of each texel

3.
$$s = x - i - 0.5$$
$$t = y - j - 0.5$$

$$\boxed{p' = (1-t)[(1-s)p_{i,j} + s \cdot p_{i+1,j}] + t[(1-s)p_{i,j+1} + s \cdot p_{i+1,j+1}]}$$

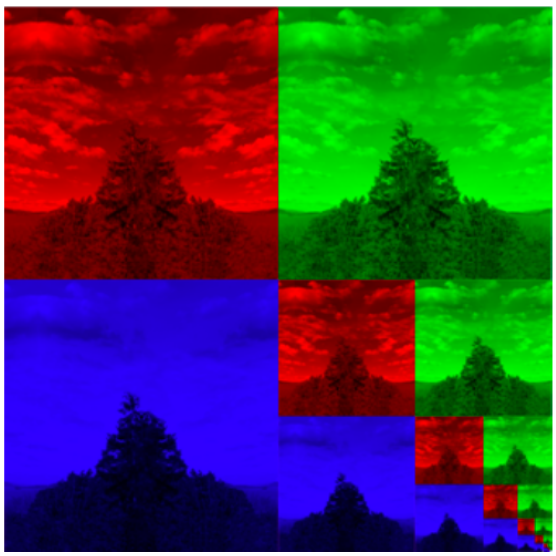
The techniques is much **smoother** but requires **4 texture access** and **3 interpolations** per pixel. In the 1D case it requires 2 texels , 1 interpolation and in the 3D case it requires 8 texels and 7 interpolations.

## 14.2 Minification Filtering

Also minification can use **nearest pixel** and **linear interpolation** but with poor results as they both try to simply guess an intermediate value rather than averaging the texels that fall inside a pixel. A good alternative is the **MIP-Mapping** technique.

### 14.2.1 MultiploInParving-Mapping

This techniques precomputes a set of scaled versions of the texture each one **halved in size** by averaging the values of the pixels and fetches the pixel from the one that is **closest** to the on-screen pixel size. A MIP-Map requires 33% extra space.



The pixel of the chosen image can be then **interpolated** or chosen with **nearest pixel** approach.

## 14.3 UV Intervals

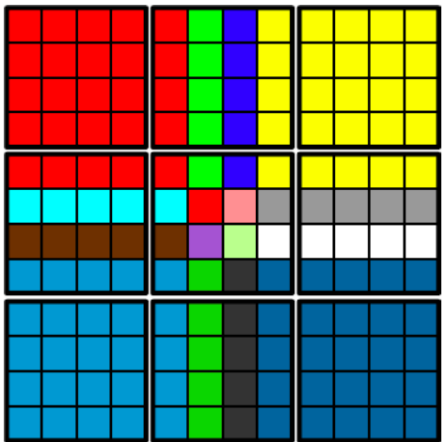
Sometimes the u,v values can fall **outside** the $[0, 1]$ range. Several behaviours and several alternatives are possible,with the main 4 being :

- **Clamp**
- **Repeat**
- **Mirror**
- **Constant**

These techniques are applied at the **texel level** , so first the i,j indexes of the texels have to be computed ( already seen )

### 14.3.1 Clamp

Extends the colors of the border to the texels that are outside the $[0, 1]$ range.



Focusing on coordinate u , i is the index of the texel required by nearest pixel or linear interpolation. The value of the index of the texel fetched from the texture can be determined as :

$$i_{Act} = \begin{cases} 0 & i < 0 \\ i & 0 \leq i < w \\ w - 1 & i \geq w \end{cases}$$

Same goes for v and index j :

$$j_{Act} = \begin{cases} 0 & j < 0 \\ i & 0 \leq j < h \\ h-1 & j \geq h \end{cases}$$

### 14.3.2 Repeat

The repeat strategy repeats the pattern indefinitely many times both in the horizontal and vertical direction. It can be computed as :

$$i_{Act} = mod(i, w)$$

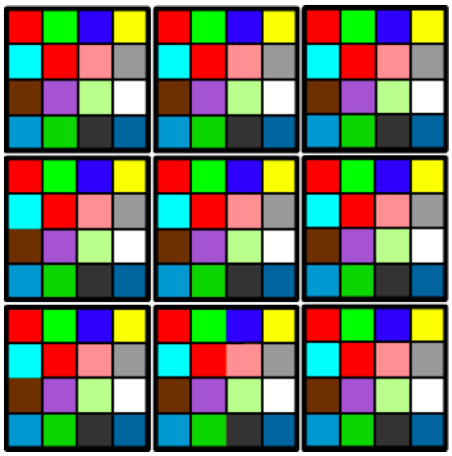
$$j_{Act} = mod(j, h)$$

where

$$mod(i, w)$$

is a **non-negative integer less than w** such that exists an integer k :

$$k \in Z \quad , 0 \leq mod(i, w) < w, \quad k \cdot w + mod(i, w) = i$$



### 14.3.3 Mirror

Similar to repeat but at each replication it flips the texture.

$$i_{Tmp} = mod(i, 2w) \quad \{0 \leq i_{Tmp} \leq 2w\}$$

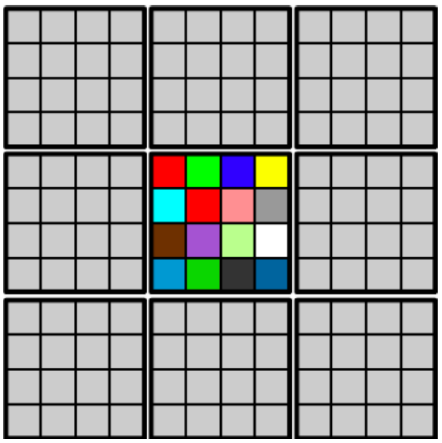
$$i_{Act} = \begin{cases} i_{Tmp} & i_{Tmp} < w \\ 2w - i_{Tmp} - 1 & i_{Tmp} \geq w \end{cases}$$

The second case is when the texel should be flipped.

### 14.3.4 Constant

The constant behaviour replaces the sample of the texture that fall outside the $[0, 1]$ range with a default color $c_d$ :

$$c = \begin{cases} c_d & i < 0 \\ c[i] & 0 \leq i < w \\ c_d & i \geq w \end{cases}$$



This is the **only technique** that return the texel **color** instead of the position.

# 15 Animations

Two important concepts in animations are **Quaternions** and **Bezier Curves**

## 15.1 Quaternions

When dealing with movements considering Euler angle rotations the **gimbal lock** represented an important problem : it causes to loose one degree of freedom due to the alignment of two rotating axis in the same direction. This phenomenon causes **unrealistic movements**. **Quaternions** are a solution to this problem : it encodes a rotation in space with **4 numbers** linearly dependent ( instead of 3). Quaternions are **extensions** of complex numbers that have **three imaginary components**:

- Complex number $\rightarrow a + ib$
- Quaternions $\rightarrow a + ib + jc + kd$ where the three imaginary components are called **vector part** subject to the following relation

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

$$i \cdot j = k$$

$$...$$

A complete algebra can be defined with Quaternions :

- Sum of two quaternions

$$(a_1+ib_1+jc_1+kd_1)+(a_2+ib_2+jc_2+kd_2) = (a_1+a_2)+i(b_1+b_2)+j(c_1+c_2)+k(d_1+d_2)$$

- Product with scalar

$$\alpha(a + ib + jc + kd) = \alpha a + i \cdot \alpha b + j \cdot \alpha c + k \cdot \alpha d$$

- Product of two quaternions

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

- Norm
$$||a + ib + jc + kd|| = \sqrt{a^2 + b^2 + c^2 + d^2}$$
- Angle
$$\theta = arcos\frac{a}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$
- Rising to power $\alpha$
$$(a+ib+jc+kd)^\alpha = ||a+ib+jc+kd||^\alpha \left(cos(\alpha\theta) + \frac{ib+jc+kd}{\sqrt{b^2+c^2+d^2}} sin(\alpha\theta)\right)$$
- Unitary quaternions
$$||a + ib + jc + kd|| = \sqrt{a^2 + b^2 + c^2 + d^2} = 1$$

Quaternions can encode rotations in 3D of an angle $\theta$ along an axis oriented along a **unitary** vector $v = (x, y, z)$ :

$$\boxed{q = cos\frac{\theta}{2} + sin\frac{\theta}{2}(ix + jy + kz)}$$

Since v is unitary also **q is unitary**. For example a rotation along the x-axis only ( v= (1,0,0)):

$$q = cos\frac{\theta}{2} + isin\frac{\theta}{2}$$

If two unitary quaternions $q_1 q_2$ encode two different rotations then their product encodes the composed form :

$$M_1 \Leftrightarrow q_1 \quad M_2 \Leftrightarrow q_2 \rightarrow M_1 \cdot M_2 \Leftrightarrow q_1 \cdot q_2$$

This way Euler angles can be transformed into a quaternion:

$$R = R_y(\Phi) \cdot R_x(\theta) \cdot R_z(\phi)$$

$$\boxed{q = \left(cos\frac{\psi}{2} + jsin\frac{\phi}{2}\right)\left(cos\frac{\theta}{2} + isin\frac{\theta}{2}\right)\left(cos\frac{\phi}{2} + ksin\frac{\phi}{2}\right)}$$

where

$$q = \left(cos\frac{\psi}{2}cos\frac{\theta}{2}cos\frac{\phi}{2} - sin\frac{\psi}{2}sin\frac{\theta}{2}sin\frac{\phi}{2}\right)$$
$$+i\left(cos\frac{\psi}{2}sin\frac{\theta}{2}cos\frac{\phi}{2} - sin\frac{\psi}{2}cos\frac{\theta}{2}sin\frac{\phi}{2}\right)$$
$$+j\left(sin\frac{\psi}{2}cos\frac{\theta}{2}cos\frac{\phi}{2} - cos\frac{\psi}{2}sin\frac{\theta}{2}sin\frac{\phi}{2}\right)$$
$$+k\left(cos\frac{\psi}{2}sin\frac{\theta}{2}cos\frac{\phi}{2} - sin\frac{\psi}{2}sin\frac{\theta}{2}cos\frac{\phi}{2}\right)$$

The opposite operation where $q = a + ib + jc + kd$ can be easily converted into a **rotation matrix** :

$$R = \begin{bmatrix} 1-2c^2-2d^2 & 2bc+2ad & 2bd-2ac & 0 \\ 2bc-2ad & 1-2b^2-2d^2 & 2cd+2ab & 0 \\ 2bd+2ac & 2cd-2ab & 1-2b^2-2c^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## 15.2 Bezier Curves

Bezier curves are **non-linear interpolation technique** that uses several intermediate points to control the shape of the curve.Their are characterized by their degree : **linear** ,**quadratic** , **cubic**...
Bezier curves of degree N are defined by **N+1 values** and are computed in a **recursive** way. Consider a linear interpolation ( function similar to the one see in the first chapters ) between two values a and b at intermediate point $\alpha \in [0, 1]$

$$lerp(a, b, \alpha) = (1 - \alpha)a + \alpha b = I(0, \alpha, 1, a, b)$$

A Bezier curved of degree N defined by points $x_0, ..., x_N$ passes in $x_0$ at $\alpha = 0$ and in $x_N$ at $\alpha = 1$ . The function $Bezier_n(x_0, ..., x_N, \alpha)$ returns the value of the Bezier curve of degree n at intermediate point $\alpha$ :

$$\boxed{Bezier_1(x_0, x_1, \alpha) = lerp(x_0, x_1, \alpha)}$$

$$\boxed{Bezier_N(x_0, ..., x_N, \alpha) = lerp(Bezier_{N-1}(x_0, ..., x_{N-1}, \alpha), Bezier_{N-1}(x_1, ..., x_N, \alpha), \alpha)}$$

So for N points the Bezier functions split the set in two :

- from $x_0 \to x_{N-1}$
- from $x_1 \to x_N$

Then the `lerp` function with those two as first two arguments is applied. For example in a cubic curve :

$$Q_0 = x_{01} = lerp(x_0, x_1, \alpha)$$

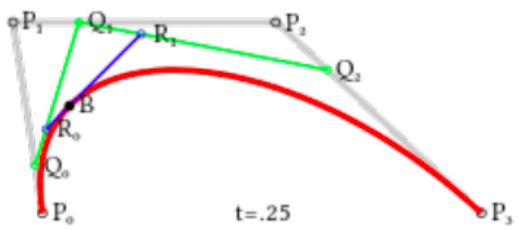
$$Q_1 = x_{12} = lerp(x_1, x_2, \alpha)$$

$$Q_3 = x_{23} = lerp(x_2, x_3, \alpha)$$

$$R_0 = x_{012} = lerp(x_{01}, x_{12}, \alpha)$$

$$R_1 = x_{123} = lerp(x_{12}, x_{23}, \alpha)$$

$$B = Bezier(x_0, x_1, x_2, x_3, \alpha) = lerp(x_{012}, x_{123}, \alpha)$$




The procedure is done with a constant interpolating value $\alpha$ repeated several times. The control points can be factorized in polynomial form :

$$Bezier_n(x_0, ..., x_N, \alpha) = \sum_{i=0}^{n} \binom{n}{i} (1-\alpha)^{n-1} \alpha^i x_i$$

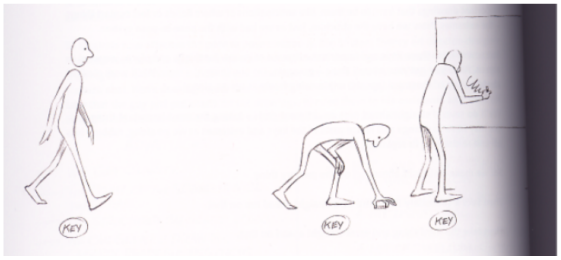
$$\boxed{Bezier_2(x_0, x_1, x_2, \alpha) = (1-\alpha)^2 x_0 + 2(1-\alpha)\alpha x_1 + \alpha^2 x_2}$$

$$\boxed{Bezier_2(x_0, x_1, x_2, x_3, \alpha) = (1-\alpha)^3 x_0 + 3(1-\alpha)^2 \alpha x_1 + 3(1-\alpha)^2 \alpha x_2 + \alpha^3 x_3}$$

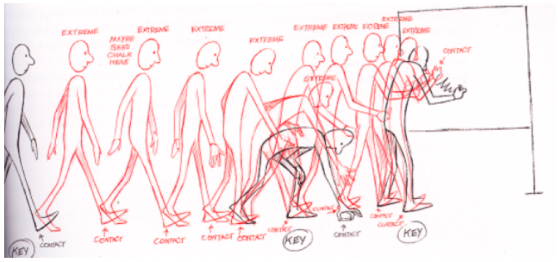
## 15.3 3D Animation

The aim of 3D animation is to reproduce the motion of objects. The scene is divided into a set of images (**frames**) recorded at a given speed (**frames per second**). To guarantee **persistence of vision** FPS are at least **24** so that the user perceives the animation as a continuous flow. While the synchronization between recorded images and the reproduction is **natural in movies** ,it is **not** in computer graphics : this is because the speed at which the display shows the images is usually faster than the one at which the animation is created. To overcome this issue all frames are **interpolated** to create a continuous evolution of the objects of the scene.
3D animation uses the **key-frame technique**:

1. First the important scenes are drawn

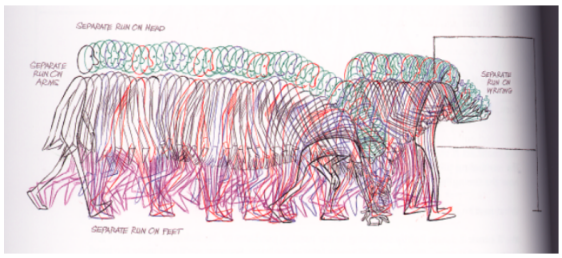



2. Then the **extreme points** are drawn (where short-time equilibrium are reached)




3. **Breakdown** poses that defined speed and acceleration

4. Finally the frames between the breakdown poses (**in-between poses**) are drawn



In computer animation the **animator** takes care of **steps 1 - 3** while the last step 4 (in-between poses) are generated through **interpolation**. This allows to:

- Use **less memory** as the poses are stored only every few frames
- Be **speed independent** as interpolating creates a continuous flow which can be sampled at un-even intervals according to the playback speed and regardless of the frame rate at which the animation was defined.
- **reduce animation authoring time** as artists do not have to manually create all frames.

Storing all the **parameters** for animation requires a lot of memory space , which often cannot be afforded. Moreover many parameters **don't change** during the animation (like time scaling or colors which remain constant). Only the values that change over time are stored : these parameters are called **animation channels** (for example the x position of a car is an animation channel). Frame in which such parameters are defined are called **key-frames**. Values of parameters in the key-frames **cannot be interpolated linearly** : the human eye is too **sensitive** to speed changes and linear interpolation would create such **speed discontinuities**

which would make the animation appear blocky. This is why 3D animation uses **Bezier curves of third degree** : this allows to define for each key frame the **value of the parameters** and its **first derivative** ( allows to have continuity in **position** and **velocity**!). The role of the animator is to work on the intermediate points.
Rotations in animations are done using **quaternions** which are not subject gimbal lock and thus allow smooth movements. Usually the following steps are adopted in computer graphics :

1. Initial rotation of the object (key-frames or application generated data), are encoded using **Euler angles**.
2. Whenever they have to be **animated** Euler angles are transformed into **quaternions**
3. **Interpolation** or updates to the angles are performed on the angles
4. **Rotation matrices** are computed from the quaternion ,encoding the final orientations of the objects
5. If Euler angles need to be computed from intermediate frames they are extracted from the quaternions

**Quaternion interpolation** has better results than **Euler interpolation** but requires some special kinds of interpolation:

- `nLerp` **Normalized Linear Interpolation**
- `sLerp` **Spherical Linear Interpolation**

To obtain optimal results these techniques are combined with **Bezier curves**.

# 16 WegGL & Shaders

## 16.1 Intro

The **OpenGL API** is a software interface for graphics hardware. A program can use OpenGL API for calling some prepackaged functionality to:

- **Define models**
- **Use Textures**
- **Perform operations (e.g. projections**

in order to **render images** (usually 3D).
The OpenGL specification is implemented as the **OpenGL library** by hardware vendors ( NVIDIA,AMD...) making OpenGL **portable**. It can also be implemented**without GPU**.
OpenGL is implemented as **client-server** system :

- **Client side**: the application code in the main CPU memory (issues commands)
- **Sever side** :hardware and memory on the graphics card (executes the commands to produce the final image)

To keep track of all the **state-variables** OpenGL uses a **state model** . This avoids **passing to many arguments** in function calls and allows to **keep the state value** until it is changed by some other function.

## 16.2 GLSL

Up to 2004 as graphics adapters evolved OpenGL was improved by adding **new functionalities**. However the many new technological improvements made issuing **fixed procedures** very expensive so the **OpenGL Shading Language (GLSL)** , a C-like language , was introduced to specify **rendering operations** only.
With GLSL new **derivative APIs** were introduced :

- **OpenGL ES** for **embedded systems** with limited system resources.

- **WebGL** , an OpenGL-style rendering within **web-browsers** using **Java Script** to access the **OpenGL ES 2.0** functionalities and the **HTML Canvas elements** for drawing.

## 16.3 WebGL - Client

### 16.3.1 Canvas

WebGL takes advantage of the **Canvas element** of HTML5:

- allows **scriptable rendering** of graphics within the browser
- provides the default **FrameBuffer**, a region of **physical memory** in the GPU used to temporarily store an image for rendering.
- supports different API (like WebGL...)
- has **width,height,ID** attributes
- the area within the canvas can be modified with **JavaScript**

```
<canvas id="my-canvas" width="600" height="400">
        Write here something to show if browser does not support the HTML5
        canvas element.
</canvas>
```

### 16.3.2 Context

The first element a program must create to draw graphics is the **Context**. It provides **internal data structure** for keeping track of the state settings and operations. Examples of state-values are *current path, current fill and stroke, line width and pattern, alpha transparency, antialiasing ,blend mode , shadows,transformation matrix, text attributes.*
It can be requested using `.getContext(contextID, *args...)` JavaScript function on the **canvas element**. The `contextID` argument can have values :

- `2D`
- `webgl/experimental-webgl`

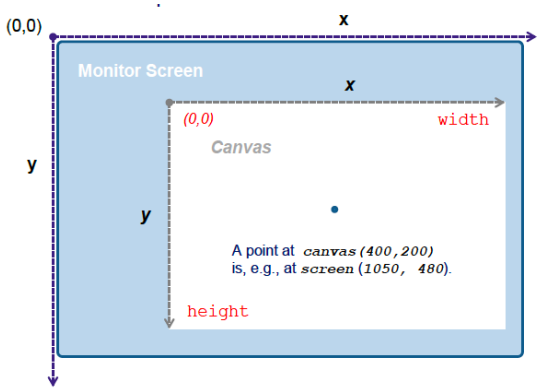
```
<script>
        var canvas = document.getElementById('my-canvas');
        var context = canvas.getContext('experimental-webgl');
</script>
```

### 16.3.3 Drawing with 2D Context

Once the **frameBuffer** (provided by the `Canvas` element) and the **context** are provided , drawing can begin. To start drawing the data for **constructing the shapes** must be specified through **geometric primitives**. **HTML 2d context** makes available :

- **Rectangles**
  - `context.rect(x,y,width,height` → creates an **invisible** rectangle object to be further defined by commands such as `.stroke()` or `.fill()`
  - `context.fillRect(x,y,width,height)` → create a filled rectangle with no border line
  - `context.stroke(x,y,width,height)` → creates an empty rectangle with border line
- **Paths (lines or circles)**
  - `context.beginPath()` → begins a path or resets the current one
  - `context.moveTo(x,y)` → moves the path to the specified point on the canvas, **without creating a line**
  - `context.lineTo(x,y)` → adds a new point and **creates a line** from that point to the last specified one
  - `context.closePath()` → creates a path from the current point back to the starting point
  - `context.arc(x,y,r,sAngle,eAngle,counterclockwise)` → creates an arc/curve
  - `context.arcTo(x1,y1,x2,y2,r)` → creates an arc/curve between two tangents.

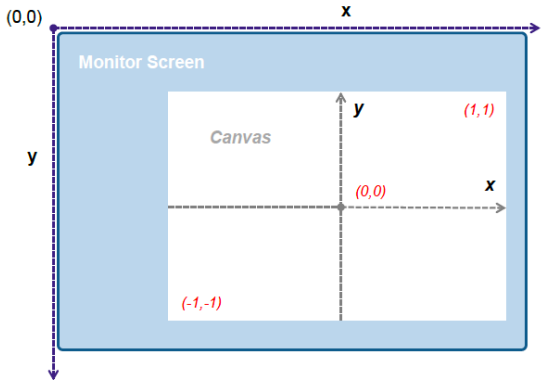
The 2D context has a **coordinate system** similar to the one of the screen, with values also expressed in pixels. The coordinates in the canvas are transparently mapped on-screen by the context.



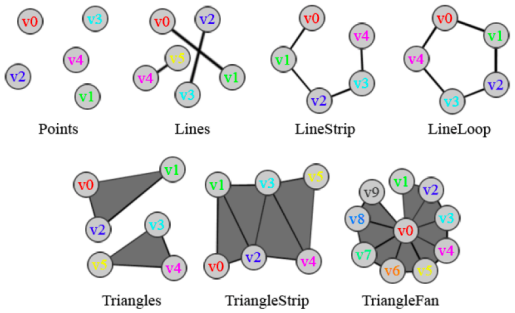

### 16.3.4 Drawing with webgl Context

With respect to the **2D context** , the **experimental-webgl/webgl** context provides:

- A different **coordinate system**
  Coordinates in webgl context start in the **middle of the canvas** with values expressed in **floating point numbers**. Its values for shapes can be **outside** the $[-1, 1]$ range provided that some **computations** are made.




- A different set of **geometric primitives**
  A webgl primitive is simply a **collection of vertices** hooked together in a predefined way. All primitives are 1d,2d or 3d ranging from simple lines to groups of triangles : all WebGL can do is visualize **points,lines and triangles** and apply **color and texture** to them.




- A different set of **modes** for drawing
  Unlike the 2d canvas context , the webgl context doesn't allow to directly set the color or location of a vertex into a scene. Furthermore shapes **cannot** be constructed using a list of fixed polygonal drawing functions (too many!). All data associated with vertices need to be streamed from JavaScript API to GPU and **only then** it will be possible to call the **draw** command. To pass vertex data, a **vertex buffer object VBO** ,which holds attributes such as **position,normals,texture,coordinates...** must be build :

  1. Define an array holding the vertexes of the shape

```
var data = [ -0.5, -0.5,
             0.5, -0.5,
             0.0, 0.5 ];
```

  2. Ask WebGL to reserve a **buffer**. Note that `gl` here corresponds to the variable returned by `canvas.getContext`

```
var VBO = gl.createBuffer();
```

  3. Set VBO buffer as active and bind to data type

```
gl.bindBuffer(gl.ARRAY_BUFFER, VBO);
```

  4. Place vertex data inside buffer. The `data` parameter should be casted to contain floats (required by JavaScript!) like : `new Float32Array(data)`

```
gl.bufferData(gl.ARRAY_BUFFER, data, gl.STATIC_DRAW);
```

5. In webgl rendering operations must use **GLSL** , which creates a program **server-side** that defines who the GPU will handle data send to it.
6. Retrieve the handle to the location of GPU where the GLSL program expects to find the input data. The `"VertexPosition"` is defined in the GLSL program

```
var vertexPositionHandle = gl.getAttribLocation(glProgram, "VertexPosition");
```

7. **Data are sent**. It activates communication that from client reach the servev input location specified by `vertexPositionHandle`

```
gl.enableVertexAttribArray(vertexPositionHandle);
```

8. Let the GLSL program know how to interpret the data.
   `GLuint index` = handle to input location
   `GLint size` = how many values define the vertex
   `GLenum type` = the type of the data `GLboolean normalized` = if data requires normalization

```
gl.vertexAttribPointer(vertexPositionHandle, 2, gl.FLOAT, false, 0, 0);
```

9. Draw by calling GLSL program
   First parameter is specifies the **primitive** to be used
   Second parameter is index of element to start drawing from
   Third parameter is number of vertexes

```
gl.drawArrays(gl.LINE_STRIP, 0, 3);
```

Another important feature is **Aspect Ratio** : when setting up the canvas width and height might be different ( remember that pixels are usually **squared**). In this case the $[-1, 1]$ coordinates are mapped **non-proportionally** to screen

coordinates : this results for example in a circle looking like an ellipse.

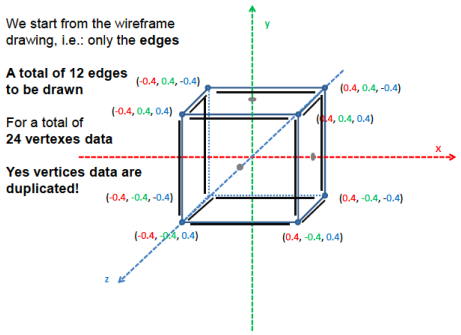
$$\text{Aspect Ratio a} = \frac{\text{canvas.width}}{\text{canvas.height}}$$

So if the **y-coordinate** of a point is multiplied by the aspect ratio **or** the **x-coordinate** divided by it, the shape will result as intended.

### 16.3.5 Drawing with webgl context : 3D

When drawing a cube in WebGL different approaches can be used:

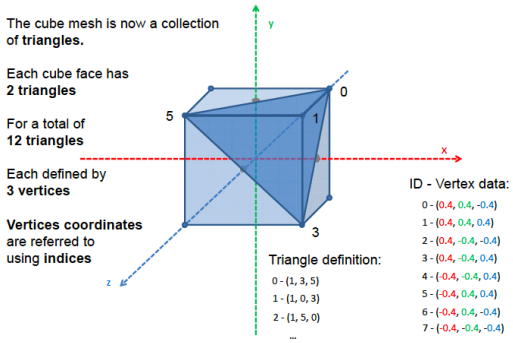
- **WireFrame Cube**
  In this case only the edges are visible. Since the cube is composed of **8 vertices and 12 edges** the total number of vertices stores is **24** ( vertices are duplicated)




  The same procedure applies as for the 2D case applies. A difference is that since we're in 3D the `gl.vertexAttribPointer()` has as second parameter **3** instead of **2**.
  The resulting figure will not appear as a cube since no **perspective** and no **view camera** have been implemented : these two can be stored in a **projection matrix** that is fed to a GLSL program (with some modifications to take the matrix into account). For the matrix you must get :

  - `matrixPositionHandle = gl.getUniformLocation(program,'pMatrix')`
  - `gl.uniformMatrix4fv(matrixPositionHandle,gl.FALSE, transposeMatrix(projectionMatrix)`

- **Mesh cube**
  In this case the cube is composed of a set of triangles. Each cube face has **2 triangles** for a total of **12 triangles**, each defined by 3 vertexes. So first define the **24 vertexes** than define a **index array** composed of 12x3 elements ( each row has 3 values indicating the vertex index making up the triangle)

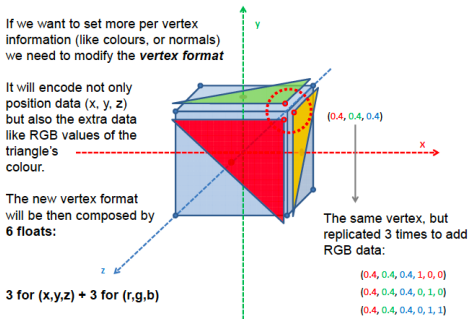



  The index array needs its **own buffer** the **IBO** buffer:

  - `IBO = gl.createBuffer()`
  - `gl.bindBuffer(gl.ELEMENT.ARRAY.BUFFER,IBO)`
  - `gl.bufferData(gl.ELEMENT.ARRAY.BUFFER,new Uint16Array(indexes), gl.STATIC.DRAW)`

  The draw function needs to be update : no longer `gl.drawArrays(gl.LINES,0,24` but `gl.drawElements(gl.TRIANGLES, indices.length,gl.UNSIGNED.SHORT.INT,0 (starting point))`

- **Mesh cube with colors**
  In this case each vertex is composed of three positional argument + **3 RGB values**.

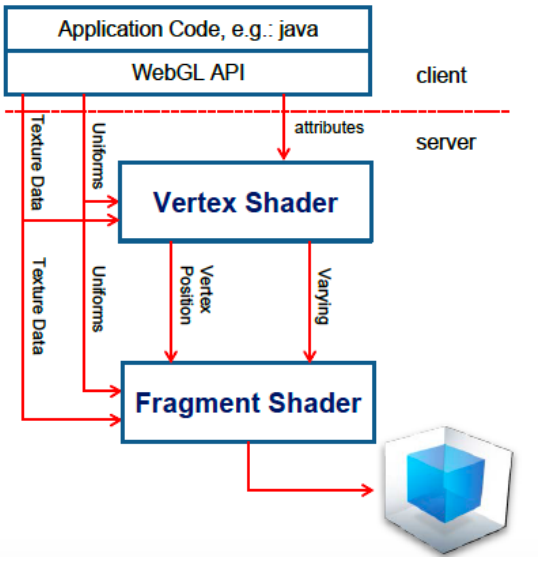



An additional `vertexColorHandler` must be defined:

- `vertexColorHandler = gl.getAttribLocation(program,'col1')`
- `gl.enableVertexAttribArray(vertexColorHandler)`
- `gl.vertexAttribPointer(vertexPositionHandler,3,gl.FLOAT,false,4*6 (4 bytes 6 values),4*3 (4 bytes staring skipping the first three positional args)`

## 16.4 GLSL - Server

GLSL creates a program that defines how the GPU will handle the data sent to it. The rendering process follows a **Rendering Pipeline**:



**Shaders** are **individual programs** written in **GLSL** that execute on graphics hardware to process vertices and perform rasterization tasks. The WebGL pipeline deals with two shaders :

- **Vertex Shader**
- **Fragment Shader**

### 16.4.1 Vertex Shader

The VS contains source code of operations that are meant to be performed on each vertex that is processed :

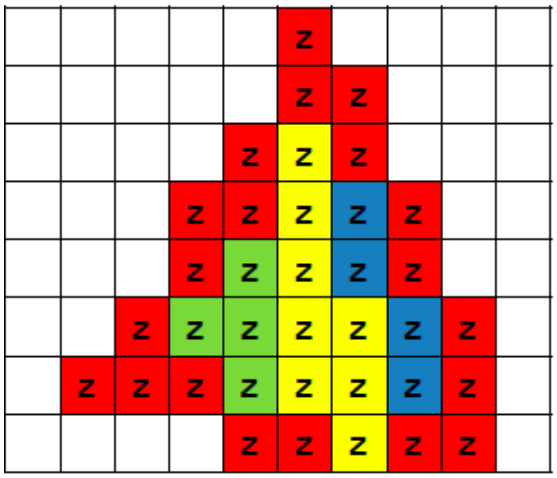
- **World-View-Projection** computation
- **Vertex color** computation
- **Light color** when using the **Goureaud** method
- ...

The VS is executed for **each vertex** so three times to render a triangle. The outputs of this step are:

- Vertices are assembled according to the **mode** argument of the drawing command
- Clipping and Screen coordinates are computed
- primitive is converted to a 2D image (**rasterization**). Each point of the image contains information as **color** and **depth**

### 16.4.2 Fragment Shader

The FS contains source code of operations that are meant to be performed on each **fragment** that results from **vertex shader rasterization**. A **fragment** is one of the 2D image grid squares along with its parameters of depth and other data :




Its operations are :

- **Anti-Aliasing** (can be done automatically)
- **Depth-test** (can be done automatically)
- **Color Blending** (can be done automatically)
- **Dithering** (can be done automatically)
- **Texturing** (must be done manually)
- **Color and light computation in Phong mode** (must be done manually)

At the end of the pipeline the image is ready to be displayed.

### 16.4.3 Data Types

Data must be passed through the different steps in the pipeline. There are different types:

- **Attributes**
  These are values that change per vertex ( for example x,y,z positions)
  - (x,y,z) → 12 bytes
  - (x,y,z,r,g,b) → 24 bytes
  - (x,y,z,nx,ny,nz) → 24 bytes

– (x,y,z,nx,ny,nz,u,v) → 32 bytes

- **Uniforms**
  These are per-program variables that are **constant** during execution. Examples are **Transformations matrices**. Uniforms can be passed to **both** shaders.

- **Texture data (Samplers)**
  These are special uniform variables used for **texturing**.They are used to identify the texture object used for each **texture lookup**.

- **Varying**
  Varying variables hold the result of VS execution that are used later in the pipeline (like partial results). These values are expected to be **interpolated** : this is because VS is executed only for each vertex but the FS is executed many more times (so must work on interpolated points)

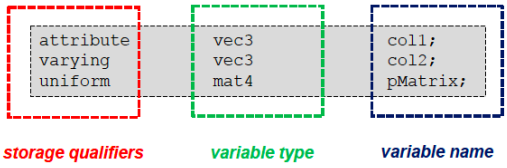
### 16.4.4 GLSL Program Language

A GLSL shader program syntax is very C-like:

```
var vs = `
                attribute vec3 pos1;
                attribute vec3 col1;
                varying vec3 col2;
                uniform mat4 pMatrix;

                void main() {
                        col2 = col1;
                        gl_Position = pMatrix * vec4(pos1, 1.0);
                }
        `;
```

The variable ,unlike C , are defined using **storage qualifiers,variable type and variable name**




Accepted **storage qualifiers** are:

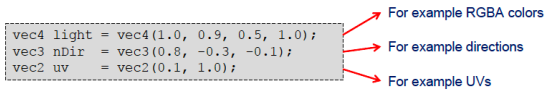
- `const` → compile-time constant or read-only function parameter

- `attribute` → linkage between vertex shader and OpenGL ES for per-vertex data
- `uniform` → value does not change across the primitive being processed. They form the linkage between a shader,OpenGL ES and the application
- `varying` → linkage between VS and FS for interpolated data

Accepted basic **variable types** are:

- `void` → no function return value or empty parameter list
- `bool`
- `int`
- `float`
- `vec2,vect3,vect4` → n-component floating point vector
- `bvec2,bvec3,bvec4` → boolean vector
- `ivec2,ivec3,ivec4` → signed integer vector
- `mat2,mat3,mat4` → 2x2,3x3,4x4 **float** matrix
- `sampler2D` → access a 2D structure
- `samplerCube` → access cubed mapped structure

**Vectors** are widely used. Here some characteristics:




```
vec3 n_Light_Dir;
float light_Color;
…
return vec4(n_Light_Dir, light_Color);
```

```
vec4 light = vec4(1.0, 0.9, 0.5, 1.0);
```

```
light.x =       light.r =       light.s =       1.0
light.y =       light.g =       light.t =       0.9
light.z =       light.b =       light.p =       0.5
light.w =       light.a =       light.q =       1.0
```

```
vec3 l1 = light.xyz;
vec2 l2 = light.rb;
```

GLSL provides also **functions** , which are similar to C:

```
vec4 light_model(int light_type, vec3 position){
    vec3 n_Light_Dir;
    float light_Color;

        [...]

    return vec4(n_Light_Dir, light_Color);
}
```

The OpenGL provides **precision qualifiers** for `int`, `float` and `sampler` variables to **increase efficiency** and **decrease memory requirements** for shaders.

| Precision Qualifiers | Descriptions | Default Range and precision | |
|---|---|---|---|
| | | Float | int |
| highp | High precision. The minimum precision required for a vertex shader (default there) | $(-2^{62}, 2^{62})$ | $(-2^{16}, 2^{16})$ |
| mediump | Medium precision. Minimum precision in FS | $(-2^{14}, 2^{14})$ | $(-2^{10}, 2^{10})$ |
| lowp | Low precision. Still able to represent all colors. | (-2, 2) | $(-2^{8}, 2^{8})$ |

These precision qualifiers can be added:

- **before** the type specification `mediump vec3 col1`
- **state for ALL variables** `precision mediump float`

There are also **Special Variables (global)** used to communicate with fixed function parts of the pipeline. They **do not** need to be declared but are **essential** :

- `gl-Position= pMatrix * vec4(pos1,1.0)` → for the VS which holds the **final** transformed vertex position in **screen-coordinates**
- `gl-FragColor=vec4(col2,1)` → for the FS which holds the **final** fragment color in **RGBA**

### 16.4.5 GLSL Compiling

A shader must be **compiled** and **linked** before being used. This is done by calling the proper functions **client-side**:

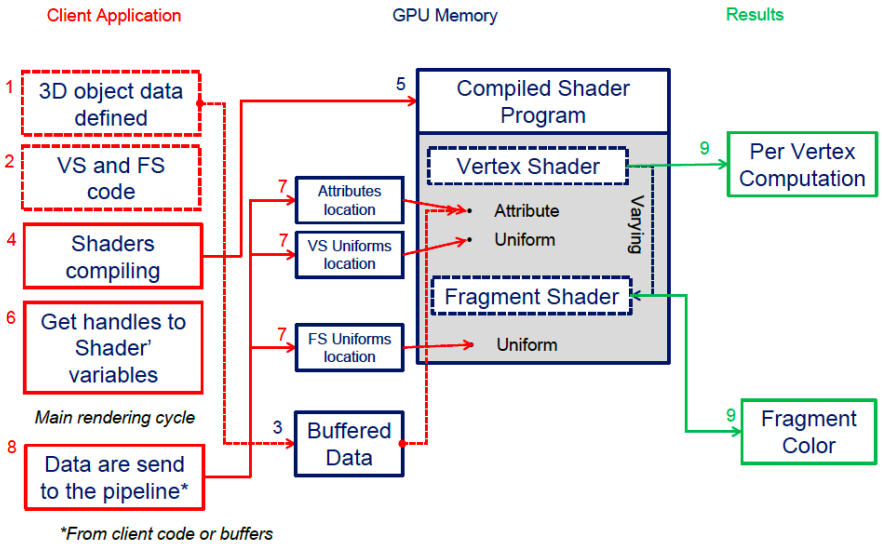
1. A shader object must be created for both **VS** and **FS** using the function `Object createShader(enum Type)` with the **handler** stored in a variable. The allowed `Types` are **Vertex-Shader** and **Fragment-Shader**
2. Shader must be loaded with a source string using `void shaderSource(Object shader, string source)`
3. Compile the shader by calling `void compileShader(Object shader)` (here a exception can occur!)
4. Create shader program object that contains the whole pipeline with `Object createProgram()` that returns a **pointer**.
5. Attach compiled shaders to the program with `void attachShader(Object program , Object shader)`
6. Link program using `void linkProgram (Object program)`

```
var v1 = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(v1, vs);
gl.compileShader(v1);

var v2 = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(v2, fs);
gl.compileShader(v2);

program = gl.createProgram();
gl.attachShader(program, v1);
gl.attachShader(program, v2);
gl.linkProgram(program);
```

### 16.4.6 Workflow Recap

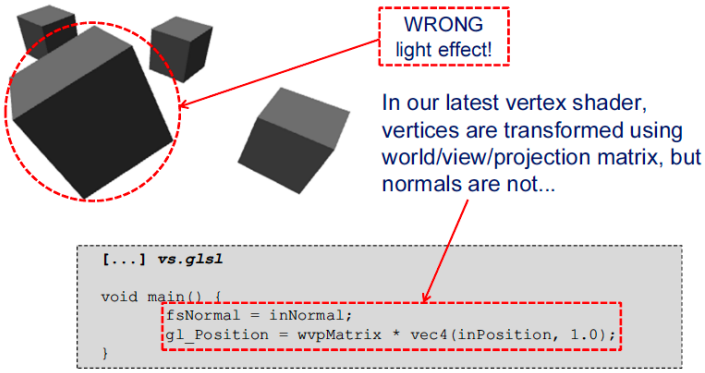



## 16.5 Shader spaces

The computation of light involves information about :

- normals
- light position/direction
- observer direction/position

The surface is defined by the direction of normals , if they are not computed in the same **space** as the **vertexes** the final light result is not going to be the one expected. When applying the **WVP Matrix** to the vertexes also normals should change otherwise this could happen :




In order to achieve the intended light results, **data** about normals,lights and eye position must be expressed using the same coordinates system or **space**. There are 4 main space systems:

- **World space**
- **Camera space**
- **Object space**
- **Static space**

### 16.5.1 World space

When the illumination model is computed in **World Space** vertices positions and normals must be transformed according to the World Transforms. Is usually requires **three** matrices:

- **WVP** projection ( as usual)
- a matrix to transform **vertex positions**
- a matrix to transform **normals** into worlds space

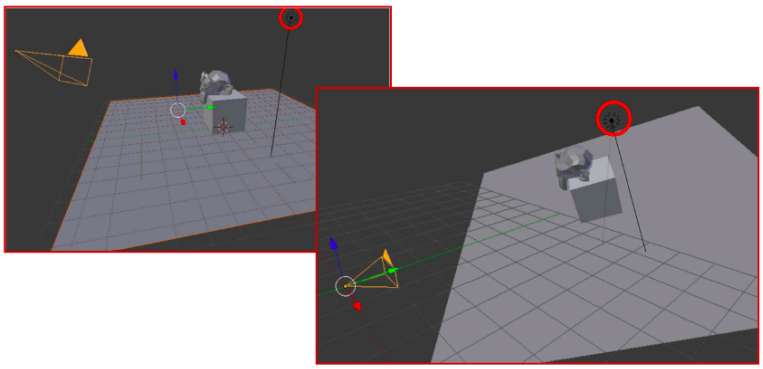
| Data | Are usually in | Must be transformed in World Space? |
|---|---|---|
| **Vertex Position** | **Local Space** | **Yes** |
| **Vertex Normals** | **Local Space** | **Yes*** |
| Light Direction | World Space | No |
| Light Position | World Space | No |
| Eye Position | World Space | No |

Camera position, lights positions and directions, can however be passed to the shaders without any additional transformation.

To transform the **normal** the **inverse** of the objects world matrix can be used if **no scalings** are performed. Otherwise the **3x3 submatrix of the inverse transpose** should be used. The world space light computation is **computationally very expensive** as many vertex multiplication are performed.

### 16.5.2 Camera space

If the illumination in is computed in the **camera space** then position and normals must be transformed according to the **world transforms** and the **view transforms**. So this time **lights** (directions and positions) must be transformed together with **objects and normals**.



This requires passing **three** matrices to the VS to project **vertices**,transform **positions** and transform **normals**. For the light:

- The **light position** is transformed applying the **view transform**

- The **light direction** is transformed applying the **inverse transpose** of the **3x3 submatrix** of the view transform (if **scaling** is used, otherwise just the 3x3 submatrix is fine).

| Data | Are usually in | Must be transformed in Camera Space? |
|---|---|---|
| **Vertex Position** | **Local Space** | **Yes** |
| **Vertex Normals** | **Local Space** | **Yes** |
| **Light Direction** | **World Space** | **Yes** |
| **Light Position** | **World Space** | **Yes** |
| Eye Position | World Space | No |

The **observer direction** ( computed for specular Phong and Blinn models) can be obtained by **normalizing** the coordinates of the considered point of the object in the 3D space ( because after view transform the center of projection is now in the origin of the axis)

### 16.5.3 Object space

In object space the objects are the center of the world. This is useful as the **normals** and **vertex positions** do not to be changed. But the **light position and direction** along with the **observer** need to be transformed into the object **local coordinates**.

- for **light directions** the **inverse transpose 3x3 submatrix** of the **inverse World matrix** should be used.
- for the **light positions** the **inverse of the World transform matrix** should be used.
- to obtain the **observer direction** both **world** and **view** matrices should be inverted and applied to the origin $((0, 0, 0, 1))$

| Data | Are usually in | Must be transformed in Object Space? |
|---|---|---|
| Vertex Position | Local Space | No |
| Vertex Normals | Local Space | No |
| **Light Direction** | **World Space** | **Yes** |
| **Light Position** | **World Space** | **Yes** |
| **Eye Position** | **World Space** | **Yes** |

### 16.5.4 Static space

In this space local and world coordinates are **identical** (world matrices are **identity** matrices).This also means that world and object spaces are coincident. No transformation must be performed. It is the most simple to use but does not allow to **move objects during application**. This can be useful in medical or scientific visualization of fixed background in VR.

### 16.5.5 Space recap

| | Operations to be performed in the client code | | | Matrices passed to the Vertex Shader | | |
|---|---|---|---|---|---|---|
| | Lights | | EYE Position | | | |
| | Position | Direction | | WVPM | PM | NM |
| **World** | - | - | - | yes | WM | **Mat3**(WM) or * |
| **Camera** | VM | mat3(VM) | no | yes | WVM | **Mat3**(WVM) |
| **Object** | $WM^{-1}$ | mat3($WM^{T}$) | $WM^{-1}$ | yes | no | no |
| **Static** | - | - | - | Only VPM | no | no |

| | |
|---|---|
| VM | View Matrix |
| WM | Object's world matrix |
| $WM^{-1}$ | Inverse of the World Matrix |
| $WM^{T}$ | Transpose of the World Matrix |
| WVPM | World View Perspective Matrix |

| | |
|---|---|
| - | No modification required |
| no | Not passed at all |
| **Mat3**(x) | Sub 3x3 matrix of the x 4x4 matrix |
| PM | Vertex Position Matrix |
| NM | Normal Matrix |

*In **World Space**, if no rotation/scale transformations are expected, it is possible to simply pass an object's unmodified world matrix as the NM. Otherwise the following 3x3 submatrix is expected:

```
mat3(inverse(transpose(WM)))
```

As a rule, all directions (lights and normals) are to be transformed. by the calling application usually require the 3x3 submatrix of the inverse of the transpose world matrix of an object, except if there are no scale modifiers (in order to optimize computation)

## 16.6 GLSL Textures

Textures are treated like any other data : declared in the client-code and handled sever-side. In the shaders **textures** can be handled using a **look-up function**

```
vec4 texture2D(sampler2D sampler, vec2 coord)
```

- `sampler2D sampler` is the identifier of texture object. It is passed from the **client code** to the **shader**.
- `coord` are the **UV** coordinates for the lookup
- `vect4` is the retuned object , a **color** in RGBa format.

Texels are extracted in the fragment shader using `sampler2D object` and `UV coordinates` by calling the `texture2D(texturefile,fsUVs)` which return a color (`vec4`)

```
// FS code
…
varying vec2 fsUVs;
uniform sampler2D textureFile;
…
void main() {
…
      vec4 diffuseTextureColorMixture = texture2D(textureFile, fsUVs);
…
}
```

The **UV coordinates** can be associated to a vertex by including them into vertex formtat

```
//*** Getting vertex and normals
var objVertex = [];
for (n = 0; n < loadedModel.meshes[i].vertices.length/3; n++){
      objVertex.push( loadedModel.meshes[i].vertices[n*3],
                      loadedModel.meshes[i].vertices[n*3+1],
                      loadedModel.meshes[i].vertices[n*3+2]);

      objVertex.push( loadedModel.meshes[i].normals[n*3],
                      loadedModel.meshes[i].normals[n*3+1],
                      loadedModel.meshes[i].normals[n*3+2]);

      if(UVFileNamePropertyIndex>=0){
              objVertex.push( loadedModel.meshes[i].texturecoords[0][n*2],
                              loadedModel.meshes[i].texturecoords[0][n*2+1]);
      } else {
              objVertex.push( 0.0, 0.0);
      }
}
```

(x,y,z, nx,ny,nz, u, v)

### 16.6.1 Using texture : steps

1. Allocate space on the GPU memory and its location stored in a variable

   ```
   var texture = gl.createTexture()
   ```

2. Specify which kind of texture will occupy that space and select it as the current one

   ```
   gl.bindeTexture(gl.TEXTURE-2D,texture
   ```

3. Load image to the texture object using HTML `Image()` to load an image file

   ```
   var image = new Image();
   ```

   This object has some properites :

   - `image.src = imageURL` set or return URL
   - `image.onLoad = function() {... }`
   - `image.webglTexture` that returns the **texture object data**

4. Define more parameters to specify **texture attributes** in the callback function defined by `.onLoad`

   ```
   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
   gl.generateMipmap(gl.TEXTURE_2D);
   ```

   - The `texImage2D` that loads the image data into the texture object.
   - The `pixelStorei` set parameters for the pixels stored in the texture.It is used to **correct the orientation** of the image with respect to the UV-axis.
   - The `texParameteri` sets textures parameters for current texture unit. Here you can set properties for the **magnification/minification** filter using `gl.TEXTURE-MAG-FILTER/gl.TEXTURE-MIN-FILTER` .
     For **magnification** valid options are `gl.LINEAR,gl.NEAREST`
     For **minification** valid options are `gl.LINEAR,gl.NEAREST,`
     `gl.LINEAR-MIPMAP-LINEAR,...`

- `generateMipmap` creates a set of textures for WebGLTexture object with image dimensions from the original size of the image down to a 1x1 image (used to create di MipMaps)

During rendering:

```
…
gl.bindTexture(gl.TEXTURE_2D, diffuseTextureObj[i].webglTexture);
gl.activeTexture(gl.TEXTURE0);
gl.uniform1i(textureFileHandle[currentShader], 0);
…
```

1. The texture must be selected again with `gl.BindTexture()`.The function uses the target specification of the texture and a reference to the texture object.

2. The binded texture is set as active unit and assigned to **level0**. Subsequent texture calls will affect only this unit.Depending on the specific implementation a different number of texture unit can be defined at one time and assigned to different levels (at least 8). More than 1 level is required if more than 1 texture is used on a face at once. Otherwise the same level can be used for different textures.

3. Active texture is loaded at the $0^{th}$ level and the level is passed to the shader as uniform where it is received using

```
uniform sampler2D textureFile
```