COMPUTER GRAPHICS

PART 1

by Yannick Giovanakis

May 17, 2018

Contents

1 Graphic Adapters

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

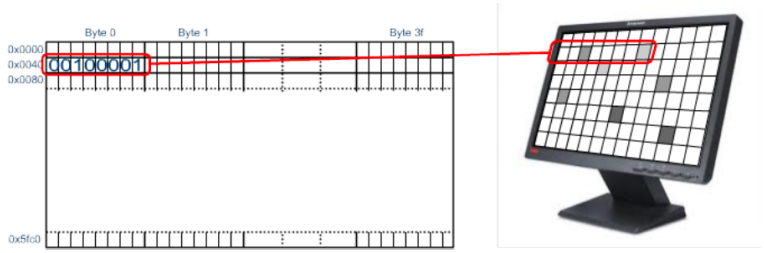
1.1 Vector Graphic Adapters

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

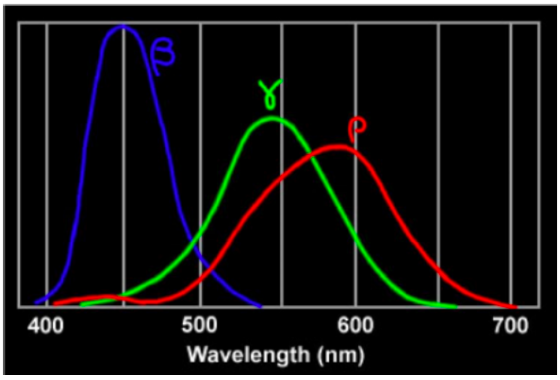
Used today , these adapters can perform complex tasks (multi - display screening, stereoscopic images ..)

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.

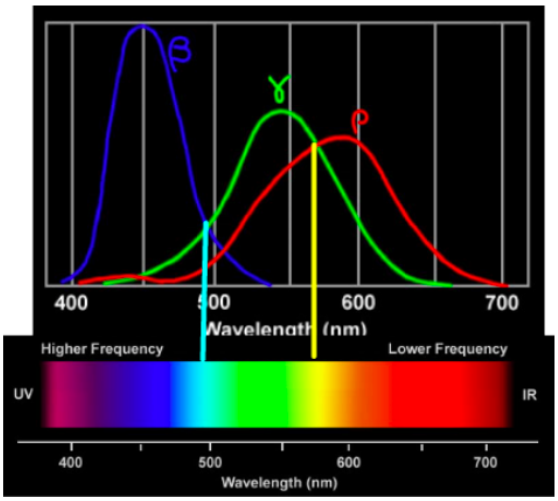
1.4.1 Human Vision

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 ρ, β, γ 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.



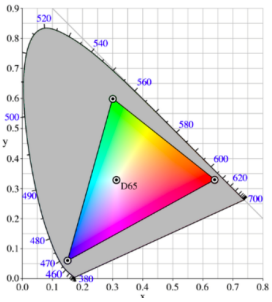
Color range

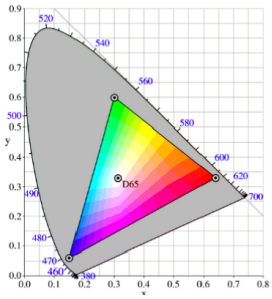
The **spectrum** that a monitor can produce is just a small portion of the entire spectrum the eye can see (grey area).
Color range of a monitor corresponds to a cube where the three colors are placed on the 3 axis.

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 ≤ 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

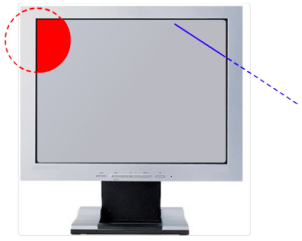


Figure 2: clipping

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} :

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

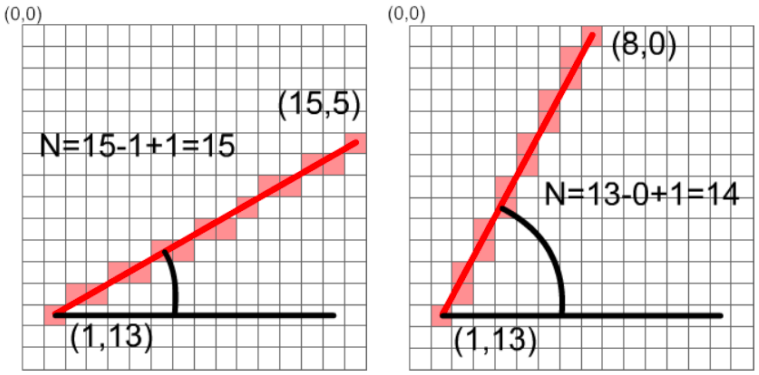
2.2 Line primitives

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)

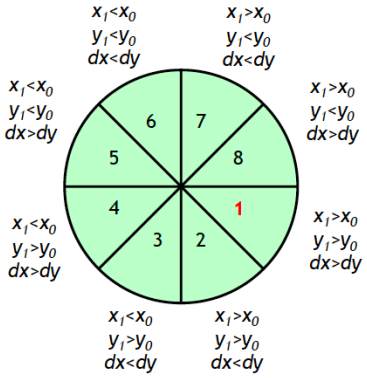
2.2.1 Interpolation Algorithm

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;
      }
}
```

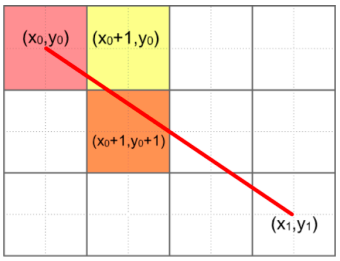
2.2.2 Bresenham Algorithm

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

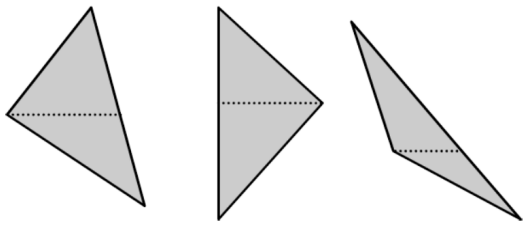
```
dy = y0 - y1;
dx = 2*(x1 - x0);
idist = 0;
x = x0; y = y0;
plot(x, y, c);
for(y = y0-1; y >= y1; y--) {
    idist += dx;
    if(idist > dy) {
        x++;
        idist -= 2 * dy;
    }
    plot(x, y, c);
}
```

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.

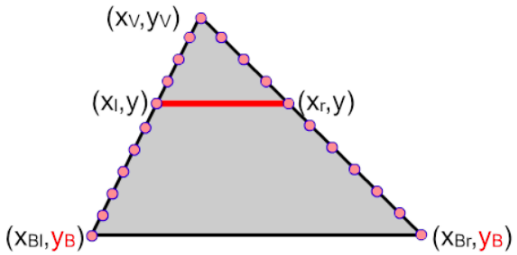


2.3.1 Triangles with edge // to x-axis

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

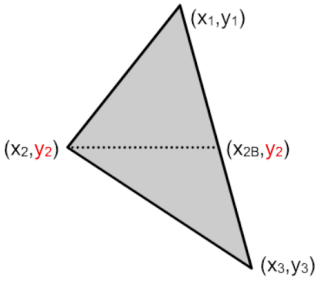
The two edges not parallel to x-axis can be considered as two lines.// The triangles is **filled** by drawing horizontal lines that connect pixels over the angled edges. The x_l, x_r coordinates can be found using **interpolation**.



```
{  /*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) :

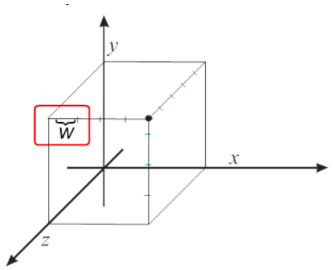
$$x_s = (s_w - 1) * (x_n + 1)/2$$

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

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**
- z 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**

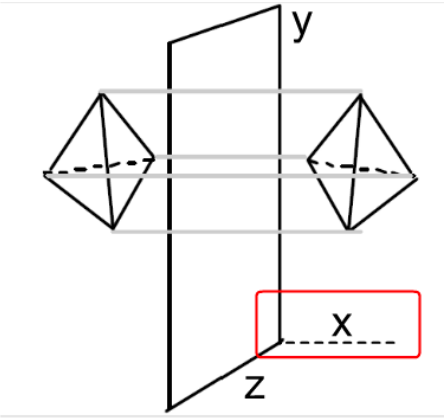


Figure 3: $s_x = -1, s_y = 1, s_z = 1$

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

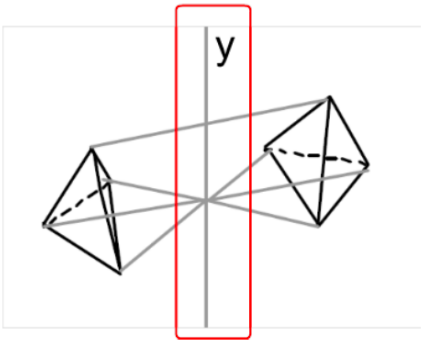


Figure 4: $s_x = -1, s_y = 1, s_z = -1$

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

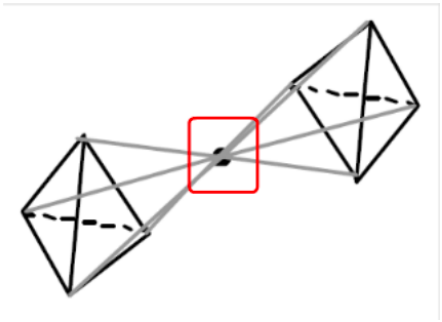


Figure 5: $s_x = -1, s_y = -1, s_z = -1$

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 α 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 α 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 α about the x-axis :

$$x' = x$$

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

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

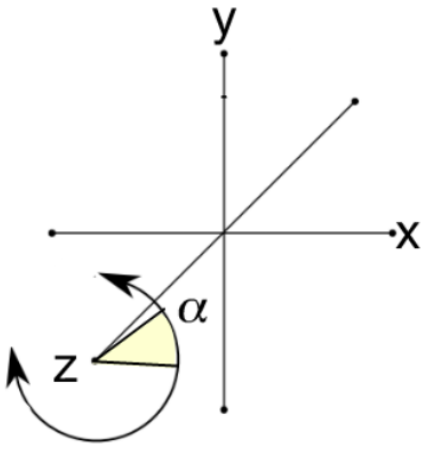


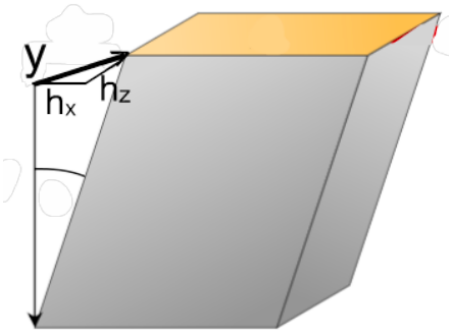
Figure 6: Z-Axis rotation

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 :

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

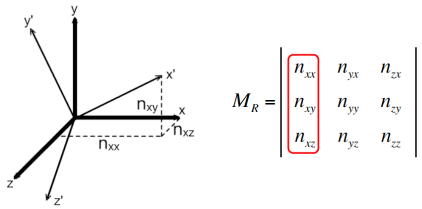
4 3D Transform

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**

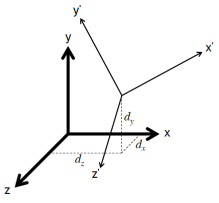
The columns of M_R represent **directions & sizes** of the new axes in the old reference system
Rotations maintain the size of the axis constant but change their direction.
Scalings maintain the direction of the axis constant , changing the size.



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

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

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

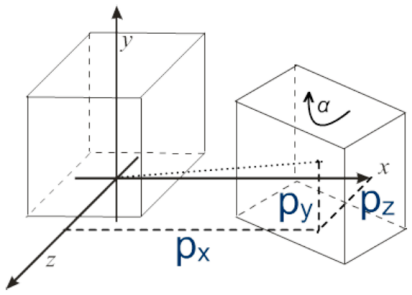
Figure 8: Scaling

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

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 α 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.

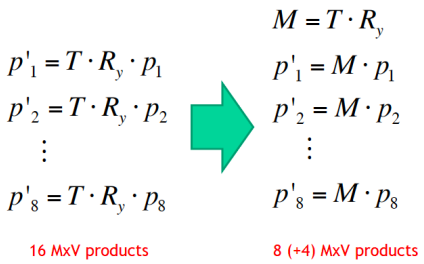
4.2.1 Properties of composition of transformations

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 :

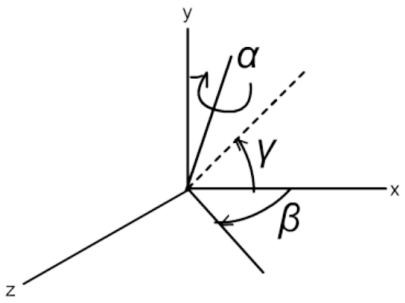
$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

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 α 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 γ, β :
- $\gamma \rightarrow$ 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 α can be done around the x-axis.
4. $R_z(\gamma)$ and $R_y(\beta)$
   To restore the original axis position.

Final transformation composition :

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

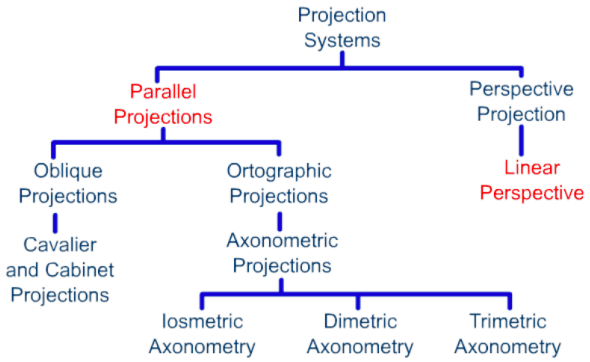
If the axis does not pass through the **origin** , a **translation T** must be applied of a point known through which the axis passes:

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

Similar procedures can be applied to :

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

5 Projections



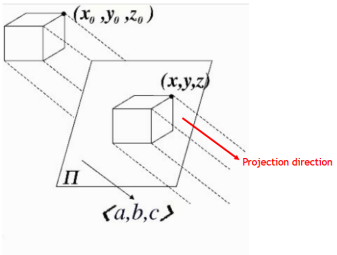
In 3D computer graphics the goal is to represent a three dimensional space on a screen with 2 dimensions:

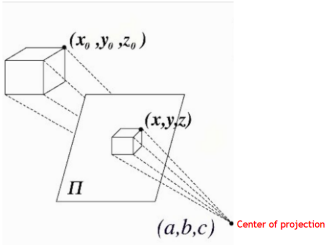
- The 3D graphics uses geometrical primitives defined in 3 dimensions
- 3D graphics produces a 2D representation of the scene to show on screen.

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

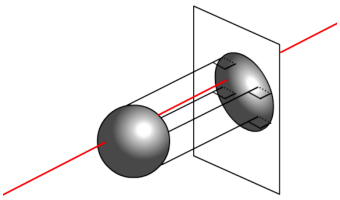
So to create a 2D projection of a 3D polyhedron it is sufficient to **project its vertices** and connect them.
In parallel projections all the rays are **parallel to the same direction**.
In perspective projections all the rays pass **through a point** called

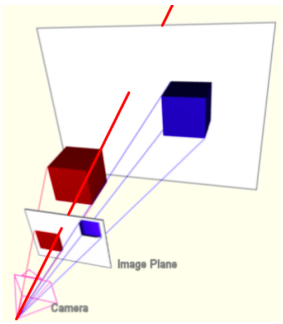




Doing a projection , we loose one coordinate so a point on screen corresponds to an **infinite** number of coordinates (consequence of moving from 3D $\rightarrow$ 2D) :

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**.

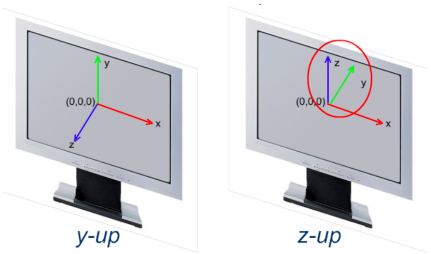




In 3D computer graphics the concept of projection becomes the **conversion** of 3D coordinates from one reference system to another.
World coordinates→ 3D Normalized Screen Coordinates

World coordinates

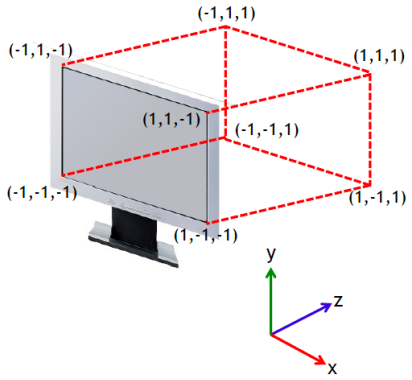
Coordinate system that describes the objects in the 3D space. It is a right-handed Cartesian coordinate system with the **origin** in the **center of the screen.** Some applications invert the z and y axis , with the y axis point inside the screen.



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.



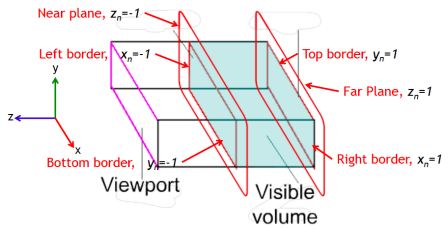
5.1 Parallel Projections

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.

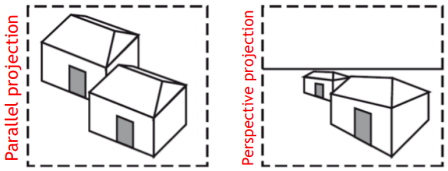
$$P_{ort} = \begin{vmatrix} \frac{1}{w} & 0 & 0 & 0 \\ 0 & \frac{a}{w} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

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

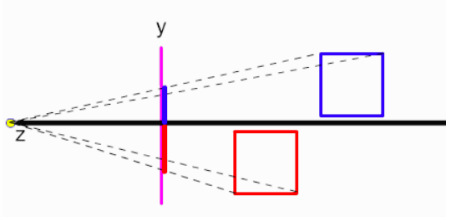
Parallel projections do **not** change the apparent size of an object with the distance from the observer. It is used mainly for drawings and is not that suitable for 3D computer graphics.

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**.

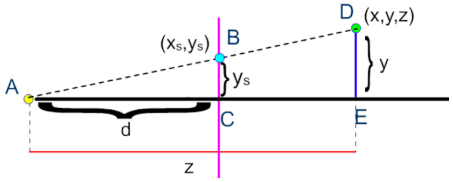


Rays intersect the **projection plane** at different points depending on the

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

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

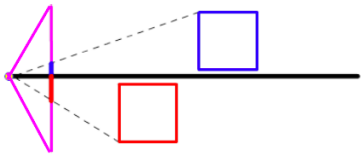
- x_s

  The same computation as for y_s occur :

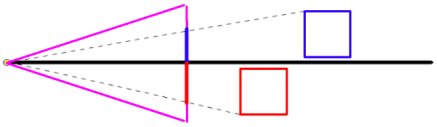
$$\boxed{x_s = \frac{d \cdot x}{z}}$$

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 :

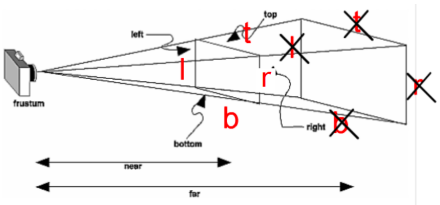
$$|x_s, y_s, -d - \frac{1}{z}, 1|$$

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 ∞.

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**:



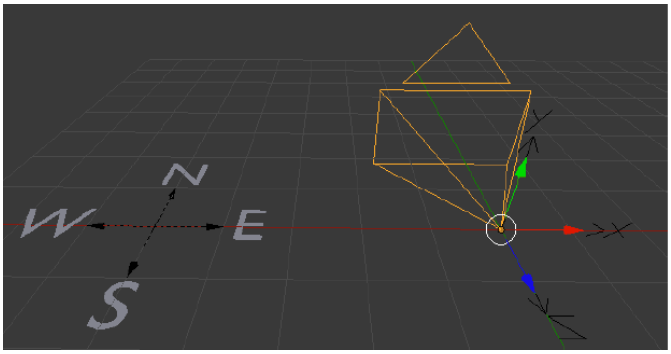
The frustum is defined by it's **near** and **far** plane and **top**,**bottom** ,**left** and **right** coordinates. The coordinates are **not constant** any more: now they depend on the **distance**.
By default t,b,l,r are defined on the **near plane** so the distance **d** corresponds to the value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

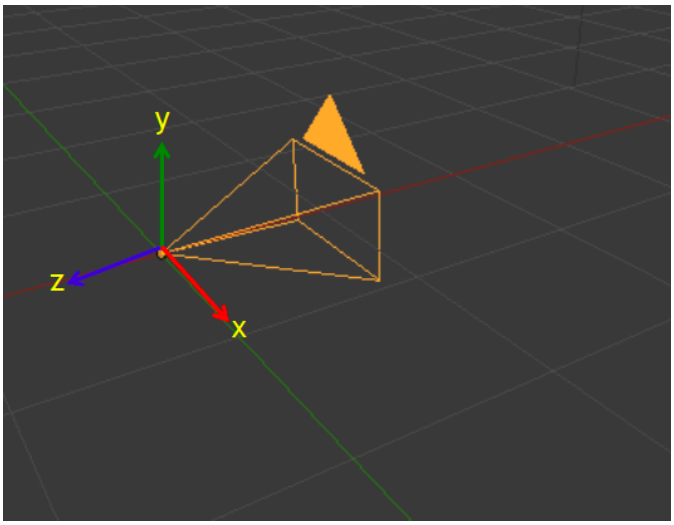
The view matrix assumes that the **projection plane** (= the screen) is the xy axis.

- 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):

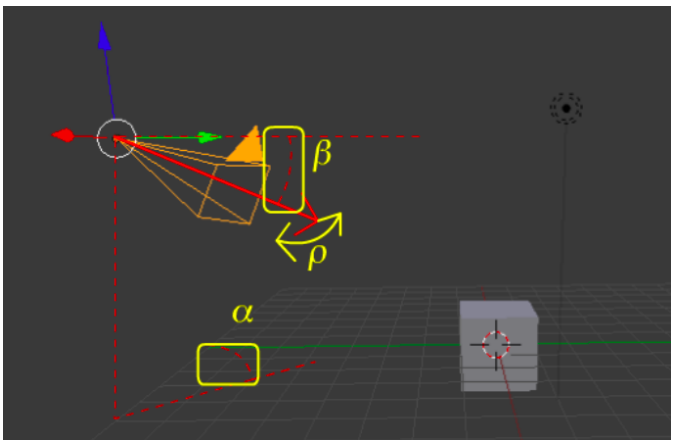
- **look-in direction matrix**
- **look-at matrix**

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
- α 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
- β vertical angle = look up and down
  - $\beta > 0° \rightarrow$ look up
  - $\beta < 0° \rightarrow$ look down
- ρ 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 :

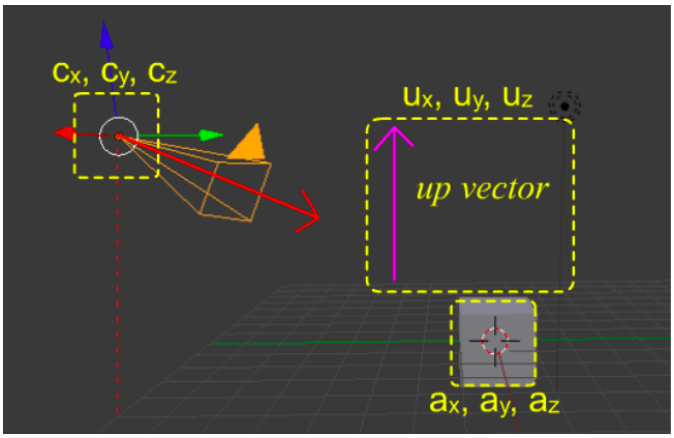
$$A' = P_{proj} \cdot M_v \cdot A$$

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)

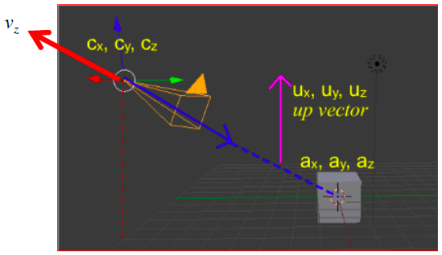
These two points are not enough as they do not decide the **roll** of the camera. To specify the orientation of the camera the **up vector** is added. It is the inverse of gravity and shows which way is up in the world. The camera bottom axis should always be **perpendicular** to the up vector : this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

$$v_x = \frac{uxv_z}{|uxv_z|}$$

$$uxv_z = |u_x, u_y, u_z|x|v_x, v_y, v_z| = |u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x|$$

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

$$v_y = v_z x v_x$$

Which leads to

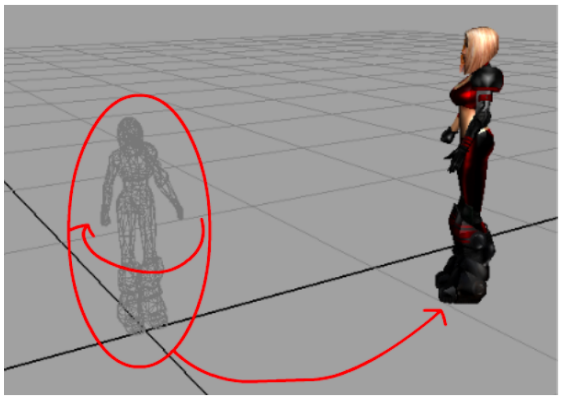
$$M_c = \begin{array}{ccc|c} v_x & v_y & v_z & c \\ 0 & 0 & 0 & 1 \end{array} = \begin{array}{c|c} R_c & c \\ 0 & 1 \end{array}$$

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

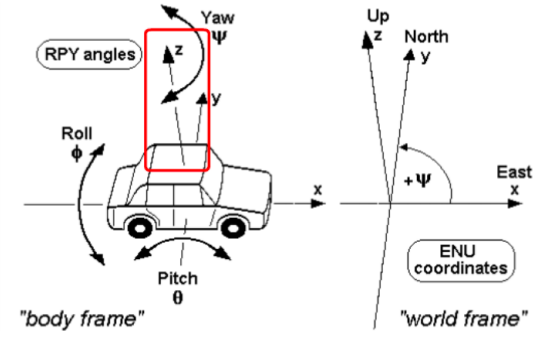
Must be performed between scaling and positioning.
To define a specific orientation in 3D space a combination of rotations along the 3-axis must be performed. In which order must these rotation be performed?
Several ways exist to compute the rotation of the object. A consistent way to specify the parameters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

- **Roll (x-axis)**
  The roll ϕ 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 θ 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 ψ defines the direction of the object and corresponds to a rotation along the vertical axis. $\psi = 0° \rightarrow$ East

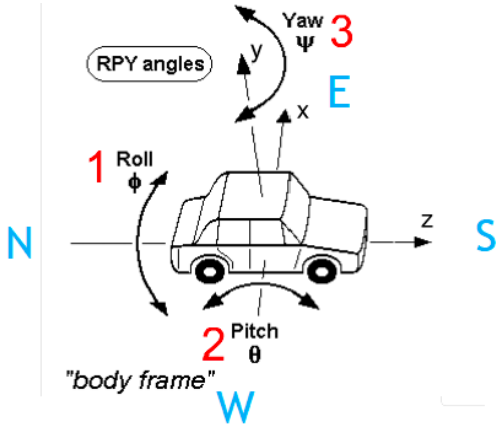
Euler angles are defined for a z-up coordinates system.



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 ϕ and Pitch θ work in the opposite way as ρ and β 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

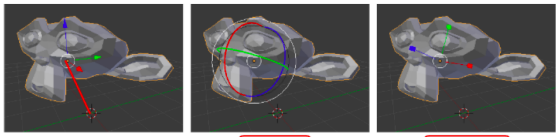
Must be performed last otherwise the coordinates would be changed during the other transformation.
When positioning the object the user wants to specify the coordinates where it should be placed in the 3D place.These coordinates should be independent of the size and orientation of the object.
Positioning 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

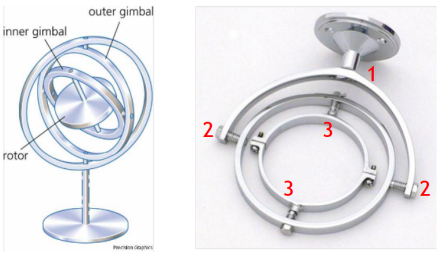


The final world matrix is :

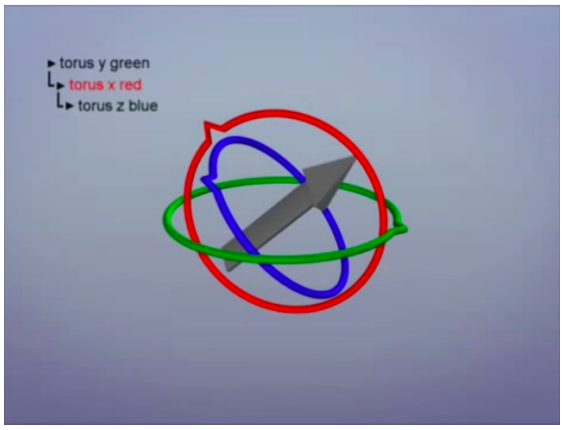
$$\boxed{M_w = T(p_x, p_y, p_z) \cdot R_y(\psi) \cdot R_x(\theta) \cdot R_z(\phi) \cdot S(s_x, s_y, s_z)}$$

6.3 Gimbal Lock

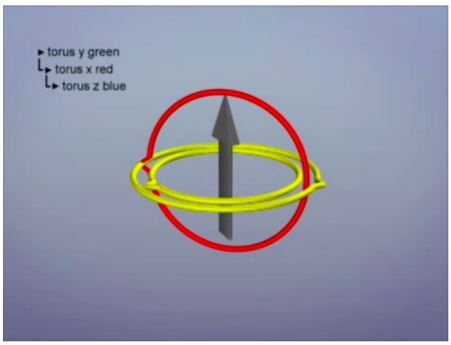
A rotation defined by the Euler Angles is perfect for **planar** movements (good for driving games or FPS).Euler Angles are a problem for applications such as flight simulators as they suffer from a problem known as **gimbal lock**. A **gimbal** is a ring that can spin around its diameter.



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**) :

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

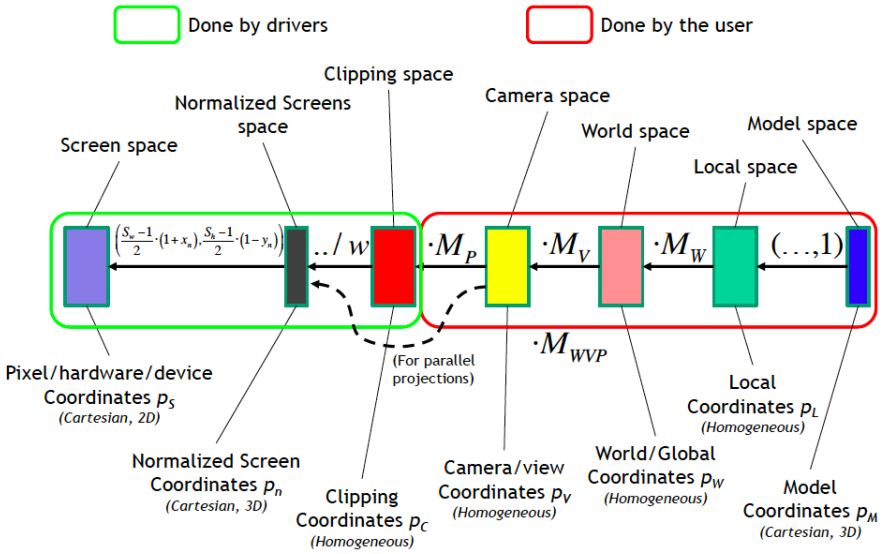
The **Normalization** step is require in case of perspective projections where the transformations produces **clipping coordinates**. As opposed to other transformations this step is done by normalizing the homogeneous coordinates that describe the points in the clipping space. Every component is divided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

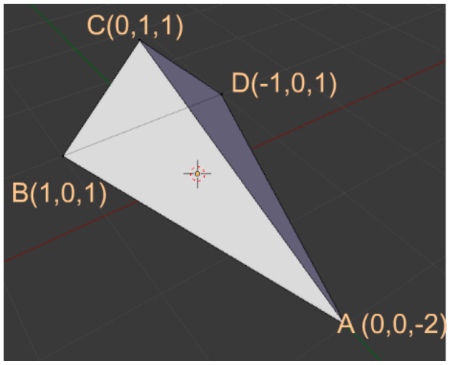
coordinates and the into pixel coordinates to show objects

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



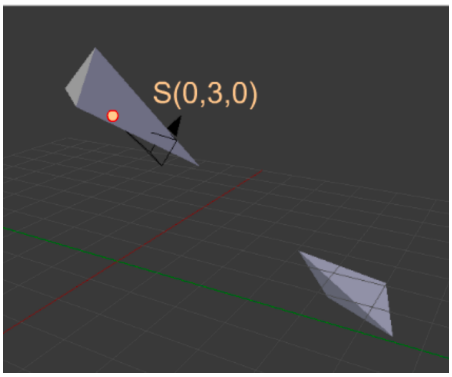
7.2 The example

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 $120°$

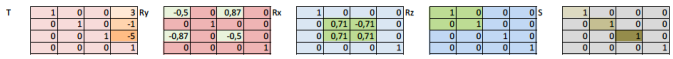


What are the pixel coordinates of the vertex of the tetrahedron seen on a 960x540 pixel screen (5:4 aspect ration with non square pixels)?
Additional informations :

- 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

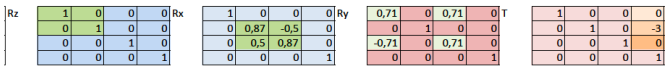
- Position $(p_x, p_y, p_z) = (3, -1, -5)$
- Rotation Yaw,Pitch,Roll$= (120°, 45°, 0°)$
- Scaling $S = (1, 1, 1)$



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

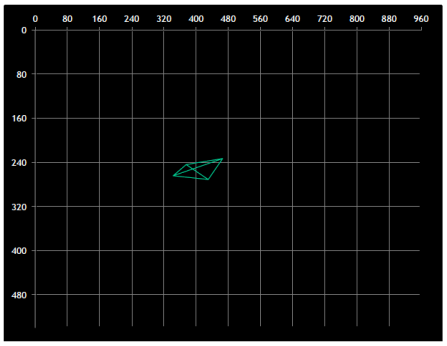
Then we multiply the points of the vertexes (to which a fourth component equal to 1 is added) with the WVP Matrix.

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 :

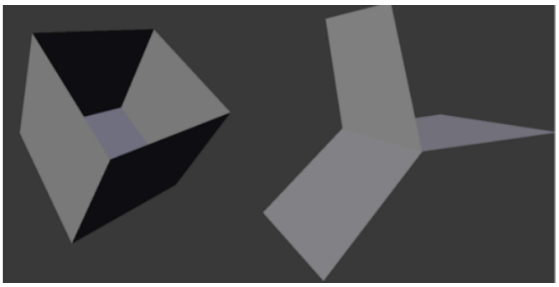
- **Meshes** (polygonal surfaces)
- **Hermite surfaces**
- **NURBS** (non uniform ration b-splines)
- **HSS** (hierarchical subdivision surfaces) **Metaballs**

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

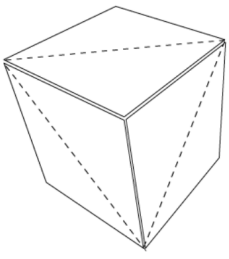
8.1 Meshes

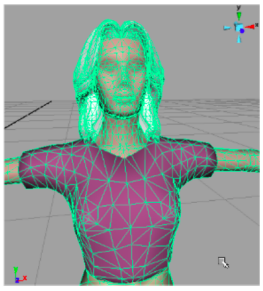
A **mesh** is a polygonal surface that can be described by a set of contiguous polygons (cube, pyramid , prism are meshes ; cones,cylinders,spheres are not meshes but can be rendered as one using tricks shown later).
A polygon that describes a planar surface portion of a surface is called a **surface**.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



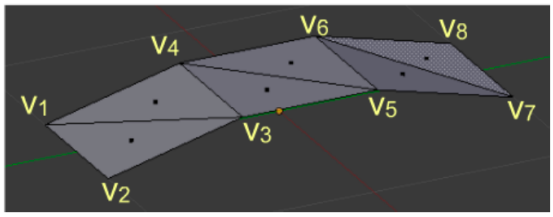
Sometimes non-2- manifold objects can be used to model very thin objects (pages...) or to obtain a special effect (opne box, magazine...). In these kind of situations the later presented **back-face culling** algorithm will **NOT** work.
Each polygon can be reduced to a set of **triangles** that shares 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.



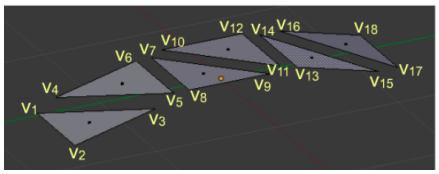


8.1.1 Mesh encoding

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

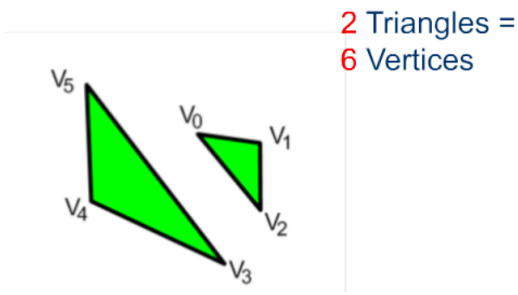


Three main types of mesh-encoding are:

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

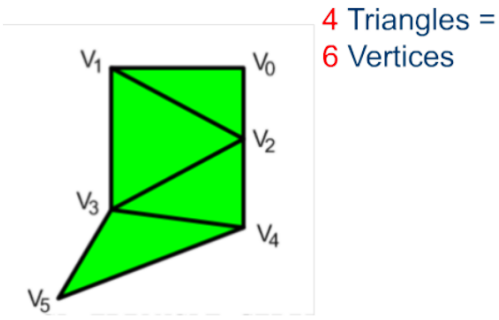
Triangle lists

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



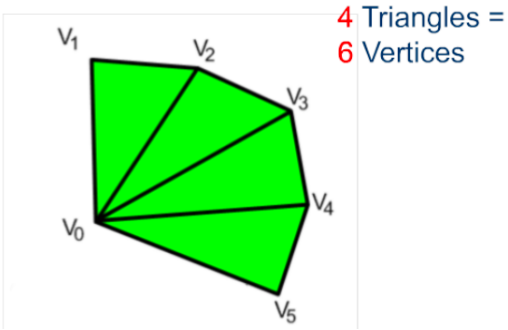
Triangle strips

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

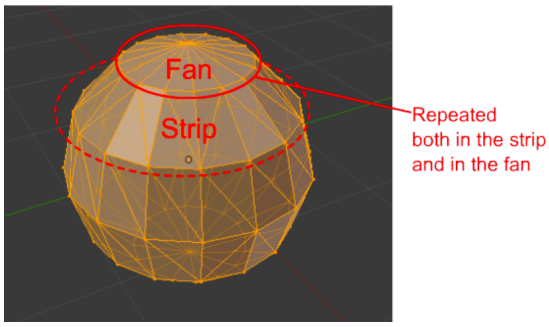
Triangle fans encode polygons where all the triangles share a **vertex**.The first two vertexes are specified independently.Then each new vertex connects both the previous one and the first of the list.



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.

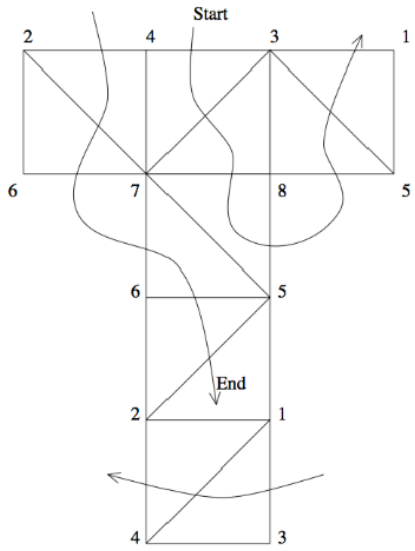
- **Index Array**:
  Are used to **indirectly** specify the triangles. For example a cube has 6 faces so we have 12 triangles. So in the index array we have 12 elements (a,b,c) where a,b,c are the indexes of the vertex array composing that triangle.

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):

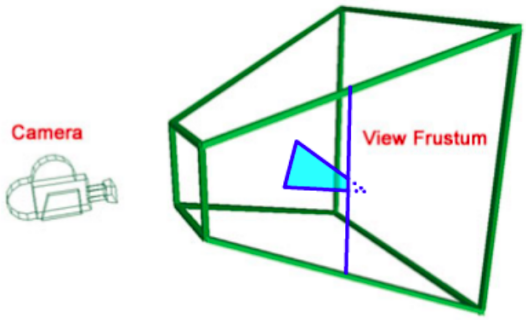
$$8_{vert} * 3_{xyz} * 4_{float} + 6_{faces} * 2_{tri} * 3_{vert} * 1_{byte} = 132Bytes$$

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:

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

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

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

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

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

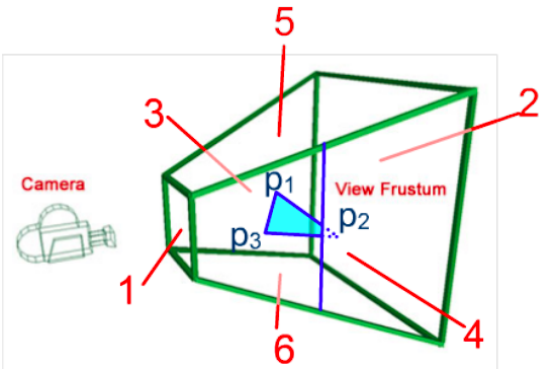
$$\frac{z}{w} < 1, \quad z < w, \quad -z + w > 0, \quad n_f = |0 \quad 0 \quad -1 \quad 1|$$

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?



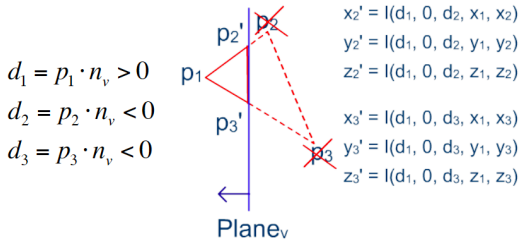
As shown the triangle and the frustum face are intersected . Computationally this is done by the **Sutherland-Hodgman Algorithm**.
Focusing one on the figure above we have a face defined by its vector n_v and the three vertexes of the triangle $p1, p2, p3$:

$$d_1 = p_1 n_v$$

$$d_2 = p_2 n_v$$

$$d_3 = p_3 n_v$$

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:

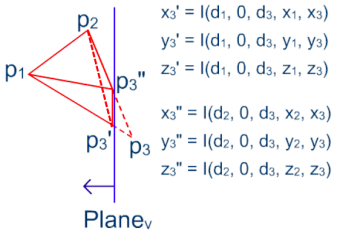
$$p_2' = (x_2', y_2', z_2')$$

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

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

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.

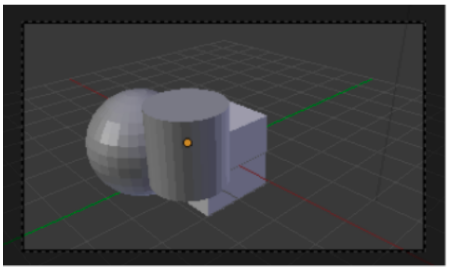
$$d_1 = p_1 \cdot n_v > 0$$
$$d_2 = p_2 \cdot n_v > 0$$
$$d_3 = p_3 \cdot n_v < 0$$



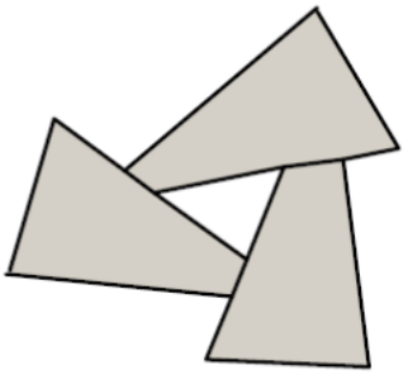
The algorithm terminates when all faces have been checked for clipping or all three points are outside the frustum.
The algorithms is simple but can produce many triangles since it can potentially double at every check

9 Hidden Surfaces

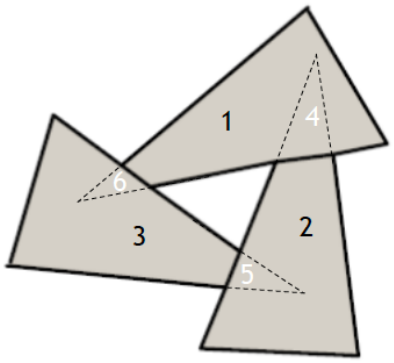
In a complex scene where many objects can overlap it is important that polygons closer to the viewer **cover** the objects behind them. If faces are not drawn in the correct order unrealistic figures will appear.Respecting the proper order of primitives visualization is called **Hidden Surfaces Elimination**.



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:

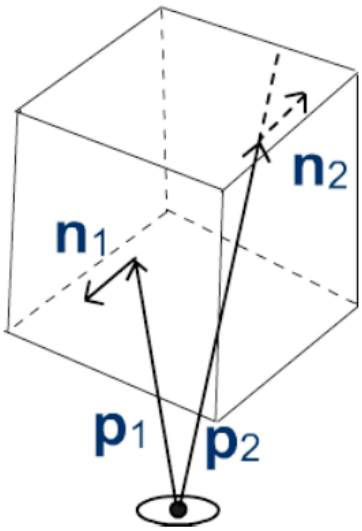
- **Back-face culling**
  Allows identifying and excluding objects that belong to the back of an object
- **Occlusion culling**
  Excludes objects that fall completely behind others (not covered in the course)
- **Z-Buffering**
  Allows to implement the painter algorithms without sorting the polygons 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



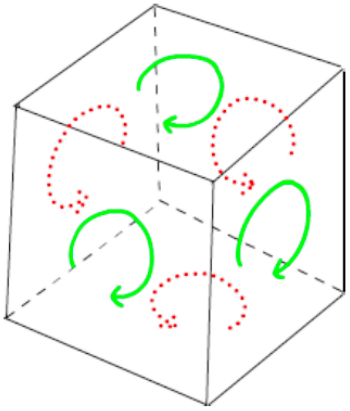
A cube is encoded as a set of 12 triangles. All the triangles are obviously **planar** surfaces. Each triangle surface has its **normal** vector n ,directed outside. The projection rays are directed from the viewer to the object. By performing a **scalar product** $p \cdot n$ we obtain:

- $p \cdot n < 0$ 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

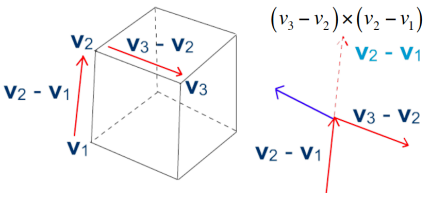
The normal vector :

- 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).

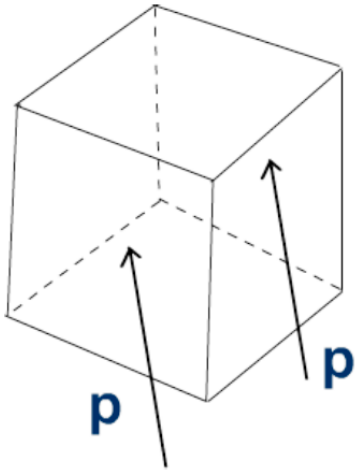


The cross-product of two vectors is a vector that is perpendicular to the plane (direction is determined by the right hand rule). In triangles the normal vector can be computed starting from the difference of two vectors (identified by the vertexes) and then the cross product of the two differences is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

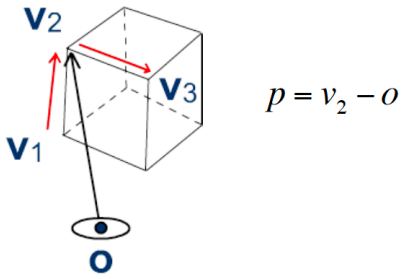


The projections rays:

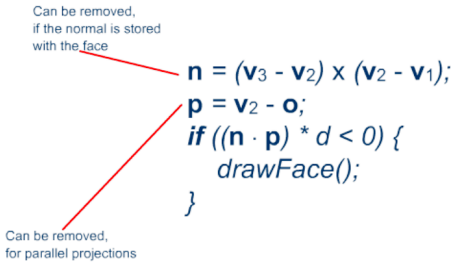
- in **parallel** projection the vector p is **fixed** and corresponds to the direction of the projection ray.



- 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 :

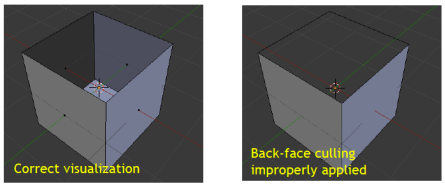


Where d is a constant of +1 if we want to accept faces ordered clockwise or -1 if we want to accept faces ordered counter clockwise.
Back-face culling can be applied :

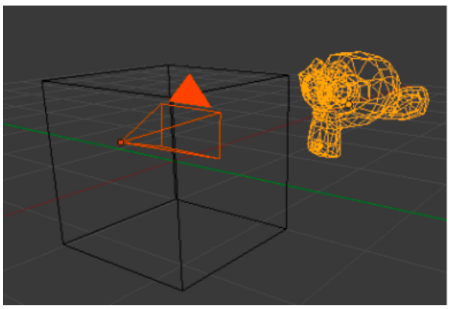
- **before** the world and view transformations.In this case the center of projection or the projection rays have to be mapped in the local space of the object by **inverting** the view and world transformations.
- **after** the world and view transformations. In this case if the vector has been stored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



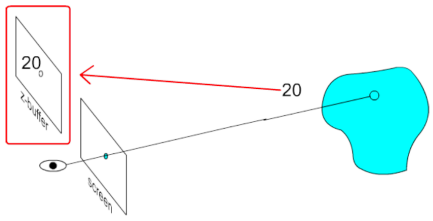
Usually engines separate non-2 manifolds from 2-manifolds for this reason.
Another special situation is when the camera is **inside** an object (for example inside a room/box). In this case the normal is oriented in the wrong direction. When designing a scene this must be taken into account if back-face 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).



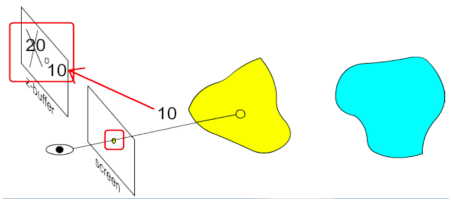
A solution to this is to create a room/skybox with the normal vectors pointing towards the **inside.**

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

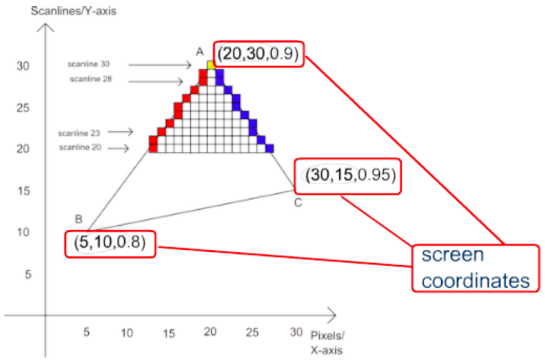
Issue 1

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

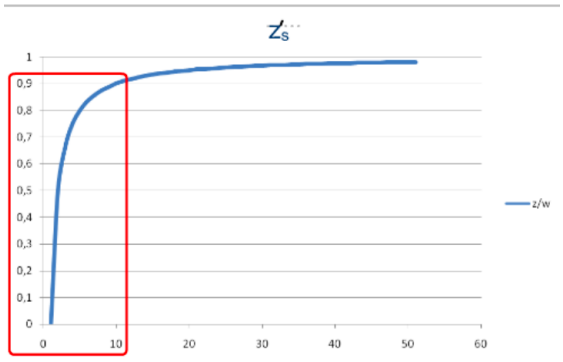
When dealing with triangles and their pixels also the z-coordinate must be interpolated : this is not always easy. **Normal screen coordinates** of x,y,z are in the range -1,1 and **pixel coordinates** are even more compressed for z in the range $z_n \in [0, 1]$.



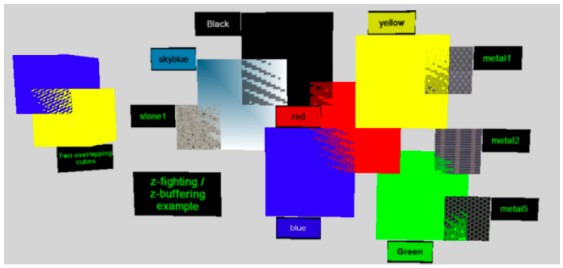
With these coordinates the vertices can be stored and from there via **interpolation** the corresponding pixels can be found. However the z-coordinate on screen has a **non-linear** behaviour : the distance between the lines becomes smaller as we move away from the plane and cannot be obtained by using 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)



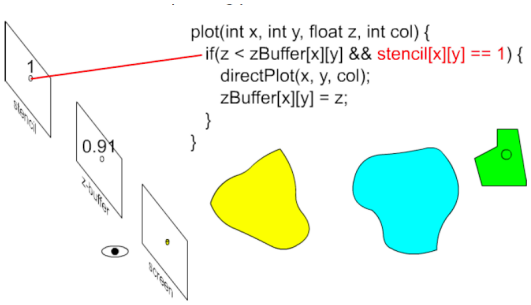
Since values a **discretized** we need more precision to store values that are further away otherwise a problem called **Z-fighting** occurs: when two almost co-planar figures are rendered the final **color** is determined by the round off error.



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.

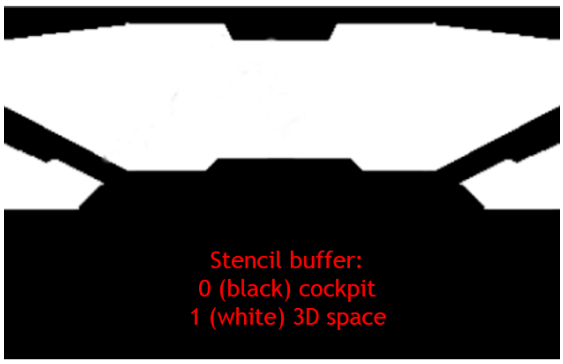
9.2.2 Stencil buffer

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.