

---

# COMPUTER GRAPHICS

---

## PART 1

BY YANNICK GIOVANAKIS

March 20, 2018

# Contents

<b>1</b>	<b>Graphic Adapters</b>	<b>3</b>
1.1	Vector Graphic Adapters . . . . .	3
1.2	Raster Graphic Adapters . . . . .	3
1.3	Accelerated Graphic Adapters . . . . .	4
1.4	Color vision . . . . .	5
1.4.1	Human Vision . . . . .	5
1.4.2	Color reproduction . . . . .	6
1.5	Image Resolution . . . . .	7
<b>2</b>	<b>2D Graphics</b>	<b>8</b>
2.1	Point primitives . . . . .	8
2.1.1	Linear Interpolation . . . . .	9
2.2	Line primitives . . . . .	10
2.2.1	Interpolation Algorithm . . . . .	11
2.2.2	Bresenham Algorithm . . . . .	12
2.3	Triangle Primitives . . . . .	15
2.3.1	Triangles with edge // to x-axis . . . . .	15
2.3.2	Triangle splitting . . . . .	16
2.4	Normalized coordinates . . . . .	17
<b>3</b>	<b>3D Graphics</b>	<b>18</b>
3.1	Affine Transformations . . . . .	18
3.1.1	Translation . . . . .	19
3.1.2	Scaling . . . . .	20
3.1.3	Rotation . . . . .	22
3.2	Shear . . . . .	25
<b>4</b>	<b>3D Transform</b>	<b>26</b>
4.1	Inversion of transformations . . . . .	27
4.2	Composition . . . . .	28
4.2.1	Properties of composition of transformations . . . . .	28
4.3	Transformations around an arbitrary axis or center . . . . .	30

<b>5</b>	<b>Projections</b>	<b>32</b>
5.0.1	Parallel Projections . . . . .	34

# 1 Graphic Adapters

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

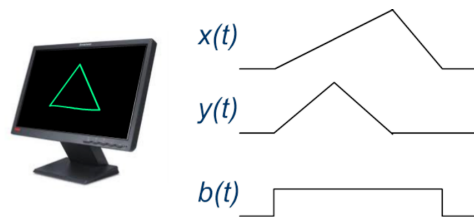
## 1.1 Vector Graphic Adapters

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.

```
move 20,20  
beam on  
move 40,60  
move 60,20  
move 20,20  
beam off
```



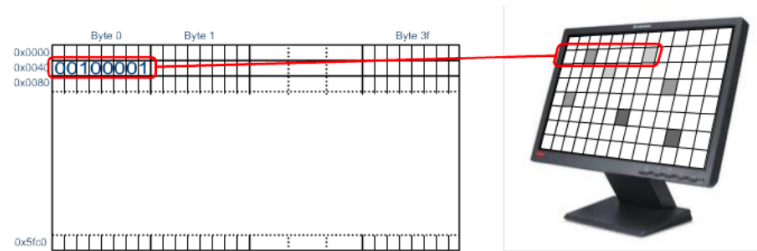
## 1.2 Raster Graphic Adapters

Raster graphic adapters divide the screen into a **matrix** of individually assignable elements called **pixels** which are assigned **colors**. If the color comes from a spatial sampling of an image the adapter can reproduce it on the monitor.

Raster adapters have a special memory called Video Memory (**VRAM**) made of cells that contain information about the color of each pixel on the screen. A

component on the video card ( RAMDAC for analog displays ) converts the information to the signal required to transfer the image to the monitor.

Images on the screen can be written by setting specific values in the VRAM ( `writeScreen()` function). Still in used but slowly dismissed due to reduction of



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

### 1.3 Accelerated Graphic Adapters

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

Commands that can be interpreted by the adapter include :

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

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

## 1.4 Color vision

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

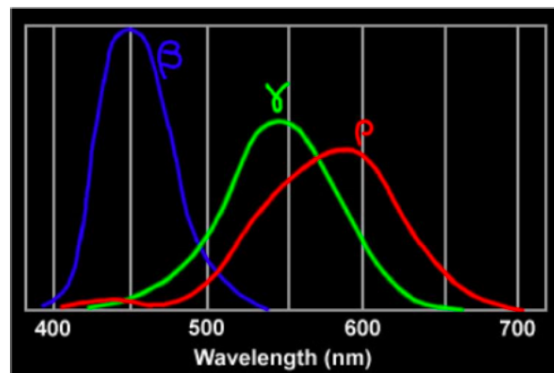
### 1.4.1 Human Vision

The color of the light is determined by the **wavelength** of the photons that transmit it. Visible light ranges from 400-700 nm wavelength.

Depending on the **light source**, lots of photons of **different wavelengths** are emitted. The photons then interact with the environment where objects, depending on their composition, **reflect or absorb** the various wavelengths at different intensities. The reflected photons are then focused on the retina where **rods** (sensible to light intensity) and **cones** (sensible to light color) transmit information to the brain through **nerves**.

There are 3 types of cones  $\rho, \beta, \gamma$  each sensible to a different portion of light spectrum.

By combining the stimuli of different cones the brain allows the vision of a given color.



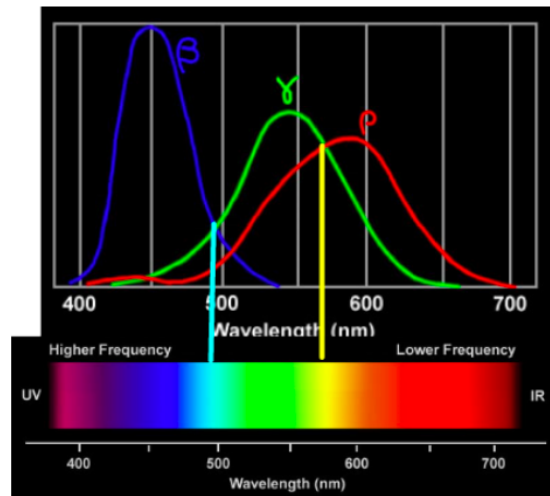
### 1.4.2 Color reproduction

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 & blue** , different hues are constructed by mixing light of these three.

Mixing two of three primary colors **cyan, magenta and yellow** are obtained.

Mixing the three in different proportions **all** possible hues can be obtained.



#### Color range

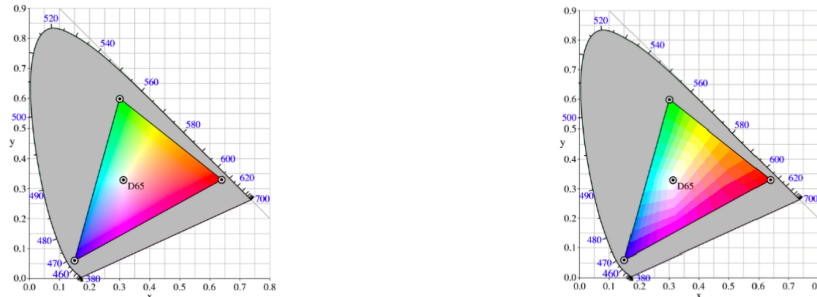
The **spectrum** that a monitor can produce is just a small portion of the entire spectrum the eye can see (grey area).

Color range of a monitor corresponds to a cube where the three colors are placed on the 3 axis.

#### Color synthesis

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

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



## 1.5 Image Resolution

The resolution of an image defines the **density of pixels** that compose it. When dealing with **raster graphics on screen** the density is relative to the **monitor size** : the resolution defines thus the number of pixels displayed on the screen on the horizontal (**width** ) and vertical (**height** )

Pixels are not always **square shaped** so the horizontal resolution  $\neq$  vertical resolution.

### Memory of Images

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

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

A first way to reduce image size was to reduce the **colors** using a **color palette** : a predefined set of colors that contains a **limited** number of entries.

The palette is encoded as an array of **RGB values** that are used to define the possible colors that can be used in an image.

If the **color depth d** (number of bits per pixel) is  $\leq 8$  a color palette is used ( the palette contains  $2^d$  colors).

If a user defined palette is used, the size of the palette must be added to the image size. With  $p$  = bits to encode a palette entry

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



## 2 2D Graphics

2D Graphics primitives are procedures that draw simple geometric shapes based on a **2D coordinates system** , a set of integer with unit the **pixel**  $\rightarrow$  *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

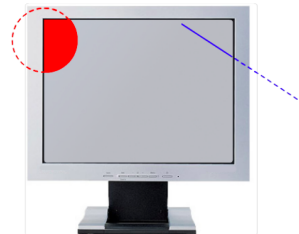


Figure 2: clipping

### 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  $y_{i-1}$  :

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

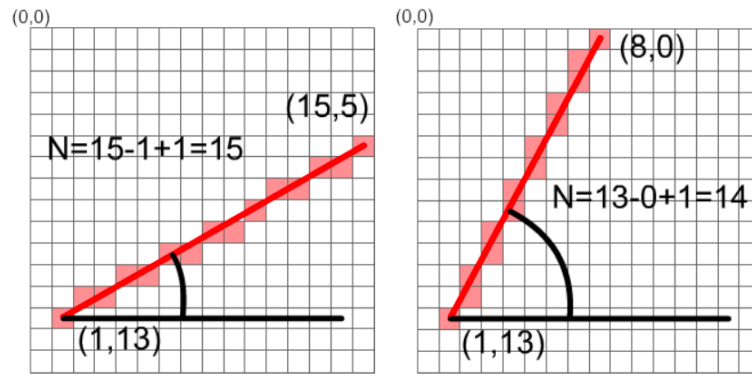
## 2.2 Line primitives

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.

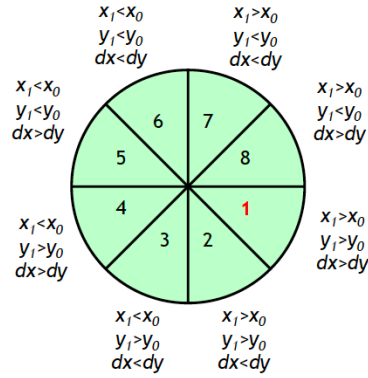
---

```
{
    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;
        }
    }
}
```

---

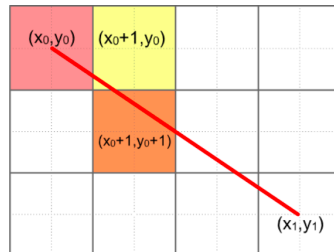
### 2.2.2 Bresenham Algorithm

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** for 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**

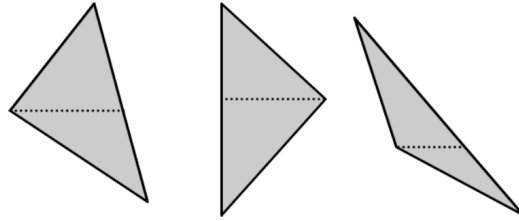
```

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.



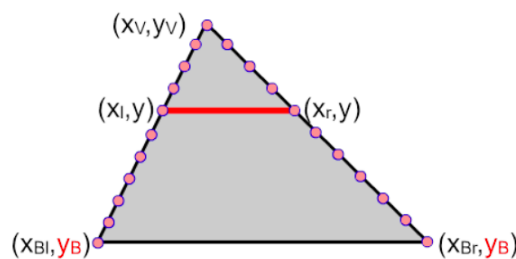
### 2.3.1 Triangles with edge // to x-axis

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**.





---

```

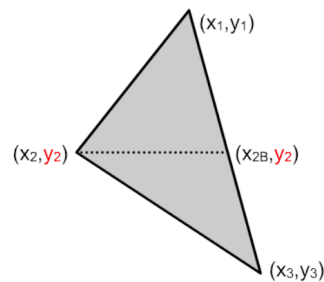
{ /*since the two lines have diff. slope , store in
   dxr ,dxl the increments in the x direction */
   dxl = (xB1 - xv)/(yB - yv);
   dxr = (XBr - xv)/(yB - yv);
   x1 = xr = xv; // starting from vertex
   /* assumption yv < yB , otherwise change loop direction */
   for ( y = yv; y <=yB ; y++) {
       for(x=round(x1);x<=round(xr); x++){
           plot(x,y,c)
       }
       x1 += 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  $(x_2, y_2)$  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  $(x_2, y_2)$

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

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

## 2.4 Normalized coordinates

Current displays are available in different **resolutions** and **sizes**. Moreover in windowed operating systems applications must be **confined** only in a portion of the screen. When changing the resolutions/window size the applications still want to show the **same image** exploiting all the features of the display. A special coordinates system called **Normalized Screen Coordinates** is normally used to address points on screen in a device in an independent way.

NSC are Cartesian coordinate system where x and y range between to **canonical values** [ OpenGL -1  $\rightarrow$  1 ]. If the window/memory area resolution is known in  $s_w, s_h$  pixels, the coordinates system  $(x_s, y_s)$  can be derived from the normalized screen coordinates  $(x_n, y_n)$  :

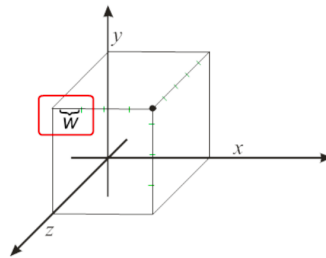
$$x_s = (s_w - 1) * (x_n + 1) / 2$$

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

### 3 3D Graphics

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

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



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

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

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

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

#### 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  $4 \times 4$  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  $\rightarrow$  rugby ball
- **Mirroring** ex: Object on the right  $\rightarrow$  symmetrical on the left

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

Now we assume that the center corresponds to the **origin**.

- **Proportional scaling**

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

$$x' = s \cdot x$$

$$y' = s \cdot y$$

$$z' = s \cdot z$$

If  $s > 1 \rightarrow$  **enlarge**

Else  $0 < s < 1 \rightarrow$  **shrink**

- **Non proportional scaling**

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

$$x' = s_x \cdot x$$

$$y' = s_y \cdot y$$

$$z' = s_z \cdot z$$

If  $s_i > 1 \rightarrow$  **enlarge**

Else  $0 < s_i < 1 \rightarrow$  **shrink**

A **scaling matrix** can be used to represent transformations :

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

- **Mirroring**

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

1. **Planar**

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

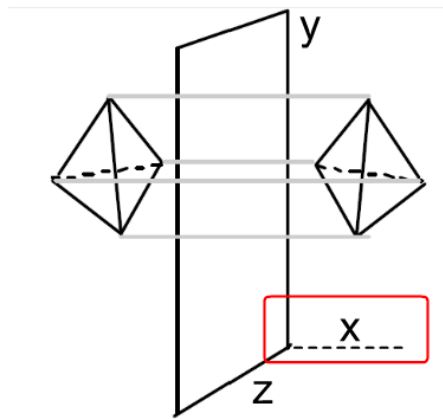


Figure 3:  $s_x = -1, s_y = 1, s_z = 1$

## 2. Axial

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

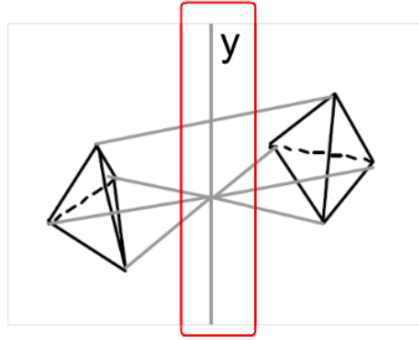


Figure 4:  $s_x = -1, s_y = 1, s_z = -1$

## 3. Central

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

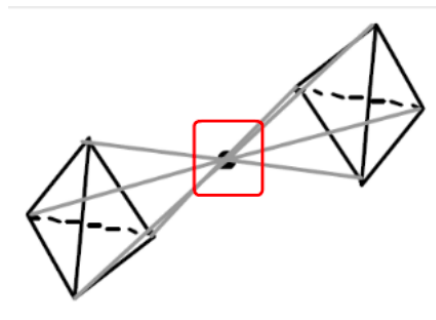


Figure 5:  $s_x = -1, s_y = -1, s_z = -1$

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

### 3.1.3 Rotation

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

transformation.

Rotation can occur also on non conventional axis but we will mainly consider rotations along x,y,z axis passing through the origin.

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

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

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

$$z' = z$$

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

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

$$y' = y$$

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

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

$$x' = x$$

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

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

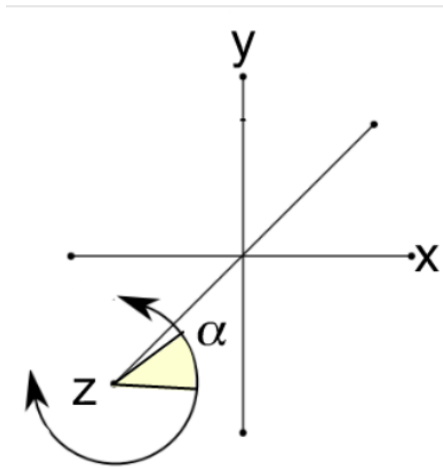


Figure 6: Z-Axis rotation

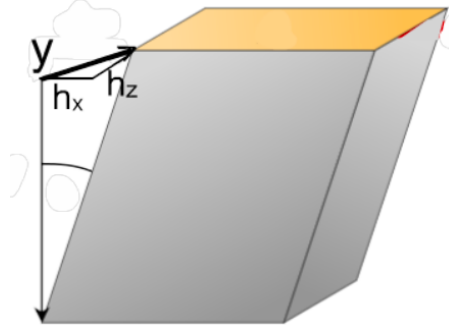


Again matrices can be used to express rotation :

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

### 3.2 Shear

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



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

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

$$y' = y$$

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

Along the **x-axis**:

$$x' = x$$

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

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

Along the **z-axis**:

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

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

$$z' = z$$

Again matrices can be used to express shear :

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

## 4 3D Transform

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

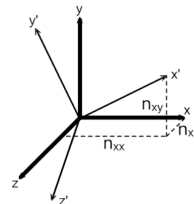
$$M = \left| \begin{array}{ccc|c} n_{xx} & n_{yx} & n_{zx} & d_x \\ n_{xy} & n_{yy} & n_{zy} & d_y \\ n_{xz} & n_{yz} & n_{zz} & d_z \\ \hline 0 & 0 & 0 & 1 \end{array} \right| = \left| \begin{array}{c|c} M_R & \mathbf{d}^T \\ \hline \mathbf{0} & 1 \end{array} \right|$$

- **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

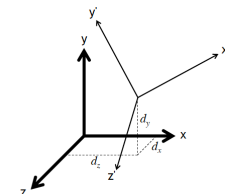
**Rotations** maintain the size of the axis constant but change their direction.

**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  $\mathbf{d}^t$  represents the position of the origin of the new coordinate system in the old one



$$M = \left| \begin{array}{c|c} M_R & \mathbf{d}^T \\ \hline \mathbf{0} & 1 \end{array} \right|$$

## 4.1 Inversion of transformations

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

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

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

Matrix  $M^{-1}$  is **invertible** if its submatrix  $M_R$  is invertible. Generally  $M^{-1}$  is always invertible except when dealing axis degeneration ( zero factor scaling for example).

Another method of inverting transformations is by using a reverse matrix :

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

Figure 7: Translation

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

Figure 8: Scaling

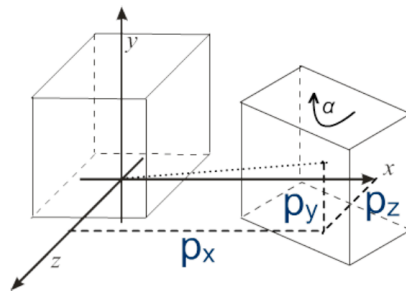
$$R_x(-\alpha) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha & 0 \\ 0 & -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad R_y(-\alpha) = \begin{vmatrix} \cos \alpha & 0 & -\sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}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

Figure 9: Reverse rotation

## 4.2 Composition

During the creation of scene an object is subject to **several** transformations. Applying a **sequence of transformations** is called **composition**. An example is the movement of a cube, sides parallel to x,y,z axis and with center in the origin.

- Translation of center to position  $p_x, p_y, p_z$
- Rotation of angle  $\alpha$  around y



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

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

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

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

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

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**.

$p'_1 = T \cdot R_y \cdot p_1$ $p'_2 = T \cdot R_y \cdot p_2$ $\vdots$ $p'_8 = T \cdot R_y \cdot p_8$		$M = T \cdot R_y$ $p'_1 = M \cdot p_1$ $p'_2 = M \cdot p_2$ $\vdots$ $p'_8 = M \cdot p_8$
16 MxV products		8 (+4) MxV products

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 :

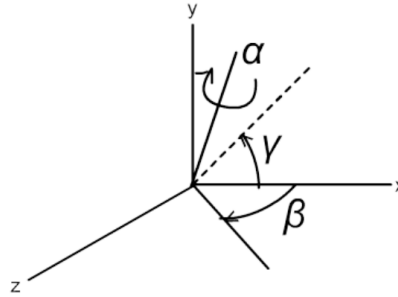
$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

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

### 4.3 Transformations around an arbitrary axis or center

#### Case : Rotation

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



In this case the two angles are  $\gamma, \beta$  :

- $\gamma \rightarrow$  how much the axis rises on the xz plane
- $\beta \rightarrow$  how much it rises on the xy plane

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

1.  $R_y(-\beta)$

Considering a rotation of  $-\beta$  along the y-axis ,the arbitrary axis now lies on the xy plane.

2.  $R_z(-\gamma)$

Then considering a rotation of  $-\gamma$  along the z-axis ,the arbitrary axis now corresponds to the x-axis.

3.  $R_x(\alpha)$

As our chosen axis corresponds to the x-axis , the rotation of the object of angle  $\alpha$  can be done around the x-axis.

4.  $R_z(\gamma)$  and  $R_y(\beta)$

To restore the original axis position.

Final transformation composition :

$$p' = R_y(\beta)R_z(\gamma)R_x(\alpha)R_z(-\gamma)R_y(-\beta) \cdot p$$

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

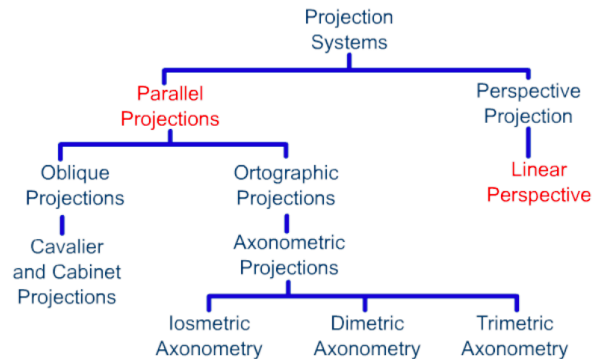
$$p' = T(p_x, p_y, p_z)R_y(\beta)R_z(\gamma)R_x(\alpha)R_z(-\gamma)R_y(-\beta)T(-p_x, -p_y, -p_z) \cdot p$$

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.

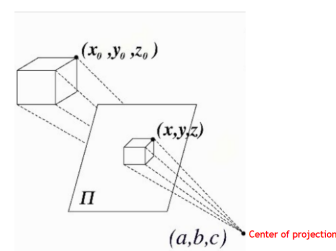
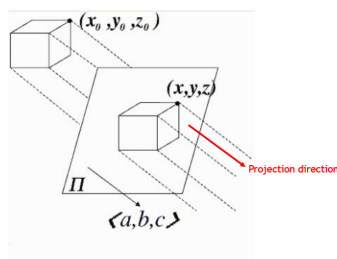
The second step is performed using **projections**. Key features :

- 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**.

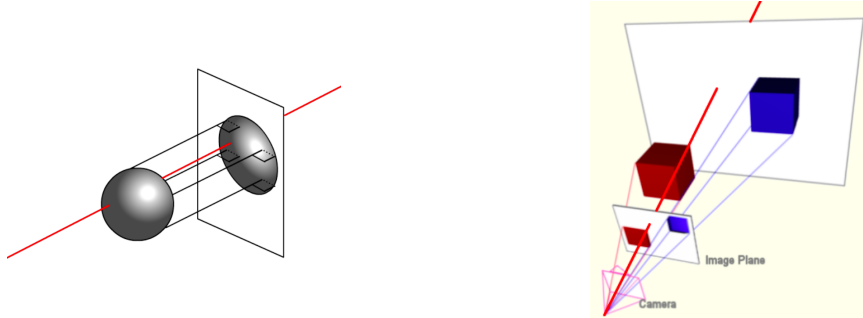
In perspective projections all the rays pass **through a point** called



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**.

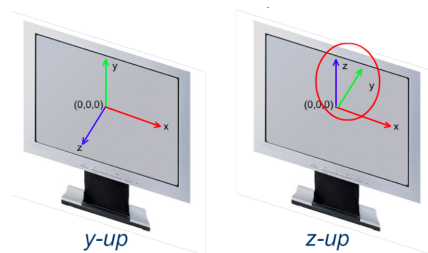


In 3D computer graphics the concept of projection becomes the **conversion** of 3D coordinates from one reference system to another.

**World coordinates** → **3D Normalized Screen Coordinates**

### World coordinates

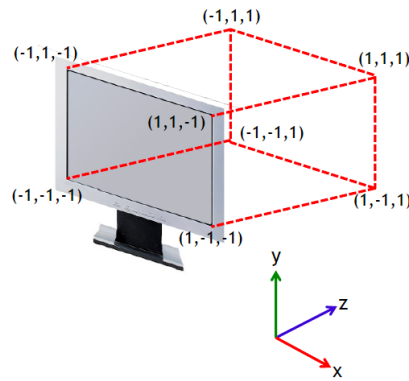
Coordinate system that describes the objects in the 3D space. It is a right-handed Cartesian coordinate system with the **origin** in the **center of the screen**. Some applications invert the z and y axis, with the y axis point inside the screen.



### 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 from the same extents (ex : -1,1). This way coordinates with a smaller z-value will be considered to be **closer** to the viewer.



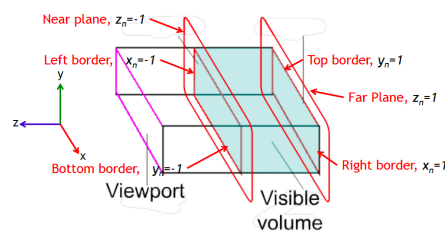
## 5.1 Parallel Projections

**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** → **near plane**
- The plane with the **maximum z component** → **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!