

3 Graphics in 2D

3.1 Aims

This Module reviews some fundamental 2D graphics operations. These operations are implemented in DirectX, but the ideas are broadly applicable. In particular, we'll look at vector graphic operations, how to use transformation matrices, issues in clipping and collision detection. We'll base this module around a single problem that explores these issues.

For this module, we are going to write a classic TV tennis game, with a few extra features. The screen represents the playing court, and is divided down the middle by a net. Each player is represented by a rectangle on the left and right sides of the screen. The ball is represented by square. Each player starts at the centre of the baseline for the serve. After the serve, the players are free to move about their respective halves of the court. Players move by rotating to face the direction that they wish to travel, and then travelling forwards or backwards. That is, there are two keys to rotate the player, and two keys to travel forward and backward. If the centre of the ball is contained within the rectangle of the player, a hit is registered. Every time the ball is hit, your algorithm is to randomly choose a position on the opponent's baseline to hit to. The ball never travels over the side lines. The speed of the ball and the players is constant. You may choose the speeds and sizes of objects for playability.



We will break this problem into a number of smaller sub-problems throughout the module.

3.2 Drawing Lines

Bresenham's algorithm is the key to drawing lines quickly. It also useful for making objects travel in a straight line from arbitrary start points to arbitrary end points.



Modify Bresenham's algorithm to implement the ball travelling back and forth across the screen. Assume that the ball is always hit back when it reaches the far side of the screen. Blit the ball onto the screen, being

careful to always blit inside the boundaries of the surface. Draw the net as well.

3.2.1 Clipping Lines

The important thing to recognise here is that if you are directly flipping pixels in the video RAM, then you are going to have to do your own clipping. The DirectDraw clipper is great for blitting, but doesn't help when you're directly accessing pixels in video RAM. The supplied `Draw_Clip_Line()` function draws within a pre-defined clipping region; feel free to use it.

3.2.2 Drawing Polygons

Clearly drawing any kind of vector based polygon means that you are going to need to store the vertices of the polygon in order so that can quickly draw lines between the vertices to create your shape. You may use a general polygon structure, such as the text defines, or tie one into the data structure of the game object that the polygon represents.

3.3 Transformation Matrices

These matrices are the same as those you saw in high school math and first year linear algebra. Some common examples of 2D matrices include:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx \\ sy \end{bmatrix} \quad \text{Scale by factor of } s$$

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{Rotation through } \phi$$

as well as other interesting operations such as reflection and shearing. Look them up in your old text books.

However, a 2 x 2 matrix cannot produce translation. This makes it not so useful for most applications where we want to investigate movement through space. For these purposes we use homogeneous coordinates. Homogeneous coordinates require the use of an N+1 x N+1 matrix to manipulate N x N coordinates. So, work in 2D space the matrix takes up the form:

$$T = \begin{bmatrix} s \cdot \cos(\phi) & -\sin(\phi) & m \\ \sin(\phi) & s \cdot \cos(\phi) & n \\ 0 & 0 & 1 \end{bmatrix}$$

which performs (in the following order) a rotation through ϕ , a scaling of s , then a translation across (m,n) . The vectors are modified to their homogeneous form by the addition of a 1 to the base of the vector to produce a 3 x 1 vector, thus:

$$v = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Using matrices of this nature, you can make a standard 3 x 3 matrix to transform all the points in your polygon without having to recalculate a whole bunch of trig functions every time. A single matrix can represent any number of functions by simple pre-multiplication. For example, to get the transformation matrix to perform a translation of (-10, 20), followed by a rotation of 60°, followed by a translation of (50, 30) you would use (noting the order):

$$T = \begin{bmatrix} 1 & 0 & 50 \\ 0 & 1 & 30 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(60) & -\sin(60) & 0 \\ \sin(60) & \cos(60) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 20 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & -0.866 & 27.6795 \\ 0.866 & 0.5 & 31.3397 \\ 0 & 0 & 1 \end{bmatrix}$$

If the order of the translations is reversed, the resulting transformation matrix, and correspondingly the resulting transform are quite different. In fact, it would be:

$$\begin{bmatrix} 0.5 & -0.866 & -10.9808 \\ 0.866 & 0.5 & 78.3013 \\ 0 & 0 & 1 \end{bmatrix}$$



Using code liberally from DEMO8_6, add the code for player drawing and player movement to your ball code. Don't worry about hitting the ball yet. Draw the player as an unfilled rectangle for now.

3.4 Colouring In

Rasterising algorithms are very nice, but we really don't have a need to cover them in detail. Use the rasterising code supplied in DEMO8_8 for the following exercise.



Now colour-in the players using the Draw_QuadFP_2D() routine.

3.5 Collision Detection

Any game with agents that interact physically is going to require some kind of collision detection. The text covers bounding circles and boxes. Here are some comments on the techniques in the text and some obvious extensions and improvements.

3.5.1 Bounding Circles

Bounding circles can be used as an approximation of a polygon, as indicated in the text. They also can be used for detecting whether an object is in range. The text talks about the problems with `sqrt()` which is a notoriously slow function. Series

expansion is one way to get a square root approximation, but there is an easier way. Generally, when using a bounding circle, we perform a comparison of the type:

```
dist = sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2));
if (dist < min_radius) {
    // Collision code
}
```

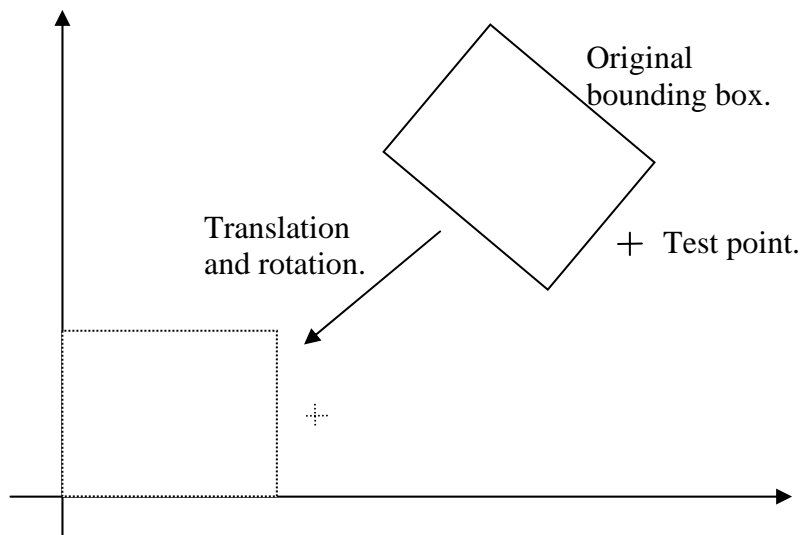
which uses the dreaded square root call. This can easily be avoided by re-writing thus:

```
dist_sq = (x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2);
if (dist_sq < (min_radius * min_radius)) {
    // Collision code
}
```

which exchanges a multiply for a square root call; much quicker.

3.5.2 Bounding Box

Finding a bounding box as the text suggests is usually not of much use, except as a quick first check before further investigation for a collision. A more useful bounding box is one oriented to the polygon, particularly if the polygon is a rectangle. To do this, we use transformation matrices to rotate and translate the bounding box and the test point to the origin. Once the bounding box and the test point have been translated and rotated, the comparison is straight forward.



Use this idea to perform the collision detection for your tennis game. Finish off your game by coding up the second player, and tuning up the size and speed of the players and ball.

3.6 Preparation for Next Week

Read the rest of Chapter 9, and Chapter 10, pages 566 - 653



