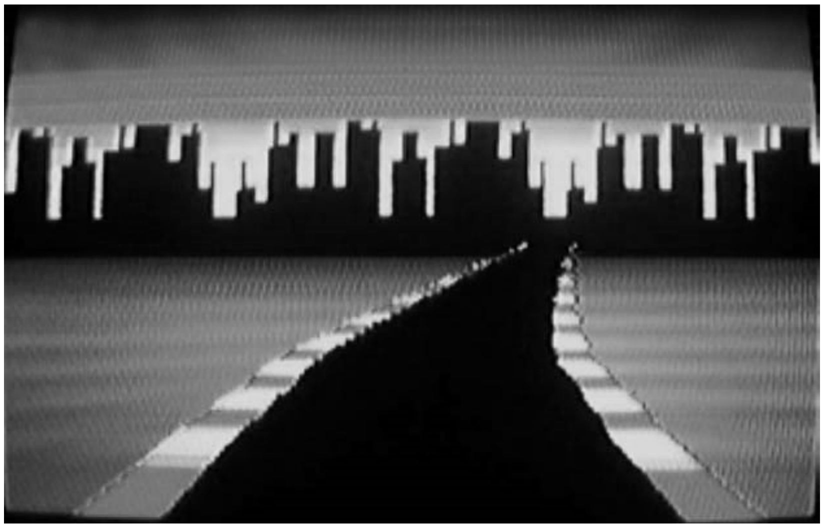
Racer City Demo - Pico Edition Tutorial

The racing demo is a reasonably complex example of detailed, full-screen graphics with a pseudo-3D perspective. It creates a smoothly animating race track with a distant city skyline as its backdrop that scrolls appropriately as the driver turns. Special details are included to enhance the visuals, such as a gradient effect in the sky and the classic dark-and-light track border that scrolls with the road. See Figure 1 for a screenshot of the racing engine demo in action.

Figure 1 – The racing engine demo.



The demo code itself is located on the CD in the following location:

CDROOT:\XGSME_HW_CD\XGSME_Sources\racer_city_pico_01.SRC

There are a handful of major elements required to make this demo work:

- The gradient sky
- The city skyline that scrolls with the turns of the track
- The track and its pseudo-3D perspective, as well as its ability to warp into the shape of a left or right turn
- The light and dark track markers that scroll along with the player's movement
- Joystick input from the player

Porting XGS Micro Edition Programs to the XGS Pico Edition

The Racer City demo was originally written on the XGS ME. The Racer City demo discussed in this chapter is a direct port of that program, not an original piece of XGS PE code. This makes it a good example of how code can be ported from one system to another, a task that can range from trivial to impossible.

The biggest difference between the XGS ME and the XGS PE are the processors; the XGS ME is based on the SX52, with **4KB** of program memory, **256** bytes of file registers and **6** globals, while the XGS PE is based on the SX28, with **2KB** of program memory, **128** bytes of file registers, and **8** globals. Aside from these differences, however, the processors themselves are nearly identical and as such, porting can actually be very easy in some cases.

The ease and feasibility of porting a program from one system to the other depends primarily on the program's size. An SX52 program that requires most or all of the 4KB of program space will be either difficult or impossible to port to the SX28. In the case of the Racer City demo, code was spread out well beyond the 2KB point, but was easily reorganized to fit within less than 1KB. In this case, the code itself wasn't too big and didn't need to be rewritten, but was spread sparsely over the 4KB despite needing less than one fourth of the space, and simply needed to be tightened up.

The other major difference between the two systems are the I/O ports. On the XGS ME a single 8-bit port represents both luminance and chroma (4-bits each). On the XGS PE the video port is 4-bit and represented by the lower 4-bits of RC while the upper 4-bits of RC are used for the audio output. The joystick on the XGS ME is also much more complex due to the serial hardware that scans the joystick switches. On the XGS PE the joystick is much simpler and directly connected to port RB's lower 5-bits. The I/O mapping for the XGS PE is shown below in Table 1.

Table 1 – I/O Mapping for the XGS Pico Edition.

Register	***Description***
RA.3-0	4 LED array
RB.4-0	Joystick input port
RC.3-0	4-bit video output port
RC.7-4	4-bit audio output port

The ported version of the racer city demo generates only black and white video and as such, sends only the luminance portion of a given color to the video output port. This is especially important to remember when porting full-color XGS ME programs, since the upper 4 bits of the `RC` register are now used as audio output, and any chroma data sent to this port will likely result in a grating, unpleasant sound from your TV speakers.

NOTE

Color is absolutely possible with the XGS PE; however, unlike the XGS ME, the Pico Edition does not have color "helper" hardware, so you must generate a color burst yourself and use software delays to create the phase differences to generate color. This is described in Chapter 11 of "Design Your Own Video Game Console". I omitted it in this demo, since the timing to generate the chroma signal complicates the demo a great deal.

One advantage of the XGS PE is that its simpler, 1-player joystick interface is much easier to read than the XGS ME. Since the joystick's output pins connect directly to the XGS PE, no shift register is involved and any joystick pin can be read at any time.

The Sky Background and City Skyline

The first major element of the racing engine demo to discuss is the background, consisting of a gradient sky and a scrollable city skyline. The skyline in particular is an illuminating look at how complex graphics can be implemented on the XGS ME using simple tricks to overcome its incredibly small amount of on-chip memory.

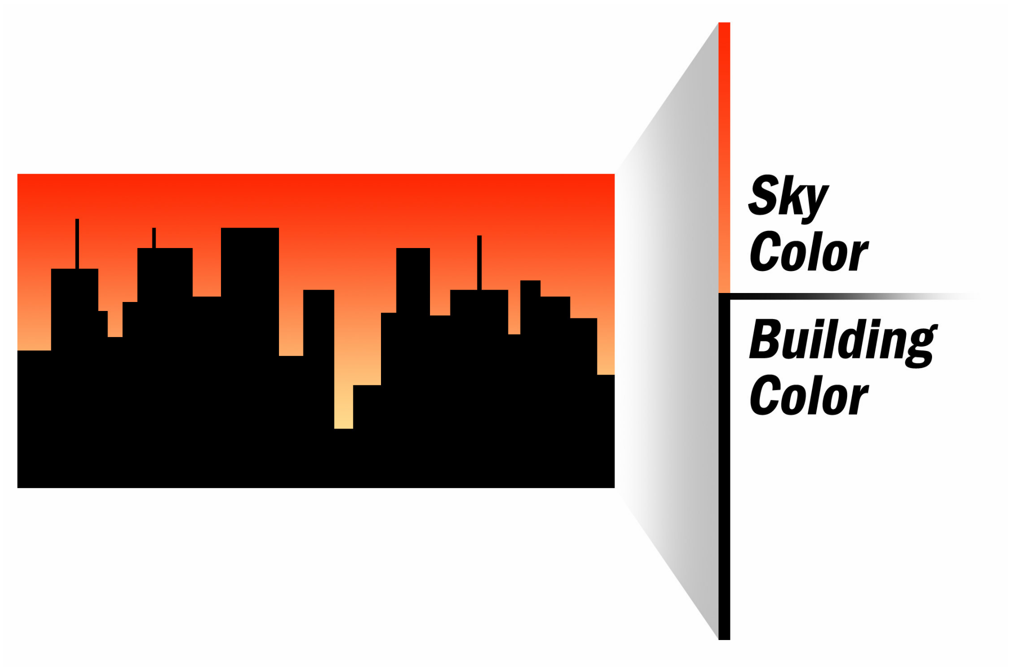
The skyline is a detailed, high-resolution graphic that covers the entire screen. It is a total of 256 pixels across, which is even wider than the screen's horizontal resolution. The skyline is a total of 44 scanlines tall, which means that a bitmap capable of containing this entire skyline would be 44 x 256 = **11K**! With only 128 file registers of RAM and 2K of program space, this is definitely impossible. So where does the skyline come from?

The City Skyline Height Table

The answer is a lookup table similar to the one we discussed in the last chapter for producing pseudo-random numbers. Since a complete 11K bitmap is not possible on the SX52 (without using the slower, off-chip SRAM), the skyline had to be represented in a form that only stored the absolute minimum amount of data. It also had to be fast, however, because each pixel drawn in the skyline only has about 20 clocks to perform its logic. This means that any sufficiently sophisticated compression algorithm would probably be too complex to implement on a per-pixel basis.

The solution was a ***height map***. A height map, in this case, allows us to think of the skyline as a simple one-dimensional entity, rather than a two-dimensional bitmap. Therefore, the skyline need only be 1D because it only changes along the X-axis. If you were to look at any 1-pixel vertical strip of the skyline, you would find that above the skyline, the strip is solid sky, and below the skyline, it is solid building. Figure 2 illustrates this.

Figure 2 - Analyzing 1-pixel vertical strips of the skyline graphic.



Because of this, when drawing any pixel in the background, we do not need to perform a 2D bitmap lookup to determine the color; all we need to know is whether or not we are above the level of the skyline at this particular point on the X-axis. If so, we draw using the sky background color. If not, we draw using the skyline. Think of the skyline table as any mathematical function (such as trig functions), except it's drawn not as a line, but as one solid color above, and another solid color below.

This effectively removes the Y-axis from the definition of the skyline and thus reduces the memory needed from an impossible 256 x 44 bytes to a more-than-possible 256 bytes. With a 256-byte single lookup table, an entire skyline can be stored:

```
skyline
        DW      10,10,10,10,20,30,20,20,15,15,35,50,35,35,40,40,20,20,20,40,40,39,39,20,20,50,50,50,40,40,50,40
```

```
       DW        10,10,30,30,40,40,40,30,30,10,35,35,35,45,35,35,50,50,50,49,48,47,46,30,30,40,40,55,40,20,20,30
       DW        10,10,10,10,20,30,20,20,15,15,35,50,35,35,40,40,20,20,20,40,40,39,39,20,20,50,50,50,40,40,50,40
       DW        10,10,30,30,40,40,40,30,30,10,35,35,35,45,35,35,50,50,50,49,48,47,46,30,30,40,40,55,40,20,20,30
       DW        10,10,10,10,20,30,20,20,15,15,35,50,35,35,40,40,20,20,20,40,40,39,39,20,20,50,50,50,40,40,50,40
       DW        10,10,30,30,40,40,40,30,30,10,35,35,35,45,35,35,50,50,50,49,48,47,46,30,30,40,40,55,40,20,20,30
       DW        10,10,10,10,20,30,20,20,15,15,35,50,35,35,40,40,20,20,20,40,40,39,39,20,20,50,50,50,40,40,50,40
       DW        10,10,30,30,40,40,40,30,30,10,35,35,35,45,35,35,50,50,50,49,48,47,46,30,30,40,40,55,40,20,20,30
```

NOTE

While the skyline height table data could be generated in numerous ways, it was literally created by hand-editing each value in the table seen above, recompiling, and making changes until it looked right. While it would be overkill for a program such as this, any program that needs a lot of high-quality lookup-table-based graphics would benefit from a graphical utility that translates mouse-drawn curves and shapes into a table as seen above.

Another Approach to Lookup Tables

To read from a table such as this, the **`IREAD`** instruction is used. **`IREAD`** accepts a 12-bit program memory address stored in **`M:W`** and returns the 12-bit program word found at that address, also in **`M:W`**. In both cases, the high-nibble is stored in **`M`**, and the low-byte is stored in **`W`**. In the case of the skyline, 12 bits are not needed and as such, only the **`W`** register is important when reading the value returned by **`IREAD`**.

Drawing the Background

Now that we understand that data structure behind the skyline, let's talk about drawing it. To draw each pixel of the background, the following steps are taken:

- Determine the current color in the background gradient using the scanline counter.
- Determine whether or not the pixel is above or below the skyline at that location along the X-axis.
- If the pixel is above the skyline, draw it using the sky background color.
- If the pixel is below the skyline, draw it using the skyline color.

In the actual implementation, each scanline is drawn in a single loop. The sky color for that scanline is determined outside of that loop, saving each pixel from having to re-calculate the color itself. Here is the code:

```
        ; 44 scanlines of lower sky are left after above loop
draw_sky_loop_l

        ; **** SET UP SCANLINE

        CALL    @Call_Start_Scanline

        ; **** DRAW SCANLINE

        ; **** Set up the scanline

        ; Put the color of the current sky scanline in <t1>
        MOV     t0, #64                        ; (2) Base the gradient on the
                                               ;     scanline counter
        SUB     t0, scanline                   ; (2)
        CLC                                    ; (1) Divide by 8 for 8 sky
                                               ;     color shades
        RR      t0                             ; (1)
        CLC                                    ; (1)
        RR      t0                             ; (1)
        CLC                                    ; (1)
        RR      t0                             ; (1)
        ADD     t0, #2                         ; (2) Increase the brightness
        MOV     t1, #COLOR_SKY_BASE            ; (2) Set the sky's base color
        ADD     t1, t0                          ; (2) Modulate the sky intensity

        ; **** Render each pixel of the scanline

        MOV     t0, #182                ; (2) <t0> is the pixel counter
        MOV     t3, #skyline                    ; (2) Start at the base of the
                                                ;     table base address
        ADD     t3, bg_scroll                   ; (2) Add background scroll
                                                ;     offset
sky_pixel_loop_l

        ; Put the next city Y pixel location in <t2>
        MOV     M, #skyline >> 8        ; (1) Point M:W at the table
        MOV     W, t3                           ; (1) Complete the address
        IREAD                                   ; (4) Read the table value
        MOV     t2, W                           ; (1) Put the value in <t2>
        INC     t3                              ; (1) Move to the next pixel

        ; If the current pixel is above the skyline, draw sky; otherwise,
        ; draw the city color
        CJA     scanline, t2, sky_pixel_l ; (4/6)
        NOP                                     ; (1) Pad this branch of the
                                                ;     conditional
        NOP                                     ; (1) Pad this branch of the
                                                ;     conditional
        MOV     VIDEO_REG, #COLOR_CITY & %00001111; (2) Draw a building color pixel
        JMP     pixel_done_l                    ; (3)
sky_pixel_l
        AND     t1,  #%00001111
        MOV     VIDEO_REG, t1                   ; (2) Render the sky
        ;JMP    $ + 1                           ; (3) Pad the jump above
        NOP
pixel_done_l

        DJNZ    t0, sky_pixel_loop_l            ; (2/4) Next pixel

        DJNZ    scanline, draw_sky_loop_l ; (2/4) Next scanline
```

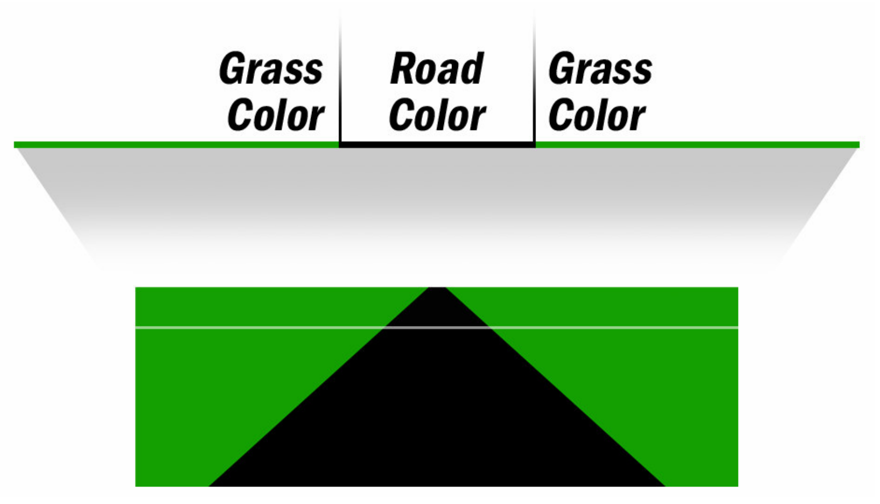
Drawing the Race Track

The second major visual element of the racing engine demo is of course the race track itself. Like the skyline, the track consists of many large, solid elements that would take massive amounts of memory if they were somehow based on raw bitmaps.

A Procedural Race Track

A solution similar to the height map used in the last section once again comes to the rescue. If you look at the track, you can see that like the skyline, it can be defined rather easily using only a couple of data points, rather than a complete 2D bitmap to cover the entire region of the screen. This time, break the track up into horizontal strips and analyze them as in Figure 3.

Figure 3 – Analyzing horizontal strips of the racetrack.



Notice that only two color changes are necessary to draw a black racetrack on a grass background. The beginning of the scanline is always the grass color, which is held until the left edge of the racetrack strip is reached, at which point the color switches to black. Black is then held until the right edge of the track strip is reached, at which point the color switches back to the grass color until the end of the scanline. It would not be difficult to expand this process slightly to include two extra data points so the dark and light stripes along the track can be drawn as well. The technique is the same either way.

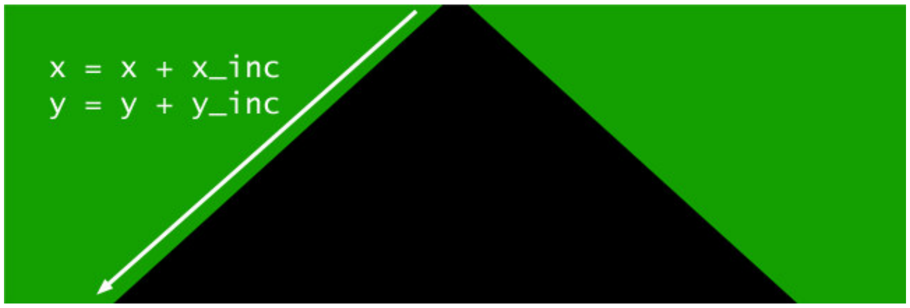
The most important lesson to learn here is the concept of using ***procedural*** graphics instead of bitmaps. Since only the smallest bitmaps can fit economically within the limited address space of the XGS ME (without using the off-chip 128K SRAM, of course), large graphics can usually only be done with clever alternatives that use code to ***generate*** an image based on a set of rules, rather than data to ***store*** it. In addition to the huge savings on memory, procedural graphics also often boast fluid movement and flexibility that would be difficult or impossible with standard bitmaps.

Adding Perspective to the Track

It is now clear that a convincing track can be drawn using just a few data points that cause color changes at the proper times along each scanline. The question now, is, where do these data points come from? The track must appear narrow near the top of the screen and wide near the bottom to give the illusion of a three-dimensional perspective. This means our data points must be close together near the top of the screen and move apart from each other as they approach the bottom.

We could perhaps use a lookup table as we did with the skyline, containing the slope of a diagonal line that matches the desired angle for the race track. This would work, but unlike the skyline, the data needed to trace the slope of a diagonal is not unpredictable and arbitrary like the shape of buildings. It is in fact so predictable that a table is not needed at all; all that is needed is a tracking variable that follows a line's slope as the scanlines are drawn. Figure 4 demonstrates this idea.

Figure 4 – Using tracking variables to follow the slope of a line as the track is drawn down the screen.



The algorithm is actually quite simple:

- Initialize a 16-bit fixed-point tracking variable near the center of the screen, representing the distant end of the track.
- At each scanline, draw the grass color from the left side of the screen to the left edge of the track. Use black to draw from the left edge of the track to the right edge. Use the grass color again to draw from the right edge of the track to the right side of the screen.
- After drawing a scanline, perform a 16-bit fixed-point subtraction on the tracking variable to slowly move it away from the center of the screen, allowing the black (road) portion of each scanline to broaden as if it were coming closer to the viewer. Using fixed-point math instead of a simple 8-bit value allows the any slope to be used easily rather than forcing the track to follow a 45 degree angle.

Two separate tracking variables, of course, are used in the actual program so both the track, as well as the dark and light border, can be drawn with reasonably correct perspective.

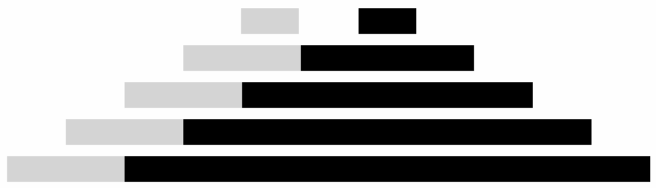
Making Turns

Equally important to the perspective of the track is giving it the ability to bend as if its path was curving. This technique is actually rather simple, and simply requires the use of yet another lookup table to apply a predefined curve to the track scanlines as they're drawn from top to bottom.

Deforming the Racetrack

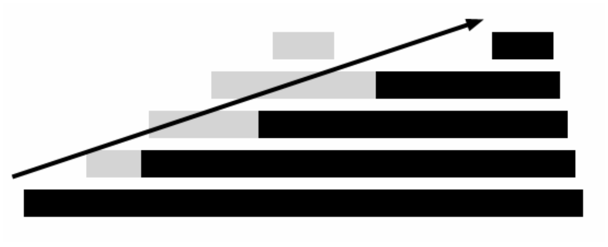
To understand this technique, imagine if a specific value were added to every data point on each scanline as they were drawn. If the value was positive, it would shift the entire track towards the right side of the screen. If it were negative, the track would shift to the left. This effect is illustrated in Figure 5.

Figure 5 – Shifting the track left or right by adding an unchanging value to each scanline's data points.



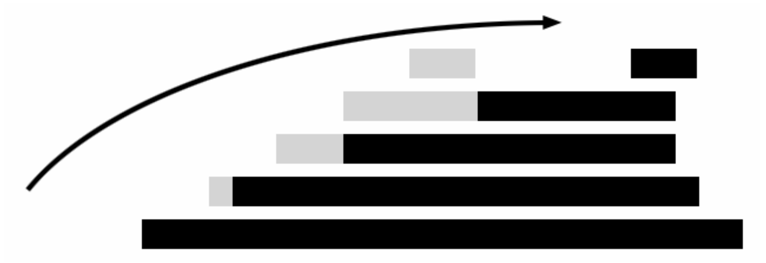
Now, imagine that another tracking variable was maintained and added to the location of every data point as each scanline was drawn, instead of the unchanging value in the last example. If this tracking variable increased after each scanline, the road would appear to veer off at an angle, as shown in Figure 6.

Figure 6 – Bending the track along a diagonal by adding an increasing value to each scanline's data points



Finally, imagine that instead of increasing the new tracking variable by an unchanging value at each scanline, thus bending the track linearly, the tracking variable instead increased in some non-linear fashion, perhaps exponentially. This would cause the track to curve, because the rate of increase would be different at the top of the track than at the bottom. Figure 7 illustrates this.

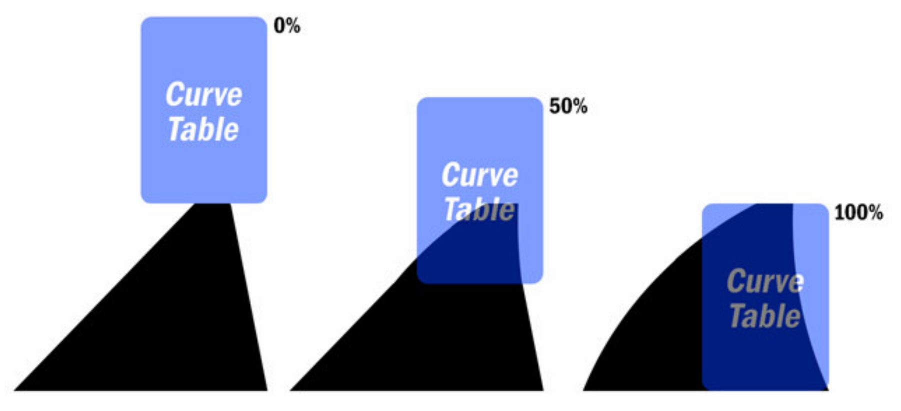
Figure 7 – Curving the track by adding a non-linear tracking variable to each scanline's data points.



The trick is to step through a curve and apply it to the scanlines as they're drawn in order to shift the track towards the desired shape. The only question remaining is how to gradually transition from a straight track to a track that moves in one direction or the other. Since the track can't suddenly switch from perfectly straight to completely curved, there needs to be a way for the curve to be applied gradually.

Figure 8 presents a solution. Imagine that the curve data is stored in a lookup table such that each index in the table corresponds to one scanline on the track. If the entire table were applied such that each scanline were affected, the entire track would warp into a full curve shape. Now, if only the first half of the table were applied to the track, thus affecting only the top half of the road scanlines, the curve would be half as apparent and appear to be halfway down the road.

Figure 8 – "Raising" and "lowering" a curve table into and out of the scanlines to control the perceived distance of the curve.



In other words, by "lowering" and "raising" the table into and out of the track scanlines, the curve shape can appear to be moving towards or away from the player, just like a real turn on an actual road.

Generating the Curve Data

As for the curve data itself, the easiest approach is simply to sample the appropriate segment of a sine wave. A C program was used to generate and format the following table based on the standard library's **`sin()`** function. The result was then simply copied and pasted into the racer demo source code.

```
curve_table

        ; Left turn curve
        DW      $E3, $E4, $E5, $E6, $E6, $E7, $E8, $E9
        DW      $EA, $EA, $EB, $EC, $ED, $EE, $EE, $EF
        DW      $F0, $F0, $F1, $F2, $F3, $F3, $F4, $F5
        DW      $F5, $F6, $F6, $F7, $F8, $F8, $F9, $F9
        DW      $FA, $FA, $FB, $FB, $FC, $FC, $FD, $FD
        DW      $FD, $FE, $FE, $FE, $FE, $FE, $FE, $FE
        DW      $FE, $FE, $FE, $FE, $FE, $FF, $FF, $FF
        DW      $FF, $FF, $FF, $FF, $FF, $FF, $00, $00

        ; Right turn curve
        DW      $1D, $1C, $1B, $1A, $1A, $19, $18, $17
        DW      $16, $16, $15, $14, $13, $12, $12, $11
        DW      $10, $10, $0F, $0E, $0D, $0D, $0C, $0B
        DW      $0B, $0A, $0A, $09, $08, $08, $07, $07
        DW      $06, $06, $05, $05, $04, $04, $03, $03
        DW      $03, $02, $02, $02, $02, $02, $02, $02
        DW      $02, $02, $02, $02, $02, $01, $01, $01
        DW      $01, $01, $01, $01, $01, $01, $00, $00
```

Note that the table is split into two blocks. The first block is the curve used to form left turns, while the second is for right turns.

Summary

All told, the there were a number of main steps in implementing the track. The first was approaching the track in a manner similar to the skyline—by recognizing that the track is really just defined by lines that mark the left and right sides of the track and the striped track border. Next, understanding how these lines can be made diagonal to produce a perspective effect for the track. Lastly, a curve table was applied to the track to varying degrees to create a flexible and easily controlled turn effect that works in both directions.

Coupled with the skyline described in the previous section, the racing engine demo creates detailed, full screen graphics without the huge memory overhead usually associated with such results. This is the trick to writing graphical programs on the XGS PE—figuring out ways to produce complex and meaningful graphic effects, images and animations without resorting to bitmaps or other large data structures.

Player Input

Player input is handled in a subroutine called **`Handle_Input()`**:

```
Handle_Input

        ; **** READ THE JOYSTICK ***********************************************

        CALL     @Call_Read_Joystick

        ; **** HANDLE STEERING *************************************************

        ; Don't allow turning if the player isn't moving
        BANK     BANK_GAME
        MOV      t0, speed
        BANK     BANK_MAIN
        CLC
        CSA      t0, #0
        JMP      :skip_steering

        ; Turn left
        SNB      data8.2
        JMP      :skip_steer_left
        BANK     BANK_GAME
        CLC
        CSA      wheel_angle, #1
        JMP      :skip_steer_left
        STC
        SUB      wheel_angle, #2
:skip_steer_left
        BANK     BANK_MAIN

        ; Turn right
        SNB      data8.3
        JMP      :skip_steer_right
        BANK     BANK_GAME
        STC
        CSB      wheel_angle, #254
        JMP      :skip_steer_right
        CLC
        ADD      wheel_angle, #2
:skip_steer_right
        BANK     BANK_MAIN

:skip_steering

        ; **** HANDLE GAS PEDAL ************************************************

        ; Speed up
        SNB      data8.0
        JMP      :skip_speed_up
        BANK     BANK_GAME
        CJE      speed, #99, :skip_speed_up
        INC      speed
:skip_speed_up
        BANK     BANK_MAIN

        ; Slow down
        SNB      data8.1
        JMP      :skip_speed_down
        BANK     BANK_GAME
        CJE      speed, #0, :skip_speed_down
        DEC      speed
:skip_speed_down
        BANK     BANK_MAIN

        RETP

End_Handle_Input
```

This fairly simple subroutine starts by calling **`Read_Joystick()`** to put the joystick status in the **`data8`** global. **`Read_Joystick()`** takes the place of the original, complex joystick reading function from the XGS ME version of this demo. Instead of reading an entire packet of data from the shift register, the lower 5 bits of the **`RB`** port now is now a direct connection to the joystick's output lines. This makes the process of checking any stick direction or the status of the fire button as easy as determining the status of a specific bit. Within the `RB` register, Table 2 below lists each bit's meaning:

Table 2 – Bit encodings for the joystick.

`RB` *Register Bit*	*Description*
0	Stick Up
1	Stick Down
2	Stick Left
3	Stick Right
4	Fire Button

Remember also that these bits are ***active low***, meaning they are set to zero when their corresponding button is pressed.

While the subroutine is very straightforward, there are a couple notes worth mentioning. First, turns cannot be made unless the user is moving. Before checking the status of the "steering" bits (left and right joystick directions), the speed is checked. If it is zero, the steering bits are ignored. Second, when the user changes the speed with the up and down joystick directions, the `speed` variable is incremented and decremented

Expanding the Demo

While the demo produces a nice effect, it's not particularly entertaining. A considerable amount of work would have to be done in order to create a true game, but here's an outline of the three most pressing omissions as of now:

- **Cars -** Needless to say, the most blatant missing elements are actual cars to drive. While you might be able to get away with pretending the player is in a first-person perspective from which his own car is not visible, there's still no excuse for not having competitors driving around. The biggest hurdle here is simply drawing a sprite of any reasonable complexity over the road, considering how dense the logic already is for drawing the road itself. Remember that in order to introduce new elements, especially those that overlap with existing ones, a considerable amount of logic must be added.
- **A Real Track -** Currently, you "design" the track as you drive along it by controlling when it turns and bends. A real track map would not be difficult to implement; as long as your location within the track (which must be thought of as one-dimensional) is known, you can determine how close you are from the next turn and, depending on the distance and the direction of the turn, "lower" the proper curve table into the track scanlines.
- **A Proper Game Interface -**Last but most certainly not least, a true game interface will complete the program. Think title screens, maybe even option screens, and perhaps screens for selecting cars or tracks.

Summary

This case study has been a more conceptual, algorithmic discussion rather than an annotated source code dump. The complete racer engine demo is a fairly complex program and while it's certainly not beyond the grasp of most intermediate programmers, it's a lot of detail to stuff into one small tutorial. So, instead of bogging you down with every conceivable detail, the purpose has been to give you an informative overview of what you can expect when writing more sophisticated programs, especially those that approach the level of a full game.

There are still numerous aspects of the demo that were not covered, such as the dark and light stripes that move along the road, the way the buildings scroll, and some other details, but this case study was not meant to be exhaustive program-wide documentation. Aside from the main eBook, ***Design Your Own Video Game Console***, of course, the next step in expanding your XGS PE skills would be to take a look at the racer engine demo source code itself, where all of the remaining details holding these concepts together come into view.