Pong on the XGS PIC

Videogames now transcend generations, each becoming seemingly more entrenched in this subculture. Presenting unique challenges to programmers across every platform, from the Atari 2600 equipped with only 128 bytes to the impossibility of scrolling with VGA, programmers required ingenuity to combat the hardware-imposed constraints of each more powerful system.

To understand the art of writing videogames, the aspiring programmer must simply write them. One game stands alone in its simplicity and its recognizability across every generation of gamer: Pong.

The first part of the document will remain universal for every programming platform, simply motivating the various pieces of the game bit by bit. Logically, the next section will be primarily for the XGS PIC created by Nurve Networks. As of this writing, a version of Pong has been posted on the XGameStation forums but was written in XGS Basic. This document will focus on utilizing the C-based drivers, but the actual execution should remain similar across multiple platforms.

Game Constructs

Graphics Engine

A basic version of Pong will have few graphical objects on the screen. The graphics engine draws, at minimum, three objects, in two colors: two paddles and a single ball. Drawing the objects is a matter of plotting pixels on the screen. Geometrically, the graphics engine can draw three rectangles on the screen to satisfy these minimum requirements. Modifying the program to draw the ball as a circle rather than a square is left as an exercise. The choice of colors for the background and the graphical objects is unimportant as long as the background and the objects are different, contrasting colors. Assuming the drivers can only plot pixels, the easiest way to make a rectangle is to utilize loops while iterating over the area of the desired rectangle. Plotting every pixel in a row until the number of desired columns is reached then moving onto the next row is likely the most intuitive way to draw a filled-in rectangle. When deciding lengths and widths for the ball and the paddles, good programming practice would suggest using constants. While fine-tuning the look, changing the values only once is much preferable to having to hunt everywhere in the program where those values are used. The obvious brute-force approach to refreshing the screen works on modern computers quite nicely without any flicker. When beginning, this approach will be used for simplicity and then discuss ways to improve the methodology to achieve faster results. When refreshing the screen, the general algorithm will be to set every pixel to the background color and then draw the objects in their respective locations with a different color on top of the recently reset background. In the optimization section, it will be shown that this algorithm is not the best way to do things, but tolerating the inefficiencies for now will result in an easier understanding of how the game as a whole is working.

Placement of the objects on the screen will require some custom terminology and pictures to describe effectively. This implementation of Pong will be faithful to some of the earliest incarnations with regard to the placement of the paddles. The paddles will

move vertically at a fixed point on the screen, rather than moving laterally at a fixed vertical point. The actual decision is fairly arbitrary, both versions will work in very similar fashions, but this will affect some game logic later. The screen of play will refer to the bounded portion of the screen where all of the action takes place. The pit is the location between the left or right edge of the screen of play and the fixed x-coordinate where the paddle will move up or down. The paddle height and paddle width are self-explanatory, but, with this implementation, the paddle height should be larger than the paddle width.

To describe the location of the game objects within the screen of play, the program does not need to know where every pixel is in the area of the object. Picking an arbitrary point will allow calculations to be used to get every other pixel in the object. There are a few obvious choices for the point, but using the top left corner will be the convention used for this program.

Game Logic

With the graphical engine finished at the basic level, the last piece of the program left to motivate is the game logic. There are exactly four pieces that will each be discussed in turn: Player input, ball movement, collision detection, and artificial intelligence (AI).

Player Input

Player interaction is perhaps the most important aspect of every game. This salient piece allows the player to influence the course of events, transforming a graphics demo into a fully fledged game. How the input is actually driven, whether by gamepad, keyboard, or some other construct, is not particularly important. The more interesting question is what the input actually does to influence game world.

This seemingly long exposition leads to a very anti-climactic and obvious conclusion: one action will move the paddle up and a different action will move the paddle down.

Ball Movement

The game must autonomously control the ball in the screen of play. The ball does not need any sort of AI to control its movement, however, as the ball should move in a predictable manner. Movement will be controlled by predefined states, as a ball can move up or down and left or right. Combinatorially, the ball now has four possible states:

1. Up and Left
2. Up and Right
3. Down and Left
4. Down and Right

With these states defined as such, the ball will always move in a diagonal manner around the screen of play.

Collision Detection

The ball now has a way to move in a predictable manner, but how does the ball transition between the various states? Quite logically, certain game events will dictate the transitions, but the next step is to define when the changes happen and to exactly which state. With this design of Pong, what should happen when the ball hits the top, and, analogously, bottom of the screen of play? Breaking this down further, to hit the top of the screen, the ball must be moving upward. To simulate a bouncing effect, the ball needs to shift to the downward direction. When the various ball states were described above, the ball has two downward states: either left or right.

Analysis must now continue in a case by case basis. Therefore, assume the ball is moving upward and left just before the collision. Moving downward is the next logical move, but should it continue moving left or switch to moving right? If the ball were to transition to a right-moving state, the ball would reverse its direction along the diagonal that it came from, never moving to the other side of the screen. The only way for it to reach the other side of the screen of play is to continue moving in a leftward direction. A similar situation holds when the ball is moving upward and right: continue moving to the right and switch to a downward direction. The downward cases are handled symmetrically.

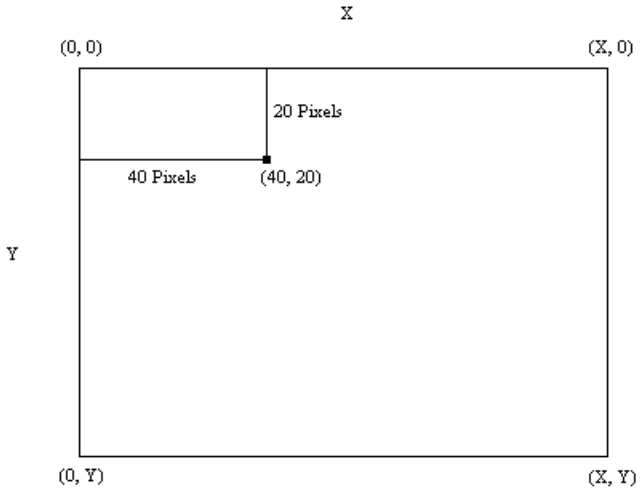
Exactly when do these transition changes happen? This will involve the definition of a collision: two objects at some state of intersection are said to collide. In this game's case, having the objects actually intersect with one another will lead to graphical anomalies; therefore, the exact state of intersection will be when the objects are tangent to one another, or simply touching but not overlapping.

A discussion of the coordinate plane is now in order. In many languages, plotting pixels works in a similar fashion. Each unit in an ordered pair is a pixel. with the x-coordinate measuring the amount of pixels from the left bound of the screen and the y-coordinate measuring the distance from the top bound of the screen. Let the current resolution be $X \times Y$, where $X,\ Y \geq 0$. X describes the maximum width of the screen while Y denotes the height of the screen. The top-left corner of the screen is (0, 0), while the bottom-right corner is (X, Y). To plot a point 40 pixels left and 20 pixels down, use the ordered pair (40, 20). Therefore, the top portion of the screen can be described as (x, 0), while the bottom portion of the screen can be denoted (x, Y), where $0 \leq x \leq X$ (see Figure 1).

It turns out that (x, 0) and (x, Y) are the collision points within the screen of play. For the former, the topmost portion of the ball will be tangent to the top edge of the screen of play, and the ball needs to change to the correct state before actually intersecting with the top edge. Since all of our game objects are described with top left corners, each game object will be represented in data with some sort of coordinate pair. Then the bottom edge case works similarly but with one slight problem. If the ball's location is ever equal to (x, Y), then it must already be intersecting the bottom edge of the screen of play. Since the ball is a square with a constant length s, the exact tangent point can now easily be calculated. By subtracting s from Y, the height of the screen yields the point where the ball's bottom edge is tangent to the bottom edge of the screen of play.

Representing this in coordinate pair form, every possible collision point can be represented by (x, $Y - s$).

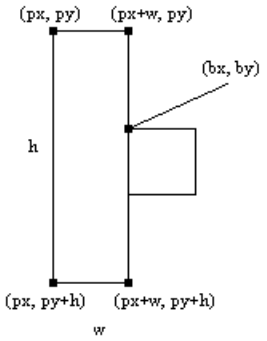
A similar set of collisions arises from the case when the ball and paddle don't collide; in essence, the scoring condition. Using the definition of the coordinate plane, (0, y) and (X, y) must be the left and right bounds of the screen of play, where $0 \leq y \leq Y$. Since top-left corners of objects are used to define their location, the definition of the left edge, (0, y), can be used to find the tangent points. Just like when finding collision points for the top and bottom edges, one case utilized the definition of an edge but an offset needed to be used to find the exact tangent point. If the ball's location were to equal (X, y), the ball would have already intersected with the right edge. Since the ball has a width of s, (X − s, y) yields all of the tangent points with the right edge of the screen of play.



The Coordinate Plane
Figure 1

All of the screen edge collisions have now been explicitly handled. The last two cases are slightly more difficult, as this is when the ball collides with the paddles on the left and right sides of the screen. The calculation is similar to when working with the wall-like top and bottom edges of the screen of play, but the wall is now bounded, or has a very specific length. Where can the ball actually collide with a paddle?

Let h denote the height of the paddle and w describe the width in pixels. Suppose that (p_x, p_y) is the location of the paddle and (b_x, b_y) describe the location of the ball. (Remember, these are the top left corners.) To find the collision point, the right edge of the paddle needs to be used. Since the width of the paddle is a constant, ($p_x + w$, p_y) will yield the top-right corner of the paddle. This is now the upper bound of the possible collision points with the paddle. Finding the lower right corner will yield the lower bound, and thus create all of the possible points where the ball can collide with the paddle. Since the height of the paddle is a constant, the lower-right corner can be found by combining the right edge calculation and h, yielding ($p_x + w$, $p_y + h$). To actually check for the collision, compare the location (b_x, b_y) with ($p_x + w$, $p_y + h$). So when $b_x = p_x + w$ and $p_y \leq b_y \leq p_y + h$, a collision occurs (see Figure 2).



Anatomy of a Pong Paddle and Collision
Figure 2

The right paddle has a symmetric issue with the calculations. Since the top-left corner is being used to represent locations, the left edge is practically given, so all points between (p_x, p_y) and $(p_x, p_y + h)$ are the possible tangent points. But now the right edge of the ball is needed to find the collision. Since s is the length of a side of the ball, $b_x + s$ yields the top right corner of the ball. The reasoning above can now work, so the collision occurs exactly when $b_x + s = p_x$ and $p_y \leq b_y \leq p_y + h$.

One last thing left to discuss: What should happen when the ball collides with the paddles? If the ball collides with the paddle, then the horizontal state needs to switch, otherwise the ball would go through the paddle. The next obvious question involves the vertical state of the ball. Suppose the ball is going up; then if the ball were to flip its vertical state, or go down, then the ball would return along its same diagonal path since the ball flips its horizontal state. Therefore, the vertical state should be the same.

An interesting trend can now be noticed between the relationships with collisions and the ball states. If the ball collides with the top or bottom edge, flip the vertical state. If the ball collides with one of the paddles, a left or right edge as it were, flip the horizontal state.

Artificial Intelligence (AI)

At this point, all of the logic so far can be made into a complete two-player game. However, it would be nice to allow for a single-player mode (and be an interesting programming exercise). The AI does not need to be foolproof but act intelligently enough to give the player a challenge to remain entertaining. The current problem rests in the definition of acting intelligently.

Referring to the notation above, let (b_x, b_y) be the ball's current location and (p_x, p_y) denote the location of the computer player's paddle. The collision detection code will handle the computation of collisions, but the AI needs to move the paddle in such a way that the paddle will generate a collision with the ball. Therefore, worrying about b_x and p_x would be extraneous. Because of the simplicity of Pong, there are few cases to analyze.

Suppose that $p_y > b_y$. What does this mean in actuality with regard to the screen of play? The top-left corner of the paddle is lower than the top-left corner of the ball on the screen. The next problem is defining the optimal strategy for the computer player. Typically, the best strategy for the computer player is analogous to a human player's

strategy. The best way to close the vertical distance between the ball and paddle is to move the paddle up, or decrement p_y.

Logically, there is one case left to handle: when $p_y \leq b_y$. In the screen of play, this case will happen if and only if the paddle's location is higher than the ball's location. These two cases for the AI mirror each other in not only the geometric qualities, but also their solution. As a player, the optimal strategy in this situation is to move the paddle downward, or increment p_y.

The algorithm can now more intuitively be described. With regard to the *y*-coordinates of the location, if the ball is above the paddle, then the paddle will move upward. The converse holds as well. But is this the optimal strategy? Something hidden lies based on arbitrary definitions created earlier to allow for easy graphical algorithms. All locations were defined to be the top-left corner of the object. Therefore, the current algorithm makes the attempt to line up the top left-corners of the paddle and the ball. Not necessarily a bad strategy, but it might not give the player an appearance of intelligence.

The logic is sound, but the comparison points are not the best choice. Any human player implicitly tries to maximize the amount of colliding points between the ball and paddle. The best location on the paddle for the ball to collide with is actually fairly intuitive: the center. Leaving the maximum amount of paddle space on both sides of a colliding ball allows for a very good error tolerance when playing. To adjust the algorithm, simply compare the midpoints of the ball and paddle to give the AI the appearance of a more intelligent player. By letting h be the height of the paddle, then $(p_x, p_y + h/2)$ yields the center of the paddle. This calculation works similarly for the ball.

Program Structure

All of the game constructs – the graphics engine, player input, collision detection, and AI – are now defined. The next step is to tie everything together into one program.

Initialization

The step of initialization is fairly nebulous at this level of abstraction, but it is still important to discuss since all the major steps are here. The obvious first step is to set up the graphical interface that will run the engine. If there are any other drivers or interfaces that need to be initialized, such as player input or sound, this is also the place to set those up.

The next major step is to create and assign starting values to all of the variables that will be used throughout the program (not temporary loop counters and their ilk). The variables actually needed are the location of all the objects, defined by their top-left corner. So, for every object, two variables are needed to account for x- and y-coordinates. Something hidden by the design is the fact that the ball not only has location, but also states as well. Since the vertical and horizontal states are independent of each other, two variables must be used to account for the ball's state of movement.

The final step is to initialize the very first frame of the screen of play. The first step is to create the background of the screen. Using the starting values for all of the location variables, plot all of the pixels needed to represent the objects.

Game Loop

The most important part of every game, the game loop's timing, needs to be immaculate on many of the older systems and architectures. The Atari 2600, for example, had 76 cycles of the processor to do all of the game logic, which includes sound as well. The processors of today are much faster, and timing is only a real consideration for graphics pushing the hardware envelope.

For the purposes of this discussion, every iteration of the game loop will correspond to the drawing of a single frame and is timed to run at approximately 30 frames per second. The game loop needs to handle everything, every iteration, but stay within the limits of the television or monitor's VSYNC. The VSYNC is the longest period of time that the monitor or television is idle and not in the process of redrawing the screen. Trying to write to video memory while the monitor or television is trying to read will lead to hardware contingencies, screen tearing, or other graphical anomalies.

What is the first thing the game loop needs to handle? Suppose, for a moment, that all of the game constructs are divided into the game logic and the graphics engine. By putting the graphics code first, and then handling, say, player input, a problem arises. The problem arises from the initialization stage. Essentially, the graphics engine will have computed the first frame before the loop actually starts and then draw the first frame again during the very first iteration of the game loop. Now, the game logic is computed, but the current frame in the loop is desynced with the calculations done with the game logic.

Therefore, the game logic should be ordered first, but that leaves a few constructs to order. Suppose that the arbitrary decision is made that the collision detection should be handled first. Then, logically, the player input is handled sometime after that in the loop. Essentially, the collision is calculated, and then the player gets to move. A contradiction can be found in at least one case, but suppose that $p_y + h = b_y + 1$, and the player is frantically trying to move downward to compensate. By the definition of the collision detector, no collision happens in this case. The player's movement was in vain, even though there would be the appearance from the graphics engine that the ball should have bounced. Taking the contradiction further, the ball would then even look like it intersected with the paddle if the player continued to move downward. This would be a frustrating situation for the player. Therefore, the input should be handled sometime before collision detection.

The AI falls into a similar situation as above. If collision detection is handled before the AI moves, that leads to the same problem. Therefore, the collision detection must happen after both the AI and player input. The order with regards to the AI and player input does not matter; the movement of one is independent of the other.

But, when should the ball move? Keep in mind that the graphics are being updated last; therefore, the ball location currently displayed by the television or monitor is the last iteration. If the program were to use this position to check for collisions, then the same problem arises. Therefore, the ball's location needs to be updated first.

The final ordering is:

1. Ball Movement

2. Player Input
3. AI
4. Collision Detection
5. Updating the Screen of Play

Implementation Details

It is now time to put the theory into practice. The implementation details diverge from the theory with regard to efficiency. In some cases, the programmer can get away with only doing a few comparisons but implicitly checking everything required. Likewise, there are some things that the theory did not account for in the initial analysis.

Paddle Movement

The theoretical analysis of both the player input and AI left out a key, but intuitive, notion: The paddles should not move off-screen. The collision detection ensured the ball would not move outside the screen of play, but the next step is to ensure that the player and AI cannot move outside this same boundary. By asserting that the paddles stay within the bounds of the screen, player frustration is kept to a minimum and there is no attempt to draw objects outside the visible screen. Graphics drivers may not explicitly check that the location is a valid value, so the graphics functions may try to write to memory outside the video buffer.

The solution is the same for both the player and AI: Before attempting to move the paddle, simply assert that the paddle is not at a certain edge of the screen. Since the paddles can only move vertically, there are only two cases. Suppose that the paddle is attempting to move upward. Before modifying the logical location, check the $y-$coordinate. If this is 0, then the paddle is currently tangent to the top of the screen and should not move upward any more.

That leaves the case where the paddle is moving downward. Since the paddle height is a constant, h, then the lowest point for the top-left corner must be $(x, Y - h)$, where x is the fixed horizontal point for the paddle and Y is the screen height. To go any lower would mean the paddle is not just tangent, but intersecting, with the bottom edge of the screen.

Collision Detection

The collision detection is by far the most complex component discussed. When working in 2-dimensional space, the implementation is actually a little simpler. Rather than have a loop check every possible tangent point on both objects or edges, checking at most two points will suffice in every case.

Focusing on the top and bottom edges, $(x, 0)$ and $(x, Y - s)$ are the only possible tangent points, where $0 \leq x \leq X$. Remember that every object is represented logically by an ordered pair that is its top-left corner. If the ball's location is (b_x, b_y) and if $b_y = 0$, then the ball is tangent to the top edge of the screen. This required only one comparison to figure out when to change to the downward state. Also, if $b_y = Y - s$, then the ball needs to also change its state to moving upward.

The left and right edges are equally simple. Recall that $(0, y)$ and $(X - s, y)$ are the possible tangent points, where $0 \leq y \leq Y$. The next step is to compare b_x to 0 and $X - s$. If b_x is equal to either of these values, then the scoring condition has occurred. Likely, the ball will be reset so as to continue play.

The final set of cases, when the ball collides with the paddle, is only slightly more difficult. Let (p_x, p_y) be the current location of the paddle. Focusing on the left paddle, the first thing to check is the correct $x-$coordinates. For the left paddle, all of the possible vertical tangent points lie on $(p_x + w, y)$, where $0 \leq y \leq Y$. Logically, if $p_x + w = b_x$, then a collision might be occurring. There are a few other points to check.

The next step is to compare the $y-$coordinates. Graphically, a collision should occur exactly when any pixel of the ball is between the top and bottom edges of the paddle. From the previous section on collision detection, all of the possible tangent points can be described as all points between $(p_x + w, p_y)$ and $(p_x + w, p_y + h)$ if h is the height of the paddle. At this point, it is possible to simplify the algorithm by comparing only two points. If the top-left corner or the bottom-left corner is tangent to the paddle, then there must exist at least one point of the ball that visually looks like it is colliding with the paddle. By checking these two points, the algorithm checks all points on the left side of the ball implicitly. The right paddle works in a similar fashion, except the points to check are the top- and bottom-right corners of the ball, found by $(b_x + s, b_y)$ and $(b_x + s, b_y + s)$.

When implementing collision detection, the order of comparisons matters. Calculations, and, therefore. time, can be saved by checking the salient points in a certain order. Take, for instance, the left paddle. Suppose that the $y-$coordinates were checked first in the algorithm. By nature, this is a complicated expression in many programming languages. After this expression is checked, the next step would be to compare the $x-$coordinates. During runtime, it might be likely that the player is following the ball closely along the vertical plane, so the first comparison will be true at least a few times during play, even when the ball is on the other side of the screen. The $x-$coordinate expression, however, will be true exactly once for quite a few frames. If the comparisons were flipped, the costly $y-$coordinate calculation can be saved to be used only exactly when needed, since it might happen to be true a few times when it is not needed.

Therefore, the collision detection for the paddles will look something like this:

1. Check if $b_x = p_x$
2. Check if $b_y \leq p_y$ and $b_y \leq p_y + h$, or if $b_y + s \leq p_y$ and $b_y + s \leq p_y + h$
3. Change ball's state

Each subsequent step must be true to continue on to the next one. The screen edge collisions work exactly as described, since there is only a single point to check.

At this point, the discussion of theory is complete. I highly encourage the reader to try to implement this on their own before viewing the next section. By implementing it yourself first, a greater understanding will be achieved while viewing someone else's implementation. Should you get stuck, feel free to glance at the relevant section for help. Due to the nature of the XGS PIC, the syntax is a variant of C and easily transferable to many other programming languages.

Implementation on the XGS PIC

A few notes before beginning. I am currently using MPLab v. 8.15a and the v. 1.0 (file designation V010) drivers written by Joshua Hintze. I will be focused primarily on writing Pong with the NTSC drivers, but the NTSC and VGA drivers have identical function calls. The main differences lie in the included source files and constant names. Here is a quick overview of the files Pong will utilize:

1. XGS_PIC_SYSTEM_V010.h
2. XGS_PIC_SYSTEM_V010.c
3. XGS_PIC_GFX_DRV_V010.h
4. XGS_PIC_GFX_DRV_V010.h
5. XGS_PIC_NTSC_160_192_2_V010.h
6. XGS_PIC_NTSC_160_192_2_V010.s
7. XGS_PIC_GAMEPAD_DRV_V010.h
8. XGS_PIC_GAMEPAD_DRV_V010.c
9. XGS_PIC_SOUND_DRV_V010.h
10. XGS_PIC_SOUND_DRV_V010.c
11. p24HJ256GP206.gld

The first two files will configure the clock rate of the processor, allowing the PIC to run at an exact multiple of the NTSC frequency. Precise timings are driven by the need to produce graphics on the TV screen. If these timings are off, then the graphics will get distorted in some way. Typically, these distortions manifest themselves from flickering to screen tearing. If every instruction has a cycle count, then these timings will only help if the total sum of all instructions is less than or equal to the VSYNC time of the TV screen. Otherwise, it is likely that the graphics will become garbled. Fortunately, this program is simple enough to not run into this problem.

The next two files contain functions integral to the graphics engine. These drivers provide functions to modify the video memory. The two files directly after those provide the actual assembly language code that will draw the image onto the TV screen from the video memory. These files also set the screen resolution to 160 × 192 pixels with a 2-bit representation for each pixel in memory. This allows up to four ($2^2 = 4$) colors to be used for each pixel. Also, many constant values are provided in the header file to ease color generation.

The final set of files is non-graphical input and output. Files 7 and 8 provide functions to read the button presses from the gamepad on either input port on the XGS PIC. The last two files provide helper functions to output sound.

The last file is the linker script. This file is essential to the compiler, as the linker script defines all of the registers and memory areas specific to the processor for the compiler.

Initialization

Before setting up any individual part of the game, all the drivers should be set up. The first thing that must be done for every program of this nature is to configure the

clock speed of the XGS PIC. The SYSTEM files provide a handy function for doing exactly that: the `SYS_ConfigureClock(. . .)` function. Sadly, this is the first place that the NTSC and VGA programs begin to diverge. The argument specifies whether to set the clock speed to an exact multiple of NTSC or VGA timings, with `FCY_NTSC` or `FCY_VGA`, respectively. Again, this must be the first line of the main function.

Configuring the sound and gamepad drivers for use is relatively simple. Both provide a single `Init()` function that will ensure correctness before making any of the other related function calls. Calling both `Gamepad_Init()` and `SND_Init()` sets up the gamepad and sound drivers.

The graphics drivers are slightly more complex to set up, requiring a few distinct steps. Like many of the other drivers, the first step is to call an `Init()` function that configures the drivers to the current parameters. The next step is to define global variables that will be used by the graphics drivers. There are three variables to create and initialize, the foremost being the section of memory that represents the next frame to draw, or the video buffer. The drivers require this to be represented as a one-dimensional array of unsigned char. To make this program as modular as possible, each of the different screen resolutions (and screen types) files provide a constant with the same name that defines the size of the video buffer. This handy constant has the name `VRAM_BUF_SIZE`. The declaration of the video buffer will look something like this:

```
unsigned char g_VRAMBuffer[VRAM_BUF_SIZE] attribute((far));
```

The attribute function will set the array's location into the correct place in memory.

Next, the drivers need the palette map and any palettes that will be used during the operation of the game. The palettes contain colors that will be used to draw a frame of the screen. Both of these are arrays with useful constants provided by the resolution setting file. The constants `PALETTE_MAP_SIZE` and `MIN_PALETTE_SIZE` will provide a palette map and palette with the correct sizes. The palette maps are used to provide indirection. Instead of having only four colors for the entire screen, a subset of the screen can be assigned a different palette. The palette that the map points to provides the actual color information. The type for the arrays is again unsigned char, so the declaration will look something like:

```
unsigned short g_PaletteMap[PALETTE_MAP_SIZE];
unsigned char g_Palettes[MIN_PALETTE_SIZE];
```

The graphics drivers setup is almost complete. The final step is to provide some initial values for the palette map and palette. Since this program only needs two colors, only one color palette is needed because each palette can provide at most 4 colors. Keeping with the tradition of black and white pong for now, the palette and palette map can be initialized with the `memset()` function. Every portion of the screen will only utilize two colors, so the first color palette is sufficient. Setting each index in palette map to the first palette (index 0) will map the entire screen to this palette.

The final step is to give the palette actual colors. The resolution file provides some more constants to utilize: the codes for the colors themselves. The `NTSC_BLACK` and `NTSC_WHITE` constants provide the exact colors needed. Using `memset()` will

provide reasonable values for all of the colors in the palette, even if they won't be used. Setting everything in the palette to NTSC_BLACK means that one of the other indices must be set to NTSC_WHITE manually. The initialization of the palette map and palette will look something like:

```
memset(g_PaletteMap, 0, sizeof(g_PaletteMap));
memset(g_Palettes, NTSC_BLACK, sizeof(g_Palettes));
g_Palettes[1] = NTSC_WHITE;
```

Like the sound and gamepad drivers, the graphics drivers have an initialization function GFX_InitBitmap(…) that needs to be invoked before calling any of the other functions, as it sets up pointers and variables in the other functions. The initialization function takes a few parameters: the screen height, width, the number of bits used to determine colors, and pointers to the video memory, the palettes, and palette map. Except for the three variables declared above, the resolution file provides constants for the screen height and width, as well as the number of bits per pixel with SCREEN_HEIGHT, SCREEN_WIDTH, and SCREEN_BPP. These constants have the same name in all of the resolution files, making the switch between modes and resolutions a snap.

Initialization of all the drivers is now complete. The current code should look something like:

```
#include "XGS_PIC_SYSTEM_V010.h"
#include "XGS_PIC_GAMEPAD_DRV_V010.h"
#include "XGS_PIC_GFX_DRV_V010.h"
#include "XGS_PIC_NTSC_160_192_2_V010.h"
#include "XGS_PIC_SOUND_DRV_V010.h"

// Global variables needed for the graphics drivers
unsigned char g_VRAMBuffer[VRAM_BUF_SIZE] __attribute((far));
unsigned short g_PaletteMap[PALETTE_MAP_SIZE];
unsigned char g_Palettes[MIN_PALETTE_SIZE];

int main(void) {
// INITIALIZATION
// Setting up the processor for graphics mode

SYS_ConfigureClock(FCY_NTSC);
// Setting up the graphics drivers

GFX_InitBitmap(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP,
g_VRAMBuffer, g_Palettes, g_PaletteMap);

memset(g_PaletteMap, 0, sizeof(g_PaletteMap));
memset(g_Palettes, NTSC_BLACK, sizeof(g_Palettes));
```

```
g_Palettes[1] = NTSC_WHITE;
// Setting up the gamepad drivers
Gamepad_Init();

// Setting up the sound drivers
SND_Init();

return 0;
}
```

Ball Demo

This section will construct a simple demo that will move a ball around the screen. The motivation behind this demo is to have testable code in a few distinct phases so debugging will be easier. The subsequent sections will add to the previous sections code, adding functionality until Pong is restored in its former glory on the XGS PIC. The simplest first step is to get a ball moving around the screen without any intervention.

The first thing to consider is the definition of the ball in context. The ball has a location defined by its upper-left corner. It was also defined to be a square. If the square has a side length of *s*, the obvious way to draw the ball is to plot pixels by row and then by column. Having a nested loop structure accomplishes this quite nicely, but one thing remains. What is the value of *s,* and how should it be represented? The side length should remain constant throughout the lifetime of the program, so it is safe to say that the value of *s* should only change at compile time. Therefore *s* should be a constant declared with the `#define` preprocessor directive, enabling modularity of code. Should *s* be used in multiple places in the code, changing it once will automatically modify for any other place, greatly enabling the ability to tweak the code. The constant *s* has a fairly unrecognizable name, so choosing something like `BALL` makes it more clear exactly what this constant is representing.

With a side length defined properly, the next step is to code exactly how the ball should be drawn. This leads to a similar question with the storage of *s*,: Where should this code be written? Having the game loop as uncluttered as possible is a huge perk to readability. A function with a nice name will enhance the readability of the game loop, and, of further benefit, if the same code must be used twice (and it will be), then no copying and pasting is required. Should the function have parameters?

This question is analogous to the first question: What would make the function the most modular? The ball changes location once per frame; therefore, the old coordinate pair representing the location may have nothing to do with the current one (consider when the ball is reset upon the scoring condition). Hence, having parameters that accept integers representing the *x* and *y* values of the location would be of great benefit. There is one further thing that could be added: a color index. Adding this has inherently little value at this point, as the color typically will only be the index with `NTSC_WHITE`, but tweaking the program to do "silly" things once the engine is complete eats up oodles of time with a friend. To draw a ball, there needs to be some notion of side length. During the course of the program, the side length will never change, so defining a constant with name `BALL` to represent some integer value gives an explicit side length

that can be used while drawing the ball and for the collision detection. The final function should look something like this:

```
void drawBall(unsigned x, unsigned y, unsigned colorIndex) {
      int i, j;
      for (i = x; i < x+BALL; i++)
            for (j = y; j < y+BALL; j++)
                  GFX_Plot_2BPP(i, j, colorIndex, g_VRAMBuffer);
}
```

Now that there is a way to draw a ball on the screen, how should the location be represented? The location can vary wildly between any two subsequent frames; hence the location should be stored somewhere in the main memory. Since only two values are needed to represent the location (*x* and *y* in the coordinate pair), an array would be overkill. Two appropriately-named, unsigned integers will be easier to understand. The difference between using signed and unsigned integers is slight, but, as the coordinates are always positive in this coordinate plane, using unsigned integers ensures that positive integers are always being used. The declarations should look something like this:

```
unsigned int ball_x;
unsigned int ball_y;
```

The ball needs a starting location. When a player scores or the system is first turned on, the ball should start in the middle of the screen and move towards one of the paddles. Thinking of the bigger picture for a moment, how many times will the ball be reset, or recentered, in the middle of the screen? The obvious answer is when the program first starts, the scoring conditions will also reset the ball. Using pointer variables, writing a function will greatly enhance the readability of the code. To center the ball, it needs to be placed halfway down the screen with regards to the screen width and height, as well as accounting for the ball's side length. The `resetBall(…)` function should look something like:

```
void resetBall(unsigned int *ball x, unsigned int *ball y) {
      (*ball x) = SCREEN_WIDTH/2-BALL/2;
      (*ball y) = SCREEN_HEIGHT/2-BALL/2;
}
```

The penultimate piece needed for this stage is to define the ball's various states. Since the vertical and horizontal states are independent, two variables are needed to account for all four combinations. Rather than use arbitrary integers and attempt to remember what they mean, declaring some constants will smooth things along. Four constants are needed to account for all of the various states. Instead of using integers, utilizing unsigned short variables will save space, especially since each variable will hold only one of two distinct values. Hence the constants and variable declarations should look similar to:

```
#define BALL                4
#define BALL_UP       0
#define BALL_DOWN     1
#define BALL_LEFT           0
#define BALL_RIGHT    1
...
int main() {
...
unsigned short state_x;
unsigned short state_y;
...
}
```

Currently, this program nearly has the ability to move a ball around the screen. With all the drivers set up and the variables declared, linking things together is a snap. The first objective is to set up the first frame before the game loop begins. Hence, all the variables should have well-defined values before the screen is displayed. The function `resetBall(…)` can be used to give the ball an initial location. Next, the ball needs to have an initial state. It does not particularly matter which way the ball moves initially, as long as the state is well-defined before the game loop begins.

Now that the ball has a state and location, the video memory needs to be adjusted. The first step is to set the entire screen to black. The graphics drivers provide the function `GFX_FillScreen_2BPP(…)` that takes a color index in the current palette and the pointer to the video buffer as arguments. The `2BPP` variant needs to be used because of the resolution file's bits per pixel setting. "Drawing" the ball is as simple as modifying the video memory with `drawBall(…)`. After all of the variable declarations, the code should look akin to

```
...
int main() {
...
unsigned short state_x = BALL_UP;
unsigned short state_y = BALL_LEFT;

// Setting up the first frame of the game
GFX_FillScreen_2BPP(0, g_VRAMBuffer);

resetBall(&ball_x, &ball_y);
drawBall(ball_x, ball_y, 1);

GFX_StartDrawing(SCREEN_TYPE_NTSC);

}
```

Enter the game loop. All of the game's logical operations need to be done during the VSYNC of the display to prevent hardware contingencies and graphical anomalies.

The graphics drivers provide two incredibly useful functions, `WAIT_FOR_VSYNC_START()` and `WAIT_FOR_VSYNC_END()`, that, as long as the code does not go over the cycle limit, can ensure the game loop is not doing anything productive while the screen is in the process of being displayed. Every iteration of the game loop will start with `WAIT_FOR_VSYNC_START()` and end with `WAIT_FOR_VSYNC_END()`. Everything else is sandwiched between these two function calls.

As described earlier, the first step in the game loop is to move the ball. This involves checking the state and then moving in the correct direction. Simply incrementing and decrementing the location variables is not quite enough, as the ball needs to stay within the screen of play. Hence the ball movement and collision detection for the upper, lower, leftmost and rightmost bounds are inextricably tied. Comparisons can be saved by checking the state first, then figuring out if the ball should move or to change the state. The vertical and horizontal states need to be checked separately, since they operate independently. Collisions to the left and right bounds of the screen are the scoring conditions, but, for now, having the ball bounce around the screen is the goal of this demo. The comparisons sandwiched between the VSYNC functions should look something like:

```
if (state_y == BALL_UP)
      if (ball_y == 0)
            state_y = BALL_DOWN;
      else
            ball_y--;
else
      if (ball_y+BALL == SCREEN_HEIGHT)
            state y = BALL_UP;
      else
            ball_y++;

if (state_x == BALL_LEFT)
      if (ball_x == 0)
            state_x = BALL_RIGHT;
      else
            ball_x--;
else
      if (ball_x == SCREEN_WIDTH-BALL)
            state_x = BALL_LEFT;
      else
            ball_x++;
```

Everything is almost in order. The final step is to modify the video buffer. The intuitive solution is to repaint the screen as completely black and then redraw the ball with the `GFX_FillScreen_2BPP(…)` and `drawBall(…)` functions. Double check, compile, and run the code. The ball will look like a rectangle simply because of the resolution. Make sure everything is working correctly before going to the next step.

AI Paddle Demo

The next milestone is to modify the ball demo to include an AI-controlled paddle. The next few steps will mirror the construction of the ball demo. Every paddle is a rectangle with a height and width, so setting up some constants now will set the dimensions for both the AI and the player. Using the `#define` preprocessor directive, the `PADDLE_WIDTH` and `PADDLE_HEIGHT` constants are integer values defining the dimensions of the paddle. One final constant is needed for both a graphical concern and collision detection. The distance between the paddle and the leftmost or rightmost edge of the screen should be the same for both the AI and the player to have the screen look symmetrical. This constant, `PIT`, will be used to determine the axis that the paddle moves along, as well as providing the location of one of the paddle edges.

The program currently has no notion of how to draw a paddle. Because there are two paddles on the screen at all times, having a function that takes the location and color of the paddle provides a great boon. The `drawPaddle(. . .)` function will look nearly identical to the `drawBall(. . .)` function, except for a few name changes:

```
void drawPaddle(unsigned x, unsigned y, unsigned colorIndex) {
      int i, j;
      for (i = x; i < x+PADDLE_WIDTH; i++)
            for (j = y; j < y+PADDLE_HEIGHT; j++)
                  GFX_Plot_2BPP(i, j, colorIndex, g_VRAMBuffer);
}
```

Because of the property that the paddle's horizontal location is fixed, only one variable is needed to store the location. But this requires another constant that is built on some of the other constants previously defined. Since the width of the screen is `SCREEN_WIDTH` and the distance between the rightmost edge of the paddle and the width of the screen is `PIT`, `SCREEN_WIDTH-PIT` yields the location of the right side of the paddle. This is close to what is needed but not sufficient. All of the locations of game objects are top-left corners, so using `SCREEN_WIDTH-PIT-PADDLE_WIDTH` provides the correct left edge of the paddle. The constant `P2_X` must be defined after `PIT` and `PADDLE_WIDTH` are declared.

Now for the interesting part: the AI algorithm. The obvious approach to have the midpoint of the paddle always follow the ball provides an extremely brutal opponent and little chance for the player to score. A more forgiving algorithm follows the ball if it's in the correct state and past a certain section of the screen. Since the AI-controlled paddle is on the right side of the screen, the ball must be in a right-moving state for a collision to occur. When choosing an $x-$location to start the AI control, the closer the choice is to 0, the more difficult the AI, as the AI will have more time to catch up to the ball's current $y-$coordinate. Using a third of the screen as a dead zone provides a challenging opponent.

So, if the ball is in a right-moving state and past the midpoint of the screen, the AI should attempt to match the midpoint of the paddle with the midpoint of the ball to maximize the possibility of a collision. Hence, if the ball is currently above the midpoint of the paddle, the paddle should move upward, or decrement the $y-$coordinate of the

location. Likewise, if the ball is currently below the midpoint of the paddle, the paddle should move downward, or increment the *y*–coordinate of the location.

One last consideration: The AI paddle should not roll off the screen. This can happen if the paddle is tangent to the top bound of the screen and the ball is still moving upward. Another case exists symmetrically if the ball is moving downward. The easiest solution is to check the location of the paddle before moving the paddle in the respective direction. A similar concern arises for player input in the next phase. The complete AI algorithm will look something like:

```
if (state_x == BALL_RIGHT && ball_x >= SCREEN_WIDTH/3)
      if (ball_y+BALL/2 > p2_y+PADDLE_HEIGHT/2) {
            if (p2_y < SCREEN_HEIGHT-PADDLE_HEIGHT)
                  p2_y++;
      }
      else
            if (p2_y > 0)
                  p2_y--;
```

Since the screen is being reset to black for every iteration of the game loop, the paddle needs to be drawn for both the very first frame (before the graphics engine is started with the function `GFX_StartDrawing(…)`) as well as right before waiting for the end of the VSYNC.

The final part to this demo is to modify and add to the current collision detection code. The modification needed is fairly trivial. Now that there is a paddle on the right side of the screen, one of the two possible scoring conditions can be applied. Therefore, the subcase in the ball movement code when the ball is moving right and tangent to the right side of the screen needs to change. Instead of bouncing, the ball needs to be reset and a function already exists for doing that! Simply call `resetBall(…)` instead of changing the state.

The addition to the collision detection code is fairly simple. The first bit to check is the state of the ball. If the ball is moving right, then the possibility of a collision exists. If the ball is possibly tangent to the paddle, simply compare the locations of both the upper and lower right corner of the ball with the left side of the paddle. If either corner is tangent to the paddle, then a collision occurs. Hence the only thing to do is to change the state. The collision detection algorithm should look something like:

```
if (state_x == BALL_RIGHT)
      if (ball_x+BALL == P2_X)
            if ((ball_y >= p2_y && ball_y <= p2_y+PADDLE_HEIGHT) ||
                (ball_y+BALL >= p2_y &&
                 ball_y+BALL <= p2_y+PADDLE_HEIGHT))
                  state x = BALL LEFT;
```

That completes this phase. Compile and run the code and ensure everything is working before transitioning from graphics demo to full-fledged game.

Input and Output

The main goal is to add interactivity in this phase. All of the core game logic is already written but needs to be reapplied symmetrically to handle the left paddle. But first, a second paddle must exist for the player to move.

Creating another paddle is a simple rehash of the steps to create the computer paddle. The location of the paddle needs a constant for the fixed horizontal location and one variable to store the changing vertical location. Since the left side of the screen is 0 for the x−coordinate, the constant `P1_X` is a rename of the `PIT` constant declared earlier. The `unsigned int p1_y` likewise serves a similar function to the AI paddle. Another call to `drawPaddle(…)` needs to be made for the left paddle as well for both the first frame and at the end of every iteration of the game loop.

Staying on the vein of similar modifications for a moment, the collision detection needs to be adjusted as well. Both scoring conditions logically work in this iteration, so instead of adjusting the state when the ball reaches the left side of the screen, call `resetBall(…)`. The next modification is at the end of the right paddle collision code. Appending an "else" onto that structure implies that the ball must be moving left. The next step is to compare the rightmost edge of the left paddle and the x−coordinate of the ball. The final step is to again compare the upper and lower corners, but this time the left side of the ball. The entire paddle-collision structure should be akin to:

```
if (state_x == BALL_RIGHT) {
      if (ball_x+BALL == P2_X)
            if ((ball_y >= p2_y && ball_y <= p2_y+PADDLE_HEIGHT) ||
                (ball y+BALL >= p2_y &&
                 ball_y+BALL <= p2_y+PADDLE_HEIGHT))
                  state_x = BALL_LEFT;
}
else
      if (ball_x == P1_X+PADDLE_WIDTH)
            if ((ball_y >= p1_y && ball_y <= p1_y+PADDLE_HEIGHT) ||
            (ball_y+BALL >= p1_y && ball_y+BALL <= p1_y+PADDLE_HEIGHT))
                  state_x = BALL_RIGHT;
```

Now, for some fresher content. The game input must be sandwiched between ball movement and the collision detection code, but there is no relationship between the AI code and the game input code. The drivers, again, provide a nice function for handling gamepad input on the XGS PIC. The `Gamepad Read(…)` takes a constant as an argument that determines which gamepad to read (either `GAMEPAD_0` or `GAMEPAD_1`) and returns a number representing what button is being pressed. The only buttons necessary for this game are the up and down buttons on the D-pad, represented by `GAMEPAD_UP` and `GAMEPAD_DOWN`, respectively.

Using a switch statement to read the input, for each case, the only thing to do is to adjust `p1_y` in the correct fashion, with one hitch. Like the AI paddle, the paddle should not roll off of the screen. The solution is the exact same, simply check to make sure there is still room to move the paddle before adjusting `p1_y`. The game input code should look like:

```
switch (Gamepad_Read(GAMEPAD_0)) {
      case GAMEPAD_UP:
            if (p1_y != 0)
                  p1_y--;
            break;
      case GAMEPAD_DOWN:
            if (p1_y != SCREEN_HEIGHT-PADDLE_HEIGHT)
                  p1_y++;
            break;
}
```

The game is nearly a good representation of the original Atari game, the only thing missing are the bleeps and bloops. The sound drivers have been initialized; it's time to use them.

The sound drivers provide a function `SND_PlayTone(…)` that accepts a frequency in Hertz and plays the given note. For reference, American tuning is typically A=440 Hz. This function will continually play a tone until another call is given with an argument of 0. The tricky part is to get the sound to play for a decent length of time without halting the game loop. A working solution is to stop any sound right after the VSYNC period begins. Then when a sound starts in any of the collision detections, the sound will automatically start. This ensures that a sound is playing at maximum for 1/30 of a second, and is long enough on average to generate a recognizable tone.

Optimizations

As promised, a major inefficiency in the current implementation will be brought to light. For every frame generated by the game loop, a call to `GFX_FillScreen_2BPP(…)` is made. This function will reset every pixel in the video memory to `NTSC_BLACK` even though most of the pixels are already in such a state!

The solution involves a few modifications to how the objects are drawn on the screen. Both of the respective functions will need to know the current state of the game object, as well as all of the current parameters. The function then needs to plot a few white pixels in the correct direction(s) and a few black pixels are painted in the opposite direction(s).

Consider the paddle for a moment. If the paddle is moving upward, then a single white row needs to be painted on top of the current paddle and a single black row needs to be painted at the bottom of the paddle. Simply reverse the colors when moving downward. The ball follows a similar paradigm, except 2 rows and 2 columns will have to be painted every time the ball moves.

With this implementation, the nested loop structure to draw the entire object can be moved to initialization. Also, care must be taken to ensure that pixels are not being plotted beyond the edges of the screen. These modifications greatly improve the efficiency of the program.