Table of Contents

Overall Design Structure

Pong was divided into the following modules:

ball

controls updating the position of the ball with collision detection and displaying it

beginCOM

linked at the beginning as in the mainCOM example, this sets segments according to COM file standards

block

contains functions to draw a filled rectangle to the screen

composte

would be called "composite" but tlink only works with files that are less than 8 characters long due to DOS conventions

endCOM

linked at the end as in the mainCOM example, this sets the stack portion, _IMAGE portion which the program draws to as a temporary location and then copies to the screen

keyboard

controls the keyboard input, replacing the original keyboard interrupt

main

contains the function _main which is called in beginCOM, which starts the code running, installs interrupts, and initializes variables; when you ask the program to quit, this module uninstalls interrupts and exits back to DOS

paddle

controls the updating of the paddle based on the current keyboard input

pause

controls the "pause screen" which is displayed when you pause the game

pixel

controls writing to the pixels to the image segment, printing characters, and also contains the functions for drawing an ellipse (this is because DOS/tlink only allow so many modules to be linked at once, ellipse functions could not be put in a separate module)

score

controls updating the score

screen

draws the playing field

sound

creates different sounds for hitting the paddle, winning the round, and hitting the wall

timer

controls the main game loop which runs on the timer interrupt at 100 Hz initially

Important Individual Modules in Depth

Ball

Module Variables

_ball_position, **_ball_velocity** store the (x,y) position and (x,y) velocity of the ball. These are updated by the _init_ball, _update_ball_velocity, and _update_ball_position functions. The _update_ball_velocity function actually does nothing, all the work it performs is handled in other procedures. These variables are not initialized and are stored in the _BSS segment.

Functions

_init_ball sets the ball position so that it is connected to the paddle. When the game has started but no player has pressed the spacebar button, the ball remains connected to one of the players paddles (depending on who scored the last goal) which the paddle itself can move up and down. This function gets called every frame to maintain the correct ball position until the ball is launched off the paddle and into play, then _init_ball is not called until someone scores a point.

_update_ball_position moves the ball according to the current ball velocity. It then checks to see if a collision has occurred with the left paddle using the _collision function. If a collision has occurred, it checks to see if the ball is already going in the right direction, if it is nothing needs to be updated. If it was going in the opposite direction, the velocity needs to be reversed, the vertical ball velocity is added from the paddle's vertical velocity. A sound for hitting the paddle also needs to be played. This method of collision essentially makes the paddle a push field rather than a hard collision, but it greatly simplifies the collision as no normal needs to be calculated, and in practice you cannot tell the difference. After this it checks the right paddle the same way. It then checks to see if the ball has hit the boundary of the screen in the x direction; if it has hit the boundary someone has scored a point and it plays a different sound, registers the point on the score, and resets the round. If nobody has won the round, the last thing _update_ball_position checks is if it has run into the top and bottom edge, in which case it reflects the ball off the edge it hits and plays another different sound.

_draw_ball draws the ball to the image segment, saved in ES upon entry to the routine (in fact this is set in the screen module function _print_screen). It can either draw an ellipse for the ball, or draw a rectangle for the ball, but the rectangle looks more like the real pong, so I

chose to draw the rectangle in the final version. The function checks the x and y position of the ball and either adds or subtracts the radius of the ball to get the top left, and bottom right corners (x,y) coordinates. This is used to draw a rectangle with the DrawBlock macro defined in screen.mac which uses the _block function in the block module.

_collision performs the collision detection between the ball and a paddle. For the purposes of collision detection the ball is considered a rectangle, even it if is drawn as an ellipse. To test for a collision between two aligned rectangles, you simply need to compare each plane and if all planes are such that a collision occurred then it returns a 1, otherwise it returns a 0.

Keyboard

Module Variables

_key_status is a byte that saves the current keys that are pressed. Defined in the file keymap.mac, are macros such as KEY_A which is a bitmask to check if the A key is pressed. You can TEST the bitmask KEY_A with the byte in _key_status and if it is non-zero, the A key is pressed. This byte is updated by the keyboard interrupt which we install.

old_key_int saves the location of the old interrupt 0x09 for when we want to go back to DOS and reinstall the old interrupt. It is global only to show up in the MAP file.

Functions

key_int is the actual function which is called when a keyboard interrupt occurs. We replace this function for the old INT 0x09. When a key is pressed and this function is activated, the first thing it does is get the scancode for the pressed key. If compares this with the scancodes defined in keymap.mac and if it matches any of them (up or down), it turns the corresponding bit on or off in the _key_status global variable. No interrupt chaining occurs.

_install_key_int replaces the original INT 0x09 with our new function for it. **_uninstall_key_int** does the opposite: it replaces our interrupt with the original INT 0x09 when the program exits.

Paddle

Module Variables

_paddle_position sets the (x,y) position for each paddle. It is 4 words stored in the form: x1, y1, x2, y2. where x1 is in the location [_paddle_position] and y2 is in the location [_paddle_position+6]. This position is the top left of the paddle, everything is based off this.

_paddle_velocity sets the y paddle velocity for each paddle. Paddle 1 is stored in location [_paddle_velocity] and paddle 2 is stored in location [_paddle_velocity+2].

_paddle_intent sets the direction that the user directs the paddle to move in. For example, if the A key is pressed but not the Z key, player 1 wants the paddle to move upwards.

This means paddle 1's intent is -1. However, if the paddle is already at the top of the screen, it cannot move any further upwards, so its velocity would be 0 since it is not moving. It is stored in the same format as _paddle_velocity as two words, one for each paddle.

Functions

_init_paddle is called once at the start of a round to set the paddles to the middle of the screen and not moving. The position and velocity values for both paddles are set accordingly.

_update_paddle is a driver that is called each time through the timer interrupt. It first calls _update_paddle_intent to update the intents of the paddles based on what keys are pressed. Then it calls _update_paddle_velocity to update the velocity of each paddle based on the calculated intents. Then it calls _update_paddle_position to update the position of each paddle based on the calculated velocity. It is in this _update_paddle_position that the paddle gets clipped vertically to the screen if needed.

_update_paddle_intent reads the _key_status variable in the keyboard module and determines what keys are pressed. Depending on this it knows if the player intends to move their paddle up, down, or stay constant. If both the up key and the down key are pressed for a paddle, it decides that the player wants to keep the ball in the same position.

_update_paddle_velocity simply copies the intent of the paddles into the velocity. No scaling occurs so if a paddle is intended to move upwards, it will try to move upwards at a velocity of 1 pixel per frame.

_update_paddle_position updates the position of each paddle based on the velocity and previous position. First it adds the velocity to the current y position. This new position value is then clipped to the top and bottom borders. If it is clipped, then the paddle velocity is set to 0. This happens for both paddles.

_draw_paddles draws the paddles to the image segment already set in the ES register. It uses the current position and size of the paddles defined in paddle.mac to set the corners of a rectangle to draw with the _block function.

Pause

Functions

print_num is a function used to print numbers in ascii form. Inputs are the x,y coordinates of the 640x200 CGA screen to print at, and it will convert the input number to 2 characters and print them using the PutChar macro defined in screen.mac which uses the _write_char function in the pixel module. _write_char copies the bit pattern in the character generator ROM in BIOS to the x,y coordinate you want, based on the character you give it to print. Note: print_num will only function properly up to a score printout of 100, above that, it will start again at printing 0. Games are arbitrarily limited to a max score of 7 so this isn't a problem. We could even just print one digit for a score of 0-9 instead, but 0-99 makes more sense in case we want to raise the max score.

_pause_game is the important routine in this module, when the game is paused, the timer stops updating the positions of the ball and paddles, and instead the "pause screen" appears which displays a help message and the scores. To stop the game loop from triggering, this function sets _game_paused to be active. When the timer interrupt sees this it will exit without doing anything. It then enables interrupts so keyboard interrupts can occur (it uses key presses to determine whether to quit the game, start a new game, or continue the current game). The ES register is set to display directly to the video screen, instead of to the image segment. Since we aren't updating in real time, we can draw everything on the screen directly without any aliasing problems. It then clears the screen by writing 0 to the entire screen image. It then prints the score by writing the string stored at score_str on the first line (y=0). It then prints the string of player 1's name in player1_str and then player 1's numerical score with print_num all on the second line (y=8). It repeats this for player 2 on the third line (y=16). It then prints some help messages from help_str, help_str2, help_str3, and help_str4 on the lines y=32, 48, 56, and 64 respectively. It is now finished with the output. If the game is won, it locks the screen until a new game is called for, you cannot continue a game past the max score. Since we want to show the pause screen and the help the first time we enter the game, we have a variable that says whether this pause screen instance is from being the first time ie. _first_time, or if we pressed the Esc key. If we pressed the Esc key we must wait for it to be unpressed before continuing otherwise we would exit the game immediately because Esc would still be pressed, and if you press Esc and the pause screen it exits. After it is done waiting there, it repeatedly queries the _key_status byte until either spacebar, Esc, or N is pressed. If Esc is pressed it sets the variable _quit_now on, which the main function is repeatedly querying. When it reads _quit_now active, it will exit immediately. If the N key is pressed, it requests a new game and exits the pause screen. If the spacebar key is pressed, it continues the current game where it left off. Since the ball and paddles did not update while at the pause screen, everything is the exact way it was left. After it detects one of those keys and performs the appropriate action, it sets _game_paused inactive, so the game starts updating again.

Sound

Module Variables

sound_done sets whether the last sound to play is done or not. If it is done, update functions do not need to do anything.

active_sound sets the currently playing sound based on the ID's found in sound.mac.

sound_count is a counter for the current note, when this reaches the delay for that note, the next note is played.

sound_note is the note that the sound is on, for example if the sound has many notes to string together this will count from 0 up to the last note.

sound_length is the length of the currently playing sound in number of notes.

sound_begin stores a pointer to the beginning of the sound in memory.

Functions

_init_sound is called at the beginning of the program and sets sound_done low so that it does not update anything until the first sound is called.

_play_sound plays the sound specified as the ID in sounds.mac. First it stops interrupts just in case, so we don't try to update a sound that we haven't set all the attributes for. Then it turns the speaker on to start beeping sound. Then it sets _active_sound to the sound specified. Then it looks up the length of the sound in sound_length_table based on the ID and sets sound_length. It looks up sound_begin in the same way using sound_data_table and the ID. It then sets the pitch for the first note in the sound's data table. Then it resets the counter data and exits.

_sound_tick is a function that is called on every timer interrupt. It increments sound_count and if it reaches the limit for that note, changes to the next note and resets the count. If it reaches the end of the notes, then the sound is done playing and it turns sound_done active.

speaker_on and **speaker_off** turn the speaker on and off respectively.

set_pitch sets the pitch of the note being played by taking an input value and writing it as the count value to use in timer 2, which drives the speaker.

Timer

Module Variables

old_timer_int saves the location of the old timer interrupt, so that when we exit back to DOS we can reinstall it.

timer_count sets the count to use for timer 0 and subsequently the rate of the timer interrupt.

_game_paused sets whether we are at the pause screen and should stop updating or not.

_player_with_ball sets who has the ball connected to their paddle before launch. We play "winner's ball" so whichever player scored a point gets to serve then ext ball.

_game_running sets whether the game is running, or if someone has the ball connected to their paddle instead.

_first_time sets if this is the first time into the timer interrupt and should pause for the first time.

Functions

_install_timer_int and **_uninstall_timer_int** install our new timer interrupt and save the location of the original, and vice versa respectively.

set_speed copies the word in the variable timer_count into the count of the timer 0.

_reset_speed sets timer_count to the default value for 100 Hz and then calls set_speed.

_increase_speed lowers the count in timer_count and then calls set speed. This reduces the time between timer interrupts so the game speed appears to increase. The count is decremented by 0x400 each time this is called (when the ball hits a paddle) until the count reaches less then 0x0400 total. By this time the game is running so fast that it wouldn't last long.

timer_int is the main purpose of this module, it runs the main game loop. Each time this is called it appears as another frame in the game. First it calls _sound_tick to update any currently playing sounds regardless of whether it is paused. Then the function checks whether it is currently paused and if it is, quits right away. It then checks if it needs to pause because of _first_time, if the game is over, or if the Esc key is pressed. Otherwise it updates the paddle and ball and prints the screen. The logic in here is slightly convoluted because of the different state variables that need to be checked and whether we need to end the interrupt before we do some action or not.

Further Work

As it is now, pong is playable and works correctly with everything implemented. There is a generic interface to add more types of sounds as desired. Increasing the speed of the game as it progresses works as desired. Several sections that I would improve upon would be the timer interrupt. Right now, the logic is fairly convoluted to choose between what code paths to take based on combinations of different state variables. I would also modify the sound to run on its own timer interrupt separate from the main timer so that as the game speed increased the sounds remain the same speed instead of becoming shorter. The keyboard interface is also fairly clumsy. In order to add new keys to detect you have to copy a lot of code. To improve this I would probably convert this code into a macro and simply call it with the different keys I want to detect for in some generic method. If multiple keys are pressed at the same time sometimes it does not register the key presses and unpresses, it is unclear whether clearing the keyboard buffer did anything to help this, so it was left out of the final code. This may be a hardware limitation, but it is something to work on in a future version.