Dino Game Process

Here is how I made the Dino game.

First, I had to figure out how to use the supplied JackCompiler, so that I could test my code as I went (since I had not written a compiler myself yet). To do this, I had to:

cmd into the nand2tetris > tools directory

enter the following command: JackCompiler ..\projects\09\List

Then I began working on the class files, starting with TREX.jack.

In the bitmap code generator, I made a simple Trex in four different positions: two to alternate for running, one of which also works for jumping, and two for crouch run positions. I then made a Main.jack and function main() to try out the bitmaps and test positions for the Trex. At this point, the bitmap rendering functions involve many calculations using the location on screen to determine the RAM locations, so I think I might optimize that later, but I’m not sure; it might be nice to have the variable location placement so that I can use the same functions for different positions of a jump.

After some experimentation, I settled on having the Trex position be at 7300; this gives users plenty of time to see obstacles and also is a little above the bottom of the screen, so I will be able to add ground.

Problem: the 16 by 16 bitmap was far too small. After looking through the code and playing around with FlappyBird (url), I decided to do something similar, by rendering objects in columns of 16-pixel width, but with arbitrary height. If something needs to be wider, it is simply rendered with two columns. For each column, I can use only as many poke commands are rows, rather than poking 0 into already white space. So now I go back to the bitmap editor and start working on multi-piece graphics. I also am starting to put the bitmaps in a separate class, called Graphics, which will be cleaner code.

Since the renderings are bigger and in chunks, I further split the render functions to allow for re-rendering just the legs, for running, rather than re-render the entire Trex.

It’s been three hours and I still do not even have the basic Trex regular graphics done.

I added a tail as a third column. At this point, each chunk gets its own function and there are functions that render the different chunks together. I think it will be most convenient to have the feet chunk position as the main location for the dinosaur and offset the head as +1 word (right a block) and -512 words (up a block) and the tail as -1 word.

After three and a half hours, I think I’m finished with the Trex. It uses the same feet and tail for both standing and crouching, and two feet modes that can be switched back and forth each game clock cycle to give the impression of running without re-rendering the entire dinosaur. If we want to re-render the entire thing, however, we can, just by using the drawStand1 or drawCrouch1, which we will likely need to do for jumping and switching between stand and crouch mode.

Edit: Now, there is only one drawStand and one drawCrouch, and since both positions use the same feet, I’ve refactored drawStandFeet1 to simply drawFeet1 and same thing for drawFeet2, so Trex will simply draw such a main position and then use drawFeet1 and drawFeet2 to run.

Things I learned Day 1

* 16 by 16 pixels is far too small for a significant player sprite. The height can easily be extended with more poke commands, but to increase the width, you have to add another column, since the screen works in 16-bit words. Moving parts are most easily done in the same “chunk” of 16-pixel width, and doing so allows that part of the character to be updated without needing to update the other parts of the character.
* Locations on the screen are memory array locations with base SCREEN = 16384. So, a pixel tall row can be stacked on top of another with location -32 since there are 32 words in a screen row. Then a 16-bit chunk can be stacked by adding or subtracting 32\*16 = 512. To move to the right or left 1 chunk, use +1 or -1 respectively. All this should have been obvious from the discussion on the screen from week 4 of part 1 of the course, but it has been a while since I did that and had forgotten.

Day 2

Now I’m starting to put together a DinoGame class to run the game, and will try to set things up to get a jump on space bar / up arrow click.

Ok, so after maybe 1-2 hours, I have a pretty reasonable jumping mechanism. Of course, I had a few glitches, some of which I documented, many tries that I didn’t. Right now, I’m using a downward velocity number which gets increased each game cycle, just as gravity increases downward velocity in real life. Then, to update the Trex position, I simply find the location that is previous\_location + (downVelocity \* 32), so if downVelocity is 5, then the new location becomes 5 screen rows lower than before. I then check to make sure that the new location is not below ground by comparing it with initial\_location. If it is below ground, I simply reset downVelocity to 0 (like the ground pushing up and counteracting gravity), and set the location to be simply the initial ground state location. Whenever the Trex is on the ground, it switches its feet position each cycle.

To implement this jump without clearing the screen in between frames, I have a separate drawJump, which is pretty much like drawStand, except that it takes also as input the number of rows that were “jumped” and erases the screen for those rows in the dinosaur column. The problem right now is that it clears all but one row each time (probably some off by one error somewhere), so I need to fix that [edit: it turned out that I had been clearing rows above the Trex when it was going up and rows below when going down, while the opposite was intended].

Things I learned

* To give an object “physics-like behavior”: keep track of a velocity vector; then on commands like jump, run, etc., update that velocity vector accordingly (checking to see if the object is on the ground or in a state in which the move is valid); then every game clock cycle, update the player’s position using that velocity vector, and also update the velocity vector with independent physics (so take into account the forces of gravity, or of solid surfaces and the way they must change the current velocity in a time step).
* In DinoGame.run(), use a loop that constantly updates the different objects on the screen, then does Sys.wait(delay) for some specified delay that controls game speed.
* Not many “breakthroughs” today, but a lot of trial and error that sort of led to some results that show themselves better in the code than I can describe here.

Day 3

Today I worked on implementing crouch. After some consideration of how to represent whether the player had been crouching, just started crouching, is not crouching, etc. I settled on a simple integer number that represents the case. The different cases are important, so that if it continues crouching, I can simply swap feet positions rather than re-render the whole thing, and I need to know what things to erase (I have to erase the standing head when entering a crouch and erase the crouching head when exiting from a crouch). The code that resulted is not super elegant, but it works, and does so pretty efficiently, so I think it’s good.

There are four main things left to do in the game: add the additional objects (cacti, Pterodactyls, ground, and clouds, which will go at a different speed), generate these objects in a playable, yet seemingly random way, figure out if the player has collided with a collide-able object, and add a score tracker.

I’m going to start by making a single cactus sprite and try to get that to scroll across the screen, learn what I need to about handling collisions ect. with it, then add the other sprites. In the FlappyBird game, the Game class maintains three instances of the pipes, and the pipes themselves update by resetting their cords to the right side of the screen when they go off the screen on the left (and they also change their hight randomly on this update). I might try something similar, though the cacti will have to change their drawing and collision box as well, to enable multiple types of cacti on the map. At first, I think I will consider a cactus’s collision box to be the rectangle with the same height and width as the cactus. Since it is easy to check the interlap of two rectangles, I can then treat the Trex as two rectangles: head and body (ignoring tail collisions), and check both for collisions with the nearest cactus each cycle.

The FlappyBird game uses rectangles for the pipes that scroll across the screen and that seems like it would be easier than using sprites, since the OS has convenient rectangle drawing commands, but rectangles will not work for Pterodactyls or clouds, so I will have to do something using bitmaps. My current idea is to come up with 16 different bitmaps for the same object (1 for each index in a screen 16-bit word, then cycle through those 16 bitmaps, and every 16 clock cycles, changing the actual location of the cactus. I could actually do 8 bitmaps instead, and just have the cactus move two pixels each time step, or 4 and move four pixels each step. I will try these different things with a simple rectangle before putting the effort in to the cactus bitmap drawings.

My current thought about the ground is to have a sequence of 16 (or some other amount) bitmaps that each ground “chunk” will progress through, so that a ground object does not have to change position, but it will still look like movement.

Things I learned

* The Jack platform uses keycodes 131 and 133 for up and down arrow respectively, as opposed to regular ASCII.
* Rather than switch the screen color, draw an object, then switch back, just add eraseObject functions that access the same screen areas as the regular drawObject functions, but poke 0.