Bryce Daniel

October 13 2014

Moving Blocks

Part 1 - A game and its rules.

Any good game has to follow a set of specific rules are laws that govern its play. If these rules were to cease in their existence, there would be no game either. If a game is meant to have blocks fall and we catch them to score points (See Manual for BlockBox) we cannot have blocks spawning all willy-nilly. The blocks instead must be created at the top and within the left and right bounds of the screen. This is just one example, but we can already see how important rules are to the game. To test the game, we must look at the game model. However, we humans are so inconsistent in our gameplay. Therefore, we shall use the Auto mode that will play the game catch the majority of the blocks (as a human player would) as well as run some testing functions (to be discussed later in this paper).

Part 2 - The Model

The game BlockBox is set up in a specific fashion in order to adhere to the rules defined in The Manual. In order to represent the rules of the game into rules that the computer/iPhone can understand, the model is implemented in Xcode using the Swift language and SpriteKit API. Briefly, let me touch on SpriteKit and what I found to be its benefits as well as its shortcomings. SpriteKit allows me to create SKNodes which are essentially trees of SKSpriteNodes. For instance, I have created the game scene which is a SKNode. Everything seen in the game from the text to the falling blocks are then children of this node. With built in functions such as “addChild()” and “removeFromParent” I can control what is displayed on the screen at which time. This makes it fairly straightforward in terms of managing graphics and updating the display. However, the implementation only allows me to access the first or last node in the parent SKNode. (we will see why this is an issue later).

The model of BlockBox, without going into incredible detail is as follows:
The GameScene class is where the majority of the code lives. There is some code in other files, but those are generated by Apple and not really of any importance to us. The BlockBox code only exists in GameScene.

The scene was setup by changing the background color, adding gravity simulation and creating a PhysicsWorld which will handle all of the collision simulations etc.

The function calls on line 85 - 95 like setupBackground(), setupHeart1(), setupBox() and setupBlocks() to name a few, call their respective functions responsible for creating the objects and adding them to the world. It defines their size properties and locations relative to the screen width and height. It also defines them as physics bodies of a specific type (the BitMask type) as well as what to check collision or contact against. spawnBlocks() on line 330 is a bit more complex. It sets up the size of the blocks as well, but it also generates a column position that is then translated in to a percentage across the screen. The block is then generated within specific parameters to determine that it appears on screen. setupBlocks() on line 290 does the rest of the work, generating the blocks with a delay and repeating that action forever. The same is done for the antiBlocks, but a function called randomly() ensures that they only generate once in a while. Similarly there are functions for setting up the score and the high score labels, the auto button and the replay button at the end. The high score label works by checking to see if the score is greater than high score. If the score is greater, the high score increments as well. As for lives, there is a counter that subtracts from thirty. Each 10 points lost (set up as a switch) results in a case that removes the heart node.

But at this point, you may be wondering how does the model handle scoring and lives? Well, this is where it gets interesting. The function on line 571 didBeginContact() does all of the heavy lifting. The first if statement checks several things simultaneously. First, it asks if the two physics bodies colliding are a block and a box, if so we continue. We then check that the origin of the falling block has a greater y position than the minimum y point of the box, but has a smaller y position than the maximum of the box. It also checks the x position against the minimum and maximum x points of the box. This takes place on line 572 and ensures that only collision with the inside of the box results in a block catch. I then set up a series of actions that will delay momentarily, then fade the block away, and remove it from the parent node BlockSet. I also iterate the score, change the name of the block so that it only registers once, update the highscore if necessary, and increase the spawn gravity. If the block is instead an AntiBlock, the same procedure takes place, but a point is subtracted from the score instead.

If the block hits the ground instead, it invokes a similar collision if-block that checks if the the two colliding objects are a box and ground object, then will then change the block color to black, delay for a moment, and then remove it from the SKNode.

The box movement is fairly straightforward. Using the touchesBegan() on 632, touchesMoved() on 663 and touchesEnded() 675 I can get the location of the finger touch registered. If the box is in the same location as the touch, nothing happens. But if the box is not, it then slides over to

the proper position. There is a slight delay to ensure that it animates smoothly instead of jump around.

Part 3 - Testing the model.

The game, when the auto button is pressed, runs a stream of tests during the physics simulation. The outputs are then streamed through the output console and a shouldBeTrue boolean is monitored. Before I delve into the specific tests, there are some limitations (briefly mentioned at the beginning of the paper) that I need to discuss. The function that is commented out on line 245 determines if the score increments when the block is caught. However, do to the nature of the SKNode implementation, I can only access the first and last node. If every block is caught, the function will work just fine, however if the block is resting on the ground, the caught block is not the first node as there is an older one still in existence. I suppose I could remove the dead blocks immediately, but that would violate one of the specifications of the game which states that there needs to be dead blocks. I figure this is a trade off that is not of great import as when the block is caught, we can clearly see that the score increments as blocks are caught during the physics simulation and only when blocks are caught. This aside though, let us examine the other tests.

testThatBlockisDead() on 109
When a block hits the ground, its name should change from “LiveBlock” to “DeadBlock”. This test checks just that. How it works is it compares the minimum y value of the fallen block to the maximum y value of the ground. If it is equal or less than (less than in case of inaccurate float values) the ground maximum, then the name should also be “DeadBlock”. With a series of if-blocks we determine that if and only if the block is on the ground, is the name of the block “DeadBlock”.

testThatBlockGeneratesWithinFrame() on 130
A block should only spawn within the left and right bounds of the frame. This test check that if a block has indeed spawned, is its x position greater than the minimum x value of the frame and less than the maximum value. If the position of the block is not within these bounds, then we have an issue and there is a flaw in the model because it would be appearing off screen and would therefore be uncatchable.

func testThat30BlocksIsGameOver() on 144
When a block falls, the game subtracts a life point. When you lose thirty life points, the game is over. Using a groundTouchCount integer variable, and checking that it is equal to thirty, it should be the case that it only equals thirty when the game is over. Also the game should only be over when the count equals thirty. After running the physics simulation, it looks at the if-blocks to determine if this is true. The model should pass because the uncaught falling blocks do in fact subtract from the life points.

testThat10BlocksIsLoseLife() on 169
Similarly, when you loose 10 life points, you should lose a heart life. This one checks that if the groundTouchCount is >= 30, 20, or 10, the lostLife 1, 2, or 3 boolean should be true respectively. It should be the case that you lose a life if and only if you have lost the correct amount of life points. For instance, if the groundTouchCount is less than 10, you should not have lost a life yet. The game passes this test and the lives are subtracted at the correct rate.

testThatHighScoreisMax() on 122
This test checks that the high score is the highest your score was during a play session. It checks that the high score is always greater than or equal to the score integer value. It should never be the case that is less, and if it is, there is an issue with the game model. It also shows however that the high score does not decrease with negative blocks either because it is meant to be the highest amount of points your reached during the round.

testThatBoxMovesToTouch() on 236
This simple test ensures that the box moves to where the finger is. It checks that the touch x location equals the box x location and then prints out a pass statement. It should never be the case that the box does not equal the touch location.

testThatSceneLoads() on 26 in the file BlockBoxTests.swift
This test that the game scene in fact loads because if it did not, there is no game to play. This test was an attempt to understand the XCTestCase class provided by Apple for unit testing. When run, it supplies a green checkmark so it is the case that the scene does in fact run.

From the above tests, as well as watching the game automatically play, we can see that the game model works as desired. Everything is set up and runs as it should. The box does not move to random locations, the blocks spawn within the frame at the top of the screen, the points only increase when the block is caught in the box, and the blocks change to dead blocks once they hit the floor. BlockBox is a fully functional game ready for the App Store and with this essay to prove that the model is working, who wouldn't jump at the chance to download it and play for themselves.