3: Pentominoes

Due: Noon September 17

Description

In this assignment you will construct the first part of a simple game called Pentominoes. A pentomino is one of 12 shapes constructed out of 5 unit squares. The idea of the game is to arrange the 12 shapes to form a regular 6x10 rectangle (or other shape).

This assignment focuses on manipulating views, view geometry, and simple view animations to automatically solve the puzzles. It will also involve using the Model-View-Controller (MVC) pattern in your code. Next week's assignment incorporates gestures to allow the user to manually manipulate each piece, including moving it onto the game board, rotating and flipping the pieces.

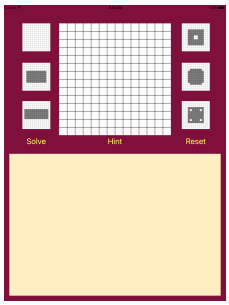
Assets will be provided for the game boards and the game pieces. Be sure to create your project in the same folder as your previous assignments. Make sure to also create a new branch from master before starting.

Warning: This assignment is significantly more involved than Assignments 1 and 2. Start early!

Walk Through

This walk-through will only describe the overall architecture of what your app's functionality. You are free to choose a different design and/or user interface (though check with me if you do something drastic). The assignment requires a moderate amount of code, but the organization and coordination of that code is critical to a well-designed solution.

1. Create a new Single View Application for iPad. Add all the provided assets to the project. All the images should be added to the Images Assets folder. The property list and Pentomino Model code should be dropped into the File Navigator. Make sure to have the "copy items if needed" checkbox checked when adding these.
2. Design your UI:
   2.1. Make sure auto layout is turned on in the Storyboard. You will use auto layout for all of the static items laid out on the storyboard in interface builder (board image, board buttons, etc..). You should not use auto layout for the individual pentomino pieces.
   2.2. Lay out the user interface on the Storyboard so that you have something approximating the display shown here. The large board is a UIImageView. The six smaller ones are customized UIButtons displaying the six different playing boards. Use UIButtons for Reset and Solve. (The Hint button won't be used until next week.) Make them look nice. The UIImageView should have size exactly 420x420. This makes each square have size 30x30.



   2.3. Tapping any Board Button should trigger a single action method that will change the playing board to the one displayed on the button. (Hint: Use a single action method and the UIView's `tag` property to distinguish the buttons.) The Reset and Solve buttons should be connected to action methods. Make sure auto layout is supported for all iPad sizes. That's it for designing the UI in Interface Builder.
3. Design your Model:

3.1. Your code needs to support the MVC design pattern. This means that you will have a model object and a main view controller. All specific information (tile image names, number of boards, etc.) and assets should be stored in the model. The main view controller should not rely on any specific information that could change (e.g., name of files, names of pieces, number of pieces, etc.). It should be as generic as possible.

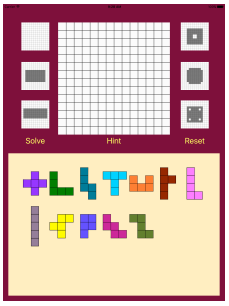
3.2. This project has 3 kinds of assets that you need to manage: images of the six playing boards, images of the 12 playing pieces (the pentominoes), and a property list containing solutions to the five boards with shaded areas. (The blank board has no solution since it has no shape to fill in.) I have provided starter code for a model class that reads the property list into an array. You will need to add functionality to this class. You need to decide the appropriate places to load other assets. The model *must not* import UIKit.

3.3. Think about what information the model needs to supply upon request from the controller. Design and implement methods to support this without creating any unnecessary dependencies between the model and controller. The controller must not depend upon the particular structure of the solutions property list. Keep the model and controller as independent as is reasonably possible.

4. Implement the game logic in your controller object.

4.1. Tapping on a particular board button should make the corresponding board become the current one (visible on the UIImageView). This should be completed with only one action method for all of the buttons.

4.2. Programmatically create and add the 12 playing pieces (`UIImageViews`) to the board approximately as shown. (Do not use Interface Builder for this.) Use the frame of the main playing board (or perhaps of a subview) to fix a position for placing the pieces just below the board. Use appropriately defined constants for spacing the pieces vertically and horizontally. You should need 2 rows for portrait orientation. You need to find an appropriate place to perform this code. You can only do it after the main imageView's frame has been determined. You may want to separate the code that creates the imageViews from the code that places them on the main view.



4.3. Implement an action to solve the puzzle. Solving the puzzle entails automatically moving each piece into a correct position for the current puzzle. Use an appealing animation for this. (The blank puzzle has no solution and therefore cannot be solved.) A solution is a mapping from tile letter names to a struct containing information for positioning the piece. (Tiles are named for the their resemblance to a letter.) The information includes coordinates x and y corresponding to unit square sizes, plus the number of (clockwise) 90 degree rotations and flips (along vertical axis). The coordinates represent the origin of the piece. To convert these coordinates into the coordinates of the board you need to multiply each value by the length of the side of a square (30). Rotations should be performed before flips (these operations are not commutative). Use `CGAffineTransform` convenience methods for the rotation and flipping. (For flipping use values -1.0 for x and 1.0 for y.) Don't worry if the animations don't always look perfect.

4.4. Implement an action method for the Reset button. The pieces should be animated back into their starting positions on the main view. Be sure to "undo" any transforms that have been applied to the pieces. You can use the identity transform for this.

5. Be sure to include App Icons in your project.

Testing

In most assignments testing of the application is imperative. Be sure that all the required features have been implemented.

1. Your project should build without errors or warnings.
2. Your project should run without crashing.
3. Each of the bullet points in the Assignment section above will be considered to verify that you've completed the assignment correctly.
4. Your project should have a clean user interface. User interface elements should be arranged logically, be aligned nicely, etc...

Hints

1. The starter code contains the class `Model` which contains initialization code for reading in the solutions property list. You will need to extend the model with additional functionality.
2. One of the most important principles of iOS programming is understanding when to perform different actions and when data will become available. The two are often related. Be sure you understand the basics of view and viewController initialization and some of the methods sent to viewControllers.
3. You may find it useful to have an array containing all the buttons for selecting boards. Instead of creating six Outlets, create one Outlet Collection as follows. When connecting the first button using the assistant editor, choose Outlet Collection instead of Outlet from the pop-up menu. This will create an IBOutlet that is an array of buttons. You can then control-drag from the remaining five buttons to this variable. Verify in the Connections Inspector that all connections were made.
4. The 12 playing pieces are all named after their shape, e.g., `PieceF@3x.png`. The "`@3x`" indicates that it is a retina image. When specifying the file name leave this out, e.g., `PieceF.png`. These images all have widths and heights that are multiples of 30 (so they will match up with the squares on the playing boards). Make your code robust! The twelve letters representing the tiles' shapes (F,…,Z) need only appear once - perhaps an array of these letters. The rest of your code should be indifferent to these letters.
5. When placing the pentominoes programmatically on the board, make sure to account for the leading and trailing space to/from the main view on the left and right sides. You will also need to add reasonable constant padding between pieces.
6. You might choose to use an additional UIView to layout the pentominoes on the main view that relies on auto layout constraints. This requires that you layout the pentominoes after the `viewDidLayoutSubviews()` function gets called.
7. Each square of the pentominoes is of size 30x30. Therefore the tallest pentomino (1x5) has height 150. This will be important when determining reasonable spacing between pieces.

8. You might want to create a datatype (class? struct?) for each pentomino. Along with the actual imageView, it might contain information like the number of times it has been flipped or rotated, its initial coordinates, etc.
9. Use only one method for all six board buttons. You can identify which button triggers the method by setting a unique tag value on each button.
10. Be sure to make no assumptions about the sizes of views. Use frame and bound properties as necessary.
11. A model should know nothing about the UI of the app, so it typically manipulates nothing with the "UI" prefix. Do not import UIKit into your model.
12. Moving a piece from its initial position to the board (and vice-versa) requires removing the piece from its current superview and adding it as a subview to its new view. You must explicitly change the piece's frame to use the new superview's coordinate system. You might use UIView's `convert(CGPoint, to: UIView?)` or `convert(CGPoint, from: UIView?)` methods for this. Also remember that a view can be visible as a subview even if its frame is not within the bounds of its superview (unless the superview's `clipToBounds` property is set to `true`).
13. Most of the images representing the pieces are not square. This means that if you perform an odd number of 90 degree rotations, the resulting width and height need to be swapped on the imageView's frame in order to maintain the proper aspect ratio.
14. When implementing the reset and solve methods, I recommend ignoring animations at first. Work out all the details with frames and coordinates before attempting to animate the movements. The animations should involve the frame, center and/or transform properties of the pieces.

Troubleshooting

Make effective use of the debugger. Set breakpoints and examine values. Be sure you understand when methods are called, especially ones involving initialization, loading views, laying out views, etc.

Submission

Your submission should be pushed on the master branch. Be sure to verify that your project builds and remove all cruft.