CS3217 Project Sprint 1 Report

Storyteller

Marcus | Pan Yongjing | Tian Fang

Requirements

Overview

Traditionally in the film industry, there is normally a drawer/designer who is in charge of storyboard design. However, hiring a dedicated person for storyboards is not always feasible for small-budget or personal projects. Additionally, more independent filmmakers are emerging nowadays as film equipment is getting more accessible. Therefore, we want to come up with an app to help filmmakers and videographers to design their storyboards with ease using their iPads. The users can effortlessly draw rough sketches to describe their shots and order different shots in various scenes using our app.

Features and Specifications

View and Select Projects

- User can view the projects that he has and select accordingly

View Scenes and select Shots

- Users can see a list of all shots grouped by the scenes they belong to.
- Shots are arranged in chronological order

Canvas

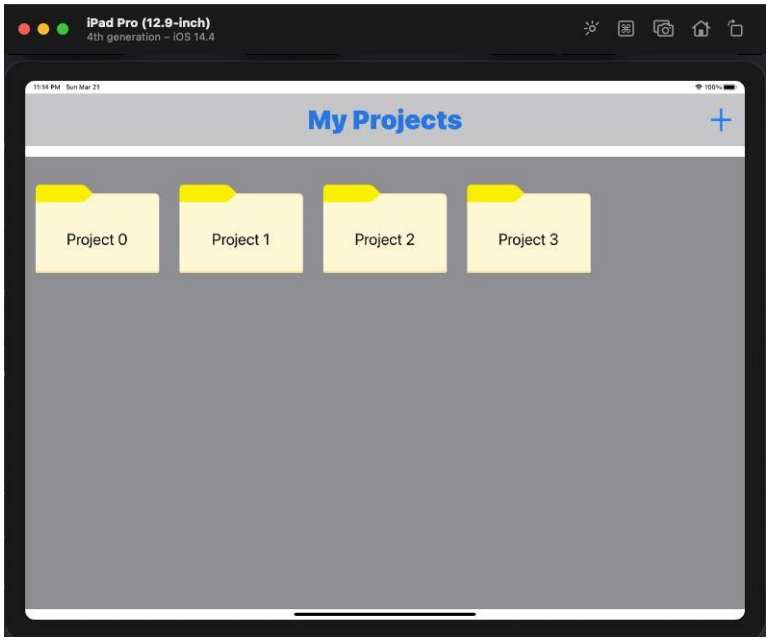
- User can draw on the canvas
- Tools on the canvas
    - Pens of different colour
    - Ruler

User Manual

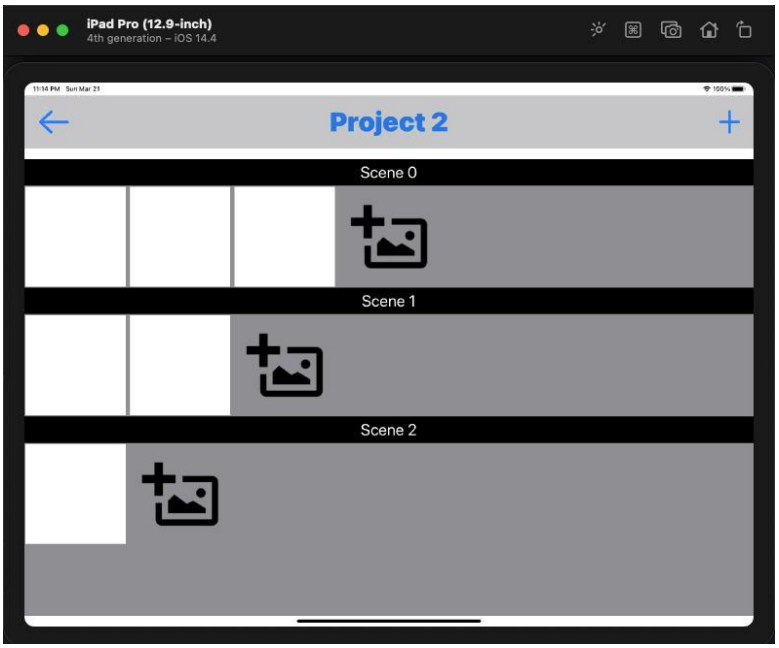
Environmental Assumptions:

- The app will be run on an iPad with iOS 14.4

1. Start the app
2. Select an existing project or create a new project by clicking the “+” button



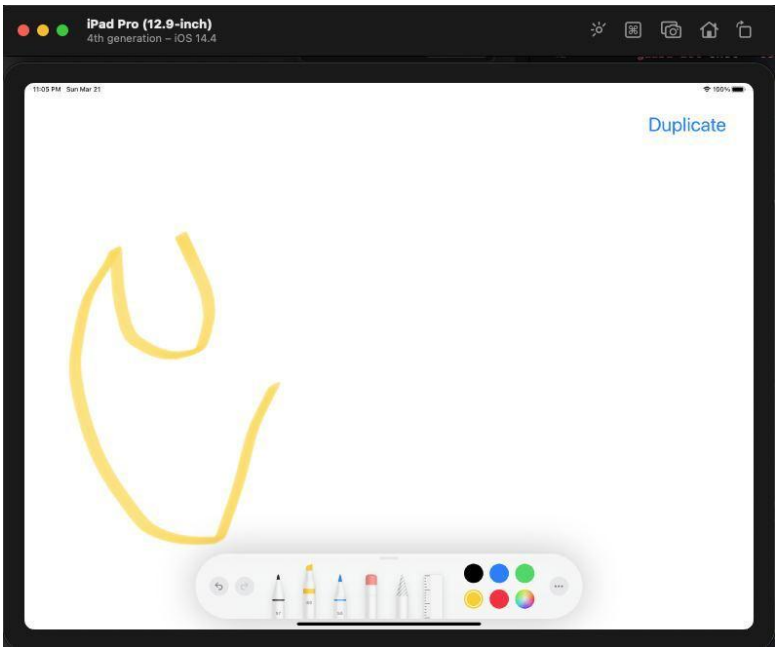
3. Once the user enters a project, all the scenes and shots in the project will be shown, and the user can select a shot to edit



   a. Click the top right “+” button to add a new scene

b. Click the “+” button behind the shots in a scene to add a new blank shot to the scene
c. Click the back button to return to the projects screen

4. Start drawing



a. Click duplicate to create and add a copy of the shot into the same scene
b. Toggle the settings on the toolkit.
    i. Drawing tools: Pen, Marker, Pencil
    ii. Ruler
    iii. Eraser
    iv. Colors
    v. Lasso Tool
c. Your work will be continuously saved as you draw or make other changes
d. Swipe from the left of the screen to return to the shots screen

Design and Architecture

Top-Level Organization

The design of the app follows the MVC pattern using Swift and UIKit. The codebase can be largely categorized into four parts: Model, Views, ViewControllers, and Storage.

Model

Files that form parts of the Model, which can be structs or enums, were stored under the Model folder. Model also includes a ModelManager that serves as an API for better separation of concerns between the Model and UI. This further helps us in separating our tasks by defining a contract of exposed methods beforehand.

Storage

Storage consists of two classes, namely StorageManager and StorageUtility. StorageManager is responsible for the encoding and decoding of Project structs to and from JSON string, while StorageUtility contains a group of functions that the StorageManager uses to retrieve or update/add JSON files.

View + ViewController

View classes are grouped together with their corresponding ViewController classes and other closely related View classes. We grouped them into Drawing, Project, and Scene components. The Drawing component contains View and ViewController classes that define the user's drawing experience. The Project component contains View and ViewController classes that are responsible for rendering the projects directory screen for the user to select a project. The Scene component contains View and ViewController classes that are responsible for rendering the screen containing a directory of shots, categorized by scene.

Interesting Design Issues

Design Consideration - *how to represent different types of layers*

1. Option 1 - Use separate protocols or use inheritance: one way to solve the problem is using separate protocols: create a protocol for each layer (e.g. `DrawingLayer`, `TextLayer`, `ImageLayer`) and let corresponding layer classes implement those protocols; another similar way is using inheritance: we have a general `Layer` class, and have different subclass for each layer(e.g `DrawingLayer extends Layer`)
    a. Pros
        i. each protocol/inheritance will only need to implement methods related to one layer
        ii. layers are not forced to implement any methods or contain any attributes that are not related to the layers (e.g. a `DrawingLayer` don't have to have an attribute named `text`)
    b. Cons
        i. the hierarchy could be complicated after more types of layers are added
        ii. need to create new protocols/subclasses for all additional layers
        iii. still need to be downcasted (using `as?`) if some operations require the specific type of the layer (e.g. when the user tap on the screen, text layer and drawing layer should respond differently)
2. Option 2 - Use enumeration and composition (current choice): Another approach is to use an enumeration `LayerType` and directly embed it in the `Layer` class
    a. Pros
        i. avoids over-complex hierarchy
        ii. a layer is guaranteed to have a layer type
        iii. easier to add more layers
    b. Cons
        i. could result in many switch cases
        ii. relies heavily on the `LayerType` as every shape has the same types of attributes (e.g. `PKDrawing` AND `text`). If the `LayerType` is not checked, a drawing layer might be misused as a text layer
3. Option 3 - Use Decorator Pattern: we can also make `Layer` an interface and have `ConcreteLayer` and `LayerDecorator` classes that implement the `Layer`,

where LayerDecorator will store a Layer and behaves like a Layer by calling methods of the stored Layer. In this way, we can have layers stack on top of each other, which is perfect for merging layers

a. Pros
    i. very suitable for merging layers into one layer
    ii. attribute/method will automatically change after putting Decorators on layers (e.g. after merging a drawing layer with an image layer, the bounding box of the result layer will be a bigger one that surrounds both the image and the drawing)
b. Cons
    i. need to have a basic ConcreteLayer. However, there is no layer suitable for this job (it cannot be layers like drawing layer because image layer's base ConcreteLayer should not be a drawing layer; it also makes no sense to make the ConcreteLayer a layer with no attributes)
    ii. hard to delete a specific Decorator from the stacks of Decorators (e.g. the user only want to delete the image in a merged layer, but not the drawing)
    iii. one has to pay extra attention to the order of the Decorators stack, which introduce another layer of complexity

We chose the second option in sprint 1. We did not use option 3 mainly due to complexity. In future sprints, we might want to find a way to incorporate the Decorator pattern if we want to implement merging layers. Moreover, when comparing options 1 and 2, there is no significant performance difference between the two options. Same as option 2, option 1 still needs to have lots of as? and if in the context where layer types do matter, and the magic of polymorphism is not shown. There is no need to jump between different files to implement the same method for different layers, and Xcode will automatically check whether there is a switch case that is not exhaustive.

Changes to be made

1. In order to implement resizing and rotating the canvas, we might need to add a background view to Shot Designer as a backdrop for the canvas.
2. We might need to incorporate the Decorator pattern to support layer merging

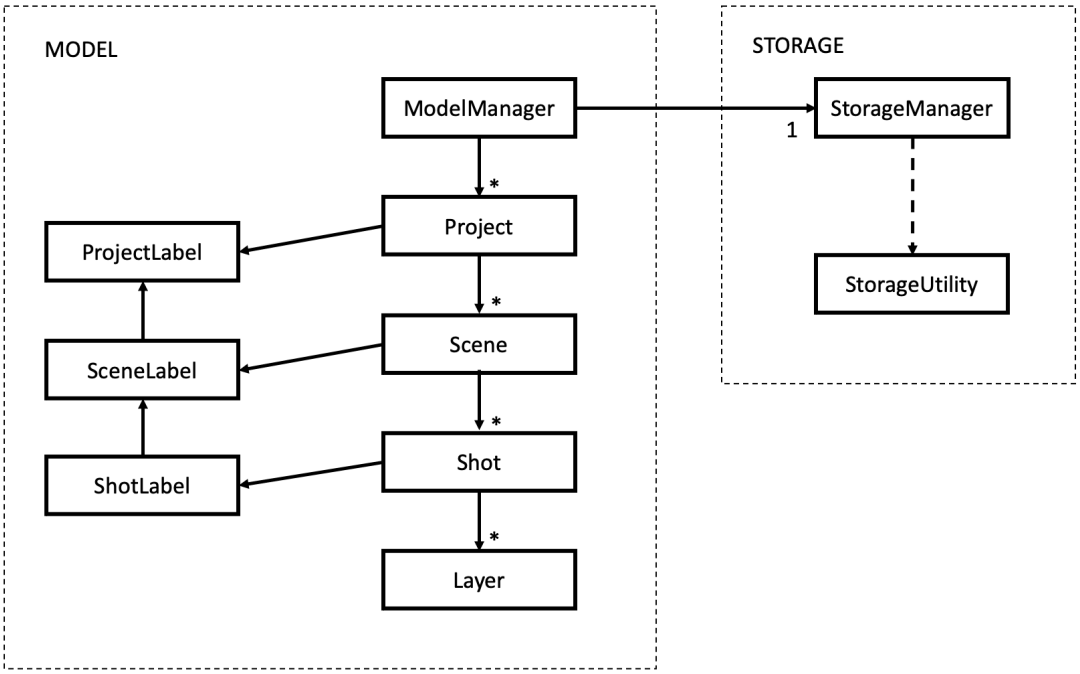
Runtime Structure

The underlying structure is rather straightforward: we have an array of projects, containing an array of scenes, containing an array of shots, containing an array of layers. The reason why we use Array instead of Set is:

1. It is more suitable for indexing (i.e. locating the desired element and retrieve it), which is crucial to CollectionView
2. The user should be able to duplicate shots/layers. Additionally, the user might need repeated scenes with the same title and shots (to increase tension for example). (we could make project uniquely identified by its title and store a Set of projects, but decide to still use Array to store projects due to the first point)

In order to uniquely identify any shot/scene/project, we used to label objects (ShotLabel, SceneLabel, and ProjectLabel). A shot has a ShotLabel which additionally keeps track of which scene it belongs to by storing a SceneLabel. SceneLabel further stores information about which Project it belongs to by storing a ProjectLabel. This way, without having any reference to the parent containers, we are still able to identify the path to a shot using only ShotLabel.

Module Structure



For the Model, we used the Facade pattern. The ModelManager acts as an entrypoint for ViewControllers to retrieve and update the Model.

There are several reasons why we decided that this is the most appropriate:

1. Simplifies the division of tasks by defining an interface between which Model structs and View+ViewController classes interact
2. Decouples the View+ViewController from the Model, so changes to the Model's internal implementation would not require changes in View+ViewController as long as the API requirements remain the same
3. More defensive as modifications made would go through the ModelManager first, which further simplifies debugging and error tracing

The ModelManager also handles persistence and continuous saving of changes made in the shot layers. It keeps a private StorageManager object and has a private saveProject() method. This allows for better access control, as only the ModelManager is able to alter the persistence storage.

Similarly, for separation of concerns, the Storage component is divided into two classes, each with its own responsibility. The first one, StorageManager, is responsible for converting Project objects, which implement Codable, into JSON

strings and vice versa. It calls one or more functions from StorageUtility, which is responsible for the actual read and write operations and acts as an interface between the codebase and the storage file directory.

Additionally, to inform `ProjectViewController` and `SceneViewController`, we use the Observer Pattern: the `ModelManager` keeps an array of observers which implements `ModelManagerObserver`. Each `ModelManagerObserver` will implement the method `modelDidChange()`, which will be called when the model changes. The two controllers `ProjectViewController` and `SceneViewController` implement the observer protocol and will thereby get refreshed every time the model is changed.

Testing

For testing of our current features, we intend to have a combination of unit tests and integration tests. The unit tests will mainly be used in the Model, while the integration tests would involve automated UI testing of the sprint product. Details of specific tests can be found in the appendix.

Reflection

Evaluation

We are behind schedule after the first sprint. For example, we have yet to implement reordering shots. Although we split the work before the start of the first sprint, we committed a huge mistake of not specifying our API beforehand, which defines how our ViewController communicates with the Model. Consequently, the person in charge of Model is waiting for others to tell him what API he needs to expose, and the person in charge of Navigation is waiting for the Model to create methods inside controllers. In the end, we each have our dummy class together with different APIs, which brings us to the next point: we should have started integration earlier. If we start integration sooner and agree on the interface, not only API issues will be fixed earlier, problems such as version control will also be discovered sooner.

Lessons

1. List all the required API at the beginning of the sprint to avoid API inconsistency (have meeting to discuss if necessary)
2. Set deadlines and buffers: In the following sprints, we plan to have the first integration in the first weekend, and we will have a three-day buffer at the end of the sprint

Known Bugs and Limitations

Duplicate buttons fail to render sometimes. However, this is only a temporary issue as we will eventually add a navigation bar.

Appendix

Test cases

Unit Tests for Models

1. LayerTests
    a. setDrawingTo(_ updatedDrawing: PKDrawing)
        i. Verify that drawing is updated to ‘updatedDrawing’
2. ShotTests
    a. updateLayer(_ layerIndex: Int, withDrawing drawing: PKDrawing)
        i. If layerIndex is negative, nothing must be changed
        ii. Else If layerIndex >= self.layers.count, nothing must be changed
        iii. Else, must update self.layers[layerIndex]’s drawing to ‘drawing’
    b. addLayer(_ layer: Layer)
        i. Verify that layer is appended to the end of self.layers
3. SceneTests
    a. updateShot(ofShot shotLabel: ShotLabel, atLayer layer: Int, withDrawing drawing: PKDrawing)
        i. If shotLabel.shotIndex >= self.shots.count, nothing must be changed
        ii. Else if shotLabel.shotIndex < 0, nothing must be changed
        iii. Else, shots[shotIndex].updateLayer(layer, withDrawing: drawing)

1. Test outcome follows 1a.

b. addShot(_ shot: Shot)
    i. Verify that shot is appended to the end of self.shots
c. addLayer(_ layer: Layer, to shotLabel: ShotLabel)
    i. If shotlabel.shotIndex >= self.shots.count or shotlabel.shotIndex < 0, nothing change
    ii. Else layer must be appended to the end of shots[shotLabel]'s layers.

4. ProjectTests
    a. updateShot(ofShot shotLabel: ShotLabel, atLayer layer: Int, withDrawing drawing: PKDrawing)
        i. If shotLabel.sceneIndex >= self.shots.count, nothing must be changed
        ii. Else if shotLabel.sceneIndex < 0, nothing must be changed
        iii. Else, scenes[sceneIndex].updateLayer(layer, withDrawing: drawing)
            1. Test outcome follows 2a
    b. addScene(_ scene: Scene)
        i. Verify that scene is appended to the end of self.scenes
    c. addShot(_ shot: Shot, to sceneLabel: SceneLabel)
        i. If sceneLabel.sceneIndex >= scenes.count or < 0, nothing must be changed
        ii. Else, Verify that shot is appended to the end of scenes[sceneLabel.sceneIndex].shots
    d. addLayer(_ layer: Layer, to shotLabel: ShotLabel)
        i. If shotLabel.sceneIndex >= scenes.count or < 0, nothing must be changed
        ii. Else, addLayer(layer, to: shotLabel)
            1. Test outcome follows 2c

UI Integration Tests

1. Starting the app should lead to a directory of saved projects
    a. Selecting the plus sign at the top right should add a new empty project
2. Selecting a project grid should lead you to a directory of shots categorized by scenes of a project
    a. Selecting the plus sign at the top right should add a new empty scene
    b. There should be one “add new shot” button at the end of the shots in each scene. Clicking this should add a new shot to the end of the scene and open the canvas.
3. Selecting a shot should lead you to the canvas view
    a. Drawing experience should match the settings selected on the toolkit as per the user manual.

Detailed Schedule + Task List

Continue implementing basic features that were left unfinished after the first sprint, such as reordering of shots and customising project titles. Responsibility for this is divided into sections of the codebase.

Marcus: Model
Pan Yongjing: View + ViewController
Tian Fang: Drawing Experience

We also intend to implement the features initially planned for the second sprint, but this is subject to further discussion at the start of the sprint. They are as follows:

- Undo and Redo
- Automatic Labelling
- Layers
- Rotate and resize canvas
- Preview previous or next shots
- Search feature