CS3217 Project Sprint 3 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

Project Navigation

- Create Projects
- Delete Projects
- Rename Projects

Scene/Shot Navigation

- Add Scenes
- Delete Scenes
- Add Shots
- Rearrange shots
- Set Background Color

Canvas

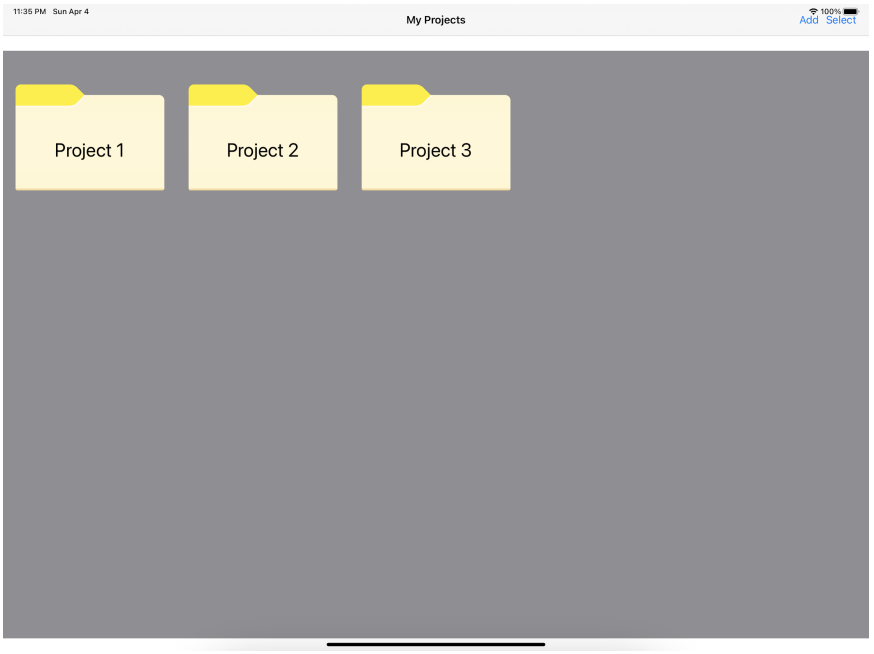
- Draw on the canvas
- Use various tools on the canvas
    - Different colored pens
    - Lasso Tools
    - Ruler
- Work with layers on the canvas
    - Add layers
    - Edit layers
    - Remove layers
- Duplicate shot
- Onion Skin

User Manual

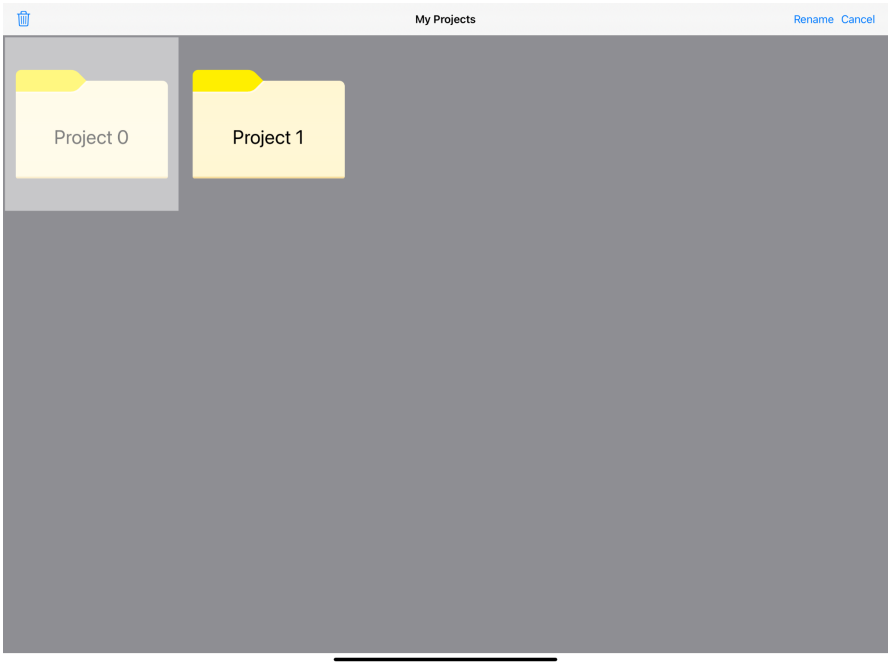
Environmental Assumptions:

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

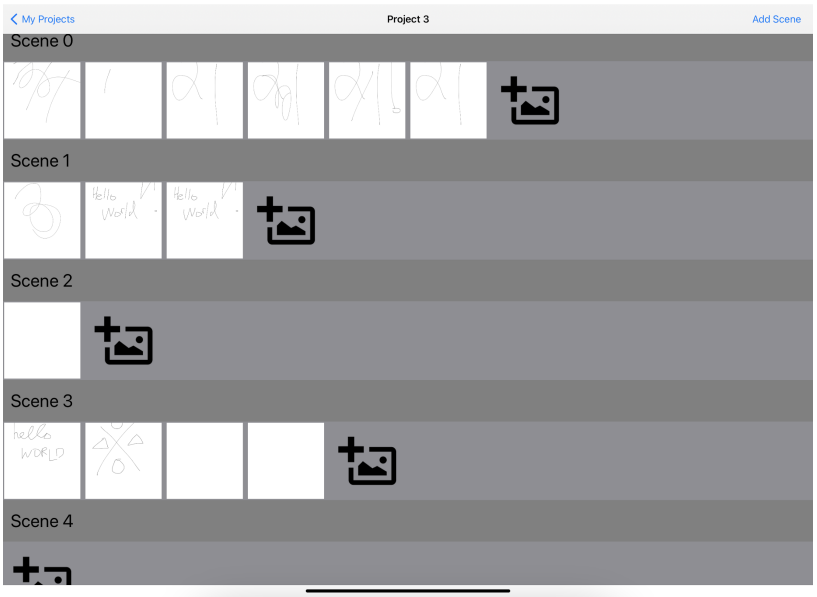
1. Start the app
2. Select an existing project or create a new project by clicking the “+” button



a. Click the top-right Select button to select projects to delete, or select a single project to rename



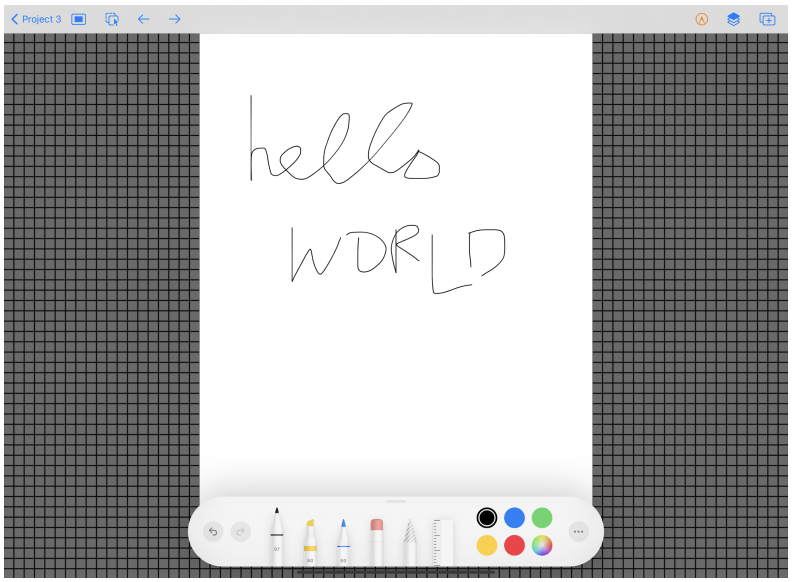
3. Once the user enters a project, all the scenes and shots in the project will be shown, and the user can select a shot to edit



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
d. Long press to reorder shots within a single scene

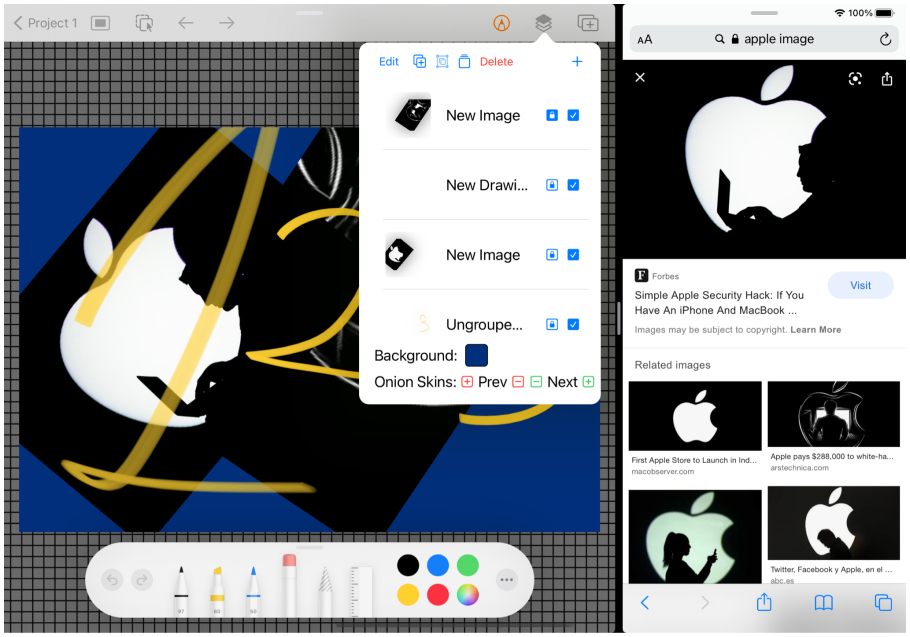
e. Pressing on the DELETE button at the top right of each scene header deletes the scene.

4. Start drawing



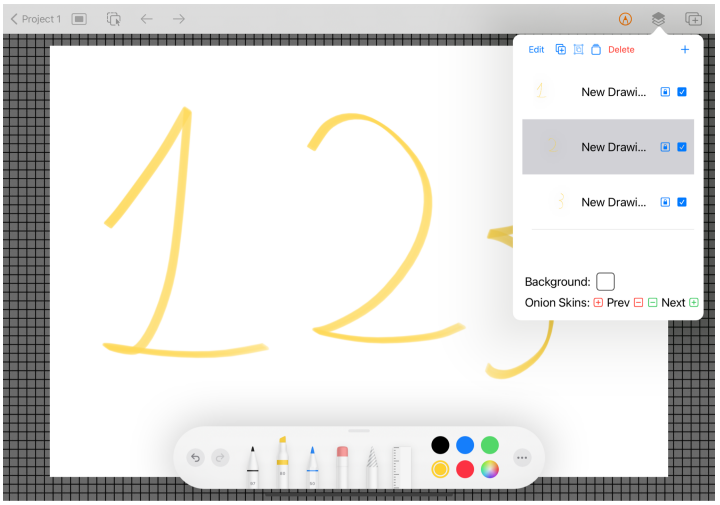
a. Click the rightmost button to create and add a copy of the shot into the same scene
b. Toggle the settings on the toolkit.
    i. Drawing tools: Pen, Marker, Pencil
    ii. Ruler
    iii. Eraser
    iv. Colors
    v. Lasso Tool
c. Your work will be continuously saved as you draw or make other changes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

5. Working with Layers



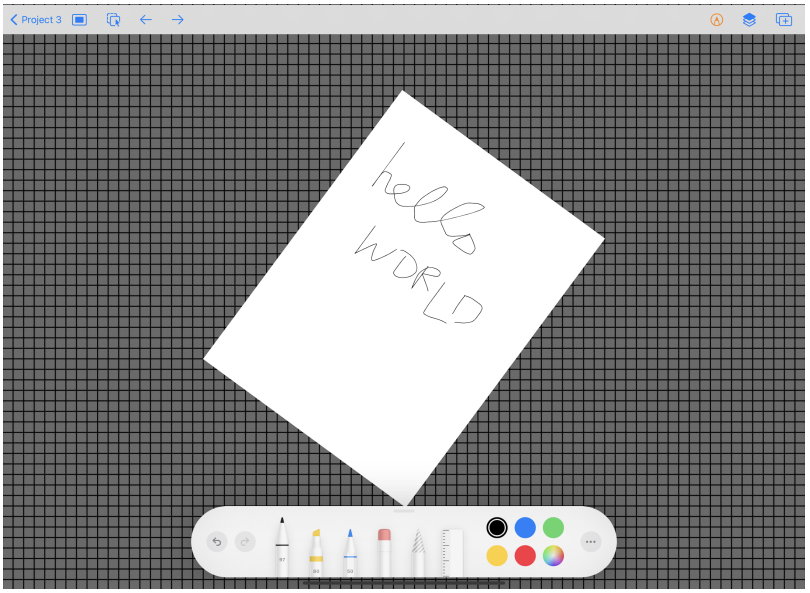
Each shot has several layers stacked on top of each other.

a. The second button from the right in the navigation bar allows you to see your layers
b. Select a layer if you wish for additional drawing on your current shot to be saved to the layer



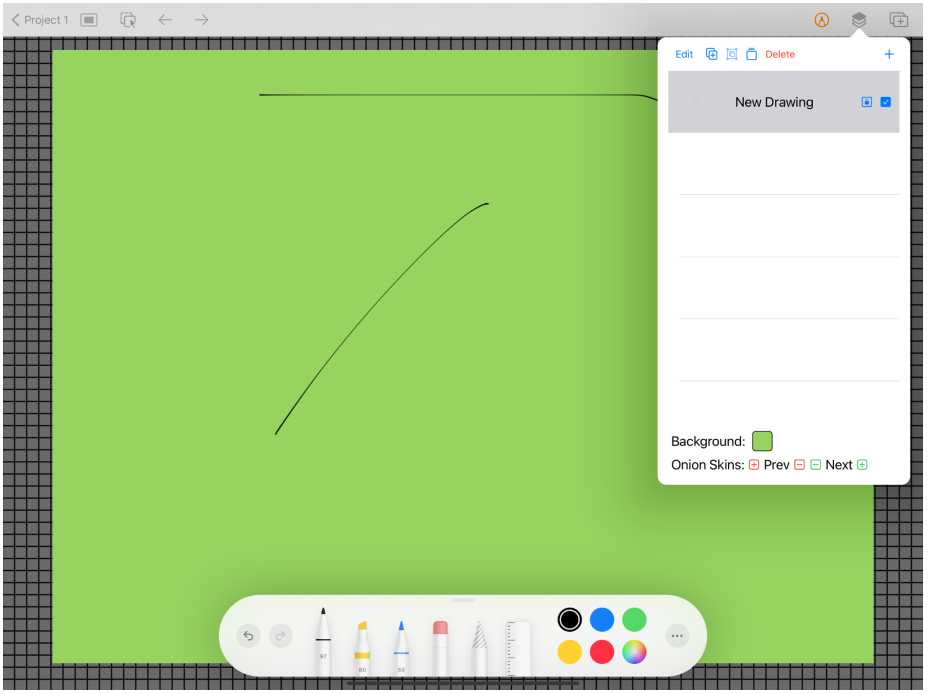
c. You can untick the checkbox in each layer to hide it
d. You can click Edit and drag layers to reorder them
e. You can click Edit, select layers, and delete them

6. Resizing, rotating, and translating canvas/layer



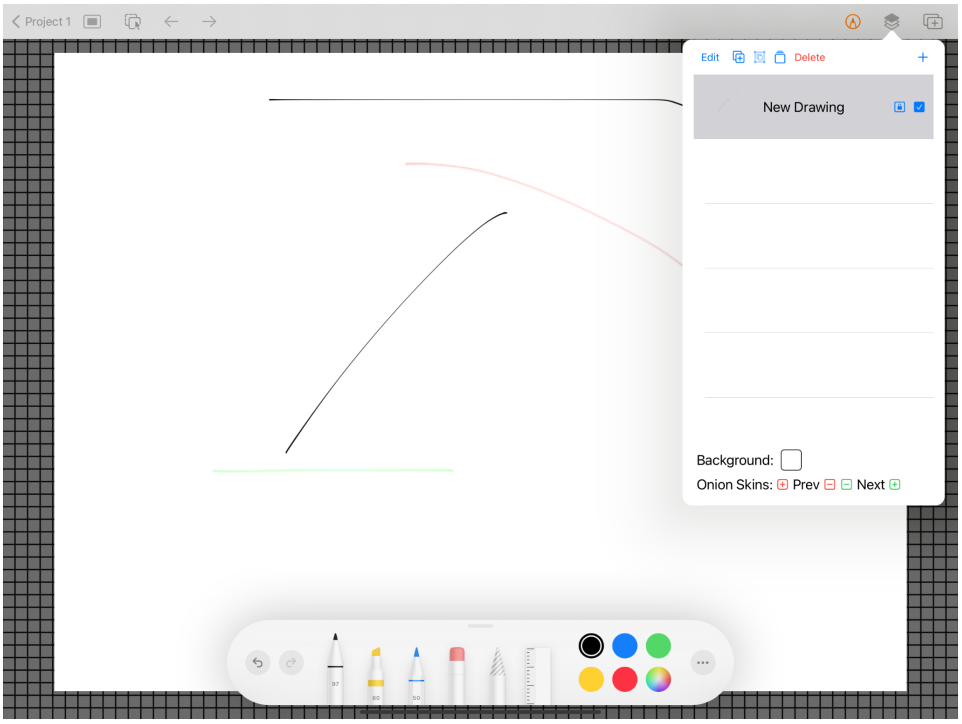
   a. To resize the canvas, simply pinch on your canvas.
   b. To rotate the canvas, you can use two fingers dragged in opposite directions
   c. Rotating and resizing be performed simultaneously
   d. Canvas can be translated using two fingers
   e. Click the second button from the left which allows you to rotate/resize/translate the current layer's drawing instead of the canvas
   f. Click the leftmost button to reset any rotation made to canvas or layer drawing.
7. Navigation
   a. You can navigate from the current shot to the next or previous shot within a scene using the left and right arrows on the navigation bar

8. Changing background color



a. Click the second button from the right at the top of the canvas
b. Click the background color button to change the background color of the shot

9. Onion Skin



a. Click the second button from the right at the navigation bar

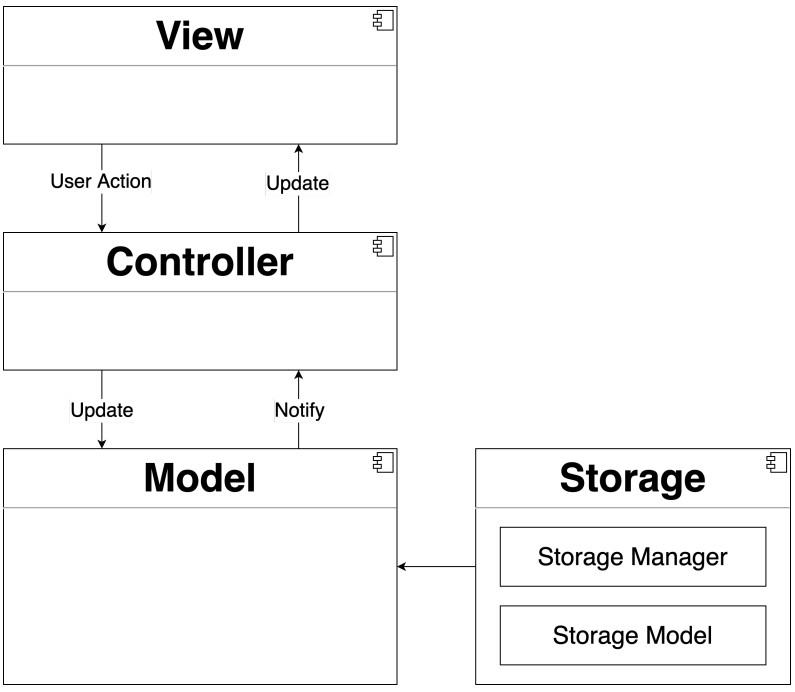
b. Click the previous (red) / next (green) PLUS onion skin button to display a semi-transparent outline of the previous / next shots (Number of click = one additional shot before/after if any)
c. Similarly, MINUS button to hide one additional shot before/after if any is displayed.

10. Grouping Layers
    a. Click Edit
    b. Select layers you wish to group
    c. Select the button to the right of Edit (“Group” button)
11. Ungrouping Layers
    a. Select a layer
    b. Click the Button to the right of “Group” button (“Ungroup” button)

Design and Architecture

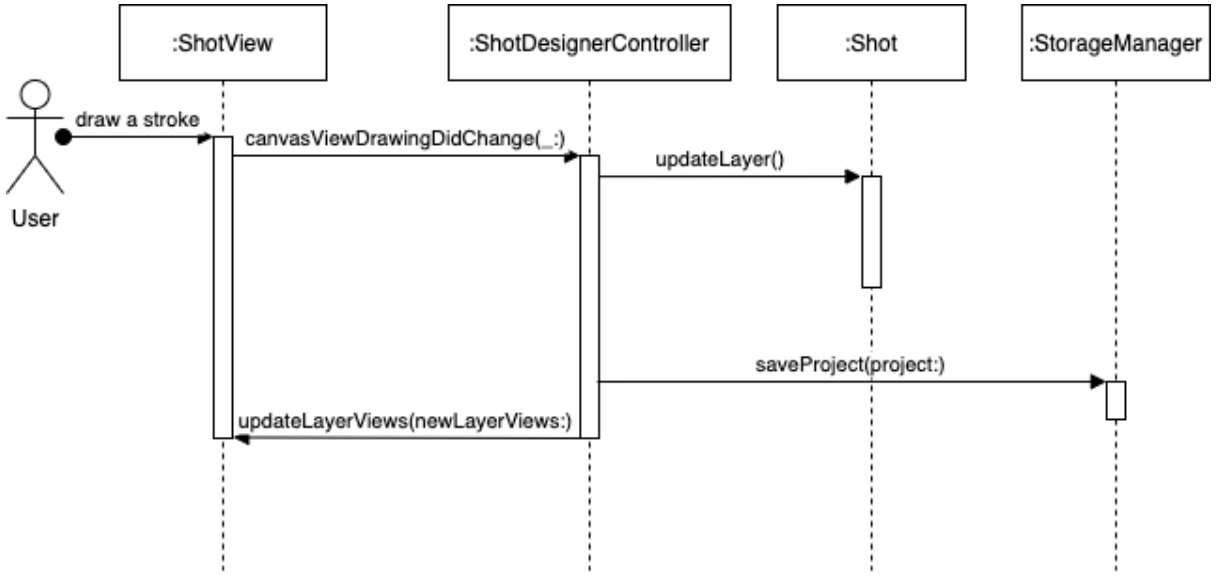
Top-Level Organization

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



Interactions between Architecture Components

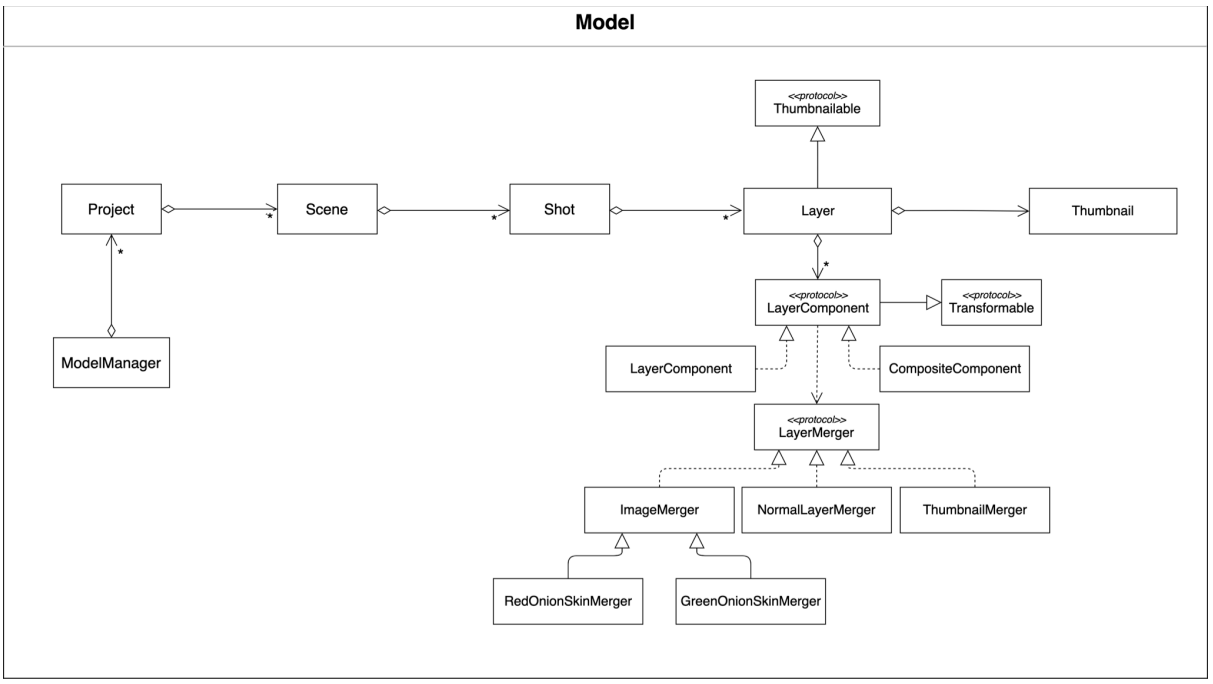
The *Sequence Diagram* below shows how the components interact with each other when the user draws a stroke:



Given below is a quick overview of each component:

- Storage
    - Handle project loading and project saving
- Model
    - Represents and stores data related to projects
    - Stores information related to Project, Scene, Shot, Layer.
- Controller
    - serves as the bridge between Model and View
    - accepts user inputs and update Model accordingly
    - updates View when Model changes
- View
    - The representation of data on the user interface.
    - Free from any contain any domain logic

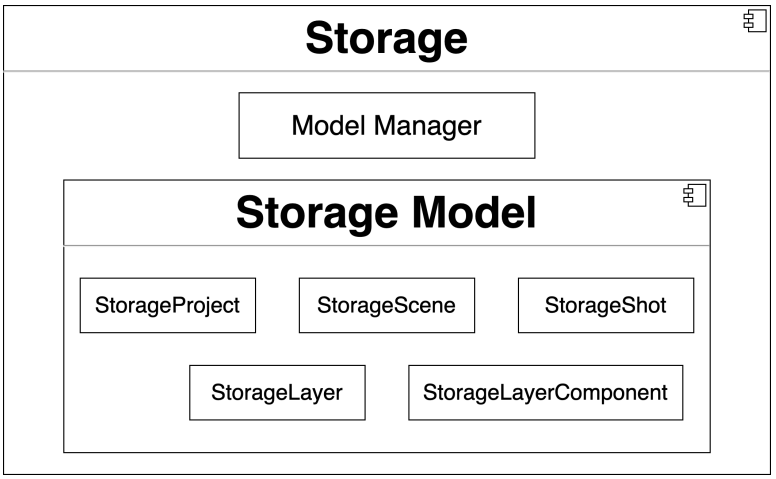
Model



As discussed during our previous sprint review, our model now uses classes (reference-type) instead of structs (value-type). This way, updating the model does not always have to go through a recursive hierarchy from the top-level (Project) to the edited component, as a reference to the relevant object can be kept immediately. Furthermore, ModelManager is no longer the single entry point for the Model. Instead, the relevant ViewController will be directly altering the relevant component of the model through the reference it keeps.

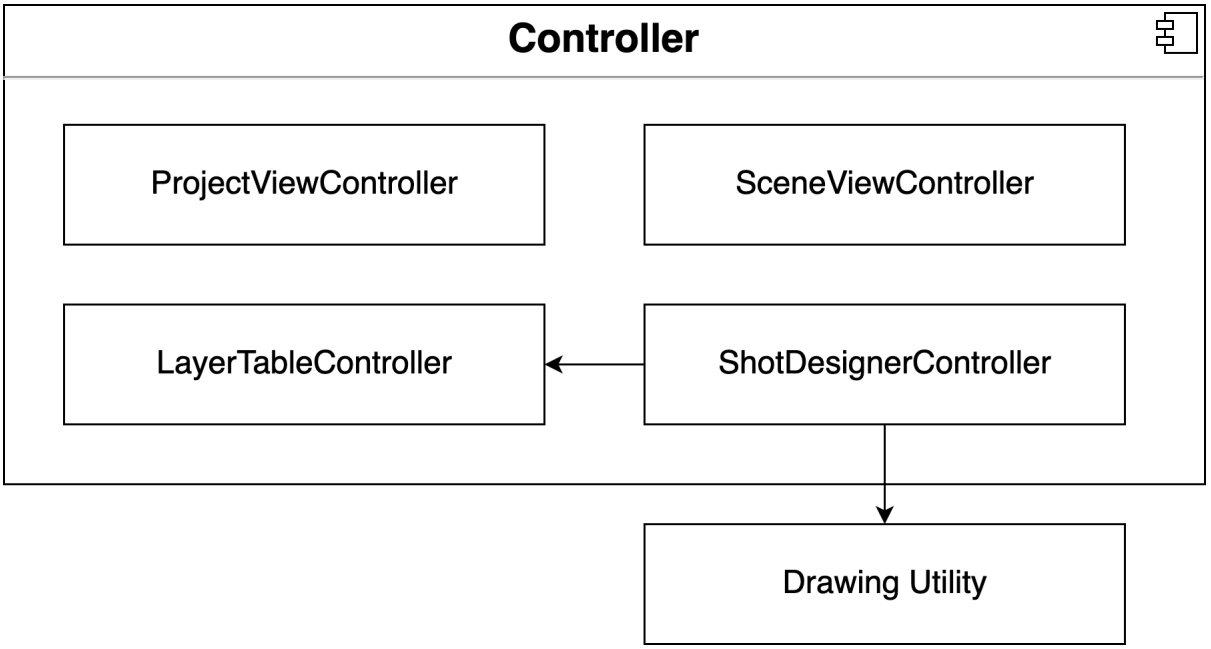
The main construction of the model flows the hierarchy **ModelManager > Project > Scene > Shot > Layer**.

Storage



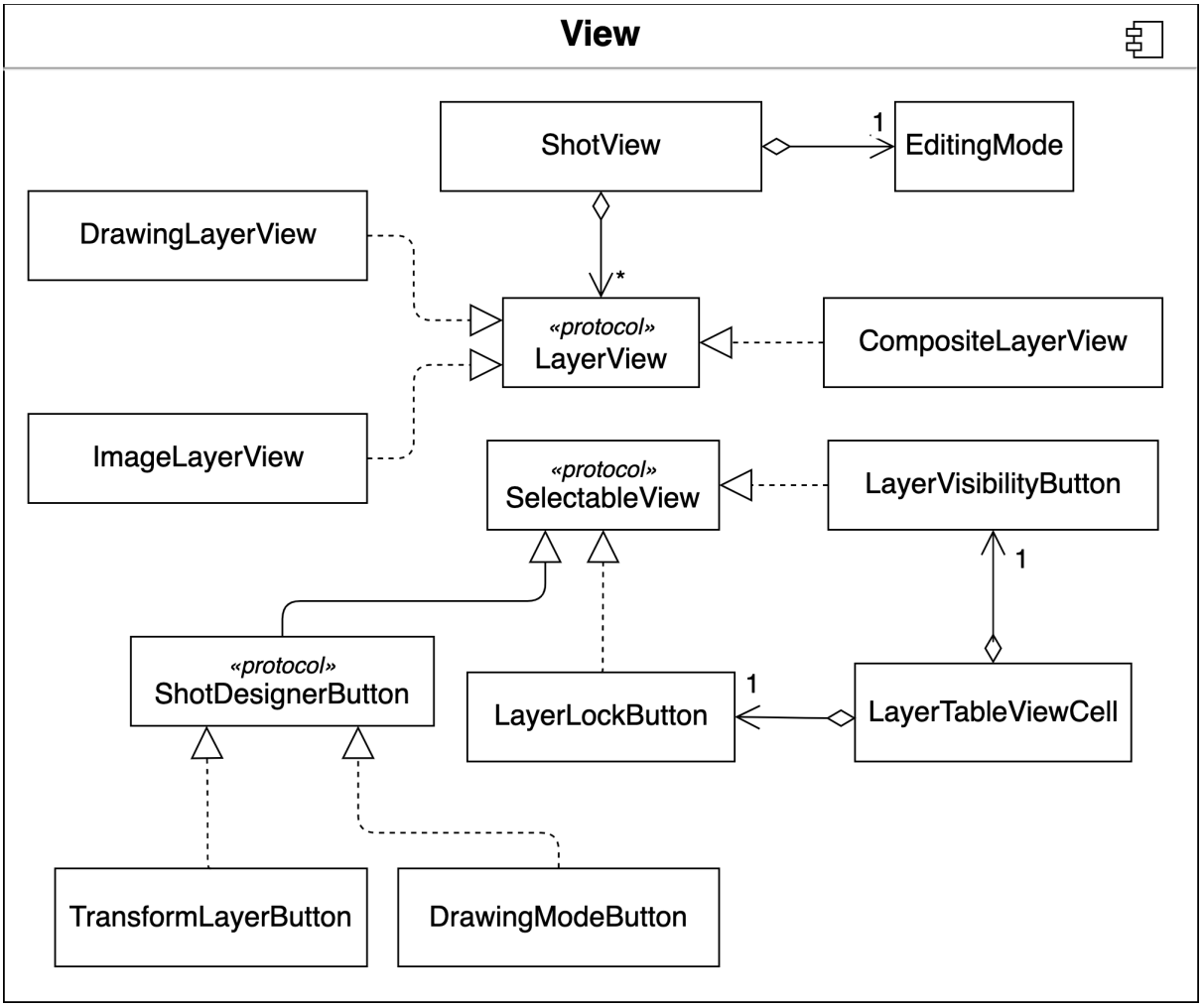
Storage has a Storage Model corresponding to the real Model. In this way the classes in the Model are not forced to implement the Codable protocol. Additionally, we have a `StorageManager` to expose methods for external usage. We are currently using Codable for our storage (i.e. classes in the Storage Model are Codable) and we are storing the user data in JSON format.

ViewController



The Project View Controller is responsible for the project folder interface that the user interacts with. The Scene View Controller gives users access to the scenes and shots of each project, and responsible for actions such as adding and organizing shots. The ShotDesignerController is the primary View controller that deals with actions on the canvas. Additionally, The LayerTableController will be presented as a popover when the user clicks the “Layers” button, and will update ShotDesignerController about activities such as “toggle layer lock” using Delegate pattern so that ShotDesignerController can act accordingly.

View



The collection view cells of Project/Scene navigator are omitted as they are just standard dynamic collection view cells. For views in the drawing part, they mimic the layer representation in the Model: ShotView contains an array of `LayerView` which is a protocol to be implemented by concrete `LayerView`. We also have another simple protocol `SelectableView` for Buttons that change their appearance when the state changes.

Interesting Design Issues (Design Considerations mentioned before will not be repeated)

Design Consideration 1 - *how to save to storage and generate shot thumbnail*

As soon as we add support for image layers, the speed problem for thumbnail generation and storage saving emerges. Specifically, when there are many complicated shots, synchronously saving them to storage or generating all

thumbnails could take several seconds, which is unacceptable for users. Hence, we've created two separate DispatchQueue to cope with this issue. See "Module Structure" for more details.

Design Consideration 2 - *how to store and update the transform*

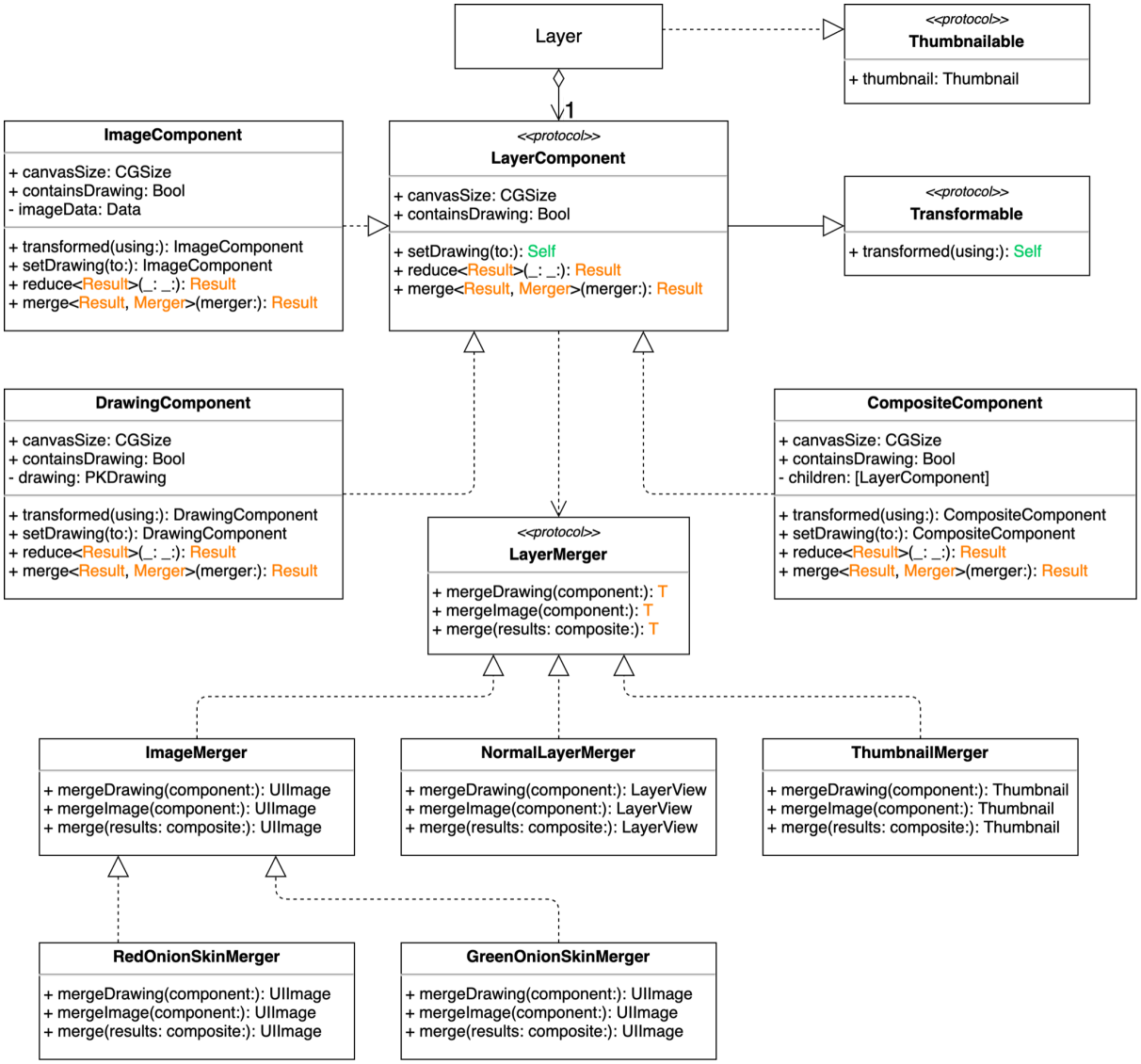
In sprint 2, we spent a lot of time on getting layer transformation to work properly. Two major difficult tasks encountered are applying transform around the correct anchor point and updating `PKCanvasView` (`PKCanvasView will refresh every time its PKDrawing is transformed`). To solve the issue of anchor point, we decided to make the anchor point the center point of the canvas. Moreover, `PKCanvasView` is only refreshed at the end of transform gestures. See "Runtime Structure" for more details.

Runtime Structure

Instead of having an array of UUID and a dictionary that maps UUID to the entities, we now have an array of projects, containing an array of scenes, containing an array of shots, containing an array of layers. Since the entities are classes, we can immediately get a unique reference to the object without the need of unique IDs / labels. 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).

Layer Structure



We've further refined the Layer structure in sprint 3. Specifically, instead of letting `CompositeComponent` keep a transform, we apply the transform to all of the leaf nodes of that `CompositeComponent` (so there is no need for `CompositeComponent` to keep a transform property). Additionally, to facilitate the process of generating thumbnails, we've added a new `Thumbnail` class that keeps data of various thumbnails (e.g. regular thumbnail, red onion skin thumbnail)

The main Layer structure is the same as that of sprint 2 (i.e., use the **composite pattern** together with the **visitor pattern**):

1. We chose the Composite pattern for Layer Structure, as shown above in the Layer diagram (note that the orange types are generic types/associated types), specifically:

    a. The LayerComponent protocol describes operations that are common to both simple(leaf) and complex(composite) elements of the layer component tree.

    b. The Leaf Component like DrawingComponent is a basic layer component of the tree that doesn't have sub-components. Leaf components usually do most of the real work, since they don't have children to delegate the work to.

    c. The CompositeComponent is a layer component that has sub-components: Leaf Components or other CompositeComponent. A CompositeComponent doesn't know the concrete classes of its children. Consequently, it works with all sub-elements only via the component interface. In the methods of CompositeComponent, it delegates the work to its sub-components, processes intermediate results, and then returns the final result to the client.

    d. Note that for the client, it only works with the LayerComponent protocol, and from its perspective, there is no difference between leaf components and CompositeComponent

2. While the tree structure is very elegant for internal operations such as `setDrawing`, it is not easy to inspect and make use of the structure from the outside. For example, layer components should not have any knowledge about how to generate corresponding `LayerView`. To address such problems of separating the Model from other logic, we make use of the **Visitor** pattern. Take generating `LayerView` as an example, in order to separate the UI elements from the Model, we will encounter the problem that external clients only work with `LayerComponent` through the protocol and they have no information about whether a component is a composite or leaf component. To avoid typecasting, we introduce the Visitor pattern through `LayerMerger` protocol to retain the magic of polymorphism, specifically:

a. The `LayerMerger` protocol declares a set of visiting methods that can take concrete `LayerComponent` such as `DrawingComponent` as arguments. The `LayerMerger` has an associated type `T`, which will be the return type for each method.

b. Each Concrete `LayerMerger` such as `NormalLayerMerger` implements several versions of the same behaviors (in this case the merge method) for different concrete `LayerComponent`. Note that a concrete `LayerMerger` should “merge” only 1 type of thing as shown in the `LayerMerger` protocol (in the case of `NormalLayerMerger`, the associated type T is now `LayerView`, so `NormalLayerMerger` will merge the layer in a way that will produce a merged `LayerView`)

c. The `LayerComponent` declares a method for “accepting” visitors (in this case the generic method `merge<Result, Merger>(merger: )`). The purpose of this method is to redirect the call to the proper visitor’s method corresponding to the current `LayerComponent`. In this way, the polymorphism magic is retained (this technique is called “Double Dispatch”)

d. In this way, concrete `LayerComponent` and concrete `LayerMerger` are separated through the `LayerMerger` protocol, and they don’t have to know each other’s concrete type to produce the result.

3. The way we apply transforms to layers is changed in sprint 3 to solve the two issues encountered in sprint 2. To solve the issue of anchor point, we decided to make the anchor point the center point of the canvas. In this way, the anchor point is unchanged after grouping, and transform is passed down the LayerComponent tree and directly applied to the leaf components through the `transformed(using:)` method. Moreover, to cope with the issue of updating `PKCanvasView`, we transform the corresponding LayerView when the transform gestures have not ended, and update the model only when those gestures end. In this way, the `PKDrawing` in the `PKCanvasView` is only updated at the end of transforms, and hence will only be refreshed once.

Module Structure

The ViewController sits between View and Model. It renders information from Model into View, and updates the relevant Model entity accordingly. This was done by changing the model from structs to classes, allowing the ViewController to get a reference to the Model entities. Furthermore, since we are able to get a unique reference to each entity, we no longer need UUID properties nor Label entities (i.e. ShotLabel, SceneLabel, etc.).

The ModelManager 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 into StorageProject objects, which implement Codable, and then into JSON strings for storage, 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.

Moreover, to inform `ProjectViewController`, `LayerTableController`, 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. Controllers such as `ProjectViewController` implement the observer protocol and will thereby get refreshed every time the model is changed.

Furthermore, `LayerTableController` uses the Delegate pattern to update `ShotDesignerController`. `ShotDesignerController` will set itself as delegate of `LayerTableController`, and `LayerTableController` will call the corresponding method of this delegate when there is any change to the layers.

Storage Module

As mentioned in the “Runtime Structure”, since we are using the Composite pattern, the storage becomes much more difficult since we cannot just simply let Swift Compiler to auto synthesize Codable. The reason for this is that now a composite component stores an array of `LayerComponent` which is a protocol which is not Codable (Note that let `LayerComponent` to extend Codable won’t work). Therefore, we’ve decided to separate the storage logic and created a dedicated StorageModel for storage. Of course, since the Storage module is an external module to the Model module, we still have the problem of knowing which concrete `LayerComponent` we are dealing with so that we can choose the corresponding encoding/decoding method. Hence, we use typecast to find the concrete type in the Storage Module. After finding the concrete type, it is stored as an associated value of an enumeration class `StorageNodeType`. After this, the rest of decoding/encoding is just retrieving data from/making nested Coding Containers.

Asynchronous Shot Thumbnail Generation and Storage Saving

To avoid UI blocking while entering the scene gallery of a project, we’ve made thumbnail generation and saving to storage asynchronous. This is crucial to shots with images as it usually takes a long time to generate images with transforms applied. Some worth mentioning details includes:

1. After the Model is updated, the process of storing the new Model is done asynchronously using the `storageQueue`
2. While thumbnail generation for shots is asynchronous, the thumbnail of a layer is generated synchronously when the layer is changed. Although it is definitely better if we can also asynchronously generate them on a background queue, thumbnail generation for images makes use of UIGraphicsImageRenderer and therefore has to stay on the main queue.
3. We keep an optional tuple `onGoingThumbnailTask` that contains the shot and its corresponding thumbnail generation `WorkItem`, or nil if there is no ongoing thumbnail task. If “generate thumbnail and save” method is called again, and there is still an ongoing thumbnail generation task for the same shot, that task will be canceled as the thumbnail will be generated using a new `DispatchWorkItem`. When the thumbnail generation is complete, it will

update the shot in the model, save it to the storage and make `onGoingThumbnailTask` nil on the main queue. The idea is similar for the `storageQueue`: if the same project is going to be saved, the previous redundant save-to-storage task will be canceled.

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

For this sprint, integration was started quite late and due to several significant changes required. This is especially the case when we have to refactor the core components of our application like the model and storage.

It took us a significant amount of time to complete the integration as constant rewriting of code and redesigning of system architecture has to take place during the process. Communication could have been improved to speed up our work, task division could have been optimized, and integration could have started earlier. Nevertheless, considering it is Week 13, we are satisfied that we could complete the project. Nonetheless, it has been an eye-opening experience to undergo a glimpse of the daily work and challenges of software engineering, and it has been a tremendously fruitful learning experience for us.

Lessons

1. When thinking of a solution, we should weigh whether it makes sense and compare it against alternative solutions. This is evident in our choice of keeping structs despite classes fitting our use case better and way more simple than labelling structs.

2. There are definitely rooms for improvement in our sprint planning. More details and more communication would have improved the efficiency of our work.

Known Bugs and Limitations

- The boundaries of the canvas after transformation is still shown
- The folders management may run into some performance issues when the number of shots increases.

Detailed Responsibilities

Responsibility is divided into sections of the codebase.

Model:

- Layer and its subcomponents: Tian Fang
- Multithreading (for thumbnail generation and storage): Tian Fang
- Rest of Model (i.e. Project, Scene, Shots etc.): Marcus and Yongjing

View + ViewController:

- Project/Scene Navigation: Yongjing
- Drawing (Shot Designer and Layer Table View): Tian Fang

Storage:

- Tian Fang

Appendix

Test cases

Unit Tests for Models

1. LayerTests
    a. setDrawingTo(_ updatedDrawing: PKDrawing)
        i. Return a layer with the updated drawing
    b. duplicate()
        i. Return a new Layer object with the exact same properties
    c. generateThumbnail()
        i. If isVisible, set self.thumbnail to empty thumbnail

        ii. Else, set self.thumbnail to a thumbnail of the layer's drawing

2. ShotTests
    a. updateLayer(_ layer: Layer, with newLayer: Layer)
        i. If layer exists, then it should be removed and replaced by newLayer at its current position
        ii. If layer does not exist, then no change
    b. moveLayer(_ layer: Layer, to newIndex: Int)
        i. If layer does not exist, do nothing
        ii. Otherwise, it should be moved to position 'newIndex'
    c. generateLayerThumbnails()
        i. Calls generateThumbnail() method for each layer of the shot, outcome for each follows 1c.
    d. addLayer(_ layer: Layer, at index: Int? = nil)
        i. If index is nil (or, no input for index parameter was provided), verify that layer is appended to the end of self.layers
        ii. Else, verify that layer is appended to the index
    e. setBackgroundColor(color: Color)
        i. The shot's backgroundColor should be set to color
    f. removeLayers(_ removedLayers: [Layer])
        i. None of the layers in the removedLayers should not exist in self.layers
    g. removeLayer(_ layer: Layer)
        i. If layer exists in self.layers, then it should be removed.
3. SceneTests
    a. updateShot(shot: Shot, with newShot: Shot)
        i. If `shot` exists, then it should be replaced with `newShot`
        ii. Else, nothing must be changed
    b. addShot(_ shot: Shot)
        i. Verify that shot is appended to the end of self.shots
    c. addShot(_ shot: Shot, at index: Int)
        i. Verify that shot should be added at index
    d. swapShots(_ index1: Int, _ index2: Int)
        i. Verify that the shots at both indices are swapped
    e. duplicate()

        i. Should return a new Scene object with the same properties as self.
    f. moveShot(shot: Shot, to newIndex: Int)
        i. If shot exists, should move shot to newIndex
        ii. Else, do nothing
    g. getShot(_ index: Int, after shot: Shot)
        i. Returns shot located `index` places after `shot` in self.shots
4. ProjectTests
    a. addScene(_ scene: Scene)
        i. Verify that scene is added to the end of self.scenes
    b. setTitle(to title: String)
        i. Verify that self.title is set to `title`
    c. duplicate()
        i. Should return a new Project object with the same properties as self.

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.
    b. Features like rotation/resizing/translation, onion skin, etc. must work according to user manual