CS3217 Project Sprint 2 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
- Add Shots
- Rearrange shots

Canvas

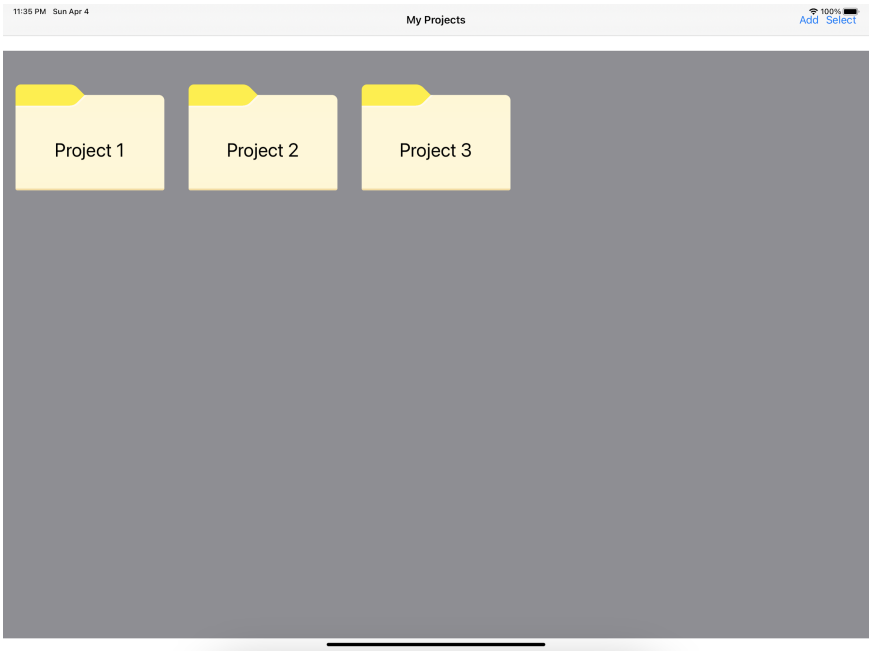
- Draw on the canvas
- Use various tools on the canvas
    - Different colored pens
    - Lasso Tools
    - Ruler
- Work with layers on the canvas
    - Add layers
    - Edit layers
    - Remove layers
- Duplicate shot

User Manual

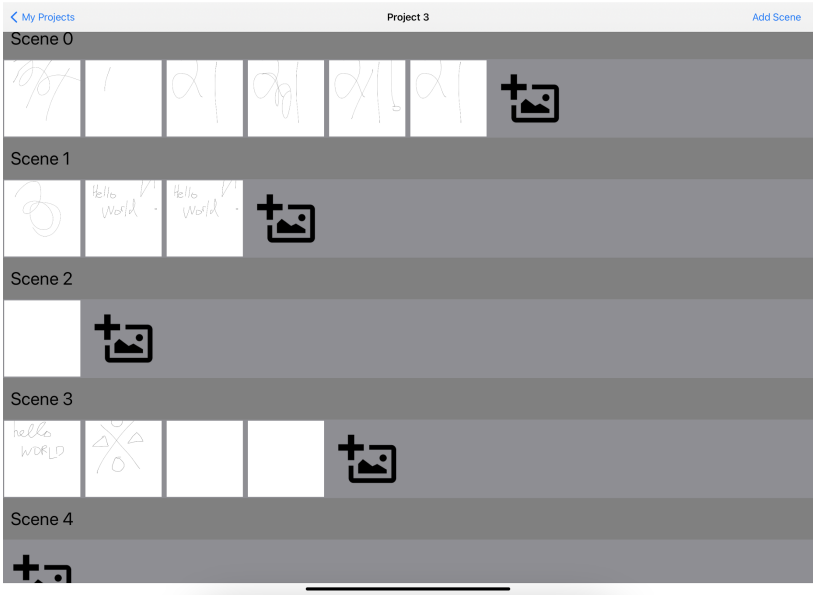
Environmental Assumptions:

- 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

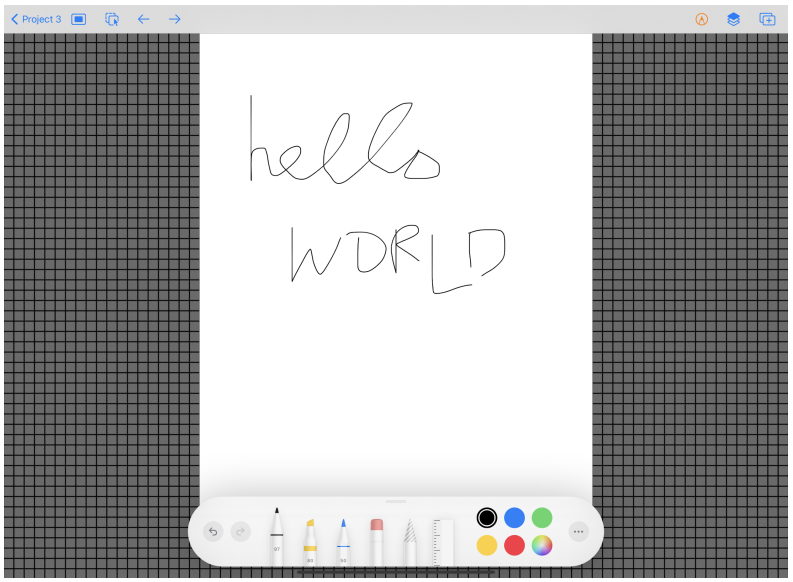


3. Once the user enters a project, all the scenes and shots in the project will be shown, and the user can select a shot to edit



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
d. Long press to reorder shots within a single scene

4. Start drawing

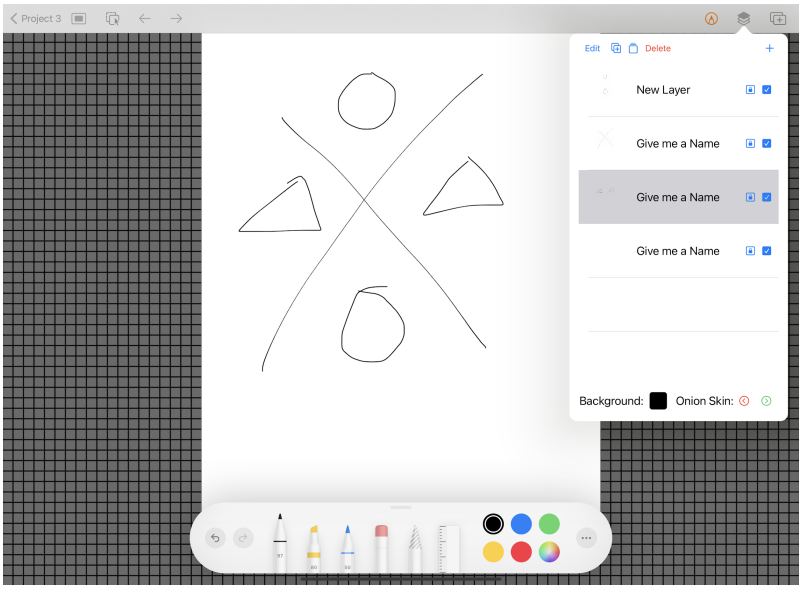


a. Click the rightmost button to create and add a copy of the shot into the same scene
b. Toggle the settings on the toolkit.

i. Drawing tools: Pen, Marker, Pencil
ii. Ruler
iii. Eraser
iv. Colors
v. Lasso Tool

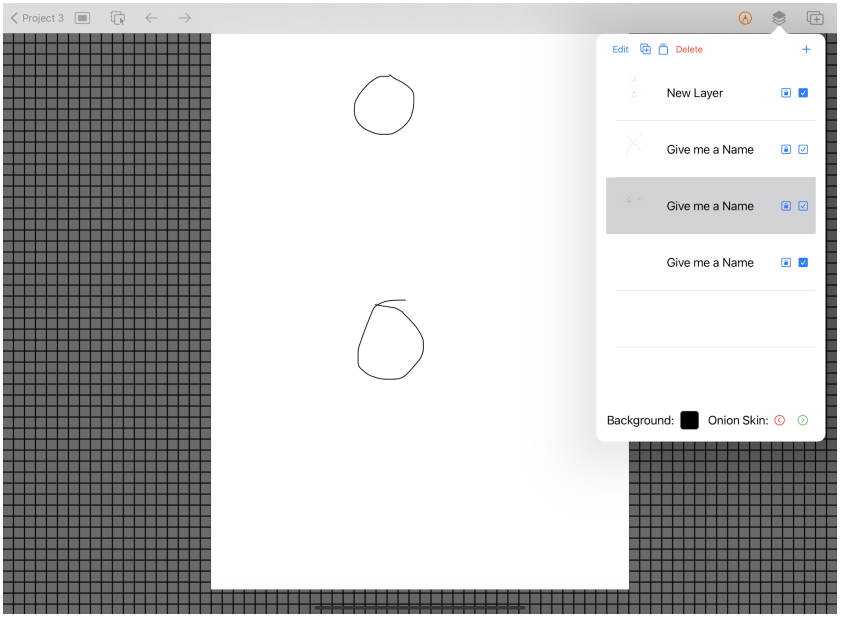
c. Your work will be continuously saved as you draw or make other changes
d. Swipe from the left of the screen to return to the shots screen

5. Working with Layers



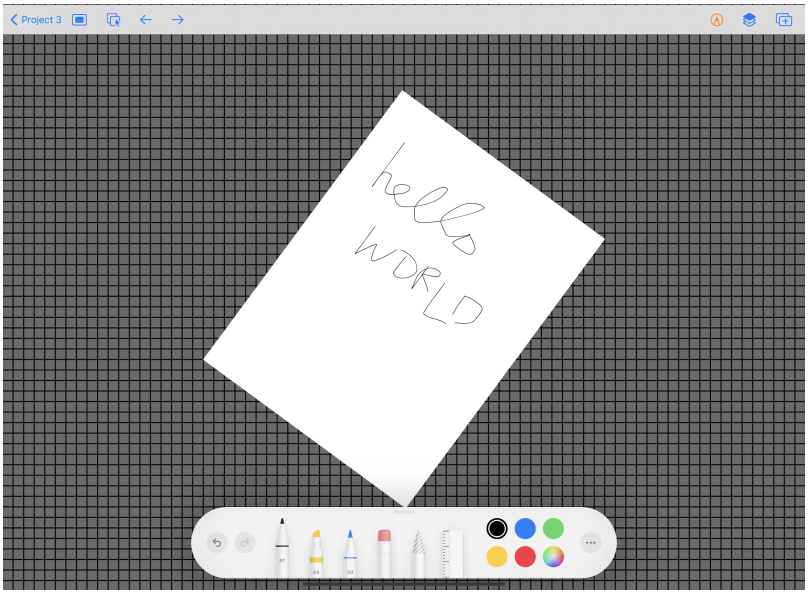
Each shot has several layers stacked on top of each other.

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

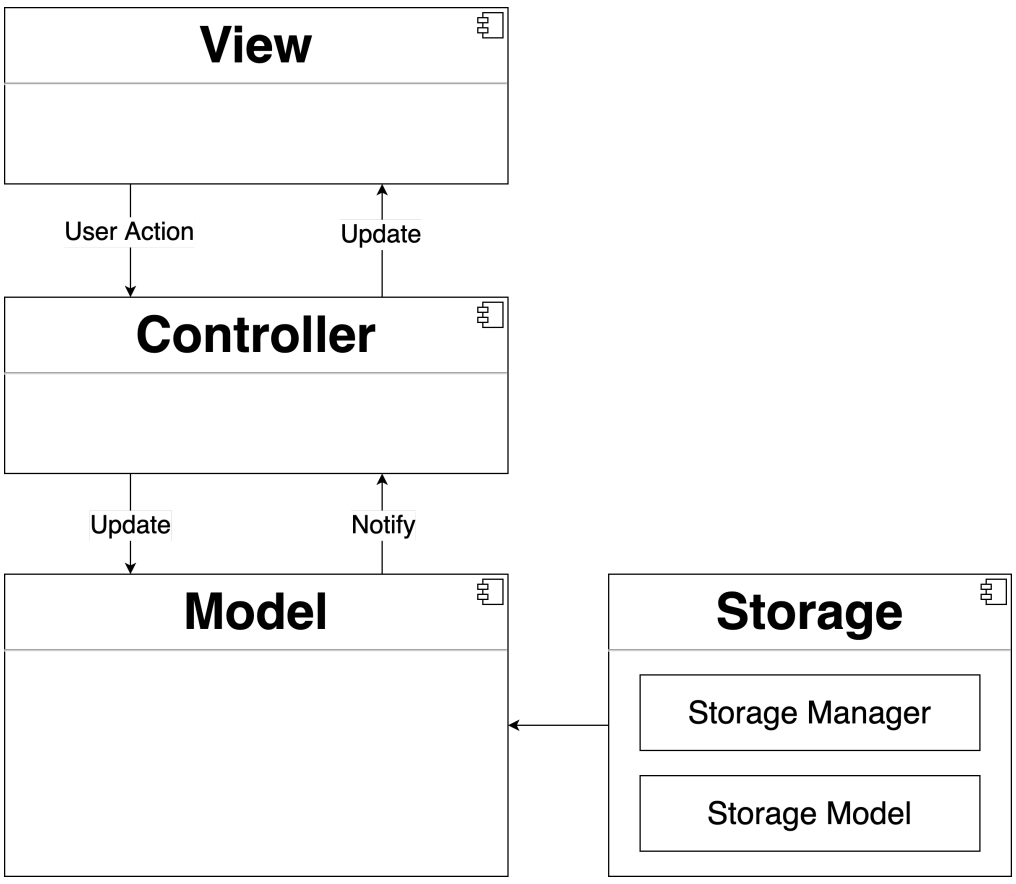
c. Rotating and resizing be performed simultaneously
d. Canvas can be translated using two fingers
e. Click the second button from the left which allows you to rotate/resize/translate the current layer's drawing instead of the canvas
f. Click the leftmost button to reset any rotation made to canvas or layer drawing.

7. Navigation
    a. You can navigate to the next or previous shot within a scene using the left and right arrows on the navigation bar

Design and Architecture

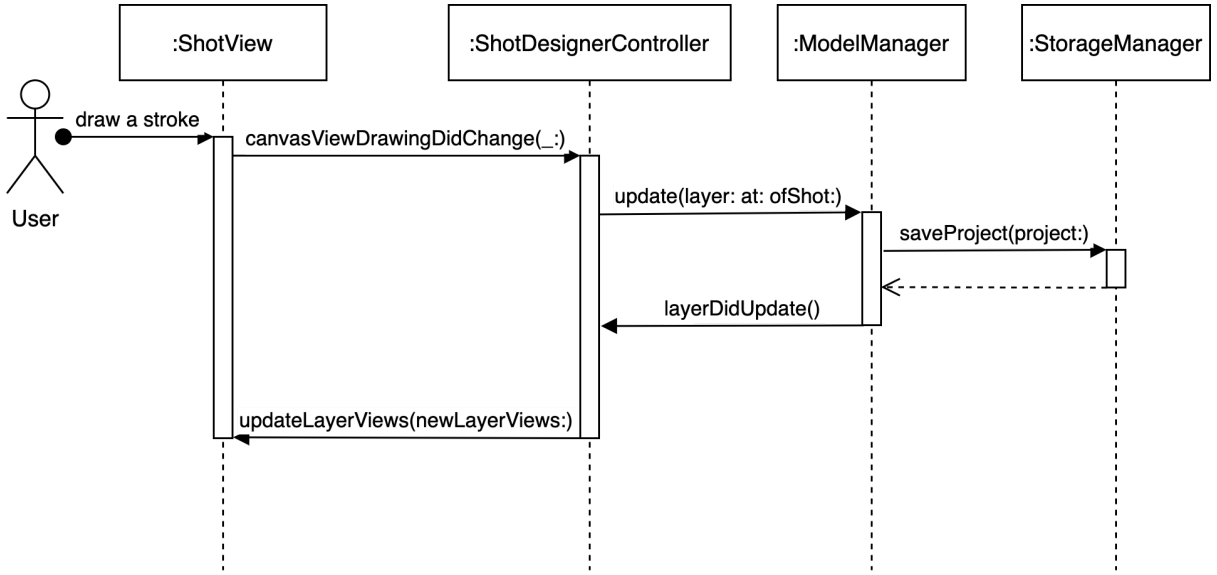
Top-Level Organization

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

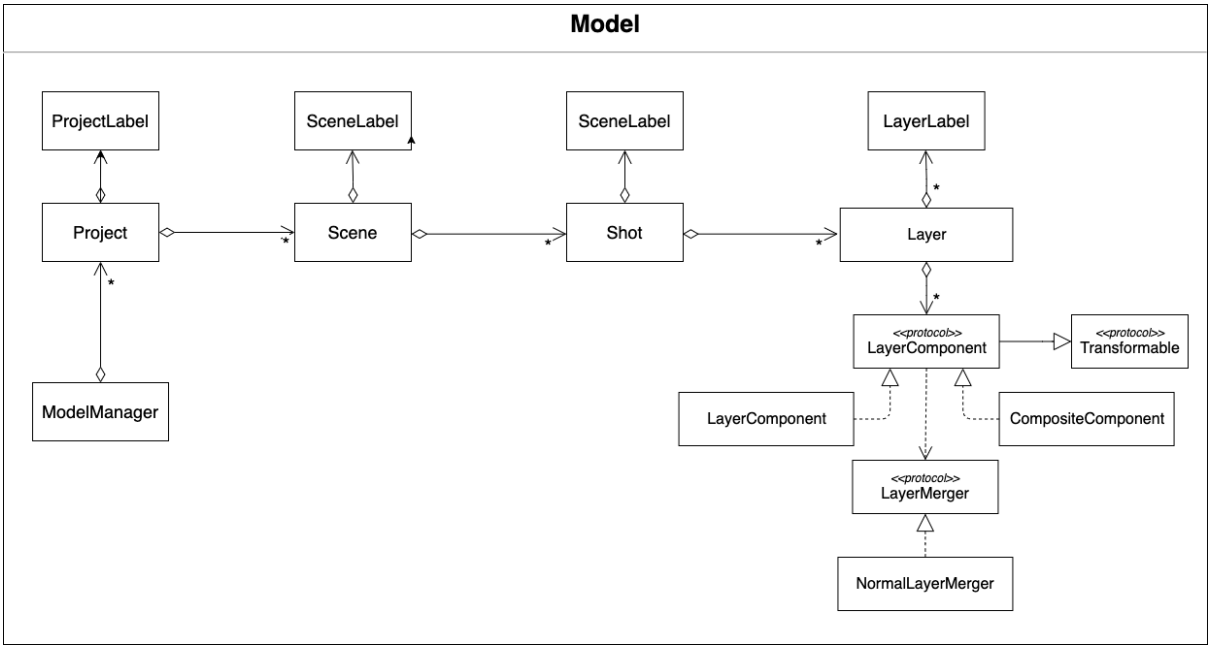
The *Sequence Diagram* below shows how the components interact with each other when the user draws a stroke:



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.
    - Include a ModelManger to expose methods for external use
- 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

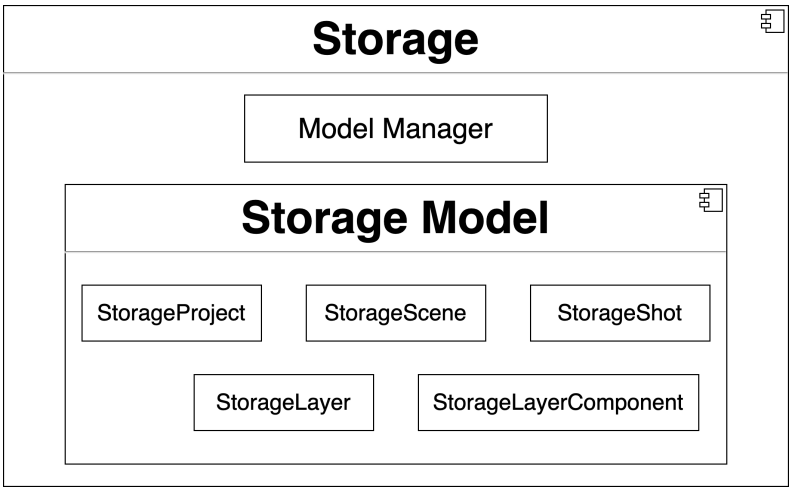
Model



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.

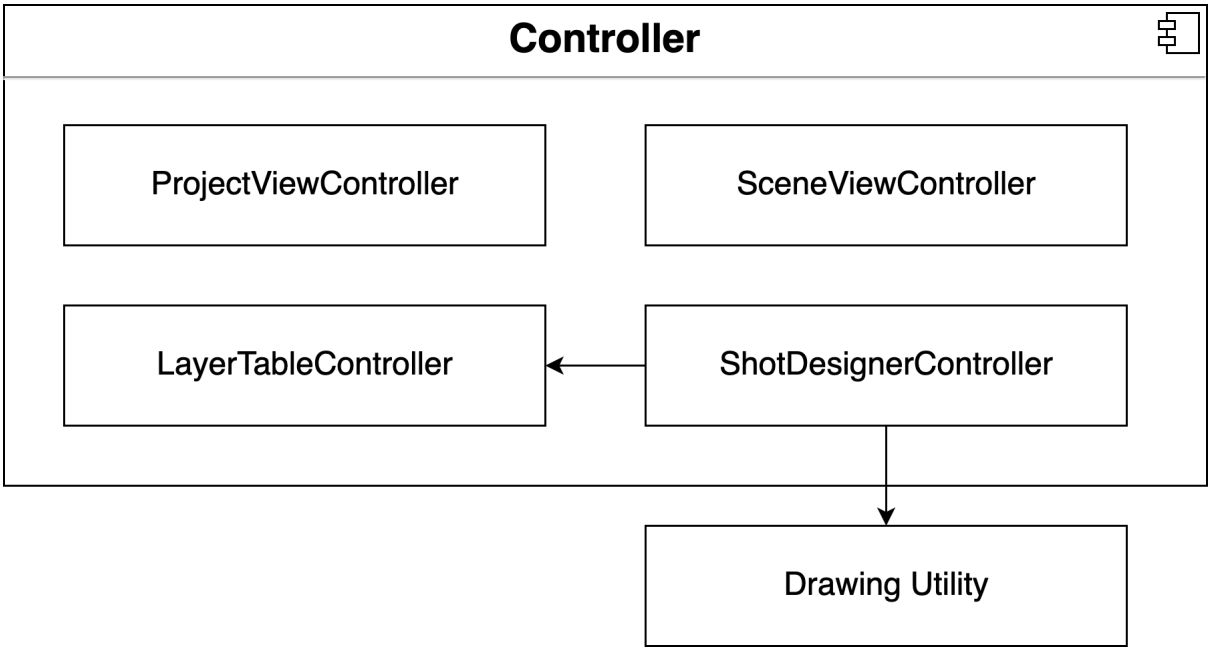
The main construction of the model flows the hierarchy **ModelManager > Project > Scene > Shot > Layer**. The ModelManager serves as an API layer for the model. The Layer encapsulates each stroke or drawing by the user.

Storage



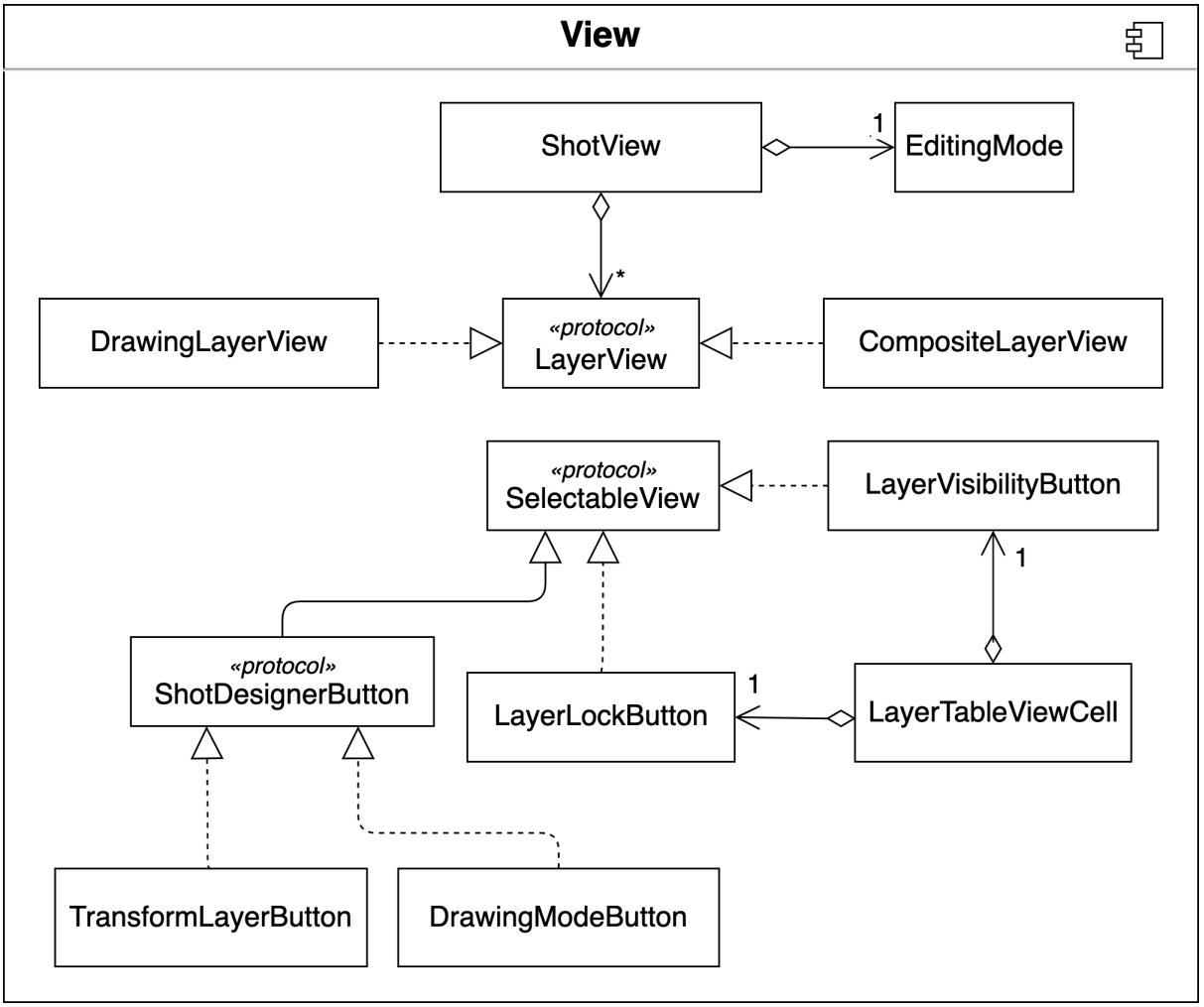
Storage has a Storage Model corresponding to the real Model. In this way the classes in the Model are not forced to implement the Codable protocol. Additionally, we have a `StorageManager` to expose methods for external usage. We are currently using Codable for our storage (i.e. classes in the Storage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

ViewController



View classes are grouped together with their corresponding ViewController classes and other closely related View classes. They are broadly categorized into the Project View, Scene View, and ShotDesigner corresponding to each screen that is displayed to the user. The Project View and View Controller are responsible for the project folders interface that the user interacts with. The Scene View and View Controller gives users access to the scenes and shots of each project, and responsible for actions such as adding and organizing shots. The ShotDesigner is the primary View controller that deals with actions on the canvas.

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 Consideration 1 - *how to represent different types of layers*

Although three solutions are proposed in report 1, none of them really work when we take the layer merging into consideration. We finally came up with an elegant solution to layer modeling by using the composite pattern together with the visitor pattern. See “Runtime Structure” for more details.

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

Originally, we wanted to transform the layer around the anchor point of the layer, which should be at the center of the *contents* of the layer. However, due to the way UIView's transform works and how drawing in PKCanvasView is transformed, we decided to transform the LayerView using its default anchor point (center) instead of the center of contents. Additionally, when transforming a drawing layer, PKCanvasView is transformed instead of the drawing inside the PKCanvasView. See "Runtime Structure" for more details.

Design Consideration 3 - *how to store the layer*

We are using Codable for our storage. However, since we are using the composite pattern for the layer and `LayerComponent` is a protocol instead of a concrete class. How do we "store" this protocol and make the `Layer` Codable? To solve this, we've separated the storage logic with the Model and created a StorageModel. See "Module Structure" for more details.

Changes to be made

1. The process of generating thumbnails is too slow when the project is large, we may need to use the Dispatch queue to help the process. Additionally, the thumbnail currently will not correspondingly transform according to Layer's transform, and we will address this after the new thumbnail generation process is implemented

Runtime Structure

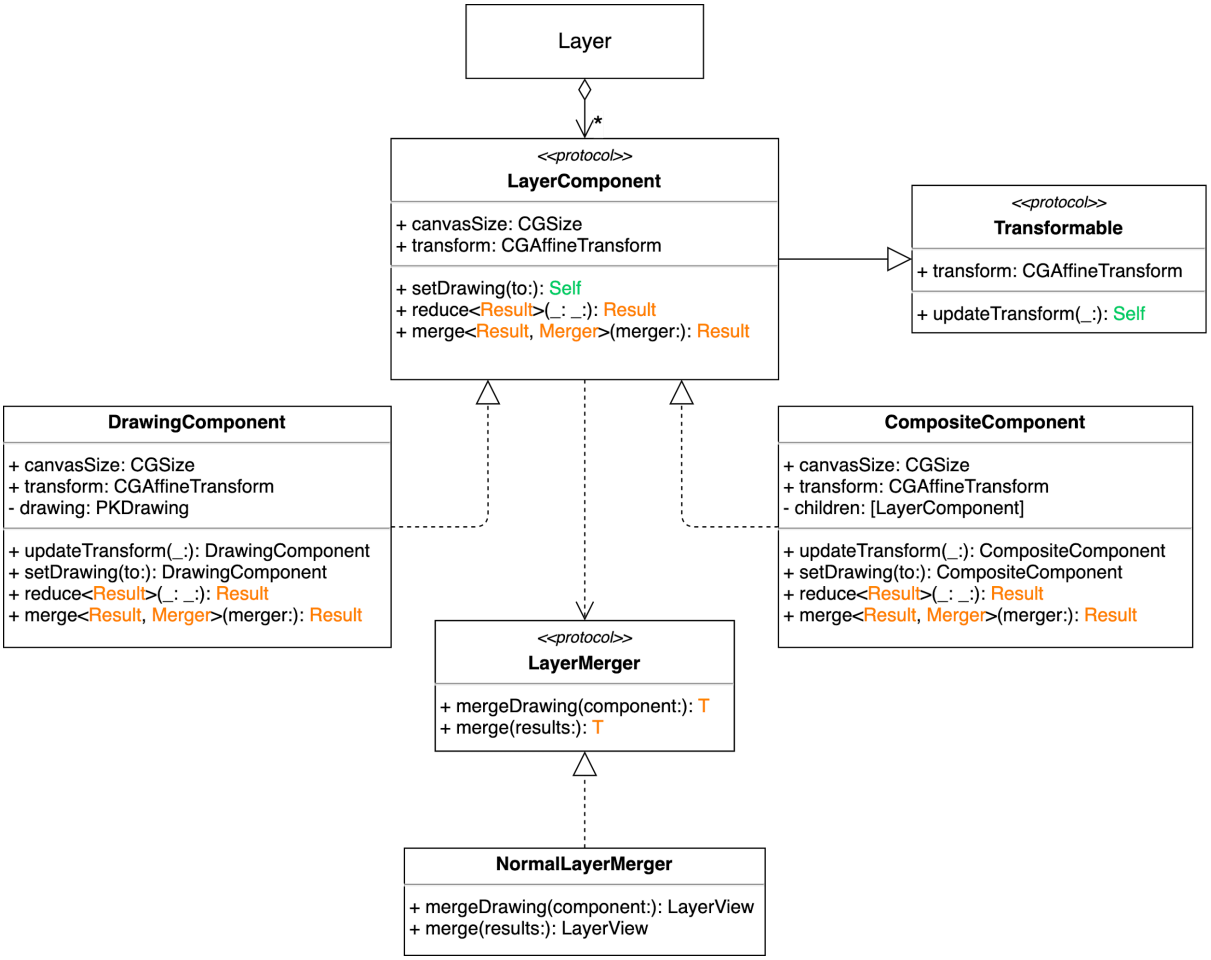
The underlying structure is rather straightforward: we have an array of project ID, containing an array of scene ID, containing an array of shot ID, containing an array of layer ID. The reason why we use Array instead of Set is:

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).

Objects themselves are stored in a dictionary that maps ID to object.

In order to uniquely identify any shot/scene/project, we used to label objects (LayerLabel, ShotLabel, SceneLabel, ProjectLabel). For example, LayerLabel stores the Project ID, Scene ID, Shot ID, and Layer ID. This allows searching for the layer when updating the model without needing a copy of the layer.

Layer Structure



In report 1, we proposed three solutions to represent `Layer`. As mentioned in report 1, “protocol/inheritance” and “enumeration with composition” will not work when taking the layer merging into consideration. Moreover, the “Decorator pattern” is also not suitable as layer components (e.g., image and text) should not really be considered as “decorations” of the drawing component. After trying out different approaches, we finally decided to use the **composite pattern** together with the **visitor pattern**:

1. The initial approach tried in sprint 2 is to make the `CompositeComponent` store an array of leaf components (like `DrawingComponent` and `ImageComponent`) that do the actual work, so that instead of a complicated tree, the `CompositeComponent` is only a “flat” tree with depth 1. However, this approach was soon discarded as it does not correctly represent the relationship between components after merging. Specifically, this approach cannot easily cope with situations like transforming a composite layer C merged using another two composite Layers A and B, which have been individually transformed. Since components in A and B may be individually transformed around their anchor point, after which A and B may also be transformed around the anchor point of A and B, transforming C will become extremely complicated.

2. The second approach starts to use a tree hierarchy for layer components: `Layer` to store a `CompositeComponent` which contains an enumeration of type `NodeType` with associated value: For example, `NodeType.composite([CompositeComponent]), NodeType.drawing(DrawingComponent)` (similar enumeration can be found in the `StorageLayerComponent`). In this way, we know which type is this composite component and we store the components in the corresponding enumeration. Although this will make Storage much simpler (we now know the type) and for external usage, not much switch statement is needed, there are still lots of switch statements inside the class, which defeat the purpose of using polymorphism and the tree structure. So we came up with the final solution.

3. We finally settled with the Composite pattern, 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

4. The final problem about the layer model we need to address is how to render Layer into `LayerView`. It is easy to implement in the Layer Model using the Composite pattern, however, we should really separate the UI elements from the Model. Then another problem will arise, for external usage, the client only works with `LayerComponent` through the protocol and has 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.

5. As a side note of Layer Structure, we’ve also tried several ways to store the transform of the Layer Component. Originally, we wanted to transform the layer around the anchor point of the layer, which should be at the center of the *contents* of the layer. It turned out to be much more complex: after a series of transform applied to layer components at different depth, it’s very hard to get orientation and anchor point right (note that UIView’s `frame` is undefined after transform other than scaling) because as the user merges layers or transforms layers, the anchor point as well as the orientation of the UIView will also change and a UIView’s transform is based on it’s local coordinate system and anchor point. So initially, we let `LayerComponent` have an array (sort of like a “transform history”) of `TransformInfo` that keeps track of the position of a transform’s anchor point relative the local coordinate system at that time, as well as the orientation, represented using a CGVector, and finally the corresponding transform(which can be scale/rotate/translate). We later changed the array to an array of `CGAffineTransform` and let the View Controller compute the transform using current orientation and anchor point. However, unfortunately, there was always something wrong with the transform as this process of getting the anchor point and orientation, and compute corresponding `CGAffineTransform` is error-prone. We eventually choose to rotate around the UIView’s default anchor point (center) instead of contents’ center. Another thing that gets in the way is PKDrawing. Firstly, unlike normal UIView, PKDRawing is not a view, so its transform is always relative to the origin (0, 0) at the top left corner (even after a series of transforms). Secondly, whenever we call its transform(using:) method, the whole PKCanvasView will

get refreshed. Consequently, the canvas is “flickering” instead of a butter-smooth 120hz drawing experience. Due to the two points above, we decided to transform the `LayerView` directly using the default anchor point.

Module Structure

For the Model, we used the Facade pattern. The ModelManager acts as an entry point 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.

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 a enumeration class `StorageNodeType` inspired by the failing second approach in the Layer Structure above. After this, the rest of decoding/encoding is just retrieving data from/making nested Coding Containers.

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, there is improvement in the integration process, and there is better synchronisation in the team with regards to each person's individual components. There were also many engineering challenges from implementing the layers components and the refactoring of the storage and model of the application. On the other hand, there is also much room for improvement on the communication front, and a lot of time and trial-and-error attempts could have been avoided with better discussions and coordination with the team up front before diving into each individual's codebase.

Lessons

1. A lot of time and effort could be saved with better communication and collaboration throughout the development process, not just during the integration of code.
2. When it comes to code collaboration, "keep it simple, stupid" is a good atittude to follow, as many times the codebase will be read, re-read, rewrtiten and refactored by others, and the worst thing that could happen is misunderstanding of codebase or gaps in information amongst the team.

Known Bugs and Limitations

- Reordering of shots (by long-pressing) can be laggy and run into some bugs when the shots are dragged outside the permissible area.
- The boundaries of the canvas after transformation is still shown
- The folder system (project view and scene view) faces some performance issues when the number of shots increases.

Detailed Schedule + Task List

Responsibility is divided into sections of the codebase.

Model:

- Layer and its subcomponents: Tian Fang
- Rest of Model: Marcus

View + ViewController:

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

Storage: Tian Fang

We also intend to implement the following features:

- Undo and Redo
- Cloud Storage
- Search feature

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.