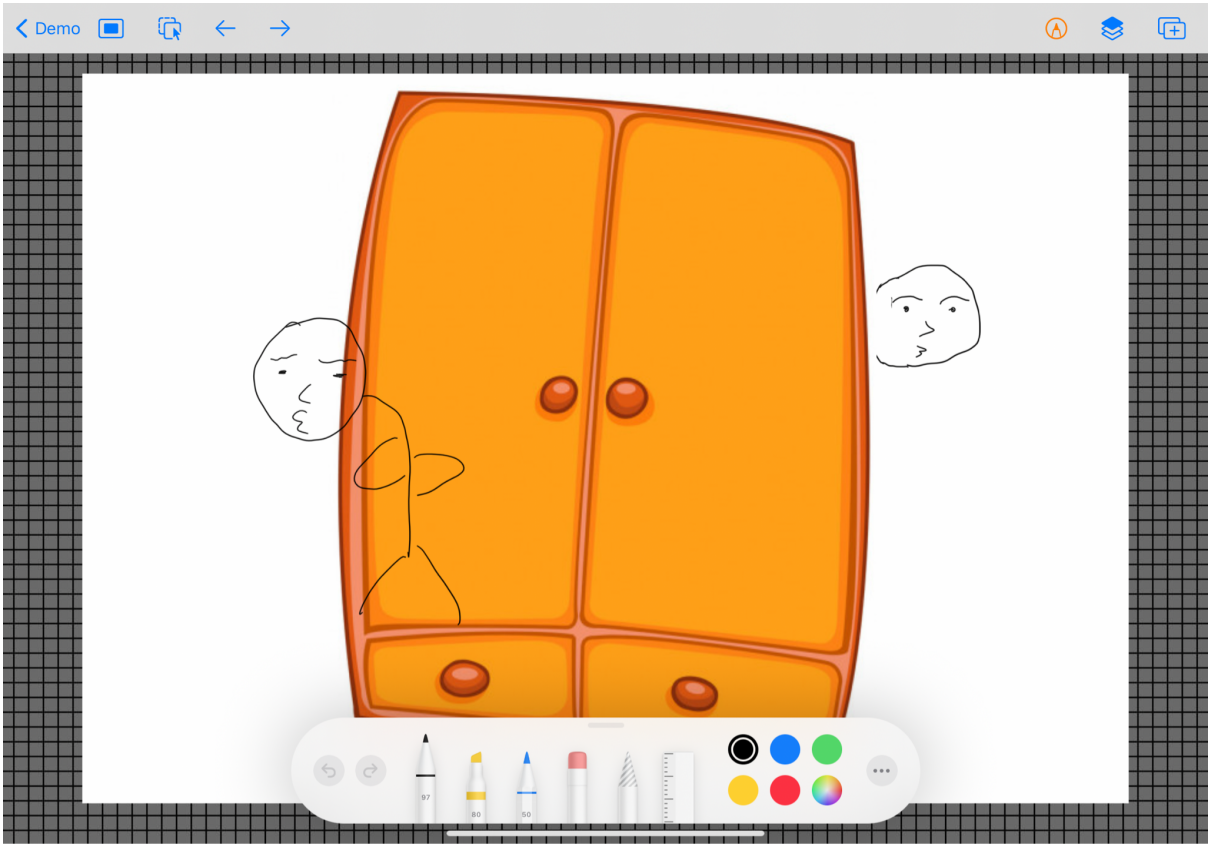
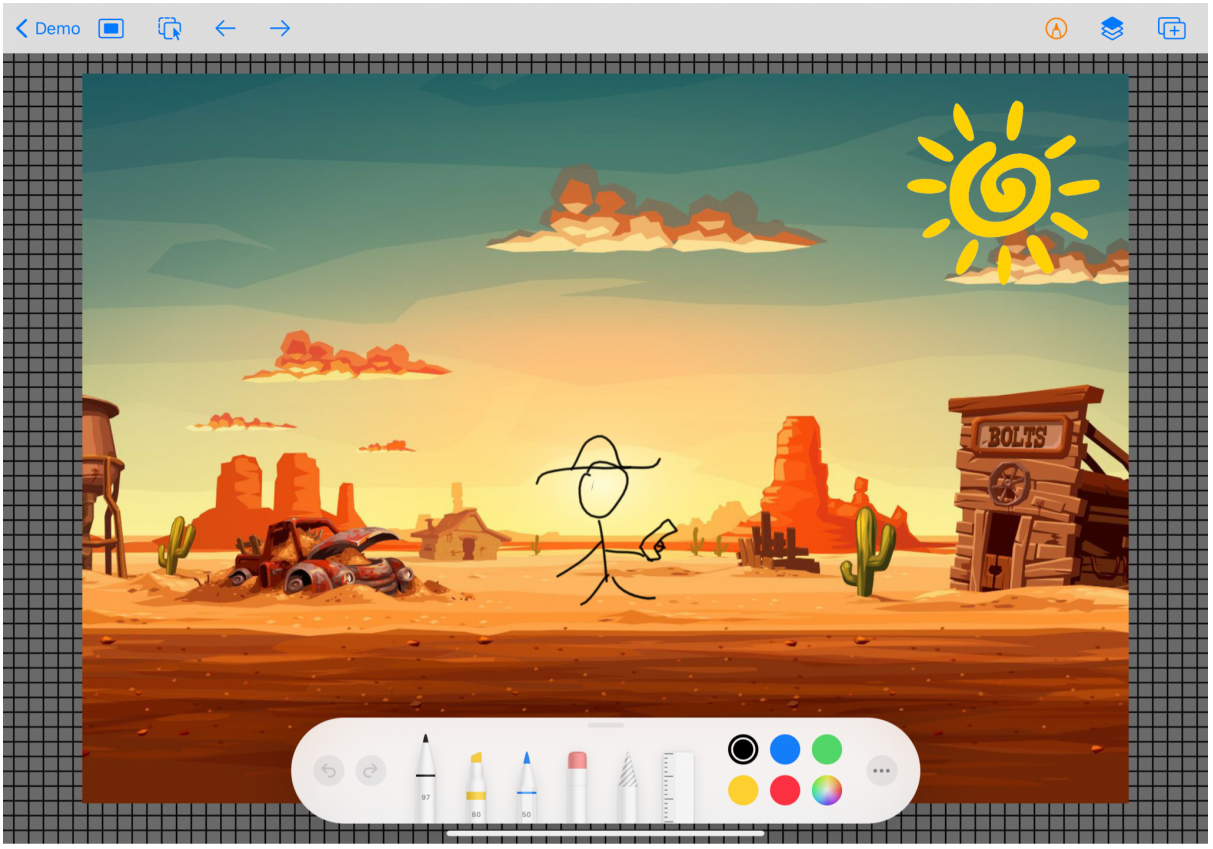
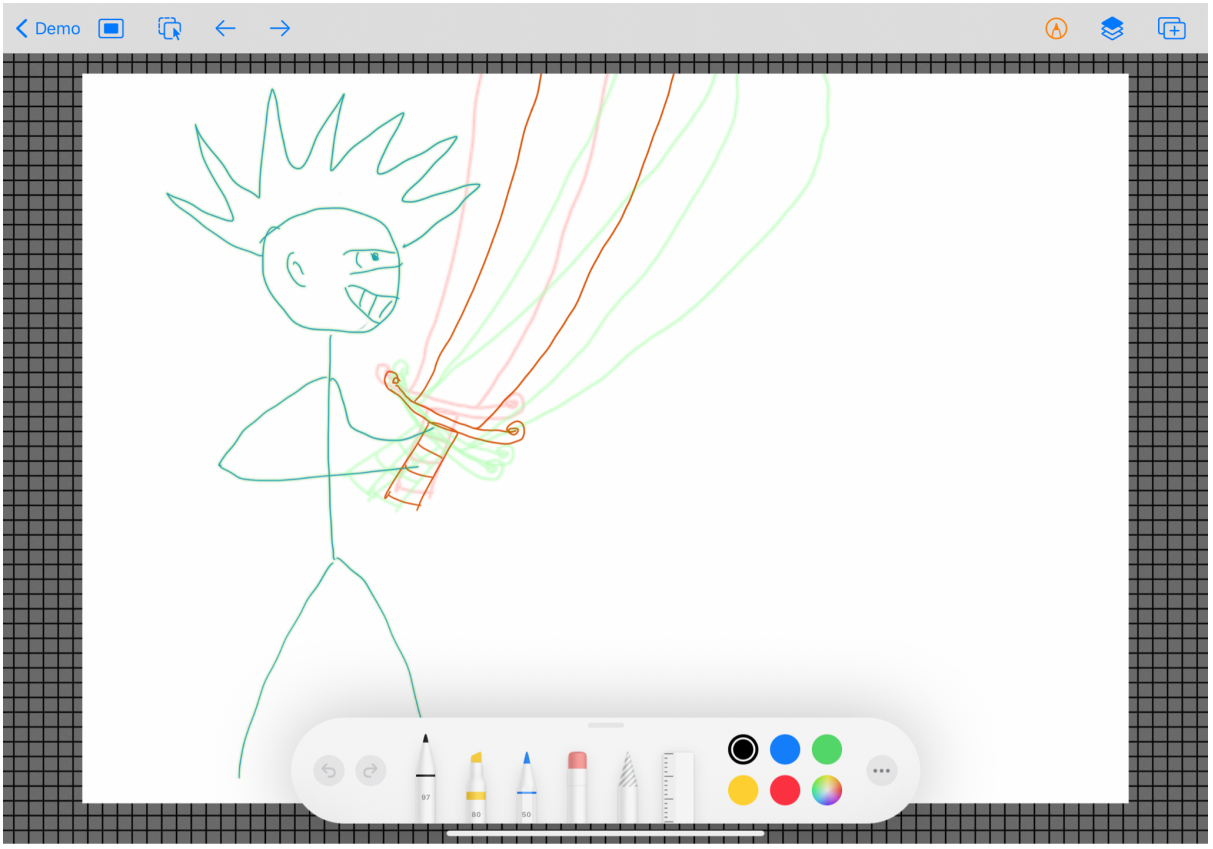
CS3217 Project Extension Report

Storyteller

Marcus | Pan Yongjing | Tian Fang


Demo


Demo
BOLTS



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

Folders/Directories Navigation

- Create Folders
- Delete Folders

- Rearrange Folders
- Rename Folders

Project Navigation

- Add Project
- Delete Project
- Rename Project

Scene/Shot Navigation

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

Canvas

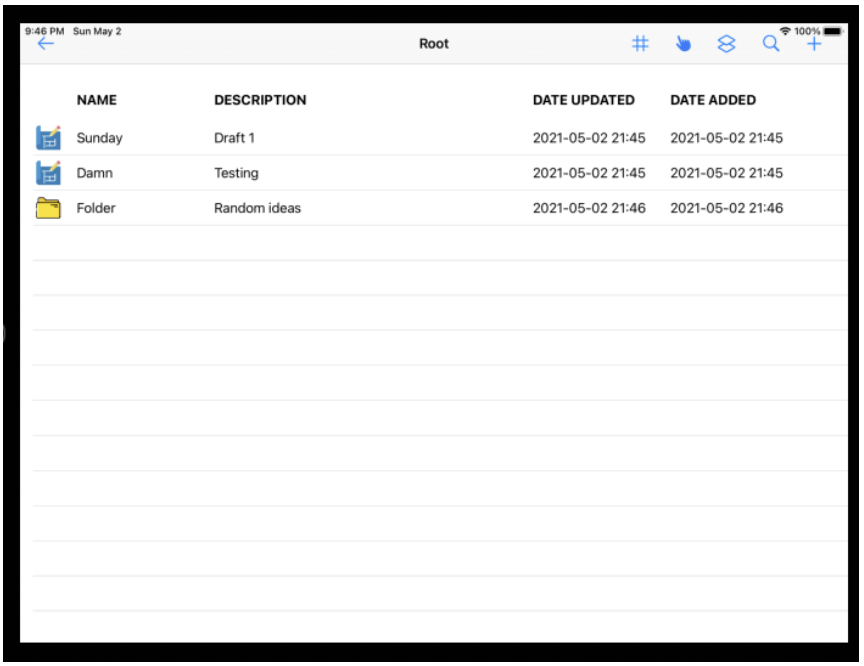
- Draw on the canvas
- Use various tools on the canvas
    - Different colored pens
    - Lasso Tools
    - Ruler
- Work with layers on the canvas
    - Add layers
    - Edit layers
    - Remove layers
- Duplicate shot
- Onion Skin
- Directories system

User Manual

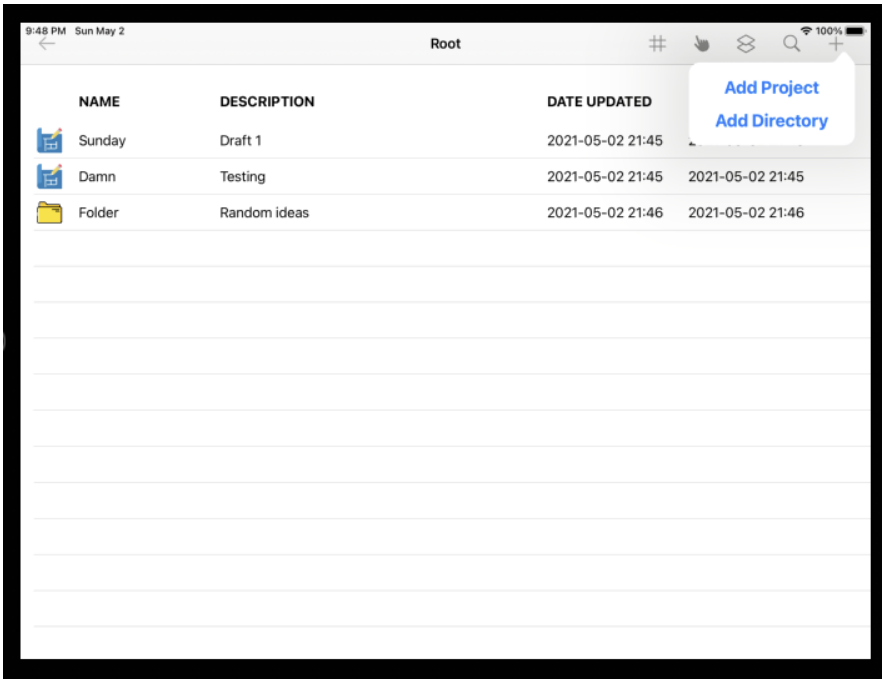
Environmental Assumptions:

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

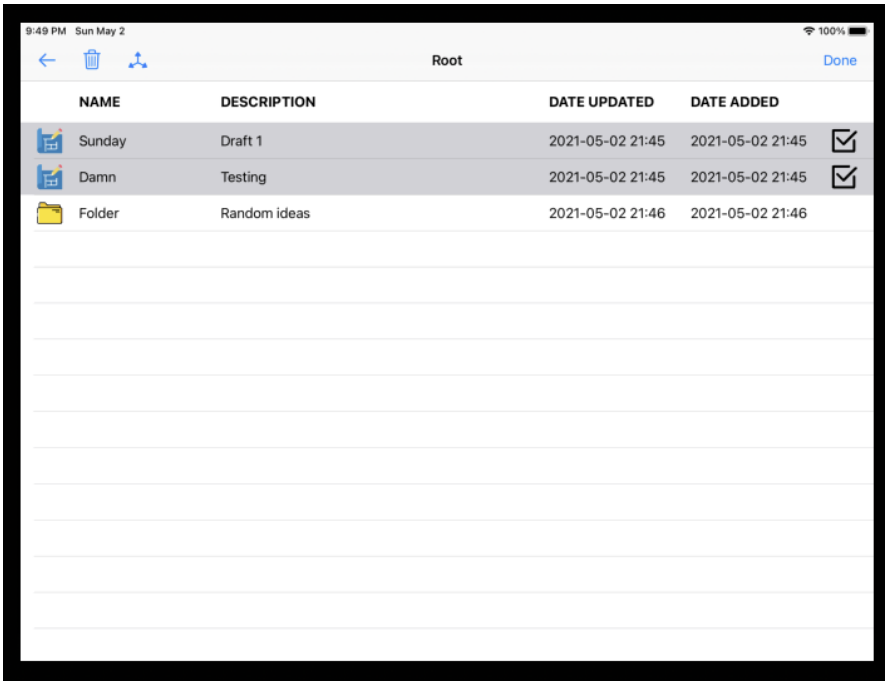
1. Start the app
2. Select a folder or a project. A folder may contain subfolders and projects.
3. Select an existing project or create a new project or folder by clicking the “+” button



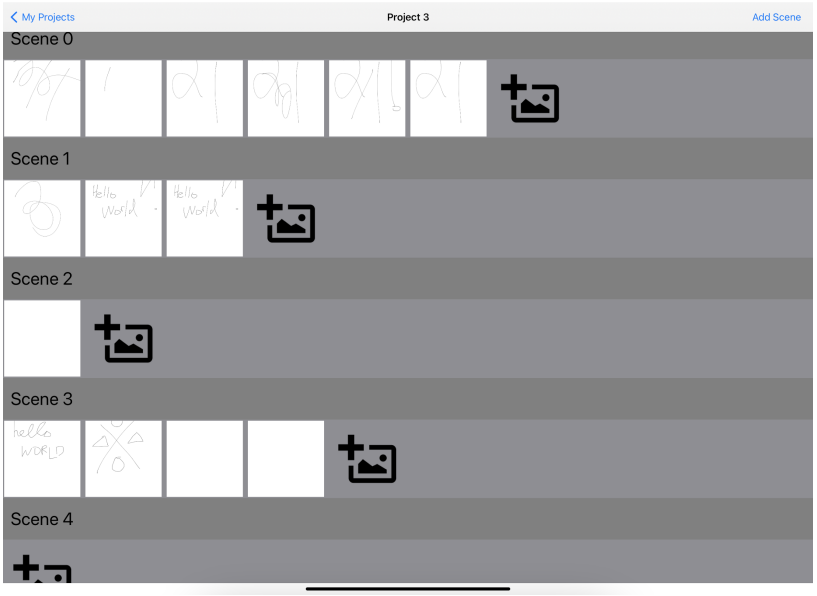
Click the top-right add button to add either a folder or a project.



A folder can contain other folders or projects. There are options to select, delete and move an existing folder to different directories.

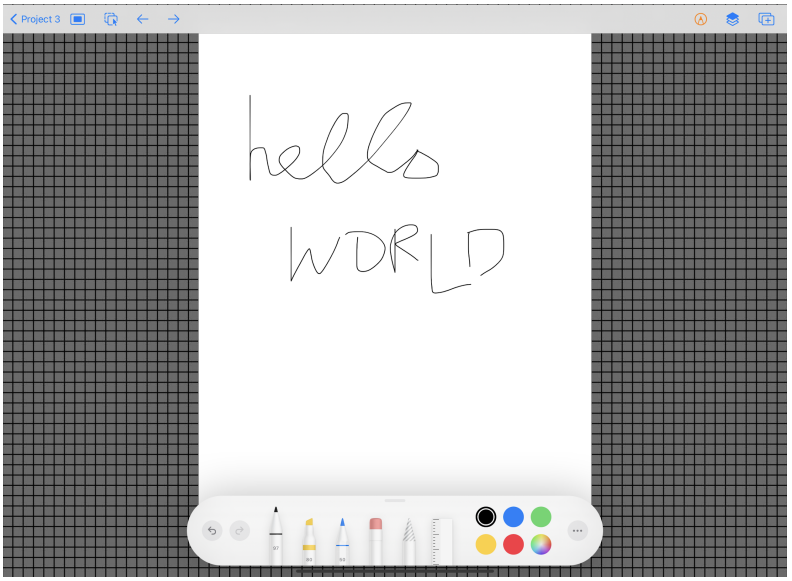


4. Once the user enters a project, all the scenes and shots in the project will be shown, and the user can select a shot to edit



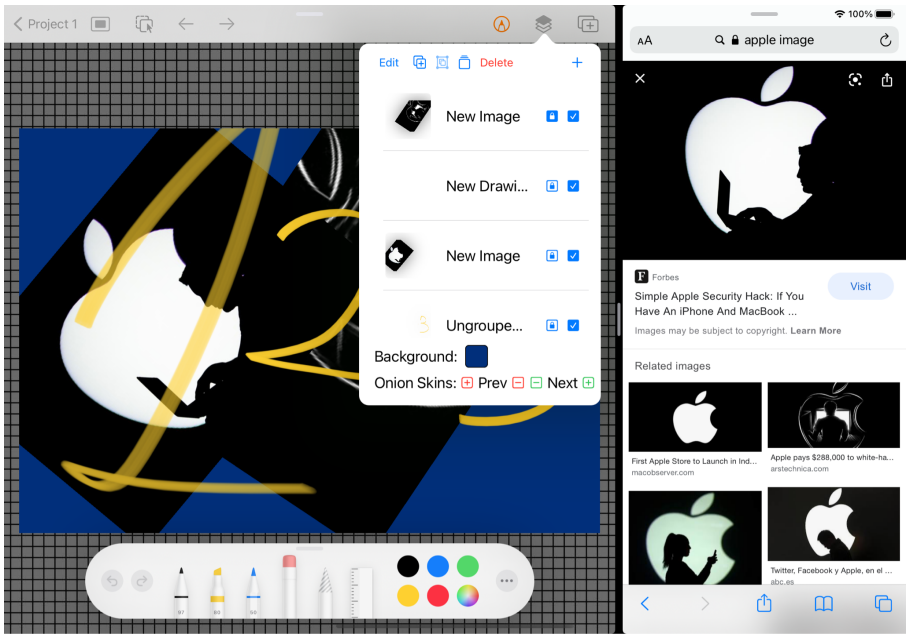
a. Click the top right “+” button to add a new scene
b. Click the “+” button behind the shots in a scene to add a new blank shot to the scene
c. Click the back button to return to the projects screen
d. Long press to reorder shots within a single scene
e. Pressing on the DELETE button at the top right of each scene header deletes the scene.

5. Start drawing



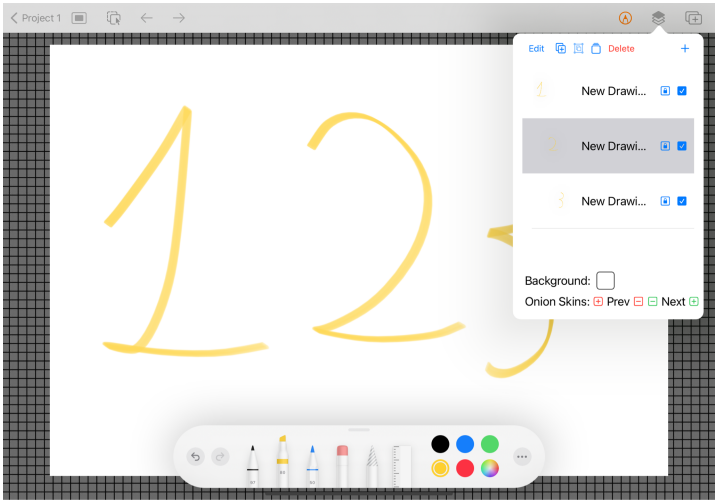
a. Click the rightmost button to create and add a copy of the shot into the same scene
b. Toggle the settings on the toolkit.
    i. Drawing tools: Pen, Marker, Pencil
    ii. Ruler
    iii. Eraser
    iv. Colors
    v. Lasso Tool
c. Your work will be continuously saved as you draw or make other changes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

6. Working with Layers



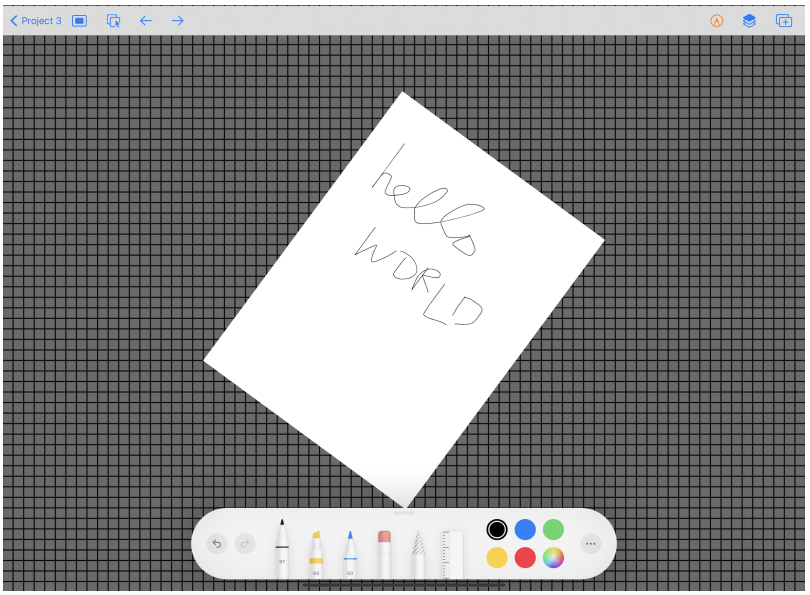
Each shot has several layers stacked on top of each other.

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

7. Resizing, rotating, and translating canvas/layer

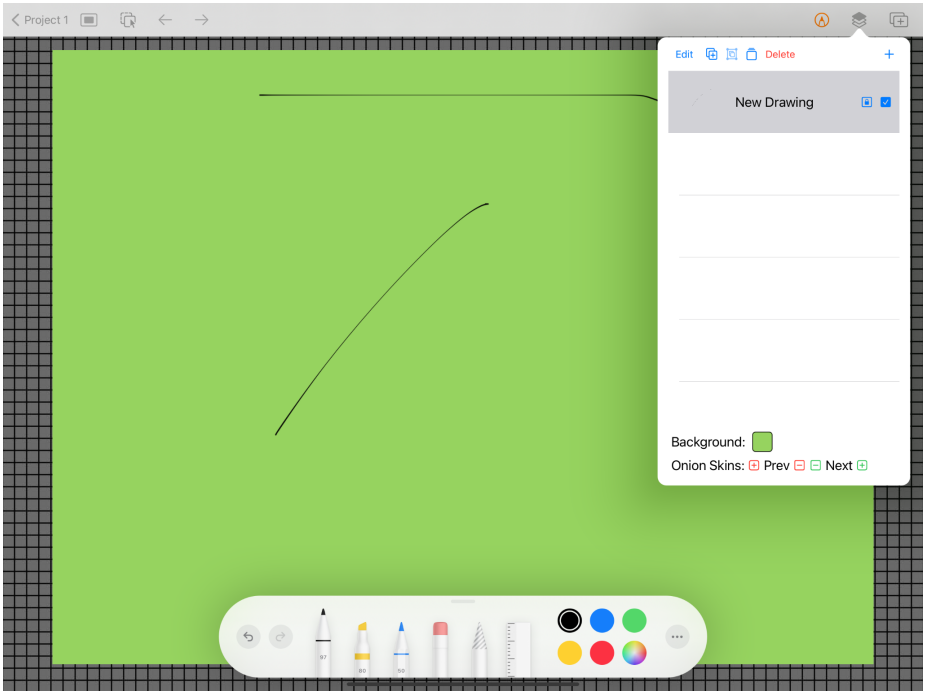


a. To resize the canvas, simply pinch on your canvas.
b. To rotate the canvas, you can use two fingers dragged in opposite directions
c. Rotating and resizing be performed simultaneously
d. Canvas can be translated using two fingers

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.

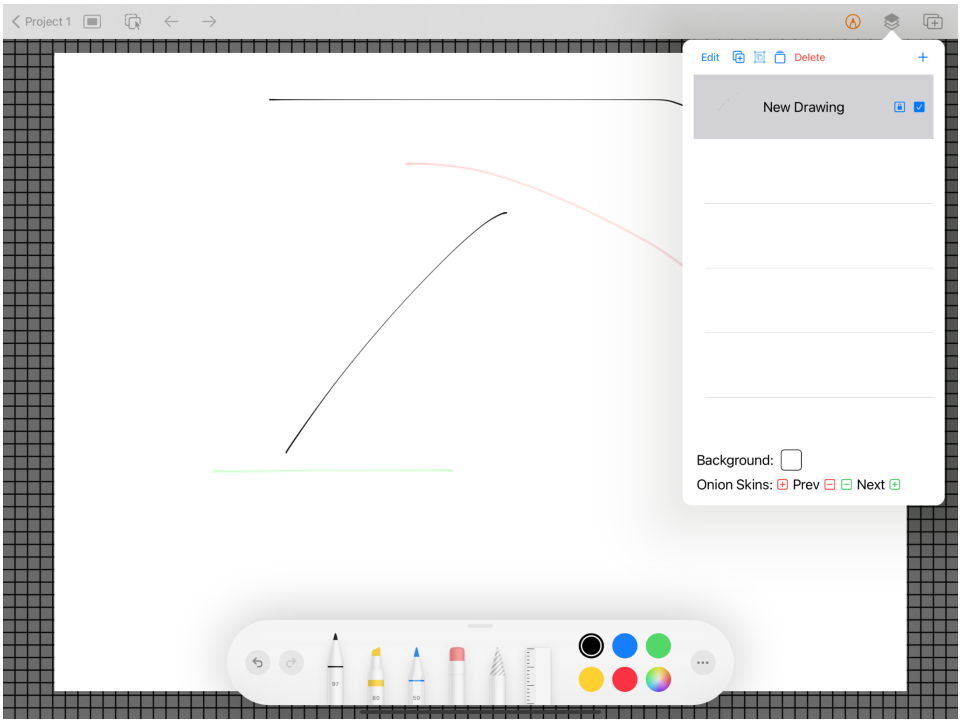
8. Navigation
    a. You can navigate from the current shot to the next or previous shot within a scene using the left and right arrows on the navigation bar

9. 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

10. Onion Skin



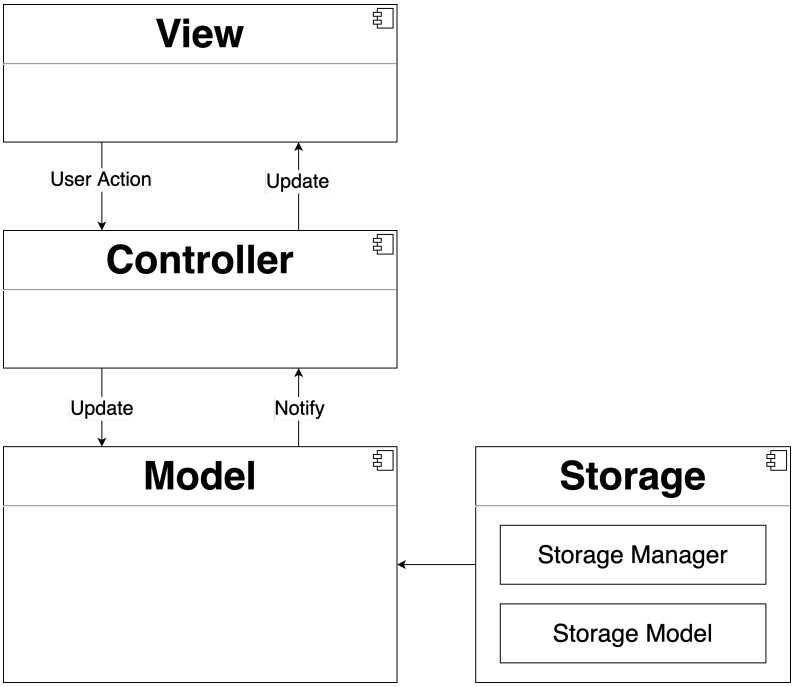
    a. Click the second button from the right at the navigation bar
    b. Click the previous (red) / next (green) PLUS onion skin button to display a semi-transparent outline of the previous / next shots (Number of click = one additional shot before/after if any)
    c. Similarly, MINUS button to hide one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

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

Design and Architecture

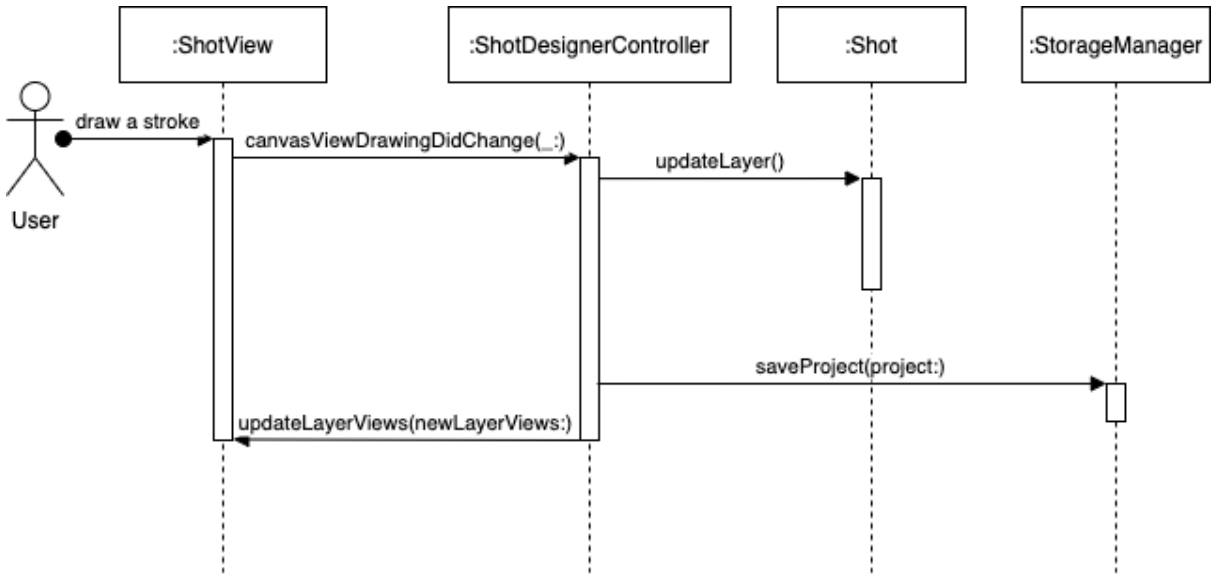
Top-Level Organization

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

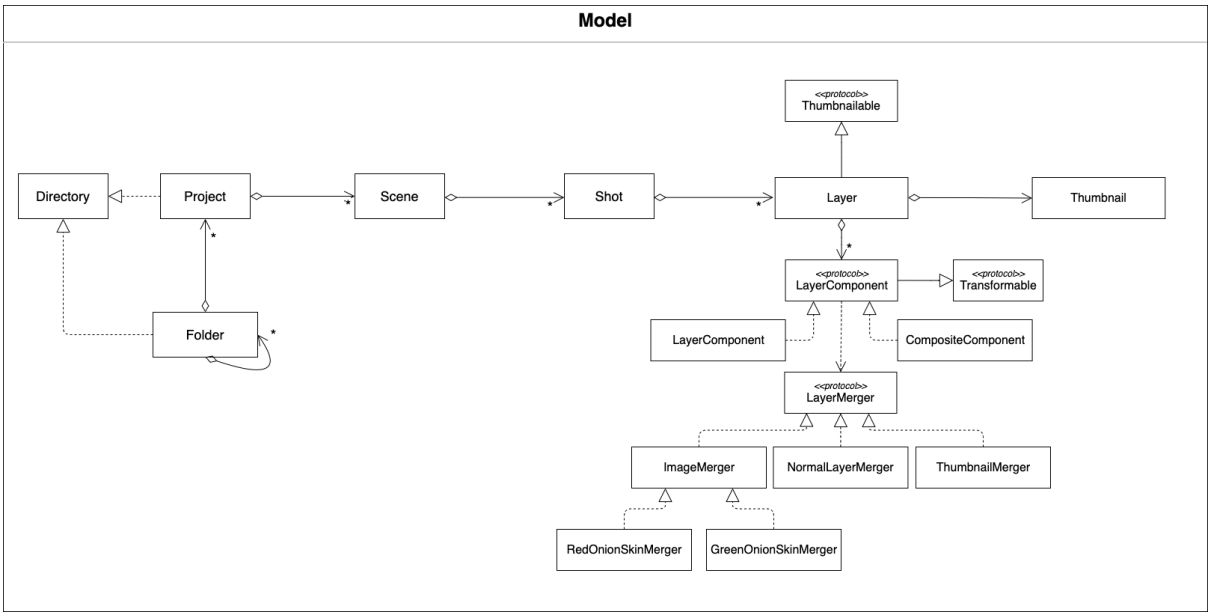
The *Sequence Diagram* below shows how the components interact with each other when the user draws a stroke:



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

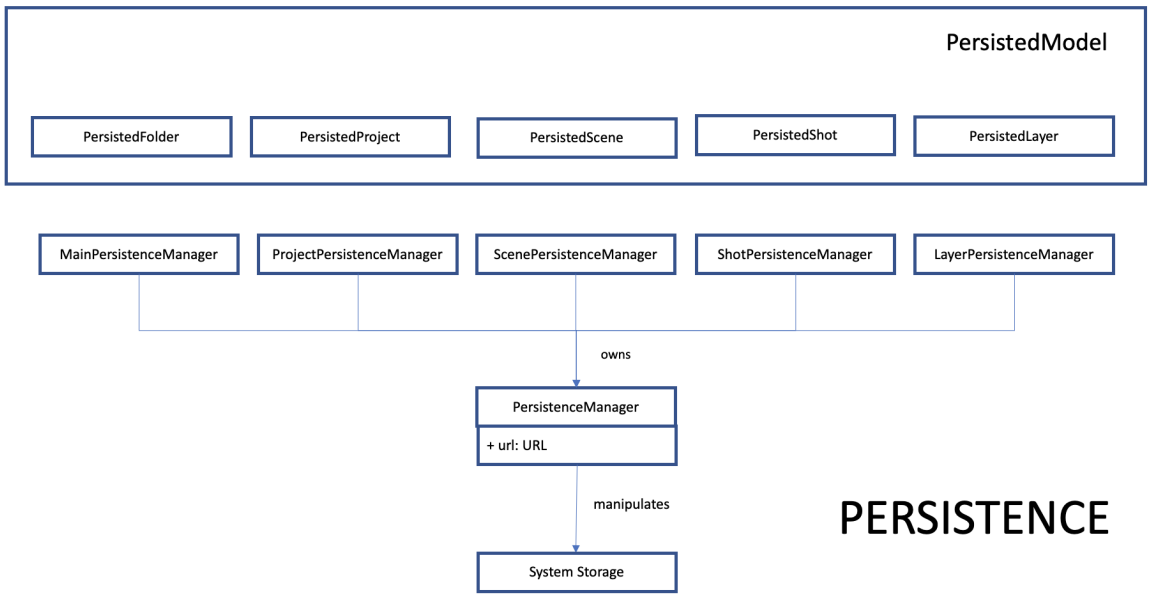


As discussed during our previous sprint review, our model now uses classes (reference-type) instead of structs (value-type). This way, updating the model does not always have to go through a recursive hierarchy from the top-level (Project) to the edited component, as a reference to the relevant object 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 **Folder > Project > Scene > Shot > Layer**. A Folder keeps an array of directories, which can be another Folder or a Project

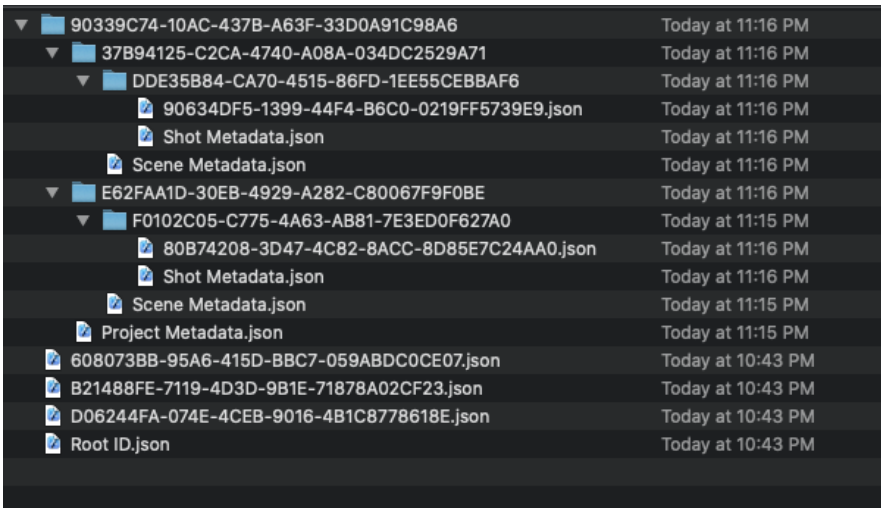
Persistence



For persistence, we no longer store the entire project each time a single layer is updated. Instead, we are structuring our device storage system into folders, where at the topmost level, we have a Project folder that contains JSON-encoded Project

Metadata (its properties as well as ordering of its scenes) and Scene folders. Each Scene folder contains JSON-encoded Scene Metadata (its properties as well as ordering of its shots) and Shot folders. Each Shot folder contains JSON-encoded Shot Metadata (its properties as well as ordering of its layers) and JSON-encoded Layers.

There are five model-specific Persistence Managers (<Model>PersistenceManager). The MainPersistenceManager manages at the top-most level of the device directory, the ProjectPersistenceManager manages each Project's device directory, and etc.
Each <Model>PersistenceManager behaves as a Facade to a PersistenceManager object (<Model>PersistenceManager owns a PersistenceManager), which in turn is initialized with a fixed URL that indicates the address of the device directory that the relevant <Model>'s data is.

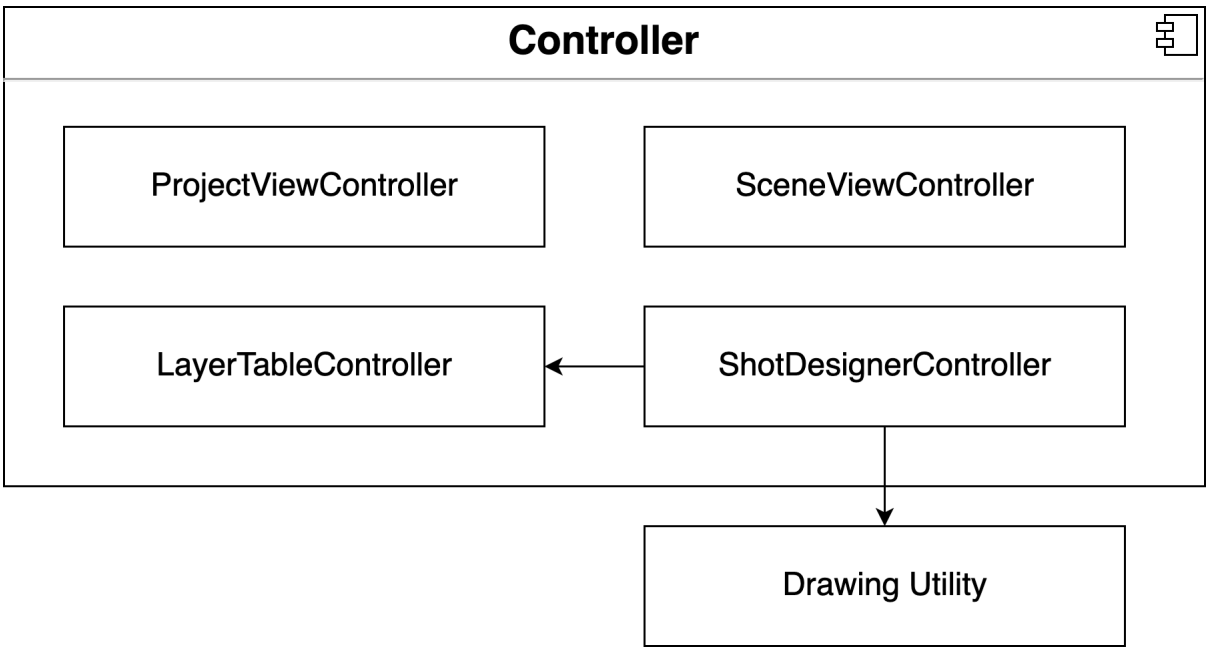


To keep track from the top level's URL address down the bottom level's URL address, each higher level <Model>PersistenceManager acts as the producer of its lower level <Model>PersistenceManager that is directly one level beneath it (e.g. ProjectPersistenceManager can produce ScenePersistenceManager which has a URL that is one path deeper)

For simplicity and uniqueness-guarantee, we are using each Model's ID to name each folder/file. This maintains consistency and avoids names that may be same as

other files (e.g. “Project Metadata”) and also, because each project is uniquely identified by its ID rather than its title (this is important when loading and saving Model, since e.g. a Project Metadata will keep track of ordered ID of Scenes only)

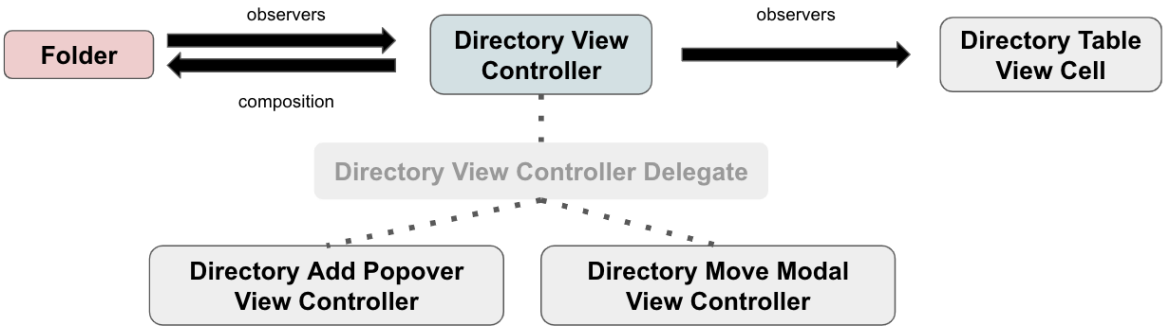
ViewController



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.

Directory View Controller



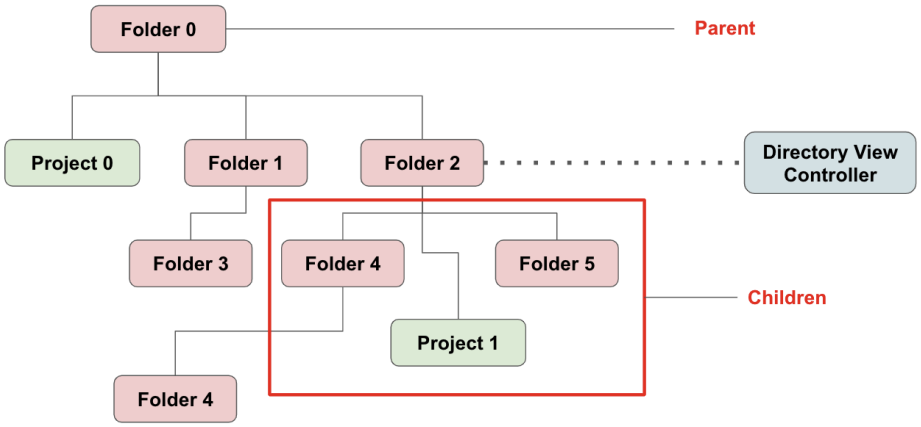
The DirectoryViewController is primarily in charge of displaying and handling functionalities on the directories page of the screen. The DirectoryViewController takes in an instance of the Folder model, which defines the location of the folder in the directories tree that will be relevant for the view controller. The view controller is also kept in sync with change in the Folder model as it implements a Folder Observer - any updates to the model can be observed by the DirectoryViewController and the necessary actions can be taken in the contents being displayed.

As the directory page consists of mainly the main table view. Information is passed to the rows and cells of the table by making the cells observers of the DirectoryViewController.

Modal view controllers and Popup view controller such as DirectoryAddPopoverViewController and DirectoryMoveModalViewController also can perform important actions with regards to the folders or projects via the Delegate of the Directory View Controller.

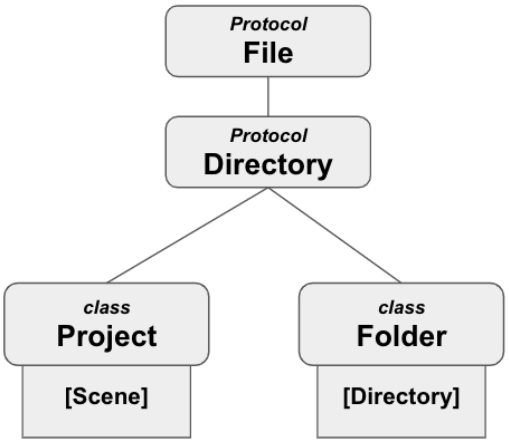
The Directory Tree

As each folder is able to contain multiple folders or projects, and given the need to allow the user to move folders either one level down or one level up.

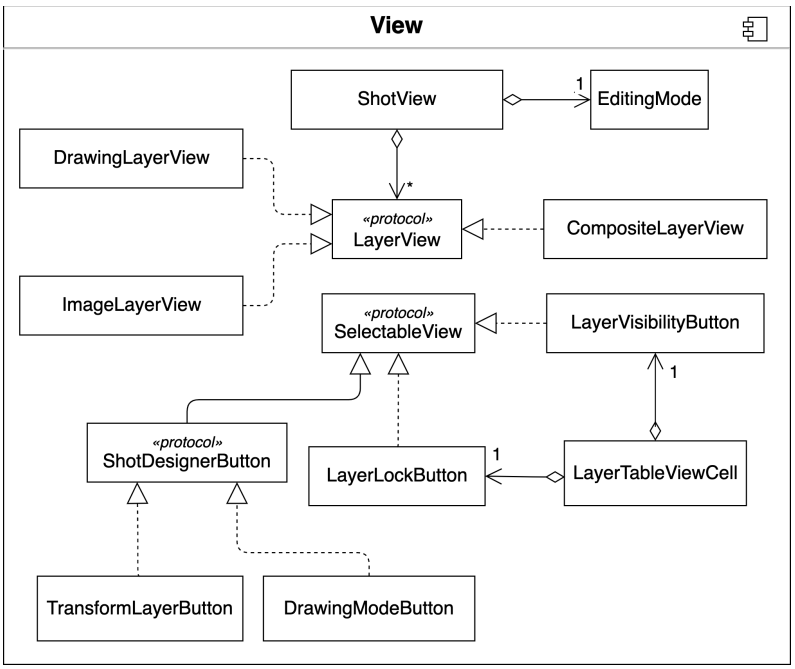


This means that a folder always needs to keep track of its direct children and parent folder. At the view controller, the user has the ability to move any of the children folders into each other or into the part.

Both Project and Folder models are classified as Directory that exist as a single row in the directory page. They have many identical properties i.e. name, description, date added, date updated etc. but their key difference is the fact that folders are also to contain other folders or projects, while a project has to contain a collection of scenes.



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 - *Inheritance vs Composition for PersistenceManager*

In our project, PersistenceManager is a class that is used to directly manipulate the device's directory for the app. Initially, we planned to have FolderPersistenceManager, ScenePersistenceManager, etc. each inherit from the PersistenceManager class. However, this means that each of these model-specific <Model>PersistenceManager will have access directly to the device storage manipulating methods. Thus, we decided to use Composition whereby <Model>PersistenceManager only exposes higher level methods (i.e. save<Model>

e.g. saveScene for ScenePersistenceManager) and the specific file manipulation is then done by PersistenceManager object owned by <Model>PersistenceManager.

Design Consideration 2 - *SQLite vs JSON Persistence*

Initially, we wanted to try implementing persistence with SQLite. However, to delete an entire project, we will need to delete the project, its scenes, its shots, and its layers one by one.

This is quite tedious, so we decided to use JSON persistence with some modifications. Instead of saving an entire encoded project each time a Layer is updated, we are only going to save a Layer. This is because the structure of Project is way too big and nested (Project > Scene > Shot > Layer) so it is way more efficient to save a Layer immediately. The model structure is maintained with a device directory system (Project folder -> Scene folder -> Shot folder -> Layer folder) so deleting a top-level module will automatically delete its children components up to the lowest level (i.e. Layer).

To decouple between Model and Persistence, we also have PersistedModel which keeps relevant information about each Model and allows it to be encoded to JSON.

Runtime Structure

Persistence for Folder

While the structure of each Project is nested up to 4 levels deep, the structure of each Folder can be even deeper. Instead of directly storing the Folder model as its own structure, we took a by-reference approach for each folder to store its children:

1. At the main directory of the App (Documents directory), there is a Root ID encoded in JSON. This represents the ID of the root Folder (the first Folder that you are led to within the app).
2. Each Folder (model) is encoded in JSON and it contains its children's ID
3. Upon loading PersistedFolder from the Model, ModelFactory recursively builds the Folder structure by referencing to children's ID.

The ModelFactory is also in charge of converting PersistedModel to Model, from a PersistedModelTree (a typealias that serves as an intermediate raw structure)

```
typealias PersistedModelTree = [(PersistedProject,
                                 [(PersistedScene,
                                   [(PersistedShot,
                                     [PersistedLayer]
                                   )])])]
```

which is created by reading the storage from top (Project) to bottom (Layer) and converting each level to its PersistedModel equivalent. The PersistedModelTree is retrieved by PersistedModelLoader.

Once we have the Projects, the ModelFactory is once again used to convert PersistedFolder objects retrieved by PersistedModelLoader into Folders recursively, with a specific Root ID indicating the root Folder, and also places Projects at the proper places.

ModelObserver

Each Model has a list of observers whom it will notify when it has updated. For our app, the observers are ViewControllers. ModelObserver is a protocol and further divided into FolderObserver, ProjectObserver, SceneObserver, ShotObserver, and LayerObserver, each of which implements ModelObserver. It's only method is modelDidChange().

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

Each Model now keeps its own <Model>PersistenceManager. For example, a Scene will have a ScenePersistenceManager because instead of saving the entire Project as an encoded JSON, we now use a folder structure in our storage system.

The persistence is purely handled by Model (as per our main architecture diagram) and not separate from ViewControllers. To ensure this, each <Model>PersistenceManager is also made private and saving/delete methods are private and called by relevant Model whenever a mutating method (e.g. updateSomething, addSomething, deleteSomething) is called.

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.

Yongjing

- Directory View + View Controller and subcomponents
- Project, Scenes and Shot View + View Controllers
- Select, add, delete, move etc folders and project in the directory page
- Directory and Folder Model

Marcus

- Persistence Manager
- Mapping of PersistedModel to Model (ModelFactory)
- Updates to Model

Tian Fang

- Layer and its subcomponents
- Multithreading (for thumbnail generation and storage)
- Drawing (Shot Designer and Layer Table View)
- Storage (Previous)

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