Туториал: Создание Color Memory Game с ежедневными push-уведомлениями

Описание

В этом проекте мы соберём простую игру для тренировки памяти на **SwiftUI** с использованием паттерна **MVVM**.

Игроку показываются случайные цвета в течение выбранного времени, после чего он должен ответить, сколько уникальных цветов было показано.

Кроме того, приложение будет ежедневно отправлять **локальное уведомление**, напоминая вернуться в игру.

Цель

Вы научитесь:

- строить интерфейс на SwiftUI
- управлять состоянием игры с помощью `ViewModel` и таймера
- использовать `UserNotifications` для планирования локальных уведомлений
- связывать push-логику с жизненным циклом приложения через `AppDelegate`

1. UI: View (экран игры)

Начнём с интерфейса.

Основные элементы:

- **Заголовок и таймер**: сверху показываем название игры и обратный отсчёт, если игра идёт.

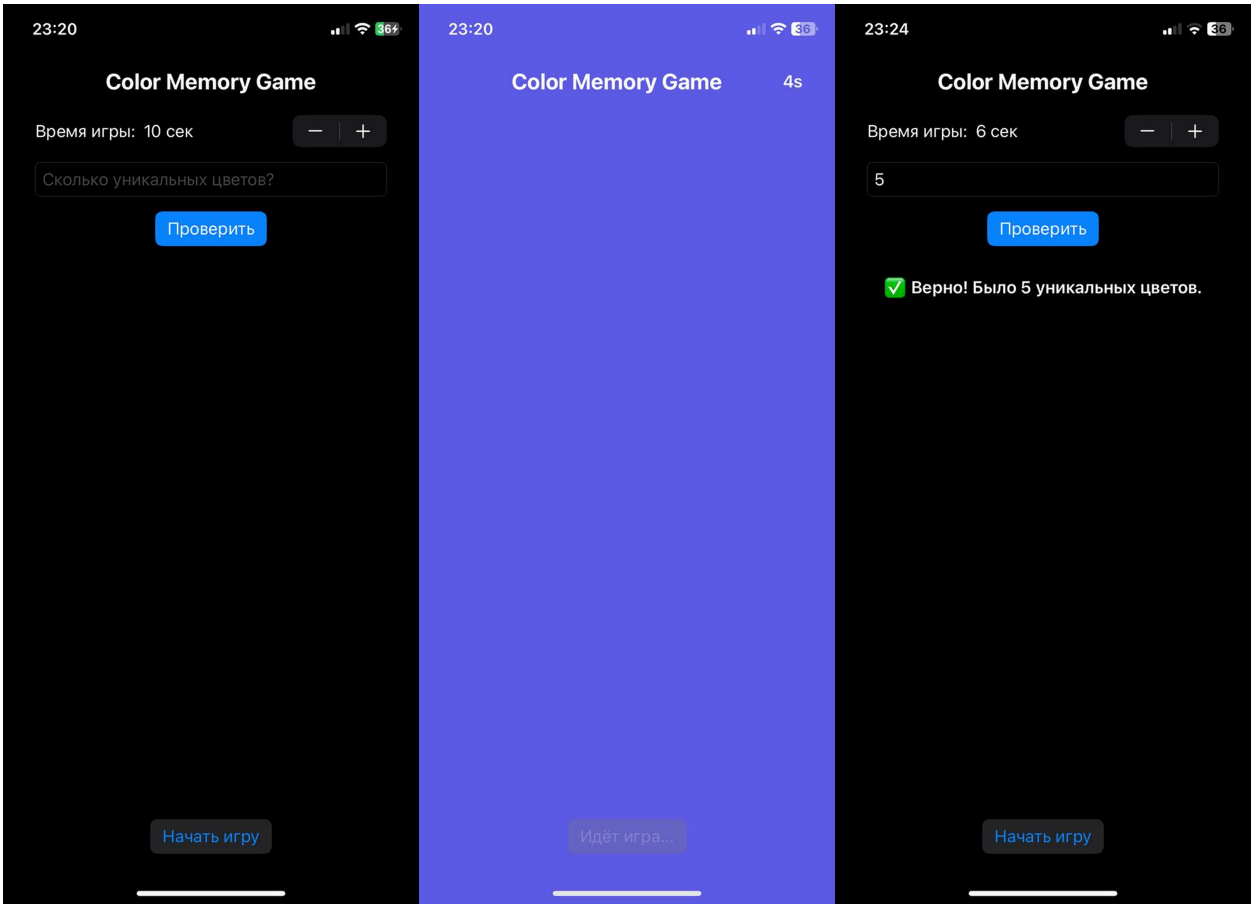
- **Настройки и ввод ответа**: пока игра не запущена, доступен `Stepper` для выбора длительности и `TextField` для ответа.
- **Кнопки**: одна для старта игры, другая для проверки ответа.

Пример:

```
struct GameView: View {
    @StateObject private var viewModel = GameViewModel()
    @State private var answer = ""

    var body: some View {
        VStack {
            HStack {
                Text("Color Memory Game")
                if viewModel.isPlaying {
                    Text("\(viewModel.timeLeft)s")
                }
            }

            if !viewModel.isPlaying {
                Stepper("Время игры: \(viewModel.gameDuration) сек",
                        value: $viewModel.gameDuration,
                        in: 5...30)

                TextField("Сколько уникальных цветов?", text: $answer)
                        .keyboardType(.numberPad)
                Button("Проверить") {
                    viewModel.checkAnswer(answer)
                }

                if let result = viewModel.result {
                    Text(result)
                }
            }

            Spacer()
```

```
            Button(viewModel.isPlaying ? "Идёт игра..." : "Начать игру") {
                viewModel.startGame()
                answer = ""
            }
            .disabled(viewModel.isPlaying)
        }
    }
}
```

Важно: в данном коде представлены базовые элементы view для демонстрации того, как они связаны с `viewModel`, вам предстоит разработать собственный вариант дизайна.



2. Логика игры: ViewModel

Вся бизнес-логика выносится в `GameViewModel`.

Основные задачи:

- хранить текущее состояние игры (`isPlaying`, `timeLeft`, `uniqueColors`)
- управлять таймером и сменой цветов
- проверять ответ игрока

2.1. Метод startGame()

Метод `startGame()` запускает саму игру. Сначала он сбрасывает всё, что связано с предыдущей сессией: очищает множество уникальных цветов, убирает результат с экрана, ставит таймер на выбранную длительность и помечает, что игра идёт.

Дальше идёт асинхронная задача через `Task`. Она запускает цикл, который повторяется столько раз, сколько секунд длится игра. Каждую секунду задача делает паузу (`Task.sleep`) - это и есть шаг таймера. После паузы выбирается случайный цвет из палитры и устанавливается как фон экрана, а сам цвет добавляется во множество уникальных цветов. Плюс уменьшается счетчик оставшегося времени.

Когда цикл заканчивается, метод ещё раз ждёт секунду, чтобы последний цвет был виден, а затем возвращает фон в нейтральный чёрный и ставит флаг `isPlaying` в `false`. В этот момент UI знает, что игра завершена, и появляется возможность ввести ответ.

Всё это асинхронно - UI не блокируется, таймер работает плавно.

```
func startGame() {
        uniqueColors.removeAll()
        result = nil
        timeLeft = gameDuration
        isPlaying = true

        Task {
            for _ in 0..<gameDuration {
                try? await Task.sleep(nanoseconds: 1_000_000_000)
                guard isPlaying else { break }
                if let newColor = palette.randomElement() {
                    currentColor = newColor
                    uniqueColors.insert(newColor)
                    timeLeft -= 1
                }
            }

            try? await Task.sleep(nanoseconds: 1_000_000_000)
            currentColor = .black
            isPlaying = false
        }
}
```

2.2. Метод `checkAnswer()`

Этот метод очень простой и понятный. Он берёт то, что ввёл игрок в поле `TextField`, пытается преобразовать в число и сравнивает с количеством уникальных цветов, которые успели появиться за игру.

Если число совпадает - устанавливается результат с отметкой «Верно!», если нет - показывается сообщение «Ошибка» с правильным количеством цветов. По сути, это мгновенная проверка ответа, которая обновляет `result`, а интерфейс тут же показывает пользователю, угадал он или нет.

```
func checkAnswer(_ answer: String) {
        guard let guess = Int(answer) else { return }
        let correct = uniqueColors.count

        if guess == correct {
            result = "✅ Верно! Было \(correct) уникальных цветов."
        } else {
            result = "❌ Ошибка. Было \(correct) уникальных цветов."
        }
    }
```

3. Push-уведомления

Чтобы игроки не забывали возвращаться к игре, мы хотим отправлять **ежедневные локальные уведомления**. Для этого удобно выделить отдельный класс - `NotificationManager`. Он будет отвечать за всё, что связано с уведомлениями: запрос разрешений и планирование напоминаний.

> Для этого создадим **синглтон** через `static let shared`. Теперь в приложении будет **один общий объект**. Синглтон позволяет не создавать новый менеджер каждый раз, а использовать его методы из любой части кода. Приватный `init()` защищает от случайного создания других экземпляров.

3.1. Метод `requestPermission()`

Этот метод запрашивает у пользователя разрешение на отображение уведомлений при первом запуске приложения.

```
func requestPermission() {
    UNUserNotificationCenter.current()
        .requestAuthorization(options: [.alert, .sound]) { _, _ in }
}
```

3.2. Метод `scheduleDailyNotification()`

Создадим **ежедневное напоминание**:

- Сначала создадим уведомление `UNMutableNotificationContent` и укажем его содержимое

```
let notification = UNMutableNotificationContent()
notification.title = "Color Memory Game"
notification.body = "Пора сыграть и потренировать память!"
notification.sound = .default
```

- **Время уведомления** задаём через `DateComponents`, например, каждый день в 21:00.

```
var dateComponents = DateComponents()
dateComponents.hour = 21
dateComponents.minute = 0
```

- Добавляем **триггер** через `UNCalendarNotificationTrigger` и с**оздаём запрос** `UNNotificationRequest` с уникальным идентификатором. После этого можно д**обавить запрос** через `UNUserNotificationCenter` для отправления ежедневных push-уведомлений

```
let trigger = UNCalendarNotificationTrigger(dateMatching: dateComponents,
                                            repeats: true)
let request = UNNotificationRequest(identifier: "dailyGameReminder",
                                    content: notification,
                                    trigger: trigger)

UNUserNotificationCenter.current().add(request)
```

4. Интеграция с AppDelegate и App

В iOS есть концепция **делегатов приложения**, и один из них - **UIApplicationDelegate**. Он отвечает за жизненный цикл приложения: когда оно запускается, переходит в фон, закрывается и так далее. Чтобы локальные уведомления работали корректно, **UNUserNotificationCenter** должен иметь своего делегата. И этот делегат обычно привязывают к AppDelegate.

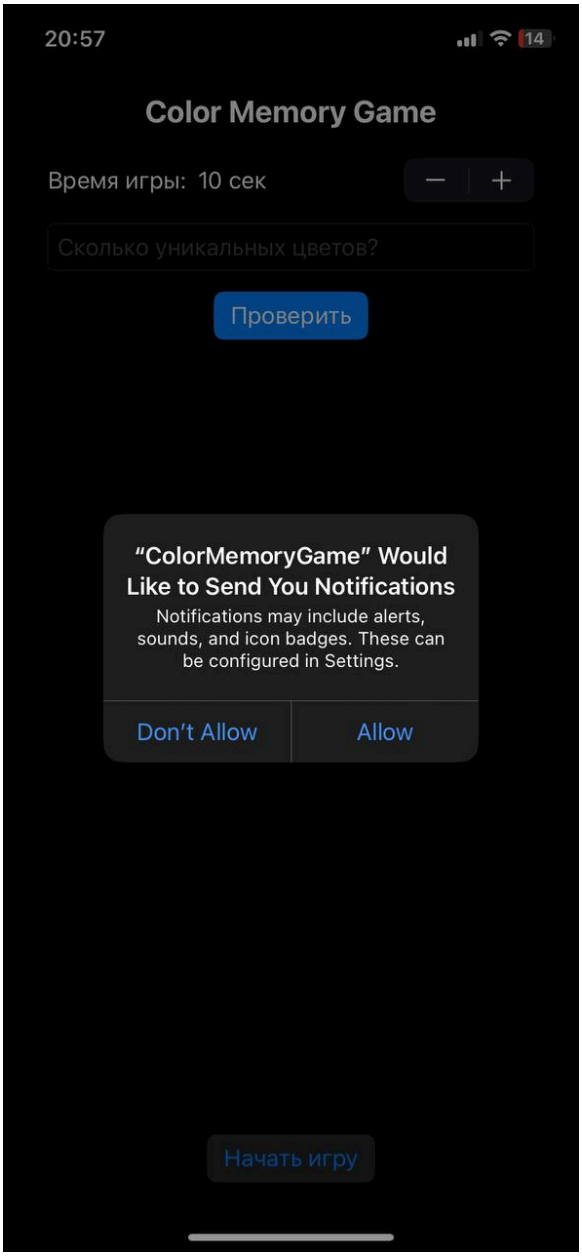
В нашем случае мы создаём **класс AppDelegate**, наследуемся от `NSObject` и реализуем протоколы `UIApplicationDelegate` и `UNUserNotificationCenterDelegate`.

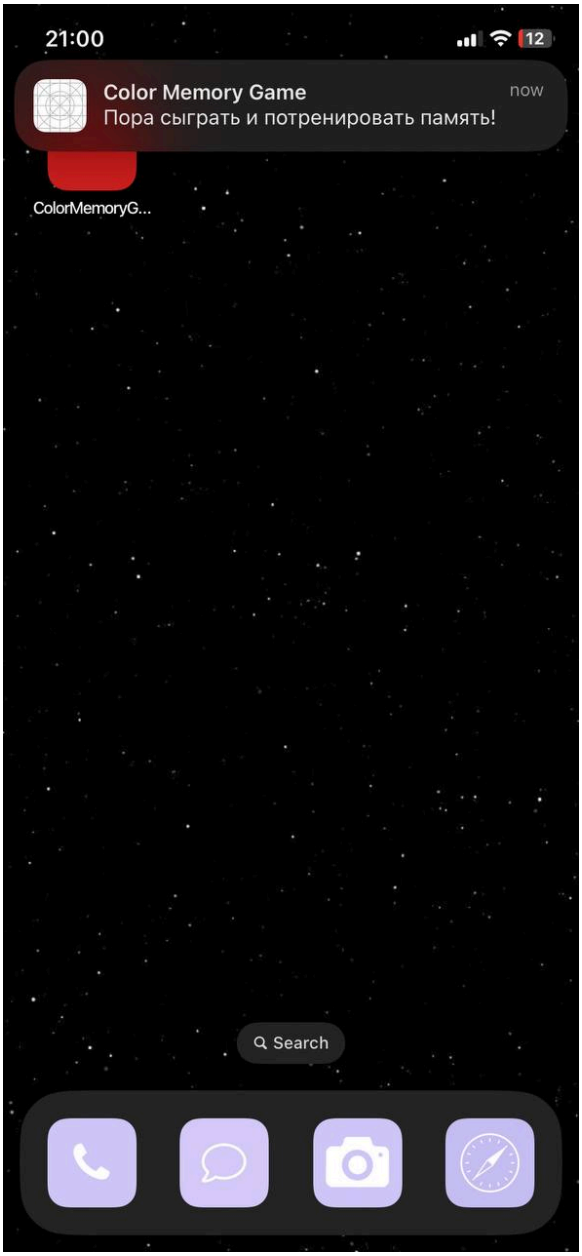
```
import UIKit
import UserNotifications

class AppDelegate: NSObject, UIApplicationDelegate, UNUserNotificationCent
erDelegate {
    func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOp
tionsKey : Any]? = nil
    ) -> Bool {
        UNUserNotificationCenter.current().delegate = self
        return true
    }
}
```

Теперь в точке входа приложения можно вызвать менеджер уведомлений:

```
@main
struct ColorMemoryGameApp: App {
    @UIApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            GameView()
```

```
            .onAppear {
                NotificationManager.shared.requestPermission()
                NotificationManager.shared.scheduleDailyNotification()
            }
        }
    }
}
```





Результат

Мы собрали приложение по шагам:

1. UI (`GameView`) - отображение игры
2. Логика (`GameViewModel`) - управление состоянием
3. Пуши (`NotificationManager`) - ежедневные напоминания
4. Интеграция (`AppDelegate`, `App`) - запуск уведомлений при старте

В итоге получилось минимальное, но рабочее приложение, которое тренирует память и напоминает вернуться к игре.