# Архитектурный мост: Фундаментальная теория перехода от Unity 3D к React Fiber Architecture для Senior-разработчиков

## Аннотация: Конвергенция инженерных парадигм

Переход высококвалифицированного разработчика из экосистемы Unity 3D в среду Enterprise React представляет собой не просто смену синтаксиса или инструментов, а фундаментальную миграцию архитектурного сознания. Для инженера, привыкшего к полному контролю над распределением памяти, управлению потоками и детерминированному игровому циклу (Game Loop), декларативная природа веб-разработки может показаться хаотичной или избыточно абстрактной. Однако при глубоком системном анализе становится очевидно, что современный React (версии 18 и выше), оснащенный архитектурой Fiber, режимом конкурентного рендеринга (Concurrent Mode) и сложными эвристиками согласования (Reconciliation), функционирует гораздо ближе к игровому движку, чем к классическим MVC-фреймворкам прошлого десятилетия.1

Данный отчет представляет собой исчерпывающее теоретическое руководство для «Второго дня» стратегического плана акселерации. Его цель — деконструировать механизмы рендеринга React через призму Unity, используя строгую терминологию: ECS (Entity Component System), Draw Call Batching, сериализация и Player Loop. Мы установим ментальную модель, в которой React рассматривается как «игровой движок пространства пользователя» (userspace game engine), работающий внутри среды выполнения JavaScript, с собственным планировщиком (Scheduler), графом сцены (Virtual DOM) и конвейером отрисовки (Commit Phase).1

## 1. Эволюция парадигмы: От Immediate Mode к Retained State

### 1.1. Императивный габитус против Декларативного контракта

В среде Unity, особенно при работе с устаревшими системами GUI (например, OnGUI) или при прямой манипуляции MonoBehaviour, разработчики привыкли к императивной модели управления. Логика изменения интерфейса неразрывно связана с прямым доступом к памяти объекта. Рассмотрим классический паттерн Unity для обновления состояния кнопки, который глубоко укоренился в сознании C#-разработчика:

C#

// Unity / C# - Императивный подход (Imperative Approach)  
public class ButtonController : MonoBehaviour {  
 public void UpdateButtonState(bool isClicked) {  
 // Шаг 1: Получение ссылки на компонент (прямой доступ к памяти)  
 var button = GetComponent<Button>();  
 var text = button.GetComponentInChildren<Text>();  
   
 // Шаг 2: Прямая мутация состояния объекта в куче (Heap)  
 if (isClicked) {  
 text.text = "Clicked";   
 text.color = Color.red;  
 } else {  
 text.text = "Click Me";  
 text.color = Color.black;  
 }  
 }  
}

Этот подход характеризуется ручной синхронизацией: разработчик несет ответственность за то, чтобы визуальное представление соответствовало логическому состоянию. Если логика обновляется, а вызов UpdateButtonState пропущен, интерфейс становится «черствым» (stale). Это аналогично режиму **Immediate Mode**, где инструкции отрисовки выдаются кадр за кадром.1

React требует перехода к **декларативной архитектуре**. В этой парадигме разработчик никогда не взаимодействует с API отрисовки (DOM) напрямую. Вместо этого он описывает *желаемое состояние* интерфейса для любого возможного набора данных. Формула, лежащая в основе этого подхода, выражается как $UI = f(State)$. React берет на себя роль «драйвера», который вычисляет разницу между текущим и желаемым состоянием и выполняет необходимые мутации.3

JavaScript

// React - Декларативный подход (Declarative Approach)  
function MyButton() {  
 // State - это "Serialized Field", изменение которого помечает объект как "Dirty"  
 const [isClicked, setIsClicked] = useState(false);  
  
 // UI - это проекция состояния.   
 // Мы декларируем ЧТО мы хотим видеть, а не КАК это изменить.  
 return (  
 <button   
 style={{ color: isClicked? 'red' : 'black' }}  
 onClick={() => setIsClicked(true)}  
 >  
 {isClicked? "Clicked" : "Click Me"}  
 </button>  
 );  
}

Для разработчика Unity этот сдвиг лучше всего иллюстрируется сравнением старого **IMGUI** (где код выполняется каждый кадр для отрисовки элементов) и нового **UI Toolkit** (Retained Mode, где создается дерево визуальных элементов, состояние которого хранится и управляется движком). React Component — это не сама отрисовка, а *чертеж* (blueprint) для отрисовки.3

### 1.2. Теоретические последствия декларативности для производительности

Декларативная модель вводит слой абстракции, который открывает возможности для оптимизаций, недоступных в строгих императивных системах. Разделяя описание интерфейса и его исполнение, React может пакетировать команды (batching), приоритизировать задачи рендеринга и даже отбрасывать кадры, если они устарели до момента вывода на экран — техники, знакомые графическим программистам, занимающимся оптимизацией пропускной способности GPU.5

В Unity, если в одном кадре выдать 1000 инструкций на перемещение GameObject, движок (в зависимости от реализации) может 1000 раз пересчитать матрицу трансформации (LocalToWorld), вызывая избыточную нагрузку на CPU (в частности, на пересчет иерархии). В декларативной модели React изменение состояния 1000 раз в синхронном цикле приведет к *единственному* проходу рендеринга благодаря механизму **Automatic Batching** (автоматическое пакетирование), полностью внедренному в React 18. Это эффективно разделяет логические обновления и графические обновления, аналогично тому, как Unity разделяет FixedUpdate (физика/логика) и Update (подготовка к рендеру).6

### 1.3. Сравнительная таблица парадигм исполнения

| **Характеристика** | **Unity (Imperative / MonoBehaviour)** | **React (Declarative / Functional)** | **Архитектурное следствие** |
| --- | --- | --- | --- |
| **Источник правды** | Распределен: состояние хранится в GameObject и скриптах. | Централизован: состояние отделено от представления (State/Props). | В React проще отслеживать потоки данных, но сложнее "хакнуть" систему быстрым прямым доступом. |
| **Мутация** | Прямая: transform.position = newVal | Косвенная: setState(newVal) | React требует иммутабельности для эффективного обнаружения изменений ($O(1)$ equality check). |
| **Синхронизация** | Ручная: вызов методов обновления UI. | Автоматическая: React Reconciliation. | Устраняет класс ошибок "рассинхронизация UI и данных". |
| **Контроль потока** | Полный: разработчик управляет каждым вызовом. | Инвертированный: Фреймворк вызывает ваш код. | React выступает как ОС для компонентов, управляя их жизненным циклом. |

## 2. Компонентная архитектура: Глубокая аналогия с Префабами

### 2.1. React Компонент как Чертеж Префаба (Prefab Blueprint)

Фундаментальной единицей построения приложения в React является Компонент. Для Senior Unity Developer критически важно перестроить восприятие: React Компонент — это не просто функция или класс, это **Префаб (Prefab)**.

* **Unity Prefab:** Сериализованный ассет (.prefab), определяющий шаблон для GameObject, включая его иерархию, компоненты (MonoBehaviours, Transforms) и значения свойств по умолчанию. Это *не* экземпляр на сцене, это *описание* того, как создать экземпляр.
* **React Component:** Функция JavaScript, определяющая шаблон для поддерева UI, включая HTML-элементы, стили и логику. Это *не* DOM-узел, это *описание* (JSX), которое возвращается движку.

Когда мы пишем <MyComponent /> в JSX, мы фактически вызываем аналог Instantiate(MyPrefab) в Unity. Мы отправляем запрос движку на создание экземпляра этого чертежа в графе сцены (в данном случае, в Virtual DOM).1

### 2.2. Иерархия и Композиция (Composition over Inheritance)

Архитектура Unity полагается на иерархию трансформаций (Transform hierarchy) — отношения «родитель-потомок», которые диктуют позицию, поворот и масштаб. React навязывает столь же строгую иерархию, но вместо распространения геометрических трансформаций, она распространяет **Данные (Data)** и **Контекст (Context)**.

В Unity сложные сущности часто конструируются путем композиции меньших, переиспользуемых префабов (например, префаб «Танк», содержащий дочерние префабы «Башня» и «Гусеницы»). React использует тот же принцип **Композиции вместо Наследования** (Composition over Inheritance). Вместо создания базового класса BaseButton и наследования от него SubmitButton, React поощряет создание компонента Button и его обертывание внутри компонента SubmitButton. Это позволяет избегать проблем "алмазного наследования" и жесткой связности, свойственных глубоким иерархиям классов в ООП.9

**Сравнение процессов инстанцирования:**

| **Этап** | **Unity (C#)** | **React (JSX)** | **Системное значение** |
| --- | --- | --- | --- |
| **Определение** | Создание Prefab Asset | Написание функции компонента | Определение структуры и поведения. |
| **Вызов** | Instantiate(prefab) | <Component /> | Запрос на выделение памяти и включение в граф сцены. |
| **Уничтожение** | Destroy(gameObject) | Условный рендеринг (cond && <Comp />) | Очистка памяти, снятие листенеров событий. |
| **Конфигурация** | SerializedFields в Инспекторе | Props передаваемые как аргументы | Начальная инициализация состояния. |

### 2.3. Маппинг Жизненного Цикла (Lifecycle Mapping)

Критически важным аспектом перехода является сопоставление жизненного цикла компонента React с методами MonoBehaviour в Unity. Понимание этого маппинга позволяет переносить паттерны управления ресурсами.

1. **Mounting (Монтирование) $\leftrightarrow$ Awake() / Start() / OnEnable()**
   * В React это фаза первоначальной отрисовки и выполнения эффектов с пустым массивом зависимостей: useEffect(() => {... },).
   * Здесь происходит инициализация, подписка на события, запуск сетевых запросов. Как и в Start(), это происходит один раз при создании экземпляра.
2. **Updating (Обновление) $\leftrightarrow$ Update() (Reactive)**
   * В Unity Update() вызывается каждый кадр безусловно. В React обновление компонента происходит *только* при изменении состояния (State) или входных данных (Props).
   * useEffect(() => {... }, [prop]) действует как реактивный Update(), который запускается только тогда, когда конкретная переменная (prop) изменилась с прошлого кадра. Это гораздо эффективнее слепого поллинга (polling) в Unity.10
3. **Unmounting (Размонтирование) $\leftrightarrow$ OnDestroy() / OnDisable()**
   * Когда компонент удаляется из DOM (например, при переходе на другую страницу или скрытии элемента), React вызывает функцию очистки (cleanup function), возвращаемую из useEffect.
   * Это критически важно для предотвращения утечек памяти: здесь нужно отписываться от событий, останавливать таймеры и закрывать соединения сокетов, точно так же, как в методе OnDestroy().1

## 3. Поток Данных: Props как Иммутабельные Сериализованные Поля

### 3.1. Контракт Иммутабельности

В Unity публичные поля (или private поля) в MonoBehaviour по умолчанию мутабельны. Скрипт на родительском объекте может получить ссылку на дочерний объект и изменить его свойства напрямую: child.health = 50;. Эта двунаправленная связность и возможность изменения состояния из любой точки кода часто приводит к запутанности ("spaghetti code") и состоянию гонки (race conditions) в сложных системах.

React навязывает строжайшую дисциплину: **Однонаправленный Поток Данных (One-Way Data Flow)**. "Props" (свойства) — это конфигурационные данные, передаваемые от родителя к ребенку.

Критическое различие: В React Props являются неизменяемыми (Read-Only).1

Дочерний компонент, получивший props.health, не имеет права выполнить props.health = 50;. Попытка сделать это нарушает фундаментальный архитектурный контракт React. Компонент должен быть "чистой функцией" относительно своих пропсов.

### 3.2. Callbacks против UnityEvents и Delegates

Чтобы изменить данные, которые "живут" в родительском компоненте (источник правды), дочерний компонент должен вызвать функцию, переданную ему через пропсы. Это концептуально идентично использованию паттерна **Delegate** или **UnityEvent** в C#.

**Паттерн в Unity C#:**

C#

// Parent (Владелец состояния)  
public class GameController : MonoBehaviour {  
 private int score = 0;  
 void Start() {  
 var enemy = GetComponent<Enemy>();  
 // Подписка на делегат/событие  
 enemy.OnDeath += HandleEnemyDeath;  
 }  
 void HandleEnemyDeath() {  
 score += 10; // Мутация состояния происходит во владельце  
 }  
}  
  
// Child (Инициатор события)  
public class Enemy : MonoBehaviour {  
 public Action OnDeath; // Делегат  
 public void Die() {  
 OnDeath?.Invoke(); // Вызов колбэка  
 }  
}

**Паттерн в React:**

JavaScript

// Parent (Владелец состояния)  
function GameController() {  
 const = useState(0);  
 // Функция-обработчик (Handler)  
 const handleEnemyDeath = () => setScore(s => s + 10);  
  
 // Передача делегата через Prop  
 return <Enemy onDeath={handleEnemyDeath} />;  
}  
  
// Child (Инициатор события)  
function Enemy({ onDeath }) {   
 return <button onClick={onDeath}>Die</button>; // Вызов колбэка  
}

Это обеспечивает **Владение Данными (Data Ownership)**. Состояние (score) остается инкапсулированным в родителе, а ребенок (Enemy) остается "глупым" (dumb) компонентом представления. Это разделение зеркально отражает принципы **Entity-Component-System (ECS)**, где данные (Component) отделены от логики обработки (System).12

### 3.3. Параллели с Data-Oriented Design (DOD)

Запрет на мутацию пропсов согласуется с принципами **Data-Oriented Design (DOD)**, которые пропагандируют трансформацию данных вместо мутации объектов. В Unity DOTS (Data-Oriented Technology Stack) системы итерируются по плотно упакованным массивам компонентов (IComponentData), читают из них и пишут в выходные буферы; они не мутируют глобальное состояние произвольно. React-компоненты работают аналогично: они принимают props (входные данные) и state (локальные данные) и выдают UI description (выходные данные).

Сравнение **Иммутабельности в React** и **Unity**:

* **Unity:** Избегает создания новых объектов ("Garbage Free" code), так как старый GC (Boehm) был медленным. Используются пулы объектов и мутируемые структуры.
* **React:** Создает новые ссылки на объекты при *каждом* обновлении состояния. Для C#-разработчика это выглядит как кошмар для GC. Однако современные JS-движки (V8) чрезвычайно оптимизированы для аллокации короткоживущих объектов (nursery generation) с помощью сборщика мусора Scavenge.1 Более того, иммутабельность позволяет использовать **проверку равенства ссылок** (prevProps === nextProps), что является операцией $O(1)$. Это позволяет пропускать рендеринг целых поддеревьев (аналог Occlusion Culling или проверки transform.hasChanged перед пересчетом физики).14

## 4. Движок Рендеринга: Virtual DOM и Согласование (Reconciliation)

### 4.1. Миф о "Медленном DOM" против Драйвера Видеокарты

Чтобы понять суть Virtual DOM, необходимо провести аналогию с графическим конвейером (Graphics Pipeline). В 3D-рендеринге вызов отрисовки (Draw Call, например, Graphics.DrawMesh) является дорогостоящей операцией, так как требует перехода через границу CPU-GPU, переключения контекста драйвера, валидации состояний и передачи данных по шине.

В веб-разработке **DOM (Document Object Model)** — это "Граф Сцены" браузера. Модификация DOM (например, element.style.width = '50%') — это эквивалент Draw Call в вебе. Это действие запускает движок компоновки браузера (layout engine), вызывая пересчет геометрии (**Reflow**) и растеризацию пикселей (**Repaint**). Это чрезвычайно дорогие операции.17

Если скрипт Unity выполнит GetComponent<Renderer>().material.color = Random.Color; 1000 раз за кадр, производительность упадет из-за накладных расходов API. Точно так же, 1000 прямых обращений к DOM "убивают" производительность страницы.

### 4.2. Virtual DOM как Command Buffer

**Virtual DOM** в React следует рассматривать как **Command Buffer** (буфер команд) или теневую копию графа сцены в Unity. Это легковесное представление UI в виде простых JavaScript-объектов, существующее исключительно в системной памяти (Heap).

Когда компонент React "рендерится", он не касается браузера. Он "рисует" в этот Virtual DOM. Это происходит мгновенно, сравнимо с манипуляцией обычными C# классами в памяти.

**Процесс Согласования (Diffing Algorithm):**

1. **Render Phase:** React выполняет функции компонентов и строит новое дерево Virtual DOM.
2. **Diffing:** Движок сравнивает новое дерево с деревом предыдущего кадра.
3. **Commit Phase:** Движок вычисляет *минимальный набор операций*, необходимых для приведения реального DOM в соответствие с новым Virtual DOM.

Этот процесс концептуально идентичен **Draw Call Batching** или использованию **Dirty Flags** в игровых движках. React "батчит" (группирует) все изменения и применяет их одним пакетом, минимизируя "накладные расходы драйвера" (Reflow браузера).19

### 4.3. Алгоритмическая сложность: Эвристика O(n)

Классический алгоритм нахождения минимального количества редакций (edit distance) между двумя деревьями имеет сложность $O(n^3)$, где $n$ — количество элементов. Для UI с 1000 элементами это потребовало бы 1 миллиард операций сравнения — неприемлемо для бюджета кадра в 16 мс.

React применяет **Эвристический Алгоритм**, снижающий сложность до $O(n)$.22 Он базируется на двух допущениях, имеющих прямые параллели с оптимизациями игровых движков:

1. **Стабильность Типов (Type Stability):** Если тип компонента изменился (например, <div> стал <span>, или Enemy стал Player), React предполагает, что все поддерево изменилось. Он уничтожает старое дерево и строит новое с нуля. Это аналог Destroy(oldObject); Instantiate(newObject); в Unity.
2. **Стабильность Ключей (Key Stability):** Разработчик предоставляет уникальные идентификаторы (keys), подсказывая движку, какие дочерние элементы остаются стабильными между рендерами.

### 4.4. Критическая роль Ключей (Keys vs InstanceID)

В Unity каждый Object имеет уникальный InstanceID, который сохраняется независимо от позиции объекта в иерархии или массиве дочерних трансформаций. Движок использует этот ID для сериализации ссылок и отслеживания объектов.

В React при рендеринге динамических списков (массивов данных) фреймворк по умолчанию не знает, был ли элемент перемещен, вставлен или удален. Он видит только новый порядок элементов.

Ошибка "Индекс как Ключ" (Index as Key Fallacy):

Использование индекса массива (0, 1, 2) в качестве ключа — распространенная ошибка новичков, приводящая к тяжелым багам.

* **Сценарий:** Список . Удаляем `A`. Новый список .
* **Логика React с индексами:**
  + Элемент с ключом 0 был A, стал B. React думает: "Элемент 0 изменил контент".
  + Элемент с ключом 1 был B, стал C. React думает: "Элемент 1 изменил контент".
  + Элемент с ключом 2 исчез. React уничтожает его.
* **Результат:** React *мутирует* DOM-узлы 0 и 1 и уничтожает узел 2. Если внутри компонентов было локальное состояние (например, введенный текст или анимация), оно "перетечет" к неправильным данным, так как физический экземпляр компонента был переиспользован.

Правильный подход (Database IDs):

Использование стабильного ID (например, UUID из базы данных) работает как Unity InstanceID.

* **Логика React со стабильными ключами:**
  + Ключ ID\_A отсутствует. Уничтожить компонент.
  + Ключ ID\_B существует. Переместить его на позицию 0.
  + Ключ ID\_C существует. Переместить его на позицию 1.
* **Результат:** React переупорядочивает существующие DOM-узлы. Внутреннее состояние компонентов сохраняется. Это аналогично **Object Pooling** в Unity, где сохранение идентичности объекта позволяет избежать затрат на повторную инициализацию.24

## 5. React Fiber: «Игровой Цикл» Веба

Наиболее сложная и глубокая аналогия для Senior-разработчика — это сопоставление **Unity Player Loop** и архитектуры **React Fiber**. Именно здесь теория "Второго дня" выходит за рамки базовых туториалов и погружается в системное программирование.

### 5.1. Исторический контекст: Stack vs Fiber

До версии React 16 использовался "Stack Reconciler". Он работал как стандартная синхронная рекурсивная функция. Как только начинался рендеринг, он должен был пройти все дерево компонентов до конца, прежде чем вернуть управление браузеру.

* **Аналогия с Unity:** Представьте, что вы поместили всю логику игры, физику и AI в один метод Update(), выполнение которого занимает 200 мс. Игра "зависнет" (drop frames), так как кадр не может быть отрисован, пока Update() не завершится.

**React Fiber** — это полная перепись ядра для решения этой проблемы. Fiber вводит возможность **приостанавливать, прерывать и приоритизировать** работу. Фактически, это реализация кооперативного многозадачного планировщика (cooperative multitasking scheduler) внутри однопоточной среды JavaScript.2

### 5.2. Анатомия Fiber-ноды (Fiber Node)

"Fiber" — это легковесный поток выполнения, или **Единица Работы (Unit of Work)**.

* **Структура данных:** Fiber Node — это JavaScript-объект, содержащий состояние компонента, пропсы и указатели на место в дереве.
* **Обход Связного Списка:** В отличие от рекурсивного стека вызовов, Fiber использует структуру односвязного списка (указатели child, sibling, return). Это позволяет алгоритму обхода быть итеративным (цикл while), а не рекурсивным. Это критически важно, так как цикл можно поставить на паузу (сохранить указатель на текущую ноду) и возобновить позже, тогда как глубокий стек рекурсии прервать невозможно без его раскрутки.30

**Аналогия с Unity:** Fiber Node сравнима со структурой Job в системе **C# Job System**. Она содержит данные, необходимые для выполнения конкретной задачи (обновление компонента), и указатели на следующую задачу.

### 5.3. Модель Двойной Буферизации (Double Buffering)

Графические движки используют двойную буферизацию (Front Buffer / Back Buffer) для предотвращения артефактов отрисовки (tearing). Один буфер отображается на экране, пока GPU пишет в другой. React Fiber использует идентичную архитектуру для дерева компонентов.

1. **Current Tree (Текущее дерево):** Дерево, которое в данный момент отображается на экране (Front Buffer).
2. **WorkInProgress (WIP) Tree (Дерево в работе):** Дерево, которое React строит в памяти (Back Buffer).

Во время обновления React клонирует узлы из Current дерева в WorkInProgress. Все изменения применяются к WIP-дереву. Только когда все WIP-дерево готово и согласовано, React "меняет буферы" (этап Commit), делая WIP-дерево новым Current-деревом. Это предотвращает отображение частично обновленного или несогласованного UI пользователю.34

### 5.4. Разделение Фаз: Render vs Commit

React разделяет свою работу на две четкие фазы, которые идеально накладываются на этапы игрового цикла:

| **Фаза** | **React Fiber** | **Unity Engine** | **Характеристики** |
| --- | --- | --- | --- |
| **Render Phase** | Вычисление изменений, diffing, построение WIP-дерева. Вызов функций компонентов. | FixedUpdate() / Логическая часть Update() | **Асинхронная / Прерываемая.** Может быть поставлена на паузу, если бюджет кадра (5-16 мс) исчерпан. Чистые вычисления, никаких видимых побочных эффектов. |
| **Commit Phase** | Применение изменений к DOM, вызов useEffect. | Render() / Draw Calls / Swap Buffers | **Синхронная / Непрерываемая.** Должна выполниться за один проход, чтобы обеспечить визуальную целостность. "Красит" результат на экран. |

Time Slicing (Временное разделение):

В конкурентном режиме React проверяет время, оставшееся в текущем кадре (используя requestIdleCallback или полифил MessageChannel). Если Render Phase занимает слишком много времени, Fiber сохраняет текущий указатель стека в куче и возвращает управление браузеру (yield), чтобы тот мог обработать ввод пользователя или анимацию. В следующем свободном слоте времени React возобновляет работу с того же места.

* **Аналогия с Unity:** Это в точности то, как работают **Корутины (Coroutines)**. Корутина прерывает выполнение (yield return null), позволяя игровому циклу продолжиться, и возобновляется в следующем кадре. Fiber — это, по сути, автоматический системный менеджер корутин для рендеринга UI.6

## 6. Синтез архитектурных паттернов

### 6.1. Иммутабельность пропсов и DOD

Строгая иммутабельность пропсов в React усиливает подход DOD. В Unity ECS (DOTS) системы итерируются по упакованным массивам данных (IComponentData). Системы не "владеют" данными; они их обрабатывают. В React компоненты не "владеют" данными, переданными им через пропсы; они их только рендерят. Это разделение **Данных** (State/Context) и **Логики/Представления** (Component) обеспечивает высокую предсказуемость и упрощает отладку. Оно устраняет "спагетти" из ссылок, где ObjectA меняет приватные поля ObjectB напрямую.

### 6.2. Синтетические События: Система Ввода (Input System)

React не вешает листенеры событий (addEventListener) на каждый DOM-узел (что было бы дорого по памяти, как GetComponent в Update). Вместо этого он использует **Делегирование Событий (Event Delegation)**. React вешает один глобальный листенер на корень документа. Когда происходит событие (клик, ввод), оно всплывает (bubbles up) до корня, и внутренняя система событий React (Synthetic Event System) перехватывает его и маппит на соответствующий компонент.

* **Аналогия с Unity:** Это похоже на **New Input System** в Unity, которая абстрагирует сырые аппаратные события в логические действия (Actions), или на глобальный InputManager, который диспетчеризирует события заинтересованным наблюдателям, вместо того чтобы каждый объект опрашивал Input.GetKeyDown каждый кадр.36

### 6.3. Виртуализация списков (Окклюзия)

В веб-разработке рендеринг списка из 10,000 элементов убивает браузер. Решение — **Виртуализация** (Windowing). Рендерится только то, что видно во вьюпорте, плюс небольшой буфер. При скролле DOM-узлы не создаются заново, а переиспользуются или подменяются (в зависимости от реализации библиотеки).

* **Аналогия с Unity:** Это прямой аналог **Occlusion Culling** (отсечение невидимой геометрии) и **LOD (Level of Detail)**. Мы не отправляем на GPU то, что камера не видит.

## 7. Практическая реализация: Архитектура Календаря

Для закрепления теории "Второго дня" мы спроектируем архитектуру сервиса "Календарь", используя принципы композиции и однонаправленного потока данных.

### 7.1. Иерархия компонентов (Scene Graph)

Структура папок и компонентов должна отражать функциональное деление, подобно префабам в проекте Unity:

src/features/calendar/  
├── components/  
│ ├── CalendarView.jsx // (Canvas/Root Panel) - Контейнер состояния  
│ ├── CalendarHeader.jsx // (UI Header) - Навигация  
│ ├── DaysGrid.jsx // (Grid Layout Group) - Сетка дней  
│ └── DayCell.jsx // (Day Prefab) - Ячейка дня

### 7.2. Реализация DayCell (Аналог скрипта Префаба)

Ниже представлен пример компонента DayCell, демонстрирующий передачу данных и событий. Обратите внимание на отсутствие прямой связи с родителем (GetComponentInParent запрещен).

JavaScript

// DayCell.jsx  
// Аналог: public class DayCell : MonoBehaviour  
  
// props - это ReadOnly данные, передаваемые при инстанцировании  
function DayCell({ date, events, onDayClick }) {  
   
 // Локальная логика презентации (как в Update для визуальных эффектов)  
 const isToday = new Date().toDateString() === date.toDateString();  
   
 // Стиль вычисляется на лету (аналог Material Property Block)  
 const cellStyle = {  
 backgroundColor: isToday? '#e6f7ff' : 'white',  
 border: '1px solid #ddd'  
 };  
  
 return (  
 // onClick - это подписка на UnityEvent  
 <div style={cellStyle} onClick={() => onDayClick(date)}>  
 <span className="day-number">{date.getDate()}</span>  
   
 {/\* Рендеринг списка событий (дочерние объекты) \*/}  
 <div className="events-list">  
 {events.map(evt => (  
 // ВАЖНО: key обязателен для Reconciliation!  
 <div key={evt.id} className="event-marker">  
 {evt.title}  
 </div>  
 ))}  
 </div>  
 </div>  
 );  
}

### 7.3. Реализация CalendarView (Аналог GameController)

Родительский компонент управляет состоянием и "инстанцирует" дочерние элементы.

JavaScript

// CalendarView.jsx  
function CalendarView() {  
 // State - сериализованные поля, управляющие логикой  
 const = useState(new Date());  
 const [events, setEvents] = useState();  
  
 // Метод-обработчик (Delegate target)  
 const handleDayClick = (clickedDate) => {  
 console.log("Day clicked:", clickedDate);  
 // Здесь может быть логика открытия модального окна  
 };  
  
 // Генерация данных для сетки (Procedural Generation)  
 const daysInMonth = getDaysInMonth(currentDate);   
  
 return (  
 <div className="calendar-container">  
 <CalendarHeader   
 date={currentDate}   
 onNextMonth={() => setCurrentDate(addMonths(currentDate, 1))}  
 />  
   
 <DaysGrid>  
 {daysInMonth.map(day => (  
 // Инстанцирование префаба DayCell  
 <DayCell   
 key={day.toISOString()} // Стабильный ID (InstanceID)  
 date={day}  
 events={events.filter(e => isSameDay(e.date, day))}  
 onDayClick={handleDayClick} // Передача колбэка (Action)  
 />  
 ))}  
 </DaysGrid>  
 </div>  
 );  
}

## 8. Стратегические рекомендации для обучения

1. **Примите ментальную модель "Чертежа":** Когда вы пишете компонент, думайте: «Я пишу скрипт, который генерирует Префаб», а не «Я рисую пиксель».
2. **Уважайте Иммутабельность:** Сопротивляйтесь желанию мутировать переменные. В C# мы оптимизируем, мутируя структуры in-place, чтобы избежать GC. В JS/React мы оптимизируем, создавая поверхностные копии (shallow copies), чтобы использовать проверку равенства ссылок ($O(1)$) для пропуска рендеринга. Движок V8 справится с мусором.
3. **Ключи — это святое:** Относитесь к пропу key с тем же уважением, что и к Primary Key в базе данных или InstanceID в Unity. Это якорь идентичности в текучей, реактивной системе.
4. **Визуализируйте Кадр:** При отладке React представляйте "Render Phase" как ваши вычисления логики/физики (которые могут происходить многократно или быть отброшены), а "Commit Phase" — как фактическую отрисовку кадра. Побочные эффекты (сетевые вызовы, подписки) должны находиться *после* рендера (в useEffect), точно так же, как вы не стали бы делать сетевой запрос внутри шейдера.

## Заключение

Переход от Unity к React — это не шаг назад в "более простую" среду, а латеральное перемещение в другой тип высокопроизводительного движка рендеринга. Сопоставляя **Компоненты с Префабами**, **Virtual DOM с Пакетированием Вызовов Отрисовки**, а **Fiber с Игровым Циклом**, Senior Unity Developer может обойти "магию" веб-туториалов и постичь механическую реальность библиотеки. React — это система рендеринга в режиме retained-mode, с двойной буферизацией и кооперативной многозадачностью. Понимание этого на архитектурном уровне гарантирует способность создавать масштабируемые, производительные и надежные энтерпрайз-приложения с той же точностью, которая применяется в AAA-геймдеве.

#### Источники

1. Интенсив React для Backend Разработчика неделя 10
2. Understanding React's Fiber Tree: A Deep Dive into React's Architecture and Rendering Process - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/gervaisamoah/understanding-reacts-fiber-tree-a-deep-dive-into-reacts-architecture-and-rendering-process-2loo>
3. Imperative vs Declarative Programming - ui.dev, дата последнего обращения: декабря 6, 2025, <https://ui.dev/c/react/imperative-vs-declarative>
4. Declarative vs Imperative Programming | by Ian Mundy - codeburst, дата последнего обращения: декабря 6, 2025, <https://codeburst.io/declarative-vs-imperative-programming-a8a7c93d9ad2>
5. Imperative vs Declarative Programming - ui.dev, дата последнего обращения: декабря 6, 2025, <https://ui.dev/imperative-vs-declarative-programming>
6. React Scheduler & Fiber Architecture Explained - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/yorgie7/react-scheduler-fiber-architecture-explained-time-slicing-concurrent-rendering-wip-management-3knh>
7. React Fiber Implementation In Detail… | by Abhishek - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@mailabhi555/react-fiber-implementation-vs-stack-implementation-916ef69a707c>
8. Which do YOU use: Prefabs or AddComponent? : r/Unity3D - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/Unity3D/comments/3qqh2a/which_do_you_use_prefabs_or_addcomponent/>
9. Question about Classes vs Prefabs : r/Unity3D - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/Unity3D/comments/qmobrm/question_about_classes_vs_prefabs/>
10. React lifecyle: phases - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/57190502/react-lifecyle-phases>
11. Thinking in React, дата последнего обращения: декабря 6, 2025, <https://react.dev/learn/thinking-in-react>
12. Data-Oriented Frontend Development - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/basal/data-oriented-frontend-development-1mk3>
13. A Complete Guide to React Architecture Patterns | by Zeel Shah - Medium, дата последнего обращения: декабря 6, 2025, <https://devshi-bambhaniya.medium.com/a-complete-guide-to-react-architecture-patterns-ea386d2ba327>
14. Understanding Mutable vs. Immutable Data in React and Their Impact on Rendering, дата последнего обращения: декабря 6, 2025, <https://dev.to/muthuraja_r/understanding-mutable-vs-immutable-data-in-react-and-their-impact-on-rendering-ldc>
15. The Power of Immutable Data Structures in React: A Deep Dive | by Harry Bloch | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@Blochware/the-power-of-immutable-data-structures-in-react-a-deep-dive-c1d8ea7f11ff>
16. Why is immutability so important (or needed) in JavaScript? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/34385243/why-is-immutability-so-important-or-needed-in-javascript>
17. How does React's Virtual DOM improve performance compared to direct manipulation of the actual DOM? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/76648822/how-does-reacts-virtual-dom-improve-performance-compared-to-direct-manipulation>
18. React Reconciliation, Fiber, and Virtual DOM Explained (Without the Jargon), дата последнего обращения: декабря 6, 2025, <https://dev.to/ad99526/react-reconciliation-fiber-and-virtual-dom-explained-without-the-jargon-51ng>
19. Draw call batching - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/2021.3/Documentation/Manual/DrawCallBatching.html>
20. Unity Draw Call Batching: The Ultimate Guide [2021] - The Gamedev Guru, дата последнего обращения: декабря 6, 2025, <https://thegamedev.guru/unity-performance/draw-call-optimization/>
21. Render and Commit - React, дата последнего обращения: декабря 6, 2025, <https://react.dev/learn/render-and-commit>
22. Reconciliation - React, дата последнего обращения: декабря 6, 2025, <https://legacy.reactjs.org/docs/reconciliation.html>
23. Reconciliation | React, дата последнего обращения: декабря 6, 2025, <https://shripadk.github.io/react/docs/reconciliation.html>
24. React Key vs Id - JavaScript - The freeCodeCamp Forum, дата последнего обращения: декабря 6, 2025, <https://forum.freecodecamp.org/t/react-key-vs-id/614749>
25. How and when to use keys in React - Cassidy Williams, дата последнего обращения: декабря 6, 2025, <https://cassidoo.co/post/react-keys/>
26. Performance implication of duplicate keys on React children in large lists - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/40579359/performance-implication-of-duplicate-keys-on-react-children-in-large-lists>
27. What should I use for key instead of id? : r/reactjs - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/reactjs/comments/1buk6zc/what_should_i_use_for_key_instead_of_id/>
28. An Introduction to React Fiber - The Algorithm Behind React - Velotio Technologies, дата последнего обращения: декабря 6, 2025, <https://www.velotio.com/engineering-blog/react-fiber-algorithm>
29. What is React Fiber and How It Helps You Build a High-Performing React Applications, дата последнего обращения: декабря 6, 2025, <https://sunnychopper.medium.com/what-is-react-fiber-and-how-it-helps-you-build-a-high-performing-react-applications-57bceb706ff3>
30. дата последнего обращения: декабря 6, 2025, <https://medium.com/@aliWhosane/react-fiber-data-structure-demystified-d3794470a8a4#:~:text=The%20data%20structure%20is%20a,%2C%20props%2C%20state%2C%20etc.>
31. A deep dive into React Fiber - LogRocket Blog, дата последнего обращения: декабря 6, 2025, <https://blog.logrocket.com/deep-dive-react-fiber/>
32. React Fiber — Secret ingredient in React's Reconciliation | by Pankaj Tanwar - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@pnk.tanwar/react-fiber-secret-ingredient-in-reacts-reconciliation-6f156be43dcc>
33. React Fiber Architecture - An Overview, дата последнего обращения: декабря 6, 2025, <https://tusharf5.com/posts/react-fiber-overview/>
34. A Deep Dive into React Fiber — The Engine Behind Modern React | Code With Seb Blog, дата последнего обращения: декабря 6, 2025, <https://www.codewithseb.com/blog/deep-dive-into-react-fiber-the-engine-behind-modern-react>
35. Introduction to React Fiber - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/jehnz/introduction-to-react-fiber-48c4>
36. React Synthetic Events Explained: Complete Guide for Developers - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/yorgie7/react-synthetic-events-explained-complete-guide-for-developers-3o2j>
37. Understanding React's Synthetic Event System - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/42327331/understanding-reacts-synthetic-event-system>