# Стратегический Отчет: Архитектурная Миграция от Unity CLR к V8 Runtime (Фаза III, День 1)

## Введение: Эпистемологический сдвиг инженерной парадигмы

Данный отчет представляет собой исчерпывающую теоретическую базу первого дня интенсива «React для Backend Разработчика», разработанную специально для специалистов уровня Senior Unity Developer и.NET Architect. Документ служит фундаментальным мостом между детерминированной, строго типизированной средой выполнения Unity (.NET CLR/IL2CPP) и событийно-ориентированной, динамической средой выполнения JavaScript (V8 Engine).

Переход от разработки игровых движков к разработке Enterprise-интерфейсов на React требует не просто изучения нового синтаксиса, а глубокой реконфигурации ментальных моделей управления памятью, потоками и жизненным циклом объектов. В Unity инженер привык к полному контролю над каждым тактом процессора внутри Game Loop. В среде браузера инженер должен принять модель «инверсии управления», где рендеринг является не гарантированным этапом каждого кадра, а побочным эффектом изменения состояния, управляемым сложной эвристикой Event Loop.

Настоящий анализ деконструирует архитектурные паттерны React и JavaScript через призму низкоуровневого устройства игровых движков, сопоставляя механизмы JIT-компиляции, сборки мусора и асинхронного планирования.

## 1. Среда Исполнения: Анатомия Runtime и Модели Конкурентности

Фундаментальное различие между Unity и React приложением кроется в том, кто владеет «главным циклом» (Main Loop). Понимание этого различия критично для предотвращения блокировок UI и утечек памяти.

### 1.1. Детерминизм против Реактивности

#### 1.1.1. Unity PlayerLoop: Императивный Детерминизм

В экосистеме Unity приложение существует внутри жестко структурированного цикла PlayerLoop.1 Это иерархическая система, где каждый кадр (Frame) состоит из последовательного вызова подсистем:

1. **Initialization:** Awake, OnEnable.
2. **Physics:** FixedUpdate (выполняется с фиксированным шагом времени, независимо от частоты кадров).
3. **Input:** Опрос устройств ввода.
4. **Game Logic:** Update, LateUpdate.
5. **Rendering:** Culling, Batching, отправка команд в GPU.

Ключевая характеристика Unity — **доминирование потока**. Главный поток (Main Thread) агрессивно потребляет ресурсы CPU, стремясь выполнить все инструкции в пределах бюджета кадра (например, 16.6 мс для 60 FPS). Если логика в Update() занимает больше времени, происходит падение FPS, но цикл продолжает свою работу, пытаясь нагнать время.2 Разработчик императивно указывает движку, *что* делать в каждый момент времени.

#### 1.1.2. V8 Event Loop: Кооперативная Реактивность

JavaScript в браузере (движок V8) работает на основе **Event Loop** (Цикла Событий). Это система, основанная на прерываниях и очередях сообщений. В отличие от Unity, где Update() вызывается безусловно, JS-движок большую часть времени может находиться в состоянии ожидания (idle), «спящем» режиме, пока не произойдет внешнее событие (I/O, таймер, взаимодействие с UI).4

Архитектура Event Loop состоит из следующих компонентов, которые необходимо мапить на знания Unity-разработчика:

| **Компонент V8** | **Аналог в Unity /.NET** | **Описание и Поведенческие Различия** |
| --- | --- | --- |
| **Call Stack (Стек вызовов)** | Stack (Потоковый стек) | LIFO структура. В JS стек один на весь контекст (вкладку). Если стек занят вычислением, браузер «зависает» полностью: не работает рендеринг, не обрабатываются клики. В Unity зависание Update останавливает игру, но не обязательно ОС (в редакторе). |
| **Web APIs (Browser Context)** | Native Plugins / OS Threads | Потоки вне V8 (написанные на C++), которые обрабатывают таймеры (setTimeout), сетевые запросы (fetch/XHR) и события DOM. Это аналог ThreadPool или нативных вызовов в Unity.6 |
| **Task Queue (Macrotasks)** | Очередь Update следующего кадра | FIFO очередь. Сюда попадают колбэки от таймеров и I/O. Они выполняются по *одному* за итерацию цикла. |
| **Microtask Queue (Jobs)** | LateUpdate / SynchronizationContext | Очередь с *высшим приоритетом*. Сюда попадают промисы (Promise, async/await) и MutationObserver. Микрозадачи выполняются **сразу** после очистки стека, но **до** того, как браузеру будет разрешено перерисовать кадр (Render).7 |

Критическое архитектурное следствие:

В Unity, если вы напишете while(true) в корутине без yield return null, вы повесите редактор. В JavaScript, если вы создадите бесконечную цепочку Микрозадач (например, рекурсивный Promise без задержки), вы повесите вкладку браузера навсегда, так как Event Loop никогда не перейдет к фазе рендеринга. Движок будет вечно разгребать очередь микрозадач.8

### 1.2. Сравнительный Анализ Моделей Исполнения

Согласно плану обучения 4, необходимо провести глубокое сравнение характеристик сред выполнения.

#### 1.2.1. Параллелизм и Поточность

* **Unity (.NET):** Поддерживает истинную многопоточность. Вы можете создать new Thread(), использовать Task.Run() для выноса тяжелых вычислений (например, поиск пути A\*) на рабочие потоки. Память (Heap) является общей (Shared Memory), что требует примитивов синхронизации (lock, Monitor, Mutex) для предотвращения состояния гонки (Race Conditions).9
* **JavaScript (V8):** Строго однопоточная модель исполнения основного кода.
  + **Web Workers:** Единственный способ достичь параллелизма. Однако, Web Workers **не имеют доступа** к DOM и не имеют общей памяти с основным потоком (за исключением SharedArrayBuffer и Atomics, которые являются продвинутой темой).
  + **Взаимодействие:** Общение происходит через сериализацию сообщений (postMessage), что накладывает оверхед на копирование данных. Это похоже на взаимодействие между микросервисами или процесс IPC, а не на легкие потоки.NET.9

**Инсайт для C# разработчика:** В JS нет Thread.Sleep(). Любая попытка заблокировать поток (например, синхронный XMLHTTPRequest или тяжелый цикл for) останавливает всё приложение. "Сон" реализуется только через асинхронное планирование (setTimeout), что эквивалентно возврату управления в Event Loop.

#### 1.2.2. Асинхронность: State Machines vs Promises

* **C# Async/Await:** Компилятор Roslyn преобразует async методы в структуру конечного автомата (IAsyncStateMachine). Когда await не завершен, метод возвращает управление, а продолжение (continuation) планируется в SynchronizationContext (в Unity это возвращение в главный поток) или в ThreadPool.10
* **JS Async/Await:** Это синтаксический сахар над Promises. Промисы — это объекты, представляющие будущее значение.
  + await в JS не освобождает поток для *других* вычислений в том же стеке. Он «разрезает» функцию: первая часть выполняется синхронно, а все, что после await, оборачивается в микрозадачу и кладется в Microtask Queue.11
  + **Важное различие:** В C# Task.Delay(1000).Wait() заблокирует поток. В JS невозможно синхронно ожидать асинхронную операцию. Вы *обязаны* использовать await или .then(), что заражает весь стек вызовов асинхронностью (Red/Blue function problem).

#### 1.2.3. Компиляция: AOT vs JIT

* **Unity (IL2CPP):** Использует Ahead-of-Time (AOT) компиляцию. C# код транслируется в C++, затем компилируется в нативный машинный код под целевую архитектуру. Это дает предсказуемую производительность и быстрый старт, но усложняет использование рефлексии и генерации кода в рантайме.
* **V8 (JIT):** Использует многоступенчатую Just-in-Time компиляцию.
  1. **Ignition (Интерпретатор):** Быстро запускает код, генерируя байт-код.
  2. **Sparkplug (Non-optimizing compiler):** Компилирует байт-код в быстрый машинный код без тяжелых оптимизаций.
  3. **TurboFan (Optimizing compiler):** На основе собранной статистики (Profiler) о типах данных генерирует высокооптимизированный код.
  + **Проблема "Прогрева" (Warm-up):** Первый вызов функции в JS всегда медленнее последующих. В Unity производительность метода стабильна (за исключением первого промаха кэша инструкций).
  + **Deoptimization (Deopt):** Если предположения компилятора о типах нарушаются (например, функция всегда получала int, а пришел string), TurboFan выбрасывает оптимизированный код и возвращается к интерпретатору. Это явление («Deopt loop») может убить производительность приложения, и аналога ему в статическом C# практически нет.12

## 2. Управление Памятью: От SOH/LOH к Generational Heap V8

Unity-разработчики часто боятся GC spikes (фризов при сборке мусора). В V8 сборщик мусора (Orinoco) работает иначе, и стратегии оптимизации памяти должны быть пересмотрены.

### 2.1. Механика Аллокации: Bump Pointer

#### 2.1.1..NET CLR Аллокатор

В.NET (и Mono) ссылочные типы выделяются в управляемой куче (Managed Heap).

* **Small Object Heap (SOH):** Использует механизм **Bump Pointer** (указатель просто сдвигается на размер объекта). Это чрезвычайно быстрая операция, сравнимая со стековой аллокацией.13
* **Large Object Heap (LOH):** Объекты > 85 КБ. Использует список свободных слотов (Free List). Аллокация здесь медленная, вызывает фрагментацию. Unity-разработчики избегают частых аллокаций в LOH.14

#### 2.1.2. V8 Heap Architecture

V8 также использует поколенческую (Generational) модель, но с нюансами динамической типизации.15

* **New Space (Nursery - Ясли):** Небольшая область (1-8 МБ), где рождаются *все* новые объекты.
  + **Аллокация:** Использует Bump Pointer, как и.NET SOH. Аллокация "бесплатна".16
  + **Scavenger (Minor GC):** Сборка мусора в яслях использует алгоритм Чейни (Semi-Space Copying). Выжившие объекты копируются в "To-Space", а "From-Space" очищается целиком.
  + **Эффективность:** Стоимость Scavenge пропорциональна количеству *живых* объектов, а не количеству мусора. В React-приложениях, где создается множество временных объектов (пропсы, замыкания, объекты событий), которые живут один кадр, Scavenge работает молниеносно. Большинство объектов умирают молодыми (Generational Hypothesis).17

Инсайт для миграции:

В Unity (особенно старых версий с Boehm GC) мы используем Object Pooling для всего, чтобы избежать аллокаций. В V8 создание миллионов мелких короткоживущих объектов — это нормальный паттерн. Не нужно писать пулы для простых JS-объектов или замыканий, так как Scavenger уберет их быстрее, чем вы будете управлять пулом. Однако, долгоживущие объекты попадают в Old Space, где сборка (Mark-Sweep-Compact) уже стоит дорого.

### 2.2. Скрытые Классы (Hidden Classes) vs MethodTable

В этом разделе мы ответим на вопросы из Промта первого дня 4: сравнение V8 Hidden Classes с C# Structs.

#### 2.2.1. Статическая типизация C#

В C# класс class Point { int x; int y; } имеет жесткую структуру в памяти.

* Каждый объект имеет заголовок (Header), указывающий на **MethodTable** (таблицу методов и метаданных типа).18
* Смещения полей (x по смещению 4 байта, y по смещению 8 байт) вычисляются при компиляции (JIT).
* Доступ к полю — это одна инструкция CPU: MOV EAX,.

#### 2.2.2. Динамическая природа JS и Hidden Classes

В JS объекты — это, по сути, словари (Hash Maps). obj.x может означать поиск строки "x" в хеш-таблице. Это медленно. Чтобы достичь скорости C++, V8 использует **Hidden Classes** (Shapes/Maps).19

1. **Transitions (Переходы):**
   * const p = new Point(); -> V8 создает Map0 (пустой).
   * this.x = x; -> V8 создает Map1 (Map0 + поле 'x') и строит переход Map0 -> Map1.
   * this.y = y; -> V8 создает Map2 (Map1 + поле 'y') и строит переход Map1 -> Map2.
2. **Offset Caching:**
   * Map2 знает, что x находится по смещению 0, а y по смещению 1.
   * Если код многократно обращается к объектам с Map2, V8 (TurboFan) генерирует машинный код, использующий прямые смещения, как в C#.

Проблема Полиморфизма (The Polymorphic Trap):

Если вы нарушаете порядок инициализации полей, вы создаете разные скрытые классы.

JavaScript

// C# Developer: "Порядок не важен"  
// JS Engine: "Это два разных типа!"  
const p1 = { x: 1, y: 2 }; // Map\_A  
const p2 = { y: 2, x: 1 }; // Map\_B (другая ветка переходов)

Функция, принимающая то p1, то p2, становится **полиморфной**.22

* **Monomorphic (1 Map):** Супер-быстро (как C# non-virtual call).
* **Polymorphic (2-4 Maps):** Быстро (сравнение Map ID и переход по switch/if).
* **Megamorphic (>4 Maps):** Медленно (деоптимизация до поиска в хеш-таблице, аналог Reflection в C# или Dictionary<string, object>).23

**Рекомендация:** Всегда инициализируйте поля в конструкторе в *одном и том же порядке*. Инициализируйте опциональные поля null, чтобы зафиксировать форму объекта (Shape) сразу.

### 2.3. Сжатие Указателей (Pointer Compression)

И V8, и современный.NET 8 используют технику **Pointer Compression**.24

* В 64-битной системе указатель занимает 8 байт. Это раздувает кеш процессора.
* Обе среды используют 4-байтные смещения относительно базового адреса кучи.
* **Ограничение V8:** Из-за использования 32-битных смещений, размер кучи V8 в браузере жестко ограничен ~4 ГБ. В Unity (C#) вы можете выделить десятки гигабайт (если не используете сжатие или имеете несколько куч). В React это означает, что вы *не можете* загрузить огромный датасет в память целиком. Виртуализация списков (Virtual List) становится обязательной не только для производительности рендеринга, но и чтобы избежать OOM (Out Of Memory) краша вкладки.26

## 3. Замыкания и Управление Состоянием: Ментальная Модель Hooks

Переход от классов к функциональным компонентам React требует понимания замыканий (Closures) на уровне их аллокации в памяти.

### 3.1. Замыкания в C# vs JS

#### 3.1.1. C# Capture Mechanics

В C# лямбда-выражение () => count++ заставляет компилятор генерировать **скрытый класс** (Display Class).28

* Локальные переменные, захваченные лямбдой, становятся полями этого класса.
* Экземпляр этого класса создается в куче (Heap Allocation).
* Если лямбда захватывает переменную цикла foreach (до C# 5.0), возникала проблема захвата *одной* переменной на все итерации. В современных версиях C# это исправлено.

#### 3.1.2. JS Closures и Lexical Environment

В JS каждая функция создает замыкание над своим лексическим окружением (Lexical Environment).30

* Функция "помнит" ссылку на Scope, в котором была создана.
* Это фундаментальный блок React Hooks. useState не магия, это использование замыкания внутри React, которое хранит состояние между вызовами функционального компонента.

### 3.2. Проблема Stale Closures (Устаревшие Замыкания)

Это самая частая ошибка Unity-разработчиков в React. В Unity, ссылаясь на поле класса в корутине, вы всегда обращаетесь к актуальному значению через this. В React функциональный компонент пересоздается при каждом рендере.

**Сценарий:**

JavaScript

function Counter() {  
 const [count, setCount] = useState(0);  
  
 useEffect(() => {  
 const timer = setInterval(() => {  
 console.log(count); // ПРОБЛЕМА: Всегда выводит 0!  
 }, 1000);  
 return () => clearInterval(timer);  
 },); // означает "запустить один раз при старте"  
}

**Анатомия проблемы:**

1. **Render 1:** count = 0. Функция useEffect выполняется. Создается замыкание для setInterval. В этом замыкании count — это переменная из скоупа Render 1 (где она равна 0).
2. **Render 2:** count становится 1. Компонент перерисовывается. Но useEffect *не перезапускается* (так как deps ``).
3. Таймер продолжает вызывать функцию из Render 1. Она "видит" старый снимок (Snapshot) переменной count.

Это концептуально похоже на передачу структуры по значению (pass-by-value) в C#: вы передали копию данных, а не ссылку на живую переменную. Чтобы исправить это, нужно либо добавить count в зависимости ([count]), что пересоздаст таймер, либо использовать useRef (аналог ссылочного типа/боксу в куче), значение внутри которого (ref.current) мутирует и доступно актуальным из любого замыкания.32

## 4. Глубокое Сравнение: Native DOM vs Unity Scene Graph

Практическое задание первого дня включает работу с Native DOM. Это необходимо, чтобы почувствовать боль императивного управления состоянием.

### 4.1. DOM как Граф Сцены

* **DOM (Document Object Model):** Это древовидная структура C++ объектов внутри браузера (движок Blink/WebCore).
* **Unity Hierarchy:** Это древовидная структура C++ объектов (GameObject/Transform) внутри движка.

### 4.2. Стоимость моста (The Bridge Cost)

В Unity вызов transform.position = newPos требует перехода из Managed (C#) в Native (C++). Это имеет накладные расходы (Marshaling).

В JS вызов div.style.width = '100px' требует перехода из V8 (JS) в Layout Engine (C++).

* **Reflow (Layout):** Изменение геометрии элемента вызывает пересчет позиций всех соседних элементов. Это аналог Canvas.ForceUpdateCanvases() или перестроения LayoutGroup в Unity UI. Это *очень* дорого.34
* **Repaint:** Заполнение пикселей. Аналог растеризации.

Почему React быстрый?

React использует Virtual DOM (Fiber Tree). Это легковесная копия дерева в памяти JS (простые объекты).

1. **Reconciliation:** При изменении состояния React перестраивает Virtual DOM.
2. **Diffing:** Сравнивает новое дерево со старым. Вычисляет минимальный набор изменений ("патч").
3. **Commit:** Применяет эти изменения к реальному DOM за один проход (Batching).

Это полный аналог **Draw Call Batching** или использования **Command Buffers** в Unity.4 Вместо тысячи отдельных вызовов API (которые вызывают пересчет макета тысячи раз), мы накапливаем изменения и применяем их пакетом.

## 5. Практический Анализ AI Промта (День 1)

В плане 4 дан специфический промт для AI-ментора. Разберем, какой технический ответ ожидается, чтобы вы могли верифицировать работу нейросети.

**Промт:** "Сравни V8 Hidden Classes с C# Structs/Vtables... Разбери работу Event Loop на примере Unity PlayerLoop... Напиши код, эмулирующий зависание..."

### 5.1. Ожидаемый ответ по Hidden Classes

AI должен объяснить, что Hidden Classes (Maps) — это механизм, превращающий динамический доступ к свойствам (dictionary lookup) в статический (offset access).

* **C# Struct:** Оффсеты жестко зашиты компилятором.
* **V8 Map:** Оффсеты вычисляются в рантайме при построении графа переходов. Если граф стабилен, JIT подставляет эти оффсеты как константы (Inline Caching).
* **Boxing аналогия:** Изменение формы объекта (добавление поля z после создания) заставляет V8 отказаться от оптимизированного кода и вернуться к медленному доступу, что похоже на *Unboxing* (извлечение из типизированной структуры в общий object) или использование Reflection.36

### 5.2. Ожидаемый ответ по Event Loop vs PlayerLoop

* **Microtasks (Promises):** Это аналог **LateUpdate** или **Coroutines**, которые выполняются *в том же кадре*, до рендеринга. Они используются для финализации данных состояния.
* **Macrotasks (setTimeout):** Это аналог логики, запланированной на **следующий кадр** Update. Они не блокируют текущий рендеринг, если разбиты на части.

### 5.3. Код эмуляции зависания

Задача: показать, как разбить длинный цикл while, чтобы не повесить браузер.

JavaScript

// Unity Style (Bad in JS)  
function heavyWork() {  
 let i = 0;  
 while(i < 1000000000) { // Блокирует поток на секунды  
 i++;  
 }  
 console.log("Done");  
}  
  
// React/JS Style (Chunking)  
function heavyWorkChunked() {  
 let i = 0;  
 function processChunk() {  
 const start = performance.now();  
 // Работаем не более 5 мс (бюджет кадра)  
 while (i < 1000000000 && performance.now() - start < 5) {  
 i++;  
 }  
 if (i < 1000000000) {  
 // Планируем продолжение в следующей макрозадаче  
 // Это дает Event Loop шанс обновить UI и обработать инпут  
 setTimeout(processChunk, 0); // Аналог yield return null  
 } else {  
 console.log("Done");  
 }  
 }  
 processChunk();  
}

Этот пример демонстрирует концепцию **Time Slicing** (нарезка времени), которая является основой **React Concurrent Mode**.37 В Unity мы делаем то же самое, распределяя нагрузку по кадрам в корутине.

## 6. Архитектурный Синтез для Проекта "Календарь"

Применяя теорию Дня 1 к задаче создания фронтенда для микросервиса Календаря:

1. **Компоненты как Префабы:**
   * Создайте структуру папок, аналогичную Prefabs: папка компонента содержит .tsx (логика+представление) и .module.css (стили/материалы).
   * DayCell — это префаб. Его пропсы (date, events) — это параметры Initialize().
2. **Виртуализация Списка (Capstone Project):**
   * Рендеринг 5000 событий в DOM убьет производительность (слишком много C++ объектов макета).
   * Используйте технику **Windowing** (виртуальное окно). Рендерите только те div, которые попадают в Viewport.
   * Это прямой аналог **Occlusion Culling** и **LOD** (Level of Detail) в Unity. Мы не тратим ресурсы GPU (в данном случае Layout Engine браузера) на то, что игрок (пользователь) не видит.
3. **Иммутабельность Данных:**
   * В Unity мы часто мутируем состояние: player.health -= 10.
   * В React/JS мы создаем копию: setPlayer(p => ({...p, health: p.health - 10})).
   * Почему? Это позволяет использовать **Reference Equality Check** (oldState === newState). Если ссылки разные — рендерим. Если одинаковые — пропускаем. Это аналог проверки Dirty Flag для оптимизации перерисовки Canvas.

## 7. Заключение

Переход на React для Unity-разработчика — это упражнение в самодисциплине. Вы теряете прямую власть над потоком исполнения и памятью, но получаете мощную декларативную модель, которая (благодаря V8) работает с впечатляющей скоростью, *если* вы соблюдаете правила статической типизации и предсказуемости форм объектов.

Ваша задача на этой неделе — научиться «думать как компилятор V8»: создавать стабильные формы объектов, избегать утечек памяти в замыканиях и уважать приоритеты Event Loop, не блокируя основной поток.

### Источники

4 План обучения «Интенсив React для Backend Разработчика неделя 10».

1 Документация Unity PlayerLoop и Execution Order.

4 Архитектура V8 Event Loop и асинхронность.

15 Внутреннее устройство V8: Hidden Classes, Memory Layout.

13 Управление памятью в.NET CLR (SOH/LOH).

34 Unity Draw Call Batching и оптимизация рендеринга.

30 Проблема Stale Closures в React.

37 React Concurrent Mode и Time Slicing.

#### Источники

1. Event function execution order - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/Manual/execution-order.html>
2. what is the difference between Update & FixedUpdate in Unity? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/34447682/what-is-the-difference-between-update-fixedupdate-in-unity>
3. Difference between Update method and FixedUpdate in Unity?, дата последнего обращения: декабря 6, 2025, <https://gamedev.stackexchange.com/questions/93850/difference-between-update-method-and-fixedupdate-in-unity>
4. Интенсив React для Backend Разработчика неделя 10
5. What Is An Event Loop IN Javascript: A Beginner Guide - F22 Labs, дата последнего обращения: декабря 6, 2025, <https://www.f22labs.com/blogs/what-is-an-event-loop-in-javascript-a-beginner-guide/>
6. Understanding the Event Loop in JavaScript: Microtasks, Macrotasks, and Asynchronous Execution - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/nishanthan-k/understanding-the-event-loop-in-javascript-microtasks-macrotasks-and-asynchronous-execution-3037>
7. Difference between microtask and macrotask within an event loop context - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/25915634/difference-between-microtask-and-macrotask-within-an-event-loop-context>
8. Why doesn't event loop have timeout for microtasks? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/72916243/why-doesnt-event-loop-have-timeout-for-microtasks>
9. Multi-Threaded v/s Event Loop Architecture | by Yamoksh Verma | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@yamokshverma/multi-threaded-v-s-event-loop-architecture-bcbab483d02b>
10. The complete asynchronous programming primer for Unity | by MY.GAMES - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/my-games-company/the-complete-asynchronous-programming-primer-for-unity-e1b00d26e96b>
11. Understanding the Event Loop, Task Queue, and Microtask Queue in JavaScript, дата последнего обращения: декабря 6, 2025, <https://rehmat-sayany.medium.com/understanding-the-event-loop-task-queue-and-microtask-queue-in-javascript-3a00b61783ec>
12. JavaScript engine fundamentals: Shapes and Inline Caches - Mathias Bynens, дата последнего обращения: декабря 6, 2025, <https://mathiasbynens.be/notes/shapes-ics>
13. Fundamentals of garbage collection - .NET | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals>
14. Why Large Object Heap and why do we care? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/8951836/why-large-object-heap-and-why-do-we-care>
15. Visualizing memory management in V8 Engine (JavaScript, NodeJS, Deno, WebAssembly) | Technorage - Deepu K Sasidharan, дата последнего обращения: декабря 6, 2025, <https://deepu.tech/memory-management-in-v8/>
16. Garbage collection in V8, an illustrated guide | by Irina Shestak - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@_lrlna/garbage-collection-in-v8-an-illustrated-guide-d24a952ee3b8>
17. Memory Management in V8, garbage collection and improvements - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/jennieji/memory-management-in-v8-garbage-collection-and-improvements-18e6>
18. Performance Improvements in .NET 8 - Microsoft Developer Blogs, дата последнего обращения: декабря 6, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/>
19. Hidden Classes: The JavaScript performance secret that changed everything, дата последнего обращения: декабря 6, 2025, <https://dev.to/maxprilutskiy/hidden-classes-the-javascript-performance-secret-that-changed-everything-3p6c>
20. Maps (Hidden Classes) in V8 - V8 JavaScript engine, дата последнего обращения: декабря 6, 2025, <https://v8.dev/docs/hidden-classes>
21. JavaScript Object Internals-From Hidden Classes to Hash Maps | by Berkay ÇIRAK, дата последнего обращения: декабря 6, 2025, <https://medium.com/@berkaycrk/javascript-object-internals-from-hidden-classes-to-hash-maps-7d96e07feb1b>
22. Inline caching - Wikipedia, дата последнего обращения: декабря 6, 2025, <https://en.wikipedia.org/wiki/Inline_caching>
23. The Hidden JavaScript Engine Optimization That Makes Your Code 40% Slower, дата последнего обращения: декабря 6, 2025, <https://javascript.plainenglish.io/the-hidden-javascript-engine-optimization-that-makes-your-code-40-slower-26994e837830>
24. When V8 compacts or reallocates memory, does it change all the references?, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/73750682/when-v8-compacts-or-reallocates-memory-does-it-change-all-the-references>
25. Pointer Compression in V8 - V8 JavaScript engine, дата последнего обращения: декабря 6, 2025, <https://v8.dev/blog/pointer-compression>
26. How does the compressed pointer implementation in V8 differ from JVM's compressed Oops? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/59276104/how-does-the-compressed-pointer-implementation-in-v8-differ-from-jvms-compresse>
27. Electron and the V8 Memory Cage, дата последнего обращения: декабря 6, 2025, <https://electronjs.org/blog/v8-memory-cage>
28. Captured Variables : Understanding lambda expression and closures C# - Sushant Pant, дата последнего обращения: декабря 6, 2025, <https://sushantpant.com.np/captured-variables-lambda-expression-and-closures>
29. Capturing variables in Lambda Expressions : r/ProgrammingLanguages - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/ProgrammingLanguages/comments/18kg86w/capturing_variables_in_lambda_expressions/>
30. React: stale closure - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/animusna/react-stale-closure-81a>
31. Why Do JavaScript Closures Capture Variables by Reference, Not Value? - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@conboys111/why-do-javascript-closures-capture-variables-by-reference-not-value-6fd481280a95>
32. Be Aware of Stale Closures when Using React Hooks - Dmitri Pavlutin, дата последнего обращения: декабря 6, 2025, <https://dmitripavlutin.com/react-hooks-stale-closures/>
33. Why would a value get stale using closure in React? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/72075881/why-would-a-value-get-stale-using-closure-in-react>
34. Draw call batching - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/2021.3/Documentation/Manual/DrawCallBatching.html>
35. Static Batching, Explained! Learn about this powerful Draw Call Optimization technique. : r/Unity3D - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/Unity3D/comments/13pmik1/static_batching_explained_learn_about_this/>
36. Hidden Costs of Boxing in C#: How to Detect and Avoid Them - elmah.io Blog, дата последнего обращения: декабря 6, 2025, <https://blog.elmah.io/hidden-costs-of-boxing-in-c-how-to-detect-and-avoid-them/>
37. React Concurrent Mode: Optimizing React Performance - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/usman_awan/react-concurrent-mode-optimizing-react-performance-2mln>
38. Customizing the Player loop - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/Manual/player-loop-customizing.html>
39. Understanding the JavaScript Event Loop: How Asynchronous Magic Happens | by Sade Shure Garmamo | Towards Dev - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/towardsdev/understanding-the-javascript-event-loop-how-asynchronous-magic-happens-921c40454f1d>