# Архитектурный синтез: DOM браузера и Scene Graph Unity — Сравнительный системный анализ (День 3)

## 1. Введение: Инверсия контроля и смена парадигмы исполнения

Переход от разработки на движке Unity к созданию архитектуры веб-приложений (Enterprise Frontend) требует фундаментального пересмотра ментальной модели управления ресурсами и потоком исполнения. В экосистеме Unity инженер выступает в роли верховного оркестратора кадра. Посредством детерминированного игрового цикла (Game Loop), строго регламентированных фаз Update(), FixedUpdate() и LateUpdate(), разработчик явно диктует движку момент расчета физики, симуляции искусственного интеллекта и отправки команд на отрисовку (Draw Calls). Движок служит циклу, заданному программистом.

В среде выполнения браузера, и в частности в объектной модели документа (DOM) и конвейере рендеринга (Rendering Pipeline), эта иерархия инвертируется. Браузер является оркестратором, а код разработчика — лишь гостевым плагином, реагирующим на стимулы внешней среды. Браузер оптимизирует выполнение не ради детерминированной симуляции каждого кадра, а ради минимизации задержки (latency) и плавности потока контента в ответ на асинхронные события. Для Senior Unity-разработчика это требует перехода от ментальной модели «polling» (опрос состояния каждый кадр) к «reactive» (реакция на флаги инвалидации).

Данный отчет представляет собой теоретическое ядро третьего дня программы переподготовки. Его цель — деконструировать внутреннее устройство движка рендеринга Blink (Chrome) и среды исполнения JavaScript V8 через призму паттернов высокопроизводительных вычислений (HPC), знакомых архитекторам.NET/C++. Мы проанализируем DOM не просто как структуру разметки, а как retained-mode граф сцены (Scene Graph) с уникальными характеристиками распределения памяти, стоимости обхода и критическими алгоритмами распространения «грязных битов» (dirty bits), управляющими инвалидацией макета.

## 2. Объектная модель: Сдвиг парадигмы от Scene Graph к DOM Tree

### 2.1. Структурный изоморфизм и расхождение в памяти

На высоком уровне абстракции граф сцены Unity (Unity Scene Graph) и DOM-дерево браузера изоморфны. Обе структуры представляют собой иерархические направленные ациклические графы (DAG), описывающие сущности, подлежащие рендерингу.

В Unity иерархия строится на экземплярах GameObject, связанных через компоненты Transform. Отношения «родитель-потомок» обеспечивают распространение матриц локальных трансформаций в мировые координаты (Local-to-World Matrices). В браузере иерархия состоит из объектов Node (в частности, HTMLElement), связанных через API DOM (parentNode, childNodes). Отношения «родитель-потомок» здесь распространяют не только геометрические трансформации, но и вычисленные стили (Computed Styles) и ограничения макета (Layout Constraints).

Однако паттерны распределения памяти и доступа к данным в этих системах радикально расходятся. В Unity компонент Transform представляет собой высокооптимизированную структуру C++ (с управляемой оберткой в C#), спроектированную для плотного размещения в памяти (особенно с развитием DOTS и ECS, разделяющих данные и поведение). Движок Unity стремится к тому, чтобы данные трансформаций были когерентны в кэше процессора (cache coherency), минимизируя промахи кэша (cache misses) при расчете мировых позиций тысяч объектов за кадр.1

В архитектуре браузера DOM-узлы являются «тяжеловесными» объектами. В движке Blink (основа Chromium) каждый узел — это сложный C++ объект (например, blink::Node), который несет на себе значительно больше состояния, чем GameObject в Unity. Один элемент div инкапсулирует:

* **Состояние стилей (Style State):** Вычисленные CSS-значения, инлайн-стили, списки классов.
* **Состояние макета (Layout State):** Клиентские прямоугольники (bounding client rects), смещения прокрутки (scroll offsets), геометрию клиента.
* **Состояние событий (Event State):** Список слушателей событий и пути распространения всплытия/перехвата.
* **Дерево доступности (Accessibility Tree):** Маппинги на API доступности операционной системы (A11y).

**Архитектурный инсайт:** Трактовка DOM-узла как легковесной сущности — аналогичной частице в Unity Particle System или простому спрайту — является катастрофической ошибкой оптимизации. Создание 10 000 DOM-узлов вычислительно аналогично инстанцированию 10 000 MonoBehaviour с компонентами CanvasRenderer, GraphicRaycaster и LayoutElement, а не 10 000 сущностей ECS. Накладные расходы памяти включают не только аллокацию в куче V8 (Managed Heap) для JavaScript-обертки, но и соответствующие структуры C++ в движке рендеринга (Native Heap), а также потенциальные слои (Layers), создаваемые в композиторе.3

### 2.2. Механика обхода и когерентность кэша

В Unity итерация по дочерним элементам Transform часто оптимизируется внутренней компоновкой памяти движка. Движок старается держать данные трансформаций в непрерывном блоке памяти, чтобы обеспечить векторизацию вычислений (SIMD) и минимизировать циклы ожидания CPU во время физических и рендер-проходов.

DOM, напротив, представляет собой структуру, требующую постоянного «перепрыгивания» по указателям (pointer-chasing). Обход DOM-дерева (например, использование element.firstChild, element.nextSibling) подразумевает переходы между разрозненными участками кучи. Движок V8 и Blink используют изощренные оптимизации, такие как скрытые классы (Hidden Classes или Shapes) и инлайн-кэширование (Inline Caching), но фундаментальная природа DOM остается динамической и фрагментированной.3 Она не гарантирует пространственную локальность ссылок, свойственную массиву структур (Array of Structs) или структуре массивов (Struct of Arrays) в Unity DOTS.

**Импликация для архитектора:** Алгоритмы, полагающиеся на обход DOM-дерева в каждом кадре (например, поиск конкретного узла путем перебора childNodes внутри цикла requestAnimationFrame), будут вызывать значительно больше промахов кэша CPU по сравнению с итерацией по предварительно кэшированному массиву JavaScript Array или типизированному массиву Float32Array. Это подтверждает необходимость применения паттерна «Кэшированная ссылка» (Cached Reference): подобно тому, как GetComponent<T>() вызывается в Awake() и кэшируется в поле класса, DOM-запросы (querySelector, getElementById) должны выполняться единожды, а ссылки сохраняться в переменных замыкания или полях класса.6

### 2.3. Стоимость абстракции: Managed vs. Unmanaged

Unity-разработчик привык к границе между управляемым кодом (Managed Code, C#) и нативным кодом (Native Code, C++). Пересечение этого моста (маршалинг) влечет за собой накладные расходы. В браузере существует аналогичный барьер между JavaScript Engine (V8) и Rendering Engine (Blink/C++).

Доступ к свойству, такому как element.id, не является простым чтением поля JavaScript-объекта. Это триггер вызова связующего кода (bindings) в движок C++ для извлечения данных из внутренней репрезентации Node. Хотя современные JIT-компиляторы (TurboFan в V8) агрессивно оптимизируют эти аксессоры, они никогда не бывают «бесплатными». Запись в DOM (element.style.width =...) обходится еще дороже, так как она помечает C++ объект как «грязный» (dirty), запуская цепочки инвалидации, которые мы подробно рассмотрим в разделе 3.3

| **Характеристика** | **Unity Scene Graph** | **Browser DOM Tree** |
| --- | --- | --- |
| **Базовая единица** | GameObject + Transform | Node / HTMLElement |
| **Модель памяти** | Оптимизирована под contiguity (особенно DOTS) | Pointer-based heap allocation (V8 + C++) |
| **Стоимость создания** | Средняя (Instantiate overhead) | Высокая (JS Wrapper + Blink C++ Object) |
| **Доступ к данным** | Прямой (struct fields) или Managed wrapper | Binding calls (V8 <-> Blink) |
| **Иерархия** | Строгая (Transform matrix propagation) | Сложная (Style inheritance, Layout flow) |

## 3. Конвейер рендеринга: Кадр из жизни браузера

Для овладения производительностью браузера необходимо спроецировать пиксельный конвейер (Pixel Pipeline) веб-страницы непосредственно на цикл рендеринга Unity. Жизненный цикл кадра в браузере представляет собой последовательное выполнение четко определенных фаз, строго упорядоченных для формирования растрового изображения. Нарушение этого порядка ведет к феноменам, эквивалентным потере производительности при нарушении батчинга в Unity.

### 3.1. Сравнительный анализ: Unity Loop vs. Browser Pipeline

Чтобы понять, где теряется производительность, сопоставим фазы отрисовки кадра в двух системах.

| **Фаза Unity** | **Фаза Браузера** | **Ответственность системы** | **Архитектурный аналог** |
| --- | --- | --- | --- |
| **Asset Loading** | **Parsing** | Построение графа сцены из исходных данных. | Resources.Load / Парсинг HTML в DOM и CSS в CSSOM.8 |
| **Logic Update** | **JavaScript** | Выполнение логики / изменение состояния приложения. | Update() / Задачи в Event Loop (Task Queues).9 |
| **N/A** | **Style Recalc** | Резолюция того, какие визуальные правила применяются к узлу. | Прямого аналога нет (в Unity материалы назначаются явно). |
| **Physics/Layout** | **Layout (Reflow)** | Расчет геометрии (x, y, width, height). | Система Layout в Canvas / Решение констрейнтов физическим движком.10 |
| **Draw Calls** | **Paint** | Заполнение пикселей (растеризация) в текстуры. | Camera.Render / Исполнение шейдеров фрагментов.11 |
| **Post-Process** | **Composite** | Сборка слоев на экране. | Image Effects / GPU buffer swapping / Sorting Layers.12 |

### 3.2. Детальная декомпозиция этапов

#### 3.2.1. Style Calculation (CSSOM — Объектная модель стилей)

Прежде чем может произойти расчет макета, браузер должен ответить на вопрос: «Как выглядит этот узел?». Для этого он комбинирует дерево DOM с деревом CSSOM. Это процесс сопоставления (matching), при котором движок обходит дерево и применяет CSS-селекторы.

* **Сложность:** Стоимость пропорциональна количеству DOM-элементов, умноженному на сложность CSS-селекторов.
* **Аналогия с Unity:** Представьте, если бы каждый кадр Unity был вынужден заново оценивать, какой Material использовать для каждого GameObject, проверяя его имя и тег против списка из 10 000 строковых правил. Это объясняет, почему сложные селекторы (например, .box:nth-last-child(odd).title) могут быть чрезмерно дорогими при динамических обновлениях.8 В Unity материалы обычно присваиваются статически или кэшируются, в то время как в браузере это динамический процесс разрешения правил (cascade resolution).

#### 3.2.2. Layout (Reflow): Решатель блочной модели

Это прямой эквивалент системы компоновки UI в Unity (VerticalLayoutGroup, HorizontalLayoutGroup, GridSystem). Браузер вычисляет точные экранные координаты и размеры для каждого видимого элемента.

* **Решатель ограничений (Constraint Solver):** В отличие от системы Transform в Unity, которая в основном строго иерархична (родитель движется -> потомок движется), система компоновки веба основана на ограничениях потока. Размер потомка может влиять на размер родителя (shrink-to-fit), а размер одного сиблинга может смещать последующие сиблинги (block flow).
* **Поток вычислений:** Движок макета (в Chrome это **LayoutNG**) обычно выполняет рекурсивный проход. Он спускается вниз по дереву, чтобы вычислить доступное пространство, и поднимается вверх, чтобы сообщить о требуемом размере.14
* **Производительность:** Это самая дорогая операция в конвейере. Она ограничена скоростью CPU и блокирует основной поток (Main Thread). Сложность макета может варьироваться от O(N) до O(N^2) в зависимости от глубины вложенности и использования Flexbox/Grid.15

#### 3.2.3. Paint: Растеризация

На этом этапе браузер заполняет пиксели. Это включает в себя отрисовку текста, теней, границ, фоновых изображений и векторной графики.

* **Растеризатор:** Современные браузеры часто используют многопоточный растеризатор (например, библиотека Skia в Chrome), который может выполняться на отдельных потоках (Raster Threads).
* **Инвалидация:** Браузер пытается перерисовывать только «грязные прямоугольники» (dirty rects) — регионы, которые изменились. Однако изменение макета (Layout) часто инвалидирует большие участки экрана, принуждая к перерисовке значительной части страницы.16

#### 3.2.4. Composite: Сборка на GPU

Это этап, где браузер демонстрирует свою мощь. Он разделяет страницу на «Слои» (Layers), которые по сути являются текстурами на GPU. Поток композитора (Compositor Thread), отдельный от основного потока JS, берет эти слои и отрисовывает их на экранный квад.12

* **Аналогия с Unity:** Это фактически рендеринг Render Textures на экран. Если вы меняете трансформацию слоя (CSS transform или opacity), GPU может обработать это без повторной растеризации текстуры (без Paint). Это ключ к анимациям на 60 FPS, так как операции выполняются на GPU, минуя медленный CPU-Layout.

## 4. Layout (Reflow) — Движок геометрических вычислений

Самая критическая концепция для разработчика третьего дня — это **Reflow** (или Layout). Понимание этого механизма — это разница между приложением, работающим на 60 FPS, и приложением, лагающим на 10 FPS. Это аналог Physics.Simulate() или перестроения мешей Canvas в Unity, но с более высокой чувствительностью к порядку операций.

### 4.1. Система «Грязных битов»: Ленивая оценка

Браузеры, как и Unity, используют паттерн «Dirty Bit» (флаг загрязнения) для избежания избыточных вычислений.

* **Маркировка загрязнения (Marking Dirty):** Когда скрипт модифицирует свойство стиля, влияющее на геометрию (например, element.style.width = '50%'), движок не вычисляет новую ширину мгновенно. Вместо этого он помечает узел как «грязный» и устанавливает флаги на всех его предках вплоть до корня документа (Document Root).19
* **Ленивая оценка (Lazy Evaluation):** Браузер предпочитает быть ленивым. Он стремится отложить пересчет макета до конца текущей задачи выполнения JavaScript, чтобы выполнить расчет макета одним пакетом (batch). Это аналогично тому, как Unity ждет конца кадра для рендеринга, или как Canvas откладывает перестроение геометрии до момента перед отрисовкой.20

### 4.2. Смертельная ловушка: Принудительный синхронный макет (Layout Thrashing)

Это наиболее распространенный убийца производительности для разработчиков, переходящих с C#. В Unity чтение transform.position является дешевым, потому что трансформ обычно обновляется, или значение просто берется из кэшированной структуры. В DOM чтение геометрического свойства может вызвать катастрофу.

**Сценарий:**

1. **Запись (Write):** Вы меняете ширину элемента (div.style.width = '100px').
   * *Состояние:* DOM теперь «грязный». Дерево макета (Layout Tree) невалидно.
2. **Чтение (Read):** Вы немедленно запрашиваете высоту этого элемента или *любого другого элемента* (div.offsetHeight).
   * *Угроза:* Чтобы вернуть корректную высоту, браузер *обязан* применить изменение ширины и запустить алгоритм Layout немедленно. Он не может ждать конца кадра. Он выполняет **принудительный синхронный макет** (Forced Synchronous Layout).

Цикл Layout Thrashing:

Если вы делаете это внутри цикла, вы создаете Layout Thrashing.

JavaScript

// АНТИ-ПАТТЕРН: Layout Thrashing  
// Аналогия: Вызов Physics.Simulate() внутри цикла for для каждого объекта.  
for (let i = 0; i < items.length; i++) {  
 let item = items[i];  
 // ЧТЕНИЕ: Браузер вынужден запустить Layout  
 let width = container.offsetWidth;   
 // ЗАПИСЬ: Браузер инвалидирует Layout  
 item.style.width = (width / 2) + 'px';   
}

В приведенном выше коде для каждой итерации браузер должен: Рассчитать макет -> Инвалидировать макет -> Рассчитать макет -> Инвалидировать макет. Если items.length равно 100, вы выполнили 100 полных проходов макета за один кадр. На практике это имеет сложность O(N^2) из-за необходимости обхода дерева при каждом расчете макета.21

Решение: Пакетная обработка (Batching)

Разделение фаз чтения и записи позволяет браузеру пакетировать инвалидации и выполнять один проход макета.

JavaScript

// ПАТТЕРН: Batching (FastDOM)  
// Фаза чтения (Read phase)  
let width = container.offsetWidth; // Один расчет макета (если он был грязным)  
  
// Фаза записи (Write phase)  
for (let i = 0; i < items.length; i++) {  
 // Только инвалидация; пересчет не триггерится здесь  
 item.style.width = (width / 2) + 'px';   
}  
// Браузер выполняет ОДИН расчет макета в конце кадра.

### 4.3. Аналогия с Unity: Canvas.ForceUpdateCanvases()

Ближайшим эквивалентом в Unity является метод Canvas.ForceUpdateCanvases(). Обычно Unity UI перестраивает макет в конце кадра. Вызов этого метода принуждает к немедленному перестроению. Представьте, что вы вызываете ForceUpdateCanvases() внутри цикла foreach, итерирующего по 100 элементам UI. Деградация производительности будет идентична DOM Layout Thrashing. В браузере, однако, этот «вызов» скрыт за невинным доступом к свойствам вроде offsetHeight или clientWidth.24

## 5. Paint и Composite — Стратегии растеризации

Как только макет (Layout) разрешен, браузер должен превратить векторные данные в пиксели. Этот этап вводит концепции **Repaint** (Перерисовка) и **Compositing Layers** (Слои композиции).

### 5.1. Инвалидация отрисовки и грязные прямоугольники

Подобно устаревшему OnGUI в Unity или системам грязных прямоугольников в 2D-движках, браузер отслеживает, какие пиксели нуждаются в обновлении. Если вы меняете цвет текста кнопки, перерисовке подлежит только регион этой кнопки. Однако, если вы меняете цвет фона у body, инвалидируется весь экран.

**Reflow всегда триггерит Repaint.** Если элемент перемещается (Reflow), он должен быть нарисован в новой позиции (Repaint), а фон, который он освободил, должен быть перерисован (Repaint).17

### 5.2. Композитинг: Быстрый путь GPU

Самая мощная оптимизация в браузере — это **Composite**. Браузер может повысить (promote) определенные элементы до собственных «Слоев композиции» (Compositing Layers). Слой — это, по сути, текстура на GPU. Как только элемент оказывается на своем слое, браузер может изменять его **Transform** (перемещение, масштаб, вращение) и **Opacity** (прозрачность), не запуская Layout или Paint. Он просто модифицирует вершины квада или альфа-канал на этапе композитинга.

Свойство CSS will-change:

Это свойство сообщает браузеру: «Я собираюсь анимировать трансформ этого элемента». Браузер реагирует, загружая этот элемент на GPU как отдельную текстуру (слой).

* **Аналогия с Unity:** Это похоже на перенос динамического UI-элемента на собственный Canvas (Sub-Canvas). В Unity UI Canvas является единицей батчинга геометрии. Разделение Canvas изолирует перестроение геометрии. Аналогично, создание слоя в DOM изолирует элемент от цикла отрисовки остальной страницы.26

**Предупреждение о производительности:** Создание слишком большого количества слоев (например, \* { will-change: transform }) потребляет чрезмерное количество видеопамяти (VRAM), аналогично наличию тысяч уникальных материалов или Draw Calls в Unity. Это перегружает шину данных и композитор, вызывая падение FPS.

## 6. Паттерны высокопроизводительной манипуляции DOM

Понимание конвейера диктует паттерны для эффективной манипуляции сценой (DOM). Мы рассмотрим методы, эквивалентные оптимизациям Unity.

### 6.1. DocumentFragment: Внеэкранный буфер (Off-Screen Buffer)

При добавлении множества элементов в DOM (например, спавн списка предметов инвентаря), добавление их по одному к живому документу вызывает множественные Reflow/Repaint (или, по крайней мере, заставляет браузер тратить ресурсы на управление грязным состоянием).

DocumentFragment — это легковесный объект, подобный DOM-узлу, который не имеет родителя. Он существует только в памяти, будучи отсоединенным от живого дерева DOM.

* **Механизм:** Вы добавляете 100 элементов во Фрагмент. Reflow не происходит, так как Фрагмент не рендерится. Затем вы добавляете Фрагмент в живой DOM. Браузер перемещает *дочерние элементы* Фрагмента в DOM за одну операцию, вызывая только **один** Reflow.
* **Аналогия с Unity:** Это концептуально схоже с инстанцированием иерархии префабов, когда корневой родитель находится в состоянии inactive (SetActive(false)), и последующим включением корня только после полной сборки иерархии. Это предотвращает запуск циклов Awake/OnEnable/Update для частично инициализированных объектов и лишние перестроения Canvas.28

### 6.2. Виртуализация (Windowing): Объектный пул веба

Инженер Enterprise-бэкенда часто работает со списками из 10 000+ записей (логи, транзакции, пользователи). Рендеринг 10 000 DOM-узлов смертелен для производительности (раздувание памяти, огромное дерево Layout).

* **Проблема:** Размер дерева Layout растет, делая обход дорогим. V8 тратит мегабайты на хранение объектов.
* Решение: Виртуальный скроллинг (Virtual Scroll).  
  Это Object Pooling, примененный к DOM.
  1. **Пул:** Мы создаем фиксированный массив div-элементов (например, 20 штук для списка, отображающего 10).
  2. **Скролл:** Вместо движения камеры (как в Unity), мы перемещаем данные.
  3. **Переработка (Recycling):** Когда пользователь скроллит вниз, верхний div выходит за пределы видимости. Мы проверяем его позицию, концептуально «деактивируем» его, перемещаем в низ списка, «реактивируем» с новыми данными (текст/картинка) и корректируем его transform для правильного позиционирования.
  4. **Контейнер:** Используется «фантомный» контейнер с высотой totalItems \* itemHeight для сохранения размера ползунка прокрутки.

**Идеальный изоморфизм:**

* **Unity:** scrollRect.content.anchoredPosition меняется -> Пересчет видимых индексов -> Перепозиционирование объектов из пула.
* **JS:** scrollTop меняется -> Пересчет видимых индексов -> Перепозиционирование DOM-узлов (recycle).31

## 7. Архитектура событий: Всплытие и Делегирование

Событийная модель браузера существенно отличается от C# delegates или Unity Events (UnityEvent).

### 7.1. Модель распространения событий (Propagation Model)

События в DOM распространяются в три фазы:

1. **Захват (Capturing):** От корня Window вниз к Цели (Target).
2. **Цель (Target):** На самом элементе.
3. **Всплытие (Bubbling):** От Цели обратно вверх к Window.

Этот двунаправленный поток сложнее стандартной системы сообщений Unity (SendMessage распространяется вниз, обычные C# события — это прямой вызов). Однако система UI Unity (EventSystem) использует рейкастинг (Raycasting), который концептуально близок к Всплытию — события передаются вверх по иерархии GameObjects, пока компонент-обработчик (например, IPointerClickHandler) не перехватит их.

### 7.2. Делегирование событий: Оптимизация памяти

Если у вас есть список из 1 000 кнопок «Удалить», наивным подходом будет присоединение addEventListener('click',...) к каждой кнопке.

* **Стоимость памяти:** 1 000 объектов EventListener, 1 000 замыканий (closures) и накладные расходы V8 на связывание.
* **Стоимость инициализации:** Время, затраченное на создание функций при загрузке.

Паттерн: Делегирование (Event Delegation)

Прикрепите один слушатель событий к родительскому контейнеру (ul). Когда происходит клик на li (потомок), событие всплывает к ul. Обработчик проверяет event.target, чтобы определить, какой именно потомок был кликнут.

* **Экономия памяти:** Снижение накладных расходов с O(N) до O(1).
* **Аналогия с Unity:** Вместо того чтобы вешать скрипт с OnMouseDown на каждую плитку в сетке карты (Grid Map), вы вешаете один скрипт на GridManager, который пускает луч (Raycast) или проверяет координаты события, чтобы определить, какая плитка была нажата. Это централизует логику и радикально снижает количество активных «подписчиков» в системе, разгружая Event System.34

## 8. Shadow DOM и Инкапсуляция

Shadow DOM привносит в веб концепцию изоляции компонентов, которая является стандартом в Unity (Prefab), но исторически отсутствовала в HTML/CSS.

### 8.1. Изоляция стилей

В стандартном CSS стили глобальны. Правило .button { color: red; } повлияет на *каждую* кнопку на странице. Это вызывает «протечку стилей» (style bleeding) и кошмар поддержки — проблемы, редко встречающиеся в Unity, где внутренняя структура Префаба изолирована.

**Shadow DOM** позволяет компоненту (Custom Element) прикрепить скрытое дерево DOM (Shadow Tree).

* **Механизм:** «Теневой корень» (Shadow Root) присоединяется к хост-элементу. Элементы внутри теневого корня изолированы от CSS основного документа. Глобальные стили не проникают внутрь, а теневые стили не выходят наружу.
* **Внутренняя реализация:** В Blink это создает границу области видимости (scope boundary) на этапе расчета стилей. Каскад CSS блокируется на границе тени.
* **Аналогия с Unity:** Это напоминает строгую инкапсуляцию в **UI Toolkit** (VisualElements) или вложенных префабах (Nested Prefabs), где внутренние модификации защищены от контекста сцены. Это обеспечивает компонентную модель на уровне движка, позволяя создавать переиспользуемые виджеты, не опасаясь конфликтов имен.37

## 9. Практический синтез: Мышление Fullstack-архитектора

Для Senior Unity-разработчика освоение теории третьего дня заключается не в заучивании методов DOM API, а в интериоризации **модели стоимости** (cost model) браузера.

* **DOM — это Render Tree:** Относитесь к нему с той же осторожностью, что и к графу сцены. Минимизируйте глубину и количество узлов.
* **Layout — это Physics:** Относитесь к Reflow с тем же страхом, что и к Physics.Simulate(). Никогда не триггерите его внутри циклов.
* **Batching — это закон:** Будь то DocumentFragment (батчинг инстанцирования) или переключение классов (батчинг изменений стилей), всегда группируйте операции. Читайте данные, затем пишите данные. Никогда не чередуйте.
* **Virtualization — это Pooling:** Никогда не рендерите данные, которые не находятся на экране. Используйте пулы DOM-узлов для списков.

Взглянув на браузер через призму архитектора игрового движка — идентифицировав цикл рендеринга, решатель макета и узкие места памяти — разработчик перерастает уровень «написания скриптов» и начинает проектировать высокопроизводительные, масштабируемые приложения уровня Enterprise.

## 10. Детальный план исследований и практических заданий (День 3)

### Темы теории (Deep Dive)

1. **Архитектура DOM (DOM Architecture):** Внутреннее устройство Node, Element, DocumentFragment. C++ подложка объектов V8 (Blink C++ bindings). Анализ накладных расходов памяти (Memory Overhead) на каждый узел.
2. **Критический путь рендеринга (The Critical Rendering Path):** Детальный разбор этапов Parse -> Style -> Layout -> Paint -> Composite. Сравнение с Render Loop Unity.
3. **Reflow & Repaint:** Триггеры (CSS свойства), стоимость вычислений, стратегии минимизации. Понятие Layout Boundary.
4. **Событийный цикл и Рендеринг:** Как requestAnimationFrame встраивается в конвейер (сравнение с Update). Различия между Task, Microtask и Rendering Task.
5. **Виртуализация (Virtualization):** Математика «оконного» рендеринга (Windowing). Алгоритмы ресайклинга DOM-узлов.

### Промт для глубокого исследования (Day 3 Theory Generation)

**Role:** You are a Senior Graphics Engineer specializing in Browser Engine Architecture (Blink/Gecko) and Unity Engine Internals.

**Task:** Generate a comprehensive technical "RFC" (Request for Comments) document comparing the **Unity Canvas Layout System** with the **Browser Layout Engine (Reflow)**.

**Key Points to Cover:**

1. **Algorithmic Complexity:** Compare the cost of Canvas.ForceUpdateCanvases() with accessing offsetWidth in a loop. Explain the O(N) vs O(N^2) implications in deep hierarchies. Provide benchmark methodology.
2. **Dirty Flag Propagation:** Explain how Blink marks nodes as dirty (LayoutInvalidation) using "Invalidation Sets" and how this compares to Unity's OnRectTransformDimensionsChange and hierarchy dirtiness.
3. **Memory Overhead:** Quantify (approximately) the memory footprint of a DOM Node (C++ backing + V8 wrapper + Event Listeners) vs. a Unity GameObject + RectTransform. Why is the DOM historically "heavier"?
4. **Virtualization Strategy:** Provide a pseudo-code implementation of a **Virtual Scroller** in Vanilla JS, explicitly commenting on where "Object Pooling" concepts apply (e.g., "Recycling a DOM node is like returning a GameObject to the pool").
5. **Compositing & Threading:** Explain how will-change: transform promotes an element to a GPU layer and how the **Compositor Thread** handles scrolling independently of the Main Thread. Draw an analogy to **Sorting Layers** or **Sub-Canvases** in Unity to isolate repaint costs.

### Практические задания (Laboratories)

#### Лабораторная работа 1: "The Thrashing Room" (Комната Трэшинга)

* **Сценарий:** Создать скрипт, который обновляет ширину 1 000 элементов div на основе ширины их соседей.
* **Задача:** Реализовать «наивный» подход (чтение и запись в одном цикле) и замерить время выполнения через performance.now(). Увидеть в Chrome DevTools Performance вкладке «пилу» из Layout событий.
* **Оптимизация:** Переписать код, используя паттерн "Batching" (сначала прочитать все ширины в массив, затем записать все новые ширины).
* **Цель:** Добиться ускорения в 10-100 раз. Понять цену синхронного рефлоу.
* **Аналог Unity:** Сравнить с вызовом Physics.SyncTransforms() или Canvas.ForceUpdateCanvases() внутри цикла.

#### Лабораторная работа 2: "The Virtual List" (Виртуальный Список)

* **Сценарий:** Дан JSON-массив из 100 000 записей лога.
* **Задача:** Реализовать скроллируемый список, который рендерит в DOM только видимые 20 элементов (+ буфер).
* **Требование:** Использовать абсолютное позиционирование (transform: translateY) и **ресайклинг** (переиспользование) DOM-узлов при скролле. Не удалять и не создавать узлы заново.
* **Метрика:** Стабильные 60 FPS при скролле и низкое потребление памяти (heap snapshot).
* **Аналог Unity:** Реализация LoopScrollRect с пулом объектов.

#### Лабораторная работа 3: "Event Radar" (Радар Событий)

* **Сценарий:** Сетка из 100x100 ячеек (10 000 элементов).
* **Задача:** Реализовать кликабельность ячеек.
  + *Вариант А (Плохой):* Добавить onclick на каждую ячейку. Замерить использование памяти.
  + *Вариант Б (Хороший):* Добавить **один** слушатель на контейнер. Использовать event.target и dataset для определения кликнутой ячейки.
* **Визуализация:** Подсвечивать кликнутую ячейку.
* **Цель:** Увидеть разницу в потреблении памяти (Listener Count) в Chrome Task Manager.

Этим завершается теоретическая база третьего дня. Инженер должен выйти из этого этапа с пониманием того, что каждый вызов API браузера имеет свою цену в тактах CPU и байтах памяти, и что высокопроизводительный фронтенд требует такой же дисциплины, как и разработка AAA-игр.

#### Источники

1. Dirty Flag - Game Programming Patterns, дата последнего обращения: декабря 6, 2025, <https://gameprogrammingpatterns.com/dirty-flag.html>
2. Performance Comparison: Moving 100,000 Nodes in Game Engines : r/godot - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/godot/comments/16xb34y/performance_comparison_moving_100000_nodes_in/>
3. Optimizing V8 memory consumption, дата последнего обращения: декабря 6, 2025, <https://v8.dev/blog/optimizing-v8-memory>
4. In a JavaScript engine like V8, does an object consume more memory than a function that returns it? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/76168588/in-a-javascript-engine-like-v8-does-an-object-consume-more-memory-than-a-functi>
5. Общий неделя 9
6. I heard that you should use GetComponent as little as possible so would this be a correct/well organized interpretation of that? : r/unity - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/unity/comments/uvo90h/i_heard_that_you_should_use_getcomponent_as/>
7. Which is better practice in Unity? GetComponent<>() or making the variable public?, дата последнего обращения: декабря 6, 2025, <https://gamedev.stackexchange.com/questions/181799/which-is-better-practice-in-unity-getcomponent-or-making-the-variable-publi>
8. Part 2 : How Browsers Render Pages - A Frontend Developer's Guide to the Rendering Pipeline - Aabir Hassan, дата последнего обращения: декабря 6, 2025, <https://aabir-hassan.medium.com/part-2-how-browsers-render-pages-a-frontend-developers-guide-to-the-rendering-pipeline-29599fd97637>
9. JavaScript execution model - MDN Web Docs, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Execution_model>
10. Inside the Browser Rendering Pipeline - AG, дата последнего обращения: декабря 6, 2025, <https://aleksandargjoreski.dev/blog/browser-rendering-pipeline/>
11. Rendering performance | Articles - web.dev, дата последнего обращения: декабря 6, 2025, <https://web.dev/articles/rendering-performance>
12. Compositor Thread Architecture - The Chromium Projects, дата последнего обращения: декабря 6, 2025, <https://www.chromium.org/developers/design-documents/compositor-thread-architecture/>
13. Populating the page: how browsers work - Performance - MDN Web Docs - Mozilla, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/How_browsers_work>
14. RenderingNG deep-dive: LayoutNG | Chromium - Chrome for Developers, дата последнего обращения: декабря 6, 2025, <https://developer.chrome.com/docs/chromium/layoutng>
15. The Core of Web Performance: A Deep Dive into Reflow (Layout) and Repaint - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@huangjianzhong/the-core-of-web-performance-a-deep-dive-into-reflow-layout-and-repaint-053d18d7f168>
16. What's the difference between reflow and repaint? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/2549296/whats-the-difference-between-reflow-and-repaint>
17. Performance Optimization — Thoroughly Understanding and Deconstructing Reflow, Repaint, and Compositing. | by Jude Wei | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@weijunext/performance-optimization-thoroughly-understanding-and-deconstructing-reflow-repaint-and-d5d9118f2cdf>
18. RenderingNG architecture | Chromium - Chrome for Developers, дата последнего обращения: декабря 6, 2025, <https://developer.chrome.com/docs/chromium/renderingng-architecture>
19. How browsers work | Articles - web.dev, дата последнего обращения: декабря 6, 2025, <https://web.dev/articles/howbrowserswork>
20. Optimizing Unity UI - Unity Learn, дата последнего обращения: декабря 6, 2025, <https://learn.unity.com/tutorial/optimizing-unity-ui>
21. Are layout thrashing, reflow the same meaning in HTML? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/66450070/are-layout-thrashing-reflow-the-same-meaning-in-html>
22. Avoid large, complex layouts and layout thrashing | Articles - web.dev, дата последнего обращения: декабря 6, 2025, <https://web.dev/articles/avoid-large-complex-layouts-and-layout-thrashing>
23. How To Fix Forced Reflows And Layout Thrashing - DebugBear, дата последнего обращения: декабря 6, 2025, <https://www.debugbear.com/blog/forced-reflows>
24. Scripting API: Canvas.ForceUpdateCanvases - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Canvas.ForceUpdateCanvases.html>
25. How does forced synchronous layout influence the browser event loop? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/67916109/how-does-forced-synchronous-layout-influence-the-browser-event-loop>
26. Unity UI performance optimization tips, дата последнего обращения: декабря 6, 2025, <https://unity.com/how-to/unity-ui-optimization-tips>
27. How to use and how works CSS' will-change property? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/26907265/how-to-use-and-how-works-css-will-change-property>
28. Efficient DOM Manipulation with document.createDocumentFragment() - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/vivek96_/efficient-dom-manipulation-with-documentcreatedocumentfragment-54i6>
29. DocumentFragment - Web APIs | MDN, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/API/DocumentFragment>
30. Performance Optimization in DOM Manipulation | by Carlos A. Rojas - Client-Side JavaScript, дата последнего обращения: декабря 6, 2025, <https://blog.carlosrojas.dev/performance-optimization-in-dom-manipulation-6669ae153847>
31. VirtualizedScrollRectList Class (MixedReality.Toolkit.UX.Experimental) | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/api/mixedreality.toolkit.ux.experimental.virtualizedscrollrectlist?view=mrtkuxcore-3.2>
32. disas69/Unity-PooledScrollList - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/disas69/Unity-Pooled-Scroll-List>
33. List Virtualization - Patterns.dev, дата последнего обращения: декабря 6, 2025, <https://www.patterns.dev/vanilla/virtual-lists/>
34. Delegate Events VS UnityEvent, which one is superior? (If you don't know what UnityEvents are, you should probably read this. It's going to change your life.) : r/Unity3D - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/Unity3D/comments/35oekm/delegate_events_vs_unityevent_which_one_is/>
35. Why does Event Delegation save memory? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/65929396/why-does-event-delegation-save-memory>
36. Comparing Memory Consumption of Delegated Event Listeners and Individual Event Listeners in Javascript - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/mohamadharith/comparing-memory-consumption-of-delegated-event-listeners-and-individual-event-listeners-in-javascript-1l3e>
37. What is the difference between Native and ShadowDom ViewEncapsulation?, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/53409943/what-is-the-difference-between-native-and-shadowdom-viewencapsulation>
38. Pros and cons of using Shadow DOM and style encapsulation - Manuel Matuzovic, дата последнего обращения: декабря 6, 2025, <https://www.matuzo.at/blog/2023/pros-and-cons-of-shadow-dom/>
39. Using shadow DOM - Web APIs | MDN, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM>