GPU & UI Optimization

Session 2

Agenda

- GPU Optimization
  - What's GPU?
  - Difference Between GPU & CPU
  - NVIDIA GPU versus CPU
  - Rendering Optimization Terms
    - Draw Call
    - Set pass Call
    - Batching
    - Others
- Unity Tools
  - Stats
    - Triangles Count
    - Vertices Count
    - Shadow Caster
    - Other
  - Frame Debugger
  - Profiler
    - Rendering
      - Used Textures
      - Render Textures
      - Render Texture Changes
      - Used Buffer
      - Vertex Buffer
      - Index Buffer
      - Others
    - UI
    - UI Details

Unity

Agenda

- GPU Optimization Techniques
  - Static Batching
  - Dynamic Batching
  - Enable GPU Instancing
  - SRP Batcher
  - Optimization Priority
  - Reducing Shadow Usage
  - Texture Optimization
    - Texture Size
    - Crush Compression
    - Texture Atlas
  - Combine Meshes
  - Others

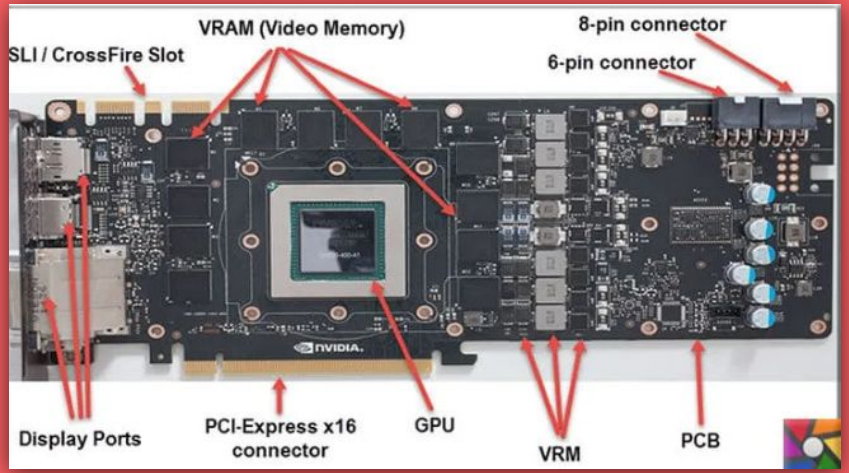
Agenda

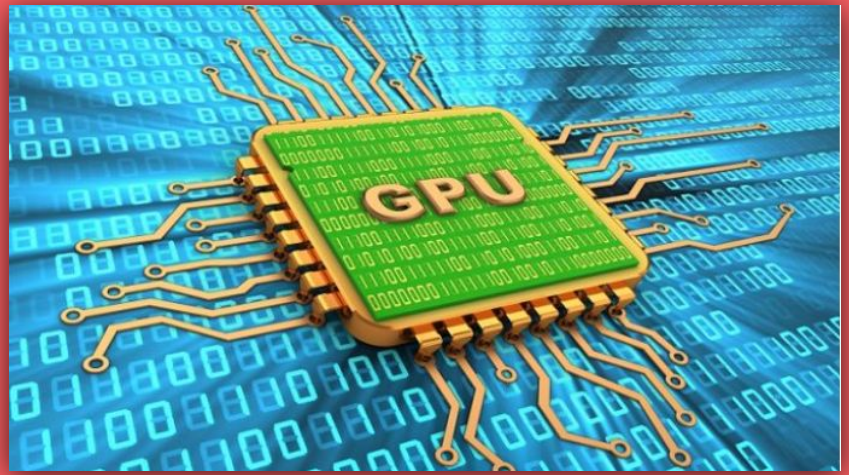
- UI Optimizations
  - Optimization Techniques
    - Divide Canvases
    - Sprite Atlas
    - Graphic Raycaster
    - Camera.main
    - Hiding Canvas
    - Animator on UI Elements
    - Tint
  - UI Optimization Tips
- Unity Optimization Tips

What's GPU?

(*Graphic Processing Unit*)

- The GPU is a specialized electronic component used to **accelerate graphics rendering** and **perform parallel computations** in a computer system.
- Modern GPUs have evolved beyond graphics to **become vital for accelerating complex computations** in **AI**, **scientific research**, and **data analysis**, notably in **deep learning** and **cryptocurrency mining**, owing to their **robust parallel processing capabilities**.





Difference Between CPU & GPU

The key difference between a CPU and a GPU lies in their **design**, **architecture**, and **primary functions**:

1. Architecture:

- CPU: Optimized for **single-threaded performance** with a few powerful cores.
  *(Example: AMD EPYC 9654 - **96 Cores**)*
- GPU: Optimized for **parallel processing** with **many simpler cores.**
  *(Example: GeForce RTX 4090 - **16.384 Cores**)*

2. Parallelism:

- CPU: Efficient at **serial tasks**; less suited for high parallelism.
- GPU: Excellent at **parallelism**, capable of **simultaneous execution** of numerous threads.

Difference Between CPU & GPU

3. Function:

- CPU: General-purpose processor for **diverse tasks.**
  *(**OS**, **Apps**, **File Management**, **Networking**, **Security** ….)*
- GPU: Specialized for **graphics rendering** and **parallel computing.**

4. Workload:

- CPU: Ideal for **complex decision-making**, **control flow**, and **single-threaded tasks** like running the OS and managing applications.
- GPU: Suited for **data-parallel tasks** such as **graphics rendering**, **deep learning**, and **large-scale scientific computations**.

*In summary, **CPUs are versatile but optimized for single-threaded performance**, while **GPUs excel in parallel processing**, making them **vital for parallelizable tasks** like **graphics** and **AI-related workloads**.*

NVIDIA GPU VERSUS CPU

Rendering Optimization Terms

- Draw Call
- Set pass Call
- Batching
- Others

Unity

Draw Call

❑ Draw Call

- ❑ The Draw Call is a **command sent to the GPU to render an object** or **a group of objects** in a graphics scene.
- ❑ Each draw call represents a unit of work for the GPU, and it often involves setting up **shaders**, **textures**, and **other rendering parameters** like **vertices of the mesh, transform**. *(Draw it here with these informations)*
- ❑ Optimizing draw calls is crucial for GPU performance because **excessive draw calls can lead to a significant overhead**, especially in scenes with many objects.

Simply, we can assume that every single material usage generates one draw call in Unity, for now.

CPU Workload on Draw Call

The Workload

- Before sending a draw call to the GPU, **the CPU performs** tasks such as **scene setup**, **shader compilation**, **data preparation**, **state configuration**, **culling**, **sorting**, and **draw call generation**.
- These tasks prepare the scene and specify how objects should be rendered, and they are essential for the GPU to carry out the rendering process effectively.

So, there is also excessive workloads for CPU, during draw call process.

Unity

Are you curious for more about CPU's Draw Call Workload?

1. **Scene Setup**: Before rendering begins, the CPU is responsible for setting up the scene. This includes tasks like loading 3D models, textures, and shaders, as well as arranging objects in the virtual environment.
2. **Shader Compilation**: Shaders are small programs that determine how objects are rendered. The CPU compiles these shaders into a format that the GPU can execute. Shader compilation typically occurs during application initialization.
3. **Data Preparation**: The CPU prepares the data required for rendering, such as vertex data (e.g., positions, normals, texture coordinates), material properties, and transformation matrices. This data is often stored in buffers that the GPU can access.
4. **State Configuration**: The CPU configures the rendering state, which includes setting rendering parameters such as the render target (framebuffer), depth and stencil buffers, blend modes, and rasterization settings. These settings define how the GPU will process the geometry.
5. **View and Projection Matrices**: The CPU computes view and projection matrices to transform 3D world coordinates into the 2D screen space where rendering occurs. These matrices are used to position and project objects in the scene.
6. **Culling and Sorting**: The CPU may perform culling techniques to determine which objects are visible and which are not. Additionally, it may sort objects to optimize rendering order, such as rendering objects closer to the camera first (front-to-back rendering) to reduce overdraw.
7. **Uniform Updates**: If there are dynamic changes in the scene, the CPU updates uniform variables in shaders. These variables might include camera transformations, lighting parameters, and other variables that affect rendering.
8. **Draw Call Generation**: The CPU finally generates draw calls based on the objects to be rendered, the shaders to be used, and the associated data. Each draw call specifies what should be rendered and how it should be rendered.
9. **API Calls**: The CPU communicates with the graphics API (e.g., DirectX, OpenGL, Vulkan) to issue draw calls and other rendering commands. These API calls prepare and send the rendering instructions to the GPU.

Set Pass Call

Set pass Call

- A set pass call (or sometimes called a "**render pass**") is a command used in graphics programming to **define a rendering state**.
- Each pass can have its own **shaders**, **render targets**, and **other settings**.
- **Changing rendering states** can be **expensive** in terms of GPU performance.
- Developers aim to **group together objects** and operations that share the **same rendering settings to reduce the number of pass changes**.

Batching

Batching

- **Combining multiple objects** into a single draw call to reduce the number of rendering commands and **improve efficiency**.
- Unity **cannot pack everything into a single batch**. It can batch together quite a large amount of geometry vertices, but it can **set the material only once per batch**.

Agenda

- Unity Tools
  - Stats
    - Triangles Count
    - Vertices Count
    - Shadow Caster
    - Other
  - Frame Debugger
  - Profiler
    - Rendering
      - Used Textures
      - Render Textures
      - Render Texture Changes
      - Used Buffer
      - Vertex Buffer
      - Index Buffer
      - Others
    - UI
    - UI Details

The Stats Window

This window shows **real-time rendering statistics**, which is incredibly useful for optimizing performance.

Tris (Triangles):

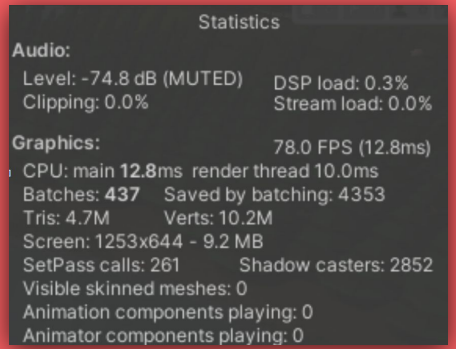
- Tris count indicates the **number of triangles used to render objects**. More triangles mean more detailed 3D models, affecting GPU performance. Optimization involves simplifying geometry and using LOD techniques.

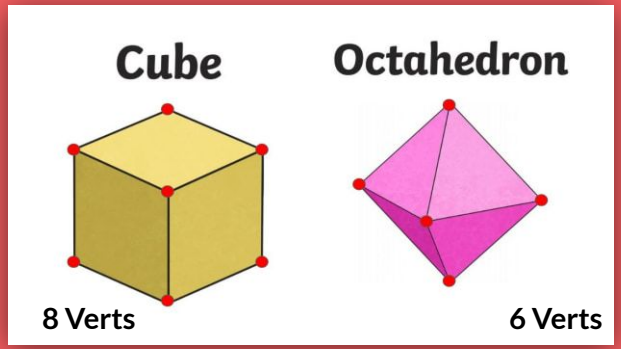
Verts (Vertices):

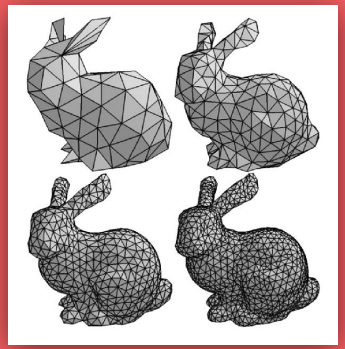
- Vertices are individual points in 3D space used to define the shape of 3D models.
- Verts count shows the **number of vertices defining 3D models**. Higher vertex counts impact CPU and GPU performance. Optimization includes reducing unnecessary vertices and LOD usage.

Shadow Casters:

- Shadow Casters are **objects that cast shadows**. Rendering **shadows** is **resource-intensive**.
- Controlling which objects cast shadows optimizes performance.

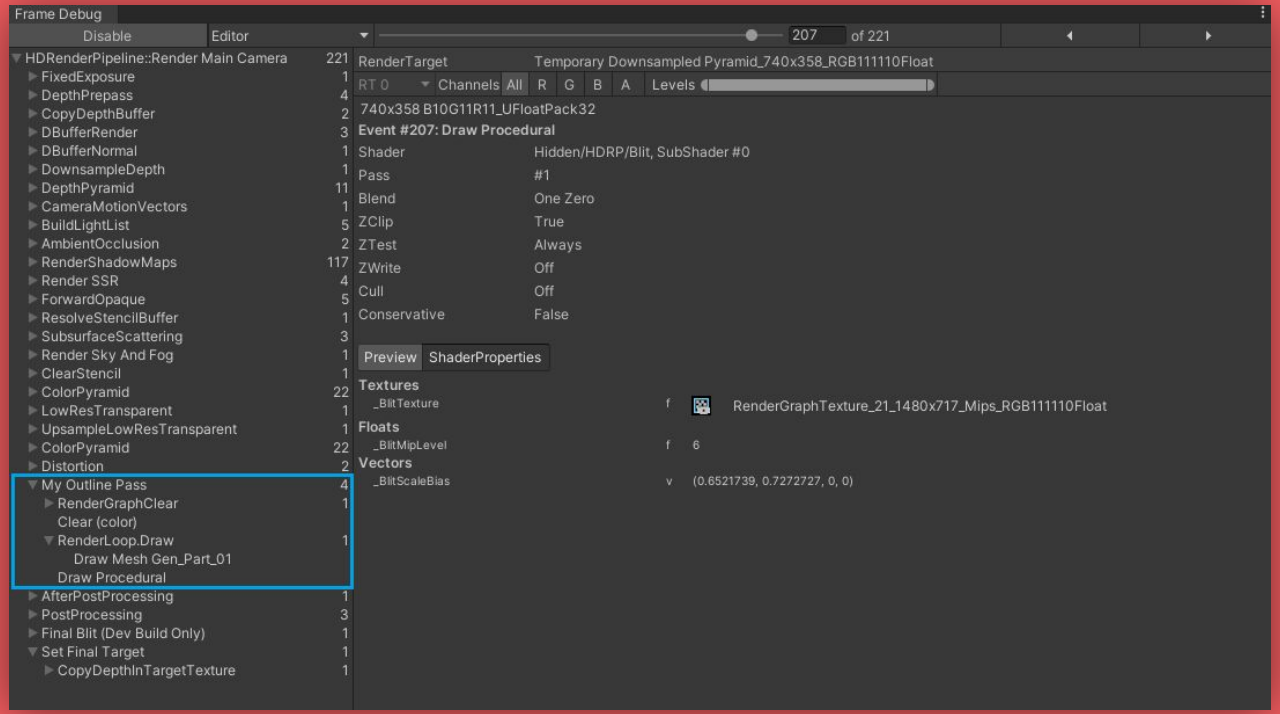






Frame Debugger

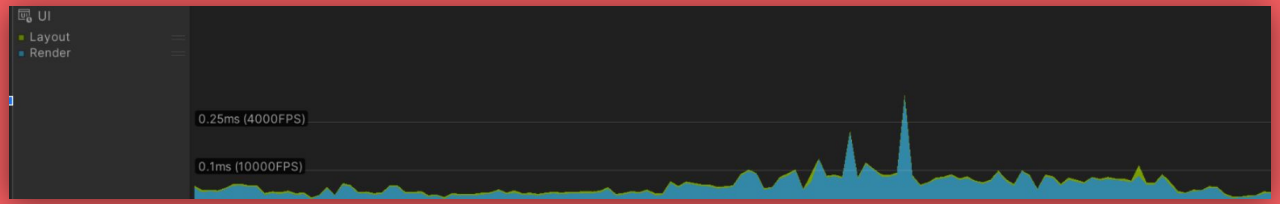
The Frame Debugger can **step through each event** and **display the graphical state** of the **scene** at that **point in the rendering process**.

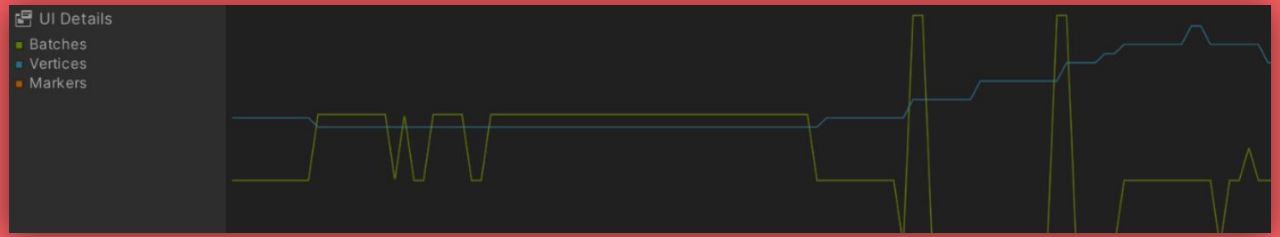


Unity Profiler

The Profiler gathers and displays data on the performance of your application in areas such as the CPU**,** memory**, renderer** and audio**.**

➢ **Let's deep dive to Rendering, UI and UI Details profiling sections.** 🤓

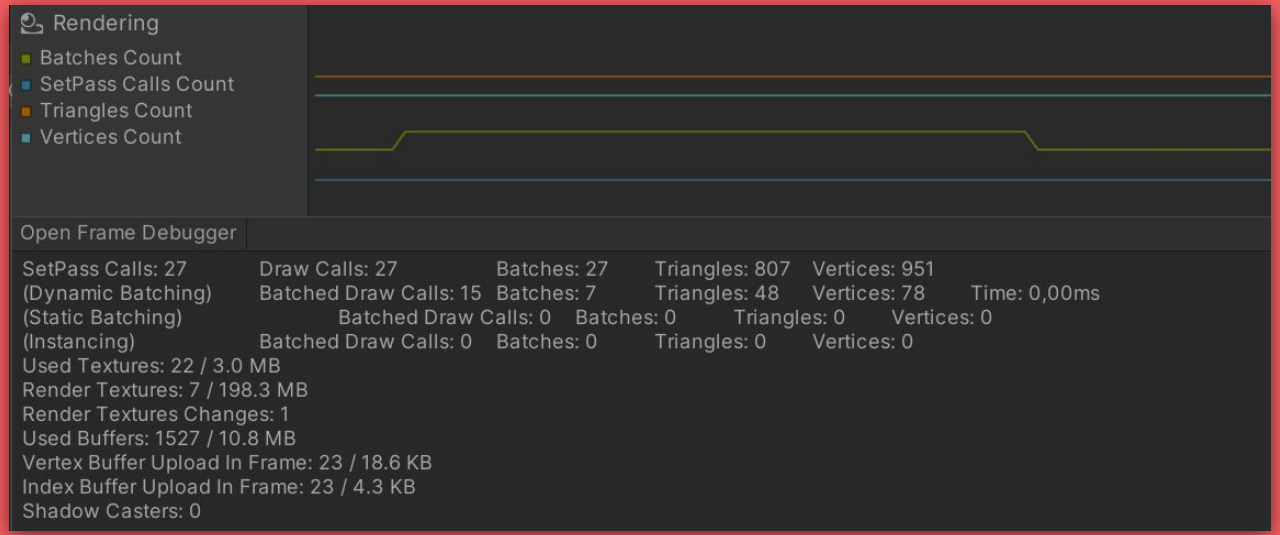




Rendering Profiler

The Rendering **Profiler** displays rendering statistics and informations.

➢ Let's briefly delve into some less familiar terms such as **Used Textures**, **Render Textures**, **Render Texture Changes**, **Used Buffer**, **Vertex Buffer and Index Buffer Uploads In Frame.** 👀



Rendering Profiler

➢ Used Textures

The number of textures Unity used during the frame and the **amount of memory the textures used**.

➢ Render Textures

A **Render Texture** is a type of Texture that **Unity creates** and **updates at run time**.

The number of Render Textures Unity used during the frame and the **amount of memory the Render Textures** used.

➢ Render Texture Changes

Represents the number of times Render Textures are **switched** or **changed during rendering**.

➢ Used Buffer

The total number of **GPU buffers** and **memory** they **used**. This includes **vertex**, **index** and **compute buffers** and **all internal buffers** required for rendering.

➢ Vertex Buffer Uploads In Frame

The amount of geometry that the CPU uploaded to the GPU in the frame. This represents the **vertex/normal/texcoord data**.

➢ Index Buffer Uploads In Frame

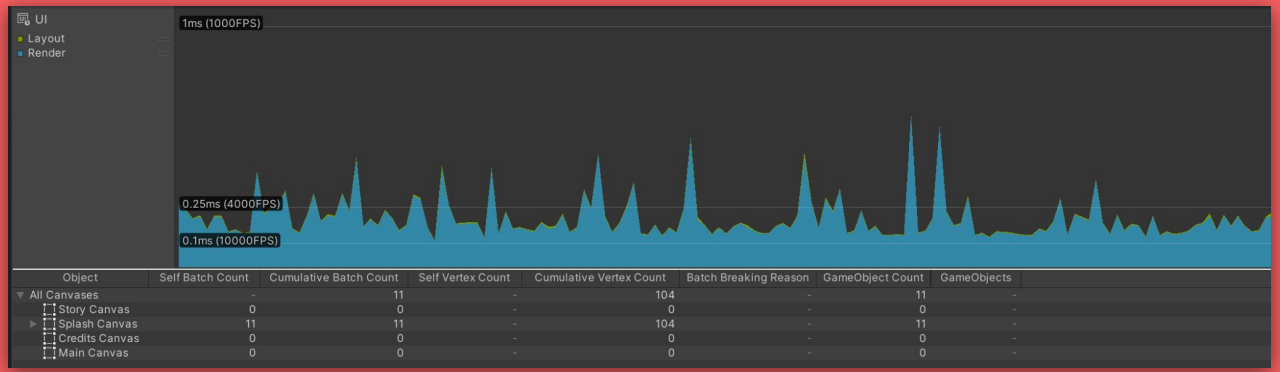
The amount of geometry that the CPU uploaded to the GPU in the frame. This represents the **triangle indices data**.

UI Profiler

The UI Profiler helps you **identify bottlenecks** and **performance issues** related to **UI rendering**, **layout calculations**, and **UI interaction**. It allows you to pinpoint areas that need optimization.

Layout: How much time Unity has spent performing the layout pass for the UI. This includes calculations done by **HorizontalLayoutGroup**, **VerticalLayoutGroup**, and **GridLayoutGroup**.

Render: How much **time** the UI has **spent** doing its **portion of rendering**. This is either the **cost of rendering** directly to the graphics device or rendering to the main render queue.

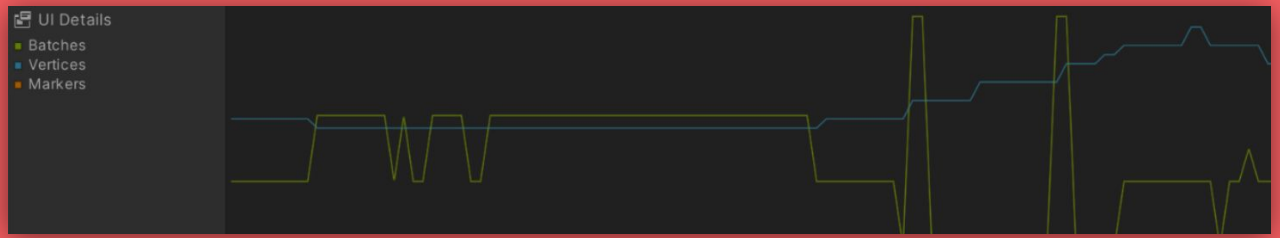


UI Details Profiler

Batches: Displays the **total number of draw calls** that are batched together.

Vertices: The **total number of vertices** that are used to render a section of UI.

Markers: Displays event markers. Unity records markers when the **user interacts with the UI** (for example, a **button click**, or a **slider value change**) and then draws them as vertical lines and labels on the chart.



Agenda

- GPU Optimization Techniques
  - Static Batching
  - Dynamic Batching
  - Enable GPU Instancing
  - SRP Batcher
  - Optimization Priority
  - Reducing Shadow Usage
  - Texture Optimization
    - Texture Size
    - Crush Compression
    - Texture Atlas
  - Combine Meshes
  - Others

Static Batching

Unity Static Batching is an **optimization technique used to reduce the number of draw calls** by combining static objects in your scene into a single batch. To utilize Static Batching effectively, you need to meet specific conditions and considerations:

- Objects are **static** and **won't be moved**, **rotated** and **scaled**.
- Objects **share** the **same material and shader**.
- Objects **have Mesh Renderer** components.
- Parent objects have the **"Static" flag** enabled to combine children.
- Objects **won't be modified during runtime**.
- Use it for small to medium-sized objects. ***(~15K Vertices)***
- Consider alternatives for animated or dynamic objects.
- **Perform batching** during the **build process** or in the Editor's Lighting window **after baking lighting**.

LET'S TAKE A LOOK TO SAMPLE 👀

Unity

Dynamic Batching

Depending on the shader being used objects will dynamically batch if they're below a certain amount of vertices:

- 180 vertices for a shader using vertex position, normal, UV0, UV1 and tangents
- 300 vertices for a shader using vertex position, normal and a single UV *(This is the most common limit for simple objects with basic shaders)*

Unlike static batching, **dynamically batched objects can be moved around and have physics with a rigidbody**. They do also have their own set of limitations though:

- Must be the exact same scale to batch (mirrored scales are also not supported, e.g if you have an object with -1 scale vs an object with 1 scale)
- Need to share the exact same material reference(s) to batch together
- Must be rendered together in order! If the render order has something else rendering between a batch then it'll be broken up into multiple batches
- Each object must be below the vertex limits mentioned above, depending on the shader it's using

(Note: If you're developing a Unity PC game keep in mind that dynamic batching has a CPU overhead for batching so unless you're being bottlenecked by heavy GPU rendering time you probably want to disable dynamic batching in your project!)

Dynamic Batching CPU Overhead

- **Object Management:** ***Tracking*** *and* ***managing dynamic objects*** *for batching.*
- **Transformation Updates:** ***Updating*** *object* ***transformations*** *during runtime.*
- **Batching Process:** *Preparing* ***rendering data*** *and* ***issuing draw calls****.*
- **CPU-GPU Synchronization:** *Ensuring* ***GPU synchronization*** *to* ***avoid rendering conflicts****.*
- **Complex Shaders:** ***CPU-intensive*** *shader* ***computations****.*
- **Render Thread:** ***Coordinating*** *with the* ***render thread*** *for rendering.*

In practice, dynamic batching remains a valuable optimization technique for reducing draw call counts and improving rendering performance, but it's essential to be aware of its limitations and to use it judiciously in scenarios where it provides the most significant benefits.

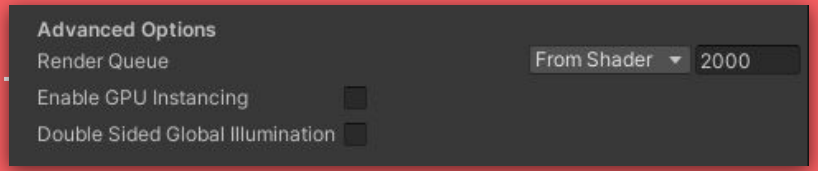
Enable GPU Instancing

GPU instancing is a **draw call optimization method** that renders multiple copies of a mesh with the same material in a single draw call.

- ❏ Must have the **same material**
- ❏ Must have the **same mesh**
- ❏ **Efficient** and primarily used for **semi-static** objects

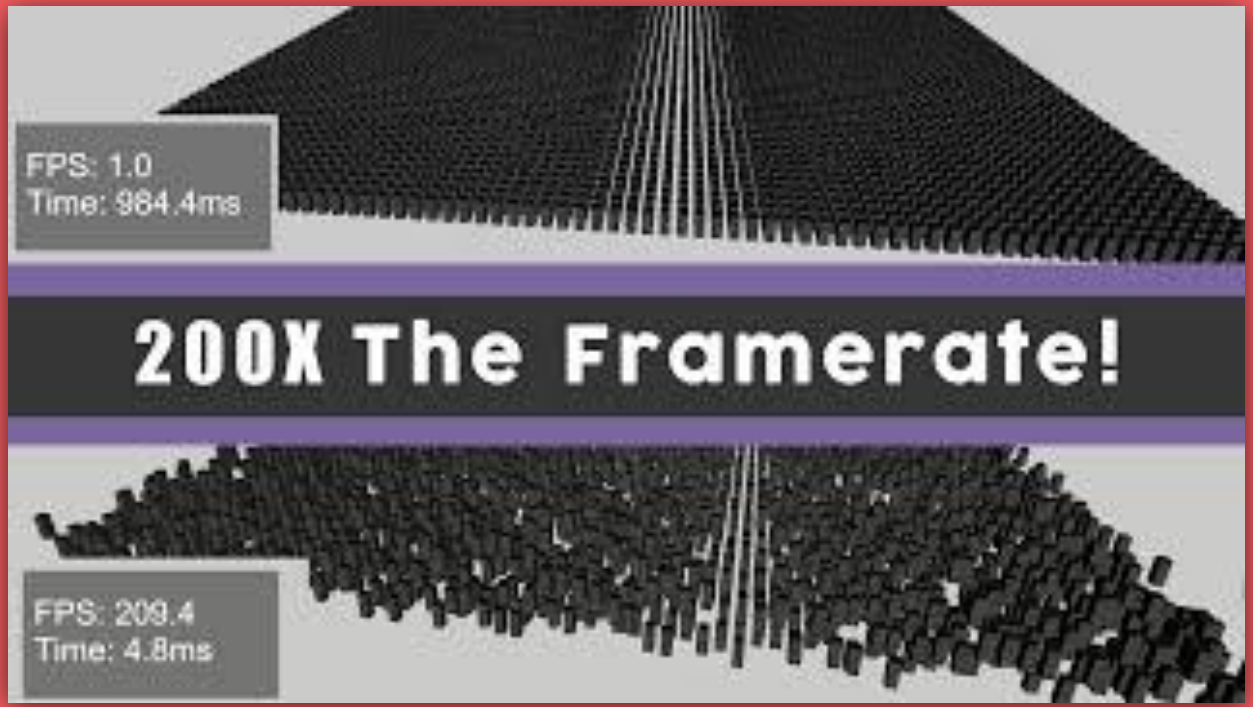
Examples of semi-static objects in a game might include:

- **Trees** and **bushes** that sway gently in the wind but are otherwise stationary.
- Decorative objects like **lamps**, **signs**, or **furniture** that can be **bumped or moved by the player**.



Unity

Summarizing Optimization Techniques



SRP Batcher

The SRP Batcher is a **rendering optimization feature** in Unity's Scriptable Render Pipelines that **reduces CPU overhead** and **draw call counts** by efficiently batching render operations.

- ➢ Batches together draw call for objects that share the **same shader variant**, **even with different materials**.
- ➢ Must contain either a **mesh** or **a skinned mesh**. ***(Only in SRP Batcher)***
- ➢ The shader must be **compatible** with the **SRP Batcher**.

Render Pipeline Compatibility

Render pipeline compatibility

Feature	Built-in Render Pipeline	Universal Render Pipeline (URP)	High Definition Render Pipeline (HDRP)	Custom Scriptable Render Pipeline (SRP)
Static Batching	Yes	Yes	Yes	Yes

Render pipeline compatibility

Feature	Built-in Render Pipeline	Universal Render Pipeline (URP)	High Definition Render Pipeline (HDRP)	Custom Scriptable Render Pipeline (SRP)
Dynamic Batching	Yes	Yes	No	Yes

Render pipeline compatibility

Feature	Built-in Render Pipeline	Universal Render Pipeline (URP)	High Definition Render Pipeline (HDRP)	Custom Scriptable Render Pipeline (SRP)
GPU instancing	Yes	Yes (1)	Yes (1)	Yes (1)

Render pipeline compatibility

Feature	Built-in Render Pipeline	Universal Render Pipeline (URP)	High Definition Render Pipeline (HDRP)	Custom Scriptable Render Pipeline (SRP)
SRP Batcher	No	Yes	Yes	Yes

Optimization Priority

You can use multiple draw call optimization methods in the same scene, but there is a Unity Prioritizes!

1. SRP Batcher and Static Batching
2. GPU Instancing
3. Dynamic Batching

Unity

Shadow Usage

Shadow usage is often considered **inefficient** when it comes to reducing draw call counts because shadows involve **additional rendering calculations** and processing that can increase the **overall workload** on **both the CPU and GPU**.

- **Use shadow distance** and **quality settings** to balance visual fidelity and performance.
- Consider using **baked (precomputed) shadows for static objects** to **reduce runtime calculations.**
- Implement **Level of Detail (LOD)** techniques to reduce the rendering load on objects that are farther from the camera.
- **Use GPU Instancing** and **Static Batching** to **reduce draw call** counts for objects that cast and receive shadows.

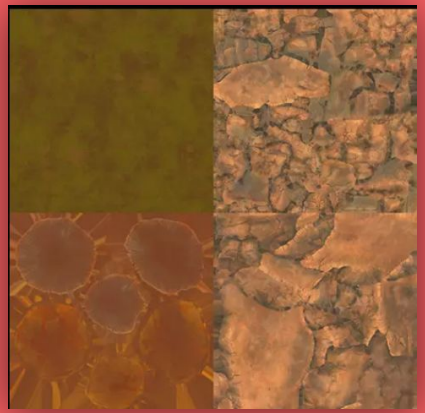
Material Shader Optimization

- **Transparent Material:**
  - Standard Shader: Use "Transparent" mode.
  - Legacy Shader: Use the "Transparent" shader and control transparency with alpha.
- **Specular Highlights:**
  - Standard Shader: Built-in support for highlights.
  - Legacy Shader: Use "Bumped Specular" for similar highlights.
- **Normal Mapping:**
  - Standard Shader: Supports normal mapping.
  - Legacy Shader: Use "Bumped Diffuse" with a normal map.
- **Reflections:**
  - Standard Shader: Advanced reflection features.
  - Legacy Shader: Basic reflections with "Reflective" shader.

*Use Unity's Standard Shader for **modern projects with PBR**, **wide platform support,** and following current best practices. Use Legacy Shaders for older or **specific visual styles,** or **for simpler**, **lightweight projects**. Choose based on your project's needs and goals.*

Texture Optimization

- Texture Size
- Atlas Mapping
- Texture Compression



Unity

Combine Meshes

- Mesh Baker Package
- Export to FBX *for 3D Artist*

Unity

Agenda

- UI Optimizations
  - Optimization Techniques
    - Divide Canvases
    - Sprite Atlas
    - Graphic Raycaster
    - Camera.main
    - Hiding Canvas
    - Animator on UI Elements
    - Tint
  - UI Optimization Tips
- Unity Optimization Tips

Divide Canvases

Description:

The Canvas is the basic component of Unity UI. It generates meshes that represent the UI elements placed on it, regenerates the meshes when UI elements change, and issues draw calls to the GPU so that the UI is displayed.

Generating these meshes can be expensive. UI elements need to be collected into batches so that they're drawn in as few draw calls as possible. Because batch generation is expensive, we want to regenerate them only when necessary.

Problem:

When an element is changed, the whole canvas is reanalyzed.

Many users build their entire game's UI in one single canvas with thousands of elements. So, when they change one element, they can experience a CPU spike costing multiple milliseconds.

Solution:

Divide up your canvases.

Sprite Atlas

Description:

A Texture Atlas is a large texture with a group of different textures. Every object (effect particles, textures) that share the same material are on the same draw call and will be sent to the GPU at once.

Problem:

Huge impact on the performance of your game when you have a ton of effects/textures. Text also is a major cause of extra draw calls as they are on a separate atlas which causes extra batching.

Solution:

Pack your sprites together to reduce draw calls by using Texture Atlases. Tip: TexturePacker is a really good tool for creating texture atlases.

Unity

Graphic Raycaster

Description:

The Graphic Raycaster is the component that translates your input into UI Events. It translates screen/touch input into Events, and then sends them to interested UI elements. You need a Graphic Raycaster on every Canvas that requires input, including sub-canvases.

Problem:

The Graphic Raycaster performs intersection checks on UI elements that are interested in receiving input. However, not all UI elements are interested in receiving updates.

Solution:

Disable Raycast Target property for all non-interactive elements. For example, a text on a button. Turning off the Raycast Target will reduce the number of intersection checks the Graphic Raycaster must perform each frame.

Camera.main

Description:

When setting up a Canvas, you can specify which Camera to anticipate where interaction events should come from. There is an optional setting for World Space canvases called the Event Camera.

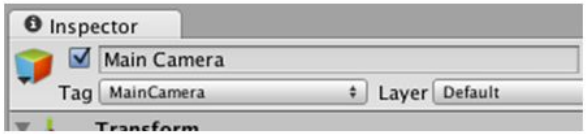
If you leave the Event Camera field blank on a World Space Canvas, it uses the game's main camera (Camera.main).

Problem:

Unity will access Camera.main between 7–10 times per frame, per Graphic Raycaster, per World Space Canvas. Camera.main also calls Object.FindObjectWithTag every time it's accessed!

Solution:

Avoid the use of Camera.main. If you use World Space canvases, always assign an Event Camera. Do not leave this setting empty!



Unity

Hiding Canvas

Description:

At times, you may want to hide your UI canvas or a group of components. There are a few ways to do it.

Problem:

Any way of hiding the canvas that leaves it enabled, even if invisible, leaves it prone to dirtying, which can reduce performance.

Solution:

Disable the entire canvas. If you want only a certain group of elements to disappear, group them on their own separate canvas.

Unity

Animators on UI Elements

Problem: Using animators on your UI

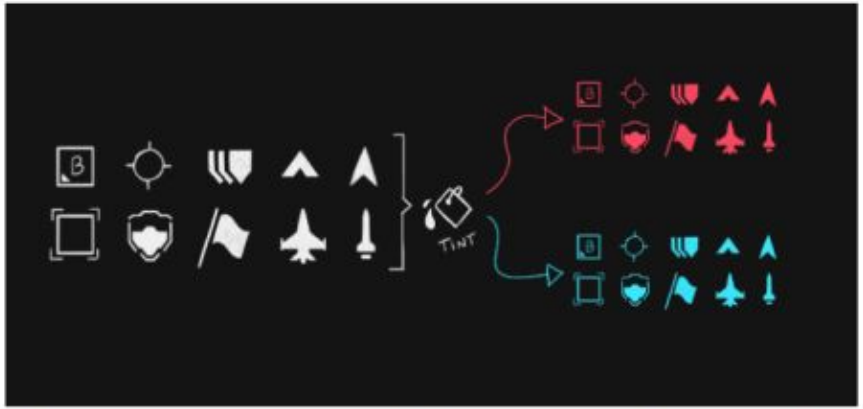
Animators will dirty their UI Elements on every frame, even if the value in the animation does not change.

Solution: Use code for UI Animation.

Only put animators on dynamic UI Elements that always change. For Elements that rarely change or that change temporarily in response to events, write your own code or use a tweening system. There are a number of great solutions for this available on the Asset Store.

Tint

In this scenario, if there are ten different icons that need to appear in three colors, we would have a total of 30 files. However, using tint, we will export only 10, but in white. Then we will apply the tint inside Unity with the necessary color through the inspector or via code.



On the left, we have the example of 10 icons used in the HUD that were exported in white and on the right the result after applying the necessary Tint.

When applied, **in addition to making maintenance much easier for everyone on the team, tint can drastically reduce the number of UI files in the project in the long run.**

Unity

UI Optimization Tips

- ★ **Avoid to use animators** on your UI. if you use, put animators on dynamic elements in different canvas or sub canvas.
- ★ **Limit the usage** of “**Best Fit**” and “**Rich Text**” in text components.
- ★ **Limit** the **usage** of **Outline/Shadow** components on Text objects.
- ★ **Avoid nesting** transform **hierarchy**.
- ★ Avoid the **expensive UI Elements** (Large grid, layout groups)
- ★ Disable **Raycast Target**, if it’s unnecessary.

Unity Optimization Tips

- ★ Set animation frame to 30 - ***especially for mobile phones***
- ★ Set application target frame to 30 - ***especially for mobile phones***
- ★ **Reduce vertex** and **polygon counts** as much as possible.
- ★ **Avoid Write/Read** enabled meshes. **Why?** 🤔
- ★ **Limit** Camera **clipping distance**.

What else, your ideas? 🤔