**OPTIMIZATION RULES & TRICKS (Basics)**

**DEVELOPERS:**

1. **Minimize Draw Calls:** Reduce the number of draw calls by combining meshes and materials where possible.



To reduce draw calls in Unity:

Note: It is the best practice to optimize stuff in any 3D modeler tool, however in the case of emergency one may apply these tricks.

**(i) Mesh Combining:** Merge meshes at runtime using `Mesh.CombineMeshes()` to decrease draw calls(for small objects only, otherwise bake a static combined mesh), Use “Mesh Baker” for advanced features related to it.

link:

<https://assetstore.unity.com/packages/tools/modeling/mesh-baker-5017>

**(ii) Static Batching:** Mark static GameObjects to automatically batch them if they share the same material. Note :It will increase build size at the build time.

**(iii) Dynamic Batching:** Unity can batch dynamic(moving) GameObjects with the same material, but with limitations.

**Limitations:**

**Vertex Count Limit:** Dynamic batching is most effective for small batches of GameObjects with a low vertex count. When the combined vertices of dynamic GameObjects exceed a certain threshold (usually around 900 vertices per batch), Unity stops batching them dynamically, resulting in separate draw calls for each batch.

**Material Properties:** Dynamic batching only works for GameObjects that use the same material with identical properties. If GameObjects have different material properties (e.g., different colors or textures), Unity cannot dynamically batch them together, and they will be rendered separately.

**GPU Overhead:** While dynamic batching reduces CPU overhead by combining draw calls, it can increase GPU overhead due to duplicated vertex processing for each batch. This overhead can become significant if there are many small dynamic batches in the scene.

**Shader Complexity:** Dynamic batching may not work well with complex shaders that use features like vertex displacement or procedural effects. Such shaders can prevent Unity from batching GameObjects dynamically, leading to separate draw calls.

**Skinned Mesh Rendering:** Dynamic batching does not work with skinned mesh renderers, which are commonly used for character animations. Skinned meshes require a different rendering approach and cannot be dynamically batched with other GameObjects.

**(i). Material Instances:** Use instances of the same material for GameObjects with different properties.

You may do so at runtime

**Example1: Runtime Instance**

public class MaterialInstances : MonoBehaviour

{

[SerializeField] Color color = Color.white;

[SerializeField] Material mat;

// Start is called before the first frame update

// Update is called once per frame

private void Start()

{

mat = gameObject.GetComponent<MeshRenderer>().materials[0];

}

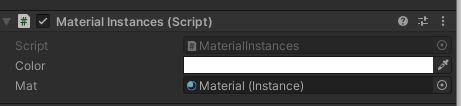
void Update()

{

mat.color = color;

}

}



**Example2: Editor**

You may make instances in the editor.

using UnityEngine;

using System.Collections;

using UnityEditor;

public class MaterialInstancer : Editor

{

[MenuItem("GameObject/Instance Material")]

public static void Instance()

{

if (Selection.activeGameObject == null || Selection.activeGameObject.GetComponent<MeshRenderer>() == null)

{

Debug.LogError("No Valid Object Selected");

return;

}

Material mat = Selection.activeGameObject.GetComponent<MeshRenderer>().sharedMaterial;

Selection.activeGameObject.GetComponent<MeshRenderer>().sharedMaterial = new Material(mat);

}

}

\*\***Texture Atlases**\*\*: Combine textures into one atlas to reduce material count and draw calls.

\*\***Shader Variants**\*\*: Minimize unique shader combinations to avoid unnecessary draw calls.

Use as few as possible shaders, prefer mobile shaders

**Key points:**

- \*\*Memory Usage\*\*: Combining can increase memory usage.

- \*\*Dynamic Objects\*\*: Static or less dynamic objects benefit more.

**2. Use Texture Atlases:** Combine multiple textures into a single atlas to reduce texture switches and memory overhead.

**Ask your 3d modular to create the atlas.**

**Hint: Size should be “Power of Two” 2:1 e.g. 1024\*512 at least keep it 1:1 E.g. 512\*512.**

As a last resortuse Atlas Creator tools, such as **Mesh Baker**. Note: Mesh baker is All Problem one solution related to Mesh Baker.

https://assetstore.unity.com/packages/tools/modeling/mesh-baker-5017

**3. Optimize Meshes:** Reduce the polygon count of your 3D models while maintaining visual quality.

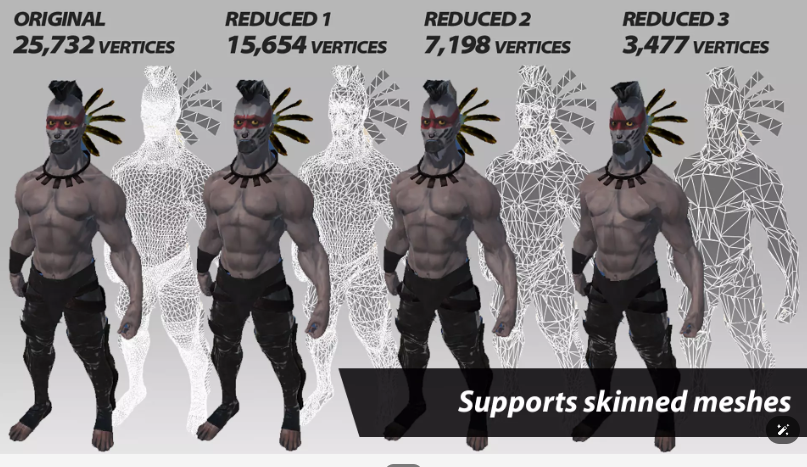
Use low poly GameObjects, check the objects polygones count.



In this example we can see the total vertices, Low Count is better.  
there is more than one mesh filter, these are sub-meshes, and Low count of sub-mesh is better.

Ask your 3D modeler to optimize the mesh otherwise you may use a tool named “Mesh Simplify”.

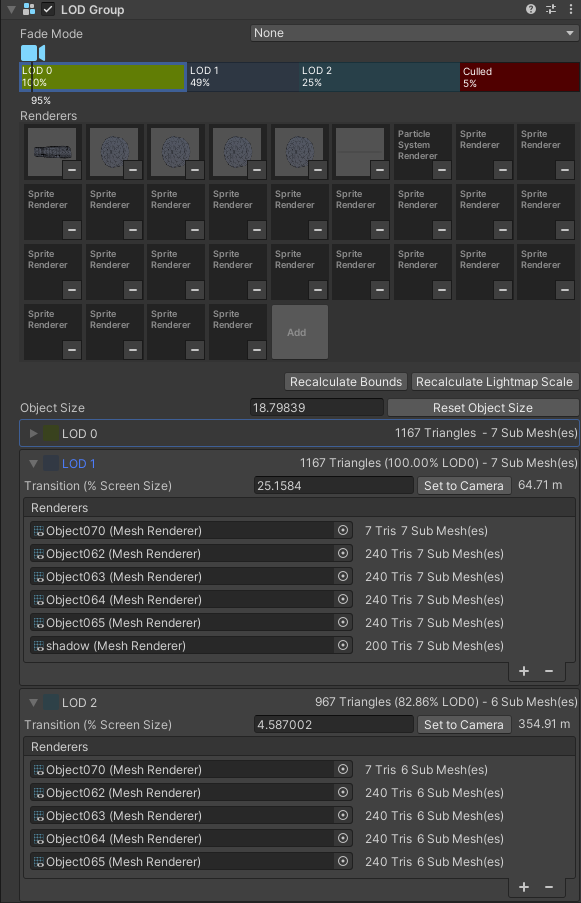
https://assetstore.unity.com/packages/tools/modeling/mesh-simplify-43658



**Note:** Max **Verts** Count should be below **40k** in total, less is better.

****

**4. Level of Detail (LOD):** Implement LOD for distant objects to decrease rendering complexity.



Use LODs in cars and buildings depending on the environment. Remove details step by step.

**5. UI Optimization:** Simplify UI elements and use Unity’s UI tools efficiently.

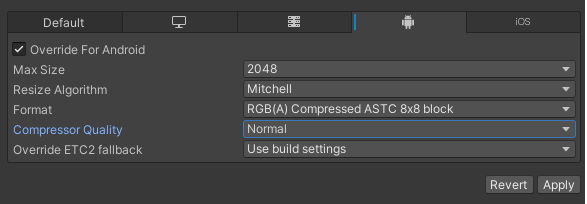
UI optimization in Unity involves streamlining user interface (UI) elements to improve performance and user experience. Here's how to achieve it:

**Use Simple Shapes:** Whenever possible, opt for simple shapes like rectangles, circles, or squares instead of complex custom shapes for UI elements. These shapes require fewer resources to render.

**Optimize Text:** Minimize the use of dynamic text and prefer static text where applicable. Dynamic text that changes frequently can impact performance, so reserve it for essential information. Additionally, use text sparingly and choose appropriate fonts to reduce memory usage.

E.g. Typewriting Effect, Text Tweens

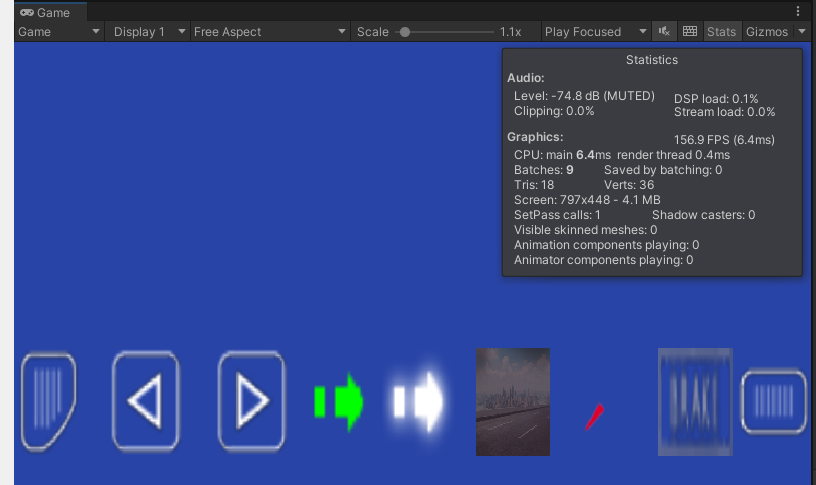
**Image Compression:** Compress images used in UI elements to reduce their file size without sacrificing visual quality. Unity provides tools for image compression and optimization.



Hint: Use ASTC 8\*8

**Sprite Atlas:** Utilize sprite atlases to combine multiple UI images into a single texture atlas. This reduces the number of draw calls and improves rendering performance.

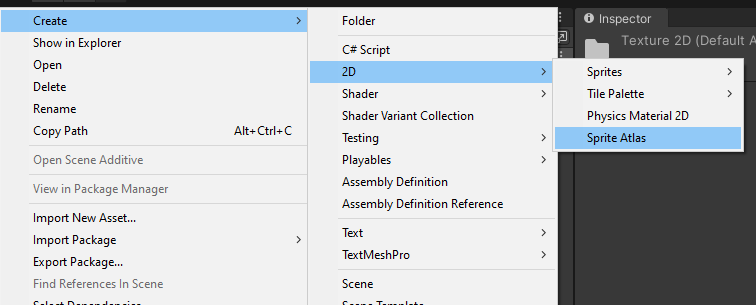
BEFORE ATLAS (Batches Count = 9, 9 Buttons 9 Batches 1 for each button)

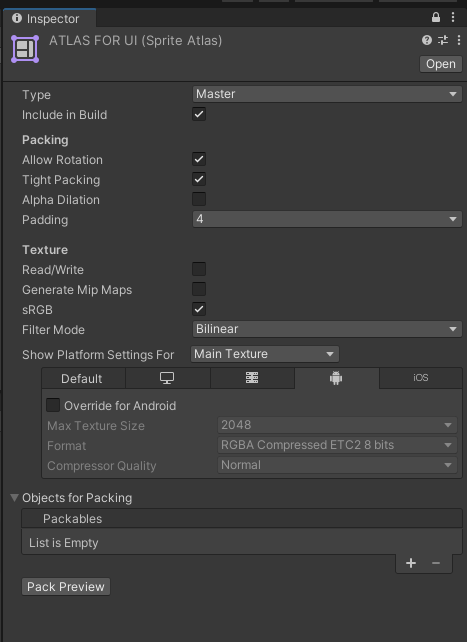


(i) Go to the Project Settings/Editor and choose Sprite Atlas V1\_ Always Enabled



(ii) Right Click in Project/Create/2D/Sprite Atlas and Rename the Atlas File just generated.

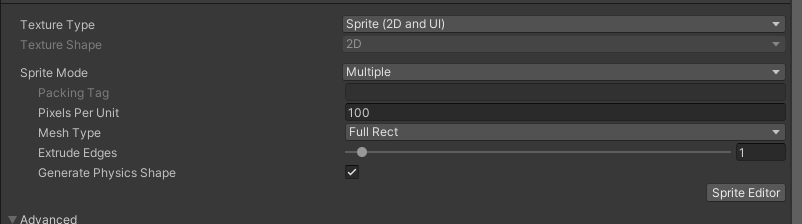




(iii)Lock At the Top in the inspector, Drag and drop all the sprites in Object for Packing, Folder Also works, No need to Separately add children of a folder.  
(iv)Play the Game, Same UI utilizing Single Batch Now 

**UI Batching:** Combine multiple UI elements into a single canvas where possible. Unity automatically batches UI elements within the same canvas, reducing draw calls and improving performance.

Example: Make Atlas in Sprite Editing tool E.g. Photoshop



Click on Sprite Editor and Install Sprite Editor plugin in already not imported.



Now Click “Sprite Editor”/ Slice/SliceIf Any slice in not okay, click it and adjust it manually.

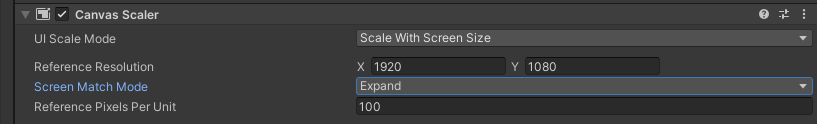


BEFORE



AFTER

**UI Scaling:** Design UI elements to scale efficiently across different screen sizes and resolutions. Use Unity's anchoring and layout components to ensure UI elements adapt correctly to various devices.



**Dynamic UI Updates:** Minimize the frequency of dynamic UI updates, such as animations or transitions, especially on low-end devices. Optimize animations to run smoothly without causing lag or frame drops.

You may use DoTween to animate anything.

Read the doc below

https://dotween.demigiant.com/documentation.php

**UI Element Pooling:** Implement object pooling for frequently instantiated UI elements to reduce memory allocation and garbage collection overhead.

Don’t destroy Ui particle effects after using them at a specific place, instead, you may disable them and enable them again.

**6. Use Object Pooling:** Instead of instantiating and destroying objects frequently, use object pooling to reuse objects. This reduces the overhead of memory allocation and garbage collection.  
The easiest way is to load the objects you need to instantiate later on again and again and keep them disabled and then Enable them.

E.g. enabled(false), enabled(true)

Better Option is to use UnityEngine.Pool

Read the doc below

<https://docs.unity3d.com/ScriptReference/Pool.ObjectPool_1.html>

**7. Use Lightweight Shaders:** Use simple shaders whenever possible. Complex shaders can be performance-intensive, especially on mobile devices.

Warning: Don’t Use Standard Shader, until you have no other option, keep in mind Standard Shader will ruin your game project performance now or later for sure.

Our common tasks can be done through mobile diffuse shader. Otherwise, you may get help from ChatGPT or mistral.ai to modify according to the requirements.   
here is the sample code for mobile defuse  
Shader "Mobile/Diffuse" {

Properties {

\_MainTex ("Base (RGB)", 2D) = "white" {}

}

SubShader {

Tags { "RenderType"="Opaque" }

LOD 150

CGPROGRAM

#pragma surface surf Lambert noforwardadd

sampler2D \_MainTex;

struct Input {

float2 uv\_MainTex;

};

void surf (Input IN, inout SurfaceOutput o) {

fixed4 c = tex2D(\_MainTex, IN.uv\_MainTex);

o.Albedo = c.rgb;

o.Alpha = c.a;

}

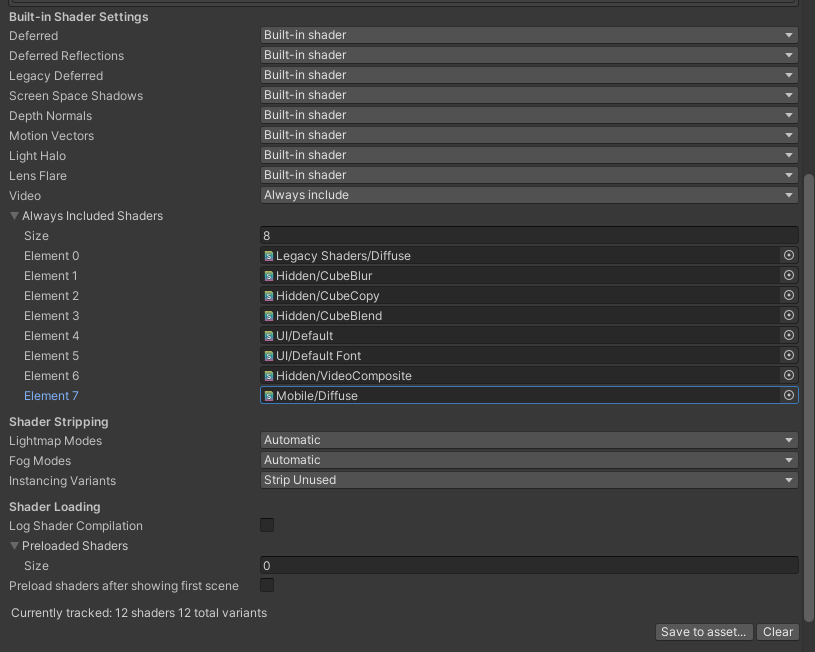
ENDCG

}

Fallback "Mobile/VertexLit"

}

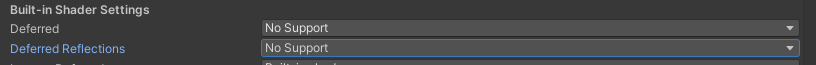
Add the frequently used shaders in the always included shaders list in Project Settings/Graphics/Always Included Shaders.



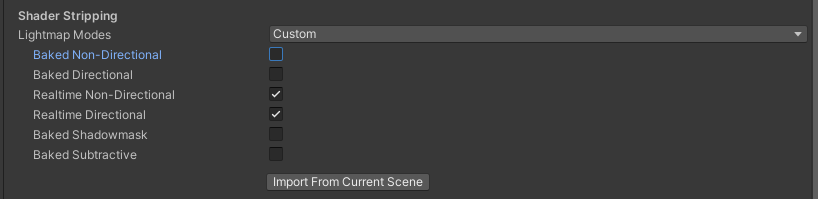
Also remove unrequired shaders E.g.

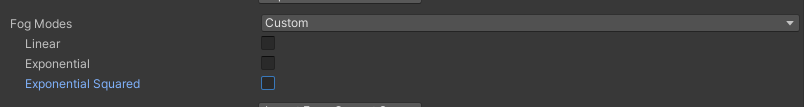
In camera settings if we have rendering path set to forward.  


Then

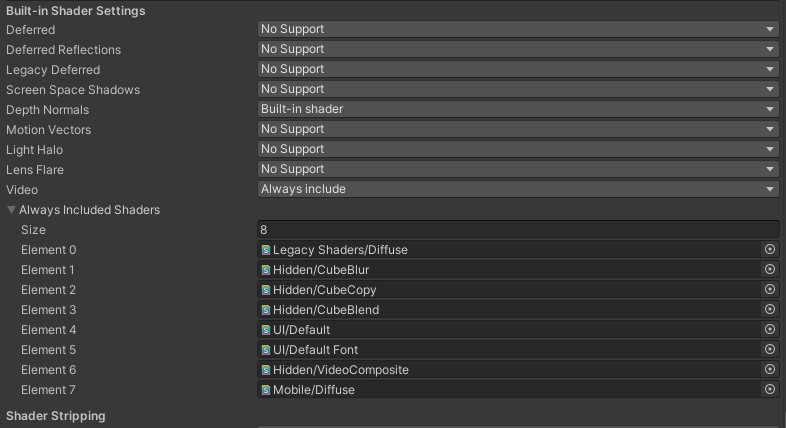


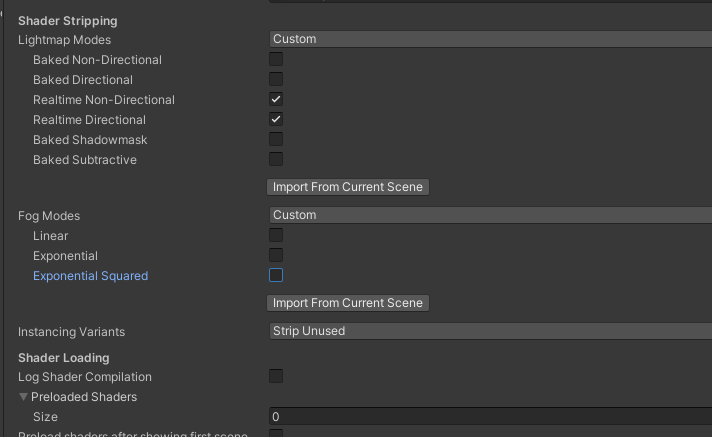
If scene has no Lens flare or light halo then   


Game has no baked light then  


If no fog is used in scene   


Over all





You may preload any foreign shader for better performance, however, please only preload lightweight and small shaders for better performance.  
  
  
  
  
  
  
  
CONTINUES…