BEST PRACTICES GUIDE

9 ways to optimize your game development

Unity expert tips to help you ship

The Explorer: 3D Game Kit — Made with Unity

Introduction

Nearly all games begin life the same way: with a creative vision and a studio's hope to launch the best game possible. It's a great undertaking, and one that can be very rewarding. But there are myriad challenges on the journey to ship a finished and performant game.

With that in mind, Unity's Integrated Success Services (ISS) experts have prepared this guide – organized around 9 important development areas – to help you better understand *and avoid* common memory, performance, and platform issues.

About Integrated Success Services

Our ISS program partners dedicated developer relations engineers and other experts with game studios to optimize their projects and keep them running smoothly. Whether they're helping studios increase productivity, hit tough deadlines, or expand into new territory, Unity's ISS experts have substantial experience and insight to draw on.

Their strategic analysis and optimization recommendations – crystallized here – will help you prevent launch-delaying problems *before they occur* and ensure your players have the best possible experience when you launch. We hope these best practices will guide you on your journey to success.

Please note that the screenshots provided are from different versions of Unity and may not exactly reflect your own Unity version.

1. Planning

Research feature requirements and target platforms

Before starting your project or doing any significant amount of work on it, thoroughly research your feature requirements and target platforms. Ensure that all of the intended platforms actually support what you need (e.g., instanced rendering is not supported on lower-level mobile devices). Be sure you consider the limitations as well as possible workarounds or compromises you're prepared to make.

As well, define the minimum specifications for each target platform and procure multiple hardware units for both your development and QA teams, as you will need a good range of target hardware to test with during development. This will allow you to quickly gauge and adjust realistic performance and frame budgets, and be able to monitor them all the way through development.

Define memory and performance budgets

Once you've established target specifications and feature support, define budgets for memory and performance. This can be tricky, and during development you may need to refine and adjust them, but starting out with a reasonable plan is much better than having no plan and simply throwing anything you like into your project.

To start, determine your target frame rate and your ideal CPU performance budget. For mobile platforms, don't forget that thermal throttling can kick in and reduce both CPU and GPU clock speeds, so allow overhead for them in your planning.

From your CPU budget, try to determine how much time you want to spend on the various systems required: rendering, effects, core logic, and so on.

Memory budgets can be quite difficult to determine. Assets are a major memory consumer, and the main area over which you (as a developer) have control. For example, how much memory overall do you want to commit to assets? Textures, meshes and sounds will all consume large amounts, and this can spiral out of control if you're not careful. Avoid oversized textures, keeping dimensions appropriate for their visible size on-screen as well as the target platform(s)' screen resolution. Similarly, ensure that meshes have vertex and polygon counts appropriate for their use case. An object that only appears far away does not need a highly detailed mesh.

Finally, think about how much memory you can afford for systems in your project. For example, you may implement a particular system that pre-computes a lot of data to reduce the amount of per-update CPU computation, but is this reasonable? Is that one system consuming a disproportionately high amount of memory? Perhaps it is the most important system in terms of performance, and so the trade-off is worth it. These are issues you need to try to plan for.

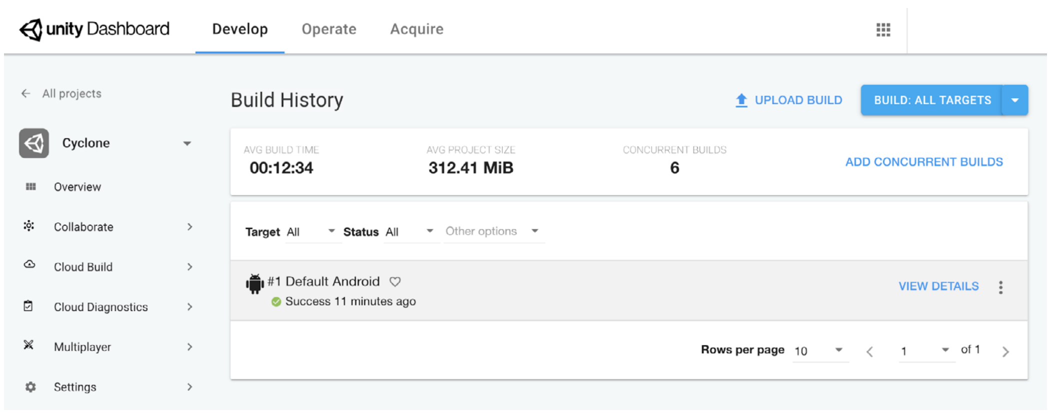
Put in place build and QA processes

It is very important to put in place a build and QA process. Building and testing locally is useful up to a point, but is time-consuming and error-prone. There are numerous possible solutions to consider (e.g., Jenkins is very popular). You may choose to configure your own build machine(s) dedicated for the purpose, or use one of the many cloud-based services to reduce the overhead and cost of maintaining your own machines. You may also consider the Cloud Build feature of Unity Teams. Take some time to evaluate and choose a solution that you feel fits your needs.

It's a good idea to plan for how features will be published to your release builds. Hand-in-hand with Version Control, consider how you want your development branches to be built and verified. Automated tests that run as part of the build process can catch many problems, but not everything. If you do run automated tests, it is worth collecting metrics so that you have a history of test performance across builds. This will help you spot regressions more quickly. You may also need to include some manual quality assurance and approval in your process. The build process can merge branches to your release build once they are verified.

Consider starting production from scratch

After prototyping your project, seriously consider starting your production phase from scratch. Decisions made during prototyping usually favor speed, and so it's highly likely your prototype project consists of many "hacks" and is not a solid foundation to start from.



Build History

AVG BUILD TIME 00:12:34 | AVG PROJECT SIZE 312.41 MiB | CONCURRENT BUILDS 6

Target All | Status All | Other options

#1 Default Android
Success 11 minutes ago

Rows per page 10 | 1 of 1

Create automated builds of your project using Unity Cloud Build

2. Development and workflow

Most developers want to spend their time being creative and productive, and not constantly battling problems with workflows. However, poor choices can lead to inefficiencies and mistakes by the team during development, affecting the quality of the final product. To mitigate this, find ways to streamline common tasks and make them more robust, which will minimize breakages that could block your team.

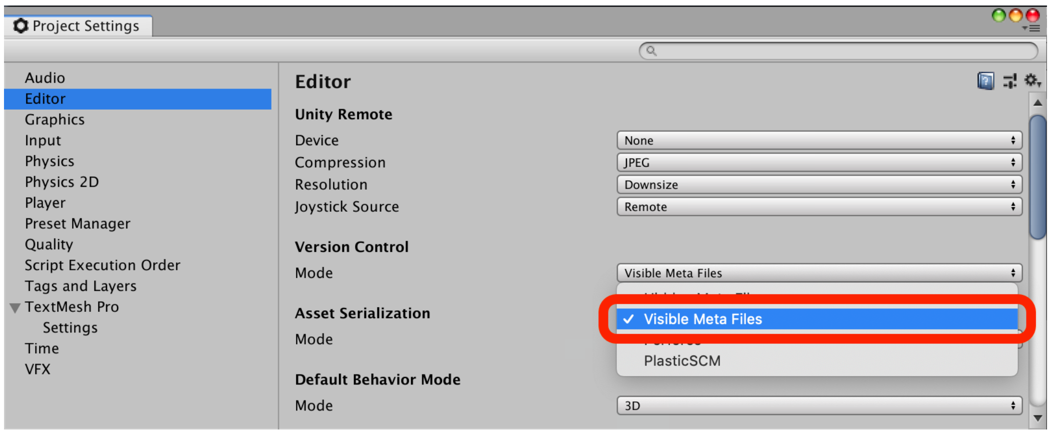
Automate repetitive manual tasks

Manual tasks that are repeated often are a prime candidate for automation (e.g., via custom scripts or Editor tools). A relatively small amount of time and effort can translate into a large amount of cumulatively saved time across the team during the life of the project. Automating tasks also removes the risk of user error.

In particular, ensure that your build process is fully automated and can be built entirely via a single action, either locally or on a Continuous Integration server.

Implement version control

All developers should be using version control of some kind, and Unity has built-in support for multiple solutions.



Editor

Unity Remote

Device: None
Compression: JPEG
Resolution: Downsize
Joystick Source: Remote

Version Control

Mode: Visible Meta Files

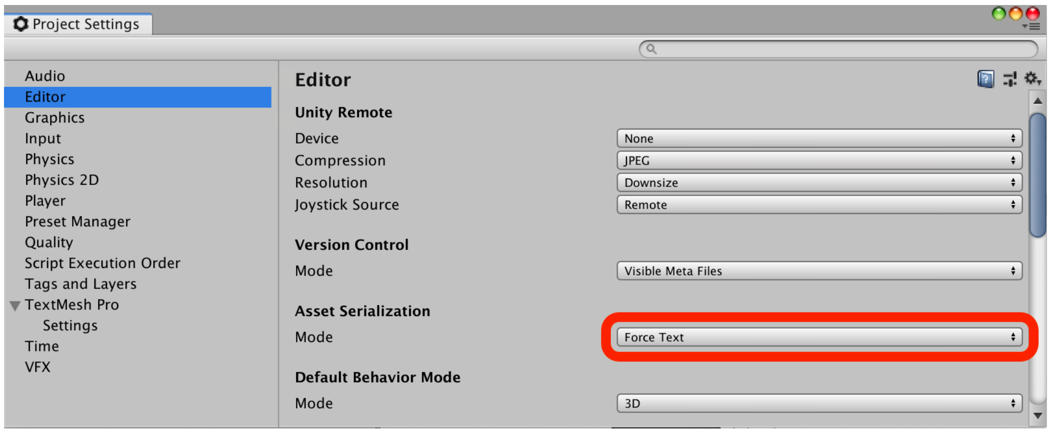
✓ Visible Meta Files
PlasticSCM

Asset Serialization

Mode

Default Behavior Mode

Mode: 3D



Editor

Unity Remote
Device: None
Compression: JPEG
Resolution: Downsize
Joystick Source: Remote

Version Control
Mode: Visible Meta Files

Asset Serialization
Mode: Force Text

Default Behavior Mode
Mode: 3D

Version Control settings within the Unity Editor

To start, ensure that your project's Editor Settings have Asset Serialization Mode set to Force Text, which should be the default.

Unity also has a built-in YAML (a human-readable, data-serialization language) tool specifically for merging scenes and Prefabs. Ensure that this is also set up. For more information, see SmartMerge here.

If your version-control solution supports commit hooks (e.g., Git) you can make use of its commit hooks to enforce certain standards. See this Unity Git Hooks page for more information.

You should keep all active development work off your main branch(es) so that you always have a solid working version of your project. Use branches and tags to manage milestones and releases.

Enforce a policy of good commit messages within the team. Clear, descriptive messages will help you track down problems later during development, while empty or meaningless messages just add noise.

Finally, good version control can help you quickly hunt down where a problem got introduced. Git, for example, has a bisect feature that allows you to mark a known good revision and a known bad revision, then you can use a "divide and conquer" approach to check out revisions in between to test and flag as good or bad. Spend some time learning which features your version-control system has and identify those that may be of use during development.

Take advantage of the Cache Server

Switching between target platforms within the Unity Editor, particularly on large projects, can be very slow, so we recommend you use the Unity Cache Server to help with this. You can run multiple cache servers, optionally using different ports, if you need to support multiple projects or versions of Unity.

The Cache Server also makes switching platforms much faster for local users, so be sure to take advantage of it.

Watch out for plug-ins

If your project contains a lot of plug-ins and third-party libraries, there's a good chance that unused assets within them are being built into your game because many plug-ins come with embedded test assets and scripts.

If you're using Asset Store assets, check which dependencies they pull into your project. For example, you may be surprised to find that you have several different JSON libraries.

Finally, strip out any resources from plug-ins that you don't need, including old assets and scripts that may remain from your prototyping phase.

Focus on collaboration with Scenes and Prefabs

It's important to think about how you want your development team to work together on content. Large, single Unity Scenes do not lend themselves well to collaboration. We recommend that you break your levels down into many smaller scenes so that artists and designers can collaborate better on a single level while minimizing the risk of conflicts. At runtime, your project can load scenes additively using the SceneManager.LoadSceneAsync() API passing the parameter mode = LoadSceneMode.Additive.

Unity 2018.3 and later include Improved Prefabs, which support the nesting of Prefabs. This also lets you break down Prefab content into smaller discrete items that can be worked on independently by different team members without risk of conflicts. Even so, there will be times when team members will need to access and work on the same asset. Agree on how you will handle this within the team. This may simply require a communication policy whereby team members alert each other via a Slack channel, email, etc. If your workflow supports it, you could also handle this through a file check-in/check-out mechanism in your version-control solution.

3. Profiling

Profile your project often

Don’t let issues build up in your project – try to profile often, not just late in your schedule or when your project starts exhibiting performance problems. Having a good feel for the typical “performance signature” of your project can help you spot new performance issues more easily. If you see a new glitch or spike, investigate it as soon as possible.

Here are some important profiling tips:

- Do not try to optimize your project based on assumptions – always optimize based on what you find during profiling.
- Use both Unity and platform/vendor-specific profiling tools to get the clearest picture of what is happening across your project.
- While profiling your project within the Unity Editor can be useful, always profile on the target platforms themselves.
- Consider automated testing to catch regressions for further investigation. As well, consider saving profiling captures for manual comparison via tools like Profile Analyzer.

Unity tools

Unity's own profiling tools typically require a Development build of your project, so while performance is not the same as a final non-Development build, it should still give a fairly good overview as long as you have selected a Release rather than a Debug build configuration.

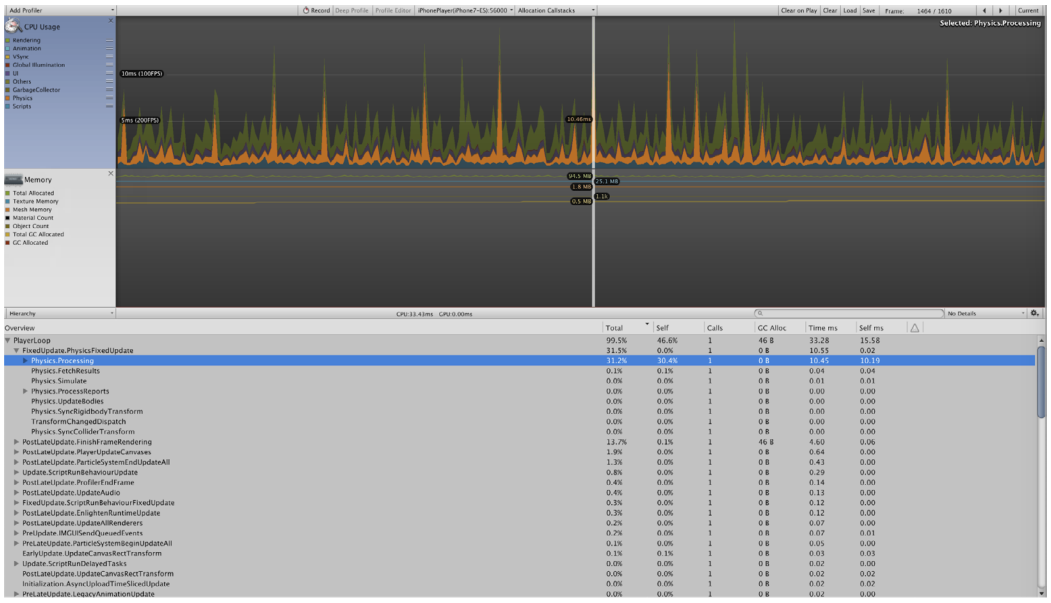
Inspect key project areas with the Unity Profiler

Use the Unity Profiler often to inspect key areas of your project.

Add meaningful Unity Profiler samplers to your scripts to give you more meaningful information without having to resort to Deep Profiling, which greatly impacts performance and only works in the Unity Editor as well as those platforms supporting Just-In-Time (JIT) Mono compilation (i.e., Windows, macOS, and Android).

Don't forget that in addition to being able to reorder the individual Profiler tool tracks (such as CPU, GPU, Physics, etc.), you can add and remove them as well. With Unity 2018.3 and later, removing those tracks, which are not currently needed, reduces CPU overhead in the runtime on the target platform. In previous Unity versions, all profiling data was collected by the runtime regardless, increasing Profiler overhead.

Within Profiler's Hierarchy view, sort by the Total column to identify the most expensive areas in terms of CPU cost. This will help you zero-in on areas that need investigation.

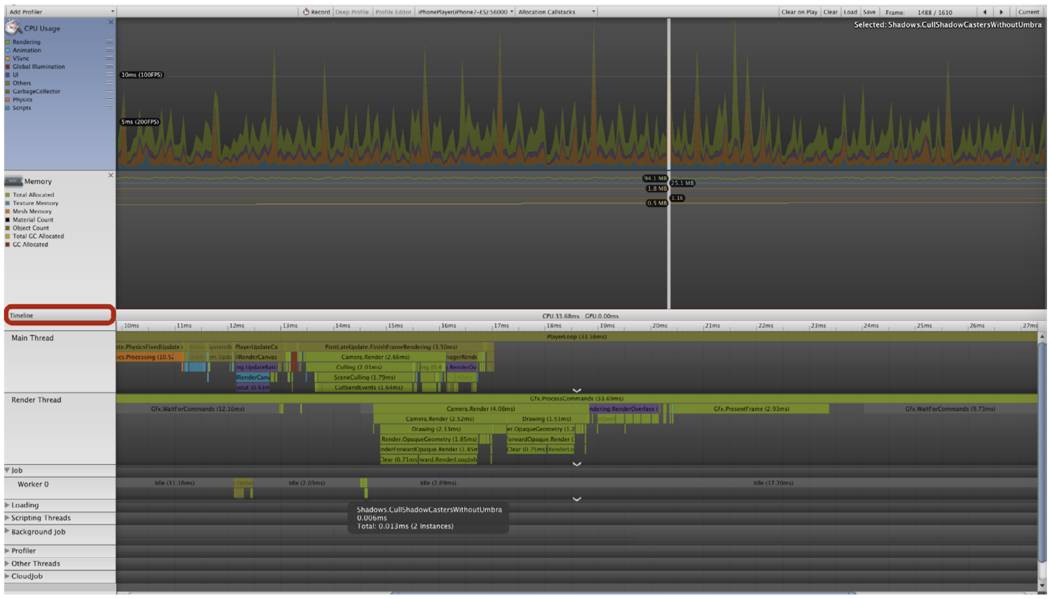


Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	99.5%	46.6%	1	46 B	33.28	15.58
FixedUpdate.PhysicsFixedUpdate	31.5%	0.0%	1	0 B	10.55	0.02
Physics.Processing	31.2%	30.4%	1	0 B	10.45	10.19
Physics.FetchResults	0.1%	0.1%	1	0 B	0.04	0.04
Physics.Simulate	0.0%	0.0%	1	0 B	0.01	0.01
Physics.ProcessReports	0.0%	0.0%	1	0 B	0.00	0.00
Physics.UpdateBodies	0.0%	0.0%	1	0 B	0.00	0.00
Physics.SyncRigidbodyTransform	0.0%	0.0%	1	0 B	0.00	0.00
TransformChangedDispatch	0.0%	0.0%	1	0 B	0.00	0.00
Physics.SyncColliderTransform	0.0%	0.0%	1	0 B	0.00	0.00
PostLateUpdate.FinishFrameRendering	13.7%	0.1%	1	46 B	4.60	0.06
PostLateUpdate.PlayerUpdateCanvases	1.9%	0.0%	1	0 B	0.64	0.00
PostLateUpdate.ParticleSystemEndUpdateAll	1.3%	0.0%	1	0 B	0.43	0.00
Update.ScriptRunBehaviourUpdate	0.8%	0.0%	1	0 B	0.29	0.00
PostLateUpdate.ProfilerEndFrame	0.4%	0.0%	1	0 B	0.14	0.00
PostLateUpdate.UpdateAudio	0.4%	0.0%	1	0 B	0.13	0.00
FixedUpdate.ScriptRunBehaviourFixedUpdate	0.3%	0.0%	1	0 B	0.12	0.00
PostLateUpdate.EnlightenRuntimeUpdate	0.3%	0.0%	1	0 B	0.12	0.00
PostLateUpdate.UpdateAllRenderers	0.2%	0.0%	1	0 B	0.07	0.00
PreUpdate.IMGUISendQueuedEvents	0.2%	0.0%	1	0 B	0.07	0.01
PreLateUpdate.ParticleSystemBeginUpdateAll	0.1%	0.0%	1	0 B	0.05	0.00
EarlyUpdate.UpdateCanvasRectTransform	0.1%	0.1%	1	0 B	0.03	0.03
Update.ScriptRunDelayedTasks	0.0%	0.0%	1	0 B	0.02	0.00
PostLateUpdate.UpdateCanvasRectTransform	0.0%	0.0%	1	0 B	0.02	0.02
Initialization.AsyncUploadTimeSlicedUpdate	0.0%	0.0%	1	0 B	0.02	0.02
PreLateUpdate.LegacyAnimationUpdate	0.0%	0.0%	1	0 B	0.02	0.00

Profiling the most expensive code functions with Unity Profiler Hierarchy view

Similarly, sort by the GC Alloc column to reveal areas that are generating managed allocations. Aim to reduce allocations as much as possible, especially those that occur regularly or even every frame, and focus on allocation spikes that may cause the managed heap to grow quickly and trigger garbage collections. Keep the managed heap size (reported as Mono under the Memory Profiler track) as low as possible because the more memory you use, the more fragmented it will become, leading to expensive garbage collections.

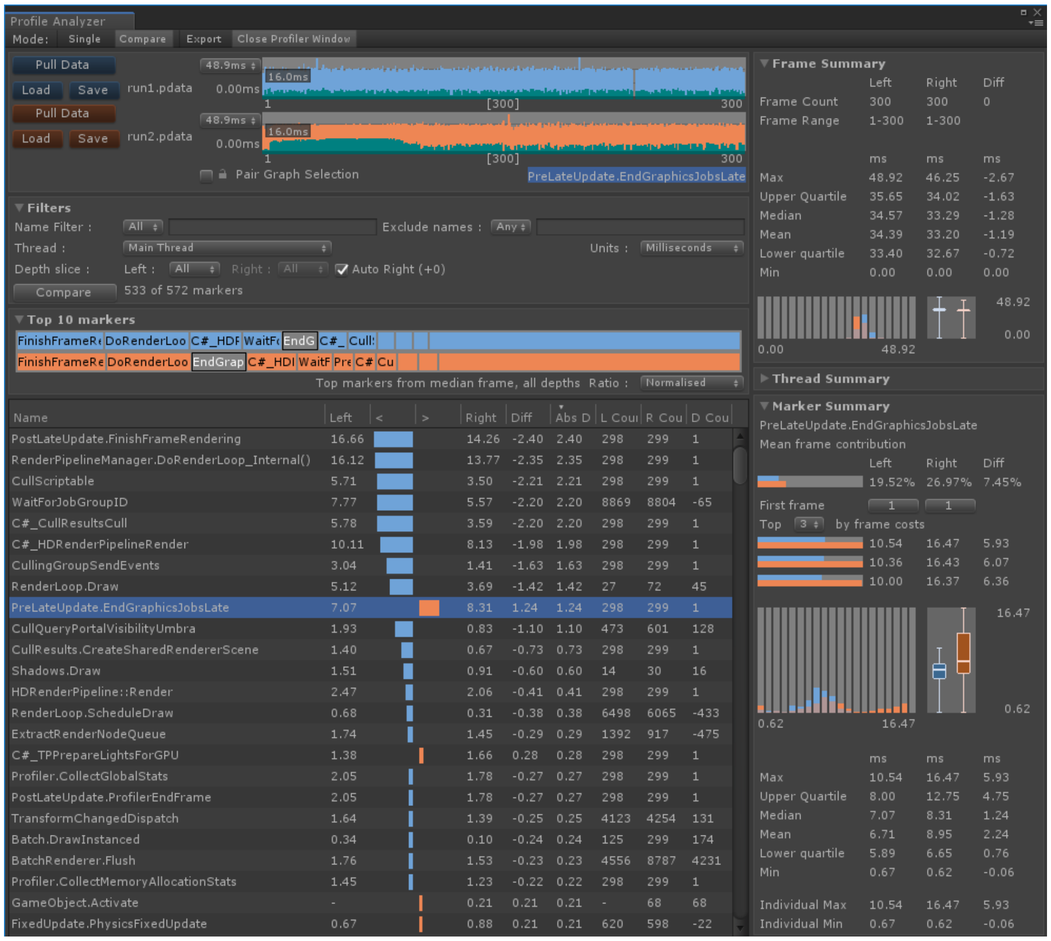
Unlike Hierarchy view, which focuses on just the Unity Main thread, Profiler's Timeline view gives you a valuable visual overview across multiple threads, including Unity Main and Rendering threads, Job System threads, and user threads (if appropriate Profiler samplers have been added).



Profiling across threads with Unity Profiler Timeline view

Put the Profile Analyzer to work on regression testing

The Profile Analyzer lets you import data from Unity Profiler – either what is currently captured or from a capture file previously saved to disk – and allows you to perform various analyses. In addition to seeing profiling markers broken down by mean, median, and peak costs, you can also compare sets of data. This is useful, for example, for regression testing where you compare data captured before and after changes.



Filters
Name Filter : All
Exclude names : Any
Thread : Main Thread
Units : Milliseconds
Depth slice : Left : All Right : All ✓ Auto Right (+0)
Compare 533 of 572 markers
Top markers from median frame, all depths Ratio : Normalised

Name	Left	<	>	Right	Diff	Abs D	L Cou	R Cou	D Cou
PostLateUpdate.FinishFrameRendering	16.66			14.26	-2.40	2.40	298	299	1
RenderPipelineManager.DoRenderLoop_Internal()	16.12			13.77	-2.35	2.35	298	299	1
CullScriptable	5.71			3.50	-2.21	2.21	298	299	1
WaitForJobGroupID	7.77			5.57	-2.20	2.20	8869	8804	-65
C#_CullResultsCull	5.78			3.59	-2.20	2.20	298	299	1
C#_HDRenderPipelineRender	10.11			8.13	-1.98	1.98	298	299	1
CullingGroupSendEvents	3.04			1.41	-1.63	1.63	298	299	1
RenderLoop.Draw	5.12			3.69	-1.42	1.42	27	72	45
PreLateUpdate.EndGraphicsJobsLate	7.07			8.31	1.24	1.24	298	299	1
CullQueryPortalVisibilityUmbra	1.93			0.83	-1.10	1.10	473	601	128
CullResults.CreateSharedRendererScene	1.40			0.67	-0.73	0.73	298	299	1
Shadows.Draw	1.51			0.91	-0.60	0.60	14	30	16
HDRenderPipeline::Render	2.47			2.06	-0.41	0.41	298	299	1
RenderLoop.ScheduleDraw	0.68			0.31	-0.38	0.38	6498	6065	-433
ExtractRenderNodeQueue	1.74			1.45	-0.29	0.29	1392	917	-475
C#_TPPrepareLightsForGPU	1.38			1.66	0.28	0.28	298	299	1
Profiler.CollectGlobalStats	2.05			1.78	-0.27	0.27	298	299	1
PostLateUpdate.ProfilerEndFrame	2.05			1.78	-0.27	0.27	298	299	1
TransformChangedDispatch	1.64			1.39	-0.25	0.25	4123	4254	131
Batch.DrawInstanced	0.34			0.10	-0.24	0.24	125	299	174
BatchRenderer.Flush	1.76			1.53	-0.23	0.23	4556	8787	4231
Profiler.CollectMemoryAllocationStats	1.45			1.23	-0.22	0.22	298	299	1
GameObject.Activate	-			0.21	0.21	0.21	-	68	68
FixedUpdate.PhysicsFixedUpdate	0.67			0.88	0.21	0.21	620	598	-22

Frame Summary

	Left	Right	Diff
Frame Count	300	300	0
Frame Range	1-300	1-300	
	ms	ms	ms
Max	48.92	46.25	-2.67
Upper Quartile	35.65	34.02	-1.63
Median	34.57	33.29	-1.28
Mean	34.39	33.20	-1.19
Lower quartile	33.40	32.67	-0.72
Min	0.00	0.00	0.00

Marker Summary
PreLateUpdate.EndGraphicsJobsLate
Mean frame contribution

	Left	Right	Diff
	19.52%	26.97%	7.45%
First frame	1	1	
Top 3 by frame costs			
	10.54	16.47	5.93
	10.36	16.43	6.07
	10.00	16.37	6.36
	ms	ms	ms
Max	10.54	16.47	5.93
Upper Quartile	8.00	12.75	4.75
Median	7.07	8.31	1.24
Mean	6.71	8.95	2.24
Lower quartile	5.89	6.65	0.76
Min	0.67	0.62	-0.06
Individual Max	10.54	16.47	5.93
Individual Min	0.67	0.62	-0.06

Comparing before-and-after Profiler data to check for improvements or regressions

Capture and visualize memory consumption

The Memory Profiler (available in the Package Manager via Preview packages) is a powerful tool for capturing and visualizing some (but not all) memory used by the application. Its Tree Map view is especially useful for immediately visualizing memory consumption of common asset types such as Textures and Meshes, as well as other managed types such as strings and project-specific types.

Check these areas of asset consumption for potential problems. For example, Textures that seem unusually large may indicate incorrect import settings or overly high resolutions. Duplicate assets may also be found here. It is possible for different assets to have the same name; however, multiple assets of the same name and the same size may be duplicates and should be investigated. Incorrect AssetBundle assignments is one cause of this.

Typically, but not always, projects tend to have a larger ratio of Texture memory to Mesh memory. For that reason, investigate if you see higher Mesh memory usage compared to Texture memory usage.

Examine rendering and batching flows

The Frame Debugger tool can be very useful for examining the flow of rendering and areas such as batching. This tool will tell you why a batch is broken, for example, which may allow you to optimize your content.

When you see how your frame is built it may reveal other problems, such as multiple renders of the same content (e.g., duplicate cameras have been observed in real-world projects) and rendering of content that is completely obscured (e.g., continuing to render a 3D scene behind a full-screen 2D UI).

Platform/vendor-specific tools

While Unity's profiling tools provide a lot of functionality, it is also worth investing time to learn to effectively use the native profiling tools for your target platform(s). For example:

Platform	Tool(s)
iOS and macOS	Xcode and Instruments
Android	Android Studio
Windows	VTune
Windows, Xbox One	PIX
Windows	NVidia NSight
PlayStation 4	Razor

These toolsets offer more powerful profiling options, such as sampling- and instrumentation-based CPU profiling, full native memory profiling, and GPU profiling, including shader debugging. You would typically use these tools on a non-Development build of your project.

You can measure performance across all CPU cores, threads, and functions, not just those with Unity Profiler markers.

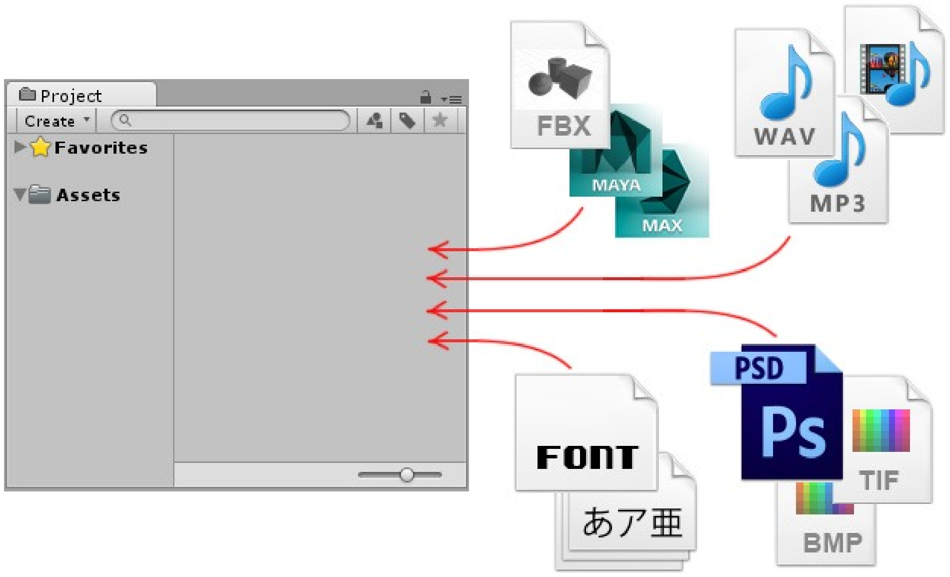
Memory profiling (e.g., via Instruments' Allocations and VM Tracker tools on iOS) can reveal large consumers of memory that don't appear in Unity's Memory Profiler, such as native allocations made by third-party plugins.

4. Assets

The asset pipeline is hugely important. Much of your project's memory footprint will be taken by assets such as textures, meshes, and sounds, and suboptimal settings can cost you dearly.

To avoid this, it's essential to set up a good flow of art content made to the right specifications. If possible, involve an experienced technical artist from the very beginning to help define this process.

To start, define clear guidelines on which formats and specifications you will use.



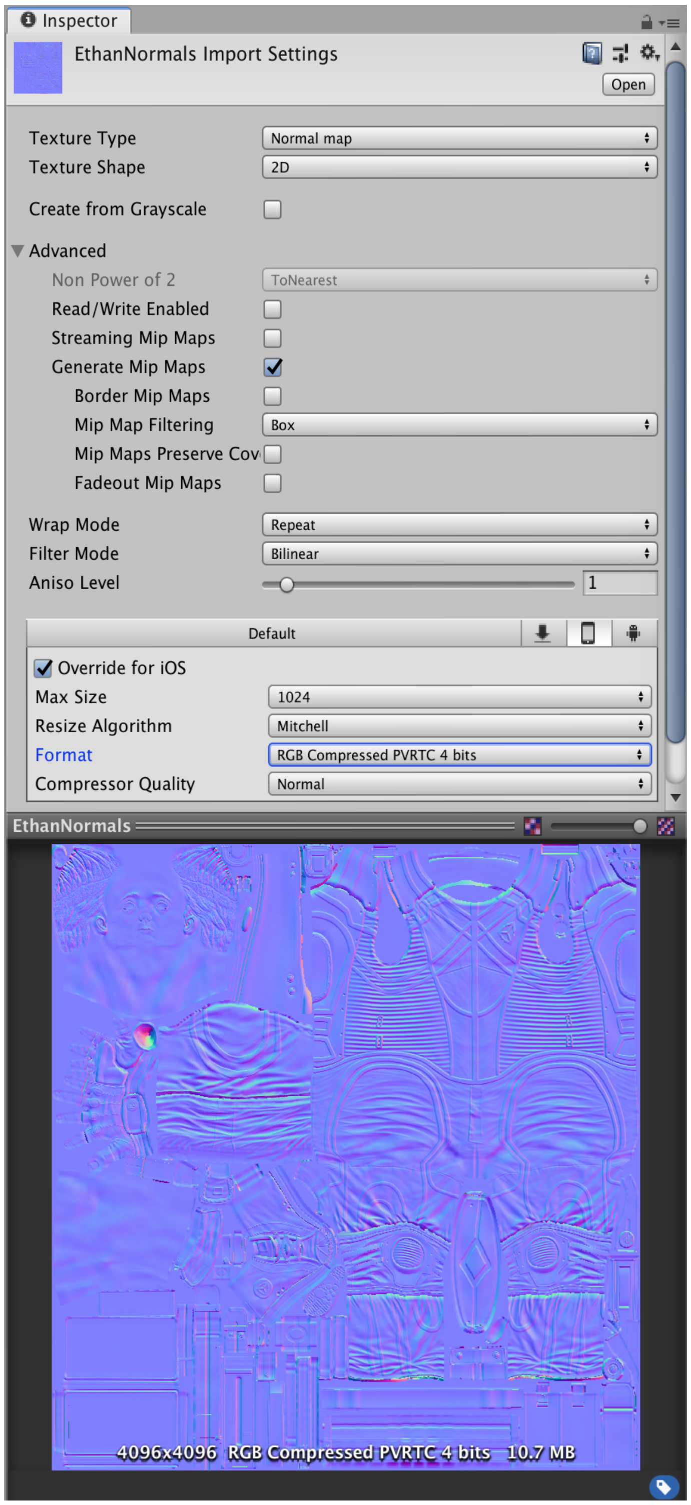
Unity can import many types of asset files

It is very important to have correct Import Settings across all assets in your project. Make sure that your team understands the settings for the asset types they're working on, and enforce rules to ensure that all your assets have consistent settings.

As well, don't simply use the Default settings for all platforms. Use the platform-specific override tabs to optimize assets (e.g., to set different maximum texture sizes or audio quality).

Consider setting up an automated way to apply Import Settings for new assets using the AssetPostprocessor API, as shown in this project.

See also: Art Asset best practice guide



Inspector

EthanNormals Import Settings

Open

Texture Type	Normal map
Texture Shape	2D
Create from Grayscale	☐
Advanced	
Non Power of 2	ToNearest
Read/Write Enabled	☐
Streaming Mip Maps	☐
Generate Mip Maps	✓
Border Mip Maps	☐
Mip Map Filtering	Box
Mip Maps Preserve Cov	☐
Fadeout Mip Maps	☐
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	1

Default

Override for iOS	✓
Max Size	1024
Resize Algorithm	Mitchell
Format	RGB Compressed PVRTC 4 bits
Compressor Quality	Normal

EthanNormals

4096x4096 RGB Compressed PVRTC 4 bits 10.7 MB

Set up texture imports correctly

Textures usually consume the most memory of all asset types, so ensure your Import Settings are correctly set up.

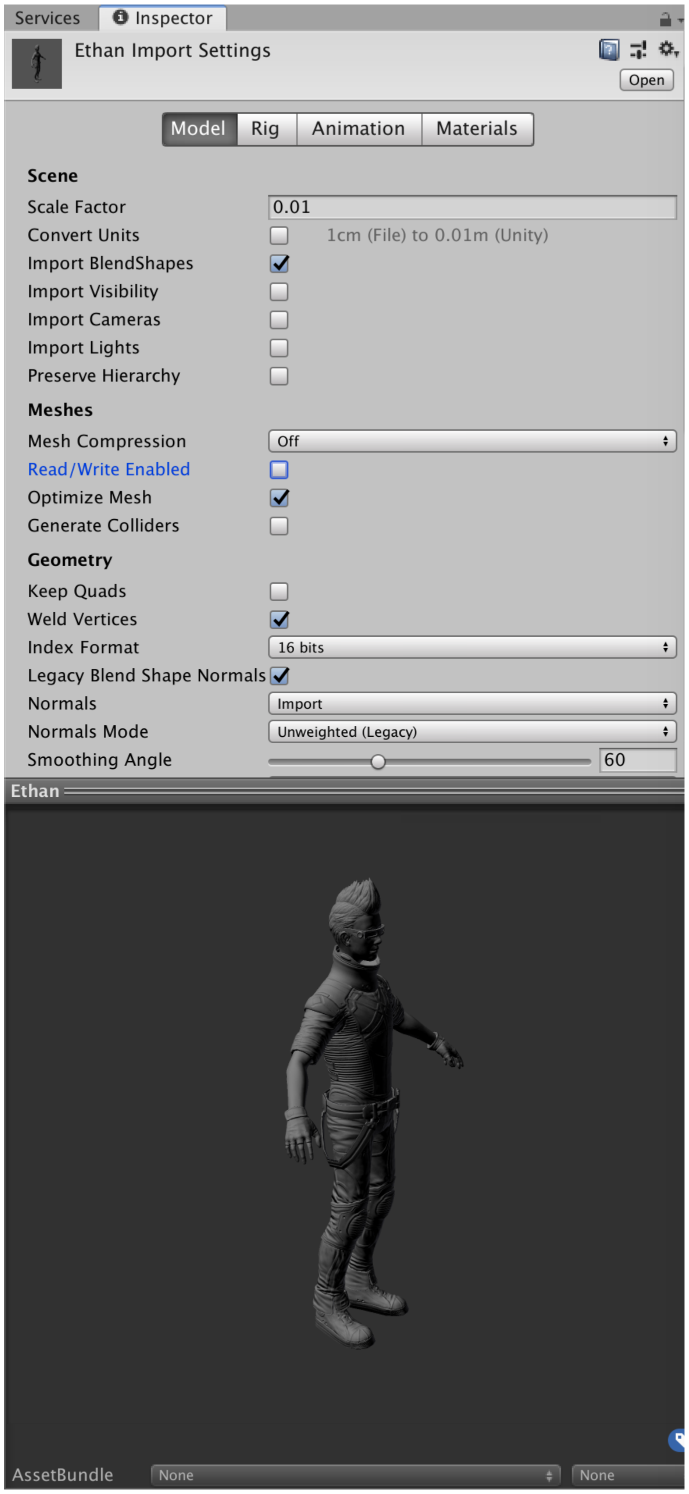
The Read/Write Enabled option should never be enabled unless you absolutely need to access the pixel data from scripts. This option keeps a copy of the pixel data in CPU-addressable memory as well as the copy uploaded to GPU-addressable memory, thus doubling the overall memory footprint. Fortunately, at runtime you can force a Texture asset to discard the CPU-addressable copy via the Texture2D.Apply() API, passing the parameter *makeNoLongerReadable=false*;

Only enable mipmaps if required. Typically this is the case for textures used in a 3D scene, but is usually not required for 2D. Leaving this option enabled when not required will increase the memory footprint of a texture by around one third.

Use compression when possible, although not all content retains sufficient visual fidelity when compressed, which is especially true of UI textures. Understand the pros and cons of the compression formats applicable to your target platforms and choose your formats wisely. For example, using ASTC rather than PVRT on iOS typically yields higher-fidelity results, but the format is not supported on lower-end devices that do not have the Apple A8 chip or later. Some developers choose to include multiple compressed versions of their assets, allowing them to use the highest-quality version supported by a particular device.

Use sprite atlases as much as possible in order to group sprites together, which improves batching and reduces the number of draw calls. To do this, use Unity SpritePacker or Unity SpriteAtlas, depending on which version of Unity you are using, or one of many third-party tools such as Texture Packer.

Setting up Import Settings for a Texture asset



Services | Inspector

Ethan Import Settings

Open

Model | Rig | Animation | Materials

Scene

Scale Factor	0.01
Convert Units	☐ 1cm (File) to 0.01m (Unity)
Import BlendShapes	✓
Import Visibility	☐
Import Cameras	☐
Import Lights	☐
Preserve Hierarchy	☐

Meshes

Mesh Compression	Off
Read/Write Enabled	☐
Optimize Mesh	✓
Generate Colliders	☐

Geometry

Keep Quads	☐
Weld Vertices	✓
Index Format	16 bits
Legacy Blend Shape Normals	✓
Normals	Import
Normals Mode	Unweighted (Legacy)
Smoothing Angle	60

Ethan

AssetBundle None None

Reclaim memory by disabling this Mesh option

Again, only enable the Read/Write Enabled option if you need to read mesh data from scripts. This option was enabled by default in all versions of Unity until Unity 2019.3 (where it is off by default), and is commonly left as-is. You can reclaim memory in many projects by disabling this option.

If your project uses different model source files (e.g., FBX) to import visible meshes and animations, ensure that the meshes are stripped out of those source files. Otherwise, the mesh data will still be built into the final output, wasting memory.

Setting up Import Settings for a Model asset

Select the best compression format for Audio Clips

For most Audio Clips, a Load Type setting of Compressed In Memory is a good default. Make sure that you select the appropriate compression format for each platform. Consoles have their own custom formats, and Vorbis is a good choice for everything else. On mobile, Unity does not use hardware decompression, therefore there is no advantage to selecting MP3 for iOS.

Choose suitable sample rates for the target platform. For example, short sound effects on mobile devices should be 22050 Hz at most, and many can be much lower than this with a negligible effect on the final quality.

Ensure that long music or ambient sounds have their Load Type set to Streaming, otherwise the entire asset will be loaded into memory at once.

Setting Load Type to Decompress On Load will incur CPU cost and memory by decompressing a sound into raw 16-bit PCM audio data. This is usually only desirable for short sounds like footsteps or sword clashes, which often have multiple concurrent instances.

Use original (pristine) uncompressed WAV files as your source assets where possible. If you use any compressed format (such as MP3 or Vorbis), then Unity will decompress it and recompress it during build time, resulting in two lossy passes, which degrades the final quality.

If you plan to place sounds within 3D space in your project, you should either author them as mono (single channel) or enable the Force To Mono setting. That's because a multi-channel sound used positionally at runtime will be flattened to a mono source in real-time, thus increasing CPU cost and wasting memory.

Use AssetBundles, not Resource folders

Use AssetBundles rather than placing assets inside Resource folders. Assets inside Resource folders get built into a sort of internal AssetBundle, and the header information for this is loaded during startup. As a result, having a lot of assets stored this way can lengthen startup time.

To avoid duplicating assets, explicitly assign all assets to AssetBundles, especially those that have dependencies. For example, imagine two material assets that use the same texture. If the texture is not assigned to an AssetBundle and the two materials are assigned to separate AssetBundles, then the texture will be duplicated into each of those AssetBundles alongside the referencing material. Assigning the texture explicitly to an AssetBundle prevents this.

You can use the AssetBundles Browser tool to set up and track assets and dependencies across AssetBundles, and the AssetBundle Analyzer tool to highlight duplicate assets as well as those with potentially suboptimal settings.

Plan your strategy for building content into AssetBundles, as there is no real one-size-fits-all solution. Many people group AssetBundles by logical content; for example, in a racing game, you could put all assets required for each car type into its own unique AssetBundle.

However, for platforms that support patching, take care with this strategy. Due to the compression algorithm used, a relatively small change to one or more assets can result in much of the binary data of an AssetBundle changing, which means that an efficient patch delta cannot be generated. Therefore, this can be a huge problem if you're using a small number of large AssetBundles. For patching, use many smaller AssetBundles rather than just a few larger ones.

5. Programming and code architecture

By following these best practices and making smart architectural decisions, you will ensure higher team productivity and better user experiences once your game launches

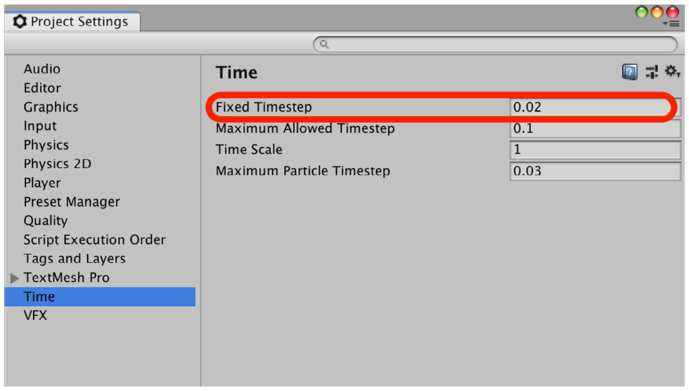
See also: How to have a better scripting experience

Avoid abstract code

Overly engineered and abstract code can be very hard to follow, especially for new hires on your team, and it generally makes it hard to analyze and discuss changes. This type of code will also generate more code, lead to longer build times (including IL2CPP builds, which will have more IL to transpile), and the final code may be less performant.

Understand the Unity Player Loop

Ensure that you have a good enough understanding of the Unity frame (or player) loop. For example, it is very important to know when Awake, OnEnable, Update and other methods are called. You can find more information in Unity's documentation.



Fixed Timestep	0.02
Maximum Allowed Timestep	0.1
Time Scale	1
Maximum Particle Timestep	0.03

The default Fixed Timestep setting is often a cause of extra CPU processing

The difference between Update and FixedUpdate is especially important. FixedUpdate methods are controlled by the project's Fixed Timestep value, which by default is 0.02 (50 Hz). This means that Unity will ensure that FixedUpdate methods are called 50 times per second, which may mean multiple calls in a single frame. If your frame rate drops, this problem worsens since the number of FixedUpdate calls will increase. This can result in a cycle known colloquially as the "spiral of death," because it sometimes exhibits severe glitches.

Physics system updates are part of the FixedUpdate phase, so games with a lot of physics content can suffer quite badly here. Some projects also run various game systems via FixedUpdate, so be careful to check for, and avoid, these kinds of performance issues. For these reasons, we highly recommend you set your Fixed Timestep value to closely match your target frame rate.

Choose the right frame rate

Choose an appropriate target frame rate. Mobile projects, for example, commonly need to balance fluid frame rates against battery life and thermal throttling. Running a game at 60 FPS with most of the frame time occupied with CPU and/or GPU load will lead to shorter battery life and faster throttling of CPU and GPU clock speeds by the device itself. For many projects, aiming for 30 FPS is a perfect compromise. Consider also dynamically adjusting frame rate during runtime via the Application.targetFrameRate property.

The new On-Demand Rendering feature, available as of the Unity 2019.3 beta, allows you to reduce the frequency of rendering without affecting other systems such as the Input system.

Often, the frame rate is not taken into account in scripting logic or animation. Don't assume a constant value for your updates, but instead use Time.deltaTime in Update methods and Time.fixedDeltaTime in FixedUpdate methods.

Avoid synchronous loading

Performance spikes are common when loading scenes or assets, often because the loads are performed synchronously on the Unity Main thread. For that reason, design your title to be robust using asynchronous loading to minimize these spikes. Use the AssetBundle.LoadFromFileAsync() and AssetBundle.LoadAssetAsync() APIs instead of AssetBundle.LoadFromFile() and AssetBundle.LoadAsset().

Setting up your project to be asynchronous will also help in other ways. Fluid user interaction will always be maintained. Server authentications and exchanges can easily happen in parallel with scene and asset loading, helping to reduce overall startup and loading times. A fully asynchronous design also means it will be easier for you to migrate to the new Addressable Asset System in the future.

Use pools of pre-allocated objects

Where objects are frequently instantiated and destroyed (bullets or NPCs are good examples), use pools of pre-allocated objects where the objects are recycled and reused instead. While there may be some CPU cost in resetting specific components on these objects for each reuse, it will be cheaper than complete instantiation. This approach also greatly reduces the number of managed allocations in your project.

Reduce usage of standard behavior methods as much as possible

All custom behaviors inherit from an abstract class, which defines methods for Update(), Awake(), Start(), and others. If a particular behavior does not require one of these methods (a comprehensive list can be found in the Unity documentation), then remove it rather than leave it empty, otherwise the empty method(s) will still be called by Unity. These methods incur a small amount of CPU overhead due to the cost of calling managed code (C#) from native code (C++).

This can be particularly problematic for Update() methods. If you have a lot of objects with Update() methods, this CPU cost can rise and become non-trivial. Consider the “manager pattern,” where one or more manager classes implement an Update() method but are then responsible for updating all of the individual objects. This greatly reduces the number of transitions between native and managed code. See this blog post for details.

As well, decide whether different systems in your project need to be updated each frame. Systems can often run at lower frequencies and so can be called in turn – a simple example would be two systems that are updated on alternate frames.

We also recommend time-slicing, where a system handles many items but spreads the load out over several frames, processing only a certain number of items per frame. This also helps to keep peak CPU load down.

A more advanced form of this is where you implement a comprehensive “budgeted update manager,” where each of the project’s systems is assigned a maximum per-frame time budget, then each system implements a manager based on a standard interface that performs as much work as it can within its allotted time. This approach can really help manage peak CPU load over the project’s lifetime. Systems designed to follow this pattern can be more flexible.

Be sure to cache expensive API results

Cache data as much as possible. Commonly seen API calls such as GameObject.Find(), GameObject.GetComponent(), and accessing Camera.main can be expensive, so avoid calling them in Update() methods. Instead, call them in Start() and cache the results.

Avoid string operations during runtime

Avoid lots of string operations such as concatenation during runtime. These can generate a lot of managed allocations, leading to garbage-collection spikes. Only regenerate strings (e.g., player scores) when they actually change, rather than on every update. Also, you can use the StringBuilder class to help reduce the number of allocations considerably.

Avoid unintended debug logging

Unintended debug logging can often cause spikes in a project. APIs like Debug.Log() will continue to log even in non-Development builds, which is often a surprise to developers. To prevent this, consider wrapping calls to APIs like Debug.Log() in your own class, using the Conditional attribute on the methods like this: **[Conditional("ENABLE_LOGS")]**. If the define used for the Conditional attribute is not present, the method and all call sites will be stripped out.

Don't use LINQ queries in critical paths

While LINQ queries can be very attractive for their power and ease-of-use, avoid using them in critical paths (i.e., regular updates), as they can generate a lot of managed allocations and be expensive in terms of CPU cost. If you must use them, be sensible and limit them to occasional situations such as level initialization, as long as they don't contribute to unnecessarily large CPU spikes.

Use non-allocating APIs

Unity has some APIs that generate managed allocations, such as Component.GetComponents(). APIs that return an array like this are allocating internally. Sometimes there are non-allocating alternatives, which should always be used instead. Many of the Physics APIs have newer non-allocating alternatives; for example, use Physics.RaycastNonAlloc() rather than Physics.RaycastAll().

Avoid static data parsing

Projects often process data stored in readable formats such as JSON or XML. While this is frequently in response to data downloaded from a server, it is also common to do so with embedded static data. This kind of parsing can be slow and typically generates a lot of managed allocations. Instead, for static data built into the title, use ScriptableObjects with custom Editor tools.

See also: 3 cool ways to architect your game with Scriptable Objects

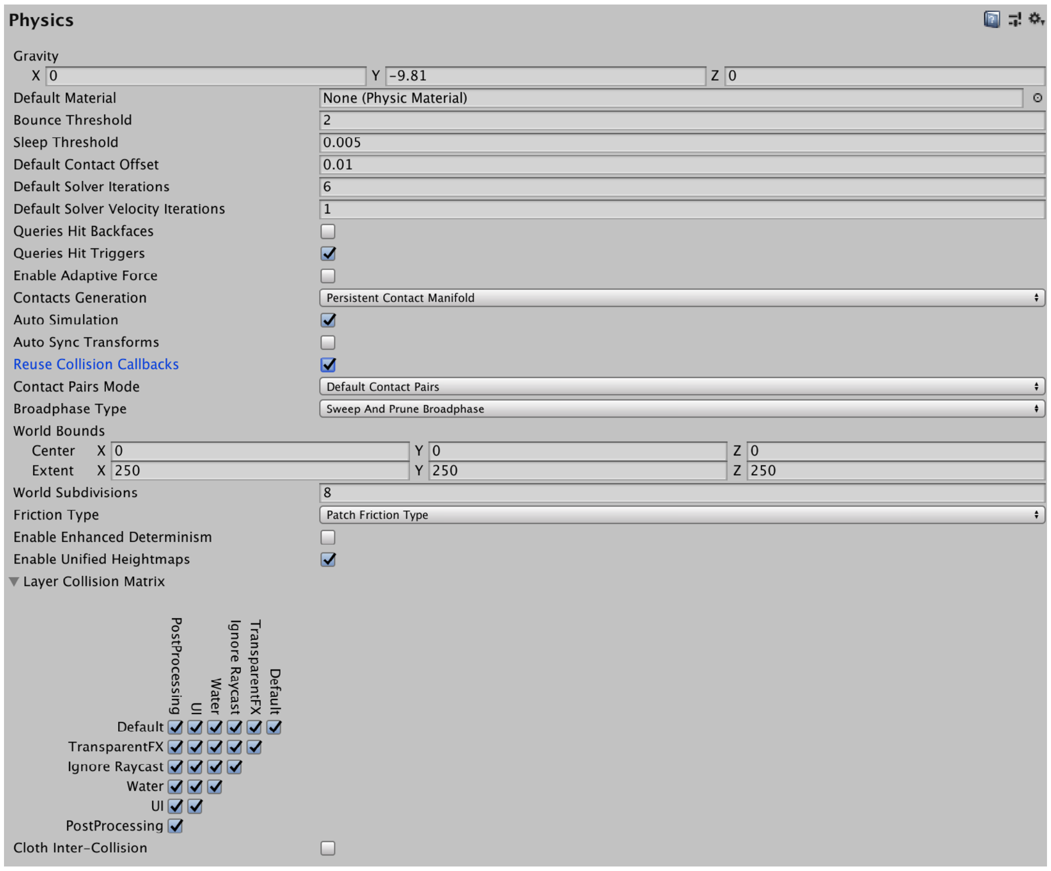
6. Physics

Due primarily to potential multiple executions per frame during FixedUpdate, Physics is often seen to be overly expensive in titles. However, there are other issues that you should also be aware of that may impact performance.

Enable the Prebake Collision Meshes option in Player Settings, where possible, to generate the runtime physics mesh data at build time. Otherwise, this data is generated at runtime when an asset is loaded, and content with many physics meshes can incur a high CPU cost on Unity's Main thread during generation.

Mesh colliders can be expensive, so substitute more complex mesh colliders with one or more simpler primitive types, as required, to approximate the original shape.

Settings



Physics

Gravity

X	0	Y	-9.81	Z	0

Default Material	None (Physic Material)
Bounce Threshold	2
Sleep Threshold	0.005
Default Contact Offset	0.01
Default Solver Iterations	6
Default Solver Velocity Iterations	1
Queries Hit Backfaces	☐
Queries Hit Triggers	✓
Enable Adaptive Force	☐
Contacts Generation	Persistent Contact Manifold
Auto Simulation	✓
Auto Sync Transforms	☐
Reuse Collision Callbacks	✓
Contact Pairs Mode	Default Contact Pairs
Broadphase Type	Sweep And Prune Broadphase

World Bounds

Center	X	0	Y	0	Z	0
Extent	X	250	Y	250	Z	250

World Subdivisions	8
Friction Type	Patch Friction Type
Enable Enhanced Determinism	☐
Enable Unified Heightmaps	✓

Layer Collision Matrix

	PostProcessing	UI	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓		
Water	✓	✓	✓			
UI	✓	✓				
PostProcessing	✓					

Cloth Inter-Collision	☐

Configuring the Physics settings within the Unity Editor

Consider disabling the Auto Sync Transforms option, which was enabled by default after its introduction in Unity 2017, in order to preserve backward compatibility. It ensures that any modification of any transform will be automatically and immediately synchronized to its underlying Physics object. However, in many cases it is not essential that this be done until the point at which the Physics simulation is stepped, yet may take noticeable CPU time during a frame. Disabling the option will defer synchronization to a later point but may well save CPU time overall.

If you are using collision callbacks, enable the Reuse Collision Callbacks option. This will avoid managed allocations on each callback by reusing a single internal Collision object during callbacks, rather than creating a new one for each callback. As well, when using callbacks, be mindful of doing complex work during them, as this can really add up in heavy scenes with a lot of collision events. Adding Unity Profiler markers to your callbacks will make them visible and easier to track down should they become a performance problem.

Finally, ensure that your Layer Collision Matrix is optimized by checking only the desired combinations of layers.

7. Animation

Animation is often a surprisingly expensive feature in projects as developers tend to use Animators excessively.

Animators are primarily intended for humanoid characters, but are often used to animate single values (e.g., the alpha channel of a UI element). Although Animators are convenient to use for their state machine flow, this is a relatively inefficient use case. Internal benchmarks show that on a device such as a low-end iPhone 4S, the performance of Animators beats that of Legacy Animation only when around 400 curves are being animated.

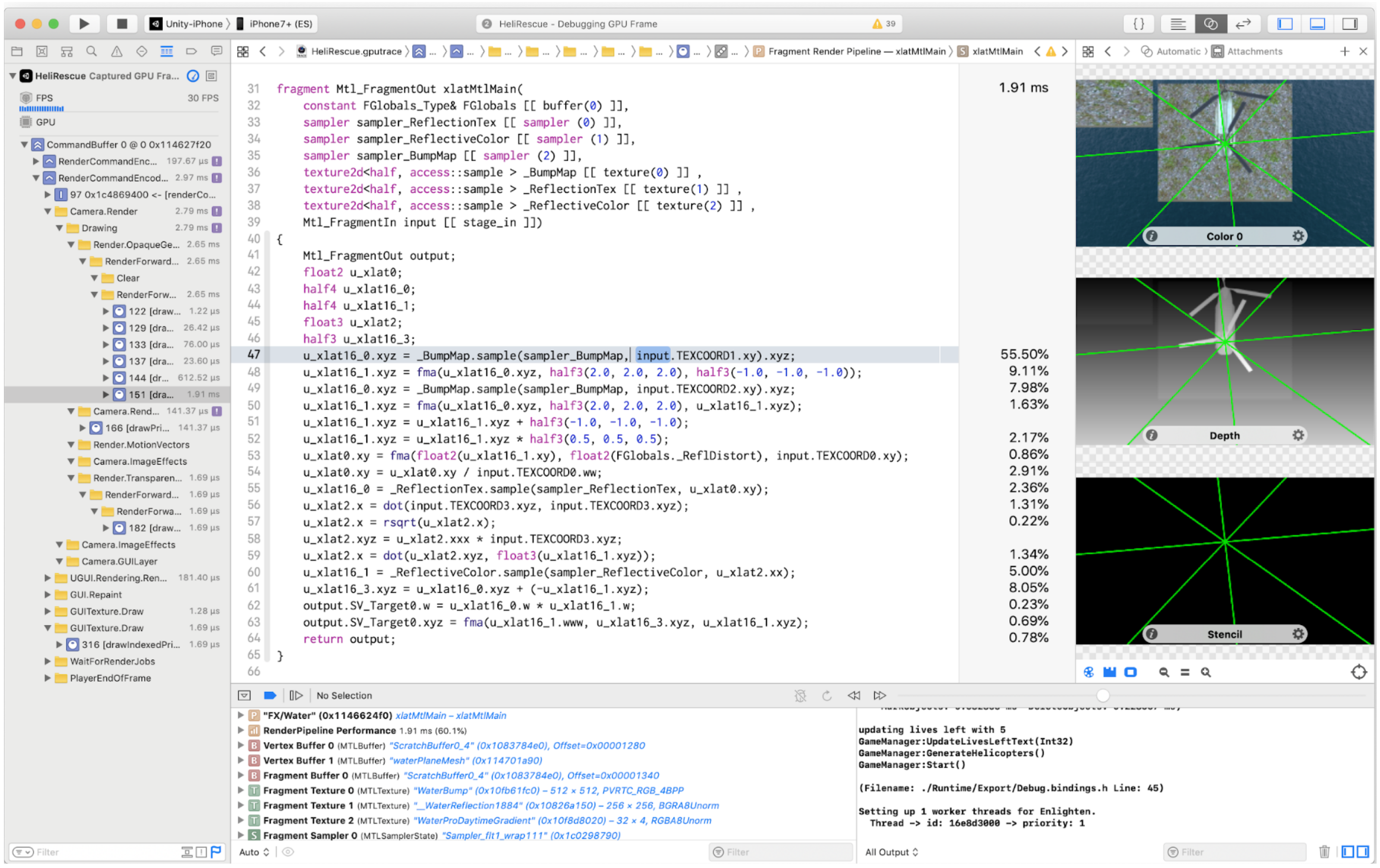
For simpler use cases (such as pulsing an alpha value or size), consider more lightweight implementations such as writing your own utility scripts or use a third-party library like DOTween.

Finally, be careful if you are manually updating Animators because they normally use Unity's Job System to process in parallel but manual updates force them to process on the Main thread.

8. GPU performance

We recommend doing GPU profiling to reveal how much of your GPU performance is spent on vertex, fragment, and compute shaders. This will let you investigate the most expensive draw calls and identify the most expensive shaders, potentially providing significant optimization gains.

See also:
[Optimizing graphics performance](#)
[Modeling characters for optimized performance](#)
[Shader profiling and optimization tips](#)



```
fragment Mtl_FragmentOut xlatMtlMain(                                                         1.91 ms
    constant FGlobals_Type& FGlobals [[ buffer(0) ]],
    sampler sampler_ReflectionTex [[ sampler (0) ]],
    sampler sampler_ReflectiveColor [[ sampler (1) ]],
    sampler sampler_BumpMap [[ sampler (2) ]],
    texture2d<half, access::sample > _BumpMap [[ texture(0) ]] ,
    texture2d<half, access::sample > _ReflectionTex [[ texture(1) ]] ,
    texture2d<half, access::sample > _ReflectiveColor [[ texture(2) ]] ,
    Mtl_FragmentIn input [[ stage_in ]])
{
    Mtl_FragmentOut output;
    float2 u_xlat0;
    half4 u_xlat16_0;
    half4 u_xlat16_1;
    float3 u_xlat2;
    half3 u_xlat16_3;
    u_xlat16_0.xyz = _BumpMap.sample(sampler_BumpMap, input.TEXCOORD1.xy).xyz;                 55.50%
    u_xlat16_1.xyz = fma(u_xlat16_0.xyz, half3(2.0, 2.0, 2.0), half3(-1.0, -1.0, -1.0));        9.11%
    u_xlat16_0.xyz = _BumpMap.sample(sampler_BumpMap, input.TEXCOORD2.xy).xyz;                  7.98%
    u_xlat16_1.xyz = fma(u_xlat16_0.xyz, half3(2.0, 2.0, 2.0), u_xlat16_1.xyz);                 1.63%
    u_xlat16_1.xyz = u_xlat16_1.xyz + half3(-1.0, -1.0, -1.0);
    u_xlat16_1.xyz = u_xlat16_1.xyz * half3(0.5, 0.5, 0.5);                                      2.17%
    u_xlat0.xy = fma(float2(u_xlat16_1.xy), float2(FGlobals._ReflDistort), input.TEXCOORD0.xy);  0.86%
    u_xlat0.xy = u_xlat0.xy / input.TEXCOORD0.ww;                                                2.91%
    u_xlat16_0 = _ReflectionTex.sample(sampler_ReflectionTex, u_xlat0.xy);                       2.36%
    u_xlat2.x = dot(input.TEXCOORD3.xyz, input.TEXCOORD3.xyz);                                   1.31%
    u_xlat2.x = rsqrt(u_xlat2.x);                                                                0.22%
    u_xlat2.xyz = u_xlat2.xxx * input.TEXCOORD3.xyz;
    u_xlat2.x = dot(u_xlat2.xyz, float3(u_xlat16_1.xyz));                                        1.34%
    u_xlat16_1 = _ReflectiveColor.sample(sampler_ReflectiveColor, u_xlat2.xx);                   5.00%
    u_xlat16_3.xyz = u_xlat16_0.xyz + (-u_xlat16_1.xyz);                                         8.05%
    output.SV_Target0.w = u_xlat16_0.w * u_xlat16_1.w;                                           0.23%
    output.SV_Target0.xyz = fma(u_xlat16_1.www, u_xlat16_3.xyz, u_xlat16_1.xyz);                 0.69%
    return output;                                                                               0.78%
}
```

```
"FX/Water" (0x1146624f0) xlatMtlMain – xlatMtlMain
RenderPipeline Performance 1.91 ms (60.1%)
Vertex Buffer 0 (MTLBuffer) "ScratchBuffer0_4" (0x1083784e0), Offset=0x00001280
Vertex Buffer 1 (MTLBuffer) "waterPlaneMesh" (0x114701a90)
Fragment Buffer 0 (MTLBuffer) "ScratchBuffer0_4" (0x1083784e0), Offset=0x00001340
Fragment Texture 0 (MTLTexture) "WaterBump" (0x10fb61fc0) – 512 × 512, PVRTC_RGB_4BPP
Fragment Texture 1 (MTLTexture) "_WaterReflection1884" (0x10826a150) – 256 × 256, BGRA8Unorm
Fragment Texture 2 (MTLTexture) "WaterProDaytimeGradient" (0x10f8d8020) – 32 × 4, RGBA8Unorm
Fragment Sampler 0 (MTLSamplerState) "Sampler_flt1_wrap111" (0x1c0298790)
```

```
updating lives left with 5
GameManager:UpdateLivesLeftText(Int32)
GameManager:GenerateHelicopters()
GameManager:Start()

(Filename: ./Runtime/Export/Debug.bindings.h Line: 45)

Setting up 1 worker threads for Enlighten.
  Thread -> id: 16e8d3000 -> priority: 1
```

Shader debugging and profiling on iOS using Xcode's GPU Capture tool

Pay attention to overdraw and alpha blending

Mobile platforms in particular are greatly impacted by alpha blending and overdraw. It is not uncommon to find a significant amount of GPU rendering time taken by large, barely visible overlays or effects with several layers of alpha-blended sprites containing a lot of zero-alpha pixels.

As well, avoid drawing unnecessary transparent images and, for cases where an image has large, fully transparent areas (e.g., a full-screen vignette overlay), consider making a custom mesh to avoid rendering those zero-alpha areas.

Keep shaders simple, with few variations

On mobile, try to keep shaders as simple as possible. Rather than using the Standard shader, use custom shaders, which you can make as lightweight as possible. Use simplified versions or even disable effects for lower-end target devices.

Try to keep the number of shader variations as low as possible, as this can impact performance and also have a dramatic effect on runtime memory usage.

Don't rely on too many Camera components

Avoid relying on more Unity Camera components than you really need to achieve your rendering. For example, it is not uncommon to find projects that use several cameras to build up UI layers. Each Camera component incurs overhead whether it does any meaningful work or not. On more powerful target platforms this might be negligible, but on lower-end or mobile platforms this can be up to 1 ms of CPU time each.

Consider static vs dynamic batching

Enable Static Batching on environment meshes that share the same materials. This allows Unity to merge them in such a way as to greatly reduce draw calls and rendering state changes, while still benefiting from object culling.

Profile to find if Dynamic Batching is a win for your project, which is not always the case. Objects must be "similar" and within fairly strict and relatively simple criteria in order to be dynamically batched. Unity's Frame Debugger will help you see why certain objects were not batched.

Don't forget forward rendering and LOD

Avoid too many dynamic lights when you use forward rendering. Every dynamic light adds a new render pass for every illuminated object.

Also, use Level of Detail (LOD) where possible. As objects move into the distance, use simpler meshes with simpler materials and shaders to significantly help GPU performance.

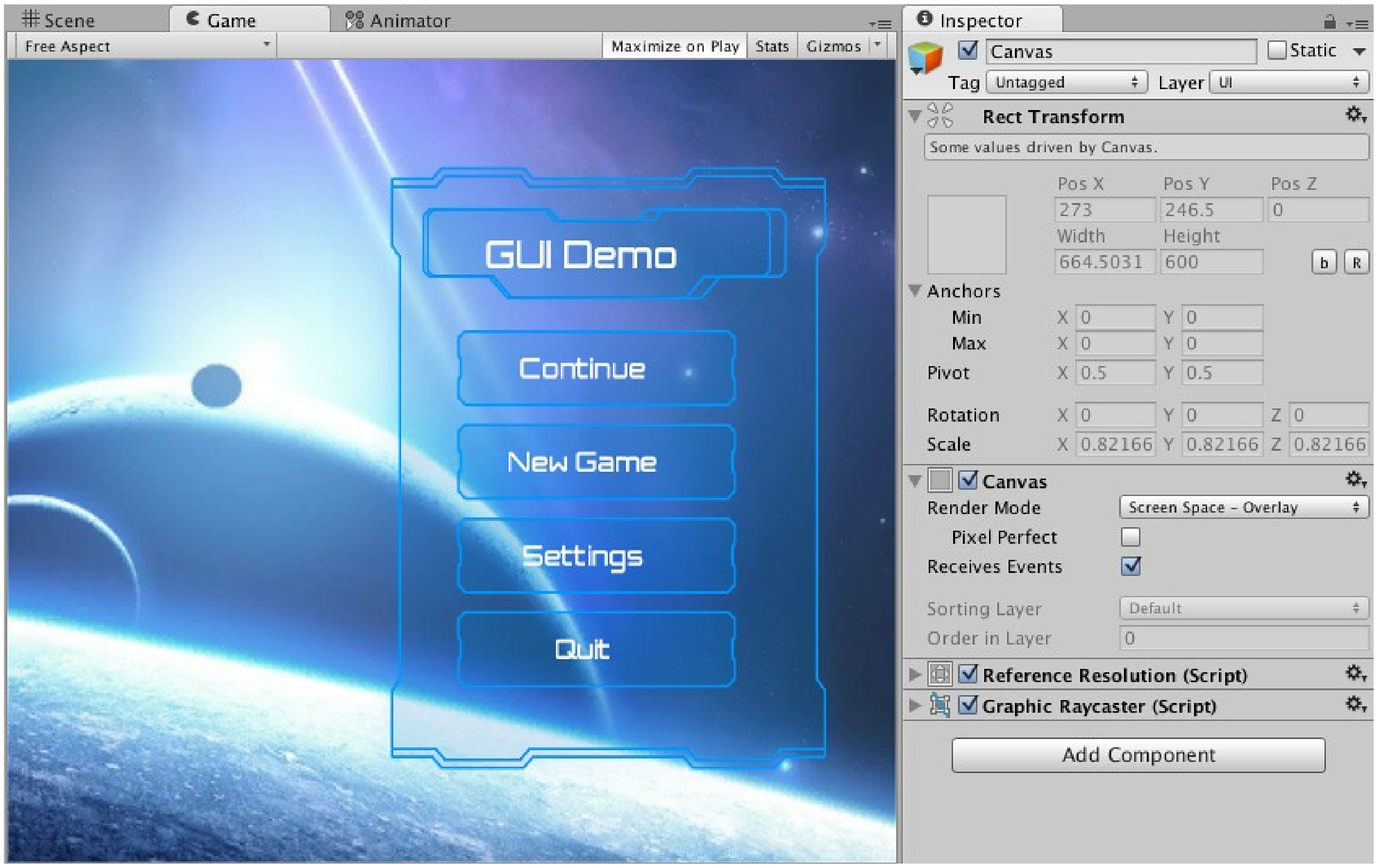
9. User interface (UI)

Unity UI (also known as UGUI) is often a source of performance issues in projects.
See Unity UI Optimization Tips for details.

Consider multiple resolutions and aspect ratios

Unity UI makes it easy to build UIs that can adjust positions and scale to account for different screen resolutions and aspect ratios. However, sometimes a single layout/design does not work well across all devices, so it might be better to create different versions of a UI (or parts of it) to provide the best experience on different devices.

Always extensively test your UI across a wide range of supported devices to ensure that the user experience is good and consistent across them all.



```
Canvas                          Static
Tag Untagged    Layer UI
Rect Transform
Some values driven by Canvas.
Pos X     Pos Y     Pos Z
273       246.5     0
Width     Height
664.5031  600
Anchors
  Min     X 0       Y 0
  Max     X 0       Y 0
Pivot     X 0.5     Y 0.5
Rotation  X 0       Y 0       Z 0
Scale     X 0.82166 Y 0.82166 Z 0.82166
Canvas
Render Mode       Screen Space – Overlay
  Pixel Perfect   ☐
Receives Events   ✓
Sorting Layer     Default
Order in Layer    0
Reference Resolution (Script)
Graphic Raycaster (Script)
```

Configuring a Unity UI Canvas

Avoid using a small number of Canvases

Do not place all UI content on a single or on a small number of "monolithic" Canvases. Each Canvas maintains a mesh for all of its content, and when any single element changes, this mesh gets rebuilt.

Separate content into separate Canvases, preferably by their update frequency. Keeping dynamic elements separate from static ones will avoid the CPU costs of constantly rebuilding static mesh data.

Watch out for Layout Groups

Layout Groups are another common source of performance issues, especially when nested. When a UI Graphic component inside a Layout Group changes – for example, when a ScrollRect is moving – UGUI will search for parent Layout Groups recursively up the scene hierarchy until it reaches a parent without one. The layout of everything underneath will subsequently be rebuilt.

Try to avoid Layout Groups where possible, especially if your content isn't really dynamic. In cases where Layout Groups are only used to perform the initial layout of content, which subsequently doesn't change, then consider adding some custom code to disable those Layout Group component(s) after the content has been initialized.

List and Grid views can be expensive

List and Grid views are another common UI pattern (e.g., inventory or shop screens). In such cases, where there may be hundreds of items with only a small number visible at once, don't create UI elements for them all, as it will be very expensive. Instead, implement a pattern to reuse elements and bring them into view on one side as they move off the other side. A Unity engineer has provided an example in this GitHub project.

Avoid numerous overlaid elements

It is common to see UIs with areas constructed of many overlaid elements. A good example of this might be a card Prefab in a card-battler game. While this approach allows for a lot of customization in designs, it can greatly impact performance with lots of pixel overdraw. Furthermore, it may result in more draw batches. Determine if you can merge many layered elements into fewer (or even one) elements.

Think about how you use Mask and RectMask2D components

Mask and RectMask2D components are commonly found in UI. Mask utilizes the render target's stencil buffer to draw or reject the pixels being drawn, bearing the cost almost entirely on the GPU. In contrast, RectMask2D performs bounds-checking on the CPU to reject elements outside of the mask. Complex UIs with lots of RectMask2D components, especially when nested, can incur significant CPU costs in performing the bounds checks. Take care not to use excessive numbers of RectMask2D components, or if GPU load is less than CPU load, consider whether it is preferable to switch to Mask components to balance the overall load.

Atlas your UI textures to improve batching

Ensure that UI textures are atlased as much as possible in order to improve batching. While it can make sense to use many atlases for logical groups of textures, take care that the atlases are not over-sized and/or sparsely populated. This is a common source of memory wastage.

Also, ensure that atlases are compressed where possible. While it is often undesirable to compress UI textures, due to the appearance of artefacts, this depends on the nature of the content itself.

In most cases, mipmaps are not required on UI textures, so ensure that this Import Setting is disabled on them unless you specifically require it (for world-space UI, for instance).

Be careful when you add a new UI window or screen

Glitches in projects often occur when a new UI window or screen is introduced. There can be many reasons for this (e.g., because of the on-demand loading of assets and the sheer cost of instantiating a complex hierarchy with many UI components).

As well as trying to reduce the complexity of the UI itself, consider caching such UI components especially if they are used relatively often. Disable and re-enable it rather than destroying and re-instantiating each time.

Disable Raycast Target when not needed

Be sure to disable the Raycast Target option on UI Graphic elements that don't need to receive input events. Many UI elements don't need to receive input events, such as the text on a button or non-interactive images. However, UI Graphic components have the Raycast Target option enabled by default. Complex UIs could potentially have a large number of unnecessary Raycast Targets, so disabling them can save significant amounts of CPU processing.

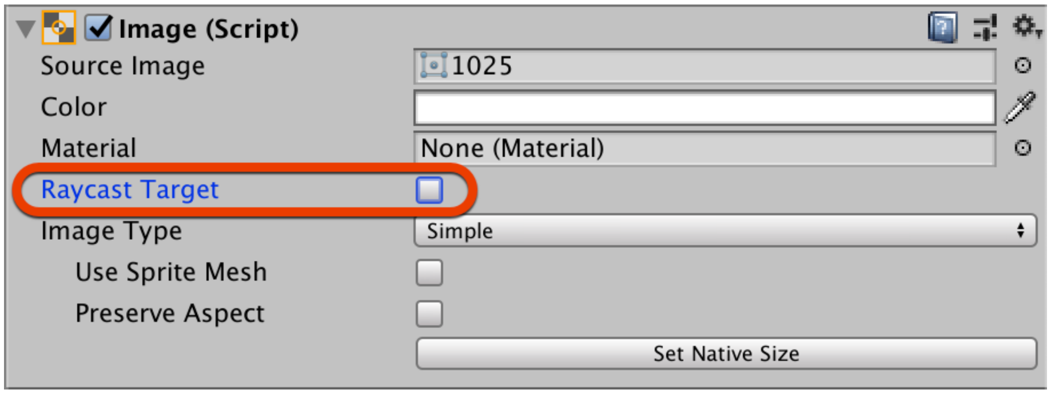


Image (Script)
Source Image: 1025
Color:
Material: None (Material)
Raycast Target: ☐
Image Type: Simple
Use Sprite Mesh: ☐
Preserve Aspect: ☐

Disable Raycast Target on elements that do not need input events

Next steps

We hope that the best practices and tips in this guide helped ensure your project has a robust structure and that you have the right tools and workflows in place to design, develop, test, and launch your game. As your needs evolve and deadline pressure increases, be sure to seek other shared Unity resources.

For more information

You can find additional optimization tips, best practices, and news on the Unity Blog, at the #unitytips hashtag, on the Unity community forums, and on Unity Learn.

Contacting Unity ISS

Need personalized attention? Consider Unity Integrated Success Services. ISS is much more than a support package. With a dedicated Developer Relations Manager (DRM), your project will be bolstered by a Unity expert who will quickly become an extension of your team. Your DRM will provide you with the dedicated technical and operational expertise required to preempt issues and keep your projects running smoothly right up to launch and beyond. To get in touch, please fill out our Contact form.

unity