Preview 0.32- DOTS Runtime & Project Tiny: Getting Started

Table of contents

Document Change log

- *Jan 25, 2021 - 0.32 preview*
- *Nov 2, 2020 - Project Tiny 0.31 is available*
- *Oct 26, 2020 - New demo "Skelebuddies Halloween Run", upcoming preview and roadmap update*
- *Sept 16, 2020 - added “Multi / Single Threaded options” in* Build Configurations *section*
- *Aug 26, 2020 - Project Tiny 0.29 is available*
- *July 27, 2020 - Project Tiny 0.28 is available*
- *July 8, 2020 - Project Tiny 0.27 was skipped*
- *May 27, 2020 - Project Tiny 0.26 is available*
- *May 1, 2020 - Project Tiny 0.25 is available*
- *April 15, 2020 - Project Tiny 0.24 is available*
- *March 24, 2020 - Project Tiny 0.23 is available*
- *March 6, 2020 - 2D Foundational Features for Project Tiny 0.22*
- *Feb 25, 2020 - Project Tiny 0.22 - Added new package to include all dependencies, bug fixes.*

- *Jan 31, 2020 - "Project Tiny" 0.21- Added support for Unity.Physics.com, bug fixes.*
- *Initial version Dec 12, 2019 - Project Tiny Fall update "Tiny Racing" 0.20 is available*

What Is Project Tiny and DOTS Runtime?

Unity embarked on Project Tiny to deliver a highly-modular runtime giving developers the tools needed to produce small, light, and fast games/experiences like Mobile, Playable Ads and Instant social experiences.

Over the lifetime of the project (see Unite Copenhagen 2019 Project Tiny roadmap session recording), the goals have not changed but our path to achieving them has. Rather than a separate Editor mode of its own, Project Tiny aligned closer with DOTS and now shares the regular DOTS conversion authoring workflow. This workflow allows you to work with Unity GameObjects as usual, with all their rich authoring capabilities, while converting into optimal ECS-based data at runtime. (For more information about this authoring flow, see Converting Scene Data to DOTS).

Compared to "regular" Unity content, Project Tiny content targets the new DOTS Runtime and has no dependency on the existing UnityEngine. The DOTS Runtime is a new execution environment focused on DOTS code, with a very lightweight small core runtime that can be extended by modules providing additional features. Our goal is to ensure that you pay code size and execution cost only for the features that you actually use. All functionality is provided as pure DOTS modules, delivered as assemblies, distributed via packages, and is interacted with using DOTS and ECS methods.

Project Tiny is part of a spectrum of "regular" Unity and pure DOTS capabilities. Our goal is to ensure that if a project is compatible with the DOTS Runtime, it also works in DOTS Hybrid / Unity. (We're not there yet.)

System Requirements

The following system requirements will vary and should not be considered as final.

For Development

- **Unity 2020.1.10f1** or later is required.
- **Windows 10**:
  - Rider or Visual Studio 2019 with the following components installed:
    - Desktop development with C++
    - .NET desktop development
    - Windows 10 SDK
  - **Note:** If using Rider, ensure you have disabled the Resharper Build feature as it is not compatible with Tiny's build process.
- **MacOS**:
  - XCode 10.0 or later
  - Make sure `xcode-select -p` returns `/Applications/Xcode.app/Contents/Developer` if not please run `sudo xcode-select -r` to reset the default Xcode path.
- **iOS**:

    - XCode 10.0
    - Unity iOS Build Support module.
- **Android**:
    - Unity Android Build support module installed via the Unity Hub which will install the required Android SDK, NDK, and OpenJDK ((see installation instructions for details)
- **Web**
    - Unity WebGL Build Support module is **No longer required** for Web Build & Run

For Runtime

(note that the following minimum system requirements will evolve and are not final)

- Mobile
    - Native builds:
        - iOS 10+
        - Android 4.4+
    - Browser/Webviews builds:
        - asm.js /WebGL: iOS 10+, Android 4.4+
        - WebAssembly /WebGL: iOS 11+, Android Chrome 71 (Android 4.4.3+)
    - Notes:
        - For iOS, Apple requires that all applications use WKWebView for web content, which is Safari/WebKit based. So other iOS browsers are built on top of Safari, and therefore have the same level of support. The particular version of Safari is tightly coupled to iOS version, so we don't differentiate in our requirements.
        - For Android, we are currently testing on Chrome browser and System WebView, though we may consider adding other browsers to our support matrix in the future.

- Desktop (Windows/Mac OS)
    - Native builds:
        - Windows 10+
        - macOS Sierra 10.12+
    - Web browser builds:
        - Firefox 58+
        - Chrome 52+

About Linux support

Currently, Linux support is Experimental. There are no guarantees on building DOTS Runtime applications targeting a specific flavor of Linux on any given platform. Making DOTS Runtime Linux builds requires running the Unity Editor on Linux.

With that said, some common issues have come up in experimental usage within Ubuntu 20.04 AMD64 which have known solutions:

`sudo apt-get install liblz4-dev`	Resolves dependency on liblz4 in conversion pipeline

`sudo apt-get install ca-certificates-mono`	Fixes a potential issue downloading dependencies during the build process

`wget http://security.ubuntu.com/ubuntu/pool/main/o/openssl1.0/libssl1.0.0_1.0.2n-1ubuntu5.5_amd64.deb`
 	
`sudo dpkg -i libssl1.0.0_1.0.2n-1ubuntu5.5_amd64.deb`
	Resolves dependency on libssl1.0.0 in the build process

`sudo apt-get install xorg-dev`	Resolves various X11 dependencies
`sudo apt-get install libudev-dev`	Resolves device API dependencies
`sudo apt-get install mono-devel`	Fixes multiple issues reported as “missing netstandard 2.0.0”

Though these aren’t as commonly reported, below are a handful of additional solutions if the issue occurs:

`sudo apt-get install libtinfo5`	Resolves dependency of libncurses-dev used in some versions of clang

`wget http://mirrors.kernel.org/ubuntu/pool/main/libf/libffi/libffi6_3.2.1-8_amd64.deb`
 	
`sudo dpkg -i libffi6_3.2.1-8_amd64.deb`
	Resolves another dependency in some versions of clang

Filing Bugs and Providing Feedback

To file bugs, please use the Unity Bug Reporter in the Unity Editor, accessible via Help > Report a Bug. Please include “Project Tiny” in the title to help our staff triage things appropriately! For more details on how to report a bug, please visit this page.

For general feedback, please visit the Project Tiny Form.

Getting started: use Sample Projects

Sample projects are the easiest way to get all components required for Project Tiny. Just go to https://github.com/Unity-Technologies/ProjectTinySamples, clone/download everything, and then open the sample project in one of the folders.
You’ll find the following projects:

- Tiny3D: A "Hello World" style sample with minimal assets & code. It's a great starting point for a new "clean" project.
- TinyRacing: Intended as a lightweight example of the type of content you can build with Project Tiny. It is a complete game slice showing a number of elements such as accepting input, implementing simple AI, handling collisions, and similar. Please explore and play around with the sample project to get a feel for what developing with pure DOTS looks like.
- TinyPhysics: AIllustrates how to combine simple Unity Physics behaviors and common input methods (keyboard, touch, mouse).
- TinyFactory: Demonstrates how Animations can be added to a scene (no code).
- TinyAquarium: Demonstrates Particle System the first iteration of the particle system which mirrors a subset of the functionalities available with the built-in Particle System (aka shuriken).
- RuntimeGeometry3D: Demonstrates how to do Dynamic meshes creation at runtime.
- 2D Entities Starter: a template project, which has everything setup for you to get started with the 2D Entities package.
- TinyGems and TinySpaceship: 2D sample projects that showcase some of the features that can be found inside the 2D Entities package.

A number of areas in these samples are currently implemented in a way that is not final due to missing features, such as UI. We'll improve the sample projects as a richer feature set becomes available.

Within the project folder, there is a "Build" folder that contains predefined build configurations for a variety of platforms. For example, if you are on Windows, you should be able to click on the "Windows-DotNet" asset and select "Build and Run" in the inspector window to build and run the sample for Windows.

Feature Status Summary

Feature Status Summary as of the current preview version

Platform Support	Desktop platforms, the Web (both asmjs and WebAssembly), Android, and iOS are available. Single HTML file export (for playable ads scenario) is now available.
Mobile Specific features	Screen orientation control, suspend/resume support added. Advanced build settings support.
Jobs	The Jobs API is fully supported.
Burst	Burst is now available for Windows (DotNet/IL2CPP), MacOS (DotNet/IL2CPP), and iOS/Android (IL2CPP) with Debug, Development & Release builds. Other platforms/options coming in future releases.

3D Graphics	A lightweight 3D renderer is available in Unity.Tiny.Rendering including PBR, dynamic mesh. The capabilities of this renderer will be expanded in the future.
3D Particles	Initial support for particles added to preview 0.24 for (3D only for n now) as a subset of Unity built-in particle system (aka shuriken)
2D Graphics & 2D Physics	Support for 2D foundational features available
Input	Lightweight input is available via the Unity.Tiny.Input assembly, Touch input as a priority with Multi-Touch added in 0.25. Touch is supported only for Mobile (iOS/Android) & Mobile/Web, not on desktop (Windows/Mac). Accelerometer/Gyro support is available for Android and iOS.
Audio	Lightweight audio is available via the Unity.Tiny.Audio assembly. 3D audio, pitch control also available
Animation	Rudimentary support for animation workflows using the TinyAnimationAuthoring component. It allows users to author animation that works in Tiny from the Animation window. See TinyFactory sample. Currently supports transform-based animation + any float field in user code can be animated, as long as it can be bound to in its MonoBehaviour counterpart. See assemblies: Unity.Tiny.Animation and its siblings.
3D Physics	Unity.Physics is available to use with Project Tiny. See TinyRacing and TinyPhysics samples (https://github.com/Unity-Technologies/ProjectTinySamples/).
Interop	Using native plugins is supported, see Native Plugins section. For C# / Javascript Interop see Calling Javascript functions section.
UI	Lightweight UI solution partially available - see Tiny UI section
Text Rendering	A lightweight solution to render text for both UI and for in-game text will be coming this year.
Networking	Support for HTTP client planned.
Custom Shaders	Initial implementation started with preview 0.31. See Shaders section for details.

Skinned Mesh Renderer	Simple GPU/CPU skinned mesh is available. See Skinned Mesh Renderer and BlendShape section

Anatomy of a Tiny Project

A Unity project that's compatible with the DOTS Runtime is very similar to a normal Unity project with the following differences:

1. Scenes must be composed entirely of DOTS Subscenes containing convertible components, or use Convert to Entity components on GameObjects in a top-level scene. (You can use unconverted GameObjects as authoring aides, but these are not preserved in play mode or the built application.)
2. Run-time code must be compiled into an assembly using an Assembly Definition.
3. Your code can only use DOTS APIs. No UnityEngine APIs are supported (with a very few exceptions for some source code compatibility, like logging).
4. Building (and Running) must be done using a Build Configuration asset, with a DOTS Runtime build profile.

These requirements are discussed in more detail below. We'll use the TinyRacing project as an example where you can see all of these requirements in action.

Scenes and Subscenes

Because DOTS Runtime is pure DOTS, scenes must be convertible to their DOTS Entity and Component representation. There are no GameObjects or MonoBehaviours in the DOTS Runtime world. **Anything that is not convertible will not exist at runtime**.
For more details, the Converting Scene Data to DOTS talk from 2019 Unite Copenhagen gives general information.
This area is under heavy development. The setup described below will be simplified in future releases. Additionally, much better guidance will be given in the Editor as to what can and can't be converted.
Using Sub-Scenes to enable conversion is no longer required. Conversion from GameObject to Entities is now enabled at scene Root. Using subscenes to organize content is still possible.

In TinyRacing, the scene in "Scenes/TinyRacing.scene" contains a single subscene (the "DOTS Subscene" scene).

GameObject and Behavior Authoring

With the GameObject conversion authoring workflow, DOTS runtime data is created from GameObjects and MonoBehaviours. A portion of the standard behaviors and assets have conversion code defined -- for example, MeshRenderers, MeshFilters, Material and Texture assets. However, your game logic needs to be defined entirely using ECS components and executed using ECS systems.

A detailed list of supported behaviors is coming soon. Additionally, we're working on extending the Editor to provide guidance in the UI around what's supported and how it will be converted.

The “Root Assembly”

DOTS Runtime, and with it Project Tiny, only supports pure DOTS code. All functionality provided has been designed with DOTS in mind and takes full advantage of DOTS, C# Jobs, and Burst. In order to enforce the separation of pure DOTS code from code that references UnityEngine types or only works in the Unity Editor, your game scripts must be compiled into one or more assemblies defined using Assembly Definition assets. You must designate one Assembly Definition as the **Root Assembly** in your Build Configuration asset.

The root assembly must reference any assemblies containing the classes and types your code uses. For example, it must reference Unity.Entities for core DOTS functionality, but also may reference Unity.Tiny.Input and Unity.Tiny.Rendering to gain access to the input and rendering functionality provided by Project Tiny. If you create additional assemblies for your project, they also must be referenced by the root assembly, either directly or through other referenced assemblies. Any scripts that are not part of an assembly are not included in your app.
See Assembly Definition properties for information about setting references and other assembly properties.
In TinyRacing, the root assembly is defined by the TinyRacing.asmdef asset in the “Scripts/TinyRacing” folder.

DOTS .NET Subset

All code written for DOTS Runtime is built against a subset of the .NET framework optimized for code size and performance and organized in what we call **Base Class Library**. Much of standard .NET is NOT available. We’ve created a list of what’s included in the Project Tiny Base Class Library:
https://docs.unity3d.com/Packages/com.unity.tiny.all@latest?subfolder=/manual/tinybcl.html
This area is under heavy development, and what is and is not available in this profile is not finalized.
Our goal is to provide chunks of opt-in functionality to provide maximum flexibility in what is and isn’t used by your projects. Thus, just because your code builds using the Unity C# Project, does not guarantee that it will build using the DOTS Runtime build system (which will use its own DOTS C# Solution).

Instead, you need to generate a separate DOTS C# Project, which targets the limited .NET subset. See below for more information about this project.

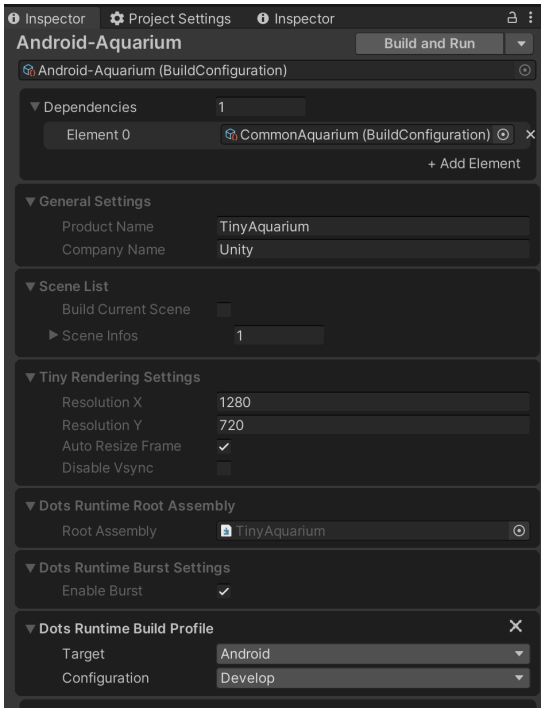
Notable limitations of the DOTS .NET subset include:

- No runtime reflection (System.Reflection and friends)
- No heavyweight functionality (e.g. System.Xml)
- No non-generic collections (List<T> is available, List is not)
- Many of the more complex generic collections are missing
    - While this will be fixed, more efficient collections are already available as part of Unity.Collections: NativeList, NativeSet, NativeHashMap, NativeMultiHashMap
- No generic ToString and Equals functionality (each type must explicitly define their own version of these functions when needed)

Build Configurations

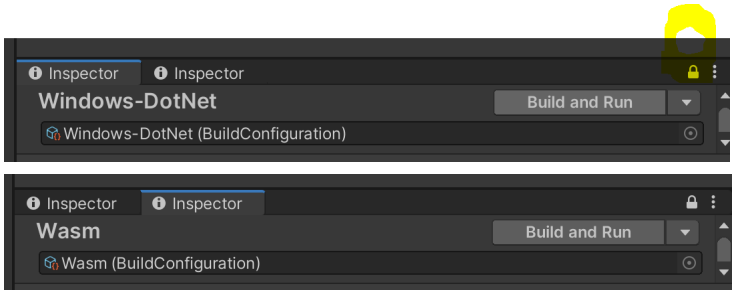
You build DOTS Runtime and Project Tiny apps with the new Build Configuration mechanism instead of the usual Unity Build Settings window.

The new Build Configurations are assets that live in your project that define a build: the individual settings and the steps (Build Pipeline) used to execute those steps. It lets you easily define multiple build configurations and store them with your project, as well as extend with your own build data and build steps. Build Configurations can inherit their settings from another Build Configuration asset.



Build Configurations are used to create DOTS Runtime/Project Tiny builds, based on the Build Profile component present in the Build Configuration asset. Build Configurations can be inspected to view their data, as well as trigger builds.

Tip: you can add inspectors, lock them to your favorite Build Configurations and dock them in a corner of the editor:



This area is under heavy development. While the Build Configuration functionality is very powerful, its design is still a preview. It may change significantly in the future, but the core functionality will remain the same. The UI and overall user experience will also be significantly improved.

In TinyRacing, various Build Configuration assets are defined in the “BuildConfigurations” folder.

Authoring and Editor-only Code

The DOTS GameObject authoring workflow requires that authoring happens using GameObjects and MonoBehaviours. These are converted to an ECS representation using conversion systems. For your own components, there are three ways to define the authoring MonoBehaviour and runtime components. In increasing order of complexity (and power):

- GenerateAuthoringComponent attribute. Any IComponentData can have a [GenerateAuthoringComponent] attribute placed on it. This will automatically generate a MonoBehaviour that can be used for authoring with the same fields as those present in the component.
- A MonoBehaviour that implements IConvertGameObjectToEntity. This interface defines a Convert method that will be called on the MonoBehaviour and must create any ECS components it needs using the provided EntityManager and Entity.
- A GameObjectConversionSystem. This is the lowest level and most powerful mechanism. These systems execute at conversion time and have a full view of both the original Unity Scene data as well as the destination ECS EntityManager data.

For all of these approaches, the IComponentData to be used at runtime *must be defined in an assembly that’s built as part of the DOTS Runtime build*: either the Root Assembly or one referenced by it.

For IConvertGameObjectToEntity and GameObjectConversionSystem, the MonoBehaviour or conversion system code *must be in a separate assembly that is not referenced by the Root Assembly*. (Compilation errors at build time will result if this isn’t true.) Unity recommends that you add a “.Authoring” suffix in the name of these assemblies.

In the TinyRacing project, you can see a mix of GenerateAuthoringComponent used directly in the TinyRacing assembly, as well as a separate TinyRacing.Authoring assembly that defines some more complex conversion code. It uses a custom GameObjectConversionSystem to handle some hand-baked UI and textures for displaying numbers at runtime.

Runtimes and Packages

When talking about DOTS, it’s important to understand that Unity has two runtimes:

- “classic” Unity is using the UnityEngine Player Runtime, and enable "Hybrid" mode with DOTS components powering part of your projects like the Megacity project,
- “new” DOTS Runtime which currently only Project Tiny is using, like the Tiny demos based on DOTS runtime.

The new DOTS Runtime is built by assembling the required packages to ultimately form a complete product. The first product built on the DOTS Runtime is Project Tiny which is made of multiple packages grouped under com.unity.tiny.all.

Then additional optional features are available also as packages. For example com.unity.physics which works with DOTS runtime but also with the “classic” UnityEngine Player Runtime

In the future, most new features in Unity will be built as packages supporting both runtimes.

We understand that it is a bit confusing with all these packages. We are currently working on ways to improve this, and make it easier to get a better overview.

Setting up a New Project

New project setup will be greatly simplified in the near future via a Project Tiny template.
For now sample projects like TinyRacing/Tiny3D/TinyFactory (available here https://github.com/Unity-Technologies/ProjectTinySamples) are **the easiest and recommended** way to get all components required for Project Tiny.

If you really want to set up a project on your own, follow the steps below.
If you want to work with **2D features** (*com.unity.2d.entities*), check the 2D Projects section.
For manual setup of a new 3D project, follow these steps:

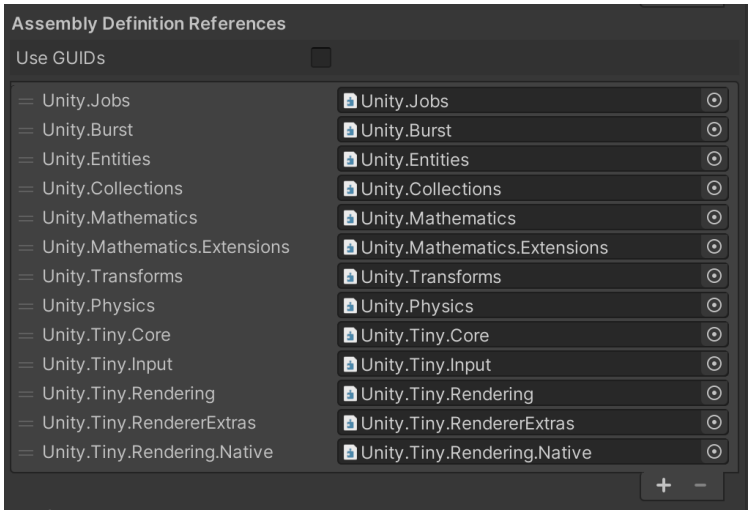
Project and Package Setup

1. Start with a Universal Render Pipeline project template.
   The Project Tiny renderer will be designed to match a subset of baseline URP rendering. (Note: still very much a work in progress.)
2. Add the main Project Tiny Full package (com.unity.tiny.all) which includes references to all packages and all features for targeting the Web, Android, iOS and the desktop platforms.

Code Setup

1. Create a new folder inside Assets named “GameSystems”
2. Inside GameSystems, create a new Assembly Definition asset
    a. Right-click Create and select Create > Assembly Definition
    b. Name it “GameSystems”
    c. Inside the inspector, add the following assemblies as references:
        i. Unity.Entities
        ii. Unity.Collections
        iii. Unity.Transforms
        iv. Unity.Mathematics
        v. Unity.Tiny.Core
        vi. Unity.Tiny.Rendering

    For example in TinyPhysics sample the list of referenced assemblies looks like this:



3. Inside GameSystems, create two C# Script files named:
    a. RotateComponent
    b. RotateSystem
4. Edit RotateComponent.cs and replace the contents with the following. This defines a new ECS component and makes it available for use in the Editor.

```
using Unity.Entities;
[GenerateAuthoringComponent]
public struct RotateComponent : IComponentData
{
    public float Speed;
}
```

5. Edit RotateSystem.cs and replace the contents with the following. This defines a new ECS system that uses the above component to change an entity's rotation.

```
using Unity.Entities;
using Unity.Transforms;
using Unity.Mathematics;
public class RotateSystem : SystemBase
{
    protected override void OnUpdate()
    {
        var dt = Time.DeltaTime;
        Entities.ForEach((ref Rotation rot, ref RotateComponent rc) => {
            rot.Value = math.mul(rot.Value, quaternion.RotateY(dt * rc.Speed));
        }).ScheduleParallel();
    }
}
```

Scene Setup

1. Create a new scene called Main.unity
2. Remove the audio listener component from the camera
3. Create a cube
    a. Right click Create > 3D Object > Cube
    b. Move the cube forward (in Z) a bit and scale it up so that it's visible in the camera
4. Remove the Box collider component from the cube
5. Add a component to the cube -- search for "Rotate Component"
    a. This is the component we declared earlier in code setup

6. Set the speed to 0.2

Build Setup

7. Create a new Build Configuration asset via right clicking in the Project window, and then Create > Build > DOTS Runtime Build Configuration
8. Drag or select your Root Assembly assembly definition into the Root Assembly field.
    a. This is the “GameSystems” assembly created above
9. Add a Scene List build configuration component, if one isn’t present.
    a. To add a component, press the + button in the bottom of the inspector.
10. Add your Main.unity scene to the scene list.
    a. In the Scene List section, set the Size to 1 (or the number of scenes you want to include in the build).
        i. Yes, this way of interacting with arrays is clunky. It’ll be fixed soon.
    b. Drag the Main.unity scene asset to the array slot below the Size field.
11. Select your build target and build type.
    a. For example, “Windows .NET” and “Develop”
12. Press Build and Run in the upper right.
13. For WASM and AsmJS builds, make sure to add the “Emscripten Settings” component to the build configuration and add to the EmccArgs list an entry with the string “-s TOTAL_MEMORY=128MB”, or however much maximum memory your application is allowed to use. The recommended minimum is 128MB.

Running and Debugging

Build Configurations

Build Configurations are the main entry point for building and running your project. For example, in the TinyRacing project, inspecting the “Build/Windows-DotNet” (or Mac, etc.) build configuration will show a “Build” or “Build and Run” button in the upper right. Pressing Build will build and place the result (by default) in a Builds folder at the top level of your project. Build and Run will build and (if possible on the target platform) run the resulting build.

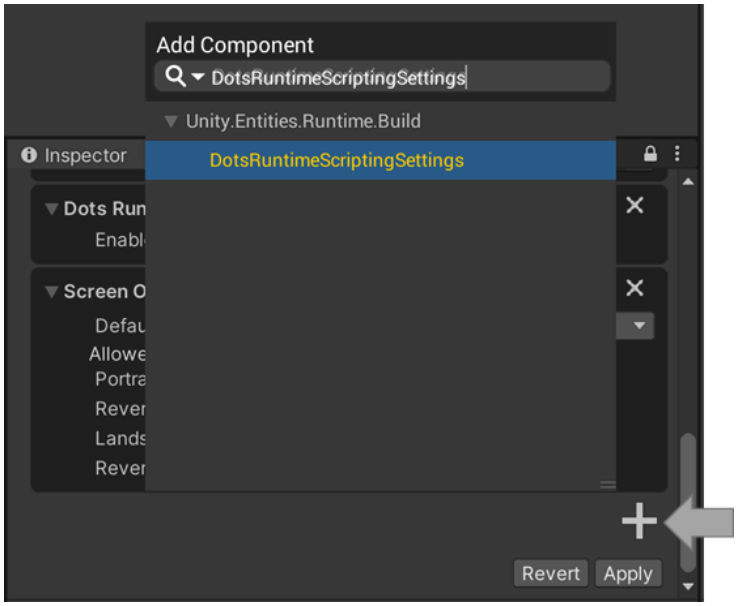
On the Web, you will have to manually spin up a web server to launch your build. (The “http-server” npm package can help here.) Playing directly from a file:// URL is not supported due to web browser security constraints.

Rendering specific build settings can be added via the "Dots Runtime Scripting Defines" components. Currently supported defines are:

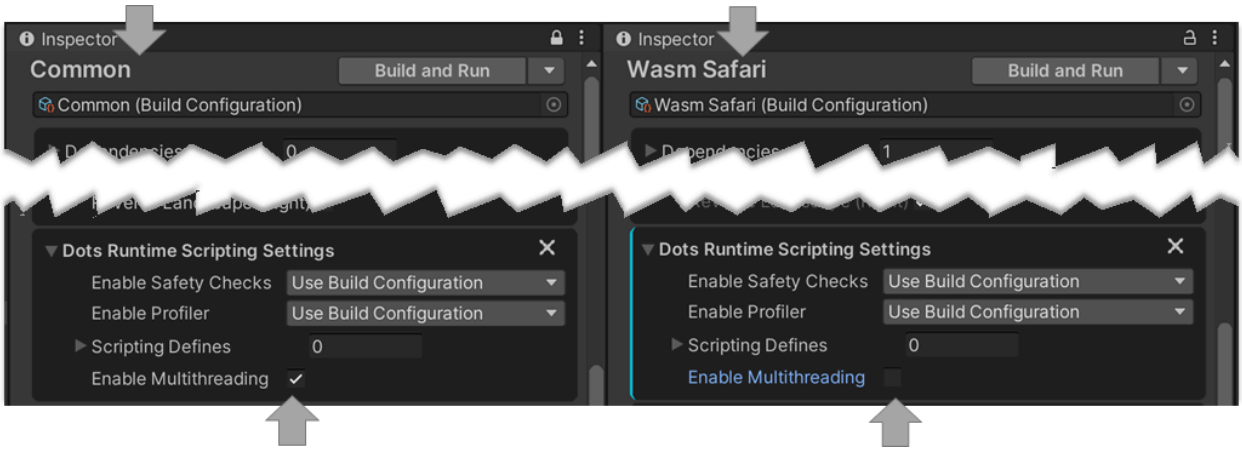
RENDERING_FORCE_OPENGL
Force opengl rendering. Useful to test Metal/DirectX/Vulkan only issues.
RENDERING_ENABLE_TRACE
Enable rendering tracing.
This has a potentially very high performance cost.
Best used in combination with full DEBUG builds which run the rendering layer in debug mode.

Multi / Single Threaded options
Multithreading is available for all supported build targets.

To enable Multithreading, click on the “+” sign at the bottom of the Build Configuration inspector and add the DotsRuntimeScriptingSettings component:



Then you can check “Enable Multithreading”.
Note that Multithreading is not supported on Safari. One way to workaround is to set multithreading on in the Common build configuration inherited by others, and then create a specific WASM build configuration for Safari. The web template loading the build will have to be changed to select the correct build.

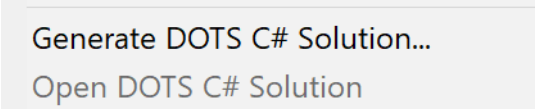


In the future, we plan to add an automatic fallback to Single threading for Safari and do more work on Burst to improve web performance and size.

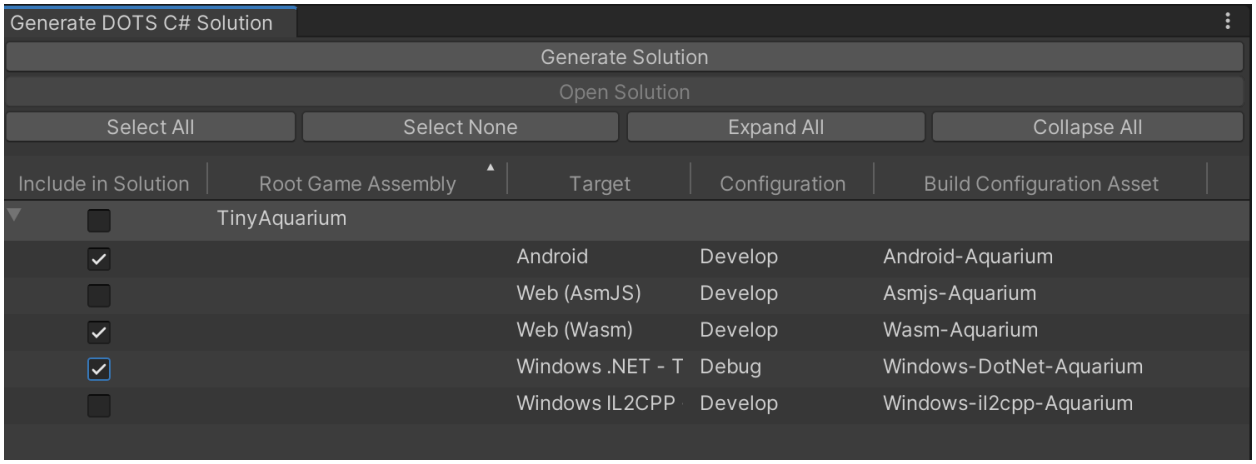
Note, due to the nature of how multithreading is supported in the DOTS Runtime Tiny configuration, enabling Multithreading will automatically enable Burst Compilation as well. It is not possible to have multithreading enabled without burst compilation enabled in DOTS Runtime unless building for the non-Tiny configuration, Net Standard 2.0 on desktop.

DOTS C# Solution

Because of the environment constraints detailed earlier in the DOTS .NET Subset section, a separate DOTS C# Solution from the regular Unity one needs to be created to enable debugging for example. This DOTS C# Solution targets the .NET framework and assemblies that are used during build with the DOTS Runtime, and can be built via **Assets** > **Generate DOTS C# Solution**.



You'll get the following window allowing you to select the build configurations you want to be included in the solution:



The "Generate DOTS C# Solution" window list will be populated with the build configuration assets available in the Unity project. Selecting any buildconfig in this window will show the corresponding build configuration asset in the inspector.
You can also double-click to expand/collapse rows in the view, multi-select, and toggle via spacebar.
Once the solution is generated you can launch it via
Assets > **Open DOTS C# Solution** or right click in the Project Explorer and it will launch your preferred IDE. If you make code-only changes, you can build and run directly from your IDE (currently tested with Visual Studio and Rider, but will be expanded in the future).
Some details to note:

- The project solution is named the name of the project, with "-Dots" at the end.
- After a Build/Build&Run is triggered at least once from the Editor for a specific Build Configuration Asset, it will be preselected in the list of the DOTS C# Solution window.

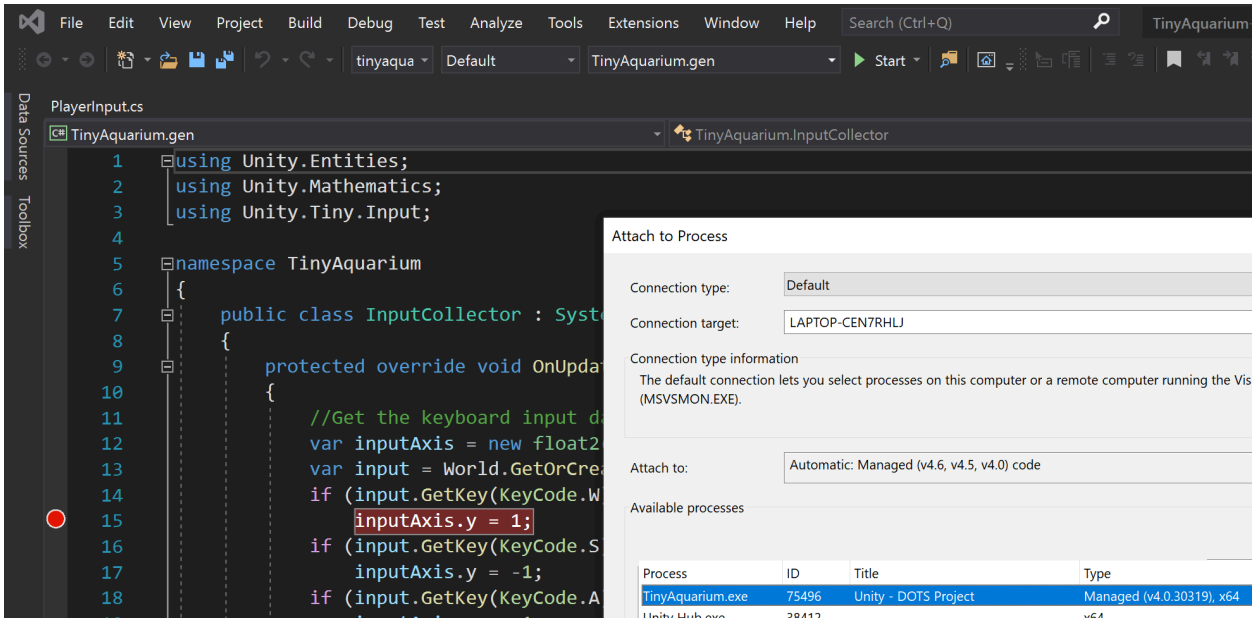
.NET vs il2cpp Builds

DOTS Runtime builds can be built using the regular (mono or Microsoft) .NET runtimes or Unity's il2cpp runtime. When built for .NET, the end result is a pure .NET application (with native code shared libraries). .NET builds are intended for development, while il2cpp builds are intended for final builds.

Debugging with a DOTS Runtime/Project Tiny build

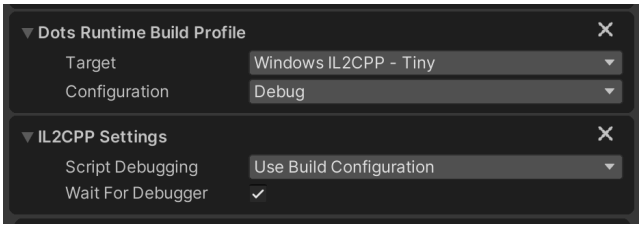
After creating the DOTS C# Solution (see DOTS C# Solution section) and opening it in your IDE, you can set breakpoints and run and debug.

When you do a .NET based build from Unity with the Build & Run available in the Build Configuration inspector, you need to attach to the process and not “Attach Unity Debugger”:

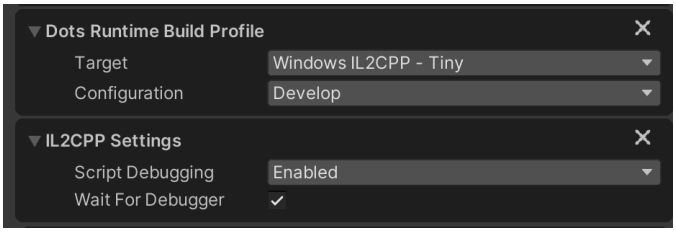


However, if you want to debug an IL2CPP build for Web, Windows, macOS, iOS, or Android, managed code debugging should be enabled at build time. It can be enabled in two ways:

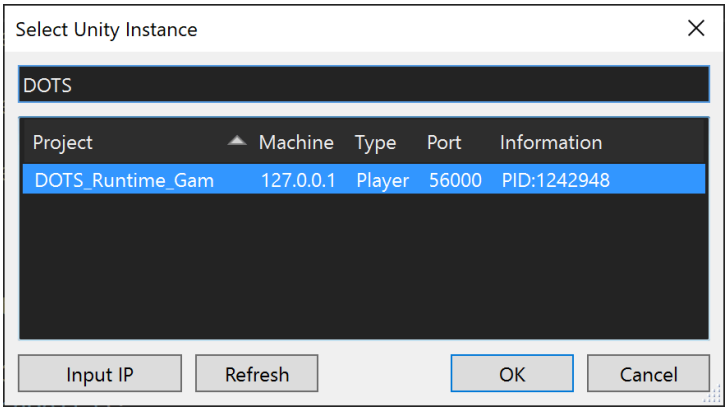
- By default, the "Debug" configuration will enable managed code debugging. The "Develop" and "Release" configurations will not enable managed code debugging by default.
- Note that L2CPP Debug builds are faster to make, have Script Debugging enabled by default, but it will affect performance and your app/game will run notably slower. Release/Develop builds are longer to make and don't have Script Debugging by default, but will run faster.
- For more control, add the "IL2CPP Settings" build component. The "Script Debugging" option allows fine grained control of script debugging for any build configuration.
    - In this case "Use build configuration" value means that the selected Configuration will be followed. In this case “Debug” which has Script Debugging enabled by default.



- In this “Develop” Configuration, the IL2CPP Settings for Script Debugging is enabled

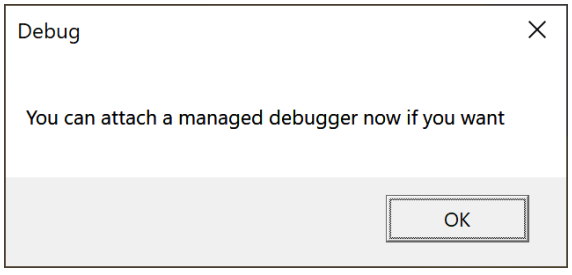


Once a player with managed code debugging is built and running, any IDE that supports debugging of Unity players can be attached to it. For example in Visual Studio, you use Debug > Attach Unity Debugger and select the “DOTS_Runtime” player:

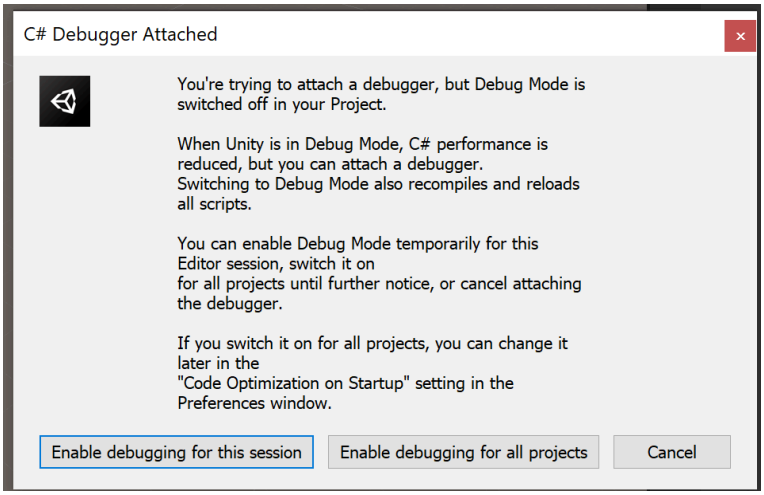


The Unity manual contains details about how this works.

The "IL2CPP Settings" object also provides a "Wait For Debugger" option. This option is disabled by default. It will cause the player to display a dialog before any user managed code is executed. The player will not continue until this dialog is acknowledged by the user. While this dialog is displayed, the user can attach a debugger to the player process.



You will see this dialog if you are trying to attach a debugger but haven't selected the correct set of options:



When managed debugging is enabled for the Web platform, a special executable must be running: websocket_to_psoix_proxy. This executable is part of the Emscripten installation, and will be started and maintained by the Unity editor when the Build and Run or Run options are used from the build settings.

Note that the web browser must support the Shared Array Buffers feature in order to use managed debugging. In some browsers, this is enabled by default. In others, it must be manually enabled. See https://caniuse.com/#feat=sharedarraybuffer for up-to-date information about which browsers support it.

Debugging conversion systems:

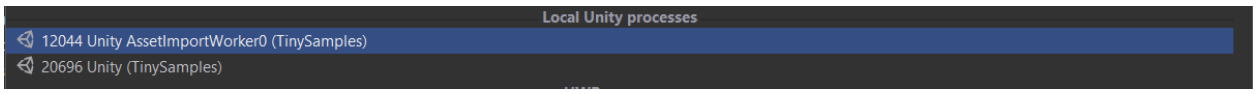
Conversion happens in the Editor when making a build, and is one of the first steps of the build process. The converted gameobject scenes will be serialized as entity scene files and will be deployed to the build directory.
Conversion is incremental and triggered by a scripted importer, conversion systems only run if the Editor detects the following:

- Modification in one of the scenes of the SceneList component.
- Modification of a dependent asset (texture, font, mesh, ..) used in one of the scenes.
- Modification of the build configuration used to build
- Incrementation of the version number of one of the conversion systems (see attribute ConverterVersion).

To force re-run all conversion systems in your project you can dirty your global entity cache asset (Assets/GlobalEntitySceneDependency.asset) by doing in the Editor: Dots->Clear Entities Cache.

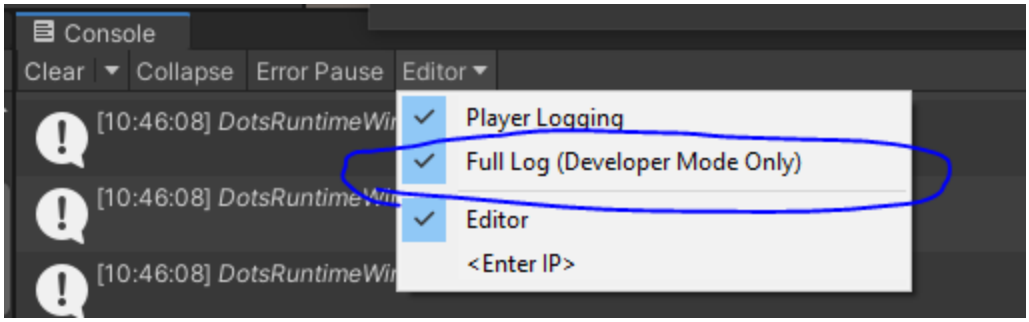
To attach your IDE to the Editor and step into the code:

- Make sure to set up the option Code Optimization On Startup to Debug (Preferences -> General in the Editor).
- Attach your IDE to an Asset Import Worker process not the standard Unity process. (Conversion is run in a scripted importer).

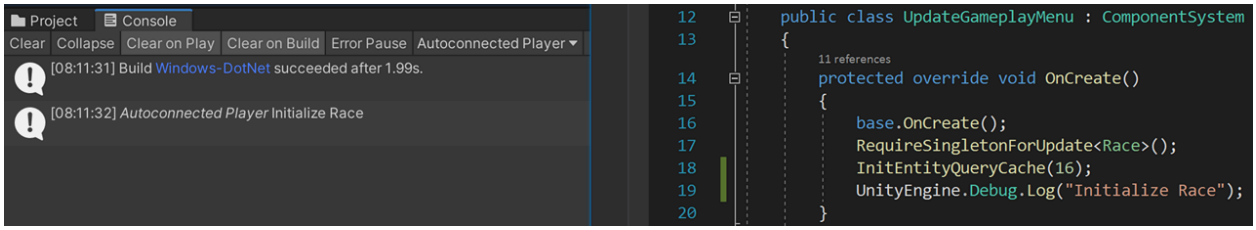


Using Debug.Log()

On Desktop, Web, Android, and iOS, you can use Debug.Log() to display messages in the console.
You need to ensure “Full Log (Developer Mode Only)” is enabled
For Web, the Editor and the running Web build must be on the same computer.
For Mobile, the Editor and the device need to be on the same Wifi. After pressing “Build and Run”, the Editor should automatically connect to the mobile device. If that fails for any reason, you may also enter the IP of the mobile device to connect via the “<Enter IP>” option.



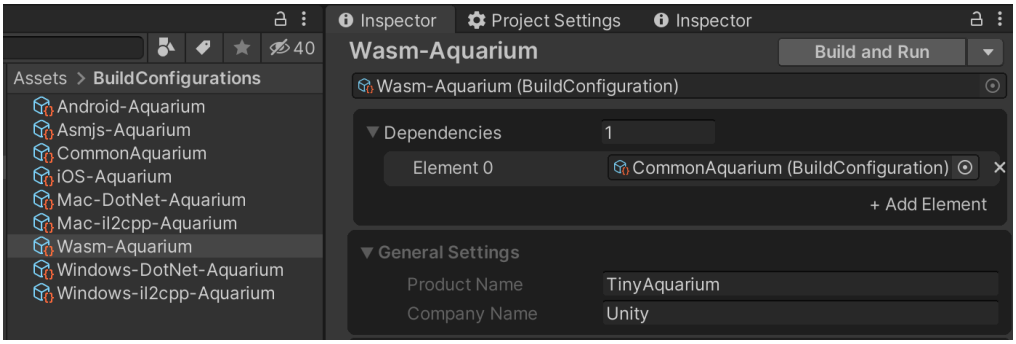
Once connected, Debug.Log() messages go to the console.



As mentioned below for Profiling, Debug.Log() with Web builds are only supported with one instance of the Editor running at the same time.

Web Build & Run

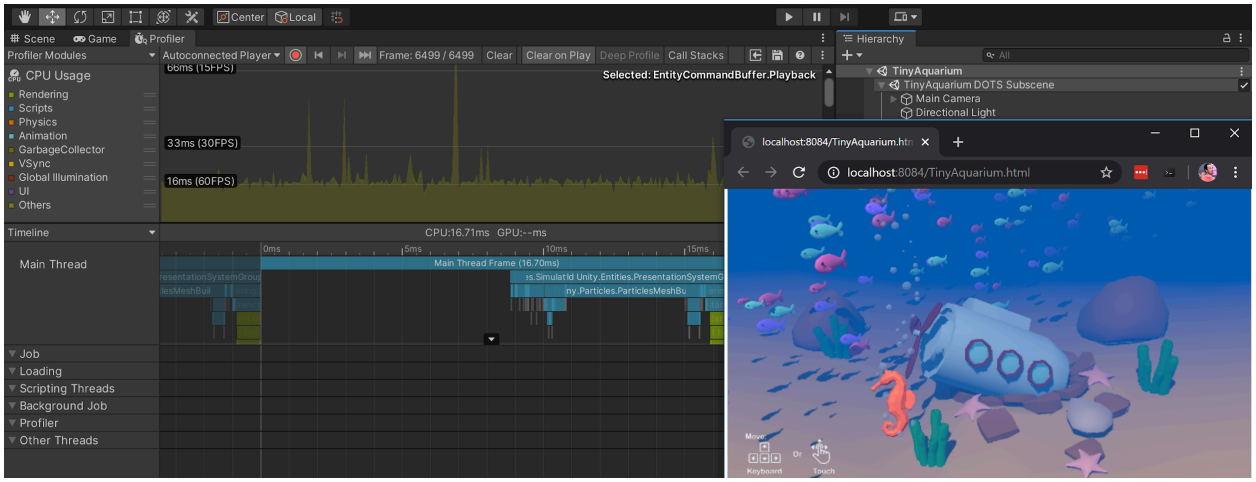
Added with 0.25, you can now use Build & Run (from the Build Configuration inspector) for Web builds:



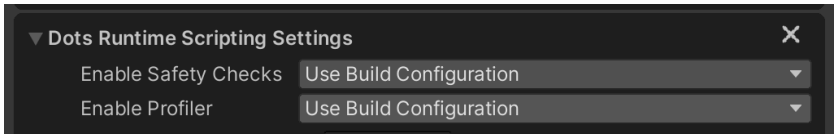
To enable the “Run” part for web builds you **No longer need to** install the Unity WebGL Module, as Project Tiny/DOTS Runtime for Web now includes its own local HTTP-WebServer.

Profiling

You can use the Profiler on Windows, Mac, iOS, Android and Web.The Profiler reports only CPU information for now.
For Desktop and Mobile devices, the Editor Profiler will auto-connect to the local runtime started by “Build & Run”.
For iOS & Android, the Editor and the device need to be on the same Wifi. If auto-connection fails for any reason, enter the IP of the mobile device to connect via the “<Enter IP>” option.
For Web build, the runtime in browser will initiate the connection to the Editor, both must be running on the same computer, and you must have only one instance of the Editor running at the same time.



Profiling is enabled by default for Development and Debug builds. In the Build Configuration, you can control how to enable/disable Profiling via the Dots Runtime Scripting Settings:



Play-in-Editor

Starting with preview 0.28, the Play-in-Editor (in Game view via DOTS Hybrid mode) is no longer supported and will be replaced in the future with an embedded DOTS Runtime player taking over and replacing the Game view.
You can still play some Project Tiny samples in the Editor, but some features will not work properly. For example Input requires to have "`#if UNITY_DOTSPLAYER / using Unity.Tiny.Input;`" and a few more ""`#if UNITY_DOTSPLAYER`" in your code to work in both Editor & standalone.
For now we recommend you use "Built and Run" with the Windows/Mac DotNet build configurations for the best result and still fast iteration. The Build process is incremental and relatively fast when you've made only small changes.
Stay tuned for further improvements in this area.

GameObject/DOTS Features and properties mapping

This section highlights which features & properties are valid for Project Tiny, meaning they will be converted from GameObjects to Entities and available at runtime.

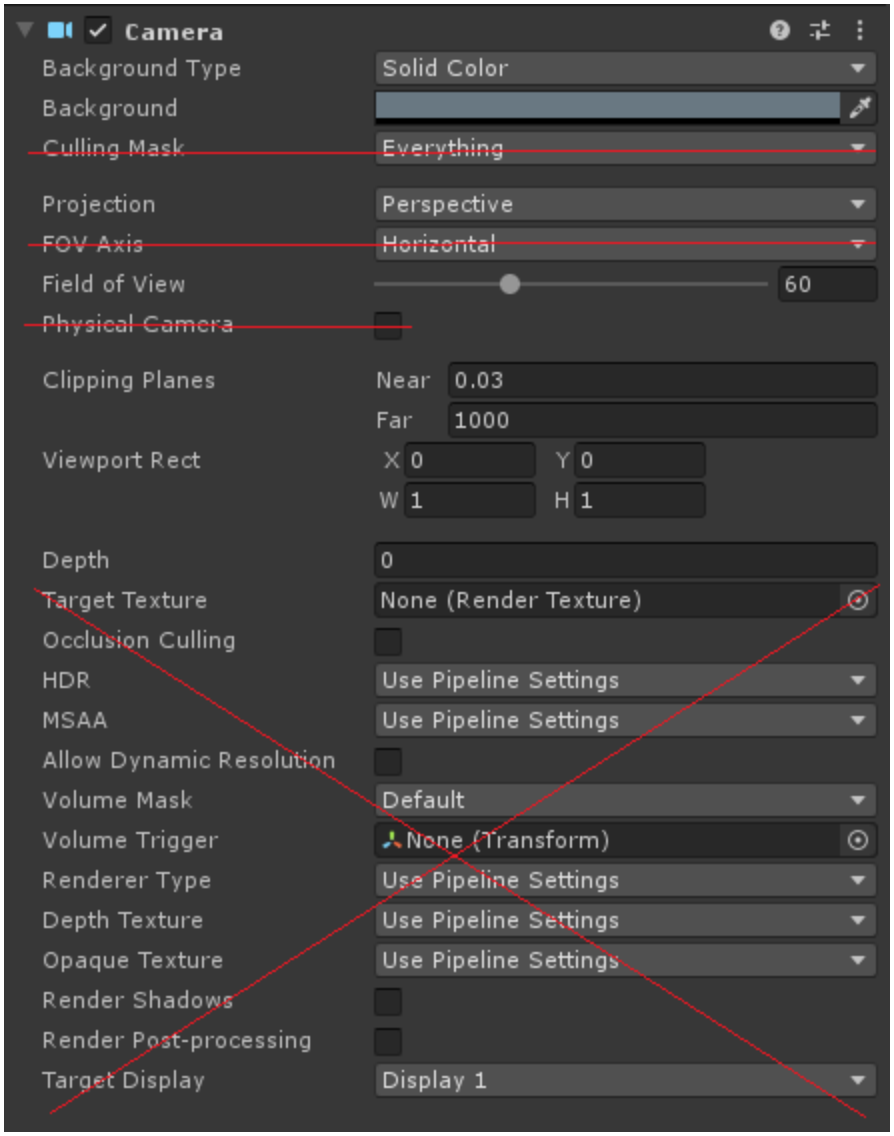
Camera

Starting with preview 0.31, using multiple cameras is supported: it works very similarly than Unity "classic" cameras. You create multiple cameras in the Editor, and use depth to order them. Use Viewport Rect for sizing, rendering object layers and camera culling mask are used.

There are some known issues as of preview 0.31:

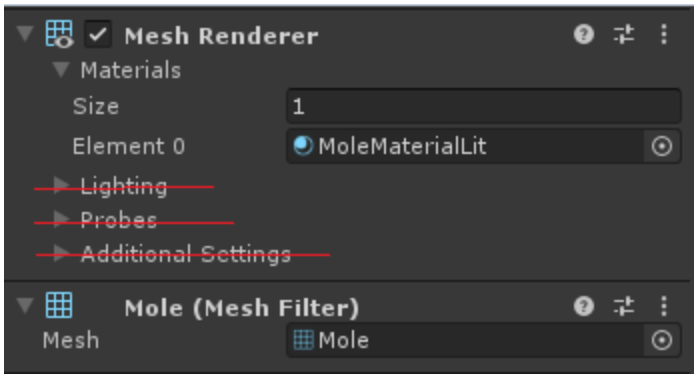
- *The stacking ui mode for cameras does not convert properly yet. Use order/depth instead.*
- *Disabling a camera at runtime might crash*

Camera supported functionalities:



Mesh Renderer

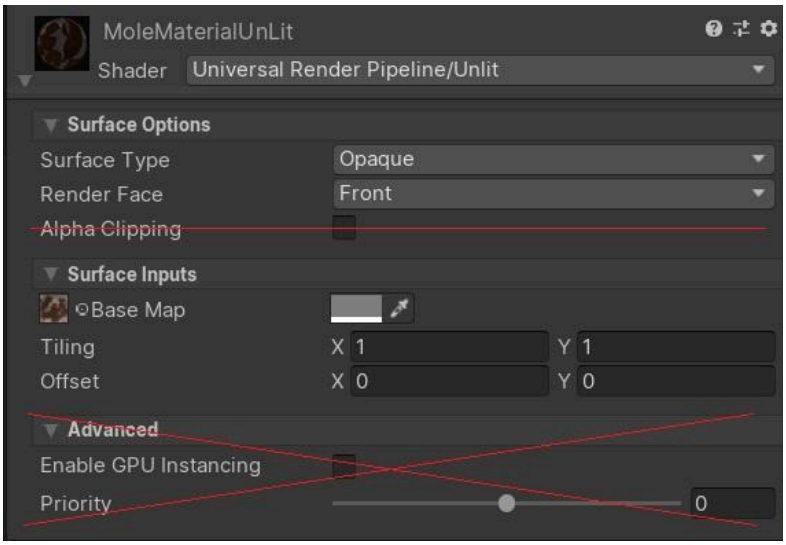
Each single submeshes of the mesh given by the mesh filter, and its associated material (material are converted first) we create an entity with a MeshRenderer, a Simple/LitMeshRender tag component. One entity mesh is created for each shared mesh asset in a subscene and contains the whole mesh data (in a blob asset) that each mesh render is pointing to. One entity material is created for each shared material in a subscene. Additional properties are not supported at the moment.



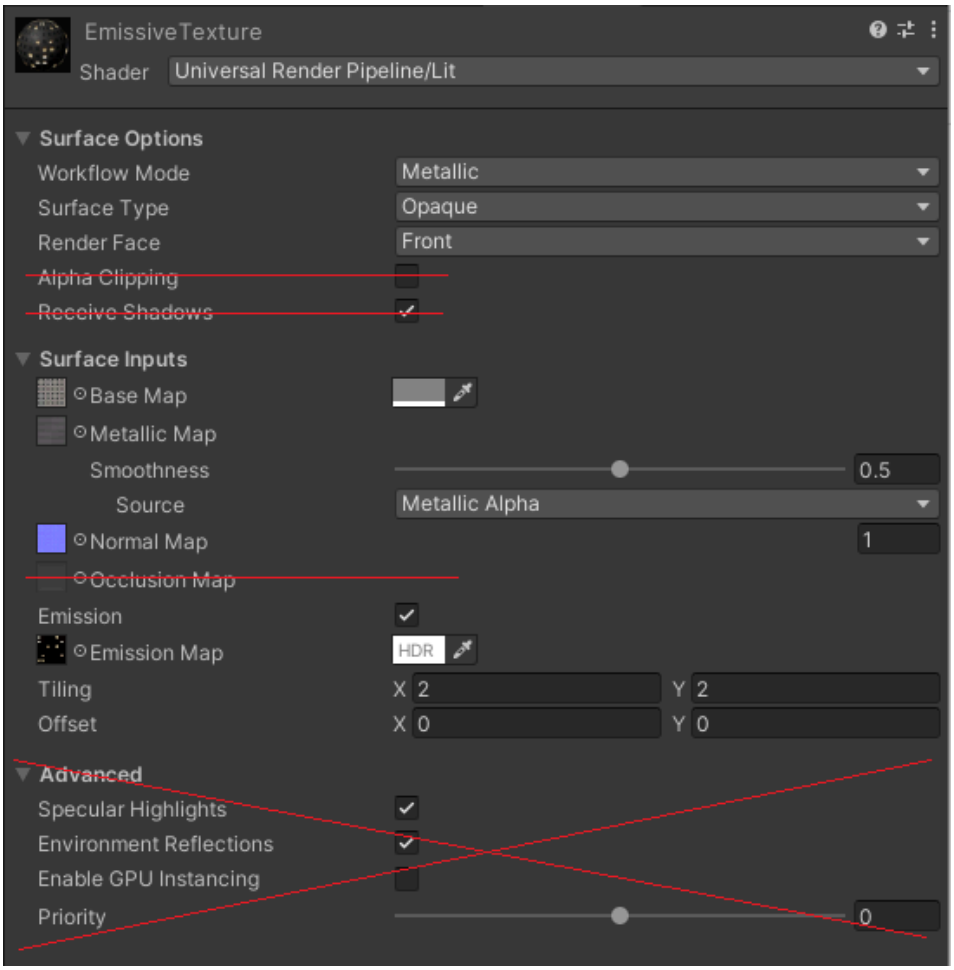
Material

Only 4 URP materials are supported right now:
Lit/Unlit and ParticleLit/ParticleUnlit as described in the Particle section below

- **Universal Render Pipeline/Unlit (SimpleMaterial)**



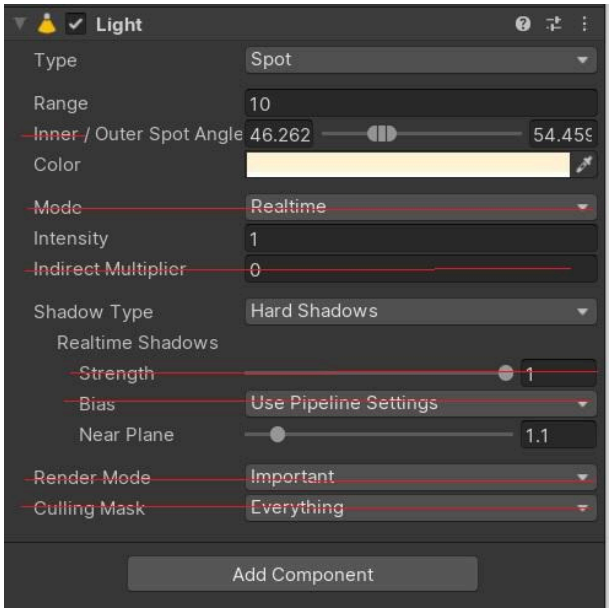
- **Universal Render Pipeline/Lit (LitMaterial)**
  Only the metallic workflow is supported:



Light

Supported built-in functionalities:

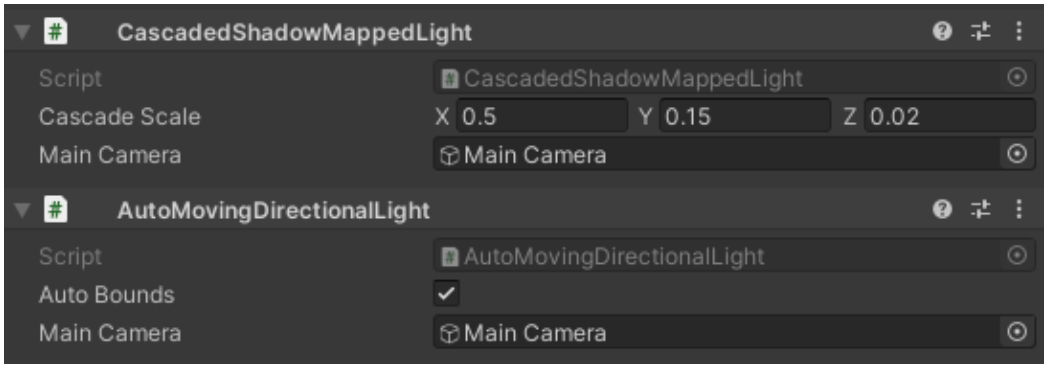
Directional lights, Point Lights and Spot Lights are supported. Shadows are supported as well for Directional Lights and Spot Lights.
Spot light without shadow is not supported.
A Maximum of 8 lights (directional+point+spot) and a maximum of 2 lights with shadows



Additional functionalities

2 monobehaviours have been introduced:

- CascadedShadowMappedLight that does 4 cascades shadow mapping. The cascade scale component must be clamped between 0 and 1 with x>y>z.
- AutoMovingDirectionalLight is recommended for automatically updating directional light's position and size.



Ambient Light

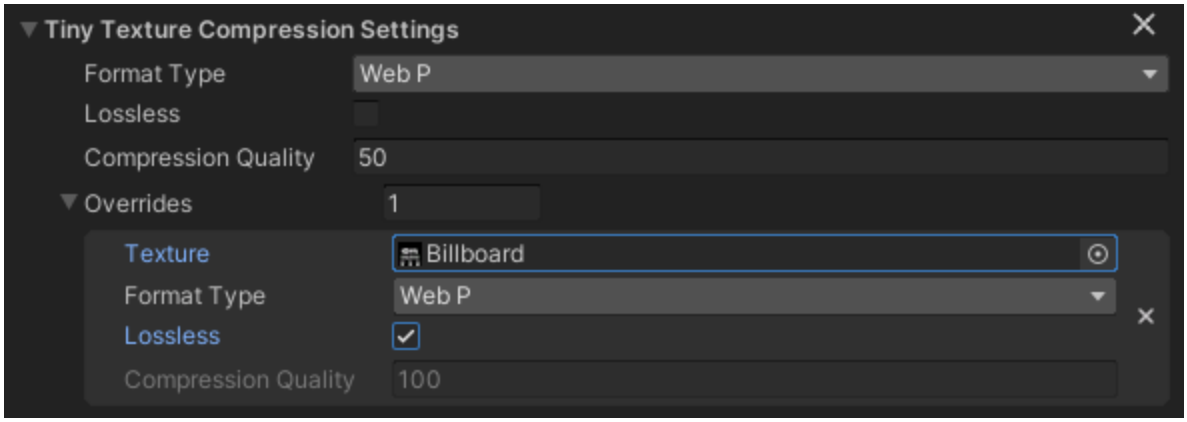
The ambient light is parameterizable per scene/subscene. Before updating the ambient light make sure to open and activate the scene you want to edit. Double click on the subscene to activate it and In The Editor go to Window - Rendering - Lighting. In the environment tab adjust the ambient light color and intensity, only solid colors are supported at the moment.

Texture Compression Settings

Textures are now exported as Webp or PNG (Webp lossless by default). Webp is a image compression format natively supported by most web browsers except on Safari, iOS Safari and IE where we do have a fallback to decode to RGBA. On other platforms, we decode webp images to RGBA on the runtime.

Currently there is a Build Component (TinyTextureCompressionSettings) to specify the format globally and/or per texture.

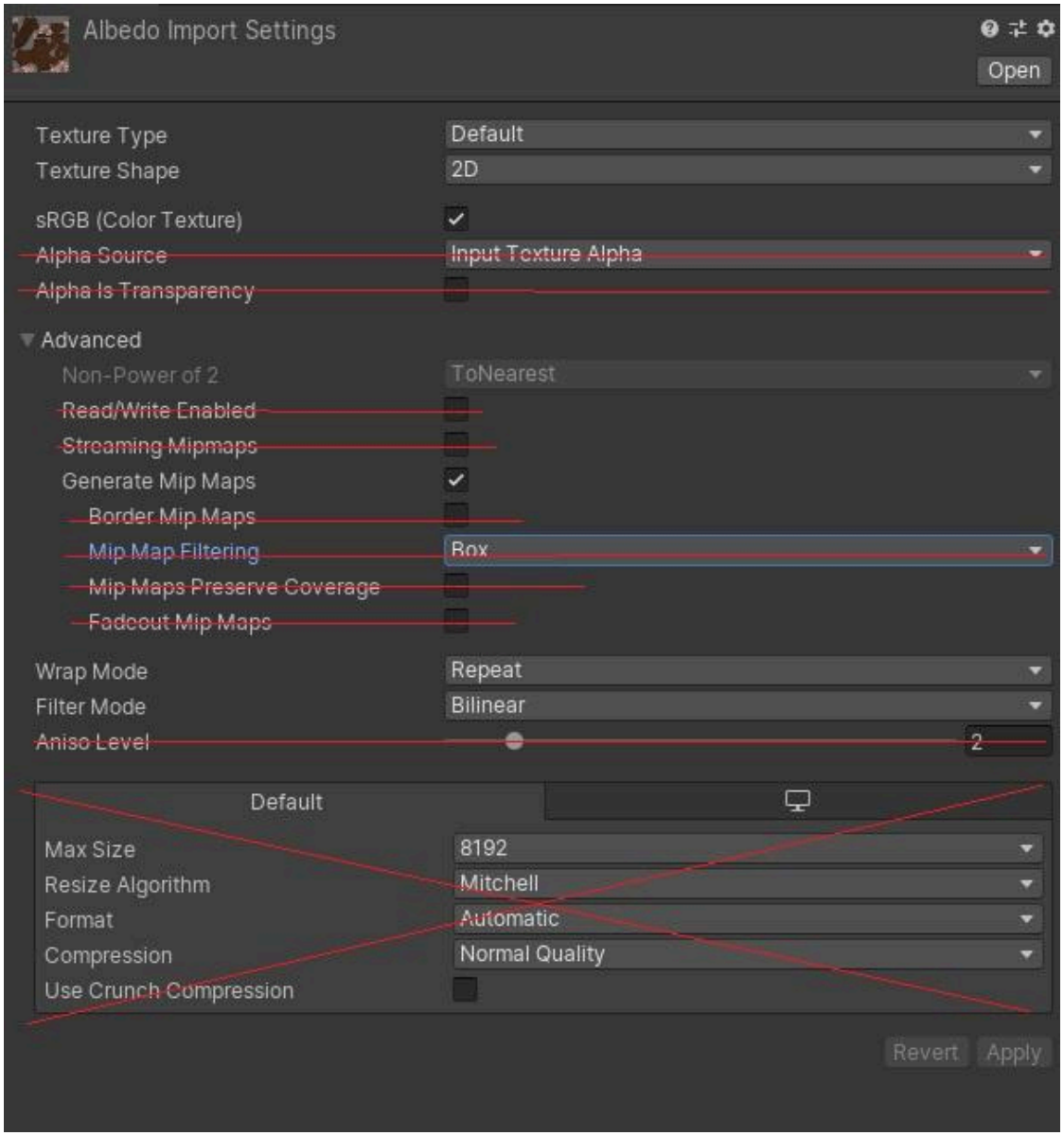
The options for the webp compression are lossless if you want to preserve the quality or lossy with a quality parameter (from 0 to 100) to compress with a higher ratio.



Texture Importer Settings:

NPOT textures with mipmaps or repeat wrap mode are not supported.

For the filter mode only Point and Trilinear are supported, the default filtering value is Linear.

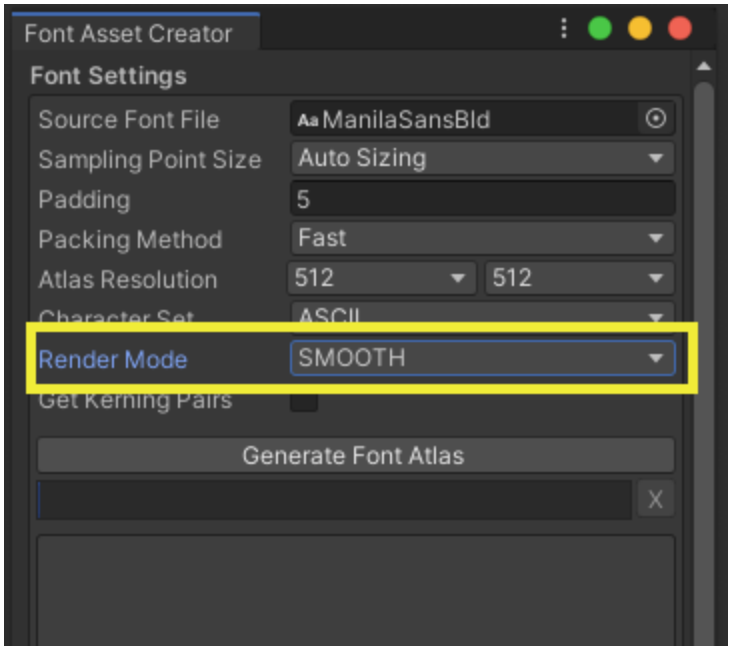


Text Rendering

Text can be rendered in the scene. TextMeshPro is used for creating the font assets, but TextMeshPro is *not* used for authoring. Functionality is very limited in this initial release.

Creating a compatible font asset

In order to create a font asset compatible with the Text Renderer component, first create a TextMeshPro Font Asset (Window > TextMeshPro > Font Asset Creator). Select your font (TTF/OTF/etc.). The “Render Mode” *must be SMOOTH or SDF(AA)* currently:



Configure the other settings as needed, and press Generate Font Atlas. Then press Save. Give the font asset a name.

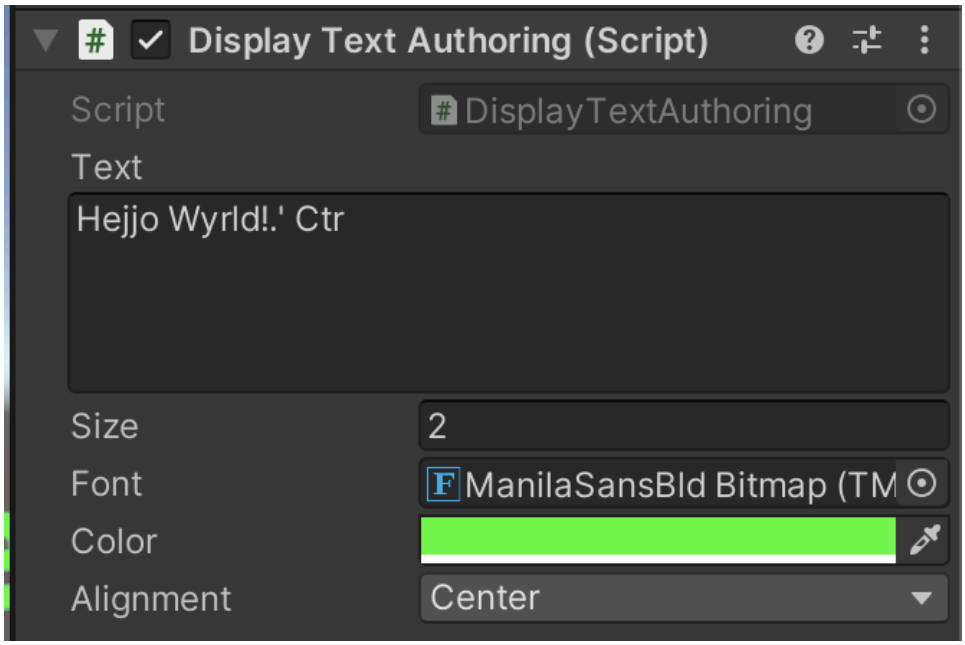
Displaying text in a scene

The following limitations currently exist, but will be removed in the near future:

- Text does not use a RectTransform. Text is drawn at a 3D world position, and can be left, center, or right justified at that position. There is no anchoring.
- Text cannot span multiple lines.
- Text cannot auto-size to fit a particular area.
- Text has only a single solid fill color. No separate outline color is supported.

Creating text in the editor

A new “Text Renderer Authoring” component is available:



This exposes just the features that are available at runtime. In the editor, it creates a dynamic mesh for display. It does *not* create a hybrid renderer mesh; the text is only visible while a subscene is opened for editing.

Creating text dynamically at runtime

At runtime, you need an entity with a TextRenderer component and a TextRendererString buffer component. The TextRenderer needs a reference to the font “material”, which will be created by conversion from a font asset. You’ll need some way to find this entity. (For example by having an authoring component with fields that link to your fonts. Eventually fixed by addressables or similar.)

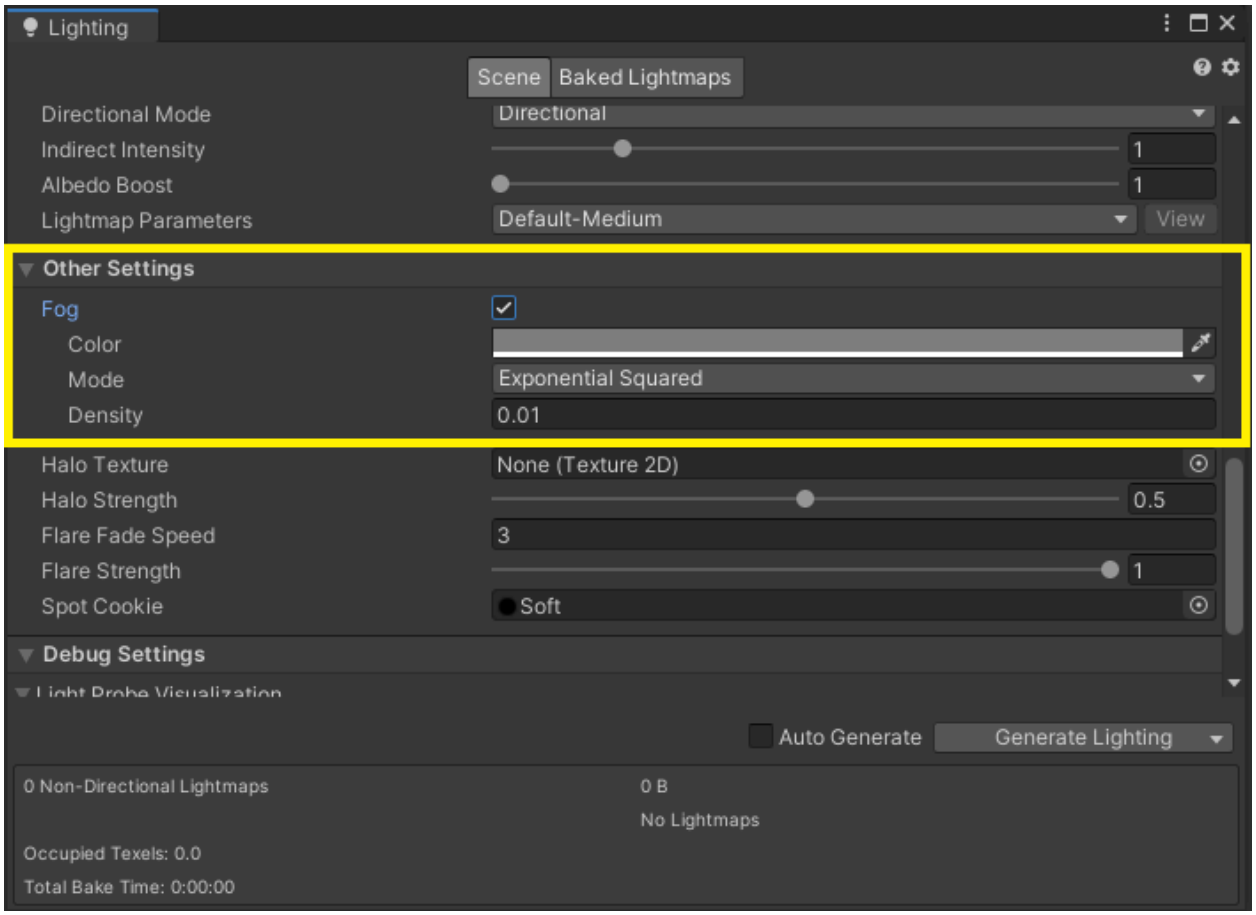
There are helper functions on `TextLayout` that can help you set the DisplayText buffer element from a string: `TextLayout.SetEntityTextRendererString(EntityManager, ui.LapLabel, $"LAP {playerLap} / {race.LapCount}");`

Changing text at runtime

If you change either the string or text properties, you must add the TextRendererNeedsUpdate component to cause the mesh & data to be regenerated. If you use SetEntityTextRendererString, it will add this component for you automatically.

3D FOG

Tiny supports adding distance-based fog to a scene using the Unity Editor or by adding it from script. To add fog from the Editor, make sure that your Subscene is currently selected (see “Scene Setup” in the Getting Started Guide). Open the Lighting Settings (Window > Rendering > Lighting Settings) and scroll down to Other Settings.



All fog settings are supported in Tiny. See the Lighting Window docs for descriptions of these settings.

Fog can be added from script by adding the Fog component to the scene Entity.

2D

To support the development of 2D Projects and 2D instant games that are small in size and both load and run quickly on mobile and web platforms, Project Tiny ships with the following foundation set of 2D features:

- Sprite
- Sprite Renderer
- Runtime 2D rendering pipeline

Once you have installed the required Editor version and third party tools, download the pre-configured project template `Tiny2D` from the ProjectTinySample repository. This project comes with Assets, Scripts and a Scene set up for Project Tiny development, so that you can go straight into developing your new Project Tiny application. In addition, two 2D game demos created with Project Tiny are available in the repository - TinyGems and TinySpaceship.

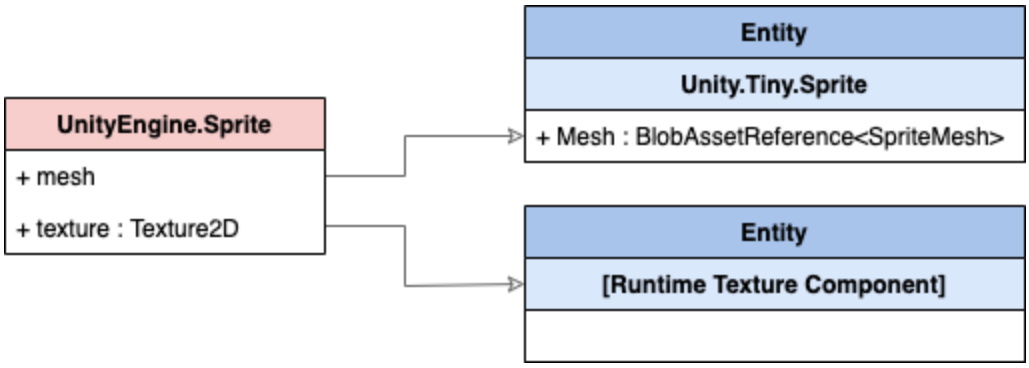
GameObject conversions

When working with DOTs, GameObjects located inside SubScenes are converted into their Entity Component System (ECS) counterparts. This converted data is used during runtime in the DOTs systems. Refer to ECS' GameObject Conversion documentation for more details.

In the section below, we have listed the conversion of the components and assets in Tiny 2D.

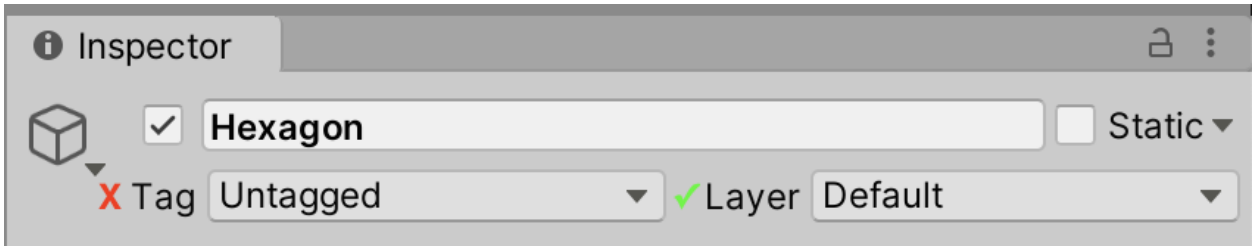
Sprite

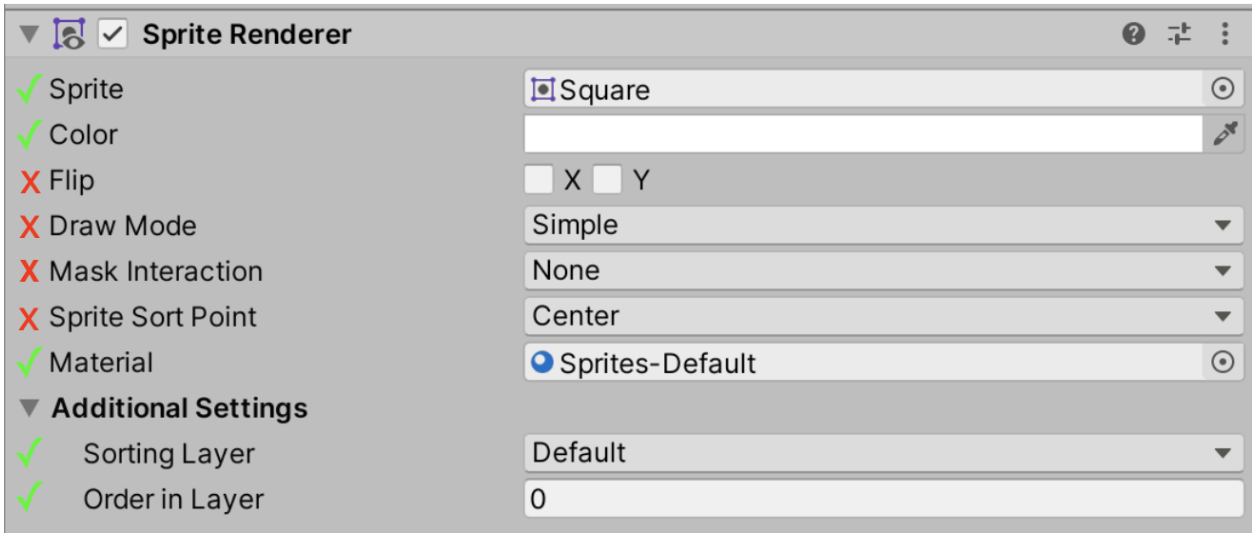
Here is a visual representation over how the `UnityEngine.Sprite` properties map over in ECS:



Sprite Renderer

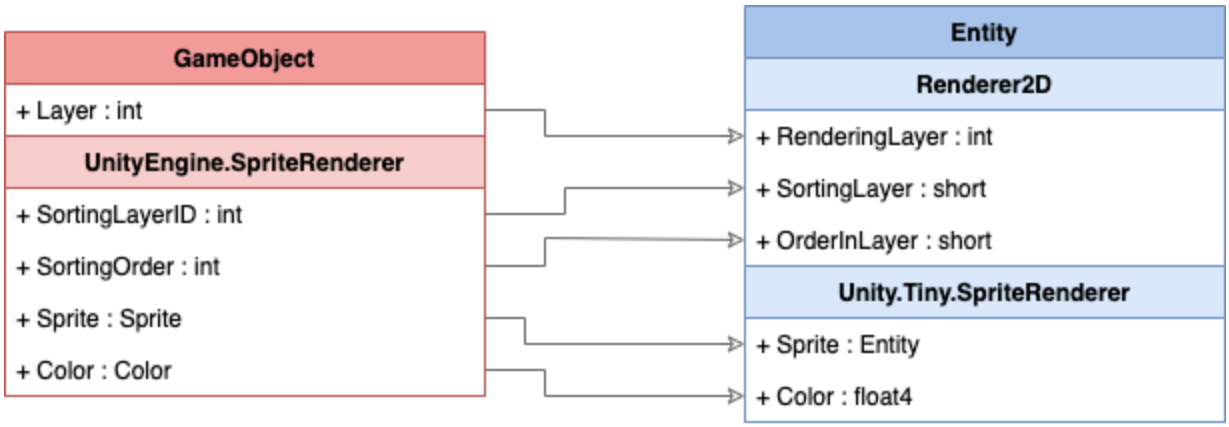
Only the following Sprite Renderer properties with green check marks are converted over to the ECS environment:



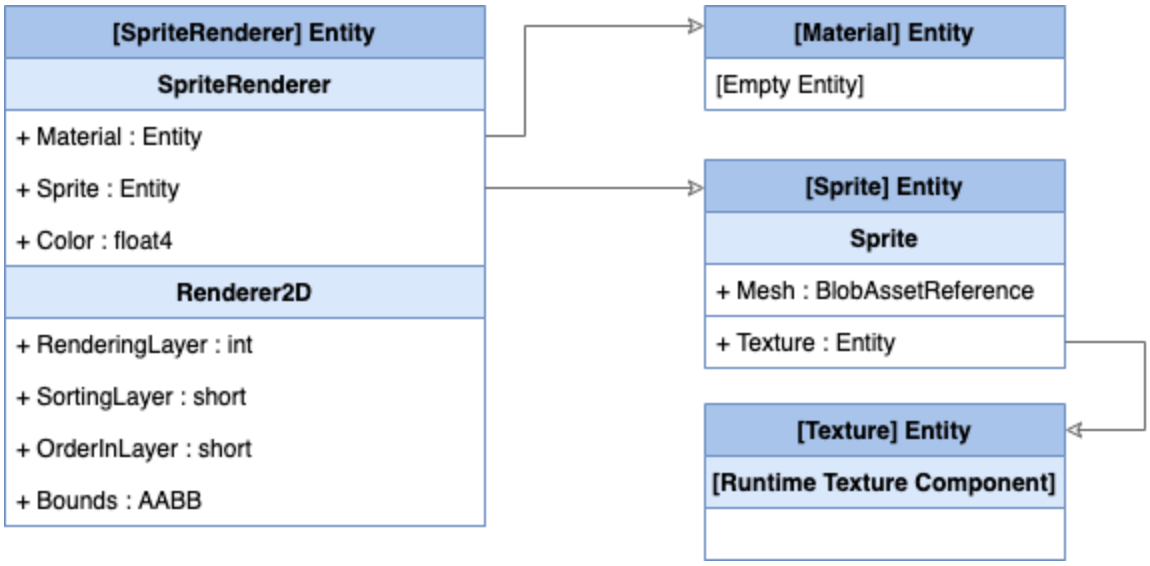


Property	Conversion details
Layer	Converted as is. Is used for selective rendering from camera.
Sprite	The Sprite data is converted and the `Unity.Tiny.SpriteRenderer` stores a link to the `Unity.Tiny.Sprite` Entity
Color	Converted into a `float4`.
Sorting Layer	Converted into a `short`.
Order in Layer	Converted into a `short`.

Here is a visual representation over how the `UnityEngine.SpriteRenderer` properties map over in ECS:



And here is a visual representation of how the different 2D rendering Entities and their components interact with each other:



2D Entities Physics

2D Entities Physics is a package that provides tools to work with 2D physics in Project Tiny. The following features are partially supported by 2D Entities Physics:

- Primitive Colliders
- Compound Collider
- Rigidbody 2D
- Physics broadphase
- Broadphase querying API

Refer to the 2D Entities Physics package documentation for more information.

Particle System (3D)

Overview

The Tiny particle system implementation mirrors a subset of the features available in Unity's Built-in Particle System (aka Shuriken). This allows users to add simple particle systems to Tiny applications using the Unity Editor or by creating them from script. When created from the Editor, the Unity particle system object is converted into Entities and consumed by Project Tiny/DOTS Runtime.

Creating Particle Systems in the Editor

Particle systems can be created in the Editor the same way they can for Unity: by adding a Particle System component to your scene and modifying its properties in the Inspector window (see Shuriken usage docs).

Gotchas:

- **Unsupported Features**
  Although only a subset of the Shuriken modules and properties are supported, the Inspector window will display all options in the full Shuriken feature set. In a few cases, the default values for an unsupported property will not match the default value in the Editor, which may cause your preview in the Scene view to not match your Tiny application. While tuning your particle effects, we recommend frequently referring to the list of supported features in Tiny and running your Tiny application (using “Build & Run”) to avoid relying on unavailable features.

- **Scene Hierarchy**
  Like all other Unity objects that get converted and consumed by Tiny, particle systems must be added into the Subscene as opposed to the top-level scene (see “Scene Setup” in the Getting Started Guide).

- **Play in Editor**
  Tiny particles do not support Playmode (i.e. they will not render in the Game view). You can preview the particles in the Scene view, but it is possible that there will be differences between the preview and Tiny application due to the Editor using the DOTS Hybrid Renderer. The most reliable way to test particles is to use “Build & Run” to view them in your Tiny application.

Creating Particle Systems from Script

Tiny particle systems can be created from script by adding the ParticleEmitter component, an emitter source shape component (e.g. EmitterConeSource), and the ParticleMaterial component to an Entity. Refer to the Project Tiny API docs for additional components. Be sure to reference the docs that correspond to the version of Project Tiny you are using since the particle system implementation is under heavy development.

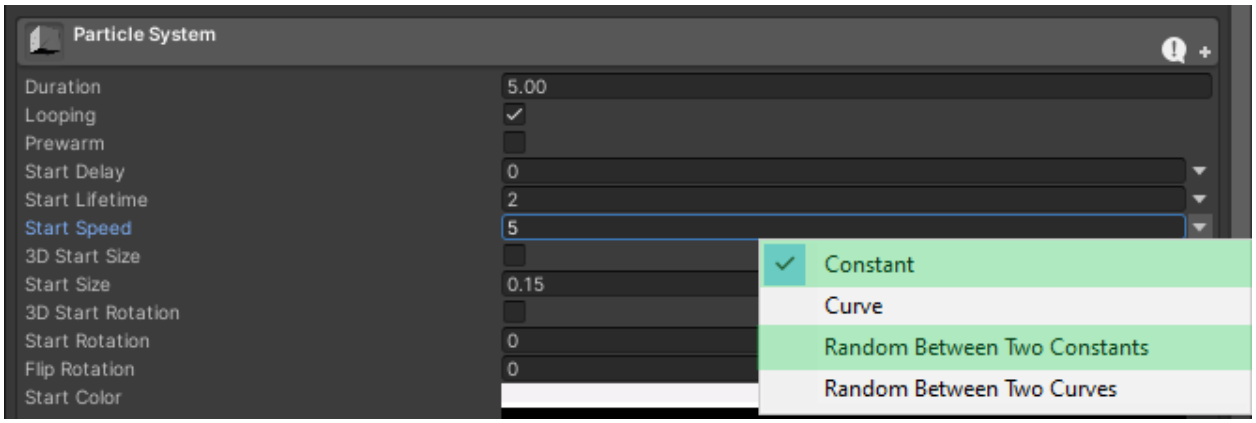
Supported Features

The Tiny particles properties mirror a subset of the Shuriken properties. The Shuriken properties are broken down into groups called “modules”. This section will cover which properties in each module are supported in Tiny. If a module is not listed, then no properties in that module are supported currently. For unsupported properties, the default values shown in the Inspector

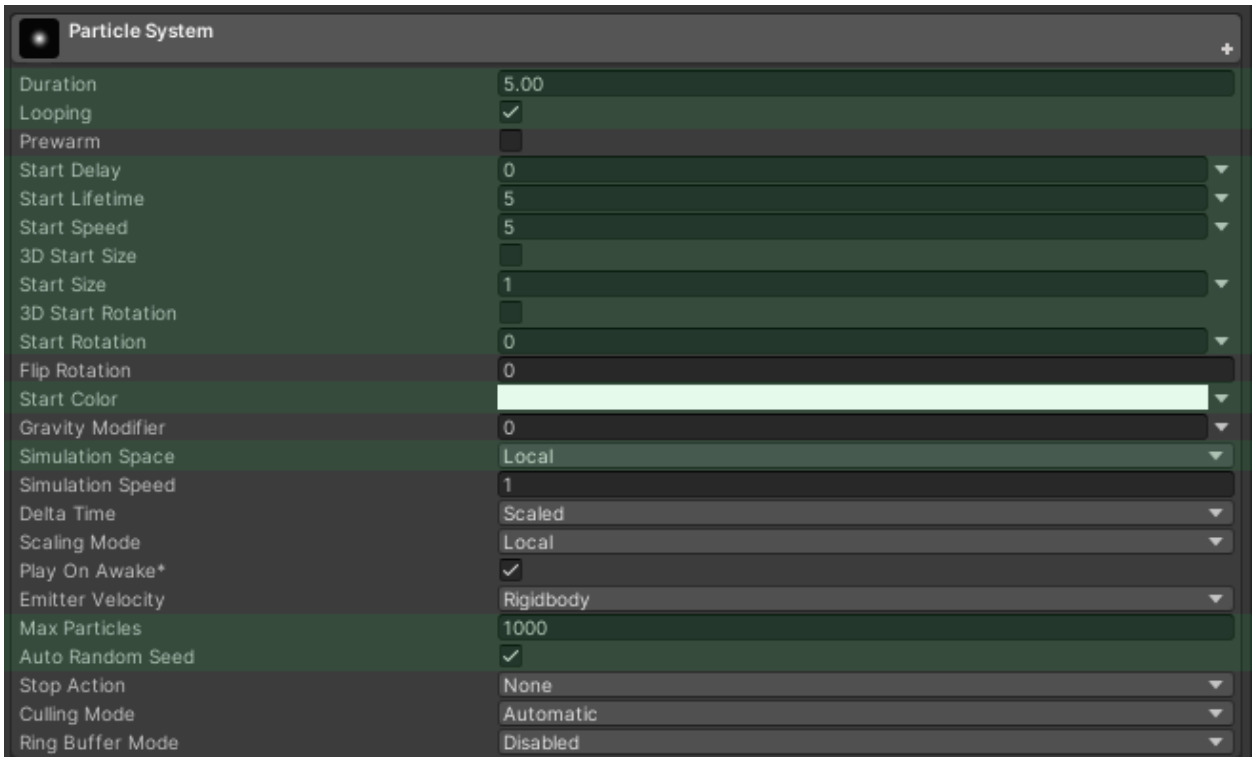
window are used unless stated otherwise. Refer to the Shuriken modules docs for descriptions of the properties.

Varying Properties Over Time

Many of the Shuriken properties in the different modules support a dropdown menu that allows you to select how this property should vary over time (see Shuriken usage docs). For all properties that support this dropdown, only **Constant** and **Random Between Two Constants** are supported.



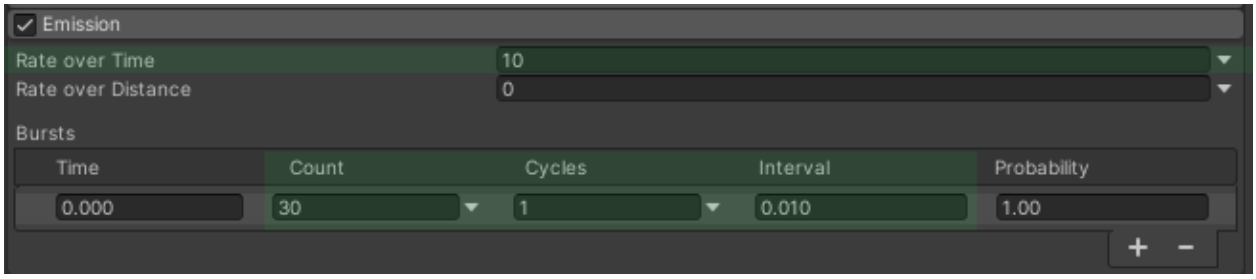
Main Module



Supported Properties

- Duration
- Looping
- Start Delay
- Start Lifetime
- Start Speed
- Start Size/Enable 3D Start Size
- Start Rotation/Enable 3D Start Rotation
- Start Color (**Color** or **Random Between Two Colors**)
- Simulation Space (**Local** or **World**)
- Max Particles
- Auto Random Seed/Random Seed

Emission Module

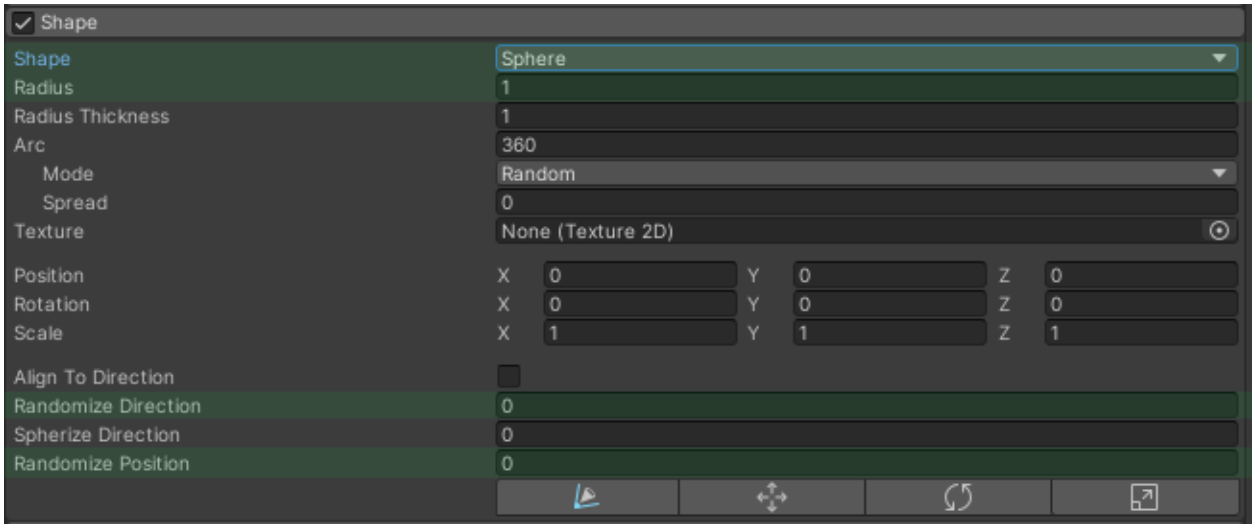


Supported Properties

- Rate over time
- Bursts (only one burst per particle system)
- Burst Settings (**Count**, **Cycles**, **Interval**)

Shape Module

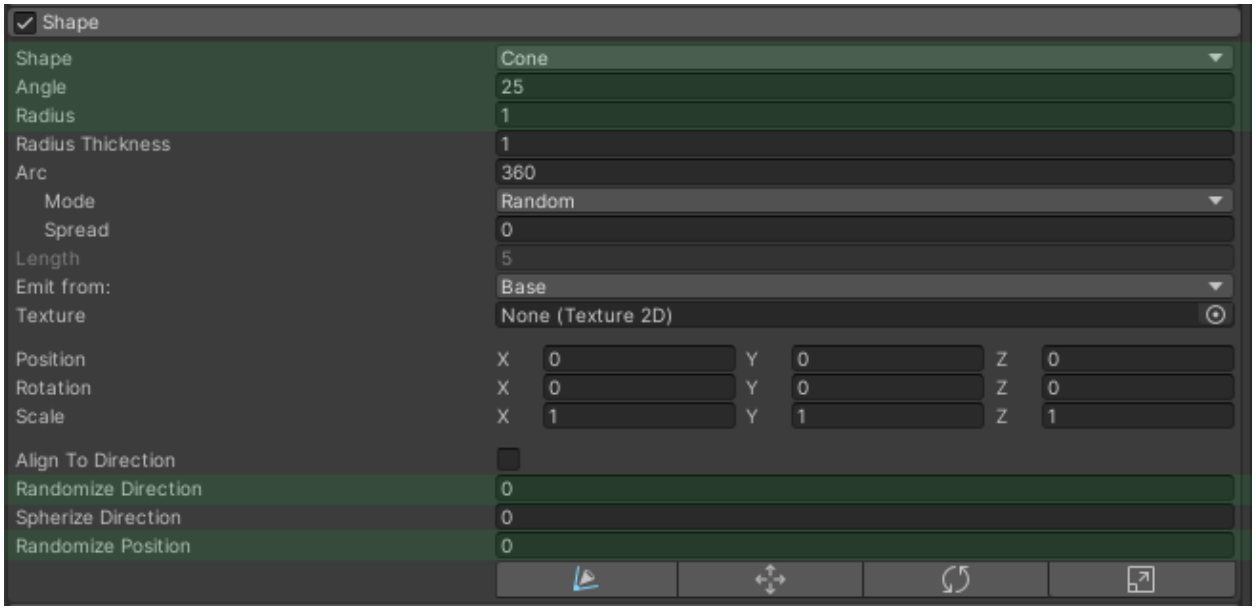
Sphere, Hemisphere



Supported Properties

- Radius
- Randomize Direction
- Randomize Position

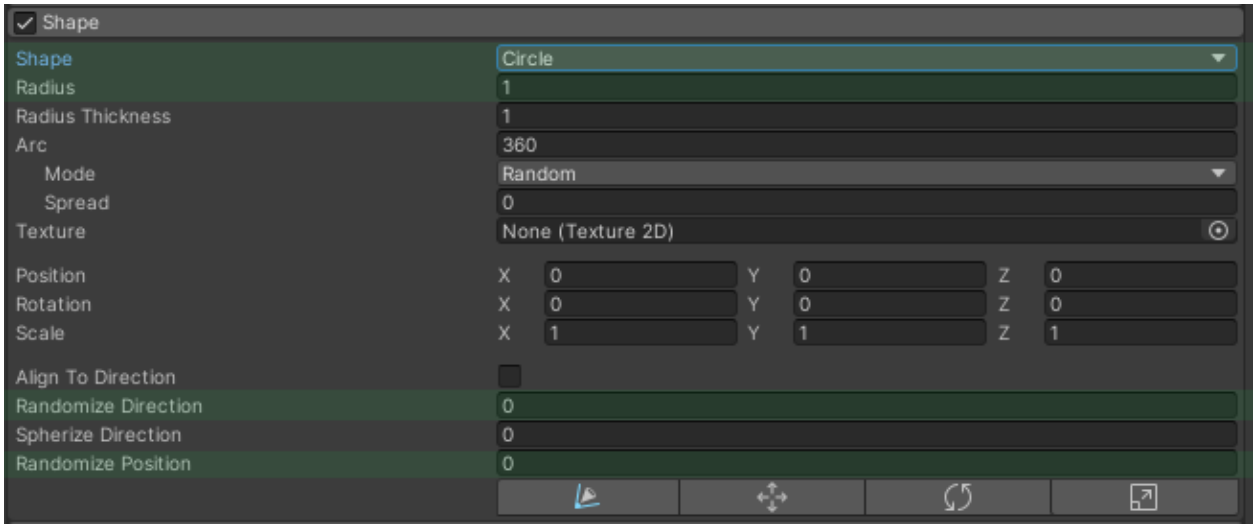
Cone



Supported Properties

- Angle
- Radius
- Randomize Direction
- Randomize Position

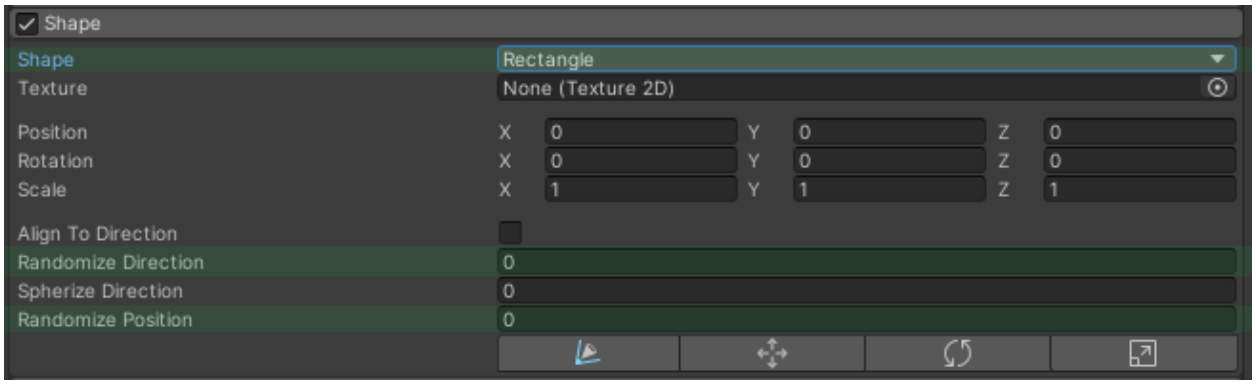
Circle



Supported Properties

- Radius
- Randomize Direction
- Randomize Position

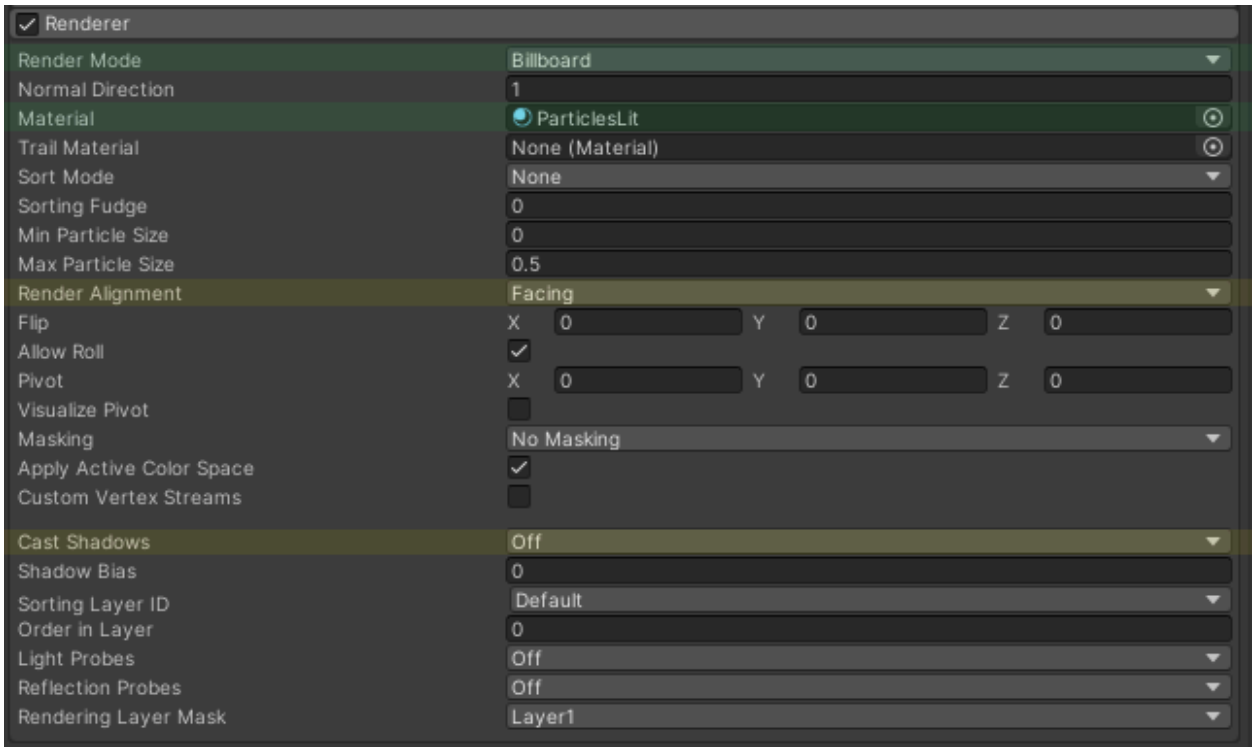
Rectangle



Supported Properties

- Randomize Direction
- Randomize Position

Renderer Module



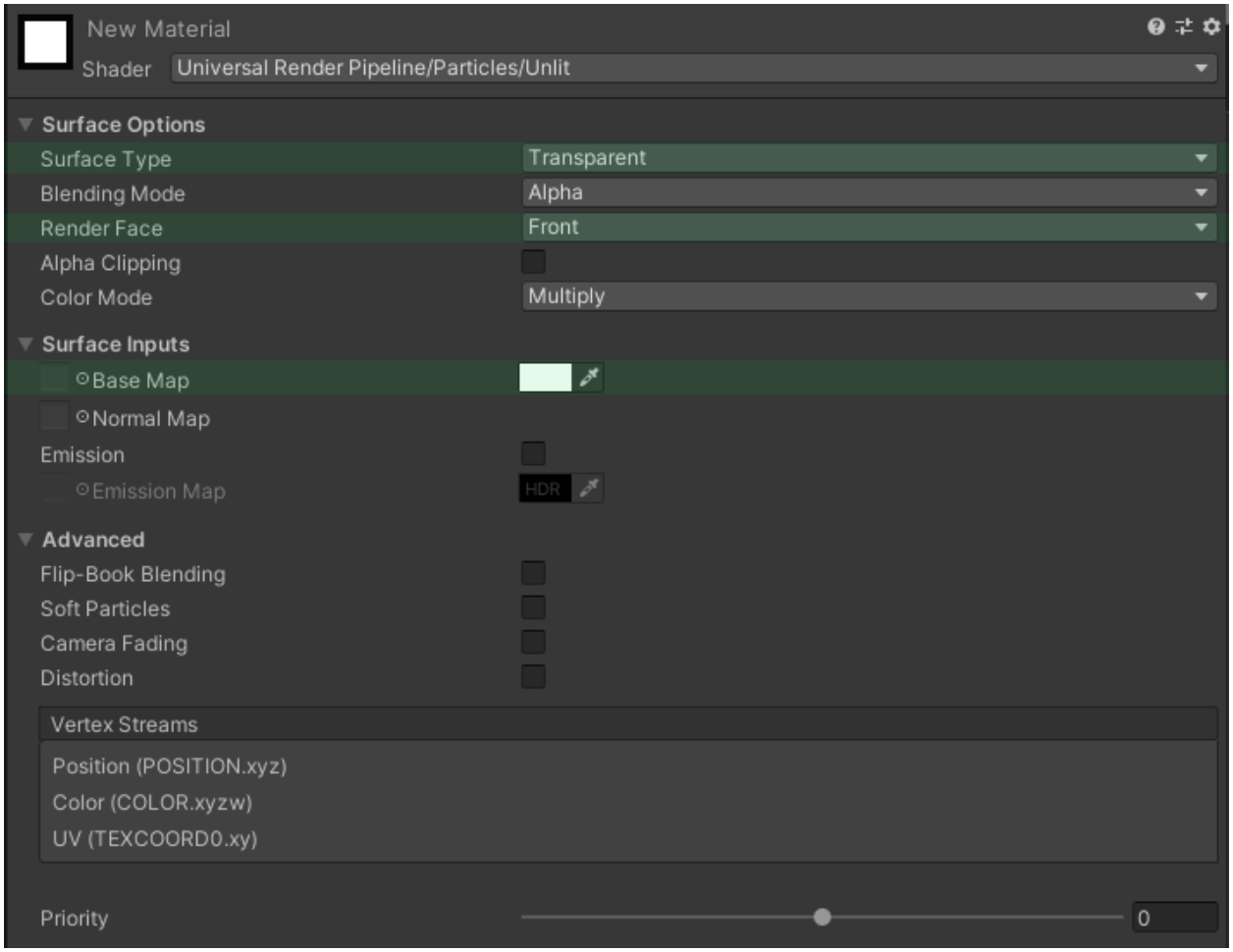
Supported Properties

- Render Mode (**Billboard** or **Mesh**)
- Material (see below)
- Render Alignment
    - **Facing** if **Render Mode** is set to **Billboard**
    - **Local** if **Render Mode** is set to **Mesh**
- Cast Shadows
    - **On** if **Material** is **Opaque**
    - **Off** if **Material** is **Transparent**

Material

Only materials that use the **Universal Render Pipeline/Particles/Unlit** or **Universal Render Pipeline/Particles/Lit** shaders are supported. Refer to the Universal RP docs for descriptions of the following material properties.

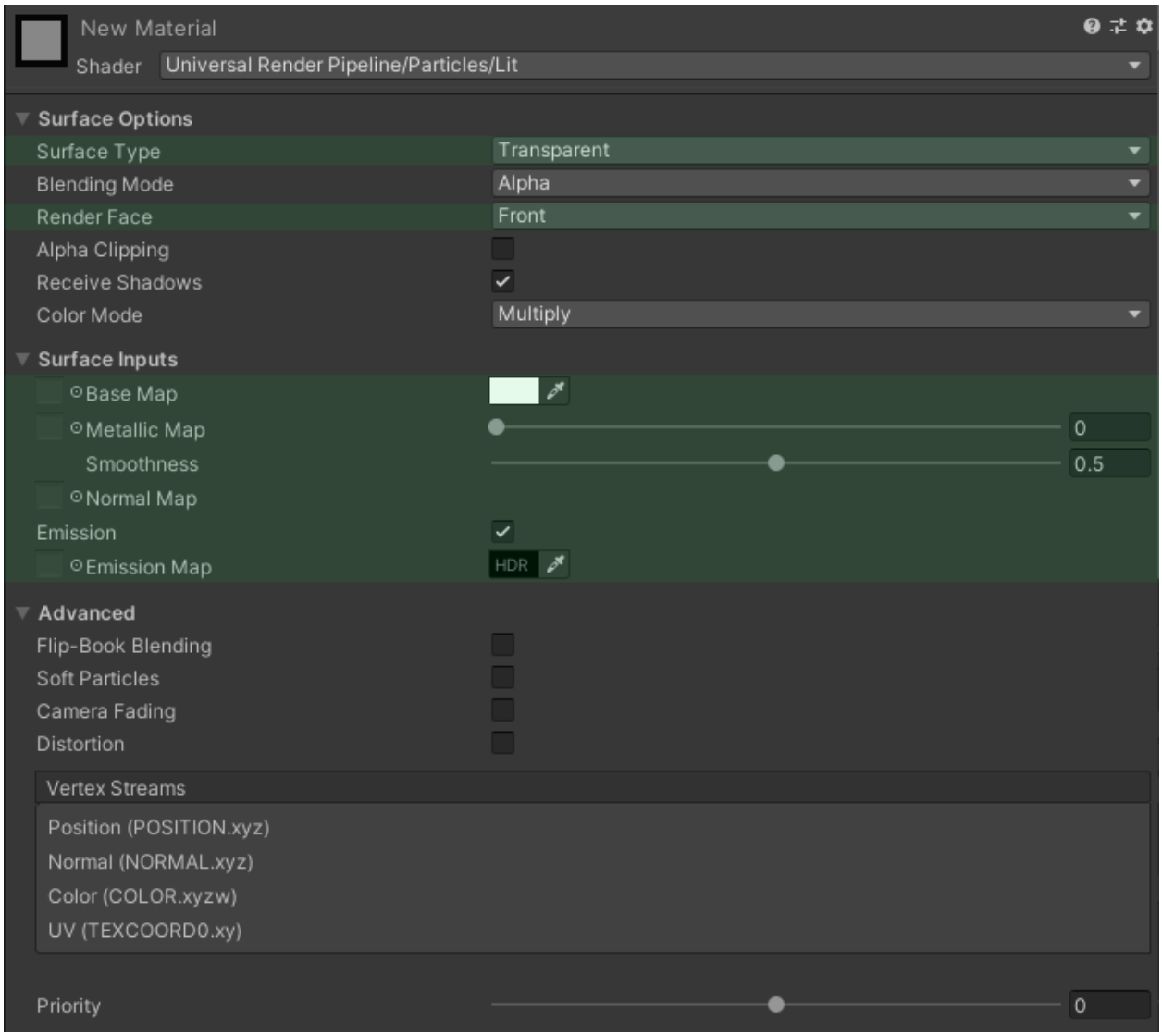
Universal Render Pipeline/Particles/Unlit



Supported Properties

- Surface Type
- Render Face
- Base Map (texture and color)

Universal Render Pipeline/Particles/Lit



Supported Properties

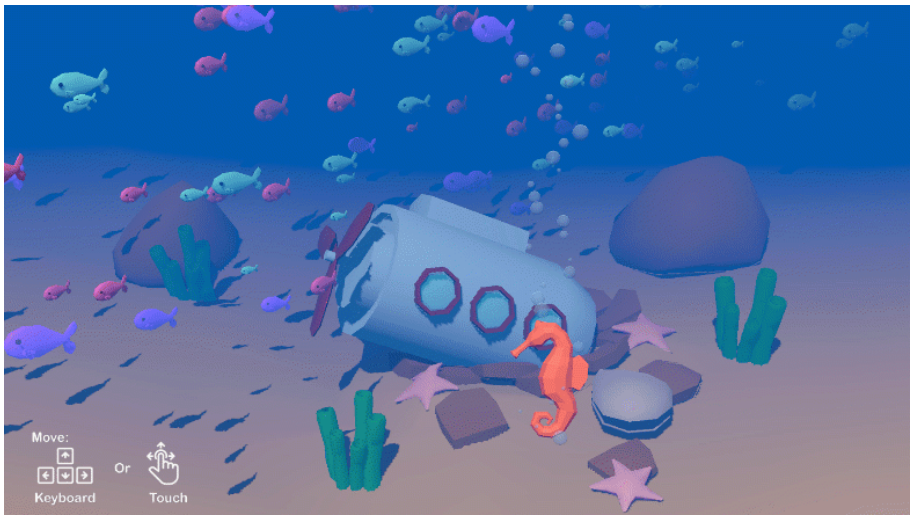
- Surface Type
- Render Face
- Base Map (texture and color)
- Metallic Map (texture, metallic constant, and smoothness constant)
- Normal Map (texture and multiplier)
- Emission Map (texture and color)

Summary of the Modules and properties supported at the moment:

- Particle System General Settings
    - Duration
    - Start Lifetime
    - Start Speed
    - Start Size/Enable 3D Start Size

        - Start Rotation/Enable 3D Start Rotation
        - Start color
        - Gravity modifier
        - Simulation Space "Local" or "World"
        - Max Particles
- Emission Settings
    - Rate over time
    - Bursts (only one burst per particle system)
    - Burst Settings (Count, Cycles, Interval)
- Shapes Settings
    - Cone (Radius, Angle)
    - Circle (Radius)
    - Rectangle
- Renderer Settings
    - Render Mode - Mesh and Billboard are supported
        - Mesh uses Local Render Alignment and Billboard uses Facing Rendering Alignment
    - Material - only Universal Render Pipeline/Particles/Unlit and Universal Render Pipeline/Particles/Lit are supported
- Other notes
    - Constants and random value between two constants are supported.
    - Animation curves are not yet supported.

We recommend you explore the TinyAquarium sample in https://github.com/Unity-Technologies/ProjectTinySamples which demonstrates how to set up particles.



Shaders

Custom shader support for shaders written in a combination of ShaderLab and HLSL is in progress. Currently there is initial support for custom shaders that have fixed parameters. More specifically, shaders must have material properties that must match the ‘Universal Render

Pipeline/Lit' shader, and vertex attributes and uniforms that match the Tiny lit shader. **The fixed parameters restriction is temporary and will not be addressed any further in this document**.

Tiny shaders are compiled using the Unity Shader Compiler during conversion and the compiled shader data is formatted so it can be used by Project Tiny Renderer (based on Bgfx) . Supported platforms are DX11/DX12, OpenGL, OpenGL ES 2.0, and Metal. Vulkan support is coming soon.

Shader Conversion

Tiny shader compilation happens when running scene conversion from the Editor. The conversion system only runs during the Tiny build if something in the scene has changed and it needs to be reconverted. Modifying a shader used in the scene and reimporting it in the Editor (reimport should happen automatically if Auto Refresh is enabled) should trigger the scene conversion/shader compilation when you build. You can also force the conversion to run by doing DOTS -> Clear Entity Cache(s) in the Editor before building.

Supported Features

Tiny shaders can be written like Unity shaders with some exceptions. Please refer to the following sections for differences between Tiny and Unity shaders. Refer to Unity shader documentation for any other guidance on shader authoring.

Uniforms

Supported Types

Uniform types are restricted to those supported by Bgfx . Matrix 3x3 uniforms are not supported because they are not supported in Unity and the cross compilation of them is not well tested.

Unity Built-in Uniforms

A limited set of the Unity built-in uniforms are supported. The supported uniforms are the ones that have a one-to-one mapping to Bgfx built-in uniforms. If an unsupported uniform is used by a shader, it will be treated as a custom uniform and Bgfx will expect that it will be set by the user. Below is the list of supported Unity uniforms and the corresponding uniforms in Bgfx.

Unity	Bgfx
“UNITY_MATRIX_M” or “unity_ObjectToWorld”	“u_model”
“UNITY_MATRIX_V” or “unity_MatrixV”	“u_view”
“UNITY_MATRIX_I_V” or “unity_MatrixInvV”	“u_invView”

“UNITY_MATRIX_P” or “glstate_matrix_projection”	“u_proj”
“UNITY_MATRIX_VP” or “unity_MatrixVP”	“u_viewProj”
“UNITY_MATRIX_MV” or “unity_MatrixMV”	“u_modelView”
“UNITY_MATRIX_MVP” or “unity_MatrixMVP”	“u_modelViewProj”

Include “UnityShaderVariables.cginc” in your shader to have access to these variables. It is worth noting that the mapping of uniforms from Unity to Bgfx is done by name, so declaring your own uniform that has the same name as one of the supported Unity variables will result in the built-in Bgfx uniform being used. This can be advantageous if a small number of built-in uniforms is needed by your shader because it can result in a reduced uniform buffer size on some platforms.

Uniform Buffers

Unity shaders allow for grouping uniforms into buffers on platforms that support this via the CBUFFER_START/CBUFFER_END macros (see Predefined Macros). Please note that there is no performance advantage to grouping uniforms that are only used within a single shader stage (e.g. vertex) in a Tiny shader because all uniform buffers are consolidated into a single buffer per shader stage.

Samplers

Decoupled samplers and textures (i.e. reusing a sampler with multiple textures) is not supported. Inline sampler states are not supported. See Sampler States.

Sampler bind locations (indices) are implicitly assigned in the order that the samplers are declared in the shader starting at 0. This can cause confusion since the index must be specified when binding the sampler and samplers are omitted if they are not used by the shader. We have plans to improve this experience in the future.

Vertex Attributes

All vertex attributes supported by Unity shaders are supported in Tiny shaders. The following is the list of supported vertex shader input semantics:

POSITION
TANGENT
NORMAL
COLOR
TEXCOORD0
TEXCOORD1
TEXCOORD2
TEXCOORD3

TEXCOORD4
TEXCOORD5
TEXCOORD6
TEXCOORD7
BLENDWEIGHTS
BLENDINDICES

Include Files

Shader include files are supported. Include files can be

- TextAssets created in the Editor
- Internal files that are not recognized by the Editor (e.g. *TinySamples\Packages\com.unity.tiny\Unity.Tiny.Rendering.Native\shadersrc~\common\simplelit.cginc*)
- Unity built-in include files (see below)

Shaders are only recompiled when the scene they are used in gets reconverted. Modifying an include file that is recognized as an Asset in the Editor will trigger the reconversion. If modifying an internal include file that is not recognized as an Asset by the Editor, you must clear the entities cache (in the Editor: DOTS -> Clear Entity Cache(s)) to force the shader to be recompiled. For more details on the shader conversion see the Conversion section.

Unity Built-in Include Files

Unity built-in include files can be used in shaders, but please be aware that not everything in these files has been tested with Tiny shaders. Be aware of any functions in the include files that use unsupported built-in uniforms (see section Unity Built-in Uniforms).

ShaderLab

Shaders are written in a combination of ShaderLab and HLSL but only the HLSL snippets are used in Tiny. More details on supported ShaderLab constructs (e.g. Properties) will be added once custom uniforms are supported.

Variants

Currently there is no support for shader variants so shader keywords will be ignored. Shader code can be shared across shaders using include files.

ShaderGraph

ShaderGraph shaders are not supported but may be in the future.

Platform Specific features

Game save system

The game save system allows data to be saved to a file in persistent storage, so that data lives beyond the lifetime of each gaming session. Each piece of data has a string key and a data value. All of the game save functionality exists in the Unity.Tiny.GameSave namespace.
It's available on iOS/Android/Windows/Mac and Web.

Read from Persistent Storage
The GameSaveSystem calls into platform-specific file functions that read from persistent storage. Tiny's system information APIs can be used to learn where to read/write the game save file on a particular platform.

At startup, the GameSaveSystem will read from a file. This operation is initiated by creating a GameSaveReadFromPersistentStorageRequest component. That component includes the file path to use to read in the game save file. The GameSaveSystem looks for these components in its OnUpdate function and reads in the data into a local data structure that it manages. Your game code can then read from and write to this data structure through the GameSaveSystem read/write APIs.

Read/Write APIs and Supported Data Types
There are several functions that can be used to read from and write to the current game save file. We currently support many data types, including all C# basic types, FixedString64, FixedString128, Unity ECS components, and Unity ECS dynamic buffer components.

Here are some functions that show how to read some common types from the game save file:
public unsafe GameSaveResult Read<T>(FixedString64 key, ref T value) where T : unmanaged
public unsafe GameSaveResult Read<T>(FixedString64 key, ref T value, in T defaultValue) where T : unmanaged
public unsafe GameSaveResult ReadDynamicBuffer<T>(FixedString64 key, ref DynamicBuffer<T> value) where T : unmanaged

If the read is successful, GameSaveResult.Success if returned; if not, you may get GameSaveResult.NotFound if the key name look-up fails, or other possible errors.

After modifying these values in your game code, they can be written to the game save file via:
public unsafe GameSaveResult Write<T>(FixedString64 key, T value) where T : unmanaged
public unsafe void WriteDynamicBuffer<T>(FixedString64 key, ref DynamicBuffer<T> value) where T : unmanaged

There are also ReadBytes and WriteBytes functions that allow you to work with a contiguous block of bytes. This may be helpful if you want to serialize the data yourself or add support for a type that the existing system does not support. These function prototypes are:
public unsafe GameSaveResult ReadBytes(FixedString64 key, void data, int length);*
public unsafe void WriteBytes(FixedString64 key, void data, int length);*

Write to Persistent Storage
At an appropriate time, after some modifications have been made, the game save data can be written back out to a file. This is done by creating a

GameSaveWriteToPersistentStorageRequest component. This component includes the file path to use to write out the game save file.

Data Migration considerations

After the initial release or your game or app, you may need to modify some types that have been stored in your user's game save files. The system is designed to be able to automatically handle the addition and removal of fields within any of our supported types. Any modifications more complex than that will have to be handled by the user.

If all your persistent data is stored in basic types by your code, then data migration is easy and automatic. The system doesn't actually do anything. For more complex types, like components, the GameSaveSystem will store detailed type information in the game save file.

When v2.0 of your game is released, any fields that were removed from a component will just be skipped over when that component is read. New fields will not be found in the game save file, so your code will need to set them to a reasonable default, or pass in the default to the version of our read APIs that take default values.

On each read request, the system checks to see if the type being read has been modified. If it is different, then the system will perform an upgrade process for that one piece of data. It knows each field's name, type, and offset, and can handle all fix-up related to fields that have been added, removed, or whose offset has changed. After this update is performed, the new type information will be written to the game save file, so the next time this data is looked up, the fast code path can be taken. It should be noted that if ReadBytes and WriteBytes are used, the GameSaveSystem has no knowledge of the underlying type, so it cannot perform any kind of data migration. The responsibility for updating falls to the user in that case.

Multiple GameSave Files

It is possible to have multiple game save files (up to 16). When creating GameSaveReadFromPersistentStorageRequest and GameSaveWriteToPersistentStorageRequest components, the gameSaveIndex in those components must be set to a value from 0-15 to indicate which game save index to use for that particular game save file. By default, these components just use zero, and so you are always reading/writing to the same game save slot in memory.

When working with multiple game save files, you also need to set the current game save index. This is done by setting GameSaveSystem.gameSaveIndex to the index of the game save file you want to use. This needs to be done before Read<T> and Write<T> calls to read/write each data item.

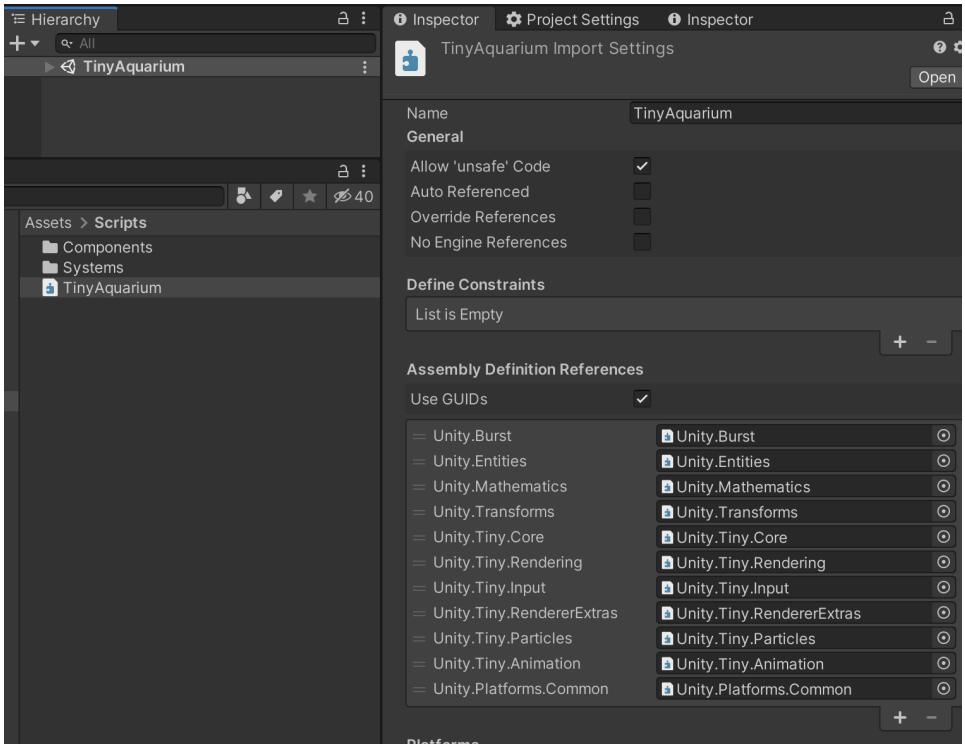
Suspend, Resume & Quit Events (iOS, Android)

With preview 0.24, we added support for Suspend, Resume & Quit Events on iOS and Android, which allow you to add

Internal implementation notes:
In **Unity.Platforms** workspace (defined in **Unity.Platforms.Common** assembly) there are two events and events handlers declared which can be used to notify systems if the application is suspended/resumed or closed:

```
	// sent when application is suspended or resumed
	// Suspend value is true in case of suspend, false for resume
	public struct SuspendResumeEvent
	{
		public bool Suspend { get; }
	}
	// sent when application is closed
	public struct QuitEvent
	{
	}
	public delegate void SuspendResumeEventHandler(object sender,
SuspendResumeEvent evt);
	public delegate void QuitEventHandler(object sender, QuitEvent evt);
	public static event SuspendResumeEventHandler OnSuspendResume;
	public static event QuitEventHandler OnQuit;
```

To make systems react to these events in an application you need a reference to **Unity.Platforms.Common** must be added to your application asmdef file.



The following code should be added to OnStartRunning/OnStopRunning methods of a system (typically some initialization system):

```
using Unit.Platforms;
	protected override void OnStartRunning()
	{
			PlatformEvents.OnSuspendResume += OnSuspendResume;
			PlatformEvents.OnQuit += OnQuit;
	}
	protected override void OnStopRunning()
	{
			PlatformEvents.OnSuspendResume -= OnSuspendResume;
			PlatformEvents.OnQuit -= OnQuit;
	}
```

And these methods should be implemented in the System:

```
	public void OnSuspendResume(object sender, SuspendResumeEvent evt)
	{
			// process suspend/resume based on evt.Suspend value
	}

	public void OnQuit(object sender, QuitEvent evt)
	{
			// process application closed event
	}
```

Possibilities to fire QuitEvent are very limited on both iOS and Android. On iOS this event is being called only if a user closes the currently active app, on Android also there is no guarantee that Activity.onDestroy() method (which fires QuitEvent) will be called if an application is killed. This means that an app can be killed by system or by user and be not notified about this. Suspend event is always being fired, so application should use it to store its state in case if it would be later killed without notifying.

The way these events are implemented on both Android and iOS, handler methods are being called from the main thread between Update cycles. It's important to keep in mind that in case of Suspend and Quit events all processing should be implemented directly in handler, because after suspend/quit there would be no new game cycle initiated.

Accelerometer/Gyro (iOS & Android)

On Mobile (iOS & Android), Accelerometer/Gyro are supported, as well as controlling screen orientation. These features are demonstrated in the TinyRacing sample which also showcased the added Audio Pitch control (3D was), setup on CarAI entities to vary the sound of their engines.
All available sensors in Tiny are implemented as singleton components.
Currently available sensors components for iOS and Android:

```
public struct AccelerometerSensor : IComponentData
{
    public TimeData LastUpdateTime;
    public float3 Acceleration;
}
```

Reports the acceleration of the device measured on a device both due to moving the device around, and due to gravity pulling the device down. **Acceleration** property stores current device acceleration vector value.

```
public struct GyroscopeSensor : IComponentData
{
    public TimeData LastUpdateTime;
    public float3 AngularVelocity;
}
```

Reports angular velocity of a device. **AngularVelocity** property stores current angular velocity vector value.

```
public struct GravitySensor : IComponentData
{
    public TimeData LastUpdateTime;
    public float3 Gravity;
}
```

Reports the direction of the gravity vector relative to a device. **Gravity** property stores current gravity vector value.

```
public struct AttitudeSensor : IComponentData
```

```
{
    public TimeData LastUpdateTime;
    public quaternion Attitude;
}
```
Reports the orientation of a device. **Attitude** property stores current device orientation quaternion value.

```
public struct LinearAccelerationSensor : IComponentData
{
    public TimeData LastUpdateTime;
    public float3 Acceleration;
}
```
Reports the acceleration measured on a device due to moving the device around, without gravity effect. **Acceleration** property stores current device acceleration vector value.
For all sensors **LastUpdateTime** value is TimeData struct which stores information of when these sensors data were last updated.

Input system provides this API to use platform sensors:

public bool **IsAvailable<T>**() where T : struct, IComponentData
Returns true if a sensor with type T is available on the current hardware/platform.

public void **EnableSensor<T>**() where T : struct, IComponentData
If a sensor is available on the hardware/platform, create a singleton component for this sensor and start getting data from this sensor, does nothing if a sensor is not available.

public void **DisableSensor<T>**() where T : struct, IComponentData
If a sensor is available on the hardware/platform, stop getting data from this sensor, does nothing if a sensor is not available.

public void **SetSensorSamplingFrequency<T>**(int freq) where T : IComponentData
If a sensor is available on the hardware/platform, sets sampling frequency in Hz for this sensor, does nothing if sensor is not available. When setting sampling frequencies, there may be limits on the range of frequencies supported by the underlying hardware/platform. Also for iOS there is one underlying physical sensor for Attitude, Gravity and LinearAcceleration, so changing sampling frequency for any of these logical sensors changes sampling frequencies for other two as well.

public int **GetSensorSamplingFrequency<T>**() where T : struct, IComponentData
If a sensor is available on the hardware/platform, returns sampling frequency in Hz for this sensor, returns 0 if sensor is not available. For Android due to platform limitations this method returns a value which was set previously via **SetSensorSamplingFrequency** method (default value is 50). For iOS this method returns actual sampling value as reported by system.

public bool **CompensateForScreenOrientation** { get; set; } = true;
If this setting is enabled, rotation values reported by sensors are rotated around the Z axis as follows:

for devices with natural Portrait orientation (all phones, iPads, some Android tablets)
ScreenOrientation.Portrait - values remain unchanged.

ScreenOrientation.PortraitUpsideDown - values rotate by 180 degrees.
ScreenOrientation.LandscapeLeft - values rotate by 90 degrees.
ScreenOrientation.LandscapeRight - values rotate by 270 degrees.
for devices with natural Landscape orientation (some Android tablets)
ScreenOrientation.LandscapeRight - values remain unchanged.
ScreenOrientation.Portrait - values rotate by 270 degrees.
ScreenOrientation.PortraitUpsideDown - values rotate by 90 degrees.
ScreenOrientation.LandscapeLeft - values rotate by 180 degrees.
This setting is enabled by default.

To use sensors app should reference **Unity.Tiny.Input** in asmdef file.
To start using any kind of sensor it is required to check if this sensor is available in the system:
var availableTSensor = **World.GetExistingSystem<InputSystem>().IsAvailable<TSensor>**();
If sensor is available, it is possible to enable it to start getting data:
World.GetExistingSystem<InputSystem>().EnableSensor<TSensor>();
In OnUpdate method it is required to check if data from sensor are available:
var dataAvailableTSensor = **HasSingleton<TSensor>**();
It makes sense to stop checking this once sensor singleton becomes available, because this is a relatively expensive request.
After sensor singleton became available, it is possible to use its data:
var dataTSensor = **GetSingleton<TSensor>**();

Multi-touch (iOS & Android)

Tiny supports multi-touch events for both Android and iOS. To use multi-touch app should reference **Unity.Tiny.Input** in asmdef file.

The TinyPhysics sample & tutorial demonstrates how you can use it (https://github.com/Unity-Technologies/ProjectTinySamples/tree/master/TinyPhysics - check the PointerSystemBase.cs to start)

Touch events are accessible via these InputSystem methods:

```
public bool IsTouchSupported()
```

Returns true if the current device produces touch input responses. This value may not be accurate until a first touch occurs.

```
public int TouchCount()
```

Returns the number of currently active touches

```
public Touch GetTouch(int index)
```

Retrieves information for a specific touch point. The index ranges from 0 to the value returned by **TouchCount**.

Information about touches is passed as Touch structs:

```
	public struct Touch
	{
	//  Specifies the difference, in pixels, between the touch point's X
coordinate
	//  in the current frame and the previous frame. This tells you how far the
	//  touch point has moved horizontally in the browser or application window.
	//  Positive values indicate rightward movement, and negative values leftward
movement.
	public int deltaX;

	//  Specifies the difference, in pixels, between the touch point's Y
coordinate
	//  in the current frame and the previous frame. This tells you how far the
	//  touch point has moved vertically in the browser or application window.
	//  Positive values indicate upward movement, and negative values downward
movement.
	public int deltaY;

	//  A unique identifier for the finger used in a touch interaction.
	public int fingerId;

	// Specifies the life cycle state of this touch. The TouchState
	// enum defines the possible values
	public TouchState phase;

	//  Specifies the absolute X coordinate of the touch, in pixels on the browser
	//  or application window. A value of 0 corresponds to the leftmost edge of
	//  the window. The higher the value, the farther right the coordinate.
	public int x;

	//  Specifies the absolute Y coordinate of the touch, in pixels on the browser
	//  or application window. A value of 0 corresponds to the bottommost edge of
	//  the window. The higher the value, the farther up the coordinate.
	public int y;
	};
```

phase value can be one of the **TouchState** enum:

```
	public enum TouchState
	{
	//  The initial reported state for new touch points.
	Began,

	//  Specifies that the touch point's position changed between the
	//  previous frame and this frame.
	Moved,

	//  Specifies that the touch point's position did not change between
	//  the previous frame and this frame.
	Stationary,
```

```
        //  Specifies that the touch point is expired. Expired Touch points
        //  are reported in Ended state for one frame, before being cleared
        //  from the list of registered touches.
        Ended,

        //  Specifies that the operating system or an event interrupted the
        //  touch interaction.
        ///
        //  For example if an incoming call causes the the operating system
        //  to deliver a popup message, the application or tab switches
        //  from one context to another. Also happens when the screen is rotated.
        Canceled
        }
```

Code example to process touch events:

```
        protected override void OnUpdate()
        {
                var Input = World.GetExistingSystem<InputSystem>();
                if (Input.IsTouchSupported() && Input.TouchCount() > 0)
                {
                for (var i = 0; i < Input.TouchCount(); i++)
                {
                        var itouch = Input.GetTouch(i);
                        var pos = new float2(itouch.x, itouch.y);
                // process touch with pos coordinates
                }
                }
}
```

Native Plugins (iOS & Android)

Android

There is support for aar, jar, c++, Java and Kotlin files.
To add aar, jar, Java, Kotlin files to the project it is required to place them in the "**android~**" folder in your assembly. These files would be automatically copied to specific directories inside the Gradle project. C++ files should be placed in the "**cpp~**" folder in your assembly. During the build process they would be compiled into a static library and then this library would be linked with other static libraries to one project specific dynamic library.

From c++ code these helper methods are available (**<AndroidWrapper.h>** should be included)

JavaVM* **Unity_Get_JavaVM()**

Returns current Java VM which can be used for JNI calls.

jobject **Unity_Get_AndroidActivity()**

Returns pointer to current Activity.

Also **JavaVMThreadScope** helper class is available. This class can be used if it required to call Java methods from C++ using such pattern:

```
void java_wrapper_method()
{
    JavaVMThreadScope javaVM;
    JNIEnv* env = javaVM.GetEnv();
    jclass clazz = env->FindClass(className);
    jmethodID method = env->GetStaticMethodID(clazz, methodName, methodSignature);
    env->CallStaticVoidMethod(clazz, method);
}
```

When an instance of **JavaVMThreadScope** class is being created it attaches the current thread to JavaVM and after that it is possible to use **GetEnv**() method to get a current pointer to the current JNI environment. When an instance of **JavaVMThreadScope** class is being destroyed it detaches current thread from JavaVM. Also in Debug configuration **JavaVMThreadScope** destructor checks and logs exceptions which might happen while calling Java methods.

From Java it is possible to access current Activity via:

```
com.unity3d.tinyplayer.UnityTinyActivity getActivity()
```

To use these helper methods and class it is required to reference "**Unity.Tiny.Android**" assembly from asmdef file.

For now in Tiny there is no direct access to Java native methods from C#. So it is required to use intermediate C++ level to do this.

iOS

There is support for m/mm and c++ files

m/mm/c++ files should be copied in the "**cpp~**" folder in your assembly. During the build process they are compiled into a static library and this library is then added to the XCode project.

From m/mm/c++ code this helper method is available (**<IOSWrapper.h>** should be included)

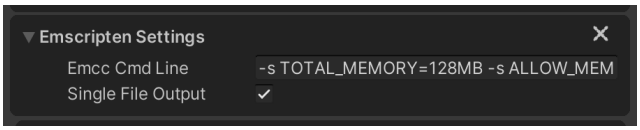
```
UIViewController* Unity_Get_ViewController()
```

Returns pointer to current UIViewController.

To use this helper method it is required to reference "**Unity.Tiny.IOS**" assembly from asmdef file.

Single HTML File Output (Web)

For Web builds you can choose to enable the "Single File Output" option, which can be found under "Emscripten Settings" in the Build Configuration. When selected, Unity produces a single HTML file, that contains all the supporting files and assets encoded as base64. This format is required for some distribution platforms.



Interacting with browser scripting (Javascript/C# interoperability)

When building content for the web, you might need to communicate with other elements on your web page. Or you might want to implement functionality using Web APIs which Project Tiny does not currently expose by default. In both cases, you need to directly interface with the browser's JavaScript engine. This can be achieved in the following ways.

JavaScript plugins

The recommended way of using JavaScript in your project is to add your JavaScript code to your project, and then call the added functions directly from your C# script code. In order to do so, you need to create a JavaScript plugin, embedding JavaScript functions which should be called from C#. JavaScript plugins should be put under *"js~"* folder next to the *".asmdef"* file. A JavaScript plugin should have a *.js* extension and the following syntax:

```
mergeInto(LibraryManager.library, {
  MyJavaScriptFunction1: function (...) {
    ...
  },
  MyJavaScriptFunction2: function (...) {
    ...
  },
  ...
  MyJavaScriptFunctionN: function (...) {
    ...
  },
});
```

It is possible to use multiple JavaScript plugins in your project.

Calling JavaScript functions from C# scripts

JavaScript functions can be called with boolean, number and string arguments and also return values of those types. Strings however should be handled in a specific way. When a string is passed from C# to JavaScript, the corresponding JavaScript function receives a pointer to a null-terminated UTF-8 string on the heap, which can be converted to a JavaScript string using a predefined *UTF8ToString()* function. When a JavaScript function needs to return a string to C#,

it should first allocate a null-terminated UTF-8 string on the heap (using predefined *lengthBytesUTF8()*, *_malloc()* and *stringToUTF8()* functions) and then return a pointer of that allocated string to C#. This is demonstrated in the following example:

```
mergeInto(LibraryManager.library, {

  // this function prints a “Hello” message to the console
  PrintHello: function () {
    console.log("[JavaScript] PrintHello: " + "Hello");
  },

  // this function prints a string, provided as an argument, to the console
  PrintString: function (pStr) {
    var str = UTF8ToString(pStr); // create a JavaScript string from a
null-terminated UTF-8 string allocated on the heap, pointed by pStr
    console.log("[JavaScript] PrintString: " + str);
  },

  // this function adds two numbers and returns the result
  AddNumbers: function (x, y) {
    console.log("[JavaScript] AddNumbers: " + x + " + " + y);
    return x + y;
  },

  // this function returns a string to C#
  ReceiveString: function () {
    var str = "A string passed from JavaScript to C#";
    var bufferSize = lengthBytesUTF8(str) + 1; // calculate the size of
null-terminated UTF-8 string
    var buffer = _malloc(bufferSize); // allocate string buffer on the heap
    stringToUTF8(str, buffer, bufferSize); // fill the buffer with the
string UTF-8 value
    return buffer; // return the pointer of the allocated string to C#
  },

});
```

You can now call JavaScript functions defined in the JavaScript plugin directly from C# if you declare them using *[DllImport("__Internal")]* syntax:

```
using System;
```

```
using Unity.Entities;
using System.Runtime.InteropServices;

public class MySystem : SystemBase
{

    [DllImport("__Internal")]
    private static extern void PrintHello();

    [DllImport("__Internal")]
    private static extern void PrintString(string str);

    [DllImport("__Internal")]
    private static extern int AddNumbers(int x, int y);

    [DllImport("__Internal")]
    private static extern string ReceiveString();

    protected override void OnCreate()
    {
        PrintHello();
        PrintString("A string passed from C# to JavaScript");
        Console.WriteLine("[C#] AddNumbers: " + AddNumbers(5, 7));
        Console.WriteLine("[C#] ReceiveString: " + ReceiveString());
    }

    protected override void OnUpdate()
    {
    }
}
```

Expected console output:

```
[JavaScript] PrintHello: Hello
[JavaScript] PrintString: A string passed from C# to JavaScript
[JavaScript] AddNumbers: 5 + 7
[C#] AddNumbers: 12
[C#] ReceiveString: A string passed from JavaScript to C#
```

Working with arrays

When calling a JavaScript function from C# with an array argument, the function will receive a pointer to the first element of the array, which is stored on the heap. If the size of the array is not known in advance, then the size should be also provided as an argument to the JavaScript function. Array elements can be accessed from JavaScript using the following predefined typed arrays: *HEAP8* (*sbyte*), *HEAPU8* (*byte*), *HEAP16* (*short*), *HEAPU16* (*ushort*), *HEAP32* (*int*), *HEAPU32* (*uint*), *HEAPF32* (*float*) and *HEAPF64* (*double*). Note that when accessing a value of a typed array, the pointer should be shifted accordingly to the size of the used type.

Here is an example of processing different types of arrays:

```
mergeInto(LibraryManager.library, {

  // this function adds 10 to each element of a byte array
  ProcessByteArray: function (pArray, size) {
    var array = HEAPU8.subarray(pArray, pArray + size);
    console.log("[JavaScript] ProcessByteArray: " + array);
    for (var i = 0; i < array.length; i++)
      array[i] += 10;
  },

  // this function adds 20 to each element of a short array
  ProcessShortArray: function (pArray, size) {
    // short takes 2 bytes
    var array = HEAP16.subarray(pArray >> 1, (pArray >> 1) + size);
    console.log("[JavaScript] ProcessShortArray: " + array);
    for (var i = 0; i < array.length; i++)
      array[i] += 20;
  },

  // this function adds 40 to each element of a uint array
  ProcessUintArray: function (pArray, size) {
    // uint takes 4 bytes
    var array = HEAPU32.subarray(pArray >> 2, (pArray >> 2) + size);
    console.log("[JavaScript] ProcessUintArray: " + array);
    for (var i = 0; i < array.length; i++)
      array[i] += 40;
  },

  // this function multiplies each element of a float array by 0.5
  ProcessFloatArray: function (pArray, size) {
    // float takes 4 bytes
```

```
    var array = HEAPF32.subarray(pArray >> 2, (pArray >> 2) + size);
    console.log("[JavaScript] ProcessFloatArray: " + array);
    for (var i = 0; i < array.length; i++)
      array[i] *= 0.5;
  },

  // this function multiplies each element of a double array by 0.25
  ProcessDoubleArray: function (pArray, size) {
    // double takes 8 bytes
    var array = HEAPF64.subarray(pArray >> 3, (pArray >> 3) + size);
    console.log("[JavaScript] ProcessDoubleArray: " + array);
    for (var i = 0; i < array.length; i++)
      array[i] *= 0.25;
  },

});
```

And the corresponding C# code would be:

```
using System;
using Unity.Entities;
using System.Runtime.InteropServices;

public class MySystem : SystemBase
{
    [DllImport("__Internal")]
    private static extern void ProcessByteArray(byte[] array, int size);

    [DllImport("__Internal")]
    private static extern void ProcessShortArray(short[] array, int size);

    [DllImport("__Internal")]
    private static extern void ProcessUintArray(uint[] array, int size);

    [DllImport("__Internal")]
    private static extern void ProcessFloatArray(float[] array, int size);

    [DllImport("__Internal")]
    private static extern void ProcessDoubleArray(double[] array, int
size);
```

```
    string ArrayToString<T>(T[] array)
    {
        var str = "";
        for (var i = 0; i < array.Length; i++)
            str += (i == 0 ? "" : ",") + array[i];
        return str;
    }

    protected override void OnCreate()
    {
        var byteArray = new byte[] { 1, 2, 3 };
        ProcessByteArray(byteArray, byteArray.Length);
        Console.WriteLine("[C#] processed byteArray: " +
ArrayToString<byte>(byteArray));

        var shortArray = new short[] { 1, 2, 3 };
        ProcessShortArray(shortArray, shortArray.Length);
        Console.WriteLine("[C#] processed shortArray: " +
ArrayToString<short>(shortArray));

        var uintArray = new uint[] { 1, 2, 3 };
        ProcessUintArray(uintArray, uintArray.Length);
        Console.WriteLine("[C#] processed uintArray: " +
ArrayToString<uint>(uintArray));

        var floatArray = new float[] { 1, 2, 3 };
        ProcessFloatArray(floatArray, floatArray.Length);
        Console.WriteLine("[C#] processed floatArray: " +
ArrayToString<float>(floatArray));

        var doubleArray = new double[] { 1, 2, 3 };
        ProcessDoubleArray(doubleArray, doubleArray.Length);
        Console.WriteLine("[C#] processed doubleArray: " +
ArrayToString<double>(doubleArray));
    }

    protected override void OnUpdate()
    {
    }
}
```

Expected console output:

```
[JavaScript] ProcessByteArray: 1,2,3
[C#] processed byteArray: 11,12,13
[JavaScript] ProcessShortArray: 1,2,3
[C#] processed shortArray: 21,22,23
[JavaScript] ProcessUintArray: 1,2,3
[C#] processed uintArray: 41,42,43
[JavaScript] ProcessFloatArray: 1,2,3
[C#] processed floatArray: 0.5,1,1.5
[JavaScript] ProcessDoubleArray: 1,2,3
[C#] processed doubleArray: 0.25,0.5,0.75
```

Using arrays might be particularly beneficial if you perform heavy interaction between C# and JavaScript, which involves processing of a significant amount of data. Instead of allocating a new block of memory on each JavaScript call, all the communication can be performed through a large enough buffer preallocated on C# side.

Calling C# callbacks from JavaScript

In some situations you need to call a C# callback from JavaScript. In order to do this, you need to create a JavaScript plugin function which receives a C# callback as an argument, so that it can be called later using a *dynCall_*()* function with the appropriate function signature.

Function signature is combined from the following letters: “v” (void), ”i” (integer, pointer, bool), ”f” (float) and “d” (double). The first letter corresponds to the return type, while the remaining letters correspond to the function arguments. For example:

- *static void callback(float, double)*
  corresponds to a “vfd” signature and can be called using *dynCall_vfd()* function.
- *static bool callback(int, string)*
  corresponds to a “iii” signature and can be called using *dynCall_iii()* function.

The first argument of a *dynCall_*()* function is a pointer to the callback function in the function table. The remaining arguments of *dynCall_*()* correspond to the arguments provided for the callback function.

Below is an example, demonstrating how to call C# callback functions from JavaScript. One of the callbacks is called just once and prints a predefined message to the console. Another callback is called every second and prints the provided arguments (iteration index and current time string) to the console, it also returns a bool, indicating whether callback iterating should continue.

```
mergeInto(LibraryManager.library, {

  SetSimpleCallback: function (pSimpleCallback) {
    console.log("[JavaScript] SetSimpleCallback");

    setTimeout(function () {
      // _v signature corresponds to `void ()`
      dynCall_v(pSimpleCallback);
    }, 0);
  },

  SetTimerCallback: function (pTimerCallback) {
    console.log("[JavaScript] SetTimerCallback");

    var iteration = 0;
    var intervalID = setInterval(timerCallback, 1000);

    function timerCallback () {
      iteration++;

      // allocate the time string
      var timeString = new Date().toLocaleTimeString();
      var bufferSize = lengthBytesUTF8(timeString) + 1;
      var buffer = _malloc(bufferSize);
      stringToUTF8(timeString, buffer, bufferSize);

      // _iii signature corresponds to `bool (int, string)`, because
      // a bool is represented by an integer with value 0 or 1,
      // and a string is also represented by an integer (a pointer to the
allocated string)
      var continueIterating = dynCall_iii(pTimerCallback, iteration,
buffer);

      // free the allocated time string
      _free(buffer);

      // we can stop iterating depending on the value returned by the
callback
      if (!continueIterating)
        clearInterval(intervalID);
    }
  },
```

```
});
```

Then you can send C# callbacks to JavaScript in the following way:

```
using System;
using Unity.Entities;
using System.Runtime.InteropServices;

public class MySystem : SystemBase
{
    internal class MonoPInvokeCallbackAttribute : Attribute
    {
        public MonoPInvokeCallbackAttribute() {}
        public MonoPInvokeCallbackAttribute(Type t) {}
    }


    // this callback does not take any arguments and does not return any
value
    // it just prints a predefined message to the console
    [MonoPInvokeCallback (typeof (Action))]
    private static void SimpleCallback()
    {
        Console.WriteLine("[C#] SimpleCallback called");
    }

    [DllImport("__Internal")]
    private static extern void SetSimpleCallback(Action callback);


    delegate bool TimerCallbackDelegate(int iteration, string timeString);

    // this callback takes the current iteration index and current time
string as arguments,
    // prints the provided arguments to the console and returns a bool,
indicating whether iterating should continue
    [MonoPInvokeCallback (typeof (TimerCallbackDelegate))]
    private static bool TimerCallback(int iteration, string timeString)
    {
        Console.WriteLine("[C#] TimerCallback called with iteration: " +
```

```
iteration + " and timeString: " + timeString);
        return iteration < 5;
    }

    [DllImport("__Internal")]
    private static extern void SetTimerCallback(TimerCallbackDelegate
timerCallback);

    protected override void OnCreate()
    {
        SetSimpleCallback(SimpleCallback);
        SetTimerCallback(TimerCallback);
    }

    protected override void OnUpdate()
    {
    }
}
```

Expected console output:

```
[JavaScript] SetSimpleCallback
[JavaScript] SetTimerCallback
[C#] SimpleCallback called
[C#] TimerCallback called with iteration: 1 and timeString: 9:08:25 AM
[C#] TimerCallback called with iteration: 2 and timeString: 9:08:26 AM
[C#] TimerCallback called with iteration: 3 and timeString: 9:08:27 AM
[C#] TimerCallback called with iteration: 4 and timeString: 9:08:28 AM
[C#] TimerCallback called with iteration: 5 and timeString: 9:08:29 AM
```

Adding raw JavaScript code to the build

In some situations you might need to append raw JavaScript code to the generated build. For example, this can be some JavaScript library or some other code which has to be called from a JavaScript plugin. One way to achieve this would be to modify the default "*tiny_shell.html*" template file and add the necessary JavaScript code there. Another way to do this, would be to simply put all the necessary JavaScript code under under *"prejs~"* or *"postjs~"* folders next to the *".asmdef"* file.

The contents of all the *.js* files under *"prejs~"* folder will be appended at the top of the JavaScript runtime of the generated build, while contents of all the *.js* files under *"postjs~"* folder will be appended at the bottom of the JavaScript runtime. Functions declared in the appended JavaScript code can be accessed from JavaScript plugins, but can not be imported into C# scripts.

Let's consider the following C# script:

```
using System;
using Unity.Entities;
using System.Runtime.InteropServices;

public class MySystem : SystemBase
{
    [DllImport("__Internal")]
    private static extern void CallRawJavaScriptFunction();

    protected override void OnCreate()
    {
        CallRawJavaScriptFunction();
    }

    protected override void OnUpdate()
    {
    }
}
```

The following "*Assets/.../js~/MyPlugin.js"* plugin:

```
mergeInto(LibraryManager.library, {
  CallRawJavaScriptFunction: function () {
    RawJavaScriptFunction();
  },
});
```

And the following "*Assets/.../prejs~/MyRawJavaScript.js"* file:

```
function RawJavaScriptFunction() {
  console.log("[JavaScript] RawJavaScriptFunction");
}
```

Then the expected console output will be:

```
[JavaScript] RawJavaScriptFunction
```

Using web templates

When building content for the web, you might want to use your own html code for the main page or add additional files, such as css styles, JavaScript libraries or images, to the build. This can be easily achieved by using web templates.

Web template structure

Web template is a folder which contains files which should be merged with the generated build. Web template should contain a “tiny_shell.html” file which is treated as the main html. The “tiny_shell.html” file should contain a “{{{ TINY_SHELL }}}” (Case Sensitive) macro which will be expanded at build time into the JavaScript code necessary to load the build.

The default Tiny template can be found under the following folder: “Packages/com.unity.dots.runtime/LowLevelSupport~/WebSupport/WebTemplates/Default”

The default template contains the following “tiny_shell.html” file:

```
<!doctype html>
<html lang="en-us">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1,
user-scalable=0">
</head>
<div id="waitForManagedDebugger" style="display:none">
   <h1 style="text-align:center;">
     <div id="waitForManagedDebuggerMessage"></div>
     <button onclick="doneWaitingForDebugger()">Continue</button>
   </h1>
</div>
<body style="margin:0px; border:none; overflow:hidden; display:block;">
<div id="error_log"></div>
<canvas id="UT_CANVAS" tabindex="1" style='display:block;
touch-action:none;'></canvas>
```

```
<script>
{{{ TINY_SHELL }}}
</script>
</body>
</html>
```

All the other files and subfolders from the web template folder will be automatically copied unprocessed to the output directory at build time. A web template can reside in any folder on the drive. For example, it might be convenient to store custom web templates under the “Assets” folder.

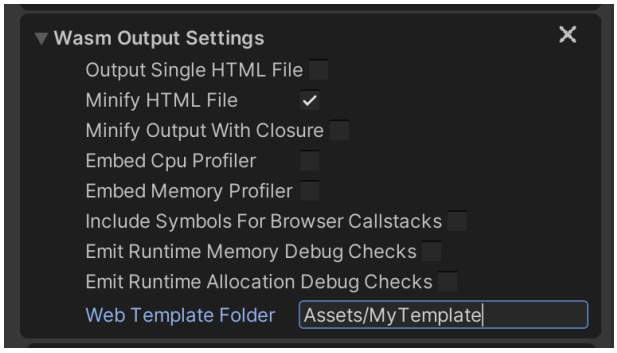
Creating a custom web template

Let’s consider an example, when we would like to add a favicon to the generated build. So let’s create an “Assets/MyTemplate” folder, and put the following files there:

- **Assets/MyTemplate/favicon.png**, which will be used as the website icon;
- **Assets/MyTemplate/tiny_shell.html**, which contains the following html code:

```
<!doctype html>
<html lang="en-us">
<head>
<meta charset="utf-8">
<link rel="icon" href="favicon.png" type="image/png"/>
</head>
<body>
<script>
{{{ TINY_SHELL }}}
</script>
</body>
</html>
```

Now we can specify the location of our custom web template folder “Assets/MyTemplate” in a “Web Template Folder” field under the “Wasm Output Settings” component (the specified path can be either absolute or relative to the project root):



When building the project, the contents of the provided “tiny_shell.html” file will be used as the code of the main build html, while the additional “favicon.png” file from the web template folder will be copied into the output build folder.

If the “Web Template Folder” field is left empty, then the default Tiny template will be used for the generated build.

Screen orientation

Screen & Device Orientation APIs

Two related parameters are being used to control orientation in the API:

- Device orientation – how the device is being physically held (can be Unknown if device lies on the table face up or face down)
- Screen orientation – how the screen is being oriented

For both device and screen orientation values **ScreenOrientation** enum (defined in **Unity.Tiny** workspace in **Unity.Tiny.Core** assembly) is being used.

```
  [Flags]
public enum ScreenOrientation
{
    Unknown = 0,
    Portrait = 1,
    PortraitUpsideDown = 2,
    ReversePortrait = 2,
    Landscape = 4, // home button below screen
    LandscapeLeft = 4, // device rotated left, home button to the right of the
screen
    LandscapeRight = 8, // device rotated right, home button to the left of the
screen
    ReverseLandscape = 8, // home button above screen
```

```
    AutoRotationPortrait = Portrait | ReversePortrait,
    AutoRotationLandscape = Landscape | ReverseLandscape,
    AutoRotation = Portrait | ReversePortrait | Landscape | ReverseLandscape
}
```

To make this enum more familiar for both Android and iOS developers and also to avoid problems with devices with natural Landscape orientation some duplicate values are added. But unlike iOS, to avoid confusion, we are using the same values for both device and screen orientation. LandscapeLeft/LandscapeRight have the same meaning as in big Unity. Required screen orientation can be set as a mask of enabled values. Some commonly used mask values are also included in the **ScreenOrientation** enum. When the mask value is being set or the physical device orientation is changed actual screen orientation is calculated as an intersection of the current device orientation and the screen orientation mask. If there is no intersection, then these rules are applied:

- If possible the system tries to follow the current device portrait/landscape orientation. For example, if current device orientation is Landscape, but current mask allows ReverseLandscape (not Landscape), then ReverseLandscape would be selected.
- If there is no way to follow current device portrait/landscape orientation, then screen orientation would be selected from the mask using this priority: Portrait, Landscape, ReversePortrait, ReverseLandscape.

These methods are declared in WindowSystemBase class (defined in **Unity.Tiny** workspace in **Unity.Tiny.Core** assembly) and are implemented in platform specific Window systems:

```
    void SetOrientationMask(ScreenOrientation orientation);
```

- Android/iOS: Sets current orientation mask, rotates screen if required. Mask cannot be 0.
- Other platforms: Does nothing

Note: For iPhone devices without a physical Home button (iPhone X, XS, XR and similar), PortraitUpsideDown screen orientation is disabled. Trying to set the mask with this only orientation on such devices doesn't change the current orientation mask. Also on such devices if the current mask allows this orientation screen would never rotate to this orientation even if the device is being held in Portrait Upside-down orientation.

```
ScreenOrientation GetOrientationMask();
```

- Android/iOS: Returns current orientation mask
- Other platforms: Returns current orientation (the same as GetOrientation())

```
ScreenOrientation GetOrientation();
```

- Android/iOS: Returns current screen orientation
- WebGL: Returns Unknown value (considering implementation for later)

- Other platforms: Returns Landscape if window width >= window height, Portrait otherwise

These methods can be called like this:

```
World.GetExistingSystem<WindowSystem>().SetOrientationMask(ScreenOrientation.Auto
Rotation);
var mask = World.GetExistingSystem<WindowSystem>().GetOrientationMask();
var orientation = World.GetExistingSystem<WindowSystem>().GetOrientation();
```

In **Unity.Platforms** workspace there are two events and events handlers declared which can be used to notify systems if actual device and screen orientation are changed:

```
// sent when physical device orientation is changed, when this happens system
also
// checks if current screen orientation should be updated based on current
allowed
// orientation mask
public struct DeviceOrientationEvent
{
    public int Orientation { get; }
}
// sent when actual screen orientation is changed
public struct ScreenOrientationEvent
{
    public int Orientation { get; }
}
```

To make system react to these events reference to Unity.Platforms.Common must be added to application asmdef file

```
public delegate void ScreenOrientationEventHandler(object sender,
ScreenOrientationEvent evt);
public delegate void DeviceOrientationEventHandler(object sender,
DeviceOrientationEvent evt);
public static event ScreenOrientationEventHandler OnScreenOrientation;
public static event DeviceOrientationEventHandler OnDeviceOrientation;
```

Orientation values in these structs actually are of **ScreenOrientation** enum type.

To make system react to these events this code should be added to OnStartRunning/OnStopRunning methods:

```
protected override void OnStartRunning()
{
    PlatformEvents.OnDeviceOrientation += OnDeviceOrientation;
    PlatformEvents.OnScreenOrientation += OnScreenOrientation;
}
```

```
protected override void OnStopRunning()
{
    PlatformEvents.OnDeviceOrientation -= OnDeviceOrientation;
    PlatformEvents.OnScreenOrientation -= OnScreenOrientation;
}
```

And these methods should be implemented in this System:

```
public void OnDeviceOrientation(object sender, DeviceOrientationEvent evt)
{
      // process new device orientation (ScreenOrientation)evt.Orientation
}
public void OnScreenOrientation(object sender, ScreenOrientationEvent evt)
{
      // process new screen orientation (ScreenOrientation)evt.Orientation
```

DeviceOrientationEvent handlers are being called from the main app thread between update cycles. **ScreenOrientationEvent** handlers are being called from the current WindowSystem OnUpdate method.

If orientation is locked in system device settings, then **DeviceOrientationEvent** is not being sent or automatically processed. But once orientation is unlocked (even while the app is running) sending/processing of **DeviceOrientationEvent** is restored. While orientation is locked it is still possible to force screen orientation change from scripts using **SetOrientationMask** method.

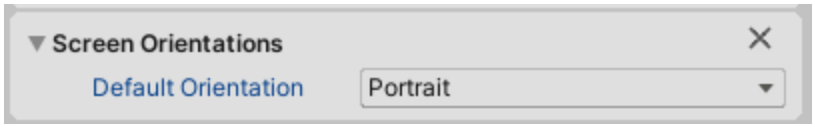
On iOS **DeviceOrientationEvent** may be sent with **ScreenOrientation.Unknown** value. This happens when iOS reports one of the **Unknown**/**FaceUp**/**FaceDown** values for the current device orientation.

Non mobile platforms never send **DeviceOrientationEvent** / **ScreenOrientationEvent**.

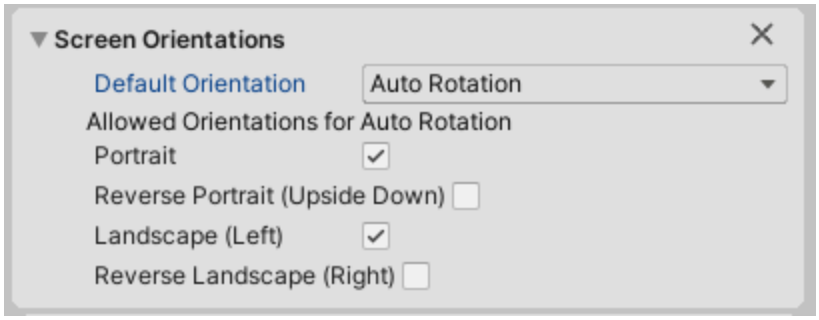
Screen Orientations build component

Both Android and iOS build targets support Screen Orientations component which can be used to limit rotation possibilities for an application. By default this component allows auto-rotation to all 4 possible orientations.

If this component is added to Build Configuration, it is possible to limit the screen orientation to some particular value:



Or to set default orientation to Auto Rotation and limit orientations using checkboxes:



For both Android and iOS using this component in Build Configuration means that the application will never be able to rotate to orientations which are not allowed. Trying to set orientation mask to some value which has no common values with orientations from this build component is NO-OP (but warning message is logged). Otherwise the actual orientation mask is calculated as the intersection of the mask set in Screen Orientation component and the mask which is set using **SetOrientationMask** method.

DeviceOrientationEvent is never being sent if the device is rotated to orientation which is not allowed in the Screen Orientations build component.

Audio

Audio Authoring

Audio is authored in the Unity Editor using the same audio components that are available in regular Unity, however Project Tiny Audio supports a subset of those components and only supports certain properties within those components. Project Tiny supports the AudioListener and AudioSource components.

AudioListener
Project Tiny supports one AudioListener component at a time. If there are zero listeners or two or more listeners, then all 3d audio will be silent.

AudioSource
Project Tiny supports the following properties on an AudioSource component:

- AudioClip,
- Play On Awake,

- Loop,
- Volume,
- Pitch,
- Stereo Pan,
- Spatial Blend,
- Volume Rolloff (Linear and Logarithmic),
- Min Distance, and Max Distance.
- SpatialBlend only supports being set to 0.0 (fully 2d) or 1.0 (fully 3d) and this value is only read at game object conversion time (build time) in the Editor as it affects which audio components we create.

Audio Assets

Project Tiny Audio is not currently integrated with the existing Unity audio asset pipeline, but importing pre-compressed content into Unity is supported. Uncompressed WAVs, Compressed MP3s, and Compressed Vorbis files are all supported. Sounds should be imported into Unity in the form you want in game. **Once imported, the AudioClip properties are not supported, so you can't re-compress in a different format for each platform**.

If one of your target platforms is the Web, then it is recommended that you use MP3s for all audio content because this is supported by all web browsers and is also supported by our mobile platforms (Android and iOS). If only targeting mobile platforms, Vorbis is probably a better option since it achieves better compression rates.

At present, on startup on the Android and iOS platforms, all Project Tiny audio assets are loaded and decompressed in memory. The benefit to this is that every time we play a sound, we do not have to pay the CPU cost to de-compress it. On the web, we start with all sounds compressed in memory and decompress each sound the first time it is played. When we hit a memory limit for decompressed sounds, we start unloading them, prioritizing the least-recently played and the largest sounds. That memory limit is currently hard-coded to 50MB.

Scripting with Audio Components

When we build a Project Tiny application, the game objects and components that we author are converted into a different form for the runtime.

AudioListeners just become AudioListeners, so that conversion is straight-forward. AudioSources, on the other hand, can end up being converted into multiple runtime components:

- AudioSource: clip, volume, loop properties
- Audio2dPanning: pan property
- Audio3dPanning
- AudioDistanceAttenuation: volume rolloff, min distance, max distance properties
- AudioPitch: pitch property

The AudioSource component is always created, while the others are created optionally based on the AudioSource authoring component's property values. If SpatialBlend is set to greater than 0.0 and VolumeRolloff is set to Linear Rolloff or Logarithmic rolloff, then we create an Audio3dPanning and AudioDistanceAttenuation component; otherwise, we create an Audio2dPanning component. If pitch is set to something other than 1.0, then an AudioPitch component is created.

Alternatively, you can create Project Tiny audio components at runtime instead of relying on the Editor's AudioSource conversion workflow described above.

To play an AudioSource, an AudioSourceStart component is added to the same entity as the AudioSource component. To stop an AudioSource, an AudioSourceStop component should be added to the same entity.

To adjust audio parameters at run-time, your C# code needs to adjust the properties on these Tiny audio components.

Here is some DOTS example code to show how to play a sound:

```
var audioEntity = mgr.CreateEntity();
AudioSource source = new AudioSource();
source.clip = eClip; // eClip is the entity that has the AudioClip component we
want to play.
source.volume = 1.0f;
source.loop = false;
mgr.AddComponentData(audioEntity, source);
mgr.AddComponent(audioEntity, typeof(AudioSourceStart));
```

When the sound starts to play, the AudioSystem removes the AudioSourceStart component. Similarly, an AudioSourceStop component can be added to an entity that is currently playing a sound to stop it. In this case, the AudioSystem will remove the AudioSourceStop component when it actually stops the sound.

If you want to control left/right panning manually yourself, you can add the following code, and then adjust panning.pan from -1.0f to fully play out of the left speaker to 1.0f to fully play out of the right speaker:

```
Audio2dPanning panning = new Audio2dPanning();
panning.pan = 0.0f;
mgr.AddComponentData(audioEntity, panning);
```

For a 3d sound, instead of the Audio2dPanning component, you can use the Audio3dPanning and AudioDistanceAttenuation components:

```
Audio3dPanning panning = new Audio3dPanning();
mgr.AddComponentData(audioEntity, panning);
AudioDistanceAttenuation distanceAttenuation = new AudioDistanceAttenuation();
distanceAttenuation.rolloffMode = AudioRolloffMode.Logarithmic;
distanceAttenuation.minDistance = 3.0f;
distanceAttenuation.maxDistance = 50.0f;
mgr.AddComponentData(audioEntity, distanceAttenuation);
```

With those 2 components, the AudioSystem will now automatically adjust the left/right speaker volume based on the audioEntity's 3d position in the world relative to the listener's position and orientation. It will also adjust the volume of the sound based on how far the audioEntity is away from the listener. If it is 3 game units or less away, it will play at full volume. It will get quieter as it moves further away, until it becomes silent at 50 game units away.

To adjust the pitch of a sound, you can add the following code:

```
AudioPitch pitchEffect = new AudioPitch();
pitchEffect.pitch = 1.1f;
mgr.AddComponentData(audioEntity, pitchEffect);
```

In the context of Tiny applications, where most applications are heavily memory-constrained, changing the pitch of a sound can be a good way to get some variety out of one sound, instead of using a container of 5-10 similar sounds. If you want to adjust the pitch as a sound is playing, for example to achieve a Doppler effect, this pitch parameter can be modified as the sound is playing back.

Audio Tips

Looping and MP3 files

Looping MP3 files does not result in perfect, seamless loops. The need for MP3 files really comes from the Safari web browser, so if your project is not targeting the Web or it is targeting the Web but you don't need Safari support, then it is recommended that you instead use Vorbis compression because it will give you better compression results and seamless looping.

If your project is targeting the Web and you want to support all browsers, then there are some strategies that can be used to try to loop your sounds as well as possible:

- For short loops, use uncompressed audio to bypass the problem with MP3s.
- Try looping your MP3 file and listen to the results. Some clips loop better than others and sometimes the results are acceptable.
- Set up your loops so they end in silence and only use looping when it is acceptable to have a silent gap at loop time. This avoids the need for a seamless loop.
- For ambience, try layering multiple loops together. If they loop at different times, sometimes this will help hide the silent gap in one of the loops.
- Don't actually loop your sound, but cross-fade instead. Play the sound once. As it nears completion, fade the volume out, and play a second instance of the same sound and fade in the new instance.

DOTS Runtime Audio

Project Tiny Audio is just one audio engine that will ultimately be available in the DOTS Runtime. There is another, more full-featured solution that is being developed that includes the DOTS Audio package layered over the low-level DSPGraph package.

For authoring, in addition to new workflows, DOTS Audio will likely offer a similar conversion from the existing AudioSource component in the Unity Editor. However, it will convert to its own, different set of DOTS Audio components and will likely support more AudioSource properties than Project Tiny Audio currently does.

On the scripting side, user code will all be interacting with these new DOTS Audio components, instead of the Tiny Audio components described in this document. For this reason, if you are heavily investing in Tiny audio scripting code, we recommend writing a thin layer of audio code between your game code and Tiny Audio. If you have simple functions to do all of the basic, common operations outlined above, it should be much easier to later change these implementations to instead use DOTS Audio to give you more flexibility in the future.

JSON

Reading and writing JSON at runtime is enabled after referencing Unity.Tiny.JSON in your game's asmdef file. It utilizes FixedString in order to maintain burst compatibility in jobs, which adds a limitation of 256 for the length of keys and 4096 characters for the length of string values. These numbers are subject to change based on feedback in the future. For most use cases, the general JSON container TinyJsonInterface allows for parsing, reading, writing, and removing of JSON fields with a dictionary-like interface. For writing JSON objects from scratch, TinyJSONStreamingWriter enables high performance emitting of JSON fields using a stack like interface.

Reading JSON

Existing JSON input can be read with the TinyJsonInterface using its constructor:
Note: Due to current limitations, this constructor relies on a .NET string as input and is not burst compatible. All other usages are burstable and use FixedString or HeapString.
From there, fields can be read by accessing the root level object and using the [] accessor.
Specific types are accessible using both implicit and explicit conversion operations.

```
float floatVal = json.Object["floatKey"];
bool boolVal = json.Object["boolKey"];
```

Or

```
var floatVal = json.Object["floatKey"].AsFloat();
var boolVal = json.Object["boolKey"].AsBool();
```

Note: Calling "As*()" on an incorrect JSON type will result in a runtime exception. The type can be obtained by calling tinyJsonObject.Type.
Nested object/array views are accessible directly from the root type. They can also be cached in a variable.

```
bool boolValInNestedObject =
json.Object["nestedField"]["nestedBoolKey"];
int intValInArray = json.Object["arrayField"][0];
```

Or

```
var nestedObject = json.Object["nestedField"];
bool boolVal = nestedObject["nestedBoolKey"];

var array = json.Object["arrayField"].AsArray();
int intValInArray = array[0];
```

Don't forget to call `json.Dispose()` when you are finished.

Enumerating:
Iterating through both objects and arrays is supported.

```
foreach (var jsonKeyValue in json.Object)
foreach (var jsonObj in json.Object["arrayKey"].AsArray())
```

Writing JSON

Writing JSON fields using the TinyJsonInterface works very similar to reading them. Since TinyJsonInterface is a mutable container, you can write JSON from scratch or modify / append an existing JSON string depending on which constructor you use.
Much like a dictionary, when writing a key that doesn't exist the class will create one. Otherwise, the existing field is modified.

```
var json = new TinyJsonInterface(Allocator.Temp);
var rootObj = json.Object;
```

Writing primitive / string types:

```
rootObj["boolKey"] = false;
rootObj["intKey"] = 21;
rootObj["floatKey"] = 10.3f;
rootObj["stringKey"] = "Hello World!";
```

Writing nested objects: Nested objects can be read and modified just like the root object.
`var nestedObj = rootObj.CreateEmptyObjectField("nestedObjKey");`

Writing arrays: Arrays act like lists and do not have a limited capacity. Existing indices can be modified or values can be appended.

```
var array = rootObj.CreateEmptyArrayField("arrayKey");
array.Append(1);
array.Append(10.3f);
array.Append(false);
array.Append("Hello World!");
var nestedArray = array.AppendEmptyArray();
var objInArray = array.AppendEmptyObject();
```

Overwriting existing array values

```
array[0] = 2;
array[1] = -22.3f;
array[2] = true;
array[3] = "Have a nice day!");
```

Deleting:
You can remove a field from an object by specifying the field to remove:
`json.Object.RemoveField("key");`

You can remove a value from an array by specifying an index:
`array.RemoveAt(0);`

Finalizing Write:
Fields will be written out to JSON in alphabetical order. The return value is a heap string.
`var heapString = json.ToJson();`

Don't forget to call `json.Dispose()` when you are finished.

High Performance JSON Writing

The TinyJsonStreamingWriter exists as an alternative to the TinyJsonInterface when users have the option to write JSON from scratch and know the exact layout in advance. Performance is much better at the cost of usability. Deletions or modifications are not supported. Writing to nested objects and arrays works in a stack-like manner. That being said, for small, simple JSON objects the TinyJsonStreamingWriter is easy to use and highly recommended.

Construction:
`var writer = new TinyJsonStreamingWriter(Allocator.Temp);`
Writing primitive / string fields:

```
writer.PushValueField("boolKeyTrue", true);
writer.PushValueField("intKey", 20);
writer.PushValueField("stringKey", "stringValue");
writer.PushValueField("floatKey", 21.11f);
```

Writing nested objects:

```
writer.PushObjectField("nestedObject");
writer.PushValueField("nestedInt", 20);
writer.PushValueField("nestedString", "hello");
writer.PopObject();
```

Writing arrays:

```
writer.PushArrayField("key");
writer.PushValueToArray(true);
writer.PushValueToArray(20);
writer.PushValueToArray(11.11f);
```

```
writer.PopArray();
```

For objects inside arrays or nested arrays use `PushArrayToArray()` and `PushObjectToArray()`

Finalizing Write: Write to either a mutable JSON interface or to a heapstring.

```
var heapStringRes = writer.WriteToString();
var resultObj = writer.WriteToInterface();
writer.Dispose();
```

Note: Exceptions will be thrown at runtime if the user pops too many times, forgets to pop enough times, tries to push object key/value fields to an array, or tries to push array values to an object.

Tiny.UI

[in progress]
The initial plan for UI was to deliver a subset of UI Toolkit (formerly known at UI Element) but we decided to postpone this approach and instead offer a lightweight UI framework that is a subset of Unity UI (UGUI).
Authoring of UI in the Editor is supported through Unity UI (UGUI). Simple right click in the Hierarchy and add UI to a Canvas, or as a child of other elements.
All `Render Graph Modes` are supported.

General Support:

- Canvas: Only **Screen Space - Overlay.** (This is the default mode and most useful. The other modes require scaling, which we haven't implemented yet.)
- Only Text Mesh Pro. Text is only supported through the Text Mesh Pro (TMP) variants of the UI controls
- Positioning, anchoring, stretching all work. **Rotation & scale do not**.
- Colors can be applied to components. Materials can not.
- Canvas Scalar is not currently supported.

Supported UI

- **Panel**
- **Canvas**

- **Button - TextMeshPro**. Including source images and sprite sheets. “Color Tint” transitions (Highlighted, Pressed, Selected, etc.) are supported, but not sprite swap. (Sprite swap is planned for the future.)
- **Text - TextMeshPro**. Note that with text, the text field (yellow outline) needs to match the Rectangle Transform (blue dots). Normally that’s the case; but the current system will ignore any differences and produce unexpected results. Basic text options are supported, including font, size, and color. Vertical and horizontal alignment are supported. Multi-line has limited support.
- **Image**. Slicing is fully supported, including from sprite sheets. Tiling is not supported.
- **Slider** (with the same support/limitations as Buttons and Text). Direction, Min/Max, Whole Number are supported.
- **Toggle**

Getting Events and UI Elements

Note that the UI entity tree at runtime is not quite the same as the tree you will create in the editor. Currently, the discrepancy is primary when using text. However, future GUI elements will increase the divergence.
There’s also just the general question of how to locate UI elements without tagging them with your own components.
To make this easy, Tiny.UI provides a way to get an entity by its name in the editor:

Entity e = World.GetExistingSystem<ProcessUIEvents>().GetEntityByUIName("Info Text");

The other information you generally need is the UIState. You can iterate through it:

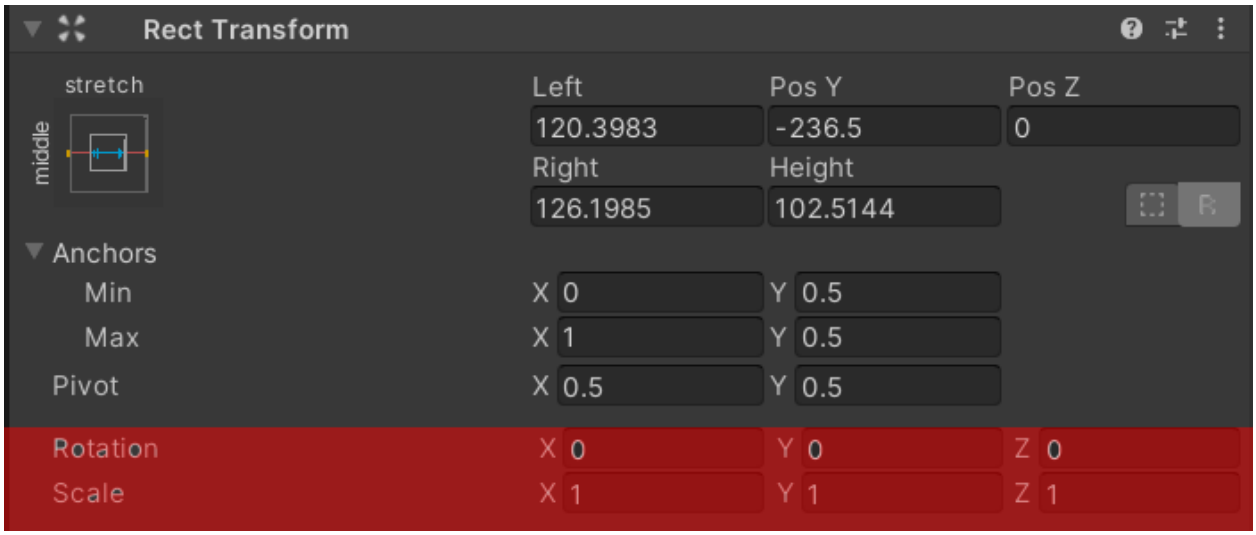
```
            Entity eClicked = Entity.Null;
            Entities.ForEach((Entity e, in UIState sr) =>
                {
                    if (sr.IsClicked)
                    {
                        eClicked = e;
                    }
                }).Run();
```

Or look it up for a particular element with GetUIStateByUIName().

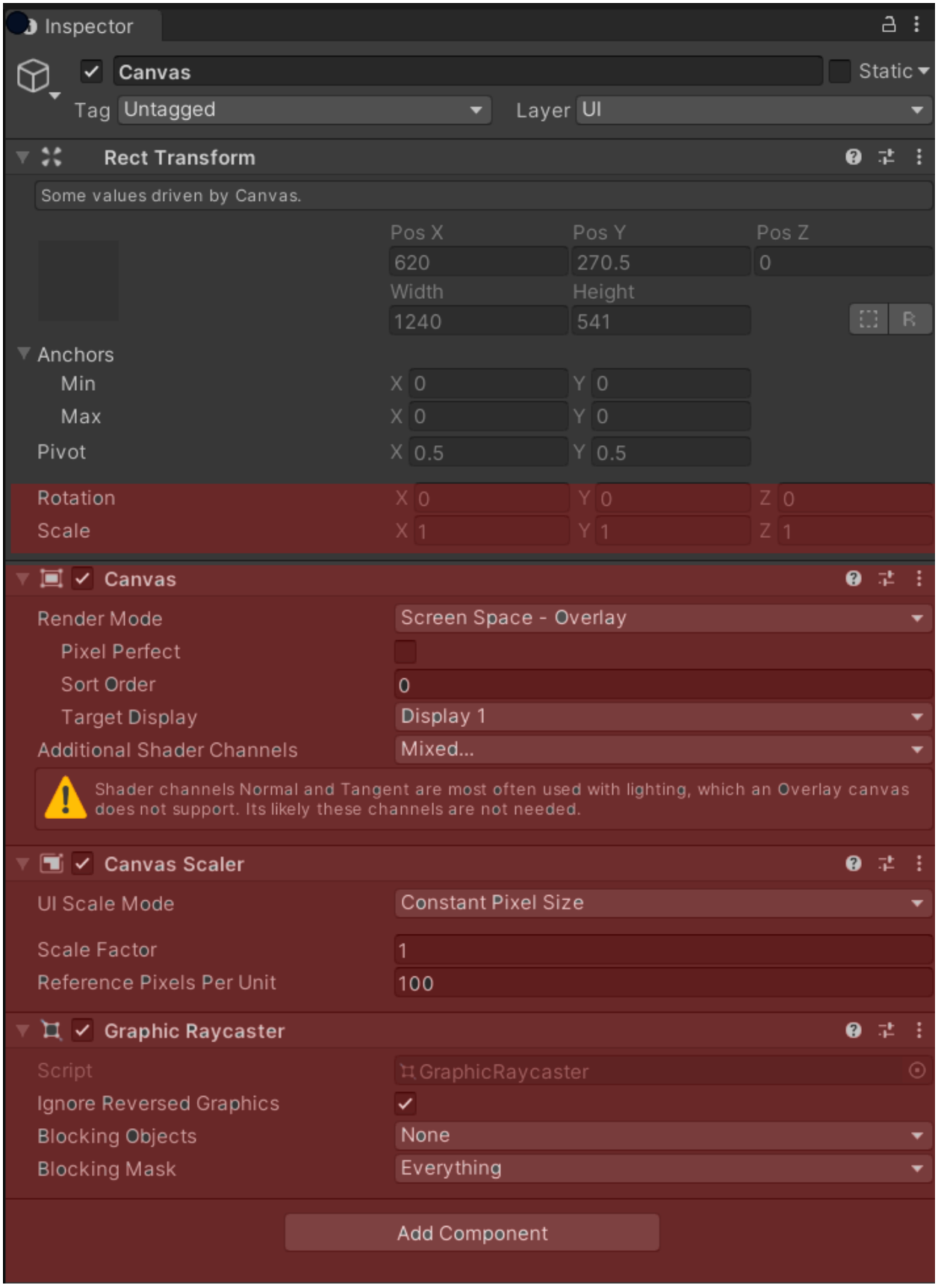
`Hidden`
The RectTransform has a `Hidden` flag. If set, it will keep this entity, and all child RectTransform entities, from rendering. This allows easy & fast (and efficient) visibility toggling.

Supported Features / Properties - Inspector view

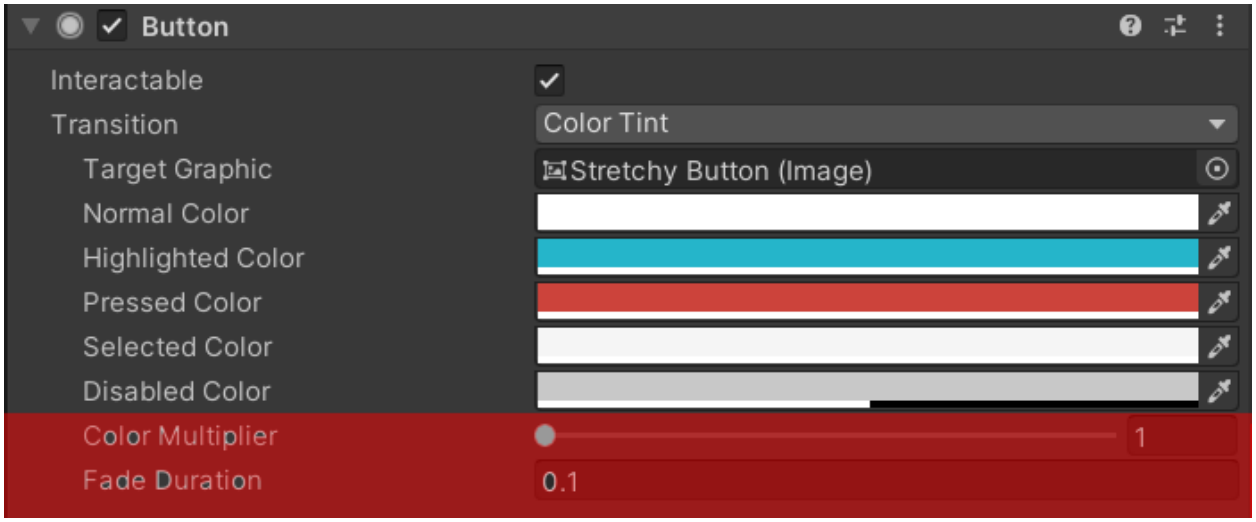
All UGUI components:



Canvas:

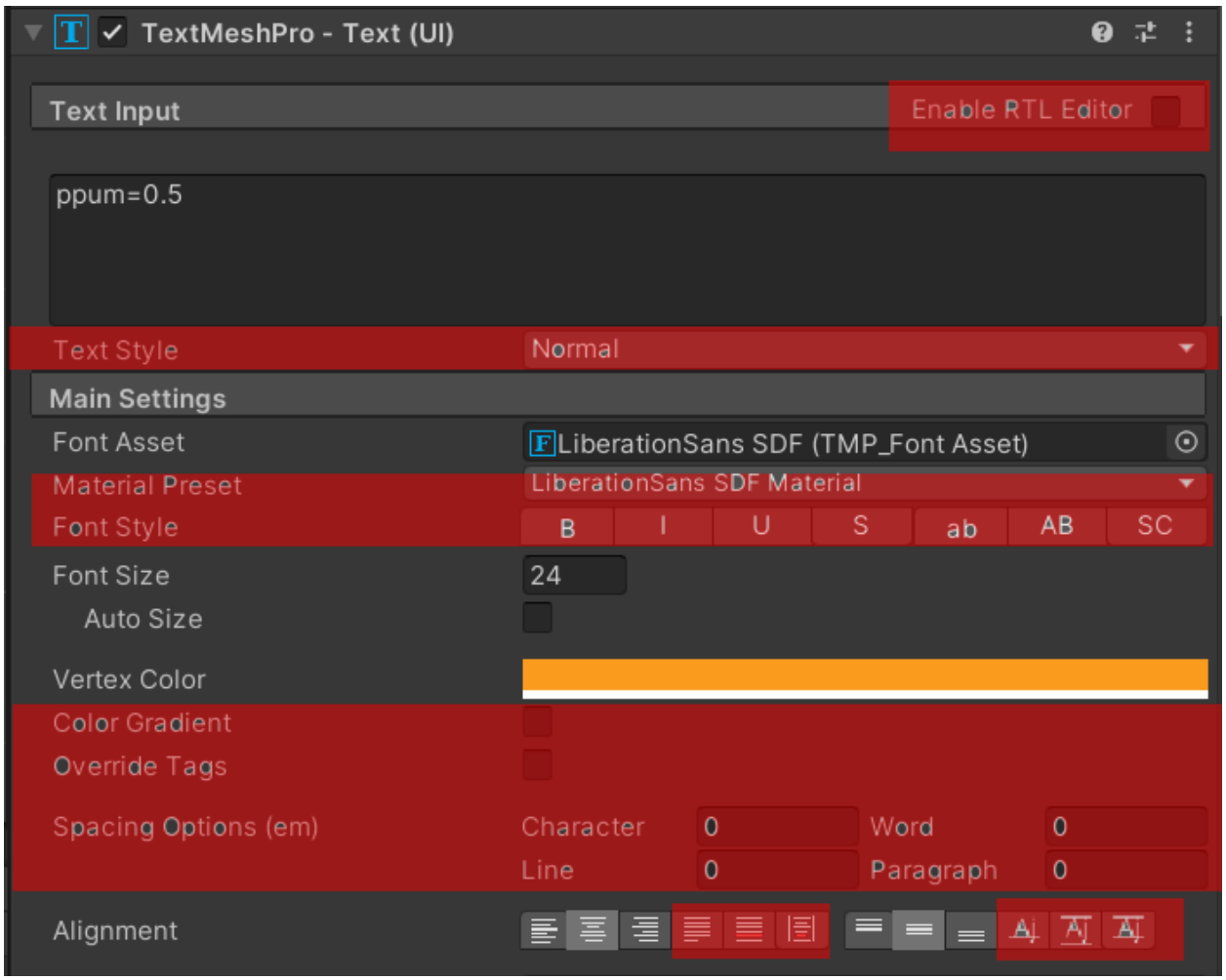

Inspector
Canvas
Static
Tag Untagged
Layer UI
Rect Transform
Some values driven by Canvas.
Pos X
Pos Y
Pos Z
620
270.5
0
Width
Height
1240
541
Anchors
Min
X 0
Y 0
Max
X 0
Y 0
Pivot
X 0.5
Y 0.5
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Canvas
Render Mode
Screen Space - Overlay
Pixel Perfect
Sort Order
0
Target Display
Display 1
Additional Shader Channels
Mixed...
Shader channels Normal and Tangent are most often used with lighting, which an Overlay canvas does not support. Its likely these channels are not needed.
Canvas Scaler
UI Scale Mode
Constant Pixel Size
Scale Factor
1
Reference Pixels Per Unit
100
Graphic Raycaster
Script
GraphicRaycaster
Ignore Reversed Graphics
Blocking Objects
None
Blocking Mask
Everything
Add Component

Buttons:



Text:

NOTE that you *can* set the text properties (bold, italic, etc.) in the Font Asset.



UI Code snippets

This section includes various code snippets to illustrate common actions.

Modifying a button's rect transform based on its state (Make sure to name the UI element something unique in the inspector):

```
var uiSys = World.GetExistingSystem<ProcessUIEvents>();
var buttonEntity = uiSys.GetEntityByUIName("StartButton");
var buttonState = GetComponent<UIState>(buttonEntity);
if (buttonState.IsClicked)
{
    var buttonTransform = GetComponent<RectTransform>(buttonEntity);
    buttonTransform.Hidden = true; // note: this will hide all children also
    SetComponent(buttonEntity, buttonTransform);
}
```

Change a button color:

```
var selectableInfo = GetComponent<Selectable>(uiEntity)
selectable.NormalColor = new float4(1, 0, 1, 0);
SetComponent(uiEntity, selectableInfo);
```

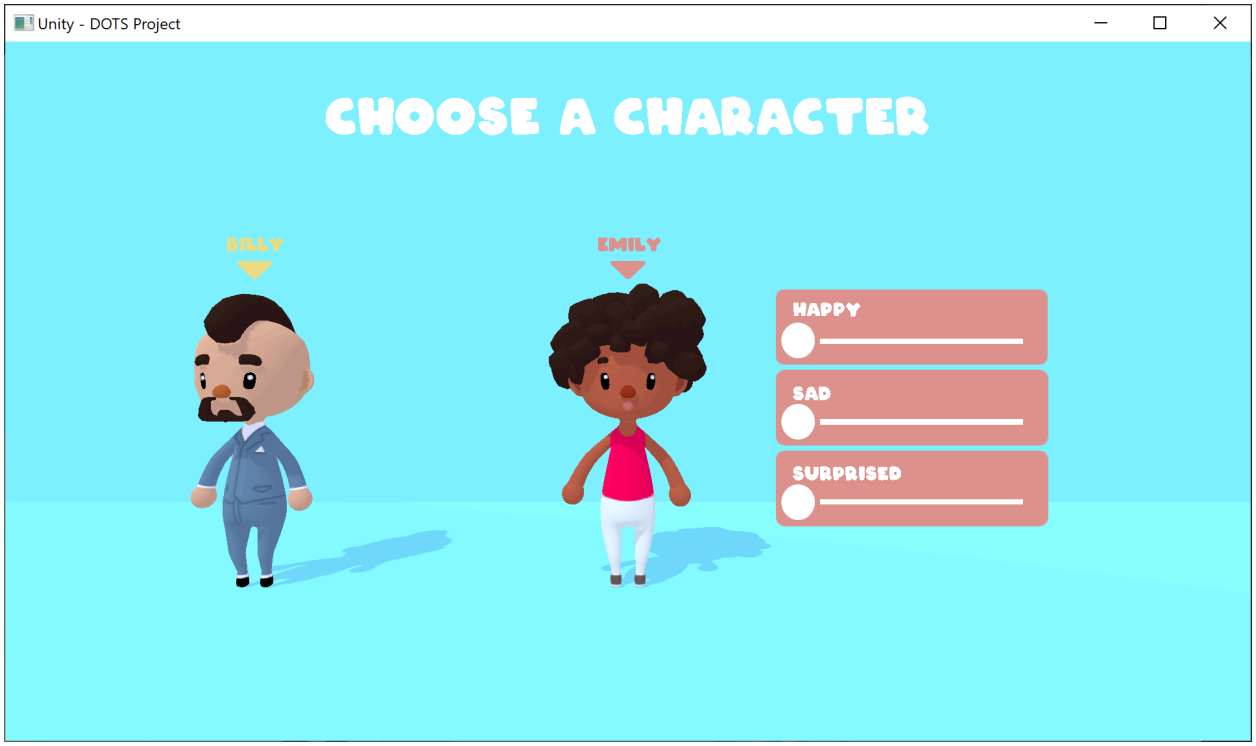
Setting text:

```
var uiSys = World.GetExistingSystem<ProcessUIEvents>();
var textEntity = uiSys.GetEntityByUIName("MyText");
TextLayout.SetEntityTextRendererString(EntityManager, textEntity, "HELLO,
WORLD");
```

Skinned Mesh Renderer and BlendShape

After reading this introduction, **we recommend you explore the sample project** available here:

https://github.com/Unity-Technologies/ProjectTinySamples/tree/master/BlendShapeDemo



Overview

SkinnedMeshRenderer is a minimalistic conversion and mesh deformation system that supports mesh skinning and blendshape. It supports CPU and GPU mesh skinning. Blendshape uses CPU only.

For mesh skinning, on the conversion side, we will generate several SkinnedMeshRenderer entities, those SkinnedMeshRenderer entities are used for switching CPU mesh skinning or GPU mesh skinning at the runtime. We will convert only one entity for CPU mesh skinning. But for GPU mesh skinning, due to the limitation of uniform buffer size, we may split into several entities based on the number of bones which the model have.

You can check the rules below:

```
//if bonecount == 0
//type        canUseCPUSkinning   canUseGPUSkinning
//original    false                false

//if bonecount <= MeshSkinningConfig.GPU_SKINNING_MAX_BONES, then one
SkinnedMeshRenderer may split to
//type        canUseCPUSkinning   canUseGPUSkinning
```

```
//original    true                true

//if bonecount > MeshSkinningConfig.GPU_SKINNING_MAX_BONES, then one
SkinnedMeshRenderer may split to servals
//type        canUseCPUSkinning   canUseGPUSkinning
//original    true                false
//additional  false               true
//additional  false               true
//additional  false               true
```

For blend shape, on the conversion side, we just convert all the blend shape weights from SkinnedMeshRenderer and the blend shape data will be converted from mesh. Also it will share with other SkinnedMeshRenderer entities.
At Runtime, we will use the GPUSkinning setting in DisplayInfo to choose CPU or GPU skinning. Also it will choose corresponding SkinnedMeshRenderer entities to do the mesh skinning based on canUseCPUSkinning and canUseGPUSkinning.

Getting Started

Just adding the SkinnedMeshRenderer component, setup the mesh, choose the material with shader Universal Render Pipeline/Lit or Universal Render Pipeline/Unlit

The Runtime API

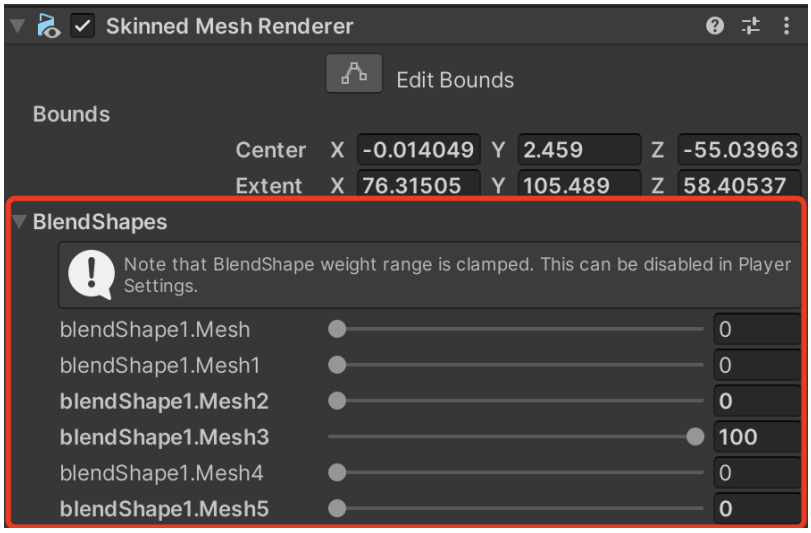
Change BlendShape.
You can add a dynamic buffer with component SetBlendShapeWeight, then add all the modifications to the buffer.

```
EntityCommandBuffer ecb = new EntityCommandBuffer(Allocator.Temp);
Entities.ForEach((Entity e, DynamicBuffer<BlendShapeWeight>
smrBlendShapeWeightBuffer, ref SkinnedMeshRenderer smr) =>
{
   SetBlendShapeWeight setWeight = new SetBlendShapeWeight();
   setWeight.NameHash = BlendShapeChannel.GetNameHash("blendShape1.Mesh3");
   setWeight.ModifiedWeight = 0;
   DynamicBuffer<SetBlendShapeWeight> setWeightsBuffer =
ecb.AddBuffer<SetBlendShapeWeight>(e);
   setWeightsBuffer.Add(setWeight);
}).WithoutBurst().Run();
```

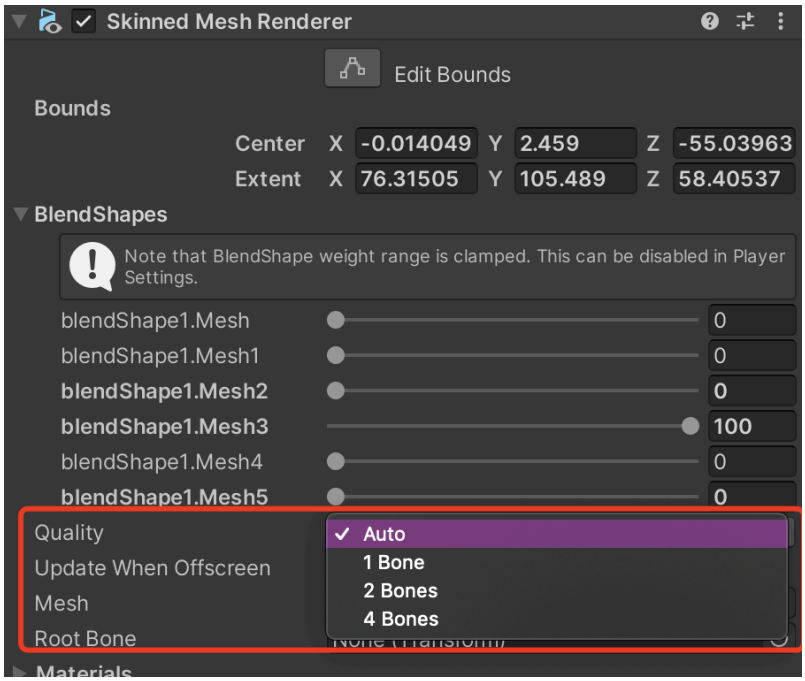
```
ecb.Playback(EntityManager);
ecb.Dispose();
```

Conversion Support

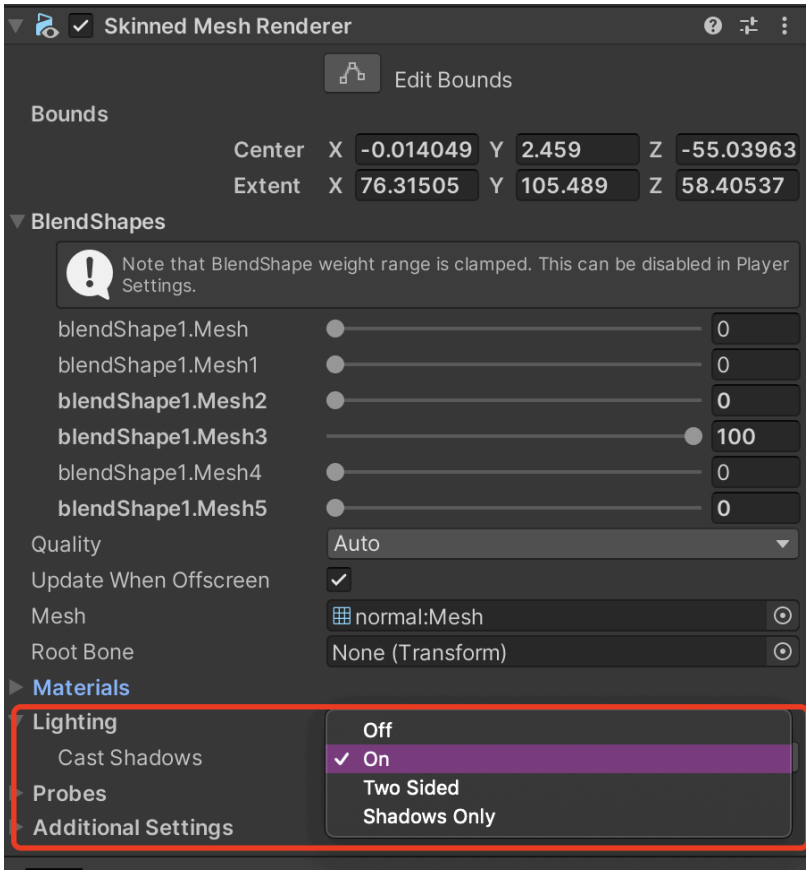
Here’s an overview of the features that are available (converted to Entities).
SkinnedMeshRenderer



All the blend shape weights will be converted and the blend shape name will convert into a name hash.

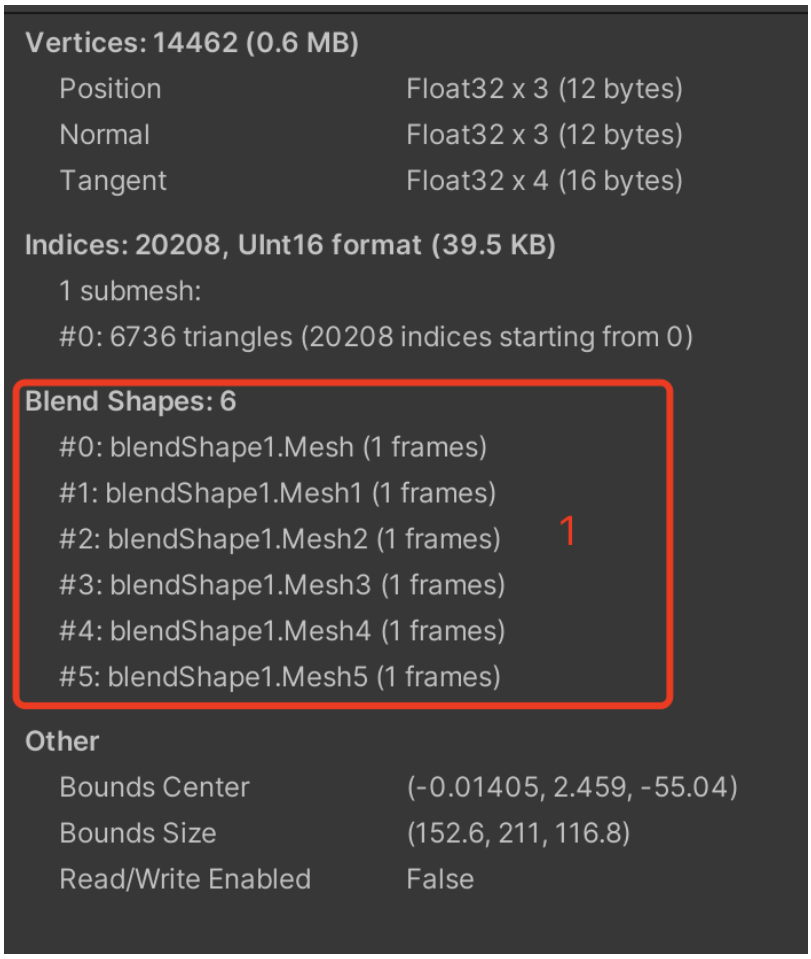


Quality Auto, the number of bones will be determined by QualitySetting.skinWeights. Others fixed options can be selected.



ShadowCastingMode will also convert to corresponding enumeration

Mesh



All the blend shape data in mesh will be also converted.

Frequently Asked Questions

At runtime, I get an error about “Cannot find TypeIndex for type hash”

For example: “`System.ArgumentException: Cannot find TypeIndex for type hash 9701635103632511287. Ensure your runtime depends on all assemblies defining the Component types your data uses.`”

The conversion systems generated data that refers to a type that is not present in your output executable. You can get the name of the actual type by looking in the build output directory (normally in the “Builds” folder inside your project folder, and then in a folder named the same as your build configuration), inside “Logs/SceneExportLog.txt”. Looking for the given TypeIndex shows:

```
0x86a321b9ac79d137 -    9701635103632511287 - Unity.Rendering.RenderBounds
```

Which gives you the “Unity.Rendering.RenderBounds” type name. There are two fixes:

1. If the output is actually supposed to use this type, then ensure that the root assembly references the assembly that contains the type.
2. If the output is not supposed to use this type, then you can filter out the conversion system that is generating it by adding a “Conversion System Filter” build configuration component to work around this:
    a. Add a “Conversion System Filter Settings” component
        i. Set the Size to 1.
        ii. Select the assembly you want to add in the first slot.

Note that the conversion systems from the hybrid rendering assembly has been automatically filtered out using the system flag WorldSystemFilterFlags.HybridGameObjectConversion in a dots runtime build. Only conversion systems using WorldSystemFilterFlags.GameObjectConversion or WorldSystemFilterFlags.DotsRuntimeGameObjectConversion are running at runtime.

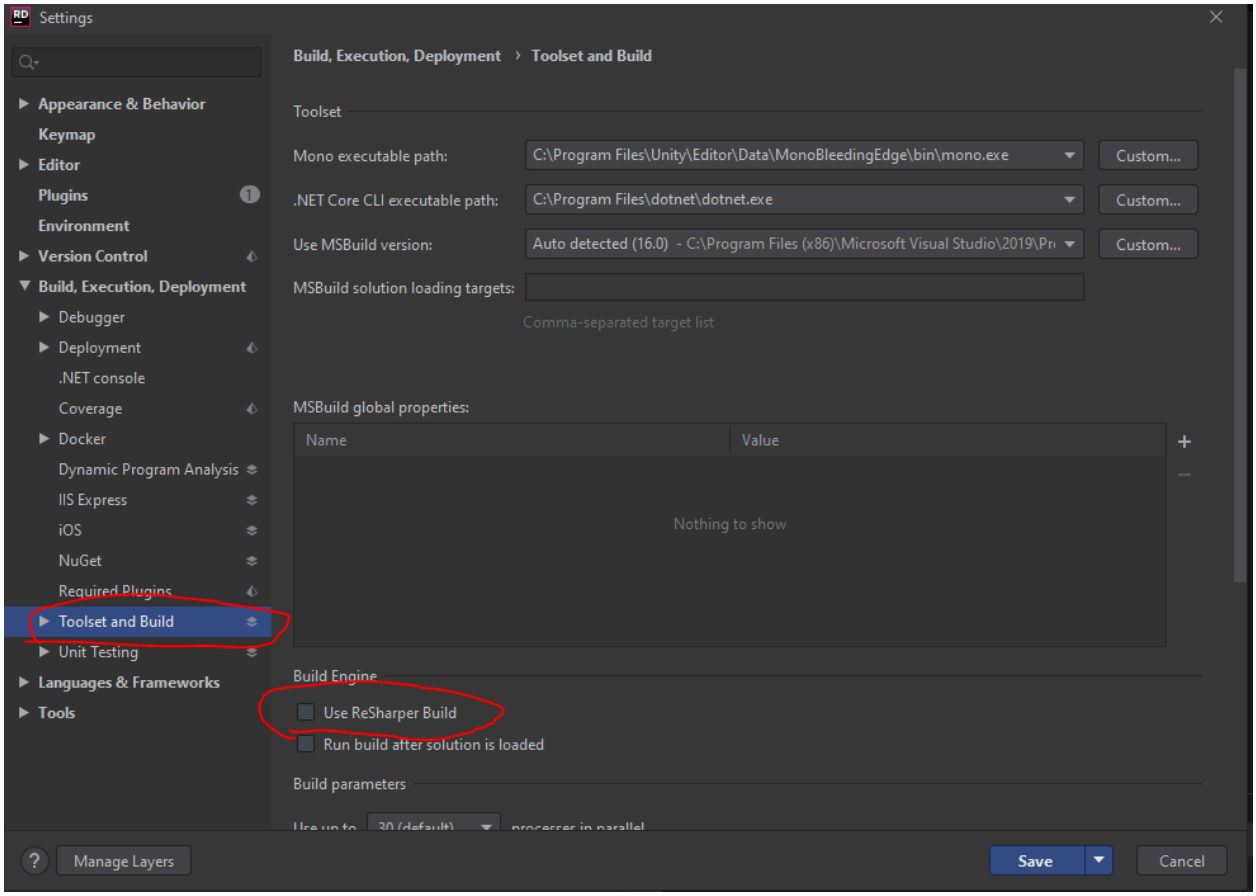
Is there a precise roadmap or launch date for Project Tiny?

We still don't have a firm date for our "V1" enabling hypercasual style scenarios. We want to ensure first that the feature set is complete enough to enable projects to go in production, so the cut line isn't defined yet. We'll share an update as soon as possible so early adopters can look at our plans and provide more feedback.

My code changes are not appearing in my game when building from Rider!

Ensure you do not have “Resharper Build” enabled:

- File > Settings > Build/Execution/Deployment > Toolset and Build
- Ensure “Use ReSharper Build" is unchecked



Resharper Build cannot deduce the final executable produced by our build pipeline, and will not detect that the application should be rebuilt.

Known Issues

The following issues will be fixed in a future release:

- asm.js builds: low performance in Chrome on all build modes.
    - Workaround: use WebAssembly
- Rendering in the scene view may look different than the runtime
    - Ongoing work to visually unify the renderers. Will be addressed over time in future releases.
- The Editor platform must be set to `PC, Mac & Linux Standalone`, unexpected errors may showup if the Editor is switched to different platforms
- Managed debugger may cause memory leaks on development and debug builds.
    - We are actively working on this issue and will be resolved in the next release
- Fixed timestep may take longer time per frame on Mac dotnet builds
    - Workaround: Use Il2cpp builds
    - It has been fixed in 0.33