The sound of Megacity

The audio system in Megacity has been written from the ground up specifically for this demo. It's a tech demo, so for us it was all about showing the potential of a lot of new technologies that form the low level foundation of all the new stuff we are building, so don't expect to see a full-blown user-friendly audio system here. That said, we are thrilled that this new tech stack enables us to open up the audio code to you in a way that wasn't possible before in Unity.

In particular this new foundation consists of:

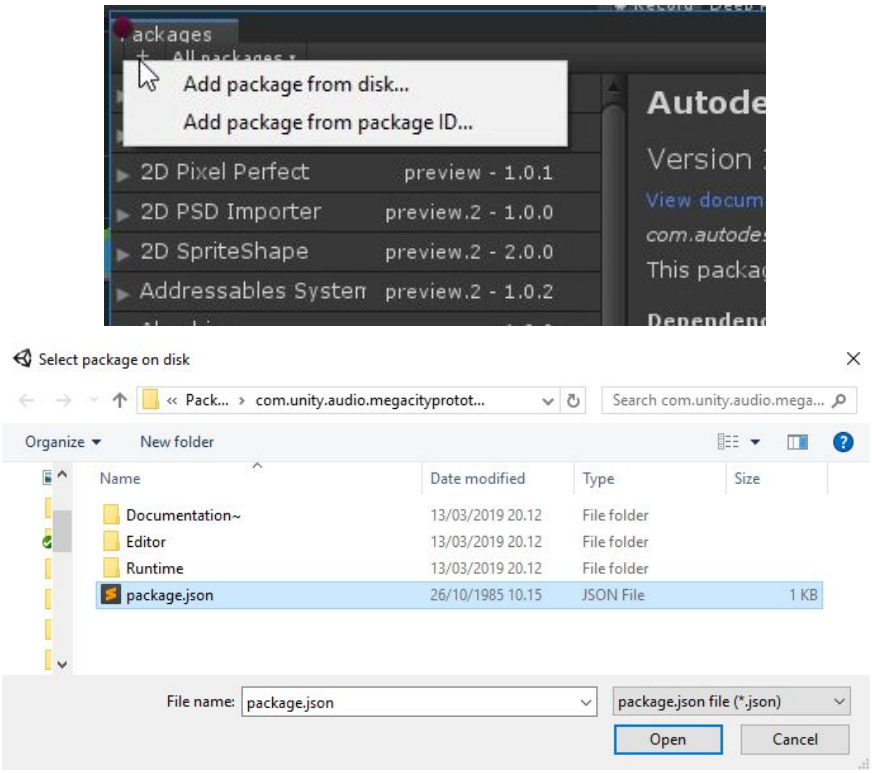
- DOTS Audio (the DSPGraph audio rendering network)
- DOTS ECS (the entity component system and its integration with the prefab system)
- Burst (the compiler that facilitates writing low-level code such as audio effects in C#)

This document will explain how these technologies interoperate to create the immersive audio experience of Megacity and will introduce the tools in the editor that helped us develop the systems and monitor their performance at various stages in the process.

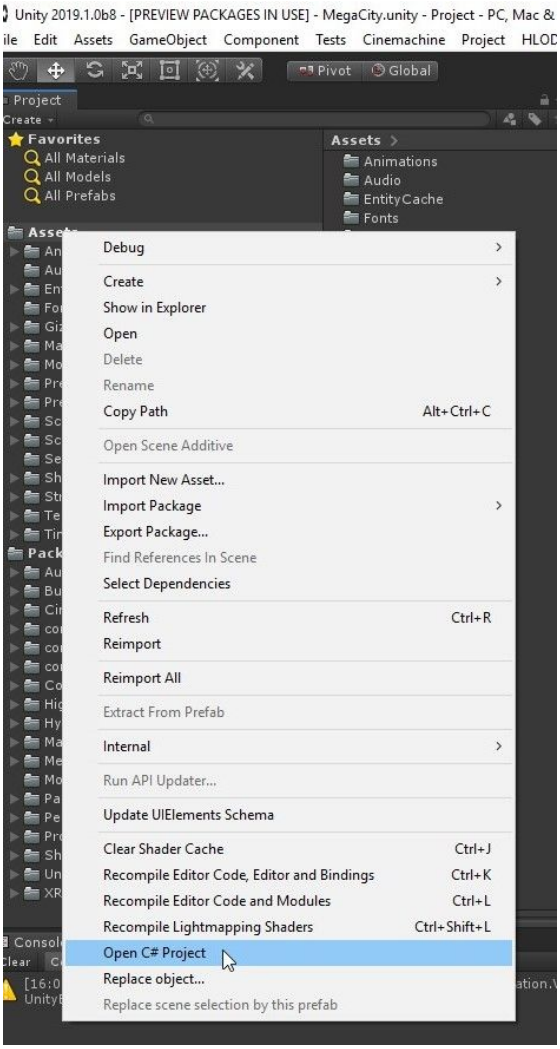
For a quick conceptual overview of the systems we suggest watching the presentation from the ECS Track of Unite LA 2018 Graph Driven Audio in an ECS World. Beware that some technical details have changed after the talk was held. We will try to note these in the following.

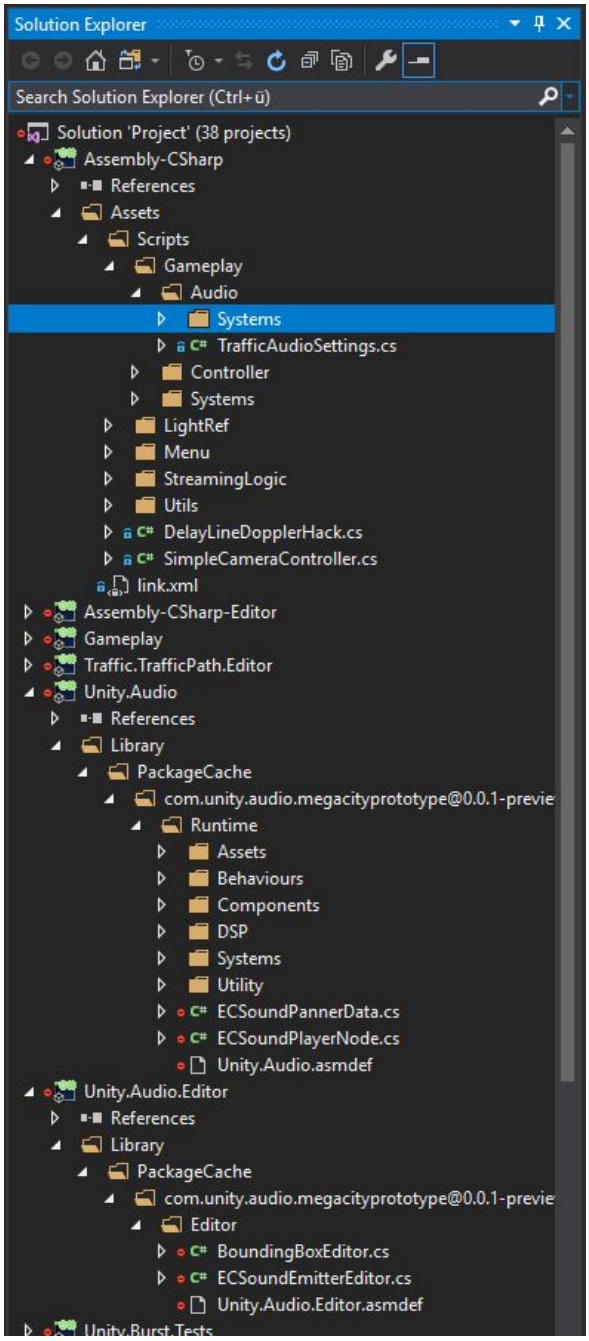
Setup

Most of the generic audio code we refer to in this document resides in the package com.unity.audio.megacityprototype which (like other non-embedded packages) is not automatically added to the C# Visual Studio solution when you choose "Open C# project" from the Project Browser. The .cs files can still be opened by double-clicking on the files in "Megacity Audio System" under the Packages folder in the Project Browser. To add the audio package to the Visual Studio project in order to set breakpoints in the code etc. you need to use the "Add package from disk…" option in the Package Manager to add the audio package and then regenerate the C# project using "Open C# Project" in the Project Browser, as shown below. The latter step will generate Unity.Audio.csproj and Unity.Audio.Editor.csproj which you can now add to the C# solution.



Note that in the screenshot above we just point to the package in the Packages folder of the project. Beware that these will be overwritten when updating the project, so if you plan on making changes, it's better to copy this folder to a different location outside of the project and point package manager to that one instead.

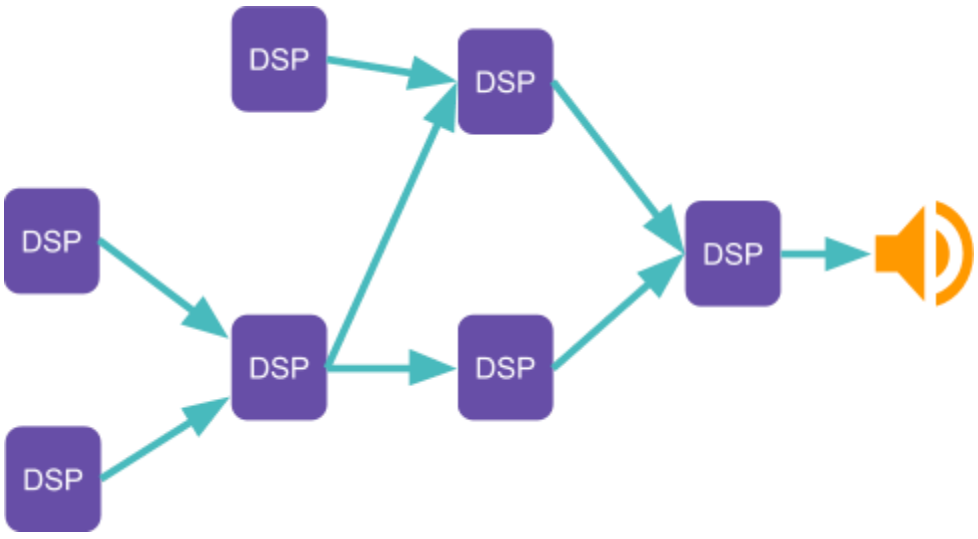

Unity 2019.1.0b8 - [PREVIEW PACKAGES IN USE] - MegaCity.unity - Project - PC, Mac &
ile Edit Assets GameObject Component Tests Cinemachine Project HLOD
Pivot Global
Project
Create
Favorites
All Materials
All Models
All Prefabs
Assets
Animations
Audio
EntityCache
Fonts
Debug
Create
Show in Explorer
Open
Delete
Rename
Copy Path Alt+Ctrl+C
Open Scene Additive
Import New Asset...
Import Package
Export Package...
Find References In Scene
Select Dependencies
Refresh Ctrl+R
Reimport
Reimport All
Extract From Prefab
Internal
Run API Updater...
Update UIElements Schema
Clear Shader Cache Ctrl+J
Recompile Editor Code, Editor and Bindings Ctrl+K
Recompile Editor Code and Modules Ctrl+L
Recompile Lightmapping Shaders Ctrl+Shift+L
Open C# Project
Replace object...
Replace scene selection by this prefab


Solution Explorer
Search Solution Explorer (Ctrl+ü)
Solution 'Project' (38 projects)
Assembly-CSharp
References
Assets
Scripts
Gameplay
Audio
Systems
TrafficAudioSettings.cs
Controller
Systems
LightRef
Menu
StreamingLogic
Utils
DelayLineDopplerHack.cs
SimpleCameraController.cs
link.xml
Assembly-CSharp-Editor
Gameplay
Traffic.TrafficPath.Editor
Unity.Audio
References
Library
PackageCache
com.unity.audio.megacityprototype@0.0.1-previe
Runtime
Assets
Behaviours
Components
DSP
Systems
Utility
ECSoundPannerData.cs
ECSoundPlayerNode.cs
Unity.Audio.asmdef
Unity.Audio.Editor
References
Library
PackageCache
com.unity.audio.megacityprototype@0.0.1-previe
Editor
BoundingBoxEditor.cs
ECSoundEmitterEditor.cs
Unity.Audio.Editor.asmdef
Unity.Burst.Tests

The DSP Graph

The first system we will look at is the new audio mixing engine in Unity called the DSP Graph. This is a buffer-based rendering system that runs in parallel with the main thread. In the current version of Unity there is one default graph that is automatically called whenever the audio driver requests new sound data, but conceptually it is possible to have multiple concurrent DSP Graphs, because internally the DSP Graph uses Unity's job system to set up topological dependencies between the data read and generated by nodes and to execute these on the job system's worker threads.

One notable difference to the C# job system is that in the case of the default graph the job scheduling is happening from the audio thread, which is necessary because the audio thread will run at a different rate from the main thread's frame update. That update rate is given by the DSP block size defined in Unity's Audio Project settings (typically 512, 1024 or 2048 sample blocks with target platform-specific variations), and the system's sample rate (typically 44100 or 48000 samples/second), so somewhere in the range of 10-45ms. Therefore sometimes the audio rendering will be faster and sometimes slower than the frame rate, and if you're lucky they align once in an aeon. This is not a problem though, and in fact a situation we're already used to from the existing audio system, where commands are enqueued from the main thread and shortly thereafter picked up by the mixing thread. What is new though is that DSP Graph provides a more structured approach to this in the form of the `DSPCommandBlock` that handles both topological changes such as adding or removing nodes as well as state changes such as changing node parameters or the connection weights. Megacity makes extensive use of connection weights in order to create a dynamic mix. More about this later.



The DSP Graph can be visualised using the custom DSPGraphInterceptor.cs utility included in the audio package. To use it though, you need to uncomment the first line that defines

`ENABLE_DSPGRAPH_INTERCEPTOR`. The reason it is this way is that currently it is not an integrated part of the DSP Graph API and as such it works by intercepting the calls we make to the DSP Graph command buffer API and thus maintains its visualisation data structures separately of the DSP Graph. That said, the difference between using the raw DSPGraph API and the interceptor code is minimal, so it is really only the creation of command blocks that changes from this (in vanilla DSPGraph code):

```
DSPGraph graph = …;
DSPCommandBlock cmd = graph.CreateCommandBlock();
```

To:

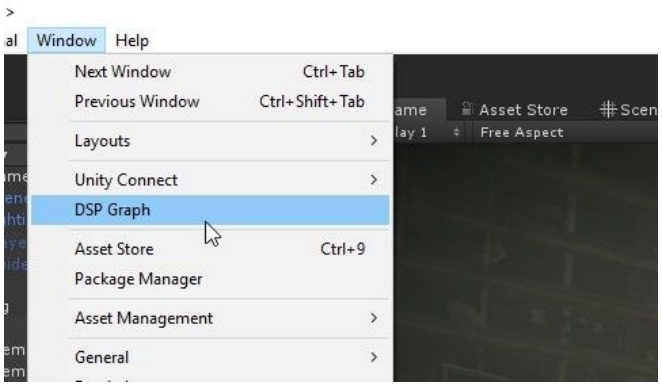
```
DSPCommandBlockInterceptor cmd =
     DSPCommandBlockInterceptor.CreateCommandBlock(graph);
```

After this the `DSPCommandBlockInterceptor` provides all the API from `DSPCommandBlock` that we needed for the purposes of the demo as well as some additional functions such as:

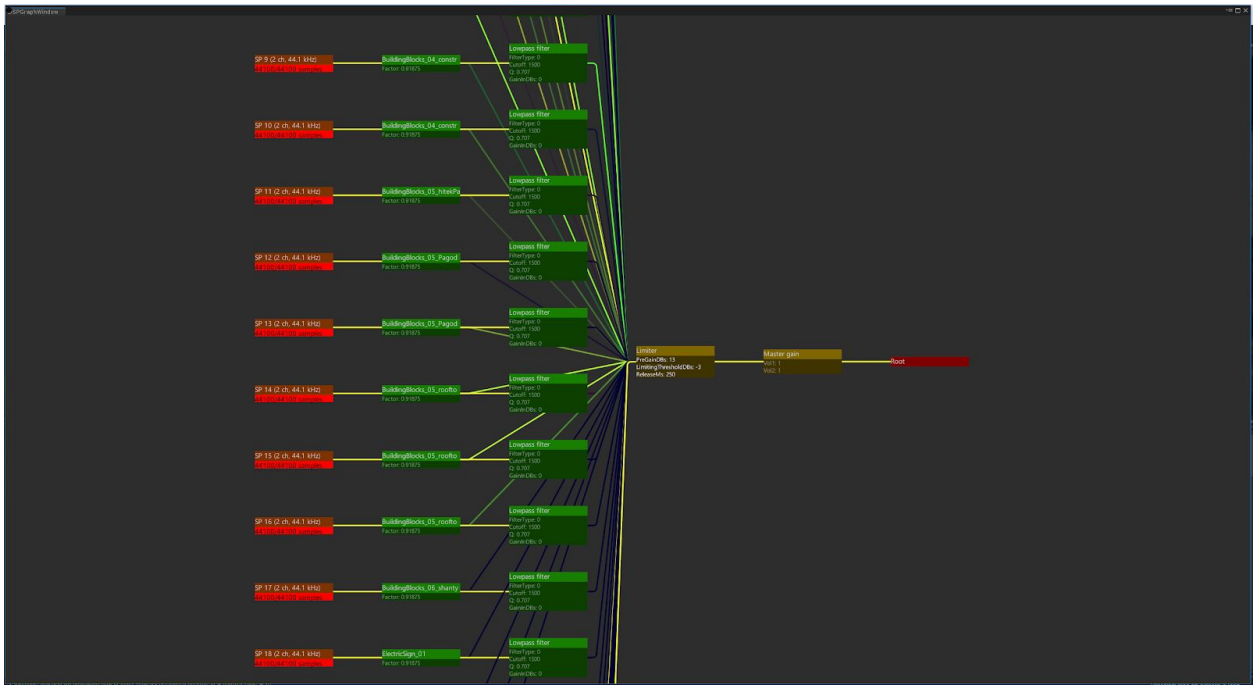
```
DSPNode carSound = …;
DSPCommandBlockInterceptor.SetNodeName(carSound,
     "My Summer Car",
     DSPCommandBlockInterceptor.Group.MainVehicle);
```

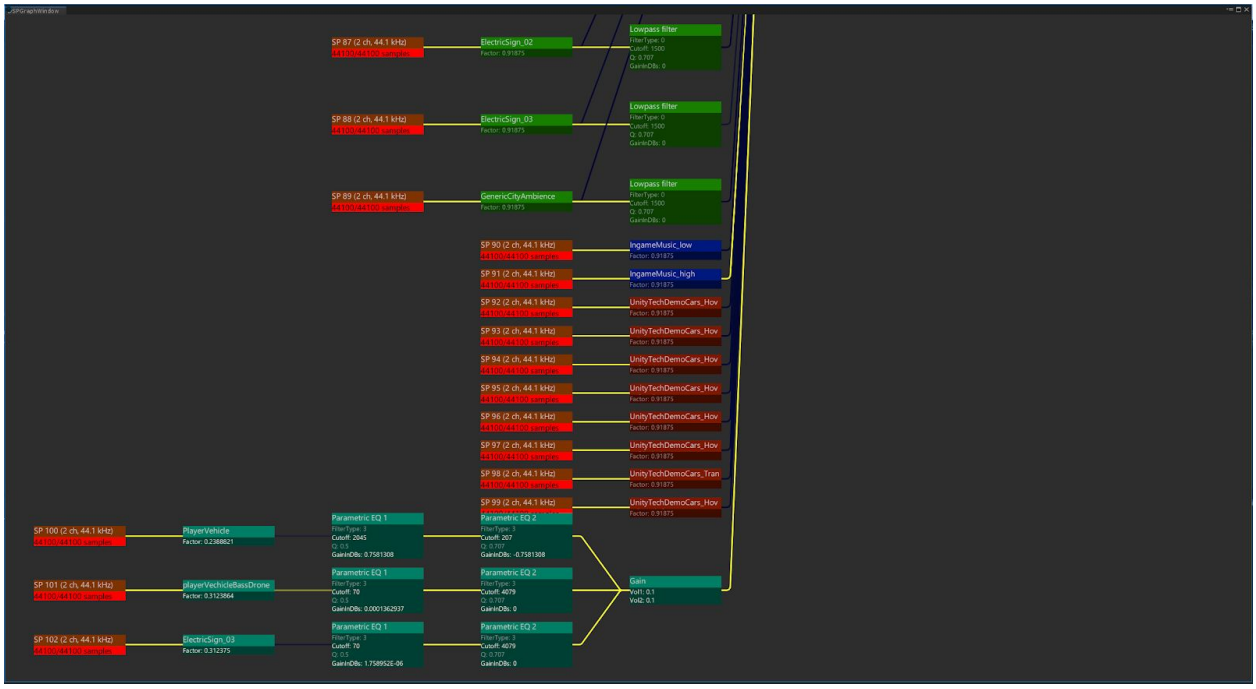
that allow naming a DSPNode and adding it to a sound group that defines its colour.

After uncommenting `ENABLE_DSPGRAPH_INTERCEPTOR` you can now see a live visualisation of the nodes and their connections in the window that pops up when selecting “DSP Graph” from the “Window” menu of the Unity editor. Note that since DSPGraphInterceptor.cs is inside a package, it is read-only, so you need to make it writable to change this. Many text editors will do this after prompting, but if this is not the case, you need to right-click the file in Windows Explorer or MacOS finder to change the access file attribute.



There won’t be anything inside the window before you start playing the game. This is because the DSP graph is first built at startup. You will also notice that the visualisation looks different from what was shown in the Unite talk. This is because the version used in the talk used IMGUI for rendering while the current version uses UIElements for drawing the nodes and a tiny bit of GraphView for drawing the connections. This also means that zoom and pan is taken care of by GraphView: Use the scrollwheel on the mouse to zoom in or out and hold down the middle button or the left button + alt (Windows) / command (Mac) to pan the view.





The graphs are read from left to right, where the leaves to the left are instances of `AudioSampleProvider` that wrap codecs that decode compressed audio into a stream at a rate of 44100 float values per second, similar to AudioSources.

The red bar at the bottom of each the `AudioSampleProvider` nodes shows the current buffering state of each decoder and should always be filled. If you fly the car into an area with busy traffic and do some fast flybys with cars from the opposite direction, you will see more sources dynamically popping up at the bottom of the graph for the special effect flyby sounds. As the flyby sound effect plays out you can see in the visualisation how the buffer is being drained.

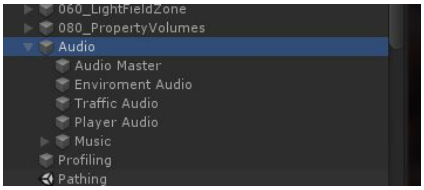
On the right side of the upper graph you see the output node of the DSP graph, which - since this is the default graph that is driven by the sound card - is the final signal that the player gets to hear. All remaining nodes have both inputs and outputs and are thus effects. The intensity of the wires between the nodes shows how much of the source signal flows into the receiving node. Furthermore the effect nodes have text labels showing the controllable parameters that they expose. Here the intensity shows how recently a parameter was changed, so parameters that were just changed appear white and then are gradually dimmed over time.

What is also instantly visible in the graphs is that the outputs of audio nodes can be read multiple times. This is where audio is different from other systems such as animation systems, and this is also a feature we use to great extent to reduce computations, as is explained in the following section. Also, why do the connections in the graph have those fancy colors? Well, we'll cover that in the section that explains the ambience sounds, but since this is a quite complex system, let's first take a look at simpler systems like the main vehicle and music, as these will help understand some basics about DSP Graph.

Note that the DSP Graph visualisation utility does a lot of dynamic memory allocations that will affect the frame rate, so make sure to comment out the define whenever you don't need it.

Main audio setup

We keep all shared audio setup in the main Audio prefab. This provides an easy way for the people who don't work with audio or want to do profiling without it to disable individual audio systems by simply disabling the sub-gameobjects that represent environment and traffic sound fields, main vehicle sounds and music. Many of these sub-gameobjects also correspond to different ECS job component systems.

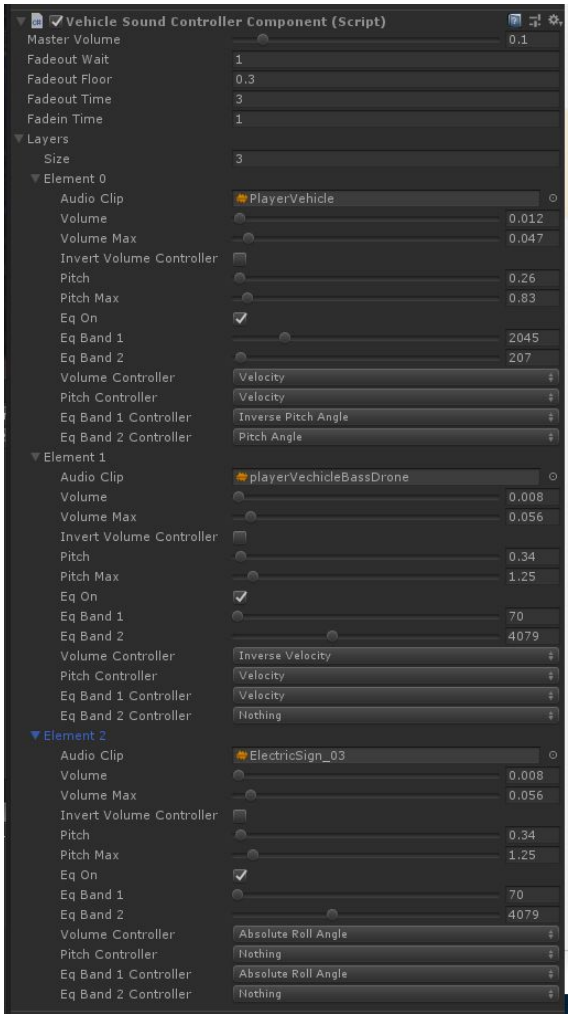


Relevant source files:

`AudioMaster.cs`
MonoBehaviour that manages basic ECS world that is used by all ECS audio systems and is used by these to query whether audio is enabled or disabled.

The main vehicle sounds

The main vehicle sounds should give the player a feeling of empowerment, so it is important that it tracks the actions on the controller to great detail. Therefore it's not just one sound playing, but a stack of 3 sounds with dynamic mixing as well as modulated filters.
The basic engine noise is generated by the "PlayerVehicle" audio clip with the additional "playerVehicleBassDrone" sound and "ElectricSign_03" added on top in amounts controlled by whether the car is accelerating, rising, falling or turning.



Relevant source files:

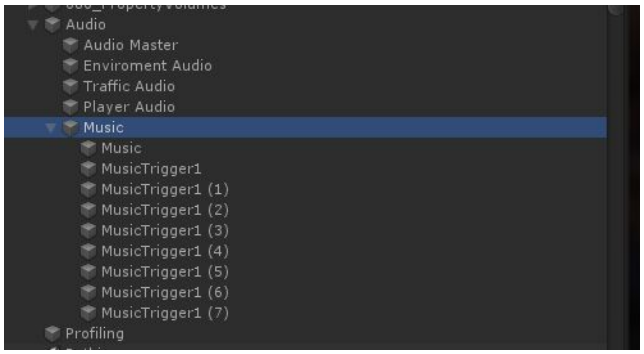
`VehicleSoundControllerComponent.cs`
Shared component data defining the settings and sound layers of the main vehicle described in the next section.

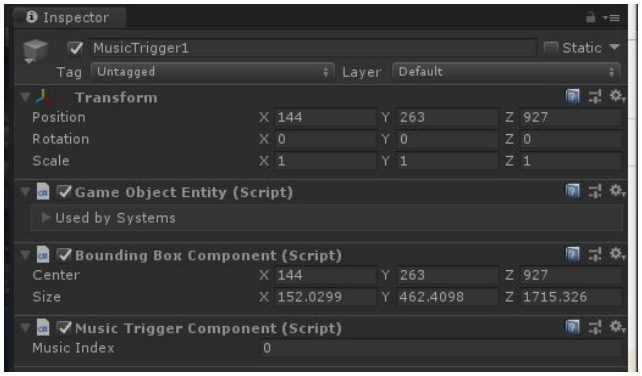
`VehicleSoundControllerSystem.cs`
Shared component data defining the settings and sound layers of the main vehicle described in the next section.

Music

Since Megacity is a tech demo, the music is very simple and basically just consists of a low and a high intensity part, which have the same musical content but just vary slightly in brightness and complexity of the sounds. Its main function is to emphasise certain parts of the level coarsely following the amount of content that is visible on the screen. As such the music system

consists of a main music player component that defines synchronous layers and 8 associated trigger boxes that switch the player component to a particular layer over a specified crossfade time. These trigger boxes are aligned with passages in the level such that the area around the starting position and open spaces trigger fades to the mellow “low” version while areas with dense traffic trigger the energetic “high” version of the soundtrack.





Relevant source files:

`MusicPlayerComponent.cs`
Sets up the relevant DSP nodes and manages crossfades initiated by the box trigger system.

`MusicTriggerComponent.cs`
MonoBehaviour wrapper for the shared component that holds the index of the music to

play.

`BoxIndexComponent.cs`
MonoBehaviour wrapping the BoundingBox that defines a range that can trigger a fade towards a particular indexed piece of music.

`BoundingBoxEditor.cs`
Editor gizmo for setting up bounding boxes.

`BoxTriggerSystem.cs`
The system that checks points (in this case only the player location) against a number of boxes and triggers music changes whenever the box that collides with the player changes.

Ambience sounds

Megacity is all about streaming a very large scene, so it was clear from the beginning that we needed a special way to fill the scene with sound that can deliver a rich sound experience that maps closely to the scene geometry and ties into the streaming system. When we started working on the project, live editing wasn't available, so placing sounds manually on the building blocks would have been a very time consuming task.

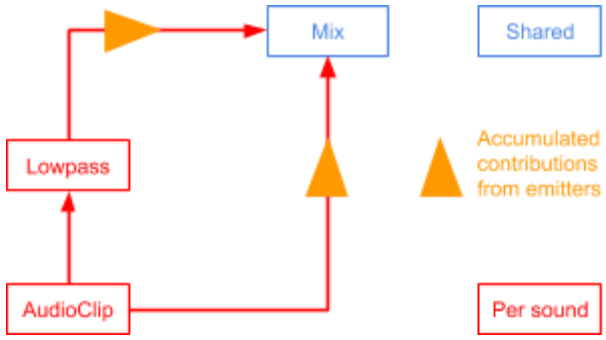
We went for a different route noticing how the large amount of air conditioners mounted on the buildings provided a surprisingly accurate (in terms of audio) representation of the geometry in the scene. What we needed was just to view the aircons as a point cloud of sound emitters that approximate all sound coming from the buildings. Note that these are not all aircon sounds, but all kinds of sounds you might imagine being produced inside the building, such as televisions, radios, people talking, dogs, birds, electrical devices, construction work etc.

Now that we have an easy way to place sounds on the buildings, there are still way too many sounds to place these manually. Sure, we could have one ambience sound per building block, but this would result in a very slowly changing sound field, and what we wanted was a very lively and immersive sound experience, especially as you fly close to the building and to do that every emitter would need its own sound. Especially for the street lights it is important that they have their own sound and distinct location as you're flying near it.
But still -- even with ECS, Burst and DSPGraph, with roughly 3000 aircons per block there is no way we can play unique sounds for each of the aircons and each of the street lights. This is where the ability to connect an output to multiple target nodes comes in.

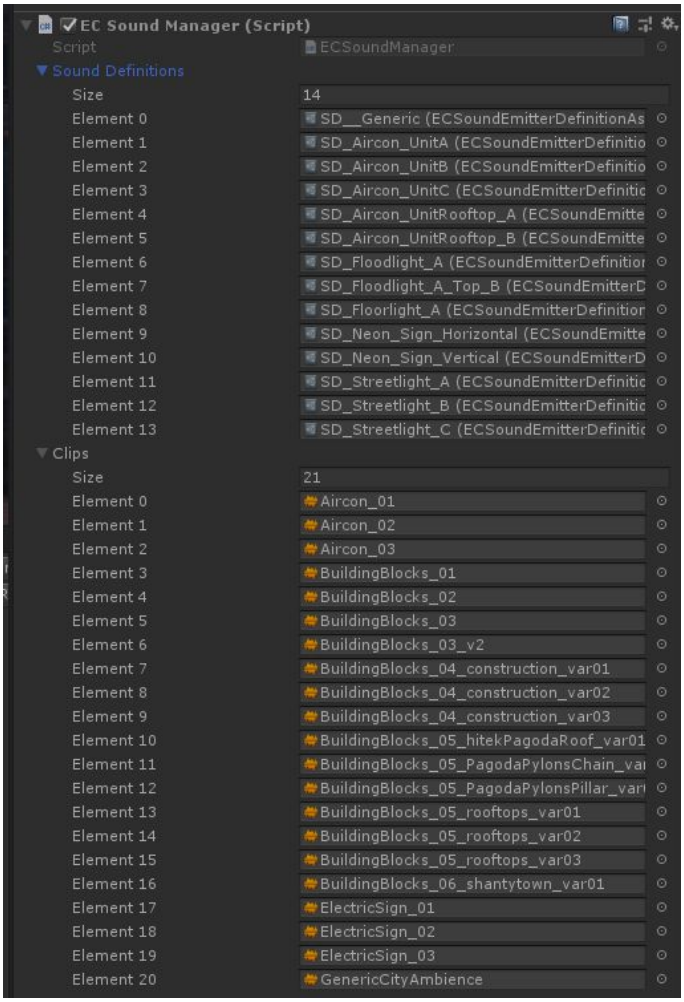
One thing that is common for the examples of sounds listed above is that they are looping textures without distinct events in them that would be easily noticeable when listening to them for a while. This enables us to reuse the same sound in multiple different locations without it

being too obvious, just like a level or heightmap landscape designer would only blend between a couple of characteristic textures using alpha maps to create a slowly varying texture that still carries the details of the source textures but blends them in ways that hides the repetition of the pattern. We do exactly the same for the field of sound emitters in Megacity.

In fact for the sounds coming from the buildings we have only 21 actual sounds playing at all times. These sounds are set up in the a MonoBehaviour that rigs up the basic graph that is shown in the first image of the DSP Graph visualizer. The structure that is repeated for each of these sounds is the one shown below:



Here Mix denotes the rightmost final output node. All the red and orange parts are instantiated for each of the 21 sounds. The picture below shows the main setup of these sounds in the Audio prefab. The sound definition concept will be explained later.



The connections between the nodes are not single-valued weights (shown as orange triangles), but actually represent vectors of volume multipliers for the left and right channel. Due to time constraints we limited Megacity to stereo, and the current version of Unity only supports mixing up to 4 channels. This is subject to change as we find a better way to express this API. And now it's time to reveal the mystery of the colors of the connections: This is simply a combination of red and green, where the red component visualises the volume of the left channel whereas the green component visualises the volume of the right channel.

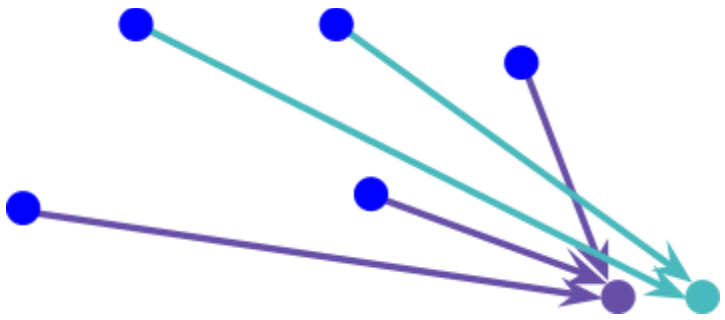
We also use the weights to implement dynamic panning of the sounds and to simulate whether the sound is in front of or behind the player. This is done by controlling the mix such that sounds in front are routed primarily into the final output Mix, whereas sounds that are behind the player are routed primarily to the Lowpass to cut some of the high frequencies and thus simulating the effect of the ears blocking the sound.

As mentioned the basic structure is repeated for each sound, resulting in the overlaid connections seen in the graph and this is where we make heavy use of the DSP Graph's ability

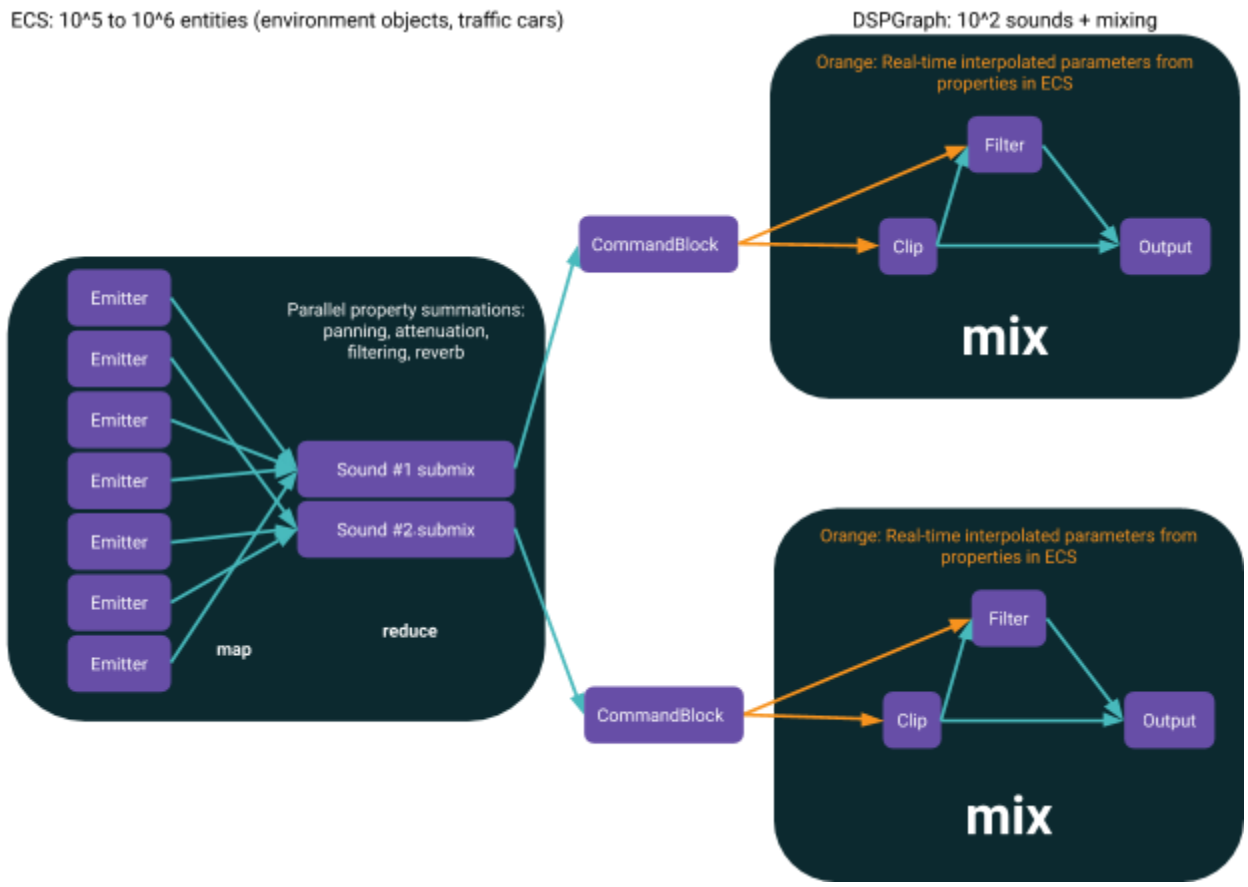
to have diamond-shaped mix structures, where a signal from one node is read by multiple nodes and the processed result is summed at the input of yet another node.

Before we move on, a quick note about which parts are implemented in native Unity code and which parts are written in user C# code. This is subject to change as we get Burst integration in more parts of the code, but at the time of writing all mixing of audio data between nodes is done by the native DSP Graph engine inside of Unity. Likewise applying the left/right mix gains during the mix as well as interpolation of parameters is done by the native part of DSP Graph. Also all `AudioSampleProvider` instances are calling into native libraries for the various decoders that Unity supports for its different AudioClip types. We are investigating ways to move the mixing code and perhaps even codecs to Burst also. Stay tuned.

But wait, we still haven't explained how we get 100000 concurrently playing sounds in the scene with this setup! Well, in a way we have, because the basic per-sound topology described above was designed with just that in mind. If you look at the orange triangles representing the weights, these are not even single weights for one sound, but actually represent a collection of emitters that play the same sound in synchrony. This means that we only need to allocate an `AudioSampleProvider` and its associated lowpass filter and mixing topology for groups of emitters that play the same sound, typically 1000-5000 emitters. We then use ECS jobs to sum all contributions of the emitters belonging to a group and apply the results as weights to the connections. This is shown below where we have a cloud of blue emitters that add small sound level contributions to the purple and cyan sounds.



Before we take a look at how the actual panning is calculated, let's look at the final graph to see how emitters contributions are collected using ECS systems and passed to the DSP Graph via command blocks:



A small but important technical detail is that the calculation of the mix contributions happens in two passes that are orchestrated by the `ECSoundFieldMixSystem` component system. This splits up the workload of calculating the emitter contributions into smaller workloads (`CalculateMixFieldJob`) whose results are then combined in a final dependent job (`MixIntegrateJob`).

Relevant source files:

> `ECSoundEmitterComponent.cs`:
> The struct `ECSoundEmitter` defines the emitter position and orientation and is baked into the .entities subscene data). The file also contains the structure `ECSoundEmitterDefinition` which contains parameters that define the spatial propagation of sound as well as the indexed range of playable sounds for this specific emitter.
>
> `ECSoundPannerData.cs`:
> Provides storage and methods to calculate the mix contributions of a single emitter given `ECSoundEmitterDefinition` data describing its properties and the listener and source positions.

`ECSoundFieldMixSystem.cs`:
The ECS component system that performs the calculation of all emitter contributions and writes them to `ECSoundFieldFinalMix` component data that is then applied to the sounds in the DSP Graph by `ECSoundSystem.cs`.

Runtime-editable assets

Quick recap: We found a quick way to use existing level geometry to place hundreds of thousands of emitters all over the scene geometry and a way to gather these to write the accumulated levels into a small number of actual playing sounds in the DSP Graph. At this point we still haven't discussed how the emitter component data actually gets stored and how we access it in the ECS jobs.

Megacity uses the hybrid ECS in a lot of places, including the storage of emitters. What this means is that the entity component data is wrapped inside a MonoBehaviour in the editor, and extracted from this during a scene bake. This way the final runtime only contains the minimal amount of data laid out in a way that is efficient for streaming.

Two new key concepts in ECS are those of baking and working with subscenes. Baking is the process by which the representation that is optimized for making modifications in the Unity editor using MonoBehaviours and Components is converted to a representation that is optimized for compact memory layout and fast load times. Subscenes are an important tool for splitting up large scenes into smaller ones that can be dynamically streamed in and out at runtime, but they also help reducing bake time, since a small modification in a specific part of the scene doesn't require the whole scene to be baked again, but only the subscene it belongs to. If these concepts are new to you, it may be a good idea to take a look at https://github.com/Unity-Technologies/EntityComponentSystemSamples which has a simplistic HelloECS example that demonstrates the use of subscene baking and links to further documentation.

The sound emitter data `ECSoundEmitter` consists of 3 pieces. This is data that is tied to the specific emitter and is only changed when adding completely new types of emitters. During a bake we extract the `definitionIndex` from the `ECSoundEmitterDefinitionAsset` that the MonoBehaviour references and store it in the emitter. Likewise we get the position and forward vector from the Transform and store it in position and `coneDirection`. The `definitionEntity` and `soundPlayerEntity` fields are resolved by `ECSoundFieldMixSystem` at runtime on first occurrence (i.e. right after they got streamed in) after which the resolution is no longer necessary and the job only needs to calculate the mix contribution of the emitter.

We should mention that it is not good practice to keep them in the serialized data of `ECSoundEmitter`, and that it would be better to keep them in separate component data added at runtime, or, even better, store the references to the definition as a shared component data. There are several reasons why this is not possible in this particular case, among others that the sound definitions belong to the main scene, so there is no way for subscenes to reference these, and also both sound player and sound definition entities are only created upon entering playmode. Furthermore at the time of writing adding separate component data at runtime was not an option, because `EntityCommandBuffer.AddComponent` was not burst'able due to its dynamic type lookup, and so we abandoned the streaming solution that was outlined in the Unite LA talk. We want to emphasize that the idiomatic way to do this is to use the approach described in the talk, but the current solution was chosen to match the performance targets given the supported feature set of ECS at the time.

```
public struct ECSoundEmitter : IComponentData
{
	// Baked data
	public int definitionIndex;
	public float3 position;
	public float3 coneDirection;

	// Resolved on first load
	public Entity definitionEntity;
	public Entity soundPlayerEntity;
}
```

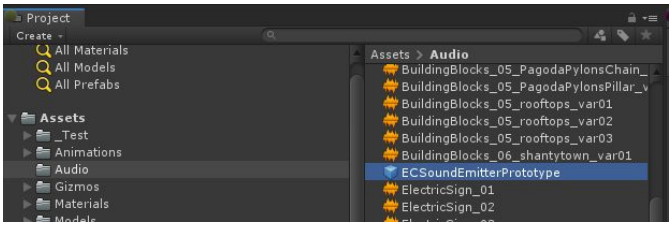
We should also note that emitters are still streamed in like any other subscene, as can be seen in the Entity Debugger window, and this keeps the number of emitters at a reasonable size at all times. As a subscene typically contains 3000 emitters, the overhead of searching for the matching definition and sound player entities for newly streamed-in emitters does not cause large deviations in the workload of the emitter mix contribution jobs.

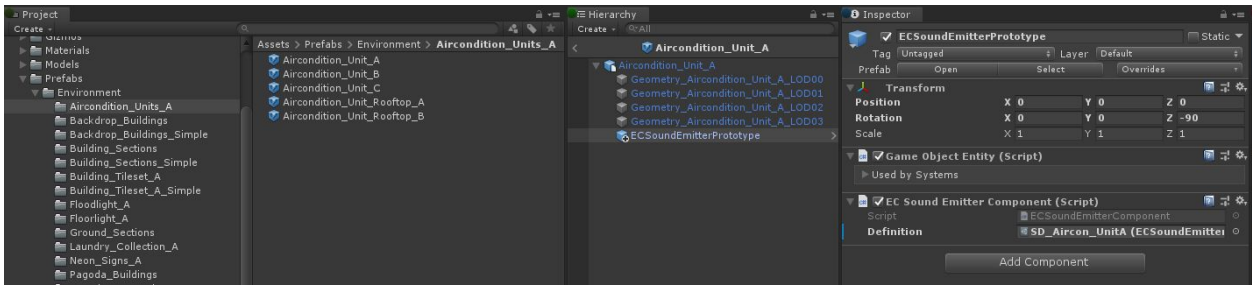
The mixing of emitter contributions does not happen in a single pass. Instead, `ECSoundFieldMixSystem` splits the work into two parts, the first of which goes wide and calculates mix of all emitters being processed on the same thread by adding emitter contributions to a shared NativeArray at an index defined by the job's thread index, while the second single job accumulates all these results into final mix levels that are applied to the looping sounds by `ECSoundSystem`.

The `ECSoundEmitter` data is produced by the editor representation `ECSoundEmitterComponent` which is just a normal `MonoBehaviour` that implements `IConvertGameObjectToEntity`. It has a function `Convert` that is invoked for each game object at subscene conversion time in order to add the necessary component data to the

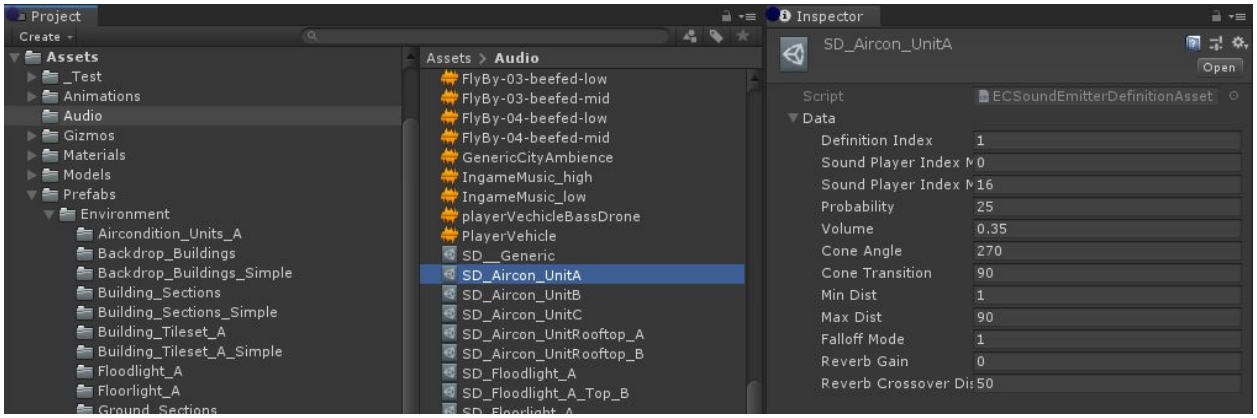
associated entity that becomes the light-weight runtime representation of the emitter in the baked subscene.

All emitters are based on the `ECSoundEmitterPrototype` prefab. All it does is set up a GameObject that has an `ECSoundEmitterComponent` that references an `ECSoundEmitterDefinitionAsset`.

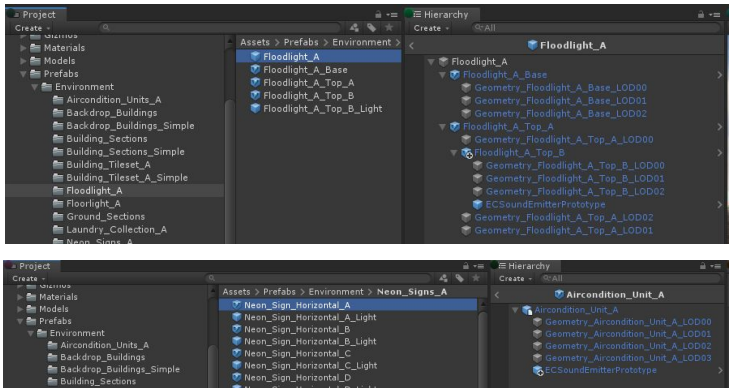




All sound definition assets are prefixed with “SD_” in the Assets/Audio folder. This is just a convention we use to make it easily searchable. The sound definition contains just one field of the type `ECSoundEmitterDefinition` which is a plain struct that is reflected from the ScriptableObject into the associated runtime-instantiated entities so that the information can be read from ECS jobs.



Aside from the aircon prefabs shown above, the other two use cases for the `ECSoundEmitterPrototype` prefab are the floodlights and neon lights, shown below:



Relevant source files:

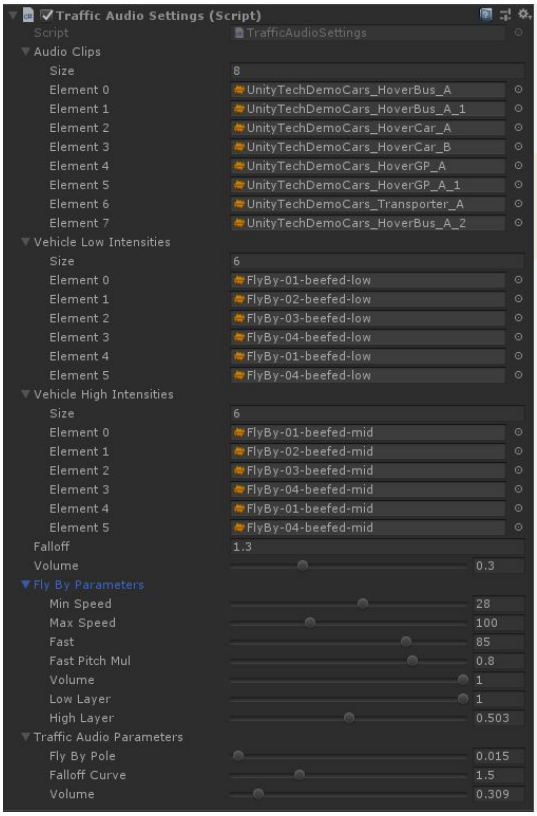
> `ECSoundEmitterComponent.cs`
> The struct `ECSoundEmitterComponent` defines the classical Unity component that is used to wrap emitters in prefabs. Note that because `ECSoundEmitterComponent` is a MonoBehaviour the file name must be identical.
>
> `ECSoundEmitterDefinitionAsset.cs`
> A ScriptableObject that stores the shared `ECSoundEmitterDefinition` data. At startup the `ECSoundManager` reflects this data into entities that are then accessible to the systems that process `ECSoundEmitter` components.

Audio in the traffic system

The traffic system is very similar to the emitter system conceptually, but since it does not deal with static scene geometry there is no data that needs to be baked for it. It obtains all live positions of the vehicles from the game code traffic system.

Like the static ambience emitter system it sums up volume contributions of a number of `VehicleEmitter` components and applies the result to 8 looping engine sounds that represent the different types of cars in the scene.
Since the number of vehicles is relatively low compared to the number of ambience emitters in the scene the `CalculateMixContributions` job of `TrafficAudioFieldSystem` stores the individual mix contribution within each emitter and combines them in the final `FoldEmitterFieldsToSamples` job, but the thread index based submixing could have been used here too.

In addition to the diffuse sound field there are also flyby sounds that are triggered whenever the main vehicle passes by other cars within a specified threshold. As the vehicle sound field system's `CalculateMixContributions` job computes the distance between the other cars and the main vehicle, it also maintains a list of nearby flyby candidates that is processed by the `SelectFlyByEmitters` job in order to select flyby sounds to actually play.



Unlike the other systems, the traffic audio are not located in the audio package, but are placed in the Megacity project itself. This is due to the tight integration with the rest of the traffic system from which live position data is obtained, but also to reflect that this is more game-specific code than the rest. Hence both the fly-by and traffic sound field systems are located in the Assets/Scripts/Gameplay/Audio/Systems folder.

Relevant source files:

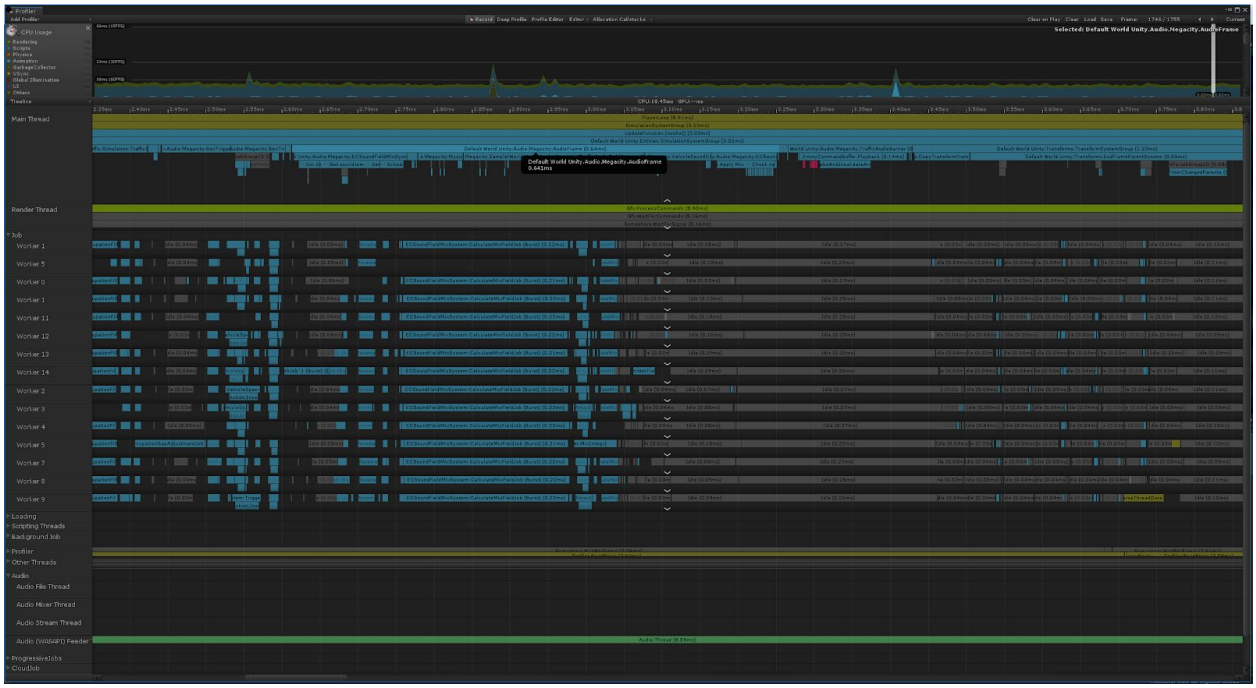
`FlyBySystem.cs`
Triggers one-shot fly-by sounds for cars that pass by within a certain radius.

`TrafficAudioSystem.cs`
The main system that drives the jobs that calculate and gather car emitter levels and orchestrates the triggering of fly-by sounds.

Debugging tools

While the DSP Graph with its live view of connection volumes and node parameters is a good tool for quickly tracking down logical errors in the audio code, the two most important tools in monitoring the performance of the audio are built-in tools:

First let's look at the Timeline view of the Profiler Window (Window > Analysis > Profiler) which is probably already well-known, so we'll focus on how to spot the ECS bits in this window:



As seen in the top picture, the timeline provides a convenient way to see all the ECS systems and barriers on the main thread. It's this order of component systems that determines when the jobs get scheduled, so therefore it is important to think about and influence this order by adding the

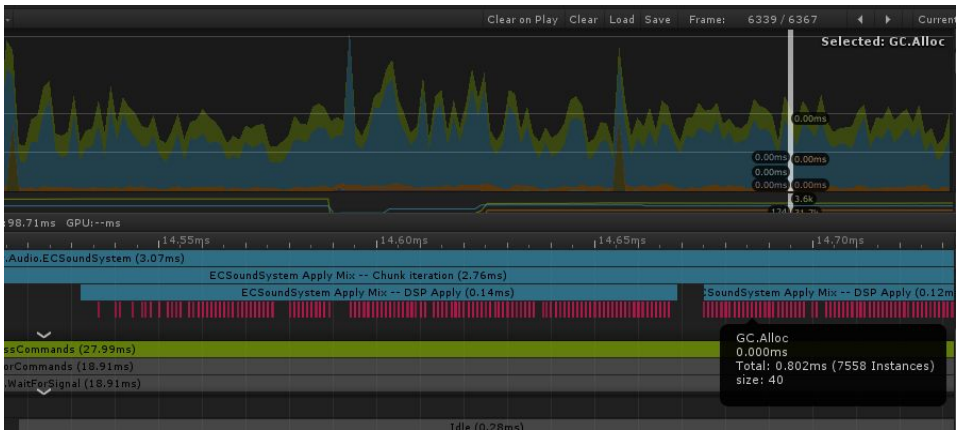
`[UpdateAfter(typeof(SomeOtherComponentSystemDependency)]` and/or
`[UpdateBefore(typeof(SomeOtherComponentSystemDependency)]`

attributes to the component system to specify the point in time at which it should be executed. These attributes can also be added to a ComponentSystemGroup such that other component systems can be specified to be updated within this group using the

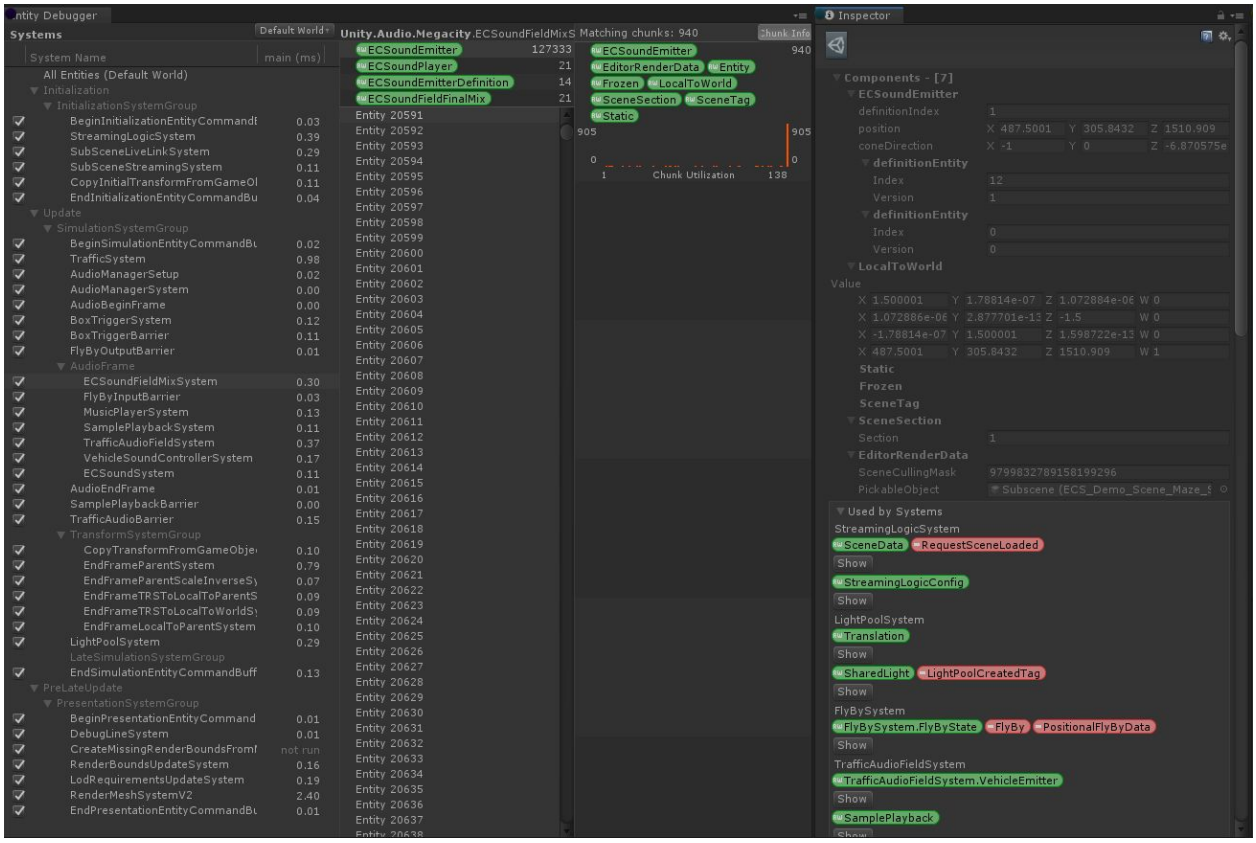
`[UpdateInGroup(typeof(SomeComponentSystemGroup)]`

Attribute. This way it is easy to set up hierarchies of dependencies at the granularities that allow for individual parallelisation of jobs. To see how the latter is used in the audio system look at the declaration and references of `AudioFrame` which is also seen as the highlighted block in the upper image.

Note that due to its simplicity the DSPGraphInterceptor does make a lot of dynamic allocations (and thus triggers garbage collection) which will be clearly visible both as spikes in the CPU Usage pane of the Profiler window as well as red boxes in the timeline view, as shown in the image below. We provide this visualization tool primarily as an aid in understanding what's going on in the DSP Graph and because we needed a quick visualisation for the presentation, so we will of course make sure that a future visualiser released as an actual feature will be allocation-free.



While the timeline is useful for optimising component system and group ordering for maximum concurrency, the Entity Debugger (Window > Analysis > Entity debugger) is often better suited for debugging logical issues. As seen below, we can click on a specific audio system and get stats on the number of entities currently loaded. For the sound emitters this is a variable number, since emitters get streamed in and out along with the buildings. Furthermore for a given component type used by a system we get a detailed description of the other component data residing in the chunk as well as a graphic visualisation of the amount of chunk fragmentation to the right. It is also possible to click on a specific entity and inspect the component data inside it. So, for an emitter for instance, we get a detailed reading of its position, orientation and index of its sound definition.



To the left we also see a breakdown of the precious main thread time spent in the different system, presented in a way that is easier to access than having to search for it in the timeline of the profiler window.

Writing DSP effects in C#

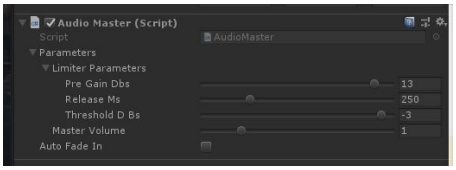
Since DSPGraph by itself doesn't provide any DSP processing tools other than per-channel attenuation we needed to implement the lowpass filter from scratch. There is also a Gain node that is still in for historic reasons, because we used this before adding the connection attenuation feature, so we recommend looking at this for the simplest possible effect or base for custom effects.

The lowpass filter we use is actually a multimode filter, meaning that the same code kernel supports calculation of both lowpass, bandpass and highpass at the same time. The details and derivation of this particular implementation of a so-called state-variable filter can be found on page 31 of https://cytomic.com/files/dsp/SvfLinearTrapOptimised2.pdf by Andrew Simper.

At the time when this code was written, there was no way to perform memory allocation for DSPNodes. This has changed since, as `IAudioJobUpdate` now provides a mechanism

through `ResourceContext` that allows allocating memory that will be freed upon destruction of the `IAudioJob`. So given the limitations back then, the code is a bit more convoluted than it needs to be in the sense that the dynamic m_Channels array containing per-channel state information for the filter is allocated outside of the job and handed to the job through the custom `UpdateJob` which gets executed right before (and on the same thread as) the DSP job, such that it can swap in the allocated array of channels before the DSP needs it. Likewise, when destructing the effect, we use a custom `DisposeJob` to unbind the channels array from the job such that it can be safely deleted on the main thread while the DSP job continues to execute until it's destroyed. This is also the reason why the DSP job's `Execute` has a check for the channel length being zero, as this indicates that either the channels array wasn't initialised yet, or the job is in the process of being destroyed. This all becomes a lot easier with memory allocation through `ResourceContext`, and to see how it works you may take a look at `SoundPlayerUpdateJob` in `ECSoundPlayerComponent.cs` which uses this mechanism to allocate a resampling buffer.

The finishing touch is added by the limiter effect that pumps up the volume to a consistent level both when the soundscape is idle or when a loud flyby sound is played.



Finally for XBoxOne compatibility we added a simple stereo to 7.1 converter, which simply copies the left and right channels from the stereo signal to the corresponding channels in the 7.1 frame and writes zeroes in the rest. This was needed because 7.1 is the native format for XBoxOne even when something else is specified. However, due to the flexibility of DSPGraph it is entirely possible to have a DSP node like this converter, that consumes inputs in one format and outputs in a different one. Thus the rest of the graph still only renders in stereo and therefore has comparable CPU usage as the other platforms. This is something that the old AudioSource based system does not allow us to do.

Relevant source files:

```
GainNode.cs
StateVariableFilterNode.cs
Limiter.cs
StereoTo7Point1Node.cs
```

Killing your darlings

We prototyped a lot more than what ended up in the demo. We had some ideas about reverb zones for the different parts of the level as well as distance-controlled reverb mixes for the near and far sounds that also appeared on some of the slides in the Unite 2018 presentation. While the concept showed a lot of potential, we didn’t have enough time to tweak them and therefore decided to leave it early on, but we are sure it will reappear in some shape in a future production.