Learning - Quartz usage for sound programming

This page describes the research into Unreal's audio engine system called Quartz.

Quartz allows developers to quantize (apply a "grid" to) in-game audio playback and subscribe to relevant events.

- Intro and goal
- Tools and tutorials
- Demo level and setup
- Implementations without Quartz
- Implementations with Quartz
  - Simple intro + loop
  - Gameplay-driven audio
    - 1. Metronome
    - 2. Boss fight
    - 3. Water platforms
  - Audio-driven gameplay
    - Setup
    - Splicing accurate samples in Audacity
    - Quartz implementation
    - Gameplay objects
    - Final result
- Conclusions
  - Difficulty
  - Verdict
  - Misc.

Intro and goal

Quartz is a system within Unreal Engine that allows developers to accurately play audio clips at any exact synchronized timing without having to worry about game-thread latency or hitches. Without using Quartz, attempting to sync audio clips (such as a music intro → looping segment) will result in audible inconsistencies in timing. By using the Quartz system you can play and queue audio samples independent of any potential game latency.

The following use cases will be looked into and attempted during this R&D period:

- Playing audio/music as separate synchronized layers and fading in/out layers based on gameplay.
- Playing a single non-looping audio sample that seamlessly transitions to a looping audio sample afterwards.
- Queuing a next audio sample to start playing at the next most suited moment based on the current playing audio (e.g. every 2s in a 120bpm soundtrack).
- Audio-dependent gameplay. Specifically triggering gameplay actions to the beat of a music track.

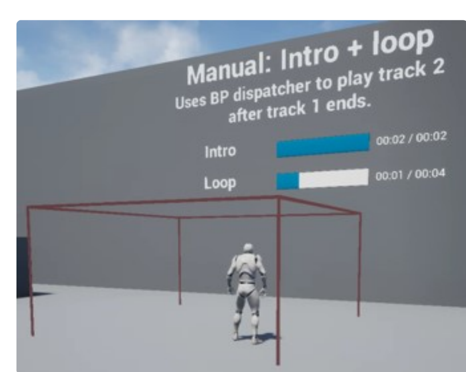
Fore more information about Quartz, see Unreal's Quartz Overview page.

Tools and tutorials

The following tools, in addition to Unreal Engine, are used during this R&D project:

Category	Title	Purpose
Software	Audacity	Editing and exported audio files.
Tutorial	Quartz overview	Official Quartz overview page and terminology reference.
	Rough introduction to Quartz	General entry-level video tutorial before diving into specifics.
Miscellaneous	BPM measure tool	Webpage used for detecting a track's BPM (beats per minute).
	Beat Finder (Audacity plugin)	Plugin for Audacity that can also detect BPM. I've found this to be less reliable than the online measure tool.

Demo level and setup



For quick prototyping and showcasing I've set up an Unreal project with the Third Person BP template. Along a test level are trigger boxes per audio implementation method that each execute logic relevant to that method. When the player leaves the trigger box all audio is stopped. I had previously tried setting this up entirely with widgets but found myself running into issues with requiring AudioComponents that cannot be destroyed from a widget and needing some sort of gameplay element to use as audio input (e.g. the velocity of the third person character).

I've also added a widget that visualizes the audio duration and elapsed time via a progress bar.

Implementations without Quartz

To quickly indicate the audible issues when not using Quartz I've set up the following scenarios in my demo level:

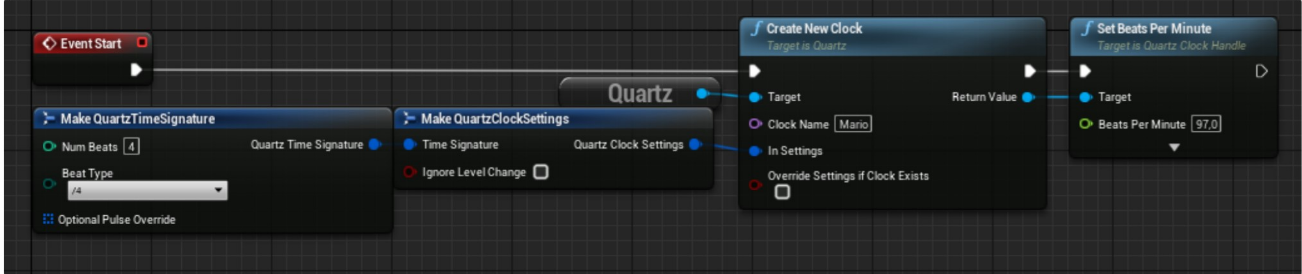
Scenario	Implementation	Observed result	Notes
Play intro, then looping segment.	One SQ for the intro, then a second SQ plays after the first SQ's `OnPlayFinished` dispatcher.	Audible and inconsistent delay between SQ 1 and SQ 2.	
	A single SQ that contains the intro as well as the looping segment on a `Delay` node.	Occasional minor gap in-between sounds but surprisingly hard to notice.	I've set up the SQ to "Prime on load", instructing Unreal to load all of the SQ's wave assets into the cache upon loading this SQ to further minimize the delay in-between sounds.

Implementations with Quartz

Simple intro + loop

After first reading most of the ***Quartz Overview*** page I followed the ***Rough Introduction to Quartz*** video tutorial.

First a **clock** is created using the Quartz subsystem. This will be our main managing object that executes events related to this clock (or "metronome"). After creating the clock I'll need to specify the speed at which it ticks.



This first set-up with Quartz is tested using two audio samples from the Super Mario Bros. main theme; the intro and main loop have been split into separate files. Using the BPM measure tool its BPM was determined to be 97.
There are various other methods you can use to set the clock's speed (milliseconds per tick, ticks per second, 30s notes per minute), but for now I've done this by specifying the beats per minute.

After creating a clock it will be required various times later to be referenced so I'll save it as a variable. Additionally, I've encapsulated the clock creation in a parent class function that also adds the clock handle to a Set, ensuring that when the pawn leaves this triggerbox it can easily iterate through all relevant clocks and set them as invalid.

> Using Unreal Engine 4.26 there doesn't appear to be a node for actually invalidating clocks or stopping them. I can, however, pause clocks and reset their "transport" (which seems to indicate playback position in time).
>
> This page suggests Unreal Engine 4.27 has the option to directly delete a clock by its handle.

After some minor troubleshooting I've found that manually calling `ResumeClock` on the clock handle is required to make it start ticking and fire off time-relevant events.

After creating and starting the clock I instruct the intro audio to start playing immediately and the main loop to start playing after 5 bars.

The result is a trigger box that, upon overlap, first plays the intro to Super Mario Bros. and then starts looping (part of) the main song perfectly timed after the intro. Success!

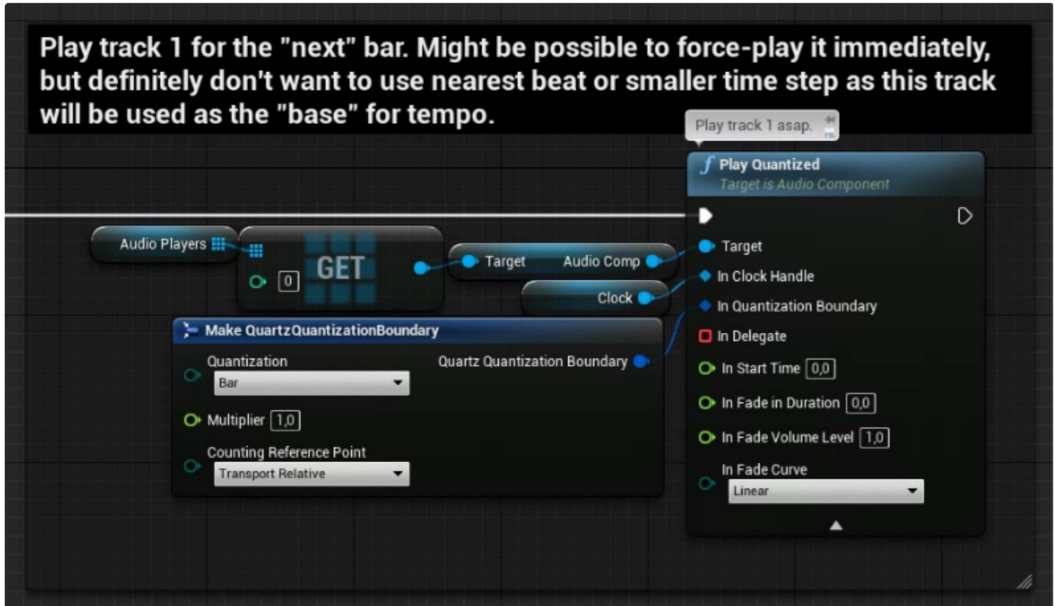
Gameplay-driven audio

The next Quartz example I set-up were three increasingly complex cases of switching from track A to B at the next most convenient time (e.g. at the nearest bar).

1. Metronome

To ensure I don't mess up anything tempo-related I started with another metronome example. Track 1 of the metronome is a 4-bar loop with 4 beats per bar. Track 2 is another 4-bar loop but with 8 beats instead of 4.

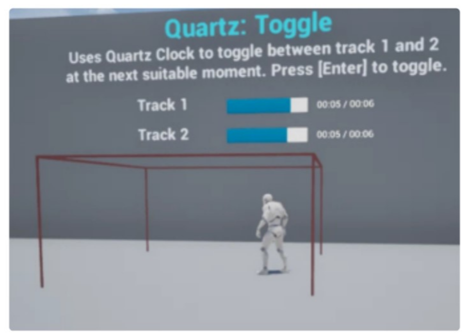
Again I create a Quartz clock and set its BPM (140).

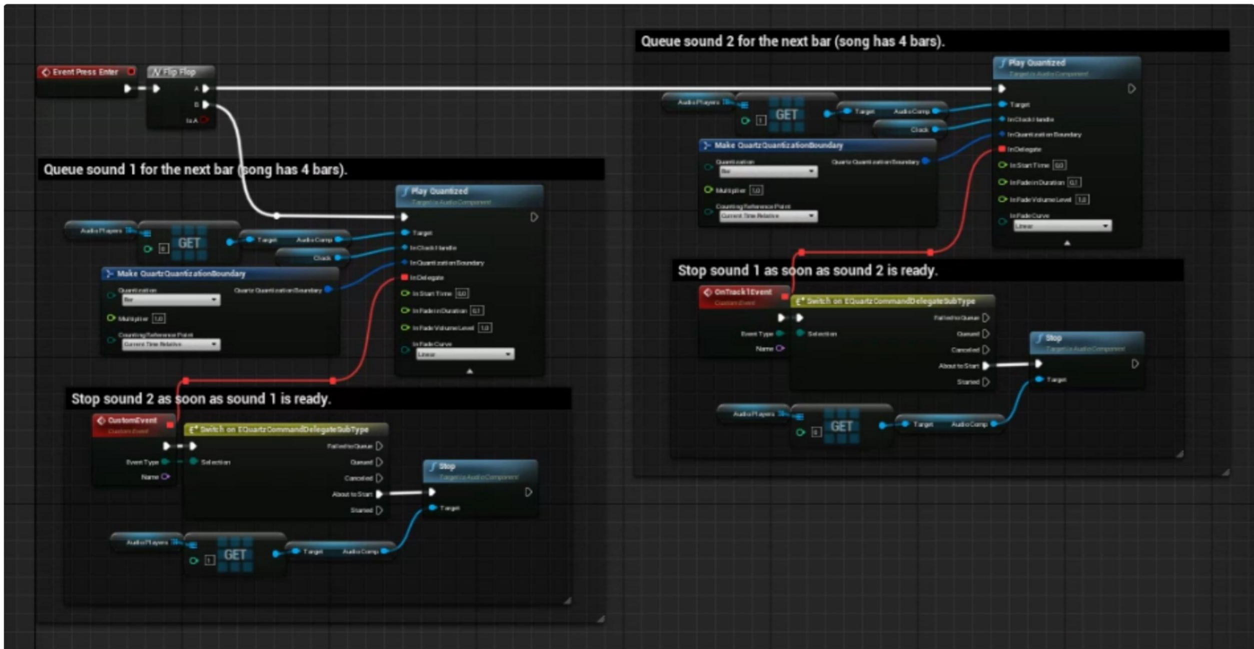


Playing track 1 at the next bar after the player overlaps with the trigger-box.

The first metronome track is played at the next bar after the player activates this trigger-box. You can load it in faster by setting the quantization to a smaller step (e.g. every beat, at 1/32, at 1/16, and so forth) but as this track is used as the "base" that would mean having to adjust the timing of all subsequent tracks to match the base's. I've not yet looked into the possibility of preparing a quantized sound *before* starting a clock to ensure it'll be played immediately.

By pressing the `Enter` key while inside this trigger-box, Quartz will switch from the current track to the alternative metronome track at the next bar. Right when the next track is about to play (of which you can be informed by subscribing to its delegate), the other track is stopped.





Crude setup for switching between two tracks at the next bar.

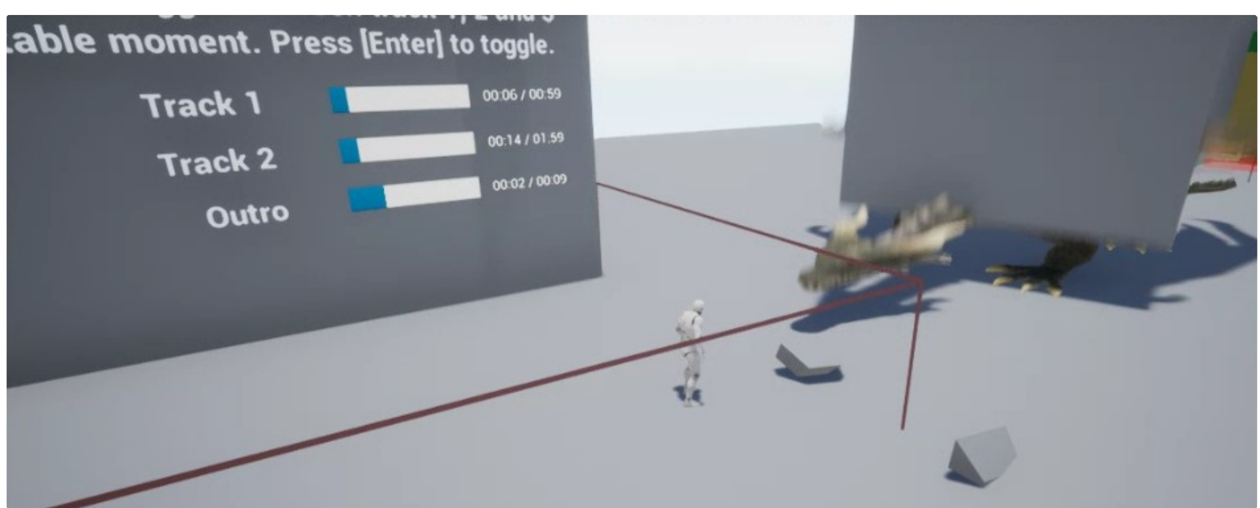
2. Boss fight

The next set-up is an example of how Quartz can be used with minimal effort to synchronize a music track to fit gameplay. For now, the `Enter` key is still used to manually cycle between stages.

The music for this example is [taken from Metroid](). Its BPM took some effort to determine as I wasn't expecting it to be **72.5** (instead trying 73 and 72 and running into timing issues).

1. When the player enters the trigger-box the camera will begin to shake.
   a. Tense music fades in. ([0:06] in the linked video)
2. A large dragon breaks through the wall and faces the player.
   a. Tense music switches to its main loop intended to play during the "boss fight". ([1:05] in the linked video)
3. The dragon is defeated, for now implemented as a large cube that falls onto and squashes it.
   a. Music switches to its simple outro. ([0:00] in the linked video)

After the dragon crashes through the wall and appears on screen, Quartz switches from track 1 to track 2 at the next beat. The song uses a standard 4/4 signature so this transition never waits more than ~1.5s.



As the dragon gets flattened, Quarts switches from track 2 to track 3 (outro) at the next beat.

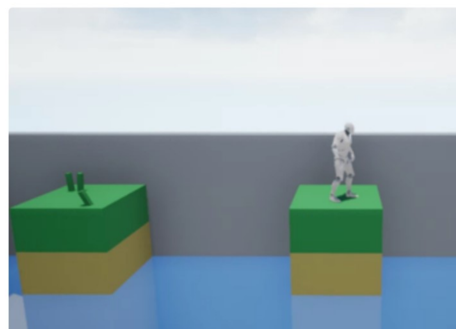
The Blueprint implementation for this example is virtually identical to that of the metronome example, albeit with an additional track to cycle between and shorter quantization (next beat instead of next bar).

3. Water platforms

The final example of gameplay-driven audio is a simple collection of platforms above water. Instead of cycling manually, the music in this example responds to the proper corresponding gameplay conditions.
Music used in this example is taken from Rayman 3 and has a BPM of 160.

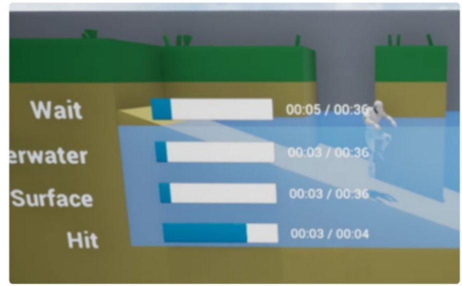
1. As the player enters the area. a simple music loop plays consisting primarily of drums.
   a. Starts at [3:00] in the video.

2. If the player falls into the water at any point, the music switches to a calmer track that contains bubble sounds and a single reverse cymbal hit plays.
   a. Loop starts at [0:00] in the video.
3. When the player exits the water to retry the platforming, the calmer track continues playing but drums fade in.
   a. Loop starts at [0:36] in the video.
4. Step 2 and 3 can loop indefinitely. Above the water drums can be heard, while underwater they cannot.



As in all Quartz use-cases, the clock is started at the correct BPM (160) as the player enters this area's main trigger-box.

The simple drum loop continues playing as the player jumps over the platforms and the player can complete the platforming segment without ever falling into the water and thus without ever changing the music.



The music transition from the drums to the calm underwater loop without drums is admittedly a little jarring, but the accompanying cymbal hit helps.

As visible in the screenshot to the right, both the underwater and the surface loops start playing at the same time. However, the surface loop is played at 0% volume. This is to ensure that the next transition from underwater → surface can seamlessly blend between the two tracks at the correct point in time for both tracks (which have an identical rhythm and duration). Note that this is currently implemented as two different audio components, but SoundCue has a `"Cross fade by param"` node that should work for this as well. The reverse cymbal hit plays at the next bar and contains a delay inside the .wav asset itself to let it sync with the 3rd beat in that bar.

Fading between the underwater/surface loops is done without Quartz (although Quartz was used to ensure both started at the same time), instead simply fading their volume from 100% to 0% (and setting up the SoundCue to continue playing even while silent).

Audio-driven gameplay

Setup

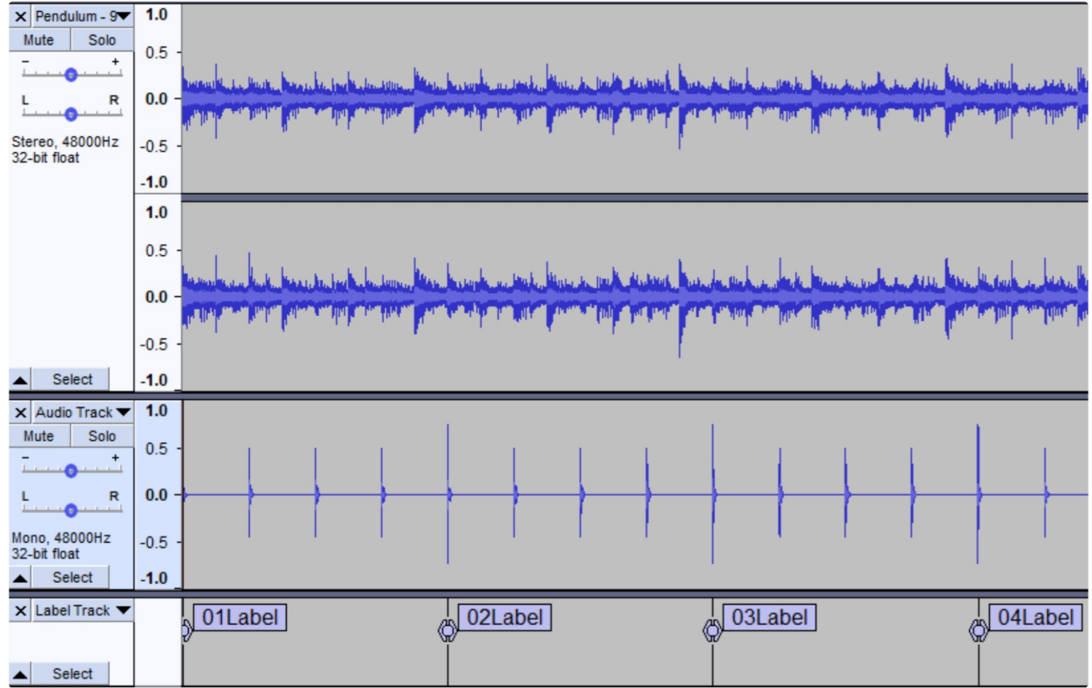
Now it's time for the opposite, which is having gameplay based on the timing of audio tracks. I've set up a separate level for this with the goal of creating a platforming segment where many different gameplay elements behave according to the beat of the background music.

The background music that plays is 9000 Miles by Pendulum. As with the gameplay-drived audio chapters, I've split up the song in different segments to play according to the stage of the parkour level.

Splicing accurate samples in Audacity

The only way to make this song work was to split it up into different segments that were 100% accurately split in order to loop and transition seamlessly. The following actions were done in Audacity:

1. Find out the song's BPM. It's either 87 or 174 (double), likely the latter. Which of the two exactly doesn't really matter.
2. Remove the initial silence that exists in most .mp3 files.
3. Add a rhythm track (basically an audible metronome) with the determined BPM. Play the song for a while and confirm the metronome is 100% accurate for this song.
4. Run the "Regular interval labels" tool to place a label every X seconds. We know the BPM, so 60 / BPM = the interval in seconds per beat. I've multiplied the value by 4 (=2.75862068966s) as the song has a standard 4/4 signature, so it only places a label every bar (every 4 beats).
5. With the labels created, selection of audio now snaps to these labels. This makes it very easy to select a segment of the song and export only that selected audio as a seamless looping segment.



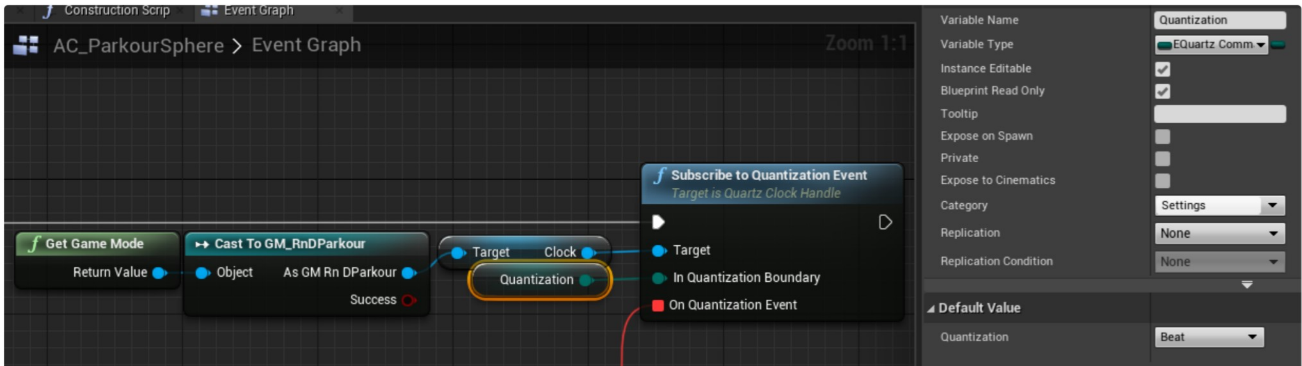
Alternatively it's also possible to just calculate the BPM back to samples. Samples are displayed in Audacity per project at the bottom left (e.g. 48000Hz); this is the amount of samples per second. So using 60 / BPM * sample rate = 33103.4482759 samples per beat in this song. Multiply that by 4 and you have the amount of samples per bar. So if we want to split the song at the start of any bar, the time (in samples) at which the song is split must be a multiplication of ~132414 (as that's the amount of samples per bar).

Quartz implementation

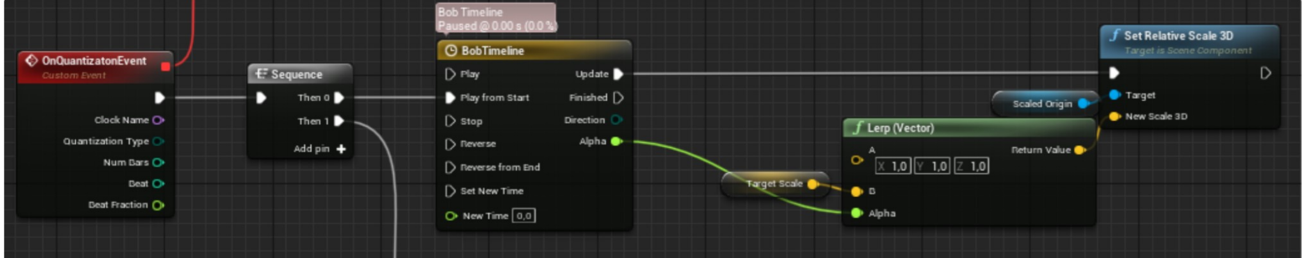
At `BeginPlay` the GameMode will create a clock with a 4/4 signature and a 87 BPM. It starts playing quantized background music via the same implementation method as the previous implementations.

> ℹ While Quartz clocks are not replicated, the GameState would be a better location to store and manage clocks that need to be used by many different other actors.

Other actors can now request a reference to this clock and subscribe to whatever quantization event (metronome) is applicable.

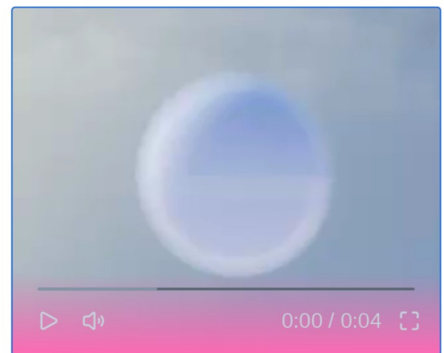


This actor has its quantization boundary exposed as a setting. Most instances of this actor will trigger every beat (1/4), but others are set to trigger twice as often (1/8).



Every beat, the above actor will play a timeline in which it visually quickly grows and shrinks as if bouncing to the beat.

Final result:

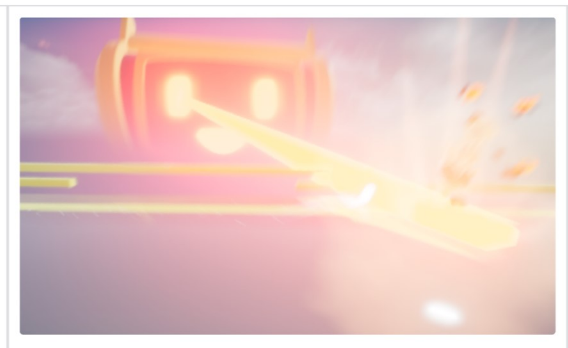


Gameplay objects

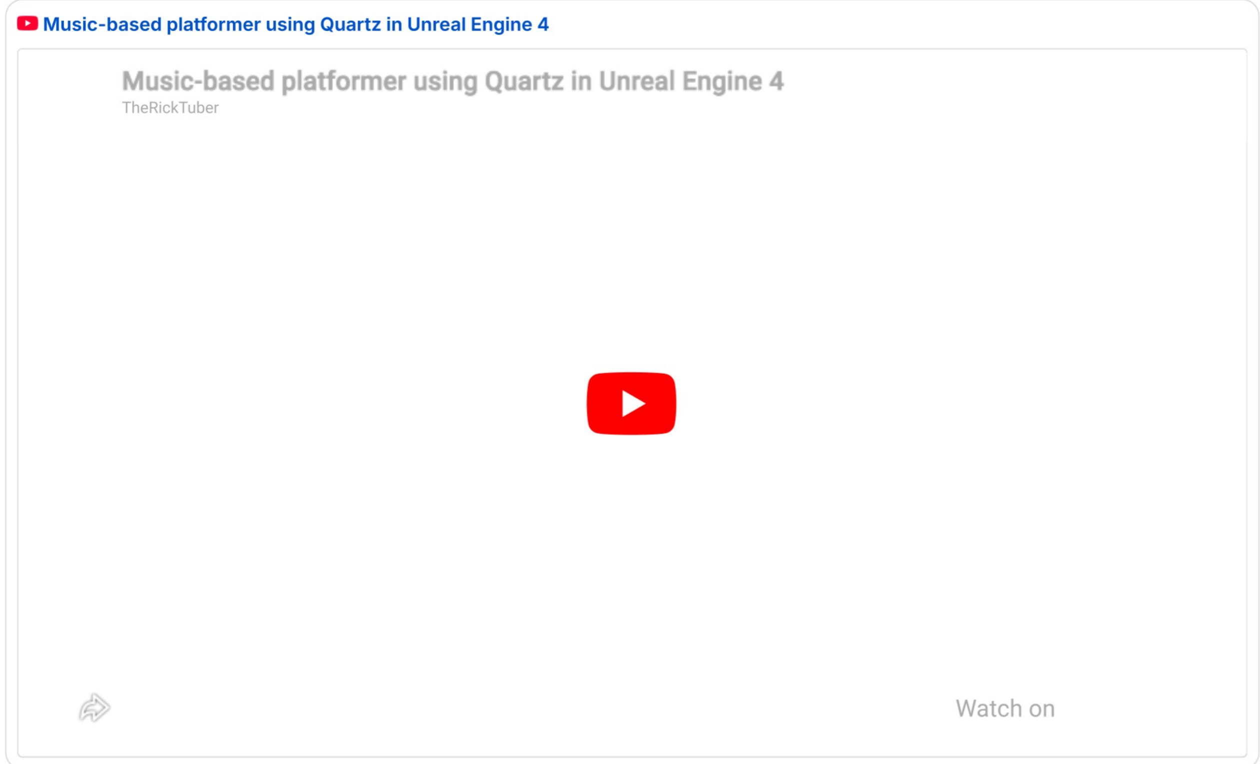
Using the same implementation with different quantization times, the following gameplay elements were added to the level:

Element	Description	Image
Flashing lights	Using the GM to control MPC values, most emissive materials in the level will flash to the beat of the music. The flashing becomes increasingly strong (1/4 → 1/8, add	

	screen disco effects) as the player progresses through the level.	
Player speed	If the player presses `Shift` in sync with the beat of the music, they will flash green and gain a speed boost.	
Swinging scythes	Scythes swing between various angles to the tempo of the music.	
Lightning	The bouncing sphere actors zap lightning between each other at every beat.	
Gates	Gates and other obstacles move from location A to location B between every beat.	

BVEE	A giant BVEE will fire lasers at the player every bar near the end of the level, because of course he does.	

Final result



Conclusions

Difficulty

Despite being a complete beginner when it comes to music terminology, figuring out how to use Quartz was surprisingly easy. It helps that a time signature of 4/4 is applicable 90% of the time and the different quantization values (every bar, every 1/4th, 1/8th, etc) can just be interpreted as increasingly smaller steps. With the exception of "beat", which depends on the time signature and can be overridden when setting up the clock; so for a standard 4/4 music track using the 1/4th quantization would be the same as using beat.

However I did already have some experience with Audacity and editing tracks to make them loop seamlessly and detecting a track's tempo, which is absolutely essential when working with Quartz. Obvious tip is obvious, but make sure you take the time to properly detect and confirm a song's BPM before using it in-game as an incorrect BPM will obviously break everything. I'd go as far as to say that when something *does* sound off, always start by verifying the tempo as it's likely the cause.

Verdict

Quartz is surprisingly easy to use in Unreal. Most tracks are simple enough to take no more than 15 minutes to split into various segments to transition between or play/stop depending on gameplay. Similarly, having audio-dependent gameplay is also extremely easy to set up and I've not noticed any notable performance issues in my parkour/platforming example.

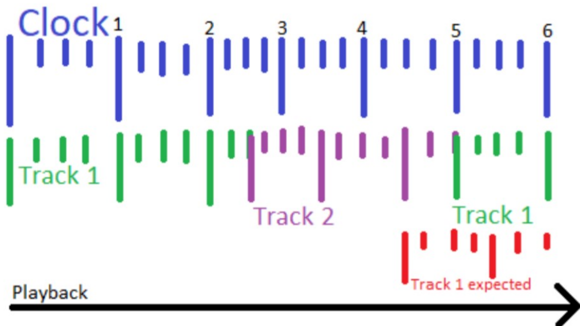
Generally it doesn't take a lot of time per music track to implement basic transitions via Quartz. If an ENVERSED entertainment project were to require tracks delivered by a professional composer, I would absolutely recommend going the extra mile with Quartz and implement these tracks as best we can.

Misc.

- Quartz uses standard audio components with standard SoundCue and SoundWave assets. You don't need special components or files and - even though I've created different SoundCue assets per implementation method for the sake of keeping relevant assets contained in different folders per method - you can use the same audio assets for implementations with and without Quartz.
    - However it is impossible to use only one single audio component if you want to queue a transition between audio tracks via Quartz. `Play quantized` only queues audio

component playback for whichever sound it is currently configured to. And even if you were to use only one audio component and try to simply quickly change it to a different sound to play right when the event triggers (ignoring the fact that this would cause loading issues), calling `Play quantized` on an audio component that's currently playing will cause it to stop playing until the quantized event triggers.

- Having access to a music track's properties such as tempo and time signature would speed up development but is not required. It's pretty easy to determine these properties and split tracks up into seamless looping segments via Audacity or similar audio editing tools before implementing the audio in Unreal.
  - Setting up some temporary debugging setup like printing a string or playing a sound at every quantization event will make it even easier to confirm the track's tempo and time signature in Unreal.
  - Time signature can be particularly hard to gauge but again, should just be a matter of generating different rhythm tracks in Audacity and just listening to check what works best.
- The shorter/faster the quantization at which you queue the next audio, the more you start to deviate from the Quartz clock which can lead to some surprising results if not keeping this in mind:
  - Track 1 starts at the same time as the Quartz clock. Standard 4/4 time signature.
  - Track 2 starts at the next beat. Let's say this happens to occur halfway through a bar, so at 2/4.
  - You now want to switch back to track 1 at the next bar of track 2, so you use "next bar" as quantization. This transition will however trigger halfway through track 2's bar as track 2 had a 2/4 offset from the clock metronome due to starting track 2 at 2/4. The bars of the Quartz clock do not match the bars of track 2.



  - This could be resolved by either saving the resulting offset somewhere and add it to step 3's queue time or resetting the clock's transport (playback progress) at the same time as starting track 2, although since the transport reset node seems to queue itself for the next

bar this may not be an easy solution. I've not looked into a solution for this problem during R&D.

- It can be tricky to judge a song's BPM regarding whether it should be half/double or not, e.g. whether it's 80 or 160. This doesn't really matter for implementation in Quartz.
- The audio progress bar widgets I made during demo level setup occasionally break and will be stuck at 0% for the rest of the level. This doesn't affect audio playback.
  - This appears to happen when a quantized event is queued and the clock is then paused before this event can trigger, then later reset and played again. At this point the audio component's "playback percent" dispatcher will always return 0.
  - This issue can probably be avoided entirely in 4.27 when clocks can be properly deleted and a new one can simply be created.
- Unrelated to Quartz, but the audio component's "playback percent" dispatcher will also continue past 1.0 "percent" for looping audio. For the progress bar widgets I've simply subtracted the whole integer from the playback percent float, keeping only the decimal.
- Quartz is fun.
  - It helps that Quartz is one of the few recent Unreal subsystems to have a properly documented overview page.