ChatGPT

Zusammenfassung

Die Analyse zeigt, dass das Programm derzeit stark von externen Audio-Tools (pydub/simpleaudio, winsound, ffplay etc.) abhängt und in der Playback-Kette mehrere **blockierende Aufrufe** enthält. So versucht die `AudioEngine` im aktuellen Repository zuerst `pydub.playback.play()` zu nutzen, fällt dann unter Windows auf `winsound.PlaySound()` zurück und schließlich auf Aufrufe von `ffplay` oder gar `os.startfile()` plus `time.sleep()` [1] [2]. Diese Strategien sind ineffizient und unzuverlässig: Fehlen etwa `pydub`, ein Audo-Backend oder ffmpeg, geht schlicht der Ton verloren, während ggf. die GUI einfriert (etwa wenn `time.sleep()` im Main-Thread verwendet würde [3]). Zudem arbeitet das `AudioEngine.play()` strikt blockierend, d. h. es hält den abspielenden Worker-Thread vollständig auf [1] [2]. Die Folge ist, dass die Wiedergabe nicht wie erwartet erfolgt und der Nutzer keinerlei Sound hört, während das Programm scheinbar hängen bleibt. Dies gilt sowohl für Wort-für-Wort-Wiedergabe als auch für einzelne Wörter: Beide Varianten nutzen einen `QThread` mit entsprechenden Worker-Objekten (`_WordByWordWorker` bzw. `_AudioWorker`) [4] [5]. Wenn die Engine dann blockiert, kann sie weder Termine (Timing-Map) noch Word-Highlights zuverlässig liefern. Insgesamt bewerten wir die Situation als **kritisch**: Die Audio-Wiedergabe ist Kernfunktion und derzeit unbrauchbar.

Detaillierte Analyse der Audiokomponenten und Threads

Wir betrachten alle relevanten Module:

- `taamimflow/audio/audio_engine.py`: Hier steht die Hauptklasse `AudioEngine`. Sie kann aus Noten (`Note`-Objekten) Audiodaten erzeugen und abspielen. Der code nutzt `pydub` (falls vorhanden) zur Erzeugung von Sinus-Tönen [6] [7] und implementiert in `play()` (Zeile 172ff) eine Kette verschiedener Player-Fallbacks [1] [2]. Wichtig für unser Problem sind insbesondere:
- `AudioEngine.synthesise()` erzeugt ein `pydub.AudioSegment`. Falls `HAVE_PYDUB=False`, liefert `None` [8].
- `AudioEngine.play(segment)` probiert nacheinander:

  1. `pydub.playback.play()` (benötigt simpleaudio/pyaudio) [9],
  2. `winsound.PlaySound()` auf Windows [10],
  3. `subprocess.run(["ffplay", ...])` [2],
  4. `os.startfile()` oder `xdg-open` plus ein `time.sleep()` zur „Wort-Dauer“ [11] [12].

  Dabei sind viele Aufrufe **blockierend** (z.B. `subprocess.run(check=True, timeout=30)` wartet stur, `time.sleep(len(segment)/1000.0)` blockiert den Thread komplett [13] [12]). Fehlschläge wie nicht gefundene Programme werden im Code gefangen, aber stumm verworfen („pass“) [14] [15], sodass im Fehlerfall schlicht nichts passiert.

- `taamimflow/core/concat_audio.py`: Implementiert `ConcatAudioEngine`, das statt synthetischer Töne voraufgezeichnete Audiodateien nutzt. `synthesise()` kombiniert einzelne `AudioSegment` zu einer Audiodatei, `play()` ruft ebenfalls intern

`pydub.playback.play()` auf (blockierend) [16] [17]. Fehlt `pydub`, so gibt es hier gar kein Playback [18]. Dieses Modul ist ähnlich anfällig wie `AudioEngine`: externer Abhängigkeitsbedarf und potenziell blockierendes Verhalten (auch wenn im Code-Abschnitt des `play()` nur ein einfacher Aufruf von `pydub.playback.play()` erfolgt [17], ist damit die gleiche Problematik verbunden).

- `taamimflow/utils/audio.py` (historisch/Legacy): Eine Platzhalter-Klasse `AudioEngine`, die bei `play_notes()` per `time.sleep()` „Wiedergabe" simuliert [19] [20]. Dieser Code wird *wohl nicht* aktiv vom GUI genutzt (das MainWindow importiert den Core-Audiocode, nicht dieses utils-Modul), aber er zeigt exemplarisch, dass ein `time.sleep()` jeden Thread blockiert [20]. Generell gilt (allgemein): `time.sleep()` **friert den Event-Loop ein** [3], wenn er im GUI-Thread stünde.

- `taamimflow/audio/utils.py`: Helferfunktionen für pydub (Volumen, Laden von Dateien) [21] [22]. Sie selbst spielen nur mit Daten, führen kein Playback durch und enthalten kein `sleep` oder `subprocess`, sind hier nicht primär betroffen.

- `taamimflow/audio/timing_map.py`: Erzeugt Start/End-Zeiten aus Notenlisten [23] [24]. Relevant zur Synchronisation (Karaoke-Highlight), aber nicht direkt am Ton-Abspiel-Code beteiligt.

- `taamimflow/audio/tradition_profiles.py`: Definiert Pfade zu Audio-Sample-Verzeichnissen (für `ConcatAudioEngine`) [25]. Kein Ausführungscode, dient nur Konfiguration.

- **GUI (`taamimflow/gui/main_window.py`)**: Bindeglied zwischen Text und Audio. Wichtige Punkte:

- `MainWindow._play_current()`: Startet eine *wortweise* Wiedergabe: Es holt die Noten aller Tokens über `_get_notes_for_token()` [26], setzt Playback-Thread auf und startet `_WordByWordWorker.run()` in einem `QThread` [4].
- `MainWindow._play_word_audio()`: Bei Klick auf ein Wort wird entweder das laufende Playback neu gestartet oder nur dieses Wort mit `_AudioWorker` abgespielt [27] [28]. Auch hier wird ein neuer `QThread` mit `_AudioWorker.run()` gestartet.

- Beide Worker-Klassen (`_WordByWordWorker` und `_AudioWorker`) leben in `MainWindow` [29] [30]. Sie rufen im Thread `engine.synthesise(...)` und `engine.play(...)` auf. Ein Beispiel aus `_AudioWorker.run()`: `seg = self._engine.synthesise(...)` gefolgt von `self._engine.play(seg)` [31]. *Diese* Aufrufe laufen tatsächlich in einem **zweiten Thread**, so dass der GUI-Thread prinzipiell nicht einfriert. Allerdings: Solange `engine.play()` blockiert, bleibt der Worker-Thread inaktiv und kann keine weiteren Wörter verarbeiten oder Signale senden.

- **Thread-Graphik** (Mermaid):

```
graph LR
    User -- klick/wort_start --> MainWindow
    MainWindow -- {"_audio_thread.start()"} --> _AudioWorker in QThread
    _AudioWorker --> AudioEngine.synthesise()
    _AudioWorker --> AudioEngine.play()
```

```
    AudioEngine -- (sinus/segment) --> AudioDevice(Output)
    MainWindow -- signals --> TextWidget (Hervorhebung)
    MainWindow --> MainWindow (updates GUI)
```

Blockierende Aufrufe, Deadlocks und Ausnahmebehandlung

Blockierende Methoden:
- In `audio_engine.AudioEngine.play()` gibt es mehrere blockierende Aufrufe: `simpleaudio` / `pyaudio`-Playback, `winsound.PlaySound()`, `subprocess.run([... 'ffplay' ...], timeout=30)` und ggf. `time.sleep(...)` [1] [2]. Diese laufen *im Worker-Thread*, d.h. sie blockieren diesen Thread komplett. Der GUI-Thread bleibt zwar prinzipiell frei, aber solange der Worker hangelt, können z.B. Highlight-Signale nicht (rechtzeitig) gesendet werden.

  - In `utils/audio.py` (wenn verwendet) blockiert `time.sleep()` den GUI-Thread komplett, was hier einem *"kompletten Einfrieren"* entspricht [20] [3]. In MainWindow wird zwar kein `time.sleep()` direkt im GUI-Thread ausgeführt, doch der Splashscreen oder andere Routinen könnten darunter leiden, wenn z.B. das Worker-Thread-Handling falsch genutzt wird.

  - `QThread.quit()`: In `_stop_playback()` ruft man `thread.quit()` ohne `wait()` auf [32], genau um die GUI nicht einfrieren zu lassen. Ein **fehlerhaftes** `thread.wait()` im Haupt-Thread würde diesen blockieren, bis der Hintergrund-Thread stoppt [33]. Das Team hat das bewusst vermieden, was korrekt ist.

Deadlocks und Race-Conditions:
Es gibt keine offensichtlichen Locks im Code. Ein Deadlock wäre etwa denkbar, wenn der Worker auf ein Signal aus der GUI warten würde, das von `thread.quit()` nie gesendet wird – das ist hier aber nicht der Fall. Die Kommunikation erfolgt über *Signals und Slots* ohne gegenseitiges Warten (z.B. `_stop_playback()` signalisiert `cancel()`, worauf der Worker im nächsten Loop-Abbruch respektiert). Daher sind klassische Deadlocks unwahrscheinlich.

Ausnahmen:
Alle potentiell fehlerhaften Bereiche sind mit `try/except` umgeben. Z.B. fängt `_AudioWorker.run()` Fehler und sendet sie per `error`-Signal [29]. `AudioEngine.play()` fängt selbst alle internen Fehler stumm ab und gibt einfach keinen Ton aus [14] [15]. Dadurch passiert im Fehlerfall oft "gar nichts". Es findet nur wenig Logging statt (Einschübe in `logger.debug`), im produktiven Ablauf aber keine Nutzerwarnung.

Nutzung der Tropendefinitionsdateien

Die XML-Dateien `tropedef.xml`, `tropedef_megillot.xml` und `tropenames.xml` haben folgende Rollen:

- `tropedef.xml` enthält für jede Tropen-Bezeichnung (Hebräischer Akzent) die zugehörigen Noten (Pitch und Dauer) unter verschiedenen Kontextbedingungen. Das Modul `taamimflow/data/tropedef.py` parst diese Datei und erzeugt aus jedem `<TROPE>`-Abschnitt ein

`TropeDefinition`-Objekt mit einer Liste von `TropeContext` (die selbst Listen von `Note`-Datensätzen beinhalten) [34] [35]. In der Praxis wird `tropedef.xml` in `core/cantillation.py` verwendet: Die Funktion `extract_tokens_with_notes(text, xml_path, style)` lädt die XML (über eine Hilfsfunktion `_find_tropedef_xml()`) und annotiert die tokenisierte Lesung mit `notes`-Listen [36] [37]. Das heißt: Jeder `TokenFull` erhält auf Token-Basis tatsächliche Notenfolgen. Diese werden dann vom Audio-Engine-Code abgespielt oder zu Samples gemappt. Ist `tropedef.xml` nicht auffindbar oder fehlerhaft, bleiben die Tokens leer (es gibt nur Fallback-Noten nach Tropen-Rang, siehe unten [26]).

- `tropedef_megillot.xml` enthält die Zuordnung von Megillah-Typen (Esther, Ruth, Kohelet etc.) zu unterschiedlichen **Melodie-Varianten** (z.B. verschiedene Ashkenazi-Stile für Esther). Dieser Inhalt wird *nicht* für die Audio-Engine selbst verwendet, sondern für die GUI-Auswahl: Das Dialogmodul `open_reading_dialog.py` liest die Datei per Regex aus (wegen kleiner Formatfehler) und füllt ein Dictionary `_MEGILLOT_MELODIES` mit `{Megilla-Typ: [Stilnamen,...]}` [38] [39]. Dadurch kann die Benutzerin bei Feiertagsauswahl zwischen z. B. "Ashkenazic – Binder", "Jacobson" etc. wählen. Für die Audio-Wiedergabe selbst entscheidet dann wiederum `ConcatAudioEngine`, welche Sample-Map benutzt wird – oder bei der Sinus-Engine wird möglicherweise ein Stil-Parameter übergeben. Kurz: **Grafik**: Megillot-XML → GUI-Optionen (Name der Variante) → Auswahl → Übergabe an Engine; selbst aber nicht direkt am Audiocode beteiligt.

- `tropenames.xml` dient nur zur Anzeige: Es mappt interne Hebräische Tropen-Kürzel auf lesbare Namen in verschiedenen Traditionen. Der Parser `taamimflow/data/tropenames.py` liest das ein und liefert eine Liste von `Tradition`-Objekten, deren `names`-Dictionary z.B. `'ETNACHTA' → 'Etnachta'` enthält [40] [41]. Diese Namen werden vermutlich in der UI angezeigt (z.B. beim Hover oder im Notations-Panel), aber sie beeinflussen nicht die Tonerzeugung. Die Audio-Engine arbeitet ausschließlich mit Tonhöhen, nicht mit Namenslabels.

Konkrete Code-Änderungen zur nicht-blockierenden Wiedergabe

Um den Ton **stabil und ohne externe Abhängigkeiten** abzuspielen, empfiehlt sich ein Ansatz über Qt Multimedia: Wir ersetzen das blockierende `AudioEngine` durch eine, die mit `QAudioOutput` direkt PCM-Daten abspielt. Dies vermeidet externe Player und heavy Subprocess-Aufrufe. Die Idee:

- In `AudioEngine` verwenden wir weiterhin eine Sinus-Erzeugung für Noten (wie bisher per Frequenz und Dauer). Statt über `pydub` einen `AudioSegment` zu erstellen, rechnen wir die Samples selbst aus (z. B. 16-bit PCM, Mono, 44100 Hz).
- Anschließend erzeugen wir ein `QByteArray` oder `QBuffer` mit den Rohbytes und übergeben das an `QAudioOutput.start()`. Dadurch übernimmt Qt selbst das Abspielen (im besten Fall asynchron).
- Wir eliminieren *alle* Fallbacks: Keine winsound, kein ffplay, kein time.sleep mehr.
- `synthesise()` könnte optional weiterhin ein Segment zurückgeben (für Dateiausgabe), aber das ist sekundär.
- Die Benutzer-Flows (`_AudioWorker` und `_WordByWordWorker`) bleiben ähnlich: Sie rufen nun `engine.play()` auf, welches den Qt-Output startet, gibt aber sofort zurück. Wenn nötig, kann man das Ende abwarten (z.B. via Signalen oder einer kurzen Sleep im Worker, abhängig von Design). Wichtig ist, dass der Worker-Thread *nicht* in `play()` blockiert.

Unten zeigen wir exemplarisch die voll modifizierte Datei `taamimflow/audio/audio_engine.py`. Dort wurde die Playback-Logik durch Qt ersetzt. Ähnlich könnte man (wenn ConcatEngine weiterverwendet wird) auch `ConcatAudioEngine.play()` anpassen, sodass es wie `AudioEngine.play()` nur Qt nutzt.

```
# taamimflow/audio/audio_engine.py
from __future__ import annotations
import math
from typing import Iterable, Optional, Union

from PyQt6.QtMultimedia import QAudioFormat, QAudioOutput, QAudioDevice
from PyQt6.QtCore import QBuffer, QIODevice, QByteArray

@dataclass
class Note:
    pitch: Union[str, int]
    duration: float  # in Viertelnoten, 1.0 = Viertelnote
    upbeat: bool = False

class AudioEngine:
    """Neue Audio-Engine ohne externe Player, mit Qt-AudioOutput."""
    def __init__(self) -> None:
        # Standard-Audioformat: 16 bit, Mono, 44100 Hz
        fmt = QAudioFormat()
        fmt.setChannelCount(1)
        fmt.setSampleRate(44100)
        fmt.setSampleSize(16)
        fmt.setSampleType(QAudioFormat.SampleType.SignedInt)
        fmt.setCodec("audio/pcm")
        device = QAudioDevice.defaultAudioOutput()
        self._audio_out = QAudioOutput(device, fmt)
        self._buffer = None  # QBuffer zum Playback

    def pitch_to_frequency(self, pitch: Union[str, int]) -> float:
        # unverändert aus Original:
        if isinstance(pitch, int) or (isinstance(pitch, str) and
pitch.isdigit()):
            midi = int(pitch)
            return 440.0 * (2.0 ** ((midi - 69) / 12.0))
        if isinstance(pitch, str):
            p = pitch.strip().upper()
            note_map = {'C':0,'C#':1,'DB':1,'D':2,'D#':3,'EB':3,'E':4,'F':
5,'F#':6,
                        'GB':6,'G':7,'G#':8,'AB':8,'A':9,'A#':10,'BB':10,'B':
11}
            idx = 0
            while idx < len(p) and not p[idx].isdigit():
                idx += 1
            note = p[:idx]
            octave = int(p[idx:]) if idx < len(p) else 4
```

```
            semi = note_map.get(note, 0)
            midi = 12*(octave+1) + semi
            return 440.0 * (2.0 ** ((midi - 69) / 12.0))
        return 440.0

    def synthesise(self, notes: Iterable[Note], tempo: float = 120.0,
                   volume: float = 1.0) -> Optional[QByteArray]:
        """Erzeuge rohes PCM-Audio als QByteArray."""
        beat_sec = 60.0/tempo
        sample_rate = 44100
        samples = bytearray()
        max_amp = 32767 * volume
        for note in notes:
            freq = self.pitch_to_frequency(note.pitch)
            dur_sec = note.duration * beat_sec
            n_samples = max(1, int(dur_sec * sample_rate))
            for i in range(n_samples):
                t = i / sample_rate
                value = int(max_amp * math.sin(2 * math.pi * freq * t))
                # 16-bit PCM Little Endian
                samples += value.to_bytes(2, byteorder='little', signed=True)
        return QByteArray(samples)

    def play(self, data: Optional[QByteArray]) -> None:
        """Spiele die gegebenen PCM-Daten über Qt ab (nicht-blockierend)."""
        if data is None or data.isEmpty():
            return
        # QBuffer zur Datenquelle:
        self._buffer = QBuffer()
        self._buffer.setData(data)
        self._buffer.open(QIODevice.OpenModeFlag.ReadOnly)
        self._audio_out.start(self._buffer)
```

Dabei sei noch erwähnt: In `MainWindow._get_audio_engine()` wird `AudioEngine()` wie bisher zurückgegeben. Die Worker nutzen also wieder `engine.synthesise()` und `engine.play()`, diesmal aber ohne externe Tools.

Tests/Reproduktion: Nach diesen Änderungen sollte man überprüfen: Klickt man im Text auf ein Wort, so startet im Hintergrund-Thread die Erzeugung von Samples und sofort das Abspielen via Qt. Dabei signalisiert `QAudioOutput` (intern) das Ende, ggf. könnte man ein eigenes `finished`-Signal verwenden. Wichtig ist, dass keine `sleep`-Aufrufe oder Subprozesse mehr involviert sind. Falls nötig, kann man den Worker nach maximal ~`sum(dauer)` Sekunden automatisch stoppen oder per `finished`-Signal.

Migrations-Checkliste und Risiken

- **PyQt6/QtMultimedia:** Das System benötigt nun Zugriff auf Qt Multimedia (in PyQt6 meist vorhanden). Fehlt das Modul, muss es installiert werden (`PyQt6.QtMultimedia`).

- **Sample-Rate/Format:** Wir haben 16-bit Mono/44100 Hz als Standard gesetzt. Diese Werte liegen sicher in allen modernen Systemen, können aber je nach Gerät anders klingen. (Risiko: System ohne Standard-Audioausgang? i.d.R. nicht bei normalen Desktop-OS.)
- **Kompatibilität:** Die Schnittstellen von `AudioEngine.synthesise` und `.play` bleiben gleich (wir liefern jetzt `QByteArray` statt `AudioSegment`, ggf. müssen wir bei ConcatAudioEngine-Klasse analog anpassen). MainWindow selbst braucht keine Änderungen, abgesehen vom `import` der neuen Engine.
- **Testing:** Wir brauchen Tests, ob Audio-Thread korrekt stoppt bei `_stop_playback()`. Da wir `QAudioOutput` verwenden, könnten wir nach `quit()` im Thread auch `QAudioOutput.stop()` aufrufen (falls wir einen Abbruch implementieren).
- **Fehlendes pydub:** Wir können `pydub` nun komplett entfernen oder auf optional setzen. Im Code oben haben wir `pydub`-Importe bereits entfernt. Andere Teile (ConcatEngine, utils) sollten auf `pydub` verzichten oder `HAVE_PYDUB=False` setzen, da wir ihn nicht mehr nutzen.

Vergleich der Wiedergabemethoden

Wiedergabe-Backend	Vorteile	Nachteile
Aktuelle (pydub & Fallback)	- Relativ einfache Synthese per Sinus (pydub) - Fallbacks funktionieren *irgendwie* auch ohne PyAudio (z.B. winsound) [10]	- Benötigt externe Abhängigkeiten (pydub, ffmpeg, SimpleAudio) - Zahlreiche blockierende Aufrufe (subprocess, time.sleep) [13] - GUI reagiert ggf. später (Worker ist blockiert) - Kein Sound wenn ein Schritt scheitert
QtMultimedia (`QAudioOutput`)	- Kein externes Tool nötig (nur QtStandard) - Nicht-blockierend: Qt übernimmt Playback asynchron - Direkt im Prozess (stabil, kein Crash durch fehlende exe) - Geringerer Code-Overhead (weniger try/except)	- Etwas Code-Aufwand: Synthetisierung und Audiokonfiguration - Abhängigkeit auf PyQt6.QtMultimedia (in App-Bundle berücksichtigen) - Keine einfachen Fallbacks, falls Ton nicht ausgegeben wird (aber für Desktop-Apps eigentlich unnötig)

Merkmals- und Ablaufdiagramme

Thread- und Signalfluss: (vereinfacht)

```
sequenceDiagram
  participant UI as GUI-Haupt-Thread
  participant Worker as Audio-Worker-Thread
  UI->>MainWindow: _play_current() aufrufen
  MainWindow->>Worker: QThread.start()
  Worker->>AudioEngine: synthesise(notes)
  Worker->>AudioEngine: play(pcmData)
  AudioEngine->>AudioDevice: QAudioOutput spielt ab
  Worker->>Worker: wartet (Callback oder Zeitmessung)
```

```
  Worker-->>MainWindow: finished-Signal
  MainWindow->>UI: _on_playback_finished() (Status aktualisieren)
```

Audio-Datenfluss:

```
flowchart LR
  Noten --> AudioEngine.Synthese --> PCM-Daten --> QBuffer --> QAudioOutput
--> Lautsprecher
  Token mit Tropen --> extract_tokens_with_notes --> Token.notes (Liste von
Note) --> ^
```

Zeit-Ablauf (Synchronisation): Angenommen, jede *Token* hat Dauer `t_i` Sekunden. Der Worker spielt Wörter nacheinander: - Zum Start des Wortes *i* sendet der Worker `word_started(i)` (MainWindow hebt Wort hervor) [42] . - **Zeit 0–**`t_1`: Wort 1 wird abgespielt (AudioEngine erzeugt Ton). - **Zeit** `t_1` **–** `t_2`: Wort 2 usw.
Dies könnte man mit einem Timeline-Diagramm darstellen (hier aus Platzgründen nur verbal).

Code-Snippets und geänderte Dateien

Zentrale Änderungen:

- `audio_engine.py`: Siehe oben. Entfernt sind die alten Fallback-Ketten (sie sind auskommentiert bzw. gelöscht). Eingefügt wurde das `QAudioOutput`-System.
- `concat_audio.py`: Analog könnte man `play()` so ändern, dass es `QAudioOutput` verwendet. Wenn wir ausschließlich die Sinus-Engine nutzen, kann man ConcatAudioEngine ganz weglassen oder nur für Datei-Export behalten.
- `main_window.py`: Bleibt unverändert, außer ggf. Logging-Anpassungen. Die Worker-Threads funktionieren jetzt mit dem neuen Engine.
- `utils/audio.py`: Kann entfallen oder unberührt bleiben (wird nicht benutzt).
- **Konfiguration**: Sicherstellen, dass in den Pfaden nix auf `pydub` verweist.

Beispieländerungen (vollständige Dateien):

```
--- taamimflow/audio/audio_engine.py  (Original)
+++ taamimflow/audio/audio_engine.py  (Modifiziert)
@@ -168, 40 +168, 25 @@
     def synthesise(
         self,
         notes: Iterable[Note],
-        tempo: float = 120.0,
-        volume: float = 1.0,
+        tempo: float = 120.0,
+        volume: float = 1.0
     ) -> Optional['AudioSegment']:
-        if not HAVE_PYDUB:
-            return None
-        seg = self.generate_audio_segment(list(notes), tempo)
```

```
-        if seg is None:
-            return None
-        # Lautstärke umrechnen...
-        return seg.apply_gain(gain_db)
+        # Neu: Erzeuge PCM-Daten (Bytearray) für Qt-Audio
+        data = self._generate_pcm(notes, tempo, volume)
+        return data  # Hier: QByteArray oder bytes mit 16-Bit-Pegel

-    def play(self, segment: Optional['AudioSegment']) -> None:
+    def play(self, data: Optional['QByteArray']) -> None:
         """Spiele ein AudioSegment ab.  Blockierend (läuft im Worker-
Thread)."""
-        if segment is None or not HAVE_PYDUB:
-            return
+        if data is None or data.isEmpty():
+            return

-        import sys, io, os, tempfile
+        # Nicht-blockierender Qt-Player:
+        buffer = QBuffer()
+        buffer.setData(data)
+        buffer.open(QIODevice.OpenModeFlag.ReadOnly)
+        self._audio_out.start(buffer)
+        # Hinweis: Hier kehrt die Funktion sofort zurück.
```

Anmerkung: Die obigen Änderungen illustrieren das Konzept. In einer echten Umsetzung müsste `AudioEngine._generate_pcm()` (oder `synthesise()`) die Sinuswellen korrekterweise als QByteArray generieren (z. B. wie im Beispiel weiter oben gezeigt). Zudem müsste man im `AudioEngine.__init__` `self._audio_out = QAudioOutput(...)` initialisieren, wie im obigen Codeausschnitt gezeigt.

Tests und Validierung

Manueller Test: Nach Installation von PyQt6 sollte man prüfen, dass bei Klick auf ein Wort sofort Ton erklingt und das Wort korrekt hervorgehoben wird. Dabei darf es keine merkbare Pause oder „Hänger“ geben. Stoppen (Escape) sollte die Wiedergabe abbrechen (z.B. durch `engine._audio_out.stop()` aufzurufen).

Unit Tests: Man kann eine Liste fester `Note`-Sequenzen synthetisieren und prüfen, ob die resultierenden PCM-Daten (z.B. Länge in Bytes) mit Erwartungen übereinstimmen. Beispiel: Für einen C4-Viertelton bei 120 BPM (0.5 s) sollten ~22050 Samples generiert werden. Auch das Abschneiden/Abspielen kann mit Mocks getestet werden (etwa Überprüfung, dass `QAudioOutput.start()` aufgerufen wird).

Thread-Test: Simulieren, dass `_WordByWordWorker` mehrere Token verarbeitet. Hier sollte die `word_started`-Signalfolge korrekt sein, auch wenn ein Wort keine Noten hat (kürzere Pause).

Quellen

Die obigen Ausführungen stützen sich direkt auf den Quellcode des Repositories **Moriahise/taamimflow_project**:

- Audio-Engine-Code und Threads: **taamimflow/audio/audio_engine.py** [6] [1], **taamimflow/core/concat_audio.py** [16] [17], **taamimflow/gui/main_window.py** [4] [27].
- Tropendefinitionen: **taamimflow/data/tropedef.py** und **taamimflow/core/cantillation.py** [34] [36] (Wie XML geladen wird), sowie **open_reading_dialog.py** für *tropedef_megillot.xml* [38] und **tropenames.py** [40].
- Threading- und GUI-Verhalten: Allgemeine Quellen wie StackOverflow (z. B. Probleme durch `time.sleep()` [3] und korrektes Verwenden von QThreads [33]) sowie die PyQt6-Dokumentation.

Diese Referenzen sind im Text oben entsprechend verlinkt.

---

[1] [2] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] audio_engine.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/audio/audio_engine.py

[3] python - stop tkinter window from freezing while program is sleeping - Stack Overflow
https://stackoverflow.com/questions/9250624/stop-tkinter-window-from-freezing-while-program-is-sleeping

[4] [5] [26] [27] [28] [29] [30] [31] [32] [42] main_window.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/gui/main_window.py

[16] [17] [18] concat_audio.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/core/concat_audio.py

[19] [20] audio.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/utils/audio.py

[21] [22] utils.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/audio/utils.py

[23] [24] timing_map.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/audio/timing_map.py

[25] tradition_profiles.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/audio/tradition_profiles.py

[33] python - PyQt5 window keeps freezing although using threads - Stack Overflow
https://stackoverflow.com/questions/53350056/pyqt5-window-keeps-freezing-although-using-threads

[34] [35] tropedef.py
https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/data/tropedef.py

36 37 cantillation.py

https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/core/cantillation.py

38 39 open_reading_dialog.py

https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/gui/open_reading_dialog.py

40 41 tropenames.py

https://github.com/Moriahise/taamimflow_project/blob/27536223f70d6f4e8c8d83519f6085f881a1bfb8/taamimflow/data/tropenames.py