Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Balogh Ákos

SZAKTERÜLETI KERETRENDSZER ZENEI VIZUALIZÁCIÓHOZ

KONZULENS

Dr. Mezei Gergely

BUDAPEST, 2020

Tartalomjegyzék

1 Bevezetés 7

1.1 A dolgozat felépítése 8

2 Felhasznált technológiák, módszertanok 9

2.1 Nyelvfeldolgozás 9

2.1.1 Szakterületi nyelvek 10

2.2 MPS 10

2.2.1 A nyelvtan megadása 11

2.2.2 Kódgenerálás 16

3 Megvalósítás 17

3.1 Motivációk 17

3.2 Specifikáció 18

3.3 A DSL felépítése 19

3.3.1 Nyelvtan 19

3.3.2 Szerkesztő 21

3.3.3 Kódgenerálás 22

3.4 Az asztali alkalmazás felépítése 26

3.4.1 Felhasznált könyvtárak 26

3.4.2 Az alkalmazás vezérlése 27

3.4.3 Az alkalmazás felépítése 27

3.4.4 Ablakkezelés 28

3.4.5 Eseménykezelés 30

3.4.6 A beállítások kezelése 32

3.4.7 Az XML állomány értelmezése 32

3.4.8 A zenék lejátszása 36

3.4.9 A soros port kezelése 41

3.5 A beágyazott rendszer felépítése 42

3.5.1 LED-szalag 42

3.5.2 Mikrokontroller 43

3.5.3 Tápegység 49

4 Az eredmények értékelése 51

4.1 Áttekintés 51

4.2 A szakterületi keretrendszer értélése 51

4.2.1 Az MPS keretrendszer értékelése 51
4.2.2 A szakterületi nyelv értékelése 52
4.3 A kliensalkalmazás értékelése 54
4.3.1 Az implementálás értékelése 54
4.3.2 Az értelmező értékelése 55
4.3.3 A spektrumszámítás értékelése 55
4.4 A hardver értékelése 56
4.4.1 A kommunikáció értékelése 56
4.4.2 A mikrokontroller értékelése 57
4.4.3 A LED-szalag értékelése 57
4.4.4 A tápellátás értékelése 58
5 Összefoglalás 59
Irodalomjegyzék 61

HALLGATÓI NYILATKOZAT

Alulírott **Balogh Ákos** kijelentem, hogy ezt a dolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye.

Kelt: Budapest, 2020. 10. 28.

...…………………………………………….

Balogh Ákos

Összefoglaló

Korábbi tanulmányaim során megismerkedtem a szakterületi nyelvek témakörével, melyek széleskörű felhasználhatóságát azóta is lenyűgözőnek találom. Eleinte a célom egy olyan nyelv megalkotása volt, amely a közismert táblás játékok szabályrendszerét képes leírni. A feladat sikeres teljesítését követően a villamosmérnöki szemléletmódhoz közelebb eső, a beágyazott rendszerekhez kapcsolódó témakört kerestem. Így jutottam el egy olyan rendszer ötletéhez, amelyben változtatható fényerejű és színű LED sorok vizualizálnak zeneszámokat. Az ötlet újdonsága, hogy a felhasználó egy intuitív szakterületi nyelv segítségével testreszabhatja a vizualizációs algoritmust, így minden zenéjéhez egyedi megjelenítést adhat meg. A TDK dolgozatomban az ötlet megvalósítását mutatom be, valamint esettanulmányok segítségével illusztrálom az elkészült rendszer működését.

A szakterületi nyelv leírásához a JetBrains Metaprogramming System (MPS) fejlesztőrendszert alkalmaztam. Az MPS a nyelv definíciója alapján egy projekcióalapú szerkesztőfelületet ad, ami a felhasználót végigvezeti a vizualizáció leírásának folyamatán. Az MPS biztosítja, hogy a felhasználó ezen a magas, könnyen érthető absztrakciós szinten fogalmazhassa meg a logikát, amiből aztán alacsonyabb szintű kódot generálok. A generált kód értelmezését egy modern C++-ban megírt asztali alkalmazás végzi, amely a zenék lejátszásáért, spektrumának számításáért, valamint mindezek grafikus megjelenítéséért is felelős. Az asztali alkalmazás soros porton keresztül kommunikál egy STM32 mikrokontrollerrel, mely előállítja a spektrum és a színpaletta szerint a LED-ek vezérlésére alkalmas adatokat, és ezt a LED-szalagok felé továbbítja is. A felhasználó vizualizációs programja és a zene spektruma közösen szabályozza a színhatásokat, valamint a fényerőt.

Ahogy a fentiekből látszik, dolgozatomban egy magasszintű szoftveres megoldást, valamint egy keretrendszert adok, ami az hardverszintű vezérlést teljesen elrejti a felhasználó elől, mégis teljes szabadságot nyújt neki a zenei vizualizáció testreszabásában.

Abstract

During my studies I became familiar with the field of domain specific languages, whose broad possibility of applications amazes me to this day. Initially my goal was to create a language which makes the description of well-known board games possible. Having accomplished this, I was looking for something a bit more relevant to the field of electrical engineering. In the end, I settled on a project that uses colourful LED-strips to visualise musical pieces. The novelty of the project is that the user can use an intuitive domain specific language to customise the visualisation algorithm, which can make every music visualisation unique. In my TDK thesis I illustrate the idea's implementation, furthermore, I showcase the working of the entire system through real world examples.

To create the domain specific language, I used the JetBrains's Metaprogramming System (MPS) framework. MPS provides a projectional editor based on the language definition, which guides the user through creating their own visualisation programs. MPS makes sure the user only needs to use the higher level, easy to understand domain specific language, without ever touching the hardware or the intermediate software. Using MPS, I generate a lower level code equivalent to the user's visualisation program, which is interpreted in a modern C++ desktop application. This application is also responsible for playing the music, calculating the spectrum, and displaying the results on a graphical interface. The music spectrum and user defined visualisation program both contribute to the luminance and colour of the individual LEDs. The desktop application communicates with an STM32 microcontroller through a specified serial port, which further transmits the appropriate LED data to the strips themselves.

In conclusion, in my thesis I showcase a high-level software solution and the corresponding framework, which hides the hardware level control from the user, while also maintaining total freedom in customising their musical visualisation.

1 Bevezetés

Szoftverek fejlesztése terén számos programozási nyelv és szoftvercsomag áll a fejlesztők rendelkezésére. A fejlesztés során érdemes figyelembe venni, hogy a projekt megvalósításához melyik programozási nyelv lenne a legalkalmasabb. A választott programozási nyelv befolyásolhatja a fejlesztéshez szükséges időt, skálázhatóságot, karbantarthatóságot, de akár a program futási idejét is.

Általában, ha a fejlesztés fő szempontja a karbantarthatóság és átláthatóság, érdemes egy olyan magas absztrakciós szintű programozási nyelvet választani, mely összetett nyelvi eleme könnyíti meg a fejlesztést. A magas absztrakciós szint egyik előnye az, hogy a programot általánosabb érvényű utasításokkal írhatjuk le, így a fejlesztés során nem szükséges a végrehajtó hardver paramétereit figyelembe venni. Ez a könnyebbség azonban magával vonja azt, hogy a program nehezen optimalizálható célhardverekre, ezzel a futási teljesítmény romlik.

Beágyazott endszerek fejlesztése esetén fő szempont a futási teljesítmény, mivel a rendelkezésre álló erőforrások erősen korlátozottak lehetnek, így az alacsonyabb absztrakciós szintű programozási nyelvek preferáltak, pl. C. Alacsony szintű programozási nyelveken a szoftver jól optimalizálható a célhardverre, viszont éppen emiatt nehéz a beágyazott szoftvert módosítani és karbantartani. A dolgozatomban bemutatok egy olyan keretrendszert, mely összeköti a beágyazott rendszerek alacsony absztrakciós szintű világát egy a modern szoftverfejlesztési elveknek megfelelő, magas absztrakciós szintű szakterületi nyelvvel.

A keretrendszer bemutatásához felvázolok egy problémát, melyet alacsony absztrakciós szintű nyelven meglehetősen nehéz lenne megvalósítani, azonban a végrehajtásához mégis le kell menni egészen a hardverig. Az dolgozat tárgya egy általam létrehozott projekt, ami zeneszámok vizualizációját valósítja meg. A vizualizáció médiuma egy LED-ekből álló szalag, melynek a zenéhez illeszkedő vezérlését a felhasználó egy magas absztrakciós szintű szakterületi nyelven állíthatja össze. Ahhoz, hogy a felhasználó vizualizációs programja értelmezhető legyen a LED-szalag által, a vizualizációs programon számos transzformációt kell végrehajtani. Ezen transzformációknak a fő feladata az absztrakciós szintek áthidalása, egészen a magas szintű szakterületi keretrendszertől a megjelenítésért felelő mikrokontrollerig. A projekt három fő részből áll:

- Az első rész a vizualizációs program leírását lehetővé tevő szakterületi nyelv és az ahhoz tartozó keretrendszer. A felhasználónak elegendő ezzel a szakterületi nyelvvel

megismerkednie, ugyanis a nyelven leírt program értelmezésének és végrehajtásának módja a szakterületi nyelv absztrakciós szintje mögé van rejtve.

- A második rész a szakterületi keretrendszer által előállított vizualizációs forráskód értelmezése. Az értelmezés pontos folyamata rejtve van a felhasználó elől, azonban a végrehajtás aktuális állapotát a felhasználó az értelmezőbe épített grafikus felületen nyomon tudja követni.
- A harmadik rész az értelmező által transzformált vizualizációs logika valódi végrehajtása a LED-szalagokat vezérlő beágyazott rendszeren.

1.1 A dolgozat felépítése

A teljes keretrendszert az alábbi fejezetekben mutatom be:

- A 2. fejezetben bevezetem a szakterületi nyelvek tárgyalásához szükséges terminológiát, egy általános, szöveges programozási nyelv elemzésének folyamatát, valamint bemutatom a szakterületi nyelv megvalósításához felhasznált MPS fejlesztőkörnyezetet is. Az MPS környezet részletes tárgyalására a dolgozat keretein belül nincsen lehetőség, így azon részeire szorítkozom, melyek a szakterületi keretrendszer megértéséhez elengedhetetlenek.
- A 3. fejezetben bemutatom a projekt megvalósítását, a projektet alkotó három alapvető logikai egységére lebontva. Külön fejezetekben tárgyalom a szakterületi nyelv megvalósítását MPS-ben, a nyelven leírt vizualizációs programnak megfelelő forráskód előállítását, annak értelmezését és feldolgozását, valamint magát a végrehajtó beágyazott rendszert is.
- A 4. fejezetben részletesen értékelem a megvalósítást a tesztelés során gyűjtött visszajelzések alapján, ezek alapján felvázolom a projekt fejlesztési lehetőségeit, valamint a jövőbeli terveket is.

2 Felhasznált technológiák, módszertanok

2.1 Nyelvfeldolgozás

A számítógépek elterjedésével együtt járó szoftveres fejlődés eredményeként számos új programozási nyelv és keretrendszer született meg. Egy új programozási nyelv létrehozása összetett feladat. Mindenekelőtt definiálni kell a nyelvtant, mely meghatározza, hogy a nyelv milyen elemeket tartalmaz. A nyelvi elemek általában szövegesek, de akár grafikusak is lehetnek. Ezután definiálni kell egy fordítási folyamatot, mely a leírt programot gépi kóddá alakítja át. Szöveges nyelv esetén a fordítás lépései [1]:

- Lexikai elemzés: a szövegesen tárolt programot önálló nyelvi elemekre, azaz tokenekre osztjuk. A tokenek általában az operátorok, a nyelv kulcsszavai, azonosítók.
- Szintaktikai elemzés: a tokenek sorozatát tovább alakítja egy könnyebben feldolgozható adatszerkezetté. A szintaktikai elemzés kimenete az absztrakt szintaxisfa (abstract syntax tree, AST).
- Szemantikai elemzés: a szintaxisfát vizsgálja szemantikai problémák után kutatva. Szemantikai problémának minősül a kódban érvénytelen azonosítók használata, vagy nem kompatibilis típusú kifejezések leírása.
- Optimalizáció: a szemantikailag helyes szintaxisfában optimalizációs lépések végrehajtása. Kódot optimalizálni rendkívül összetett feladat, néhány példája a *common subexpression*[1] keresése, vagy *copy propagation*[2].
- Kódgenerálás: a szintaxisfát gép által értelmezhető utasítássorozatra fordítjuk.

A fordítás eredménye egy olyan adatszerkezet, mely megfelel a programban leírtaknak, és a számítógép tudja értelmezni. Az értelmezés formája nyelvtől függően eltérő lehet. A kapott adatszerkezet tartalmazhat közvetlenül végrehajtható gépi kódot, vagy akár köztes nyelven tárolt utasításokat, melyeket egy értelmező (interpreter) hajt végre. A dolgozatban tárgyalt nyelv is egy értelmező által kerül végrehajtásra.

[1] Több értékadás esetén felhasználható az előző eredmény, ha a kifejezés nem változik.

[2] Értékadás forrásváltozóinak felcserélése, ha értékük azonos.

2.1.1 Szakterületi nyelvek

Számítógépes programok leírására számos programozási nyelv áll a fejlesztők rendelkezésére. A fejlesztés során érdemes figyelembe venni, hogy a program leírására melyik nyelv lenne a legalkalmasabb. A nyelv megválasztása befolyásolja a fejlesztéshez szükséges időt, skálázhatóságot, és egyéb tulajdonságokat is.

Fontos szempont a programozási nyelv kiválasztásakor a nyelv által biztosított lehetőségek vizsgálata. Azokat a nyelveket, melyeket széleskörűen felhasználható nyelvi elemekkel láttak el, általános célú programozási nyelveknek (general purpose language, GPL) nevezzük, pl.: C++. Ezzel szemben azokat a nyelveket, melyek nyelvi elemei egy szűk problémakör leírását teszik lehetővé, szakterületi nyelveknek (domain specific language, DSL) nevezzük [2], pl. SQL.

2.2 MPS

A projekthez tartozó szakterületi nyelv megvalósításához a JetBrains által fejlesztett Metaprogramming System (MPS) névre keresztelt szoftvercsomagot választottam [3][4], melyet kifejezetten szakterületi nyelvek létrehozására terveztek. Az MPS rendszer meglehetősen összetett, így részletes tárgyalásához érdemes bevezetni néhány alapfogalmat:

- Projekcióalapú szerkesztő: MPS-ben a programok leírására nem nyers szövegként, hanem az úgynevezett projekcióalapú szerkesztőn keresztül van lehetőség. Projekcióalapú szerkesztővel közvetlenül a programnak megfelelő szintaxisfa kerül bevitelre, így nincsen szükség lexikai és szintaktikai elemzésre.
- Koncepció: legkisebb nyelvi elem. Minden nyelv koncepciókból épül fel, a nyelvtant koncepciók kapcsolata adja. A koncepciókhoz kulcsszavakkal rendelhető különleges működés, mely meghatározza a koncepció fajtáját. Egy lehetséges fajta a gyökérkoncepció, mely egy programot leíró szintaxisfa gyökerét jelenti.
- Modul: minden koncepció modulokból épül fel. A koncepciók működését a modulok határozzák meg. Modul lehet pl. a koncepcióhoz tartozó projekció-alapú szerkesztőt megvalósító modul, a koncepcióból kódot generáló modul, valamint a koncepció változóinak értelmezési tartományát meghatározó modul is.
- Sandbox: a nyelvhez opcionálisan tartozhat egy „homokozó”. A homokozóban próbálhatjuk ki az egyes koncepciókat, valamint gyökérkoncepciók példányosításával teljes programokat is leírhatunk.

- Generátor: a nyelvhez opcionálisan tartozhat egy generátor. A generátor a nyelvhez tartozó transzformációs lépéseket foglalja magába, melyek megadják, hogy a szerkesztőben leírt programból hogyan állítható elő a kimenet. A kimenet lehet futtatható kód, de akár valamilyen köztes nyelven írt program is.
- Nyelv: egybetartozó koncepciókat, Sandboxot, generátort, és egyéb nyelvi modulokat összefogó logikai egység. Egy nyelv felhasználható közvetlenül a Sandboxban, vagy felhasználható importként egy másik nyelvben.
- Projekt: egy MPS projekt, melyben a fejlesztés zajlik. Tartalmazhat több egymástól teljesen független nyelvet, Sandboxot, generátort is.

2.2.1 A nyelvtan megadása

MPS-ben a nyelvi elemeknek koncepciók felelnek meg. Mivel az MPS projekcióalapú szerkesztőt használ, a nyelvtan megadásához elegendő az egyes koncepciók relációinak meghatározása, nincsen szükség szintaktikai, az érvényes karaktereket leíró, vagy egyéb szabályok explicit megadására. A koncepciók közti relációk az alábbiak lehetnek:

- Gyökérkoncepció: egy szintaxisfa gyökéreleme, hozzá tartozó leveleket definiál. A gyökérkoncepciók példányosítására a Sandboxban van lehetőség. Egy nyelvben több gyökérkoncepció is lehet, így többféle gyökér köré is építhetünk egymástól független szintaxisfákat. Egy Sandboxban több gyökérelem is lehet, akár azonos típusúak is.
- Leszármaztatás: két koncepció közti szülő-gyerek kapcsolat. A gyerek a szülő mindegyik tulajdonságát örökli (változók, modulok). A modulokat a leszármazott felüldefiniálhatja. Egy koncepciónak egyetlen őse lehet, azonban többszintű öröklés esetén a leszármazott az összes őskoncepció tulajdonságaival rendelkezni fog. Közös ős alapján a koncepciók kollekciókba szervezhetők.
- Tartalmazás: másik koncepció példányának tartalmazása. A tartalmazás számossága lehet 1, 0..1, 0..n, 1..n. Amennyiben a tartalmazás számossága nagyobb, mint 1, a tartalmazás kollekciónak tekinthető.
- Referencia: másik koncepció példányaira való hivatkozás. A hivatkozás számossága lehet 1, 0..1.

A koncepciók működését modulok létrehozásával határozhatjuk meg. MPS-en belül számos modul áll rendelkezésre. A VSR megvalósításához elegendő három modul széleskörű felhasználása, melyeket alább egy-egy példán keresztül mutatok be.

2.2.1.1 Egy koncepció definiálása

Egy koncepció létrehozásakor az alábbi adatszerkezetet kell kitölteni. Az adatszerkezet három fő részből áll, a fejlécből (concept, extends, implements), a példányosítás metaadataiból (root, alias, short description), valamint a koncepció tagváltozóiból (properties, children, references).

```
concept <no name> extends    BaseConcept
                  implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  << ... >>

  references:
  << ... >>
```

1. ábra: Koncepciót definiáló adatszerkezet

A fejlécben a concept kulcsszó után lehet megadni a koncepció azonosítóját, az extends kulcsszó után lehet meghatározni a koncepció egyetlen ősét, továbbá az implements kulcsszó után lehet megadni a koncepció által megvalósított interfészeket. A megvalósított interfészek száma nincsen korlátozva, sőt, interfész típusú koncepció létrehozásával sajátot is definiálhatunk. A fejlécben ezeken kívül megadható más kulcsszó is, pl. abstract (a koncepció absztrakt, így nem példányosítható, de képezheti más koncepciók ősét), vagy final (a koncepció nem képezheti más koncepció ősét).

A példányosítás metaadatainak azokat a paramétereket tekintem, melyek befolyásolják a koncepció példányosításának tulajdonságait. Ilyen tulajdonság, hogy a koncepció gyökérkoncepció-e, azaz létrehozható közvetlenül a Sandboxban. Ezt az instance can be root logikai változó határozza meg. Az alias és a short description kulcsszavak után megadható egy-egy szöveges állomány, mely a projekcióalapú szerkesztőben kerül felhasználásra. Az alias általában egy rövid, pár karakterből álló azonosító, melynek begépelésekor a szerkesztő

létrehozza a koncepció egy példányát. A short description mezőben a koncepció egy rövid leírását adhatjuk meg, mely a szerkesztő automatikus kódkiegészítő listájában jelenik meg.

A koncepció tagváltozóit a properties, children, valamint a references mezők kitöltésével hozhatjuk létre. A properties mezőben sorolhatók fel az MPS-be épített típusú változók példányai (integer, string, boolean). A children mezőben sorolhatók fel azon tagváltozók, melyeknek típusa általunk definiált koncepció. A references mezőben hozhatók létre más koncepciók példányaira mutató hivatkozások.

```
concept Song extends    BaseConcept
             implements INamedConcept

  instance can be root: true
  alias: <no alias>
  short description: song

  properties:
  filepath : string

  children:
  timestamps : ControlPoint[1..n]

  references:
  << ... >>
```

2. ábra: Egy koncepció definíciója

A 2. ábra példája esetében a koncepció azonosítója a Song. A koncepció őse a BaseConcept, mely az összes koncepció közös ősének tekinthető, a Java nyelv Object típusához hasonlóan. A Song koncepció megvalósítja az INamedConcept interfészt, mely egy string típusú, name azonosítójú változó létrehozásával ekvivalens. A koncepció gyökérkoncepció, így a példányai a Sandboxban kerülnek létrehozásra. Mivel a példányokat nem a projekcióalapú szerkesztőn belül hozzuk létre, nincsen szükség alias definiálására. A koncepció tartalmaz egy string típusú, filepath azonosítójú változót, mely a zenefájl elérési útját tárolja. Tartalmaz továbbá egy timestamps azonosítójú kollekciót, melynek típusa a ControlPoint koncepció és számossága 1..n.

2.2.1.2 Editor modul

Koncepciók példányosításakor a koncepcióhoz rendelt szerkesztőmodul (editor) kerül megjelenítésre. Amennyiben nincsen szerkesztőmodul definiálva, alapértelmezetten egy XML-sémához hasonló szöveges szerkesztő kerül generálásra. Általában a szerkesztőmodulok konstans stringekből és beviteli mezőkből állnak. A beviteli mezők tartozhatnak egyetlen

változóhoz, de akár egy teljes kollekcióhoz is. A szerkesztőmodulnak az a feladata, hogy a példányra jellemző stringek és beviteli mezők egymáshoz képesti elhelyezkedését és kinézetét rögzítse. A beviteli mezők lehetnek egyszerű szövegdobozok, de akár összetett grafikus elemek is, pl. JavaFX grafikus komponensek. Az alábbi ábrákon egy szerkesztőmodul forráskódja és a modul realizációja látható.

```
<default> editor for concept Song
  node cell layout:
    [-
    song
      title { name }
      file path { filepath }
      visualisation:
      (/ % timestamps %                /)
         /empty cell: <default>
    -]
```

3. ábra: Egy szerkesztőmodul

```
song
  title 01 - We Will Rock You
  file path 01 - We will rock you.flac
  visualisation:
  @ 0 ms ( 100 | 100 )
    << ... >>

  @ <no stamp> ms ( <no volume> | <no brightness> )
    << ... >>
```

4. ábra: A modul realizációja

A 3. ábra mutatja a modul forráskódját. Ahogy korábban elhangzott, a modul beviteli mezőkből és konstans stringekből áll. Konstans string a song, title, file path, valamint a visualisation szavak. Ezek félkövér stílussal kerülnek megjelenítésre a forráskódban és a realizációban egyaránt. A forráskódban kétféle beviteli mező látható, közöttük az eltérés az értékadás számossága. Egyetlen változó esetén a forráskódban a változó azonosítóját kapcsoszárójelek közé zárva hozható létre egy típushelyes beviteli mező. Erre példa a name és a filepath változók string típusú beviteli mezeje. Kollekciók esetén meg kell határozni, hogy a kollekció egyes példányai hogyan helyezkedjenek el egymáshoz képest. A 4. ábra példája esetén a kollekció elemeit függőlegesen egymás alá rendezem, melyet a (/--/) operátor valósít meg (vertical child node list). A kollekció azonosítóját az operátorban kell megadni. Kollekciót kizárólag koncepciókból lehet létrehozni. A kollekció egyes elemeinek példányosításakor a típushoz rendelt szerkesztőmodul jelenik meg. A 4. ábra példája esetében a példányok szerkesztőmoduljait az @ operátor jelzi.

2.2.1.3 Constraints modul

Koncepciók példányosításakor a tagváltozók értékeit a szerkesztőmodul beviteli mezőin keresztül módosíthatjuk. Ezek a beviteli mezők típushelyesek, tehát pl. egész szám típusú változónak nem vihetünk be szöveges adatokat. Amennyiben a változók bevitelekor nem csak a típushelyesség, hanem a változó bizonyos értékek közé szorítása is követelmény, a koncepcióhoz megszorításokat kell rendelni (constraints). A modulban felsorolhatjuk a

koncepció megszorítani kívánt változóit, és egy-egy függvényt rendelhetünk hozzájuk, melyek a visszatérési értékükkel jelzik, hogy a változó milyen tartományban érvényes. Amennyiben a bevitt változó értéke az értelmezési tartományon kívül esik, fordítási hibát kapunk.

```
property {index}
  get <default>
  set <default>
  is valid (propertyValue, node)->boolean {
    if ((propertyValue >= 0) && (propertyValue < 128)) { return true; }
    return false;
  }
```

5. ábra: Egy változó értékének megszorítása

Az 5. ábra esetében az index azonosítójú változó kerül megszorításra. Az érvényességével visszatérő függvény a propertyValue argumentumán keresztül kapja meg a változó szerkesztőben bevitt értékét. A példán a változó értéke 0 és 127 között mozoghat, ilyenkor a visszatérési érték igaz, egyébként hamis.

2.2.1.4 TextGen modul

Az egyes koncepciók közvetlen forráskóddá alakítását a koncepcióhoz rendelt szöveggenerátor modulok valósítják meg (TextGen). A modulban meg kell határozni, hogy a koncepció változóit milyen formátumban alakítjuk forráskóddá. A formátumot az elvárt szintaktika, és egyéb beillesztési feltételek adják, pl. a jelenlegi indentációs szint.

```
text gen component for concept Color {
  (node)->void {
    append {RGBA(} ${"" + node.r} {, } ${"" + node.g} {, } ${"" + node.b} {, 255)};
  }
}
```

6. ábra: Egy TextGen modul

A 6. ábra egy színkódot tartalmazó koncepció TextGen modulját illusztrálja. A koncepció a színkódot RGB formátumban tárolja, melyeknek egyes bájtjait az r, g és b egész típusú változók tárolnak. A változók értékei a ${--} operátorral érhetők el. Az operátorban string típusnak kell állnia, hogy a generálás sikeres legyen. Kézenfekvő megoldás a változók értékét egy-egy üres stringhez appendálni, majd ezt a forráskódba beilleszteni.

2.2.2 Kódgenerálás

Az MPS lehetőséget ad arra, hogy a saját nyelvünkön leírt programokat lefordítsuk. Ahhoz, hogy a programunk lefordítható legyen, a programban felhasznált koncepciókhoz generálási szabályokat kell hozzárendelni. A generálási szabályok határozzák meg, hogy a koncepció egy példányán milyen transzformációs lépéseket kell elvégezni, hogy a generálás kimenete előálljon. Az egy nyelvhez tartozó generálási szabályokat a nyelv generátor modulja foglalja össze.

Általában a generátor a DSL-lel leírt szintaxisfából indul ki, ezt tekintjük a generátor bemeneti modelljének, a generálás végeredményét pedig a kimeneti modellnek. Azért van szükség ilyen általánosításra, mert a kimeneti modell lehet forráskód és futtatható állomány is, de akár képezheti egy másik generátor bemeneti modelljét is. Sok esetben a generálás nem hajtható végre egyetlen lépésben. Egyetlen generálási lépés végrehajtását a bemeneti modell egyszeri redukálásának tekintjük. Több generálási lépés esetén a kimeneti és bemeneti modell közti állapotokat tranziens modellek írják le.

Az MPS sokféle generálási szabályt biztosít, pl. gyökérkoncepció alapján új forrásfájl létrehozása (root mapping rule), bemeneti koncepció elvetése (abandon roots), több koncepció összefűzése tranziens modellé (weaving rule). Minden generálási szabály esetén meg kell határozni egy mintát (template), amely alapján az egyes példányokhoz tartozó kimeneti modell előállítható.

A kimeneti modell forráskóddá való alakítását generátor makrók vezérlik. Ezek a makrók felelősek azért, hogy a kimeneti modellben tárolt szintaxisfa elemei a forráskód megfelelő helyére kerüljenek, az elvárt szintaxissal. Számos generátor makró áll rendelkezésünkre, pl. property macro (beépített típusú változó értékének forráskóddá alakítása), reference macro (hivatkozott példány belső változóinak elérése), loop macro (kollekció bejárása). A korábban említett generálási mintákat generátor makrókkal kiegészítve állíthatjuk elő a kívánt forráskódot.

3 Megvalósítás

3.1 Motivációk

Korábbi tanulmányaim során megismerkedtem a beágyazott rendszerek alacsonyszintű világával, valamint a szakterületi nyelvek magas absztrakciós szintjében rejlő lehetőségekkel is. A TDK dolgozatom témájához igyekeztem egy olyan projektet keresni, mely közelebb áll a villamosmérnöki szemlélethez, mégis magában foglal valamilyen magasabb szintű szoftveres megoldást. Így jutottam el addig az ötletig, hogy zeneszámok látványos vizualizációját valósítsam meg.

Az ötlet felvázolása után kutatni kezdtem, hogy a zenei vizualizáció eddig milyen formában került megvalósításra, ami számomra is elérhető. Számos megoldást találtam, melyek a zene spektruma alapján modulálnak valamilyen megjelenítő eszközt, legyen ez akár LED-szalag[3] (SparkFun projekt), vagy összetettebb hardveres elemeket is tartalmazó beágyazott rendszer[4] (Acrylic Tower).

Mindegyik megoldás kifejezetten látványos és kellemes élményt nyújt, azonban számos hiányossággal rendelkeznek, melyet egy összetettebb projekttel lehet csak kiküszöbölni. Számomra a SparkFun projektben látható LED-szalagos vizualizáció nyújtotta a legjobb élményt, ezért a saját projektemben is hasonlót szerettem volna megvalósítani. A SparkFun projektben a LED-ek fényereje a spektrum szerint változnak, azonban számomra nagy hiányosság, hogy a LED-ek színe látszólag a spektrumtól és a zene hangulatától függetlenül változik. Az én elképzelésem szerint a LED-ek színének megválasztása éppoly fontos a hangulat közvetítésében, mint a fényerő modulálása.

Mivel a hangulathoz illesztett színvilág erősen szubjektív, szerettem volna lehetőséget biztosítani arra, hogy a felhasználók a színpalettát a saját ízlésükhöz igazítsák. Az eredeti elképzelésem szerint a színvilágot a felhasználók úgy adhatták volna meg, hogy hozzárendelik az egyes zenei stílusokhoz szerintük legjobban illő színt. Az aktuális színvilágot ezen hozzárendelés felhasználásával egy algoritmus alapján választottam volna ki. Az algoritmus a zene spektruma és műfaja alapján határozta volna meg a zene hangulatát, ami alapján

[3] https://learn.sparkfun.com/tutorials/interactive-led-music-visualizer/all

[4] https://www.youtube.com/watch?v=GtKIkkLkrwU

kiválasztható a felhasználó által elvárt színvilág. Sajnos azonban az, hogy a zene spektrumából a hangulatára következtessünk, egy máig megoldatlan kérdés. Manapság sikerült eredményeket elérni mesterséges intelligencia köré épített analizátorokkal [5], azonban ennek az implementálási meglehetősen nehéz, ráadásul az algoritmus pontatlan is, így nem alkalmaztam a projektben.

Ennek ellenére mindenképpen ragaszkodtam egy olyan projekt megvalósításhoz, amely a felhasználó igényei szerint változtatja a vizualizáció színpalettáját. Ez megvalósítható azzal, ha a színpaletta leírását rábízom a felhasználóra magára. A színpaletta leírására alkalmazható lenne akár a beágyazott rendszer C nyelvű forráskódjának kiegészítése, azonban ez még tapasztalt programozóknak is kihívást jelentene. Erre megoldást nyújt egy olyan szakterületi nyelv létrehozása, melynek absztrakciós szintje a végfelhasználók igényeihez illeszthető, mégis lehetővé teszi a zenék színpalettáinak leírását. A szakterületi nyelven könnyebben leírható a vizualizációs logika, viszont értelmezése összetett feladat.

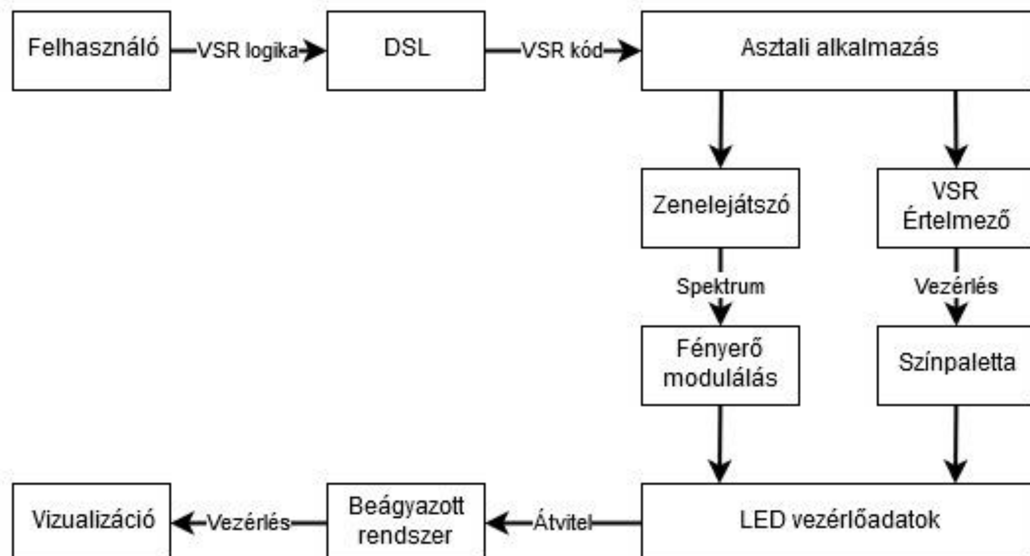
3.2 Specifikáció

Az előző fejezet megfontolásai alapján az alábbi specifikációt tűztem ki:

- A vizualizáció médiuma legyen színes LED-szalag!
- A szalag egyes LED-jeinek fényereje legyen időfüggő, a zene spektrumának változásaihoz igazodva.
- A szalag színpalettája legyen időfüggő, a felhasználó igényeihez igazítva.
- A felhasználó az időfüggő színpalettát egy magas absztrakciós szintű szakterületi nyelven tudja leírni.
- A szakterületi nyelven leírt program értelmezésének menete legyen elrejtve a felhasználó elől.
- A felhasználó rendelkezzen valós-idejű visszajelzéssel a vizualizáció jelenlegi állapotáról.
- A felhasználó tudja valós-időben módosítani a lejátszás alapvető paramétereit, pl. maximális hangerő és maximális fényerő, zene megállítása.
- A vizualizációs program végrehajtásáért egy külön beágyazott rendszer legyen felelős.

- A vizualizáció megvalósítása legyen skálázható hardver és szoftver tekintetében is.

A specifikációt megvalósító keretrendszer hatásvázlatát az 7. ábra illusztrálja. Az illusztráció egyes részeit részletesen az alábbi fejezetekben tárgyalom.



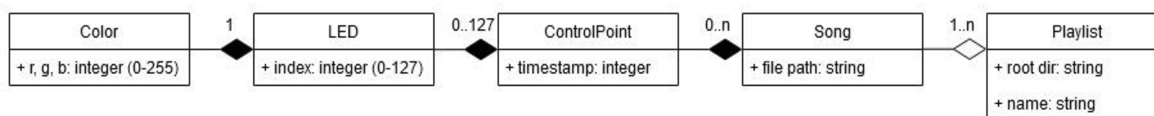
7. ábra: A megvalósítás hatásvázlata

3.3 A DSL felépítése

A vizualizációs logikát leíró szakterületi nyelvet (Visualiser, VSR) az MPS keretrendszerben valósítottam meg. A szakterületi nyelvhez létrehoztam egy projektet, ami magában foglalja a nyelvtant, a nyelvtannak megfelelő szerkesztőt, valamint a generálási szabályokat is. A nyelvtani szabályoknak megfelelően létrehoztam egy projekcióalapú szerkesztőt, mely egy intuitív felhasználói felületen vezeti végig a felhasználót a vizualizációs program létrehozásán. A projekthez rendelt generálási szabályok gondoskodnak arról, hogy a VSR nyelven leírt program olyan állománnyá forduljon, amit az asztali alkalmazás könnyen tud értelmezni.

3.3.1 Nyelvtan

A VSR alapvetően a LED-sorok színpalettájának leírására alkalmas nyelv. A színpaletta a felhasználó igényeinek megfelelően módosítható, és akár időfüggő is lehet. Időfüggő esetben a színpaletták a zeneszámok egyes időpillanataihoz kerülnek hozzárendelésre. Ezt a működést az alábbi koncepciók bevezetésével valósítottam meg, melyek közül az egyiknek részletesen is bemutatom a felépítését. A koncepciók relációit az alábbi diagram illusztrálja:



8. ábra. A VSR koncepcióinak relációi

- Color: egy szín RGB kódját tároló koncepció. Tartalmaz három egész számot, melyek értékei a 0 és 255 között mozoghatnak.
- LED: a szalag egyetlen LED-jét reprezentáló koncepció. Tartalmaz egy egész számot, mely a LED indexét jelöli, valamint tartalmazza a Color koncepciónak egy példányát. A LED szalagon 128 LED található, emiatt az index paraméter értéke 0 és 127 között érvényes.
- ControlPoint: egy teljes színpalettát megvalósító koncepció. Tartalmaz egy egész számot, mely a színpalettához rendelhető időpillanatot tárolja ezredmásodperc felbontásban. Tartalmaz továbbá egy LED-ekből álló kollekciót, melyben felsorolhatjuk a szalag egyes LED-jeinek színét, ezzel a színpalettát meghatározva. Nem szükséges mindegyik LED-hez közvetlenül színt rendelni. Azon LED-ek színe, melyek nem kerültek rögzítésre, a környezetében lévő LED-ek színéből számítható. A színpaletta megadása mellett megadható még az időpillanathoz tartozó maximális hangerő és maximális fényerő értéke is, százalékos értékben.
- Song: egyetlen zeneszámnak megfelelő koncepció. Tartalmaz egy stringet, melyben a zenefájl relatív címe adható meg. Tartalmaz továbbá egy ControlPoint-okból álló kollekciót, melyben felsorolhatjuk a zene egyes időpillanataihoz rendelt színpalettákat. A felsorolt színpaletták képzik az egyes zeneszámok vizualizációs logikáját. Nem szükséges minden időpillanathoz színpalettát rendelni. A rögzített időpillanatok közötti színpalettát az előző és következő paletta átmenete adja.
- Playlist: egy teljes lejátszási listát tároló koncepció. Tartalmaz két stringet, valamint egy Song koncepciókból álló kollekciót. Az egyik string a lejátszási lista nevét tárolja, mely a generálás végeredményéül kapott forrásfájl nevét is képzi. A másik stringben a zenefájlok elérési útjának gyökérkönyvtárát adhatjuk meg. A Song koncepcióban tárolt relatív címek ezen gyökérkönyvtártól számítandók. A lejátszási lista zenéit a Song-ot tároló kollekció feltöltésével adhatjuk meg. Ebben a kollekcióban a Song koncepció egyes példányaira mutató referenciákat gyűjthetjük

össze. A lejátszási lista vizualizációs logikája a felsorolt zeneszámok vizualizációs logikájából áll össze.

3.3.2 Szerkesztő

Ahogyan korábban elhangzott, a szintaxisfákat gyökérelemek példányai köré tudjuk felépíteni. A gyökérelemek a Sandboxban példányosíthatók. Az előző fejezetben tárgyalt koncepciók közül a Song és Playlist koncepciók lehetnek gyökérelemek. Egy Sandboxon belül a gyökérelemek száma nincsen korlátozva. Általában egy Sandboxon belül egyetlen lejátszási lista és számos zeneszám kerül példányosításra. Egy zeneszám példányosításakor az alábbi ábrán látható szerkesztőfelület jelenik meg.

```
song
  title 01 - We Will Rock You
  file path 01 - We will rock you.flac
  visualisation:
  @ 0 ms ( 100 | 0 )
    << ... >>

  @ 1200 ms ( 100 | 100 )
    LED 40 : Color( 153 , 0 , 0 )
    LED 80 : Color( 32 , 0 , 0 )

  @ <no stamp> ms ( <no volume> | <no brightness> )
    LED <no index> : Color( <no r> , <no g> , <no b> )
```

9. ábra: Egy zeneszám szerkesztése

A zeneszámok szerkesztőfelületén kötelező megadni a zeneszám címét, valamint a zenefájl elérési útját, ezek hiánya fordítási hibát okoz. A vizualizáció kulcsszó után sorolhatjuk fel az egyes vezérlési pontokat. Az ábrán három vezérlési pont példánya is látható:

1. Sorban a legfelső egy olyan vezérlési pont, ami 0-1000ms-ig érvényes. 1000ms előtt egyetlen LED-nek sincs meghatározva a színe, a lejátszás hangereje 100%, valamint a maximális fényerő 0% értékeken van (minden LED ki van kapcsolva).

2. A második vezérlési pont 1200ms-hez van hozzárendelve. Ezen vezérlési pont már rögzíti a 40-es és 80-as indexű LED-ek színét, előbbinek egy világosabb (RGB: 153, 0, 0), utóbbinak egy sötétebb piros színt (RGB: 32, 0, 0). Mind a hangerő, mind a maximális fényerő 100%-ra van állítva, így a LED-ek már világítani fognak. A nem felsorolt LED-ek színét az értelmező program fogja meghatározni.

3. A harmadik vezérlési pont adatszerkezetei kitöltésre várnak, ezeket a piros szövegrészek jelölik. A szövegrészek utalnak arra is, hogy melyik mezőben melyik változó értékét adhatjuk meg.

Az összes kívánt zene példányosítása és vizualizációs logikájának megadása után a 10. ábra szerint szervezhetjük őket egy lejátszási listává. Az ábrán látható szerkesztő a Playlist koncepció példányosításakor jelenik meg. A szerkesztő fejlécében meg kell adni a lejátszási lista nevét, valamint a zenefájlok gyökérkönyvtárát. Ezek hiányában szintén fordítási idejű hibát kapunk. A fejléc kitöltése után fel kell tölteni a songs kulcsszóval jelölt kollekciót. A kollekcióba a Sandboxban létrehozott Song példányokra mutató referenciákat tudunk behelyezni, új Song példányt ezen a szerkesztőn keresztül nem lehet létrehozni. A listában a zenék szerkesztőjében megadott cím jelenik meg. Az alábbi ábrán látható a lista feltöltésének menete az automatikus kódkiegészítés menü segítségével.

```
Playlist Queen
root directory: D:\Music\Queen\News of the World\
songs:
01 - We Will Rock You
02 - We are the champions
03 - Sheer heart attack
04 - All dead, all dead
05 - Spread your wings
06 - Fight from the inside
07 - Get down, make love
08 - Sleeping on the sidewalk
09 - Who needs you
1
  10 - It's late              ^Song (p.s.p.Queen)
  11 - My melancholy blues    ^Song (p.s.p.Queen)
  01 - We Will Rock You       ^Song (p.s.p.Queen)
  Press Ctrl+Alt+B to Show item trace
```

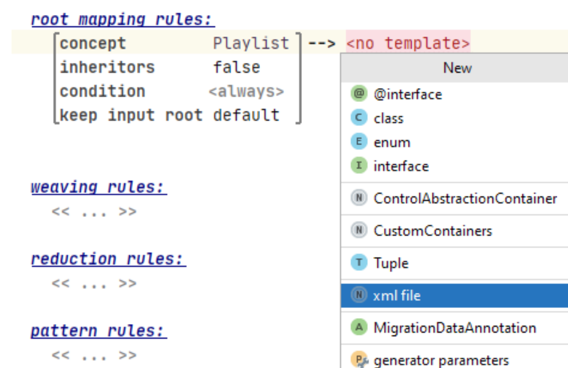
10. ábra: A lejátszási lista szerkesztése

3.3.3 Kódgenerálás

Az előző fejezetekben bemutatásra került a VSR nyelvhez tartozó projekcióalapú szerkesztő, és a szerkesztőn keresztül elérhető nyelvi elemek. A szerkesztőn keresztül könnyen előállítható a vizualizációs logika AST-je, azonban ez az AST közvetlenül nem értelmezhető, ezért annak lefordítására van szükség. A fordítás kimenete lehet tetszőleges formátumú szöveges állomány, így akár C++ forráskód is, ami közvetlenül felhasználható az asztali alkalmazás fordítása során. A VSR esetében azonban nem célszerű C++ kódot generálni, mert akkor a lejátszási lista cseréjéhez újra kéne fordítani a teljes programot. Ezen megfontolások

miatt egy leíró fájlformátum mellett döntöttem, melyben könnyen tárolható a vizualizációs logika. A leíró fájl elérési útját az asztali alkalmazás indítási argumentumként át tudja venni.

Az ismert leíró formátumok közül az XML-re esett a választásom, mivel az MPS natívan támogatja, beépített modulok importálásával. Mivel az XML formátumot kizárólag a generáláskor használom fel, elegendő a projekt generátor moduljába importálni az XML nyelvtanát megvalósító nyelvi modult. Az importálás után létrehozhatók gyökérkoncepciókhoz rendelt redukciós szabályok, melynek kimenete XML fájl. Egy ilyen redukciós szabály (root mapping rule) létrehozását a 11. ábra mutatja.



11. ábra: XML redukciós szabály

A generálási cél létrehozása után meg kell határozni, hogy az MPS a Playlist koncepcióból milyen minta alapján generáljon forráskódot. A generálási mintának önmagában is érvényes XML dokumentumnak kell lennie. Mivel az MPS natívan támogatja XML dokumentumok generálását, már a minta szerkesztésekor jelzi a szintaktikai hibákat. Amennyiben a kiinduló minta sziktaktikai hibát tartalmaz, fordítási hibát kapunk még a generátor meghívása előtt.

Az MPS-ben leírt AST forráskódra való leképezését generátor makrók végzik. Ezeket a makrókat a generátor kódban $ előtag jelzi. Az MPS számos generátor makrót biztosít, ezek közül azonban elengedő két típus használata a lejátszási listák generálásához. Az egyik típus kollekciókon való iterálást végez (loop macro, $LOOP$), a másik pedig koncepciók példányaihoz rendelt változók értékéből generál szöveget (property macro, $[…]).

A generálás három szintre bontható. A legalacsonyabb szint egyetlen vezérlési pontból állít elő szöveges állományt. A legalacsonyabb szintet megvalósító generálási mintát a 12. ábra mutatja.

```
<ControlPoint timestamp="$[100000]" volume="$[100]" brightness="$[100]">
  $LOOP$[<bin index="$[100]" color="$[RGBA(255, 255, 255, 255)]"></bin> ]
</ControlPoint>
```

12. ábra: Egyetlen vezérlési pont generálása

Az ábrán látható iteratív makró a vezérlési pontokhoz rendelt LED kollekción iterál végig. A kollekció összes LED-jéhez hozzárendelésre kerül egy bin XML node. A bin node az index és color attribútumokban tárolja az aktuális LED paramétereit. Minden egyes LED új sorban kerül a dokumentumba. A bin node-okat egyetlen ControlPoint node fogja közre. A ControlPoint node attribútumai tárolják a vezérlési pont időpillanatát, hangerejét és fényerejét.

A generálás második szintje az egyes zeneszámok forráskódjának generálása, melyet a 13. ábra illusztrál. Egy zeneszámhoz számos vezérlési pont rendelhető hozzá. Ezeken a vezérlési pontokon egy iteratív makró fut végig, mely minden egyes vezérlési pontra elvégzi az alacsony szintű, 12. ábra által mutatott generálást. A ControlPoint node-okat egy Song node tartalmazza. Ezen Song node attribútuma tárolja a zenefájl relatív elérési útját.

```
<Song file="$[fpath]">
  $LOOP$[<ControlPoint timestamp="$[100000]" volume="$[100]" brightness="$[100]">
           $LOOP$[<bin index="$[100]" color="$[RGBA(255, 255, 255, 255)]"></bin> ]
         </ControlPoint>
         ]
</Song>
```

13. ábra. Zeneszámok generálási mintája

A generálás harmadik szintje a teljes lejátszási lista generálása. Ez a működés szintén egy iteratív makró alkalmazásával valósítható meg. Ezen iteratív makró a Playlist gyökérkoncepció egy példányához rendelt zenéken iterál végig, és mindegyikre végrehajtja a 13. ábra generálási mintáját. Mindegyik szintet összevetve az alábbi ábrán látható generálási mintát kapjuk.

```
xml $[Playlist].xml

<?xml version = "1.0" encoding = "default" standalone = "default" ?>
<Playlist name="$[Playlist]" mroot="$[musicroot]">
  $LOOP$[<Song file="$[fpath]">
           $LOOP$[<ControlPoint timestamp="$[100000]" volume="$[100]" brightness="$[100]">
                    $LOOP$[<bin index="$[100]" color="$[RGBA(255, 255, 255, 255)]"></bin> ]
                  </ControlPoint>
                  ]
         </Song>
         ]
</Playlist>
```

14. ábra: A teljes generálási minta

Látható, hogy az összes zeneszámot közrefogja egy Playlist XML node. A node-hoz két attribútum kerül hozzárendelésre, az egyik a lejátszási lista nevét, a másik pedig a zenék elérési

útjának gyökérkönyvtárát határozza meg. A generált fájl nevét a lejátszási lista neve adja. A generált fájlt ellátom a szabványos XML prolog sorral. A generálás végeredményének első pár sora az alábbi kódrészletben látható. Egy hosszabb lejátszási listából generált XML állomány akár több ezer soros is lehet, aminek kézi értelmezése és módosítása meglehetősen nehéz. A kód értelmezése azonban könnyen algoritmizálható, melyet az asztali alkalmazásban implementálok.

```
<?xml version = "1.0"?>
<Playlist name="Queen" mroot="D:\Music\">
  <Song file="Queen\News of the World\01 - We Will Rock You.flac">
    <ControlPoint timestamp="1000" volume="100" brightness="0">
      <bin index="0" color="RGBA(0, 0, 0, 255)"></bin>
    </ControlPoint>
    <ControlPoint timestamp="1200" volume="100" brightness="100">
      <bin index="40" color="RGBA(153, 0, 0, 255)"></bin>
      <bin index="80" color="RGBA(32, 0, 0, 255)"></bin>
    </ControlPoint>
    <ControlPoint timestamp="1900" volume="100" brightness="100">
      <bin index="40" color="RGBA(153, 0, 0, 255)"></bin>
      <bin index="80" color="RGBA(32, 0, 0, 255)"></bin>
    </ControlPoint>
    <ControlPoint timestamp="2000" volume="100" brightness="100">
      <bin index="40" color="RGBA(32, 0, 0, 255)"></bin>
      <bin index="80" color="RGBA(153, 0, 0, 255)"></bin>
    </ControlPoint>
…
```

3.4 Az asztali alkalmazás felépítése

A szakterületi nyelven leírt vizualizációs logikából generált XML állomány absztrakciós szintje túl magas a LED-ek közvetlen, alacsony szintű vezérléséhez, ezért szükséges egy köztes feldolgozási lépés bevezetése. Az absztrakciós szintbéli különbség áthidalására egy olyan alkalmazás kell, mely egyaránt képes alacsony szintű utasítások végrehajtására, pl.: soros porton való kommunikáció, valamint magas szintű nyelvi elemek kezelésére is, pl.: XML fájl értelmezése. Ennek az alkalmazásnak a megvalósítására modern C++-t választottam, mely a megfelelő könyvtárak felhasználásával össze tudja kötni a hardvert a szakterületi nyelvvel.

3.4.1 Felhasznált könyvtárak

Az alkalmazástól elvárt, hogy fel tudja dolgozni az MPS-el generált XML állományokat, azaz a létrehozott lejátszási listát időzítéshelyesen, a vizualizációs logikának megfelelően végrehajtsa, továbbá a végrehajtás aktuális állapotát a felhasználó felé egy grafikus felületen mutassa. Az XML állományok feldolgozásához a RapidXML-, a zenefájlok metaadatainak értelmezéséhez a TagLib-, a lejátszáshoz és megjelenítéshez az SFML-könyvtárakat használtam fel.

- Simple and Fast Multimedia Library (SFML, https://www.sfml-dev.org/): az SFML egy könnyen felhasználható, nyílt forráskódú multimédia könyvtár. A könyvtár egy közös interfészt biztosít a háttérben lévő OpenGL és OpenAL könyvtárak köré, mely jelentősen megkönnyíti játékok vagy multimédiás alkalmazások fejlesztését. SFML-en keresztül elérhető hálózat- és ablakkezelő modul is.

- TagLib (https://taglib.org/): audio fájlok metaadatainak írására és olvasására szolgál. A könyvtár nyílt forráskódú, objektum orientált szemlélettel, így könnyen felhasználható C++ projektekben. Támogatja az összes elterjedt audio formátumot, így az általam felhasznált FLAC formátum metaadatait is.

- RapidXML (http://rapidxml.sourceforge.net/): XML fájlok értelmezésére szolgáló könyvtár. C++-ban íródott, a fő szempont a bemeneti fájl lehető leggyorsabb feldolgozása, hogy a könyvtár beágyazott rendszereken is használható legyen. A feldolgozási sebesség ára, hogy az absztrakciós szintje meglehetősen alacsony, az XML fájl adatait C-stílusú memóriacímekkel lehet kinyerni.

3.4.2 Az alkalmazás vezérlése

A felhasználók a programot az alábbiak szerint tudják vezérelni:

- a maximális hangerő és fényerő szabályozása
- a spektrumszámítás engedélyezése és a spektrumszámítás eredményének kiküldése a kontroller felé
- a zenelejátszás szüneteltetése és újraindítása, valamint a lejátszási lista következő elemének automatikus lejátszása
- a zenelejátszó előre vagy hátrafelé való tekerése
- a lista következő vagy előző elemére való ugrás

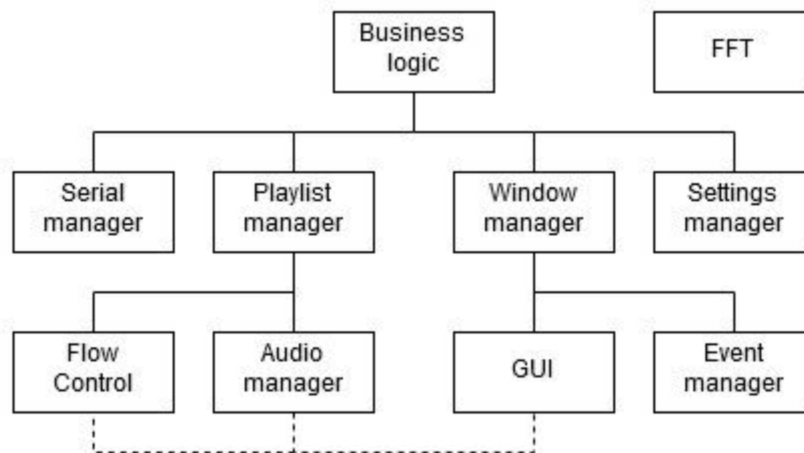
A program jelenlegi verziójában a vezérlés főként billentyűzeten keresztül zajlik, azonban ugyanez a működés később grafikus felületen keresztül is elérhetővé tehető. Látható, hogy a lista elemeit párosával gyűjtöttem össze. Ennek az az oka, hogy igyekeztem a vezérlést a lehető legkevesebb billentyűvel megvalósítani, így a hasonló működéseket azonos gombra helyeztem. A párok közül az adott gomb lenyomásakor minden esetben végrehajtásra kerül valamelyik működés, ezt egy globális módosító billentyű állapota befolyásolja. Ez a kettős működés az egérre is igaz, pl.: az egérgörgő fel-le görgetése szabályozza a maximum hangerőt, amennyiben a módosító billentyű le van nyomva (shift), ugyanaz az egérgörgő a maximális fényerőt módosítja. A vezérlőbillentyűk egyelőre rögzítve vannak a program forráskódjában, azonban a jövőben könnyen kiszervezhetők egy konfigurációs fájlba. A kezelési útmutató megtalálható a mellékletben.

3.4.3 Az alkalmazás felépítése

A programot eredetileg C++20-ban szerettem volna megvalósítani, GCC10 fordítóval, aminek új funkciója a forrásállományok modulokként[5] való kezelése. Sajnos azonban az SFML nem támogatja ezt az új nyelvi szabványt, ezért a fejlesztést C++17-ben folytattam. Ettől függetlenül a program moduláris felépítésű, melyet C++17-ben is meg lehet valósítani, csak más nyelvi elemekkel. Ebben az esetben a program egyes funkcionalitási csoportját megvalósító részegységet tekintem modulnak. Minden egyes modul egy osztálynak feleltethető meg, melyekből pontosan egy példány kerül létrehozásra a Singleton tervezési mintát követve.

[5] https://en.cppreference.com/w/cpp/language/modules

Ennek előnye, hogy a modulok között megosztott erőforrások a program futási ideje alatt minden esetben érvényesek, konzisztensek lesznek. Ilyen megosztott erőforrás a zenei mintákat tároló buffer, vagy a jelenleg játszott zene metaadatai. Az alkalmazás moduljai a 15. ábra szerint kapcsolódnak. A folytonos vonalak tartalmazást jelölnek, a szaggatott vonal pedig a modulok közti megosztott erőforrásokat jelöli.



15. ábra: Az értelmező moduljainak kapcsolódása

- Window manager: az ablak élettartamát és a hozzá tartozó rajzolási felületet vezérli.
- Event manager: az ablakban keletkezett eseményeket feldolgozó osztály.
- Settings manager: a program beállításait tároló adatszerkezet interfészét biztosítja.
- Playlist manager: a lejátszási lista metaadatait, bufferelését megvalósító osztály.
- Flow Control: a vizualizációs logika értelmezéséért felelős osztály.
- Audio manager: az aktuális zenefájl lejátszásáért felelős osztály.
- Serial manager: a soros port megnyitását, bezárását és az adatátvitelt vezérli.
- FFT: a spektrumszámításhoz szükséges függvényeket biztosítja. Az előző modulokkal ellentétben nincsen szükség a példányosítására, mivel a tagfüggvényei statikusak.

3.4.4 Ablakkezelés

Az ablakkezelő osztály feladata a program inicializációjakor az ablak létrehozása, valamint a program végét jelző feltételek teljesülésekor az ablak bezárása. Az ablak mérete a program beállításait tároló konfigurációs fájlból kerül beolvasásra (settings.ini), az ablak címe pedig a parancssori argumentumként kapott lejátszási lista nevéből határozható meg.

SFML-ben a rajzolható objektumok általában egy ablakban kerülnek megjelenítésre. A könyvtár ezt úgy valósítja meg, hogy a rajzolás az ablak objektum egyik tagfüggvényén keresztül érhető el, mely paraméterként a rajzolandó objektumot várja. Ebből következik, hogy minden egyes grafikus elem megjelenítésekor rendelkezésre kell állnia a rajzolás célpontját képző ablak példányának. Ahhoz, hogy az egyes modulok működése jól behatárolható maradjon, érdemes a grafikus felület megvalósítását egy külön osztályba szervezni, melyet az ablakkezelő tartalmaz. A grafikus felület megvalósítása is további elemekre bontható, pl. külön osztály valósítja meg a spektrum megjelenítését, és a zene adatainak kiírását. Ennek hatékony implementálására bevezettem egy ősosztályt, mely a leszármazottjaitól elvár minden működést, ami a megjelenítéshez szükséges. Például ilyen működés, hogy a grafikus elemek nem rendelkezhetnek default konstruktorral, minden esetben explicit meg kell adni a rajzolási célt képző ablak memóriacímét[6]. Közös a leszármazottakban az is, hogy rendelkeznek egy a rajzolásért felelős tagfüggvénnyel. Mivel az ablakkezelő osztályból mindösszesen egyetlen példány létezhet, minden leszármazottnál konzisztens a rajzolási célpont.

3.4.4.1 Grafikus felület

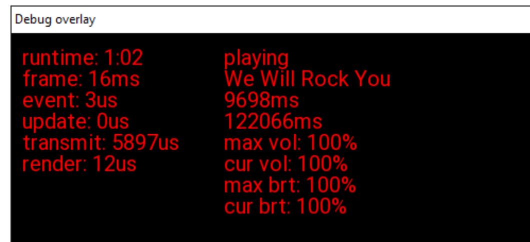
A program felülete az SFML által biztosított alapvető grafikus elemekből épül fel. Ilyen alapvető grafikus elem lehet egy háromszög, téglalap vagy ellipszis. Ezen elemek mellett az SFML könyvtár TrueType szöveg megjelenítésére is képes, amennyiben rendelkezésre áll a betűtípust leíró fájl. Ezek felhasználásával az alábbi ábrán látható letisztult, minimalista interfészt valósítottam meg, melyen kizárólag a zene állapota és a vizualizáció előnézete látszik. A vizualizáció előnézete a zene spektruma és az időfüggő színpaletta kompozíciója.

16. ábra: A program grafikus felülete

[6] C-stílusú memóriacím, későbbi verziókban std::unique_ptr.

3.4.4.2 Debug overlay

A fejlesztés megkönnyítésére létrehoztam egy kiegészítő ablakot (debug overlay), mely igény szerint megnyitható és bezárható a főprogram futásától függetlenül. Ebben a kiegészítő ablakban valós időben megjelenítem a főprogram belső állapotváltozóinak egy részét.



17. ábra: Debug overlay

3.4.5 Eseménykezelés

Az eseménykezelő minden egyes futási ciklusban lekérdezi az ablakkezelőtől, hogy az előző kirajzolt képkocka óta keletkezett-e valamilyen felhasználói esemény. Amennyiben igen, akkor megkezdi az esemény feldolgozását. Az SFML könyvtár az ablak eseményeit egy közös enumerációban tárolja, legyen az egér, billentyűzet, vagy egyéb eseményforrás. Ez alapján az enumeráció alapján bevezetek egy asszociatív tárolót (std::multimap), melynek kulcsa az esemény azonosítója az enumerációból, értéke pedig a megfelelő függvénypointer[7]. Az eseményeket feldolgozó függvények szignatúrája azonos, ezért könnyen kollekcióba szervezhetők. A függvények visszatérés nélküliek, és paraméterként a teljes eseményt leíró adatszerkezetet várják. Az eseményeket leíró adatszerkezetet az ablakkezelő osztály biztosítja az eseményfeldolgozó felé. Az adatszerkezet egyértelműen meghatározza a bekövetkezett esemény típusát, valamint a típus alapján további paramétereket is, pl.: egérgörgetés esemény esetén a görgetés irányát, billentyű lenyomása esetén a lenyomott gomb azonosítóját. Az eseménykezelő függvénypointerekkel való feltöltése a program inicializációjakor, az eseménykezelő konstruktorában történik meg. A tároló az események típusa szerint van rendezve.

Az alkalmazás vezérlése (3.4.2) fejezetnek megfelelően minden eseményhez kettő függvénypointert is hozzárendelek. Az esemény bekövetkeztekor a módosító billentyű állapota határozza meg, hogy a kettő pointer közül melyikhez kerül a vezérlés. A programban az egér

[7] Modern C++ stílusban std::function példánya, melyet std::bind-al kötök egy adott objektum adott tagfüggvényéhez.

mozgatása és a kattintások, billentyűk lenyomása, valamint az ablak paramétereit befolyásoló események kerülnek regisztrálásra, pl.: ablak átméretezése. Egy esemény bekövetkeztekor az eseménykezelő kiszűri a regisztrált események közül azokat, melyekkel a felhasználói esemény típusa megegyezik. Ezután az összes kiszűrt eseménykezelő függvény egymás után lefut. A függvényeknek saját maguknak kell rendelkezni arról, hogy az eseményt leíró adatszerkezet függvényében milyen formában futnak le, pl.: a zenét megállító függvény csak akkor fut végig, ha az ehhez hozzárendelt gomb került lenyomásra. Azért választottam ezt a megoldást, mert így az egy eseményhez hozzárendelt működések száma nem korlátozott, azonos eseményre akár több modul is reagálhat. Megközelítőleg 30 esemény kerül regisztrálásra a konstruktorban. 30 eseménynél az eseménykezelő futási ideje mikroszekundum nagyságrendbe esik, amennyiben nem számítjuk bele az eseményt ténylegesen feldolgozó függvények futási idejét. Ezt is beleszámolva, a leghosszabb ideig tartó teljes eseményfeldolgozási ciklus 2 ezredmásodperc alatt fut le, amely az ablak átméretezésekor következik be. A teljes eseményfeldolgozást az alábbi függvény hajtja végre:

```
void vs::man::event::exec(const sf::Event event) {
    std::pair <emap::iterator, emap::iterator> match;
    match = eventmap.equal_range(event.type);

    for (emap::iterator i = match.first; i != match.second; ++i) {
        fpair f = i->second;

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::LShift)) {
            f.second(event);
        } else {
            f.first(event);
        }
    }
}
```

A függvény működése:

1. Paraméterként adott az eseményt leíró típus. A típust az SFML könyvtár biztosítja, magát a példányt pedig az ablakkezelő modul.
2. Az eseménykezelő tárolójából a kulcs szerint kiválasztom azt a tartományt, ahol a tárolt események típusa megegyezik a paraméterként kapott leíró adatszerkezetben tárolt típussal.
3. Végigiterálok a kijelölt tartományon, majd az iterátor segítségével kimásolom a tárolóból a függvénypointerekből álló párt.

4. A módosító billentyű állapotától függően meghívom az első vagy második függvényt.

3.4.6 A beállítások kezelése

Az alkalmazás számos beállítását újrafordítás nélkül módosíthatjuk. Ezek a beállítások egy konfigurációs fájlból kerülnek beolvasásra. A konfigurációs fájl felépítése megfelel az inicializációs fájloktól elvárt, szekciókba rendezett kulcs-érték pároknak. Az inicializációs fájl feldolgozását egy általam korábban írt könyvtár végzi, mely az iniparser[8] névre hallgat. A könyvtár a szöveges állományt egyszeri átfutással egy fésűs lista adatszerkezetbe helyezi, melyből a szekció és kulcs azonosító alapján lekérdezhetők a tárolt értékek. A könyvtár az értékeket szöveges formában tárolja el. A settings manager osztály egy absztrakciós réteget biztosít az asztali alkalmazás számára, így maga a szöveges állomány és annak struktúrája elrejtésre kerül. A settings manager egy statikus, publikus asszociatív tárolót biztosít (std::map), mely az osztály példányosításakor feltöltésre kerül az inicializációs fájlból kiolvasható értékekkel. A beállítások listáját egy C stílusú enumerációban gyűjtöttem össze, ez képzi az asszociatív tároló kulcsait. Az inicializációs fájlból a szöveges értékeket leképzem az asszociatív tároló egyes értékeire, szintén szöveges formában. A tároló feltöltése után az inicializációs fájlt bezárom, mivel legközelebb a program újraindításakor van rá szükség. A tárolóból az értékeket a felhasználásukkor a megfelelő típusra kell konvertálni, pl.: egész számmá, vagy logikai értékké. A konverzió kiküszöbölhető a C++17-ben bevezetett std::any típus felhasználásával, melyhez nem csak a beolvasott kulcs értékét, hanem típusát is hozzá lehet rendelni, erre azonban a dolgozat készítéséig nem jutott időm. Az alábbi kódrészlet a settings manager felhasználását mutatja, az ablak létrehozásakor. Az x és y változókba konverzió után bekerül az asszociatív tárolóból az ablak szélessége (*WINWIDTH*) és az ablak magassága (*WINHEIGHT*) kulcsokhoz rendelt érték.

```
unsigned x = std::stoi(settings::smap[settings::WINWIDTH]);
unsigned y = std::stoi(settings::smap[settings::WINHEIGHT]);

window.create(sf::VideoMode(x, y), header, sf::Style::Fullscreen);
```

3.4.7 Az XML állomány értelmezése

Az MPS-el generált XML állomány tartalmazza a lejátszandó zenék listáját és a hozzájuk tartozó vizualizációs logikát. A lejátszási listához szükséges a zenefájlok elérési

[8] https://github.com/akosbalogh01/iniparser

útjának és formátumának megadása, a vizualizációs logikához pedig szükséges az elérhető zenék egyes időpillanataiban a spektrum színpalettájának meghatározása.

3.4.7.1 A lejátszási lista kezelése

A lejátszási listák legalább egy elemből állnak, de tetszőlegesen hosszúak lehetnek. Amennyiben az alkalmazás indításakor nem adjuk meg a lejátszási listát tartalmazó fájl címét, vagy a fájl formátuma nem helyes, az alkalmazás hibaüzenettel leáll. Az alkalmazás debug módban elindítva a parancssorra továbbítja a program moduljainak állapotát. Az alábbi ábrán a listát alkotó zenei fájlok ellenőrzésének kimenete látható. Amennyiben a fájl létezik, és érvényes metaadatokkal rendelkezik, a program ezeket a metaadatokat kiírja a parancssorra. Látható, hogy a lista harmadik eleme nem található. Ilyenkor az alkalmazás nem áll le, viszont a hiányzó zenefájlhoz tartozó vizualizációs logikát figyelmen kívül hagyja.

```
Parsed song: (Rock) Queen, News of the World: We Will Rock You
Parsed song: (Rock) Queen, News of the World: We Are The Champions
Not found: D:\Music\Queen\News of the World\03 - Sheer_heart_attack.flac
Parsed song: (Rock) Queen, News of the World: All Dead, All Dead
Parsed song: (Rock) Queen, News of the World: Spread Your Wings
```

18. ábra: A zenefájlok ellenőrzése

Miután ellenőrzésre került mindegyik zenefájl, a lejátszási lista első elemei betöltésre kerülnek a memóriába. Az egyidejűleg bufferelt zenék száma a konfigurációs fájlban módosítható, alapértelmezetten három. A többszörös bufferelésnek az az előnye, hogy számváltáskor szünet, ill. megakadás nélkül folytatódhat a lejátszás. Ideális esetben a teljes lejátszási lista egyidejűleg bufferelhető lenne, azonban ez driver szintű limitációk miatt nem lehetséges. A bufferelést és a lejátszást az SFML könyvtár hajtja végre, amely a bufferelt zenéket azonnal az OpenAL driver audio bufferébe továbbítja. Ennek az SFML-en keresztül elérhető maximális mérete 256MB, melyet egy teljes lejátszási lista könnyen meghaladhat. Mivel a zenefájlok mérete jelentősen eltérhet, megfelelő biztonsági tartalékot figyelembe véve három fájl bufferelését állítottam be alapértelmezettnek.

A lejátszás a felhasználó vezérlése alapján szüneteltethető és folytatható, előre és hátra tekerhető, valamint a listában való előre és hátra ugrás is elérhető. Amennyiben lejátszás közben engedélyezve van a vizualizáció, akkor a program automatikusan kiszámolja a zene spektrumát, ráilleszti a szakterületi nyelven leírt program alapján az időfüggő színpalettát, és mindezt a grafikus felületen megjeleníti.

3.4.7.2 A vizualizációs logika értelmezése

A teljes lejátszási lista vizualizációja a listában lévő zeneszámok vizualizációs logikájából áll. Az egyes zeneszámok vizualizációja egyértelműen meghatározható, ha bizonyos időpillanatokban ismerjük a spektrum színpalettáját. Az ilyen időpillanathoz rendelt színpalettákat vezérlési pontoknak tekintem. Egy vezérlési pontban az időt ezredmásodperc pontosan adhatjuk meg. A színpaletta megadása a spektrum egyes oszlopaihoz rendelt szín egyértelmű meghatározását jelenti. A szakterületi nyelv lehetőséget biztosít arra, hogy az oszlopok színét egyesével megadjuk, azonban ez nem szükséges. A színpaletta meghatározható úgy is, hogy csak némelyik oszlop színét rögzítjük. Ilyenkor a rögzített oszlopok között a színeket lineáris interpolációval kapjuk.

19. ábra: Egy vezérlési pont színpalettája

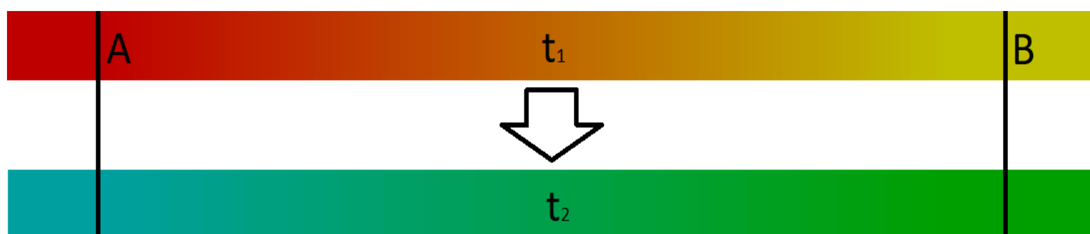
A 19. ábra egy vezérlési pont színpalettáját mutatja. A vezérlési pontban két oszlop színe került megadásra. Az oszlopok helyét az ábrán az A és B jelzések mutatják. A színpalettán az első rögzített oszlop előtti, valamint az utolsó rögzített oszlop utáni színek megegyeznek a legközelebbi rögzített oszlop színével, pl. az A-tól balra lévő oszlopok színe megegyezik az A oszlop színével. Az A és B rögzített oszlopok között pedig a lineáris interpoláció működése látható, ami a rögzített színek között folytonos átmenetet eredményez. Az előbbi ábra palettáját az alábbi XML kódrészlet valósítja meg:

```
<ControlPoint timestamp="170000" volume="85" brightness="75">
    <bin index="10" color="RGBA(255, 0, 0, 255)"></bin>
    <bin index="110" color="RGBA(255, 255, 0, 255)"></bin>
</ControlPoint>
```

A kódrészletben látható, hogy a vezérlési ponthoz rendelt időpontot a timestamp XML tag tárolja. A vezérlési pontokhoz hozzá van rendelve még a lejátszás hangereje, valamint a globális fényerő is, százalékban értelmezve. A különböző vezérlési pontokhoz rendelt hangerő és fényerő értékek között is folytonos az átmenet, ezek is interpolálásra kerülnek. A spektrum oszlopainak színét az index és color tag-ek tárolják. Látható, hogy a tárolt színek formátuma RGBA, azaz az RGB adatok mellé még egy átlátszósági (alfa) komponens is rögzítésre kerül. A program jelenlegi verziójában az átlátszóságot nem használom fel, azonban az SFML könyvtár natívan támogatja, így érdemesnek láttam a DSL-t is erre felkészíteni. Az MPS projekten belül az átlátszósági paraméter a szín koncepciókhoz van rendelve, azonban a

szerkesztő modulon keresztül nem elérhető, így nem is módosítható a vizualizációs logika létrehozása közben. Mivel nem módosítható, minden esetben az alapértelmezett érték kerül az XML fájlba, amely a teljesen átlátszatlan 255-ös értéket jelenti. Az XML-ből kiolvasott RGBA kódot egy reguláris kifejezésként értelmezve könnyen kiolvashatók az egyes komponenseknek megfelelő bájtok értékei.

Az alábbi ábrán két eltérő vezérlési pont közötti átmenet látható. A két vezérlési pontban közös, hogy a spektrumnak az A és B szakaszokkal jelölt oszlopainak a színét rögzíti, azonban ezek a színek eltérőek. A két vezérlési pont közötti időablakban az egyes oszlopok színe szintén lineáris interpolációval számítható.



20. ábra: Két vezérlési pont közti színátmenet

A vizualizációs logika egyértelmű meghatározásához elegendő a vezérlési pontok megadása, így az MPS által generált XML fájlba is csak a vezérlési pontok kerülnek bele. Az oszlopok közti interpolációt, valamint a vezérlési pontok közötti interpolációt mind az értelmező végzi. Az értelmező tárolja a vezérlési pontokat tároló adatszerkezetet is. Mivel az egy zenéhez tartozó vezérlési pontok száma csak futásidőben derül ki, a vezérlési pontok tárolására egy dinamikus méretű tömböt használok (std::vector). A vektor egyes elemei tartalmazzák a színpaletták rögzített oszlopait, valamint a vezérlési pontokhoz rendelt időpillanatokat is. A lejátszott zene kezdetétől eltelt idő alapján ki lehet választani, hogy éppen melyik két vezérlési pont között tart a lejátszás, és ez alapján pedig elvégezhető a színpaletta interpolációja a vezérlési pontok között. Az interpoláció végeredménye egy megosztott erőforrás az értelmező, a grafikus, valamint a soros portot vezérlő modulok között, így a számított színek rendelkezésre állnak az ablakban való megjelenítéshez és a LED-ek felé való továbbításhoz is.

3.4.7.3 Hibakezelés

Mivel az XML fájlt MPS-el állítom elő, szintaktikai hibára nincsen lehetőség. Kizárólag akkor fordulhat elő szintaktikai hiba, ha a felhasználó kézzel módosítja a program bemenetét képző fájlt, ami specifikáción kívüli felhasználást jelent. Amennyiben a felhasználó mégis

kézzel módosítja a fájlt, és szintaktikai hibát vét, az XML fájl beolvasásakor a RapidXML könyvtár kivételt generál, melynek hatására a program leáll.

Szemantikai hiba abban az esetben fordulhat elő, ha a felhasználó az MPS-en belül a vezérlési pontok listájában későbbi vezérlési ponthoz korábbi időpontot rendel hozzá. Mivel a kódgenerálás előtt a kollekció elemei nem kerülnek időpont szerint rendezésre, a végeredményül kapott kód hibás lehet. Az ilyen hibák az értelmező szintjén kerülnek detektálásra. Ilyenkor a hibás időpillanat utáni vizualizációs logikát érvénytelennek tekintem, és a színpalettát konstans fehérre állítom, a zeneszám végéig.

Az MPS-en belül jelenleg nincs megkötés arra, hogy a felhasználó azonos időpillanathoz több vezérlési pontot rendeljen, vagy hogy egy oszlophoz több színt is rendeljen. Az értelmező az azonos indexel rendelkező vezérlési pontok és oszlopok közül az elsőt veszi figyelembe. A DSL következő verzióiban érdemes lehet kiegészíteni a nyelvtant olyan megkötésekkel, amik az azonos időpillanatokat vagy oszlopindexeket nem teszik lehetővé.

3.4.8 A zenék lejátszása

A zenefájlok lejátszását az SFML könyvtárral valósítom meg. A könyvtár számos formátumot támogat, többek között a Vorbis és FLAC formátumokat is, sajnálatos módon viszont nem támogatja MP3 zenefájlok közvetlen lejátszását. Támogatott formátum esetén a könyvtár a zenefájlból elő tud állítani egy audio buffert, amiben a zene hullámformáinak mintái találhatók. A zenefájlok tárolják még a minták lejátszásához szükséges adatokat, pl. a csatornák számát, vagy az egy másodpercre eső minták számát.

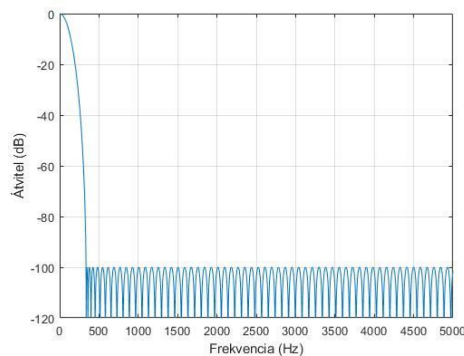
Az SFML képes lejátszani audio fájlokat, azonban nincsen felkészítve a zenék metaadatainak beolvasására. Ennek a megvalósítására a TagLib könyvtárat használtam fel. Erre a két könyvtárra építve létrehoztam egy saját Music osztályt, amely egyaránt képes tárolni a zenei mintákat és a zenék metaadatait is. A lejátszási lista ennek az osztálynak a példányaiból áll. A zenei minták a példányok audio bufferében nem állnak mindig rendelkezésre. A bufferek élettartamát a példányokat tartalmazó lejátszási lista modul vezérli, amely a Music osztály metódusaival vezérli a bufferek példányba való beolvasását és törlését. A lejátszási listát vezérlő modul garantálja, hogy a lejátszani kívánt zeneszám buffere rendelkezésre álljon a lejátszás pillanatában. Az aktuálisan lejátszott zenét reprezentáló osztály egy megosztott erőforrás az alkalmazás moduljai között (std::shared_ptr), így a spektrumszámításért felelős modul hozzáfér az audio mintákhoz, az értelmező hozzáfér a lejátszás aktuális állapotához, a grafikus modul pedig hozzáfér a zenei metaadatokhoz.

3.4.8.1 A spektrum előállítása

A vizualizáció tényleges megvalósításához szükséges az aktuálisan játszott zene spektrumának előállítása. A fejlesztés során felmerült több alternatíva is a spektrum előállítására, pl.: diszkrét Fourier-transzformáció (DFT) [6], Wavelet-transzformáció [7], vagy rezonátoros Fourier-analizátor [8]. A kliensalkalmazás jelenlegi verziójában a spektrumot DFT-vel állítom elő.

A spektrumon ábrázolható maximális frekvencia a zene mintavételi frekvenciájának a fele. Általában a zenék mintavételi frekvenciája 44100Hz, melyből a maximális ábrázolható frekvencia 22050Hz-re adódik. A spektrumot 512 lineáris frekvenciamenetű sávra osztottam, melyből a DFT szimmetriájából adódóan 256 sáv tartalmaz hasznos információt. 256 frekvenciasáv esetén egy sáv szélessége megközelítőleg 86Hz.

Kezdetben a spektrumot a DFT definíciója szerint állítottam elő, azonban ez meglehetősen lassúnak bizonyult, ugyanis a számításhoz szükséges idő a 10ms nagyságrendbe esett. Alternatívaként felmerült az FFTW könyvtár [9] használata, mely a kevésbé számításigényes Fast Fourier Transform algoritmust alkalmazza. Az algoritmus a könyvtártól függetlenül is felhasználható, melynek egy lehetséges implementációja a lábjegyzetben mellékelt linken[9] található. Ezen implementáció felhasználásával a spektrum számításához szükséges idő pár ezredmásodpercre adódik, mely belefér a program képfrissítése által szabott időkorlátnak, így nem volt szükség az FFTW könyvtárra. Mivel a spektrum számítását véges mintaregisztrátumon végezzük, a spektrum csúcsai szivárogni fognak. A szivárgás minimalizálására érdemes a feldolgozott mintákat ablakozni.

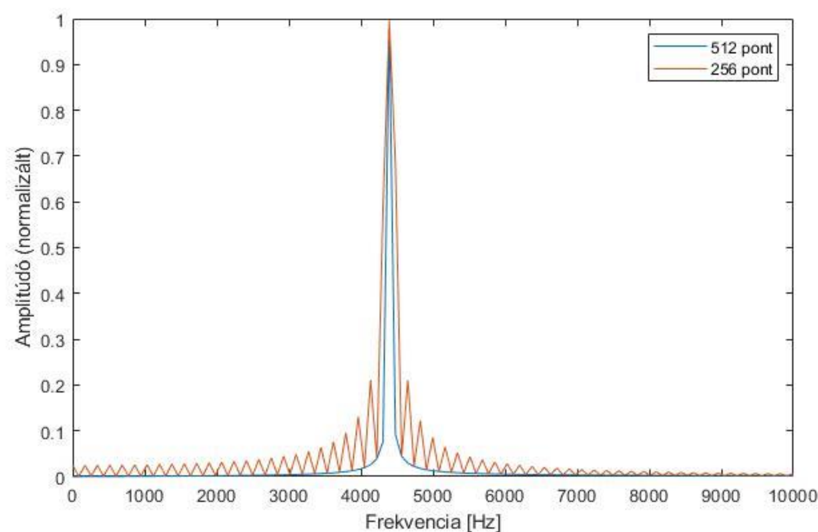


21. ábra: Chebysev-ablak

[9] https://www.geeksforgeeks.org/iterative-fast-fourier-transformation-polynomial-multiplication/

A 21. ábra a választott ablakozó függvény átvitelét mutatja. A fejlesztés során számos ablakozó függvényt megvizsgáltam, melyek közül a Chebysev-ablak bizonyult a legjobbnak a szivárgás minimalizálása szempontjából. Az ablakot a Matlab chebwin függvényével állítottam elő. Az ablak vágása 86Hz-en megközelítőleg -10dB azaz a szomszédos sávokba átszivárgó spektrális komponenseket 10dB-el csökkenti. 172Hz-en, azaz két sávval arrébb a levágás -40dB-re adódik.

Végeredményét tekintve a spektrumot LED-ek fényerejének modulálására használom fel. Az általam választott szalagokon 128 LED található, ezért a 256 hasznos frekvenciasáv közül csak minden második amplitúdóérték kerül felhasználásra. Célszerűnek tűnhet a spektrumot eredendően 256 sávra osztani, melynek szimmetriája miatt 128 amplitúdóérték lenne hasznos. A 128 frekvenciasáv egybeesik a LED-ek számával, így mindegyik felhasználásra kerülne, ezzel számítási kapacitást spórolva. A két eljárást hasonlítja össze a 22. ábra, melyen egyetlen szinuszjel spektruma látható. A sárga diagram az eredendően 256 sávra számolt spektrumot ábrázolja, a kék diagram pedig az 512 sávra számolt spektrumot mutatja, ha a végeredményből minden második amplitúdóértéket elhagyjuk. A kapott adatszerkezet mindkét esetben 128 mintából áll, azonban a kék diagram kedvezőbb spektrális tulajdonságokat mutat, az elhagyott értékek ellenére is. A kedvezőbb spektrális tulajdonságok abból adódnak, hogy 512 sávhoz 512 zenei mintát veszünk figyelembe, amely egy nagyobb időszeletet jelent, így pontosabb képet kapunk a spektrális komponensekről. 512 minta esetén a Chebysev-ablak vágási frekvenciája is alacsonyabb, így a szomszédos sávok közti szivárgást jobban csillapítja.



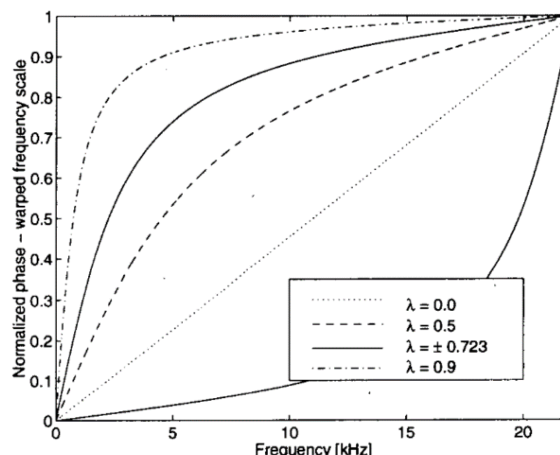
22. ábra: Mintaszámok összehasonlítása

3.4.8.2 A spektrum torzítása

Az emberi hallás sajátossága, hogy az alacsony frekvenciájú hangok között kisebb frekvenciabeli eltérést is jelentősnek érzékelünk, míg magasabb frekvenciákon ugyanaz a frekvenciabeli különbség nem jelent ugyanakkora különbséget az érzékelt hangmagasságban. Mivel az alacsony frekvenciákat jobban tudjuk diszkriminálni, általában a zenei hangok alapharmonikusai is alacsony frekvenciájúak, továbbá felharmonikusai is alig haladják meg a pár kHz-es frekvenciát.

A Fourier-transzformáció által előállított spektrum frekvenciamenete lineáris, így a zenei hangok alacsony és magas frekvencián is azonos szélességű frekvenciasávokban kerülnek csoportosításra. Ennek hátránya, hogy alacsony frekvenciákon egy frekvenciasávon belüli hangok között is különbséget tudunk tenni, azonban ez a spektrumon nem jelenik meg. Ennek az ellenkezője is igaz, miszerint magas frekvenciákon a frekvenciasávok közti váltásokat is alig tudjuk megkülönböztetni, annak ellenére, hogy a spektrumon könnyen észrevehető eltérések tapasztalhatok. Erre az egyik lehetséges megoldás a korábban említett Wavelet-transzformáció, amivel logaritmikus frekvenciamenetű spektrumot állíthatunk elő, vagy amennyiben a lineáris frekvenciamenethez ragaszkodunk, a spektrum torzítása [10].

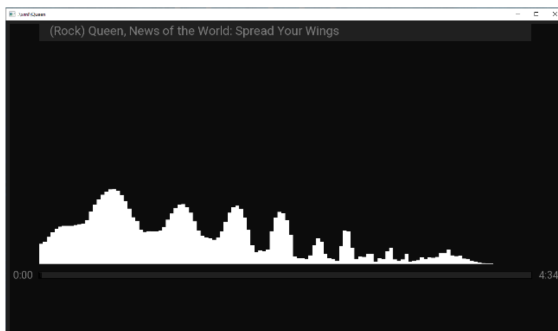
A spektrum torzítása megvalósítható úgy, hogy a zenei mintákon egyfajta digitális szűrést alkalmazunk a DFT elvégzése előtt. A szűrés lényegében a minták időtartománybeli késleltetése a frekvencia függvényében. Általános esetben a szűrés all-pass filterek sorozatával végezhető el. Az all-pass filterek átviteli függvényének amplitúdója egység nagyságú, így a zenei minták amplitúdóján nem változtatnak, azonban rendelkezhetnek frekvenciafüggő fázistolással. A gyakorlatban elegendő egyetlen elsőfokú all-pass filter alkalmazása, melynek a fázistolása egyetlen λ paraméterrel jellemezhető.



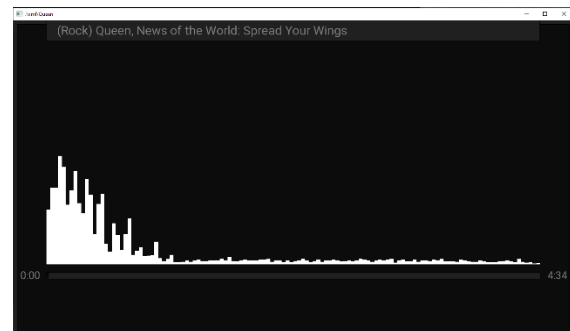
23. ábra: Fázistolás a frekvencia függvényében

Az előbbi ábrán a fázistolás került ábrázolásra a frekvencia függvényében, több λ paraméter esetén is. Amennyiben a λ-t zérusnak választjuk, a frekvenciamenet lineáris, így a szűrés után visszakapjuk az eredeti mintaregisztrátumot. Kitüntetett szerepe van a $\lambda = 0.723$ értéknek. Empirikus mérések alapján belátható, hogy a $\lambda = 0.723$ érték mellett illeszkedik legjobban a torzított spektrum az emberek által érzékelt hangmagasságbeli különbségekhez. A torzítás implementálásakor a λ értékét 0.7-re kerekítettem.

Az alábbi ábrákon látható a spektrum torzításának végeredménye az asztali alkalmazáson belül. A két kép ugyanazon zenének ugyanazon időpillanatában készült. Látható, hogy torzított esetben az egyes felharmonikusok csúcsai egymástól távolabb, valamint a magasabb frekvenciasávok felé eltolva jelennek meg a spektrumban, ezzel látványosabb vizualizációt eredményezve.



24. ábra: Torzított spektrum



25. ábra: Eredeti spektrum

3.4.8.3 Az RGB adatok számítása

A LED-ek RGB adatait az értelmező állítja elő a vizualizációs logika és a spektrum alapján. A vizualizációs logikából előállítható egy időfüggő színpaletta, mely egy adott időpillanatban leírja a szalag összes LED-jének színét. Ez a színpaletta adja a LED-ek RGB adatainak maximális értékét. A maximális értékhez képest a szalagokon az RGB adatok leosztott értékei jelennek meg. RGB kódolás esetén a három színkomponenst azonos értékkel leosztva azonos színárnyalatot kapunk, csak kisebb intenzitással, pontosabban a szalag esetében kisebb fényerővel. A leosztás eredményeként kapott RGB adatok kerülnek felhasználásra a grafikus megjelenítés során is. A maximális fényerő három lépésben kerül leosztásra:

- A felhasználó által beállított maximális fényerő alapján. A felhasználó futásidőben módosíthatja a LED-ek maximális kivezérelhetőségi tartományát egy globális, százalékban értelmezett paraméter beállításával. Ez a leosztás az összes LED-re érvényes.

- A vizualizációs logika által meghatározott leosztás alapján. A vizualizációs logika egyes vezérlési pontjaihoz rendelhetünk egy leosztást, szintén százalékban értelmezve. Ez a leosztás szintén változhat futásidőben, a vizualizációs programtól függően, viszont a felhasználó nem tudja befolyásolni. Ez a leosztás is érvényes az összes LED-re.
- A spektrum egyes frekvenciasávjainak amplitúdója alapján. Ez a leosztás a LED-ek fényerejét külön-külön befolyásolja. A szalagon található összes LED hozzárendelhető a spektrum egyes frekvenciasávjaihoz. DFT esetén meghatározható a spektrum amplitúdójának elméleti maximuma. Az elméleti maximum alapján az egyes frekvenciasávok amplitúdója normálható 0 és 1 közé. A normalizált érték alkalmas az egyes LED-ekhez tartozó RGB adatok újbóli leosztására.

3.4.9 A soros port kezelése

A LED-szalagok RGB adatai a mikrokontroller felé virtuális soros porton keresztül kerülnek kiküldésre. A virtuális soros port azonosítóját az operációs rendszer határozza meg, ezért számítógépek között eltérhet, így nem kódolható bele konstansként az alkalmazásba. A port azonosítója mellett a maximális adatátviteli sebesség is eltérhet számítógépek között. Ennek áthidalására a port azonosítóját és Baud rátáját kihelyeztem a program beállításait tároló konfigurációs fájlba, és a program indulásakor olvasom be.

A soros port vezérlését a Serial manager osztály valósítja meg. Az osztály a konstruktorában megkísérli megnyitni a portot a beállításoknak megfelelően. Az osztály egy belső, logikai típusú állapotváltozóban tárolja, hogy a kommunikációs csatorna nyitva van-e vagy sem. A soros port az osztály destruktorában, és felhasználói vezérlés alapján kerülhet bezárásra. Amennyiben a port nyitva van, a serial manager osztály példányán keresztül a transmit függvény meghívásával lehet bájt tömböket küldeni a mikrokontrollernek.

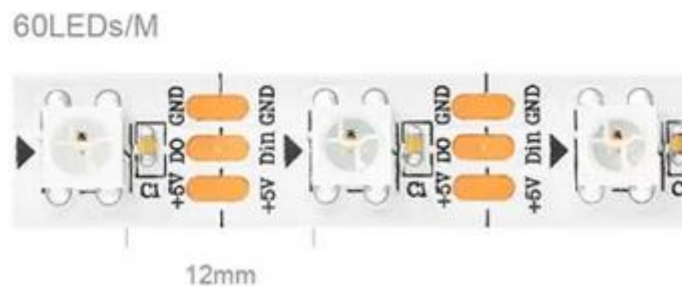
A kliensalkalmazás és a mikrokontroller közti kommunikáció egyirányú. Az alkalmazás minden képfrissítéskor kiküldi a nyers RGB adatok tömbjét. A kiküldés lehet szinkron és aszinkron, azaz az asztali alkalmazás megállítható az átvitel ideje alatt, vagy a küldési parancs kiadása után tovább is futhat. Aszinkron átvitel esetén a kiküldést a hardverközeli driver párhuzamosítja. Aszinkron esetben a maximális küldési időablak 16ms, mely az asztali alkalmazás képfrissítési rátájából számítható. Általában a küldéshez szükséges idő belefér a képfrissítések közötti időablakba, így alapértelmezetten szinkron átvitel mellett döntöttem.

3.5 A beágyazott rendszer felépítése

Az asztali alkalmazás által előállított zenei vizualizáció megjelenítéséért egy beágyazott rendszer felelős. A beágyazott rendszernek elsődlegesen eleget kell tennie a LED szalag specifikációjának, továbbá meg kell felelnie feldolgozási sebesség, áramfelvétel, valamint könnyű felhasználhatóság szempontok szerint is.

3.5.1 LED-szalag

A LED-szalagnak a specifikáció alapján egyénileg vezérelhető, változtatható színű és fényerejű LED-ekből kell állnia. A számomra elérhetők közül a WS2812B típusú LED-sorra esett a választásom [11]. A megvalósítás szempontjából releváns paraméterek a tápfeszültség, LED-enkénti áramfelvétel, valamint a LED adatok továbbításának protokollja.



26. ábra: A LED-szalag egy részlete

A 26. ábra mutatja, hogy szalagon három galvanikusan elválasztott vezetőfelület fut végig, melyből egyik a földelés, másik a tápvonal, a harmadik pedig az adatvezeték. A szalag a tápvonal és a földelés között 5V-os egyenáramú feszültséget vár el. A szalagon a LED-ek kaszkádosítva vannak, azaz sorosan egymás után vannak kapcsolva. A LED-ek egymástól függetlenül vezérelhetők, így az eredetileg 5 méteres szalagot kedvünkre méretre szabhatjuk. A szalagról legkisebb lemetszhető egység egyetlen műanyag tok, mely három fényforrásból, egy vörös, egy kék, valamint egy zöld LED-ből áll. Az egy tokban található LED-ek vezérlése három vezérlőbájt kiküldésével zajlik, mely a kívánt szín RGB kódjának felel meg. A bájtok sorrendje GRB, azaz először a zöld, aztán a piros, majd a csoporthoz tartozó kék LED bájtja kerül kiküldésre. A kaszkádosítás működési elve az, hogy mindegyik csoport felhasználja és levágja a fogadott adatok közül az első három bájtot, a többit pedig továbbítja a következő csoport felé.

3.5.2 Mikrokontroller

A szalag a LED-ek vezérlő bájtjait egy soros adatvezetéken várja. Soros adatvezeték révén, a vezérlésre felhasználható lehet egy USB-soros port átalakító, vagy akár az alaplapi RS-232 portok egyike is. A fejlesztés során kipróbáltam mindkét lehetőséget, azonban mindkét esetben technikai korlátokba ütköztem. Az alaplapomon található soros port baud rate-je nem éri el a közvetlen vezérléshez szükséges átviteli sebességet, a rendelkezésemre álló FT232RL-M USB-soros átalakító [12] pedig nem rendelkezik megfelelő mennyiségű memóriával, hogy egy teljes szalagra vonatkozó keretet egyszerre tárolni tudjon. Ezen problémák áthidalására egy mikrokontroller felhasználása mellett döntöttem.

Rendelkezésemre állt egy STM32 Nucleo F446RE fejlesztőkártya [13], amelynek számítási-, valamint memóriakapacitása megfelelőnek bizonyult a szalag vezérléséhez. A kontrolleren futó beágyazott szoftver ANSI C nyelven íródott. Egy mikrokontroller felhasználása biztonságtechnikailag is kedvező, mert így a LED-szalag és az alaplap között nincsen közvetlen kapcsolat, azaz a hardver bármely nem várt meghibásodása esetén is legfeljebb a mikrokontroller veszik oda. További előny, hogy egy kontroller felhasználásával a szalag vezérlése USB porton keresztül megvalósítható, így az eszköz akár laptopokról is használható.

3.5.2.1 Kommunikáció az asztali alkalmazással

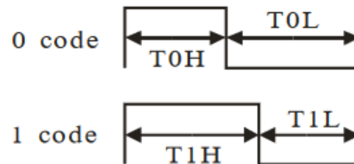
A mikrokontroller az asztali alkalmazással USB-re illesztett virtuális UART (Universal Asynchronous Receiver Transmitter) porton keresztül kommunikál. A kommunikáció keretrendszerét az STM32 HAL (Hardware Abstraction Layer) könyvtára biztosítja.

A kontroller hardver moduljai a HAL könyvtár segítségével könnyen konfigurálhatóak, a megfelelő inicializációs struktúrák létrehozásával, valamint a megfelelő vezérlőparancsok kiadásával. Az UART-ot 1Mbaud átviteli sebességre, egyetlen stop bitre, paritás bit nélkül, 8 bites szóhosszra konfiguráltam, ami megközelítőleg 10kB/s átviteli sebességet eredményez. A saját konfigurációm esetén egy csomag 128 LED RGB adataiból áll, ami összesen 384 bájtot jelent. 10kB/s adatátviteli sebességgel számolva ez megközelítőleg 4 ezredmásodpercet jelent.

A program jelenlegi verziójában az RGB adatok nyersen, keretezés nélkül kerülnek kiküldésre a kontrollernek. Összetettebb szalagkonfigurációk esetén előnyös lehet a csomagot egy kerettel ellátni, amely tartalmazhat hibaellenőrző adatokat, vagy több szalag esetén a szalagok indexét is. Indexelhető szalagokkal megvalósítható lenne az, hogy többcsatornás zene esetén mindegyik csatorna spektruma külön vezérelhetne egy-egy szalagot.

3.5.2.2 Kommunikáció a LED szalaggal

Az RGB bájtok egyes bitjeit névlegesen egy 800kHz-es négyszögjel kitöltési tényezője határozza meg. Amennyiben a négyszögjel kitöltési tényezője 10-30% között van, 0-s bitként, amennyiben 60-90% között van, a szalag a kiküldött adatot 1-es bitként értelmezi.



27. ábra: A feszültségszintek időtartományai

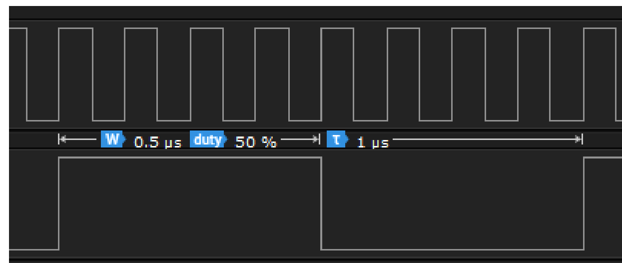
A gyakorlatban a szalag specifikációja nem kitöltési tényezőt, hanem a bemenet magas és alacsony feszültségi szinten töltött idejét határozza meg. A bitek elvárt jelalakjai meghatározhatók a 27. ábra és az alábbi táblázat segítségével, melyben a minimális és maximális időtartamok kerültek feltüntetésre:

T0H	0 bit, magas feszültség időablaka	250ns – 450ns
T0L	0 bit, alacsony feszültség időablaka	700ns – 1000ns
T1H	1 bit, magas feszültség időablaka	650ns – 950s
T1L	1 bit, alacsony feszültség időablaka	250ns – 450ns

Ez nem egy szabványos soros kommunikáció, így a Nucleo nem rendelkezik olyan célhardverrel, ami közvetlenül végre tudná hajtani. Az elvárt négyszögjel manuálisan előállítható hardveres időzítők által generált PWM (Pulse Width Modulation) jellel, GPIO (General Purpose Input Output) lábak vezérlésével, de akár SPI (Serial Peripheral Interface) busz szabványon kívüli felhasználásával is.

A felsoroltak közül az SPI buszra esett a választásom. Az SPI egy soros, szinkron adatátviteli interfész, melynek adatbuszára több periféria is illeszthető. Az adatbusz két adatvezetékből, a MOSI és MISO (Master Output Slave Input és Master Input Slave Output) vezetékből áll, mely kétirányú kommunikációt tesz lehetővé, akár egyidejűleg is. Több periféria esetén a vezérelni kívánt perifériát egy chip select jel kiadásával választhatjuk ki. A szalagok vezérlése megvalósítható a MOSI vezetékkel, valamint az órajel figyelmen kívül hagyásával. A buszon legkisebb küldhető adatmennyiség egy bájt. Megfelelő bitmintázatú bájtok esetén az SPI MOSI adatvonalán előállíthatóak tetszőleges kitöltési tényezőjű négyszögjelek. A

négyszögjelek kitöltési tényezőjét a kiküldött bájtokban található 1-es bitek száma határozza meg. Az alábbi ábrán egy 50%-os kitöltési tényezőjű négyszögjel, valamint a hozzá tartozó órajel látható, melyet a hexadecimális F0 bájt kiküldésével valósítottam meg.



28. ábra: SPI MOSI négyszögjel

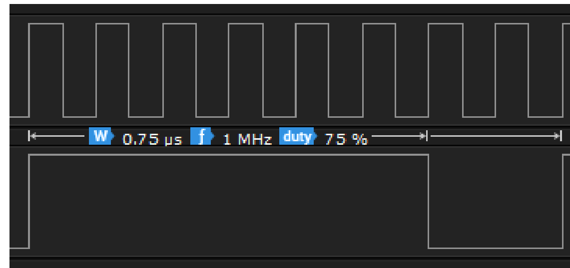
Az adatvezetéken előállítható négyszögjel frekvenciája az SPI busz órajelének frekvenciájától függ. Megfelelő SPI busz frekvencia esetén a keletkező négyszögjel alkalmas lehet LED-szalagok vezérlésére. Az SPI buszon egyetlen órajelciklus alatt egyetlen bit kerül átvitelre, így egyetlen bájt továbbításához nyolc órajelciklusra van szükség. Az adatvezetéken a LED-szalagot vezérlő hullámformát vezérlőbájtok kiküldésével valósítom meg. Mivel egy vezérlőbájt átviteléhez nyolc órajelciklus szükséges, a szalagot vezérlő négyszögjel frekvenciája az SPI busz órajelfrekvenciájának nyolcadára adódik.

A 27. ábra és a mellékelt táblázat alapján a rövid időablaknak (T0H, T1L) 250ns-et, a hosszú időablaknak (T0L, T1H) pedig 750ns-et választottam. Ezeknek a megfelelő összegzése esetén 1us periódusidőt kapunk, amely egy 1MHz-es négyszögjelnek felel meg. Figyelembe véve a nyolcszoros leosztást, ez megvalósítható, ha az SPI busz frekvenciáját 8MHz-nek választom. A Nucleo az SPI busz órajelét a processzor órajelgenerátorának leosztásával állítja elő, ezt az osztót prescalernek nevezzük. Érdemes a processzor órajelének olyan frekvenciát választani, melyből egy osztással pontosan előállítható a kívánt órajelfrekvencia. Ezeket figyelembe véve a processzor órajelét 128MHz-re állítottam, melyből egy 16-szoros prescaler pontosan 8MHz-es SPI frekvenciát állít elő.

8MHz-es órajel esetén egy bitidő 125ns-re adódik, mellyel a rövid időablaknak választott 250ns-ot 2 bit, a hosszú időablaknak választott 750ns-ot 6 megegyező bit kiküldése valósítja meg. Az alábbi ábrán a szalag felé továbbított 1-es bit hullámformája[10] látható, melyet

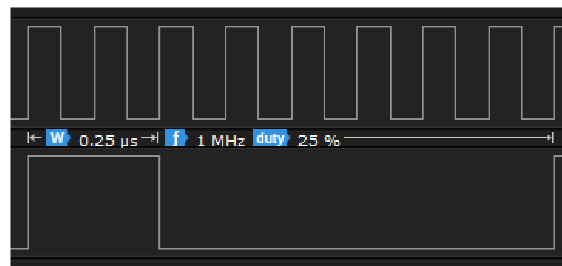
[10] A hullámformákról mellékelt ábrákat egy Saleae logikai analizátorral készítettem. Ennek a mintavételi frekvenciája nem elég nagy a pontos időzítések kimérésére, de a hullámformákat jellegre helyesen mutatja.

a 0b1111`1100 bájt kiküldése valósít meg. Az ábrán feltüntetésre került a négyszögjel frekvenciája és a kitöltési tényezője is, mely valóban egy 750ns-es pulzust valósít meg.



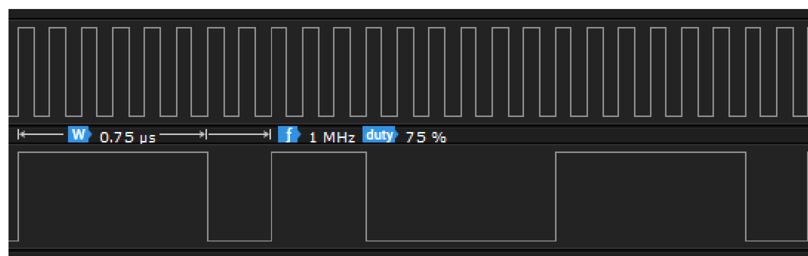
29. ábra. 1-es bit hullámformája

A szalag felé továbbított 0-s bit hullámformáját a 30. ábra mutatja. Az előző ábrához hasonlóan feltüntetésre került a négyszögjel kitöltési tényezője és frekvenciája, valamint a 250ns-es pulzus szélessége is. A 0-s bit hullámformáját a 0b1100`000 bájt kiküldése valósítja meg.



30. ábra: 0-s bit hullámformája

A gyakorlati tapasztalat azt mutatja, hogy a szalag ennél lazább időzítéseket is képes értelmezni. A pontos időzítések mérésekkel meghatározhatók[11]. 0-s bitként értelmezi a 0b1000`0000 és 0b1110`000 bájtokat, valamint 1-es bitként értelmezi a 0b1111`0000 és 0b1111`1110 konverziós bájtokat is. Ennek figyelembevételével a 0-s bit konverziós bájtjának a 0b1100`0000, az 1-es bit konverziós bájtjának a 0b1111`1100 bitmintát választottam.



31. ábra. Eredő kimeneti hullámforma

[11] https://wp.josh.com/2014/05/13/ws2812-neopixels-are-not-so-finicky-once-you-get-to-know-them/

A korábban meghatározott konverziós bájtok felhasználásával előállítható eredő kimeneti hullámformát a 31. ábra mutatja. Az ábrán a felső hullámforma az órajel, az alsó pedig az SPI MOSI adatvonala. Az adatvonalon a szalag felé a 101 bitsorozat került továbbításra. Látható, hogy 1-es bit esetén a kapott négyszögjel kitöltési tényezője 75% körüli érték, 0-s bit esetén pedig ennél jóval kevesebb, 25% körüli, tehát a keletkező hullámforma megfelel a szalag által elvárt 0-s és 1-es bitek jelalakjának.

Az asztali alkalmazástól fogadott RGB adatok ezen konverziós bájtok segítségével átalakíthatók egy olyan vezérlőtömbbé, mely alkalmas a LED szalag egyes csoportjainak vezérlésére. Ez a vezérlőtömb úgy állítható elő, hogy az RGB adatok mindegyik bitjét leképezzük a számolt konverziós bájtok egyikére. Alternatív megoldásként, amennyiben a kontrolleren kevesebb erőforrás áll rendelkezésre, ugyanilyen hullámformák előállíthatók az RGB bitek három SPI bitre való konvertálásával is. Ebben az esetben kisebb a program memóriaigénye, valamint alacsonyabb SPI buszfrekvencia is elegendő, azonban a három bites logikát nehezebb implementálni.

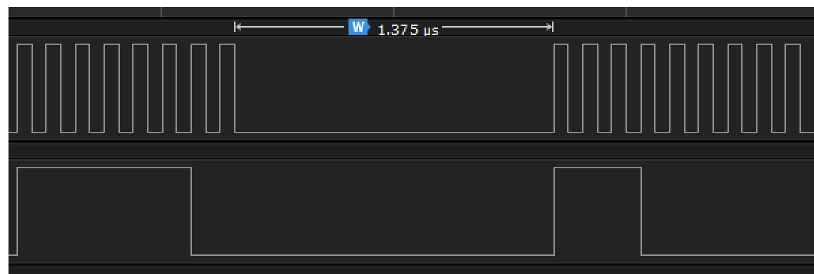
SPI kommunikáció esetén a perifériák közti szinkronizációt az órajel biztosítja. Definiálni kell, hogy az adatbuszon az órajel egyes fázisaiban mikor értelmezhető az adatvezetéken lévő érték. Általában, a legalapvetőbb SPI kommunikáció esetén az órajel felfutó élekor kerül megvizsgálásra az adatvezeték szintje, majd ehhez a szinthez tartozó logikai érték belekerül egy shiftregiszterbe. Ez azt eredményezi, hogy az órajel két magas félperiódusa között akár szünet is lehet, mégis az SPI kommunikáció érvényes marad. Szabványos SPI perifériák között ez az órajelkésleltetés nem okoz gondot, azonban a LED szalag vezérlésénél glitchekhez vezet.

32. ábra. MOSI glitch

A 32. ábra egy órajelkihagyást mutat. A kiküldött adatok ebben az esetben is érvényes SPI kommunikációnak felelnek meg, azonban az órajelkésleltetés miatt az adatvezetéken lévő négyszögjel kitöltési tényezője megváltozik. A szalag felé a 000 bitsorozat került továbbításra, melyet azonban az 010 biteknek értelmezett, a kitöltési tényező változása miatt. Ilyen

órajelkésleltetést a tranzakció alatti megszakítások okozhatnak. Általános esetben a glitchekre a DMA (Direct Media Access) nyújt megoldást. DMA adatátvitel esetén az SPI a processzortól függetlenül eléri az átküldeni kívánt memóriaterületet, így a processzor megszakításai sem okoznak az átvitelben késleltetést.

A probléma kiküszöbölésére a LED-szalag esetében alkalmazható az SPI modul Texas Instruments módja is, az alapértelmezett Motorola mód helyett. A TI mód a szigorúbb időzítések mellett előírja, hogy az adatvonal alapállapota logikai alacsony szinten legyen. A LED-szalag felépítése miatt az adatátvitel bizonyos ideig felfüggeszthető (~50us), amennyiben a bemeneti logikai érték 0. A 33. ábra egy ilyen glitchet illusztrál, TI mód esetén. Látható, hogy az órajel kihagyása alatt az adatvonal végig a 0 szinten van, így a szalag a glitch ellenére is helyesen értelmezi a bitet.



33. ábra: MOSI glitch TI mód esetén

3.5.2.3 Beágyazott szoftver

Ebben a beágyazott rendszerben a mikrokontroller feladata meglehetősen egyszerű, melyet a kontrolleren futó szoftver architektúrája is tükröz. A mikrokontroller feladata az asztali alkalmazás által számított RGB adatok fogadása, átalakítása, majd továbbítása a LED szalagok felé.

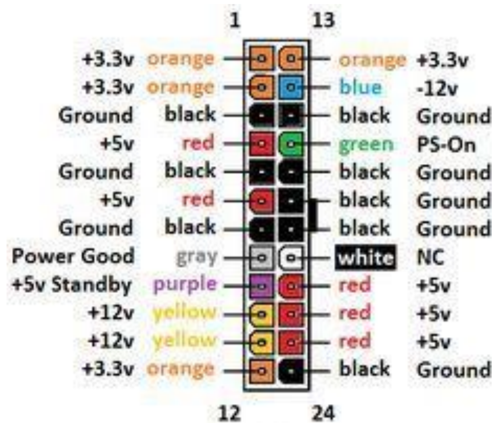
Ennek megvalósítására egy megszakításvezérelt architektúrát alkalmaztam. Mivel előre ismert az egy keretben vezérelt LED-ek száma, ismert az is, hogy egy keretben az asztali alkalmazás hány bájtot küld a mikrokontroller felé. Amikor az UART interfészen keresztül megérkezik a megfelelő mennyiségű bájt, egy megszakítás beállít egy flaget. A program a főciklusában megvizsgálja a flag értékét. Amennyiben beállításra került, az UART adatbufferében rendelkezésre állnak az aktuális RGB adatok. Ezen buffer alapján feltöltök egy másik buffert, melyben az UART buffer minden egyes bitjéhez egy LED szalag konverziósbájtot rendelek. A hozzárendelés végén egy SPI tranzakcióval továbbítom a szalag felé a LED-eket vezérlő adatokat. A továbbítás végén a buffereket nullázom, valamint a flaget alaphelyzetbe állítom, ezzel felkészítve a programot a következő RGB csomag fogadására.

3.5.3 Tápegység

Maximális áramfelvétel a maximális fényerejű fehér fény (RGB: 255, 255, 255) esetén mérhető, ami 70mA-re adódik. A választott szalagon méterenként 60 LED csoport helyezkedik el. Egy 5 méteres szalagon így 300 LED csoport található, melynek maximális áramfelvétele 22A körüli érték. A spektrumszámítás szempontjából kényelmes a kettő valamely hatványára illeszteni a csoportok számát, ami ebben az esetben 256 csoportot jelent. 256 csoport esetén szalag hossza 4,5 méter, maximális áramfelvétele 18.5A. 10%-os biztonsági tartalékkal számolva a tápegység kimeneti áramerőssége legalább 20 Amperes lehet. Figyelembe véve az 5V-os tápfeszültséget, 256 csoport maximális teljesítménye 200W-ra adódik. Ezeknek a paramétereknek eleget tesz egy átlagos asztali számítógép tápegysége, melyeken általában található kellő terhelhetőségű 5V-os sín. Rendelkezésemre állt egy Corsair RM550x [14] tápegység, mely a terhelhetősége mellett passzív hűtésű, így nem zavarja a zenét egy hangos ventilátor.

Tesztelés gyanánt kipróbáltam régebbi, rosszabb minőségű tápegységeket is. A régebbi tápegységeken nagyobb hangsúly volt az 5V-os sín terhelhetőségén, a számítógépekben szintén megtalálható 12V-os és 3.3V-os sínekhez képest. Ez abban mutatkozik, hogy míg a modern Corsair táp 5V-os terhelhetősége 25A, például egy régebbi Chieftec 5V-os terhelhetősége 34.5A. Elméletileg tehát alkalmasabb lehet egy régebbi, korábbi ATX[12] szabvány szerint tervezett tápegység, melynek nagyobb a terhelhetősége. A gyakorlatban azonban az tapasztalható, hogy hirtelen fényerőváltáskor, amikor az áramfogyasztás is hirtelen megugrik, a régebbi tápegységek automatikusan kikapcsolnak. Ennek kompenzálására illesztettem egy-egy kondenzátort a tápvonal és föld közé, közvetlenül a szalagok bemenetére és közvetlenül a tápegység kimeneti csatlakozójára is. Ezek a kondenzátorok szükség esetén, azaz hirtelen fényerőugrásnál képesek töltést leadni, ezzel kiküszöbölve a tápegység leállását. A modern Corsair tápegység kondenzátorok nélkül is tudja kezelni a fényerő ugrásait, a biztonság kedvéért azonban mégis ráforrasztottam őket. A tápfeszültség biztosításához a tápegység 24 pines ATX csatlakozóját használtam fel. A csatlakozó lábainak kiosztása alább látható.

[12] Advanced Technology eXtended, asztali számítógép tápegységeire és alaplapjaira vonatkozó szabvány



34. ábra: ATX 24pin kiosztása

A csatlakozón összeforrasztottam az 5V-os kivezetésként szolgáló 21-22-23-as lábakat. A csatlakozóra illesztett kondenzátor a 23-as és 24-es pinek között helyezkedik el, a LED szalagok és Nucleo földelését a 24-es pinre kötöttem. A tápegység bekapcsolva tartásához rövidre zártam a 16-os engedélyező pint a szomszédos 15-ös pinnel, ezzel leföldelve azt. Amennyiben a 16-os pin, a szabvány szerinti zöld vezeték földre van húzva, a tápegység bekapcsolva marad.

A fejlesztőkártya egy USB csatlakozón keresztül kapcsolódik a számítógéphez. Alapértelmezetten az USB csatlakozó biztosítja a kártya tápellátását is. Ebben az esetben, mivel eltérő a szalag és a Nucleo áramforrása, a LED szalag földelése és a Nucleo SPI buszának földelése között feszültségkülönbség jelenhet meg. Ez a feszültségkülönbség befolyásolja a jelszint komparálási feszültségét, ami a LED vezérlőtömb hibás fogadását eredményezi. Erre megoldás a Nucleo és a LED szalagok földelésének közösítése. A fejlesztőkártya egy jumper áthelyezésével üzemeltethető külső 5V-os tápforrásról. Mivel a LED szalagok szintén 5V-os tápfeszültséget várnak el, kézenfekvő megoldás a Nucleo-t a szalagokkal közös tápvonalra helyezni, ezzel a közös földelést is biztosítva.

4 Az eredmények értékelése

A dolgozatban részletesen bemutatásra került a zenei vizualizáció jelenlegi implementációja, mely a projekt 1.0-ás verziójának tekinthető. A nagy verziószám az értelmező verziójával egyezik meg, a kis verziószámot pedig az értelmezőt nem érintő módosítások adják. Részlegesen implementálásra került az 1.1-es verzió, valamint koncepcionálisan elkészült a projekt 2.0-ás verziója is.

4.1 Áttekintés

Végeredményét tekintve a projekttel sikeresen megvalósítottam zeneszámok vizualizációját. Mivel a vizualizációs élmény erősen szubjektív lehet, ezért nehéz az elkészült alkalmazást felhasználói szempontból objektíven értékelni. A keretrendszer tesztelésére több órányi zenének készítettem el a vizualizációját, melyekkel igyekeztem különböző időtartamú és stílusú lejátszási listákat létrehozni. A saját tapasztalataim szerint a keretrendszer alkalmas zenék széleskörű vizualizációjára, ráadásul kellemes élményt is nyújt hangulatvilágítás szempontjából.

A keretrendszer széleskörű tesztelését hallgatótársaim segítségével végeztem, akik a saját ízlésük szerint számos lejátszási listát hoztak létre. A visszajelzések pozitívok voltak, ami a projekt sikerét mutatja. Ugyan a hallgatótársaim visszajelzése is szubjektív, mégis a segítségükkel számos fejlesztési lehetőséget sikerült összegyűjteni, melyeket az alábbi fejezetekben foglaltam össze. A fejlesztési területeket a projekt fő irányvonalai szerint csoportosítottam, azaz a szakterületi nyelv, a kliensalkalmazás, valamint a hardver mentén.

4.2 A szakterületi keretrendszer értélése

4.2.1 Az MPS keretrendszer értékelése

A szakterületi nyelv elkészítéséhez az MPS jó választásnak bizonyult. A dolgozat keretein belül csak azok a részei kerültek bemutatásra, melyek elengedhetetlenek voltak a VSR nyelv dokumentálásához, azonban a keretrendszer ennél jóval összetettebb, mind a megvalósítható nyelvi elemek, mind a szerkesztő, mind a kódgenerálás vonatkozásában.

A fejlesztés során az MPS alternatívájaként felmerült, hogy az értelmező bemenetét képző XML állományt XSD (XML Schema Definition) segítségével állítsam elő. XSD alapján a nyelvhez rendelhető lenne akár egy grafikus szerkesztő is, amivel a vizualizációs

dokumentumot az MPS projekcióalapú szerkesztőjéhez hasonlóan hozhatnánk létre. Az XSD-vel való megvalósításhoz képest az MPS számos előnnyel rendelkezik:

- Összetettebb validációs lépések: a generálási folyamat kiegészíthető statikus kódellenőrző scriptekkel.
- Összetettebb szerkesztő: a projekcióalapú szerkesztő összetettebb típusok szerkesztését is lehetővé teszi, a szerkesztést magát mindenhol automatikus kódkiegészítéssel segíti, valamint a szerkesztő interaktív grafikus elemeket is tartalmazhat.
- Absztrakciós szint: a projekcióalapú szerkesztő egy absztrakciós szintet biztosít a generált XML állományhoz képest. Amennyiben a jövőben a vizualizáció fájlformátumát cserélni kell, pl. szubrutinok leírása XML-ben nehezen valósítható meg, akkor ezt a szerkesztőfelület módosítása nélkül megtehetjük.

4.2.2 A szakterületi nyelv értékelése

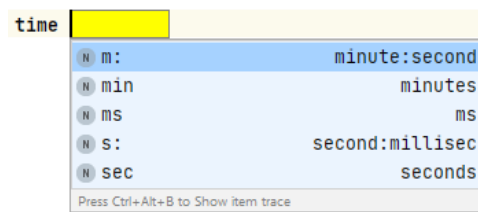
A szakterületi nyelv 1.0-ás verziójának megvalósítása a 3.3 fejezetben olvasható. A keretrendszer tesztelése során számos fejlesztési lehetőség merült fel. A jelenlegi verzió fő problémája a vezérlési pontok leírásának skálázhatósága. Hosszabb zeneszámok esetén a vezérlési pontok száma könnyen elérheti a százas nagyságrendet, mely nehezen átláthatóvá válhat. Erre megoldást nyújthat a nyelvtan kiegészítése összetettebb nyelvi elemekkel. Ilyen nyelvi elemek lehetnek összetett adatszerkezetek (v1.1), vagy összetett vezérlési utasítások (v2.0).

4.2.2.1 VSR 1.1

A nyelv kiegészíthető összetett adatszerkezetekkel, úgy, hogy a generált XML formátuma nem változik, így az értelmező jelenlegi verziója is képes feldolgozni. Az összetett adatszerkezeteket a nyelv 1.1-es verziójában valósítottam meg. Az összetett adatszerkezetek lehetővé teszik az időpontok és színek flexibilisebb megadását, azonban bonyolultabbá teszik a generálási logikat. A dolgozat elkészítéséig sajnos a generálási logikát nem tudtam megvalósítani, azonban a létrehozott adatszerkezeteket alább bemutatom.

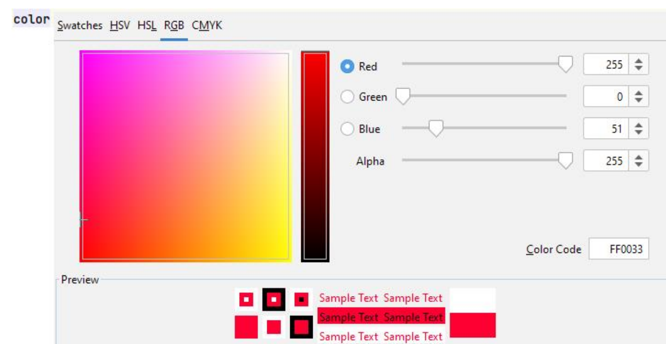
Az egyes adatszerkezeteknek létrehoztam egy-egy külön nyelvet (Time, Color), melyet később a fő VSR nyelvbe importálva használok fel. A nyelvek modulokként való kezelésének előnye, hogy mind az importált nyelvet később tetszőlegesen módosíthatjuk és kiegészíthetjük, a fő VSR nyelv változtatása nélkül. A Time nyelv számos időformátumot támogat, ezeket a 35.

ábra mutatja. A nyelv bevezet két operátort is (@, +), mellyel az egyes vezérlési pontok időpontja abszolút módon és az előzőhez képest relatív módon is megadható, ezzel jelentősen kezelhetőbbé téve hosszabb zenék leírását.



35. ábra: Támogatott időformátumok

A Color nyelv számos színformátumot támogat, többek között RGB, HEX, valamint CMYK formátumokat is. Az MPS szerkesztőjében rejlő lehetőségeket kiaknázva létrehoztam egy interaktív grafikus felületet a színek kiválasztására. A színválasztó felületet a 36. ábra illusztrálja.



36. ábra: Grafikus színválasztó

4.2.2.2 VSR 2.0

A nyelv kiegészíthető számos vezérlőutasítással, melyek nem a vizualizációs logika, hanem az értelmező működését befolyásolják. Ehhez szükséges az értelmező kibővítése, mely a nyelv 2.0-ás verziójában kerülne megvalósításra.

Jelenleg az értelmező működését kizárólag a zenék vezérlési pontjai befolyásolják, azonban a kliensalkalmazás ezen kívül számos más funkciót is megvalósít, pl. a zenelejátszás megállítása, zenék átugrása, a spektrum kiszámítása. Ahhoz, hogy a szakterületi nyelvből ezek a funkciók elérhetőek legyenek, be kell vezetni az egyes utasításoknak megfelelő nyelvi elemeket, pl. időzített megállítás (stop, delay), a spektrum megfagyasztása (freeze), vagy a következő zenére ugrás (skip).

Ezen nyelvi elemek bevezetése magával vonja, hogy a lejátszási lista nem csak zenék vizualizációjának hívását tartalmazhatja, hanem az értelmező egyéb függvényeit is meghívhatja. Ennek általános kezelésére érdemes bevezetni egy szubrutinokat kezelő nyelvet. A szubrutinokat kezelő nyelvet érdemes úgy létrehozni, hogy a zenék vezérlési pontjai is alkothassanak szubrutinokat, ezzel a vizualizációs logika leírását megkönnyítve, pl. ismétlődő zenei részek vizualizációja lehet egyetlen szubrutin meghívása.

Szubrutinok leírása, vagy egyéb vezérlési utasítások kiadása nehezen valósítható meg XML nyelven, ezért a 2.0-ás verzióhoz érdemes megfontolni más fájlformátum felhasználását. A közismert leíró formátumok közül egyik sem használható fel kényelmesen ilyen célra, ezért a legjobb megoldás egy saját fájlformátum bevezetése. Ennek megvalósításához MPS-ben létre kell hozni egy nyelvet, amely kizárólag a szöveges állomány szintaktikáját valósítja meg. Ezt a nyelvet generálási célnak felhasználva a saját fájlformátumunk szerint is előállítható a vizualizációs program.

4.3 A kliensalkalmazás értékelése

4.3.1 Az implementálás értékelése

A fejlesztés kezdetekor az asztali alkalmazás megvalósításához a C# és C++ nyelveket vettem számításba. C#-ban jelentősen egyszerűbb lett volna a teljes alkalmazás megvalósítása, azonban C++-ban több tapasztalatom van. Figyelembe véve továbbá, hogy a spektrumszámítás akár nagy teljesítményigényű is lehet, C++ mellett döntöttem.

Ahogyan a 3.4 fejezetben elhangzott, a kliensalkalmazást modern C++-ban valósítottam meg, pár könyvtár felhasználásával. Eredeti célom a C++20-as nyelvi szabvány alkalmazása volt, azonban a projekt kezdetekor nem állt rendelkezésre olyan fordító, amely a teljes szabványt lefedte volna. A fejlesztés elején a GCC10 és MSVC fordítókat vizsgáltam meg. Ezek közül a GCC10 legfrissebb verzióját integráltam CodeBlocks fejlesztőkörnyezetbe, és ezzel kezdtem meg a fejlesztést. A fejlesztés közben azonban kiderült, hogy az SFML könyvtár nem támogatja a C++20-as szabványt, így végül a CodeBlocks-ba beépített MinGW fordítót alkalmaztam. A fejlesztés végére kiderült, hogy a C++ egy jó választás volt, ugyanis az implementálás során több ponton is teljesítménykorlátba ütköztem. Általában a MinGW-vel fordított programok lassabbak a MSVC vagy GCC által fordított binárisoknál, ezért a jövőben érdemes lehet a projektet másik toolchain-re áthelyezni.

A könyvtárakat tekintve a TagLib jó választásnak bizonyult, mert a zenék metaadatait egy meglehetősen egyszerű interfészen keresztül teszi elérhetővé. A RapidXML könyvtár a teljesítményre helyez nagy hangsúlyt, ezért a megvalósítása és interfésze is C-stílusú memóriacímeket tartalmaz, melyekkel könnyű hibát véteni. C++ projekt esetén érdemes lehet a könyvtár köré egy wrappert írni, mely modern nyelvi elemekkel teszi elérhetővé a beolvasott XML állomány egyes elemeit.

Az SFML könyvtár sok szempontból megkönnyítette a fejlesztést, ezért jó választásnak tekintem, és a projekt későbbi verziókban is alkalmazni fogom. Hátránya azonban, hogy nem támogat modern fordítókat, és ezáltal néhány modern nyelvi elemeket sem. A spektrumszámításnál is korlátba ütköztem a könyvtár használatával. A könyvtár zenéket kezelő osztálya nem teszi közvetlenül elérhetővé a zenei mintákat, ezért alternatív megoldásokra kellett szorítkoznom. Többek között ezért van szükség a zenei minták teljes bufferelésére, melyet az 3.4.7.1 fejezetben részleteztem, ahelyett, hogy közvetlenül az adattárolóról streamelném. Ez a limitáció megkerülhető egy saját osztály bevezetésével, mely az SFML-es implementációra épít és publikus tagfüggvényein keresztül elérhetők a zenei minták is, azonban erre a fejlesztés során nem jutott idő.

Az alkalmazás a soros port kezelését Windows operációs rendszerre specifikus könyvtárral valósítja meg (windows.h), de a jövőben szeretnék kidolgozni egy multiplatform megoldást, amit akár egy külön könyvtárban valósítanék meg.

4.3.2 Az értelmező értékelése

Az értelmező bemenetét a szakterületi nyelven leírt programból generált XML állomány képzi. Jelenleg a szakterületi nyelv lehetőséget nyújt szemantikai hiba elkövetésére, a 3.4.7.3 Hibakezelés fejezet szerint. Ennek kiküszöbölése a DSL szintjén meglehetősen összetett feladat, ezért érdemes lehet inkább az értelmező hibakezelését kiegészíteni validációs lépésekkel. A validációs lépések akár a program inicializációjakor, akár futásidőben ellenőrizhetnék a vizualizációs logika érvényességét. A validációs lépések közé akár egy hibajavító algoritmus is beiktatható, mely adott szabályok szerint korrigálja az esetleges szemantikai hibákat.

4.3.3 A spektrumszámítás értékelése

A kliensalkalmazás szerves részét képezi a spektrumszámítási algoritmus. A fejlesztés során számos algoritmust megvizsgáltam, melyek közül az 1.0-ás verzióban a DFT mellett

döntöttem, spektrumtorzítással kiegészítve. Az algoritmus a jelenlegi verzióban is ellátja a feladatát, azonban mind számítási kapacitás, mind a vizualizációs élmény szerint lehet rajta javítani.

Korábban említésre került a Wavelet-transzformáció. A DFT-vel szemben ez a transzformáció logaritmikus frekvenciamenetű spektrumot állít elő. Ez a vizualizáció szempontjából előnyös lehet, ugyanis alacsony frekvenciákon nagyobb felbontást biztosít. Alacsony frekvenciákon a nagyobb felbontás lehetővé teszi, hogy a különbözőnek érzékelt zenei hangok a spektrumon is jól elkülöníthetők legyenek. DFT esetén ezt a működést a spektrum torzításával közelítem, számítási kapacitás árán. Elméletileg a Wavelet-transzformáció számításigénye nem sokkal nagyobb, mint a DFT-jé, ráadásul a spektrumtorzítási lépés is kihagyható, így jelentős számítási kapacitást spórolhatunk meg. Sajnos azonban a Wavelet kipróbálására már nem jutott idő a dolgozat elkészítéséig.

A fejlesztés során felmerült egy alternatív DFT számítási mód, a rezonátoros Fourier-analizátor is. Ez a számítási mód a zene frekvenciakomponenseit egy rezonátorokból álló szabályozási rendszerrel határozza meg. A rezonátorok különböző frekvenciákra vannak hangolva, ezáltal a bemenő jel frekvenciakomponensei szerint eltérően rezonálnak. Az egyes rezonátorok amplitúdója és fázisa adja a spektrumot. Ennek az az előnye, hogy a rezonátorok frekvenciája tetszőlegesen megválasztható, így a spektrum frekvenciamenete is tetszőleges lehet, és ezek mellett a számításigénye a DFT-vel azonos, ezért a jövőben érdemes lehet kipróbálni.

A kliensalkalmazást felkészítettem arra, hogy a jövőben más spektrumszámítási algoritmusokat is implementálok. Az alkalmazás moduláris felépítése lehetővé teszi, hogy tetszés szerint ki lehessen cserélni a spektrumot előállító modult, akár futásidőben is. Ennek tesztelésére a DFT mellett implementáltam egy egyszerűsített vezérlést is, mely kizárólag a LED-ek színét vezérli, konstans fényerő mellett. Számos algoritmus megvalósítása után érdemes lehet a szakterületi nyelvet is kiegészíteni olyan vezérlőparancsokkal, amikkel cserélhető az aktuális vizualizációs mód.

4.4 A hardver értékelése

4.4.1 A kommunikáció értékelése

A jelenlegi verzióban a kliensalkalmazás és a mikrokontroller közti kommunikáció a nyers RGB adatok kiküldéséből áll. A jövőben érdemes lehet a kiküldött adatcsomagot

keretezni. Keretezéssel megvalósítható lenne több szalag aszimmetrikus vezérlése, valamint az adatátvitel hibáinak detektálása is, pl. ellenőrző összeg számításával. Az adatcsomag fogadásához és feldolgozásához szükséges idő közvetlenül befolyásolja a vezérelhető LED-ek számát. Aszimmetrikus vezérlés esetén a szalagok számával lineárisan nő az adatátvitelhez szükséges idő, mely könnyen meghaladhatja a kliensalkalmazás által biztosított pár ezredmásodperc szélességű időablakot. Ez kompenzálható az adatátviteli sebesség növelésével vagy összetettebb átviteli protokoll alkalmazásával, pl. az RGB adatok tömörítésével.

4.4.2 A mikrokontroller értékelése

A teljes beágyazott rendszert tekintve a Nucleo megfelelő választásnak bizonyult a LED-szalagok vezérléséhez, sőt, még a projekt következő verzióinak megvalósításához is alkalmazható. Számítási kapacitás tekintetében, a 4.4.1 fejezet szerint kiegészített kommunikáció feldolgozása is kényelmesen belefér a rendelkezésre álló időablakba. A keretezéssel több szalag is vezérelhető lehet, mely a kontrolleren egyszerűen megvalósítható azzal, hogy a külön szalagokat külön SPI buszra helyezzük.

A kontrolleren rendelkezésre álló hardverekkel akár másfajta vizualizáció is realizálható ugyanazon RGB adatok feldolgozásával, pl. lézerek dőlési szögét befolyásoló szervomotorok vezérlésével, anélkül, hogy a kliensalkalmazáson bármilyen módosítást kellene alkalmazni.

4.4.3 A LED-szalag értékelése

A saját tapasztalataim és a hallgatótársaim visszajelzése alapján is jobbnak mutatkozik a vizualizációs élmény, ha a LED-ek alacsony fényerővel vannak vezérelve. Ez abból adódik, hogy a LED-ek érzékelt fényereje nemlineárisan követi a fogadott RGB adatok abszolútértékét. Alacsony értékeknél szomszédos bájtok között is jelentős fényerőbeli eltérés figyelhető meg, azonban magas értékeknél a LED-ek fényereje közel statikus. Ebből következik, hogy alacsony fényerőnél a LED-ek érzékelt dinamikatartománya nagyobb, mint nagy fényerőnél.

Ezt figyelembe véve a jövőben érdemes lehet a jövőben a LED-ek nemlinearitását az értelmező szintjén figyelembe venni, és a kivezérlési tartományt az alacsonyabb értékekre korlátozni. A kísérleti tapasztalatok alapján azonban a dinamikatartomány növelése is korlátokba ütközik. Nagy dinamika esetén a fényerő hirtelen ugrásai hosszú idő után a szem fáradását okozhatják. Ennek kiküszöbölésére bevezettem egy exponenciális szűrést, mely a kliensalkalmazás két ciklusa között szűri az egyes fényerő értékeket, ezzel kisimítva a hirtelen változásokat.

A visszajelzések alapján érdemes lehet megfontolni a jövőben olyan szalag alkalmazását, melyen nagyobb a LED-ek közötti távolság. Ennek az lenne az előnye, hogy az egymás melletti LED-ek fénye kevésbé lapolódna át egymásra, így az egyes spektrális komponensek vizualizációi jobban elkülöníthetők lennének. Ugyanezen típusú szalagnak van olyan variánsa, melyen méterenként 30 LED található a jelenlegi 60 helyett, így még a vezérlésen sem kellene sokat változtatni.

4.4.4 A tápellátás értékelése

A jelenlegi verzióban a hardvert egy asztali számítógép tápegysége látja el árammal. A jövőben érdemes lehet költséghatékonyabb megoldást keresni, mondjuk egy megfelelő terhelhetőségű fali adapter felhasználásával. A fejlesztés során azonban kézenfekvőnek bizonyult asztali tápegységeket használni, széleskörű elérhetőségük és jó dokumentációik miatt. Asztali tápegységnél maradva a jövőben érdemes lehet 12V-os tápfeszültségű szalagokat alkalmazni, ugyanis a tápegységek 12V-os sínje jóval nagyobb áramot képes leadni, mint az 5V-os sín. Az általam felhasznált Corsair tápegység 12V-os terhelhetősége megközelítőleg 45A, mely majdnem kétszerese az 5V-os terhelhetőségnek.

5 Összefoglalás

A TDK dolgozatomban bemutattam az általam elképzelt zenei vizualizációs keretrendszer megvalósítását. A projekt komplexitása miatt néhány tervezési megfontolás és megvalósítás nem fért a dolgozat kereteibe, ugyanakkor igyekeztem a legfontosabb részeket a megfelelő részletességgel és pontossággal leírni.

Ehhez a dolgozat elején bevezettem a szakterületi nyelvek tárgyalásához szükséges definíciókat, majd ezek felhasználásával röviden bemutattam egy szakterületi nyelvek létrehozására tervezett keretrendszert, az MPS-t. Az MPS keretrendszerrel a szakdolgozatom elkészítése közben ismerkedtem meg, és a benne rejlő lehetőségeket máig nem tudtam teljesen kiaknázni. A rövid bemutatás ára, hogy az MPS egyes területeit csak említés szintjén, vagy egyáltalán nem tudtam bemutatni, pl. nyelvorientált programozás támogatása, scriptelhetőség, vagy összetett nyelvgenerálási rendszerek. Ennek ellenére úgy gondolom, hogy az MPS-t leíró fejezetek egy átfogó képet adnak a fejlesztőkörnyezet működéséről és a széleskörű felhasználási lehetőségeiről.

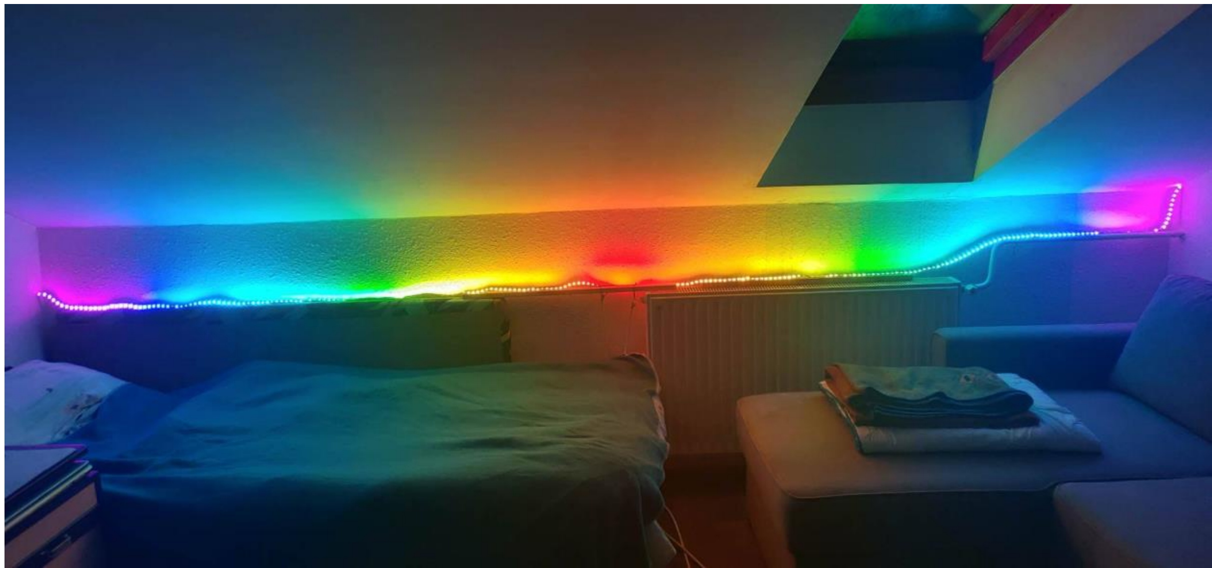
A szakterületi nyelv implementációja után részletesen bemutattam az azt értelmező kliensalkalmazás felépítését. Rávilágítottam az egyes tervezési döntésekre, és a fejlesztés azon lépéseire, melyek végeredményeként előállt az alkalmazás. Kitértem a moduláris felépítésére, azon belül is a legfontosabb modulok felépítésére és relációira. Különös figyelmet fordítottam a vizualizációs logika értelmezéséről szóló fejezetekre, mellyel szintén azt igyekeztem illusztrálni, hogy egy látszólag egyszerű projekt lépéseit is jól meg kell fontolni, hogy a fejlesztés eredményes legyen. A kliensalkalmazás keretein belül, a matematikai háttér részletezése nélkül bemutattam azon megfontolásokat is, melyek a spektrum számításával kapcsolatban merültek fel.

A kliensalkalmazás felépítésének bemutatása után részletesen dokumentáltam a teljes beágyazott rendszert. A beágyazott rendszer tárgyalásakor nagy hangsúlyt fektettem arra, hogy milyen mérnöki döntések voltak szükségesek ahhoz, hogy a teljes projekt életre keljen, így bemutattam a felhasznált kommunikációs protokollokat és hardveres megoldásokat is.

A teljes dokumentáció után kitértem arra, hogy a projektet milyen környezetben teszteltem, valamint, hogy a tesztek alapján milyen visszajelzéseket kaptam. Ezek alapján a visszajelzések alapján igyekeztem kiemelni a megvalósítás előnyeit, valamint értékeltem a jelenlegi megvalósítás kérdéseit és hiányosságait is. Az eredmények értékelésekor igyekeztem

a környezetemtől visszajelzésként kapott kérdésekre szavatosan válaszolni, és ezen válaszokon keresztül a projekt jövőbeli fejlesztési lehetőségeit is bemutatni.

Összefoglalva, szerintem a projekt sikeresen demonstrálja a magas absztrakciós szintű nyelvek beágyazott rendszerekre való illesztésekor felmerülő kérdéseket, továbbá mindezekre egy kézzelfogható megoldást nyújt. Remélem, hogy a Kedves Olvasónak ugyanakkora örömöt nyújt a dolgozat tanulmányozása, mint nekem annak az elkészítése, valamint a végeredmény használata.



37. ábra: A VSR működés közben

Irodalomjegyzék

[1] A. Aho, R. Sethi, J. Ullman és M. S. Lam: *Compilers: Principles, Techniques, and Tools* (2006, Pearson Education Inc.)

[2] Markus Voelter: *DSL Engineering Designing, Implementing and Using Domain Specific Languages* (2013, http://dslbook.org)

[3] JetBrains Metaprogramming System (MPS), https://www.jetbrains.com/mps/

[4] Fabien Campagne: *The MPS Language Workbench, Volume I* (third edition, 2014, http://campagnelab.org/publications/our-books/)

[5] M. B. Mokhsin, N. B. Rosli, S. Zambri, N. D. Ahmad és S. R. Hamidi: *Automatic music emotion classification using artificial neural network basedon vocal and instrumental sound timbres* (2014, Journal of Computer Science 10 (12): 2584-2592)

[6] Balogh L., Kollár I., Németh J., Péceli G., Sujbert L.: *Digitális jelfeldolgozás, hallgatói segédlet* (2008, BME Méréstechnika és Információs Rendszerek Tanszék)

[7] Truong, Nguyen, Gilbert, Strang: *Wavelets and Filter Banks* (1996, Wellesley College)

[8] Dabóczi Tamás: *Fourier analízis hatékonyan* (2016, BME Méréstechnika és Információs Rendszerek Tanszék)

[9] Matteo Frigo, Steven G. Johnson: *The Design and Implementation of FFTW3* (2005, Proc. IEEE, vol. 93, no. 2, pp. 216-231)

[10] A. Härmä, M. Karjalainen, L. Savioja, V. Välimäki, U.K. Laine, A.J. Huopaniemi: *Frequency-Warped Signal Processing for Audio applications* (2000, J.AudioEng.Soc., Vol.48, No.11)

[11] WS2812B LED-szalag specifikációja, https://www.digikey.com/en/datasheets/parallaxinc/parallax-inc-28085-ws2812b-rgb-led-datasheet

[12] FT232R USB-UART IC specifikációja, https://www.hestore.hu/prod_getfile.php?id=8553

[13] STM32 Nucleo F446RE mikrokontroller specifikációja, https://www.st.com/en/evaluation-tools/nucleo-f446re.html#documentation (2015)

[14] Corsair RM550x asztali számítógép tápegység specifikációja, https://www.corsair.com/corsairmedia/sys_master/productcontent/RMx_2018_Manual.pdf (2018, pp. 11)

Köszönetnyílvánítás

Ezúton is szeretnék köszönetet nyilvánítani azoknak, akik szakmai visszajelzésükkel segítettek a fejlesztésben, valamint a projekt dokumentálásában. Külön köszönettel tartozom Csorvási Gábornak a beágyazott rendszerről szóló fejezet lektorálásáért, valamint mentoromnak, Dr. Mezei Gergelynek a félévek óta tartó támogatásáért.

Köszönettel tartozom a családomnak itthon és külföldön,
akik nélkül nem sikerülhetett volna.

Special thanks to my family at home and abroad,
without whom I couldn’t have made it.

#metoo