UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Iluminare ambientală pentru sisteme desktop

propusă de

Lucian-Andrei Bosînceanu

Sesiunea: *iunie, 2019*

Coordonator științific

Lect. Dr. Cosmin-Nicolae Vârlan

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Iluminare ambientală pentru sisteme desktop

Lucian-Andrei Bosînceanu

Sesiunea**:** *iunie, 2019*

Coordonator științific

Lect. Dr. Cosmin-Nicolae Vârlan

Cuprins

Introducere 4

Contribuții 6

1 Prezentarea problemei 7

1.1 Ochiul uman și reacția acestuia la lumina unui ecran 7

1.2 Fenomenul de *lumină ambientală* și avantajele acestuia 8

1.3 Lumina ambientală ca problemă Input-Output 9

1.4 Concluzie 9

2 Soluția propusă 10

2.1 Deschiderea aplicației și meniul principal 10

2.2 Opțiuni generale 11

2.2.1 Opțiunea Frame Type 11

2.3 Opțiunile modului video 13

2.4 Opțiunile modului audio 15

2.5 Opțiunile modului ambiental 16

2.5.1 Editarea unui profil 17

2.5.2 Editarea culorilor LED-urilor în mod individual 17

2.5.3 Crearea gradientelor 18

2.5.4 Gestionarea animației curente 18

2.6 Concluzie 19

3 Realizarea componentei hardware 20

3.1 Descrierea componentelor 20

3.1.1 Banda LED ws2812 5V 20

3.1.2 Arduino UNO 20

3.1.3 Sursă de alimentare 5V 10A 20

3.1.4 Componente pentru construcția ramei 20

3.1.5 Unelte și alte ustensile 21

3.2 Construcția sistemului 21

3.2.1 Diagrama circuitului și asamblarea acestuia 21

3.2.2 Construcția ramei 23

3.3 Programarea microcontroller-ului Arduino 24

3.3.1 Tehnologii folosite 24

3.3.2 Protocolul de comunicare 24

3.3.3 Descrierea algoritmului 25

3.4 Concluzie 27

4 Implementarea aplicației de procesare 28

4.1 Tehnologiile folosite 28

4.1.1 Java SE 8 28
4.1.2 JavaFX 2.0 28
4.1.3 SceneBuilder 28
4.1.4 Maven 28
4.1.5 TarsosDSP 28
4.1.6 Gson 29
4.1.7 JSSC 29
4.2 Arhitectura aplicației 29
4.3 Model 30
4.3.1 Clasa LedFrame 30
4.3.2 Clasa FrameColorPreset 30
4.3.3 Modulul config 30
4.4 Controller 34
4.4.1 Modulul frame 34
4.4.2 Modulul processing_units 36
4.4.3 Modulul source 38
4.4.4 Modulul *workers* 39
4.4.5 Modulul algorithms 43
4.4.6 Modulul io 50
4.5 View 51
4.5.1 Modulul ui 51
4.5.2 AlertManager 53
4.6 MainApp 53
4.7 Concluzie 53
5 Concluzie 54
Bibliografie 55
Anexa 1 – codul încărcat pe *Arduino* 56
Anexa 2 – VideoWorker 58
Anexa 3 – Audio Worker 59
Anexa 4 – Algoritmul de procesare audio *FrequencyWave* 61
Anexa 5 – Algoritmul de preprocesare *DynamicSensibilityDetection* 62
Anexa 6 – SettingsViewManager 63

Introducere

Faptul că suntem mereu în preajma ecranelor a devenit o realitate de zi cu zi. La birou sau acasă, lumina monitoarelor sau a televizoarelor ne acaparează și ne pune o presiune imensă asupra ochilor, în special noaptea. Ochiul uman este făcut să perceapă o medie a intensității luminoase. Fie că ne uităm la un peisaj din natură, un ecran al televizorului sau o cameră parțial iluminată, pupila se dilată sau se contractă pentru a percepe media luminii a întregii imagini pe care o privim. Când ochiul uman este expus la lumina ecranelor noaptea, acesta trebuie să depună un efort suplimentar în a se adapta întrucât media intensității luminoase nu este dată doar de lumina monitorului, cât de întreaga scenă privită, inclusiv zonele întunecate din preajma dispozitivului. Astfel, o expunere îndelungată în fața ecranelor poate provoca, pe termen scurt, dureri de ochi, disconfort, iritații, uscarea corneii, dureri de cap, iar pe termen lung, tensiune oculară crescută, îmbătrânirea prematură a ochiului sau chiar pierderi de vedere.

Există mai multe soluții pentru a preveni aceste probleme. Una dintre ele este limitarea luminii albastre radiate de ecrane prin folosirea unor filtre speciale, cum ar fi ochelari specializați sau produse software dedicate. O altă soluție, cea pe care se concentrează lucrarea de față, este fenomenul de *bias lighting*[1]. Acest fenomen presupune atașarea unei surse de lumină în spatele ecranului. În acest mod, se creează lumină ambientală care va determina un efort mai mic al ochiului în a se adapta. Alte beneficii ale *luminii ambientale* vor fi discutate în primul capitol.

Această soluție nu este întru totul una nouă. Dacă în trecut erau folosite lămpi care proiectau lumină în spatele ecranului, începând cu anii 2000 s-a făcut tranziția spre becuri LED care erau atașate ecranelor și care erau alimentate prin portul USB sau HDMI. În anul 2002, Philips a patentat tehnologia *Ambilight,* care a îmbunătățit fenomenul de *lumină ambientală* prin a-l face unul adaptiv și dinamic. Astfel, LED-urile tehnologiei *Ambilight* își schimbă culoarea pe baza culorilor pixelilor de pe marginea ecranului, creând, astfel, o experiență spectaculoasă și captivantă. Philips a aplicat această tehnologie doar televizoarelor pe care le produce, nu au făcut și soluții dedicate calculatoarelor personale.

Hyperion este o aplicație open source care își propune adaptarea tehonologiei *Ambilight* la orice televizor prin intermediul unui program de procesare care rulează pe un Raspberry Pi și care prelucrează imagini captate direct de pe portul HDMI. Această soluție este compatibilă și cu monitoarele calculatoarelor, dar nu funcționează pentru sistemul de operare *Windows*.

Prismatik este o implementare a conceptului de *lumina ambientală* și pentru sisteme *desktop*. Pe PC-ul utilizatorului rulează un program scris în C++ care captează și analizează imaginea de pe ecran, urmând ca informațiile obținute să fie transmise, prin portul serial, la un *microcontroller Arduino,* care va aprinde o bandă de LED-uri atașate, simulând, astfel, tehnologia *Ambilight*. Totodată, această soluție are implementată și procesare audio care face ca LED-urile să pulseze în ritmul muzicii.

O soluție asemănătoare cu *Prismatik* este *AmbiBox,* care are aceleași funcționalități, dar, în plus, permite schimbarea profilelor de configurare prin apăsarea unor taste sau de la distanță, pe baza unui serviciu web.

Analizând aceste soluții am observat că ele sunt dificil de instalat sau de configurat. Ba chiar unele dintre ele (de exemplu *Hyperion*) necesită *hardware* adițional, ceea ce crește prețul asamblării. În cadrul proiectului de licență, mi-am propus realizarea unui sistem de *lumină ambientală* dinamică, dedicat sistemelor *desktop*. Deși există deja alte soluții la această problemă, abordarea mea își propune să îndeplinească următoarele obiective:

[1] Referit în restul lucrării ca *lumină ambientală.*

O1. Să ruleze pe cât mai multe sisteme de operare;
O2. Să fie simplu de folosit și de configurat. Dacă se poate, doar să pornesc un executabil și să funcționeze, fără a trece printr-o corvoadă de configurări doar pentru a porni aplicația (cum se întâmplă în cazul *Prismatik* sau *AmbiBox*);
O3. Să aibă o arhitectură flexibilă și extensibilă, care să permită adăugarea facilă de noi opțiuni de configurare;
O4. Partea hardware să fie ușor de realizat și cât mai ieftină, iar produsul final să nu fie unul invaziv;
O5. Să aibă trei moduri de funcționare (video, audio și ambiental), cu multiple opțiuni de personalizare.

Această lucrare este structurată în 5 capitole care acoperă descrierea problemei și modul în care a fost rezolvată:

În capitolul 1 (Prezentarea problemei) este descrisă anatomia și fiziologia ochiului, precum și impactul negativ al luminii ecranului asupra acestuia. Totodată, este prezentat și fenomenul de *lumină ambientală*, împreună cu avantajele sale.

Capitolul 2 (Soluția propusă) prezintă cum se poate implementa un astfel de sitem, dar se concentrează pe funcționalitățile aplicației de procesare, sub forma unui ghid de utilizare.

Capitolul 3 (Realizarea componentei hardware) descrie componentele care sunt necesare în realizarea sistemului de *lumini ambientale*, motivația alegerii acestora, cum se îmbină între ele și cum este programat *microcontroller-ul* pentru a fi pregătit de comunicarea cu aplicația care rulează pe calculator.

Capitolul 4 (Implementarea aplicației de procesare) conține informații despre tehnologiile folosite în realizarea aplicației de procesare, dar și cum a fost structurată și implementată.

Contribuții

În cadrul proiectului de licență am realizat o implementare proprie a unui sistem de *lumină ambientală* dinamică (în stil *Ambilight*) pentru ecranele calculatoarelor personale. Acest lucru a presupus trei aspecte:

1. Asamblarea unei rame de LED-uri care se poate atașa pe partea din spate a unui monitor, precum și conectarea ei la un *microcontroller Arduino*, dar și la o sursă de alimentare. *Microcontroller-ul* a fost programat în așa fel încât să poată controla rama pe baza unor informații primite pe portul serial;
2. O aplicație *desktop*, scrisă în Java, de procesare în timp real a imaginii afișate pe ecranul unui calculator sau al sunetului recepționat de microfonul implicit, cu scopul de a obține un tablou unidimensional care conține culorile ce ar corespunde LED-urilor ramei;
3. Serializarea și transmiterea acestui tablou prin portul serial la *Arduino*, pe baza unui protocol de comunicare.

Aplicația de procesare are trei moduri de funcționare:

- Video – asemănător tehnologiei *Ambilight*, LED-urile din spatele monitorului vor lua culorile pixelilor aflați pe marginea ecranului;
- Audio – sunetul înregistrat de microfonul implicit este prelucrat cu scopul de a realiza o animație a LED-urilor în tandem cu ritmul muzicii;
- Ambiental – permite configurarea amănunțită a unui profil de culoare al LED-urilor pentru a reda *lumină ambientală* statică sau animată.

1 Prezentarea problemei

Tematica acestui capitol se axează pe impactul negativ al ecranelor asupra ochilor noștri, ce presupune un sistem de *lumină ambientală* dinamică și care sunt avantajele acestuia.

1.1 Ochiul uman și reacția acestuia la lumina unui ecran

În cele ce urmează, vor fi prezentate elemente de anatomie și de fiziologie ale analizatorului vizual, cu scopul de a înțelege mai bine cum sunt percepute culorile și imaginile de către acesta, aspecte importante pentru a justifica necesitatea fenomenului de *lumină ambientală*. Din punct de vedere al relevanței față de subiectul lucrării, prezentarea nu va fi una exhaustivă, ci se va concentra pe aspectele importante. În partea dreaptă a paginii se află o figură care descrie anatomia analizatorului vizual, elemente ale acesteia fiind referite în cele ce urmează.

Astfel, pentru ca un obiect să fie perceput de către ochiul uman, acesta trebuie să reflecte lumina. Razele de lumină reflectate intră prima dată în contact cu **corneea**, un înveliș transparent și curbat, parte a scleroticii (țesutul alb care protejează exteriorul ochiul). Fiind curbată, aceasta va răsturna imaginea, creierul fiind cel responsabil de reorientarea ei pentru a o percepe așa cum este în realitate. Raza de lumină este refractată apoi printr-un mediu gelatinos, numit **umoare apoasă,** până când ajunge la **cristalin**. Nivelul de lumină care ajunge la acesta este controlat de către **pupilă** (pata neagră din centrul ochiului). (Cristescu, Sălăvăstru, Voiculescu, Niculescu, & Cârmaciu, 2006)

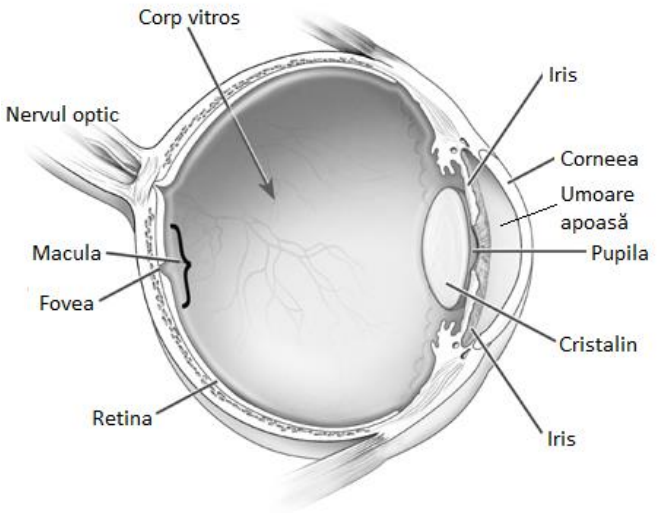

Fig. 1 - Anatomia analizatorului vizual[2]

Aceasta se dilată pentru a permite unei cantități mai mari de lumină să ajungă la cristalin, ideal în condițiile în care luminozitatea mediului este scăzută, sau se contractă pentru a proteja interiorul ochiului de lumina foarte puternică. Dilatarea pupilei este controlată de mușchii **irisului,** țesut pigmentat care determină culoarea ochilor. Cristalinul joacă rolul unei lentile flexibile care, prin intermediul mușchilor ciliari, își poate schimba forma sa curbată pentru a se concentra pe obiecte aflate în apropiere sau la distanță. Lumina refractată de către cristalin ajunge în **corpul vitros,** un mediu care, la fel ca și umoarea apoasă, este umplut cu un lichid gelatinos responsabil cu hrănirea țesuturilor ochiului.

În cele din urmă, după trei refracții, lumina obiectului care se dorește a fi perceput ajunge în partea posterioară a ochiului, la **retină**. Aceasta este un țesut fotosensibil, al cărui rol constă în a converti luminii percepută în impulsuri electrice, care apoi sunt transportate prin **nervul optic** către creier. **Macula**, sau pata galbenă, este zona retinei în care ajunge lumina întregii imagini percepute de către analizatorul vizual, în centrul acesteia fiind **fovea**, sau punctul de maximă sensibilitate vizuală, care percepe doar razele obiectului asupra căruia privirea este atentă. Pe suprafața retinei, concentrate în zona maculei, se află celule fotoreceptoare de două tipuri:

- Bastonașe – celule extrem de sensibile la lumină, specializate în a funcționa în condiții de luminozitate scăzute. Ele sunt responsabile de vederea periferică, precum și de cea

[2] Imagine preluată și adaptată de pe situl organizației *National Eye Institute* - https://www.nei.nih.gov/health/eyediagram

scotopică, specifică luminozității crepusculare. Sunt specializate în a percepe, mai degrabă, dinamica obiectelor decât culoarea acestora;

- Conuri – celule specializate în a funcționa în condiții de luminozitate sporită, responsabile de vederea centrală și fotopică, specifică celei din timpul zilei. Acestea sunt celulele care percep culorile și detaliile obiectelor, însă, pentru aceasta, au nevoie de multă lumină, ele nefiind sensibile la nivele scăzute de luminozitate. Celulele cu conuri sunt concentrate în zona **foveii.**

Vederea normală, sănătoasă, este caracterizată de un balans între funcționalitățile celor două tipuri de celule fotoreceptoare. Însă, ce se întâmplă când ochii sunt expuși unui ecran luminat noaptea? În astfel de condiții, există o singură sursă intensă de lumină înconjurată de o zonă aflată în beznă. Deși ochii se concentrează asupra unui singur punct intens luminat, media intensității luminoase a întregii scene privite este mai mică decât cea a ecranului. Acest fapt determină analizatorul vizual să creadă că este într-o ambianță crepusculară, activând, astfel, celulele cu bastonaș, care, la rândul lor, forțează dilatarea pupilei pentru a crește cantitatea de lumină percepută, dar într-o măsură mai mare decât ar fi necesar. Mai multă lumină percepută determină supraîncărcarea celulelor de tip con. Totodată, întrucât ecranul este singura sursă de lumină din peisaj, intensitatea luminoasă a acestuia determină gradul de dilatare al pupilei, însă aceasta nu este extrem de flexibilă pentru a ține pasul cu schimbările bruște și haotice de luminozitate. Pupila se va dilata și ea în ritm haotic, de la un diametru mai mic la unul mai mare și invers.

Supraîncărcarea cu informație a celulelor bastonaș, cât și ritmul haotic de dilatare al pupilei, au ca rezultat obosirea ochiului, ceea ce cauzează uscarea, usturimea sau lăcrimarea acestuia, disconfort specific activității de uitat la televizor sau monitor noaptea. Așa cum a fost menționat și în introducere, expunerea pe termen lung la lumina ecranelor poate cauza dureri de ochi, migrene, oboseală, stări de amețeală, creșterea tensiunii oculare, probleme de vedere sau pierderea acesteia etc. (Sheppard & Wolffsohn, 2018)

1.2 Fenomenul de *lumină ambientală* și avantajele acestuia

O soluție simplă la această problemă este adăugarea de surse noi de lumină în cameră, precum aprinderea becului sau folosirea de lămpi ambientale, care vor face ca media intensității luminoase să crească. Deși e mai bine pentru ochi, sursele noi de lumină se vor reflecta de ecran, ceea ce duce la o strălucire (*glare*) care scade calitatea imaginii, făcând culorile să pară șterse. Astfel, este nevoie de o sursă de lumină care să nu se reflecte de ecran. Așa a luat naștere conceptul de *lumină ambientală,* principiul de bază al acestuia fiind adăugarea unei surse de lumină în spatele ecranului. Tipul de lumină preferat pentru acest fenomen este cea produsă de o bandă LED, întrucât este ieftină, durabilă și consumă puțin.

Un avantaj major al acestei tehnologii este reprezentat de faptul că ochii vor fi mai puțin solicitați, întrucât acum este mai multă lumină în cameră și nu mai este necesară dilatarea haotică a pupilei. De asemenea, un astfel de sistem crește contrastul culorilor redate pe ecran, fenomen cunoscut și sub numele de iluzia contrastului.

În imaginea din dreapta se poate observa cum, deși bara gri din mijloc are aceeași culoare pe toată lungimea ei, aceasta este percepută diferit în funcție de conținutul fundalului. În partea din stânga, unde fundalul este închis, nuanța de gri pare mai ștearsă și mai deschisă, în timp ce în partea din dreapta este percepută ca fiind mai închisă și mai bogată în culoare. Folosirea unui sistem de *lumină ambientală* creează un fundal luminat, ca în partea din dreaptă a iluziei, ceea ce determină o mai bună percepție a nuanțelor de natură închisă, precum negru și gri, dar și a celorlalte culori.

Fig. 2 – Iluzia constrastului

În plus, folosirea unui sistem de *lumini ambientale* dinamice, prin intermediul căruia culorile de pe marginea ecranului sunt transpuse LED-urillor, creează o experiență vizuală deosebită, captivantă, în care acțiunea de pe micul ecran se extinde dincolo de marginile acestuia. (Fitzpatrick, 2017)

1.3 Lumina ambientală ca problemă Input-Output

Problema pe care această lucrare își propune să o rezolve este aceea de a creea un sistem de *lumini ambientale* dinamice dedicat sistemelor *desktop*, capabil să îndeplinească o serie de obiective descrise în introducere. Față de un televizor, utilizatorul are mai mult control asupra conținutului redat pe ecran, ceea ce îi oferă mai mult confort și flexibilitate în folosirea unui astfel de sistem, în special dacă acesta folosește un calculator mai mult decât un TV. Sistemul trebuie să aibă trei moduri fundamentale de funcționare:

- Video – conținutul de pe ecranul utilizatorului este în permanență analizat cu scopul de a capta culorile de pe marginea ecranului;
- Audio – LED-urile benzii reacționează într-un anumit fel la semnalul audio recepționat în preajma sistemului;
- Ambiental – să existe posibilitatea asocierii oricărei culori LED-urilor, precum și animații la nivelul întregii benzi, pe baza preferinței utilizatorului.

Analizând aceste funcționalități, problema se poate generaliza astfel:

Input: *byte[] source* – un semnal digital serializat într-un tablou de octeți. În funcție de modul de funcționare curent, acesta poate fi o imagine, un *sample* audio sau chiar trecerea timpului.

Output: *List<Color> colors* – o listă de culori, în format RGB, de dimensiune egală cu numărul de LED-uri, care ilustrează starea benzii în funcție de conținutul variabilei *source*.

1.4 Concluzie

În acest capitol am studiat anatomia și fiziologia analizatorului vizual la om, impactul negativ al ecranelor luminoase asupra acestuia, precum și necesitatea unui sistem de *lumini ambientale*. În următorul capitol va fi prezentată o soluție de implementare a unui astfel de sistem.

2 Soluția propusă

Pentru realizarea unui sistem de *lumini ambientale*, ca cel descris în capitolul anterior, sunt necesare atât o componentă *hardware*, cât și una *software*. Partea *hardware* este reprezentată de o bandă cu LED-uri individual adresabile, care permite modificarea nominală a culorii lor. Banda are nevoie de o sursă de alimentare pentru a putea opera, dar mai ales este necesar un *controller* care va aprinde LED-urile. În acest caz, se va folosi un *Arduino UNO*. Acesta nu este suficient de puternic pentru a face procesare audio sau video în timp real, de aceea este nevoie de o aplicație *software* dedicată, care să ruleze pe calculatorul utilizatorului. Astfel, aplicația de procesare se va ocupa de obținerea culorilor LED-urilor, urmând ca acestea să fie transmise, prin portul USB, la *Arduino*. Un program încărcat pe *microcontroller* va capta aceste informații si va aprinde banda în mod corespunzător. Componenta *software* a fost proiectată în așa fel încât să fie agnostică de o anumită construcție *harware* a ramei de LED-uri, permițând, astfel, multiple opțiuni de configurare care oferă libertate și flexibilitate din acest punct de vedere.

În capitolul de față sunt descrise funcționalitățile aplicației de procesare, punctual, ca un ghid de utilizare. Informații despre implementarea celor două părți vor fi prezentate în următoarele două capitole. În cele ce urmează, se presupune că partea *hardware* a fost deja asamblată și conectată la calculator.

Majoritatea ferestrelor aplicației au aceste trei butoane, cu următoarea semnificație:

Apply – va salva modificările făcute și va actualiza comportamentul ramei, fără a închide fereastra curentă;

Ok – va salva modificările făcute, va închide fereastra curentă și va actualiza comportamentul ramei;

Cancel – va închide fereastra curentă, fără a actualiza comportamentul ramei. Dacă până să se apese pe acest buton s-a apăsat cel de ***Apply***, atunci setările de la acel moment vor rămâne salvate.

2.1 Deschiderea aplicației și meniul principal

Aplicația se pornește dând dublu-click pe executabilul acesteia. Vom fi întâmpinați de meniul principal al aplicației. Acesta conține două butoane de tip *choicebox* – *Mode* și *Settings*. Primul permite selectarea modului de funcționare al aplicației între video, audio, ambiental și *none*. Alegerea unuia dintre acestea va declanșa automat schimbarea comportamentului ramei. Al doilea buton conține o listă de setări posibile, acestea incluzând câte o intrare pentru fiecare mod de funcționare, dar și o intrare pentru opțiunile generale. Alegerea oricărui element din listă va declanșa deschiderea unei noi ferestre corespunzătoare tipului de setări respectiv.

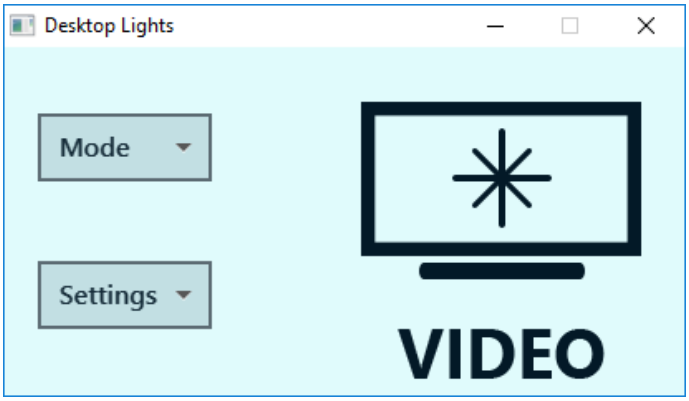

Fig. 3 - Meniul principal

Această abordare simplistă a meniului principal se bazează pe ideea că, odată ce configurările pentru toate cele 3 moduri de funcționare au fost făcute, utilizatorul va dori doar să comuteze cât mai ușor între ele, fără ca interfața să fie poluată cu alte opțiuni. Dacă aplicația este deschisă pentru prima dată, modul selectat va fi *None* și se recomandă modificarea opțiunilor generale înainte de comutarea către un alt mod!

2.2 Opțiuni generale

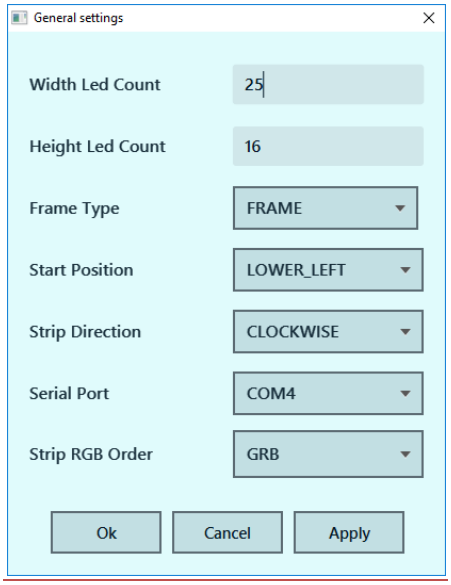

Fig. 4 - Meniul opțiunilor generale

Opțiunile generale se accesează din meniul principal, prin apăsarea butonului *General Settings* din meniul *Settings*. Acestea vizează o serie de configurări cu privire la construcția ramei, dar și la comunicarea dintre aplicația de procesare și *microcontroller*.

Width Led Count – câte LED-uri are rama pe lungime;

Height Led Count – câte LED-uri are rama pe lățime;

Start Position – poziția începutului benzii LED, considerând că rama este atașată de spatele monitorului, iar observatorul privește ecranul. Primul LED se poate afla doar în unul dintre cele patru colțuri, nu și pe laturi;

Strip Direction – direcția generală a benzii LED, considerând perspectiva de mai sus. Aceasta poate fi în sensul acelor de ceas sau în sens trigonometric;

Serial Port – numele portului USB la care este conectat *Arduino*. Acest meniu detectează automat toate dispozitivele conectate și va conține doar numele porturilor lor. Selectați din listă pe acela corespunzător;

Strip RGB Order – ordinea de transmitere a octeților corespunzători unei culori. În funcție de banda LED utilizată, nu este obligatoriu ca ordinea să fie cea standard (RGB). Selectați din listă ordinea corespunzătoare.

2.2.1 Opțiunea Frame Type

Această opțiune este mai complexă și necesită mai multe explicații. Fiecare opțiune din această listă reprezintă o arhitectură posibilă pentru rama de LED-uri. În continuare va fi descrisă semnificația fiecăreia, precum și o serie de avantaje și dezavantaje.

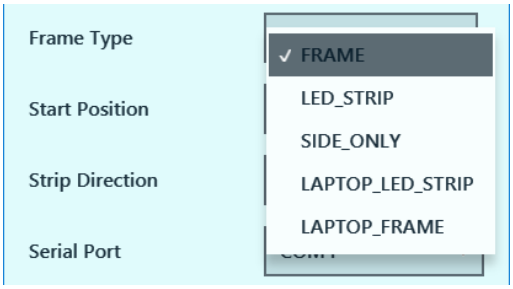

Fig. 5 - Tipuri de construcție ale ramei

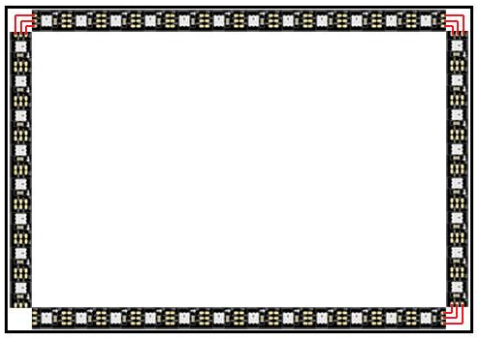

Fig. 6 - Arhitectura FRAME

FRAME – pe laturile de pe lățime există exact *Height Led Count* LED-uri, iar pe lungime sunt *Width Led Count*. Nu există LED-uri în colț, culoarea acestora fiind redată de extremitățile segmentelor. Modul de asamblare al ramei descris în capitolul **Error! Reference source not found.** corespunde acestei arhitecturi. Un avantaj major îl reprezintă estetica ramei, dar și faptul că LED-urile pot fi aliniate în așa fel încât să proiecteze lumina la un unghi de 45°. Pe de altă parte, rigiditatea construcției cauzează dificultăți de prindere pe monitoare cu spatele denivelat sau curbat, dar și de asamblare în general.

În plus, dacă un LED se strică, înlocuirea unui segment nou este foarte dificilă.

LED_STRIP – se folosește direct bandă LED care se modelează în forma unei rame. Există LED-uri dedicate pentru fiecare colț. Asamblarea este facilă, dar necesită atenție sporită pentru că unele colțuri se numără de două ori pentru a se respecta dimensiunile specificate. În acest sens, convenția este următoarea: considerând că banda este prinsă de spatele monitorului și că perspectiva este din față, segmentul din stânga conține ***Height Led Count – 1*** LED-uri, cel de sus ***Width Led Count -1***, cel din dreapta ***Height***, iar cel de jos ***Width – 2***.

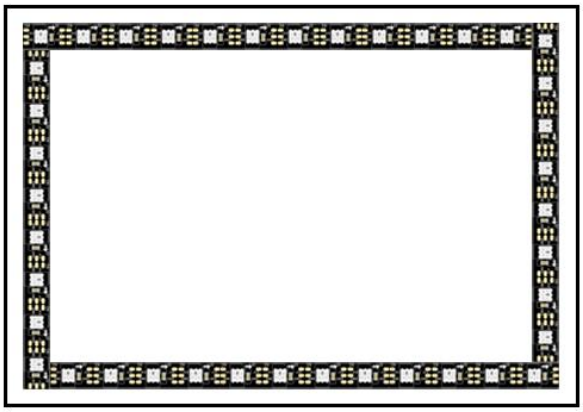

Fig. 7 - Arhitectura LED_STRIP

Avantajele acestei construcții sunt ușurința de asamblare, cât și faptul că banda nu este deloc secționată, ceea ce duce la absența unor întârzieri cauzate de parcurgerea curentului electric a firelor de legătură dintre segmente. În plus, banda se poate mula pe spatele oricărui ecran. Cu toate acestea, estetica lasă de dorit, în special dacă se recurge la lipirea benzii folosind bandă adezivă, dar dacă se dorește obținerea rapidă a unui rezultat, atunci este o soluție bună.

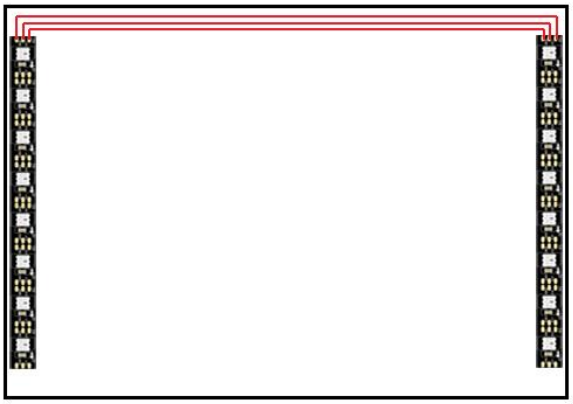

Fig. 8 - Arhitectura SIDES_ONLY

SIDES_ONLY – principiul de asamblare este același ca la ***FRAME***, numai că se folosesc doar segmentele de pe lățime, conectate prin fire electrice. Această soluție este potrivită în cazul ecranelor curbate și are avantajul că se poate construi în așa fel încât să se utilizeze un număr mic de LED-uri (maxim 14). Astfel, nu mai este necesară alimentarea externă de la o sursă de curent. Dezavantajul este că toate funcționalitățile ramei nu vor arăta la fel de bine și că pot exista anumite întârzieri cauzate de trecerea curentului electric prin firele de legătură.

LAPTOP_FRAME – aceleași principii de funcționare, avantaje și dezavantaje ca la modul ***FRAME***, doar că nu mai există partea de jos a ramei. Acest model de arhitectură este potrivit în cazul utilizării unui laptop, întrucât luminile din partea de jos nu sunt atât de vizibile.

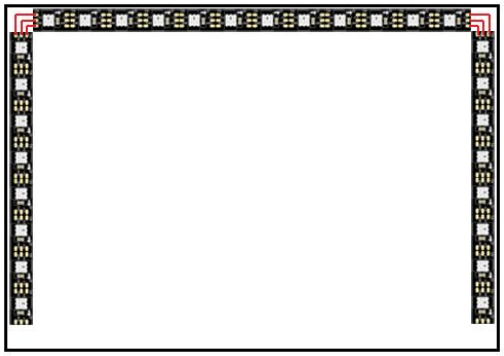

Fig. 9 - Arhitectura LAPTOP_FRAME

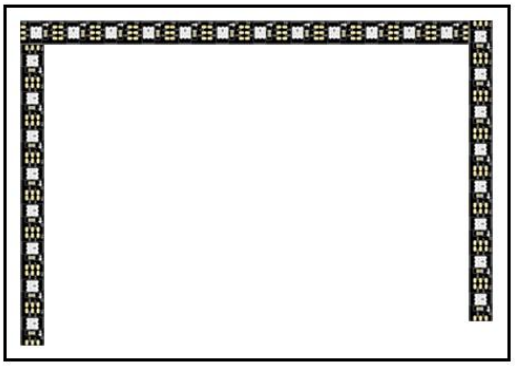

Fig. 10 - Arhitectura LAPTOP_LED_STRIP

LAPTOP_LED_STRIP – aceleași principii de funcționare, avantaje și dezavantaje ca la modul ***LED_STRIP***, doar că nu există partea de jos a ramei.

2.3 Opțiunile modulului video

Opțiunile modulului video sunt accesibile din meniul principal alegând opțiunea *Video Settings* de la butonul *Settings*. Acestea sunt împărțite în două categorii: generale și de procesare. Opțiunile generale permit modificarea următorilor parametri:

Display – numărul monitorului care va fi procesat video, în cazul în care sunt mai multe conectate. Cel cu numărul 0 este ecranul principal.

Updates per seconds – câte capturi de ecran pe secundă să fie procesate și de câte ori să se transmită culorile la *Arduino*. Valorile posibile sunt 15, 25, 30, 45, 60. O valoare mai mare va crește fidelitatea culorilor, dar și puterea de procesare necesară.

Use brightness – dacă să se aplice sau nu modificarea intensității luminoase. Dacă nu este bifată, culoarea fidelă va fi redată.

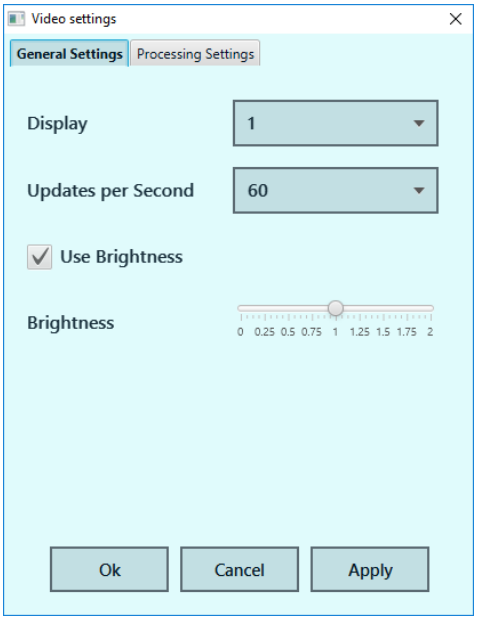

Fig. 11 - Opțiunile generale ale modului video

Brightness – dacă opțiunea de mai sus este bifată, atunci această bară glisantă este activă și permite reglarea intensității luminoase a LED-urilor. Valorea variază de la 0 la 2, cu semnificația că la 0 rama este stinsă, la 1 este culoarea fidelă, iar la 2 este intensitatea fidelă dublată.

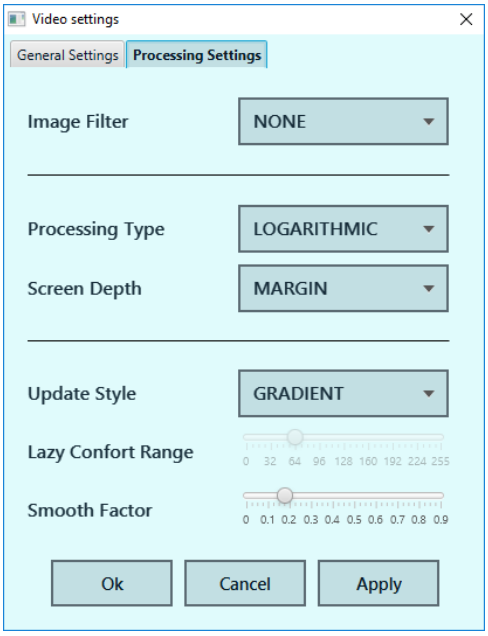

Fig. 12 - Opțiunile de procesare ale modului video

Opțiunile de procesare permit următoarele configurări:

Image Filter – ce filtru de culoare să se aplice asupra capturilor de ecran. Opțiunile posibile sunt *NONE*, *GRAYSCALE*, *RED*, *GREEN* și *BLUE*. Acestea vor constrânge rama să lumineze doar cu o singură culoare, a cărei intensitate va varia în funcție de conținutul de pe ecran.

Processing Type – ce algoritm de procesare al imaginii să fie folosit. UNIFORM va reda culoarea fidelă, dar cu un cost de procesare mai mare, în timp ce LOGARITHMIC va furniza o culoare aproape de cea fidelă, dar cu un cost de procesare (mult) mai mic.

Screen Depth – cât de adânc să se facă procesarea grafică. În cazul *MARGIN*, se analizează doar marginea ecranului (mai exact, pe lățime, adâncimea este numărul de pixeli ai lungimii ecranului / ***Width Led Count***, iar pe lungime numărul de pixeli ai lățimii ecranului / ***Height Led Count***), la *QUARTER* se analizează până la un sfert din ecran, iar la *HALF* se procesează până în centru.

Update Style – cum se realizează tranziția de la un cadru la altul. În modul *INSTANT*, la fiecare cadru, culoarea fidelă va fi redată pur și simplu, într-un mod destul de agresiv. *GRADIENT* va determina o tranziție mai lină între cadre. *LAZY* va determina schimbarea culorii actuale a LED-urilor doar dacă noua culoare este diferită, într-o anumită măsură, dar într-un mod agresiv. *LAZY GRADIENT* reprezintă o combinație dintre cele două tipuri, culoarea schimbându-se lin doar dacă aceasta diferă.

Lazy Confort Range – bară glisantă care determină gradul de diferență dintre două culori ca ele să fie considerate diferite. Modul de funcționare se bazează pe faptul că nuanța de culoare de la cadrul precedent este comparată cu cea actuală și se calculează diferențele în modul dintre cele 3 componente RGB. Culoarea se va schimba doar dacă toate cele trei diferențe sunt mai mici decât *Lazy Confort Range*. Valoarea 0 semnifică faptul că rama nu se va modifica vreodată, în timp ce, pentru 255, rama se va comporta ca în cazul *INSTANT*. Această opțiune este disponibilă doar dacă *Update Style* este *LAZY* sau *LAZY GRADIENT*.

Smooth Factor – bară glisantă care semnifică finețea cu care se realizează trecerea de la o coloare la alta, în cazul în care *Update Style* este *GRADIENT* sau *LAZY GRADIENT*. Această variabilă ia valori în intervalul [0, 1], unde 0 reprezintă faptul că nu se fac deloc tranziții, în timp ce 1 determină același comportament ca în cazul *INSTANT*. Valorile recomandate sunt între [0.1, 0.5].

2.4 Opțiunile modului audio

Opțiunile modului audio sunt accesibile din meniul principal, alegând opțiunea *Audio Settings* de la butonul *Settings*. Acestea sunt de două feluri: de bază și avansate. Setările de bază se concentrează strict pe partea audio și permit modificarea următorilor parametri:

Animation Style – tipul de animație pe care îl va întreprinde rama în tandem cu ritmul muzicii. Acesta poate fi de două feluri: *PULSE*, în care intensitatea culorilor LED variază în funcție de ritmul muzicii, și *WAVE*, potrivit căruia partea de jos a ramei va fi mereu luminată, în timp ce LED-urile de pe laterală se vor lumina de jos în sus, în mod sincronizat cu muzica.

Base Preset – buton cu valori multiple din care se poate alege un profil de culoare care va dicta aspectul ramei în timpul animațiilor.

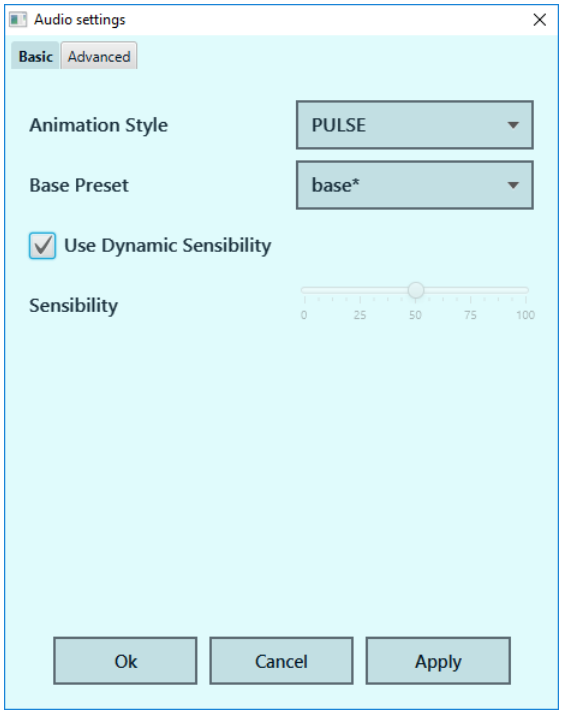

Fig. 13 - Opțiunile de bază ale modului audio

Sensibility – semnalul audio este analizat în așa fel încât se obține o metrică, numită *signal*, care determină ritmul muzicii la momentul actual (despre ce semnifică, mai exact, și cum este calculată, consultați secțiunea 4.4.5.2). Valoarea opțiunii *Sensibility* reprezintă intensitatea muzicii care va determina aprinderea maximă a LED-urilor. Volumul actual al muzicii influențează și el metrica *signal*, deci *Sensibility* trebuie modificat în așa fel încât să se țină cont de asta. Pentru uz obișnuit, valoarea implicită 50 este potrivită.

Use Dynamic Sensibility – bifarea acestei opțiuni va determina ajustarea automată a sensibilității în funcție de ritmul muzicii, în așa fel încât intensitatea luminoasă a LED-urilor să fie aproximativ la jumătate dacă volumul nu se schimbă semnificativ. Această opțiune este potrivită pentru atunci când valoarea 100 a *Sensibility* nu mai este de ajuns (în cazul unei petreceri, de exemplu).

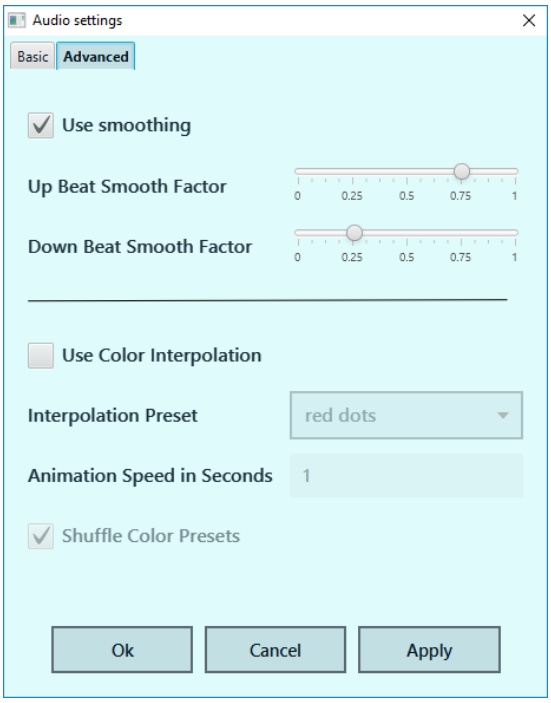

Fig. 14 - Opțiunile avansate ale modulului audio

Opțiunile avansate permit utilizatorului alterarea animațiilor de bază în anumite moduri, prezentate în cele ce urmează:

Use Smoothing – dacă această opțiune este bifată (ceea ce este recomandat), tranzițiile de la o intensitate la alta sunt modificate în așa fel încât să fie mai line. Dacă această opțiune nu este bifată, animațiile vor reda intensitatea curentă pur și simplu, ceea ce creează o clipire obositoare a LED-urilor.

Up Beat Smooth Factor – variabilă care determină cât de fin să se facă tranzițiile de la o intensitate mai scăzută spre una crescută. Se recomandă o valoare ridicată pentru ca animația să rămână sincronizată cu muzica.

Down Beat Smooth Factor – această metrică determină gradul de finețe al tranzițiilor de la o intensitate ridicată la una scăzută. Se recomandă o valoare scăzută pentru un efect plăcut, dar nu prea mică pentru a nu desincroniza animația.

Use Color Interpolation – dacă utilizatorul optează pentru această opțiune, atunci culorile LED-urilor se vor schimba, pe baza unei animații de interpolare (vezi 4.4.5.2), spre culorile unui alt profil, ales de la butonul cu alegeri multiple corespunzător opțiunii ***Interpolation Preset***, și înapoi la profilul inițial.

Animation Speed in Seconds – câmp de text în care se introduce, în secunde, durata animației de interpolare, asta în cazul în care este folosită. Aceasta semnifică durata de trecere de la un profil de culoare la un altul, deci revenirea la culoarea inițială se petrece în două intervale de timp.

Shuffle Color Presets – dacă această opțiune este bifată, atunci se va schimba profilul de culoare cu un altul existent în momentul în care animația de interpolare se termină. În acest mod, animația devine mai captivantă.

2.5 Opțiunile modulului ambiental

Opțiunile modulului ambiental pot fi accesate selectând *Ambient Settings* de la butonul *Settings* din meniul principal. Acestea sunt împărțite în două categorii: gestionarea profilelor și a animației curente. În ceea ce privește profilele, putem să îl selectăm pe cel curent de la butonul corespunzător opțiunii ***Selected Preset***. De asemenea, putem adăuga un profil nou, apăsând pe *Add*, sau putem șterge sau edita profilul selectat prin apăsarea butonului potrivit. Profilele de culoare marcate cu asterisc (*) vin preinstalate și sunt salvate în interiorul executabilului. Profilele create de utilizator sunt salvate, în format *JSON*, în directorul *Users/Documents*.

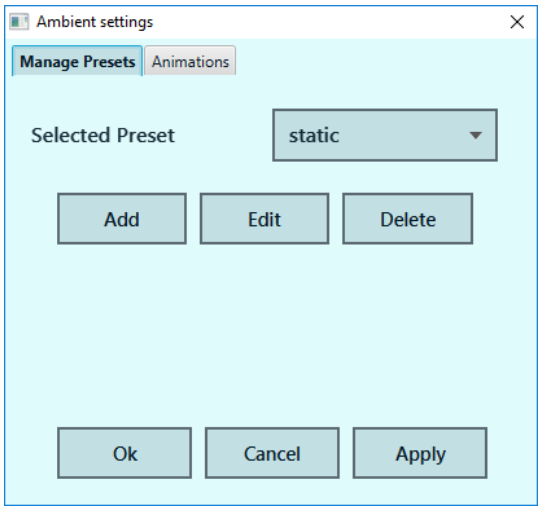

Fig. 15 - Meniul de gestiune al profilelor de culoare

2.5.1 Editarea unui profil

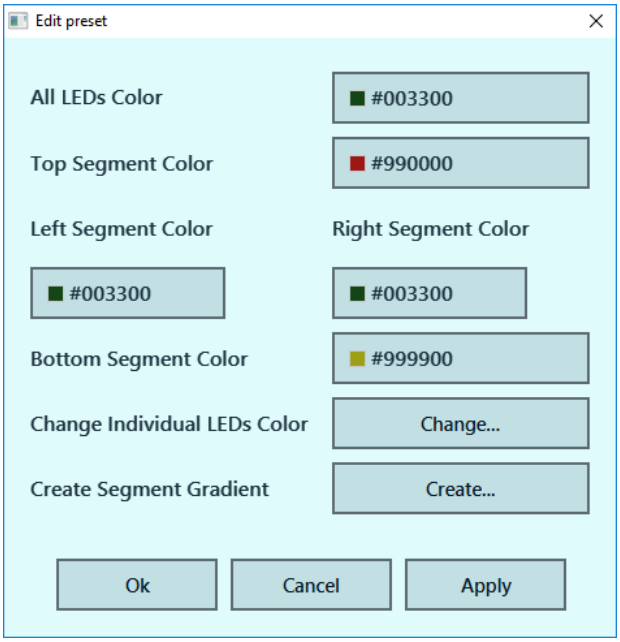

Fig. 16 - Opțiunile de editare ale unui profil

După ce a fost selectat profilul care se dorește a fi editat din lista de la ***Selected Preset***, se poate apăsa butonul *Edit* pentru a deschide meniul de personalizare. Acesta include următoarele opțiuni:

All LEDs Color – prin intermediul unui *color picker*, se poate alege o culoare care să fie asociată întregii rame;

Top, Left, Right, Bottom Segment Color – tot utilizând un *color picker*, se poate alege o culoare care să fie vizibilă doar pentru un anumit segment al ramei;

Change Individual LEDs Color – se va deschide o nouă fereastră, cât tot ecranul, care permite schimbarea culorii pentru fiecare LED în parte;

Create Segment Gradient – dacă se apasă pe butonul ***Create…,*** atunci se va deschide o fereastră prin intermediul căreia se pot aplica gradiente segmentelor ramei.

2.5.2 Editarea culorilor LED-urilor în mod individual

Acest meniu permite vizualizarea culorii fiecărui LED în parte. De asemenea, dimensiunea dreptunghiurilor semnifică și zona care va fi procesată de modul video, în cazul în care ***Screen Depth*** este setat pe *MARGIN*. Poziționarea LED-urilor pe ecran variază în funcție de numărul acestora și de construcția ramei.

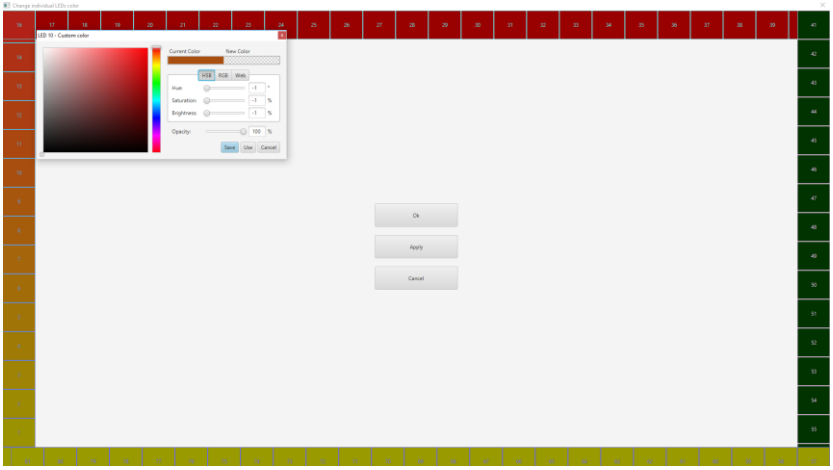

Fig. 17 - Fereastra de editare individuală a culorilor ramei

Dacă se apasă pe oricare dintre aceste dreptunghiuri, atunci se va deschide un *color picker* care permite schimbarea culorii LED-ului respectiv. În centrul ferestrei se află 3 butoane (*Ok*, *Apply* și *Cancel)* care au semnificația standard, prezentată la începutul capitolului.

2.5.3 Crearea gradientelor

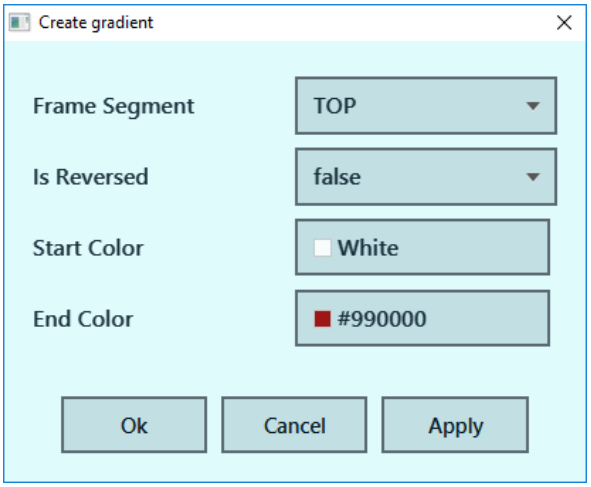

Fig. 18 - Meniul de creare al gradientelor

Meniul de creare al gradientelor permite noi moduri creative de a înfrumuseța aspectul culorilor LED-urilor ramei. Acest lucru se realizează prin intermediul următoarelor opțiuni:

Frame Segment – se alege din listă segmentul de ramă pentru care se dorește crearea unui gradient. Opțiunile disponibile sunt TOP, BOTTOM, LEFT și RIGHT;

Is Reversed – dacă se bifează această opțiune, atunci gradientul va fi creat contrar direcției ramei pentru care s-a optat în meniul *General Settings*, opțiunea ***Frame Direction***;

Start Color – utilizând un *color picker*, se alege culoarea de start a gradientului;

End Color – prin intermediul unui *color picker*, se optează pentru culoarea de sfârșit a gradientului.

În momentul în care se apasă butonul *Apply*, segmentul de ramă ales va fi luminat în așa fel încât primul LED al acestuia va avea culoarea de start, iar ultimul va avea culoarea de final, în timp ce restul vor face trecerea lină, în trepte, de la una la cealaltă.

2.5.4 Gestionarea animației curente

Din cadrul acestui meniu, se pot schimba anumite aspecte legate de înfățișarea modului ambiental:

Animation Style – specifică stilul animației curente. Se poate opta pentru STATIC (nicio animație), FADE (rama se va aprinde și se va stinge progresiv), INTERPOLATE (culorile curent se vor transforma treptat în culorile asociate unui alt profil) sau SPIN (culorile LED-urilor se vor interschimba între ele în așa fel încât par că se rotesc în sens trigonometric).

Speed in Seconds – cât de mult durează, în secunde, terminarea animației.

Interpolation Preset – în cazul animației de interpolare, specifică spre ce alt profil de culoare se va realiza trecerea.

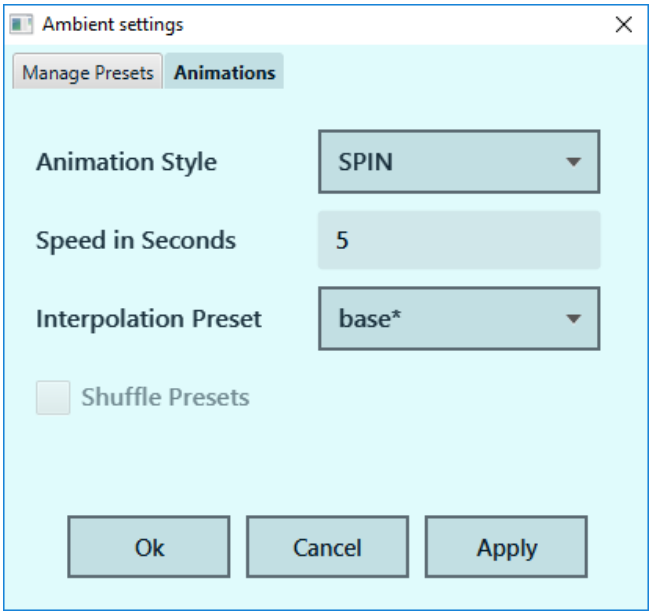

Fig. 19 - Setările animațiilor specifice modului ambiental

Shuffle Presets – bifarea acestei opțiuni va determina, pentru animații compatibile *FADE* și *INTERPOLATE*), schimbarea profilului de culoare cu unul aleator din cele existente în momentul terminării animației.

2.6 Concluzie

Toate aceste opțiuni de configurare permit personalizarea destul de amănunțită a modului în care sistemul de *lumină ambientală* funcționează. De asemenea, ușurința de utilizare a aplicației reiese din meniurile bine structurate și facil de folosit. În concluzie, obiectivele O1, O2 și O5 din introducerea lucrării sunt îndeplinite. În continuare, se va prezenta modul de implementare a componentei *hardware*.

3 Realizarea componentei hardware

3.1 Descrierea componentelor

3.1.1 Banda LED ws2812 5V

Această bandă este specială pentru că fiecare LED este individual adresabil, ceea ce permite o setare precisă a culorii, indiferent de poziție. Gama de culori conține 16 milioane de nuanțe posibile, codificate în stil RGB. O astfel de bandă are o densitate care variază de la 30 la 144 LED-uri/m (eu am folosit de 60). Cu cât sunt folosite mai multe LED-uri, cu atât intensitatea luminoasă este mai mare, dar și consumul este pe măsură. Un alt avantaj al acestei benzi este faptul că, în cele mai multe cazuri, poate fi secționată la fiecare un LED.

Folosirea LED-urilor este una extrem de avantajoasă. Acestea se aprind instantaneu, intensitatea luminoasă nu scade pe parcursul duratei de viață, lumina este confortabilă pentru ochi, durabilă (aprox. 50 mii de ore) și consumă puțin (0.3 watts/LED). (Gayral, 2017)

3.1.2 Arduino UNO

Arduino UNO este un *microcontroller* al cărui *hardware open source* permite realizarea de proiecte în domeniul *embedded* și *IOT*. Acesta vine dotat cu un procesor de 16 MHz, 2 KB SRAM, 1 KB EEPROM, 32 KB Flash Memory, și funcționează la un voltaj de ieșire de 5V, fapt ce garantează absența riscului de accidentare prin curentare în cazul utilizării neprielnice. Specificațiile nu sunt tocmai impresionante și justifică necesitatea procesării audio și video externe. De exemplu, o captură de ecran, în format *PNG*, de dimensiune *full HD*, ocupă, pe discul meu, 152 KB, deci nu ar putea fi încărcată în întregime în memoria SRAM a *microcontroller-ului*. Cu toate acestea, *Arduino UNO* are marele avantaj că nu are un sistem de operare preinstalat, deci codul încărcat pe acesta va rula fără întreruperi, ceea ce este ideal în cazul procesării în timp real.

3.1.3 Sursă de alimentare 5V 10A

Pentru ca atât Arduino, cât și rama, să funcționeze, este necesară alimentarea acestora. O soluție la îndemână este încărcarea de la portul USB al unui calculator, însă, în funcție de numărul de LED-uri, s-ar putea să nu fie de ajuns. Un singur LED are nevoie de 60mA pentru a se aprinde la intensitatea maximă. Portul USB al calculatoarelor moderne poate alimenta maxim 15 LED-uri, întrucât acesta oferă un curent de 900mA.

Rama pe care am construit-o are 82 LED-uri, în total acestea având nevoie de un curent de 82*0.06 = 4.92A. O sursă de alimentare de 5V și 10A este suficientă în acest caz, ea putând alimenta până la 166 LED-uri. Asta ne conferă versatilitate, întrucât putem să folosim o bandă de dimensiune variabilă. Este necesar ca sursa de curent să funcționeze la 5V, altfel ar duce la arderea benzii în cazul în care voltajul este mai mare.

3.1.4 Componente pentru construcția ramei

Aceste componente sunt necesare doar dacă se dorește construirea unei rame cu un aspect plăcut și care se poate atașa neinvaziv pe spatele unui monitor. Pe de altă parte, se poate folosi direct doar bandă LED care se prinde de spatele monitorului. Pentru avantajele și dezavantajele fiecărei dintre aceste metode, consultați secțiunea 2.2.1. Principiile de construcție ale ramei se bazează pe realizarea unei structuri solide, elegante, care permite proiectarea luminii la un unghi de 45° pentru un aspect mai plăcut. Cu scopul de a respecta aceste principii, se poate folosi:

- colțar de plastic cu latura de 1 cm;
- capac cu lățimea de 1cm de la bară de plastic de depozitat fire electrice;

- cablu sau un obiect cilindric cu diametrul de 0.5 cm (eu am folosit cablu Ethernet secționat longitudinal);
- pistol de lipit cu silicon pentru asamblare.

3.1.5 Unelte și alte ustensile

În mod absolut necesar, este nevoie și de:

- un pistol de lipit electric (letcon) cu vârf ascuțit, subțire și cu puterea de 40W;
- fludor pentru lipirea firelor;
- bucată de sacâz pentru a curăța vârful letconului și pentru a ușura, astfel, procesul de lipire;
- cablu de alimentare pentru sursă, care se poate conecta la priză;
- fire de tip *jumper* tată (care vor fi conectate în pinii plăcii *Arduino*);
- 2 fire electrice, puțin mai groase, pentru conexiunea dintre sursă și banda LED;
- voltmetru digital pentru măsurarea tensiunii sursei;
- șurubelniță;
- Bandă izolatoare.

3.2 Construcția sistemului

În această secțiune va fi prezentat pas cu pas modul de asamblare al sistemului, incluzând aici realizarea circuitului electric și construcția ramei.

3.2.1 Diagrama circuitului și asamblarea acestuia

Schema întregului circuit poate fi analizată mai jos și reprezintă componenta de maximă importanță a întregii componente *hardware*. Aceasta descrie conexiunea dintre *Arduino*, banda LED și sursa de curent. Deși în figură este desenat cu negru, cablu este, în realitate, alb, dar s-a optat pentru această culoare din motive de lizibilitate.

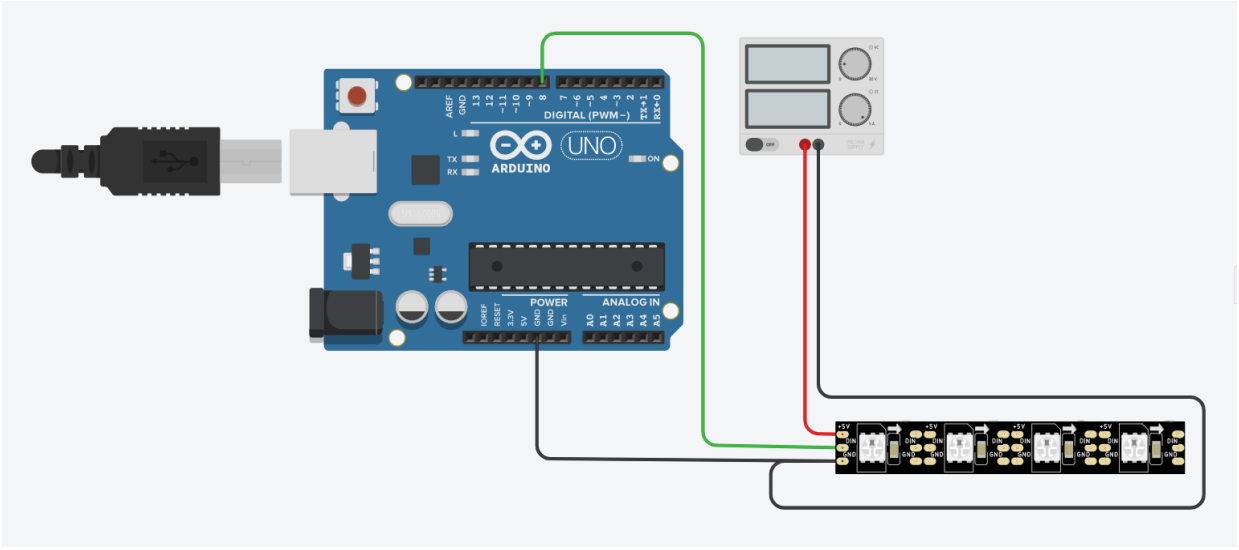

Fig. 20 - Diagrama circuitului[3]

Pentru a ajunge la acest circuit, se execută următorii pași:

[3] Realizată utilizând *www.tinkercad.com*

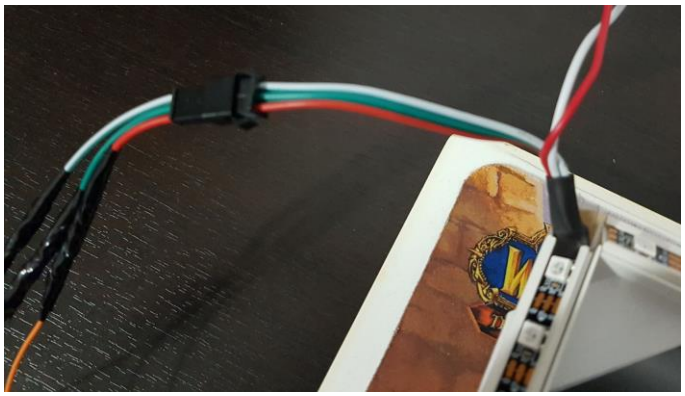

Fig. 21 - Conectorul benzii

Construirea întregului sistem începe cu analizarea benzii LED. Dacă este proaspăt cumpărată, la capătul inițial ar trebui să aibă 5 fire. Trei dintre ele sunt grupate în aceeași mufă și sunt de culoare roșie, verde și albă, iar celelalte două sunt libere, de culoare roșie și albă. Firele albe sunt prinse de pinul GND al benzii, cel verde de DIN, iar cele roșii de +5V.

Se identifică acest capăt și se verifică faptul că săgețile de pe bandă pleacă din el, deducând astfel că este cel inițial. Finalul benzii conține un conector cu trei fire de culoare roșie, verde și albă, care se poate conecta în mufa de la capătul inițial. Se secționează acest capăt întrucât poate fi folosit la ușurarea conexiunii dintre bandă și *Arduino*, și se conectează la cel inițial, prelungindu-l, așa cum este vizibil în Fig. 21.

Utilizând un letcon, se extind toate cele trei fire ale conectorului final în așa fel încât extremitățile sale să poată fi introduse în pinii *Arduino*. Firul cel alb se conectează în pin-ul GND al plăcuței, iar cel verde în pin-ul 8, prin intermediul căruia *Arduino* va controla banda LED. Extremitatea roșie, cea corespunzătoare pentru alimentare, nu trebuie conectată nicăieri, întrucât sursa este cea care are această responsabilitate. Dacă totuși se optează pentru alimentarea de la portul *USB*, atunci se conectează firul roșu la pin-ul 5V.

Se analizează sursa de alimentare cu scopul de a identifica cele cinci socluri ale sale. Privind imaginea de pe lateral, de la stânga la dreapta, aceste socluri se numesc: fază, nul, împământare, voltaj minus și voltaj plus. La capătul lor se află un potențiometru, care permite, cu ajutorul unei șurubelnițe, reglarea voltajului de ieșire. În cazul în care cablul de alimentare la priză nu este conectat deja la sursă, atunci trebuie secționat și îndepărtat tot plasticul de la toate cele trei fire ale acestuia.

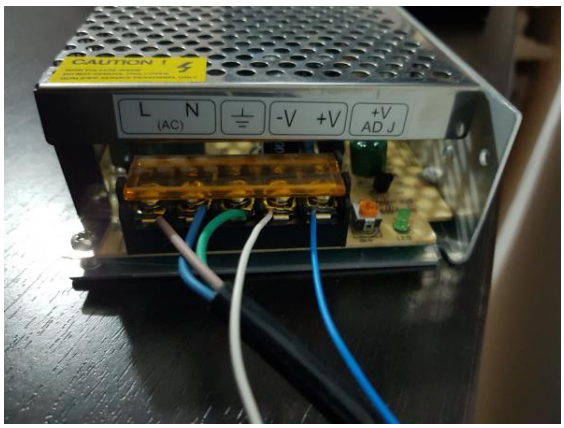

Fig. 22 - Soclurile unei surse

Culorile celor trei fire nu sunt întâmplătoare, ci sunt o convenție de culoare pentru a identifica rolul fiecăruia. Acestea nu sunt universal valabile și variază de la o regiune la alta. În imaginea de mai sus, cablul respectă convenția de culoare a Uniunii Europene, care prevede folosirea culorii maro pentru fază, albastru pentru nul și verde/galben pentru împământare. Fiecare fir este conectat la soclul potrivit și se folosește șurubelnița pentru fixarea acestuia. Odată conectat cablul în mod corespunzător, sursa se bagă în priză și se măsoară cu un voltmetru tensiunea de ieșire pentru soclurile V- și V+. Voltmetrul trebuie să indice exact 5V. Dacă nu, se calibrează voltajul de la potențiator folosind o șurubelniță cu mâner de plastic.

După ce sursa a fost calibrată, se scoate din priză și se conectează câte un fir la soclurile V- și V+. Este recomandat ca aceste fire să aibă culoarea albă pentru V- și roșie pentru V+ pentru a respecta convenția conectorului benzii. Firul V- de la sursă se conectează cu firul alb liber (corespunzător GND) al benzii, iar cel de la V+ cu firul roșu liber, cel de la +5V.

Orice conexiune cu rama se recomandă a fi făcută utilizând un mecanism în așa încât să fie ușor decuplabilă, eventual fără a utiliza o șurubelniță. Acest lucru garantează mai multă flexibilitate, întrucât orice componentă devine ușor interschimbabilă, în cazul în care se defectează. În plus, întreg sistemul devine mai ușor de relocat dintr-o parte în alta.

În acest moment, construcția de bază a sistemului este gata și se poate opta pentru o arhitectură a ramei de tip *LED_STRIP*, așa cum a fost descrisă în capitolul anterior. Pentru a realiza acesta, se calculează numărul de LED-uri dorit pe lungime **W** și pe lățime **H**, se taie o bucată de bandă de dimensiune 2 * (**W** + **H**) – 4 LED-uri și se atașează pe spatele ecranului, în formă de dreptunghi, utilizând bandă adezivă. O soluție mai elegantă va fi descrisă în secțiunea următoare.

Odată fixată rama de monitor, se conectează sursa la priză, apoi se introduce *Arduino* la o mufă *USB* a calculatorului. Se încarcă în memoria plăcuței codul din Anexa 1 – codul încărcat pe *Arduino*, utilizând un utilitar dedicat, precum *Arduino IDE*, apoi se pornește aplicația de procesare și se folosește conform modului descris în capitolul anterior.

3.2.2 Construcția ramei

În această secțiune se va prezenta un mod de construcție al unei rame de LED-uri care să fie estetică, și care să permită proiectarea luminii la un unghi de 45° față de normala părții dorsale a monitorului. Pentru realizarea ei sunt necesare componentele descrise în secțiunea 3.1.4. Rezultatul final poate fi consultat în imaginile de mai jos.

Fig. 23 – Rama, privită de sus

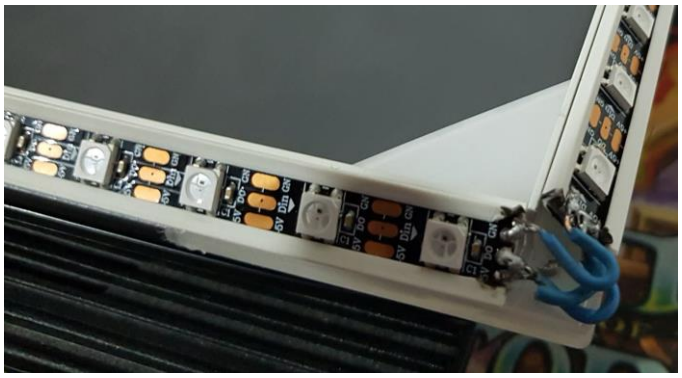

Fig. 24 - Colțul ramei, privit de aproape

Realizarea unei astfel de rame presupune îndeplinirea cu succes a următorilor pași:

- Se măsoară lungimea și lățimea monitorului în număr de LED-uri, respectând regula că laturile paralele sunt egale. În caz contrar, rama va ieși strâmbă sau nu vom avea cum să facem lipiturile între segmente. Se măsoară bucăți de capac de fire electrice în așa fel încât ar încăpea exact numărul de LED-uri necesare unei laturi. Se taie 4 astfel de bucăți;
- Se secționează bucătile de bandă LED necesare și apoi se glisează în aceste capace;
- Se măsoară 4 bucăți de colțar în așa fel încât au dimensiunea capacelor + 2 cm. De pe o parte și de alta se secționează câte 1 cm de la capetele aceleiași laturi de colțar, având grijă să nu se rupă complet;

- Se taie 4 bucăți din cablul ales, la dimensiunea capacelor. Se plasează cablul în interiorul colțarului, apoi se pune capacul pe deasupra și se analizează unghiul și estetica. Ar trebui să fie un unghi de 45 de grade, iar distanțele dintre capac si colțar sunt estetic neglijabile. Dacă nu, se face o tăietură de-a lungul cablului până se aduce la grosimea dorită, sau se folosește un alt tip mai subțire;

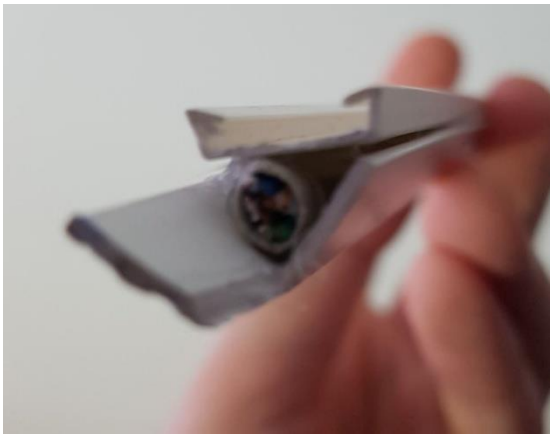

Fig. 25 - Asamblarea ramei

- Se încălzește pistolul de lipit cu silicon. Se lipește cablul în interiorul colțarului, apoi se lipește capacul deasupra, ca în Fig. 25;
- Se lipesc segmentele de ramă între ele, la unghi drept, având grijă ca bucățile de colțar nesecționate să se suprapună, iar sensul benzii să fie același. Pentru a le lipi la un unghi drept, puteți folosi o cutie sau colțuri de plastic care se utilizează la protecția ramelor de tablou din magazinele mari. Asigurați-vă că sunt lipite bine și că întreaga structură este stabilă;
- Utilizând un letcon și fire electrice, conectați segmentele de bandă între ele în așa fel încât pinii de la capătului unuia să fie lipiți de cei de la începutul următorului segment (+5V cu +5V, DIN cu DIN, GND cu GND). Capătul de final al ultimului segment nu trebuie lipit de primul segment, ci se va lăsa liber. Acest proces este unul destul de dificil și necesită multă răbdare. Înainte de a-l începe, verificați încă o dată ca sensul benzii segmentelor să fie același, precum și că întreaga structură este stabilă și că nu se mișcă;
- La finalul acestui procedeu, conectați primul LED al benzii la *Arduino* și la sursă, așa cum a fost descris în secțiunea anterioară. Rama este acum pregătită pentru a fi pusă în funcțiune. Dacă LED-ul sursei clipește intermitent sau nu toate segmentele sunt luminate, deconectați *Arduino* de la calculator și opriți imediat alimentarea sursei! Verificați ca toate lipiturile să fie corecte și că banda are același sens al LED-urilor, apoi încercați din nou.

3.3 Programarea microcontroller-ului Arduino

Această secțiune descrie partea de control *software* al întregului ansamblu *hardware*, incluzând aici tehnologiile folosite, structura protocolului de comunicare dintre *Arduino* și aplicația de procesare, precum și descrierea algoritmului încărcat pe *microcontroller*.

3.3.1 Tehnologii folosite

PlatformIO IDE – mediu de lucru *open source*, extensie pentru *Visual Studio Code*. Permite crearea și gestionarea de proiecte *embedded*, precum și încărcarea de cod C++ în memoria plăcii *Arduino*. Deși o alternativă la fel de viabilă și mai ușor de utilizat este *Arduino IDE*, am optat să folosesc prima variantă pentru că e mai comfortabil de editat codul sursă;

FastLED – bibliotecă *open source* care permite controlul direct și facil al benzii LED. Are suport pentru un număr foarte mare de modele de bandă, inclusiv cel care este folosit în cadrul acestui proiect (ws2812).

3.3.2 Protocolul de comunicare

Înțelegerea protocolului de comunicare dintre aplicația de procesare care rulează pe PC și programul de control al benzii LED de pe *Arduino* este esențială pentru a putea studia codul aferent

celui din urmă. Protocolul este destul de simplu și se bazează pe faptul că aplicația de procesare trimite comenzi astfel structurate:

HEADER || COMMAND_TYPE || COMMAND_PACKAGE

Semnificația este următoarea:

- HEADER – patru octeți corespunzători șirului de caractere “send”;
- COMMAND_TYPE – octet care memorează tipul comenzii. Deocamdată sunt două coduri:
  - 0 – LIGHT_COMMAND – această comandă determină aprinderea LED-urilor pe baza culorilor stocate în COMMAND_PACKAGE;
  - 1 – TURN_OFF_COMMAND – stinge toate LED-urile ramei, comandă folosită la comutarea dintre modurile de funcționare sau la oprirea aplicației de procesare;
- COMMAND_PACKAGE – parametru opțional care conține informații adiționale asociate comenzii curente, serializate ca un șir de octeți;
- || - operație care denotă concatenarea șirurilor de octeți.

COMMAND_PACKAGE pentru LIGHT_COMMAND este structurat în felul următor:

$$N0X_0Y_0Z_0 \dots iX_iY_iZ_i \dots N-1X_{N-1}Y_{N-1}Z_{N-1}, 0 \leq i \leq N-1$$

Cu următoarea semnificație:

- N – numărul de LED-uri care se dorește a fi aprins
- $iX_iY_iZ_i$ – indexul i al LED-ului curent și cei trei octeți de culoare corespunzători acestuia. Aceștia denotă cantitatea de roșu, verde și albastru a culorii, în ordinea unei permutări alese din aplicația de procesare. În cazul benzii LED folosite în acest proiect, ordinea acestor octeți trebuie să fie GRB pentru a obține culoarea fidelă.

După ce comanda este procesată, *Arduino* va trimite și el un octet de răspuns înapoi, dar doar în cazul comenzii TURN_OFF_COMMAND. Valoarea acestuia corespunde simbolului “&”.

3.3.3 Descrierea algoritmului

Codul de pe *microcontroller-ul Arduino* a fost scris în C++ și poate fi consultat în Anexa 1. În prima parte a acestuia sunt definite o serie de constante numerice precum:

- *DATA_PIN* – pinul Arduino corespunzător de controlul benzii LED. Se va modifica această valoare în așa fel încât să corespundă pinului în care este conectat firul verde (cel conectat de pinul DIN al benzii);
- *NUM_LEDS* – numărul de LED-uri al benzii. Valoarea ei se va modifica în funcție de numărul acestora. O valoare mai mare decât numărul de LED-uri actual este permisă, dar poate cauza întârzieri;
- *BRIGHTNESS* – intensitatea implicită a luminozității LED-urilor. Se recomandă valoarea maximă 255 pentru că se dorește controlarea acesteia prin intermediul aplicației de procesare;
- *HEADER_SIZE* – dimensiunea antetului folosit în cadrul protocolului de comunicare;
- *LIGHT_COMMAND* – valoarea octetului corespunzător acestei comenzi;
- *TURN_OFF_COMMAND* – similar ca mai sus.

Urmează apoi o serie de variabile globale, precum:

- *CRGB leds[NUM_LEDS]* – un tablou unidimensional, cu elemente de tip CRGB, pentru memorarea culorilor actuale ale benzii LED. CRGB este un tip de dată definit de biblioteca *FastLED* și reprezintă o culoare codificată în spațiul RGB, reprezentată pe trei octeți, câte unul pentru fiecare cantitate de roșu, verde și albastru;
- *Const uint8_t header[HEADER_SIZE]* – tablou unidimensional care conține literele antetului corespunzător comenzilor primite de la aplicația de procesare;
- Alte variabile care sunt folosite doar în interiorul anumitor funcții, dar au fost declarate globale pentru a evita supraîncărcarea memoriei de pe stivă.

Contrar programelor clasice scrise în C++, a căror execuție începe mereu cu funcția *main(),* cele specificei zonei *embedded* au o altă structură, bazată pe folosirea a două funcții *void,* fără parametri, care nu lipsesc niciodată: *setup()* (se execută o singură dată, la început) și *loop()* (codul acesteia buclează la infinit, până când se oprește alimentarea sau până cand *Arduino* este resetat). (Banzi & Shiloh, 2015)

În funcția *setup()* este inițializată biblioteca *FastLED* prin faptul că se asociază tabloul *leds*, având dimensiunea *NUM_LEDS*, cu banda LED de tip *WS2812*, controlată de *Arduino* prin pinul *DATA_PIN*. În plus, se inițializează pragul de referință pentru intensitatea luminoasă la valoarea *BRIGHTNESS,* din rațiuni explicate *a priori*. Totodată, se inițializează comunicarea serială prin apelul *Serial.begin(115200),* parametrul acestei funcții simbolizând viteza de transfer a datelor, exprimată în biți pe secundă (*baud rate*). Aceasta este valoarea maximă pe care o suportă *Arduino UNO,* iar unele cazuri nu este de ajuns. De exemplu, un pachet de date pentru luminarea unei rame cu 82 de LED-uri are dimensiunea: 4 (antetul) + 1 (tipul comenzii) + 1 (numărul de LED-uri) + 4 * 82 (4 octeți pentru fiecare LED – indice, roșu, verde, albastru) = 334 octeți. Dacă se trimit 60 de astfel de pachete pe secundă, atunci numărul de octeți crește la 334 * 60 = 20040, deci 160320 biți, număr care este mai mare decât viteza de transfer, ceea ce poate cauza întârzieri de până la 16 cadre.

În continuare, funcția *loop()* începe cu o buclă *while,* al cărei scop este citirea de pe portul serial și identificarea antetului unei comenzi. Un octet de pe comunicarea serială se citește utilizând funcția *Serial.read()*. Dacă s-a citit un octet care pare a fi început de antet de comandă, atunci se mai citesc, într-o buclă *for,* alte caractere, atât timp cât acestea sunt în ordinea din antet. Dacă octetul curent, citit ultimul, nu corespunde cu cel care ar trebui sa fie în antet, atunci a fost o alarmă falsă și se continuă așteptarea primirii unei secvențe care să corespundă. Pe de altă parte, dacă s-a citit o secvență antet, atunci următorul octet de pe portul serial este memorat în variabila globală *currentCommand*, acesta simbolizând codul unei comenzi. După aceea, se apelează funcția *solve_command()* care să soluționeze comanda curentă, apoi se iese din bucla *while*. Apelarea funcției anterior menționată are ca efect modificarea, într-un anume fel, a tabloului unidimensional *leds*, urmând ca un apel către *FastLED.show()* să ilumineze propriu-zis LED-urile ramei. Urmează apoi o linie de cod interesantă, și anume că se citește ce a mai rămas în *buffer-ul* de pe serial și se ignoră.

De ce se întâmplă acest lucru? Inițial, în dezvoltarea aplicației, protocolul de comunicare era mult mai simplificat. Se citeau caractere într-un *buffer* până la întâlnirea simbolului "|", care semnifica sfârșit de comandă. De exemplu, pentru a ilumina LED-ul 10 în alb, se trimitea comanda "10 255 255 255|" și se extrăgea fiecare număr din comandă folosind *strtok* și *atoi,* apoi se verificau să fie valide. Când indexul curent corespundea cu numărul de LED-uri al benzii, atunci se apela *FastLED.show()*. În acest mod, am observat că, în funcție de modul curent de funcționare, un număr de LED-uri de la începutul benzii nu se aprindeau, apoi urma un LED care avea culori ciudate, apoi restul erau normale, ceea ce era o problemă majoră. Căutând o soluție, am descoperit un articol, scris de către dezvoltatorii bibliotecii *FastLED,* despre faptul că citirea de pe serial nu mai este

blocantă cât timp sunt transmise date către bandă prin comanda *FastLED.show()*. Acest fenomen se întâmplă doar în cazul benzilor LED cu 3 fire, așa cum este modelul ws2812 folosit în proiect. Durata de transmisie a datelor pentru un singur LED este de 30µs, deci 2.46 milisecunde pentru 82, timp în care tot ce a fost transmis pe serial se pierde! Așa se explica modul ciudat de funcționare al ramei utilizând vechiul protocol. Se pierdeau datele corespunzătoare primelor LED-uri, apoi urma unul cu un comportament ciudat pentru că mecanismul de citire de pe serial își revenea în mijlocul comenzii corespunzătoare acestuia, ceea ce nu garanta recepționarea corectă a indexului. (Garcia, 2019)

Deși pentru acest model de bandă nu se poate preveni pierderea datelor, există un mecanism de corecție descris în articol și implementat de către mine. Mecanismul se bazează pe ideea de a prefixa pachetele de date cu un antet specific, iar după ce se apelează funcția *show()*, să se elibereze tot ce era în *buffer-ul* de pe serial, întrucât conținutul acestuia a fost corupt de către lipsa întreruperilor. Astfel, anumite pachete cu date despre LED-uri nu vor avea efect, dar dacă se transmit mai multe pe secundă, atunci pierderea calității efectului luminos este neglijabilă.

Revenind la explicarea codului sursă, soluționarea comenzii se realizează prin intermediul funcției *solve_command()*, care, pe baza codului memorat în variabila *currentCommand*, decide dacă să apeleze una din funcțiile corespunzătoare celor două comenzi posibile:

- *solve_light_command()* – conform protocolului, funcția citește un octet care corespunde numărului de LED-uri care urmează a fi aprinse, apoi se citește, în variabila globală *light_command_buffer*, pachetul de date care descrie culorile acestora. După aceea, prin intermediul unei bucle *for*, se parcurge conținutul *buffer-ului* și se extrag informațiile necesare actualizării tabloului *leds*;
- *solve_turn_off_command()* – se parcurge întreg tabloul *leds* cu scopul de a seta valoarea fiecărui element pe 0, corespunzător culorii negre. Această combinație va stinge rama în momentul apelării funcției *show()*. După execuția acestei comenzi, se va trimite simbolul "&" către aplicația de procesare pentru a marca îndeplinirea cu succes.

Folosirea acestei arhitecturi de comenzi, în combinație cu protocolul de comunicare stabilit, este una foarte avantajoasă pentru că permite multă flexibilitate. Adăugarea de comenzi noi este facilă, iar editarea uneia dintre acestea se realizează local, doar în corpul funcției, fără a schimba întreaga arhitectură.

3.4 Concluzie

În acest capitol a fost prezentat un mod de construcție și de control al unui sistem de *lumini ambientale*. Cu siguranță eixstă și alte moduri de implementare, dar conceptele de bază rămân aceleași: folosirea unei benzi cu LED-uri individual adresabile, alimentată de o sursă de curent și controlată de un *microcontroller*. În total, realizarea unui astfel de sistem, pornind de la zero, costă aproximativ 400 RON, incluzând în preț toate componentele și ustensilele menționate în subcapitolul 3.1. Folosirea unui letcon poate fi destul de dificilă la început, cu atât mai mult pentru că este necesar un grad mare de precizie. Considerând aceste aspecte, decizia că obiectivul O3 a fost îndeplinit aparține cititorului.

4 Implementarea aplicației de procesare

Aplicația de procesare este cea care rulează pe calculatorul utilizatorului și procesează, în timp real, capturi de ecran, sunetul captat de microfonul implicit sau trecerea timpului, cu scopul de a obține o listă de culori corespunzătoare LED-urilor benzii. Acestea sunt serializate și trimise la *Arduino*, pe baza protocolului descris în secțiunea 3.3.2. Procesarea este deterministă, adică un anumit semnal va produce mereu aceeași listă de culori, și, în plus, este flexibilă și personalizabilă, fapt ce se realizează prin folosirea unui mecanism de algoritmi interschimbabili. Întreaga implementare a aplicației, precum și tehnologiile folosite în acest proces, vor fi descrise pe parcursul acestui capitol.

4.1 Tehnologiile folosite

4.1.1 Java SE 8

Java este un limbaj de programare orientat obiect care facilitează dezvoltarea aplicațiilor *desktop*. Un rol important îl joacă *Java Virtual Machine*, care permite rularea acestei aplicații pe multiple sisteme de operare. Totodată, *IDE-urile* acestui limbaj (eu am folosit *IntelliJ IDEA 2018.1.6 Ultimate Edition*) facilitează procesul de dezvoltare și gestionare al codului aplicației.

4.1.2 JavaFX 2.0

JavaFX este un *API* media care permite dezvoltarea de interfețe grafice cu utilizatorul pentru aplicații desktop (și nu numai), care pot rula fără dificultăți pe multiple platforme. Această bibliotecă grafică este foarte versatilă și avantajoasă, din multiple motive:

- Interfața grafică este descrisă prin intermediul unui limbaj de marcare numit *FXML*. Astfel, deși JavaFX permite acest lucru, programatorul nu mai trebuie să implementeze aspectul interfeței și se poate concentra doar pe logica ei. Această separare crește flexiblitatea și robustețea codului;
- Implementează o mulțime de elemente grafice utile precum butoane, meniuri, grafice, câmpuri de text, *check box*, *choice box*, *color picker* etc;
- Stilul aplicației poate fi personalizat prin intermediul fișierelor de tip *CSS*;
- Dacă există, folosește placa video pentru a accelera randarea grafică. (Oracle)

4.1.3 SceneBuilder

SceneBuilder este o aplicație desktop care permite editarea fișierelor *FXML* în mod vizual. Procesul de implementare al interfeței, utilizând acest program, este unul facil și flexibil, pe bază de *drag and drop*. Permite previzualizarea rezultatului final, precum și conexiunea dintre fișierul *FXML* rezultat și clasa *controller* asociată.

4.1.4 Maven

Maven este o tehnologie care facilitează procesul de *build* al proiectului, dar și de gestiune al dependințelor externe. Am folosit *Maven* pentru a adăuga în proiect 3 librării, care vor fi descrise mai jos. Pentru a adăuga aceste biblioteci, am adăugat referința lor de pe *Maven Repository*[4] în fișierul *pom.xml*.

4.1.5 TarsosDSP

TarsosDSP (Digital Signal Processing) este o bibliotecă *open source*, pur Java, care facilitează procesarea în timp real a sunetului captat de microfon și nu numai. Funcționalitățile acesteia sunt

[4] https://mvnrepository.com/

exploatate în cadrul modulului Audio, așa cum vom vedea în cele ce urmează. Deși Java conține în mod implicit un *API* de captare și procesare audio (*Java Sound API*), acesta este lent și cauzează întârzieri de până la o secundă. *TarsosDSP* s-a dovedit a fi o soluție rapidă și eficientă, potrivită pentru procesarea în timp real.

4.1.6 Gson

Gson este o bibliotecă de la Google care permite serializarea și deserializarea obiectelor în format *JSON*. Acest lucru se realizează într-un mod ușor de folosit. Biblioteca a fost necesară pentru a salva în memoria externă profilele de culoare (în directorul *User/Documents*). Am ales acest format pentru a crește flexibilitatea în cazul dezvoltării unor viitoare funcționalități care ar permite modificarea stării ramei de la distantă, prin intermediul unui serviciu web, de exemplu.

4.1.7 JSSC

JSSC (Java Simple Serial Connector), este o librărie care oferă suport pentru comunicarea pe portul serial, folosită pentru a transmite și a recepționa date de la *microcontroller-ul* Arduino.

4.2 Arhitectura aplicației

Aplicația de procesare a fost dezvoltată folosind clasicul șablon de proiectare *MVC* (*Model - View - Controller*). Principalul avantaj al acestuia este reprezentat de separarea responsabilităților, fapt care permite o facilă gestionare a procesului de dezvoltare, cu atât mai mult cu cât este necesară și dezvoltarea unei interfețe cu utilizatorul.

Principalul element al *modelului* este modulul *Config*, acesta conținând o serie de proprietăți pe baza cărora funcționează cele trei moduri ale aplicației: video, audio și ambiental. În plus, tot în partea de model, există reprezentări pentru profilele de culoare, cât și pentru o ramă de LED-uri.

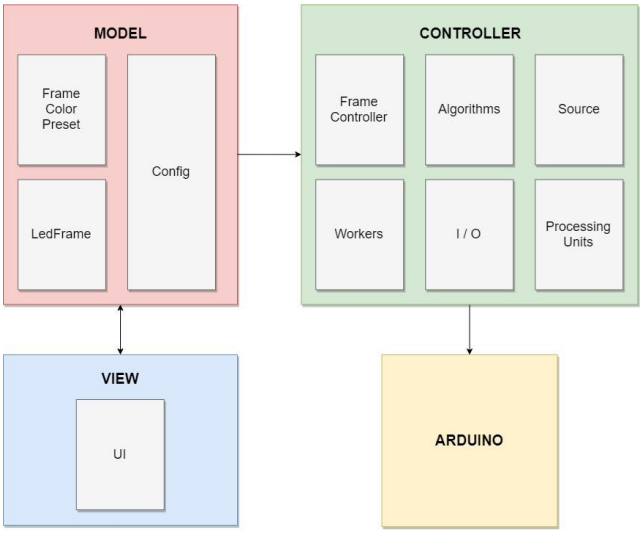

Fig. 26 - Arhitectura MVC a aplicației[5]

În ceea ce privește partea de *controller,* modulul *Workers* este cel mai important, întrucât acesta conține buclele principale ale programului, câte una pentru fiecare funcționalitate. Un *worker* este responsabil de captarea și serializarea semnalului curent, acest lucru făcându-se prin interacțiunea cu modulul *Source*. În plus, acesta creează și gestionează un *Frame Manager*, care face parte din modulul *Frame Controller*. *Frame Manager* joacă un rol fundamental în aplicație, și anume acela de a creea și gestiona unități de procesare (*Processing Units*), împreună cu un *LedFrame* (de la Model). Dacă un *LedFrame* este o reprezentare în memorie a culorii tuturor LED-urilor ramei, o unitate de procesare poate fi privită ca un înveliș de funcționalitate aplicat unui singur LED de la nivelul *hardware*, ea fiind responsabilă de gestionarea culorii acestuia.

Fiecărei unități de procesare îi sunt asociate algoritmi din modulul *Algorithms*, interacțiunea dintre aceste două module bazându-se pe șablonul de proiectare *Visitor*. Algoritmii de procesare sunt de trei feluri (preprocesare, procesare si postprocesare), și sunt împărțiți pe cele trei funcționalități ale aplicației. Astfel, se creează 9 clase de algoritmi care sunt gestionate de un *Algorithm Manager*.

[5] Atât diagrama aceasta, cât și toate celelalte în stil UML, au fost realizate utilizând https://www.draw.io/

Frame Manager este acela care declanșează funcția *process()* a unităților de procesare. Fiecare, rând pe rând, va rula algoritmii asociați, în ordinea preprocesare-procesare-postprocesare. În acest mod, se obțin noile culori ale ramei, care vor fi trimise la modulul *I/O (Input/Output)*. Acesta le serializează în mod corespunzător și le trimite la programul care rulează pe *Arduino*. În plus, modulul *I/O* gestionează și serializarea/deserializarea profilelor de culoare (*Frame Color Preset*) care pot fi create de setările modulului ambiental.

Interfața cu utilizatorul a fost creată folosind *JavaFX*. Fiecărei fereastre prezentată în capitolul 2 îi este asociată un document *FXML* care descrie înfățișarea ei, precum și o clasă *controller* pentru funcționalitatea acesteia. Principalul rol al interfeței este acela de a-i permite utilizatorului să gestioneze în mod facil parametrii modulului *Config*, cât și să comute între modurile de funcționare ale aplicației.

Arhitectura aplicației a fost gândită în așa fel încât extinderea ei cu noi funcționalități să fie ușor de realizat. Acest obiectiv a fost atins prin folosirea șablonului de proiectare *Visitor* între unitățile de procesare și algoritmii asociați acestora. Astfel, adăugarea unei animații sau a unei funcționalități noi ar presupune:

- Crearea unui algoritm de procesare care aparține unei clase corespunzătoare;
- Asocierea acestuia unui *Algorithm Manager* potrivit;
- Extinderea unui *factory* asociat unității de procesare căreia i s-ar potrivi noul algoritm;
- Introducerea de noi perechi cheie-valoare în modulul *config;*
- Extinderea interfeței grafice cu elemente de funcționalitate capabile să gestioneze noile configurări.

4.3 Model

4.3.1 Clasa LedFrame

Modelează rama de LED-uri. Are 3 date membre care o descriu, și anume numărul de LED-uri pe lățime, pe lungime, și o listă de obiecte de tipul *javafx.scene.paint.Color*, care reprezintă culorile LED-urilor actuale. Are un singur constructor care primește ca parametru două variabile de tip int *widthLedCount* și *heightLedCount*, semnificând numărul de LED-uri pe lungime și pe lățime.

4.3.2 Clasa FrameColorPreset

Modelează un profil de culoare, reținând despre acesta numele, culoarea întregii rame, precum și a fiecărui segment și LED în parte. Are, de asemenea, și o variabilă de tip boolean *isEditable* pentru a marca dacă acest profil poate fi editat sau șters în meniul de gestiune al profilelor de culoare. Toate datele membre, mai puțin variabila *isEditable*, sunt de tip String pentru a putea fi serializate și deserializate cu ușurință de către biblioteca *Gson*.

În interiorul clasei sunt și trei metode de gestionare al profilului: *saveColorState*, *deleteColorState* și *loadColorState*. Acestea interacționează cu clasa *FileManager* din modulul *io* cu scopul de a serializa, deserializa sau șterge fișierul *JSON* corespunzător profilului.

4.3.3 Modulul config

Principala clasă a modulului este *ConfigManager*. Aceasta implementează șablonul de design *Singleton* și conține, ca date membre, 4 obiecte statice corespunzătoare celor patru categorii de configurări ale aplicației: generale (*GeneralConfig*), video (*VideoConfig*), audio (*AudioConfig*) și ambientale (*AmbientConfig*). Constructorul clasei este privat, iar inițializarea celor patru obiecte se face prin apelarea unei metode statice *init()* în momentul în care este pornită aplicația. De asemenea, există patru accesori statici pentru obiectele cu configurări.

În interiorul modulului se află un pachet numic *config*, care conține patru clase, câte una pentru fiecare categorie de setări ale aplicației. Toate funcționează pe aceleași principii:

- Rețin perechi cheie-valoare. Fiecare clasă de configurare conține o listă de variabile *String*, de tipul *private static final*, care stochează numele cheilor. În acest mod, se evită greșeli de scriere a numelor acestora în momentul în care sunt folosite.
- Se bazează pe *Preferences API*. În acest mod, aplicația este ușor de transferat de la un utilizator la altul pentru că aceste preferințe sunt salvate pe disc, separat de aplicație. Înainte de a folosi *Preferences*, am folosit *Properties API* și fișiere de tipul *config.properties* în care memoram configurările, câte un fișier pentru fiecare categorie. Procedând astfel, a fost mai ușor să testez aplicația până să fac interfața grafică prin intermediul căreia să modific configurările. (Oracle)

Numele și conținutul acestora este descris în cele ce urmează.

4.3.3.1 GeneralConfig

Nume cheie configurare	**Tip de dată valoare**	**Valori posibile**	**Semnificație**
WIDTH	Integer	Numere naturale	Numărul de LED-uri de pe lungimea ramei.
HEIGHT	Integer	Numerere naturale	Numărul de LED-uri de pe lățimea ramei.
MODE	FrameMode (enum)	VIDEO, AUDIO, AMBIENT, NONE	Modul curent de funcționare al ramei.
CONSTRUCTION_TYPE	FrameConstruction Type (enum)	FRAME, LED_STRIP, SIDE_ONLY, LAPTOP_LED_STRIP, LAPTOP_FRAME	Tipul de arhitectură al ramei (dar nu al benzii LED, atenție!).
START_POSITION	FrameStartPosition (enum)	UPPER_LEFT, UPPER_RIGHT, LOWER_LEFT, LOWER_RIGHT	Colțul ecranului corespunzător începutului benzii LED.
DIRECTION	FrameDirection (enum)	CLOCKWISE, COUNTER_CLOCKWISE	Direcția săgeților de pe banda LED.
SERIAL_PORT	String	Nume de profil .json	Portul serial pe care se face transmisia datelor.
STRIP_RGB_ORDER	StrigRgbOrder (enum)	RGB, RBG, GRB, GBR, BRG, BGR	Ordinea de transmisie a octeților corespunzători unei culori.

4.3.3.2 VideoConfig

Nume cheie configurare	Tip de dată valoare	Valori posibile	Semnificație
DISPLAY	Integer	0, 1, … MAX_ECRANE	Id-ul monitorului caruia i se vor face capturi de ecran.
UPDATE_RATE	Integer	15, 25, 30, 45, 60	De câte ori pe secundă sunt trimise culorile procesate la *Arduino*.
PROCESSING_TYPE	VideoProcessingType (enum)	UNIFORM, LOGARITHMIC	Tipul algoritmului de procesare video.
USE_BRIGHTNESS_REGULATOR	Boolean	True/False	Memorează dacă utilizatorul vrea să aplice algoritmul de postprocesare responsabil de reglarea intensității luminii LED.
BRIGHTNESS	Double	[0, 2]	Multiplicatorul intensității luminoase, în cazul în care se foloseste algoritmul de reglare al intensității.
IMAGE_FILTER	VideoImageFilter (enum)	GRAYSCALE, RED, BLUE, GREEN, NONE	Dictează algoritmul de preprocesare video utilizat. NONE semnifică folosirea niciunuia dintre algoritmii de preprocesare.
UPDATE_STYLE	VideoFrameUpdateStyle (enum)	INSTANT, GRADIENT, LAZY, LAZY_GRADIENT	Specifică ce algoritm de postprocesare să fie utilizat după cel de reglare al intensității luminoase.
PROCESS_DEPTH	VideoProcessingDepth (enum)	MARGIN, QUARTER, HALF	Cât de adânc să se facă procesarea capturii de ecran.
SMOOTH_FACTOR	Double	[0, 1]	Finețea cu care se face tranziția de la o culoare la alta.
LAZY_COMFORT_RANGE	Integer	[0, 255]	Gradul de apropiere dintre culori pentru ca acestea să fie considerate din aceeași gamă de valori.

4.3.3.3 AudioConfig

Nume cheie configurare	Tip de dată valoare	Valori posibile	Semnificație
PROCESSING_TYPE	AudioProcessngType (enum)	PULSE, WAVE	Tipul de animație ritmică al LED-urilor.
FRAME_PRESET	String	Nume de profil .json	Numele unui profil care va dicta culorile de bază ale ramei în timpul animațiilor.
DYNAMIC_SENSIBILITY	Boolean	True/False	Dacă să se folosească sau nu algoritmul de calibrare automată a sensibilității audio.
SENSIBILITY	Double	[0, 100]	Frecvența maximă acceptată, prag în funcție de care se stabilește intensitatea culorii LED-urilor.
RELAXED	Boolean	True/False	Dacă să se folosească sau nu algoritmul de relaxare al tranzițiilor de la o intensitate luminoasă la alta.
UPBEAT_SMOOTH_FACTOR	Double	[0, 1]	Cât de fin să se realizeze tranziția de la o intensitate scăzută la una crescută.
DOWNBEAT_SMOOTH_FACTOR	Double	[0, 1]	La fel ca mai sus, dar de la crescut la scăzut.
INTERPOLATION	Boolean	True/False	Dacă să se interpolze sau nu culorile LED-urilor de la un profil la altul.
FRAME_INTERPOLATION_PRESET	String	Nume de profil .json	Numele unui profil de culoare spre care să se facă interpolarea, pornind de la FRAME_PRESET.
BASE_ANIMATION_SPEED	Integer	Multiplu de 1000, mai mare ca 0	Cât de multe milisecunde durează interpolarea de la un profil la altul.
SHUFFLE_PRESETS	Boolean	True/False	Dacă să se schimbe profilul de culoare în momentul terminării interpolării sau nu.

4.3.3.4 AmbientConfig

Nume cheie configurare	Tip de dată valoare	Valori posibile	Semnificație
ANIMATION_STYLE	AmbientAnimationStyle (enum)	STATIC, FADE, INTERPOLATE, SPIN	Tipul de animație al LED-urilor.
ANIMATION_SPEED	Integer	Multiplu de 1000, mai mare ca 0	Cât de mult durează, in milisecunde, animația.
FRAME_PRESET	String	Nume de profil .json	Numele unui profil care dictează culorile LED-uril pe parcursul animațiilor.
INTERPOLATION_PRESET	String	Nume de profil .json	Numele unui profil de culoare spre care se realizează animația de interpolare, în cazul în care animație este de așa natură.
SHUFFLE_PRESET	Boolean	True/False	Dacă să se schimbe sau nu profilul de culoare cu unul aleator din cele existente în momentul terminării unei animații compatibile.

4.4 Controller

4.4.1 Modulul frame

Clasa de importanță majoră a acestui modul este *FrameManager*. Aceasta gestionează asocierea dintre un obiect de tip *LedFrame* (vezi 4.3.1) și o listă de unități de procesare (*ProcessingUnit*, vezi 0). Rolul său fundamental este acela de a declanșa execuția algoritmilor asociați unităților de procesare stocate în lista *processingUnits*, cu scopul de a obține culorile pentru LED-urile ramei, memorate în *ledFrame*. Acest lucru se realizează prin funcția *update()*, care este apelată din interiorul modulului *workers*.

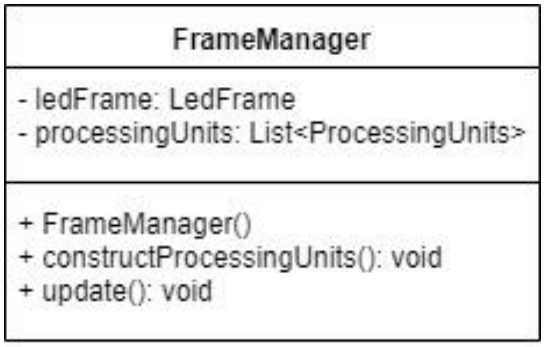

Fig. 27 - Structura unui FrameManager

Constructorul *FrameManager()* se folosește de o altă clasă pentru a inițializa *ledFrame*, și anume de *LedFrameBuilder*. Aceasta are o metodă publică numită *build()*, fără niciun parametru, care extrage din modulul *config* informații despre arhitectura ramei, precum tipul de construcție, numărul de LED-uri pe lungime și pe lățime, și construiește un obiect de tipul *LedFrame,* care conține o listă de elemente de tip *Color*. În realizarea unui *LedFrame*, *LedFrameBuilder* se folosește de clasa *LedCountComputer*, responsabilă de calcularea numărului de LED-uri necesare pentru arhitectura curentă.

De asemenea, un *FrameManager* are și metoda *constructProcessingUnits()* pentru a construi lista *processingUnits*. Acest lucru se realizează prin intermediul unei clase ajutătoare numită *ProcessingUnitsArrayFactory,* care are o singură metodă publică, și anume *build()*. Metoda *build()* nu are parametri și returnează o listă cu elemente de tip *ProcessingUnit.* Pentru a realiza acest lucru, inițial extrage din modulul *config* date referitoare la direcția ramei, precum și poziția primului LED, dar și tipul de construcție al ramei *FrameConstructionType*. După aceea, calculează un indice de start, numit *startingPoint,* și decide, pe baza tipului de construcție, instanțierea unui *StyleFactory* specific care să construiască lista de unități de procesare. Descrierea acestora, precum și semnificația și modul de calculare al indicelui de start, sunt prezentate în cele ce urmează.

4.4.1.1 Construcția listei de unități de procesare

StyleFactory este o clasă abastractă care se ocupă de construcția de liste de unități de procesare. Are o singură metodă publică, *buildFrame(),* care este apelată din interiorul metodei *build()* a clasei *ProcessingUnits ArrayFactory*. De asemenea, clasa are mai multe variabile protejate, pe baza cărora se construiesc unitățile de procesare. Valoarea lor, însă, este strâns legată de modul de construcție al ramei ales.

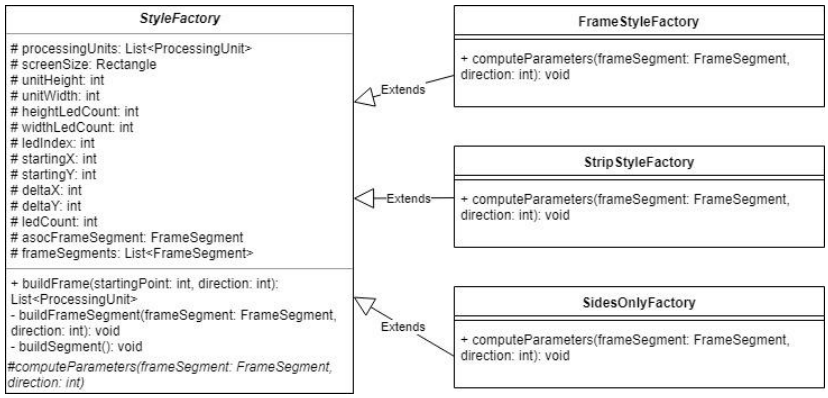

Fig. 28 - Arhitectura StyleFactory

Din această cauză, *StyleFactory* are o singură metodă privată, *computeParameters()*, implementată de toate clasele care o extind: *Frame, Laptop, LaptopStrip, SidesOnly, Strip – StyleFactory.* Rolul funcției este acela de a inițializa variabilele clasei de bază.

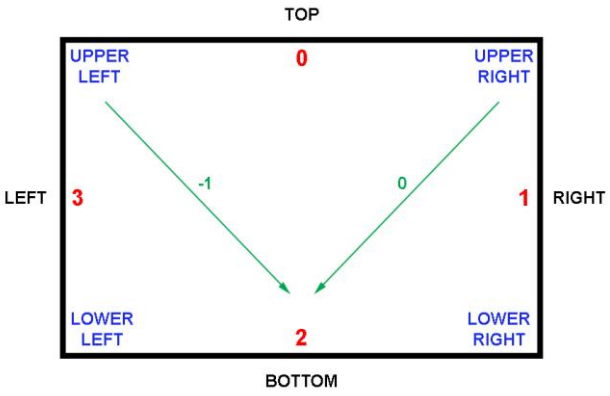

Fig. 29 - Calcularea punctului de start

Modul de funcționare al metodelor din clasa *StyleFactory* este în strânsă legătură cu valoarea variabilei *startingPoint*. Pentru a înțelege cum este calculată, priviți Fig. 29. Se dorește aflarea numărului roșu. Din modulul *config* se citește valoarea colțului de start și se inițializează valoarea variabilei *startingPoint* cu primul număr roșu care urmează, în sensul acelor de ceas. Se citește apoi, tot din modulul *config*, direcția generală a benzii. Dacă aceasta este conform acelor de ceas, atunci la valoarea variabilei *startingPoint* se adaugă 0, altfel se scade -1, cu mențiunea că dacă se obține -1, atunci se modifică în 3.

StyleFactory conține, ca dată membru, o listă cu obiecte enum de tip *FrameSegment*. Fiecare clasă derivată va inițializa în constructor această listă, în funcție de arhitectura ei specifică, astfel: *Frame + Strip StyleFactory (Top, Right, Bottom, Left), Laptop + LaptopStrip (Top, Right,* ***null****, Left)*, și *SidesOnlyStyleFactory* (***null****, Right,* ***null****, Left*). Metoda *buildFrame()* din clasa *StyleFactory* va parcurge această listă, circular și în întregime, începând cu poziția *startingPoint*, spre dreapta, dacă direcția este conform acelor de ceas, sau spre stânga, în caz contrar. Pentru fiecare element al listei, va apela funcția *buildFrameSegment()* care, la rândul ei, va apela metoda *computeParameters()*, apoi *buildSegment()*. Metodele de tip *computeParameters()* primesc ca parametru un obiect enum de tip *FrameSegment* și direcția generală a benzii, apoi inițializează poziția punctului de start al zonei de ecran care va fi procesată de către unitățile asociate (*startingX, startingY*), câte LED-uri sunt (*ledCount*), cum variază coordonatele X și Y de la unitate la alta (*deltaX, deltaY*), dar și care este segmentul de ramă care urmează a fi analizat (*associatedFrameSegment*).

Metoda *buildSegment()* este aceea care va construi, propriu-zis, unitățile de procesare, câte un segment de ramă o dată. În funcție de modul de funcționare curent, extras din modulul *config*, va inițializa un *ProcessingUnitFactory* specific, apoi, într-o buclă *for*, pentru fiecare LED asociat acelui segment, va apela metoda *getUnit()*, obținând o unitate de procesare care va fi adăugată în lista *processingUnits()*. Această listă reprezintă rezultatul final al metodei *buildFrame()*, care va fi returnat către clasa apelantă *ProcessingUnitsArrayFactory*, ajungând, în cele din urmă, ca dată membră a clasei *FrameManager()*.

4.4.2 Modulul processing_units

Așa cum a fost prezentat în subcapitolul 4.2, unitățile de procesare sunt un înveliș de funcționalitate aplicat fiecărui LED. Acestea sunt responsabile de modificarea culorii LED-ului, pe baza algoritmilor atașați. Ele sunt de trei tipuri, câte unul pentru fiecare funcționalitate a aplicației: video, audio și ambiental. Modulul *processing_units* este divizat în două pachete: *units* și *factories*. Primul conține clasele asociate acestor unități de procesare, iar al doilea implementează șablonul de proiectare *Factory Method* pentru a controla instanțierea acestora.

4.4.2.1 Pachetul units

Principala clasă a pachetului *units* este *ProcessingUnit*, o clasă abstractă care are ca date membre informații despre LED, precum indexul, segmentul de ramă asociat și culoarea curentă, dar și trei liste cu elemente de tipul *PreProcessingAlgorithm*, *ProcessingAlgorithm* și *PostProcessingAlgorithm*.

Metoda principală a acestei clase este *process()*, care determină aplicarea algoritmilor de procesare asociați fiecărei unități în parte. Se parcurg și se execută, în ordine, algoritmii de preprocesare, apoi cei de procesare și, în final, cei de postprocesare. Acești algoritmi primesc o referință la unitatea curentă și îi modifică anumite date membre în funcție de natura lor. În urma aplicării tuturor algoritmilor, rezultă o nouă culoare curentă care va fi returnată spre *Frame Manager*, cu scopul de a fi stocată în *Led Frame*.

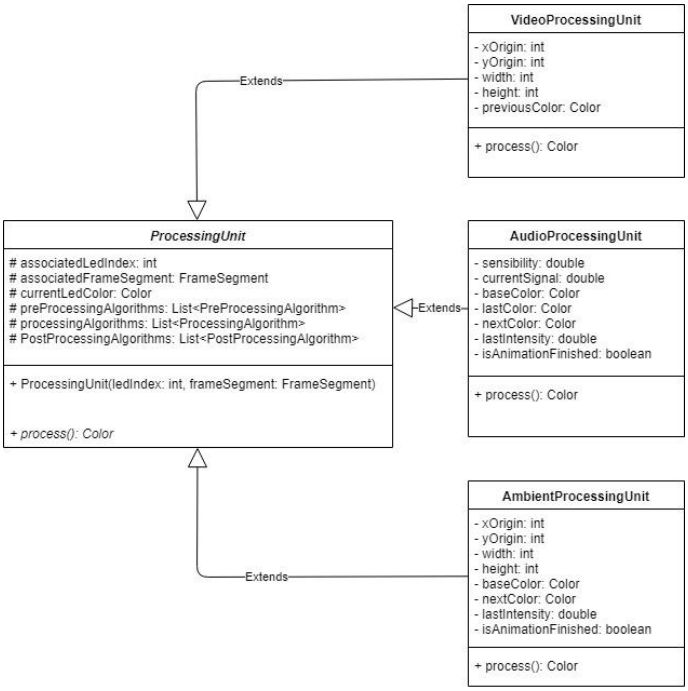

Fig. 30 - Arhitectura unităților de procesare

VideoProcessingUnit – este derivată din *ProcessingUnit* și conține, în plus, informații despre sectorul de imagine asociat. Fiecare unitate de procesare video este responsabilă de prelucrarea unei bucăți din captura de ecran curentă, cu scopul de a obține culoarea LED-ului. În mod evident, aceste sectoare de imagine sunt alese în așa fel încât să fie în apropierea LED-ului de pe rama fizică, memorându-se despre acestea poziția colțului din stânga-sus (*xOrigin* și *yOrigin*), precum și dimensiunea (*width* și *height*). Inițial se memora chiar bucata de imagine, într-o variabilă de tip *BufferedImage*, dar s-a renunțat la această variantă pentru a eficientiza consumul de memorie, nefiind necesară duplicarea informației. Variabila *previousColor* stochează culoarea precedentă a LED-ului și este relevantă pentru algoritmii de postprocesare *FrameUpdate*.

AudioProcessingUnit – conține informații necesare aplicării algoritmilor audio. *CurrentSignal* este o variabilă de importanță majoră, aceasta indicând valoarea curentă a *metricii Signal* (vezi 4.4.4), pe baza căreia se calculează intensitatea luminoasă a LED-urilor în cadrul algoritmilor de procesare. Similar, *sensibility* dictează valoarea pe care trebuie să o ia *currentSignal* pentru a fi considerat semnal de maximă intensitate. *BaseColor* reprezintă culoarea de bază a LED-ului curent, extrasă dintr-un profil de culoare. *LastColor* memorează culoarea precedentă a LED-ului, informație necesară pentru algoritmul de postprocesare *Relaxing Pulse*. *NextColor* stochează următoarea culoare spre care se face interpolare, în cazul în care s-a optat pentru acest algoritm de preprocesare. *LastIntensity* este un câmp care memorează care a fost stadiul animației de interpolare la ultima actualizare a ramei, informație vitală pentru a determina dacă această animație s-a terminat, folosită tot de algoritmul responsabil de această animație. *IsAnimationFinished* este un indicator pentru a determina dacă animația de interpolare s-a sfârșit, acest fapt fiind necesar în cazul în care s-a optat pentru schimbarea profilelor de culoare la final de animație.

AmbientProcessingUnit – îmbină date membre de la ambele unități de procesare de mai sus. Ca și unitățile video, se memorează poziția și dimensiunea sectorului de image asociat, dar nu pentru a interacționa cu captura de ecran, ci pentru a desena aceste zone în fereastra specifică setărilor *Edit Individual LED Color* (vezi 0). În plus, se stochează date care apar și la unitățile audio, scopul lor fiind identic, și anume acela de control al animațiilor.

4.4.2.2 Pachetul factories

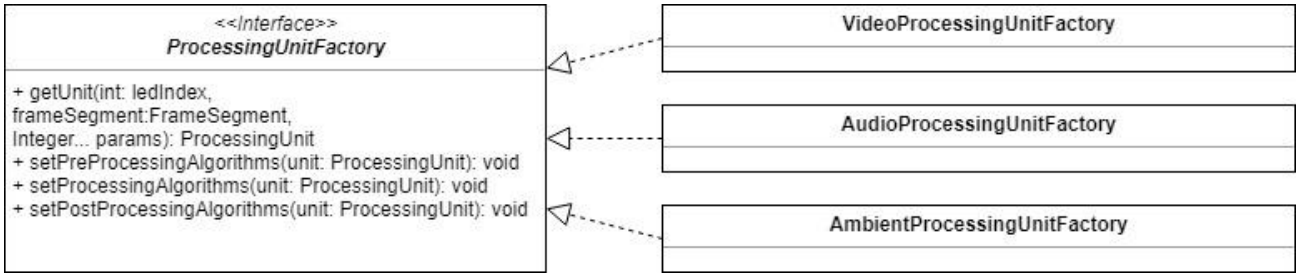

Fig. 31 - Arhitectura ProcessingUnitFactory

Instanțierea unităților de procesare este realizată prin intermediul șablonului de proiectare *Factory Method*, întrucât nu se poate ști dinainte tipul unităților de care este nevoie. În plus, acest tip de arhitectură permite adăugarea facilă de noi tipuri de unități de procesare în viitor. (Gamma, Helm, Johnson, & Vlissides, 1995) (Shvets, 2013)

Întreaga structură se bazează pe interfața *ProcessingUnitFactory*, metoda ei principală fiind *getUnit()*, care primește ca parametru indexul LED-ului, precum și segmentul de ramă asociat, dar și un număr variabil de parametri de tip întreg, stocați în *params*. De asemenea, există trei metode de adăugare ai algoritmilor de preprocesare, procesare și postprocesare asociați, care vor fi apelate în interiorul funcției *getUnit()*.

Există trei clase care implementează această interfață, câte una pentru fiecare tip de unitate de procesare. Un caz special este reprezentat de unitățile video și ambientale, ale căror metodă *getUnit()* extrage valori din lista *params* dată ca parametru. În ordine, aceste valori trebuie să corespundă variabilelor *xOrigin, yOrigin* (colțul din stânga sus al sectorului de imagine procesat de unitatea aferentă), *width* și *height* (dimensiunile sectorului). Aceste variabile sunt necesare în apelarea constructorilor potriviți.

Metodele de setare ai algoritmilor urmează aceeași structură, și anume că se extrag date referitoare la tipul acestora din modulul *config*, apoi se folosește clasa *AlgorithmManager* pentru a obține instanțele dorite, urmând ca apoi să le și a le adăuga în lista lor aferentă. Mecanismul este descris mai jos, în secțiunea 4.4.5.4.

4.4.3 Modulul source

Clasa principală a acestui modul este *SourceManager* care, prin intermediul șablonului de proiectare *Singleton*, pune la dispoziția utilizatorului obiecte care implementează interfața șablon *Source<T>*. Scopul acesteia este acela de a oferi o descrie a modului în care ar trebui să arate o clasă care gestionează o sursă de informație care poate fi procesată cu scopul de a obține o listă de culori ale LED-urilor. Clasele care o implementează sunt *VideoSource, AudioSource, TimeSource* și *ColorPresetSource*, fiecare folosite în a alimenta cu date de intrare modurile de funcționare. Structura acestui modul poate fi analizată în imaginea de mai jos.

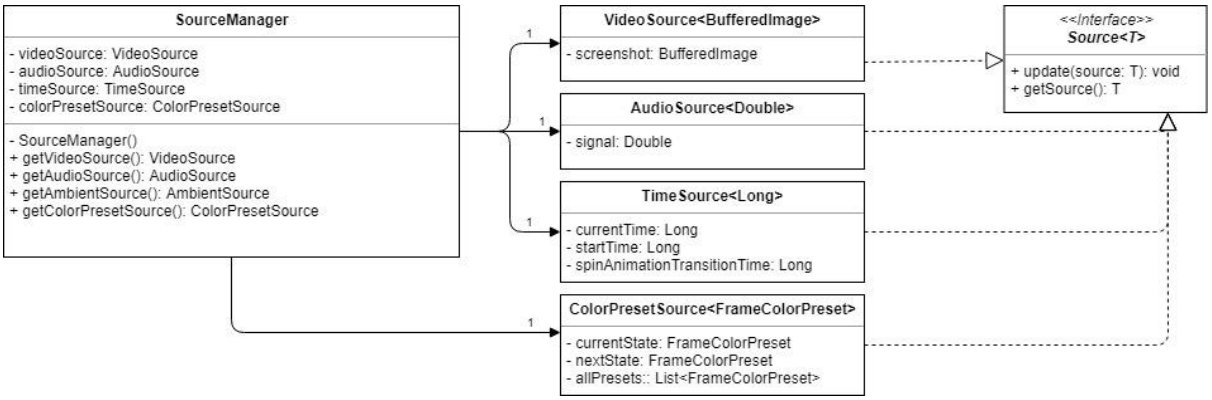

Fig. 32 - Arhitectura modulului source

Metoda *update()* este declanșată doar de către un *worker* care ar avea nevoie de acel tip de dată, în timp ce *getSource()* este folosită de algoritmi sau de unitățile de procesare care trebuie să-și actualizeze anumite date membre înainte de execuția algoritmilor asociați lor.

VideoSource stochează o singură imagine corespunzătoare ultimei capturi de ecran realizate. Java are suport pentru realizarea capturilor de ecran prin intermediul clasei *java.awt.Robot*. Aceasta are o funcție numită *createScreenCapture()* care primește ca parametru un dreptunghi corespunzător zonei de ecran care se dorește a fi capturată, și returnează un obiect de tip *BuffereImage* care conține imaginea rezultată.

Mecanismul de capturi de ecran al aplicației este gestionat de clasa *ScreenshotTaker*, conținută în modulul *source*. Ca date membre, aceasta conține un obiect *Rectangle* care denotă dimensiunile dorite pentru captura de ecan, precum și o instanță a clasei *Robot*. Acestea sunt inițializate în constructorul clasei, care, mai întâi, extrage indexul *displayID* al ecranului pentru care se dorește captura, precum și dimensiunile lui, stocate în obiectul *Rectangle*, printr-un apel către:

```
GraphicsEnvironment.getLocalGraphicsEnvironment().getScreenDevices()[displayId].getConfigurations()[0].getBounds();
```

Clasa conține metoda publică *take()*, al cărei singur rol este returnarea imaginii *BufferedImage* rezultate în urma apelului *createScreenCapture()*.

4.4.4 Modulul *workers*

Acest modul este cel mai important din toată aplicația, întrucât acesta conține buclele principale ale programului. Clasa principală este *MainWorker* și implementează șablonul de proiectare *Singleton*. Aceasta conține un obiect de tip *Worker*, numit *activeWorker*, care stochează o instanță a unui *VideoWorker*, *AudioWorker* sau *AmbientWorker*, în funcție de modul de funcționare curent. Metoda statică *start()* este apelată în funcția *Main()* a programului, întrucât se dorește începerea procesării încă de când se pornește aplicația, și are ca scop extragerea modului curent din modulul *config*, pe baza căruia instanțiază un *worker* corespunzător. După aceea, creează un nou fir de execuție, printr-un apel *new Thread(activeWorker).start()*, care va începe procesarea corespunzătoare modului dorit. Acest lucru este posibil deoarece interfața *Worker* este extinsă din *Runnable*. Funcția *stop()* din *MainWorker* apelează metoda *stop()* din *activeWorker*, apoi îl egalează cu *null*. Metoda *restart()* face un apel către *stop()*, apoi *start()*, și este apelată de fiecare dată când în interfață se apasă butonul *Ok* sau *Apply*.

Urmând această arhitectură, se asigură faptul că doar un singur *worker* funcționează la un moment dat. Este necesar ca aceștia să ruleze pe un fir de execuție separat pentru a nu bloca interfața grafică. Mai jos este o diagramă UML care descrie clasele acestui modul:

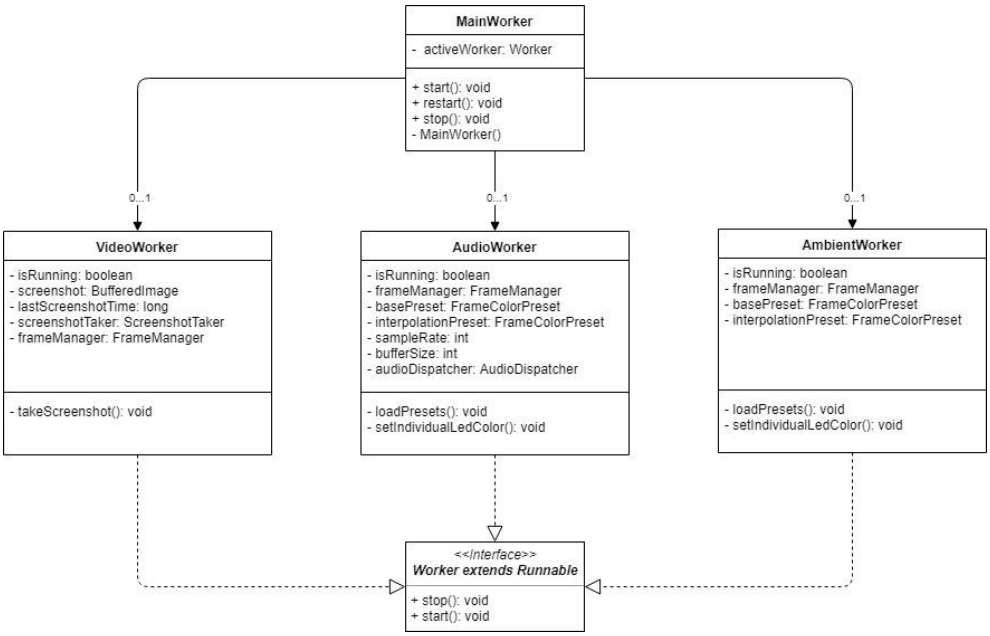

Fig. 33 - Ierarhia și arhitectura claselor modulului workers

VideoWorker este responsabil de întreaga procesare video, conținută în interiorul funcției *start()*. Codul acestuia este inclus în Anexa 2 – VideoWorker. Aceasta începe cu o etapă de inițializare în care se citește rata de actualizare *updateRate* din modulul *config*, semnificând de câte ori pe secundă se analizează ecranul și se trimite către *Arduino* culorile de pe marginea ecranului, apoi calculează *refreshRate*, adică numărul de milisecunde dintre aceste actualizări. După aceea, instanțiază un obiect de tip *FrameManager*, precum și un *ScreenshotTaker*, din modulul *source*. Se construiesc, apoi, unitățile de procesare cu un apel *frameManager.constructProcessingUnits()*. În final urmează o buclă *while*, care rulează cât timp *isRunning* este *true*, fapt ce se va întâmpla până la un apel *MainWorker.stop()*. Întreg codul buclei este pus într-un *if* care se asigură că ecranul este analizat doar la intervale de timp cel puțin egale cu *refreshRate*. Acest lucru se verifică prin faptul că diferența dintre durata de timp, în milisecunde, dintre momentul actual (*System.currentTimeMillis()*) și când s-a făcut ultima dată o captură de ecran (*lastScreenshotTime*) să fie mai mare de *refreshRate*. Dacă a venit momentul să se analizeze ecranul, atunci se face o captură a acestuia printr-un apel la *takeScreenshot()*, se actualizează timpul *lastScreenshotTime* ca fiind cel curent, se încarcă imaginea în *VideoSource*, apoi se apelează *frameManager.update()*, care declanșează apelarea funcției *process()* a tuturor unităților de procesare pe care le gestionează. În final, după ce noile culori ale ramei au fost actualizate în obiectul *LedFrame*, gestionat de *frameManager*, se apelează funcția *sendLedColors()* din clasa *SerialCommunicator*, modulul *io*, cu scopul de a serializa și trimite spre *Arduino* comanda care să aprindă LED-urile. Dacă execuția acestei bucle este întreruptă, atunci se apelează *frameManager.turnOff()*, care trimite pe portul serial comanda corespunzătoare stingerii LED-urilor.

O limitare, descoperită destul de târziu, este aceea că Robot, clasa Java responsabilă de capturile de ecran, nu poate realiza mai mult de 12 capturi pe secundă. Acest lucru poate fi corectat prin folosirea unor librării cu cod nativ C++ care să crească viteza la 60 capturi pe secundă, precum JavaCV. Aplicația, în forma curentă, nu face acest lucru, din rațiuni de stabilitate.

AudioWorker se ocupă de gestionarea procesării audio, utilizând biblioteca *TarsosDSP*. Totodată, el este cel care calculează metrica *signal*. Ne vom concentra doar asupra funcției *run()*, întrucât aceasta este de interes. Ca la *VideoWorker*, se creează un *frameManager* și unitățile de procesare asociate lui. După aceea, într-un bloc *try-catch*, rulează cod specific bibliotecii *TarsosDSP*. Pentru a-l putea înțelege mai bine, trebuie studiat modul în care funcționează această bibliotecă. Firul de execuție (*pipeline*) al acesteia este descris în următoarea imagine:

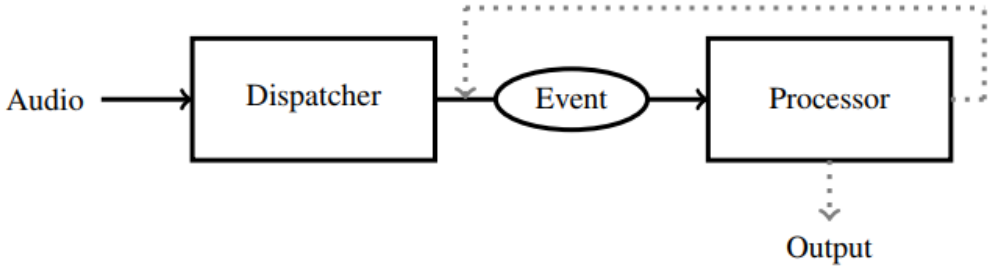

Fig. 34 - Firul de execuție al bibliotecii TarsosDSP[6]

Conform articolului "*TarsosDSP, a Real-Time Audio Processing Framework in Java"*, totul începe cu un *AudioDispatcher*, responsabil de captarea semnalului audio din diferite surse (de la microfon sau dintr-un fișier audio) și împărțirea lui în blocuri de date *float*, cu valori cuprinse între [-1, 1]. Aceste blocuri sunt încapsulate într-un *Event* și trimise către un *AudioProcessor*, care va procesa conținutul acestora cu scopul de a obține un rezultat (*Output*), sau, opțional, va modifica într-un anume fel *Event-ul* curent cu scopul de a fi procesat de un alt *AudioProcessor*. (Joren Six & Leman, 2014)

Înțelegerea captării audio este în strânsă legătură cu definiția unei unde sonore. Modelul matematic al acesteia este o linie sinusoidală care variază în timp. Axa verticală denotă amplitudinea sunetului, adică intensitatea sonoră cu care este perceput. O melodie este compusă din mai multe sunete, fiecare cu unda sa, dar care sunt îmbinate într-o singură linie sinusoidală, ca în imaginea din dreapta. Această linie este una continuă, care, din păcate, nu poate fi prelucrată ca atare de dispozitivele digitale, spre deosebire de cele analogice.

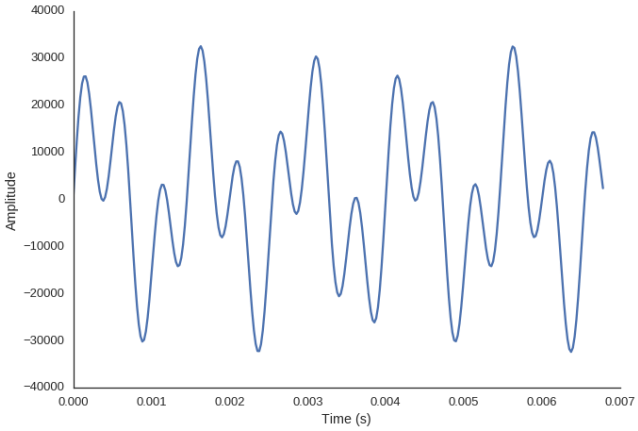

Fig. 35 - Sunetul ca o sinusoidă în domeniul timpului[7]

Astfel, pentru ca un aparat digital să poată reprezenta în memorie o astfel undă sonoră, se măsoară, de foarte multe ori pe secundă, amplitudinea sunetului, dând iluzia de continuitate. Numărul de măsurători pe secundă se numește *sample rate*, iar dimensiunea zonei de memorie în care acestea sunt stocate poartă numele de *buffer size*. (Rocchesso, 2003)

Din nefericire, o reprezentare în spațiul Amplitudine – Timp a semnalului sonor captat nu este foarte utilă, întrucât, așa cum a fost menționat mai sus, aceasta este rezultatul compunerii undelor sunetelor din componența sa. În cel mai bun caz, se poate calcula intensitatea audio exprimată în decibeli. Există, totuși o rezolvare la această problemă, și anume folosirea Transformării Fourier Discrete (*Discrete Fourier Transform - DFT*).

[6] Imagine preluată din articolul ***TarsosDSP, a Real-Time Audio Processing Framework in Java***, menționat în bibliografie.

[7] Imagine preluată de aici - http://amyboyle.ninja/DSP-Audio

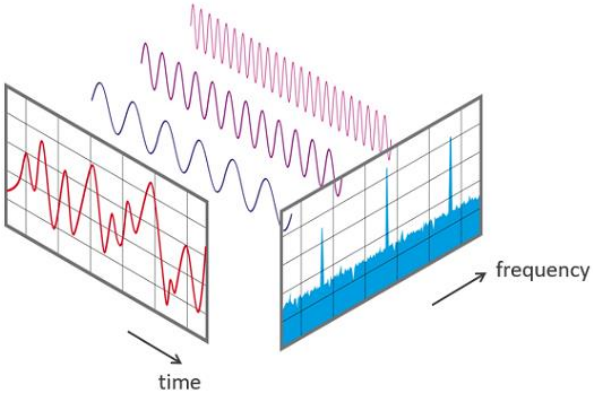

Fig. 36 - Vizualizarea Transformării Fourier Discrete[8]

Scopul acestei transformări este reprezentat de trecerea din domeniul timpului în cel al frecvențelor, unde frecvența, exprimată în Hertz, exprimă numărul de cicluri complete ale sinusoidei într-o secundă. În acest mod, semnalul audio este descompus în undele sale componente, iar pentru fiecare dintre ele se știe frecvența, așa cum se poate observa în imaginea din stânga. Acest lucru este util pentru că, de exemplu, se poate diminua zgomotul dintr-o melodie prin micșorarea frecvenței sunetelor problematice, se pot realiza comprimări audio, se poate altera sunetul în diferite moduri cu scopul de a obține diferite efecte etc. (Gaydecki, 2019)

Transformarea Fourier Discretă primește la intrare un *buffer* x_N, cu dimensiunea N egală cu *buffer size*, amintit mai sus. Acest *buffer* conține o listă cu valori ale amplitudinii sonore înregistrate în ultimele $\frac{buffer\ size}{sample\ rate}$ secunde. La ieșire returnează o listă cu valori ale amplitudinii, dar în domeniul frecvențelor, X_N. Acest lucru este realizat prin rezolvarea ecuațiilor:

$$X_k = \sum_{n=0}^{N-1} x_n * e^{\frac{-2i\pi}{N}kn} = \sum_{n=0}^{N-1} x_n * [\cos\left(\frac{2\pi kn}{N}\right) - i * \sin(\frac{2\pi kn}{N})], k \in \{0, 1, \dots N-1\}$$

Cu această ocazie, amintim *Teorema lui Nyquist*, care spune că dacă se folosește un anumit *sample rate*, atunci frecvența maximă care se poate înregistra este jumătate din acesta. Astfel, în urma aplicării algoritmului de mai sus, X_k va memora o medie a semnalului corespunzător frecvențelor din intervalul $[k * \frac{sample\ rate}{2*buffer\ size}, (k+1) * \frac{sample\ rate}{2*buffer\ size} - 1]$. Rezolvarea acestor ecuații se realizează în $O(N^2)$, iar pentru un *sample rate* de 44100 (urechea umană poate auzi până la 22000 Hz) și un *buffer size* de 1024, rezultă 43 de aplicări pe secundă, ceea ce nu se pretează procesării în timp real. O variațiune a acestui algoritm este *Fast Fourier Transform* (FFT) care, pe baza tehnicii de programare *Divide et Impera* și a exploatării periodicității funcțiilor trigonometrice, poate rezolva aceste ecuații în $O(NlogN)$, cu condiția ca *buffer size* să fie putere a lui 2. (McGee, 2007-2009)

Metrica *signal*, cea folosită de către algoritmii de procesare audio, și cea care este considerată că "dictează" ritmul muzicii este, în realitate, valoarea maximă din lista X_N, dar care este parcursă doar până la N/2, deoarece jumătățile sunt simetrice în urma aplicării FFT. Am ales această valoare întrucât corespunde amplitudinii sonore al celui mai relevant sunet din întreaga melodie.

Revenind la *AudioWorker*, codul din metoda *run()* sintetizează tot ce a fost prezentat mai sus. Acesta urmează firul de execuție al bibliotecii *TarsosDSP* și începe prin instanțierea unui *AudioDispatcher* care să capteze audio de la microfonul implicit la un *sample rate* de 44100 și un *buffer size* de 1024, putere a lui 2. La acest *dispatcher* se adaugă un nou *AudioProcessor*, personalizat să îndeplinească funcționalitatea pe care ne-o dorim. Metoda *process()* primește ca parametru un *AudioEvent* din care se extrage informația audio într-un tablou unidimensional de valori *float, audioData*. Se instanțiază apoi un obiect *FFT,* specific bibliotecii *TarsosDSP*, care conține o implementare a algoritmului FFT amintit mai sus. Se aplică transformarea FFT tabloului *audioData*, apoi se parcurge jumătate din acesta în căutarea valorii maxime *intensity*, care apoi

[8] Imagine preluată de aici - https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft

este folosită în actualizarea *AudioSource* din modulul *source*. După aceea, ca la *VideoWorker*, se actualizează unitățile de procesare printr-un apel către *frameManager.update()*, apoi se trimit culorile rezultate spre *Arduino*, prin comanda *sendLedColors()*. În final, se returnează false, ceea ce înseamnă că procesarea audio nu s-a terminat. A se observa că nu există o buclă implicită, ci este creeată una de către firul de execuție al *TarsosDSP*. Se vor trimite date către *Arduino* de fiecare dată când s-a înregistrat un *buffer* cu semnale cu dimensiunea de *buffer size*. În cazul de față, de 43 de ori pe secundă. Codul sursă al clasei *AudioWorker* este disponibil în Anexa 3 – Audio Worker.

AmbientWorker este rezultatul unei combinații dintre cei doi *worker-i* precedenți. Ca și *AudioWorker*, va încărca profilele de culoare și va seta culoarea curentă a unităților de procesare și, implicit, a LED-urilor ramei. Codul din funcția *run()* va fi similar, de data aceasta, cu cel din *VideoWorker*: se va creea un *frameManager* și unitățile de procesare aferente, se va calcula distanța de timp între două actualizări ale ramei (care va fi aceeași ca în cazul procesării video; variabila din modulul *config* dictează numărul de actualizări pe secundă a ambelor funcționalități, fapt ce nu este evident utilizatorului final al aplicației). Apoi, într-o buclă *while* care rulează cât timp *isRunning == true*, se actualizează *TimeSource* cu timpul curent, apoi unitățile de procesare obțin noile culori ce urmează a fi trimise către *Arduino*.

4.4.5 Modulul algorithms

Așa cum a fost prezentat în introducerea acestui capitol, aplicația implementează nouă categorii de algoritmi rezultate în urma încrucișării dintre natura lor (preprocesare, procesare și postprocesare) și tipul de unitate pentru care este destinat (video, audio și ambiental). Modulul *algorithms* este împărțit în două pachete: *processing_algorithms* (conține structura și implementările acestora) și *algorithm_manager* (descrie mecanismul de instanțiere și de gestiune). În continuare, vor fi prezentate implementările tuturor algoritmilor aplicației, urmând ca, în încheierea secțiunii, să fie descrisă gestiunea acestora.

4.4.5.1 Algoritmii de procesare video

La baza tuturor algoritmilor există trei interfețe numite *PreProcessing Algorithm*, *ProcessingAlgorithm* și *PostProcessingAlgorithm*, fiecare dintre acestea având o singură metodă void, cu nume sugestiv, care primește ca parametru o unitate de procesare. Aceste funcții vor modifica structura internă a unităților, conform naturii algoritmilor, cu scopul de o obține o nouă culoare curentă.

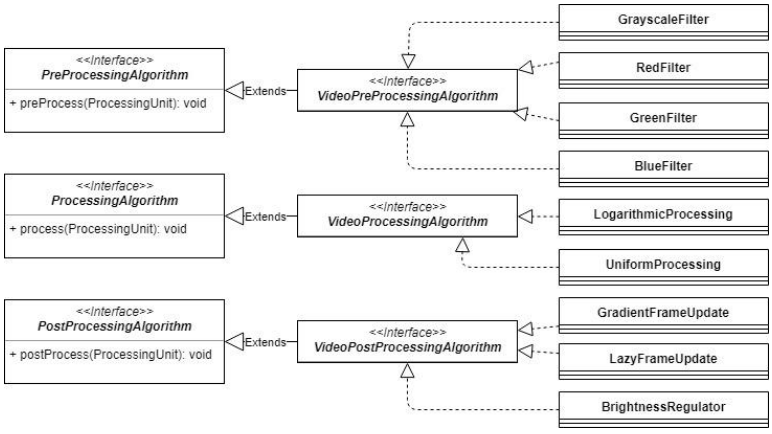

Fig. 37 - Arhitectura algoritmilor de procesare video

Din aceste interfețe sunt extinse alte trei, ale căror singură schimbare este doar numele acestora, pentru a delimita mai bine categoriile amintite mai sus.

Algoritmii de preprocesare video au ca scop alterarea în diferite moduri a capturii de ecran curente. De exemplu, algoritmul ***GrayscaleFilter*** presupune parcurgerea pixel cu pixel al sectorului de imagine asociat unității de procesare curente, cu scopul de a-l transforma în alb și negru. Acest lucru se realizează prin extragerea valorilor pentru roșu *R*, verde *G* și albastru *B* ale pixelului curent și înlocuirea culorii acestuia cu o medie ponderată a numerelor extrase. Astfel, culoare curentă este înlocuită din (R, G, B) în (M, M, M), unde $M = 0.2126 * R + 0.7152 * G + 0.0722 * B$. Deși la fel de bine aceste ponderi pot fi egale cu 0.33, în realitate se preferă aceste ponderi întrucât culoarea

verde contribuie mult mai mult la luminozitatea unei culori, iar albastru cel mai puțin. ***RedFilter***, ***BlueFilter*** și ***GreenFilter*** funcționează în mod similar, doar că nuanța de culoare a pixelului curent este înlocuită din (R, G, B) în, respectiv, (R, R, R), (B, B, B) sau (G, G, G). Efectul final al acestor algoritmi este acela că lumina ramei va fi monocromă, fie albă, verde, roșie sau albastră, și se va comporta ca un joc de umbre, în funcție de conținutul redat pe ecran. De asemenea, ei nu alterează în niciun mod datele membre ale unităților de procesare primite ca parametru, ci doar pregătesc imaginea pentru algoritmii care urmează. Doar unul dintre aceștia poate fi activ la un moment dat.

Algoritmii de procesare video sunt cei care au un impact major asupra întregului proces. Aceștia analizează sectorul de imagine asociat unității de procesare curente și decid, pe baza acestuia, care să fie culoarea LED-ului. Metoda de obținere a acestei culori este asemănătoare, dar diferă complexitatea algoritmului. ***UniformProcessing*** parcurge sectorul pixel cu pixel și extrage, pentru fiecare pixel i în parte, valorile pentru roșu R_i, verde G_i și albastru B_i, și le însumează, fiecare separat. În final, culoarea curentă a LED-ului extrasă din sector va fi media aritmetică a fiecărei componente în parte, adică:

$(\frac{\sum_{i=0}^{W*H} R_i}{W*H}, \frac{\sum_{i=0}^{W*H} G_i}{W*H}, \frac{\sum_{i=0}^{W*H} B_i}{W*H})$, unde W și H sunt dimensiunile în pixeli al sectorului de imagine.

Complexitatea acestui algoritm este O($W * H$) și, dacă sectoarele de imagine sunt mari (pot fi și fâșii până la jumătatea ecranului), iar rata de actualizări pe secundă a ramei are și ea o valoare considerabilă, atunci se poate dovedi ineficient în a ține pasul. Avantajul acestui algoritm este că obține culoarea cea mai fidelă.

Un algoritm mai eficient, de complexitate O(log($W * H$)), este ***LogarithmicProcessing***, care se aseamănă cu cel precedent, doar că numărul de pixeli procesați este mult mai mic. După ce se termină de extras valorile pentru roșu, verde și albastru ale pixelului curent, următorul pixel analizat nu este cel imediat de lângă el, ci unul aflat la *XStep* distanță. Când se termină de analizat o linie de pixeli, următoarea nu este cea aflată imediat sub ea, ci una la *yStep* distanță. În acest mod, se procesează pixeli care sunt la distanțe egale unii de alții.

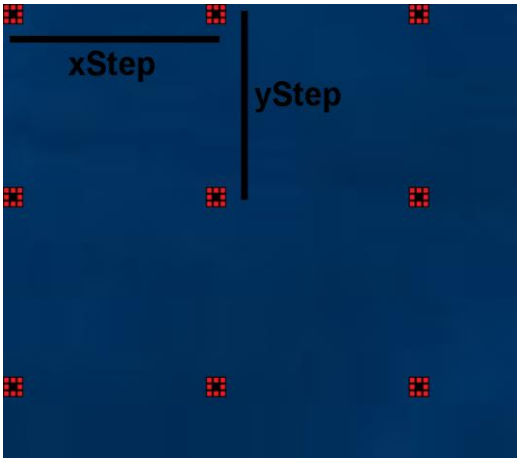

Fig. 38 - LogarithmicProcessing

Figura din stânga ilustrează cel mai bine această parcurgere. Punctele negre sunt pixelii considerați pentru procesare, iar cele roșii sunt vecinii lor. Distanța de la un pixel la celălalt pe orizontală este *xStep*, iar pe verticală *yStep*. Întrucât numărul de pixeli procesați este foarte mic, nu îi putem considera doar pe aceștia ca dictând culoarea finală pentru că media culorii lor este foarte ușor influențabilă. Să ne imaginăm că această imagine ar fi cerul înstelat, iar în unele puncte negre ar fi stele luminoase, cât un pixel de mari. Culoarea fidelă, obținută de algoritmul ***UniformProcessing***, ar fi foarte aproape de negru absolut întrucât stelele au o pondere mult prea mică, însă lucrurile ar sta diferit în cazul procesării logaritmice.

De aceea, pentru ca punctele de excepție (*outliers*) să nu aibă pondere mare în calcularea mediei, se procesează și vecinii pixelului curent. Pentru ca numărul de pixeli analizați să fie cât mai mic, se dorește ca, pe o linie, să se proceseze doar $\log_{10} W$ pixeli, iar numărul de linii parcurse să fie $\log_{10} H$. Astfel, valorile pentru *xStep* și *yStep* vor fi următoarele:

$xStep = W / \log_{10} W$; $yStep = H / \log_{10} H$. În acest mod, pentru un sector de 76 x 67 pixeli, cum este în cazul folosirii unei rame cu 25 LED-uri lungime și 16 lătime și a unui ecran cu rezoluția 1920 x 1080, se analizează doar 4 pixeli și vecinătatea lor. Culoarea obținută astfel este foarte

aproape de cea fidelă întrucât, în cazul unei utilizări normale, în timpul unui film, de exemplu, arareori se întâmplă să existe un număr mare de culori diferite într-o bucată de imagine atât de mică. Pot exista, totuși, situații limită care ar duce la diferențe mai consistente, așa că rămâne la alegerea utilizatorului dacă preferă o mai bună performanță sau culori mai fidele. Un astfel de caz pe care l-am observat în utilizarea aplicației este acela când se fac capturi de ecran ale *desktop-ului*. Dacă există iconițe la marginea ecranului, atunci algoritmul ***UniformProcessing*** produce culoarea fundalului, dar mult mai spre alb datorită acestora, în timp ce ***LogarithmProcessing*** obține o culoare apropiată de cea care s-ar obține în absența iconițelor.

Algoritmii de postprocesare video au rolul de a altera culoarea curentă a LED-ului pentru a acomoda anumite preferințe ale utilizatorului. De exemplu, algoritmul ***BrightnessRegulator*** modifică intensitatea culorii curente pe baza unei ponderi a cărei valoare este modificată de către utilizator prin intermediul interfeței grafice. Modul de funcționare al algoritmului este următorul: se extrage culoare curentă a unității de procesare primite ca parametru și se transformă din spațiul de culoare RGB în cel HSB (*Hue, Saturation, Brightness*) prin intermediul clasei *Utils* din Java FX. Componenta *brightness* este apoi înmulțită cu ponderea *brightnessModifier*, care este extrasă din modulul *config* și a cărei valoare variază de la 0 la 2. Dacă *brightness* depășește valoarea 1, atunci se recalibrează la 1, întrucât, în Java FX, valorile componentelor de culoare variază între 0 și 1, nu între 0 și 255. După aceea, culoarea modificată este transformată înapoi în spațiul RGB și se setează ca fiind cea curentă.

Dacă se utilizează rama în acest fel, fără niciun algoritm de postprocesare, se va observa că tranziția culorilor este în tandem cu conținutul redat pe ecran, ceea ce poate fi supărător pentru ochi dacă se face trecerea brusc de la o culoare la alta. Pentru a corecta acest lucru, se folosesc algoritmi de postprocesare de tip relaxare. ***GradientFrameUpdate*** este unul dintre ei și are datoria de a face tranzițiile culorilor mai line și mai plăcute pentru ochi. Pentru aceasta, se folosește de variabilele *currentColor* și *previousColor* ale unității de procesare curente. *PreviousColor* reține culoarea LED-ului de la ultima execuție a întreg lanțului de algoritmi de procesare video, în timp ce *currentColor* conține culoarea curentă obținută până acum și care s-ar afișa dacă nu ar fi modificată de algoritmii de postprocesare. La fiecare execuție a algoritmului ***GradientFrameUpdate***, se iau aceste două culori și se face diferența, pe componentele roșu, verde și albastru, dintre *currentColor* și *previousColor*. Aceste diferențe, care NU sunt în modul, sunt memorate în variabilele *deltaRed*, *deltaGreen* și *deltaBlue*. Se extrage, apoi, din modulul *config*, valoarea *smoothFactor*, o pondere, între 0 și 1, care denotă cât de fin să se facă tranziția de la culorile precedente la cele actuale. Noile valori pentru componentele roșu, verde și albastru sunt calculate astfel: *updatedRed = previousColor.getRed() + deltaRed * smoothFactor,* similar și pentru celelalte două. Culoarea astfel obținută o va înlocui pe cea curentă. Efectul acestui algoritm în practică este acela că atunci când se trece de la o culoare la alta, este nevoie de mai multe actualizări ale ramei până când se ajunge la culoarea de pe ecran, realizându-se, astfel, o trecere graduală, interpolată.

LazyFrameUpdate este un algoritm de postprocesare al cărui principiu de funcționare se bazează pe faptul că tranziția de culoare de la ultima actualizare la cea curentă se realizează doar dacă noua culoare este "total diferită" față de cea precedentă. Există o variabilă, în modulul *config,* care dictează gradul de diferență între culori, numită *LazyUpdateStyleConfortRange,* pe scurt - *range*. Înainte de a vedea cum funcționează, trebuie mai întâi să îi înțelegem rostul.

Pentru aceasta, să ne imaginăm că există un spațiu tridimensional în care axele acestuia corespund componentelor roșu, verde și albastru din codificarea RGB a culorilor. Pe fiecare axă sunt reprezentate valorile de la 0 la 255. Astfel, o culoare RGB va fi un punct în acest spațiu, așa cum este vizibil și în imaginea din dreapta. Considerăm că două culori se aseamănă dacă și numai dacă există un cub cu latura *range* care conține, în interiorul acestuia, punctele corespunzătoare celor două culori.

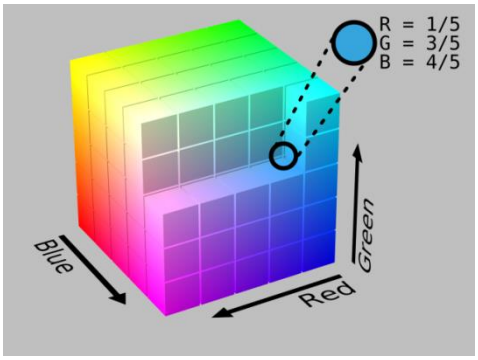

Fig. 39 - Spațiul RGB[9]

Dacă acest lucru nu se întâmplă, atunci culorile sunt considerate "total diferite". Astfel, implementarea acestui algoritm este una trivială. Ca la ***GradientFrameUpdate***, se calculează, pe componente, diferențele în modul, de data aceasta, dintre *currentColor* și *previousColor*, și se memorează în variabilele *deltaRed*, *deltaGreen* și *deltaBlue*. Dacă aceste valori sunt toate trei simultan mai mici decât *range*, atunci se consideră că sunt culori asemenea și *currentColor* va fi egal cu *previousColor*, altfel sunt "total diferite" și culoarea curentă nu se schimbă. Valoarea variabilei *range* aparține intervalului de la 0 la 255 și este un număr întreg.

Față de categoriile precedente, pot fi și are sens să fie mai mulți algoritmi de postprocesare care să se aplice unității video curente, doar că trebuie să se execute mai întâi ***LazyFrameUpdate***, apoi ***GradientFrameUpdate***, întrucât ordinea inversă influențează rezultatul în mod negativ (culoarea modificată în gradient nu va diferi în mod fundamental de culoarea precedentă).

4.4.5.2 Algoritmii de procesare audio

Structura este asemănătoare ca cea de la algoritmii video, doar că numele interfețelor extinse din cele de bază sunt diferite, pentru a denota categorii noi. În mod similar, aceștia primesc unități de procesare și, în funcție de natura algoritmului, modifică datele membre din componența lor.

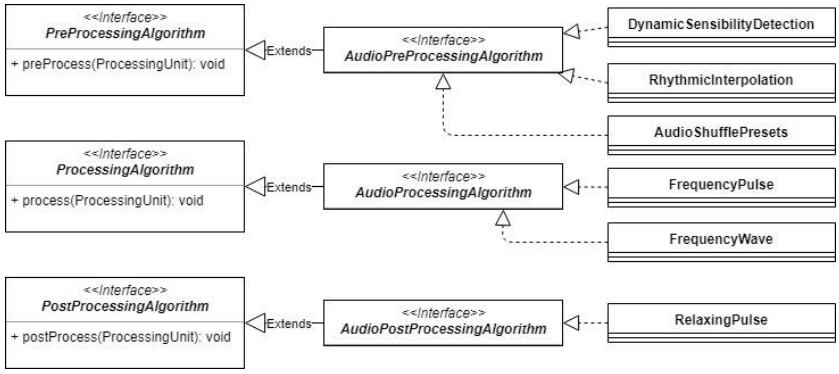

Fig. 40 - Arhitectura algoritmilor de procesare audio

Vom începe prin a studia algoritmii de procesare audio. Funcționalitatea lor se bazează pe faptul că unitățile de procesare și LED-urile lor aferente au deja o culoare de bază, dată prin intermediul unui profil de culoare. Apoi, pe baza metricii *signal*, stocată în variabila *currentSignal* a unităților de procesare audio, modifică într-un anume fel culoarea curentă, ca o variațiune a celei de bază. Metoda fiecărui algoritm de a face asta diferă.

Algoritmul ***FrequencyPulse*** începe prin a calcula ponderea *colorIntensity* ca fiind un raport dintre *currentSignal* (valoarea metricii *signal*) și *sensibility (*valoarea maximă *signal* care poate fi percepută de către unitatea de procesare*)*. Această pondere trebuie să fie pozitivă, maxim 1, deci se calibrează în mod corespunzător dacă raportul este supraunitar. Culoarea curentă a LED-ului va fi culoarea lui de bază ale cărei componente sunt înmulțite cu ponderea *colorIntensity*. Efectul final al acestui algoritm va face ca intensitatea luminoasă a întregii rame să varieze în funcție de ritmul muzicii, ceea ce creează iluzia unui puls.

[9] Imagine sub licență Cc-by-sa-3.0, prezentată ca atare, preluată de aici - https://github.com/mjhorvath/mike_illustrations

Algoritmul ***FrequencyWave*** funcționează în mod similar, adică tot pornește de la premisa că există culori de bază pentru LED-urile ramei, dar tehnica de actualizare diferă. Conceptul este următorul: partea inferioară este mereu aprinsă, iar LED-urile de pe laterale se aprind, de jos în sus, în funcție de valoarea *currentSignal* stocată în unitățile de procesare. Dacă această valoare este mai mare sau egală cu *sensibility*, atunci întreaga ramă va fi iluminată conform profilului de culoare setat. O procesare de acest tip dă iluzia faptului că ritmul melodiei este un val, de aici și numele algoritmului. Algoritmul începe prin a extrage din modulul *config* numărul de LED-uri de pe lungime și de pe înălțime, apoi calculează cota maximă de semnal *ledQuota* a unei LED, făcându-se raportul dintre *sensibility* și înălțimea ramei. După aceea, se calculează numărul de LED-uri care ar putea fi alimentate de *currentSignal* ca fiind raportul dintre semnalul curent și *ledQuota,* rezultat memorat în variabila *onLedCount*. În funcție de segmentul de ramă și indexul LED-ului asociate unității de procesare, se calculează înălțimea față de bază *unitAudioHeight*. Dacă unitatea de procesare se află deja pe partea din jos a ramei, atunci se consideră culoarea curentă ca fiind cea de bază și se oprește execuția algoritmului. În caz contrar, se determină valoarea *unitAudioHeight* așa cum este prezentat în codul din Anexa 4 – Algoritmul de procesare audio *FrequencyWave*. În final, pe baza acestei valori, se hotărăște dacă LED-ului curent va fi aprins sau nu, calculând diferența dintre *onLedCount* și *unitAudioHeight*. Dacă diferența e pozitivă, atunci se va aprinde LED-ul curent, dacă este negativă, culoarea curentă va fi negru, iar dacă este egală cu zero, atunci LED-ul va fi aprins, dar nu la intensitate maximă.

În concordanță cu acești algoritmi, dar în special cu ***FrequencyPulse***, există un singur algoritm de postprocesare numit ***RelaxingPulse***. Ideea de funcționare este similară cu cea de la algoritmul ***GradientFrameUpdate***, prezentat în secțiunea anterioară. Fără nicio postprocesare, intensitatea luminoasă a LED-urilor va fi în tandem cu ritmul muzicii, ceea ce creează haos vizual care deranjează ochiul. Această problemă este diminuată cu ajutorul algoritmului ***RelaxingPulse***. Execuția lui începe cu extragerea a două ponderi din modulul *config*, și anume *upBeatSmoothFactor* și *downBeatSmoothFactor,* care dictează finețea cu care se face tranziția de la o actualizare la alta a culorilor ramei, valorile acestora fiind între 0 și 1. Prima pondere semnifică gradul de rafinament al tranziției culorilor de la o intensitate audio scăzută la una crescută, în timp ce a doua dictează trecerea de la una crescută la alta scăzută. Are sens ca aceste două ponderi să difere întrucât se dorește ca tranzițiile *upBeat* să fie aproape instante întrucât acestea sunt sufletul melodiei, ele dictează ritmul general. O valoare scăzută sau aproape de 0.5 duce la desincronizări sau întârzieri în a ține pasul cu ritmul. Pe de cealaltă parte, ponderea *downBeat* ar trebui să aibă o valoare scăzută pentru ca trecerea de la parte intensă la una mai liniștită a melodiei să se realizeze lin, treptat. Necesitatea ca valorile acestor ponderi să fie în acest mod este justificată de următoarea experiență: imaginați-vă producerea unui sunet provocată de ciupirea unei corzi de chitară. Inițial, în primele fracțiuni de secundă, el este perceput ca fiind foarte gălăcios și tare, apoi se relaxează și, lin, dispare. Ne dorim același efect și în cazul ramei, doar că sunetul chitarei este înlocuit de intensitatea luminoasă a LED-urilor: bruscă inițial și lină în final.

Odată explicată semnificația ponderilor, algoritmul ***RelaxingPulse*** este trivial. Se calculează diferențele, dar nu în modul, pe componente dintre culoarea curentă și cea precedentă. Dacă toate aceste diferențe sunt negative, atunci înseamnă că s-a trecut de la o intensitate mai mare la una mai mică, deci noua culoare curentă va fi cea anterioară la ale cărei componente RGB se adaugă produsul dintre diferența de culoare și ponderea *downBeatSmoothFactor*. În caz contrar, se execută aceeași operație, doar că se folosește celalaltă pondere – *upBeatSmoothFactor*.

Algoritmii de preprocesare audio îndeplinesc anumite funcționalități de care ar depinde ceilalți algoritmi, de aceea au fost lăsați, intenționat, la urmă. Unul dintre ei se numește ***DynamicSensibilityDetection*** și a fost inclus în Anexa 5 – Algoritmul de preprocesare *DynamicSensibilityDetection*. Rolul acestuia este de a analiza valorile metricii *signal* din ultima secundă, cu scopul de a ajusta automat sensibilitatea unităților de procesare audio. În acest fel,

dacă o melodie își menține un ritm monoton, atunci rama va lumina la jumătate din intensitate, iar când vor fi fluctuații de ritm, acestea vor fi mult mai vizibile și spectaculoase. Astfel, utilizatorul este scutit de a găsi echilibrul perfect dintre sensibilitatea unităților și volumul de *output* al sistemului audio. De asemenea, este potrivit în cazul în care volumul muzicii forțează metrica *signal* să depășească pragul de 100 al aplicației, la o petrecere, de exemplu.
La nivel programatic, acest obiectiv este atins prin folosirea unei cozi *frequencySample* în care, la fiecare actualizare a ramei, se pune valoarea *currentSignal* a primei unități de procesare (nu trebuie să ne facem griji întrucât toate unitățile au aceeași valoare) în coadă. S-a optat pentru o coadă întrucât trebuie să se execute inserări și eliminări rapide, în timp constant. Aceste operații se pot executa eficient prin utilizarea unei cozi. După ce a fost actualizată coada, se calculează media aritmetică *sampleMean* a tuturor elementelor acesteia, dar doar dacă numărul lor este egal cu marginea superioară *sampleRate*. În momentul în care numărul de elemente este egal cu *sampleRate* pentru prima dată, atunci se calculează media prin parcurgerea tuturor elementelor componente. După aceea, când se adaugă un nou element în coadă, se elimină și se reține cea mai veche valoare în variabila *lastFrequency*, apoi se actualizează *sampleMean,* în timp constant, prin adăugarea diferenței *currentSignal – lastFrequency*, împărțită la *sampleSize*. După ce se obține noua medie, dacă aceasta este mai mare ca 2 (adică există muzică pe fundal și nu este liniște), atunci sensibilitatea ajustată a unităților *adjuestedSensibility* devine 1.5 * *sampleMean*. În acest mod, dacă ritmul rămâne aproximativ constant, *sampleMean* nu variază și rama va fi luminată la 50% din intensitate, întrucât *currentSignal* va fi aproximativ jumătate din sensibilitatea ajustată.

Algoritmul ***RhythmicInterpolation*** funcționează identic cu cel de ***InterpolateAnimation***, prezentat în secțiunea următoare. Scopul acestuia este de a realiza o animație de tranziție a culorilor ramei de la un profil de culoare de bază la unul prestabilit. Inițial se dorea obținerea unei astfel de animații a cărei viteză să fie dictată de volumul melodiei, dar, în acest mod, se creea un haos vizual neplăcut, așa că am renunțat la această idee.

AudioShufflePresets este un algoritm de preprocesare care presupune schimbarea profilului de culoare în momentul în care se termină o animație a ramei. În cazul modului audio, acest algoritm are sens doar dacă este folosit împreună cu ***RhythmicInterpolation***. Există o variantă a algoritmului adaptată pentru unități de procesare ambientale, fructificată mai bine de acest mod de funcționare. Modul de funcționare este următorul: se verifică dacă unitatea curentă are indexul LED-ului asociat 0 și că are variabila *isAnimationFinished* setată pe *true*. Dacă este cazul, se alege un nou profil de culoare, în mod aleatoriu, din lista acestora, stocată în *ColorPresetSource*, așa încât el nu coincide cu cel actual și nici nu este unul proaspăt creat, fără să fi fost editat. După aceea, profilele de culoare din *colorPresetSource* sunt permutate așa încât *currentState* devine *nextState*, iar *nextState* ia valoarea profilului ales anterior. Culorile *baseColor* și *nextColor* sunt actualizate în mod corespunzător, iar valoarea variabilei *isAnimationFinished* este setată pe *False*.

4.4.5.3 Algoritmii de procesare ambientală

Structura algoritmilor de procesare ambientală este aceeași ca și la categoriile anterioare. Astfel, vor fi prezentate doar funcționalitățile algortimilor finali.

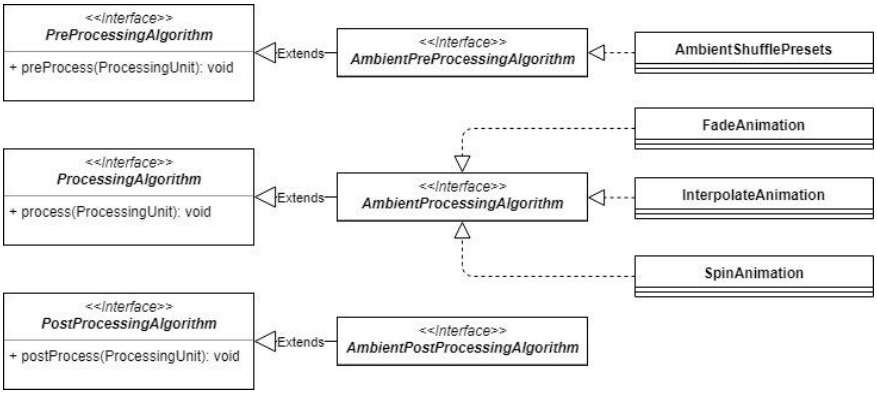

Fig. 41 - Arhitectura algoritmilor de procesare ambientală

În ceea ce privește algoritmii de preprocesare ambientală, există doar unul singur, ***Ambient ShufflePresets***, care funcționează întocmai ca ***AudioShufflePresets***, descris mai sus.

Algoritmii intesanți sunt cei de procesare ambientală, responsabili de crearea animațiilor specifice acestui mod de funcționare. ***FadeAnimation***, de exemplu, realizează o animație în care LED-urile se aprind, treptat, spre un profil de culoare prestabilit, iar când ajung la intensitate maximă, se sting, progresiv, până când rama nu luminează deloc. Pentru a realiza acest lucru, este nevoie de un mecanism matematic care să simuleze această periodicitate. Inițial am folosit funcția sinus, dar am ales să exploatez, în final, restul împărțirilor, întrucât e mai puțin costisitor în a fi calculat. Ceea ce se dorește este obținerea unei pondere, între 0 și 1, care dictează în ce procent de desfășurare a animației sunt la momentul actual. Știind deja cât timp a trecut până acum de la începutul execuției animației, *elapsedTime* (valoare, în milisecunde, este extrasă din modulul *Source*), dar și cât de mult durează o animație, *animationTime* (valoare cunoscută din modulul *config*, exprimată tot în milisecunde), se poate calcula această pondere, *intensityPercentage*, ca fiind: *elapsedTime % animationTime / animationTime*. Se aplică, mai întâi, restul pentru a vedea cât de mult timp a trecut de la sfârșitul ultimului interval de animație, apoi se împarte la timpul de animație pentru a obține un procent subunitar. În acest mod, algoritmul ***FadeAnimation*** devine trivial: la fiecare apelare a sa, află ponderea *intensityPercentage* și calculează culoarea curentă ca fiind cea de bază, înmulțită, pe componente, cu aceasta.

Aplicarea acestui calcul nu obține, din păcate, rezultatul dorit. Când rama este la intensitatea luminoasă maximă, se va stringe brusc și apoi se va aprinde treptat, efect pe care nu ni-l dorim. Este nevoie de un mod de a distinge între animația de *fadeIn* și cea de *fadeOut*. Acest lucru se realizează cu ajutorul unei variabile *boolean isFadeIn*, care este inițializată pe *True*. Dacă unitățile de procesare rețin ponderea *intensityPercentage* de la ultima actualizare, atunci se poate deduce ușor când a început o nouă animație: diferența dintre ponderea calculată acum și cea de la pasul anterior este negativă. Când se întâmplă acest lucru, putem comuta valoarea *isFadeIn*. În acest mod, culoarea curentă se obține prin înmulțirea celei de bază cu un *intensityModifier*, a cărui valoare este *intensityPercentage* dacă *isFadeIn* este true, sau 1 – *intensityPercentage* în caz contrar. Astfel, se obține și efectul de *fadeOut* dorit. Dacă valoarea variabilei *isFadeIn* s-a schimbat din adevărat în fals, atunci se consideră că întreaga animație de *fade* s-a terminat și se actualizează câmpul *isAnimationFinished* a unităților de procesare.

Algoritmul de procesare ambientală ***InterpolateAnimation*** realizează o animație de tranziție a culorilor de la un profil de culoare de bază, prestabilit, la unul de interpolare, ambele extrase din modulul *config*. În momentul în care s-a ajuns la profilul țintă, culorile ramei vor tranziționa către cele de bază. Acest fenomen se realizează tot pe baza ponderii *intensityPercentage*, al cărei mod de funcționare a fost descris mai sus. Algoritmul calculează diferențele, pe componente, dintre culoarea finală, cea spre care se dorește a se ajunge, și cea inițială, de bază, de la care s-a plecat. Culoarea curentă a LED-ului va fi culoarea de bază, la ale cărui componente sunt adăugate produsul dintre diferența de culoare pe acea componentă, înmulțită cu *intensityPercentage*. Algoritmul se bazează pe aceeași idee de a calcula diferența de intensități de la ultima actualizare la cea curentă pentru a deduce dacă animația s-a sfârșit. În acest caz, va actualiza variabila respectivă a unității de procesare curente, dar va interschimba și culoarea de bază cu cea de interpolare pentru a se putea realiza, fără probleme, și animația de revenire la culoarea inițială.

Algoritmul ***SpinAnimation*** este diferit ca abordare față de ceilalți doi de până acum. Scopul acestuia este de a realiza o animație în care culorile LED-urilor se interschimbă circular , în sens trigonometric. Modul de funcționare se bazează pe faptul că, în *TimeSource*, se stochează durata de timp necesară unui LED pentru ca acesta să se deplaseze cu o poziție, în contextul în care străbate întreaga ramă în *animationTime* milisecunde. Această metrică este numită *spinAnimationTransitionTime*, și se calculează în momentul inițializării unui *AmbientWorker*. Algoritmul calculează la ce poziție este acum primul LED, memorată în variabila

leadingLedPosition, pe baza a cât timp s-a scurs până acum, apoi deduce poziția la care ar trebui să fie LED-ul curent, adăugând la indexul acestuia valoarea *leadingLedPosition* și făcând restul împărțirii cu numărul total de LED-uri al ramei, din rațiuni de circularitate. Culoarea curentă a acestuia va fi extrasă din *ColorPresetSource*, alegându-se cea corespunzătoare poziției anterior calculată.

La momentul actual, nu a fost necesară implementarea unor algoritmi de postprocesare ambientală, dar există suport pentru adăugarea lor ulterioară, în cazul în care se va dori acest lucru.

4.4.5.4 Gestiunea algoritmilor

Gestiunea algoritmilor a fost proiectată cu scopul de a garanta faptul că există doar câte o instanță a fiecărui algoritm de procesare în parte. Se dorește acest lucru pentru a optimiza consumul de memorie, nefiind necesar câte un același algoritm pentru fiecare unitate de procesare. Acest lucru se realizează prin intermediul unui *AlgorithmManager* care implementează șablonul de proiectare *Singleton*, și care conține trei obiecte: *(Video|Audio|Ambient)AlgorithmManager*. La rândul lor, fiecare dintre aceștia conține alți trei manageri, specifici tipului de algoritm: preprocesare, procesare și postprocesare. În final, se obțin nouă astfel de manageri, fiecare conținând algoritmii specifici categoriei din care aparțin. Figura de mai jos ilustrează doar o parte a acestei arhitecturi, specifice categoriei de algoritmi de procesare video.

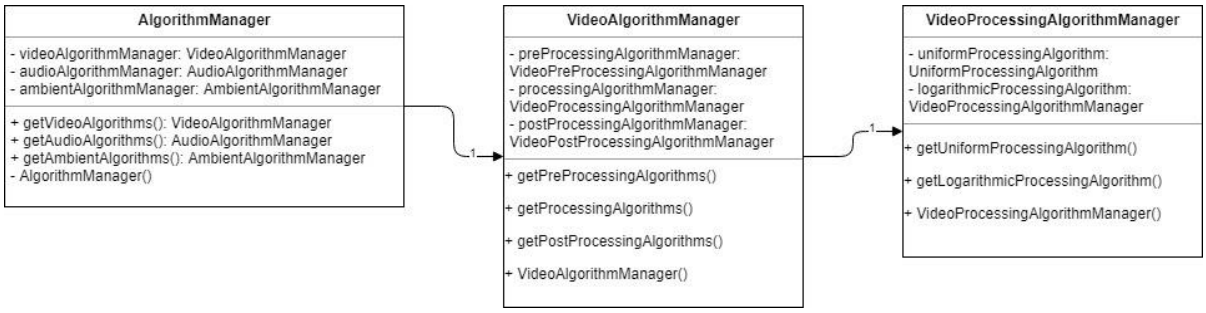

Fig. 42 - Arhitectura sistemului de gestiune ai algoritmilor

Constructorii tuturor acestor manageri instanțiază obiectele conținute în aceștia, mai puțin *AlgorithmManager*, care face asta doar în funcțiile accesor *get*. În acest mod, înlănțuirea de apeluri:

AlgorithmManager().getVideoAlgorithms().getProcessingAlgorithms().getUniformProcessingAlgorithm() - oferă mereu aceeași instanță a algoritmului. Bine-nțeles, nimic nu oprește instanțierea directă a algoritmului acolo unde este nevoie de el, dar în acest mod crește claritatea codului. Aceste înlănțuiri de apeluri sunt utilizate în cadrul claselor care extind *ProcessingUnitFactory*, din modului *processing_units*.

4.4.6 Modulul io

4.4.6.1 Comunicarea serială

Comunicarea prin intermediul portului serial este realizată cu ajutorul clasei *SerialCommunicator*, pe baza protocolului descris în secțiunea 3.3.2. În acest sens, clasa conține mai multe date membru *static final byte* necesare implementării protocolului, precum antetul comenzilor sau codurile corespunzătoare lor. Pe lângă clasa anterior menționată, există și *SerialPortSingleton*, care oferă acces către un singur obiect de tip *SerialPort*, specific bibliotecii *JSSC*, pe baza unui parametru *String* semnificând numele portului *USB* prin care se va realiza comunicarea. Clasa

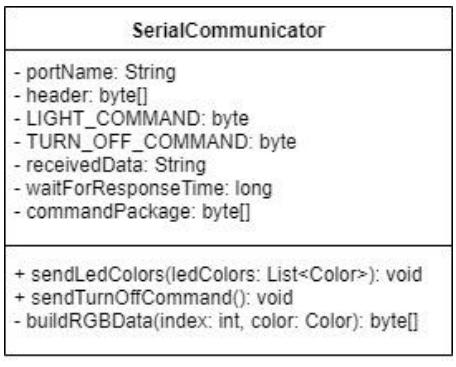

Fig. 43 - Structura clasei SerialCommunicator

SerialCommunicator conține două metode publice statice:

- *sendLedColors(List<Color> ledColors)* – primește ca parametru o listă de culori care se doresc a fi trimise LED-urilor benzii. Metoda extrage numele portului din modulul *config* și obține un obiect de tip *SerialPort* prin intermediul clasei *SerialPortSingleton*, apoi îl folosește pentru a trimite către *Arduino* antetul comenzii, octetul corespunzător *LIGHT_COMMAND*, precum și dimensiunea listei. După aceea, pentru fiecare element din listă, apelează metoda *buildRGBData()* pentru a construi șirul de octeți corespunzători culorii curente, memorați în variabila *commandPackage*, urmând ca acesta să fie trimis prin portul serial. Metoda *buildRGBData()* primește ca parametru indexul LED-ului curent, precum și culoarea acestuia, și citește permutarea RGB *rgbOrder* din modulul *config*, care va fi un șir de 3 litere curespunzător unei permutări ale literelor "R"-"G"-"B", urmând ca apoi să construiască *commandPackage* astfel: pe poziția 0 se pune indexul, iar pe pozițiile *commandPackage[rgbOrder.indexOf(X) + 1]* pune valoarea corespunzătoare culorii X, unde X ∈ {"R", "G", "B"};
- *sendTurnOffCommand()* – similar ca la metoda precedentă, obține o instanță a clasei *SerialPort* care se folosește, prin apelarea metodei *writeBytes()* aferentă, la transmiterea antetului și a octetului corespunzător comenzii *TURN_OFF_COMMAND*. Transmiterea se face într-o buclă *while*, la intervale de timp egale cu *waitForResponseTime*, până când se recepționează de la *Arduino* caracterul "&", semnificând reușita comenzii.

Citirea de pe portul serial se realizează prin intermediul unei clase *inline* care implementează interfața *SerialPortEventListener*, specifică librăriei *SerialPort*. Aceasta are o singură metodă de tip void, numită *serialEvent()*, care primește ca parametru un eveniment *SerialPortEvent*, tratat în corpul funcției. Dacă evenimentul este unul de tip *RXCHAR*, asta înseamnă că sunt caractere în *buffer-ul* de intrare care pot fi citite. Se extrage obiectul de tip *SerialPort*, similar ca la metodele precedente, și se apelează funcția asociată *readString()*, care va returna caracterele citite în variabila *receivedData*.

4.4.6.2 Gestiunea fișierelor

Profilele de culoare gestionate de către aplicație sunt serializate și salvate pe disc, în directorul *User_Home/Documents/Desktop_Lights*. De serializarea și deserializarea acestora se ocupă clasa *FileManager*, care are o serie de metode statice pentru a citi sau a scrie fișiere din/în acel director. Aceasta conține inclusiv metoda *createHomeDirectory()*, care se ocupă stric5 de crearea și inițializarea directorului *Desktop_Lights*. Când se pornește pentru prima dată aplicația, această funcție este apelată pentru a serializa profilele de culoare prestabilite, care nu pot fi editate direct din interfața grafică.

Există și o clasă ajutătoare *JsonParser*, al cărei singur rol este să pună la dispoziția programatorului două metode publice statice, *getJsonFromObject* și *getObjectFromJson*, care folosesc biblioteca *Gson* în a obține un șir de caractere care simbolizează serializarea *JSON* a unui obiect, dar și invers, un obiect dintr-un *string JSON*. Aceste metode sunt folosite exclusiv de către *FileManager* cu scopul de a serializa și deserializa obiecte de tipul *FrameColorPreset*.

4.5 View

4.5.1 Modulul ui

Fiecărei ferestre prezentate în capitolul 2 îi corespunde, în aplicație, două fișiere: unul care descrie înfățișarea acesteia, într-un format XML special numit *FXML*, și o clasă Java care se ocupă de

controlul ei. Pentru ca cele două să funcționeze în mod corespunzător și să se obțină rezultatul dorit, este necesară conectarea lor. Clasa Java răspunzătoare de control va avea ca date membre obiecte din *API-ul* JavaFX care apar în descrierea interfeței grafice, precum *Button, CheckBox, Slider, TextField, ChoiceBox etc*, prefixate de o adnotarea ***@FXML***. Totodată, dacă sunt metode care trebuiesc apelate în momentul în care există o anumită interacțiune cu un element grafic, atunci și acestea pot fi adnotate în același mod. De cealaltă parte, fișierul *FXML* va avea mereu un element rădăcină, deobicei o variațiune a unui *Pane,* pentru care se poate seta valoarea atributului *controller* ca fiind egală cu numele clasei responsabilă de control. În plus, nodurilor XML corespunzătoare elementelor grafice li se va seta atributul *id* ca fiind numele variabilei asociate din clasa *controller*. Pentru ca acest lucru să meargă, variabilele trebuies adnotate ca mai sus. Dacă există noduri *Button*, atunci li se pot seta atributul *onAction* ca fiind numele unei funcții din *controller*. Se poate acest lucru și din clasa *controller,* asociind un *Listener* butonului respectiv printr-un apel către *SetOnMouseClicked()*.

La *runtime*, conexiunea dintre cele două este realizată prin intermediul clasei *FXMLLoader,* parte a *API-ului* JavaFX. Acestuia i se asociază o locație de unde să extragă resursa specifică unui fișier FXML, prin folosirea unui *ClassLoader*. Prin intermediul funcției *load()* se obține, în final, un obiect Java de tipul rădăcinii conținutul XML al fișierului. În cadrul aplicației, toate aceste încărcări de fișiere sunt realizate într-o singură clasă, *SettingsViewManager,* utilizând două metode publice, incluse în Anexa 6 – SettingsViewManager. În continuare, așa cum reiese din codul anexat, se creează un *Stage*, adică o nouă fereastră de interfață grafică, careia i se asociază o scenă *Scene* care conține aspectul și elementele grafice încărcate din fișier. Se instanțiază și un *SettingsController* la care se asociază scena instanțiată anterior, după care fereastra este vizibilă și se așteaptă închiderea ei. La final se returnează o valoare de adevăr care reprezintă dacă fereastra a fost deschisă și a avut un impact asupra stării aplicației sau nu. De exemplu, din *MainLayoutController,* clasa corespunzătoare ferestrei care se deschide când se pornește aplicatia, se poate deschide meniul corespunzător setărilor video. Se apelează *showSettingsDialog()* din *SettingsViewManager* pentru meniul cu opțiuni video, iar acesta apare pe ecran. Dacă fereastra a fost închisă sau se apasă pe buton *Cancel*, atunci se returnează *false*, adică nu există schimbări în starea internă a configurărilor. Dacă se apasă pe butonul *Ok*, atunci fereastra este închisă și se returnează *true*. Se revine la *MainLayoutController* și, dacă există modificări, atunci le va salva și va reinițializa *MainController* printr-un apel *restart()*.

Se poate observa o abordare abstractă și generică, acest lucru realizându-se prin crearea unei interfețe *SettingsController* care este implementată de toate clasele *controller*. Aceasta conține 5 metode specifice unui *controller*, 3 pentru funcționalitatea butoanelor *handleOk(), handleApply()* și *Cancel()*, și două care apar în *SettingsViewManager – isOkClicked()* și *setDialogStage()*.

Din punct de vedere alorganizării proiectului, fișierele *FXML* sunt grupate într-un același pachet *view,* dar care este structurat logic pe directoare. Întrucât acestea nu conțin cod, pachetul corespunzător lor a fost plasat în directorul *resources*. Fișierele *controller* sunt organizate identic, dar sunt plasate într-un pachet numit *ui,* parte din *controller*.

Structura și modul de funcționare ale tuturor claselor *controller* sunt similare. Acestea conțin obiecte adnotate din *API-ul* JavaFX pentru elementele grafice, precum și o funcție *initialize(),* tot adnotată, care este apelată în momentul în care interfața *controller-ului* este afișată pe ecran. Aceasta conține cod care încarcă din modulul *config* anumite date specifice, urmând ca apoi să fie folosite pentru a popula elementele grafice. Acestor elemente le sunt asociate funcții *listener* pentru a dicta un anumit comportament în cazul unor situații specifice, de exemplu dacă apăsarea unui *checkbox* declanșează blocarea unei alte opțiuni. Funcția *handleApply()* va extrage datele din

elementele grafice și le va salva în modulul *config*, validându-le *a priori*, apoi va declanșa repornirea *MainWorker*. Funcția *handleOk()* are un comportament similar, doar că va închide fereastra și va seta variabila *isOkClicked* pe *true*. *HandleClose()* închide fereastra și atribuie valoarea *false* variabilei *isOkClicked*. Singura excepție de la acest mod de funcționare este clasa *EditPresetIndividualLedController*, responsabilă de fereastra care permite editarea manuală a fiecărui LED în parte, al cărei aspect este generat procedural întrucât numărul de butoane nu este mereu același.

Stilul interfeței cu utilizatorul a fost creat utilizând un fișier *AppTheme.css*, care, similar tehnologiei CSS folosită în dezvoltarea aplicațiilor *WEB*, asociază anumite caracteristici aspectuale elementelor grafice, precum culoare, spațieri, font, opacitate etc. Fiecare fișier *FXML* a creat o legătură cu *AppTheme.css* prin intermediul setării atributului *stylesheets* a nodului rădăcină.

4.5.2 AlertManager

Așa cum se poate observa în Anexa 2 – VideoWorker, în cazul apariției unei probleme, se apelează metoda *AlertManager.showCustomError()*, care va face să apară pe ecran o fereastră cu un mesaj de eroare. Acest lucru este posibil prin intermediul clasei *AlertManager*, deținătoare a două metode: *showError()*, care primește ca parametru un obiect *MyException*, o extindere personalizată a clasei *Exception*, și *showCustomError()*, care are 3 parametri de tip *String*, corespunzători pentru titlul, antetul și conținutul erorii. Ambele metode au același rol – crearea unei ferestre de dialog cu un mesaj de eroare.

JavaFX are o clasă proprietară *Alert* responsabilă cu afișarea ferestrelor cu mesaje de eroare. Cu toate acestea, folosrea ei este incomodă pentru că are nevoie de multe operații de tip *set* înainte de a fi instanțiată. Aplicația de proesare ușurează acest proces prin intermediul unui *AlertBuilder*, o clasă care implementează șablonul de proiectare *Builder* și care facilitează crearea de obiecte *Alert* personalizate.

4.6 MainApp

Această clasă este punctul de intrare în aplicație, întrucât conține metoda *main()*. Aceasta extinde clasa abstractă *Application*, fapt specific aplicațiilor realizate cu ajutorul JavaFX. *MainApp* conține, ca date membre, un *Stage* static, numit *primaryStage* care, așa cum amintit mai sus, corespunde unei ferestre grafice. În plus, există și un obiect de tip *AnchorPane*, *mainLayout*, în care se va încărca interfața grafică din fișierul *FXML* corespunzător.

Metoda principală a acestei clase este *start(Stage primaryStage)*, apelată în momentul deschiderii aplicației. În interiorul acesteia se inițializează *ConfigManager*, se creează în User Home directorului aplicației, în cazul în care nu există deja, și se pornește *MainWorker*. Pentru că acest *worker* rulează pe un alt fir de execuție, în continuare se încarcă fără blocaje interfața din *MainLayout.fxml*, apoi este făcută vizibilă către un apel *primaryStage.show()*.

4.7 Concluzie

În acest capitol au fost descrise tehnologiile, arhitectura și implementarea aplicației de procesare. Consider că exemplele detaliat prezentate justifică faptul că aceasta are o arhitectură flexibilă, care să permită adăugarea facilă de noi funcționalități, fapt ce îndeplinește obiectul O3 prezentat în introducerea lucrării.

5 Concluzie

Această lucrare a avut ca scop descrierea realizării unui sistem de *lumini ambientale* dinamice, având ca obiectiv final obținerea unui produs ușor de folosit, extensibil și configurabil. Consider că acest scop a fost atins, așa cum reiese din argumentele prezentate până acum. Cu toate acestea, există chestiuni care ar putea fi îmbunătățite.

În ceea ce privește funcționalitățile de bază ale ramei, fiecare dintre ele poate fi rafinată cu noi adăugiri ca să devină mai rapidă, mai dinamică sau mai spectaculoasă. De exemplu, modul video poate fi îmbunătățit prin folosirea unor biblioteci cu cod nativ C++ precum *JavaCV* pentru a crește viteza capturilor de ecran, în acest mod obținând tranziții mai fine între culori. Totodată, se poate implementa un algoritm de identificare și ignorare al marginilor negre, fapt ce ar îmbunătăți experiența vizionării filmelor al căror raport de aspect nu corespunde cu cel al ecranului. Modul audio ar putea fi extins cu algoritmi care procesează spectrul audio mai detaliat cu scopul de a identifica și a reacționa doar la anumite tipuri de sunete (*beat detection*). Noi animații pot fi adăugate modului ambiental care să exploateze mai bine trecerea timpului, precum unele care ar simula un ceas sau un cronometru. În plus, opțiunile generale ar putea fi extinse în așa fel încât începutul ramei să se poată afla oriunde pe marginea ecranului, nu doar în colțuri. De asemenea, posibilitatea de a crea profile pentru opțiunile video, audio sau ambientale ar fi un plus în experiența de interacțiune cu aplicația a utilizatorului, care i-ar economisi timp.

Din punct de vedere al viitorului proiectului, mi-ar plăce ca această soluție să rămână una dedicată strict sistemelor *desktop.* Un mod posibil și inovativ de extindere al proiectului se poate realiza prin transformarea aplicației de procesare într-un centru de comandă capabil să sincronizeze culoarea LED-urilor ramei cu a altor periferice *RGB,* precum tastaturi sau mouse-uri. De exemplu, compania *Razer* pune la dispoziția utilizatorilor săi *Razer SDK*, o bibliotecă special dezvoltată pentru a programa efecte luminoase pentru perifericile *RGB* compatibile. Aplicația ar putea interacționa cu această bibliotecă pentru a sincroniza aspectul ramei cu cel al perifericelor. În cazul companiilor care nu pun la dispoziție un astfel de *API* pentru produsele lor, dar care au programul lor dedicat pentru controlul culorilor, se pot implementa roboți *software* orchestrați de către aplicația de procesare. Deși viteza unor astfel de roboți trebuie investigată întrucât intuiesc că nu s-ar preta procesării în timp real, prin apelarea lor periodică și prin folosirea unor algoritmi de interpolare s-ar putea obține un efect plăcut.

Consider că ideea de sincronizare cu periferice *RGB* ar fi una de succes și ar face aplicația să iasă în evidență întrucât nicio altă soluție asemănătoare nu abordează această posibilitate. În plus, piața dispozitivelor de acest fel, deși încă o nișă, crește de la an la an și devine din ce în ce mai populară. În mod sigur utilizatorii unor astfel de periferice ar fi intesați să își îmbunătățească experiența vizuală cu un sistem de *lumini ambientale,* cu atât mai mult cu cât acesta ar fi compatibil deja cu componentele *hardware* pe care le dețin. Acest fapt asigură cererea care ar putea alimenta un *startup* de succes.

Bibliografie

Banzi, M., & Shiloh, M. (2015). *Getting Started with Arduino.* USA: Maker Media.

Cristescu, D., Sălăvăstru, C., Voiculescu, B., Niculescu, C. T., & Cârmaciu, R. (2006). *Biologie - Manual pentru clasa a XI-a.* București: Corint.

Fitzpatrick, J. (2017, Noiembrie 13). *What Bias Lighting Is and Why You Should Be Using It*. Retrieved from How-To Geek: https://www.howtogeek.com/213464/how-to-decrease-eye-fatigue-while-watching-tv-and-gaming-with-bias-lighting/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns Elements of Reusable Object-Oriented Software.* USA: Pearson Education.

Garcia, D. (2019, Ianuarie 20). *Interrupt problems*. Retrieved from Github: https://github.com/FastLED/FastLED/wiki/Interrupt-problems

Gaydecki, P. (2019). *Digital Signal Processing.* University of Manchester.

Gayral, B. (2017). LEDs for lighting: Basic physics and prospects for energy savings. *Computes Rendus Physique, vol. 18, nr. 7-8* , pp. 453-461.

Joren Six, O. C., & Leman, M. (2014). TarsosDSP, a Real-Time Audio Processing Framework in Java. *AES 53RD INTERNATIONAL CONFERENCE.* London.

McGee, K. J. (2007-2009). *An Introduction to Signal Processing and Fast Fourier Transform (FFT).*

Oracle. (n.d.). *Core Java Preferences API*. Retrieved from Oracle Docs: https://docs.oracle.com/javase/8/docs/technotes/guides/preferences/index.html

Oracle. (n.d.). *What is JavaFX?* Retrieved from Oracle Docs: https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm

Rocchesso, D. (2003). *Introduction to Sound Processing.* Firenze: PHASAR Srl.

Sheppard, A. L., & Wolffsohn, J. S. (2018). *Digital eye strain: prevalence, measurement and amelioration.*

Shvets, A. (2013). *Design Patterns Explained Simply.*

Anexa 1 – codul încărcat pe *Arduino*

```
#include <FastLED.h>

#define DATA_PIN     8
#define NUM_LEDS    82
#define BRIGHTNESS 255
#define HEADER_SIZE 4
#define LIGHT_COMMAND 0
#define TURN_OFF_COMMAND 2

CRGB leds[NUM_LEDS];

const uint8_t header[HEADER_SIZE] = { 's', 'e', 'n', 'd'};
uint8_t currentByte;
uint8_t currentCommand;
bool isHeader;
uint8_t light_command_buffer[NUM_LEDS * 4];

void solve_command();
void solve_light_command();
void solve_turn_off_command();

void setup() {
    FastLED.addLeds<WS2812, DATA_PIN>(leds, NUM_LEDS);
    FastLED.setBrightness( BRIGHTNESS );
    Serial.begin(115200);
}

void loop() {
    while(true){
        while(Serial.available() == 0){}
        currentByte = Serial.read();
        isHeader = false;
        if (currentByte == header[0]) {
            isHeader = true;
            for (int i = 1; i < HEADER_SIZE && isHeader; i++) {
                while(Serial.available() == 0){}
                currentByte = Serial.read();
                if (currentByte != header[i]) {
                    isHeader = false;
                }
            }
        }

        if (isHeader) {
            while(Serial.available() == 0){}
            currentCommand = Serial.read();
            solve_command();
            break;
        }
    }
    FastLED.show();
    while(Serial.available() > 0) { Serial.read(); }
}
```

```
void solve_command() {
    if (currentCommand == LIGHT_COMMAND) {
        solve_light_command();
        return;
    }

    if (currentCommand == TURN_OFF_COMMAND) {
        solve_turn_off_command();
        Serial.print('&');
        return;
    }
}

void solve_light_command() {
    uint8_t id, rgbValue1, rgbValue2, rgbValue3;
    int bytesRead = 0;
    int packageSize = 0;

    while(Serial.available() == 0){}
    uint8_t ledCount = Serial.read();
    packageSize = 4 * ledCount;

    while(bytesRead < packageSize){
      bytesRead += Serial.readBytes(light_command_buffer + bytesRead,
packageSize - bytesRead);
    }

    for (int i = 0; i < 4 * ledCount; i+=4) {
      id = light_command_buffer[i];
      rgbValue1 = light_command_buffer[i+1];
      rgbValue2 = light_command_buffer[i+2];
      rgbValue3 = light_command_buffer[i+3];
      leds[id] = CRGB(rgbValue1, rgbValue2, rgbValue3);
    }

}

void solve_turn_off_command(){
    for (int i = 0; i < NUM_LEDS; i++)
    {
        leds[i] = 0x000000;
    }
}
```

Acest cod, precum și cel al aplicației de procesare, poate fi consultat la adresa web: https://github.com/Lucian-Bosinceanu/Bias-Lighting-for-Desktop-Systems

Anexa 2 – VideoWorker

```
public class VideoWorker implements Worker {

    private boolean isRunning;
    private BufferedImage screenshot;
    private Long lastScreenshotTime;
    private ScreenshotTaker screenshotTaker;
    private FrameManager frameManager;

    @Override
    public void run() {
        isRunning = true;

        System.out.println("Video Worker Starts");
        try {

            int updateRate = ConfigManager.getVideoConfig().getUpdateRate();
            int refreshRate = 1000 / updateRate;
            screenshotTaker = new ScreenshotTaker();
            SourceManager.getVideoSource().update(screenshotTaker.take());
            frameManager = new FrameManager();
            frameManager.constructProcessingUnits();
            lastScreenshotTime = System.currentTimeMillis();
            while (isRunning) {
                if (System.currentTimeMillis() - lastScreenshotTime > refreshRate) {
                    takeScreenshot();
                    lastScreenshotTime = System.currentTimeMillis();
                    SourceManager.getVideoSource().update(screenshot);
                    frameManager.update();
                    SerialCommunicator.sendLedColors(frameManager.getLedArray());
                }
            }
            frameManager.turnOff();
        }
        catch (AWTException e) {
            e.printStackTrace();
        }
        catch (SerialPortException e) {
            isRunning = false;
            ConfigManager.getGlobalConfig().setFrameMode(FrameMode.NONE);
            Platform.runLater(new Runnable() {
                @Override
                public void run() {
                    AlertManager.showCustomError(
                            "Critical error!",
                            "No device found for provided port - " +
ConfigManager.getGlobalConfig().getSerialPortName(),
                            "Make sure that the device is connected to correct port!"
                    );
                }
            });
            stop();
        }

    }

    private void takeScreenshot(){
        screenshot = screenshotTaker.take();
    }

    public void stop() {
        isRunning = false;
    }
}
```

Anexa 3 – Audio Worker

```
public class AudioWorker implements Worker {

    private boolean isRunning;
    private FrameColorPreset basePreset;
    private FrameColorPreset interpolationPreset;
    private FrameManager frameManager;
    private static final int sampleRate = 44100;
    private static final int bufferSize = 1024;
    private AudioDispatcher audioDispatcher;

    @Override
    public void stop() {
        isRunning = false;
        audioDispatcher.stop();
        try {
            frameManager.turnOff();
        } catch (SerialPortException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        isRunning = true;
        frameManager = new FrameManager();
        frameManager.constructProcessingUnits();

        try {
            audioDispatcher = AudioDispatcherFactory.fromDefaultMicrophone(sampleRate,
bufferSize, 0);
            SourceManager.getColorPresetSource().loadAllPresets();
            loadPresets();
            setIndividualLedColor();
            audioDispatcher.addAudioProcessor(new AudioProcessor() {
                @Override
                public boolean process(AudioEvent audioEvent) {
                    float[] audioData = audioEvent.getFloatBuffer();
                    FFT fft = new FFT(bufferSize);
                    fft.forwardTransform(audioData);
                    double intensity = 0;

                    for (int i = 0; i < bufferSize/2; i++) {
                        if (audioData[i] > intensity) {
                            intensity = audioData[i];
                        }
                    }

                    SourceManager.getAudioSource().update(intensity);
                    try {
                        SourceManager.getTimeSource().update(System.currentTimeMillis());
                        frameManager.update();
                        SerialCommunicator.sendLedColors(frameManager.getLedArray());
                    } catch (SerialPortException e) {
                        isRunning = false;
                        ConfigManager.getGlobalConfig().setFrameMode(FrameMode.NONE);
                        Platform.runLater(new Runnable() {
                            @Override
                            public void run() {
                                AlertManager.showCustomError(
                                        "Critical error!",
                                        "No device found for provided port - " +
ConfigManager.getGlobalConfig().getSerialPortName(),
                                        "Make sure that the device is connected to
correct port!"
```

```
                                );
                            }
                        });
                        stop();
                    }
                    return false;
                }

                @Override
                public void processingFinished() {
                }
            });

            SourceManager.getTimeSource().setStartTime();
            audioDispatcher.run();


        } catch (IOException e) {
            e.printStackTrace();
        } catch (LineUnavailableException e) {
            e.printStackTrace();
        }
    }

    private void loadPresets() throws IOException {
        String presetFileName = ConfigManager.getAudioConfig().getFramePreset();
        FrameColorPreset currentPreset = FrameColorPreset.loadColorState(presetFileName);
        SourceManager.getColorPresetSource().update(currentPreset);
        basePreset = currentPreset;

        String interpolationPresetFileName =
ConfigManager.getAudioConfig().getInterpolationPresetName();
        interpolationPreset =
FrameColorPreset.loadColorState(interpolationPresetFileName);
        SourceManager.getColorPresetSource().updateNextState(interpolationPreset);
    }

    private void setIndividualLedColor() {
        for (int i = 0; i < frameManager.getLedArray().size() && i <
basePreset.getIndividualColors().size(); i++) {
            frameManager.getLedArray().set(i,
Color.valueOf(basePreset.getIndividualColors().get(i)));
            AudioProcessingUnit currentUnit = (AudioProcessingUnit)
frameManager.getProcessingUnits().get(i);

currentUnit.setBaseColor(Color.valueOf(basePreset.getIndividualColors().get(i)));

currentUnit.setLedCurrentColor(Color.valueOf(basePreset.getIndividualColors().get(i)));
        }

        for (int i = 0; i < frameManager.getLedArray().size() && i <
interpolationPreset.getIndividualColors().size(); i++) {
            AudioProcessingUnit currentUnit = (AudioProcessingUnit)
frameManager.getProcessingUnits().get(i);

currentUnit.setNextColor(Color.valueOf(interpolationPreset.getIndividualColors().get(i)))
;
        }
    }

}
```

Anexa 4 – Algoritmul de procesare audio *FrequencyWave*

```
public void process(ProcessingUnit processingUnit) {
    AudioProcessingUnit updatedUnit = (AudioProcessingUnit) processingUnit;
    int unitAudioHeight = 0;
    int heightLedCount = ConfigManager.getGlobalConfig().getHeight();
    int widthLedCount = ConfigManager.getGlobalConfig().getWidth();

    if (ConfigManager.getGlobalConfig().getConstructionType() ==
FrameConstructionType.SIDE_ONLY) {
        widthLedCount = 0;
    }

    int ledIndex = updatedUnit.getLedIndex();
    double ledQuota = updatedUnit.getSensibility() /
ConfigManager.getGlobalConfig().getHeight();
    FrameSegment frameSegment = updatedUnit.getAssociatedFrameSegment();
    Color baseColor = updatedUnit.getBaseColor();

    if (ConfigManager.getAudioConfig().isRhythmicInterpolation()) {
        baseColor = updatedUnit.getLedCurrentColor();
    }

    int onLedCount = (int) (updatedUnit.getCurrentSignal() / ledQuota);
    onLedCount = onLedCount > heightLedCount ? heightLedCount : onLedCount;

    if (frameSegment == FrameSegment.BOTTOM) {
        updatedUnit.setLedCurrentColor(baseColor);
        return;
    }

    if (frameSegment == FrameSegment.LEFT) {
        unitAudioHeight = ledIndex;
    } else if (frameSegment == FrameSegment.TOP) {
        unitAudioHeight = heightLedCount;
    } else if (frameSegment == FrameSegment.RIGHT) {
        unitAudioHeight = heightLedCount - (ledIndex - heightLedCount - widthLedCount);
    }

    if (unitAudioHeight < onLedCount) {
        updatedUnit.setLedCurrentColor(baseColor);
    } else if (unitAudioHeight > onLedCount) {
        updatedUnit.setLedCurrentColor(Color.BLACK);
    } else {
        double lastOnLedQuota = (updatedUnit.getCurrentSignal() % ledQuota) / ledQuota;
        Color newColor = Color.color(
                baseColor.getRed() * lastOnLedQuota,
                baseColor.getGreen() * lastOnLedQuota,
                baseColor.getBlue() * lastOnLedQuota
        );

        updatedUnit.setLedCurrentColor(newColor);
    }
}
```

Anexa 5 – Algoritmul de preprocesare *DynamicSensibilityDetection*

```
public class DynamicSensibilityDetectionPreProcessingAlgorithm implements
AudioPreProcessingAlgorithm {

    private Queue<Double> frequencySample;
    private int sampleSize = 43;
    private double adjustedSensibility;
    private double sampleMean;

    public DynamicSensibilityDetectionPreProcessingAlgorithm() {
        frequencySample = new LinkedList<>();
        adjustedSensibility = ConfigManager.getAudioConfig().getSensibility();
        sampleMean = -1;
    }

    @Override
    public void preProcess(ProcessingUnit processingUnit) {
        AudioProcessingUnit updatedUnit = (AudioProcessingUnit) processingUnit;
        if (updatedUnit.getLedIndex() == 0) {
            updateSample(updatedUnit.getCurrentSignal());
            adjustSensibility();
        }
        updatedUnit.setSensibility(adjustedSensibility);
    }

    private void updateSample(double currentSignal) {
        if (frequencySample.size() < sampleSize) {
            frequencySample.add(currentSignal);
        } else {
            double lastFrequency = frequencySample.remove();
            frequencySample.add(currentSignal);
            updateSampleMean(currentSignal, lastFrequency);
        }
    }

    private void updateSampleMean(double currentSignal, double lastFrequency) {
        double mean = 0;
        if (sampleMean == -1) {
            for (Double value : frequencySample) {
                mean += value;
            }

            mean /= sampleSize;
        } else {
            mean = sampleMean + (currentSignal - lastFrequency) / sampleSize;
        }

        sampleMean = mean;
    }

    private void adjustSensibility() {
        if (frequencySample.size() < sampleSize) {
            return;
        }

        if (sampleMean > 2) {
            adjustedSensibility = sampleMean * 1.5;
        }
    }
}
```

Anexa 6 – SettingsViewManager

```
public class SettingsViewManager {

    public static final String STATIC_SETTINGS_CREATE_GRADIENT =
"view/settings/static_mode/EditPresetCreateGradientLayout.fxml";
    public static final String STATIC_SETTINGS_INDIVIDUAL_LED =
"view/settings/static_mode/EditPresetIndividualLedsLayout.fxml";
    public static final String STATIC_SETTINGS_ADD_PRESET =
"view/settings/static_mode/AddPresetLayout.fxml";
    public static final String STATIC_SETTINGS_EDIT_PRESET =
"view/settings/static_mode/EditPresetLayout.fxml";
    public static final String GENERAL_SETTINGS =
"view/settings/GeneralSettingsLayout.fxml";
    public static final String STATIC_SETTINGS =
"view/settings/static_mode/StaticSettingsLayout.fxml";
    public static final String VIDEO_SETTINGS = "view/settings/VideoSettingsLayout.fxml";
    public static final String AUDIO_SETTINGS = "view/settings/AudioSettingsLayout.fxml";
    public static boolean showSettingsDialog(String resourceName, String title, boolean
resizable, boolean maximized) {
        try {
            FXMLLoader loader = new FXMLLoader();
loader.setLocation(SettingsViewManager.class.getClassLoader().getResource(resourceName));
            AnchorPane dialog = loader.load();
            Stage dialogStage = new Stage();
            dialogStage.setTitle(title);
            dialogStage.initModality(Modality.WINDOW_MODAL);
            dialogStage.initOwner(MainApp.getPrimaryStage());
            Scene scene = new Scene(dialog);
            dialogStage.setResizable(resizable);
            dialogStage.setMaximized(maximized);
            dialogStage.setScene(scene);
            SettingsController controller = loader.getController();
            controller.setDialogStage(dialogStage);
            dialogStage.showAndWait();
            return controller.isOkClicked();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }

    public static boolean showSettingsDialog(String resourceName, SettingsController
customController, String title, boolean resizable, boolean maximized) {
        try {
            FXMLLoader loader = new FXMLLoader();

loader.setLocation(SettingsViewManager.class.getClassLoader().getResource(resourceName));
            AnchorPane dialog = loader.load();
            Stage dialogStage = new Stage();
            dialogStage.setTitle(title);
            dialogStage.initModality(Modality.WINDOW_MODAL);
            dialogStage.initOwner(MainApp.getPrimaryStage());
            Scene scene = new Scene(dialog);
            dialogStage.setResizable(resizable);
            dialogStage.setMaximized(maximized);
            dialogStage.setScene(scene);
            customController.setDialogStage(dialogStage);
            dialogStage.showAndWait();
            return customController.isOkClicked();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```