Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Комп'ютерно-інтегрована система генерації музичних композицій

(тема)

Виконав:
студент 2 курсу, групи СШІм-22-3
Чубов К.Р.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)
Керівник проф. Музика К.М.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ В.О. Філатов
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Чубову Кирилу Руслановичу
(прізвище, ім'я, по батькові)

1. Тема роботи Комп'ютерно-інтегрована система генерації музичних композицій

затверджена наказом університету від 1 квітня 20 24 р. № 260Ст

2. Термін подання студентом роботи до екзаменаційної комісії 4 червня 20 24 р.

3. Вихідні дані до роботи Набір шаблонів, що зберігається в базі даних проекту та можуть бути застосовані до різних складових композиції. Кожен шаблон має певний тип і впливає на визначену концептуальну складову треку, таку як: ритм, гармонія, інтервальне зміщення.

4. Перелік питань, що потрібно опрацювати в роботі
1) Аналіз предметної галузі
2) Постановка завдань
3) Проектування дизайну та архітектури програмного забезпечення
4) Розробка програми

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	01.04.2024	виконано
2	Постановка завдання та огляд існуючих рішень	06.04.2024	виконано
3	Хід розробки	09.04.2024	виконано
4	Опис ключових алгоритмів	16.04.2024	виконано
5	Огляд результатів та подальший розвиток	23.04.2024	виконано
	програми		
6	Написання пояснювальної записки	29.04.2024	виконано
7	Перевірка на академічний плагіат	30.04.2024	виконано
8	Підготовка презентації та доповіді	06.05.2024	виконано
9	Рецензування	09.05.2024	виконано
10	Захист перед ЕК	04.06.2024	

Дата видачі завдання 1 квітня 20 24 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Музика К. М.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 106 с., 69 рис., 1 дод., 35 джерел.

АВТОМАТИЗАЦІЯ, МАШИННЕ НАВЧАННЯ, МУЗИЧНІ КОМПОЗИЦІЇ, МУЗИЧНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, НЕЙРОННІ МЕРЕЖІ, ПРОГРАМНІ АЛГОРИТМИ, ТЕОРІЯ МУЗИКИ, ШТУЧНИЙ ІНТЕЛЕКТ.

Об'єкт дослідження – програмне забезпечення для музики, яке реалізує методи зберігання концептуальних даних та їх використання під час створення музичних композицій.

Предмет дослідження – знання музичної теорії, інструменти генеративного мистецтва та можливості програмних алгоритмів для розробки системи, яка може працювати над музичними композиціями на рівні контексту.

Мета дослідження – пошук засобів концептуального підходу до програмних засобів, які призначені для створення музики.

Методи дослідження включали аналіз історії генеративного мистецтва та генеративної музики, а також вивчення теорії музики для виділення концептуальних складових, які стали основою для подальшої розробки алгоритмів. Основними принципами, на яких ґрунтувалася розробка системи, були принципи чистої архітектури SOLID, принципи об’єктно-орієнтованої та функціональної парадигм програмування.

ABSTRACT

Master's thesis contains: 106 pp., 69 fig., 1 ann., 35 references.

ARTIFICIAL INTELLIGENCE, AUTOMATION, MACHINE LEARNING, MUSICAL COMPOSITIONS, MUSIC SOFTWARE, MUSIC THEORY, NEURAL NETWORKS, SOFTWARE ALGORITHMS.

The object of research is music software that implements methods of storing conceptual data and using them when creating musical compositions.

The subject of the research is the knowledge of music theory, tools of generative art and the capabilities of software algorithms to develop a system that can work on musical compositions at the context level.

The purpose of the study is to find means of a conceptual approach to software tools designed to create music.

The research methods included an analysis of the history of generative art and generative music, as well as the study of music theory to identify conceptual components that became the basis for further algorithm development. The main principles on which the system development was based were the principles of the pure SOLID architecture, the principles of object-oriented and functional programming paradigms.

ЗМІСТ

ВСТУП

Актуальність обраної теми дослідження. Тема дослідження полягає в тому, що створення музики визначається перш за все як творчий процес, під час якого композитор розповідає свою історію, користуючись різними доступними методами вираження в цій сфері. Робота над музичною композицією може бути розділена на три основні етапи.

Становлення ідеї: цей етап залежить від багатьох факторів, таких як життєвий досвід композитора, глибина його знань у певній області та культурна освіченість. Ідея може виникнути під впливом життєвих подій, природного середовища, творів мистецтва або літературних ідей. Існує безліч причин, що можуть породжувати ідеї.

Формування ідеї в реальний твір: після виникнення ідеї композитор може відхилити її або реалізувати у формі, зрозумілій для нього. Цей процес можна назвати процесом вираження себе. Наприклад, скульптор виражає свої ідеї, створюючи об’ємні фігури за допомогою ліплення, висікання, лиття тощо.

Реалізація ідеї: на цьому етапі композитор використовує доступні для нього засоби вираження, які він володіє. Це може бути музичні інструменти, розуміння різних аспектів композиції та вміння працювати з музичними матеріалами. Незалежно від форми, цей етап передбачає застосування фактичних навичок і знань, за допомогою яких композитор перетворює ідею в готовий твір.

Сучасні підходи до композиції музики, які більшість музичних програм пропонують, мають проблему у спілкуванні з автором через використання мови конкретних значень. У той час, автор у процесі створення музики працює з концепціями, або зв’язками, які мають більш абстрактний характер.

Для розуміння цієї проблеми, розглянемо докладніше, що мається на увазі під термінами «конкретні значення» та «музичні концепції». Якщо

відмовитися від глибшого вивчення, то ноти можуть розглядатися як конкретні значення в композиції музики. Основним завданням Ноти є позначення сукупності фізичних параметрів звуку, опис частоти амплітуди, коливання, тривалості. Нота дозволяє передавати цю інформацію у вигляді простого символу, не вдаючись у деталі кожного окремого процесу. Отже, нота є абстракцією фізичних характеристик звукової хвилі.

Музичні фрази складаються з нот, музичні партії – з фраз, а цілі композиції – з партій. Важливо зазначити, що з точки зору концепції, конкретні ноти не мають такого великого значення, як зв'язки, які можуть виникнути між ними. Аналогічно до того, як значення окремих слів розкривається лише в контексті речення, а значення речення – в контексті тексту, виникає зв'язок між нотами в музичній композиції, що є важливішим за окремі ноти.

Повертаючись до музики, якщо головною метою взаємодії автора та програми для написання музики є надання закінченої форми ідеям автора, то система може стати досконалою, якщо дозволить працювати не лише з конкретними нотами, але й на рівні контексту. Це означає, що система має здатність розуміти не лише окремі музичні символи (ноти), але й ідеї та концепції, що лежать в основі композиції, і має можливість взаємодіяти з ними безпосередньо.

Прикладами таких концепцій у музичному творі можуть бути різноманітні закономірності в його розвитку.

Протягом історії розвитку музики виникла велика кількість подібних концепцій. Різні народи виявляли аналогічні закономірності, які часто формували в правила (наприклад, лади, тональності, ритми та інші), а інші – у структури, які можна назвати «побажаннями» (наприклад, послідовності акордів, їх зв’язки). Кожна з таких концепцій надає відомостей твору. Наприклад, різні послідовності акордів можуть викликати сумні чи радісні відчуття, гармонійність чи напруженість. Таким чином, кожна складова фактично є як кисть, яка додає свої фарби до загального полотна композиції.

Теорія музики виникла не випадково. Це результат довгих експериментів з відбору та систематизації різних закономірностей, які виявляються під час різних комбінацій простих музичних одиниць, таких як звуки, шуми та голос людини.

Цікаво, що більшість таких закономірностей виникла хаотично – у різних культурах та з різних джерел звуку, але значна частина з них зафіксувалася, навіть якщо це сталося випадково.

Сучасний композитор, працюючи над новою композицією, виражається за допомогою відомих йому музичних закономірностей або експериментує, шукаючи нові. У будь-якому випадку результат його творчості буде включати певний набір закономірностей, навіть якщо він не має явного уявлення про них або знаходить їх випадково. Ці закономірності можна назвати «концепціями» у даному контексті.

У чому суть проблеми у використанні конкретних значень? Насправді, немає нічого неправильного у їхньому використанні. Іноді навіть простіше висловити ідею конкретно, ніж намагатися знаходити відповідні абстракції. Проте проблема полягає в тому, що рівень роботи з концепціями в сучасних DAW (програмах для написання музики) взагалі відсутній. А використання такого рівня могло б приносити безліч переваг:

– зберігання інформації про зв'язки, а не конкретних нот, спростило б редагування матеріалу при зміні певних концептуальних аспектів композиції;

– автор міг би працювати на тому рівні, який зараз для нього зручний. Він міг би редагувати композицію як на рівні концепцій, так і на рівні деталей реалізації. Це схоже на роботу широкими мазками та більш детальними, з можливістю в будь-який момент замінити одні на інші;

– це дозволило б створювати музику для людей, які мало знайомі з її теорією, оскільки їм не потрібно було б заглиблюватися у деталі реалізації, а лише відбирати концепції та передавати їх програмі.

Мета та завдання дослідження полягають у пошуку та втіленні засобів

концептуального підходу до програмних засобів, які призначені для створення музики. Важливо відзначити, що наша мета не полягає в створенні системи, яка автоматично генерує музику «з нуля». Існують вже такі системи, але вони не є надійним інструментом для втілення задумів композитора, оскільки приймають рішення лише на основі жорстко встановлених програмних алгоритмів.

Об'єктом дослідження є програмне забезпечення для музики, яке реалізує методи зберігання концептуальних даних та їх використання під час створення музичних композицій.

Предметом дослідження є знання музичної теорії, інструменти генеративного мистецтва та можливості програмних алгоритмів для розробки системи, яка може працювати над музичними композиціями на рівні контексту.

Методи дослідження включали аналіз історії генеративного мистецтва та генеративної музики, а також вивчення теорії музики для виділення концептуальних складових, які стали основою для подальшої розробки алгоритмів. Основними принципами, на яких ґрунтувалася розробка системи, були принципи чистої архітектури SOLID, принципи об’єктно-орієнтованої та функціональної парадигм програмування.

При розробці алгоритмів враховувалася мета зробити їх максимально ефективними. Для цього ми використовували підхід «розділяй та володарюй», який передбачав розбиття складної задачі на менші, що дозволяло вирішувати їх рекурсивно. В магістерській роботі для створення інтерфейсу програми, було обрано мобільну систему Android. Для досягнення поставленої мети ми використовували математичні обчислення та вирішували безліч логічних задач з метою перетворення розбіжних концепцій у функціонуючу систему.

Наукова новизна отриманих результатів полягає у революційному підході до творення музики, що використовує концептуальні закономірності

замість прив'язки до конкретних значень у процесі розробки музичної композиції у віртуальному середовищі.

Практичне значення отриманих результатів полягає у створенні системи, яка має здатність виявляти та зберігати зв'язки між різними елементами композиції. Ці зв'язки базуються на принципах музичної теорії. Створена система значно полегшує внесення концептуальних змін, таких як ритмічний малюнок або тоніка, оскільки вони автоматично впроваджуються на рівні реалізації. Через часткову автоматизацію програма дозволяє робити зміни на рівні деталізації значно швидше, ніж це можливо при ручному редагуванні, що в декілька разів прискорює процес творення музики.

1 ПОСТАНОВКА ЗАВДАННЯ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Поняття «генеративна музика». Напрями розвитку

Генеративна музика, або «Generative music» у англомовному контексті – це напрямок у музичній творчості, що ґрунтується на використанні алгоритмів та меті досягнення змінності у творчому результаті. Термін став поширеним завдяки британському музикантові-електроннику Браяну Іно, який у співпраці з компанією SSEYO у 1996 році випустив програму під назвою «Koan» для створення «генеративної музики», а також серію з 12 композицій «Generative Music 1 with SSEYO Koan software», створених за допомогою цієї програми. Р. Вуллер виділяє чотири основні напрями розвитку генеративної музики [1]:

– лінгвістичний/структурний підхід використовує алгоритми, спрямовані на формування структурно зв'язаного матеріалу, ґрунтуючись на досягненнях Генеративної граматики Чомскі та на музикознавчих дослідженнях Фреда Лердала і Рея Джекендоффа. Алгоритми базуються на деревоподібній структурі;

– інтерактивний/поведінковий підхід вимагає, щоб музика породжувалася системою без входів, і характеризується як «нетрансформаційна». Як приклад можна вказати Generative Music 1 Браяна Іно;

– креативний/процедурний полягає в створенні музики композитором, наприклад твори It's Gonna Rain Стіва Райха та In C Террі Райлі;

– біологічний/послідовний підхід описує недетерміновану музику, яка не може бути відтворена. Автори цієї концепції порівнюють процес створення музики композитора з роботою вчених-селекціонерів щодо виведення нових видів («Вірусна симфонія» Йозефа Нехвatала).

Українська аудиторія вперше зіткнулася з генеративною музикою на

академічній сцені під час проекту EM-VISIA у 2011 році, що відбувся під егідою фестивалю «Форум музики молодих», ініційованого композиторкою Аллою Загайкевич [8]. Програмне забезпечення використовувалося для створення музичного нотного тексту, в той час як виконання музики покладалося на музикантів-інструменталістів.

Метод генерації, що використовується у віртуальному генеративному мистецтві, має глибокі історичні корені та базується на елементі випадковості. Цей підхід існував і задовго до настання ери персональних комп'ютерів. У 1751 році англійський музикант Вільям Хейс винайшов оригінальний метод створення музики, який описав у своєму сатиричному трактаті, який підтверджує, що ідеї генеративного мистецтва та використання випадковості в творчому процесі існували задовго до сучасних технологій.

«Новий підхід» полягав у використанні звичайної щітки (або навіть зубної), яку треба було занурити в чорнильницю і, потім проводити пальцем по щетині, щоб розпорошити чорнило на аркуші нотного паперу. Отримані плями повинні були вказувати місце ноти на лініях нотного запису. Потім до них додаються тактові розмітки, ключі тощо. Важливо зазначити, що ці елементи також обиралися не «композитором», а відповідно до карти з колоди, яка випадала йому на долю. Після всіх цих творчих маніпуляцій «твір» був готовий до виконання [9, с. 34].

Уявлення про непередбачуваність і випадковість результатів творчого процесу набуло досить широкого поширення в Західній Європі у XVIII столітті і породило таке явище, як «Musikalisches Würfelspiel» (нім.) – музична гра в кості, яка полягала в створення «випадкових» музичних творів за допомогою кубиків. «Musikalisches Würfelspiel», як стверджували виробники цих ігор, дозволяло будь-якому аматору створити меню, контртанець, вальс чи марш – без необхідності вивчати теорію музики [10].

Ідея полягала в тому, щоб створити, наприклад, менует, складаючи готові, наперед визначені музичні частини у порядку, встановленому за

допомогою кидання грального кістки. Великі композитори брали на себе авторство подібних захоплюючих ігор [10].

У XVIII столітті виникла перша версія «секвенсера» – шарманка. За її допомогою «мелодії» та «акомпанемент» наносилися на валик за допомогою спеціальних шпильок. При перегортанні валика відбувалося відкриття доступу до потрібних труб, після чого музикант лише натискав на кнопку «Play», щоб розпочати програвання записаної на валику музики.

У XIX столітті І.Н. Мельцелем було винайдено метроном (1814 рік) і механічний музичний інструмент – пангармонікон (Panharmonicon). Для цього інструменту спочатку Л. Бетховеном була написана композиція «Битва при Вітторії» (1813 рік), більш відома як симфонічна п'єса.

У 1837 році Ч. Пейджем була винайдена «гальванічна музика», що відзначилася новаторським підходом до використання електрики у музичному процесі.

У XVIII столітті розуміння звуко висотності перейшло від простого вивчення музичних ладів до складнішого аналізу природи інтервалів. Фундаментальні поняття, такі як консонанс та дисонанс, стали розглядатися в контексті стійкості і нестійкості, а їх властивості були аналізовані у зв'язку зі специфікою музичних ладів. Також на той час з'явилося розуміння про те, що дисонанс може бути розв'язаний шляхом внутрішньо-ладових взаємодій.

На той момент лад розглядався як основний принцип музичної логіки, і таке розуміння звуко висотності іноді можна зустріти і сьогодні [12].

У XIX столітті Гельмгольц став першим у світі, хто експериментував із подібними до електронних звуків, створивши апарат, відомий як резонатор Гельмгольца. Він також написав кілька значущих праць, які досліджували фізичні та фізіологічні аспекти музики. Одна з них, книга «Вчення про звукові відчуття як фізіологічна основа теорії музики», містить детальний аналіз фізіології слуху, формування звукового тембру, консонансу і дисонансу, а також організації висоти. Одна з них, книга «Вчення про звукові відчуття як фізіологічна основа теорії музики», містить

детальний аналіз фізіології слуху, формування звукового тембру, консонансу і дисонансу, а також організації висоти. Варто зауважити, що Гельмгольц висунув принципи, які стосувалися концепцій консонансу та дисонансу, що співпередбачало ідеї про гармонію, розвинуті пізніше А. Шенбергом [13].

Пізніше, американським винахідником Е. Греєм був розроблений інструмент під назвою Harmonic Telegraph, який був унікальним у своїй конструкції. Кожна клавіша цього двооктавного інструменту володіла власним електромагнітним генератором з відповідною частотою. Звук, що видавався інструментом, був досить слабким, а тембр схожий на телефонний гудок.

Браян Іно, визнаний англійський музикант, відомий як піонер музичного жанру ambient, використовував генеративний підхід у своїй творчості. Наприклад, на його альбомі «Discreet Music» (1975) знаходиться композиція з однойменною назвою, яка триває тридцять хвилин. Ця музична композиція є експериментом із ефектом затримки відлуння, де дві прості мелодії, що схожі за структурою, записані на магнітофонну стрічку, відтворюються в різних послідовностях і накладаються одна на одну. Це призводить до випадкової зміни тембру синтезатора на виході.

Ці та подібні експерименти відіграли ключову роль у започаткуванні явища, відомого як семплювання (sampling) – процес компонування готових звукових фрагментів (семплів), переведених у цифровий формат. Без цього явища неможлива уявити сучасну музичну індустрію.

Одним із найяскравіших прикладів може бути творчість американського композитора Джона Кейджа, який є основоположником алеаторики. У його музиці центральне значення відводиться принципу випадковості. Твори Кейджа передбачають вільне переміщення слухачів, їхню відкритість та занурення в акустичні події (happening), де різні композиції можуть змішуватися, а звучання на різних майданчиках може відрізнятися [15].

Одна з найвідоміших творів Кейджа − «4,33» (1952) − можна вважати вершиною музичного генеративного мистецтва. У цій композиції композитор встановив лише тривалість звучання – 4 хвилини та 33 секунди, а все інше залишалося повністю випадковим. Без участі музикантів, у повній тиші, кожне «виконання» створювало нову «мелодію» з моменту її виникнення – завдяки різним зовнішнім звукам, таким як шелест, шум аудиторії або шум вулиці [16].

Коли композитор використовує традиційну технологію для створення музики, його можливості обмежуються конкретним арсеналом інструментів і тембрів. Загалом вважається, що робота композитора полягає у виборі та поєднанні нот різних висот, тривалостей та гучностей. Отже, інструмент, на якому виконується музика, і його темброве забарвлення зазвичай вважаються менш важливими. Проте, з часом стає очевидним, що тембральні характеристики музики грають велику роль у сприйнятті її настрою та емоційного відтінку. Іншими словами, одна й та ж мелодія, яка виконується на тромбоні та скрипці, може сприйматися слухачем по-різному.

Зі становленням музичної традиції композитори регулярно обирали конкретні інструменти або навіть цілий симфонічний оркестр для втілення своїх творчих ідей. Оркестр вважався найбагатшим засобом виразності виконавських можливостей, і цей висновок переживав час і залишався актуальним і сьогодні. Однак навіть у створенні музики для оркестру, з його широким спектром інструментів, композиторів обмежували певні фіксовані темброві характеристики, що заздалегідь визначались. Для того щоб розширити ці рамки, вони вигадували темброві доповнення, відомі як «міксти». Спочатку вони обирали найбільш природні поєднання інструментів, що гармонійно поєднувалися, а потім почали експериментувати з більш незвичайними поєднаннями тембрів. З виникненням музично-комп'ютерних технологій (МКТ) та музичного комп'ютера (МК) композиторам відкрилися безмежні можливості для

створення та використання звуків будь-якого бажаного тембру. Сучасні МКТ усувають усі темброві обмеження, залишаючи лише межі професійного програмного забезпечення та вміння композитора користуватися ним.

Сучасні дослідження у музикознавстві часто спрямовані на використання математичних методів, що полягають переважно у створенні музичних композицій з урахуванням закономірностей, виявлених під час аналізу музичних творів, про що докладно розповідали в наших дослідженнях [13]. Як зазначає М.С. Заливадний, у загальному розумінні це може бути предметом окремої теоретичної дискусії, адже воно відображає комплексність соціально-психологічних аспектів творчості у музичній та інших мистецьких сферах [18, с. 328]. Також він вказує на фундаментальний характер елементу програмування (або, точніше, алгоритмізації), який, у різних формах, має значення у всіх сферах людської діяльності [18, с. 12].

У своїх наукових дослідженнях А.Сіхра активно використовував комплексний науковий підхід до вивчення законів музики, зокрема із застосуванням ідей і методів кібернетики та теорії інформації. Він інтегрував елементи різних точних дисциплін, таких як кібернетика та теорія інформації, в рамках досягнень комплексного теоретичного музикознавства 20 століття. Одним із розділів його знаменитої книги «Музика очима науки» [19] є «Музика і кібернетика».

Генеративний підхід, що базується на елементі випадковості, спочатку з'явився у контексті експериментів у музиці та літературі. Протягом другої половини XX століття цей метод вдосконалився та перетворився на самостійний художній напрямок, відомий як генеративне мистецтво.

Цей напрямок передбачає використання автономних систем, таких як генератори псевдовипадкових чисел, для створення унікальних художніх ефектів, які можуть бути несподіваними як для глядача, так і для самого автора. Це явище має синтетичний характер, його коріння сягає ідей і принципів класичного російського авангарду, дадаїзму, сюрреалізму та

мінімалізму, які послужили благодатним грунтом для подальших творчих пошуків, уже з використанням сучасних комп’ютерних технологій.

Для глибшого розуміння суті генеративного мистецтва важливо чітко визначити його межі та виокремити можливі підвиди. Це стає особливо актуальним, оскільки вже були спроби розгляду подібних питань, а одні з найбільш серйозних належать професорові когнітивних наук університету Сассекса Маргарет А. Боден. Обидва дослідники написали десятки наукових праць про сучасні арт-практики, які існують на перетині традиційних та пост-інформаційних методів виразності, технік і засобів, внесли великий вклад у розуміння питань цифрового трансформування художньої творчості.

У своїй статті «Розкриття сутності генеративного мистецтва» Боден та Едмондс виділяють 11 взаємозалежних типів мистецтва: Електро-арт (електронне), Комп'ютерний-арт (C-art), Комп'ютерно-опосередкований-арт (CA-art), Цифровий-арт (D-art), Власне генеративний-арт (G-art), Комп'ютерно-генеративний-арт (CG-art), Еволюційний-арт (Evo-art), Робототехнічний-арт (R-art), Інтерактивний-арт (I-art), Комп'ютерно-інтерактивний-арт (CI-art) та Мистецтво віртуальної реальності (VR-art).

На основі цієї класифікації Бодена та Едмондса, В. Лукічев запропонував свою власну класифікацію (рисунок 1.1) [21].

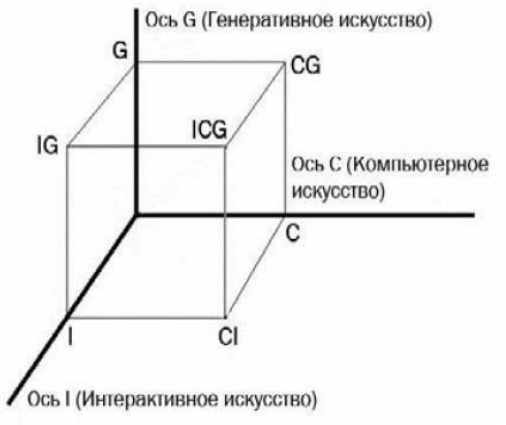


Рисунок 1.1 – Основні напрями генеративного мистецтва

Лукічев описав генеративне мистецтво через три ключові виміри [21]:

– вісь G («генеративна») – це про походження вмісту та стилю твору, які залежать від випадкових чинників;

– вісь C («комп'ютерна») – включає апаратні та програмні засоби, що використовуються для створення мистецьких творів;

– вісь I («інтерактивна») – розглядає способи взаємодії глядача з твором, можливості участі глядачів у його створенні та розвитку.

G-art (Generative art) – це створення художніх об'єктів, які є результатом принаймні частково автономних процесів фізичних характеристик, що відбуваються без безпосереднього втручання автора (G=1, C=0, I=0).

C-art (Computer art) – це художні об'єкти, створені з використанням комп'ютерних технологій, де вирішальну роль відіграє робота з комп'ютерними програмами та апаратурою (G = 0, C = 1, I = 0).

Зазначимо, що терміни «комп'ютерне мистецтво» і «цифрове мистецтво» часто використовуються як синоніми, але це не повністю точно: поняття «комп'ютерне мистецтво» має більше охоплення і включає в себе навіть використання аналогових пристроїв, які досі застосовуються музикантами, коли вони бажають передати візуальну або аудіо інформацію без втрати початкової якості, що неминуче виникає при конвертації аналогового сигналу в цифровий і навпаки [22].

Інтерактивне мистецтво (I-art) – це колекція творів, форма або зміст яких значно залежить від участі глядача.

Інтерактивне комп'ютерне мистецтво (CI) включає в себе асортимент творів, що були створені за допомогою комп'ютерів або інших мультимедійних пристроїв. Ці твори передбачають активну участь глядача у процесі мистецтва, що може бути повною або частковою, і не використовує автономні системи.

Існує безліч прикладів інтерактивного медіа-мистецтва, оскільки ця форма виявляється дуже популярною. Початково комп'ютери взаємодіяли з

користувачами за допомогою інтерфейсів, тому природно, що залучення глядача в ігровий арт-процес за допомогою мультимедіа відбувається майже непомітно. Це особливо очевидно в наші дні, коли сенсорні екрани та різні Інтернет-сервіси, такі як соціальні мережі, відео-чати та мобільні додатки, є частиною нашого щоденного життя. Отже, коли відвідуєте виставку інтерактивного медіа-мистецтва, глядач вже має певний рівень досвіду і інтуїтивно розуміє, як користуватися пристроями: на яку кнопку натискати, якої частини екрану торкатися, куди рухати рукою та інші аспекти.

CG-art (Computer-Generated art) – це форма комп'ютерного мистецтва (G = 1, C = 1, I = 0). Твори в цій категорії створюються за допомогою комп'ютера або іншого мультимедійного пристрою шляхом послідовного виконання команд, які складаються художником-програмістом. Ці команди включають в себе використання генератора випадкових чисел для візуалізації отриманих результатів. У контексті даної категорії мистецтва відсутня можливість взаємодії, як показано на схемі: значення по вісі I становить нуль.

Розробник, якщо запустив програму, може піти на обід: комп’ютер, виконуючи заданий алгоритм, самостійно створює арт-об’єкт, приймаючи «автономні рішення» там, де автор коду передбачив таку можливість. Рішення машини базуються на випадкових числах, що породжує безліч варіацій – виконавцю (або співавтору) музики залишається лише вибрати найбільш прийнятне рішення серед усього різноманіття, яке пропонується.

IG-art (Interactive Generative art) – це асортимент творів, де глядач активно бере участь у процесі їх створення, а також використовуються «традиційні» автономні системи, які генерують елементи залежно від випадкових факторів, незалежно від мультимедійних технологій. (G=1, C=0, I=1).

ICG-art (Interactive Computer-Generated art) – це набір творів, які створені за допомогою автономних комп'ютерних систем і сприймають активну взаємодію з глядачем. (G=1, C=1, I=1).

«Випадковість у квадраті»: з одного боку, машина з генератором чисел створює аудіовізуальний контент за заданими художником початковими параметрами, як у комп'ютерному ген-арті. З іншого боку, «гра випадку» виникає через дії користувачів, які своїми рухами тіла, тембром голосу, діями або бездіяльністю вносять хаос і передбачуваність, залежно від функціоналу конкретної інтерактивно-генеративної медіа-моделі.

Таким чином, генеративне мистецтво, взаємодіючи з інтерактивним та комп'ютерним мистецтвом, може виявляти властивості одного з них, додавати свої унікальні аспекти, або поєднувати у собі навіть усі три вектори, утворюючи складну автономну систему, що базується на мультимедійних технологіях і передбачає активну взаємодію з глядачем.

1.2 Огляд існуючих рішень

Визначаючи перелік умов майбутньої програми та починаючи її проектування, важливо враховувати наявні на даний момент рішення, які можуть частково вирішити проблему концептуального підходу (рисунок 1.2).

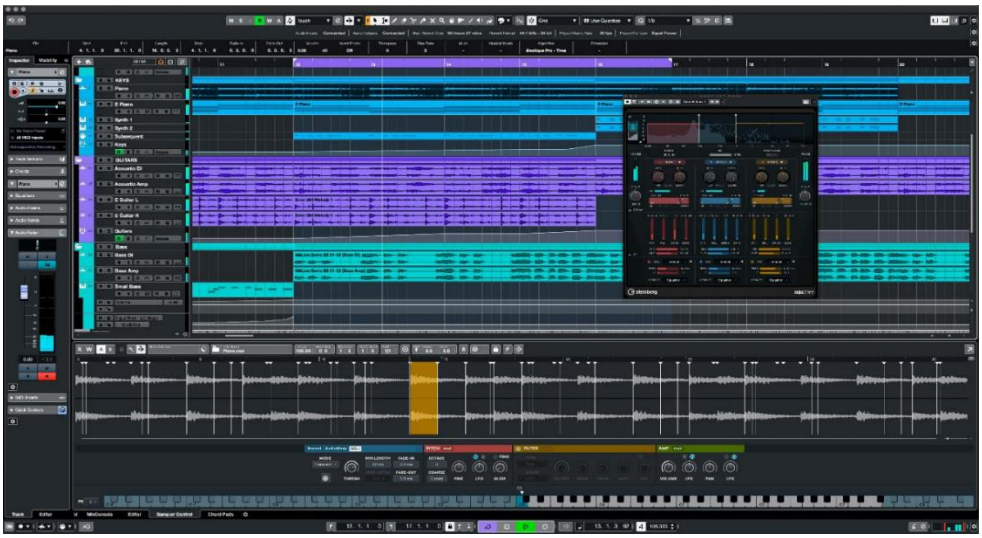


Рисунок 1.2 – Cubase 10 програмного інтерфейсу

1.2.1 DAW

Цифрова аудіо студія (DAW), яка використовується музичними продюсерами, є електронною системою для запису і редагування цифрового аудіо. Основна особливість таких робочих станцій полягає в можливості вільно маніпулювати записаним звуком. Більшість DAW також підтримують технологію MIDI. Якщо MIDI-редактори працюють із MIDI-даними, а аудіо-редактори обробляють аудіо файли, то DAW вміє все це і ще більше. Цифрові аудіо студії можна розглядати як універсальні інструменти для створення музики, оскільки вони об'єднують у собі все необхідне для перетворення музичної ідеї в якісний музичний продукт.

Кожна програма DAW має широкий спектр стандартних функцій, таких як редагування MIDI і обробка аудіо та унікальні особливості. Наприклад, у програмі Bitwig є модульна станція синтезу звуку під назвою «Grid». Програми може вдосконалити використовуючи plug-in, та DAW.

Це означає, що ви можете розширити можливості своєї DAW за допомогою VSTi та інших аудіо розширень.

Серед найпопулярніших DAW, таких як FL Studio, Cubase, Ableton Live і Logic Pro, можна відзначити широкий функціонал, особливо це стосується програм, які вже давно присутні на ринку. Проте жодна з них поки не вирішує проблему збереження та редагування музичного матеріалу на рівні понять.

Хоча в цих програмах реалізовано багато функцій, що дозволяють одночасно редагувати великі обсяги даних, наприклад, змінювати висоту і часову позицію музичних фрагментів, виділяти і переміщати їх як ціле або навіть змінювати тональність цілої композиції за допомогою однієї ручки, але концептуальні зв'язки між компонентами не збережені.

Ймовірно, така функціональність не з'явилася досі лише через відсутність чіткого розуміння того, як саме вона має бути реалізована та як зробити її зручною та зрозумілою для кінцевого користувача.

Переваги DAW:

– великий асортимент функцій;

– усі інструменти для створення музики доступні в одній програмі;

– можливість проводити докладне редагування композиції;

– можливість розширення функціоналу за допомогою сторонніх програм.

Недоліки DAW:

– відсутність збереження інформації;

– відсутність інструментів для швидкого редагування цих зв'язків.

1.2.2 Програми повної генерації

Одним із таких унікальних сервісів є онлайн-платформа Mubert. Тут користувач обирає один із трьох критеріїв для подальшого уточнення: жанр, настрій чи вид діяльності. Для категорії «діяльність» можуть бути доступні такі підкатегорії, як «концентрація», «спорт», «кафе» тощо. В якості останнього параметра користувач вибирає довжину треку. Після цього програма, використовуючи базу семплів та програмні алгоритми, генерує музичний трек відповідно до вибраних параметрів (рисунок 1.3).

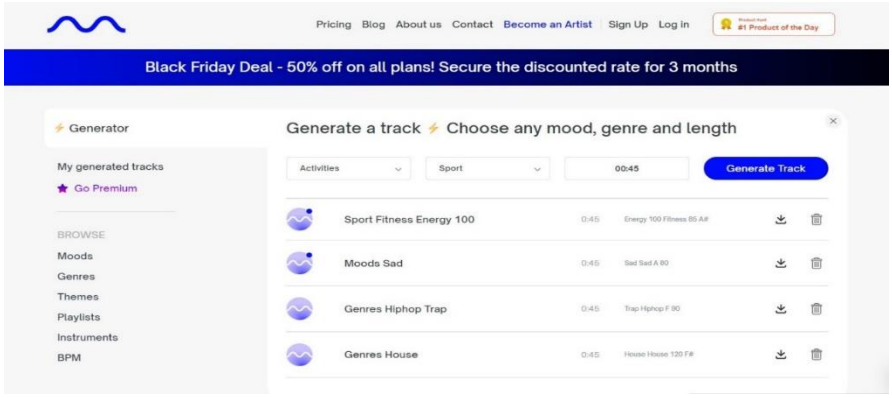


Рисунок 1.3 – Mubert Онлайн-сервіс

AIVA, програма особливо відома своїми оркестровими композиціями, які важко відрізнити від створених людиною, перший визнаний віртуальний композитор за версією SACEM [25] (рисунок 1.4).

Рис. 1.4 – AIVA віртуальний композитор

Фактично, існує значна кількість схожих програм, проте їх головним недоліком є занадто великий фокус на «автономності».

Переваги:

– концептуальний підхід;

– можливість редагування партій після завершення генерації;

– результати високоякісної генерації, яку важко відрізнити від музики, створеної людиною.

Недоліки:

– автономне прийняття рішень, що ускладнює визначення авторства результатів. Більшість програм вимагають лише вказання жанру або настрою;

– обмежена варіативність, оскільки всі результати формуються за однаковим принципом, що призводить до обмеженості у різноманітності;

– більшість програм не мають можливості детального редагування (хоча це реалізовано в AIVA);

– відсутність збереження інформації про зв'язки, що обмежує їх

можливості на рівні понять, лише до початку генерації, а після неї – лише на рівні редагування конкретних значень, таких як примітки чи зразки.

1.2.3 Міді-процесори

Це група програм, що спеціалізуються на обробці MIDI та можуть містити елементи генерації.

Прикладом такої програми є VST-інструмент Scaler (рисунок 1.5). Ця програма має широкий спектр функцій, серед яких:

– аналіз послідовності MIDI-акордів із визначенням їх типу та тональності;

– різноманітні вбудовані тональності та акорди, які можна використовувати для створення власних композицій;

– можливість перетворення партії, що виконується окремими нотами, у послідовність акордів, де ці ноти будуть основними нотами акорду;

– відтворення послідовності акордів у різних артикуляціях і ритмах, наприклад, у формі арпеджіо.

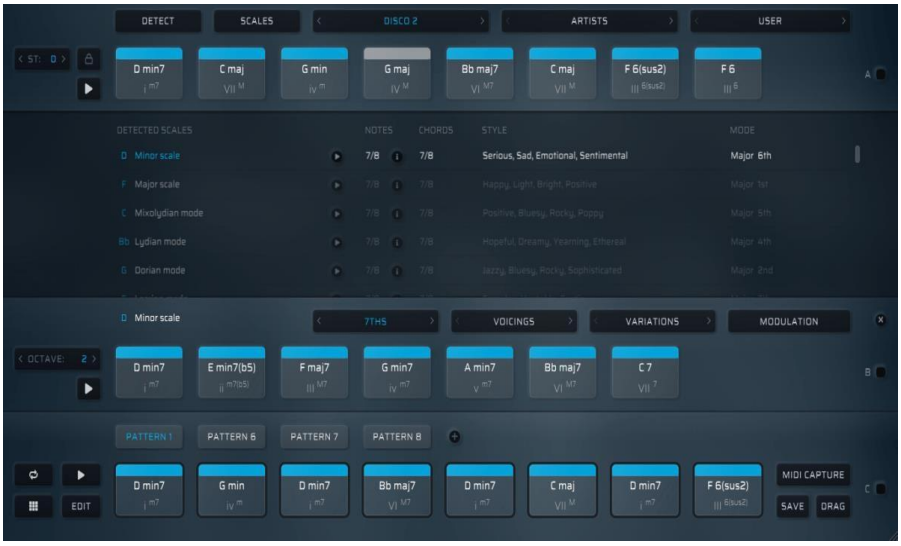


Рисунок 1.5 – Scaler 2. VST-інструмент

За характером роботи «Scaler» найближче до програми, яку ми плануємо впроваджувати. Однак він також не зберігає інформацію про зв’язки між кількома сторонами та бере участь у першому процесі генерації кожної сторони окремо. Крім того, він використовується лише для створення акордів.

Переваги:

– розширений вибір доступних тональностей і акордів, обіцяно більше 1200 варіацій;

– змініть артикуляцію, щоб грати акорди;

– експорт в формат MIDI;

– здатність використовуватися в сторонніх програмах у вигляді розширення.

Недоліки:

– обмежений функціонал щодо одночасного генерування декількох партій;

– самостійно не генерує нічого, використовується лише для перетворення вибраних й акордів в MIDI-послідовності.

1.3 Висновки до розділу 1

Генеративна музика є напрямком творчості в музиці, що ґрунтується на застосуванні алгоритмів та прагненні досягти змінності в творчому процесі. Термін набув популярності завдяки британському електронному музиканту Браяну Іно, який у 1996 році співпрацював з компанією SSEYO для створення програми Koan з метою створення «генеративної музики», а також циклу з 12 композицій «Generative Music 1 with SSEYO Koan software», створеного за допомогою цієї програми.

Генеративна музика може розвиватися у чотирьох напрямах:

– лінгвістичний напрямок використовує алгоритми, спрямовані на створення структурно зв'язаного матеріалу, заснованого на принципах

Генеративної граматики, з використанням деревоподібних структур;

– інтерактивно-поведінковий: музика генерується системою без вхідних даних і характеризується як «нетрансформаційна»;

– творчі процеси створення музики розвиваються або ініціюється композитором;

– біологічно узгоджена: це недетермінована музика або музика, яку неможливо відтворити.

Розвиток музичних комп'ютерних технологій (МКТ) і поява музичного комп'ютера (МК) дали композиторам можливість створювати та використовувати звуки різних тембрів із значною свободою.

Сучасні МКТ зняли більшість обмежень у виборі тембрів, залишаючи обмеження лише на рівні можливостей професійного програмного забезпечення та вмінь композитора управляти ним.

Наша мета не полягає в створенні системи, що повністю генерує музику автономно. Мета нашої роботи полягає у виявленні областей в процесі написання музичних композицій, які можна автоматизувати для покращення ефективності цього процесу.

2 ХІД РОЗРОБКИ

2.1 Вхідні та вихідні дані програми

Програма отримує вхідні дані у вигляді різноманітних умовних шаблонів, що зберігаються в її пам'яті. Користувач може вибирати ці шаблони залежно від своїх вподобань та застосовувати їх до різних частин музичних композицій. Ці умови, що базуються на шаблонах, визначають конкретні характеристики музичних нот у вихідній композиції.

Результатом роботи програми є набір файлів у форматі MIDI, кожен з яких містить послідовність нот, що відповідає вихідній композиції. Ці MIDI файли можна відкривати та редагувати у різних програмах для обробки музики, таких як FL Studio, Ableton, Cubase та інші.

2.2 Короткий опис MIDI-протоколу

Програма використовує midi-файли як вихідні дані, щоб забезпечити сумісність з різноманітними міді-редакторами. Це обрано через широке поширення міді-протоколу як стандартного засобу представлення музичних даних на комп’ютері. Оскільки програма базується на міді, коротке пояснення про цей формат стає важливим елементом розуміння її функціоналу.

2.2.1 Історія виникнення

MIDI протокол стартував у 1982 році як Musical Instrument Digital Interface (цифровий інтерфейс музичних інструментів), виник у зв'язку з необхідністю передачі інформації про натискання клавіш між декількома синтезаторами, які були підключені до однієї мережі. Його основна мета полягала в тому, щоб музикант міг грати на одному синтезаторі, а звук тих

же нот одночасно відтворювався на інших синтезаторах, які були з'єднані з ним послідовно.

Запровадження MIDI відкрило шлях до розвитку синтезаторної пам'яті. Музичні фрагменти, відтворені на синтезаторі, могли бути записані у форматі MIDI, забезпечуючи можливість відтворення у будь-який момент. На відміну від аналогових носіїв, таких як пластинки або магнітофони, MIDI зберігав послідовність нот, що можна було редагувати без втрати якості.

Наступним кроком у розвитку MIDI стала поява комп'ютерних програм – midi-редакторів, що дозволяли створювати та редагувати музичні фрагменти безпосередньо на екрані комп'ютера, не залежно від підключення синтезатора чи інших MIDI-сумісних інструментів.

Пізніше midi-редактори розвинулися в цифрові аудіостудії (DAW – Digital Audio Workstation), що об'єднали процес створення музичних композицій в одній програмі. Вони також включали в себе віртуальні аналоги музичних інструментів та ефектів VST та VSTi, та FL Studio, Ableton Live, Steinberg Cubase, Logic Pro, та інші.

2.2.2 Будова MIDI файлу

Звичайний MIDI-файл складається з послідовності команд, що називаються подіями у протоколі. Кожна подія включає дві частини: команду, що визначає дію, що має відбутися, та відмітку часу.

Наприклад, серед таких подій можуть бути:

– Note on – ініціація звучання ноти;

– Note off – завершення звучання ноти;

– Program change − зміна обраного інструменту.

У MIDI існує розмаїття типів подій, але основні перераховані на рисунку 2.1.

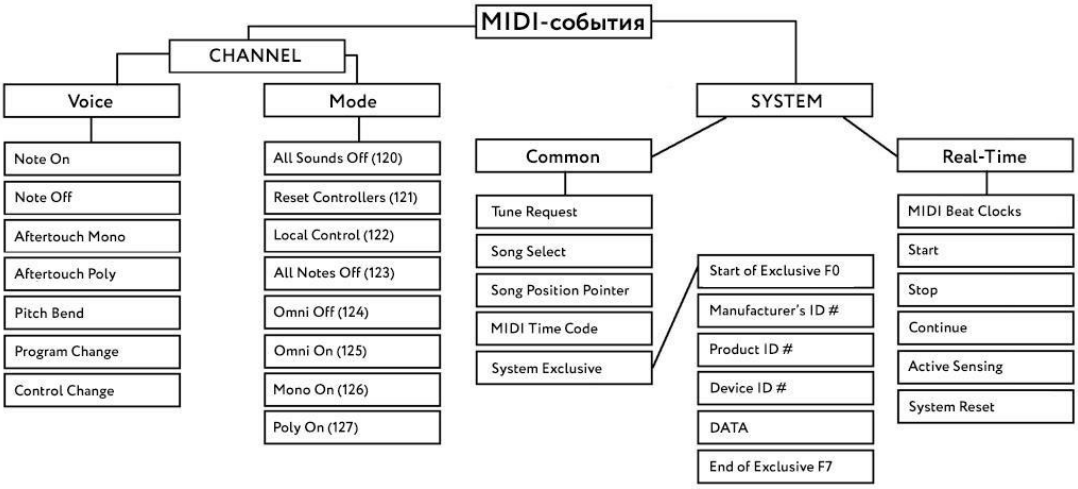


Рисунок 2.1 – Основні MIDI-події

Усю інформацію в MIDI записують у двійковому форматі, що означає, що кожній події відповідає унікальний двійковий код. Це дозволяє міді-редактору розрізняти різні команди. Під час перегляду у звичайному текстовому форматі MIDI-файл буде мати аналогічний вигляд (рисунок 2.2).

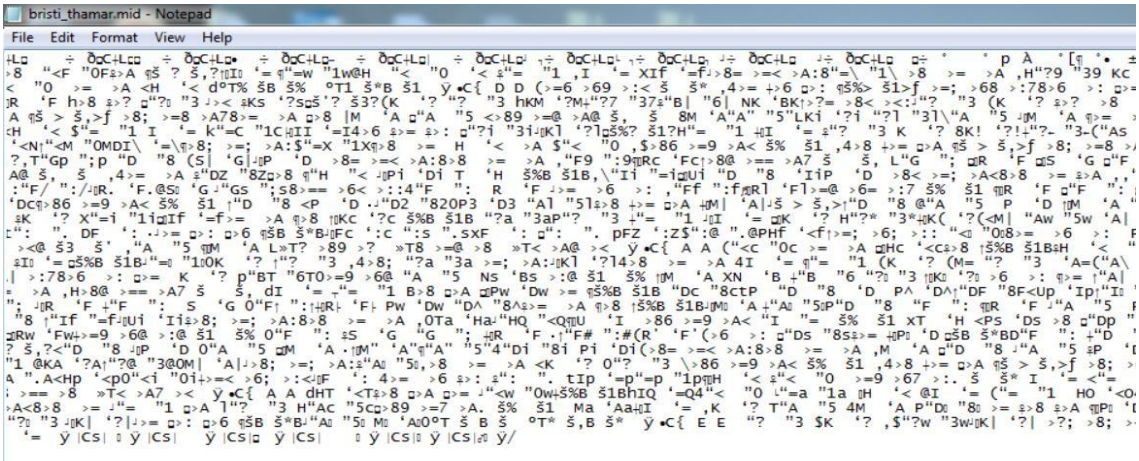


Рисунок 2.2 – Вигляд міді-файлу відкритого в текстовому редакторі

Оскільки редактор в текстовому форматі аналізу послідовність двійкових команд як зашифрований набір символів.

Перетворення файлу MIDI в текстовий формат виконується у наступному вигляді (рисунок 2.3).

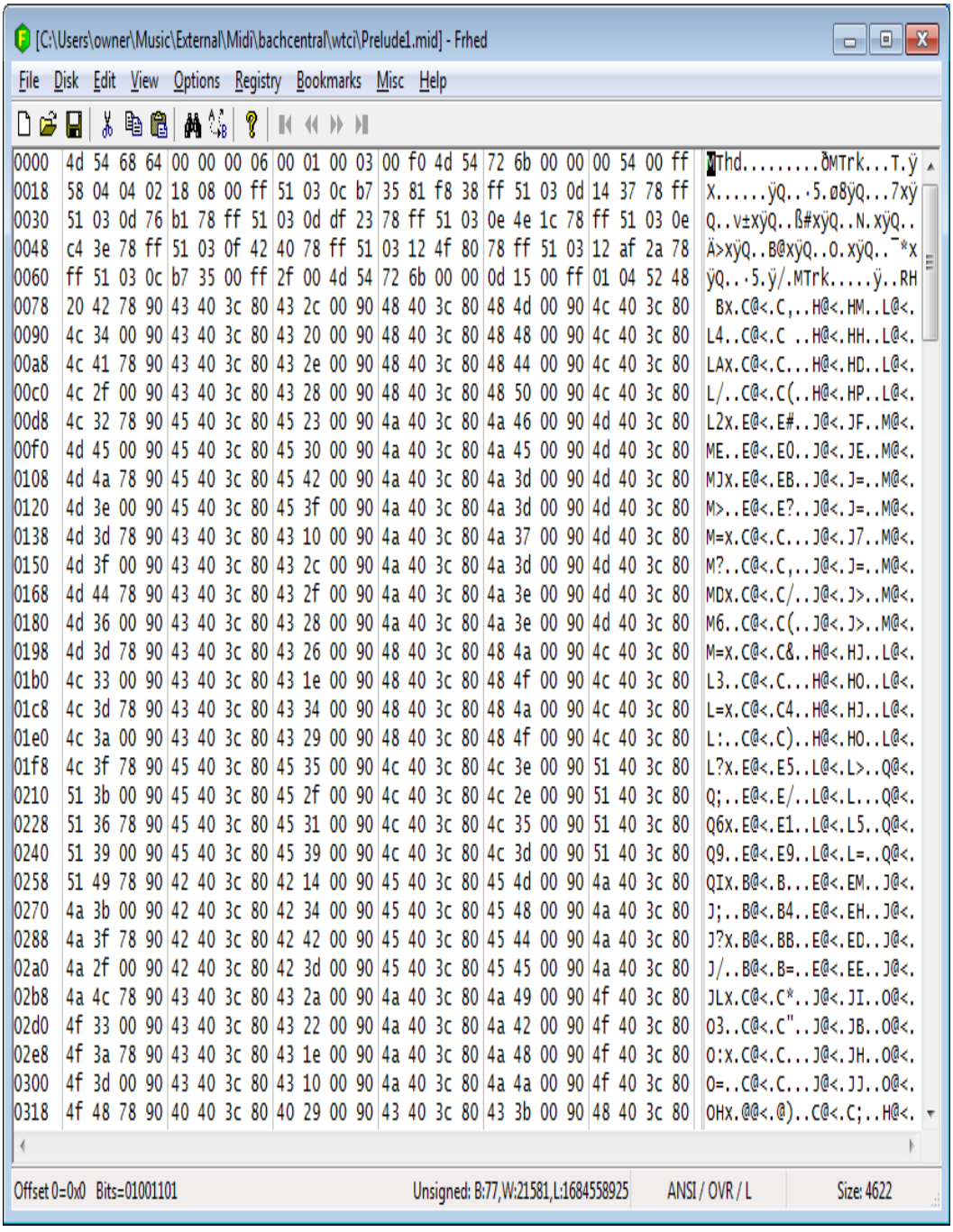


Рисунок 2.3 – Міді-файл в текстовому представленні

За допомогою команд «Note on» або «Note off» передається індекс ноти, для якої має відбутися подія. Індекс ноти визначається позначкою висоти, що відповідає певному музичному звуковому значенню. На прикладі рисунку 2.4 можна побачити різні індекси нот.

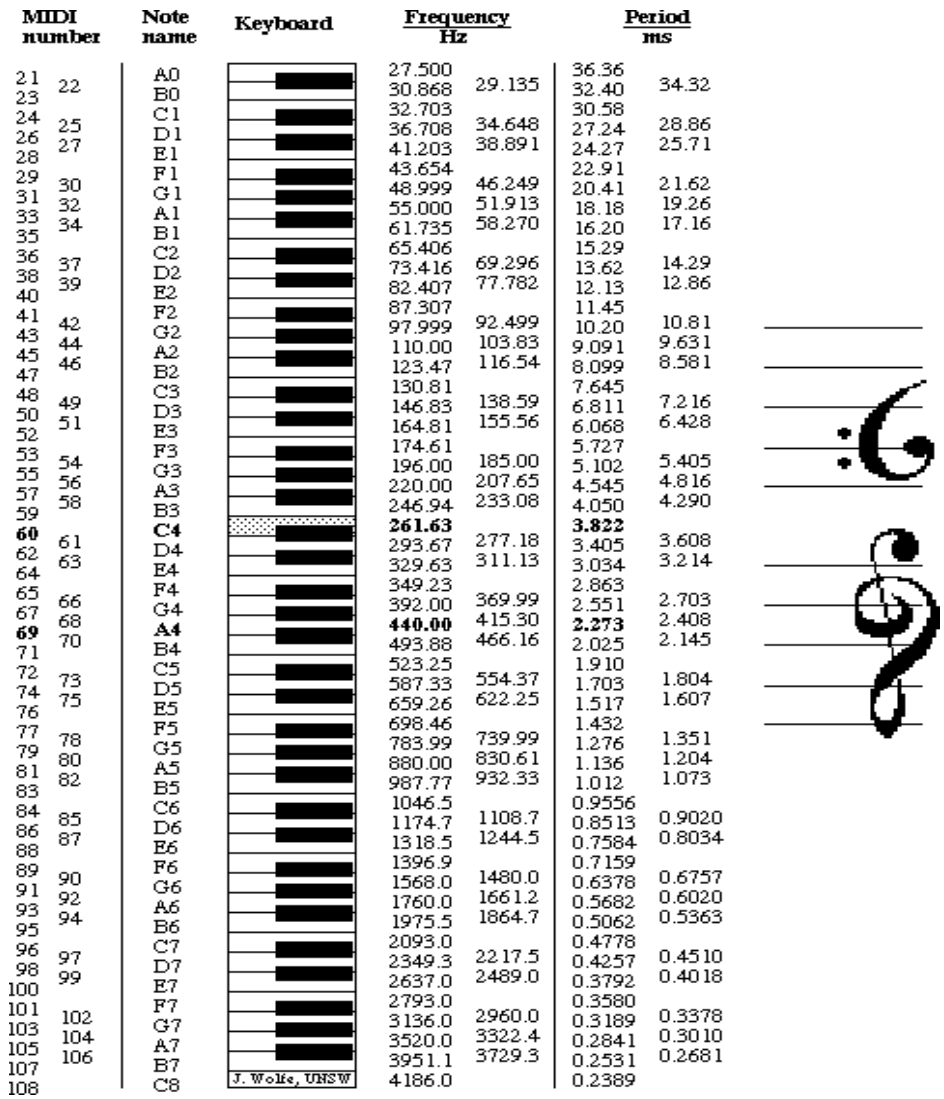


Рисунок 2.4 – Відповідність індексів висоти нот в MIDI та прийнятих позначень в музичній нотації

Отже, у міді-файлі послідовність нот записується через послідовність подій «Note on» та «Note off», які включають в себе висотний індекс ноти та часову мітку, коли клавішу було натиснуто та відпущено.

Для відкриття та редагування міді-інформації існують спеціальні програми, які називаються міді-редакторами. У них музична інформація може бути представлена у вигляді звичайного нотного запису, або у вигляді кубічної сітки, відомої як більш зрозумілий для сучасних музикантів «Piano roll». Piano roll має дві осі: позиція ноти на осі Y відображає її висоту, а позначка часу на осі X (рисунок 2.5).

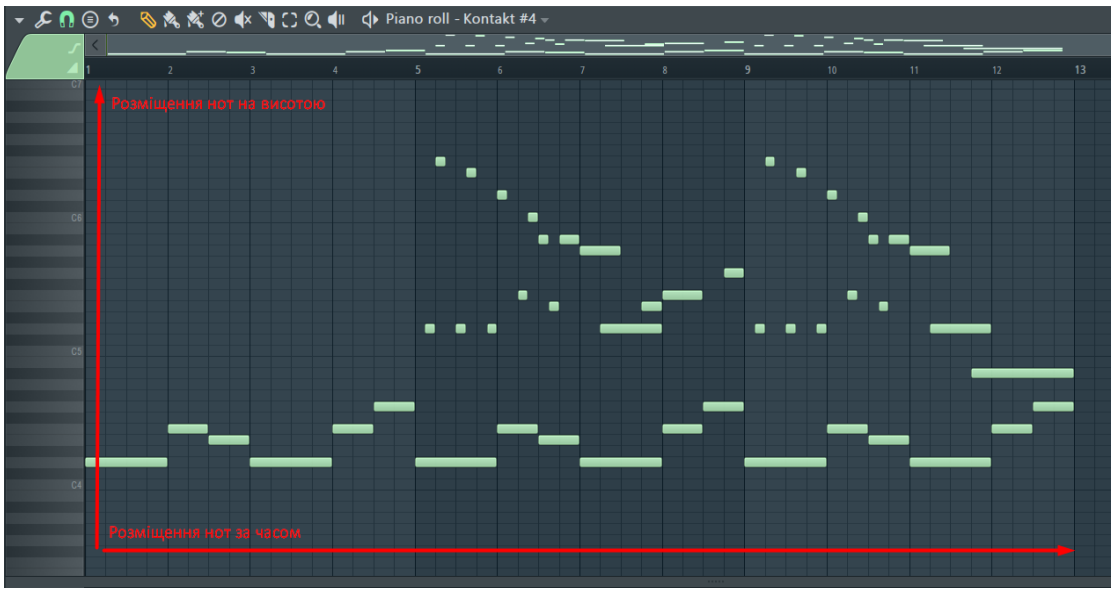


Рисунок 2.5 – Вигляд midi-файлу в Piano Roll FL Studio

2.2.3 PPQN

У MIDI час кожної події виражається не в точних одиницях, таких як секунди чи хвилини, а у відносних значеннях, відомих у протоколі як PPQN (імпульси на четвертну ноту). В академічній музичній нотації також використовуються відносні одиниці для позначення тривалості нот, наприклад цілі, половини, четверті, восьмі, шістнадцяті тощо. Справжня тривалість ноти визначається темпом, який вимірюється в чвертях за хвилину і відомий як BPM (ударів за хвилину).

Маючи приблизне значення ноти і темпу, можна визначити його реальну тривалість в конкретних одиницях виміру. PPQN (імпульси на чверть)- це величина, яка визначає розділення часу в MIDI-файлі та розбиває четвертну ноту на менші частини для більш точного позиціонування кожної події за часом.

PPQN відрізняється, в залежності від MIDI-файлах. Наприклад, у нашій програмі ми встановили його на рівні «32», вважаючи його достатнім для потреб точного часового розподілу подій. Проте це лише дискретна величина, яка може бути іншою.

Просто кажучи, в MIDI для кожної події час мітки вимірюється у кількості імпульсів, які пройшли від початку файлу до моменту виникнення події. Знавши темп BPM та розділення файлу (кількість імпульсів у четвертній ноті), можна конвертувати цей час у конкретні величини.

2.3 Про вибір платформи та мови програмування

Для розробки програми була вибрана платформа Андроїд, що сприяло зручності у написанні програми. Вже протягом двох років я активно працюю з цією платформою і добре знайомий з нею.

Усі алгоритми, що використовуються в програмі (крім тих, що залежать від конкретної платформи, наприклад, збереження до MIDI), залишаються актуальними як для мобільної, так і для інших платформ. Це відкриває можливість в майбутньому перенести програму на комп'ютерну версію.

В якості базової мови програмування обрано Kotlin, що є функціональною мовою з численними перевагами порівняно з Java. Одна з них – лаконічність. Код на Kotlin має менше самоповторень та займає менше місця, що робить його більш зрозумілим та зручним для читання. Це лише одна з переваг цієї мови, і програмісти можуть виокремити багато інших.

2.4 Набір вимог перед початком написанням програми

Перед початком розробки програми, ми чітко визначили вимоги, які вона повинна задовольняти. Ці вимоги стосуються функціональності, якості коду та інтерфейсу користувача. Основні вимоги, які ми висунули перед написанням програми, включають наступне.

Головна функція програми – допомагати музикантові швидше формувати партії та зручно вносити в них зміни.

Таке чітке визначення вимог до програми дозволяє нам зосередитися на розробці необхідного функціоналу та забезпечити високу якість програми, що задовольняє потреби користувачів.

Програма повинна мати можливість зберігати результати у форматі, який дозволяє їх використовувати і в інших програмах. Для цього на практиці реалізується підтримка MIDI-формату, який дозволяє зберігати окремі партії, створені в програмі.

Архітектура програми повинна бути грамотно спроектована, щоб у майбутньому її можна було легко розширювати, додавати нові функції та інше. Для досягнення цього в програмі використовуються принципи SOLID, шаблон MVC та зрозуміле іменування компонентів програми, їх даних та функцій.

Також важливо, щоб програма була оптимізованою. Це означає використання ефективних алгоритмів та оптимізацію використання оперативної та довготривалої пам'яті пристрою. На практиці це може означати, наприклад, перегенерацію лише сегментів композиції, які залежать від змінених сегментів партії, а не всієї композиції, при зміні умов в одній з партій.

2.5 Інтерфейс програми

Інтерфейс, будь то фізичний чи віртуальний, є точкою дотику між користувачем та системою. Він відіграє критичну роль у забезпеченні позитивного досвіду користувача та досягненні цілей системи. Тому основними вимогами до інтерфейсу завжди були його зрозумілість та ефективність. Важливо пам'ятати, що інтерфейс – це не просто візуальна оболонка, а важливий елемент, який впливає на те, як користувач сприймає та взаємодіє з системою.

Зразок інтерфейсу програми FL Studio Mobile (рисунок 2.6).

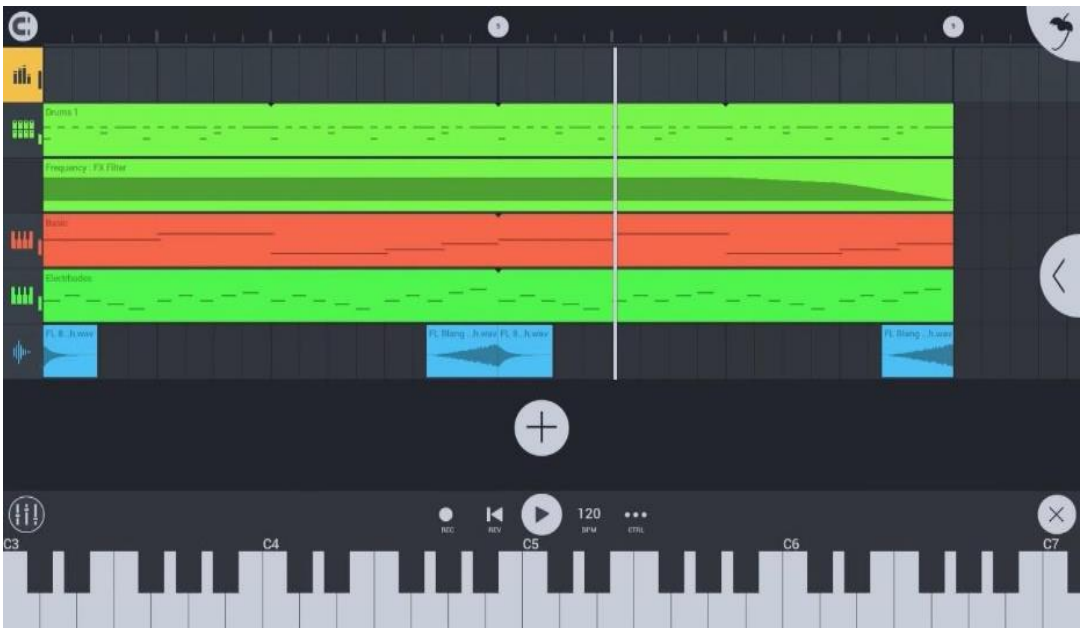


Рисунок 2.6 – Інтерфейс програми FL Studio Mobile

Паперовий макет дизайну наведено на рисунках 2.7 та 2.8.

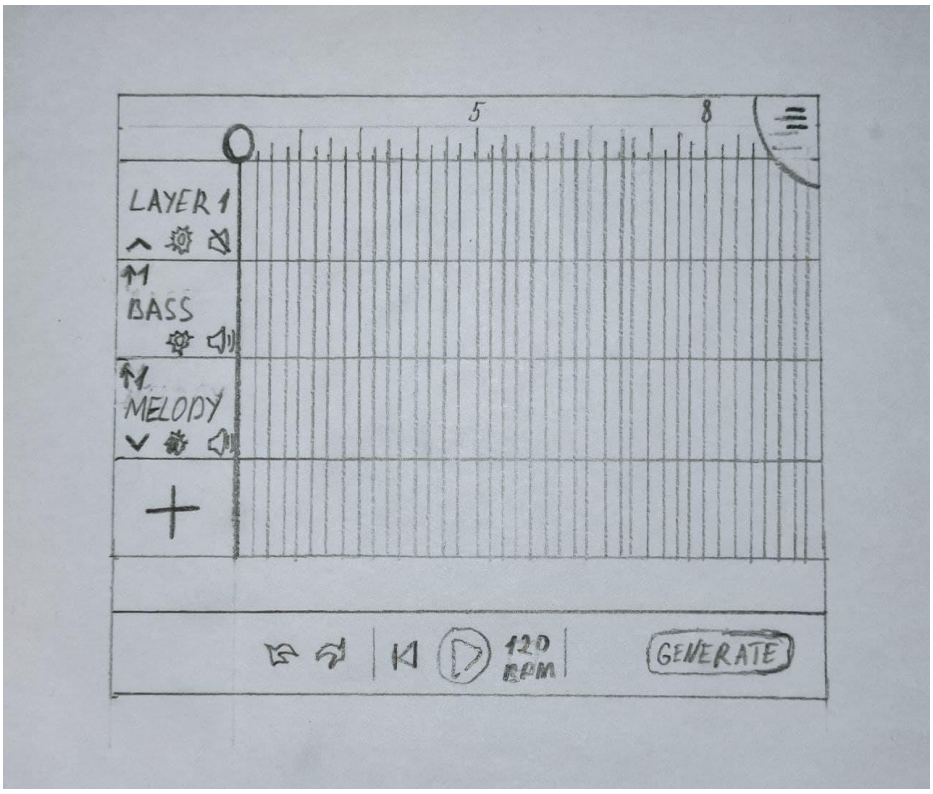


Рисунок 2.7 – Класичний дизайн майбутньої програми

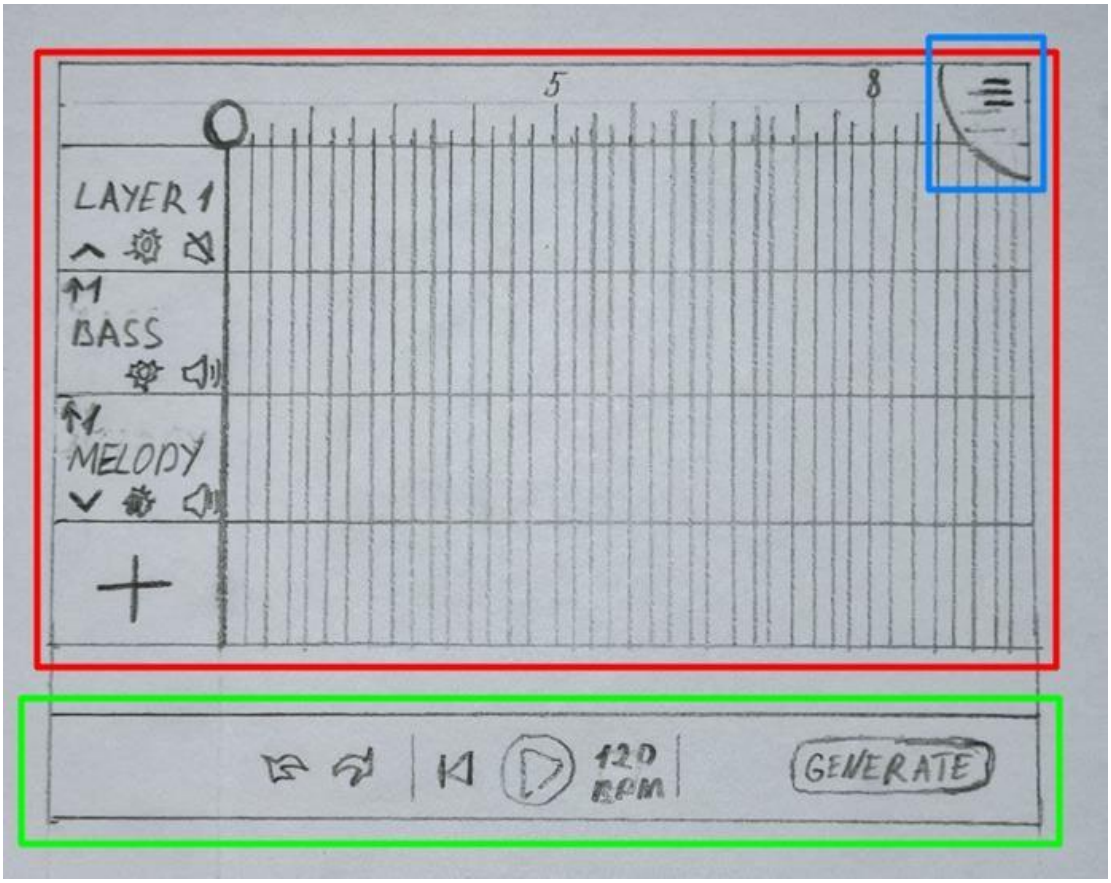


Рисунок 2.8 – Головний екран програми

2.5.1 Огляд дизайну

На головному екрані ви знайдете наступне:

– панель управління (позначена зеленим кольором), яка містить необхідні інструменти для управління програмою;

– область для редагування композиції (позначена червоним кольором), де ви можете редагувати свою музику;

– кнопки для виклику бокового меню (позначені синім кольором), які дозволяють отримати доступ до додаткових функцій та налаштувань.

Панель управління складається з:

– кнопки «Undo» та «Redo» призначені для скасування та відновлення попередніх дій, відповідно;

– кнопка «Go to start» – повертає курсор до початкової позиції

композиції;

– кнопка «Play / Pause» при першому натисканні запускає відтворення, при повторному – призупиняє його;

– «BPM» – це параметр, який дозволяє встановити швидкість відтворення композиції, вимірювану в кількості четвертних нот на хвилину;

– кнопка «Generate» запускає процес генерації, але цей процес не відбувається автоматично після внесення змін. Він активується лише після натискання на вказану кнопку. Ми вирішили використовувати цей підхід через те, що процес генерації може зайняти певний час, тому користувач може бажати запускати його у відповідь на комплексні зміни, а не після кожної найменшої зміни у вихідних умовах. Після початку генерації кнопка зникає, з'являючись лише у разі внесення нових змін до умов генерації.

Кнопка виклику бокового меню

Після натискання на цю кнопку, з правого краю екрану виїжджатиме бокове меню, що містить список операцій:

– відкрити проект (Open);

– новий проект (New);

– відкриття вікно збереження до міді (Export to midi);

– збереження проєкту зі всіма налаштуваннями (Save);

– закрити програму (Close);

– налаштувань програми (Settings).

Під час збереження у форматі MIDI, користувач може вибрати, які дані зберегти (рисунок 2.9):

– треків (червоний колір);

– таймлайн (зелений колір);

– кнопки «Додати трек» (синій колір).

– курсору (помаранчевий колір).

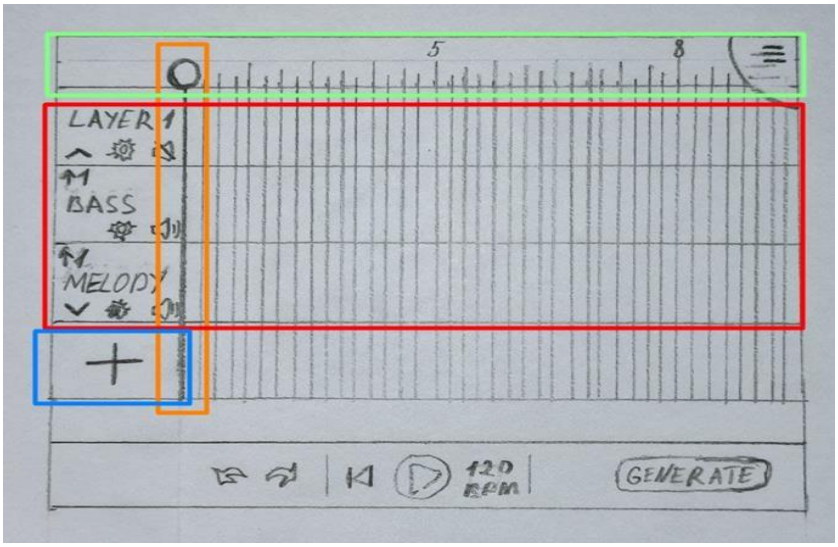


Рисунок 2.9 – Елементи редагування композиції

Таймлайн відображає поділку на часові інтервали (вертикальні лінії по всій області редагування) та нумерацію тактів (у вигляді чисел «5» та «8» на зображенні). Розтягуючи таймлайн, можна змінювати масштаб області редагування (аналогічно до багатьох інших програм).

Складається з вертикального елемента, який простягається вздовж усієї зони редагування, та круглого маркера у верхній частині, за допомогою якого можна пересувати курсор по таймлайну. Під час відтворення композиції курсор переміщується по зоні редагування та таймлайну, вказуючи поточне місце відтворення. Початок відтворення завжди здійснюється з поточного положення курсора.

Кнопка «Додати партію» відкриває діалогове вікно налаштувань партії. Поки що це вікно не було розроблене, але головне, що на ньому можна буде вибрати батьківську партію для нової, ввести ім'я та вибрати музичний інструмент для звучання партії. Після додавання, нова партія буде розміщена останньою серед дочірніх партій обраної батьківської партії..

Трек має дві складові: заголовок і область редагування, яка йому належить.

Заголовок треку (рисунок 2.10).

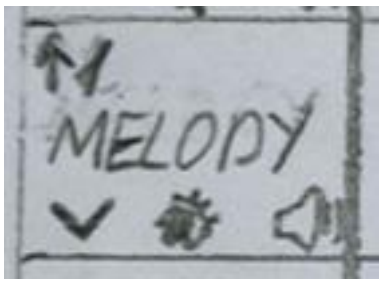


Рисунок 2.10 – Елементи заголовку треку

Показник ланцюга батьківських партій відображається як стрілка, спрямована вгору, разом із числом, розташованим під заголовком партії. Батьківська партія може скластися ланцюг наслідування. У випадку, коли партія наслідується від батьківської партії, яка не наслідується від жодної іншої, біля показника стоїть число «1».

Другою лінією відображається ім’я партії.

На третьому рядку ви знайдете рядок кнопок управління, які розташовані зліва направо наступні.

«Collapse» / «Expand» – ці кнопки дозволяють згортати або розгортати список дочірніх партій. Це корисно для економії місця на екрані: ви можете приховати список дочірніх партій, коли він не потрібен. Якщо відсутні дочірні партії, кнопка автоматично вимикається. Якщо дочірні партії розгорнуті, стрілка вказує вгору, а якщо згорнуті – вниз.

«Settings» – ця кнопка відкриває вікно налаштувань партії. Тут можна змінити її назву, батьківську партію та редагувати межі сегментів, що буде розглянуто пізніше.

«Mute» / «Unmute» – ці кнопки вмикають або вимикають звук під час відтворення партії.

Область редагування треків

На робочій області розташовані сегменти, які утворюють композицію. На рисунку 2.11 показано приклад розміщення цих сегментів.

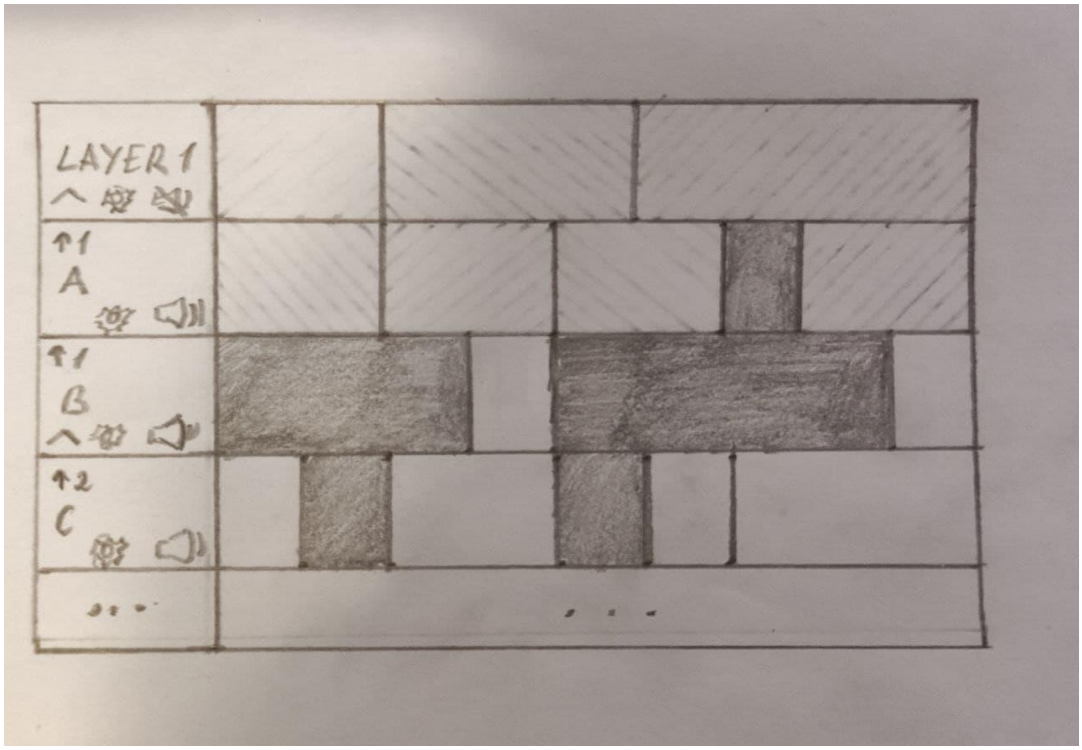


Рисунок 2.11 – Демонстрація розташування сегментів у вікні редагування.

Важливо зауважити, що це той самий основний екран, але представлений у більш простому вигляді. У реальності тут також повинні бути присутні вертикальні лінії, курсор і лінія часу. Згідно з наведеним на рисунку, виділяється чотири партії.

Layer 1 – це батьківська партія. Оскільки вона не має батьківської партії, індикація наслідування відсутня.

У цій партії є два нащадки: «A» та «B». Партія «B», з свого боку, наслідується від «C».

Сегмент – це частина треку з унікальним набором умов. На малюнку сегменти представлені як незафарбовані або у вигляді ліній.

Чорні області повністю відображають відсутність сегментів. Там, де сегментів немає, у партії панує тиша -жодних нот і жодної генерації.

При створенні нової партії відсутні сегменти. Для створення нового сегмента необхідно перейти до налаштувань партії та відкрити розділ «редагування сегментів». Важливо відзначити, що на малюнках відсутній

згаданий розділ. Під час відкриття редагування сегментів на головному екрані залишаються лише заголовки треків, лінія часу та область редагування. При цьому зникають: курсор, кнопка виклику меню та панель управління. Замість цього з'являються кнопки, необхідні під час процесу редагування:

– кнопка «Back» – скасовує всі внесені зміни та повертає головний екран до його стандартного вигляду;

– кнопки «Undo» / «Redo»;

– кнопка «Apply» – застосовує внесені зміни та переводить головний екран у стандартний вигляд.

Для того щоб додати новий сегмент, просто торкніться порожньої області та перетягніть у будь-якому напрямку: сегмент автоматично збільшиться.

Для того, щоб змінити часові межі сегмента, просто перетягніть його початок або кінець. Якщо кінець одного сегмента збігається з початком іншого, просто перетягніть між ними: їхні часові рамки автоматично зсунуться відповідно. Для роздільної зміни часових рамок, двічі натисніть межу між сегментами перед перетягуванням. Під час цього дій виникає порожній проміжок між сегментами.

Для видалення сегмента просто торкніться його: з'явиться кнопка «видалити», яку можна натиснути для видалення сегмента.

Усі зміни можна відмінити за допомогою кнопок «Undo» / «Redo». Також можна змінювати масштаб, використовуючи timeline.

Накладання умов.

Застосування умов починається лише у випадку, якщо головний екран знаходиться у звичайному режимі, без додаткового включення режиму редагування меж сегментів. Натискаючи бажаний сегмент, можна внести зміни щодо умов сегменту треку.

Після цього панель управління зникає, а знизу виїздить діалогове вікно зі списком вузлів генерації (рисунок 2.12).

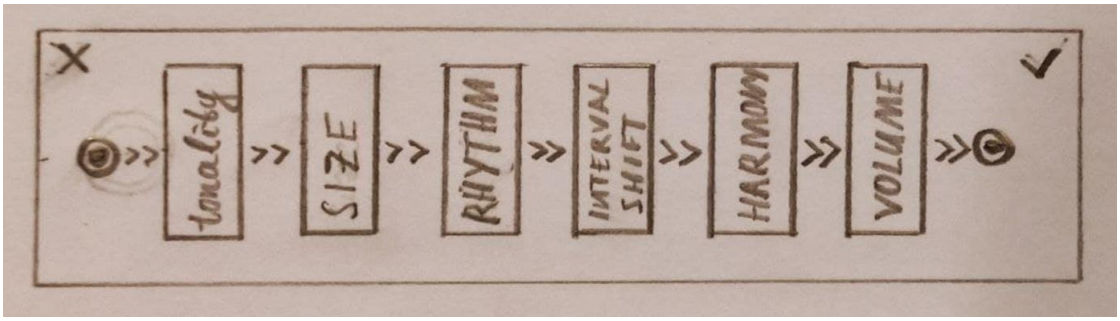


Рисунок 2.12 – Діалог з вузлами генерації.

Розглянемо більш детально вузли генерації. У загальному розумінні, вузол генерації можна уявити як окрему одиницю для визначених умов. На малюнку перелічені всі типи умов, які ми плануємо реалізувати у першій версії програми: Гармонійна; Тональна; Ритмічна; Метрична; Гучність; Інтервальна.

Кожен з цих типів умов буде представлений відповідним вузлом генерації, який дозволить користувачеві встановлювати параметри та умови для кожного аспекту композиції. Діалог вузлів генерації (рисунок 2.13).

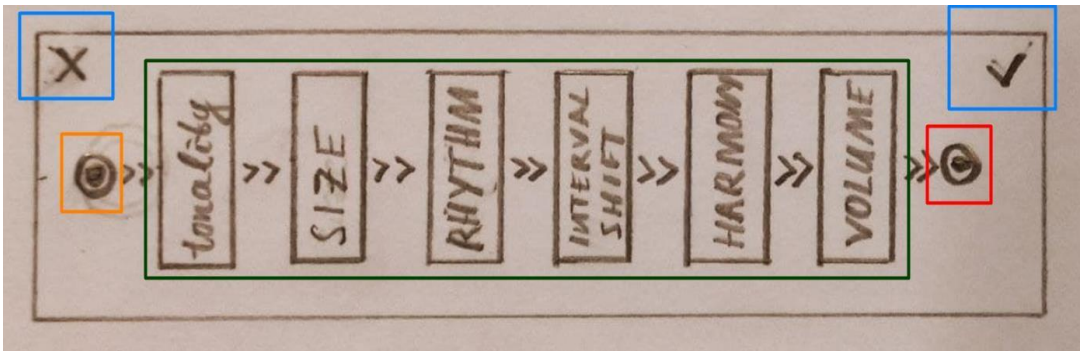


Рисунок 2.13 – Елементи діалогу вузлів генерації

Кнопки управління (виділені синім кольором).

«Скасувати» – скасовує всі зміни, зроблені для сегмента, і закриває відповідний діалогове вікно.

«Застосувати» – зберігає зміни, внесені до сегмента, і закриває

відповідний діалогове вікно.

Вхід у сегмент (позначений помаранчевим кольором) – це точка, в яку подаються результати генерації для батьківського треку. Якщо у треку немає батьківського треку, то вхід у сегмент не відображається в інтерфейсі.

Вихід із сегмента (виділений червоним кольором) – натиснувши на цей вузол, відкривається список дочірніх треків. Вузли накладання умов (позначені зеленим кольором).

Кожен вузол накладання умов може містити лише умову власного формату. Наприклад, вузол ритму не може містити умову вузла інтервалу, і навпаки.

Деякі вузли можуть бути вимкнені, тобто не застосовуватися. Як краще це реалізувати, ми ще не вирішили, але можливо, можна буде додати перемикачі для кожного вузла, якщо потрібно вимкнути їх функцію.

Вузли, які відповідають за тональність та метр, не можна вимкнути, оскільки вони обов'язкові для застосування інших вузлів. Значення цих вузлів за замовчуванням можна змінити в налаштуваннях програми. Стандартні значення цих вузлів – «До-мажор» та «4/4» відповідно.

Для того, щоб змінити часові межі сегмента, просто перетягніть його початок або кінець. У випадку, коли кінець одного сегмента співпадає з початком іншого, просто перетягніть між ними, і їхні часові рамки автоматично зсунуться відповідно (рисунок 2.14).

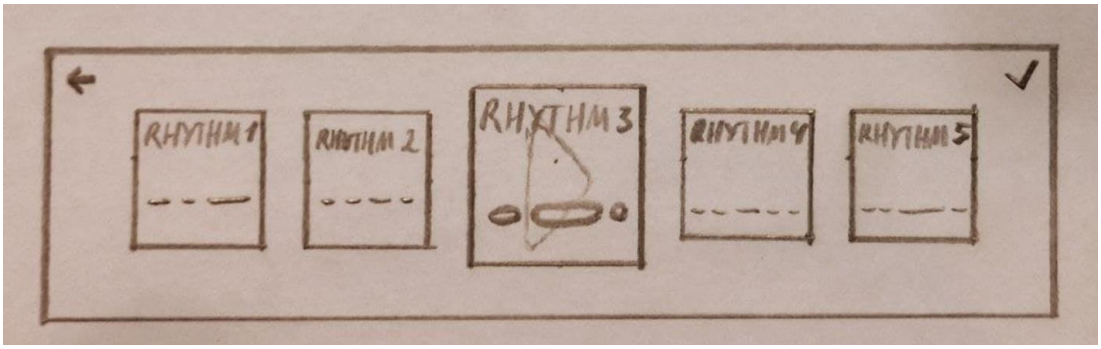


Рисунок 2.14 – Діалог вибору умови

Він подібний до попереднього, але в ньому міститься унікальний набір шаблонів для умови, яка буде зберігатися у вузлі відповідного типу. На цей раз ритмічні умови відображаються на рисунку.

Набори умов можуть бути структуровані у каталоги (хоча на рисунку це не відображено). Каталог може містити підкаталоги або простий список умов. Для вибору умови потрібно центрувати її на екрані. Це можна здійснити, натискавши на умову або прокручуючи список умов. Обрана наразі умова завжди виділяється у центрі та має більший розмір порівняно з іншими. Для підтвердження вибору необхідно натиснути кнопку підтвердження.

Коли умова центрується, на неї накладається значок «Play», який дозволяє відтворити умову.

У майбутньому планується розширення функціоналу для кожної умови, щоб користувач мав можливість редагувати та додавати власні умови. Також передбачено створення вікна перед прослуховування впливу умови.

Після вибору умови користувач повернеться до діалогу з вузлами. Змінені вузли будуть виділені іншим кольором, тоді як вимкнені залишаться без підсвічування.

2.6 Проектування архітектури

У розробці Backend програми можна виокремити два основних напрямки: компоненти, пов'язані з музичною теорією, та компоненти, відповідальні за процес генерації музики.

2.6.1 Класи музичної теорії

Класи-компоненти музичної теорії є основними елементами, з якими взаємодіє програма при формуванні умов, що зберігаються у вузлах

генерації.

Серед компонентів музичної теорії розрізняють: тональність, лад, метр (розмір), нота, акорд та музична лінія.

Тональність. Просто кажучи, у музиці тональність виконує роль орієнтира, що вказує, яким чином можна розташовувати ноти по висоті.

Лад представляє собою закономірність вибору доступних нот з усього музичного діапазону. Наприклад, мажорний лад може мати схему – «т т п т т т п» (т – тон, п – півтон), тоді як мінорний – «т п т т п т т». Крім мажорного та мінорного ладу існує безліч інших, таких як лідійський, міксолідійський, персидський та багато інших. Лади виникали як результат взаємодії музичних традицій, що склалися протягом століть у різних культурах, і стали основою для унікального звучання, що відчувається з певним насиченням.

Тоніка, по суті, є тією нотою, від якої розпочинається лад. Наприклад, мажорний лад, побудований від ноти «До», називається «До-мажор», а мінорний лад, побудований від ноти «Ре», відповідно має назву «Ре-мінор».

Залежно від ладу та тоніки ми вибираємо ноти, які входять у тональність. Ноти, що не відповідають тональності, можуть звучати дисгармонійно, навіть якщо в інших музичних контекстах вони звучать правильно.

Згідно з ладом, звукові висоти організовані таким чином, щоб створювати приємне звучання – ось у чому суть ладу, яке полягає в гармонійному розміщенні звуків.

Метр у музиці головним чином пов'язаний з ритмічною динамікою, яка ґрунтується на відмінностях у відзначенні деяких нот як гучніших, а інших як тихіших.

У програмі метр застосовується переважно для налаштування параметрів вузла «Гучність».

Метр завжди представлений у формі співвідношення двох чисел, наприклад, «4/4». Нижнє число вказує на розмір тактової одиниці, у яких

вимірюється метр, а верхнє число – на їх кількість. Також метр вказує на розмір такту; наприклад, 2/4 означає, що новий такт починається через проміжки часу, еквівалентні двом четвертним нотам. Це важливо, оскільки позначення тактів спрямоване на визначення сильної метричної долі, тому що такт завжди починається з сильної долі. Ще одне важливе аспект ритму – це акцентуація.

Нота, кожна нота має свої характеристики: висоту, час початку звучання, тривалість та гучність.

У нашому проекті кожна нота також зберігає інформацію про тональність, в якій вона була записана, а також про метр. Це спрощує використання музичних шаблонів.

Висота ноти визначається двома параметрами – індексом октави та назвою ноти. Ці величини дозволяють визначити абсолютне значення висоти ноти з урахуванням закономірності акустики, де звучання ноти у вищій октаві в два рази вище звучання ноти в поточній октаві (рисунок 2.15).

У програмі доступні наступні октави: SUBCONTRACT, CONTROVERSIAL, LARGE_OCTAVE, SECOND_OCTAVE, SMALL_OCTAVE, THIRD_OCTAVE, FIRST_OCTAVE, FOURTH_OCTAVE, FIFTH_OCTAVE.

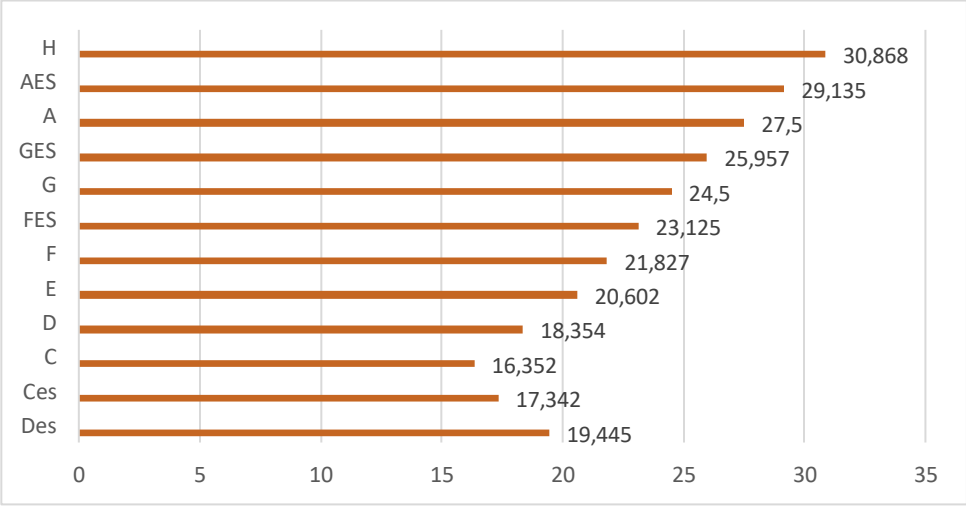


Рисунок 2.15 – Ноти та їх висоти в октавах

Позначення нот зі значенням їх висоти в найнижчій октаві.

Висота ноти визначається кількістю коливань фізичного тіла на секунду. Коли тіло коливається, ці коливання передаються повітрям, і органи слуху сприймають їх як висоту звуку.

Наприклад, для ноти «Ля» (відомої як «А» в європейській нотації) спостерігається приблизно 27,5 коливань на секунду. Кожна нота є окремою одиницею музичного матеріалу, яка має свою власну характеристику висоти.

Акорд – це набір нот, які звучать одночасно. У нашій програмі акорди представлені базовою нотою – нотою, на основі якої за гармонійним малюнком будується акорд – і набором нот, які втілюють саму гармонію.

Партія включає в себе набір нот і акордів, які розташовані в часі і відображають послідовність музичних подій.

2.6.2 Класи умов

Компоненти умови використовуються в алгоритмах генерації для перетворення одного музичного матеріалу, що складається з компонентів музичної теорії (нот, акордів, частин), в інший. Умови, як і вузли генерації, реалізуються наступними типами:

- позначення нот латинськими буквами;
- ладові;
- ритмічні;
- метричні;
- гармонійні;
- інтервальні;
- гучності.

При генерації, вузол проводить процес аналізу (парсингу), під час якого шукає та витягує необхідну інформацію з тексту, а потім накладає умову на передану йому партію в якості аргументу.

Умова позначення нот зберігається в самій програмі і не може бути розширеною «+2|+2|+1|+2|-2|+2|-1». Кожне число відображає зсув у півтонах. Ритмічний малюнок: «32|64+0:6|8:6|16:6, графічний запис на рисунок 2.16.

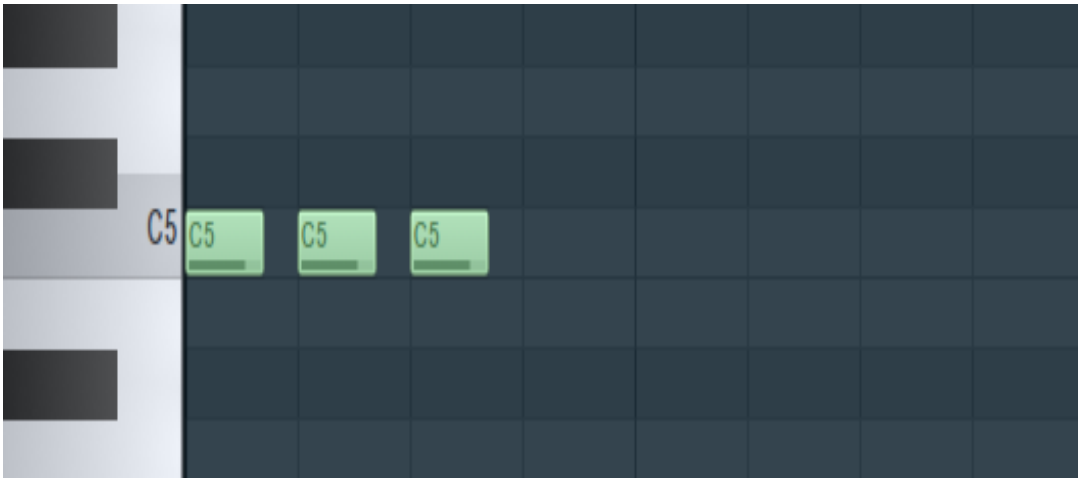


Рисунок 2.16 – Графічне відображення ритмічної умови «32|64+0:6|8:6|16:6»

Послідовність параметрів нот знаходиться справа від знаку «+», що виглядає наступним чином: «0:8|8:8|16:32». Кожен блок опису окремої ноти розділяється символом «|», тобто: «0:8», «8:8», «16:32». У кожному блоку зліва від «:», записана стартова позиція (номер імпульсу в PPQN), коли нота починає звучати, а справа – тривалість її звучання.

Зліва від символу «|», розташований PPQN розширення, у якому записаний ритмічний малюнок – «32». Зліва від знаку «+», розташований опис розміру партії: «32|64». Ці умови необхідні для забезпечення сумісності в разі незбіжності PPQN, в якому записаний ритмічний малюнок з PPQN, прийнятим в програмі.

Правильний запис тривалості ритмічного малюнку знаходиться справа від знаку «|», наприклад: «64». Важливо враховувати тривалість, оскільки ритмічний малюнок не вказує пауз між нотами, тому неочевидно,

коли після закінчення останньої ноти має завершитися малюнок. Таким чином, запис «64» показує, що ритмічний малюнок складається з двох четвертних нот.

Помилкове вказання більшої кількості бітів, наприклад, стартової позиції більше 64 або тривалості, що перевищує 64, при застосуванні умови призведе до ігнорування таких нот.

Для гармонійних та ладово-інтервальних шаблонів формат зберігання умов однаковий, але одна й та ж інформація використовується в них по-різному.

Приклад шифрування умов для ладово-інтервальних та гармонійних шаблонів: «0:0|0:4|1:0|1:4». Окремі ноти розділяються знаком «|». Зліва від знаку «:» знаходиться зміщення в октавах відносно октави наявної ноти, а справа – зміщення по ступенях. В обох випадках значення можуть бути як додатними, так і від'ємними. Повний ритмічний шаблон для розглянутої умови має вигляд: «some_rhythm_1*32|64+0:6|8:6|16:6».

2.6.1 Класи генерації

Класи генерації використовують основні елементи музичної теорії та стандартні шаблони для створення музичного матеріалу. Ці класи формують партії, які пізніше перетворюються у формат MIDI. До класів генерації входять такі складові: композиція, трек, сегмент та вузол.

Композиція відображає найвищий рівень генерації у музичному процесі. Вона складається з ієрархії треків – спеціальної структури, яка визначає, який трек успадковується від якого. При цьому композиція зберігає інформацію про те, які конкретні сегменти треків слід перегенерувати. Використовуючи цю ієрархію, вона встановлює послідовність для цього процесу.

Трек – це сукупність сегментів та партії, яка є результатом останньої генерації.

Сегмент слугує як контейнер для набору умов. Кожен сегмент містить упорядкований набір вузлів генерації. Вони використовуються лише під час генерації, оскільки процес створення партії просувається послідовно через всі сегменти.

Коли ви вибираєте умову в інтерфейсі користувача, вона додається до вузла. Під час процесу генерації, коли вузол отримує пакет для редагування, ця умова декодується (кожен вузол знає, як проаналізувати збережену умову) і застосовується до існуючого пакету, створюючи на його основі новий.

Процес генерації розпочинається з послідовного створення всіх партій, що починаються з незалежних і продовжуються вздовж ієрархії. Спочатку генеруються всі незалежні партії, після чого робиться перехід до залежних на першому рівні і так далі. Це дозволяє забезпечити, що необхідна для генерації найнижчих рівнів інформація вже була отримана на попередніх етапах генерації.

2.7 Висновки до розділу 2

Протокол MIDI (Musical Instrument Digital Interface) є основним стандартом для зберігання та передачі музичних даних на комп’ютері, який був вперше представлений у 1982 році. Всі дані записуються до MIDI-файлу у вигляді двійкової послідовності. Для обробки та редагування MIDI-файлів існують спеціальні програми, відомі як MIDI-редактори. Більшість сучасних віртуальних музичних студій оснащені функціональними можливостями MIDI-редакторів.

Вхідними даними програми є набори завчасно підготовлених шаблонів. Кожен шаблон представляє собою концептуальну умову, таку як ритмічний малюнок, інтервальна закономірність, гармонійна послідовність і інші. Ці шаблони зберігаються в базі даних програми і можуть бути в будь-який момент відредаговані або замінені іншими.

Вихідними результатами програми є міді файли партій, які формуються в процесі виконання алгоритмів програми. Під час розробки ми встановили кілька ключових вимог. Програма має забезпечувати роботу з концептуальними зв'язками, їх збереження та можливість користувача керувати ними під час створення композиції. Програма повинна реалізовувати наслідування між партіями. Алгоритми роботи програми мають бути ефективними та уникати зайвих обчислень. Архітектура програми повинна бути прозорою та легко розширюваною.

3 ОПИС КЛЮЧОВИХ АЛГОРИТМІВ

3.1 Основні положення стосовно розробки алгоритмів

Найважливішою складовою нашої роботи є її алгоритми, оскільки вони розв'язують основні завдання, поставлені перед нами.

Під час розробки алгоритмів ми вдосконалювалися за принципами, які висвітлені у серії книг «Досконалий алгоритм». Основною ідеєю цих алгоритмів є принцип «розділяй і володарюй». Емпірично доведено, що цей підхід найефективніший, коли задачу можна розбити на менші частини, які можна розв'язувати рекурсивно. Такий підхід дозволяє значно скоротити кількість операцій, необхідних для вирішення задачі.

Багато алгоритмів, побудованих на основі цієї парадигми, стали відомими. Наприклад, Merge Sort широко використовується у стандартних бібліотеках багатьох Java.

Дотримуючись парадигми «розділяй і володарюй», задачу розбивають на менші частини, які потім розкладаються на ще менші, аж поки вони не стають простими. Потім ці прості частини розв'язуються рекурсивно, спочатку з найменших, і поступово вирішуються більш складні, збираючи результати кожної частини та використовуючи їх для подальшої роботи, доки вся задача не буде розв'язана.

Ми намагалися реалізувати наші алгоритми, опираючись на концепції, згадані в серії книг «Досконалий алгоритм» [26], а також на принципи SOLID-архітектури та чистого коду. Згідно з цими концепціями, алгоритми були розділені на окремі блоки та розміщені за функціональним призначенням. Таким чином, кожен алгоритм впровадження умов виступає окремою незалежною одиницею, що працює самостійно та не залежить від інших алгоритмів. Це дозволяє модифікувати та тестувати кожен алгоритм окремо, не торкаючись іншого коду.

Незважаючи на те, що будь-яка програма побудована на алгоритмах,

ми найбільше зосереджуємося на алгоритмах генерації, оскільки вони безпосередньо вирішують завдання, поставлене в дипломі. Таким чином, далі розглянемо основні алгоритми, які використовуються в процесі генерації. Почнемо з алгоритму глобальної генерації, а потім перейдемо до розгляду алгоритмів, які використовує кожен вузол (як це працює, стане зрозуміліше пізніше). Ми також розглянемо, як умови накладаються на сторону під час процесу генерації.

Однак ми не будемо розглядати алгоритми, не пов'язані з генерацією. Таким чином, пропустимо алгоритми, пов’язані з відображенням графічного інтерфейсу, взаємодією з базою даних, збереженням файлів, конвертацією в MIDI, відтворенням MIDI-інформації та ін.

3.2 Глобальний алгоритм генерації

3.2.1 Реалізація ієрархії наслідування треків

Загальний алгоритм генерації відповідає за послідовну обробку доріжок композиції, усіх сегментів у кожній доріжці та всіх вузлів у кожному сегменті. Це означає, що доріжки генеруються в порядку їх успадкування: спочатку обробляються батьківські доріжки, потім дочірні доріжки і так далі. Основна мета – застосувати вибрані користувачем умови в правильній послідовності.

Ми крок за кроком розглянемо цей алгоритм і пояснимо, як працює кожен з його компонентів і з чого він складається. Важливо відзначити, що алгоритм генерації існує у двох форматах: для першої генерації та для повторної генерації після внесення змін. Ці алгоритми мають свої відмінності, і на початку ми розглянемо лише алгоритм генерації під час першого запуску. Пізніше ми розширимо його новими перевірками та умовами, щоб зробити його ефективнішим під час повторних запусків.

При початковому запуску генерації ми звертаємося до класу

Composition. Цей клас є глобальним у проекті та існує у єдиному екземплярі. У ньому містяться дані про треки, їх ієрархія успадкування, а також допоміжні функції, такі як generate (яка запускає генерацію), та інші.

Отже, цей клас виступає оболонкою, що містить всі головні дані проекту та функції для їх обробки.

Ієрархія наслідування треків зберігається в окремій структурі під назвою Hierarchy. Ми вирішили винести дані про батьківські та дочірні треки з об'єктів класу Track, щоб дотримуватися принципу єдиної відповідальності та уникнути складнощів у підтримці та розширенні архітектури. Таким чином, інформація про наслідування треків була розміщена у відокремленому класі – Hierarchy.

Hierarchy – це деревоподібна структура даних, де кожен вузол може мати будь-яку кількість дочірніх вузлів або не мати їх зовсім. Також кожен вузол може або не мати батьківського вузла. Наслідування треків можна уявити у вигляді наступної схеми (рисунок 3.1).

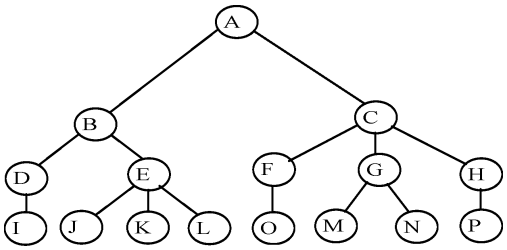


Рисунок 3.1 – Ієрархія наслідування треків

У структурі Hierarchy, що знаходиться в класі Composition, зберігається ієрархія треків. Вона дозволяє навігуватися між різними треками. Наприклад, уявіть трек «A» як найвищого рівня батьківський трек, з якого походять дочірні треки «B» та «C». Дочірні треки «B» та «C» можуть мати свої дочірні треки, які представлені, наприклад, треками «D» та «E». Ця ієрархія дозволяє організувати структуру треків і здійснювати навігацію між ними. Розглянемо код реалізації (рисунок 3.2).

```
class HierarchyNode(
    val boundTrack: Track,
    var parent: HierarchyNode? = null,
    var children: MutableList<HierarchyNode> = mutableListOf()
) {
    fun addChild(node: HierarchyNode) {
        children.add(node)
    }

    fun removeChild(node: HierarchyNode) {
        children.remove(node)
    }
}
```

Рисунок 3.2 – Код реалізації

Клас HierarchyNode є базовим елементом дерева ієрархії. Він містить посилання на конкретний трек, що прив'язаний до цього вузла (boundTrack). Також у класі зберігається посилання на батьківський вузол (parent), яке може мати значення null, і список дочірніх вузлів (children), який за замовчуванням є порожнім (рисунок 3.3).

Рисунок 3.3 – HierarchyNode

Клас Hierarchy має два основних поля. Перше – це список верхніх вузлів ієрархії, представлений як topNodes. Друге поле – це мапа map, яка забезпечує швидкий доступ до вузла ієрархії за його унікальним

ідентифікатором (ID) треку. Це здійснено для оптимізації швидкості доступу до треків за їхніми ідентифікаторами, оскільки мапа забезпечує швидкий доступ за ключем ID у порівнянні з ручним пошуком у ієрархії.

Розглянемо призначення та реалізацію функцій.

Функція findParentTrackOf Ця функція призначена для пошуку батьківського треку для вхідного треку, який передається як аргумент. Якщо вузол для вхідного треку знайдено в мапі, функція отримує батьківський вузол цього вхідного треку і повертає прив'язаний до нього трек. Якщо батьківський вузол відсутній (значення null), функція повертає null, що означає відсутність батьківського треку для вхідного треку (рисунок 3.4).

```
fun findParentTrackOf(track: Track): Track? {
    val node = map[track.id]
    if (node != null) {
        return node.parent?.boundTrack
    }
    return null
}
```

Рисунок 3.4 – Функція батьківського контролю

Функція add, функція додає новий трек до ієрархії, встановлюючи його батьківський трек. Якщо батьківська доріжка нульова, нова доріжка додається до списку незалежних доріжок topNodes (рисунки 3.5–3.7).

```
fun remove(track: Track) {
    val node = map.remove(track.id)
    if (node?.parent != null) {
        node.parent?.removeChild(node)
    }
}
```

Рисунок 3.5 – Функція remove може видаляти трек з ієрархії.

```
fun addTrackToHierarchy(newTrack: Track, parentTrack: Track?) {
    val newNode = HierarchyNode(newTrack)

    if (parentTrack != null) {
        val parentNode = map[parentTrack.id]
        if (parentNode != null) {
            newNode.parent = parentNode
            parentNode.addChild(newNode)
        }
    } else {
        topNodes.add(newNode)
    }

    map[newTrack.id] = newNode
}
```

Рисунок 3.6 – Функція додаванні треків до ієрархії

```
fun changeParent(track: Track, newParent: Track) {
    remove(track)
    add(newParent, track)
}
```

Рисунок 3.7 – Функція changeParent, може змінювати parent-трек для треку.

Функція getSortedInDeepTracks викликає метод collectNodesInDeep для кожного вузла верхнього рівня (незалежного треку) та утворює список відсортованих вузлів «в глибину» (рисунок 3.8).

```
fun getSortedInDeepTracks(): List<Track> {
    val tracks = mutableListOf<Track>()
    for (node in topNodes) {
        collectNodesInDeep(node, tracks)
    }
    return tracks
}
```

Рисунок 3.8 – Функція getSortedInDeepTracks

Функція collectNodesInDeep рекурсивно додає вузли до списку list, просуваючись «вглиб» по вузлах ієрархії (рисунок 3.9).

```
fun collectNodesInDeep(root: HierarchyNode, list: MutableList<Track>) {
    list.add(root.boundTrack)
    for (node in root.children) {
        collectNodesInDeep(node, list)
    }
}
```

Рисунок 3.9 – CollectNodesInDeep

Щоб пояснити термін «відсортованих в глибину», давайте розглянемо процес генерації. Перш ніж генерувати дочірні треки, ми маємо спочатку згенерувати батьківські треки. Таким чином, якщо дочірній трек буде згенерований перед батьківським, то його результат може бути неточним, оскільки він ґрунтується на застарілих даних.

Правильна послідовність генерації треків може бути складена різними способами. «Відсортовані в глибину» означає, що треки зберігаються у списку в такому порядку, що спочатку додається глобальний трек «A», оскільки він знаходиться на вершині ієрархії.

Розглянемо порядок додавання треків у список «в глибину», як це представлено на зображенні.

Спочатку додається глобальний трек «A».

Потім додається перший дочірній трек «B».

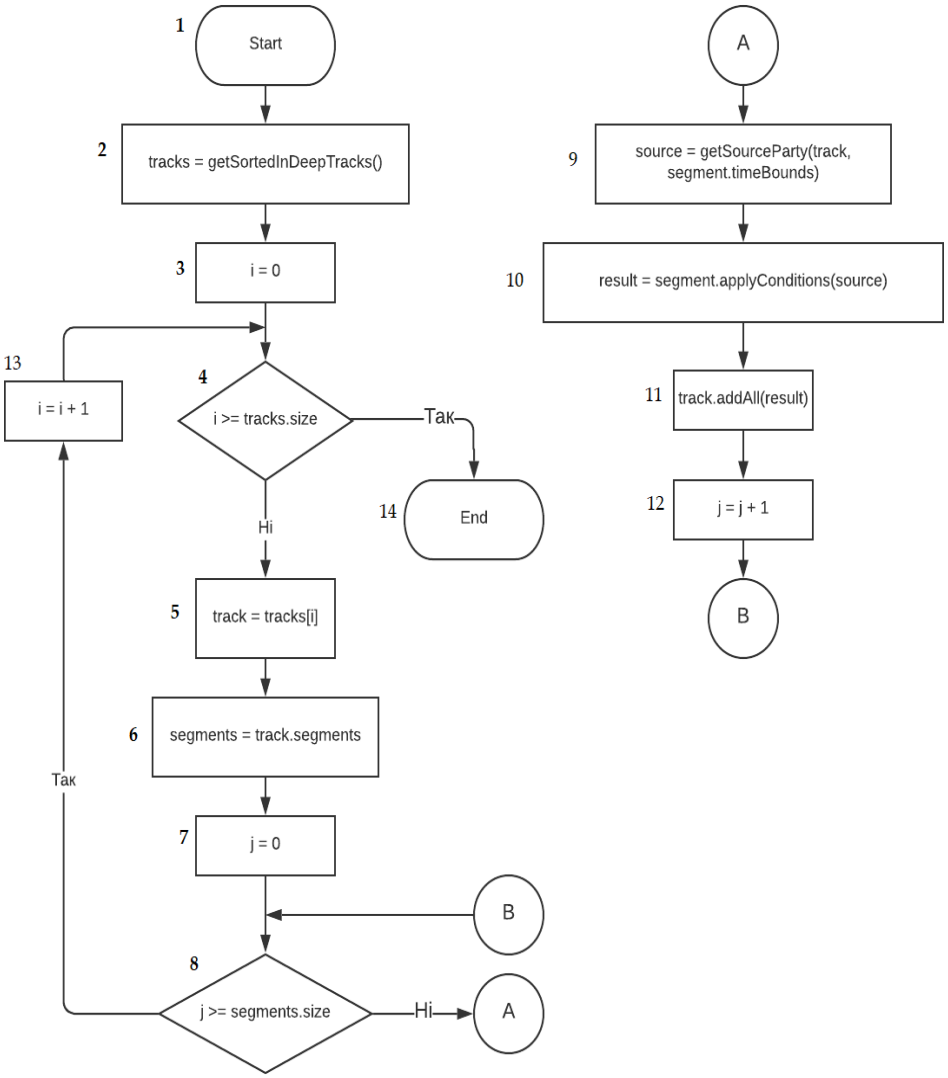
Після цього перевіряється, чи є в треку «B» дочірні треки. Якщо так, вони додаються, якщо ні – додається наступний трек-наслідник «A».

Цей процес продовжується, просуваючись в глибину за вузлами і зберігаючи порядок генерації.

Отже, результатуючий список має такий вигляд: A, B, D, I, E, J, K, L, C, F, O, G, M, N, H, P.

3.2.2 Генерація на рівні композиції

Щоб запустити процес генерації, ми викликаємо функцію generate класу Composition (рисунок 3.10).




```
fun generate() {
  val tracks: List<Track> = hierarchy.getSortedInDeepTracks()
  for (track in tracks) {
    for (segment in track.segments) {
      val source: Party = getSourceParty(track, segment.timeBounds)
      val result: Party = segment.applyConditions(source)
      track.addAll(result)
    }
  }
}
```

Рисунок 3.10 – Блок схема алгоритму генерації на рівні Composition

У фрагменті коду ми створюємо змінну tracks, у якій зберігаємо відсортований за глибиною список треків. Потім ми проходимо всі треки. Для кожного треку ми проходимо всі його складові сегменти у внутрішньому циклі.

Для кожного треку ми отримуємо «базову» частину, викликаючи функцію getSourceParty(...). У цій функції ми передаємо трек поточної доріжки та часові межі сегмента segment.timeBounds. TimeBounds – це простий клас-огортка з двома цілочисельними полями start і end.

У наступному фрагменті коду (рисунок 3.11) ми викликаємо функцію applyConditions(source) для об’єкта сегмента, передаючи пакет, який ми отримали на попередньому кроці. Результат виклику функції зберігається в змінній результату.

Після цього ми викликаємо track.addAll(result). Функція addAll додає всі ноти з партії, переданої як аргумент, до внутрішньої партії треку.

У коді також є звернення до функції getSourceParty(...), яка надає партію для подальшого накладання умов.

```
fun getSourceParty(track: Track, bounds: TimeBounds): Party {
  val parent = hierarchy.findParentTrackOf(track)
  if (parent != null) {
    return TrackUtils.getPartyFragmentOf(track, bounds)
  } else {
    return TrackUtils.generateSingleNoteParty(bounds)
  }
}
```

Рисунок 3.11 – Функція applyConditions(source)

У цьому фрагменті коду ми спочатку намагаємося отримати батьківський трек, звертаючись до функції findParentTrackOf(...). Ми вже досліджували цю функцію раніше і знаємо, як вона працює.

Потім ми перевіряємо, чи не є parent нульовим. Якщо він не є

нульовим, ми звертаємося до допоміжного класу TrackUtils, який не містить жодних даних, але містить набір допоміжних функцій, необхідних для процесу генерації.

Функція TrackUtils.getPartyFragmentOf(track: Track, bounds: TimeBounds) витягує фрагмент партії з треку, який обмежений часовими рамками, визначеними параметром bounds. Якщо нота партії знаходиться на межі цих рамок, вона обрізається таким чином, щоб перебувати повністю всередині меж. Важливо зазначити, що ця функція створює нові ноти замість повернення списку нот із партії треку. Це означає, що при редагуванні цих нот батьківська партія залишається без змін.

У випадку, якщо значення parent дорівнює null, ми викликаємо метод TrackUtils.generateSingleNoteParty(bounds), який генерує партію з однією нотою. В цій ноті зберігається інформація лише про початок звучання та тривалість, а інші характеристики (наприклад, висота, гучність і т. д.) відсутні. Інформація про часові обмеження ноти береться з параметра bounds.

На концептуальному рівні програми це означає, що для треків найвищого рівня (які не мають батьківських треків) за замовчуванням генерується «одинарна нотна партія».

Ми вирішили зберігати умови для генерації безпосередньо в параметрах кожної окремої ноти. Це означає, що нам вже необхідно мати щонайменше одну ноту на момент застосування першої умови.

Тривалість ноти встановлюється за замовчуванням з метою забезпечення єдиної форми ритмічного шаблону для подальшого застосування, зокрема для фільтрації.

В попередній версії бекенду програми існували два режими ритмічного шаблону: генеративний і фільтруючий. Перший створював ритм з нуля, тоді як другий адаптував вхідні ноти відповідно до заданих ритмічних умов.

Але існування двох режимів для одного вузла призводило до

плутанини, особливо враховуючи те, що алгоритм практично не відрізнявся, що викликало повторення в коді. За новим підходом, на початку генерації, нота вже має часові рамки, тому режим породження ритму втрачає свою актуальність.

3.2.3 Генерація на рівні сегмента

Проводячи ітерацію по сегментам треку, ми викликаємо у кожного з них функцію applyConditions(party: Party).

Треки (Track) складаються з Segment а сегменти, у свою чергу, з вузлів Node та генерація виклику функції applyConditions. У результаті роботи функція повертає партію, яку згенерував останній вузол (рисунок 3.12).

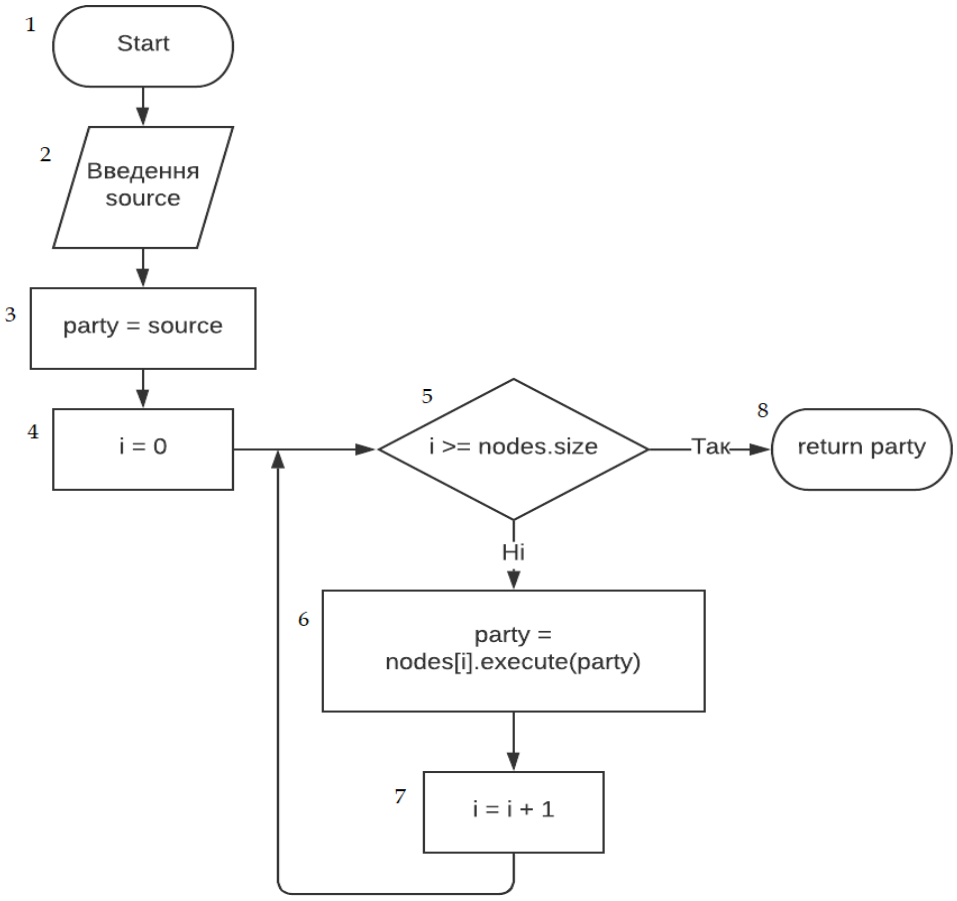


Рисунок 3.12 – Блок-схема генерації на рівні сегменту

Важливо відзначити, що кожен трек може мати власний набір умов, і вони не обов'язково однакові для всіх треків. Наприклад, сегменти незалежних треків мають два додаткових вузли генерації: вузол тональності та вузол метру. Ці вузли застосовуються раніше за всі інші і мають значення за замовчуванням.

При встановленні трека як незалежного, ці додаткові вузли автоматично додаються до списку вузлів, і видаляються, якщо трек стає дочірнім. Таким чином, на етапі застосування вузлів фільтрувати їх окремо не потрібно.

3.2.4 Генерація на рівні вузла

На рисунку 3.13 наведено блок-схему генерації на рівні вузла

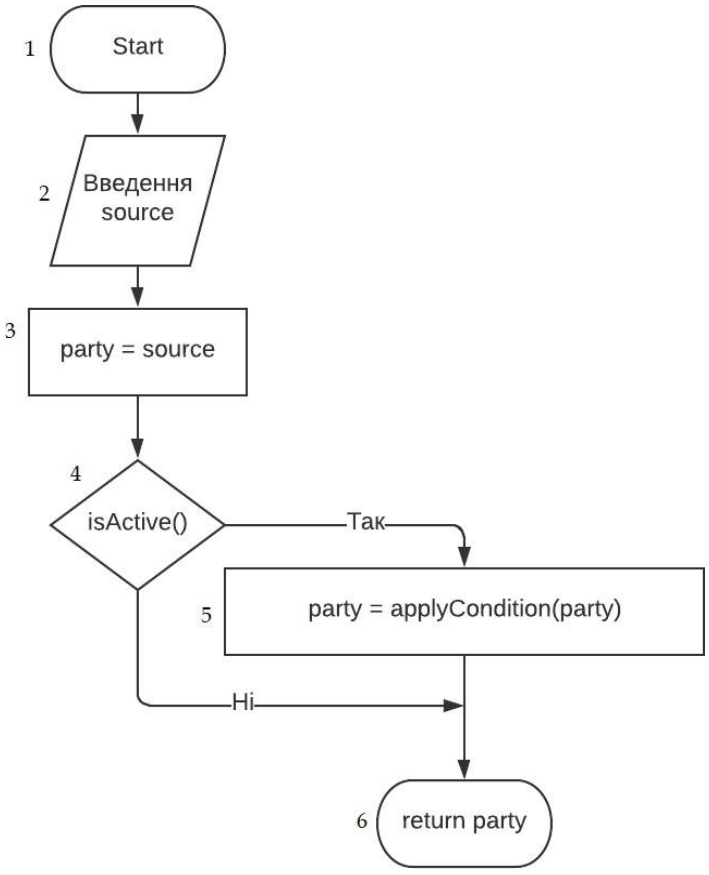


Рисунок 3.13 – Блок-схема генерації на рівні вузла

На рівні вузла генерація починається з виклику функції execute (рисунок 3.14).

```
fun execute(source: Party): Party {
   if (isActive()) {
      return applyCondition(source)
   }
   return source
}
```

Рисунок 3.14 – На рівні вузла, генерація починається з виклику функції виконання

Спочатку перевіряється активність вузла, яка залежить від двох факторів: чи ввімкнено вузол в інтерфейсі користувача (UI) і чи має він певну умову. Якщо вузол не має умови, він автоматично стає неактивним як в інтерфейсі користувача, так і в процесі генерації. У цьому випадку вузол не взаємодіє з пакетом, переданим йому як аргумент, а лише повертає його в початковому стані.

Якщо ж вузол має умову і включений у UI, тоді викликається його метод applyCondition, який має різну реалізацію для кожного типу вузла.

Важливо підкреслити, що під час генерації ми завжди створюємо нову партію, базуючись на існуючій. Це пояснює, чому кожен метод накладання умов повертає значення.

Такий підхід ґрунтується на одній із ключових концепцій функціонального програмування – незмінності даних. Багато проблем у роботі програм виникають через недбале модифікування даних з різних джерел. Тому, де це можливо, дані слід робити незмінними. Кожен вузол не змінює частину, яку він отримує як аргумент, а створює нову на її основі, копіюючи нотатки з джерела та надаючи цим копіям нові характеристики.

3.2.5 Загальний огляд алгоритму генерації

Підсумовуючи наші знання, в проекті важлива центральна точка входу – екземпляр класу Composition, що містить ієрархію треків та запускає генерацію. Ієрархія зберігається у структурі класу Hierarchy. Hierarchy включаючи в себе вузли ієрархії – HierarchyNode, з одним батьківським вузлом і кількома нащадками. Генерація розпочинається з виклику методу generate об'єкту Composition і має наступний алгоритм:

– проходимо всі треки в порядку ієрархії;

– для кожного трека послідовно переходимо до кожного сегменту;

– якщо для треку є батьківський трек, ми отримуємо з нього відрізок партії для редагування за допомогою методу TrackUtils.getTrackPartyFragment(parentTrack, bounds). У випадку, якщо трек є незалежним, для сегмента генерується партія «пустої» ноти за допомогою іншого методу того ж класу TrackUtils: generateSingleNoteParty(bounds);

– метод getTrackPartyFragment() виконує вирізання фрагмента існуючої партії треку в межах заданих bounds;

– метод generateSingleNoteParty() створює ноту з тривалістю bounds, яка не має висоти, гучності та інших параметрів, але містить інформацію про початкову та кінцеву позиції звучання ноти.

Результат обробки сегменту включає в себе партію, яка створена останнім вузлом генерації. Перед застосуванням умов, вузол перевіряється на активність. Якщо він активний, до сегменту застосовується відповідна умова. У випадку неактивного вузла, повертається партія, що була передана як параметр. Кожен дочірній клас перевантажує метод applyCondition, щоб працювати з конкретними типами умов. У цих класах створюється новий екземпляр Party, який заповнюється об'єктами MusicItem. Результат застосування сегмента зберігається в поточному треку за допомогою методу track.addAll(party).

3.3 Алгоритми накладання умов

3.3.1 Шаблони умов

Давайте розіб'ємо це детальніше. У проекті наразі існують п'ять типів умов:

- тональна;
- метрична;
- ритмічна;
- інтервально-висотна закономірність;
- гармонійна.

Всі ці умови зберігаються в базі даних у зашифрованому вигляді. Ми наводили приклад шифрування в попередній частині. Наприклад, для ритмічної послідовності шаблон в базі даних може мати наступний вигляд: some_rhythm_1*32|64+0:6|8:6|16:6.

Це звичайний рядок, який містить зашифрований шаблон ритмічного малюнка згідно певних правил. Давайте розглянемо, як він структурований. «some_rhythm_1» – це ім'я шаблону, яке також виступає унікальним ідентифікатором у базі даних. Давайте докладніше проаналізуємо це. «32|64» – перше число вказує на роздільну здатність, у якій зазначена умова (в PPQN). Друге число вказує на тривалість фрагменту, також у PPQN. «0:6|8:6|16:6» – це ритмічна умова, де окремі ритмічні одиниці розділені символом «|». Ліворуч від символу «:» – початкова позиція ритмічної одиниці (в PPQN), праворуч – її тривалість (в PPQN).

Термін «ритмічна одиниця» описує просту ноту, в якій відсутня будь-яка інформація, крім часових характеристик. У проекті кожна така одиниця перетворюється в об'єкт класу TimeBounds.

Коли користувач переглядає каталог умов певного типу (як це відбувається при виборі умов на інтерфейсі користувача, що було описано в попередньому розділі), ці умови витягуються з бази даних у

зашифрованому вигляді. Потім відбувається процес розшифрування, в результаті якого дані зберігаються у відповідному класі, щоб у подальшому швидко з ними взаємодіяти.

Наприклад, ритмічний стан після розшифровки зберігається в класі RhythmCondition, який має наступний вигляд, наведений на рисунку 3.15.

```
class RhythmCondition(
  val sequence: List<TimeBounds>,
  val size: Int
)
```

Рисунок 3.15 – RhythmCondition

RhythmCondition містить список екземплярів класу TimeBounds (ритмічних одиниць) та інформацію про тривалість ритмічного малюнка – розмір. Використання об’єкта цього класу в коді зручніше, ніж робота із зашифрованими даними. Умови інших типів також зберігаються в подібних класах-огортках.

3.3.2 Архітектура базового класу Node

Проаналізуємо архітектуру базового класу Node (рисунок 3.16).

Розпочнемо з визначення імені класу. Як видно з заголовку, клас Node є параметризованим, що вказується передачею параметра у трикутних дужках справа від імені класу (<Condition>) зазначено, що для кожного класу-нащадка Node змінна condition матиме власний тип.

Клас Node містить два поля: isEnabled – логічна змінна, яка показує, чи ввімкнений вузол на інтерфейсі користувача, condition – розшифрована умова у вигляді класу-обгортки. Далі ми розглянемо методи цього класу.

```
abstract class Node<Condition>(
  var condition: Condition? = null,
  var isEnabled: Boolean = false,
) {

  fun execute(source: Party): Party {
    if (isActive()) {
      return applyCondition(source)
    }
    return source
  }

  abstract fun applyCondition(source: Party): Party

  fun isActive() = isEnabled && condition != null

  fun requireCondition() = condition!!
}
```

Рисунок 3.16 – Архітектура базового класу Node

Метод execute викликається сегментом під час генерації, як ми раніше розглядали.

applyCondition – ця функція не має реалізації в базовому класі, але присутня у всіх його нащадках. Вона відповідає за створення нової партії на основі вхідної з накладеними умовами.

isActive – цей метод демонструє, чи є вузол активним. Вузол вважається активним, якщо він увімкнений у користувацькому інтерфейсі (isEnabled == true) та містить умову (condition != null).

requireCondition – цей метод повертає ненульове значення condition. Якщо condition == null, генерується виключення (NullPointerException).

3.3.3 Тональність

Вузол тональності зустрічається тільки в незалежних треках, і він автоматично налаштований на тонність C-maj за замовчуванням (рисунок 3.17).

```
class TonalityNode: Node<TonalityCondition>() {
   override fun applyCondition(source: Party): Party {...}
}
```

```
class TonalityCondition(
   val tonality: Tonality
)
```

Рисунок 3.17 – Вузол тональності

В проекті представлений класом TonalityNode (рисунок 3.18).

```
class TonalityNode: Node<TonalityCondition>() {

  override fun applyCondition(source: Party): Party {
    return Party().apply {
      val condition = requireCondition().tonality
      for (index in 0 until source.items.size) {
        val item = source.items[index].copy()
        item.setTonality(condition)
        item.setHeight(Height(
          FIRST_OCTAVE,
          condition.tonic
        ))
        add(item)
      }
    }
  }
}
```

Рисунок 3.18 – TonalityNode

На рисунку 3.19 наведено блок-схему накладання тонального шаблону.

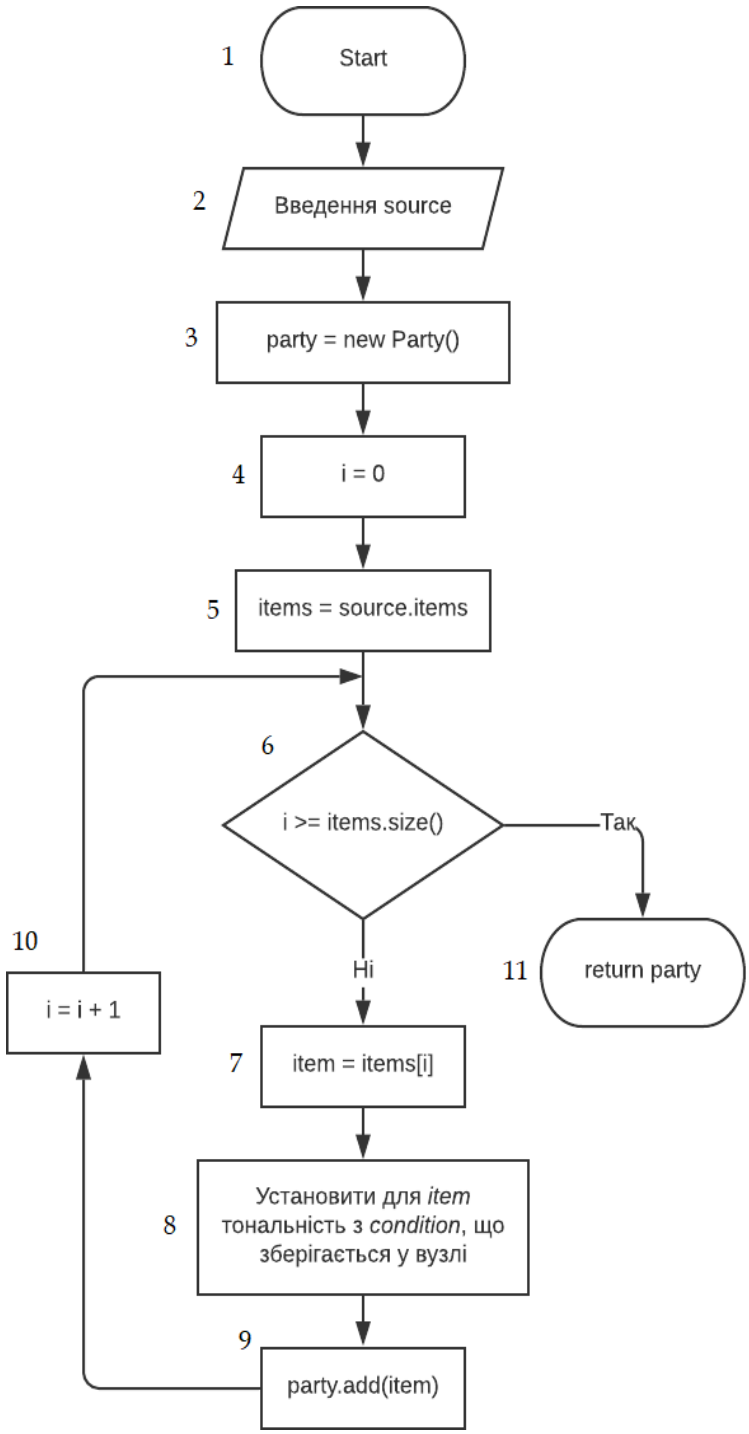


Рисунок 3.19 – Блок-схема накладання тонального шаблону

Отримуємо умову з класу-обгортки.

Послідовно проглядаємо всі елементи партії (ноти або акорди). Кожному елементу встановлюємо тональність, яку дістаємо з умови.

Висота встановлюється для елементів автоматично та дорівнює тоніці тональності на першій октаві. Додаємо ноту до партії. Коротке пояснення

роботи return Party().apply { … }: Функція apply, яка викликається одразу після конструктора класу, є нововведенням у мові Kotlin. Вона дозволяє використовувати анонімні об'єкти одразу при їх створенні. Це означає, що ми можемо провести всі необхідні маніпуляції з об'єктом без потреби зберігати його значення або посилатися на нього явно, звертаючись до його методів безпосередньо після створення, як якщо ми знаходимося всередині нього.

3.3.4 Метр

На рисунку 3.20 наведено код класу.

```
class MeterNode: Node<MeterCondition>() {

  override fun applyCondition(source: Party): Party {
    return Party().apply {
      val condition = requireCondition().meter
      for (index in 0 until source.items.size) {
        val item = source.items[index].copy()
        item.setMeter(condition)
        add(item)
      }
    }
  }
}

class MeterCondition(
  val meter: Meter
)
```

Рисунок 3.20 – Код класу

На рисунку 3.21 наведено блок-схему накладання метричного шаблону.

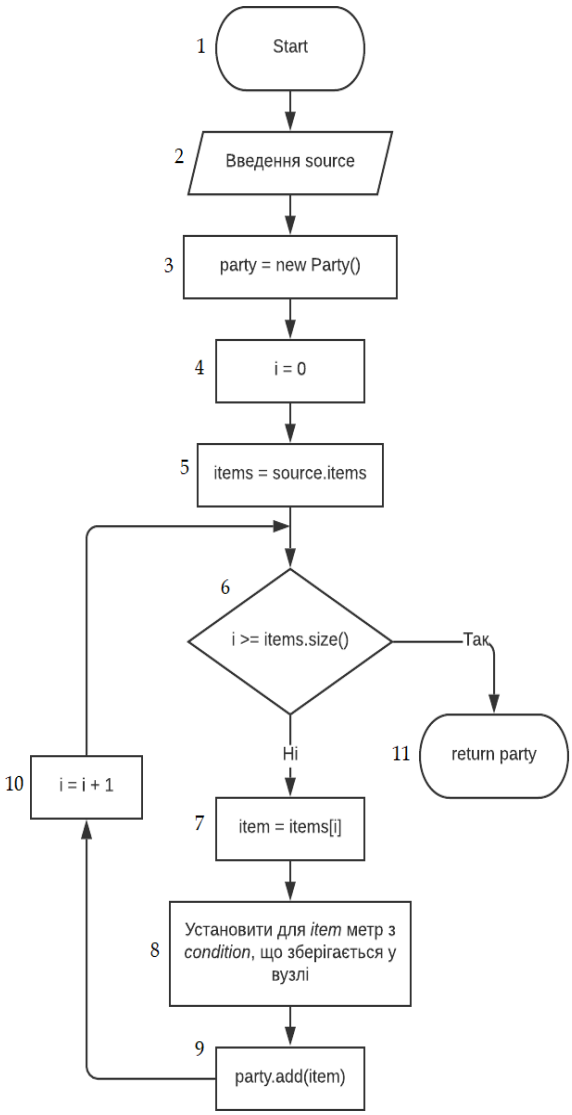


Рисунок 3.21 – Блок-схема накладання метричного шаблону

Ми отримуємо інформацію про метр з умови та послідовно переглядаємо всі елементи вхідної партії. Для кожного елемента встановлюємо метр і додаємо його до нової партії. Важливо відзначити, що під час перебору елементів вхідної партії ми створюємо копії цих елементів у всіх вузлах, а не модифікуємо наявні.

3.3.5 Інтервальна закономірність

Для інтервального та гармонійного вузла використовується однакова обгортка даних – IntervalCondition.

```
class IntervalCondition(
val sequence: List<HeightShift> )
```

Це базовий клас, що містить список зсувів за висотою, який називається «HeightShift».

```
class HeightShift( val steps: Int, val octave: Int )
```

Інформація про висоту ноти в класі Note представлена через обгортку– Height.

```
class Height(
var octave: Octave = FIRST_OCTAVE,
var letter: Letter = C )
```

Height просто упаковує дві інші важливі характеристики: обрану октаву та назву ноти. Отже, вона точно визначає положення ноти по висоті (октава та назва ноти в межах цієї октави). Звичайно, HeightShift нічого не знає про конкретну висоту, але у двох своїх полях відображає зсув, який потрібно застосувати до ноти: від однієї октави до іншої (октава) та на певну кількість ступенів ладу (steps). Давайте розглянемо, як це реалізовано в методі applyCondition класу IntervalNode. (рисунок 3.22).

```
override fun applyCondition(source: Party): Party {
  return Party().apply {
    var sIndex = 0 // index of source item
    var cIndex = 0 // index of condition item
    val shifts = requireCondition().sequence

    while (sIndex < source.items.size) {
      val item = source.items[sIndex].copy()
      if (cIndex >= shifts.size) {
        // condition sequence end reached
        // reset condition
        cIndex = 0
      }
      item.shiftHeight(shifts[cIndex])
      add(item)
      sIndex++
    }
  }
}

}
```

Рисунок 3.22 – Метод applyCondition

На рисунку 3.23 наведено блок-схему накладання інтервального шаблону.

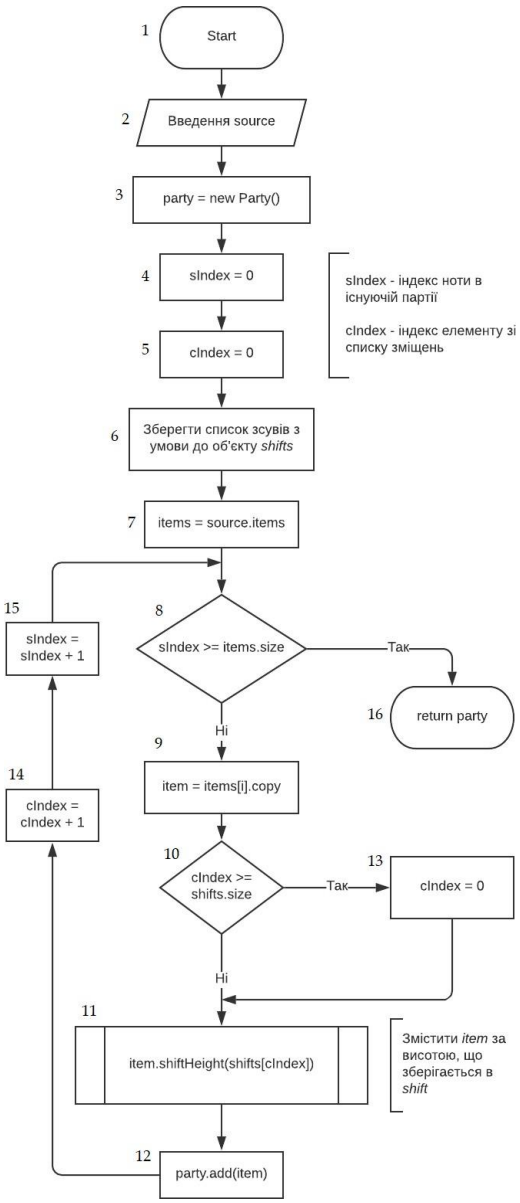


Рисунок 3.23 – Блок-схема накладання інтервального шаблону

Механізм роботи алгоритму, ми отримуємо вхідні дані у вигляді партії, яка може складатися з нот або акордів, а також умову зі списком зміщень (IntervalCondition з переліком HeightShift).

Наша мета полягає в тому, щоб послідовно переглянути всі елементи вхідної партії source та змістити кожен елемент (ноту або акорд) на відповідну висоту, яка визначена в списку умов (sequence у IntervalCondition). Ми переходимо від одного елементу до наступного як у списку sequence, так і у списку елементів партії source.

Процес триває, доки ми не дійдемо до кінця елементів source. У разі досягнення кінця sequence, ми починаємо знову з його початку та продовжуємо з першої умови.

Висотне зміщення ноти відбувається шляхом переміщення по ступенях ладу, який визначається тональністю. Основний механізм такого зміщення просто пересуває ноту вгору або вниз (в залежності від того, чи є зсув додатнім чи від'ємним) лише серед нот, дозволених у даній тональності, оминаючи інші.

Наприклад, якщо ми маємо тональність Ля-мінор (Am), то відповідними нотами будуть A, B, C, D, E, F, G (Ля, Сі, До, Ре, Мі, Фа, Соль) – всі білі клавіші фортепіано.

Таким чином, якщо в нас є нота D2 (Ре другої октави) та висотне зміщення з величинами octave = -1, interval = 3, ми змінюємо ноту на одну октаву вниз та піднімаємо її на 3 ступені вгору: E, F, G. Таким чином отримуємо ноту G1 (Соль першої октави).

У програмі висотне зміщення реалізується за допомогою методу shiftHeight (рисунок 3.24), який належить до допоміжного класу MusitUtils, куди винесені всі алгоритми, пов'язані з музичною теорією.

```
object MusicUtils {

  //...

  fun shiftHeight(
    height: Height,
    tonality: Tonality,
    shift: HeightShift
  ) {
    val letters = tonality.letters
    var index = letters.indexOf(height.letter)
    var octave = height.octave.ordinal
    val size = letters.size

    octave += shift.steps / size + shift.octave
    index += shift.steps % size

    // check is letter in bounds
    if (index >= size) {
      octave++
      index -= size
    } else if (index < 0) {
      octave--
      index += size
    }

    // check is octave in bounds
    if (octave >= Octave.values().size) {
      octave = Octave.values().size - 1
    } else if (octave < 0) {
      octave = 0
    }

    height.octave = Octave.values()[octave]
    height.letter = letters[index]
  }

  //...
```

Рисунок 3.24 – Метод shiftHeight

На рисунку 3.25 наведено схему алгоритму висотного зміщення.

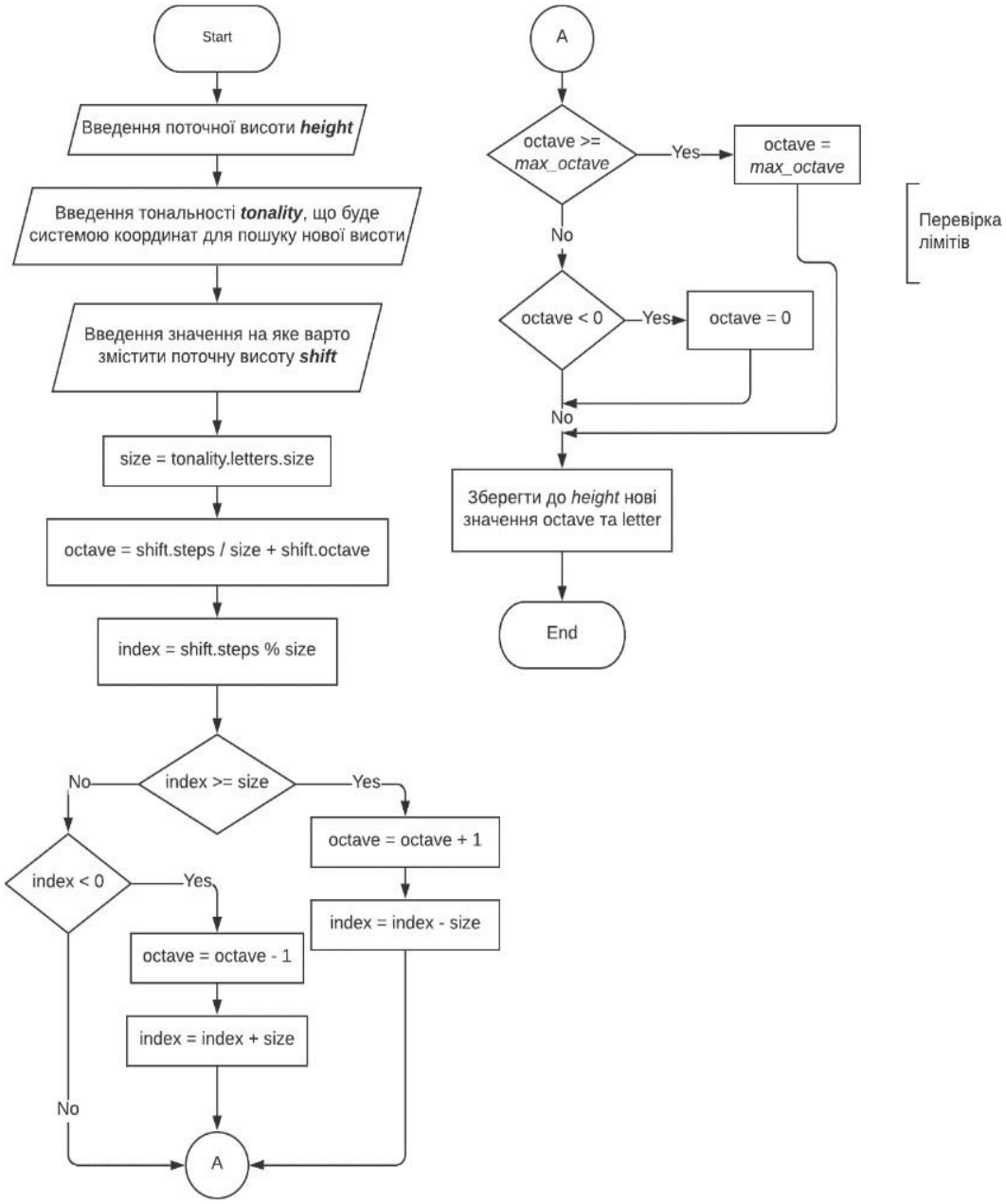


Рисунок 3.25 – Схема алгоритму висотного зміщення приймає на вхід три параметри.

Height – об'єкт, що містить висоту, яку потрібно змістити.

Tonality – тональність, за ступенями якої відбуватиметься зміщення.

Shift – об'єкт, який містить дані про зміщення.

Алгоритм реалізується наступним чином.

Отримуємо список нот, які входять до вказаної тональності, та зберігаємо їх у змінній letters.

Визначаємо порядковий номер буквеного позначення ноти серед тих, що дозволені в заданій тональності, зберігаємо його у змінній index.

Запам'ятовуємо порядковий номер октави (octave) та кількість нот, які входять до тональності (size).

Далі ми обчислюємо нове значення індексу октави. Для цього до поточного значення octave додаємо кількість октав із умови, а також ділимо зсув по ступенях на кількість ступенів в одній октаві. Це допомагає визначити, чи є додаткове зміщення за октавами, яке ми також додаємо до значення октави. Далі ми визначаємо новий індекс ноти, додаючи залишок від ділення shift.steps на size, щоб зміщення за ступенями залишалося в межах однієї октави.

Після цього ми перевіряємо, чи потрапляє новий індекс ноти в доступні межі. Якщо індекс ноти менший за допустиме значення, то насправді ми знаходимося на одну октаву нижче від необхідної. Тому ми зменшуємо значення octave на 1, а індекс збільшуємо на кількість нот в одній октаві.

Останнім кроком є перевірка значення октави за аналогічним принципом. Якщо октава перевищує допустимі межі, ми встановлюємо максимально можливе значення. А якщо вона менша за мінімально допустиме, то встановлюємо мінімальне значення. Після цього оновлені дані зберігаються у полях об'єкта height.

3.3.6 Гармонія

Гармонією в музиці називається послідовність акордів, що акомпанують мелодії.

Гармонійний вузол взаємодіє з висотним зміщенням, але відрізняється від інтервального тим, що не змінює саму висоту ноти.

Замість цього він конструює акорд на основі вказаної ноти, копіюючи її та зміщуючи копії на відповідні висоти, які визначені у послідовності IntervalCondition (рисунок 3.26).

```
class HarmonyNode: Node<IntervalCondition>() {

  override fun applyCondition(source: Party): Party {
    return Party().apply {
      val shifts = requireCondition().sequence
      for (item in source.items) {
        val baseNote = item.totalNotes()[0].copy()
        baseNote.setHeight(item.getHeight())
        val chordNotes: MutableList<Note> = mutableListOf()
        for (shift in shifts) {
          val copy = baseNote.copy() as Note
          copy.shiftHeight(shift)
          chordNotes.add(copy)
        }
        add(Chord(
          item.getTonality(),
          item.getMeter(),
          item.getHeight(),
          chordNotes
        ))
      }
    }
  }
}
```

Рисунок 3.26 – IntervalCondition

Обходимо елементи source. Для отримання всіх характеристик ноти, ми використовуємо першу ноту зі списку totalNotes. Якщо це акорд, функція поверне список нот акорду; якщо це одиночна нота, вона поверне саму ноту. Цю першу ноту зберігаємо у константі baseNote.

Потім ми встановлюємо висоту базової ноти елемента. Це важливо, оскільки вона може відрізнятися від висоти першої ноти акорду, якщо

елемент є акордом. Після цього створюємо порожній список для нот акорду, який ми будемо заповнювати пізніше.

Під час ітерації циклу ми пройдемося по списку зміщень. На кожній ітерації створимо копію базової ноти, зміщуючи її на величину shift за допомогою відомого методу з MusicUtils, і потім додамо цю зміщену ноту до списку нот акорду.

Новий акорд додаємо до партії той самий цикл для інших елементів source (рисунок 3.27).

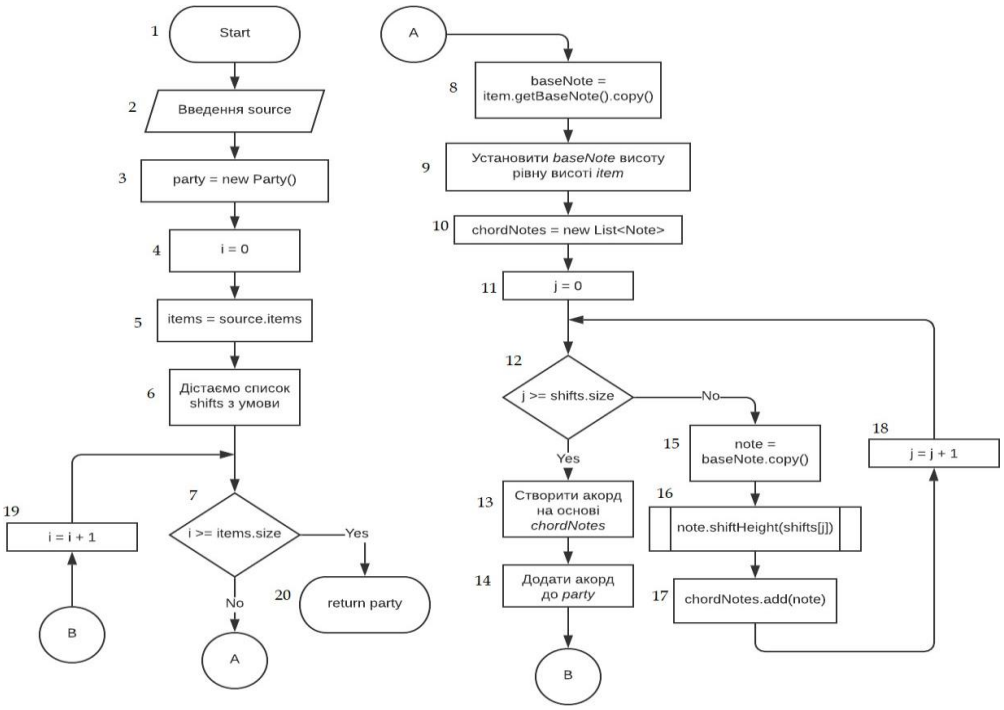


Рисунок 3.27 – Блок-схема накладання гармонійного шаблону

Дістаємо список зміщень з умови.

3.3.7 Ритм

Застосування ритмічного шаблону на прикладі візуального представлення. Для цього скористаємося редактором піано-ролів FL Studio, щоб відобразити ноти. В даному випадку ми не будемо звертати увагу на

висоту нот, а зосередимось лише на ритміці. У нас є два ритмічних рисунки 3.28 та 3.29.

Рисунок 3.28 – Ритмічний малюнок нот в партії

Послідовність «1» представляє ритмічний малюнок нот у партії. Послідовність «2» -це ритмічний патерн, який ми застосуємо до партії.

Фільтруючий ритм визначається, як система логічної кон'юнкції, що означає, що сигнал на виході буде активним лише в тих випадках, коли він одночасно активний на обох входах. Іншими словами, ми отримуємо згенеровану ноту лише у ті періоди, коли ноти з обох ритмічних малюнків співпадають.

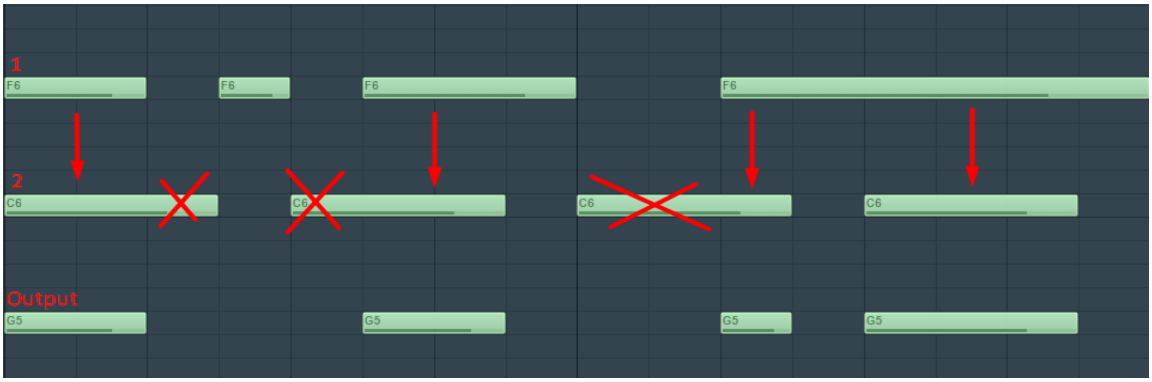


Рисунок 3.29 – Ритмічний рисунок шаблону, який накладається на партію

Ноти з першої послідовності застосовуються до нот з другої послідовності там, де вони звучать одночасно. Ноти, які не співпадають,

просто відкидаються (їх виділили на малюнку). Як результат, ми отримуємо партію з назвою «Output» (рисунок 3.30).

```
class RhythmNode : Node<RhythmCondition>() {
    override fun applyCondition(source: Party): Party {
        return Party().apply {
            var n0 = 0 // index of current note in party
            var n1 = 0 // index of current note in condition
            // sequence
            var i1: TimeBounds // item from party
            var i2: TimeBounds // item from condition
            val condition = requireCondition()
            var shift = 0
            // until party finish reached
            while (n0 < source.size()) {
                if (n1 >= condition.sequence.size) {
                    // condition sequence end reached
                    // reset condition, shift and continue
                    n1 = 0
                    shift += condition.size
                    continue
                }
                val item = source.items[n0].copy()
                i1 = item.getTimeBounds()
                i2 = condition.sequence[n1]
                val i1Start = i1.start
                val i1End = i1.end
                val i2Start = shift + i2.start
                val i2End = shift + i2.end
                // check and update indexes if condition
                // item and party item not intersects
                if (i2End < i1Start) {
                    n1++
                    continue
                } else if (i1End < i2Start) {
                    n0++
                    continue
                }
                val start = max(i1Start, i2Start)
                val end = min(i1End, i2End)
                item.setTimeBounds(TimeBounds(start, end))
                add(item)
                // update indexes
                if (i1End < i2End) {
                    n0++
                } else if (i2End < i1End) {
                    n1++
                } else {
                    n0++
                    n1++
                }
            }
        }
    }
}
```

Рисунок 3.30 – Партія з назвою «Output»

При використанні ритмічного шаблону ми працюємо в припущенні, що на вхід вузла надходить пакет, в якому всі ноти відсортовані за часом

початку звуку. Таким чином ми можемо переглядати їх послідовно, враховуючи, що час збільшується з індексом поточної ноти. Алгоритм роботи наступний (рисунки 3.31, 3.32, 3.33).

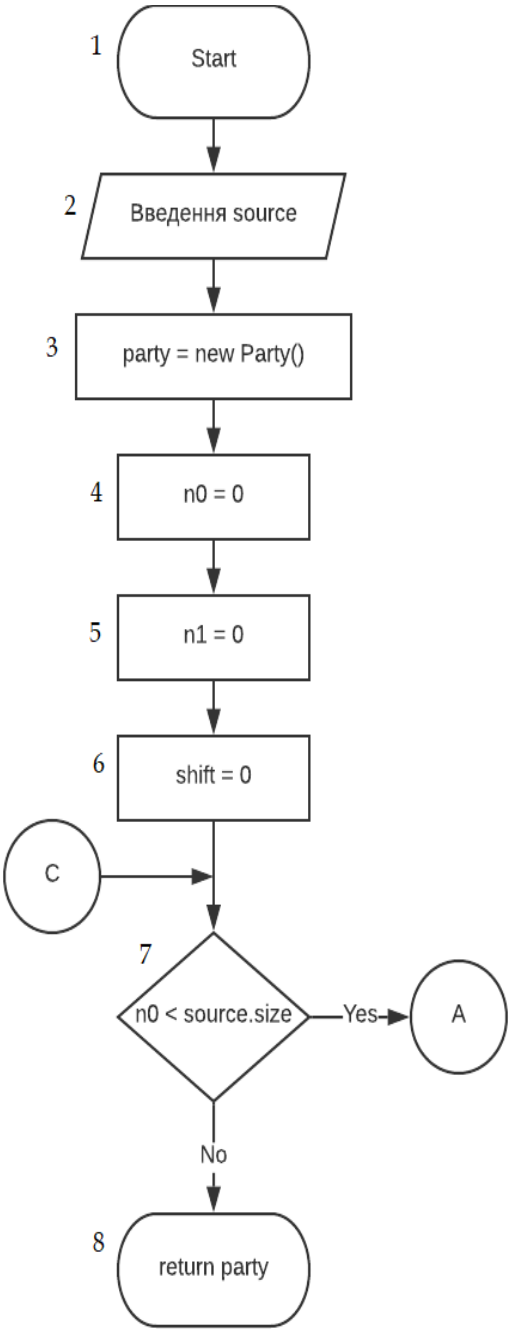


Рисунок 3.31 – Блок-схема накладання ритмічного шаблону, частина 1

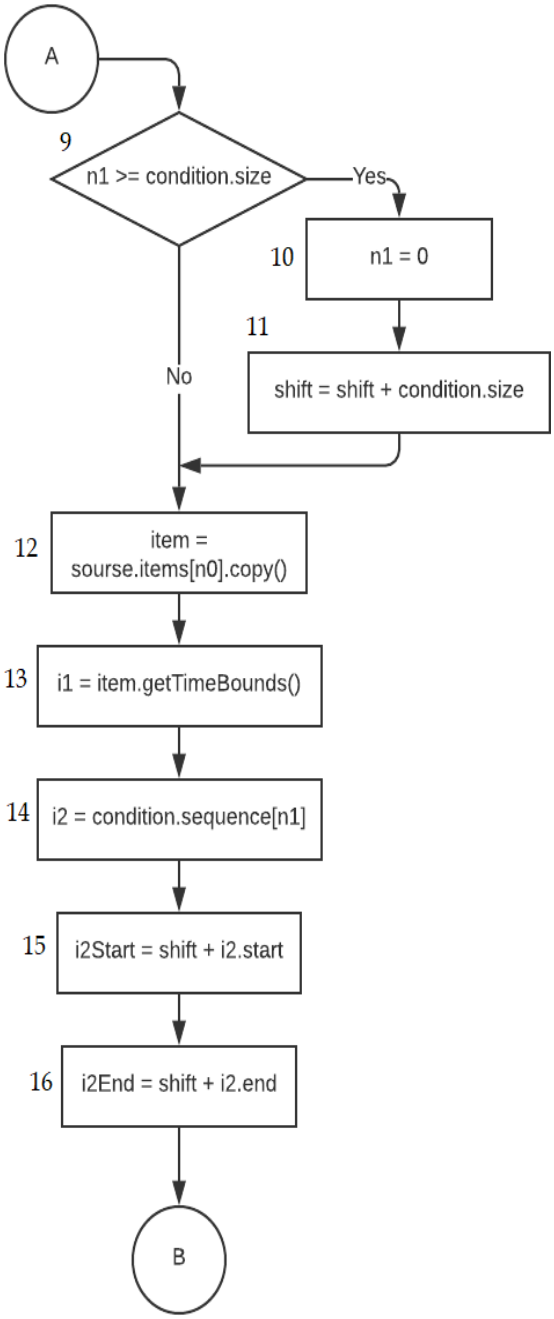


Рисунок 3.32 – Блок-схема накладання ритмічного шаблону, частина 2

Ми взяли до уваги довжину ритмічного шаблону умови (який зберігається в константі розміру об’єкта RhythmCondition), тому ми будемо послідовно проходити кожну з цих довжин. Спочатку задамо змінну shift, яка вкаже початкову часову позицію, з якої будемо розглядати кожен елемент зі списку ритмічних умов.

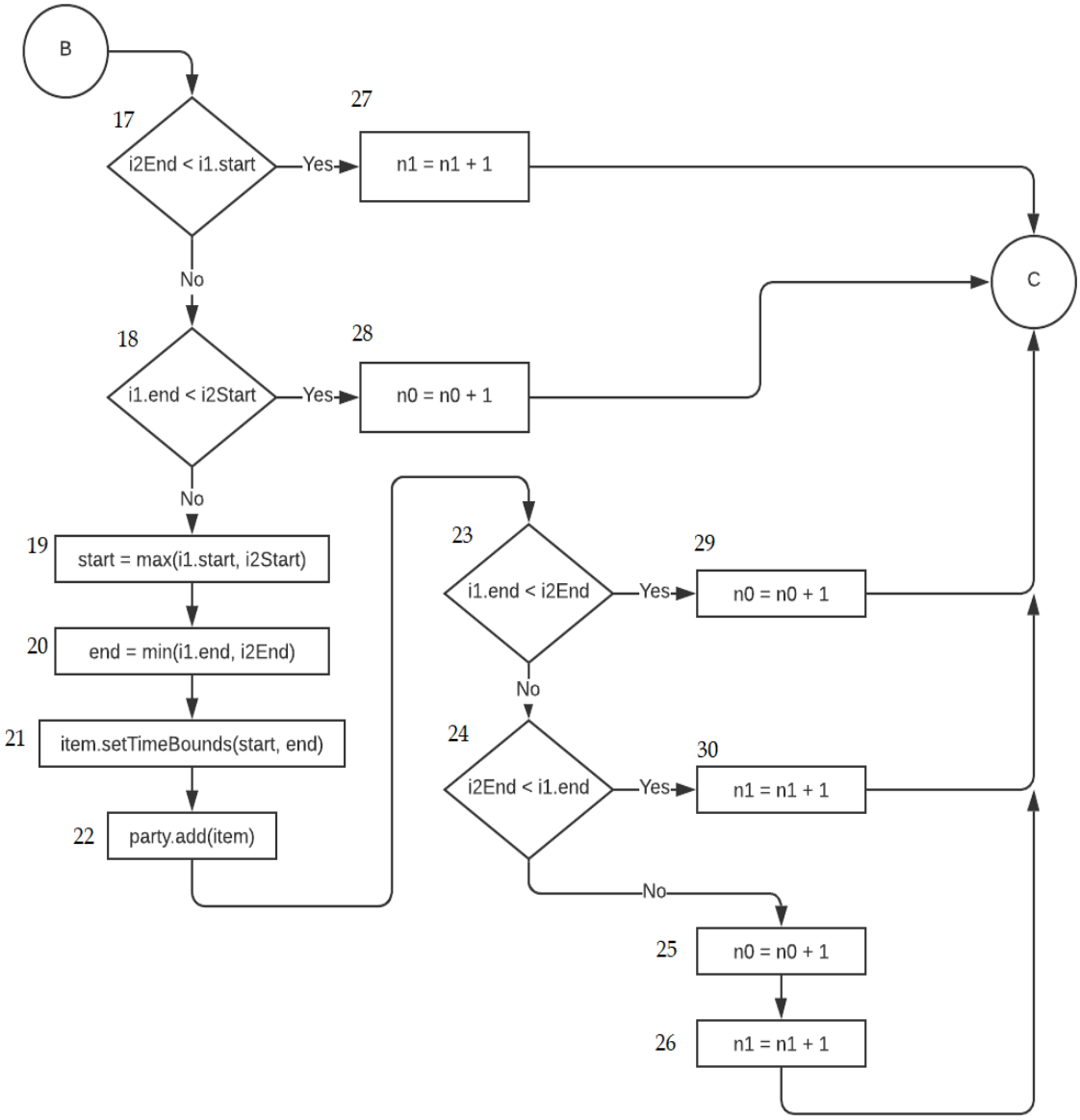


Рисунок 3.33 – Блок-схема накладання ритмічного шаблону, частина 3

Далі перейдемо до циклу, який пройде через кожен елемент джерела, поки не досягнемо кінця партії. На кожній ітерації цього циклу отримаємо поточну ноту і поточну умову. Наступним кроком буде перевірка. Якщо примітка до частини та умова не збігаються, ми переходимо до примітки до наступної частини або до примітки про наступну умову. Ця перевірка проста: якщо кінцева позиція елемента умови менша за початкову позицію примітки в частині, це означає, що примітка умови розташована перед приміткою в частині, і вони не перекриваються (рисунок 3.34).

Рисунок 3.34 – Перевірка взаємодії нот партії та стану

Тож ми маємо перейти до наступної ноти з умовою та продовжити цикл. Крім того, ми також перевіряємо, чи примітка в частині не закінчується раніше, ніж починається умова примітки (рисунок 3.35).

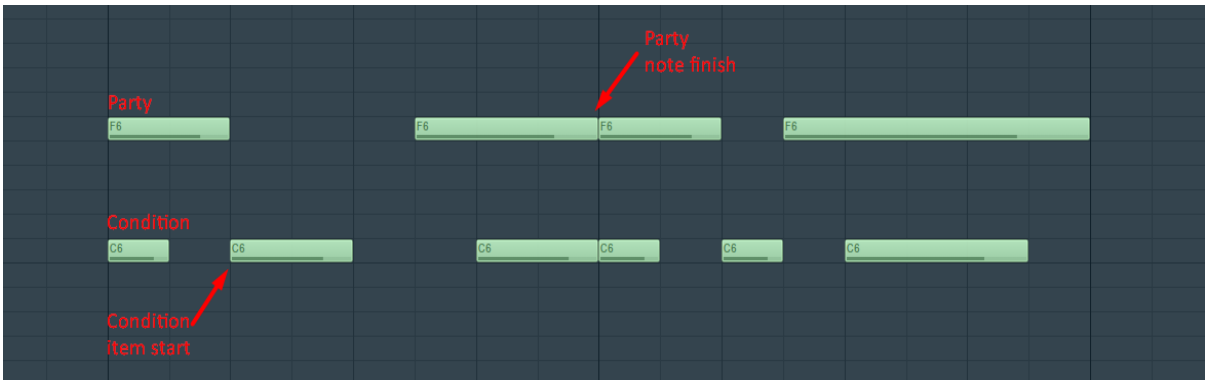


Рисунок 3.35 – Продовження перевірки взаємодії ноти партії та умови

Якщо вони дійсно не перетинаються, ми збільшуємо індекс поточної ноти в партії і продовжуємо цикл. Але якщо ноти перетинаються, початкова позиція нової ноти буде більшою з двох значень початкових позицій цих двох нот, а кінцева позиція буде меншою з цих двох чисел. Це й стане новими часовими параметрами ноти.

Після визначення часових параметрів новоствореної ноти ми переходимо до наступної ноти у послідовності, яка «закриває» кінцеву позицію цієї нової ноти. Щоб визначити, який індекс потрібно збільшити,

ми порівнюємо кінцеві позиції та збільшуємо індекс тієї послідовності, нота якої закінчується раніше. У випадку, якщо позиції кінця їх поточних нот однакові, збільшуємо індекси обох послідовностей.

На кожній ітерації також проводиться перевірка, чи не вийшов поточний індекс ноти за межі умов, за їх максимальну кількість. Якщо це так, потрібно здійснити зсув shift на тривалість умови та скинути індекс умови n1 до початкового значення. Цим ми фактично розбиваємо всю партію на рівні відрізки, кожна з яких має тривалість, яка рівна тривалості ритмічного шаблону, і працюємо з кожною окремо. Це представляє ще один приклад впровадження парадигми «розділяй та володарюй» в нашій програмі.

Загальний алгоритм можна описати так.

Розбиваємо всю партію на рівні відрізки, кожна з яких має тривалість, рівну тривалості ритмічного малюнка з умови.

Перевіряємо, чи поточні ноти умови та партії перетинаються. У випадку відсутності перетину, переміщуємося вперед, по нотам умови або партії, враховуючи, яка з них відстає.

Якщо ноти перетинаються, ми створюємо нову ноту на основі поточної ноти партії з часовими характеристиками від моменту, коли обидві ноти почали звучати, і до моменту, коли хоча б одна з них завершилася.

Залежно від того, яка нота закінчилася раніше, ми переходимо до наступної ноти в партії або в ритмічному малюнку, або в обох, якщо вони закінчилися одночасно.

Після закінчення нот у ритмічному малюнку він починається знову.

Всі інші характеристики нот (висота, тональність, гармонія і т.д.) беруться з існуючої ноти партії, і ми змінюємо лише їх часові характеристики.

3.4 Висновки до розділу 3

В розробці алгоритмів проекту використовується парадигма «розділяй та володарюй», що передбачає розбиття складної задачі на менші частини, які поділяються на ще менші, і так далі, поки не будуть отримані елементарні завдання. Потім ці завдання вирішуються рекурсивно, починаючи з елементарних і поступово переходячи на вищі рівні, доки не буде повністю розв'язана початкова задача. Цей підхід широко використовується в проектуванні складних алгоритмів, оскільки він ефективно зменшує кількість конкретних операцій, що потрібні для вирішення задачі.

З використанням парадигми «розділяй та володарюй», а також принципів чистої архітектури, алгоритми розміщені в окремих блоках та відділені за призначенням. Кожен окремий алгоритм впровадження умов функціонує як незалежна одиниця, яка не має знань про інші алгоритми. Це дозволяє змінювати та тестувати кожен алгоритм незалежно від іншого коду, що сприяє підтримці та розвитку системи.

Основний алгоритм генерації відповідає за послідовну обробку доріжок композиції, усіх сегментів доріжки та всіх вузлів сегментів. Під час генерації ми не змінюємо існуючу партію, а щоразу створюємо нову партію та заповнюємо її відредагованими параметрами. Такий підхід ґрунтується на принципі незмінюваності даних, який є однією з основ функціонального програмування. Модифікація даних з різних місць може призвести до багатьох проблем у роботі програм, особливо при багатопоточному програмуванні. Тому ми намагаємося зробити дані незмінними, де це можливо.

4 ОГЛЯД РЕЗУЛЬТАТІВ ТА ПОДАЛЬШИЙ РОЗВИТОК ПРОГРАМИ

4.1 Приклад застосування програми

Для наочного прикладу використання алгоритмів розглянемо процес генерації на конкретному прикладі. Створимо відрізок композиції на 4 такти, який складається з 4-х треків: Акорди; Базовий трек; Мелодія; Басовий трек.

Для візуалізації результатів, ми будемо користуватися програмою FL Studio. Оскільки візуальний інтерфейс нашої програми наразі не підтримує попередній перегляд MIDI, ми будемо створювати зображення результатів генерації з цієї програми.

4.1.1 Генерація базового треку

Для базового треку встановимо тривалість у 4 такти. Цей трек складатиметься з одного сегменту, також тривалістю 4 такти. Умови цього сегменту такі:

- інтервальна послідовність: «0|-2|-4|+1»;
- розмір: 4/4;
- тональність: Am;
- ритмічний малюнок: тривалість в 1 такт.

Пам'ятаємо, що для самостійного треку (тобто без батьківського треку), перед застосуванням умов, генерується «порожня» нота з часовими рамками, які дорівнюють тривалості сегменту. На наступних ітераціях до цієї ноти будуть застосовуватися тільки тональні умови, а висота ноти буде встановлена рівною тоніці тональності в першій октаві. У нашому випадку це нота «Ля» першої октави, під час візуального накладення, розмір не відображено (рисунок 4.1).

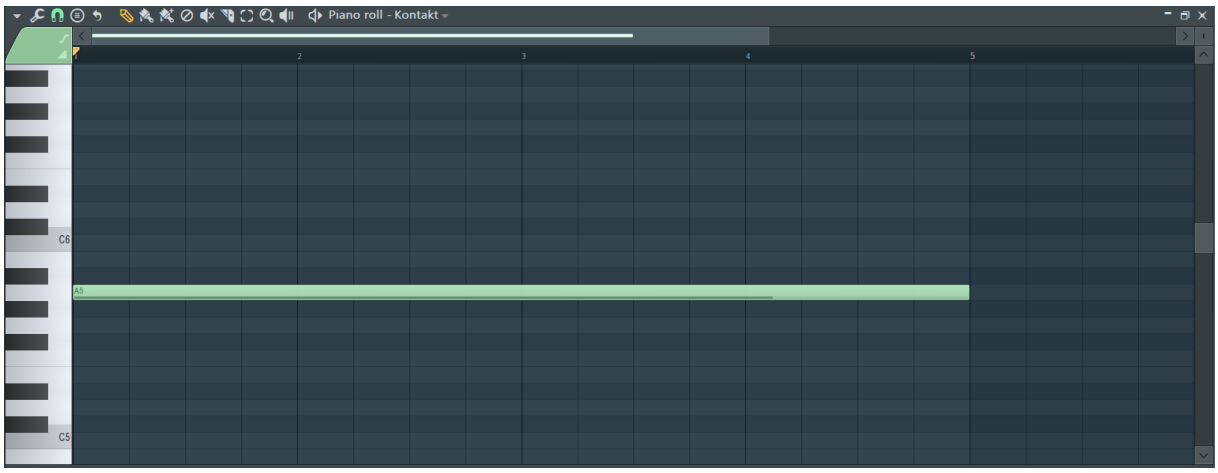


Рисунок 4.1 – Результат накладання тональності та розміру

При накладенні ритмічного малюнка суцільна нота, створена раніше, ділиться на 4 ноти, кожна тривалістю 1 такт (рисунок 4.2).

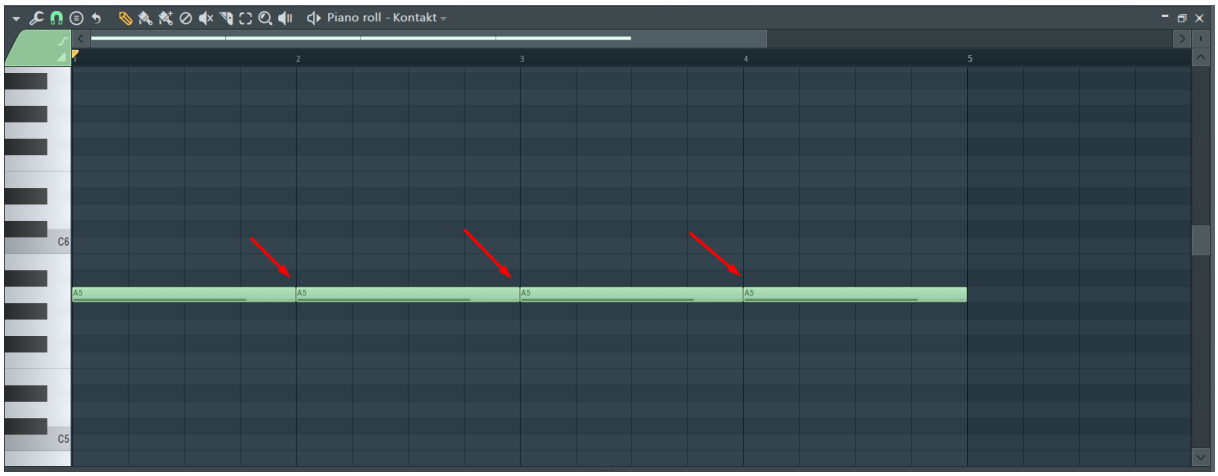


Рисунок 4.2 – Результат накладання ритмічної умови (стрілками показано межі нот)

При застосуванні інтервальної умови перша нота залишається на тоніці (нота «0»), а друга зсувається на два ступені вниз (нота «–2»). Це означає, що він пересуває дві ноти вниз від початкової позиції. У тональності ля мінор доступні всі ноти, розташовані на білих клавішах фортепіано, а ноти, розташовані на чорних клавішах (які представляють

ноти, що не входять до цієї тональності), недоступні (рисунок 4.3).

Таким чином, друга нота зміщується на дві ступені вниз, уникнувши чорних клавіш, які представляють недоступні ноти в даній тональності.

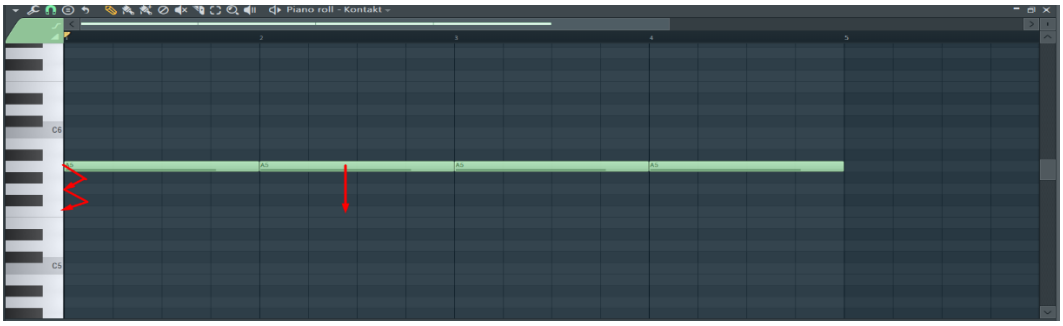


Рисунок 4.3 – Наочне зображення процесу навчання інтервальної умови

Третя нота відпускається на 4 ступені, а остання підвищується на 1 градус. (рисунок 4.4). Фінальний етап накладання інтервальної закономірності наведено на рисунку 4.5.

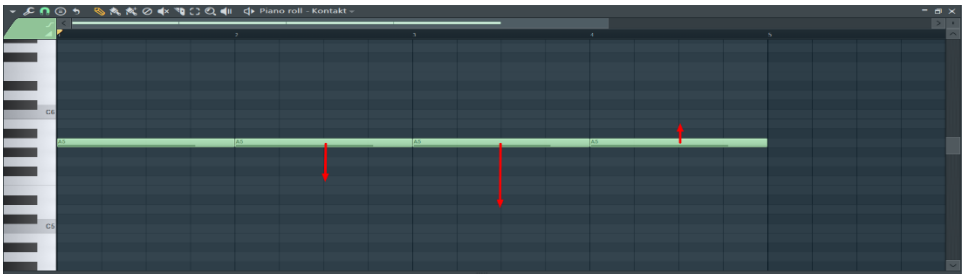


Рисунок 4.4 – Зображення інтервальної закономірності

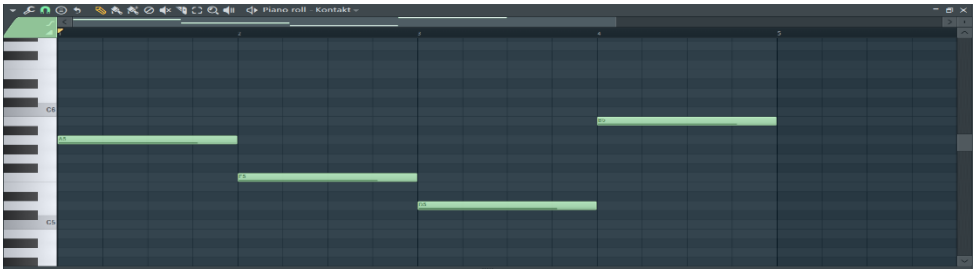


Рисунок 4.5 – Фінальний етап накладання інтервальної закономірності

4.1.2 Генерація треку басу

Фундаментальний трек складається з 2-х сегментів.

Сегмент 1, загальною тривалістю 3 такти (384 імпульси в PPQN) Умови; Інтервальна – зміщення на октаву вниз («-1:0»); Ритмічна – «32|128+0:32|32:64|64:96|112:124».

Сегмент 2, загальною тривалістю 1 такт (128 імпульсів в PPQN) Умови; Інтервальна – зміщення на октаву вниз («-1:0»). Нагадаємо коротко як розшифровуються записи; Ритмічна – «32|64+0:16|16:32|48:64». Ритмічний запис «32|128+0:32|32:64|64:96|112:124».

Зліва від знака розширення «+»: 32 -це розмір четвертої ноти в PPQN (скільки імпульсів вміщується в тривалість четвертої ноти), 128 -це розмір усього зображення в PPQN (в даному випадку, картинка має тривалість рівня 4-ї четвертої ноти).

Праворуч від знака «+» є ритмічний малюнок, де різні ноти розділені знаком «|» знак, ліворуч від «:» -початкова позиція звуку ноти, праворуч - позиція її кінця. Усі значення записуються в PPQN. Ритмічний малюнок (рисунок 4.6).

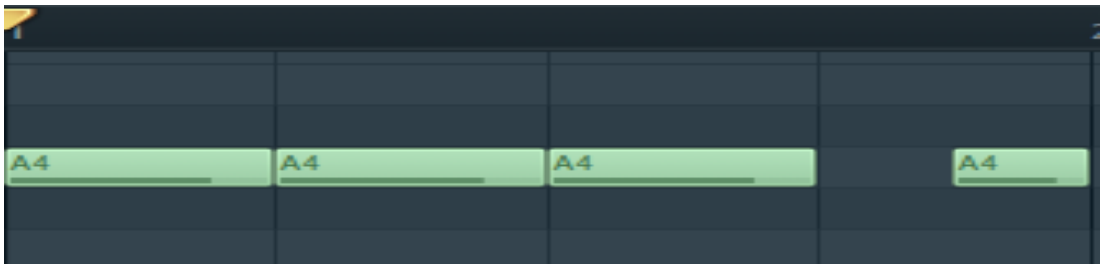


Рисунок 4.6 – Ритм-паттерн відображається на певних нотах у програмі Piano Roll FL Studio. Три чверті, восьма пауза і одна восьма нота

Після застосування умов до обох сегментів на виході маємо такий макет (рисунок 4.7).

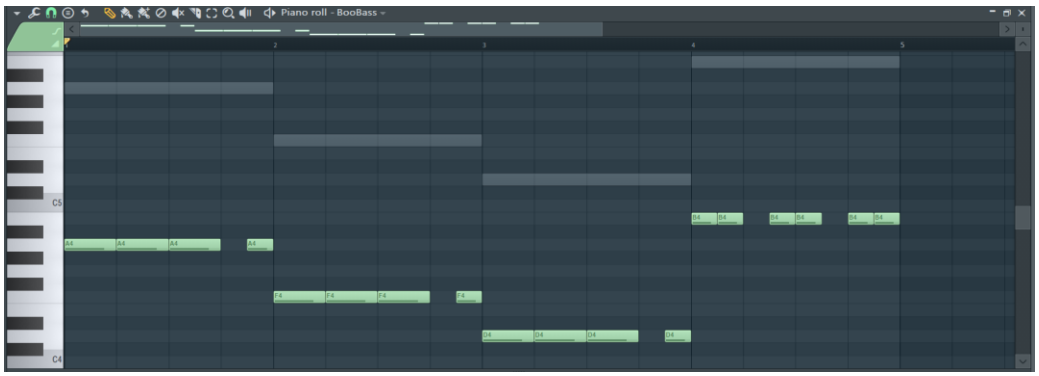


Рисунок 4.7 – Басова партія формується на основі базової партії (базова партія позначена сірим кольором)

4.1.3 Генерація треку акордів

Нехай доріжка акордів складається з двох відрізків, кожен з яких має тривалість 1,5 такту. Перший відрізок починається від початку доріжки, а другий – від середини. Умови для обох сегментів однакові: Гармонійна умова: «-1:0|-3|0|+2» Під час генерації алгоритм використовує ноти базової партії, щоб створити одночасно 4 ноти, утворюючи таким чином акорди. Перша нота зміщується на одну октаву вниз, друга – на 3 ступені вниз (досягаючи квінту акорду, але розташовану знизу). Потім додається тоніка (зміщення «0»), а завершується акорд терцією зверху (зміщення «+2»). Простір між сегментами залишається порожнім (рисунок 4.8).

Рисунок 4.8 – Результат генерації треку акордів (сірим кольором позначені видимі ноти партій базового та басового треків)

4.1.4 Генерація треку мелодії

Мелодію ускладнювати теж не будемо. Зробимо один відрізок тривалістю 4 такти. З такими умовами:

– ритмічна – «32|128+0:16|16:32|32:48|48:96|96:128»;

– інтервальна – «0|+2|+1|+1|-1» Результат генерації на рисунку 4.9.

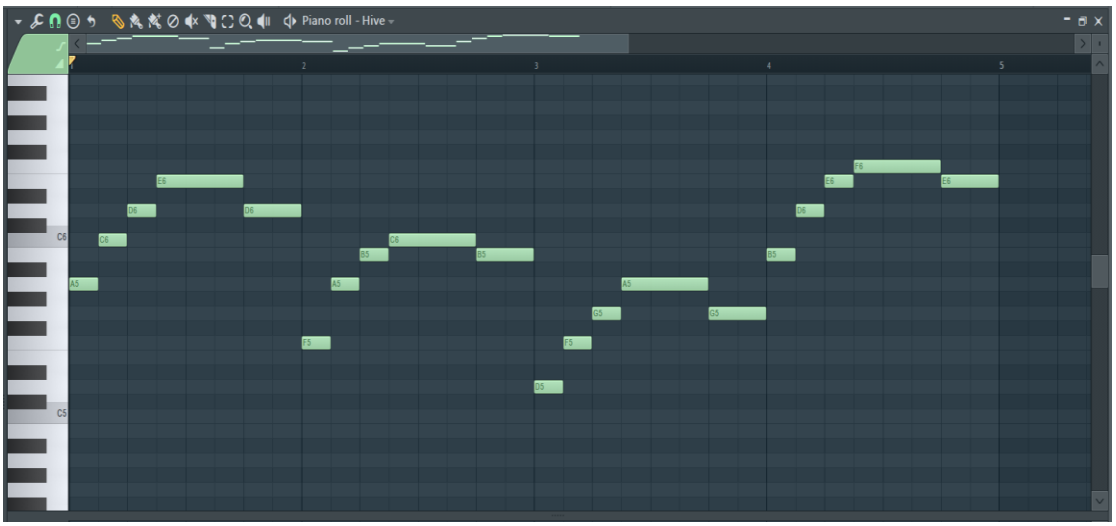


Рисунок 4.9 – Результат генерації треку мелодії

Аналізуючи результати, ми помітили, що останній акорд звучить не гармонійно. Виявляється, ми випадково використали другу тональність як останню ноту основної послідовності. Відомо, що другий ступінь тональності може звучати не дуже гармонійно. Тому замінимо його на третій ступінь. Для цього просто перейдіть до базової доріжки та в умові інтервалу змініть останню цифру послідовності з «0|-2|-4|+1» на «+2». Після перезапуску генерації всі дочірні частини автоматично адаптуються до цих змін. Регенерація зайняла всього 0,007 секунди (7 мілісекунд).

Отже, тепер у нас є такі партії (рисунки 4.10, 4.11, 4.12, 4.13).

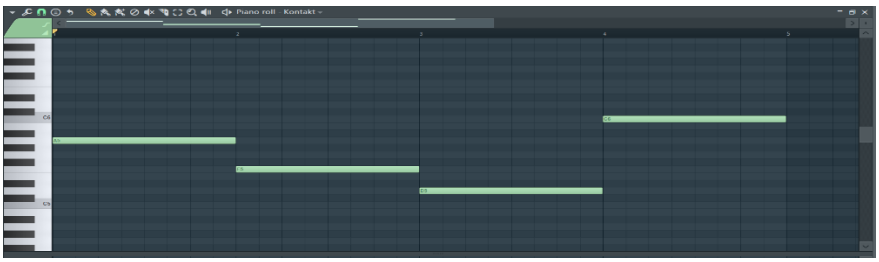


Рисунок 4.10 – Базова партія

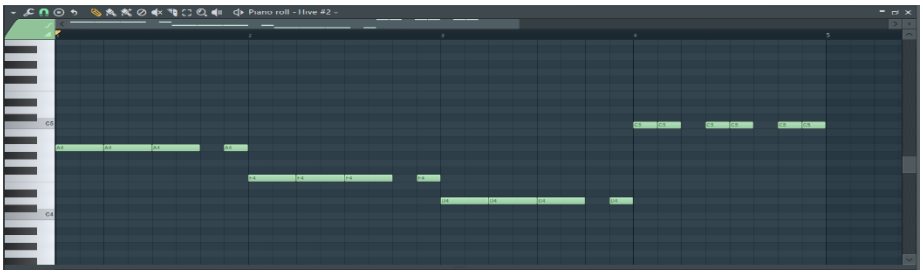


Рисунок 4.11 – Партія басу

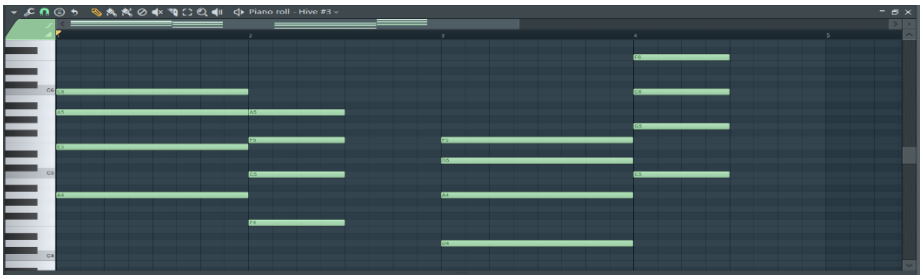


Рисунок 4.12 – Партія акордів

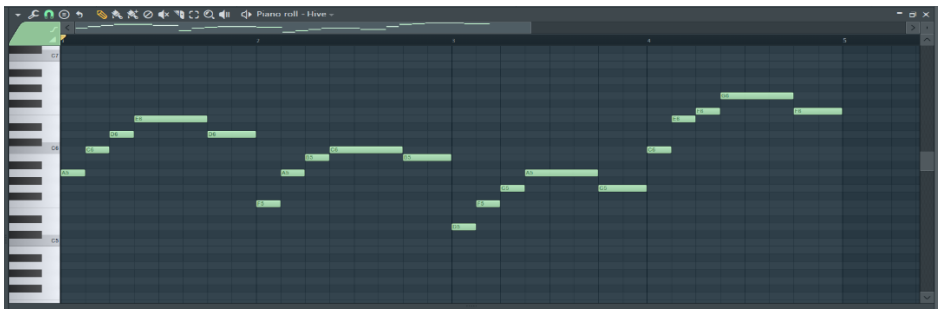


Рисунок 4.13 – Партія мелодії

4.2 Аналіз приросту продуктивності при внесенні змін

Провести таку оцінку складно, оскільки композиції мають різну кількість треків, а швидкість роботи алгоритмів залежить від характеристик комп'ютера, особливо процесора. Крім того, важко об'єктивно оцінити швидкість змін людиною. Різні виробники використовують різні засоби редагування MIDI. Те, що може зайняти кілька хвилин для одного, може зайняти менше часу для іншого. Також важливо враховувати ритм роботи людини. Зрозуміло, що спроба прискорити роботу може вплинути на швидкість роботи.

Незважаючи на це, ми провели невелике дослідження, щоб продемонструвати швидкість внесення змін. Ми взяли згенеровану композицію і почали змінювати умови вручну та програмно.

Програма перевірена на комп'ютері з процесором Intel Core i7 – 8750H 2.21 GHz (таблиця 4.1).

Таблиця 4.1 – Результати тестування програми

№	Ручне редагування, с	Програма, с
1	34	0,003
2	25	0,004
3	31	0,003
4	21	0,003
5	22	0,002
6	40	0,004
7	39	0,002
8	12	0,001
9	31	0,004
1	22	0,001

Загалом, середній час для ручного редагування складає 27,7 секунд.

Однак із змінами програмного забезпечення середній час відновлення всіх чотирьох доріжок становить приблизно 0,003 секунди (3 мілісекунди). Слід зазначити, що сюди не входить час, необхідний для зміни умови в програмі, але він не перевищує 15 секунд у всіх експериментах. Також важливо зауважити, що ця композиція є досить простою з лише чотирьма треками та простими умовами. Якби ми працювали з більш складними зв'язками, ручні зміни зайняли б набагато більше часу.

Ми також повторили експеримент із 10-трековою композицією та складнішими умовами. Виявилося, що ручне редагування зайняло 2 хвилини 3 секунди, зміна умови в програмі – 14 секунд, а повторна генерація – всього 0,003 секунди. Отже, очевидно, що збільшення швидкості впровадження змін є суттєвим.

4.3 Подальші шляхи розвитку програми

Серед можливих напрямків розвитку програми розглядаються два основних шляхи.

Першим варіантом є перетворення програми до формату VST для її використання в інших DAW (цифрових аудіо робочих станціях).

Другий варіант передбачає розвиток програми у повноцінну DAW з впровадженням аналогічних підходів до обробки аудіо, вибору інструментів та застосування ефектів обробки.

На даному етапі здійснюється переважно перший шлях розвитку. У контексті розвитку та поліпшення функціоналу плануються наступні кроки.

Покращення алгоритмів накладання умов з метою зробити їх більш зрозумілими та легкими у використанні для користувача.

Додавання можливості аналізу наявних міді-партій з метою збереження виявлених закономірностей для подальшого використання у редагуванні.

Впровадження спеціального типу умовних шаблонів, які не мають чіткої реалізації, але описують певний характер. Реалізація конкретного шаблону буде генеруватися ітеративно з можливістю змінити його, зберігаючи характер шаблону.

4.4 Висновки до розділу 4

Ми провели аналіз функціонування програми та встановили, що внесення змін за допомогою цієї програми виявляється набагато ефективнішим порівняно з ручним редагуванням. У нашому експерименті ми використовували 4 пов'язані треки: один батьківський та три дочірні. При ручному редагуванні середній час процесу становив 27,7 секунд, тоді як використання автоматичних змін займало лише 0,003 секунди, і це без врахування часу, необхідного на редагування умов (яке у всіх експериментах не перевищувало 15 секунд).

Важливо зазначити, що отримані результати залежать від складності концептуальних зв'язків. У більш складних випадках, коли зв'язків між партіями багато і вони складніші, а також коли партій більше, програма може працювати ще ефективніше, оскільки всі зв'язки будуть враховуватися при автоматичних змінах.

У майбутньому ми розглядаємо два основних напрямки розвитку програми. Перший – це перетворення програми у формат VST для використання в інших DAW (цифрових аудіо робочих станціях). Інший напрямок полягає в розробці повноцінної DAW та впровадженні аналогічних технологій в інших аспектах створення музики, таких як обробка звуку, вибір інструментів та інші, з метою створення повноцінної системи для музичного творчості.

ВИСНОВКИ

Для кращого осмислення та удосконалення процесів, що можна оптимізувати за допомогою автоматизації, ми провели дослідження розвитку музичного мистецтва, запису музики та його контексту. Виявилося, що протягом багатьох століть у розвитку музики виникали правила та закономірності, які з часом стали складовою теорії музики. Різні музичні автори використовують ці закономірності у своїй творчості, навіть якщо це відбувається неосмислено.

У сприйнятті музики, так само як і в будь-якому іншому мистецтві, ключовим є контекст, який виникає зі зв’язків між різними компонентами. Це подібно до того, як слово набуває значення в реченні, а речення – у тексті. Коли автор працює над музичною композицією, він розмірковує більш концептуально, уважно враховуючи зв’язки між її елементами, які несуть в собі певну історію та передають певний настрій. Тому важливо надати автору можливість працювати не тільки на рівні окремих компонентів, але й у контексті композиції в цілому.

Більшість існуючих програм для створення музики сконцентровані на рівні реалізації, а не на концептуальному рівні. Це дозволяє користувачам вільно маніпулювати кожним параметром незалежно, проте обмежує їх у можливості швидкого та ефективного редагування концептуальних значень. З цією метою було прийнято рішення розробити нову систему.

Для розробки цієї системи було проведено дослідження музичної теорії та генеративного мистецтва, оскільки вони базуються на концептуальних зв'язках. Проте існуючі програми-генератори музики не є достатньо гнучкими для роботи на рівні контексту, оскільки вони діють за жорстко заданими алгоритмами, а не враховують бажання користувача.

За результатами дослідження було сформульовано перелік вимог до майбутньої програми:

– програма повинна мати можливість зберігати та застосовувати

концептуальні зв’язки під час роботи над музичною композицією;

– зв’язки мають бути збережені як на рівні окремих музичних частин, так і на рівні всієї композиції. Це означає, що вони будуть встановлені не тільки між різними частинами сторін, а й між самими сторонами;

– програма повинна мати достатню гнучкість, щоб автор міг працювати як на рівні конкретної реалізації, так і на рівні контексту. При цьому концептуальні зв'язки між елементами композиції будуть збережені незалежно від рівня роботи;

– ефективність також є важливою характеристикою програми, щоб вона не виконувала зайву роботу;

– на основі цих вимог була розроблена програма, що втілює концептуальний підхід до створення музики. Вона базується на використанні шаблонів – попередньо підготовлених умов, які впливають на різні концептуальні аспекти музики, такі як ритм, динаміка, інтервали або гармонія. Кожен шаблон містить набір зв'язків, які, хоч і складаються з конкретних значень, але застосовуються тільки в певному контексті.

Шаблони, збережені в базі даних програми, можуть бути легко використані та змінені у будь-який момент. Для створення та збереження зв'язків між елементами композиції в програмі була впроваджена можливість наслідування партій. Наслідування означає, що партія може використовуватися як основа для подальших маніпуляцій у процесі додавання нових умов. З урахуванням того, що для кожної партії можна встановити різну кількість таких умов, процес генерації можна розглядати як послідовний ланцюжок накладання умов різного типу на вже створені ноти.

При розробці алгоритмів ми використовували парадигму «розділяй і володарюй», яка є важливою у сфері розробки алгоритмів на сучасному етапі. Згідно з цією парадигмою, будь-який алгоритм досягає високої ефективності, розбиваючи проблему на більш дрібні, а потім на більш

дрібні, поки вони не стануть елементарними. Потім ці елементарні задачі вирішуються рекурсивно, переходячи від найменших компонентів до розв’язання задачі найвищого рівня.

Створена програма дозволяє зберігати концептуальні зв’язки між елементами композиції, такими як окремі ноти та частини, що сприяє швидшому внесенню концептуальних змін. Це означає, що для зміни, наприклад, ритмічного малюнка на певній ділянці партії, не потрібно редагувати кожну ноту окремо; достатньо змінити або модифікувати умову, що відповідає за цей параметр. Програма автоматично виконає всі необхідні дії на рівні конкретної реалізації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Wooller R., Brown A. R. A framework for comparison of processes in algorithmic music systems. Generative Arts Practice. Sydney: Creativity and Cognition Studios Press, 2005. С. 109–124.

2. Biles A. GenJam in Transition: from Genetic Jammer to Generative Jammer. International Conference on Generative Art. Milan, Italy, 2002.

3. Chomsky N. Three models for the description of language. *IRE Transactions on Information Theory.* 1956. № 2. P. 113–124.

4. Chomsky N. Syntactic structures. The Hague, the Netherlands: Mouton, 1957.

5. Lerdahl F., Jackendoff R. A generative theory of tonal music. Cambridge, Mass: MIT Press, 1983.

6. Eno B. Generative Music. URL: http://www.inmotionmagazine.com/eno1.html (дата звернення: 27.04.2024).

7. Dorin A. Generative processes and the electronic arts. *Organised Sound.* 2001. Т. 6, № 1. P. 47–53.

8. EM-VISIA 2011. Програмка концерту 15 травня 2011 у Великому залі НМА.

9. Hideo N. Mozart Musical Game in C K. 516f*. 28.12.1997. URL: http://www.asahi-net.or.jp/~rb5h-ngc/e/k516f.html (дата звернення: 27.04.2024).

10. Pearson M. Generative art: a practical guide using Processing. 2011. Manning publications co. URL: http://www.manning.com/pearson/GenArt-Sample-Chapter-1.pdf (дата звернення: 27.04.2024).

11. Eldred M. The Quivering of Propriation: A Parallel Way to Music. URL: http://www.arte-fact.org/qvrpropn.html (дата звернення: 27.04.2024).

12. Ihmels T., Riedel J. The methodology of generative art. Media Art Net, Germany. URL: http://www.medienkunstnetz.de/themes/generative-tools/generative-art/ (дата звернення: 27.04.2024).

13. Sychra A. Hudba ocima vedy. Praha, 1965. 300 c.

14. Boden M.A. What is generative art? University of Technology Sydney. URL: http://research.it.uts.edu.au/creative/eae/intart/pdfs/generative-art.pdf (дата звернення: 27.04.2024).

15. Цифрова звукова робоча станція (англ. Digital Audio Workstation, DAW). URL: https://en.wikipedia.org/wiki/Digital_audio_workstation (дата звернення: 27.04.2024).

16. Mubert online service. URL: https://mubert.com/ (дата звернення: 27.04.2024).

17. AIWA online-resource. URL: https://www.aiva.ai/ (дата звернення: 27.04.2024).

18. Cope D. Experiments in Musical Intelligence. Madison, Winconsin: A-R Editions, 1996.

19. Cope D., Hofstadter D. R. Virtual music: computer synthesis of musical style. Cambridge, Mass.: MIT Press, 2001.

20. Desain P., Honing H. Music, mind and machine: studies in computer music, music cognition and artificial intelligence. Amsterdam: Thesis Publishers, 1992.

21. Developer Survey Results 2019. URL: https://insights.stackoverflow.com/survey/2019/ (дата звернення: 27.04.2024).

22. Dodge C., Jerse T. A. Computer Music. 2nd ed. New York: Schirmer Books, 1997.

23. Hiller L., Isaacson L. Musical Composition with a High-Speed Digital Computer. Machine Models of Music. 1958.

24. IBM-Computer-Music-Center. Music Sketcher. IBM, 1999. (дата звернення: 27.04.2024).

25. Laurson M. Patchwork. Helsinki: Sibelius Academy, 1996.

26. Lippe C. Music for piano and computer: A description. *Information Processing Society of Japan SIG Notes.* 1997. Т. 97, № 122. P. 33–38.

27. Loy G., Abbott C. Programming languages for computer music synthesis, performance and composition. *ACM Computing Surveys (CSUR).* 1985. T. 17, № 2. P. 235–265.

28. Microsoft. DirectMusic Producer 5.3.0.900. Microsoft, 2001. URL: http://www.musicmachines.net/ (дата звернення: 27.04.2024).

29. Mozer M. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multiscale processing. *Connection Science*. 1994.

30. Pachet F. Playing with Virtual Musicians: the Continuator in Practice. *IEEE Multimedia*. 2002. T. 9, № 3. P. 77–82.

31. Papadopoulos G., Wiggins G. AI Methods for Algorithmic Composition: A Survey, A Critical View, and Future Prospects. *AISB'99 Symposium on Musical Creativity*. Edinburgh, 1999.

32. Puckette M., Zicarelli D. MAX. Opcode. 1990. URL: http://www.cycling74.com (дата звернення: 27.04.2024).

33. Roads C. The Computer Music Tutorial. Cambridge, Massachusetts: MIT Press, 1996.

34. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 2nd ed. New Jersey: Prentice Hall, 2004.

35. Truax B. A Communicational Approach to Computer Sound Programs. *Journal of Music Theory.* 1976. T. 20, № 2. P. 227–300.

36. Winkler T. Composing Interactive Music. Cambridge, Massachusetts: MIT Press, 1998.