CONSERVATOIRE NATIONAL SUPÉRIEUR MUSIQUE ET DANSE DE LYON

Mémoire de recherche

2ème cycle (Master)

Année universitaire 2022 - 2023

ScoreGen : développement d'une bibliothèque logicielle C++ pour la CAO, la génération de partitions musicales et la composition algorithmique

Antoine Gabriel BRUN

Discipline principale : écriture-composition

Professeur de la discipline principale : David Chappuis

Professeur référent : David Chappuis

Accompagnant méthodologique : Jean-Jacques Benaily

Merci à David Chappuis pour son précieux regard

sur mon travail

.

Sommaire

Quatre réponses du logiciel MidJourney à l'invite suivante :

« A futuristic machine for composing music »

Avant-propos

Ce travail de recherche a pour objet le développement d'un système de CAO (composition assistée par ordinateur). Présenté dans le cadre d'un second travail de recherche, il s'éloigne parfois quelque peu de la forme mémoire pour adopter une forme *ad hoc*, incluant une discussion théorique, un compte-rendu de recherche, une présentation détaillée de la bibliothèque créée et des choix qui ont présidé à sa conception, ainsi que des exemples de réalisations.

Le centre autour duquel gravite ce document, la bibliothèque logicielle à proprement parler, reste absent de la forme finale de ce texte. Imprimé, il constituerait une annexe d'une soixantaine de pages à la présentation problématique (le code en C++ souffrant mal les retours à la ligne imposés par l'impression) et d'un examen fastidieux.

Le code source dont il est question, ainsi que plusieurs ressources audiovisuelles utiles à la lecture de ce mémoire, sont fournis pour référence à l'adresse web suivante :

`http://antoinegabrielbrun.com/ressources/scoregen-memoire-de-recherche/`

Introduction

La maîtrise d'un environnement de CAO (composition assistée par ordinateur) est aujourd'hui un incontournable de la formation des jeunes compositeurs, quelles que soient l'esthétique visée (musique contemporaine, musique pour l'image, musique électro) et la place que ce type d'outils est amenée à occuper dans leurs œuvres (génération de textures, gestion de la structure, aide à l'orchestration, synthèse mélodique ou harmonique).

Les outils de CAO existants sont en général conçus pour des musiciens non formés à la programmation informatique ; cependant, ils font tous appel, de façon explicite ou implicite, à des concepts issus du domaine de la science algorithmique : entrées et sorties, structures de données, boucles et itérations, structures conditionnelles, fonctions, algorithmes, récursivité, programmation par contraintes, encapsulation.

Pour permettre à des novices en termes de programmation de manipuler des notions parfois complexes d'algorithmique, la plupart des environnements de CAO se sont tournés vers le paradigme de la « programmation graphique », jugé plus simple à appréhender : les primitives des langages en question sont non pas des mots-clés organisés en instructions, mais des boîtiers visualisables sur l'écran et que l'utilisateur relie entre eux de façon à réaliser les fonctions désirées.

Ce choix pratique, qui sera discuté dans la suite de notre travail, n'est pas sans inconvénients ni sans limitations. Les outils ainsi conçus tendent à limiter la complexité des codes produits, à moins de perdre leur lisibilité et leur réutilisabilité (deux exigences pourtant fondamentales dans le domaine de la programmation) ; leur rigidité force les utilisateurs à de nombreuses contorsions mentales, et ces derniers sont en fin de compte tributaires d'un environnement d'exécution, menacé comme tout logiciel de péremption.

Ces limitations, les questionnements sur l'employabilité de tels systèmes dans un contexte réel de composition, et une inclination personnelle à construire les outils qui nous sont nécessaires, nous ont conduit à réaliser un nouvel environnement de CAO non graphique, basé sur un langage universellement employé, le C++, sous la forme d'une

bibliothèque logicielle capable de générer en sortie des fichiers MusicXML utilisables dans n'importe quel éditeur de partitions ou n'importe quel DAW (station audionumérique).

Cet outil, baptisé ScoreGen, s'adresse à des compositeurs que ne rebute pas la programmation informatique textuelle, ou qui acceptent de prendre le temps de cet apprentissage dans le but d'en moins perdre par la suite. Il est utilisable en C++, mais aussi dans des langages réputés plus simples d'utilisation et supportant l'usage de bibliothèques logicielles écrites en C++, comme le Python.

Nous avons porté notre attention sur les points qui nous semblaient faire défaut aux outils existants : programmes lisibles et commentables, structure modulaire, facilité de réutilisation et de modification des programmes, facilité de réalisation d'algorithmes complexes.

L'objet de ce document est de montrer comment la bibliothèque logicielle créée répond aux exigences ainsi fixées.

Pour répondre à cet objectif, nous commencerons par aborder les grandes orientations suivies durant le développement de ScoreGen, en définissant précisément ce en quoi il consiste et en mettant l'accent sur la nécessité de ce nouvel outil.

Nous fournirons ensuite une présentation détaillée de la bibliothèque et de son organisation, avec la hiérarchie des composants et la structuration choisie pour manipuler les éléments musicaux au sein de programmes.

Enfin, nous aborderons brièvement les développements que nous souhaitons réaliser sur notre outil afin de le rendre encore plus complet, interopérable et accessible.

I. Compte rendu de recherche sur ScoreGen : orientations et choix directeurs

ScoreGen, outil de programmation informatique pour la génération de partitions, est né d'un long processus de réflexion sur les objectifs visés, sur les outils existants, sur la représentation logique des objets musicaux.

Sa création a été jalonnée de choix et d'arbitrages, visant à délimiter ce que ce système est destiné à accomplir et ce qui reste en-dehors de son champ d'application. Cette première partie vise à faire une description critique des grandes orientations qui ont été retenues pour le projet.

Pour cela, nous proposons tout d'abord une description de ScoreGen. Elle sera complétée d'une discussion sur la nécessité d'un nouvel outil par rapport aux alternatives existantes, et enfin d'une présentation des choix concrets auquel ces réflexions ont abouti.

A. ScoreGen : définition et délimitation de l'objet

ScoreGen est une bibliothèque logicielle destinée à la manipulation de contenu musical et à sa génération sous forme de partitions.

1. Un système de manipulation d'objets musicaux

Tout programme informatique manipule des données. Il peut s'agir de données simples, gérées nativement par les langages de programmation, comme des nombres, des chaînes de caractères, des valeurs logiques.

D'autres données manipulées par les programmes peuvent atteindre un plus grand niveau de complexité, ou bien parce qu'elles présentent une structuration de données de bas niveau (tableaux et autres structures), ou bien en ce qu'elles représentent des objets eux-mêmes complexes ou abstraits (images, sons, fichiers, graphes, etc.)

Lorsqu'il s'agit de manipuler des éléments musicaux par programme, ou dit autrement, lorsqu'un programme doit accepter comme données des partitions musicales et leurs éléments, au moins trois difficultés se font jour :

- Aucun langage de programmation standard ne permet de manipuler nativement des données comme un accord, un rythme, un triolet ou *a fortiori* une partition dans son ensemble ;
- La représentation de ce qu'est une partition en termes structurels est sujette à débats et dépend vraisemblablement des types de musique envisagés ;
- Les formats de fichiers destinés à la représentation de la musique sont ou bien opaques (formats propriétaires gérés par des logiciels de gravure musicale), ou bien structurés d'une façon qui rend très peu pratique leur usage sous forme de chaînes de caractères dans un programme.

Ainsi, une bibliothèque qui comme ScoreGen a l'ambition de rendre possible la manipulation formelle d'objets musicaux doit imaginer et construire des formats de données capables de représenter, de créer, de modifier et de faire interagir les éléments présents sur une partition.

À titre d'exemple, la bibliothèque ScoreGen permet en premier lieu à un programmeur d'écrire quelque chose comme : « Crée une partition à une portée contenant cinq mesures en 3/4+3/8. Remplis-la de croches avec accents qui sont des accords à trois sons aléatoires dans les notes de sol mixolydien. Ajoute une deuxième voix au-dessus de la première, contenant un rythme de doubles croches de percussions dont la moitié est aléatoirement accentuée ». Le destinataire de ce jeu d'instructions est bien sûr l'ordinateur, et le langage employé n'est pas le langage naturel mais du C++ ou un autre langage utilisant la bibliothèque.

Une grande partie du code de la bibliothèque que nous avons conçue sert donc à définir des types de données manipulables par le programme : hauteurs de notes, figures rythmiques, notes et silences, mesures, signatures rythmiques, nuances, blocs de musique, partitions, etc. Ces différents types de données et leur hiérarchie ont fait l'objet d'une intense réflexion et de nombreux remaniements (un silence est-il un accord qui n'a aucune note ? un triolet peut-il être à cheval sur une barre de mesure ? un crescendo

est-il attaché à un groupe de notes ou à une position dans une mesure ?) ; ils sont présentés de façon détaillée sous leur forme définitive dans la deuxième partie de ce travail.

2. Un système centré sur la synthèse de partitions

Une fois en présence d'un système capable de représenter et de manipuler des objets musicaux, il nous a fallu définir des moyens de visualiser, ou tout du moins d'exporter, les données manipulées sous une forme utilisable.

Un point central dans la définition de ce qu'est ScoreGen est le suivant : l'objectif de la bibliothèque est de permettre la synthèse de partitions sous une forme manipulable par d'autres programmes. Ainsi :

- Nous aurions pu prévoir que notre bibliothèque permette d'afficher les objets musicaux créés sous forme d'images. La partition décrite ci-dessus aurait ainsi pu être affichée sur un écran ou imprimée, puis jouée par deux instrumentistes. Cependant nous aurions alors été dans l'incapacité d'éditer la partition générée à l'aide d'un logiciel de gravure musicale comme MuseScore, Finale ou Sibelius. Il aurait aussi été impossible de réimporter la partition dans ScoreGen ou de la faire lire en MIDI ou à l'aide d'un séquenceur par l'ordinateur.
- Nous aurions pu permettre à ScoreGen de générer des fichiers MIDI. Les partitions auraient alors pu être jouées par l'ordinateur et importées dans un logiciel comme un DAW (station audionumérique) voire, moyennant quelques imprécisions, dans un éditeur de partitions. Mais il aurait alors été impossible de donner à des interprètes humains le produit généré – le format MIDI n'offrant pas la possibilité d'inclure les notations autres que les notes, leur durée en fractions de mesures, leur puissance individuelle et quelques effets stockés numériquement.

Comme dans le premier cas de figure évoqué ci-dessus, nous avons résolument priviligié la possibilité de synthétiser une *partition* en sortie des programmes faits avec ScoreGen, dans le but de préserver la possibilité de jeu par des interprètes. Cependant, plutôt que

des images, nous avons choisi comme format de sortie MusicXML, un format dédié à la représentation de partitions.

Ce choix est discuté plus loin dans ce document ; notons toutefois, comme remarque préalable, qu'il présente tous les avantages des propositions ci-dessus et aucun de leurs inconvénients :

- Le matériel musical généré peut être converti en image, affiché sur un écran ou imprimé ;
- Il peut être fourni à des instrumentistes avec toutes les informations que l'on s'attend à trouver sur une partition ;
- Il peut être ouvert, édité, modifié ou mis en page dans n'importe quel logiciel de gravure musicale et dans n'importe quel DAW ou séquenceur.

Le MusicXML joue ici le rôle de *lingua franca* entre différents « interlocuteurs » humains et informatiques : il est synthétisé par ScoreGen, peut être lu par un humain, ouvert dans tout logiciel spécialisé et facilement converti dans d'autres formats.

3. Quelques exemples préalables d'utilisations de ScoreGen

Nous avons déjà montré un pseudo-code s'adressant à ScoreGen, et une partie sera consacrée plus loin à l'examen d'exemples de vrais codes écrits dans des situations réelles de composition.

Nous souhaitons toutefois présenter ici quelques exemples d'usages possibles de ScoreGen, sans montrer leur réalisation en code, et dans le but d'illustrer les deux points que nous venons d'aborder (manipulation de données musicales et export en partitions au format MusicXML) :

- Nous pourrions en quelques lignes de code créer une partition pédagogique montrant toutes les gammes majeures et mineures, ou tous les accords communs, puis l'éditer dans un logiciel de gravure musicale pour la mettre en page et l'imprimer.

- Nous pourrions récupérer sur Internet un répertoire de tous les chorals de Bach au format MusicXML, les charger dans ScoreGen, et utiliser un programme pour rechercher toutes les occurrences d'une figure musicale donnée (par exemple les mouvements de basses présentant plus de huit croches consécutives, les enchaînements d'accords de septième diminuée, les croisements de voix durant plus d'une note ou les chromatismes au ténor).
- Nous pourrions charger une partition pour piano et écrire un programme ayant pour mission de l'orchestrer en *Klangfarbenmelodie*.
- Nous pourrions composer une pièce pour ensemble de percussions entièrement algorithmique, où le nombre de parties soit une option choisie par l'utilisateur à chaque exécution. Deux synthèses consécutives pourraient alors engendrer la même pièce, par exemple, pour trio ou pour ensemble de quarante percussions.

Ces idées, même si aucune n'a été concrètement accomplie, sont toutes réalistes avec la bibliothèque que nous avons créée.

4. Comparaison avec des outils connexes (délimitation du champ)

Par la comparaison de ScoreGen avec des outils existants et présentant une visée similaire ou non, cette partie vise à montrer ce que ScoreGen n'est pas, et à délimiter en négatif son ambition et son champ d'utilisation.

a) ScoreGen n'est pas un logiciel de synthèse sonore

ScoreGen ne gère que des partitions formelles, et laisse ou bien à des logiciels comme Csound, Max/MSP, Ableton, REAPER, ou bien à des interprètes humains, le soin de les traduire en sons.

b) ScoreGen n'est pas un logiciel de gravure musicale

ScoreGen génère des partitions, mais se concentre sur leur contenu musical. La bibliothèque ne gère pas la mise en page, les couleurs, ni la taille des éléments sauf si elle est pertinente musicalement (appogiatures). Ce choix allège considérablement la bibliothèque, mais il implique que pour être édités, mis en page, ou même simplement

visualisés et imprimés, les objets musicaux synthétisés par ScoreGen doivent être ouverts dans un logiciel tiers.

c) ScoreGen n'est pas un langage descriptif

Au contraire de LilyPond et de MusicXML lui-même, ScoreGen est plus qu'un format de description de partitions. L'écriture d'une partition avec LilyPond nécessite de la concevoir au préalable. À l'inverse, ScoreGen peut générer une partition *via* un ou plusieurs algorithmes, impliquant éventuellement des paramètres modifiables ou des valeurs aléatoires. La sortie d'un même programme peut être différente d'une exécution à la suivante, et le programmeur n'a pas besoin d'écrire manuellement chaque élément présent dans l'objet final.

d) ScoreGen ne propose pas d'interface utilisateur

Là où la plupart des outils de CAO se présentent comme des logiciels avec une fenêtre, des interactions clavier et souris et des zones de visualisation, ScoreGen est une bibliothèque abstraite. Des interfaces et des visualisations peuvent être utilisées, mais elles sont à la charge de l'utilisateur. La logique de programmation en elle-même n'est pas visuelle (comme dans les outils OpenMusic ou PWGL) mais textuelle. Cette question est discutée en détail plus loin.

5. Définition et mise en perspective des termes employés

Avant d'aller plus avant, il nous apparaît nécessaire de définir quelques termes et de les expliquer dans le contexte de notre travail. Ces termes ont déjà été rencontrés plus haut, il s'agit ici d'en donner une explication plus précise que celle que fournit le contexte.

• **CAO.** Composition assistée par ordinateur. Il s'agit de l'ensemble des techniques utilisant un ordinateur pour assister le compositeur dans la création ou la modification d'un matériau musical. – ScoreGen est conçu pour la CAO, mais son usage n'y est pas limité (voir plus haut dans les exemples préalables d'utilisation). En plus des types de données permettant de manipuler les éléments musicaux, la bibliothèque comprend de nombreuses fonctionnalités « clés en main » pour la gestion de processus compositionnels variés.

• **Composition algorithmique.** La composition algorithmique est l'ensemble des techniques de composition où un matériau, voire la structure d'un morceau, est engendrée par un calcul. J. BRESSON la définit de façon similaire comme « toute forme d'application d'algorithmes ou de programmes (qu'ils soient complètement déterministes ou aléatoires) pour produire de la musique[1] ». Les processus de composition algorithmique ne sont pas nécessairement calculés sur un ordinateur, ils peuvent privilégier un processus physique, une mesure ou un calcul manuel. – ScoreGen ne se limite pas à la composition algorithmique, puisqu'il peut être utilisé pour générer des partitions note à note et signe à signe. Toutefois, tout l'intérêt de la bibliothèque est de permettre à un compositeur de concevoir des algorithmes générant des partitions.

• **Bibliothèque logicielle.** Une bibliothèque logicielle est un composant logiciel destiné à être utilisé dans un programme et à lui offrir des fonctionnalités. – ScoreGen n'est pas un logiciel mais une *bibliothèque logicielle*. Cela signifie que ScoreGen est un ensemble d'abstractions et de fonctions destinées à être utilisée par un programme tiers. En d'autres termes, ScoreGen offre à un programmeur la possibilité, *dans ses programmes*, de manipuler des partitions.

• **MusicXML.** MusicXML est un format libre de représentation de partitions. Il est conçu pour être structuré hiérarchiquement, pour être lisible dans un éditeur de texte et pour être facile à générer par un programme. Il s'agit d'un format standard, éditable dans tout éditeur de texte et supporté par la majorité des logiciels musicaux. – ScoreGen est conçu pour générer des fichiers au format MusicXML.

• **C++.** Le C++ est un des langages de programmation les plus répandus. Il est compatible avec tous les systèmes d'exploitation communs, peut être compilé par des centaines de compilateurs libres, et le risque de le voir rendu obsolète est faible. – La bibliothèque ScoreGen est écrite en C++/CLI, une variante du C++ disposant d'un ramasse-miettes[2]. Le choix de ce langage est détaillé plus loin.

• **Éléments musicaux.** Nous avons écrit plus haut que ScoreGen permettait de manipuler des *élements musicaux*. Ce terme recouvre ici toutes les entités appartenant à

[1] BRESSON, Jean. *Composition assistée par ordinateur : techniques et outils de programmation visuelle pour la création musicale*. Université Pierre et Marie Curie, 2017.
[2] Dispositif absent du C++ standard, et permettant de libérer automatiquement et en cours d'exécution les zones de mémoire n'étant plus utilisées par le programme.

la sémantique musicale et ayant leur place sur une partition, comme une note, un crescendo, une clé d'ut ou une altération. Le terme « élément » est à voir ici non comme « partie indivisible » mais comme « composant d'un tout » ; ainsi une mesure est un élément au même titre qu'un symbole de staccato. – ScoreGen est basé sur la décomposition de la partition en éléments organisés hiérarchiquement au sein de la partition.

B. Discussion : nécessité d'un nouvel outil de CAO

Nous avons défini à quoi sert ScoreGen, avons effleuré son fonctionnement général et l'avons comparé à d'autres outils existants. Ce dernier point nous amène maintenant à une question d'importance : quel est l'avantage de proposer un nouvel outil de CAO face aux solutions existantes ?

1. CAO : une abondance d'outils disponibles

Depuis les premières expérimentations dans le domaine de la composition assistée par ordinateur par I. Xenakis, la CAO s'est structurée autour de la double dynamique des besoins exprimés par les compositeurs et des propositions d'outils faites par des programmeurs ou des institutions.

Ce double mouvement explique la coexistance d'outils artisanaux, développés par les compositeurs eux-mêmes et souvent dépourvus de représentation structurée des éléments musicaux, avec des outils professionnels et versatiles, parfois peu flexibles lorsqu'il s'agit de les employer pour de nouveaux usages musicaux.

Le secteur de la CAO est dominé par des logiciels comme OpenMusic, PWGL et Csound ; il est complété par des outils artisanaux réalisant des tâches précises, des plug-ins pour les logiciels de gravure musicale et quelques plug-ins pour DAW. Les premiers logiciels de cette liste seront examinés plus en détail dans le partie suivante.

2. Alternatives à la conception d'un nouvel outil de CAO

Examinons donc les possibilités qui s'offrent à nous si nous voulons générer des partitions et manipuler des données musicales avec des outils existants.

a) Génération de code LilyPond

Le logiciel libre LilyPond offre la possibilité de générer des partitions *via* un langage de description textuelle. Cela permet à tout programmeur de générer assez facilement des partitions par programme : il suffit d'écrire un code qui génère le texte descriptif, puis de laisser LilyPond le convertir en partition lisible par un interprète.

Cette solution est fonctionnelle et applicable dans un contexte réel de composition. Elle peut suffire pour créer des séquences mélodiques ou rythmiques à l'aide d'un programme, ou pour créer divers autres matériaux musicaux.

Elle souffre toutefois de deux limitations.

Tout d'abord, les partitions générées sont des images ; il n'est donc pas possible d'importer les résultats dans un autre logiciel, par exemple simplement pour les écouter. Il faudrait pour cela opérer une conversion du texte LilyPond en MusicXML ou en MIDI, ce qui n'est pas un processus simple et risque d'occasionner des pertes de données.

Surtout, le texte LilyPond est peu pratique à manipuler dans un programme car il n'est pas structuré : il s'agit d'une séquence de caractères non hiérarchisée. Utiliser ce système pour générer des objets musicaux complexes peut obliger le programmeur à créer sa propre bibliothèque de manipulation des éléments musicaux… ce qu'il ne peut pas réaliser de façon exhaustive en un temps court.

b) Csound

Csound est à la fois un logiciel de synthèse sonore et un système de CAO. Il est possible d'y créer des structures musicales complexes, mais aussi de les exporter dans divers formats.

Si cette solution permet de nombreuses applications en CAO et en composition algorithmique, les objets musicaux exportés ne seront pas des partitions complètes : hors

des notes, toute indication musicale leur fera défaut. Il est par exemple inenvisageable d'utiliser Csound pour générer une partition de chœur avec des paroles. D'autre part, Csound est pensé pour le temps réel et le *live coding*, ce qui peut rendre difficile la conception de structures musicales abstraites vues dans leur ensemble. Reste enfin la dépendance vis-à-vis d'un logiciel susceptible d'être un jour abandonné au profit d'un successeur, laissant obsolètes les programmes conçus avec son aide.

c) Génération de fichiers MIDI

La génération de fichiers MIDI est une voie valide pour la création de matériaux musicaux par algorithmes. Elle est même supérieure aux possibilités de ScoreGen en ce qu'elle ne force pas l'utilisateur à respecter des rythmes écrits mais le laisse placer tout événement à un moment libre.

Pour utiliser cette méthode, un compositeur devrait construire son propre programme de génération de fichiers MIDI, ou se procurer une bibliothèque spécialisée existante.

Cependant, cette méthode présente des limitations déjà rencontrées : impossibilité de générer des partitions détaillées, absence de structuration des éléments musicaux.

d) OpenMusic et PWGL

OpenMusic et PWGL sont deux logiciels de CAO très utilisés par les compositeurs. Ils s'affranchissent de la plupart des limitations rencontrées ci-dessus grâce à des représentations structurées des matériaux musicaux et à la capacité d'export vers des formats de données standard.

Ces solutions sont au plus proche de la vocation de ScoreGen, c'est pourquoi nous prendrons plus de temps pour en discuter les avantages et les inconvénients.

Ces deux logiciels sont gratuits, bien documentés, largement utilisés et disponibles sur plusieurs architectures. Elles permettent de lire et d'écrire des fichiers MIDI et MusicXML, et intègrent la possibilité de visualiser et d'écouter des éléments musicaux en temps réel. Les structures de données utilisées sont multiples (MIDI, partitions et audio).

Ces deux logiciels représent donc deux voies royales pour le compositeur désirant utiliser la CAO.

Ils présentent toutefois assez d'inconvénients pour motiver la conception d'outils alternatifs.

Non compilés, les patchs conçus avec ces logiciels sont notoirement lents : même sur un ordinateur récent, la génération d'une partition complexe peut être très longue voire mener à un plantage logiciel, rendant le travail entrecoupé et fastidieux.

De plus, ces deux logiciels peuvent devenir obsolètes à moyen terme, pour peu que les institutions qui les mettent à jour (l'Ircam pour OpenMusic, la City University of New York pour PWGL) décident d'arrêter leur développement, et comme cela se passe invariablement pour quasiment tous les logiciels au cours des décennies[3].

OpenMusic et PWGL restent peu extensibles, obligeant les utilisateurs à adopter la logique dictée par les fonctions proposées (ou à programmer leurs propres primitives, ce qui est à la portée d'une minorité d'entre eux).

Le dernier point est le plus important, et celui qui a motivé le plus directement notre désir de créer un système alternatif : OpenMusic et PWGL font appel à la programmation graphique, c'est-à-dire que la programmation dans ces environnements consiste à disposer et à interconnecter des boîtiers réalisant chacun une fonction. Ce point, qui est le centre de notre argumentaire, fait l'objet de la partie suivante.

3. Programmation graphique contre programmation textuelle

Dans la programmation graphique, les *primitives* ne sont pas des mots-clés mais des boîtiers placés sur l'écran et reliés entre eux de façon à réaliser les fonctions désirées[4] :

[3] Nous faisons donc ici la distinction entre les *logiciels* (outils informatiques complets destinés à une classe donnée d'architectures informatiques, disposant d'une interface, et sujets à une obsolescence presque inévitable), et les *langages*, qui comme le C peuvent être maintenus et utilisés pendant des dizaines d'années.

[4] En informatique, les *primitives* sont les briques élémentaires (boîtiers graphiques, mots-clés, opérateurs…) à partir desquelles les utilisateurs peuvent construire leurs programmes ou leurs patchs.

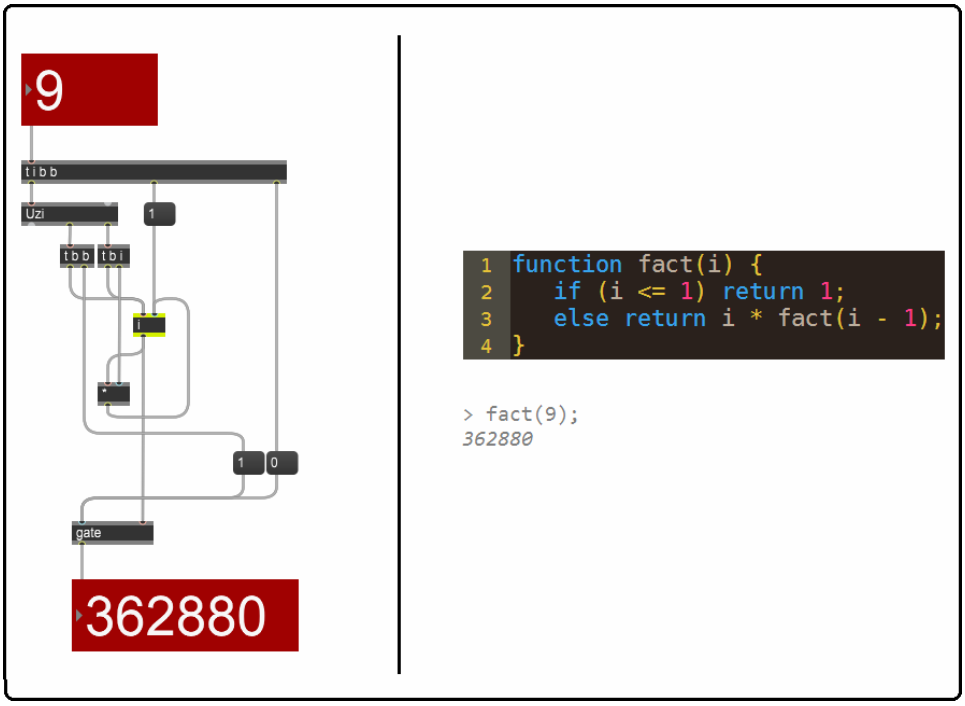


Figure 1 : La fonction factorielle*, réalisations graphique (Max/MSP) et textuelle (JavaScript)*

Comme on le voit dans la Figure 1 ci-dessus, programmation graphique et programmation textuelle ne s'opposent que sur la forme : les deux paradigmes permettent *a priori* de réaliser les mêmes opérations et de mettre en œuvre les mêmes notions algorithmiques.

L'observation de cet exemple, pensé comme une simple illustration du concept de programmation graphique, annonce déjà la thèse présentée ici : la version textuelle est plus concise et infiniment plus claire. L'auteur, parfaitement expérimenté avec les deux langages utilisés, a d'ailleurs passé pas moins d'une demi-heure à créer l'exemple de gauche, contre au plus une minute pour celui de droite réalisant la même fonction.

a) Les outils de CAO utilisent la programmation graphique

Les outils de CAO les plus utilisés (OpenMusic, PWGL, ainsi que Bach et Cage au sein de l'environnement Max/MSP) recourent tous à une représentation graphique des programmes (patchs).

Les langages de programmation graphiques ont la réputation d'être plus accessibles aux non-programmeurs, plus simples à appréhender avec la visualisation qu'ils offrent des fonctions et des flux de données. Ils permettent d'inclure des objets de débogage affichant en temps réel les dernières données échangées en plusieurs points du patch, ne

demandent pas de compilation[5], et permettent d'envisager la structure d'un programme comme un schéma visuel. Nous ne pouvons nous empêcher de penser que ces environnements offrent aussi à des compositeurs non-programmeurs une approche plus ludique de l'algorithmique que l'apprentissage du code, réputé abstrus.

Signalons que les environnements de programmation graphique proposent en général d'intégrer des extraits de code dans les patchs. L'approche habituelle consiste à proposer aux utilisateurs un type de « boîte » graphique, extérieurement semblable aux primitives du langage, mais qui une fois ouverte donne accès à un éditeur de code. La progammation du comportement de la boîte se fait alors en codant dans un langage textuel. Ainsi, OpenMusic propose un objet [lispfunction], dans lequel l'utilisateur est libre d'écrire un code en Lisp ; Max/MSP a plusieurs objets similaires utilisant entre autres JavaScript, et permet même de s'en servir pour créer programmatiquement des zones d'interface graphique (d'affichage ou interactives). Nous avons ainsi pu tester que le code en JavaScript ci-dessus, intégré dans l'environnement Max/MSP, donnait bien les mêmes résultats que sa version graphique.

Ces tentatives augmentent considérablement la puissance et la flexibilité des langages graphiques, ce qui démontre la supériorité du code sur le patch. Toutefois elles donnent accès à des éditeurs de code limités, sans possibilité de gestion de fichiers et de bibliothèques (bases stratégiques pour l'organisation de codes de grande taille), et rendent difficile l'encapsulation fine et le débogage.

b) Inconvénients propres aux langages graphiques

Nous avons vu ci-dessus la complexité que revêt la mise en place d'un algorithme itératif simple au sein d'un environnement graphique. Établissons maintenant une liste des difficultés systémiques qu'occasionnent ces environnements ; seule la première a été théorisée et nommée, les suivantes relevant de considérations personnelles.

• **Limite de Deutsch[6].** Cette observation empirique, énoncée par l'informaticien américain L. Peter Deutsch, postule : « The problem with visual programming languages

[5] Le fait que les langages de programmation graphiques soient interprétés et non compilés ne semble pas relever d'une nécessité. Il se trouve toutefois que tous les environnements dont nous avons connaissance permettent l'édition du patch en cours d'exécution, et nécessitent donc une approche interprétative.
[6] La dénomination anglaise « Deutsch limit » est plus souvent employée.

is that you can't have more than fifty visual primitives on the screen at the same time »[7] (« Le problème des langages de programmation graphiques est qu'on ne peut pas avoir plus de cinquante primitives visuelles à l'écran en même temps »). Dans sa première formulation, Deutsch l'accompagne avec malice de la question suivante : « How are you going to write an operating system? » (« [Dans ces conditions], comment allez-vous écrire un système d'exploitation ? »). Ce nombre indicatif limite *de facto* la complexité des algorithmes faciles à réaliser en programmation graphique.

En pratique, la programmation de fonctions algorithmiquement complexes est extrêmement ardue dans les contextes de programmation graphique et donne lieu à des patchs tentaculaires effectivement impossibles à visualiser d'un seul coup d'œil.

• **Difficulté à nommer les données.** La programmation textuelle classique procède en général par assignation et nommage : chaque résultat de calcul utile dans la suite d'un algorithme est stocké dans une variable ayant un nom décrivant son contenu. Cet étiquetage systématique des données vaut aussi pour les fonctions et les types de données définis par l'utilisateur ; il est toujours complété par des commentaires[8] qui explicitent la fonction de chaque section de code.

Essentiels pour la compréhension du code, indispensables pour sa réutilisabilité, et toujours cités en premier dans les listes de bonnes pratiques de programmation, le nommage des données et la présence de commentaires sont difficiles et fastidieux à mettre en place dans un patch. Voici par exemple (Figure 2 : Variables et commentaires dans un patch Max/MSP) le patch déjà montré plus haut de calcul d'une factorielle dans l'environnement Max/MSP, cette fois réécrit avec nommages et commentaires :

[7] HELLER, Jon. *Pro Oracle SQL Development: Best Practices for Writing Advanced Queries*. Apress, 2019, p. 14.

[8] Les *commentaires* sont une fonctionnalité proposée par tous les langages de programmation modernes. Ils consistent en une convention syntaxique signalant qu'une portion de code donnée doit être ignorée par le compilateur ou l'interpréteur ; le programmeur s'en sert pour documenter son travail au sein de son programme.

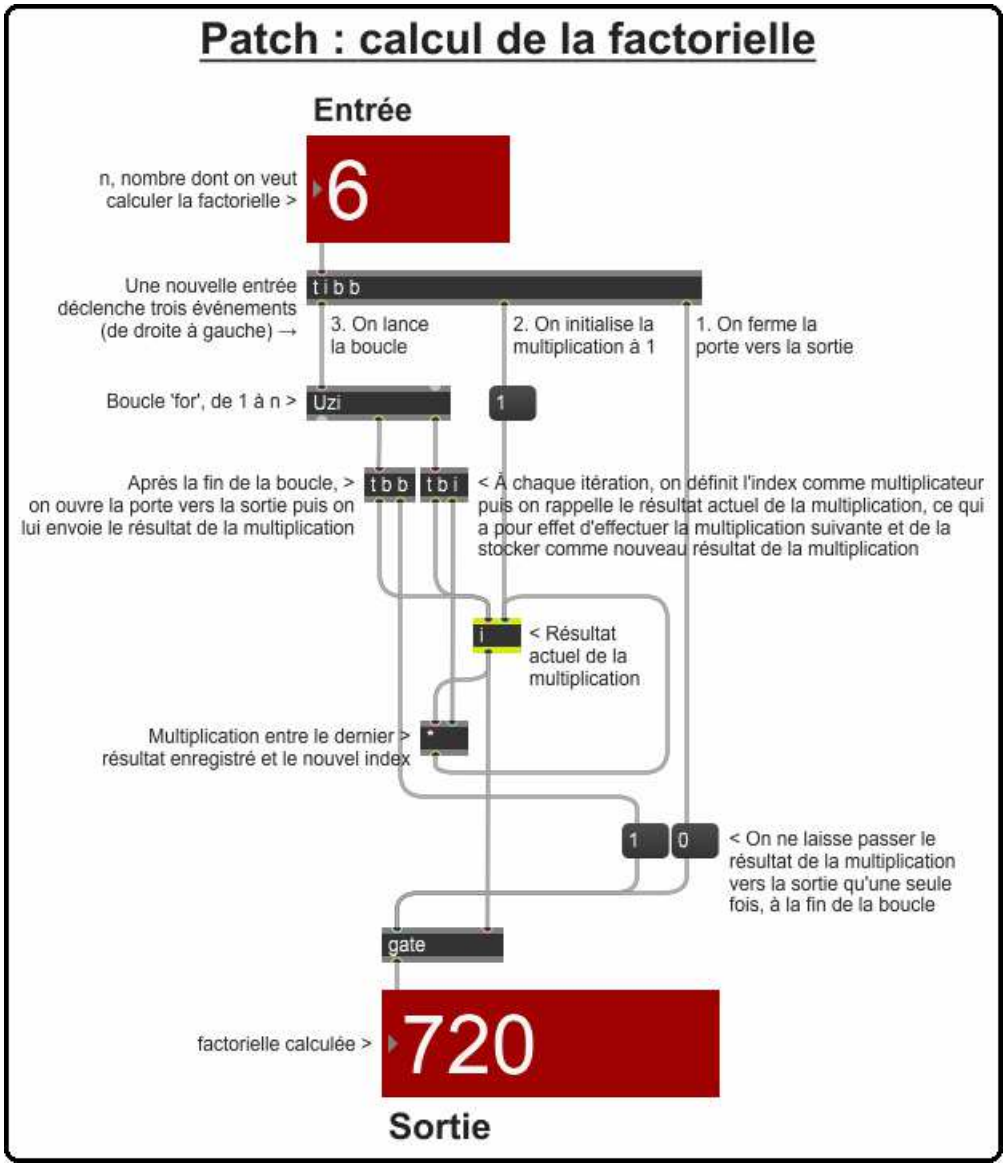


Figure 2 : Variables et commentaires dans un patch Max/MSP

Cet exemple utilise l'environnement Max/MSP, mais les mêmes considérations s'appliquent à OpenMusic ou à d'autres logiciels analogues.

• **Faible lisibilité et faible réutilisabilité.** Les exigences de lisibilité et de réutilisabilité sont au cœur de toute technique sérieuse de programmation. Les environnements graphiques, avec leur présentation non linéaire, leur verbosité et leurs commentaires encombrants ou absents, ne peuvent y répondre.

Il sera aisé de se convaincre du manque de lisibilité des patchs en demandant à un utilisateur aguerri de Max/MSP quelle tâche réalise le patch non commenté de la Figure 1, sans lui donner d'autre indication que le patch lui-même[9].

Une fois terminées, les fonctions programmées sont très difficiles à modifier ; il est plus rapide de recommencer du début. Elles resservent donc très rarement d'un projet au suivant.

• **Problèmes liés à la présentation.** De plus, s'il s'efforce malgré ces limitations de rendre ses patchs les plus lisibles possible, le programmeur devra passer un temps considérable à organiser son travail visuellement, à aligner les éléments, à faire passer les connexions aux bons endroits (voir Figure 3 : Travail supplémentaire pour l'organisation visuelle).

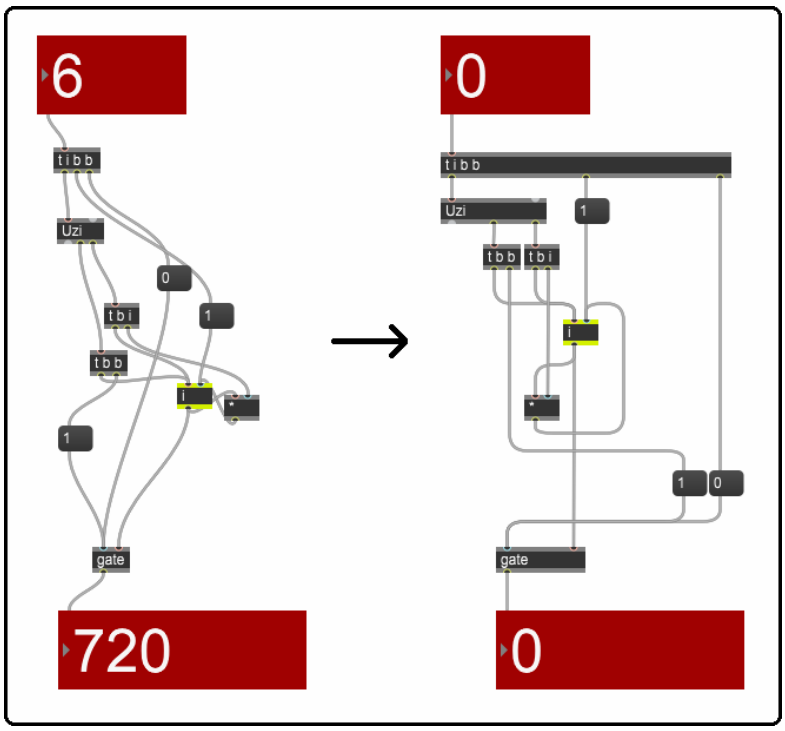


Figure 3 : Travail supplémentaire pour l'organisation visuelle

• **Rigidité structurelle.** Face à l'environnement proposé dans un système de programmation graphique, l'utilisateur se voit contraint d'adopter le mode de pensée prévu pour lui par ses concepteurs. En particulier, il doit considérer tout algorithme comme la gestion de flots de données entre des objets.

[9] Cette tâche relève de la *rétro-ingénierie*, un domaine qui ne devrait pas intervenir lorsque qu'un programmeur relit un vieux travail ou tente de modifier une création d'un collègue pour son propre usage.

Si ce paradigme est absolument pertinent pour des tâches simples (succession d'opérations élémentaires sur un jeu de données) ou pour des problèmes relevant du traitement du signal, il n'en va pas de même de tous les algorithmes que l'on peut vouloir mettre en pratique en composition.

Ainsi, dans un environnement graphique, la mise en place d'un simple algorithme de tri ou de retour sur trace, s'il n'est pas nativement implémenté, peut nécessiter de concevoir un patch extrêmement complexe – voire être quasiment impossible.

Au travers de cette liste, nous avons montré la formidable perte de temps, imputable à de nombreux facteurs, que représente l'usage d'un langage de programmation graphique pour la réalisation de tâches algorithmiques. Vu sous l'angle des avantages et inconvénients pour un compositeur novice en programmation, le temps certes considérable économisé en se dispensant d'apprendre *une fois* un langage de programmation se trouve finalement largement dépassé par le temps qu'il perdra *chaque jour* en programmation, en débogage, et en réécriture lorsque les structures construites par lui s'avèreront impossibles à relire et à réemployer.

c) Des raisons historiques devenues obsolètes

L'hégémonie paradoxale des environnements de programmation graphique en CAO tient à des causes historiques. Avant l'apport de musiciens comme J. Cage ou I. Xenakis, d'une part, et d'autre part le développement de la puissance des premiers ordinateurs, aucune raison ne poussait un compositeur à s'intéresser à la programmation informatique dans le cadre de sa pratique musicale.

Les premiers outils d'algorithmique à destination des compositeurs ont donc dû trouver des moyens de vulgariser les concepts de la programmation auprès de ce public encore profane, d'où le recours à des solutions visuelles.

Cependant, cette raison historique semble maintenant dépassée. De plus en plus de jeunes compositeurs sont issus du monde de la musique électronique et familiers de ses outils. La programmation informatique est maintenant enseignée à l'école dans les sections générales, et beaucoup de compositeurs reçoivent un enseignement spécifique de programmation et de musique algorithmique dans le cadre de leur formation. Hors de

la CAO à proprement parler, les logiciels de création sonore comme Max/MSP sont maintenant concurrencés par des outils performants fonctionnant en code textuel[10]. Enfin, la programmation, hier réservée à des spécialistes, est aujourd'hui très facile à apprendre en suivant des formations gratuites et grand public en ligne.

Toutes les réflexions présentées dans cette partie nous poussent donc à affirmer que le paysage actuel de la CAO laisse une place pour un outil spécialisé dans la production de partitions, basé sur un paradigme de programmation textuelle, et disposant d'une représentation des éléments musicaux robuste et hiérarchique.

C. Orientations suivies

Avant de parcourir systématiquement et en détail la bibliothèque et son organisation, nous souhaitons faire état des grandes orientations que nous avons données à ScoreGen.

Ces orientations, qui peuvent être vues comme les réponses que nous avons choisies lorsqu'un choix se présentait à nous lors de la création de la bibliothèque, ont des conséquences à chaque niveau du projet. Elles concernent l'usage assigné à ScoreGen (ce que la bibliothèque est prévue pour faire ou non), la représentation des structures musicales (la façon dont la bibliothèque représente et organise les éléments musicaux), ainsi que le style de programmation adopté (la façon dont le code à proprement parler est écrit).

1. Choix concernant la destination de la bibliothèque

Les premiers arbitrages concernant ScoreGen ont consisté à définir son domaine d'usage, sa finalité et les exigences que ces buts imposaient au code de la bibliothèque.

[10] Ainsi, le logiciel SuperCollider et le standard Web Audio API sont deux excellentes alternatives, uniquement textuelles, aux logiciels graphiques de traitement du signal comme Max/MSP.

a) Interopérabilité

La première exigence assignée à la bibliothèque est celle de l'interopérabilité[11]. Dans le cas de ScoreGen, le seul point de contact avec l'extérieur est la partition générée. Notre impératif était donc de nous assurer qu'un fichier de sortie de la bibliothèque puisse être également ouvert et « compris[12] » d'une façon identique par des logiciels aussi différents que Finale, Sibelius, MuseScore, Dorico, REAPER, Ableton, etc., mais aussi par les logiciels à vocation similaire comme OpenMusic.

b) Indépendance

Nous avons souhaité que les programmes écrits à l'aide de ScoreGen soient, dans la mesure du possible, indépendants de tout logiciel tiers : les programmes ne doivent pas dépendre des mises à jour de programmes comme le système d'exploitation, OpenMusic, Max/MSP ou Sibelius pour continuer à être fonctionnels. Les seules dépendances consenties car inévitables sont celles propres au langage de programmation utilisé (C++) et celle au langage de sortie (MusicXML). Nous reparlerons plus loin de ces choix, et montrerons comment la seconde peut facilement être contournée dans le cas plausible où le standard MusicXML deviendrait un jour obsolète.

c) Accès de bas niveau aux éléments musicaux

La bibliothèque doit fournir un accès de bas niveau aux éléments constitutifs de la partition. Les objets privilégiés sont donc plutôt des notes, des indications de nuances, des paroles ou des articulations, que des modes, des types d'accords, des processus génératifs ou des modèles d'accompagnement et d'orchestration.

Il est ainsi loisible à l'utilisateur de ScoreGen de définir ses propres macro-structures sans devoir se conformer aux conceptions dictées par l'outil.

Quelques exceptions ont été faites à ce principe pour implémenter des processus ou des structures semblant assez universelles et nécessaires : manipulation intuitive des tableaux, générateurs de hasard, chaînes de Markov génériques, etc.

[11] Capacité d'un système à communiquer avec d'autres systèmes en utilisant des interfaces standardisées, indépendantes du fonctionnement interne du système.

[12] Nous voulons ici non pas que deux logiciels de gravure musicale différents *affichent* une partition générée par ScoreGen à l'identique, mais bien que son contenu sémantique soit analysé de la même façon.

d) Restriction au contenu musical

Enfin, nous avons choisi de limiter les capacités de ScoreGen à un sous-ensemble de ce qui peut figurer sur une partition – en l'occurrence, aux données qui décrivent du contenu musical. Sont ainsi exclus, par exemple, les options de mise en page, de placement et d'espacement, les couleurs, les tailles d'éléments, les sauts de pages, les images, les polices de caractères, etc.

Les textes de tous types ont cependant été inclus, en ce qu'ils décrivent des contenus musicalement pertinents (nuances, indications métronomiques, paroles, modes de jeu).

Ce choix définit ScoreGen comme un moteur de *contenu musical*, celui-ci pouvant ensuite être édité et mis en page sous une forme visuellement satisfaisante grâce à un logiciel de gravure.

2. Orientations concernant la représentation des structures musicales

D'une égale importance apparaissent les choix concernant la façon dont ScoreGen stocke, manipule et organise les éléments musicaux.

a) Une structure arborescente

Le choix le plus lourd de conséquences pour l'implémentation de la bibliothèque est celui de donner à tous les éléments une place hiérarchique dans une arborescence de types de données.

Cette vision arborescente s'oppose à la forme linéaire que prend par exemple un fichier texte décrivant la partition, une séquence de notes, etc.

Tout programmeur lisant ces lignes pensera peut-être que ce choix relève du bon sens, en ce qu'il est non seulement conforme à la structuration du format de sortie MusicXLM, mais aussi à l'exigence de structuration qu'impose un système de données aussi complexe qu'une partition musicale.

Cependant, le formatage des données dans des structures arborescentes n'est pas neutre musicalement et n'a pas que des avantages, ce que nous pouvons illustrer par quatre exemples :

1. Si nous considérons qu'une note est hiérarchiquement élément d'un accord, qui lui-même appartient à une mesure, elle-même issue d'une portée, elle-même appartenant à un instrument qui lui-même prend place dans une partition, alors comment représenter les accords à cheval sur deux portées, habituels aux instruments à clavier ?

→ Impossible dans le cadre de cette vision arborescente, cette notation musicale standard pourrait être représentée si la partition présentait une collection d'accords, placées hors de toute hiérarchie, et dont chaque note serait attribuée à une portée de la partition.

2. Si toute note est hiérarchiquement affiliée à une mesure, alors il est impossible de noter un triolet à cheval sur une barre de mesure – un fait musical rare mais accepté dans les standards de l'édition musicale.

→ Un tel problème n'a pas lieu si les notes de chaque portée sont uniquement placées sur une ligne temporelle, indépendamment de tout lien hiérarchique avec les mesures.

3. Si une liaison est considérée comme hiérarchiquement attachée à une portée avec une position de départ et une position d'arrivée, alors il est impossible de figurer une liaison de harpe courant sur les deux portées.

→ Une structuration uniquement positionnelle (chaque élément étant seulement placé en deux dimensions sur la partition), désastreuse sur le plan de la sémantique musicale, permet toutefois d'éviter cette limitation.

4. Enfin, si chaque note fait partie d'une mesure, elle-même élément d'une portée, alors il est très difficile d'insérer une note ou un groupe de notes au milieu d'une phrase dans une portée. L'insertion doit ou bien ajouter des temps dans la mesure parente, ce qui laisse des vides dans les autres portées si la partition en comporte, ou bien décaler tous les éléments suivants (avec les difficultés qu'on imagine lorsque des éléments ainsi décalés doivent chevaucher une barre de mesure).

→ Cette impossibilité d'envisager simplement une insertion ou une suppression n'aurait pas cours dans un modèle considérant chaque portée comme une liste ahiérarchique d'éléments musicaux juxtaposés.

Nous avons toutefois considéré que les avantages de la structure arborescente dépassaient de loin les limitations qu'elle provoque.

Le détail de l'arborescence choisie, elle aussi très sujette à discussion, sera exposé dans la deuxième grande partie de ce travail.

b) Une structure indépendante du format de sortie

La structure des fichiers MusicXML étant arborescente, nous aurions pu suivre les types d'éléments imposés par les spécifications du langage, pour en faire la base de la représentation interne de la musique de ScoreGen.

Cependant, nous avons préféré construire la hiérarchie qui nous semblait la plus logique et la plus manipulable dans un contexte de programmation algorithmique. Nous avons inventé des conteneurs, restructuré la partition comme un tableau à double entrée (portées et mesures), complètement redéfini la représentation des multiplets qui n'était pas hiérarchique en MusicXML ; nous avons abandonné bien des éléments musicaux qui nous semblaient non pertinents dans ce cadre et avons choisi au cas par cas de représenter chaque type d'élément musical avec les données utilisées par le format MusicXML ou avec des données qui nous semblaient plus pertinentes.

Ce choix d'émancipation par rapport au format de sortie complique considérablement l'export des données manipulées par ScoreGen vers MusicXML, puisque les structures manipulées ne sont pas toujours représentées ou hiérarchisées de la même façon par ScoreGen et dans le fichier généré ; cependant il n'est pas gratuit : en plus d'assurer une manipulation du matériel musical par les programmes beaucoup plus simple, fluide et conforme au bon sens, il permet d'ouvrir la porte à l'ajout de nouveaux formats de sortie (comme le code LilyPond) dans les développements ultérieurs de la bibliothèque.

c) Emploi de structures invisibles sur la partition

Nous avons été conduit à inventer des structures de données intermédiaires destinées à simplifier la manipulation des éléments musicaux. Ainsi, les objets de type « Block » représentent un rectangle de musique, composé d'une ou plusieurs portées et d'une ou plusieurs mesures. Une partition de grandes dimensions peut ainsi être gérée en y découpant des blocs (sous-groupes d'instruments, sections temporelles). Invisibles sur la partition générée, les blocs sont un moyen précieux de structurer la partition.

Les objets de type « Group », permettant de manipuler des séquences monodiques, ont la même vocation. Ils peuvent être construits et manipulés hors de la partition, puis devenir le contenu d'une mesure ou être employés n'importe où dans la partition.

d) Usage systématique de la dérivation

Nous avons considéré que certains éléments musicaux avaient suffisamment de points communs pour être décrits comme des avatars différents d'une même entité générique. Ainsi, nous avons décidé de considérer les accords, les notes et les silences comme plusieurs versions d'un même type d'objets ; en effet, la seule différence formelle entre ces trois types d'éléments est le nombre de notes qu'ils portent (plusieurs, une, zéro), et la capacité à recevoir des articulations (oui pour les accords et les notes, non pour les silences).

Ce concept, implémenté en utilisant la possibilité de dérivation des classes en C++, est un choix structurant pour la bibliothèque. En effet, la formalisation de la parenté entre les trois types d'éléments musicaux cités ci-dessus (par dérivation d'une même classe de base abstraite) permet d'envisager des collections d'objets contenant possiblement des silences et des notes, ou de définir des fonctions s'appliquant indifféremment à ces trois types d'objets.

La même idée a présidé au fait de considérer, par exemple, les notes, les silences, les accords, mais aussi les mutiplets et les groupes, comme relevant d'un même type encore plus général (éléments musicaux plaçables sur une unique portée et possédant une durée), figuré dans le code par le type « Element ».

e) Usage d'un langage descriptif sur mesure

Enfin, nous avons choisi de permettre la création de n'importe quel élément musical *via* un langage concis basé sur les chaînes de caractères. En effet, il aurait été fastidieux pour l'utilisateur de la bibliothèque de devoir créer chaque élément séparément en utilisant la syntaxe du constructeur en C++.

Nous avons donc associé à chaque type d'objet une fonction de création rapide, préfixée de la lettre q (« quick »). Ainsi, la création d'une blanche pointée sur *do* avec un accent et une nuance *fortissimo* avec ScoreGen peut être menée à bien de deux façons, la première conforme à la structure interne des données, la seconde concise :

```
Note^ note1 = gcnew Note(
    gcnew Rhythm(RhythmFigure::_half, 1),
    gcnew Pitch('C', 0, 4),
    Articulation::accent,
    gcnew array<Words::Words^, 1>{gcnew Words::Dynamic("ff")}
);

Note^ note2 = qNote("h.@|c4<!ff>");
```

La deuxième manière permet d'illustrer non seulement la concision de ce langage descriptif, mais aussi son caractère mnémotechnique. En effet, entre les guillemets, il faut lire : h (half note = blanche), . (pointée), @ (accent), puis après le séparateur : c (*do*), 4 (4ème octave), < !ff> (texte entre chevrons ***ff***, point d'exclamation pour le type de texte « nuance »).

3. Orientations concernant la programmation

Au-delà des choix concernant les objectifs de ScoreGen et sa représentation des données musicales, certains arbitrages ont dû être rendus sur des questions propres à la programmation elle-même. Ils concernent les langages employés et certains choix d'écriture du code.

a) Une bibliothèque écrite et utilisable en C++

L'usage du C++ comme langage d'écriture de la bibliothèque tient aux caractéristiques de ce langage.

Tout d'abord, il s'agit d'un langage compilé, ce qui assure une rapidité d'exécution maximale pour les programmes conçus avec ScoreGen.

D'autre part, il s'agit de l'un des langages les plus largement implémentés, compatible avec toutes les architectures, très largement connu et documenté. Il existe des ponts pour utiliser une bibliothèque écrite en C++ avec de nombreux autres langages existants.

Enfin, le C++ est orienté objet, ce qui est particulièrement adapté pour une bibliothèque visant entre autres à définir un grand nombre de types de données.

b) Une sortie au format MusicXML

Le standard MusicXML a été choisi pour sa large prise en charge, en écriture comme en lecture, par les différents logiciels capables d'afficher ou de jouer des partitions.

De plus, MusicXML est riche, voire complet : il permet de représenter une très grande partie des notations standard, qu'il s'agisse d'articulations, de lignes, de quarts de tons ou de textes.

Enfin, le code MusicXML est conçu pour être à la fois simple à générer à l'aide d'un programme (ce que réalise ScoreGen) et simple à comprendre pour un lecteur humain.

c) Des noms en anglais

Conformément à l'usage qui prévaut en programmation, nous avons nommé tous les éléments musicaux et tous les autres objets nommés (variables, fonctions, classes…) en anglais. Nous avons en revanche conservé les commentaires du code en français, considérant que ceux-ci s'adressaient, pour l'instant, uniquement à des lecteurs francophones – et quitte à en réaliser une traduction à l'occasion d'une relecture générale du code source.

d) Une majorité d'objets non mutables

Après plusieurs expérimentations, nous avons résolu de rendre la plupart des éléments musicaux non mutables. Cela signifie qu'une fois créés, les objets ne peuvent plus subir de modification : impossible de changer *a posteriori* la durée d'une note ou sa hauteur.

Ce choix permet d'assurer la cohérence des partitions générées. En effet, il évite par exemple qu'une note déjà placée dans une mesure ne change de durée et ne compromette la cohérence entre le contenu de la mesure et sa signature rythmique.

L'utilisateur souhaitant modifier un élément existant peut toutefois y parvenir en créant un nouvel élément muté grâce aux constructeurs dédiés dans chaque classe.

Bien entendu, les objets conteneurs (comme un bloc de musique de type « Block ») restent mutables : il est possible d'y insérer une mesure ou d'y ajouter une portée, de changer le contenu d'une mesure, etc.

e) Développement d'un test unitaire

La bibliothèque est accompagnée d'un test unitaire, qui vise à assurer le bon fonctionnement de toute l'architecture au-delà des modifications pouvant lui être apportées au cours du temps.

Le lancement du test unitaire crée des objets tests de tous les types et vérifie qu'ils se comportent comme prévu.

Le test unitaire fourni tentera, par exemple, de créer des notes portant toutes les nuances, de créer des liaisons de longueur zéro, de placer divers éléments dans un multiplet (comme des appoggiatures ou d'autres multiplets), etc.

Dans cette première partie, nous avons défini ScoreGen, discuté de sa nécessité dans le panorama des solutions existantes et montré les orientations fixées pour sa réalisation. Nous allons maintenant étudier la bibliothèque plus en détail et exposer précisément sa structure et son fonctionnement.

II. Présentation détaillée de la bibliothèque et de son organisation

Nous proposons maintenant d'entrer dans le détail de l'organisation et du fonctionnement de la bibliothèque ScoreGen. Pour cela, nous suivrons un chemin permettant d'appréhender cet ensemble complexe : nous allons tout d'abord montrer une vue générale de la bibliothèque, avec ses fichiers, ses espaces de noms et sa structure générale ; puis, nous opérerons un « zoom[13] » sur l'architecture choisie pour l'arborescence des éléments musicaux. Ce double parcours nous donnera l'occasion d'un voyage dans l'intégralité de la structure de ScoreGen, et d'une description exhaustive des classes qui la composent.

A. ScoreGen : architecture générale

La bibliothèque ScoreGen suit une architecture à plusieurs niveaux superposés. La Figure 4 montre un aperçu de cette structure. L'objet de cette partie est d'en expliquer l'organisation et le fonctionnement.

[13] La description des éléments musicaux n'est, nous allons le voir, qu'une partie de ScoreGen. Après la description générale de la bibliothèque, l'examen détaillé de l'arborescence musicale représente donc un focus sur une partie du diagramme général.

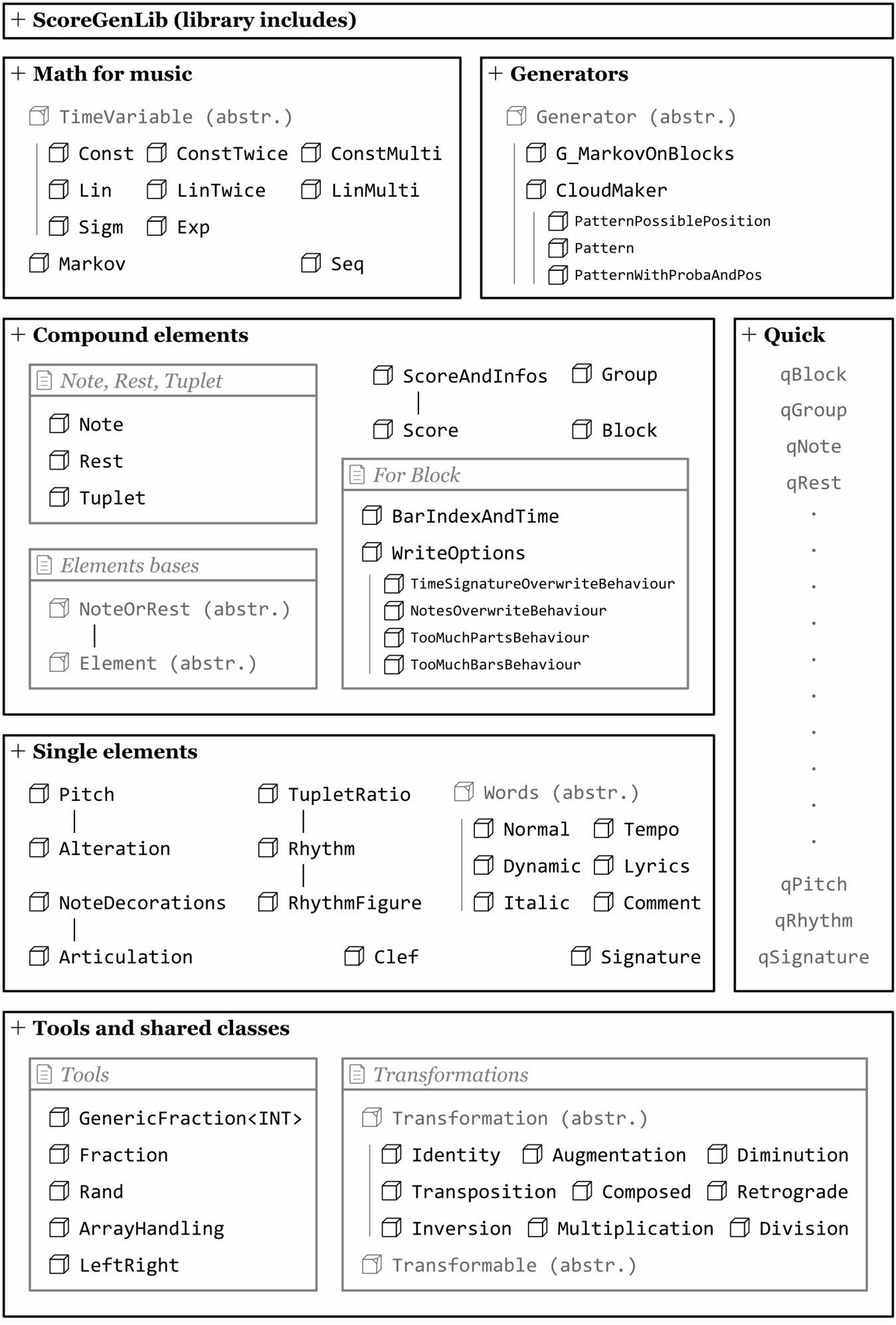


Figure 4 : Structure de ScoreGen

Le schéma présenté se concentre sur la structure générale de la bibliothèque. Il montre la hiérarchie des grandes parties qui la composent, ainsi que son organisation en fichiers lorsque celle-ci ne suit pas celle des classes. En revanche, il ne rend pas compte de la hiérarchie d'héritage et de dépendance entre les classes ; ces points seront schématisés et expliqués lorsque nous aborderons l'arborescence des éléments musicaux.

La hiérarchie des espaces de noms (*namespaces*) employés n'y est pas non plus indiquée ; elle ne sera pas étudiée dans ce travail, car elle n'est pas essentielle à la compréhension de la bibliothèque.

Voici comment lire le schéma ci-dessus :

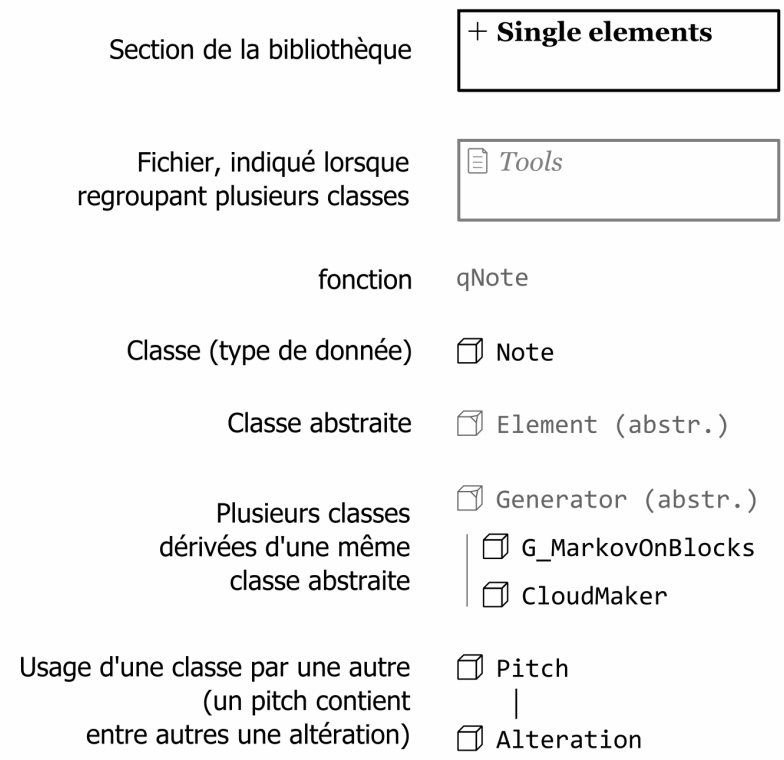


Figure 5 : Légende du schéma de structure ci-dessus

Commençons par commenter la présentation du schéma de structure dans le but de rendre sa lecture plus aisée. Nous avons placé en bas du diagramme les éléments de bas niveau, c'est-à-dire les types de données simples. Plus les éléments sont placés haut dans le diagramme, plus ils représentent des objets complexes. En toute logique, les éléments situés dans le haut du schéma utilisent les éléments simples ; ainsi, la définition d'une *note* (dans la section « Compound elements ») fait appel au type *pitch* qui apparaît plus bas (section « single elements »).

Cette représentation est cohérente avec l'organisation réelle de la bibliothèque. Celle-ci définit en effet un arsenal de types de données plus ou moins complexes, en général basés sur des types de données plus simples. La finalité de cet empilement de définitions est d'aboutir à la caractérisation complète du type *Score* (« partition »), visible au milieu de la section « Compound elements ».

Maintenant que ce choix d'organisation du schéma a été éclairci, étudions un à un et en partant du plus bas niveau les sept grands blocs qui structurent ScoreGen.

1. *Tools and shared classes* : Outils et classes partagées

Nous avons regroupé au premier niveau de ScoreGen une collection d'outils génériques et élémentaires ayant la particularité de servir en de nombreux endroits de la bibliothèque. Ce premier étage de la structure de la bibliothèque permet à tous les composants situés plus haut dans l'architecture d'utiliser les fonctions qu'il expose.

Le fichier *Tools* (Outils) regroupe les objets non spécifiquement musicaux. Il contient par exemple des classes dédiées :

- aux calculs mathématiques sur les fractions, utilisées pour les multiplets, les signatures rythmiques, les durées musicales, les augmentations et diminutions rythmiques ;
- à la génération de nombres aléatoires, utilisée en de nombreux points de la bibliothèque ;
- à la manipulation fluide des tableaux.

Le fichier *Transformations* décrit quant à lui différentes modifications abstraites appliquables à des objets musicaux. Ce dernier fichier a pris place au plus bas étage de la bibliothèque car les transformations définies sont applicables à de très nombreux objets situés plus haut dans la hiérarchie des classes.

2. *Single elements* : Composants élémentaires

Ce second niveau, superposé au premier, définit tous les objets musicaux descriptibles sans mise en œuvre de hiérarchies complexes. Il comporte les types de données suivants :

• **Articulation** et **NoteDecorations.** Les *articulations* représentent les symboles pouvant être attachés aux notes et accords, comme un accent, un point de staccato ou un effet *doit* de trompette. Elles sont en nombre limité et correspondent à une sélection non exhaustive de symboles employés de manière standard dans l'édition musicale.

Les *décorations*, quant à elles, représentent un *ensemble* d'éléments attachés à une note ou un accord : une ou plusieurs articulations et/ou liaisons.

• **Alteration** et **Pitch.** Les *altérations* correspondent aux modificateurs de hauteur comme dièse, bémol, bécarre, demi-dièse, etc. Elles sont limitées à des valeurs entre -2 (double bémol) et +2 (double dièse), et acceptent les quarts de tons[14].

Les *pitches* (« hauteurs de notes ») représentent toutes les hauteurs de notes qu'il est possible de rencontrer dans une partition. Cette classe est plus complexe qu'il n'y paraît : elle gère plusieurs représentations différentes des hauteurs (par nom de note ou en valeur MIDI) et permet aussi d'employer des hauteurs indéfinies de percussions caractérisées par une position verticale sur une portée à *n* lignes.

• **Clef.** Cette classe permet de représenter les clés présentes au début des portées. Elles sont en nombre limité et couvrent toutes les clés communes, y compris les clés de percussions et les clés octaviantes.

• **RhythmFigure**, **Rhythm** et **TupletRatio.** Les *figures rythmiques* correspondent aux figures de notes et de silences élémentaires que l'on peut rencontrer sur une partition moderne, de la sextuple croche à la double note carrée (toutes valeurs non pointées). Les figures rythmiques disposent d'une durée propre.

[14] Les altérations microtonales plus précises que les quarts de tons n'ont pas été implémentées pour l'instant. Ce choix, justifié par l'absence de notation standard pour les représenter et leur mauvaise prise en charge dans les logiciels de gravure musicale, fera peut-être l'objet d'une révision, le MusicXML acceptant tous les types de micro-altérations.

Un *rythme* est l'association d'une figure rythmique et d'un certain nombre de points de prolongement (zéro à quatre). Les rythmes disposent eux aussi d'une durée propre.

Un *ratio de multiplet* est une modification temporelle par laquelle une partie des éléments musicaux sont assignés à un temps fractionnaire. Elle est caractérisée par un rythme de base (par exemple : croche), un nombre normal (par exemple 2) et un nombre effectif (par exemple 3). L'exemple donné correspond au ratio « triolet de croches », sans présupposer du contenu rythmique de ce triolet (trois croches, un silence pointé, etc.).

• **Signature.** Cette classe représente une *signature rythmique* associée à une mesure. Elle prend en charge les signatures simples comme 3/4 ou 6/8, mais aussi les mesures composées (5+9/16) et symboliques (C ou C barré).

• **Words.** Cette classe abstraite permet de décrire n'importe quelle indication textuelle. Les textes sont instanciés à l'aide des classes dérivées, représentant respectivement un texte générique, une dynamique, un texte en italique, une indication de tempo ou un commentaire.

Chaque type d'indication textuelle peut être créée conformément à ses spécificités ; par exemple, la classe dérivée *Dynamic* donne accès à tous les textes de nuances communément utilisés (aussi bien qu'à la possibilité d'utiliser un texte personnalisé comme ***sfffzmp*** ou ***ppf***) ; les textes de tempo ne se limitent pas à une indication en chaîne de caractère mais ont une valeur sémantique (association d'une vitesse en BPM à une figure rythmique ou à un rythme).

3. *Compound elements* : Composants complexes

Les premiers éléments qui composent la deuxième couche de ScoreGen ne présentent aucune structuration des éléments musicaux : ils sont des composants isolés, et même si certains sont définis en relation avec d'autres (par exemple *Pitch* et *Alteration*), ils ne construisent aucune hiérarchie dans les objets musicaux.

À l'inverse, les *éléments complexes* qui constituent le troisième étage dessinent la structure hiérarchique de la partition. Ils utilisent les éléments définis à l'étage inférieur et entretiennent entre eux des rapports hiérarchiques.

• **Element** et **NoteOrRest** (classes abstraites)**.** Les *éléments* sont des objets qui comme les notes, les silences ou les multiplets, peuvent prendre place dans une portée et possèdent une durée.

Les objet de type *NoteOrRest* en sont une spécialisation pour les objets qui consistent en une unique note ou un unique silence ; la présence de cette classe artificielle dans ScoreGen s'explique par la similitude entre une note et un silence, différenciés seulement par leur nombre de *pitches* (zéro pour le silence).

• **Rest.** Les *silences* sont définis comme des *NoteOrRest* sans autre attribut.

• **Note.** Les *notes* sont définies comme des *NoteOrRest* comportant des *pitches* et d'éventuelles *décorations*. Une seule classe a été créée pour décrire les notes et les accords, ces deux objets musicaux étant différenciés seulement par leur nombre de hauteurs (une pour les notes, plus d'une pour les accords) et cette différentiation n'ayant jamais de conséquence sur les autres caractéristiques des objets.

• **Tuplet.** Les *multiplets* sont définis comme un *ratio de multiplet* (*TupletRatio*) et un contenu (liste d'éléments musicaux de type *NoteOrRest*).

• **Group.** Les *groupes* sont des structures non visibles sur la partition. Il s'agit de séquences d'éléments musicaux tenant sur une portée, représentant par exemple une figure musicale ou le contenu d'une mesure.

• **Block.** Les *blocs* sont un autre type de structures non visibles sur la partition. Elles consistent en une grille bidimensionnelle mutable et organisant spatialement tous les autres types d'éléments dans un « bloc de musique » pouvant comporter plusieurs portées et plusieurs mesures.

• **Score** et **ScoreAndInfos.** Les *partitions* dominent la hiérarchie des éléments musicaux. Il s'agit de structures comparables aux *blocs* mais non mutables ; elles sont donc créées à la fin de la synthèse et juste avant l'export vers MusicXML. Contrairement aux *blocs*, elles représentent plus qu'un contenu musical : elles peuvent

aussi spécifier des clés pour la lecture de chaque portée. La classe *ScoreAndInfos* encapsule une partition et des métadonnées sous forme de texte ou d'image, fournissant l'une des façons de déboguer un programme, de renseigner sur des choix effectués par l'algorithme ou de visualiser l'algorithme appliqué.

4. *Quick* : Fonctions de création rapide

Les deuxième et troisième étages de ScoreGen, qui ont fait l'objet des deux dernières parties de cette présentation, sont bordés d'une section nommée Quick (« Rapide »). Cette section transversale, qui ne contient que des fonctions préfixées du caractère *q*, donne un accès alternatif aux constructeurs de la plupart des éléments musicaux grâce à des *raccourcis* écrits dans un langage dédié.

Voici quelques exemples démontrant la concision que permettent ces fonctions :

Objet à créer	**Syntaxe en *q***
Indication de mesure en 9/8	`qSignature("9/8")`
Succession de trois silences	`qGroup("q q q")`
Deux noires de pitch MIDI 60.5 (do ½ dièse)	`qGroup("q|60.5 q|60.5")`
Triolet de croches contenant des demi-soupirs	`qTuplet("[3:2 e e e]")`
Pause pointée	`qElement("w.")`
Noire de pitch C4, *pianissimo*	`qNote("q|C4<!pp>")`
Demi-soupir triplement pointé *dolcissimo*	`qRest("e...<:dolcissimo>")`

Ce langage concis sert *a priori* surtout lorsqu'une partie du matériel à manipuler est connue du programmeur et non généré par un algorithme. Toutefois, rien n'empêche de l'utiliser pour générer du matériel musical en construisant algorithmiquement les chaînes de caractères correspondantes.

5. *Math for music* : Outils mathématiques

Les sections précédentes réalisent une description complète d'une partition en termes de types de données. Elles suffisent à réaliser des partitions algorithmiquement et à les exporter en MusicXLM. ScoreGen aurait donc pu s'arrêter à ce niveau.

Toutefois, il nous a semblé bon d'ajouter encore deux sections visant à faciliter la création de pièces ou de matériaux musicaux algorithmiques. La première, *Math for music* (Mathématiques pour la musique) présente trois outils hautement génériques et utiles dans des contextes variés de composition :

• **TimeVariable.** Presque tout processus de CAO régi par des règles fixes peut aussi à profit être appliqué avec des règles variant au cours du temps. Ainsi, si un programme est conçu pour générer 500 doubles croches aléatoires dans l'octave C4-C5, il peut être intéressant et musicalement pertinent de vouloir le modifier pour que l'ambitus, occupant cette octave au début du processus, s'ouvre au long de la génération et finisse par couvrir une plage plus ou moins grande que l'octave initiale.

Nous pourrions aussi vouloir appliquer une telle évolution à la densité de notes par rapport aux silences au cours du temps, vouloir explorer des modes aux nombres de notes croissants au cours d'une série d'accords générés algorithmiquement, ou encore faire évoluer les poids d'une chaîne de Markov.

Enfin, nous pourrions vouloir qu'une évolution sur un paramètre suive un cheminement progressif et linéaire, ou qu'elle se conforme à un dessin plus complexe.

Ce type de procédé, connu sous le nom d'*automations* dans le cadre de la musique életronique et des DAW, est universel en CAO ; de plus, la programmation de certains dessins simples comme les fonctions linéaires par morceaux peut paraître fastidieuse à certains programmeurs. C'est pourquoi ScoreGen propose des classes dédiées à cet usage.

Chacune des classes dérivées de la classe abstraite *TimeVariable* propose un type d'évolution temporelle spécifique et paramétrable. Elles ont en commun de permettre d'accéder à la valeur du paramètre au cours du temps ou bien selon une variable temporelle allant de 0 à 1, ou bien selon un index dans un intervalle – une fonctionnalité

précieuse dans le cadre de la CAO où la plupart des processus interviennent sur des listes discrètes d'objets.

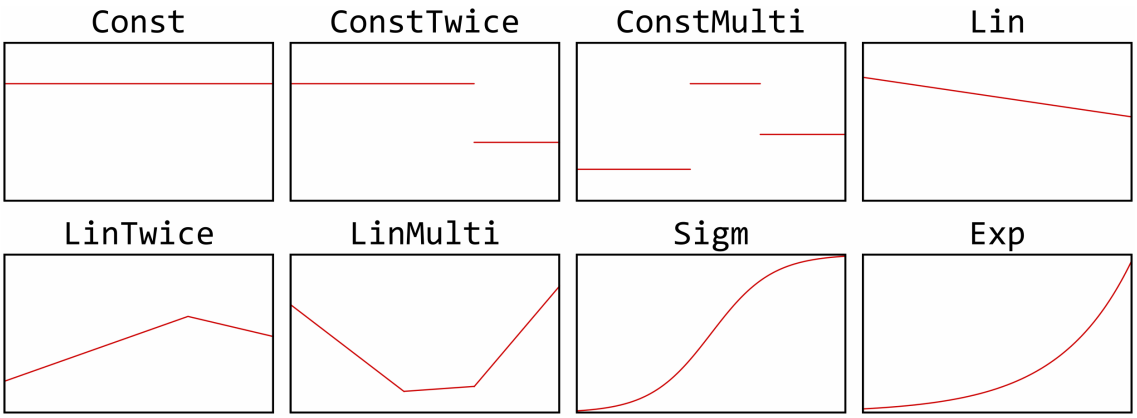


Figure 6 : Types de variables temporelles supportées par ScoreGen

Même si les fonctions linéaires par morceaux (*LinMulti*) peuvent approcher avec la précision souhaitée n'importe quelle fonction, l'utilisateur est libre de dériver lui-même la classe *TimeVariable* pour définir ses propres types d'automations.

• **Markov.** La bibliothèque prend aussi en charge un autre classique des algorithmes de CAO, à savoir les chaînes de Markov. La définition précise des chaînes de Markov dépasse le cadre de ce travail et est largement documentée par ailleurs, mais nous proposons toutefois ici un aperçu de leur fonctionnement.

Les chaînes de Markov permettent de générer une succession d'éléments se déroulant dans le temps. Les éléments en question sont fixés par l'utilisateur : il peut s'agir, par exemple, de quatre accords[15]. Pour l'exemple, choisissons C, F, Dm7 et G.

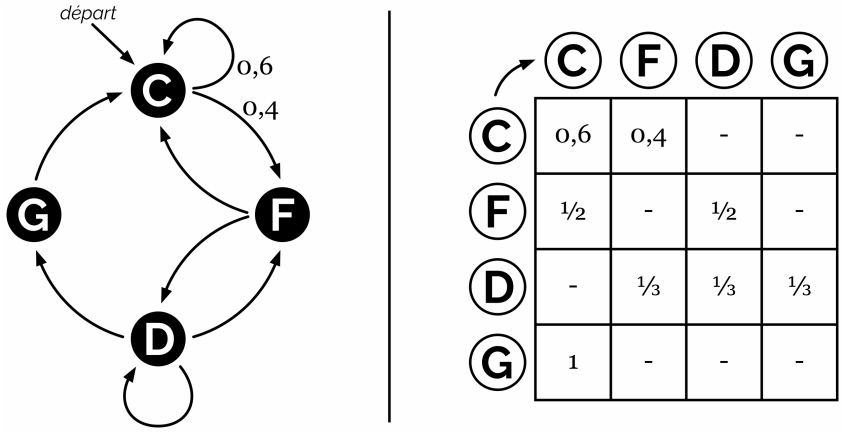
Une fois le premier accord de la série fixé par l'utilisateur (C par exemple), la chaîne de Markov se charge de générer les accords suivants. Pour cela, l'utilisateur définit un ensemble de probabilités dépendant du dernier accord déjà généré. Nous pourrions fixer ceci :

- Si le dernier accord tiré est C, on reste tantôt sur C (avec une probabilité de 60 %), tantôt on va sur F (dans les 40 % de cas restants) ;

[15] Dans ScoreGen, les chaînes de Markov sont définies de façon générique. Il est donc possible de les appliquer à des notes, à des rythmes, à des blocs, ou à tout autre type d'éléments.

- Si le dernier accord tiré est F, tantôt on revient sur C, tantôt on va sur Dm7 (avec une probabilité égale) ;
- Si le dernier accord tiré est Dm7, on reste sur Dm7 dans un tiers des cas, on va sur F dans un autre tiers des cas, et sinon on va sur G ;
- Si le dernier accord tiré est G, on revient toujours sur C.

La chaîne de Markov ci-dessus peut être symbolisée par un graphe ou par une matrice :



	C	F	D	G
C	0,6	0,4	-	-
F	½	-	½	-
D	-	⅓	⅓	⅓
G	1	-	-	-

Figure 7 : Représentation sous forme de graphe ou de matrice

Le graphe montre les quatre différents états possibles et figure le passage de l'un à l'autre par des flèches ; nous avons omis les probabilités d'emprunter l'une ou l'autre lorsque toutes les probabilités au départ d'un accord étaient égales.

La matrice, plus synthétique mais moins facile à lire, présente sous forme de tableau toutes les probabilités de passer d'un état à l'autre ; on peut ainsi lire dans la deuxième ligne que lorsque le dernier accord tiré est un F, le suivant sera un C ou un Dm7 à probabilités égales.

Voici maintenant une partition générée en utilisant ScoreGen. La chaîne de Markov ci-dessus a été générée en utilisant la classe *Markov*, et à chaque accord a été attribué un bloc à deux voix. Les blocs ainsi obtenus ont été mis bout à bout, exportés en MusicXML et ajoutés à une partition (l'ensemble occupe une dizaine de lignes de code) :

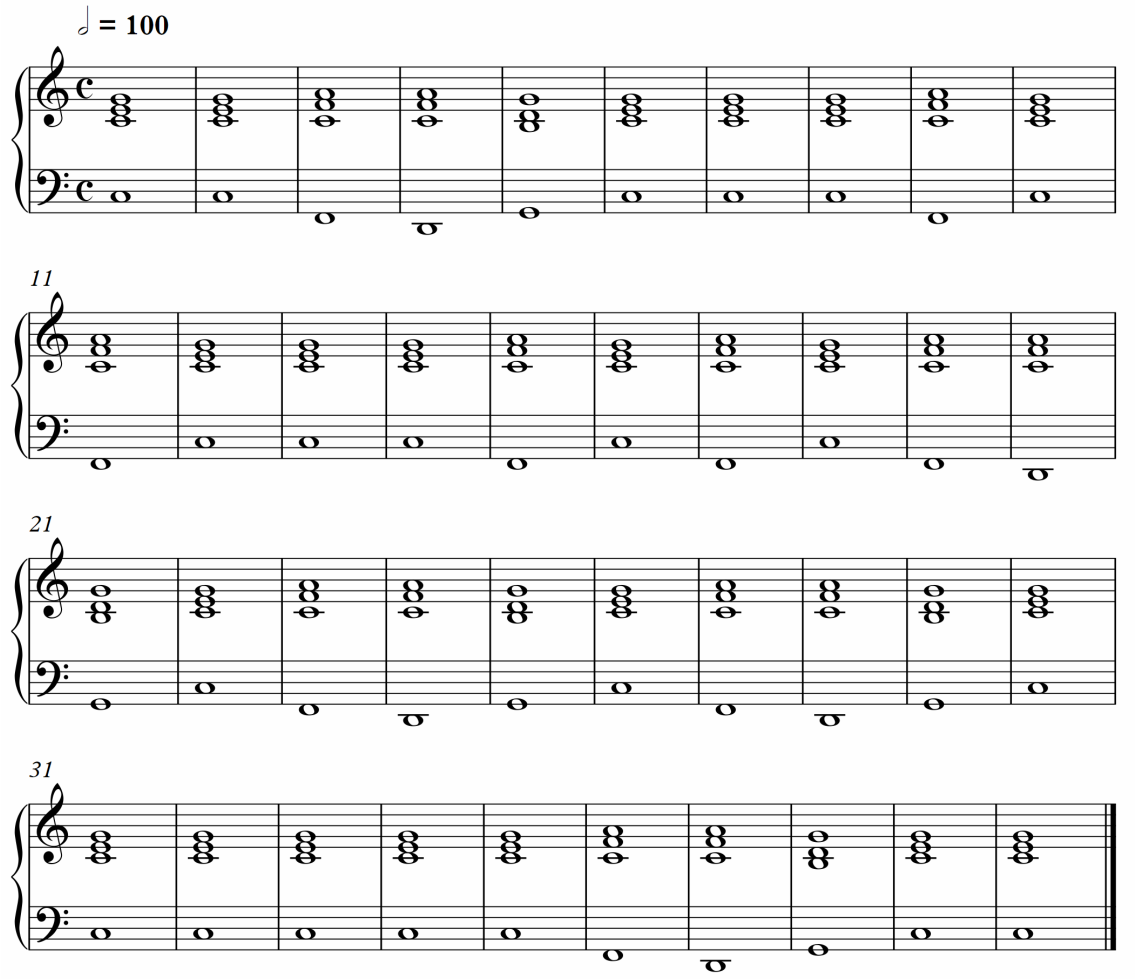


Figure 8 : Suite d'accords générée par ScoreGen en utilisant la chaîne de Markov ci-dessus

Les chaînes de Markov implémentées dans ScoreGen ont été adaptées pour l'usage spécifique qui peut en être fait en CAO. Nous avons donc étendu la définition habituelle de ces objets mathématiques pour permettre de faire varier les probabilités dans le temps. Cette puissante fonctionnalité ouvre des possibilités intéressantes pour générer des successions aux caractéristiques évolutives. Nous en donnerons plus loin un exemple dans un contexte réel de composition d'une pièce pour orchestre.

Signalons enfin que notre implémentation accepte que la somme des probabilités au départ d'un état donné soit différente de 1 ; ainsi l'utilisateur peut employer à sa convenance et sans distinction des probabilités, des pourcentages ou des nombres entiers.

• **Seq.** La bibliothèque fournit encore une classe pour la génération de suites mathématiques entières communes. Cette classe vise à éviter à l'utilisateur de devoir

reprogrammer des suites comme la suite de Fibonacci. Elle ne propose pour l'instant qu'un nombre très limité de suites et doit encore être complétée dans une prochaine version de ScoreGen.

6. *Generator* : Générateur de partitions

La classe abstraite *Generator* représente le plus haut niveau d'abstraction de la bibliothèque ScoreGen.

Un objet de ce type est un *créateur de partition* : il peut à la demande générer un objet de type *Score* ou *ScoreAndInfos*, en associant éventuellement à la composition un nom et une description.

Rien n'oblige à utiliser l'abstraction *Generator* pour créer des partitions avec ScoreGen. Toutefois, cette classe est utile lorsque l'on souhaite créer et étiqueter une série de compositions, utiliser un même programme pour générer des partitions avec des options différentes ou encore créer une méthode générique de création de partitions.

L'exemple fourni dans la bibliothèque, *MarkovOnBlocks*, dérive par exemple la classe *Generator* pour donner une méthode simple et générique de synthèse de partitions entières en juxtaposant des blocs de musique en suivant une chaîne de Markov à coefficients variables dans le temps.

La classe *CloudMaker*, enfin, reprend le même principe de méta-synthèse en proposant un moyen simple et général d'engendrer des nuages polyphoniques à partir de *Patterns* (motifs). Ce générateur permet de fixer des motifs, leur probabilité d'apparition au cours du temps, leurs schémas de transposition au cours du temps et leurs positions possibles dans les mesures.

7. *ScoreGenLib* : #includes de la bibliothèque

Toutes les couches d'abstractions qui composent ScoreGen sont enfin couronnées par un dernier module, dont le rôle est d'inclure tous les fichiers de la bibliothèque.

B. De la note à la partition : hiérarchie des objets musicaux dans ScoreGen

Nous venons de terminer la présentation exhaustive des modules qui composent la bibliothèque ScoreGen, dans son architecture en couches superposées et de complexité croissante. De ce tour d'horizon, un aspect a été seulement effleuré : il s'agit du caractère arborescent du modèle musical qui sous-tend les types de données impliqués.

Nous avons certes évoqué en certains points l'existence de classes dérivant d'autres classes (*Dynamic* de *Words*, ou encore *Note* de *Element*) ; nous avons à plusieurs reprises montré comment un type de données était basé sur un autre (*Tuplet* utilise un *TupletRatio*, et *Block* range des *Element* dans une grille bidimensionnelle). Cependant, nous n'avons pas encore mis en évidence la hiérarchie arborescente que ce double jeu de dérivation et d'inclusion dessine – alors même que la réflexion sur la structure de la partition est au cœur du travail de conception de ScoreGen.

L'objet de cette partie est d'étudier le modèle de représentation des éléments musicaux retenu pour la structuration de la bibliothèque.

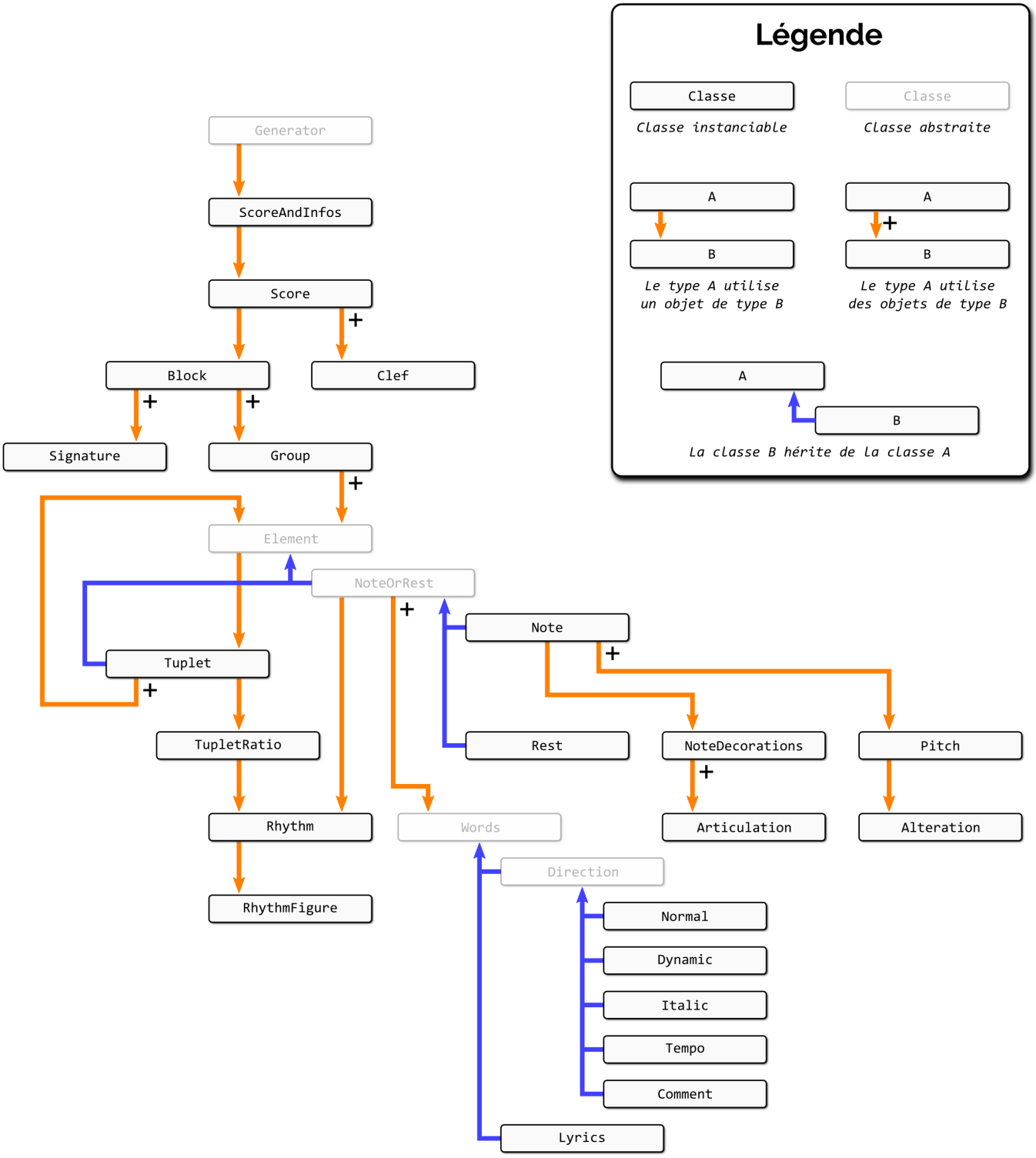


Figure 9 : Arbre des éléments musicaux dans ScoreGen

1. Les classes et leurs relations

Commençons par examiner la légende du diagramme. Celle-ci indique que tous les éléments représentés dans l'arborescence sont des classes, autrement dit des types de données personnalisés que l'utilisateur peut manipuler dans ses programmes. Certaines classes sont « instanciables », ce qui signifie que l'utilisateur peut les utiliser pour créer directement des objets ; d'autres, que nous avons représentées en teintes plus pâles, représentent des types d'objets abstraits que l'utilisateur ne peut créer directement.

Ainsi, si tous les types de textes sont regroupés sous un même type *Words* (en bas du schéma), il n'aurait pas de sens de créer un texte sans préciser s'il s'agit d'une syllabe de paroles, d'une nuance ou d'une indication métronomique. L'utilisateur ne peut donc créer un objet du type *Words* directement : cette classe est abstraite. En revanche, la classe *Dynamic* n'est pas grisée : l'utilisateur peut l'utiliser pour créer un texte de type nuance.

Il en va de même de la classe *Element* : il n'aurait pas de sens de créer un élément « abstraitement » sans préciser s'il s'agit d'une note, d'un silence ou d'un multiplet. Ce type, qui ne sert qu'à implémenter les caractéristiques *communes* à ces trois sortes d'éléments, est donc abstrait.

La deuxième ligne de la légende montre un premier type de relation que deux classes de la bibliothèque peuvent entretenir entre elles : il s'agit de l'*inclusion*, figurée par une flèche orange. Cette notation indique qu'une classe A « fait appel » à une autre classe B. Dans le contexte de cette bibliothèque, cela signifie simplement que les objets de type A « contiennent » un objet de type B.

Ainsi, on peut voir à droite du diagramme que les hauteurs de notes (classe *Pitch*) comportent une altération, et en bas à gauche que les rythmes (éventuellement pointés) contiennent une figure rythmique.

La même notation, assortie d'un signe « plus », signifie que la première classe a recours à une *collection* d'objets de la seconde classe. On pourra se convaincre qu'une partition *Score* contient plusieurs clés (une par portée), qu'un *Block* présente une série de signatures rythmiques (jusqu'à une par mesure) et de groupes (un objet de type *Group*

par case de la grille formée par les portées et les mesures) ou encore qu'un *Tuplet* contient une série d'*Element*.

Enfin, la légende montre un second type de relation que les classes de la bibliothèque peuvent entretenir, à savoir l'*héritage*. On dit que la classe B hérite (ou dérive) de la classe A lorsque les objets de type B peuvent aussi être considérés comme des cas particuliers d'objets de type A ; ou pour l'écrire autrement, si le type A représente un *cas général* des objets du type B.

On voit ainsi sur le diagramme de classes que les *Tuplet* sont un type spécial d'*Element*s, et que les paroles *Lyrics* sont un type de texte *Word*.

Dans certains cas, les classes de ScoreGen présentent plusieurs niveaux d'héritage. Ainsi, si tous les types de texte sont représentés par la classe *Word*, certains sont regroupés dans le type *Direction* (indication). De même, si les notes, les silences et les multiplets sont tous des *Element*, seuls les notes et les silences sont représentés par la classe *NoteOrRest*.

Cette double hiérarchie portée par l'inclusion et l'héritage dessine une structure globalement arborescente, au sommet de laquelle se trouvent la partition *Score* et les deux objets de la bibliothèque qui incluent directement (*ScoreAndInfos*) ou indirectement (*Generator*) une partition.

2. Notes et silences

Passons maintenant en revue les différents objets permettant de définir notes et silences dans ScoreGen. Nous allons donc étudier cette partie du diagramme :

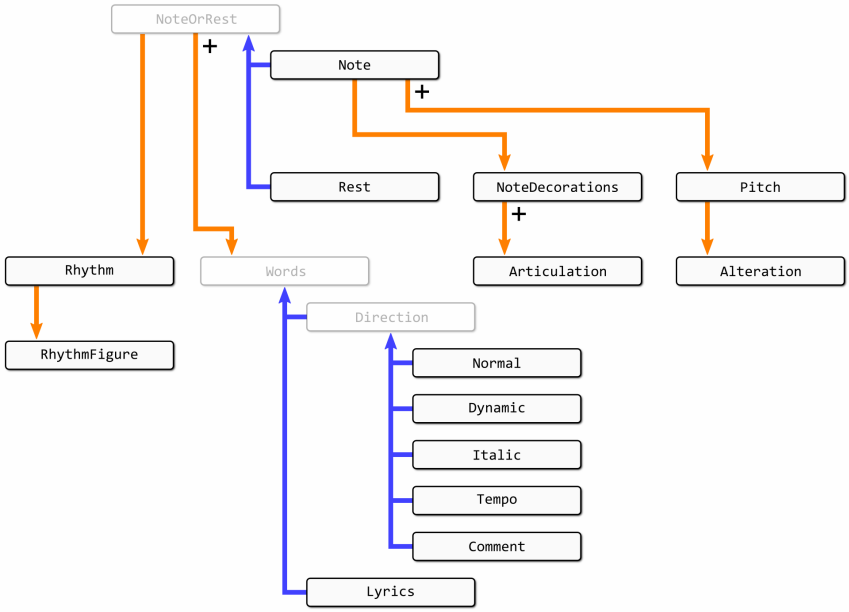


Figure 10 : Types permettant de définir les notes et les silences

Un objet abstrait *NoteOrRest* est défini comme ayant un rythme *Rhythm* et une ou plusieurs indications textuelles *Words*. Si cet objet abstrait est créé grâce à la classe instanciable *Rest* (silence), ces deux éléments suffisent – et en effet un silence ne saurait présenter d'autres caractéristiques ; sinon, l'objet *NoteOrRest* est instancié *via* la classe *Note*.

La classe *Note* fait intervenir plus d'informations que *Rest* : une note peut porter une *NoteDecoration* (articulations et liaisons qui ne sauraient convenir aux silences), et surtout une note doit préciser une ou plusieurs hauteurs de notes (*Pitch*).

Reste à aborder les classes dont nous avons parlé sans détailler leurs inclusions : le rythme d'un objet *NoteOrRest* est lui-même une figure rythmique (*RhythmFigure*) éventuellement accompagnée d'un certain nombre de points de prolongation ; les indications textuelles *Words* peuvent être de six types différents, dont cinq relèvent d'un type dérivé abstrait *Direction* ; les *NoteDecoration* comportent d'éventuelles liaisons et un certain nombre éventuellement nul d'articulations ; les *Pitch*, même lorsqu'ils sont définis directement en utilisant une valeur MIDI, utilisent en interne un nom de note, une octave et une *Alteration*.

3. Éléments

Les notes et les silence étant définis par la classe *NoteOrRest* et ses deux classes dérivées, étudions les « éléments » (class abstraite *Element*), qui représentent indifféremment des notes, des silences ou des multiplets avec leur contenu.

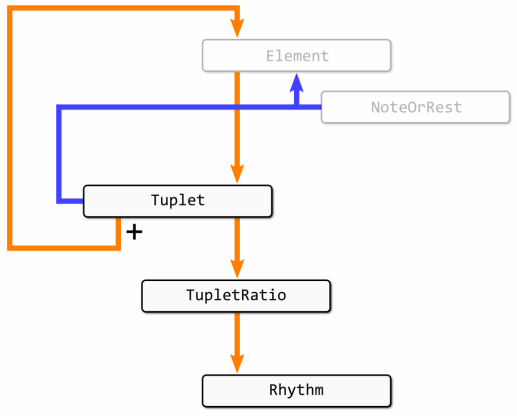


Figure 11 : Types permettant de définir les « éléments »

On voit sur cette portion du diagramme de classes que les éléments peuvent être de deux types. En effet, deux classes héritent de *Element* : la classe *NoteOrRest* et la classe *Tuplet* (multiplet).

NoteOrRest ayant déjà été étudiée, observons la classe *Tuplet*. Un multiplet est composé d'un ratio *TupletRatio*, et d'un contenu (plusieurs éléments *Element*). Ici réside l'une des principales difficultés du diagramme : la structure est *récursive*, puisqu'un multiplet est un élément qui peut lui-même contenir des éléments (par exemple un soupir ou un multiplet imbriqué).

Le *TupletRatio*, quant à lui, est simplement défini pas une valeur rythmique et une fraction (cette dernière indiquant le taux de compression temporelle appliquée par le multiplet). Par exemple, un simple triolet de croche est décrit par la valeur *croche* et la fraction 3/2. Nous pouvons à cette occasion remarquer que la classe incluse dans *TupletRatio* n'est pas *RhythmFigure* mais bien *Rhythm* : il est logiquement possible,

quoique rarissime en pratique, de définir un multiplet sur une valeur pointée (par exemple dans une mesure en 9/8, cinq noires pointées dans 3 noires pointées)[16].

Une dernière flèche du schéma n'a pas encore reçu d'explication : c'est celle qui indique que les objets de type *Element* contiennent un *Tuplet*. Cette affirmation n'est pas conforme à la logique si l'on se rappelle qu'un *Element* peut être aussi bien un multiplet qu'une note ou un soupir. Elle s'explique par le fait que tout élément offre une référence vers son *Tuplet* parent. Lorsque l'élément n'est pas dans un multiplet, la référence est nulle. Cette disposition, si elle complique l'arborescence, a l'avantage de permettre de parcourir l'arborescence des éléments vers le bas ou vers le haut.

4. Structures de haut niveau

Ces éléments définis, il reste à permettre de les ordonner au sein de structures plus vastes.

Nous aurions pu nous contenter de proposer une unique structure, à savoir la partition elle-même. Mais nous avons choisi de proposer des structures de taille intermédiaire entre les éléments et la partition, pour faciliter la manipulation de séquences et de blocs de musique, et rendre *in fine* possible de s'organiser pour construire des partitions de très grandes dimensions.

[16] Cette fonctionnalité est d'ailleurs supportée au moins dans le standard MusicXML, dans le langage LilyPond et dans les logiciels de gravure musicale Sibelius et Finale.

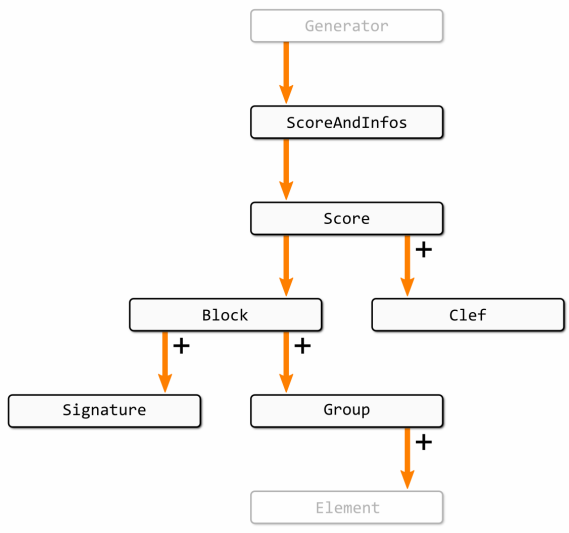


Figure 12 : Structures musicales de haut niveau

a) Groupes sur une portée

La plus simple d'entre elles est le « groupe » : elle représente une simple collection d'éléments musicaux sur une seule portée. Comme indiqué sur le diagramme, les groupes ne contiennent rien d'autre qu'une collection d'éléments. Cependant, la classe *Group* est bien plus qu'un moyen de stocker des éléments : elle offre aussi de nombreux opérateurs et fonctions permettant de réaliser toutes sortes de manipulations. Une fois qu'un objet de type *Group* est créé, il est ainsi possible de calculer sa durée, de le concaténer à un autre groupe, de le multiplier, de le transposer, de le couper à une position temporelle donnée, etc.

b) Blocs en deux dimensions

La structure supérieure au groupe est le « bloc ». Un objet de type *Block* est une grille mutable de *n* portées et de *m* mesures, représentant un « rectangle » de musique. Chaque case de cette grille est logiquement représentée par un objet de type *Group*, d'où la flèche montrant un recours multiple à cette classe. Le bloc tient aussi à jour une collection de signatures rythmiques pour les mesures qui la composent.

Le bloc est pensé comme une structure flexible. Il est possible d'insérer et de supprimer portées et mesures à tout moment. De plus, les blocs proposent les mêmes types de manipulations que les groupes. Enfin, il est possible de coller un bloc à tout endroit d'un bloc plus large, ce qui, en rendant plus aisée une division des tâches algorithmiques

suivant les sections d'un morceau ou les groupes d'instruments considérés, permet d'envisager des créations de grande envergure à l'aide de ScoreGen.

c) Score : la partition dans son ensemble

Tous ces efforts d'organisation des objets musicaux aboutissent à la structure musicale ultime : la partition. Un objet de type *Score* est composé d'un unique bloc de musique et d'une collection de clés (une pour chaque portée).

Contrairement au *Block*, le type *Score* n'offre aucune flexibilité : un objet *Score* est non mutable, c'est-à-dire que son contenu est fixé une fois pour toutes lors de sa création. La création de la partition n'est donc que la dernière étape après toutes les manipulations effectuées avec les structures de plus bas niveau ; elle est en général suivie de l'export de la partition au format MusicXML.

Les deux autres classes présentes au-dessus de *Score* sur le diagramme ont déjà été étudiées plus haut ; elles incluent toutes deux un objet de type *Score*.

5. Éléments absents du diagramme

Le diagramme que nous venons d'étudier partie par partie est conçu pour donner le meilleur aperçu possible de la structuration des objets musicaux dans la bibliothèque ScoreGen. Sa relative complexité nous a dissuadé d'y ajouter certaines informations de structure, que nous allons donc brièvement évoquer maintenant.

Tout d'abord, une large part des éléments musicaux, en plus d'hériter ponctuellement d'un autre élément musical plus général, héritent de la classe *Transformable* qui a été évoquée dans la structure générale de ScoreGen (section *Tools and shared classes*). La fonctionnalité d'*héritage multiple* propre au C++ nous a permis de faire jouer ce double héritage, l'un relevant de la structure hiérarchique des éléments musicaux, et l'autre de l'exigence de généricité dans le programme. Ainsi, chaque type d'objet implémente sa propre fonction de transposition, de rétrogradation, d'augmentation, etc., permettant d'effectuer ces opérations sur tous les éléments musicaux pertinents.

D'autre part, un autre réseau hydrographique souterrain irrigue l'arborescence des éléments : il s'agit de la conversion en MusicXML, qui est elle aussi implémentée par chaque objet. La conversion suit ainsi la hiérarchie naturelle des données, chaque objet appelant la méthode de conversion de ses objets enfants pour effectuer sa propre traduction.

Enfin, l'arborescence représentée ci-dessus est largement complexifiée par un jeu de conversions automatiques entre les types, et dont la représentation impliquerait de multiplier par deux le nombre de flèches indiquant des inclusions. Ce dispositif, extrêmement utile pour une utilisation fluide de la bibliothèque, consiste à fournir des équivalences entre certains types d'objets. Ainsi, un groupe peut être automatiquement converti en bloc à une seule portée et une seule mesure ; une note peut être vue par le programme comme un groupe à un seul élément, une partition comme un objet *ScoreAndInfos* sans informations, un bloc comme une partition où les clés ne sont pas précisées, etc.

Ce fonctionnement permet de ne jamais avoir à convertir manuellement une structure de données en une autre lorsque la conversion est évidente. Grâce à cette fonctionnalité, l'utilisateur peut toujours fournir, par exemple, un *Tuplet* là où une fonction attend un *Block* ou un *ScoreAndInfos* ; le programme se chargera d'effectuer les conversions de façon transparente.

6. Un solfège basé sur la relation

Nous avons déjà montré plus haut que la structuration des éléments musicaux sous forme d'une *arborescence* n'était en aucun cas une évidence.

Pour mieux caractériser le système mis en œuvre dans ScoreGen, prenons un instant pour nous intéresser aux éléments musicaux tels qu'ils sont enseignés par le solfège[17].

[17] Nous devons en effet constater que toute tentative de définition des éléments musicaux et de leur symbolisme entre en concurrence directe avec le système du solfège, puisque celui-ci est l'unique façon de se référer à eux dans le contexte de la didactique musicale.

Traditionnellement, le symbole musical est considéré au travers de deux types d'associations : association sémantique et association spatiale.

La première considère le symbole comme signifiant d'un couple signifiant-signifié : il est destiné à être lu ou exécuté sur un instrument, suivant un code maîtrisé par le lecteur, et possède donc une *valeur signifiante*. Nous pouvons voir cette première association comme une relation verticale, le symbole étant inféodé à une réalité musicale supérieure qu'il décrit et projette sur la partition.

La seconde envisage le symbole de façon spatiale, comme élément d'une succession ou d'un groupe. La relation ici considérée peut être perçue comme horizontale dans le plan de la page de musique : elle lie un symbole à ses *voisins* sur la partition. C'est elle qui entre en jeu lorsque l'élève est face à une ligne de notes ou de rythmes à *lire*, les symboles étant alors appréhendés par voisinage – un *do*, surmonté d'un accent, à côté duquel se trouve un *ré*, en dessous desquels figure une liaison, etc. ; elle aussi qui intervient lorsque la note est envisagée dans une harmonie (voisins du dessus et du dessous). Cette relation spatiale est conditionnée par l'usage des partitions comme support de lecture : le regard *parcourt* la partition de proche en proche pour en extraire la signification.

Comparons maintenant ce double système d'associations (sémantique et spatial) au paradigme qui sous-tend la bibliothèque ScoreGen.

Dans ce « nouveau solfège », chaque élément est aussi en relation avec son voisinage selon deux axes, ci-dessus nommés *inclusion* et *héritage*. Que ces relations nous disent-elles de ce paradigme musical ?

Tout d'abord, les deux types de relations d'inclusion et d'héritage interviennent non pas comme dans le solfège entre des *instances* d'objets musicaux mais entre leurs *natures*. Le solfège de ScoreGen représente donc une abstraction du solfège traditionnel au niveau des *types* de symboles.

D'autre part, nous pouvons remarquer que la direction horizontale (ou spatiale) est absente de notre modèle. La proximité n'y indique qu'une relation de nature ; le positionnement des éléments n'a aucune mention au sein de ce solfège relationnel. Il y a

donc ici une nouvelle abstraction, consistant à extraire la *structure* des objets plutôt que leur *forme*.

Enfin, et cela peut sembler plus surprenant, la direction verticale (ou sémantique) fait elle aussi complètement défaut au modèle. Le programme ignore tout de ce que signifie un « pitch » (pour lui simple nombre dans une case mémoire) ou un « groupe » (défini comme une liste de références vers des éléments d'un type tout aussi peu chargé de sens). S'il peut éventuellement être utilisé pour produire des données signifiantes, ce n'est que par le contact entre ses produits (partitions) et le lecteur humain auquel elles sont destinées. Il y a donc là dans notre modèle un troisième type d'abstraction, qui isole le jeu des *relations* entre objets du *sens* qu'elles peuvent revêtir.

III. ScoreGen en action : exemples de réalisations

Cette partie vise à montrer la bibliothèque ScoreGen en fonctionnement, dans un contexte réel de composition. Son but est de démontrer le caractère viable et fonctionnel du dispositif, au travers d'exemples difficiles à réaliser avec d'autres outils existants.

Nous ne montrerons pas toujours les codes complets des exemples, mais nous nous attacherons à démontrer leur fonctionnement au regard de l'explication qui a été donnée de la structure de la bibliothèque.

A. Cadre de travail

Nous utiliserons la classe *Generator* pour créer différentes partitions de démonstration. Pour cela, à chaque nouvel exemple, nous créerons en C++/CLI une classe héritée de cette classe abstraite et représentant un générateur pour notre partition.

La classe *Generator* possède seulement trois fonctions virtuelles qui doivent être implémentées par les classes dérivées : *name*, qui définit le nom de la partition générée, *description*, qui permet de donner plus d'informations sur le contenu de la partition, et *generate*, qui doit renvoyer un objet de type *ScoreAndInfos*. Nous précisons que les deux premières fonctions ne sont en aucun cas le titre et un texte de description présents sur la partition : il s'agit bien de métadonnées informatiques permettant d'étiqueter et/ou de déboguer la partition dans le cadre d'un logiciel de création de partitions utilisant ScoreGen.

Comme nous l'avons écrit, la fonction *generate* a pour type de retour *ScoreAndInfos*. Cependant, par le jeu des conversions implicites entre éléments musicaux que nous avons expliqué plus haut, le programmeur implémentant une classe héritée de *Generator* peut préférer retourner tout type d'élément (très souvent un *Group* ou un *Block*) s'il n'a besoin ni de spécifier les clés des parties ni de fournir d'autres représentations de sa partition grâce à la classe *ScoreAndInfos*.

La structure générale de nos exemples sera donc la suivante :

```
1  public ref class G_StructureDesExemples : public Generator {
2      public:
3          virtual String^ name() override {
4              return "Structure minimale d'un objet de type Generator";
5          }
6          virtual String^ description() override {
7              return "Ce générateur crée un duo avec 10 mesures vides en 9/8.";
8          }
9          virtual ScoreAndInfos^ generate() override {
10             Block^ block = gcnew Block(2);
11             block->addBars(10, qSignature("9/8"));
12             return block;
13         }
14 };
```

Figure 13 : Structure minimale d'un objet de type Generator

Expliquons ce code. La ligne 1 définit une classe nommée *G_StructureDesExemples*, qui implémente (dérive) la classe abstraite *Generator* (d'où notre convention de nommage personnelle préfixée par la lettre *G*), et se termine à la ligne 14.

Cette classe contient les trois fonctions publiques *name*, *description* et *generate* qui définissent le comportement de la classe. Les deux premières renvoient des textes descriptifs permettant d'étiqueter et de caractériser cette création.

La troisième, *generate*, se charge de créer la partition. C'est dans cette fonction que se joue tout le processus algorithmique, ici limité à sa plus simple expression. Nous commençons à la ligne 10 par instancier un objet de type *Block* vide possédant deux portées ; à la ligne suivante, nous utilisons la fonction *addBars* de la classe *Block* pour ajouter dix mesures vides, en précisant la signature grâce à la fonction rapide *qSignature*. Enfin, ligne 12, nous renvoyons le bloc ainsi créé – il sera converti automatiquement en *Score* puis en *ScoreAndInfos* grâce au jeu des conversions implicites.

Dans tous les exemples suivants, nous omettrons la structure générale de la classe héritée de *Generator* pour ne montrer que le contenu de la fonction *generate*. Dans l'exemple ci-dessus, nous aurions par exemple seulement montré ces trois lignes :

```
1  Block^ block = gcnew Block(2);
2  block->addBars(10, qSignature("9/8"));
3  return block;
```

Figure 14 : Portion de code signifiante dans le code ci-dessus

À chacun de nos exemples, nous présenterons les partitions générées ouvertes dans le logiciel Sibelius pour affichage. Sans modifier le résultat obtenu, nous nous permettrons

d’effectuer la tâche de mise en page minimale consistant à choisir le nombre de mesures par système.

Voici la partition générée par le programme expliqué ci-dessus et conforme à nos prévisions :

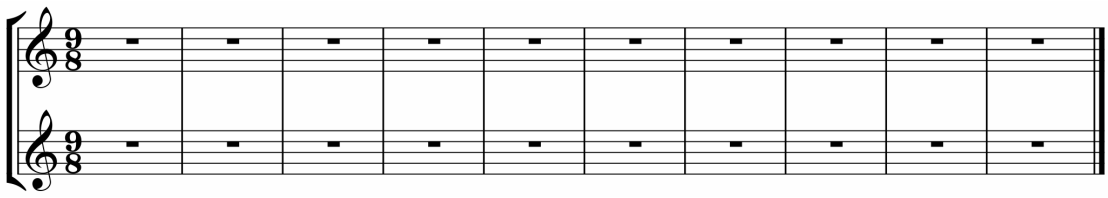


Figure 15 : Partition minimale

Nous allons maintenant monter plusieurs exemples de codes et les partitions qu’ils génèrent. Nous commencerons par des pièces de démonstration, puis nous passerons à des compositions plus ambitieuses.

B. Pièces de démonstration

Ces premières créations visent à montrer des exemples très simples de l’utilisation de la bibliothèque au sein de programmes.

1. Recopier une partition existante

Voici tout d’abord un programme chargé de générer les trois premières mesures du second *Quatuor à cordes* de B. Bartók.

```
Block^ block = gcnew Block(4);
block->addBars(2, qSignature("9/8"));
block->addBars(1, qSignature("6/8"));
block[1, 2] = qGroup("e. 16th(|E4<!p> 16th|A4 16th)|D5 q._(|C#5 q_|C#5 16th|C#5 16th)|G#4");
block[1, 3] = qGroup("q.|G4 q._|Bb4");
block[2, 1] = qGroup("q. e|Eb4<!p><:sempre tenuto> e|Eb4 e_|Eb4 e|Eb4 e|Eb4 e_|Eb4");
block[2, 2] = qGroup("e|Eb4 e|E4 e_|E4 e|E4 e|G4 e_|E4 e|E4 e|D#4 e(|E4");
block[2, 3] = qGroup("e)|F4 e|F4 e_|F4 e|F4 e|C#4 e_|D4");
block[3, 1] = qGroup("q. e|D4<!p><:sempre tenuto> e|D4 e_|D4 e|D4 e|D4 e_|D4");
block[3, 2] = qGroup("e|D4 e|D4 e_|D4 e|D4 e|D4 e_|D4 e|D4 e|D4 e(|D4");
block[3, 3] = qGroup("e)|C4 e|B3 e_|C4 e|C4 e|C4 e_|C4");
block[4, 1] = qGroup("h._(|Bb2<!p> q_|Bb2 16th|Bb2 16th)|Bb3");
block[4, 2] = qGroup("h._|Bb3 q._|Bb3");
block[4, 3] = qGroup("q.|Bb3 q._|G3");
return block;
```

Figure 16 : Programme de génération

Comme tout le contenu musical à produire nous est connu, nous pouvons recourir aux raccourcis *quick*. Après avoir créé le bloc et ajouté des mesures de deux signatures différentes, nous n'avons plus qu'à définir le contenu de chaque mesure pour chaque instrument en utilisant *qGroup*. Les éléments sont énumérés dans l'ordre où ils apparaissent sur la partition. Les durées rythmiques seules comme « e. » (*eighth* plus un point, soit « croche pointée ») désignent des silences ; les durées rythmiques suivies du symbole séparateur « | » et d'une hauteur de note, comme « q.|G4 », génèrent des notes (il faut lire ici « noire pointée » (*quarter*) sur la note *sol* à la 4ème octave). Enfin, les éléments peuvent être accompagnés de diverses informations supplémentaires : textes (telle la nuance *piano* du premier violon), liaisons de prolongation (caractère « _ » après la valeur rythmique), liaisons de phrasé (ouvertes avec « (» et fermées avec «) »), etc.

Voici la partition générée, semblable à la partition consultée pour écrire le code :

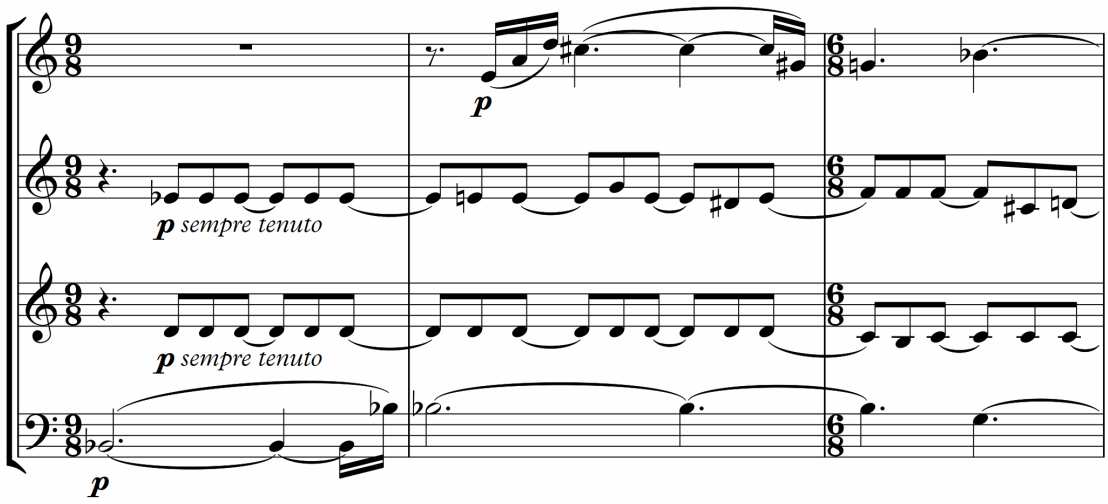


Figure 17 : Partition générée

Ce cas de figure consistant à saisir à la main tous les éléments d'une partition est peu représentatif des usages réels de ScoreGen, dont tout l'intérêt est de construire à partir d'algorithmes des partitions inexistantes. Cependant, nous avons choisi ce premier exemple pour démontrer la possibilité de décrire une partition en utilisant la bibliothèque.

2. Afficher une gamme chromatique et ses fréquences

Dans ce deuxième exemple, nous allons envisager le cas intermédiaire qui consiste à générer une partition dont toutes les caractéristiques nous sont connues d'avance, mais dont la synthèse peut être réalisée de façon algorithme.

Notre but est de créer une partition montrant une gamme chromatique sur laquelle la fréquence en Hertz de chaque note soit indiquée.

Voici le code utilisé :

```
1  int nb = 13;
2  int basePitch = 60;
3  array<Note^, 1>^ notes = gcnew array<Note^, 1>(nb);
4  for (int i = 0; i < nb; i++) {
5      int midi = basePitch + i;
6      notes[i] = gcnew Note(
7          RhythmFigure::_whole,
8          gcnew Pitch(midi),
9          qWords("." + midiToFreq(midi))
10     );
11 }
12 return gcnew Group(notes);
```

Figure 18 : Programme de génération

Les deux premières lignes représentent les options du programme (nombre de notes de la gamme, hauteur de la première note). Nous aurions aussi bien pu les définir comme variables de classe et concevoir une interface graphique permettant à l'utilisateur de générer la partition avec les options de son choix.

La troisième ligne crée un tableau destiné à contenir les *nb* notes de la gamme chromatique. Cette ligne est suivie d'une boucle *for* servant à créer chaque note.

Pour créer la note d'index *i*, nous commençons par calculer la valeur MIDI de cette note ; il s'agit de la valeur de base *basePitch*, à laquelle nous ajoutons l'index pour

augmenter d'un demi-ton à chaque itération. Ensuite, nous assignons à la case d'index *i* du tableau une nouvelle note, créée avec ces arguments (lignes 7 à 9) :

- Un rythme, en l'occurrence *ronde* (avec conversion implicite de *RhythmFigure* vers *Rhythm*) ;
- Un objet *Pitch*, créé grâce à la valeur MIDI précédemment calculée ;
- Une indication textuelle créée avec le raccourci *qWords*. Le préfixe « . » indique un texte normal, et la fonction *midiToFreq* a été définie comme fonction statique dans la classe dérivée de *Generator* créée pour cette partition[18].

La ligne de code suivant la fin de la boucle *for* retourne le groupe des éléments ainsi ajoutés au tableau.

Voici la partition générée :

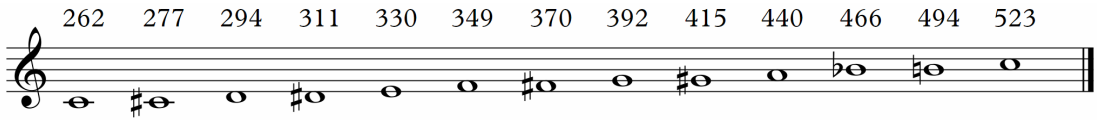


Figure 19 : Partition générée

Nous pouvons vérifier que le programme a généré une séquence chromatique et que les fréquences des notes sont exactes (par exemple le *la* à 440 Hz).

Cet exemple ne relève pas encore à proprement parler de la composition algorithmique, mais il montre que l'usage de ScoreGen est indiqué pour tout programme dont le format de sortie naturel est une partition.

3. Appliquer une suite mathématique à un paramètre musical

Nous proposons maintenant de montrer un exemple de composition algorithmique simple, dans lequel les valeurs successives d'une suite mathématique connue sous le nom de *Suite de Thue-Morse*[19] sont utilisées pour générer un rythme.

[18] La fonction *midiToFreq* implémente cette formule : **freq** = 440 * 2 ^ ((**midi** - 69) / 12)). Elle la retourne arrondie à l'entier le plus proche et sous la forme d'une chaîne de caractères.

La suite de Thue-Morse, composée de seulements deux éléments A et B, est définie par le processus de génération suivant :

- On commence par la séquence à un élément « A » :

  A

- À la suite de cette première séquence on ajoute la séquence obtenue en transformant les A en B et inversement :

  A **B**

- À la suite de cette deuxième séquence on ajoute à nouveau la séquence obtenue en transformant sur tout le début les A en B et inversement :

  A B **B A**

- On itère ensuite le processus :

  A B B A **B A A B**

  A B B A B A A B **B A A B A B B A**

  A B B A B A A B B A A B A B B A **B A A B A B B A A B B A B A A B**

  etc.

La suite infinie résultant de ce processus a la propriété remarquable de ne jamais présenter trois répétitions consécutives d'un motif : aucune lettre ne vient plus de deux fois de suite, aucun groupe de 2 lettres n'est répété plus de deux fois, ni aucune sous-séquence de quelque longueur que ce soit.

Cette propriété donne à la suite de Thue-Morse un intérêt certain en CAO, car elle assure au paramètre auquel elle est appliquée une imprévisibilité maximale.

Nous avons choisi dans cet exemple d'appliquer la suite de Thue-Morse à un matériau rythmique. Ici, les A de la suite sont associés au rythme « deux croches » (la première des deux étant accentuée), et les B de la suite sont associés au rythme plus long « trois croches » (la première étant aussi accentuée).

[19] Aussi *Suite de Prouhet-Thue-Morse* dans la littérature francophone, du nom de ses inventeurs indépendants.

Figure 20 : Rythmes choisis pour la génération

Nous avons décidé de réaliser ce rythme sur une seule hauteur, pour interprétation par un percussionniste. Enfin, nous avons écrit ces rythmes au sein d'une mesure en 5/8, car les successions de A et de B fixées par la suite employée assurent ainsi qu'aucun motif ne sera à cheval sur deux mesures.

Le programme, que nous ne reproduisons pas ici, commence par générer la suite de Thue-Morse (qui se trouve déjà implémentée dans la classe *Seq* de la bibliothèque) sur un nombre de valeurs défini par l'utilisateur de la classe (ici, 64). Il crée ensuite les éléments A et B comme des objets de type *Group*, et les concatène au sein de groupes plus larges grâce à l'opérateur d'addition que cette classe fournit. Enfin, il colle le groupe obtenu sur un bloc possédant une unique portée et le bon nombre de mesures.

Voici la partition générée, à laquelle nous avons ajouté manuellement un tempo et une nuance :

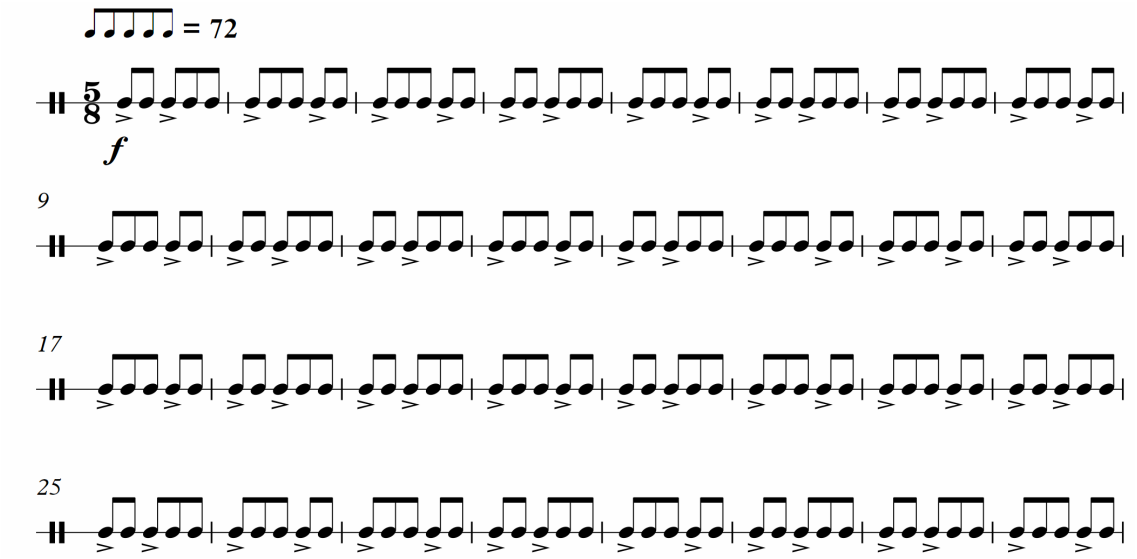


Figure 21 : Partition générée

À l'écoute, la succession rythmique possède les caractéristiques d'imprévisibilité qui avaient motivé l'emploi de la suite de Thue-Morse :

Annexe audio I

http://antoinegabrielbrun.com/ressources/scoregen-memoire-de-recherche/

Ce processus de génération d'un matériau musical à partir d'une séquence construite récursivement, ici présenté sous sa forme la plus simple, peut être largement généralisé ; M. SUPPER montre ainsi comment il est possible de jouer sur des règles de réplication plus complexes (« L-systèmes ») pour obtenir des séquences auto-similaires pouvant être musicalement intéressantes[20].

4. Appliquer une suite mathématique à un paramètre musical

Pour clore cette section dédiée aux démonstrations de fonctionnement de la bibliothèque, voici un exemple qui donne à voir un usage de la classes *TimeVariable*, en association avec un processus aléatoire.

Nous allons générer une partie de marimba se présentant comme une succession de doubles croches. Certaines de ces doubles croches seront jouées, d'autres seront laissées en silence sous forme de quarts de soupirs ; la proportion entre 0 et 1 de doubles croches jouées est nommée *densité* : à 0 toutes les doubles croches sont laissées silencieuses, à 1 toutes sont jouées. Toutes les hauteurs seront choisies aléatoirement à l'intérieur d'un intervalle.

Nous souhaitons que ce matériau soit évolutif. Pour cela, nous ferons évoluer la densité de 0,1 (très faible) à 1 (maximale), et nous élargirons progressivement l'intervalle dans lequel les hauteurs sont choisies (en partant de l'unisson C5 vers la double octave C4-C6). Toutes ces évolutions suivront un dessin linéaire.

[20] SUPPER, Martin. *A Few Remarks on Algorithmic Composition*. Computer Music Journal vol. 25, n°1, *Aesthetics in Computer Music*. Spring, 2001, p. 50.

Voici le code :

```
int nb = 192;
array<NoteOrRest^, 1>^ elements = gcnew array<NoteOrRest^, 1>(nb);
TimeVariable^ density = gcnew TimeVariable_Lin(0.1, 1);
TimeVariable^ low     = gcnew TimeVariable_Lin(60, 48);
TimeVariable^ high    = gcnew TimeVariable_Lin(60, 72);
for (int i = 0; i < nb; i++) {
    double densityNow =    density->at(i, nb);
    int        lowNow = round( low->at(i, nb));
    int       highNow = round(high->at(i, nb));
    if (Rand::double01() < densityNow) {
        int midi = Rand::integer(lowNow, highNow + 1);
        elements[i] = qNote("16th|" + midi);
    } else {
        elements[i] = qRest("16th");
    }
}
return gcnew Group(elements);
```

Figure 22 : Programme utilisant des automations

Après la définition du nombre de doubles croches et d'un tableau pour les contenir, nous définissons trois automations linéaires avec la classe *TimeVariable_Lin*. La première décrit l'évolution de la densité au cours du temps ; les deux autres représentent le bas et le haut de l'intervalle dans lequel les hauteurs sont tirées, en valeur MIDI.

À chaque itération de la boucle *for* destinée à créer les éléments, nous commençons par utiliser les trois automations pour déterminer les valeurs de densité et d'ambitus à cet instant. Nous tirons ensuite au sort la présence d'une note ou d'un silence en comparant un nombre aléatoire entre 0 et 1 à la dentité actuelle. Dans le premier cas une note est générée avec une hauteur tirée au sort dans l'intervalle actuel ; dans le second, le programme génère un silence.

Ce programme n'a été montré ici qu'à titre informatif. Il a l'avantage d'être simple, mais est encore très imparfait. Par exemple, les successions de quatre quarts de soupirs ne sont pas regroupées en soupirs, ce qui résulte en des rythmes très peu lisibles ; l'ensemble est regroupé dans une unique mesure, qui si *nb* est grand ne peut être affichée sur la largeur d'une page.

Nous avons donc modifié le code présenté ci-dessus pour afficher les rythmes de façon plus conforme à ce qu'attendrait un lecteur humain, en tenant compte du fait que certaines notes écrites en doubles croches pouvaient être avantageusement réécrites

comme des valeurs plus longues ; et nous avons collé le groupe dans un bloc écrit en 4/4. La partition générée est la suivante[21] :

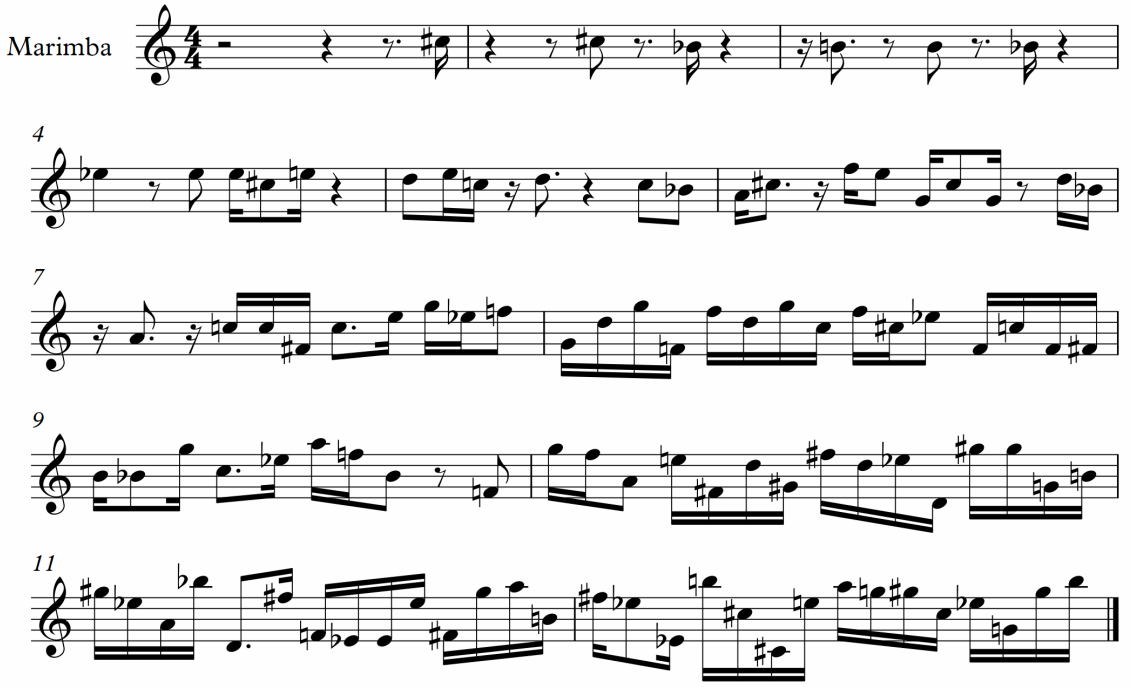


Figure 23 : Partition générée

La densification et l'ouverture progressive de l'ambitus sont bien visibles.

C. Étude de compositions

Les programmes que nous avons montrés jusqu'à présent se limitent à fournir un matériau musical. Les partitions qu'ils génèrent ne sont en aucun cas des pièces à part entière, ni même des blocs de musique exploitables.

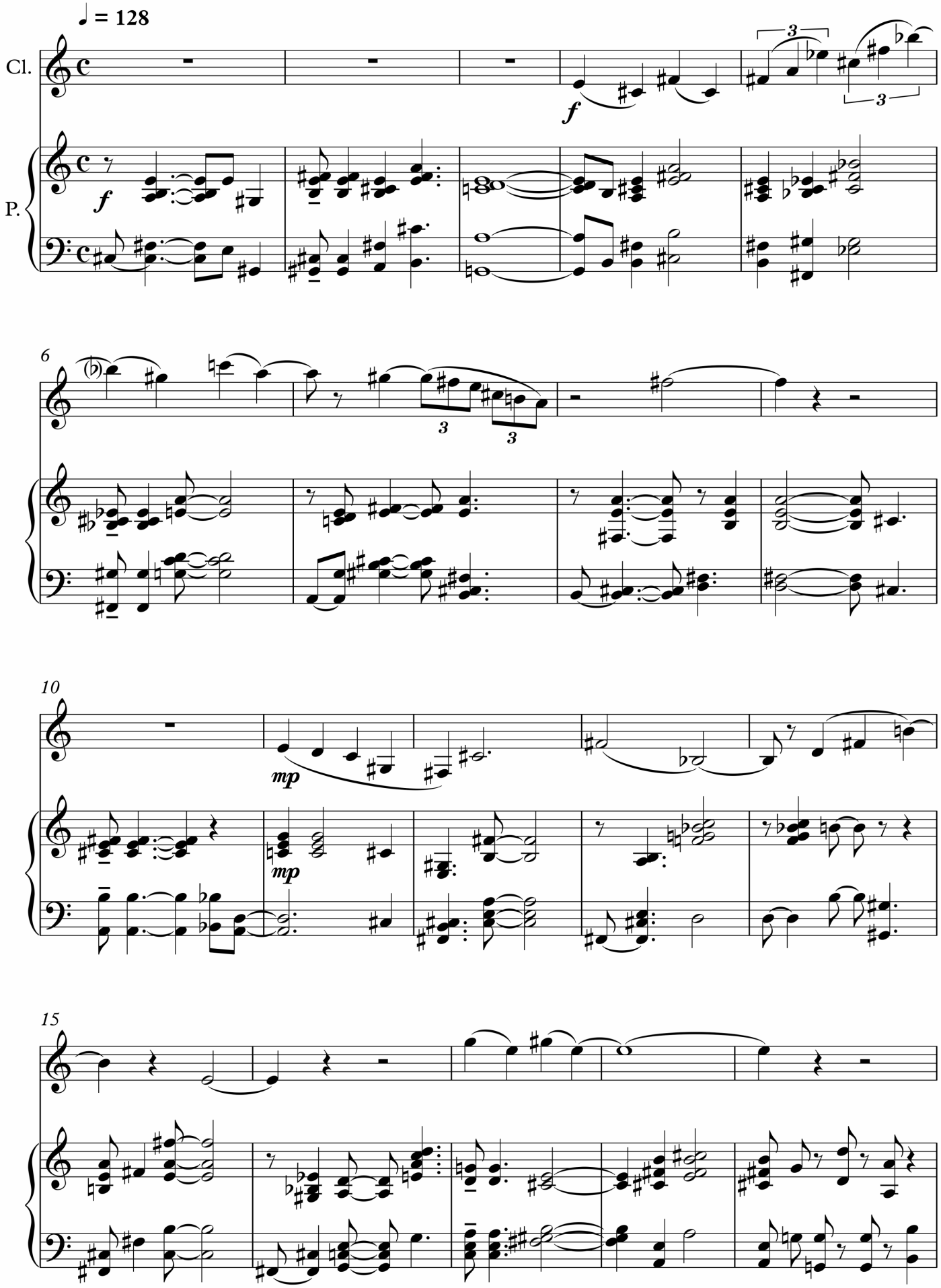
Cette section montre plusieurs exemples de créations qui, sans tous prétendre à constituer des pièces en elles-mêmes, possèdent ou bien une structure beaucoup plus complexe, ou bien un intérêt musical bien plus important. Le premier exemple, longuement expliqué, a été imaginé spécialement pour ce travail ; les suivants sont abordés plus rapidement et sont issus d'une pièce pour grand orchestre symphonique, que nous avons composée en utilisant largement ScoreGen. Nous terminerons cette suite

[21] À cause des tirages aléatoires, la partition générée change à chaque nouvelle exécution.

d’exemples par une brève évocation de l’usage pouvant être fait de ScoreGen dans le cadre de pièces de musique électronique.

1. Exemple de réalisation : pièce en mélodie accompagnée

Une fois n’est pas coutume, avant même de montrer des extraits de code ou d’expliquer un algorithme, nous allons d’abord donner à lire la partition générée par un nouveau programme utilisant ScoreGen.


♩ = 128
Cl.
P.
f
mp
mp
6
10
15

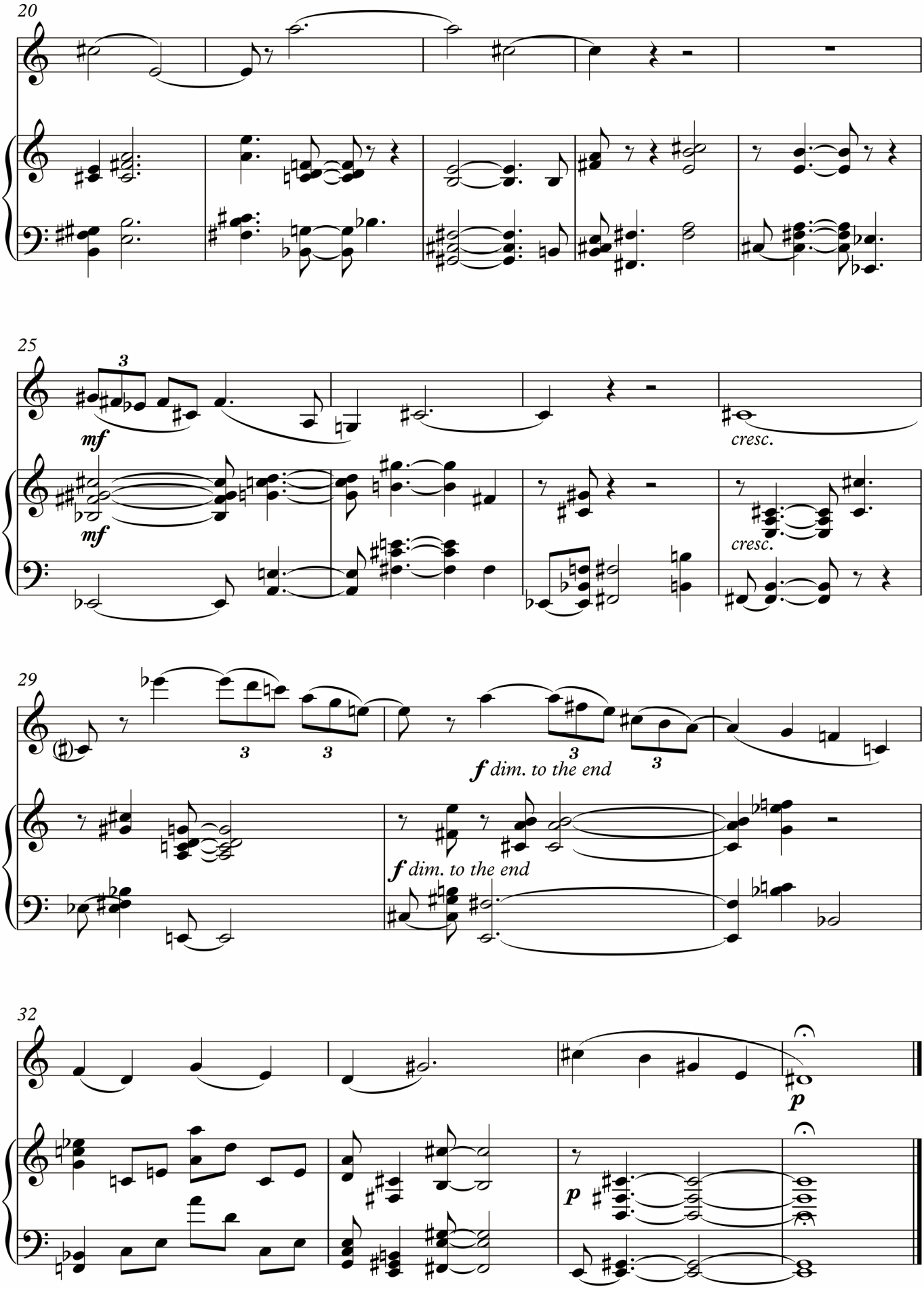


Figure 24 : Deux pages de la partition générée

Nous invitons le lecteur à écouter la simulation audio[22] fournie en annexe avant de poursuivre :

Cette pièce[23] pour clarinette et piano nous semble intéressante à plus d'un titre :

- Elle possède un degré de complexité, de variété et d'unité suffisants pour pouvoir éventuellement être considérée comme *musicalement intéressante* ;
- Le programme l'a écrite pour deux instruments en respectant non seulement leur ambitus mais aussi leurs autres possibilités instrumentales (taille des mains du pianiste, caractère *suffisamment* conjoint des parties) ;
- Plus généralement, elle est directement lisible par des interprètes humains[24], notamment grâce à une écriture rythmique cohérente ;
- Son cheminement harmonique paraît *logique* quoique étant assez erratique ;
- Sa mélodie peut *se prêter* à une interprétation expressive.

La lecture de la liste ci-dessus peut laisser sceptique quant au caractère *subjectif* des critères abordés ; nous avons indiqué en italique les expressions qui sont sujettes à l'appréciation de chacun et ne peuvent faire l'objet d'aucune évaluation objective.

Cependant, c'est peut-être là tout l'intérêt de cet exemple : il montre un cas de pièce relevant de la composition algorithmique et qui, soumise à la *partialité* d'un auditeur humain, peut arriver à remporter son adhésion au moins sur certains aspects.

Expliquons maintenant le fonctionnement du programme ayant généré cette pièce. Nous ne reproduisons pas ici le code complet (environ 750 lignes de code), mais nous nous appliquerons à expliquer son fonctionnement avec le plus de détail possible, à montrer

[22] La simulation n'implique pas d'interprète humain. Elle a été créée en important la partition générée dans le logiciel de gravure musicale Sibelius et en exportant l'audio grâce au plug-in NotePerformer.
[23] Nous décidons ici subjectivement de qualifier la sortie du programme de *pièce* et non plus comme dans les exemples précédents de *matériau musical*. Ce choix s'explique par les efforts déployés pour donner à la sortie une structure et une fin, et par la minutie dont nous avons fait preuve pour que la pièce générée par le programme ait l'apparence d'une pièce composée avec un style et des intentions musicales.
[24] À l'exception des altérations, qui ne suivent pas la logique habituelle. Ce point sera évoqué un peu plus loin.

les stratégies que nous avons déployées pour obtenir ce résultat, tout en montrant comment ScoreGen est utilisé pour le réaliser.

Cette pièce est basée sur deux temporalités superposées :

- Un temps régulier, courant sur toute la durée de la pièce, correspondant à *(i)* un certain nombre de mesures en **C**, *(ii)* une subdivision fixe en huit croches par mesure et *(iii)* une hiérarchie de temps forts et faibles[25] ;
- Une suite de durées irrégulières et imprévisibles, non synchronisées avec les mesures, et correspondant aux durées d'une succession d'accords pentatoniques durant chacun entre une et neuf croches.

Voici, comme illustration, une représentation de ces deux temporalités dans le premier système (mesures 1 à 5) :

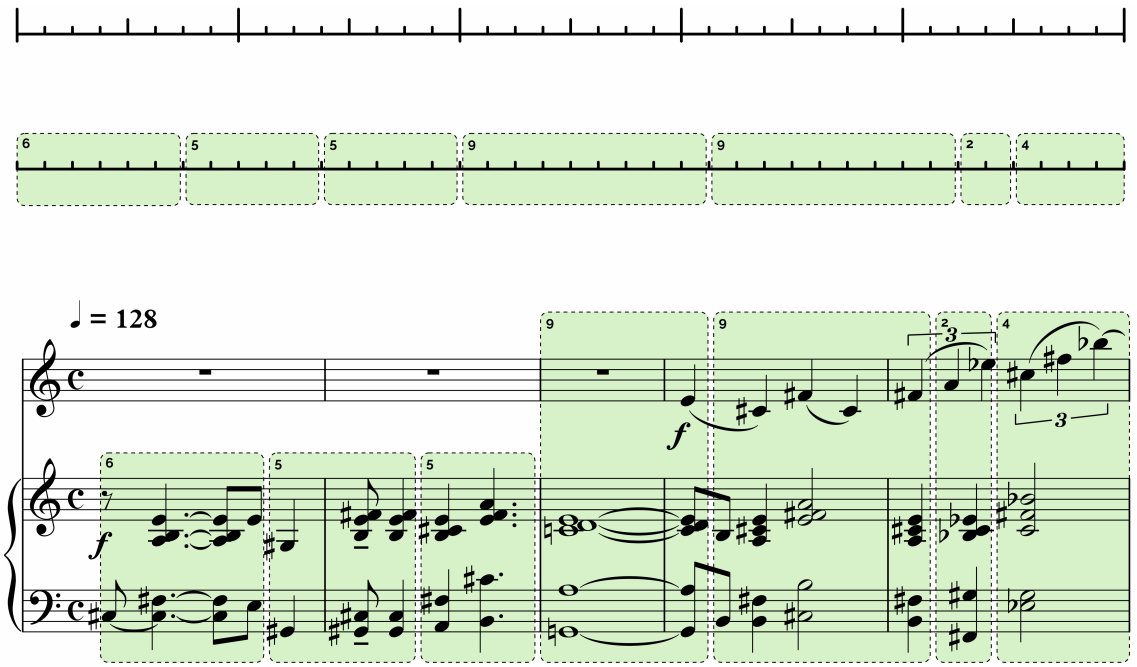


Figure 25 : Temporalités superposées dans la pièce

La première ligne de la figure montre la première temporalité (mesurée) ; la seconde montre la temporalité irrégulière des accords ; et la troisième permet de visualiser l'emplacement des accords sur le premier système de la partition. Les nombres apparaissant au coin des zones représentent les durées des accords en croches.

[25] Nous avons en fait retenu une seule croche forte dans la mesure (la première) et sept croches faibles (toutes les autres).

Nous avons écrit que les accords utilisés étaient des accords pentatoniques. Plus précisément, à chaque section harmonique (les zones du schéma ci-dessus) est associé un mode pentatonique différent parmi les douze transpositions possibles du mode *do – ré – fa – sol – la*. Dans chaque zone sont alors calculés deux accords « complets », possédant les cinq notes du mode dans un étagement différent. La réalisation de la partie de piano utilise l'un de ces deux renversements ou les deux ; nous expliquerons comment.

Nous avons donné un premier aperçu de la structure rythmique et harmonique de la pièce. À présent, reprenons le code dans l'ordre pour montrer comment le programme s'y prend pour générer cette partition à l'aide de ScoreGen.

• Tout d'abord, le programme énumère les 12 modes pentatoniques possibles, vus comme des listes de nombres entre 0 et 11. Il part pour cela du mode 0 – 2 – 5 – 7 – 9 évoqué plus haut avec les noms de notes correspondants et applique des transpositions successives au demi-ton. À chaque transposition, il s'assure de rester dans l'intervalle [0, 11] et retrie les tableaux afin de conserver des modes énumérés par ordre croissant. Les 12 modes sont ensuite mélangés dans un ordre aléatoire, pour que celui considéré comme le premier puisse être chacun des douze modes possible (et ainsi des suivants).

• Ensuite, le programme utilise l'une des suites mathématiques implémentées par la bibliothèque dans *Seq* pour générer une succession temporelle non stochastique de 48 modes parmi les modes qui viennent d'être créés. La séquence employée, nommée *fractal010210* dans la bibliothèque, a pour particularité de ne jamais répéter deux fois de suite un même élément[26]. Nous disposons donc maintenant d'une succession de modes pour notre morceau, dessinant sa structure harmonique.

• Nous avons ensuite généré une suite de 48 durées applicables à ces modes, exprimées en nombres de croches. Pour cela, nous avons d'abord tiré aléatoirement une suite de 48 nombres compris entre 1 et 9. Pour éviter que la dernière mesure du morceau ne soit incomplète, nous avons ensuite ajusté la dernière durée pour que la somme de toutes les durées soit divisible par 8.

[26] Nous aurions aussi pu effectuer un tirage aléatoire sans répétitions.

• Étant maintenant en possession de toutes les données sur le déroulement harmonique du morceau, nous avons ensuite pu calculer la durée totale du morceau, puis son nombre de mesures (durée divisée par 8).

À cette étape, le programme dispose de tous les éléments pour commencer à écrire de la musique. Il va le faire en accordant un traitement séparé aux deux instruments.

• **Pour la partie de clarinette,** nous avons commencé par décider quelles mesures du morceau seraient jouées et lesquelles seraient laissées vides[27]. Ce choix, important pour donner à l'auditeur le ressenti d'une structure, est partiellement stochastique. Nous avons fixé un minimum de trois mesures vides au début du morceau (vécues par l'auditeur comme une introduction de piano) et un minimum de trois mesures jouées à la fin (pour s'assurer que la clarinette ne laisse pas le piano accompagnateur terminer le morceau seul). Ensuite, nous avons tiré au sort pour chaque mesure si elle serait jouée ou non, avec une probabilité constante de 75 %. Enfin, pour éviter que le discours de l'instrument soliste n'apparaisse hâché, nous avons opéré une certaine réorganisation de la position des mesures vides, conduite aléatoirement, et ayant pour effet de tendre à placer les mesures vides côte à côte[28].

• Pour générer la partie de clarinette, nous avons ensuite mis en place une boucle parcourant les mesures du morceau et chargée de générer séparément et successivement le contenu de chaque mesure. Pour assurer une continuité d'une mesure à la suivante, nous avons tenu à jour une variable stockant à chaque itération la dernière note atteinte à la fin d'une mesure, pour continuer la phrase en partant d'une hauteur proche à la mesure suivante[29]. Chaque itération de la boucle appelle une même fonction spécialisée, qui a pour rôle d'écrire la mesure.

[27] Certaines mesures dites « vides » n'apparaissent pas comme telles sur la partition, car elles terminent la mesure précédente par une noire liée (voir par exemple les mesures 16 et 19).

[28] L'algorithme employé opère des échanges stochastiques entre mesures adjacentes en tentant de minimiser les arrêts et les reprises de jeu. Il est contrôlé par un paramètre entre 0 et 1 permettant de décider du taux de réorganisation (à 0 il n'y a aucune réorganisation, et à 1, toutes les mesures vides se trouvent groupées au début du morceau).

[29] Cette variable est construite grâce à une objet personnalisé, capable d'indiquer à une mesure si la mesure précédente se termine ou non par une note, et si oui, si celle-ci possède une liaison. Le traitement pour le remplissage d'une mesure est en effet différent dans ces trois cas – par exemple, une mesure suivant une mesure qui se termine par une note liée doit impérativement commencer sur la même note (cela excluant aussi de commencer sur un silence).

• Cette fonction prend notamment en arguments le bloc dans lequel écrire les notes, l'index de la mesure à réaliser, un booléen indiquant si la mesure est jouée ou non, une variable de progression parcourant linéairement l'intervalle [0, 1] au cours du morceau, les huits modes caractérisant chacune des croches de la mesure (déterminées grâce aux modes et aux durées calculées précédemment), et les informations sur la dernière note de la mesure précédente ; en toute logique, elle retourne la dernière note générée pour usage dans la mesure suivante.

• Pour générer le contenu d'une mesure, cette fonction tire au sort un algorithme de génération entre 1 et 10. Chacun des ces algorithme correspond à un rythme et à un dessin mélodique, qui selon l'algorithme sont fixés ou tirés au sort suivant des règles. La fonction s'assure que l'algorithme choisi convient en fonction de son placement dans le morceau et du nombre d'harmonies différentes apparaissant dans la mesure, puis l'exécute.

• Les dix algorithmes possibles représentent des motifs reconnaissables ; ainsi, nous pouvons facilement nous convaincre que les mesures 29 et 30 de la partie de clarinette ont été générées par la même méthode – à noter que l'algorithme a bien pris en compte la présence d'une note liée obligatoire au début de la seconde des deux mesures. De même, les mesures 31 et 32 ont été générées par le même algorithme ; il est intéressant de remarquer que cet algorithme possède deux cas de réalisations, l'un où le schéma est descendant et lié sur quatre notes, l'autre où le schéma est brisé et lié par deux.

• Chacun des dix algorithmes de génération de la mélodie se charge d'*arrondir* toutes les hauteurs de notes à la hauteur la plus proche dans le mode actuel. Ainsi, dans le mode pentatonique *do – ré – mi – sol – la*, l'algorithme qui tente d'insérer un *fa* dièse doit l'arrondir au *sol*. Cette méthode d'obtention de mélodies compatibles avec l'harmonie est toujours calculée par rapport au mode de la croche en cours pour chacune des notes du motif ; ainsi, le mode dans lequel évolue la mélodie peut changer en cours de mesure au gré des changements d'accords dans l'harmonie.

• **La partie de piano** est construite de façon assez similaire. Cependant, elle n'est pas générée mesure par mesure mais accord par accord. Le programme parcourt la série des harmonies (modes pentatoniques) dans une boucle, et appelle pour chaque harmonie successive une fonction chargée de la réaliser.

• Cette fonction prend en arguments les deux groupes où écrire les éléments (pour la main droite et la main gauche), la progression dans la pièce entre 0 et 1, le mode pentatonique caractérisant cette zone harmonique, et la durée de l'harmonie en croches.

• La fonction commence par créer deux accords différents *c1* et *c2* qui possèdent chacun les cinq notes de l'harmonie. Les ambitus possibles font l'objet d'une automation au cours de la pièce grâce à deux objets de type *TimeVariable_Lin*, pour la note basse minimale et la note aiguë maximale. Le programme s'assure que les deux accords ont une basse différente, qu'ils sont croissants (c1 < c2 en termes de note basse et de note aiguë), et qu'ils ne présentent pas de seconde plaquée dans le grave (pour éviter un effet de brouillage harmonique s'ils apparaissent plaqués dans la partie de piano finale).

• La fonction détermine ensuite la position de l'harmonie à ajouter par rapport aux barres de mesures, en calculant si l'hamonie *(a)* démarre sur un début de mesure, *(b)* est entièrement incluse dans une mesure, ou *(c)* est à cheval sur au moins[30] deux mesures.

• Enfin, comme pour la partie de clarinette, le programme appelle l'une des douze méthodes possibles de réalisation de l'harmonie demandée dans la durée indiquée. Le choix aléatoire de l'algorithme à utiliser est bien plus complexe que dans le cas de la partie de clarinette :

- tous les algorithmes de réalisation possibles ne sont pas compatibles avec tous les types de position par rapport à la mesure. Ainsi, certains algorithmes sont conçus pour marquer le début d'une harmonie sur un début de mesure ; d'autres représentent une réalisation sans temps forts et ne sont adaptées que pour des harmonies entièrement incluses dans une même mesure ; d'autres changent d'accord entre *c1* et *c2* au passage d'une barre de mesure, etc.

- de plus, tous les algorithmes de réalisation possibles ne sont pas compatibles avec toutes les durées. Ainsi, certaines réalisations nécessitent une durée d'au moins trois croches pour être employées, etc.

• Chacune des douze méthodes ajoute à la main gauche et à la main droite des éléments respectant l'harmonie demandée (en utilisant *c1*, *c2* ou les deux) et d'une durée

[30] En pratique, avec une durée maximale de 9 croches dans une mesure en **C**, une harmonie ne peut courir sur plus de deux mesures.

conforme à celle requise. Certaines correspondent à des accords plaqués, d'autre à des arpégiations des accords, d'autres ne conservent que les basses, etc. Nous avons choisi un traitement simple, visant à varier la réalisation plutôt qu'à générer un discours complexe.

Les parties de clarinette et de piano une fois réalisées, nous avons encore programmé une série de post-traitements appliqués au matériau obtenu :

• Les trois voix sont insérées dans un *Block* à trois parties, ce qui a notamment pour effet de contraindre les éléments joués au piano dans la mesure générale en **C** ;

• Une mesure de fin est ajoutée avec une ronde tenue et un point d'orgue ;

• La voix de clarinette est redessinée en changeant les octaves pour suivre une forme générale moins chaotique ;

• Une indication de tempo est ajoutée ;

• La partie de piano est réécrite suivant un algorithme complexe permettant de choisir à quelle portée écrire chaque note en fonction des déplacements de mains requis ;

• Les silences sont refactorisés en suivant les habitudes de lecture en **C** ;

• Des nuances définies par le programmeur sont ajoutées en des points de la partition déterminés en suivant avec une précision modérée le registre de la clarinette.

Ces opérations de post-traitement sont essentielles pour la synthèse d'une partition de qualité, tant pour permettre la génération d'une partition directement lisible par des interprètes, que pour favoriser l'impression d'une structure à l'écoute de la pièce.

Arrivés ici, il peut être intéressant de réexaminer ou de réécouter la partition générée avec la connaissance fine des algorithmes utilisés. Ce réexamen nous mène aux remarques suivantes :

• L'algorithme utilisé pour synthétiser les notes de la voix de clarinette mime assez efficacement le jeu d'un interprète qui improviserait sur une grille d'accords. Cela inclut l'attention portée au respect des notes de l'harmonie, mais aussi le « retard » consenti lorsque le changement de note intervient après le changement de l'harmonie au piano.

• De même, l'algorithme du piano reproduit le comportement d'un pianiste cherchant à réaliser une grille d'accords de façon simple mais non monotone.

• L'efficacité de la succession harmonique choisie tient au fait que tous les enchaînements d'accords entre deux accords pentatoniques apparaissent à l'oreille comme acceptables – nous aurions eu bien d'autres difficultés à générer un morceau d'harmonie classique par les mêmes procédés.

• La partition générée possède de nombreuses imperfections. Dans la forme, elle peut apparaître comme désagréablement monotone ; sa mélodie peut sembler globalement erratique et, par endroits, mélodiquement boiteuse. Localement, le choix de l'altération dièse ou bémol pour une même note semble illogique, rendant la lecture de certains accords malaisée (l'accord de piano de la mesure 5 présente par exemple simultanément deux notes bémolisées et deux notes diésées[31]).

• En conséquence, nous pouvons considérer que si ce programme avait été utilisé dans un contexte réel de composition – dans le but de faire jouer la pièce, suite à une commande, etc. –, l'exécution du programme de synthèse aurait certainement représenté une *première étape* de travail. Celle-ci aurait sans aucun doute été suivie de modifications, de corrections, voire d'une réécriture complète d'une pièce utilisant la sortie du programme comme matériau de base.

• Cependant, malgré les nombreuses imperfections du produit engendré, nous constatons la puissance de la pensée algorithmique à l'aune de cette considération : dans ce cas précis, *la pièce générée par le programme surpasse l'idée qui lui a donné naissance* – la pièce générée aurait difficilement pu être créée autrement par celui qui, pourtant, en a fixé tous les paramètres sur des critères issus de sa subjectivité. Nous développerons plus loin cette observation.

[31] Nous aurions pu, sinon complètement éradiquer, du moins largement améliorer ce défaut d'écriture par une étape supplémentaire de post-traitement.

2. Autres exemples issus de la pièce *Trailer!* pour orchestre

Les autres exemples que nous voulons aborder ici sont issus d'une pièce pour grand orchestre symphonique que nous avons composée avec ScoreGen un an avant la rédaction de ce mémoire[32].

Même si la structure générale de cette pièce est conçue de façon « artisanale[33] », nombre de ses éléments et matériaux constitutifs ont été générés avec ScoreGen ; certains d'entre eux ont été ensuite largement modifiés et retravaillés, d'autres apparaissent dans la partition finale pratiquement comme ils sont sortis du programme.

Nous proposons au lecteur d'écouter (et de regarder) cette pièce avant de poursuivre :

Annexe audio III

http://antoinegabrielbrun.com/ressources/scoregen-memoire-de-recherche/

La représentation qui défile dans cette présentation de la pièce correspond aux notes jouées par l'orchestre, les notes les plus graves apparaissant en bas de l'écran et les notes les plus aiguës en haut.

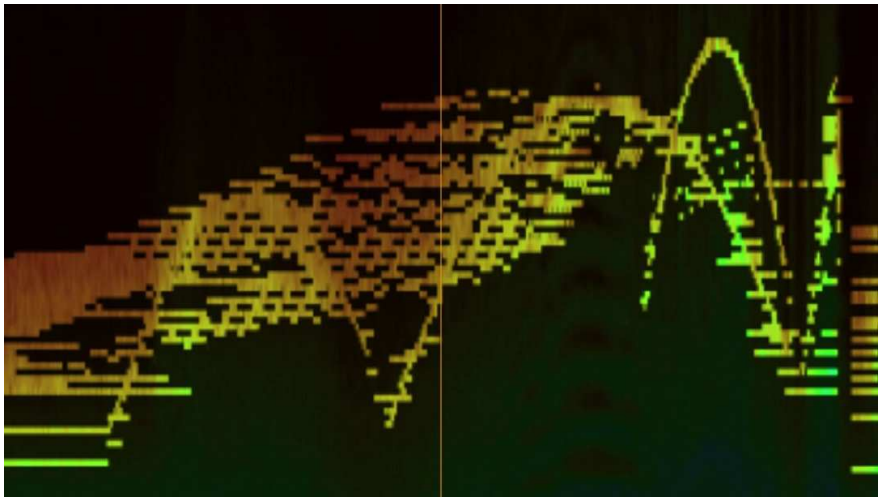


Figure 26 : Aperçu de la pièce algorithmique Trailer!

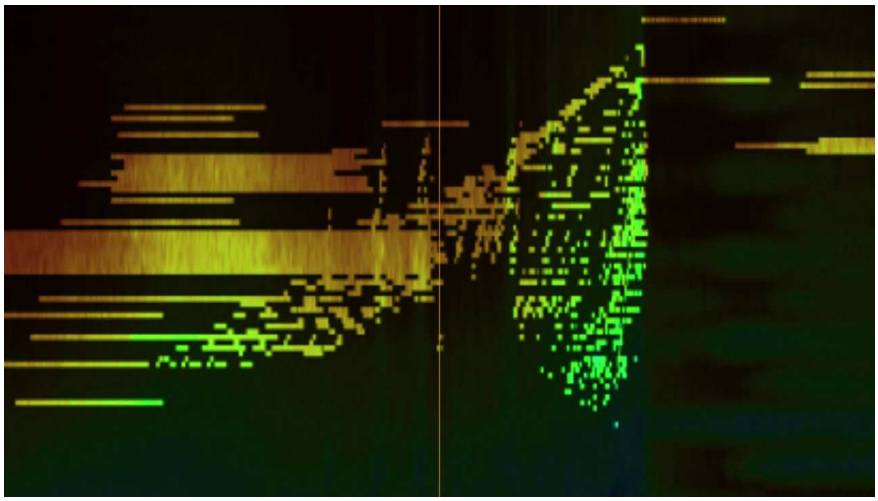
[32] La bibliothèque ScoreGen en était alors à un stade antérieur de son développement. C'est l'écriture de cette pièce, algorithmiquement exigeante, qui nous a poussé à améliorer et à compléter la bibliothèque pour qu'elle permette de mettre en place des algorithmes de complexité arbitraire.

[33] Comprendre : sans qu'intervienne une quelconque pensée algorithmique.

Effectuons une rapide visite de différents moments de la pièce où interviennent des programmes conçus avec ScoreGen.

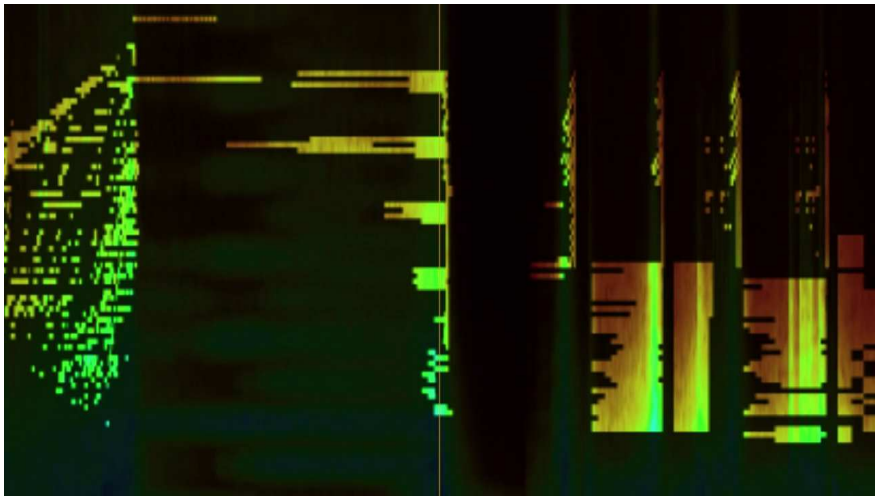
a) Usage de nuages de motifs

Le grand crescendo orchestral qui commence à la seconde 55" est produit avec le générateur *CloudMaker*. Nous lui avons fourni un grand nombre de matériaux mélodiques qui, joués à des moments aléatoires par tous les instruments, dessinent ensemble une forme prédéfinie complexe ; les notes sont fixées sur un mode à 8 sons aléatoires, donnant une sensation de grande complexité spectrale mais occasionnant une impression harmonique bien différente d'un nuage laissé sur des hauteurs chromatiques.



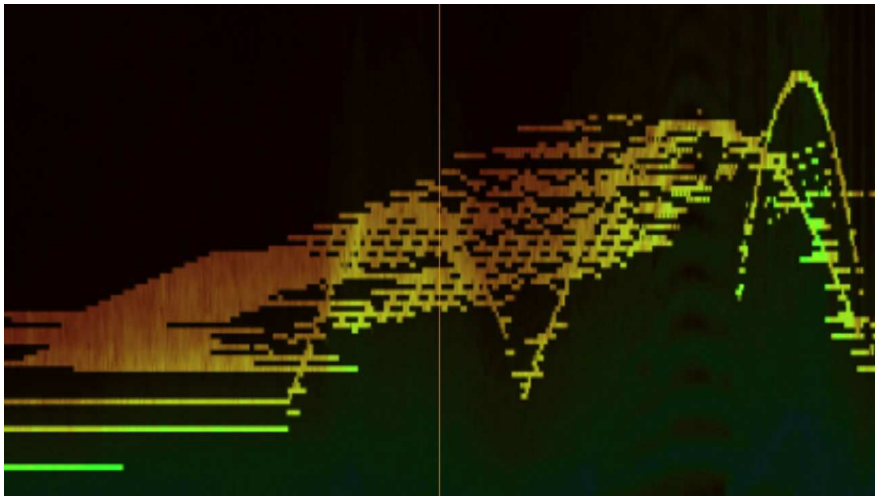
b) Réalisation d'harmonies à spectre continu

Les accords verticaux comme celui de 1'27" sont conçus algorithmiquement : ils répartissent la totalité des hauteurs d'un intervalle (ici, le plus grand intervalle accessible à l'orchestre symphonique) entre tous les instruments, en s'assurant, pour obtenir le spectre le plus continu possible, de laisser les instruments au son puissant seuls sur leur note et à l'inverse de grouper les instruments plus faibles. Le procédé est aussi employé plus loin pour créer des nappes à spectre continu sur des tenues.



c) Réalisation de mélodies paramétrées réparties

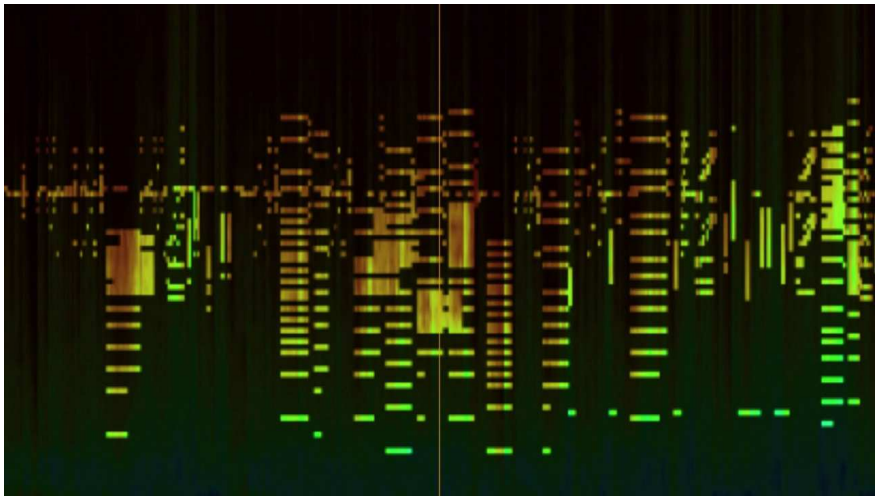
Les courbes qui apparaissent (notamment) à partir 2'24", difficilement réalisables à la main, sont des paraboles[34]. Les notes qui les constituent courent d'un instrument à l'autre et sont soutenues par les glissandos de tous les instruments qui en sont capables (cordes, clarinettes, et trombones pour les points de rebond).



[34] Nous avons choisi cette courbe mathématique car elle correspond à la trajectoire en ordonnées d'un objet lancé vers le haut puis retombant vers le sol. Nous avons ainsi pu figurer un élan puis des rebonds croissants.

d) Usage de chaînes de Markov sur la micro-structure

La section entre 2'49" et 4'06" est générée en utilisant une chaîne de Markov complexe :



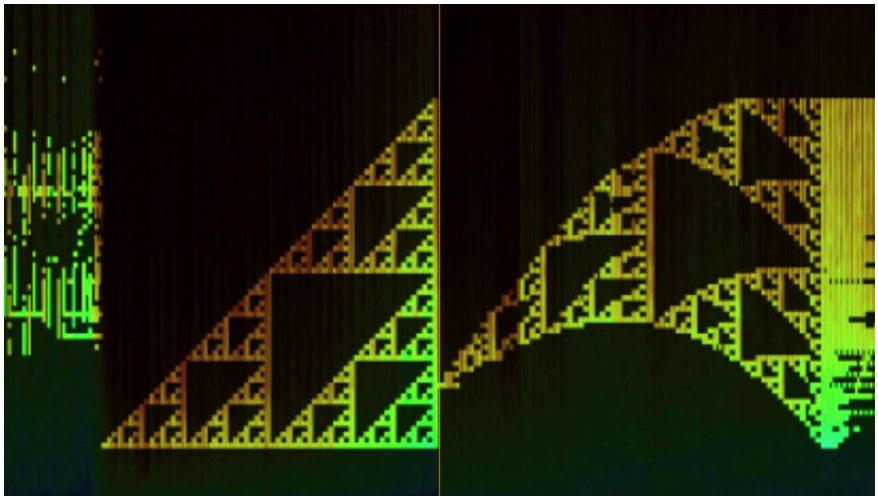
Dans cette section, les éléments juxtaposés sont des grands blocs orchestraux durant chacun entre une et quelques croches. Leurs durées différentes donnent à l'ensemble un caractère rythmique fort et instable. Le passage, qui sert de transition vers la figure suivante (voir plus bas), est évolutif :

- Les probabilités de transition entre les différents blocs juxtaposés par la chaîne de Markov évoluent tout au long du passage, ce qui est utilisé pour obtenir l'apparition progressive de nouveaux élément ;
- Tous les éléments manipulés par la chaîne de Markov évolutive sont eux-même stochastiques et réglés par des paramètres munis d'automations – ainsi leurs durées possibles, leur étalement spectral ou leur nuance peuvent évoluer et dessiner une évolution du matériau.

Cette section est un exemple intéressant de l'utilisation pouvant être faite de concepts algorithmiques au sein d'une composition ; tout d'abord, parce qu'elle met un algorithme au service d'une idée structurelle (écrire une transition) ; ensuite, parce qu'elle met en jeu un aller-retour entre composition manuelle et composition algorithmique (certains des éléments gérés par la chaîne de Markov sont écrits manuellement au préalables, comme les duos de wood-blocks et de piano ; le matériau généré est retravaillé manuellement).

e) Mise en musique de structures mathématiques

Enfin, la double structure apparaissant à 4'06", qui tient à la fois de la surprise auditive et visuelle, est basée sur la figure mathématique nommée « triangle de Sierpiński[35] ».



Elle est entièrement réalisée par ScoreGen, qui se charge d'attribuer à chaque instrument ses notes au bon moment de façon que le dessin général entendu soit celui du triangle fractal.

L'intérêt musical de cette structure tient à plusieurs facteurs :

- Ainsi placée et transcrite en notes, elle dessine un *crescendo* naturel porté par l'augmentation progressive du nombre d'instruments et par l'élargissement de l'ambitus au cours du temps ;
- Sa structure « trouée » crée une périodicité à 4, 8, 16… temps, qui instaure une enveloppe sonore à propriétés fractales et un ressenti rythmique caractéristique ;
- Vue verticalement, la disposition des trous dans la forme crée successivement et rapidement des à-plats chromatiques, des gammes par tons, et d'autres modes qui lui donnent une signature harmonique propre.

Plus subjectivement, cette structure possède un ratio entre simplicité de conception et complexité engendrée qui en fait un bon candidat pour être perçue auditivement comme à la fois intelligible et intéressante.

[35] Fractale nommée en l'honneur du mathématicien polonais W. Sierpiński.

3. Pour la musique électronique

Nous ne pouvons clore cette série d'exemples sans aborder la possibilité de destiner la sortie d'un programme basé sur ScoreGen non à une interprétation humaine, mais à une réalisation électronique.

Si la bibliothèque est, nous l'avons vu, dédiée à la production de partitions, rien n'empêche de se servir des partitions générées pour produire des séquences de notes MIDI et les utiliser dans un séquenceur.

Après les efforts déployés en première partie pour justifier la conception d'un outil spécialement conçu pour écrire des *partitions*, cette remarque mérite une justification.

Les informations contenues dans un fichier MIDI (codant des notes avec un instant de début et de fin, des informations d'intensité sonore, quelques marqueurs relatifs au tempo et aux mesures, et des effets applicables) peuvent pour la plupart être considérées comme un *sous-ensemble* de celles figurant sur une partition de musique.

Ainsi, s'il est toujours possible de convertir une partition en un flot de données MIDI et de l'enregistrer dans un fichier, la conversion inverse montre en revanche une perte d'information massive : si les emplacements des sons dans le temps et leur durée sont bien conservés, la partition sera privée de toute autre indication (articulations, liaisons, indications textuelles), le rythme aura perdu son écriture naturelle, etc. Surtout, la *structure* d'un fichier MIDI est pauvre par rapport à celle d'une partition : il s'agit d'une représentation temporelle linéaire, sans hiérarchie forte entre les événements.

Il est donc bien plus pertinent de travailler sur la riche représentation offerte par les partitions, quitte à en effectuer ensuite une *réduction* au format MIDI[36] lorsque cela est nécessaire.

[36] Précisons toutefois que la présentation sous forme de partition est plus pauvre que la représentation au format MIDI sur un point : elle ne peut stocker que (plus ou moins) huit niveaux d'intensités de notes, indiqués par les nuances (***ppp***, ***pp***, ***p***, ***mp***, ***mf***, ***f***, ***ff***, ***fff***), là où le MIDI en possède, comme pour tout autre paramètre, 128. Cette limitation pourra être levée dans une prochaine version de la bibliothèque ScoreGen en attribuant aux objets *Note* un attribut optionnel de « vélocité » (suivant la terminologie employée dans le contexte du standard MIDI) et en permettant d'exporter les objets *Score* directement sous forme de fichiers au format MIDI.

D'ailleurs, le passage par la forme partition avant l'export vers MIDI permet éventuellement d'opérer certaines modifications du matériau dans un logiciel de gravure musicale avant son importation comme un bloc de notes à l'intérieur d'un logiciel de séquençage.

Les deux pièces électroniques fournies en annexes ont été conçues en utilisant la bibliothèque ScoreGen. Nous vous proposons de les écouter avant de poursuivre :

Dans la première, ce sont les successions de notes rapides groupées en mouvements ascendant irréguliers qui ont été engendrées algorithmiquement ; le programme de synthèse fournit aussi une voix ne possédant que les notes extrémales de ces mouvements, permettant d'utiliser un autre instrument virtuel pour les souligner.

Dans la seconde, c'est l'ensemble du cheminement harmonique qui a été généré en utilisant une méthode presque identique à celle expérimentée dans la pièce pour clarinette et piano ; le programme offre toutefois une option permettant de choisir le type de mode employé, et la pièce électronique générée utilise d'abord des modes pentatoniques, puis des modes à sept sons.

D. Évaluation

Nous terminons cette troisième partie, consacrée aux exemples de réalisations basées sur la bibliothèque ScoreGen, objet de ce mémoire, par une tentative d'évaluation de la bibliothèque au vu des objectifs que nous lui avons assignés.

1. Atteinte des objectifs fixés

Les nombreux exemples d'utilisation que nous avons présentés dans les dernières pages illustrent plusieurs réussites de la bibliothèque ScoreGen.

Ils montrent que la représentation des éléments musicaux choisie pour structurer les objets de ScoreGen est *(1)* assez extensive pour permettre la génération d'un très large choix d'objets musicaux, et *(2)* suffisamment bien architecturée pour offrir une utilisation conforme à l'idée intuitive que l'on peut se faire d'une partition musicale. Ces deux remarques valident *a posteriori* les choix qui ont été faits concernant l'arborescence des éléments musicaux, et plus largement la structuration de la musique écrite sous forme de partitions.

D'autre part, l'usage fait de ScoreGen au sein de projets ambitieux démontre une grande flexibilité dans les manipulations possibles des matériaux musicaux générés ; il montre la pertinence des types d'objets proposés à l'utilisateurs, qu'ils représentent des éléments de la partition (*Pitch*, *TimeSignature*), des structures de manipulation intermédiaires (*Group*, *Block*) ou des aides à la composition algorithmique (*Seq*, *Generator*, *Markov*).

En offrant un environnement entièrement textuel, la bibliothèque ScoreGen montre qu'il n'est pas nécessaire de passer par des outils graphiques lourds, contraignants et potentiellement sujets à obsolescence pour générer des partitions ; elle s'affranchit de tout logiciel hôte en acceptant comme environnement n'importe quel compilateur de C++ sur n'importe quelle plateforme. Enfin, elle répond à l'objectif d'interopérabilité fixé dès le début de ce travail, en offrant comme format de sortie un type de fichiers largement supporté et se prêtant facilement à l'ouverture dans des éditeurs de partitions, des séquenceurs ou d'autres logiciels permettant leur conversion vers d'autres formats.

Les exemples d'utilisation montrent que la bibliothèque rend possible un usage au sein de programmes relativements concis[37], notamment grâce aux fonctions-raccourcis *q* et aux fonctionnalités de conversion automatique entre types aux structures compatibles.

Signalons enfin que l'usage de ScoreGen n'occasionne à notre connaissance plus de bugs[38] autres que ceux que l'utilisateur peut provoquer par ses propres erreurs.

[37] Lorsque les codes employés possèdent des longueurs, cela est bien plus souvent dû à la verbosité du langage C++ lui-même (*boilerplate code*) qu'à une lourdeur de la bibliothèque. En ce sens, l'utilisation de ScoreGen à l'aide d'un autre langage comme Python devrait aboutir à des codes spécialement concis.

[38] Sur, au moins, les vingt dernières partitions générées. Cet état de fait est le résultat d'une longue période de débogages, conduits au fur et à mesure du développement de la bibliothèque et à mesure qu'elle était utilisée pour produire de nouvelles partitions. Un programme sans erreur de conception

Terminons en constatant que la complexité des objets générés avec ScoreGen semble pouvoir être repoussée à l'infini, ou du moins largement plus loin que ce qu'il est raisonnable d'attendre d'environnements de CAO basés sur la programmation graphique. La production de l'exemple pour clarinette et piano nous a pris une demi-journée et un fichier de code, la création de la chaîne de Markov de la pièce *Trailer!* seulement quelques heures ; or rien n'empêche de faire croître largement la taille des codes sources et la complexité des structures musicales sous-jacentes – au prix peut-être d'une conception plus raffinée et prudente du code source, notamment en multipliant les fichiers et le nombre de types personnalisés intermédiaires. Il est aussi à noter que portée par la rapidité du C++ dont le paradigme compilatoire permet d'exploiter toute la puissance du processeur, la vitesse d'exécution des programmes n'a jamais été un frein à une quelconque réalisation – nous n'avons pas encore produit de programme utilisant ScoreGen et dont l'exécution ne fût immédiate[39].

2. Développements souhaités

Malgré ces constats, l'évidence nous force à admettre que ScoreGen est un projet en cours de construction, qui ne saurait en son état actuel être considéré comme achevé.

Certaines fonctionnalités *extrêmement basiques* des partitions musicales sont encore absentes de la bibliothèque ; ainsi l'absence provisoire de la possibilité d'inclure une simple ligne de *crescendo*, pudiquement tue dans toutes les sections précédentes, est incompatible avec l'ambition de la bibliothèque à couvrir l'éventail des notations les plus universelles que peuvent présenter les partitions[40].

---

n'existant pas, il est certain que des erreurs, subsistant dans les fonctionnalités de la bibliothèque qui ont encore été peu utilisées, seront débusquées par des utilisateurs futurs.

[39] Il est pourtant à noter que de nombreux choix de *design* de la bibliothèque ont été faits plus pour la fluidité de programmation que pour la vitesse d'exécution – il en va ainsi des objets non mutables et du recours systématique aux fonctions en *q* (qui nécessitent chacune une assez coûteuse analyse syntaxique).

[40] Lorsque nous avons eu besoin d'inclure des crescendos et decrescendos dans nos partitions, nous nous sommes jusqu'ici contenté d'utiliser les indications textuelles *cresc.* et *dim.* Cette facilité nous a pour l'instant dispensé d'implémenter la fonctionnalité de façon plus plus adaptée ; elle montre cependant aussi que les manques de la bibliothèque peuvent, pour certains, ne pas empêcher la production de signifiés dépassant ses capacités de formalisation, *via* l'emploi de textes.

De même, l'appel de certaines fonctionnalités évoluées de ScoreGen, possédant déjà leurs fonctions dédiées, se verra gratifié d'une décevante *NotImplementedException*. Ainsi, si la fonction permettant d'étirer les durées au sein d'un élément existe bel et bien, son implémentation est encore vide et lèvera une exception.

Enfin, certaines fonctionnalités sont régulièrement ajoutées à ScoreGen lorsqu'il apparaît à l'occasion d'une création qu'une fonctionnalité d'intérêt général manque à la bibliothèque. Si de nombreuses fonctions sont ainsi apparues au cours du temps dans la bibliothèque, il est encore bien des manipulations d'intérêt évident qui doivent encore, pour l'instant, être implémentées par l'utilisateur final – ainsi de la fonction *midiToFreq*, que nous avons dû réécrire pour l'un des exemples ci-dessus alors que sa place naturelle est dans la section *Math for music*, dans une hypothétique classe consacrée aux calculs sur les fréquences ou aux conversions.

Ces manquements attestent de la difficulté pour un programmeur seul de mettre sur pied un travail aussi vaste que la conception d'une bibliothèque généraliste de manipulation des partitions musicales. Cet effort, qu'ont dû réaliser en leur temps les équipes de programmeurs professionnels ayant contribué à la création des principaux logiciels de gravure musicale, demanderait de permanents regards extérieurs, une aide pour le codage des fonctions les plus difficiles[41], des discussions de spécialistes pour les choix de design logiciel, des relectures du code à la recherche de sections buguées ou implémentées de manière inefficace.

C'est pourquoi le principal développement attendu pour ScoreGen est l'ouverture du code à d'autres programmeurs grâce à la création d'un dépôt sur Git[42].

L'autre grand chantier de ScoreGen est le projet de refonte du code source en C++ pur, c'est-à-dire sans usage du modèle C++/CLI. Cela passe notamment par l'écriture d'un destructeur pour chaque classe, par une migration vers les tableaux et chaînes de caractères natifs du C++, et par une conformation du code à la norme de séparation des

[41] Ainsi, la fonction de la bibliothèque qui permet de couper un élément quelconque à une position temporelle donnée pourrait donner du fil à retordre à n'importe quelle équipe de programmeurs professionnels. Nous en voulons pour preuve l'erreur que le logiciel de gravure musicale Sibelius affiche lorsqu'un utilisateur tente de couper une mesure au milieu d'un multiplet – un cas que ScoreGen sait gérer.

[42] Git, et sa plateforme GitHub, sont l'outil le plus largement utilisé et certainement le plus puissant pour le partage de codes sources et la programmation collaborative.

fichiers de définitions et d'implémentations. Cette réécriture est un projet titanesque mais nécessaire ; en effet, même si le C++/CLI est largement supporté et exportable, il s'agit de la dernière et pernicieuse dépendance à laquelle ScoreGen souscrit. Or au vu des objectifs d'interopérabilité et de durabilité affichés par la bibliothèque, il est impensable qu'elle persiste avec une telle servitude dans ses versions suivantes.

Les autres développements souhaités, plus modestes, concernent :

- l'ajout de nouvelles structures (peut-être séparation d'une *Note* monodique et d'un objet *Chord* à plusieurs hauteurs, création de nombreuses nouvelles suites dans *Seq*, ajout de nouvelles *TimeVariable* basées sur des fonctions mathématiques définies par l'utilisateur ou sur des valeurs lues dans des fichiers, etc.),
- la rédaction d'une documentation exhaustive suivant les normes en vigueur,
- la compilation pour d'autres plateformes que Windows,
- l'adjonction de formats d'export alternatifs aux côtés du MusicXML (à commencer par LilyPond et MIDI)[43],
- et enfin les tests concernant l'utilisation de la bibliothèque avec d'autres langages que le C++.

[43] Ce développement viserait entre autres à assurer la pérennité de ScoreGen au cas où le format de sortie actuel deviendrait obsolète.

Conclusion

Motivé par la création de ScoreGen, bibliothèque logicielle en C++ dédiée à la CAO, à la génération de partitions musicales et à la composition algorithmique, ce travail de recherche rend compte des problématiques rencontrées et des solutions apportées lors de sa mise au point. Pour cela, nous avons fait état des objectifs de notre démarche, de sa nécessité au vu des outils existants, et des orientations choisies pour y parvenir ; nous avons offert une présentation détaillée de la bibliothèque, partant de sa structure générale pour ensuite parcourir la hiérarchie des éléments musicaux qui la sous-tend. Enfin, nous avons proposé un choix d'exemples d'utilisation, couvrant aussi bien des démonstrations de fonctionnement que des créations plus complexes et des codes ayant servi dans le cadre de compositions de grande envergure ; exemples dont nous nous sommes servi pour évaluer la viabilité de notre outil, autant pour en confirmer l'utilisabilité et la flexibilité que pour signaler les carences qu'il lui reste à combler.

Ce mémoire montre ScoreGen comme une ressource puissante et fonctionnelle. Elle répond aux critiques que nous avons adressées de façon étayées aux outils de programmation visuelle qui dominent le monde de la CAO, en permettant des programmes lisibles, commentables, modulaires et réutilisables. Elle offre par rapport à ces autres environnements la possibilité de réaliser avec facilité des algorithmes complexes impliquant par exemple des boucles, des récursions ou des types de données personnalisés.

Nous avons montré que les programmes écrits textuellement à l'aide de ScoreGen plutôt que sous forme de patchs graphiques leur faisait gagner en concision (limite de Deutsch), et que l'implémentation de la bibliothèque avait cherché à consolider cet avantage par l'usage de syntaxes raccourcies et d'un jeu de conversions implicites.

Enfin, l'usage montre que les avantages de ScoreGen incluent la vitesse d'exécution, l'extensibilité de la bibliothèque (notamment par héritage de classes) lorsqu'une tâche n'est pas implémentée nativement, ainsi que la possibilité d'utiliser les milliers de bibliothèques en C++ existantes au sein de programmes de CAO.

Quoique d'abord centré sur un outil, ce travail a été l'occasion de nous questionner sur bien des concepts appartenant aux domaines entre eux connexes de la CAO et de la composition algorithmique.

Ainsi, nous avons écrit plus haut dans ce travail que la fertilité de la pensée algorithmique nous semblait en partie provenir du fait que *les pièce générées en suivant un processus algorithmique tendent à surpasser l'idée qui leur a donné naissance.*

Si nous avons souhaité en conclusion de notre travail réaffirmer cette observation fondamentale, c'est qu'en elle se trouve à notre sens la justification même du concept de composition algorithmique et de sa fécondité artistique.

Nous voulons par cette considération nous enthousiasmer du fait que les objets musicaux que peut générer un jeu de règles bien choisi n'y soient pas entièrement contenus ; que la naissance d'un objet musical unique engendré par un processus dépasse, parce qu'il peut alors être perçu avec la subjectivité d'un musicien et non plus avec l'objectivité d'un programmeur, le procès dont il est né, faisant subitement expérience ; ou encore que le compositeur, qui fixe pourtant lui-même la totalité des processus régissant un programme de composition algorithmique, soit le premier à pouvoir se laisser surprendre par l'objet musical que celui-ci engendre – ainsi lorsque nous avons mis en musique l'objet bien connu du triangle de Sierpinski, pour une première fois l'entendre.

Cette transcendance de l'objet sonore par rapport à son processus de mise à jour, et le nouvel itinéraire qu'elle ouvre vers une possible sérendipité musicale, ne nous semblent d'ailleurs pas tout à fait propre à la musique électronique – elle rappelle qu'au-delà des esthétiques, l'œuvre musicale est le fruit d'un processus appelé à s'effacer devant elle. Nous pourrions cependant tout à fait envisager de la proposer comme un nouveau critère de définition de la composition algorithmique et de ses pratiques – cela ferait l'objet d'un autre travail.

Enfin, à l'heure où la plus grande partie de la recherche et de la médiatisation associées à l'informatique théorique se concentrent sur les technologies émergentes dites d'« intelligence artificielle », sur le *deep learning* et sur les réseaux de neurones profonds, notre travail nous apparaît à la fois « dans le vent » – en ce qu'il abonde dans

le sens d'une société intégralement régentée par les ordinateurs – et paradoxalement à contre-courant de celles qui nous paraissent parfois être les idées du siècle, puisque notre travail vise à employer l'informatique à une fin issue d'une volonté artistique, éventuellement destinée à des interprètes humains, plutôt que pour explorer ce que l'intelligence artificielle peut nous apporter sans être sûrs d'en ressentir au préalable le besoin. Nous nous sommes donc grandement plu à arpenter la voie déjà en tous sens sillonnée de l'algorithmique pure, dans laquelle l'humain fixe les règles et la machine les suit – et avons eu à cœur de montrer que la règle bien choisie reste au début du XXI^{ème} siècle, après cinquante ans d'algorithmique informatique, une source féconde d'émerveillement.

Bibliographie

AGON Carlos, ASSAYAG Gérard, BRESSON Jean. *The OM composer's book*. Delatour, 2006, 292 p.

BRESSON Jean. *Composition assistée par ordinateur : techniques et outils de programmation visuelle pour la création musicale*. Université Pierre et Marie Curie, 2017.

CAIRES Carlos Miguel Marques da Costa. *Algorithmes de composition : exemples d'outils informatiques de génération et manipulation du matériau musical*. Bibliothèque numérique Paris 8, 2006.

HERVE, Gaëtan. *De la formalisation des procédés compositionnels à travers le patch de CAO*. IRCAM, Journées d'Informatique Musicale, 2011.

LEACH Jeremy et FITCH John. *Nature, Music, and Algorithmic Composition*. Computer Music Journal, été, vol. 19, n°2, 1995, pp. 23-33.

POTTIER Laurent. *Le calcul de la musique : composition, modèles et outils*. Publications de l'Université de Saint-Étienne, 2009, 477 p.

SUPPER Martin. *A Few Remarks on Algorithmic Composition*. Computer Music Journal vol. 25, n°1, *Aesthetics in Computer Music*. Spring, 2001, pp. 48-53.

TRUCHET Charlotte, AGON Carlos, ASSAYAG Gérard, CODOGNET Philippe. *CAO et contraintes*. IRCAM, Journées d'Informatique Musicale, 2001.

Spécifications et documentation des langages et outils cités :

MusicXML : https://www.w3.org/2021/06/musicxml40/

C++ : https://en.cppreference.com/w/

OpenMusic : https://support.ircam.fr/docs/om/om6-manual/co/OM-User-Manual.html

PWGL : https://ccrma.stanford.edu/courses/124/resources/PWGL-book.pdf

Csound : https://csound.com/docs/manual/index.html

LilyPond : https://lilypond.org/manuals.html

Table des illustrations

Table des annexes

Code source

Bibliothèque ScoreGen – dossier compressé

Adresse web d'accès aux annexes :

`http://antoinegabrielbrun.com/ressources/scoregen-memoire-de-recherche/`

Table des matières

Mémoire de : Antoine Gabriel BRUN

2ème cycle (Master), 2022-2023

Professeur référent : David Chappuis

Accompagnant méthodologique : Jean-Jacques Benaily

ScoreGen : développement d'une bibliothèque logicielle C++ pour la CAO, la génération de partitions musicales et la composition algorithmique

Abstract

Le paysage des logiciels de composition algorithmique est marqué par la suprématie des outils de programmation visuelle, originellement destinés à des compositeurs peu formés à la programmation informatique. Cette hégémonie a de quoi surprendre lorsqu'on considère les avantages, faisant consensus parmi les programmeurs, du code sur les patchs.

L'objectif de ce travail est de présenter ScoreGen, un outil que nous avons développé pour servir d'alternative textuelle aux autres outils de génération de partitions, et d'évaluer sa capacité à produire des codes lisibles, interopérables, concis, protégés de l'obsolescence et réutilisables ; nous avons dans ce but détaillé le fonctionnement de cette bibliothèque logicielle, exploré sa logique de représentation des partitions, et fourni de nombreux exemples simples et complexes de réalisations musicales.

Abstract

The landscape of algorithmic composition software is marked by the supremacy of visual programming tools, originally intended for composers with little knowledge in program writing. This hegemony is surprising when one considers the advantages of code over patches, which are widely accepted among programmers.

The aim of this work is to present ScoreGen, a tool we have developed as a textual alternative to other score generation tools, and to evaluate its ability to produce readable, interoperable, concise, obsolescence-resistant and reusable code; to this end, we have detailed the functioning of this software library, explored its score representation logic, and provided numerous simple and complex examples of musical realisations.