UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE INGENIERÍA

Representación de instrumentos musicales en modelos matemáticos para la generación fidedigna de Música

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

PRESENTA
Joaquín Gustavo Espino de Horta

DIRECTOR DE TESIS
Dr. Rogelio Alcántara Silva

Ciudad Universitaria, Cd. Mx., Agosto 2024

INTRODUCCIÓN

La propuesta hoy presentada comenzó como una necesidad no resuelta por las herramientas disponibles y demandas hechas por los proyectos de escasos recursos presupuestarios y personal, mejor conocidos como proyectos independientes en los ámbitos de producción en software destinados a la recreación y el ocio como lo son juegos de video y multimedia, a su vez, disponer de una herramienta adicional a los ya indispensables usos del procesador de textos, hojas de cálculo, controlador de diapositivas, entre otros integrados y dependientes exclusivamente del equipo de cómputo.

Si bien, este proyecto no es pionero en su finalidad, en lo que a producción de origen totalmente computacional es referido, la composición de obras auditivas en formatos digitales sigue siendo parcialmente dependiente de la captura y edición de la información proveniente de los instrumentos musicales físicos capturados por periféricos especializados, donde en estos se requiere un compromiso absoluto en la disciplina artística, así como la ingeniería propia involucrada en el muestreo y alteración en las señales de audio.

Existen algunas herramientas de porte comercial dedicadas a la generación del audio sin la demanda de un instrumento o captura de este, aun así, estas no ha sido diseñadas en dotar a su producto de una calidad definitiva, pues ha sido concebida como una referencia o prueba de concepto, pues el método de producción basado en la adición estratégica con muestras pre-grabadas de notas musicales, tienen una clara limitante en cuanto a tiempo, diversidad en variaciones de interpretación entre muchos aspectos no deseados en vista a un producto final, pues a nivel de consumidor, no sería de su agrado la escucha de sonidos tan parecidos y sin sus variaciones.

Esta problemática no refiere a una urgencia en el mercado, aunque si aspira a la dar un nuevo abanico de opciones en cuanto la producción musical, una que permita cubrir la sincronización con el tiempo así como la correcta interpretación de sonidos hechos en instrumentos musicales, en otras palabras, dar a los compositores sean aspirantes o expertos la posibilidad de aumentar e inaugurar una mayor y más eficiente producción, brindando las bondades que nos ha traído la programación informática, referentes al propio soporte, el trabajo re-utilizable incluso en cuanto a un nuevo paradigma de almacenamiento y respuesta computacional adecuada a los temas sonoros.

Véase un anuncio publicitario compuesto con los *jingles* establecidos de la corporación, el repertorio cultural en potencia al disponer a cada usuario las herramientas para componer la futura obra maestra en una industria saturada por el monopolio de tendencias actuales, surgidas en culturas y lugares distintos, todo disponible desde la misma plataforma y capacidades de la máquina capaz de ejecutar un editor de textos comercial.

Después de todo, ya existen los conjuntos de herramientas totalmente independientes para la elaboración de sus respectivos productos, hablando también de aquellos más complejos como obras audiovisuales, construcción de modelos en tres dimensiones o combinando todos estos para la elaboración de aplicaciones gráficas orientadas al entretenimiento, mejor conocidos como *videojuegos*

VISIÓN

La siguiente imagen muestra una representación simplificada sobre la propuesta de solución al completo, donde se tiene planeadas todas sus funcionalidades y operaciones al menos de cara a un lanzamiento hacia el público interesado, más no uno comercial.

Están contempladas 5 grandes aspectos para el desarrollo de una aplicación integral, aunque se aclara desde este punto, solo se centrará en un módulo, el más significativo en cuanto una nueva propuesta se refiere. Siendo el módulo de *Tuner* (afinador).

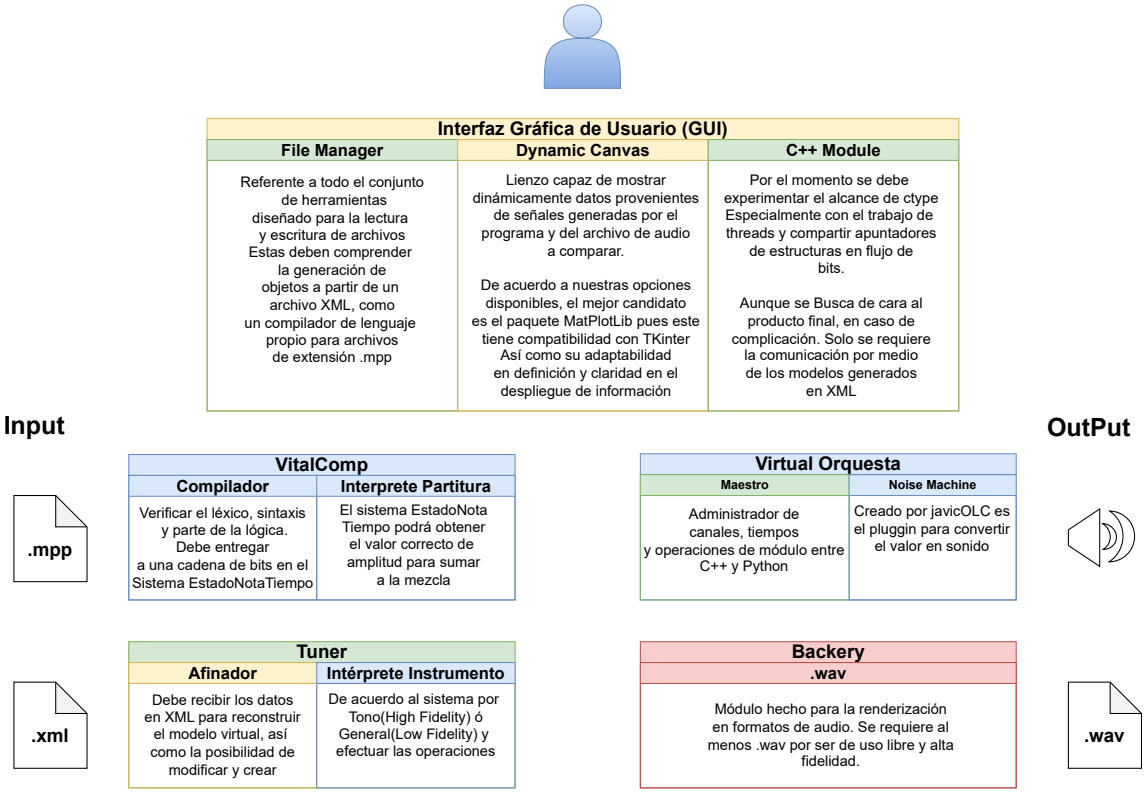

Figura 1: Diagrama General del Proyecto Completo

MÓDULOS

Profundizando en los módulos mostrados anteriormente:

Interfaz Gráfica de Usuario (GUI)

El enfoque comercial del proyecto, exige que este contenga por lo menos una visualización cercana a los estándares en herramientas de software, si bien, el público general cambia su tendencia en cuanto conocimientos informáticos básicos, sería un error cerrarnos de cara a una presentación como la manipulación para una terminal de consola. Se ha experimentado construir una GUI con el lenguaje *C++*, sin embargo no pudo encontrarse una biblioteca adecuada, mucho menos un resultado satisfactorio para delegar todo el proyecto a un único lenguaje de programación, por lo tanto, se trabajará con Python en el *FrontEnd*, es decir, todo contacto con el usuario final.

File Manager

Ya que nuestro propósito es la generación de archivos cuya información pueda interpretarse por un estándar orientado a audio, así como la preservación del trabajo en un lenguaje de programación, es imperativo usar varios sistemas de lectura y escritura de archivos en ambos lenguajes a utilizar, siendo *Python* y *C++*, estos contienen una biblioteca en sus paquetes fundamentales, siendo la función *open* y *fstream* respectivamente. Por otra parte, se necesitará de una biblioteca especializada en convertir la información de un texto etiquetado como lo es el formato XML. No solo es un estándar que permite escalar las funciones del proyecto a futuro con otros productos de software, sino que al ser un formato simple y en cierto punto, indicativo para ser editado manualmente.

Dynamic Canvas

Si bien, esta parte podría únicamente ser útil en lo que respecta la construcción del módulo afinador. Puede utilizarse en tiempo real la representación gráfica de la información generada, desde los armónicos que participan en la mezcla del sonido, una visión más amigable de cara al usuario de los eventos programados, entre otras cosas. Ya que esto está ligado a la GUI así como el afinador, su desarrollo sería exclusivamente en Python, optando por la opción que ofrece el paquete de MatPlotLib por su compatibilidad con TKinter

C/C++ Module

Debido a las características por implementar, es conveniente dejar a Python como aquel que tenga el ejecutable inicial, así como ser el eje de las herramientas que conlleve el proyecto. Este ofrece una opción denominada como *ctype* el cual permite convocar código en *C/C++* colocando y devolviendo datos primitivos, aún no se ha experimentado del todo, pues se requiere de la certeza y técnicas en cuanto el envío de apuntadores para arreglos de datos, estructuras específicas así como su correcto funcionamiento con múltiples hilos de ejecución.

VitalComp

Este módulo comprende el lenguaje de programación propuesto para la codificación de melodías en cuanto Tonos y Tiempos se refiere, así como su manejo en melodías. El nombre es en conmemoración a un profesional en la música, conocido personal, *Vitalis Elrich* a quien se le había consultado la idea inicial y gracias al estudio de necesidades y herramientas actuales en el mercado, se sugirió la propuesta sobre un lenguaje de programación.

Compilador

El compilador estará compuesto por 2 fases (independientes al análisis léxico, semántico y lógico), pues se plantean 2 paradigmas distintos en un mismo formato, pues por un lado tendríamos las necesidades logísticas en cuanto a la sincronización, designación de tiempos, incluso la posibilidad de implementar una aritmética y lógica primordiales con el fin de hacer compatible con otros códigos. Para su construcción, se emplearán los recursos de *Flex* y *Bison* para las primeras etapas, donde se obtendrá una cadena atómica verificada en la que se interpretarán parcialmente los comandos y símbolos para la adecuación del ambiente al momento de recibir la segunda fase, esta consistirá en reconocer elementos melódicos como el nombre de las notas(pueden ser católicas o protestantes), comandos referentes a acordes específicos, así como la duración, varianza en semitonos entre otras características que pueda contener una partitura y ser representada en comandas. Para este último conviene manejarse en un detector de símbolos para la reducción atómica y el complemento en una cadena de Bytes, por medio de un estado universal que tenga tiempos por defecto así como los propios modos... Así estamos solucionando el uso dinámico pues al ejecutarse condicionalmente, la computadora podrá re-compilar las partes móviles

Intérprete

Ajustando el proyecto original del usuario *Javidx9/OneLoneCoder*, se parte de modificar la estructura *sequencer*, cuya función principal es la generación automática de información en tiempos y notas para que sean añadidas en el mecanismo principal. Si bien, su versión está limitada a una sola característica, pues fue diseñada como un efecto cíclico de batería. Rescatando el concepto abstracto del secuenciador, es posible modificar a un sistema de interpretación por cadenas de Bytes, inspirado en el *pipeline de renderizado* usado en procesos gráficos de *OpenGL*. Esta consiste en la toma de 3 Bytes con información de tipo Estado, Timbre y Duración, por cada Tempo en secuencia de melodía, es decir, que clase de sonidos, su tiempo en reproducción y el estado técnico para poder formar acordes, silencios o incluso solicitar una interpretación especial(fuera de mayores ó bemoles) en relación con otras notas. Cabe abordar una sugerencia en cuanto la optimización del caso estático, pues si la máquina no contiene ninguna variación, porque de ser el propósito: generar un archivo en memoria de almacenamiento, entonces, no debería compartir el mismo mecanismo en tiempo real, ahorrando recursos de presentación y otorgando una mayor eficiencia de uso.

Virtual Orquesta

Este es un módulo enfocado al procesamiento central de la información, pues estará diseñado para invocar la interpretación de las instrucciones y ajustes dados por el usuario, administración de los múltiples hilos dedicados, pausa, reinicio, incluso a la captura en tiempo real de notas generadas por el usuario mediante una entrada estándar así como la posibilidad de escalar a un instrumento especializado.

MasterChord

Siendo esta parte administrativa, deberá cumplir con una facilidad de funciones que puedan ser citadas como servicios desde la GUI así como la respuesta de información siendo instrumentos, posición de notas, tiempo real o cualquier información correspondiente a una visualización con propósitos artísticos o técnicos. Si bien las funciones gráficas son prescindibles, se deja abierta la posibilidad de escalar al producto final.

NoiseMachine

Esta es una cabecera hecha por *Javidx9/OneLoneCoder* que nos permite la comunicación con el hardware hecho para la reproducción de audio mediante la solicitud del tiempo en valor de la amplitud otorgado por una función asignada. Esta pieza limita el proyecto a plataformas con sistema operativo Windows 7 en adelante, con compatibilidad para arquitectura de x32 bits.

Backery

Recordando los propósitos iniciales, es evidente la necesidad de respaldar los archivos en memoria de almacenamiento, por lo tanto necesitamos de alguna forma, guardar la información en un estándar para la reproducción de audio.

Tuner

Generador de Señales

Virtual Instrument

Para la realización de estos se ha tomado como base un proyecto para emulación de instrumentos del talentoso ingeniero inglés conocido en sus redes sociales electrónicas como *Javidx9*. Su proyecto *olcNoiseMaker* sienta los principios para estos requisitos:

Maestro del Audio

```
FTYPE MakeNoise(int nChannel, FTYPE dTime){
        unique_lock<mutex> lm(muxNotes);
        FTYPE dMixedOutput = 0.0;
        // Iterate through all active notes, and mix together
        for (auto &n : vecNotes){
                bool bNoteFinished = false;
                FTYPE dSound = 0;
                // Get sample for this note by using the
                // correct instrument and envelope
                if(n.channel != nullptr)
                        dSound = n.channel->sound(dTime, n, bNoteFinished);
                // Mix into output
                dMixedOutput += dSound;
                if (bNoteFinished) // Flag note to be removed
                        n.active = false;
        }
        // Woah! Modern C++ Overload!!!
        //Remove notes which are now inactive
        safe_remove<vector<synth::note>>(vecNotes,
                [](synth::note const& item)
                { return item.active; });
        return dMixedOutput * 0.2;
}
```

Plantilla de Instrumento

```
        struct instrument_base{
                FTYPE dVolume;
                synth::envelope_adsr env;
                FTYPE fMaxLifeTime;
                wstring name;
                virtual FTYPE sound(const FTYPE dTime, synth::note n, bool &bNoteFinished) = 0;
        };
        struct instrument_bell : public instrument_base{
                instrument_bell(){
                        env.dAttackTime = 0.01;
                        env.dDecayTime = 1.0;
                        env.dSustainAmplitude = 0.0;
                        env.dReleaseTime = 1.0;
                        fMaxLifeTime = 3.0;
                        dVolume = 1.0;
                        name = L"Bell";
                }
                virtual FTYPE sound(const FTYPE dTime, synth::note n, bool &bNoteFinished){
                        FTYPE dAmplitude = synth::env(dTime, env, n.on, n.off);
                        if (dAmplitude <= 0.0) bNoteFinished = true;
                        FTYPE dSound =
                                + 1.00 * synth::osc(dTime - n.on, synth::scale(n.id + 12), synth::OSC_SINE, 5.0, 0.001)
                                + 0.50 * synth::osc(dTime - n.on, synth::scale(n.id + 24))
                                + 0.25 * synth::osc(dTime - n.on, synth::scale(n.id + 36));
                        return dAmplitude * dSound * dVolume;
                }
        };
```

Acceso a Hardware

```
FTYPE(*m_userFunction)(int, FTYPE);
unsigned int m_nSampleRate;
unsigned int m_nChannels;
unsigned int m_nBlockCount;
unsigned int m_nBlockSamples;
unsigned int m_nBlockCurrent;
T* m_pBlockMemory;
WAVEHDR *m_pWaveHeaders;
HWAVEOUT m_hwDevice;
thread m_thread;
atomic<bool> m_bReady;
atomic<unsigned int> m_nBlockFree;
condition_variable m_cvBlockNotZero;
mutex m_muxBlockNotZero;
atomic<FTYPE> m_dGlobalTime;
// Handler for soundcard request for more data
void waveOutProc(HWAVEOUT hWaveOut, UINT uMsg, DWORD dwParam1, DWORD dwParam2){
        if (uMsg != WOM_DONE) return;
        m_nBlockFree++;
        unique_lock<mutex> lm(m_muxBlockNotZero);
        m_cvBlockNotZero.notify_one();
}
// Static wrapper for sound card handler
static void CALLBACK waveOutProcWrap(HWAVEOUT hWaveOut, UINT uMsg, DWORD dwInstance, DWORD dwParam1, DWORD dwParam2){
        ((olcNoiseMaker*)dwInstance)->waveOutProc(hWaveOut, uMsg, dwParam1, dwParam2);
}
// Main thread. This loop responds to requests from the soundcard to fill 'blocks'
// with audio data. If no requests are available it goes dormant until the sound
// card is ready for more data. The block is fille by the "user" in some manner
// and then issued to the soundcard.
void MainThread(){
        m_dGlobalTime = 0.0;
        FTYPE dTimeStep = 1.0 / (FTYPE)m_nSampleRate;
        // Goofy hack to get maximum integer for a type at run-time
        T nMaxSample = (T)pow(2, (sizeof(T) * 8) - 1) - 1;
        FTYPE dMaxSample = (FTYPE)nMaxSample;
        T nPreviousSample = 0;
        while (m_bReady){
                // Wait for block to become available
                if (m_nBlockFree == 0){
                        unique_lock<mutex> lm(m_muxBlockNotZero);
                        while(m_nBlockFree == 0) // sometimes, Windows signals incorrectly
                                m_cvBlockNotZero.wait(lm);
                }
                // Block is here, so use it
                m_nBlockFree--;
                // Prepare block for processing
                if (m_pWaveHeaders[m_nBlockCurrent].dwFlags & WHDR_PREPARED)
                        waveOutUnprepareHeader(m_hwDevice, &m_pWaveHeaders[m_nBlockCurrent], sizeof(WAVEHDR));
                T nNewSample = 0;
                int nCurrentBlock = m_nBlockCurrent * m_nBlockSamples;
                for (unsigned int n = 0; n < m_nBlockSamples; n+=m_nChannels){
                        // User Process
                        for (unsigned int c = 0; c < m_nChannels; c++){
                                if (m_userFunction == nullptr)
                                        nNewSample = (T)(clip(UserProcess(c, m_dGlobalTime), 1.0) * dMaxSample);
                                else
                                        nNewSample = (T)(clip(m_userFunction(c, m_dGlobalTime), 1.0) * dMaxSample);
                                m_pBlockMemory[nCurrentBlock + n + c] = nNewSample;
                                nPreviousSample = nNewSample;
                        }
                        m_dGlobalTime = m_dGlobalTime + dTimeStep;
                }
                // Send block to sound device
                waveOutPrepareHeader(m_hwDevice, &m_pWaveHeaders[m_nBlockCurrent], sizeof(WAVEHDR));
                waveOutWrite(m_hwDevice, &m_pWaveHeaders[m_nBlockCurrent], sizeof(WAVEHDR));
                m_nBlockCurrent++;
                m_nBlockCurrent %= m_nBlockCount;
        }
}
```

0.1. Resultados Esperados

El producto en su finalidad debe ser capaz de:

- Bajo en consumo de recursos computacionales
- Utilizable en muchos entornos de programación
- Facilidad en su comprensión para el usuario
- Simulación de instrumentos musicales diversos
- Diseño escalable al número de funciones y modos
- Grabar los formatos más utilizados en la actualidad

Respecto a la funcionalidad, esta deberá estar accionada desde una entrada estándar en cadena de texto. Dando como una salida la reproducción del sonido esperado, como su posibilidad de almacenarse en los formatos objetivo.

De esta manera, se espera un uso por parte de usuarios desde aplicaciones sofisticadas con una interfaz gráfica de usuario agradable a la vista, como las opciones comerciales presentadas hasta la implementación en lenguaje de alto nivel. Se tiene como panorama el paradigma de interpretación y lectura secuencial gracias a sus propiedades fundamentales arraigadas en las características de computadoras comerciales accionadas por sistemas Linux ó Windows, así como su vista hacia Android.