Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Arquitectura para parametrización de atributos en sistema de tiempo real

Tesis de Licenciatura en Ciencias de la Computación

Matias A. Garcia Marset

Director: Dr. Esteban Mocskos
Director: Dr. Mariano Camilo González Lebrero
Buenos Aires, 2021

ARQUITECTURA PARA PARAMETRIZACIÓN DE ATRIBUTOS

Durante el siglo pasado, el desarrollo de instrumentos electrónicos acompañó grandes cambios en la forma de hacer música. Además de guitarras, bajos eléctricos y *samplers*, se exploraron técnicas computacionales para la elaboración de mejores sintetizadores buscando interpretar y concebir una gran variedad de sonidos.

La simulación numérica mediante diferencias finitas, por su parte, permite capturar las características de los instrumentos brindando, además, un alto nivel de flexibilidad a la hora de modificar propiedades inherentes del objeto simulado. En particular, la simulación de una cuerda está controlada por una gran variedad de parámetros (fricción, frecuencia, masa, entre otros) que permiten la posibilidad de experimentar diferentes sonidos mediante sus variaciones. Así, una determinada configuración podría generar el sonido de una cuerda de guitarra criolla y otra produciría el de una guitarra eléctrica. No obstante, esta técnica de simulación tiene un alto costo computacional ya que debe ejecutarse en tiempo real en base a las perturbaciones introducidas por el ejecutante, imponiendo un desafío computacional en sí mismo.

Este trabajo de tesis se enmarca en un proyecto interdisciplinario que tiene por objetivo crear herramientas de simulación de instrumentos musicales que puedan ser utilizadas por diferentes públicos (desde estudiantes hasta luthiers). Se busca obtener simulaciones fidedignas así como abrir la posibilidad de crear nuevos sonidos, facilitando la experimentación para lograr sonoridades nuevas o mejoradas. Esta exploración se realiza por medio de la modificación de los distintos parámetros que controlan la simulación y se debe lograr sin que estos cambios afecten a la fluidez de la simulación, representando todo un desafío, ya que la misma debe generar resultados en el orden de los microsegundos para evitar latencias, distorsiones o interrupciones. Es decir que cualquier cambio en las propiedades, por más grande o pequeño que sea, no debe escaparse de este límite. A su vez es imperativo contar con un protocolo de comunicación entre el usuario y el simulador tal que no limite la cantidad o tamaño de los parámetros de entrada removiendo cualquier tipo de restricción a la hora de experimentar con el instrumento.

En esta tesis extendimos un prototipo de simulador de cuerdas que admitía exclusivamente `MIDI` como entrada para que también acepte un nuevo protocolo capaz de transmitir todas las propiedades deseadas sin las limitaciones propias de `MIDI`. Para esto buscamos e implementamos una arquitectura que permite ampliar los parámetros que utiliza el sintetizador en forma dinámica, sin que esto produzca interrupciones en la simulación (o incumplimientos en los límites temporales del sistema).

Así fue que se adaptó el simulador a la arquitectura propuesta, utilizando un protocolo de actualización que admite realizar cambios a los distintos componentes del sistema, enriqueciendo la interacción del usuario con el sintetizador. Adicionalmente agregamos una interfaz más amigable, mediante una aplicación *Android*, para facilitar la explotación de la herramienta a instrumentistas manteniendo muy bajo impacto en el sistema (menor a 400 µs) y una correcta latencia (en el orden de 15 ms).

Palabras claves: Simulador, Propiedades Físicas, Sonidos, Síntesis, Guitarra, Cuerdas, Parametrización, Bluetooth, Android, Simulación Numérica.

Índice general

1. INTRODUCCIÓN

La música forma parte de la cultura desde sus orígenes. De la prehistoria a la actualidad, los instrumentos musicales han co-evolucionado con la sociedad, adecuándose a las necesidades expresivas de cada época. Durante el siglo pasado, el desarrollo de instrumentos electrónicos acompañó grandes cambios en la forma de hacer música [1]. Guitarras y bajos eléctricos, sintetizadores y *samplers* ampliaron el campo de los sonidos posibles permitiendo expresar una sociedad mecanizada. En estos desarrollos existe una marcada diferencia entre lo orgánico y lo mecánico.

En la actualidad, la industria musical cuenta principalmente con dos tipos de instrumentos electrónicos clasificados por la tecnología de generación de sonidos: los sintetizadores y los *samplers* [1].

El *sampler* [2] es un dispositivo que puede reproducir sonidos que fueron previamente grabados. Mezclas complejas de estos sonidos, así como efectos posteriores, le dan a este tipo de "síntesis" un alto grado de realismo. Sin embargo, el universo de sonidos posibles está determinado por las grabaciones realizadas previamente, limitando la capacidad expresiva. Esta limitación se vuelve más sobresaliente sobre todo en instrumentos en los cuales la interacción directa con el objeto vibrante permite modular el sonido con un alto grado de sutileza (como en guitarras, violines, vientos, etc.).

Por su parte, los sintetizadores generan sonidos a partir recetas algorítmicas de relativa simpleza. Suma de ondas, restas, modulaciones de frecuencia, son algunos ejemplos de técnicas utilizadas para generar el sonido. Este tipo de síntesis posee una versatilidad y nivel de expresión alto, pero los sonidos generados resultan claramente sintéticos y no se asemejan a los generados por instrumentos reales.

Este proyecto se sostiene en la exploración de otra vía de síntesis: la creación de una nueva paleta de sonidos utilizando modelado y simulación de la física de objetos.

La síntesis de sonido mediante simulación física tiene una larga historia. Un trabajo pionero en el modelado de cuerdas es el de Hiller y Ruiz [3] del año 1971 en el que simularon el sonido de una cuerda discretizando la ecuación de la onda para calcular el valor de un punto en una cuerda vibrando. En ese momento, el cálculo de cada punto llegó a tardar minutos. Sin embargo, con el correr de las décadas fueron apareciendo numerosos modelos y acercamientos que aportaron notables mejoras. Dentro de estos trabajos, se destaca, por su bajo uso de recursos computacionales y muy buena calidad, el algoritmo de Karplus-Strong [4], que luego evolucionó al llamado modelo de guía de ondas digitales (*Digital Waveguides*) [5, 6]. Este último método es el más utilizado en productos comerciales hoy en día.

En la actualidad, existen iniciativas académicas que buscan, desde el modelado numérico con técnicas de mayor precisión (entre ellas diferencias finitas), superar las limitaciones de los otros métodos, incluso se ha explorado el uso de las placas de video para acelerar los tiempos de cómputo. Un ejemplo de esto es el proyecto NESS [7] que incluye la utilización de una gran variedad de algoritmos para diferentes instrumentos (incluidos cordófonos) usando modelos físicos para la síntesis de sonido. Si bien este proyecto tiene una perspectiva similar al abordado en esta tesis, a diferencia del objetivo principal de este trabajo, no está orientado a la simulación en tiempo real.

Es así que, utilizando un sistema en tiempo real y el poder de cómputo de las placas

de video (GPU), surge el aliciente de utilizar esta capacidad en simulaciones de diferentes instrumentos y sus propiedades. Esta técnica permite capturar el comportamiento de instrumentos existentes, así como de otros novedosos aún no construidos e, incluso, permite explorar instrumentos que no podrían ser construidos físicamente. La clave para lograr esto se esconde en el desarrollo de un controlador que permita incorporar sutilezas de la interpretación del ejecutante, como presión de los dedos, modos de perturbación, etc.

De esta manera, por medio de la conjunción de un programa de simulación y un controlador acorde, se podrán expandir las posibilidades expresivas. Esta primera etapa del proyecto, en el que se enmarca la presente tesis, está orientada a los instrumentistas de cuerdas, pero este acercamiento tecnológico deja planteado el escenario para desarrollos posteriores en otras áreas (vientos, percusión, etc.) e, incluso, generar nuevas categorías de instrumentos.

1.1. Sistemas de tiempo real

En esta sección buscaremos dar una breve introducción sobre los sistemas de tiempo real con el foco en nuestra aplicación, sin ahondar en detalles, ya que existe abundante bibliografía que trata este tópico [8].

Los sistemas de tiempo real tienen como característica distintiva la de poseer restricciones temporales junto con sus condiciones de funcionamiento lógicas. Existen varias caracterizaciones de sistemas de tiempo real, pero una de la más difundidas es: *tiempo real duro* (*"hard real-time"*) en los cuales cualquier mínimo retraso invalida al sistema completamente y *tiempo real blando* (*"soft real-time"*) en donde los retrasos tienen un impacto negativo pero el sistema puede continuar su funcionamiento [9].

Siendo *hard* o *soft*, la principal restricción sigue siendo el tiempo de respuesta. Si de alguna u otra forma se retrasa, el programa genera resultados que, al no cumplir con los objetivos de tiempo de respuesta, podrían no ser útiles. Esto nos lleva a preguntarnos: ¿a cuánto corresponde un *retraso aceptable*?

La respuesta a esta pregunta está muy atada al tipo de salida esperada. Por ejemplo si estamos aplicando filtros en una cámara en tiempo real, la respuesta esperada es vídeo, entonces buscamos una frecuencia de *frames* mayor o igual a la detectada por el ojo (de 50 Hz a 90 Hz [10], de ser menor, el ojo empezaría a detectar efectos de congelamiento. Pero ésta no es la única restricción, además es necesario que coincida la imagen resultado con lo que se está capturando (es decir, si estoy viendo como resultado una persona acercándose, quiero que efectivamente esa persona se esté acercando y no que ya esté al lado mío), en otras palabras, que no haya retraso. Esto se refiere al tiempo total de respuesta del sistema, es decir, desde que recibe un *input* (la imagen de la persona acercándose) hasta que retorna el *output* (la misma imagen filtrada), a ese tiempo se lo llamaremos **latencia**.

De esta manera el tipo de salida del programa (*output*) nos da una buena referencia de los tiempos esperados del sistema, pero no es el único parámetro importante, ya que otro punto muy relevante es la entrada (*input*). Siguiendo el ejemplo anterior, no es lo mismo si capturo una imagen *High Definition* (HD) a si capturo una a 4K. En la primera voy a necesitar procesar 2 millones de píxeles (HD), mientras que en la segunda 8 millones de píxeles (4K). Con esto vemos que los tiempos de procesamiento pueden cambiar drásticamente dependiendo del tamaño de la entrada. Entonces otra pregunta que surge es ¿cuál es el límite soportado para la entrada sin que genere retrasos inaceptables? (donde "retraso aceptable" fue definido anteriormente y, claramente, depende de la aplicación).

Por último, imaginemos que la cámara no es parte del sistema de procesamiento, es decir, que hay una cámara que captura vídeo y la envía al dispositivo que procesa y muestra en pantalla la imagen filtrada. ¿Podría ser un problema la comunicación entre estos dispositivos?

Estas preguntas nos van a ayudar a orientar el presente trabajo que busca una arquitectura flexible (que permita agregar nuevos parámetros), eficiente (que, de haberlos, genere retrasos aceptables) y que tenga en cuenta que los parámetros provienen de un dispositivo externo. Dicho trabajo lo detallaremos a continuación.

1.2. Descripción del sistema

Esta tesis se encuadra dentro de un proyecto que busca el modelado de un instrumento cordófono que sea ejecutado en un sistema embebido. El mismo debe ser capaz de generar un sonido realista de forma totalmente configurable, con fines educativos y profesionales.

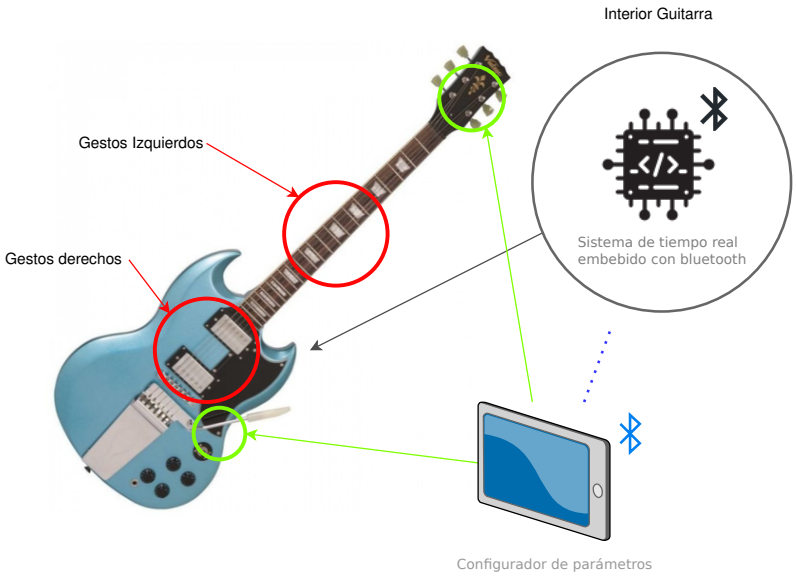


Fig. 1.1: Esquema general de componentes. El controlador contiene los sensores para interpretar los gestos del ejecutante (izquierdos y derechos), que son conectados al sistema embebido que se encuentra dentro del mismo. Éste se conecta por *Bluetooth* con un dispositivo externo que permite configurar todas las propiedades de la guitarra simulada (como la afinación, la fricción de las cuerdas, la palanca, etc)

El sistema embebido es el que contiene al sistema en tiempo real, encargado de sintetizar los sonidos basado en simulación numérica mediante diferencias finitas.

Este tipo de síntesis está muy poco explorado debido a su alto costo computacional. Recién ahora, y empleando procesadores de alto desempeño (cómo las Unidades de Procesamiento Gráfico, GPU), es posible realizarla en tiempo real para sistemas no triviales.

En el caso de considerar cuerdas, el objeto se construye mediante una serie de nodos unidos entre sí por un potencial armónico (de un resorte). Se le aplica una perturbación inicial (un dedo tocando) en algún lugar de la cuerda. La evolución del sistema se obtiene mediante la integración numérica de las leyes de movimiento de Newton aplicando un

factor de fricción para modelar el apagado del sonido. Finalmente, el sonido se obtiene de la posición o la velocidad en algún punto definido de la cuerda, esto puede pensarse como el lugar dónde estaría ubicado el *micrófono*.

Este planteo admite una gran cantidad de parámetros, la masa, la fricción y el potencial del resorte entre los nodos (que puede no ser perfectamente armónico), la posición de la perturbación, del micrófono, etc.

Este conjunto de parámetros definirán el comportamiento físico de la cuerda y por lo tanto su sonido, siendo una de las características sobresalientes de este tipo de sintetizadores que permite modelar una gran variedad de instrumentos, tanto reales cómo imaginarios.

A pesar del gran abanico de instrumentos que se pueden simular, el foco de esta tesis está puesto sobre aquellos que están basados en cuerdas. Se hará una pequeña descripción de las mismas en uno de los próximos apartados, pero la idea de instrumentos cordófonos es que aceptan muchas configuraciones y permite realizar distintos tipos de pruebas variando la carga y las condiciones experimentales. Por ejemplo, las condiciones que se pueden explorar van desde variar la fricción o masa de las cuerdas, hasta la distancia entre la cuerda y el diapasón, en el caso de una guitarra.

La motivación principal del proyecto es proveer a los músicos herramientas de calidad para la composición y análisis de instrumentos. Haciendo énfasis en el segundo, **en este trabajo desarrollaremos herramientas eficientes que permitan realizar esta experimentación, analizando el impacto de la parametrización de diferentes atributos en dicho sistema**.

1.2.1. Cuerdas

Parte del modelo físico consiste en representar una cuerda como una serie de nodos unidos entre sí tal como muestra la figura 1.2. Cada nodo tiene una masa que no tiene por qué ser igual a la de los vecinos, aunque por simplicidad se puede suponer que sí lo es. Es importante notar que cada nodo **solo se pueden mover en dirección vertical**.

Fig. 1.2: Esquema del modelo de cuerda. Cada punto representa un nodo que tiene una masa (no necesariamente igual a la de sus vecinos). Los nodos se unen entre sí por medio de potencial elástico: la fuerza que sienten es proporcional a la distancia. En nuestro modelo, los nodos solo tienen la libertad de moverse en sentido vertical.

Los nodos están unidos entre sí mediante un potencial elástico de modo que cada uno *siente* una fuerza proporcional a la diferencia de la posición vertical con cada nodo vecino. Cuanto mayor es la distancia con el vecino, mayor es la fuerza que siente el nodo para acercarse:

$$F_i = K \cdot (X_i - X_{i+1}) + K \cdot (X_i - X_{i-1}) \tag{1.1}$$

donde X es el vector que incluye las posiciones de todos los nodos usados para representar la cuerda, X_i, X_{i+1} y X_{i-1} son las posiciones de tres nodos consecutivos y a K se la denomina *constante del resorte* (representando la dificultad de comprimir un resorte).

Intuitivamente, $X_i - X_{i+1}$ representa qué tan separados están dos nodos (el i e $i+1$) en un momento dado y si el resorte es muy duro (K grande), mayor será la fuerza que mantenga los dos nodos en la distancia de reposo (para que no haya tensión en el resorte).

Para obtener un modelo que sea lo más real posible, es necesario tener en cuenta la pérdida de energía debida a la fricción con el aire y en los extremos de la cuerda por su interacción con el puente del instrumento. La fricción se puede aproximar cómo una fuerza proporcional a la velocidad y va en sentido opuesto a la misma.

$$F_i^{fricc} = -K_i^f V_i \tag{1.2}$$

donde K_i^f corresponde al coeficiente de fricción entre los dos materiales en contacto en el nodo i y V_i la velocidad del nodo i.

Luego, sumando todas las contribuciones a la fuerza (incluso eventualmente podrían considerarse otras en el modelo) y utilizando la segunda ley de Newton de movimiento:

$$F_i^{total} = m_i \cdot a_i = m_i \cdot \frac{\partial V_i}{\partial t} = m_i \cdot \frac{\partial^2 X_i}{\partial t^2} \tag{1.3}$$

donde F_i^{total} es la suma de las fuerzas, m la masa masa y a la aceleración.

Esta ecuación se integra numéricamente empleando el algoritmo de Verlet. Partiendo de velocidades y posiciones iniciales, se puede calcular la posición un instante de tiempo después (representado por Δt) según:

$$V_i\left(t + \frac{1}{2}\Delta t\right) = \left(V_i\left(t - \frac{1}{2}\Delta t\right) + a_i(t)\right)\Delta t \tag{1.4}$$

$$X_i(t + \Delta t) = X_i(t) + V_i\left(t + \frac{1}{2}\Delta t\right)\Delta t \tag{1.5}$$

Con las ecuaciones anteriores podemos calcular la velocidad y posición de cada nodo en un momento dado (t):

$$F_i^{total} = F_i^{fricc} + F_i \tag{1.6}$$

$$m_i \cdot a_i(t) = -K_i^f V_i(t) + (K \cdot (X_i - X_{i+1}) + K \cdot (X_i - X_{i-1})) \tag{1.7}$$

$$a_i(t) = \frac{-K_i^f V_i(t) + (K \cdot (X_i - X_{i+1}) + K \cdot (X_i - X_{i-1}))}{m_i} \tag{1.8}$$

Aproximamos V_i en t por $t - \frac{1}{2}\Delta t$ para el cálculo de la fuerza de fricción.

$$a_i(t) \approx \frac{-K_i^f V_i(t - \frac{1}{2}\Delta t) + (K \cdot (X_i - X_{i+1}) + K \cdot (X_i - X_{i-1}))}{m_i} \tag{1.9}$$

$$a_i(t) \approx \frac{-K_i^f}{m_i} V_i\left(t - \frac{1}{2}\Delta t\right) + \left(\frac{K}{m_i} \cdot [(X_i - X_{i+1}) + (X_i - X_{i-1})]\right) \tag{1.10}$$

Reemplazando las contribuciones a la fuerza 1.10 en la ecuación 1.4 se obtiene:

$$V_i\left(t + \frac{1}{2}\Delta t\right) = V_i\left(t - \frac{1}{2}\Delta t\right) + \left(\frac{-K_i^f}{m_i} V_i\left(t - \frac{1}{2}\Delta t\right) + \left(\frac{K}{m_i} \cdot [(X_i - X_{i+1}) + (X_i - X_{i-1})]\right)\right)\Delta t \tag{1.11}$$

$$V_i(t+\frac{1}{2}\Delta t) = V_i\left(t-\frac{1}{2}\Delta t\right)\left(1-\frac{K_i^f}{m_i}\Delta t\right)+\frac{K}{m_i}\Delta t\left[(X_i-X_{i+1})+(X_i-X_{i-1})\right] \quad (1.12)$$

Tomando un Δt adecuado se puede ir encontrando la trayectoria de cada nodo.

La ecuación (1.12) se puede simplificar a:

$$V_i\left(t+\frac{1}{2}\Delta t\right) = V_i\left(t-\frac{1}{2}\Delta t\right)(1-f_i)+Mtt_i[(X_i-X_{i+1})+(X_i-X_{i-1})] \quad (1.13)$$

donde $f_i = \frac{K_i^f}{m_i}\Delta t$ y $Mtt_i = \frac{K}{m_i}\Delta t$ son parámetros.

Estos parámetros modifican las frecuencias de resonancia de la cuerda (su afinación), la dispersión de energía (el *sustain*) y la composición armónica (que es el tono).

De esta forma hemos desarrollado las ecuaciones necesarias para calcular la velocidad de un nodo i en un tiempo t, que luego es utilizada para el cómputo de la nueva posición del nodo usando la ecuación 1.5.

Una particularidad del modelo planteado, que permite simular comportamientos más ricos, es que éstos pueden diferir entre los nodos de las distintas cuerdas e, incluso, dentro de una misma cuerda.

1.3. Entrada/Salida

1.3.1. MIDI

La interfaz digital de música (`MIDI`) es un protocolo introducido en 1982 utilizado para conectar diferentes instrumentos digitales mediante una interfaz en común [11].

Es muy popular por su simplicidad y flexibilidad a la hora de manejar diferentes instrumentos, ya que es muy eficiente (muchos sistemas operativos están optimizados para interactuar con este protocolo con muy baja latencia) y permite definir diferentes canales (en un mismo cable) dedicados cada uno a un instrumento en particular.

Cada canal transporta mensajes (de forma unidireccional) siguiendo el estándar llamado `GM` (`General MIDI`) que incluyen los valores que representan el sonido y la identificación numérica de cada instrumento (junto con criterios para la utilización de canales, más detalles en la sección 1.3.2). Estos mensajes tienen un máximo de tres bytes y cada bit tiene un propósito específico.

Otra ventaja del protocolo es la variedad de herramientas que lo utilizan. Un claro ejemplo es `Rosegarden` [12] que es una aplicación que permite generar y guardar notas musicales que luego son traducidas a mensajes `MIDI`. Ésto resulta muy útil para el almacenamiento y edición de notas musicales, pero en nuestro caso, también nos permite la realización de pruebas por medio de la estandarización de la entrada recibida en las mismas.

Por su parte, mientras que para el control del instrumento este protocolo es muy útil, el manejo de propiedades más avanzadas requiere mayor volumen de transmisión de datos. Por ejemplo, para simular una cuerda en forma discreta, se la puede discretizar en nodos (ver detalles en la sección 1.2.1 en la página 4), donde cada nodo puede tener sus propiedades o parámetros individuales (como una fricción determinada) distintas a las de sus vecinos. Teniendo en cuenta que una cuerda podría llegar tener cientos o miles

de nodos, es claro que un mensaje `MIDI` no alcanzaría para enviar todos los valores que corresponden a los parámetros de los cientos o miles de nodos utilizados.

De esta manera, la utilización de este protocolo para el manejo de diferentes tipos de parámetros requeriría una re-adaptación, o abuso de su utilización. Es por esto que en esta tesis se busca mantener el uso de `MIDI` para interacciones críticas (como las que suceden con el toque de las cuerdas), pero buscando una nueva arquitectura que sea lo suficientemente flexible para manejar cualquier tipo de propiedades, dándole la posibilidad al usuario de interactuar con las mismas en un tiempo cercano al real.

1.3.2. Audio Digital

Siendo que el sonido está representado por ondas vibrando a una determinada frecuencia, es fundamental encontrar un proceso de conversión bi-direccional que codifique estas ondas en valores que puedan ser interpretados por medios digitales (como lo es un ordenador o un sistema embebido) para su tratamiento o almacenamiento; y que luego puedan ser reconvertidas en ondas físicas para volver a ser sonidos.

Aquí es donde varios dispositivos como el ADC [13] (*Analog to Digital Converter*) permiten convertir información analógica (la onda de sonido) en digital.

A muy grandes rasgos, esto lo realiza tomando una cantidad de muestras (o *samples*) por segundo (al menos dos veces la frecuencia máxima de la onda [14]) y convirtiendo las mismas en valores numéricos. Por ejemplo, si utilizamos una frecuencia de 44.1 kHz, estamos tomando 44 100 muestras por segundo (de una onda vibrando hasta 20 kHz) generando un valor numérico para cada una, es decir, obteniendo como resultado una secuencia de 44 100 números que representan la onda en un tiempo determinado (un segundo) en formato digital. Así, una vez procesada esta secuencia de números por medios digitales, puede ser reconvertida al medio analógico utilizando dispositivos conocidos como DAC [15] (*Digital to Audio Converter*) sin pérdida alguna de calidad gracias al teorema de muestreo de Nyquist-Shannon [14]. Esta señal luego puede ser usada, por ejemplo, en un parlante que vibra reproduciendo el audio.

Existen variedad de dispositivos, algoritmos, tipos de compresiones y pasos que no detallaremos en este trabajo, pero lo importante a resaltar son dos puntos. El primero es que el sonido puede ser representado por una secuencia numérica (donde cada número es una muestra) y el segundo es que la frecuencia de salida va a definir cuántas muestras se necesitan por segundo (en el ejemplo anterior tomábamos 44100 muestras o *samples* por segundo). Es así que si tenemos un sistema que en tiempo real genera sonido, tenemos que asegurarnos de generar exactamente el número de muestras esperadas por segundo. De generarse menos, podría haber distorsiones o pérdida de sonidos. De generarse más, algunas muestras podrían perderse, generando de igual manera distorsiones o latencias innecesarias.

Para esto es que se utilizan los *buffers*, los mismos son espacios de memoria donde los datos se almacenan a la espera de ser utilizados. De esta forma, si se generan más muestras por segundo de las que se pueden procesar, las mismas son almacenadas en memoria (*buffers*) para ser procesadas posteriormente. Ésto tiene la ventaja adicional de que si durante un cierto tiempo se generan menos muestras que las necesarias, pero antes se habían logrado almacenar de más, se puede compensar manteniendo el ritmo de reproducción del sonido. Sin embargo, el abuso en la cantidad o tamaños de *buffers* puede causar que las acciones ejecutadas en un momento se perciban tiempo después (sensación

de desacople entre el gesto realizado y su impacto en el sonido) ya que el buffer resulta muy extenso y tarda en procesarse.

Así, teniendo una cantidad y tamaño adecuados de *buffers*, el software que genere el sonido puede leer directamente de ellos a la velocidad necesaria, mientras que el programa que los genera puede encargarse solo de escribir las muestras. Con esto logramos que el generador y el consumidor no tengan que estar totalmente sincronizados y haya un margen donde puedan ir a diferentes velocidades.

Por su lado, la biblioteca `STK` (*Synthesis ToolKit*) [16] es la que nos permite abstraernos del uso de dichos *buffers* y simplemente escribir los resultados llamando a la operación *tick* con las muestras generadas. A su vez, internamente, establece la comunicación con `Jack`[1] completando el camino completo para que el sonido sea generado. Esta configuración con su utilización será detallada más adelante.

1.3.3. Latencia en audio

Un aspecto importante en la ejecución musical es el sincronismo entre ejecución y percepción, es decir que los sonidos sean generados en un tiempo determinado muy preciso. Por eso, una herramienta como la tratada en este trabajo debe mantener los niveles de retraso tal que el manejo del instrumento resulte *natural*, manteniendo la latencia debajo de un límite que se determina en base a la *sensación* que siente el ejecutante del instrumento.

Es así que la **latencia** la podemos definir como el tiempo que tarda una acción en verse reflejada. Por ejemplo, en nuestro trabajo correspondería al tiempo transcurrido desde que el ejecutante realiza una acción, hasta que la misma es percibida (en forma de sonido) por el usuario. Siendo que todas las acciones no son iguales, la latencia máxima para cada una también puede variar. Operaciones con baja latencia[2] (como el toque de una cuerda) son asociadas a operaciones *hard-realtime*, mientras que latencias más laxas son relacionadas con operaciones *soft-realtime* (como la afinación de las cuerdas, entre otras).

Sin embargo, garantizar una correcta latencia en las operaciones realizadas no asegura el cumplimiento de las restricciones temporales del sistema, es decir, un sonido fluido para el oyente. Para esto tenemos que garantizar que la cantidad de muestras (*samples*) generadas por segundo sea la correcta. Este valor lo podemos calcular en base a la unidad de tiempo utilizada sobre la frecuencia de salida esperada (44.1 kHz[3]). Por lo que:

$$t_{\text{porMuestra}} = \frac{1\,\text{muestra}}{44100\,\text{muestras/s}} \approx 23\,\mu\text{s} \tag{1.14}$$

Siendo así 23 µs el tiempo esperado para la generación de una muestra, por lo que si en un ciclo[4] generamos 32 muestras, la restricción temporal sería de 736 µs aproximadamente:

$$t_{\text{porBuffer}} = t_{\text{porMuestra}} * 32\,muestras \approx 736\,\mu\text{s} \tag{1.15}$$

Es importante destacar que dicha cota representa la "velocidad" con la que el sistema debe generar valores para garantizar la fluidez esperada por el oyente, pero la latencia en que los cambios son impactados se encuentra en otro orden de magnitud.

[1] `Jack` [17] que es el encargado de conectar diferentes interfaces de audio con muy baja latencia y, en particular, con `ALSA` [18] que es el conjunto de *drivers* utilizado para generar el sonido en `Linux`

[2] 20 ms corresponde a la latencia esperada en este tipo de operaciones [19]

[3] Valor configurable

[4] Iteración en código que se encarga de generar el sonido

1.4. GPU

En 1978 surgen las placas de vídeo con la introducción del chip `iSBX 275` de Intel. Siete años más tarde, la Commodore Amiga incluía un coprocesador gráfico que podía ejecutar instrucciones independientemente del CPU, un paso importante en la separación y especialización de las tareas. En la década del 90, múltiples avances surgieron en la aceleración 2D para dibujar las interfaces gráficas de los sistemas operativos y, para mediados de la década, muchos fabricantes estaban incursionando en las aceleradoras 3D como agregados a las placas gráficas tradicionales 2D. Luego a principios de la década del 2000, se agregaron los *shaders* a las placas, pequeños programas independientes que corrían nativo en el GPU y se podían encadenar entre sí uno por pixel en la pantalla [20]. Este paralelismo es el desarrollo fundamental que llevó a las GPUs a poder procesar operaciones gráficas en órdenes de magnitud más rápido que el CPU.

En el 2006, Nvidia introduce la arquitectura G80, que es el primer GPU que deja de resolver únicamente problemas de gráficos para pasar a ser un motor genérico donde cuenta con un conjunto de instrucciones consistente para todos los tipos de operaciones que realiza (geometría, vertex y pixel shaders) [21]. Como subproducto de esto, la GPU pasa a tener procesadores simétricos más sencillos y fáciles de construir. Esta arquitectura de GPU, llamada CUDA (*Compute Unified Device Architecture*), es la que se ha mantenido y mejorado en el tiempo, permitiendo a las GPU escalar masivamente en procesadores simples, de baja frecuencia de reloj y con una disipación térmica manejable, dando como resultado un marcado poder de computo superior respecto a la CPU [Fig. 1.3].

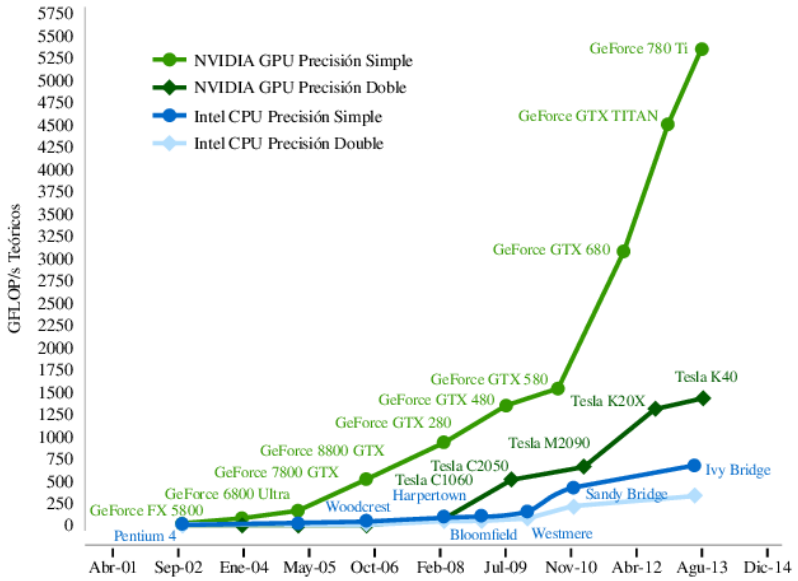


Fig. 1.3: Picos teóricos de *performance* en GFLOPS/s. Tomado de *Nvidia, Cuda programming guide* [22].

CUDA es una arquitectura y un lenguaje de programación (basado en C) que permite escribir código tanto para la CPU como para la GPU. Esto es porque la GPU nunca fue pensada como un procesador aparte, sino como un co-procesador que funciona bajo la supervisión de la CPU (toda la información pasa por la CPU antes de ir a la GPU). Lo cual hace que el modelo de programación de la GPU sea llamado "*host/device*", en donde

el *host* (la CPU) le manda los comandos al *device* (GPU) y no está al tanto de lo que está procesando la GPU, solo puede usar un conjunto de comandos para determinar el estado de la ejecución (al *host* solo le interesa que el *device* termine la ejecución y sus resultados). A pesar de haber otra alternativa para trabajar con GPUs disponible (`OpenCL`), en nuestro caso trabajaremos con CUDA ya que permite sacar el máximo provecho del procesamiento en paralelo en los dispositivos Nvidia.

El desarrollo consiste en reservar y almacenar datos en la memoria de la GPU (memoria DRAM) y una vez que el *kernel*[5] procesa esos datos, los resultados deben ser copiados nuevamente a la memoria del programa (memoria RAM, bajo el control del *host*). Notar que la GPU no puede acceder a la memoria RAM de la CPU, pero el *host* (CPU) sí puede acceder a la memoria de la GPU. Es así que CUDA provee las siguientes operaciones para el manejo de memoria:

1. **cudaMalloc**: Reserva memoria en el *device*, toma dos parámetros, el primero es el puntero a la dirección a reservar y el segundo es la cantidad de bytes a reservar.

2. **cudaFree**: Libera memoria en el *device*. Su único parámetro es el puntero al objeto a liberar

3. **cudaMemcpy**: Permite transferir memoria entre el *host* y el *device* o viceversa. Toma cuatro parámetros, puntero al destino y al origen, cantidad de bytes a copiar y la dirección donde transferir. Esta es una llamada bloqueante.

A grandes rasgos, el programador reserva y copia los datos a procesar al dispositivo (`cudaMalloc` + `cudaMemcpy`), llama al *kernel* (de CUDA, no del sistema operativo) y copia los resultados del dispositivo a la memoria local (`cudaMemcpy`). Notar que `cudaMemcpy` es una llamada a función bloqueante, por lo que el método se queda esperando los resultados del *kernel* para luego así copiarlos.

[5] Función definida por el programador que corre en el *device*

2. SIMULADOR DISCRETO DE INSTRUMENTOS

En esta sección introducimos la arquitectura original del sistema tal como fue encontrada al comienzo de esta tesis. Vamos a mostrar algunos detalles de las operaciones realizadas y daremos una descripción de los principales parámetros utilizados, introduciendo así la importancia de los mismos para que, en secciones subsiguientes, podamos trabajar sobre una nueva arquitectura que permita administrarlos.

2.1. Sistema

En el caso de este trabajo, nos vamos a centrar en un sistema informático cuyo objetivo es el de sintetizar sonido. Los componentes destacables son:

1. `Musical Instrument Digital Interface (MIDI)`: Encargado de la comunicación con el controlador o con cualquier dispositivo que tenga soporte para este protocolo.
2. `Simulador`: Realiza las operaciones principales para la simulación. Éste está escrito en `CUDA` para aprovechar los recursos paralelos brindados por la GPU.
3. `STK (Synthesis ToolKit)` [16]: es el encargado de administrar los *buffers* e interactuar con paquetes de software externos al sistema para generar el sonido.
4. `Flujo principal`: Es la lógica encargada de orquestar todos los componentes principales. La misma es ejecutada por el *main* en el código en `C++` y contiene al *loop* principal del sistema.

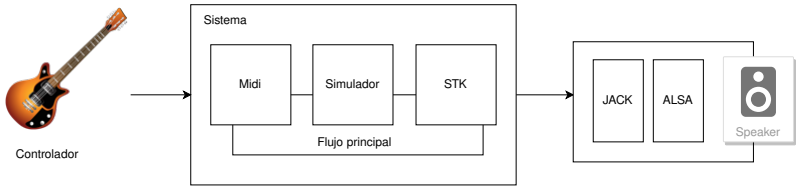


Fig. 2.1: Esquema de los principales componentes presentes en la versión original del sistema. Aquí observamos las diferentes interacciones que posee la información desde que ingresa al sistema mediante `MIDI`, hasta que sale del mismo utilizando `STK`, todo orquestado en el flujo principal

La figura 2.1 muestra cómo el instrumento genera mensajes `MIDI` que son enviados al sistema mediante dicho protocolo. Éstos son traducidos y remitidos al simulador para que, luego de realizar las operaciones necesarias, escriba los resultados en los *buffers* de salida de la biblioteca `STK`. Estos *buffers* representan las cantidad de "pasos"(floats) que se escriben antes de enviar el sonido al programa que lo maneja (`Jack` [17] en nuestro caso). Por lo tanto, si tenemos un *buffer* de 32 posiciones, cada iteración va a generar 32 pasos que serán guardados en *buffers* internos de dicha biblioteca. Luego `STK`, escribirá en los *buffers* de `Jack` que es el encargado (a través de `ALSA` [18]) de escribir en la placa de sonido. De esta manera, teniendo una frecuencia de 44 100 Hz, `STK` se encargará de ir

acumulando los *buffers* e ir mandando 44100 valores por segundo a `Jack`, generando un error en caso que no completemos los pasos suficientemente rápido y bloqueando el hilo en caso de ir más rápido de lo esperado.

Todas estas interacciones se dan en lo que llamamos el "flujo principal", que es el encargado de realizar todas las operaciones una y otra vez, junto con la inicialización de los componentes.

Por otro lado, el simulador es aquel que busca sacar ventaja del cómputo en paralelo utilizando la placa de vídeo (GPU) para trabajar con algoritmos de simulación que generen sonidos **reales** al oído del ejecutante, pero permitiendo una interacción del usuario con el instrumento mucho más flexible.

Dicha flexibilidad está definida como la capacidad del usuario de proveer una gran cantidad de variaciones a las propiedades físicas del instrumento. Aquí es donde los parámetros juegan un rol fundamental, ya que mediante su variación, es posible modular el sonido y, hasta, transformar un instrumento en otro de manera casi instantánea. Los parámetros principales y su función serán descriptos en la siguiente sección.

2.2. Parámetros de entrada

Como se mencionó en la sección 1.2.1, el **simulador** calcula la posición y velocidad de cada nodo en cada momento de la simulación, para ello requiere de una serie de parámetros que controlan su comportamiento, por ejemplo la masa y el coeficiente de fricción en cada nodo.

En un sistema como el tratado en esta tesis, los parámetros son aquellas variables de entrada del simulador que permiten regular el comportamiento del instrumento en mayor o menor medida y pueden ser manipulados por el usuario.

Cada parámetro tiene su propio significado y forma de impactar al sistema, generando distintos intereses desde el punto de vista *musical*. Sin embargo, los mismos pueden ser discriminados según el tiempo de respuesta aceptado. Por ejemplo, cuando el ejecutante toca las cuerdas de una guitarra, cada toque de cuerda requiere una respuesta que, a oídos del mismo, resulte inmediata (caso contrario el sonido saldría con retraso, generando una sensación de desagrado al músico). Por su lado, hay parámetros como la cantidad de cuerdas, que solo impactan al inicio del programa y no cambia durante la ejecución, por lo que un retraso en su aplicación, no va a afectar la naturalidad del instrumento.

De aquí proviene la importancia de caracterizar a los parámetros de acuerdo al impacto que puede producir en relación con la respuesta del sistema. Una vez identificados los más críticos, se podrá diseñar una arquitectura del sistema para que los maneje sin que un cambio en éstos impacte negativamente en el sonido generado o en el tiempo de respuesta.

A continuación vamos a dar una caracterización de nuestros parámetros para poder referenciarlos luego:

1. **Configuración**: Son aquellos que solo impactan en un momento muy concreto del programa y cualquier retraso es aceptable (por ejemplo, variables de configuración inicial del sistema).

2. **Ajustes**: Son aquellos que permiten calibrar o modificar el sistema, realizando cambios significativos en el mismo (por ejemplo propiedades físicas del instrumento, afinación, etc.).

3. **Toque**: Son aquellos que son utilizados durante la interacción del usuario con el sistema (por ejemplo la posición del dedo en la cuerda)

Notar que la respuesta esperada en el punto 3 tiene que ser instantánea (al oído del usuario), mientras que la del 2 acepta muy pequeños retrasos y la del 1 es aun más flexible (ya que no se van a modificar una vez que el sistema está funcionando).

El foco de esta tesis va a estar puesto en los puntos 1 y 2, ya que 3 se está realizando con `MIDI` por el momento (aunque el patrón general es adaptable a cualquier parámetro).

2.2.1. Parámetros de configuración y ajuste

En esta categoría nos encontramos con las propiedades que le dan al sistema diferentes características de sonido interesantes para la investigación. Algunas variables a destacar son:

Configuración
Cantidad de cuerdas
Cantidad de nodos por cuerda
Tamaño del dedo

Ajuste
Fricción de la cuerda
Frecuencia de la cuerda
Máxima fricción en punta
Ancho de las puntas
Distancia de la cuerda con diapasón
Distancia de la cuerda con el traste
Distancia entre nodos
Palanca

Que los parámetros estén en la misma categoría no significa que afecten de la misma manera al sistema. Por ejemplo, una mayor masa altera la afinación, mientras que aumentar el tamaño del dedo va a impactar con mayor sutileza el sonido (generando un sonido más apagado).

Recordar que hay parámetros de sistema que no afectan directamente al simulador, sino que requieren un cálculo previo y terminan impactando en múltiples parámetros, a éstos los hemos denominado **meta-parámetros**.

2.2.2. Meta-parámetros

Muchos de los parámetros no impactan directamente al simulador, sino que sirven para recalcular matrices que representan diferentes propiedades de los nodos de las cuerdas. A éstos los llamamos **meta-parámetros**, ya que su impacto es indirecto y requieren un procesamiento extra.

Un ejemplo de este tipo de parámetros es la fricción por nodo que fue mencionado en la introducción (f_i), donde la fricción se puede calcular como la suma de dos gaussianas en función del nodo (i) en la cuerda (j):

$$f_{i,j} = a_j \cdot \exp(-b_j \cdot i^2) + a_j \cdot \exp(-b_j \cdot (i - c)^2) + d_j \tag{2.1}$$

donde $a_j = maxFriccionEnPunta - friccion_j$, $b_j = anchoPuntas_j$, $c = \#nodos$ y $d_j = friccion_j$. Gráficamente podemos verlo como una cuerda con $\#nodos$ nodos donde la fricción en las puntas es más alta:

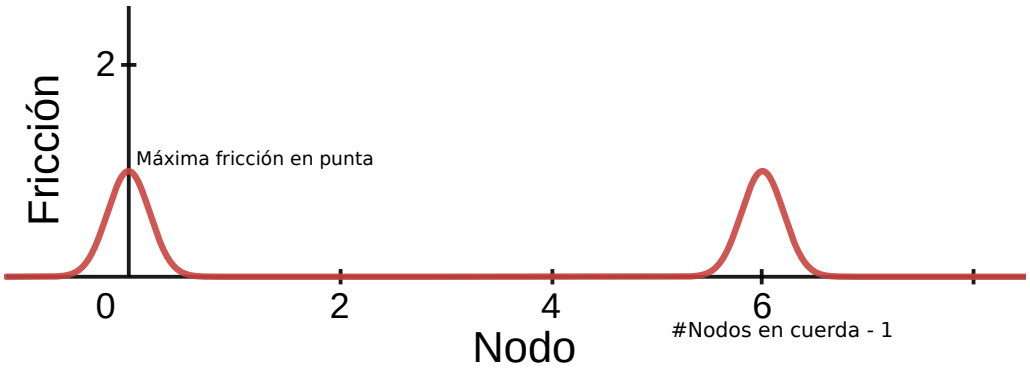


Fig. 2.2: Ejemplo de cuerda con dos gaussianas representando la fricción en las puntas. Se puede observar cómo la fricción en las puntas es mayor y va disminuyendo al aproximarse al centro. Esto modela una cuerda real donde en los extremos se encuentran el clavijero y el puente sosteniendo la cuerda.

En el ejemplo se pueden observar meta-parámetros como $maxFriccionEnPunta$, $friccion_i$, $anchoPuntas_j$ y $\#nodos$, ya que son parámetros que tienen sentido para el usuario pero al simulador solo le interesa f_i. Por lo que cada vez que se actualiza alguno de éstos, es necesario recalcular f en cada en uno de los nodos.

En nuestro cálculo, la fricción es aplicada con más intensidad en las puntas, pero también se podría aplicar homogéneamente en todos los nodos de la cuerda. Esto abre un abanico de posibilidades a la hora de experimentar con instrumentos, ya que no solo se pueden explorar los parámetros, sino que también la forma en la que se los generan a partir de meta-parámetros, incluso empleando funciones distintas a las aquí planteadas.

Esta resulta ser la característica principal de los meta-parámetros: pueden variar dependiendo de la implementación que uno elija, mientras que el simulador termina utilizando los valores que son obtenidos a partir de éstos.

Por otro lado, los meta-parámetros son de particular interés porque el costo del cálculo puede llegar a ser significativo, especialmente en sistemas que requieren realizar operaciones por debajo del mili-segundo (como es nuestro caso) y, a su vez, el tamaño de la salida generada puede ser significativo también (por ejemplo en la fricción para seis cuerdas con 400 nodos, la salida esperada contiene 2400 elementos y esto puede crecer considerablemente a medida que aumentan estos valores). Tener en cuenta que una salida más grande, implica mayor cantidad de datos a ser transmitidos y procesados.

2.3. Código

El sistema fue escrito principalmente en `C++` y CUDA. CUDA es una arquitectura junto con un lenguaje de programación que permiten la ejecución en paralelo utilizando las unidades de procesamiento gráfico (GPUs) de Nvidia.

La utilización de CUDA en el sistema es muy sencilla y cuenta, básicamente, con los siguientes pasos:

1. Reservar memoria en la placa (`cudaMalloc`).
2. Copiar datos en memoria reservada (`cudaMemcpy`).

3. Procesar dichos datos en el programa CUDA (`simulador.cu`). Aquí se realizan las operaciones matemáticas para generar el sonido, entre otras, las detalladas en la introducción [1.2.1]. Éste es el *Simulador* nombrado anteriormente.

De esta manera, el flujo principal se encarga de realizar los pasos 1, 2 y realizar la llamada a 3.

Aquí una breve simplificación del flujo principal (*main*):

Algorithm 1: Programa principal - Main

```
Result: Sonido
Inicialización de componentes;
Reserva de memoria;
while true do
    Lectura de parámetros y meta-parámetros mediante MIDI;
    if Hay cambios MIDI then
        Generación de parámetros para el simulador;
        Cálculos con meta-parámetros;
    end
    Copiar variables a la placa de vídeo;
    Llamada al simulador (kernel de la GPU);
    Escribir resultado en memoria;
    Escribir resultados en buffer de sonido;
end
```

La *inicialización de componentes* incluye la lectura de configuraciones, la inicialización de `MIDI`, `STK` y el pre-cálculo de algunas matrices utilizadas para el sonido.

Por otro lado, la *lectura de parámetros* era realizada originariamente mediante `MIDI` (para todos los tipos de parámetros).

Aquí vale notar que el simulador queda atado a las limitaciones de los mensajes `MIDI` (máximo de 3 B por mensaje, ver detalles en la sección 1.3.1) y a que el dispositivo posea una interfaz `MIDI` para conectarse. Esto es un problema ya que **queremos darle al simulador la flexibilidad necesaria para no estar sujetos a una cantidad fija de parámetros y experimentar con diferentes funciones para las meta-variables, junto a la posibilidad de utilizar una interfaz de comunicación distinta**.

Finalmente, luego de realizar los cálculos en CUDA, se escriben los resultados en el *buffer* de salida utilizando la biblioteca: The Synthesis ToolKit in `C++` (`STK`).

3. ARQUITECTURA DEL SISTEMA

Habiendo ya dado la descripción del sistema inicial y la importancia de los parámetros utilizados, ahora nos enfocamos en encontrar una arquitectura eficiente que permita conservar una baja latencia y una alta flexibilidad.

El enfoque de este capítulo es adaptar el sistema existente a una nueva arquitectura que nos permita realizar operaciones potencialmente costosas, sin afectar el flujo del mismo, es decir, manteniendo el tiempo de respuesta dentro de los límites aceptables para la percepción del ejecutante.

3.1. State y Estado

Con el objetivo de mejorar la interacción entre de los parámetros previamente mencionados en la sección 2.2 (página 12) y el sistema, vamos a definir una nueva abstracción llamada **State**. Esta clase contiene todos los parámetros (de ajuste y configuración) que representan al sistema físico y un subconjunto del estado computacional del mismo.

El *sistema físico* representa todas las variables que determinan el comportamiento del modelo físico a representar, como lo son la cantidad de nodos, la masa y otros atributos que se utilizan para modelar, en nuestro caso, las propiedades de una guitarra. Notar que estas variables solo pueden ser actualizadas por el usuario, es decir, el sistema puede evolucionar pero no cambiar su naturaleza por sí mismo.

El *estado computacional* es el conjunto de todas las variables y sus valores en un momento determinado de la ejecución. Mientras que para el **State** solo nos van a interesar las variables relacionadas con los parámetros, el *estado computacional* puede contener además variables con punteros, referencias a bibliotecas y variables que dependen del estado anterior, como es el caso de una cuerda perturbada (i.e. el ejecutante la está *tocando*) que puede quedar vibrando por un tiempo prolongado, por lo que la posición de los nodos de la cuerda en un momento va a depender de cómo estaban en el momento anterior.

De esta manera, teniendo en cuenta el universo de variables que posee el sistema, si logramos identificar y desacoplar parte del *sistema físico* (que también corresponde al *computacional*), vamos a haber identificado los parámetros de configuración y ajuste con los cuales nos interesa interactuar.

Para esto, nuestra primera aproximación va a ser crear un objeto llamado `State` que contenga todos aquellas variables que no son modificadas por el simulador pero sí por el usuario y por lo tanto definen parte del *sistema físico* en cuestión.

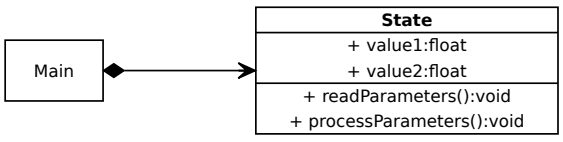


Fig. 3.1: Ejemplo de primera aproximación de arquitectura propuesta donde el código principal mantiene una referencia al objeto **State** que contiene los parámetros (value1, ..) y las operaciones para leer y realizar operaciones sobre los parámetros.

Podemos observar que mientras que antes teníamos una variable y la accedíamos directamente, ahora tenemos que acceder al objeto `State` y luego a la variable. Ejemplo:

```
...
void main() {
  ...
  int nFrames = 1000000;
  int cantMaximaNodos = 400;
  int cantCuerdas = 6;
  float ordenMasa = .5;
  ...
  /*********************************
  // ACA EMPIEZA EL LOOP PRINCIPAL *
  //*********************************/
  for (int i = 10; i < nFrames; i++) {
    ...
    float* masaPorNodo = generarMasaPorNodo(cantMaximaNodos, cantCuerdas,
    ordenMasa)
    ...
  }
  ...
}
```

Listing 3.1: Versión original del código en el que los distintos parámetros se están utilizando directamente.

```
void main() {
  ...
  State state = new State();
  ...
  /*********************************
  // ACA EMPIEZA EL LOOP PRINCIPAL *
  //*********************************/
  for (int i = 10; i < state->nFrames; i++) {
    ...
    float* masaPorNodo = generarMasaPorNodo(state->cantMaximaNodos, state->
    cantCuerdas, state->ordenMasa)
    ...
  }
  ...
}
```

Listing 3.2: Versión modificada incluyendo la instanciación de un objeto de la clase `State` que engloba todos los parámetros que controlan el comportamiento de la simulación.

Como se puede observar, el objeto `State` contiene todas las variables públicas y el acceso es directo, por lo que no hay impacto alguno en la eficiencia del programa (3.3.4). Por otro lado, éstas son inicializadas en el `State` y no en el programa (Ejemplo: `nFrames`, `cantMaximaNodos`, etc).

Siguiendo el ejemplo anterior, podemos observar que `generarMasaPorNodo` retorna una lista de valores a partir de diferentes parámetros. Éste es un ejemplo de **meta-parámetro**, ya que son solo utilizados para generar otros valores. De esta manera, podemos agregar `masaPorNodo` también al `State` (ya que es una variable utilizada por el sistema).

```
...
state->masaPorNodo = generarMasaPorNodo(state->cantMaximaNodos, state->
    cantCuerdas, state->ordenMasa)
```

```
3 ...
```

Listing 3.3: Moviendo masaPorNodo al `State`

Así, hemos logrado desacoplar las variables que representan los parámetros del programa principal de forma muy sencilla. Aunque esto es una mejora, aún tenemos que hacer estas variables sean accesibles al usuario. Es así que como primera aproximación vamos a mover los parámetros a un archivo.

3.1.1. Archivo de texto

Para que el usuario pueda interactuar con las variables del `State` necesitamos generar algún tipo de interfaz donde pueda ingresar y modificar parámetros. Para esto utilizaremos un archivo de texto, de forma tal que el usuario pueda modificarlo y luego estos cambios se vean reflejados en el `State`.

Algo interesante a notar es que una vez modificado el `State`, estos cambios impactarán instantáneamente en el sistema porque en cada iteración el flujo principal utiliza la última versión (y única) del `State`.

Para esto utilizaremos la biblioteca `libconfig` [23] que nos permite leer y guardar las variables de una manera muy conveniente. El siguiente es un ejemplo de archivo de configuración utilizado:

```
nFrames = 100000000;
samplerate = 44100;
nbufferii = 64;
canalesEntrada = 2;
npot = 2048;
dedoSize = 64;
softrealtimeRefresh = 200;
escalaIntensidad = 1.0;
distanciaEquilibrioResorte = 20.0;
distanciaEntreNodos = 50000.0;
centro = 0.25;
maxp = 0.1;
expp = 0.6;
ordenMasa = 0.1;
cuerdas = (
  {
    friccion = 0.000003;
    nodos = 400.0;
    frecuencia = 82.1069;
    maxFriccionEnPunta = 0.4;
    anchoPuntas = 0.3;
    distanciaCuerdaDiapason = -180.0;
    distanciaCuerdaTraste = -150.0;
  },
  {
    friccion = 0.000003;
    nodos = 400.0;
```

```
    frecuencia = 109.75;
    maxFriccionEnPunta = 0.4;
    anchoPuntas = 0.3;
    distanciaCuerdaDiapason = -180.0;
    distanciaCuerdaTraste = -150.0;
  }
);
```

Como se puede observar, este archivo contiene varios parámetros de configuración y ajuste. A su vez veremos propiedades sobre las cuerdas, por ejemplo, la primera cuerda tiene una frecuencia principal de resonancia de 82.1069 Hz y la segunda de 109.75 Hz.

Así, el `State` va a verse de esta manera:

```
class State {
  public:
    // Parametros de configuracion
    int  nFrames;
    int samplerate;
    int  nbufferii;
    int  canalesEntrada;
    int npot;
    unsigned int dedoSize;
    bool debugMode;
    int softrealtimeRefresh;
    float ordenMasa;

    // Parametros de ajuste
    float escalaIntensidad;
    float distanciaEquilibrioResorte;
    float distanciaEntreNodos;
    float centro;
    float maxp;
    float expp;

    // Atributos autogenerados
    int cantCuerdas;
    int cantMaximaNodos;
    float masaPorNodo;
    float friccionSinDedo;
    float friccionConDedo;
    float minimosYtrastes;
    float cuerdas;

    static string ITEMS_CUERDA[];
};

string State::ITEMS_CUERDA[] = {"friccion",
                                "frecuencia",
                                "maxFriccionEnPunta",
                                "anchoPuntas",
                                "distanciaCuerdaDiapason",
                                "distanciaCuerdaTraste",
                                "nodos"};
```

Listing 3.4: Primera versión de State.h

El hecho de extraer las variables nos trae otro beneficio que es el de poder realizar determinados cálculos (por ejemplo con meta-parámetros) fuera del flujo principal, ya que podemos saber cuándo se modifican éstos.

Sin embargo, el hecho de extraer los parámetros a un archivo no es suficiente para que se actualice el State, ya que necesitamos algún tipo de lógica que lo vaya leyendo periódicamente (especialmente para los parámetros de ajuste). De esta manera surge la pregunta *¿Cuándo y cómo cargamos el archivo de configuraciones en el sistema?*

Carga de archivo en flujo principal

Una primera aproximación es realizar la carga del archivo cada n iteraciones. Por ejemplo:

```
void main() {
  ...
  state->cnFrames = leerDeArchivo(int, "nFrames")
  state->cantMaximaNodos = leerDeArchivo(int, "cantMaximaNodos");
  state->cantMaximaNodos = leerDeArchivo(int, "cantCuerdas");
  ...
  /*********************************
  // ACA EMPIEZA EL LOOP PRINCIPAL *
  //*********************************/
  for (int i = 10; i < state->nFrames; i++) {
    ...
    if (i % N == 0) {
        state->actualizarState();
    }
    ...
  }
  ...
}
```

Listing 3.5: Carga de archivo cada n iteraciones. El método `leerDeArchivo` permite leer el valor de un parámetro desde un archivo de texto. Como resultado, se observa cómo cada N iteraciones se lee el archivo de texto.

Notemos que la lógica de carga de archivo se encuentra en el `State` y por cada llamado a `actualizarEstado()` se lee el archivo de configuración y se actualizan los parámetros.

En las experimentaciones 3.3.5 vamos a ver que esto no es una opción viable, ya que el solo hecho de leer el archivo de configuraciones y actualizar el `State` tarda más de 10 ms (recordar que nuestro límite aceptable es menos de 1 ms por iteración, ver sección 1.3.3). **Por lo que cualquier implementación que implique cargar el archivo en el flujo principal, va a tener un impacto muy negativo en el sistema**.

Carga de archivo usando un administrador de estado

Una segunda opción es la utilización de un objeto llamado **"Administrador de Estado"** (`StateManager` en el código) que se encargue de manipular el `State` de forma tal que el sistema no tenga que ocuparse de cargarlo directamente. Esta propuesta, junto con la experimentación realizada, se detallará en la siguiente sección.

3.2. Administrador de estado

El administrador de estado es un objeto nuevo que se encarga de administrar y proveer el `State` al sistema. El objetivo del mismo es abstraer completamente la forma en que el `State` es actualizado y proveer mecanismos de recuperación en caso de que haya un error en la actualización del mismo.

Es por esto que a partir de la introducción del `StateManager`, el sistema principal no va a tener acceso directo al `State`, sino que lo hará a través de este nuevo objeto. Lo mismo sucedería con cualquier otro objeto que quiera leer o escribir el `State`, lo deberá hacer a través del administrador de estado.

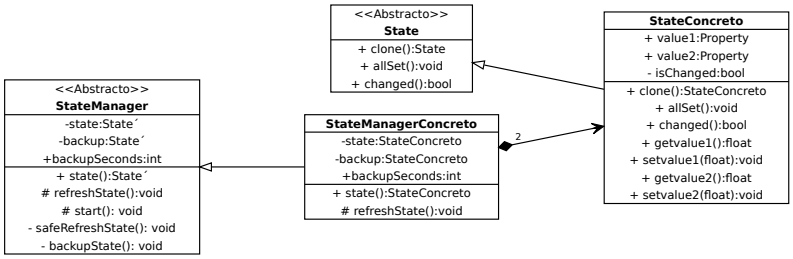


Fig. 3.2: Diseño del `StateManager` con una clase concreta que implementa el método `refreshState`. También se observan dos referencias al `StateConcreto` que es de tipo `State`. Notar que `State'` refiere a algún tipo que implemente `State`.

Como se puede observar en la figura 3.2 el `StateManager` es una clase que implementa dos métodos principales (`backupState` y `safeRefreshState`). El primero se encarga, en intervalo de tiempos regulares, de hacer una copia del `State` llamando al método `clone` de este último. Esto lo realiza en un hilo aparte para no afectar las operaciones principales y guarda la nueva copia en la variable `backup`.

```
template <class S>
void StateManager<S>::backupState() {
  while(true) {
    try {
      this_thread::sleep_for(std::chrono::seconds(backupSeconds));
      S* prevState = _backup;
      _backup = _state->clone();
      if (prevState != NULL) {
        // Elimino previo backup
        delete prevState;
      }
    } catch(...) {
      cerr << "Error: Error actualizando Backup" << endl;
    }
  }
}
```

Listing 3.6: Implementación interna del método `backupState`. Aquí se ve como cada `backupSeconds` se realiza una copia de seguridad (`clone`) del `State`. En caso de falla se ignora el error y se mantiene el anterior.

Por su parte, `safeRefreshState` se encarga, simplemente, de llamar a `refreshState` pero lo hace capturando cualquier excepción posible. Es así que, en caso que se genere una,

reemplaza el valor actual del `state` con el valor de `backup` generado por `backupState`. Esto permite evitar que todo el sistema se invalide por un error en el `StateManager`, aunque el mismo debe permitir "volver atrás"[1] unos segundos sin problemas, cosa que puede no ser posible en determinados sistemas de tiempo real.

```
template <class S>
void StateManager<S>::safeRefreshState() {
  while(true) {
    try {
      refreshState();
    } catch(...) {
      cerr << "Error: Error actualizando State - Recuperando State" << endl
    ;
      if (_backup == NULL) {
        cerr << "PANIC: Backup no disponible" << endl;
        exit(1);
      }
      // Recuperando State
      S* prevState = _state;
      _state = _backup;
      _backup = NULL;
      delete prevState;
    }
  }
}
```

Listing 3.7: Implementación interna del método `safeRefreshState`. Aquí se ve como en caso de cualquier excepción, el `State` es recuperado utilizando la última copia de respaldo.

Estos métodos son parte del `StateManager` por lo que el programador no necesita interactuar con los mismos, lo único que debe hacer es implementar el método `refreshState` que se encarga de actualizar el `State`. Ésto conlleva a que el `StateManager` no pueda ser instanciado por sí solo, ya que requiere una clase concreta que implemente el método `refreshState`.

3.2.1. Cambios en el State

Siguiendo la arquitectura propuesta (fig. 3.2) se implementaron nuevos métodos en el `State`: `clone`, `allSet` y `hasChanged`. Estos métodos son públicos y tienen la responsabilidad de:

1. `clone`: Crear una copia del `State` actual
2. `allSet`: Marcar el `State` como actualizado
3. `hasChanged`: Notificar que el `State` fue actualizado

Los dos últimos permiten al flujo principal saber cuándo hubo cambios, es decir, cuando se llama al método `hasChanged` devuelve verdadero si algún parámetro cambió desde la última lectura y `allSet` permite marcar el `State` como leído. Así, cuando no se modifica ningún parámetro, no es necesario revisar parámetro por parámetro para saber si hubo actualización (tenemos un `flag` global que nos lo indica).

[1] Deshacer los cambios del `State` hechos en los últimos segundos

Por otro lado, es importante destacar que el `State` se convertió en una clase no instanciable (abstracta), ya que requiere la implementación de los nombrados mencionados anteriormente. A esto debemos sumarle la definición e implementación de los *setters* y *getters* que deben estar en la clase concreta, para que el `State` mantenga un propósito general, i.e el `State` no dependa de los parámetros definidos.

Para esto usamos macros de `C++` [24] ya que es una herramienta muy utilizada que nos permite definir de forma limpia una variable nueva y, en tiempo de compilación, crear métodos y variables nuevas para la misma. Veamos un ejemplo:

```
class ParameterState : public State {
  public:
    ...
    PROPERTY(int, canalesEntrada);
    ...
}
```

Listing 3.8: Ejemplo de definición de parámetro en el State.

mientras que en tiempo de compilación, se genera automáticamente el siguiente código:

```
class ParameterState : public State {
    public:
        ...
        Property<int> canalesEntrada;
        bool isChanged; // Dice si algun valor del state cambio
        ...
        int getcanalesEntrada()
        {
          return canalesEntrada.value;
        }
        void setcanalesEntrada(int val)
        {
            canalesEntrada.write_mtx.lock();
            canalesEntrada.set = canalesEntrada.value != val;
            canalesEntrada.value = val;
            isChanged = isChanged || canalesEntrada.set;
            canalesEntrada.writeMtx.unlock();
        }
        bool hasChangedcanalesEntrada()
        {
            return canalesEntrada.set;
        }
        void setChangedcanalesEntrada()
        {
            canalesEntrada.set = false;
        }
        ...
}
```

Listing 3.9: Ejemplo de código auto generado utilizando macros de `C++`.

donde `property` está definido como:

```
    template <typename T>
    struct Property
    {
      Property() : set(true) {}
      T value;
      bool set;
```

```
7       std::mutex writeMtx
8     };
```

Listing 3.10: Definición de `struct Property` utilizado para encapsular cada variable del `State`.

Ésto nos permite re-utilizar muchísimo código (ya que solo hay que modificar el `template` y no parámetro por parámetro) implementando funcionalidades muy interesantes como `hasChanged...()` que nos indica si el valor fue modificado (muy útil para evitar re-procesamientos), `setChanged...()` para avisar que la variable ya fue consumida (y que `hasChanged...()` retorne `false` en la próxima iteración) y finalmente los *getters* y *setters* que permiten retornar y guardar el valor respectivamente.

Notar que los *setters* deben realizar las operaciones de forma atómica para evitar inconsistencias. Es por esto que agregamos un semáforo tipo mutex (`writeMtx`) para que, en caso en que dos o más hilos intenten actualizar el `State` de un parámetro simultáneamente, éstos deberán hacerlo en orden de llegada al recurso. Pero no es suficiente, ya que las lecturas (*getters*) no poseen mutex[2], por lo que debemos garantizar que cuando se lee un valor, éste ya se encuentra completamente actualizado y no a medio escribir. Para esto realizamos operaciones atómicas a la hora de actualizar los valores (especialmente con estructuras de datos como arreglos o diccionarios) donde, si por ejemplo debemos copiar un arreglo completo de elementos, hacemos un cambio de punteros (`swap` en `C++`) en vez de copiar uno a uno. En este caso el lenguaje lo permite, pero de no hacerlo, hay que evaluar el uso de mutex para lectura también.

El último detalle a resaltar es que, al ser la clase concreta (`StateConcreto` en fig.3.2) la que posee la implementación de los *setters* y *getters*, al `StateManager` no le alcanza una referencia al `State`, sino que necesita una referencia a la clase concreta del mismo. De no contar con esa referencia, cuando el sistema le consulte al `StateManager` por el `State`, el objeto resultante no tendrá la implementación de los *setters* y *getters*. Es por esto que la condición definida es que el `StateManager` mantenga referencias a una clase que extienda al `State` (esto lo definimos como `State'` en los gráficos).

3.2.2. Administrador de estado con lectura de archivo

Con este nuevo actor, y al cambiar la responsabilidad, debemos mover la lectura del archivo al administrador. Esto lo realizamos creando una nueva clase `FileManager` que extiende del `StateManager` e implementa el método `refreshState` donde debemos preocuparnos por agregar el código para actualizar el `State`, ya que el manejo de hilos está implementado en la definición del `StateManager` (`safeRefreshState`).

[2] Ya que la actualización de las estructuras de datos son atómicas y no hay riesgo de lectura inconsistente

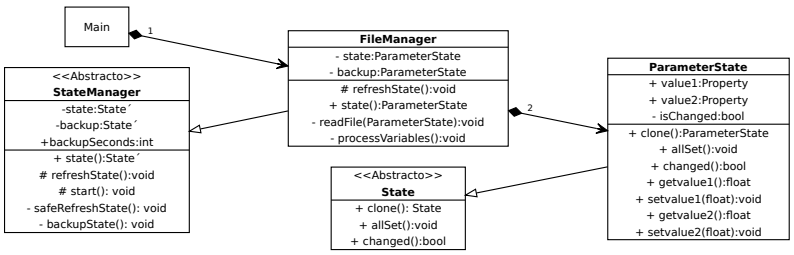


Fig. 3.3: Ejemplo de arquitectura con administrador de estado donde el flujo principal (*main*) mantiene una referencia al `FileManager` (de tipo `StateManager`) y este otro mantiene la referencia dos referencias al `ParameterState` (de tipo `State`). Notar que el código principal ya no puede acceder directamente al `State` y la actualización del `State` es totalmente transparente (depende de la implantación del `StateManager` seleccionada). `State'` refiere a algún tipo que implemente `State`.

Vale aclarar que el cálculo de los meta-parámetros ahora es parte del `refreshState` ya que los resultados son almacenados en el `State`. Quedando así dicho método encargado de leer los parámetros del archivo de texto, calcular los meta-parámetros (si correspondiera) y guardarlos en el `State`.

3.2.3. Impacto en flujo principal

Como se puede observar, ahora el flujo principal se encarga de manejar un objeto de tipo `StateManager` sin importar su implementación. Esto nos permite cambiar la lógica para actualizar el `State` sin modificar el código existente.

```
1  void main() {
2    ...
3    StateManager<ParameterState> stateManager = new FileManager();
4    ...
5    if (stateManager -> state() -> hasChanged()) {
6               ...
7               if (stateManager -> state() -> hasChangedmasaPorNodo()) {
8                   d_M = cudaFreeMallocAndCopy < float > (stateManager ->
     state() -> getmasaPorNodo(), d_M);
9                   stateManager -> state() -> setChangedmasaPorNodo();
10              }
11              ...
12              stateManager -> state() -> allSet();
13          }
14   ...
15 }
```

Listing 3.11: Ejemplo de uso de administrador de estado donde solo se indica la implementación en la inicialización y luego es transparente para el hilo principal. Por otro lado muestra como se actualiza la `masaPorNodo` en la memoria de la GPU.

Al manejarse en *threads* aparte, podemos realizar operaciones más costosas sin afectar[3] el funcionamiento de la generación de sonido en el sistema principal.

[3] Suponiendo que se utiliza un procesador *multi-thread*.

De esta manera hemos diseñamos una nueva arquitectura que nos permite realizar operaciones costosas, y abstraer cálculos sobre las variables del `State`, sin impactar directamente el flujo principal del programa.

3.3. Experimentación

En esta sección mostramos los resultados obtenidos con las diferentes arquitecturas propuestas.

3.3.1. Requisitos y Configuración - MIDI

Las pruebas fueron realizas sobre un sistema operativo Linux (Ubuntu 18.04) con el hardware descripto en la sección 7.2, junto con los programas `Rosegarden` [12] (para la generación `MIDI`) y `Jack Audio` [17] (para conectar interfaces `MIDI` y salidas de audio).

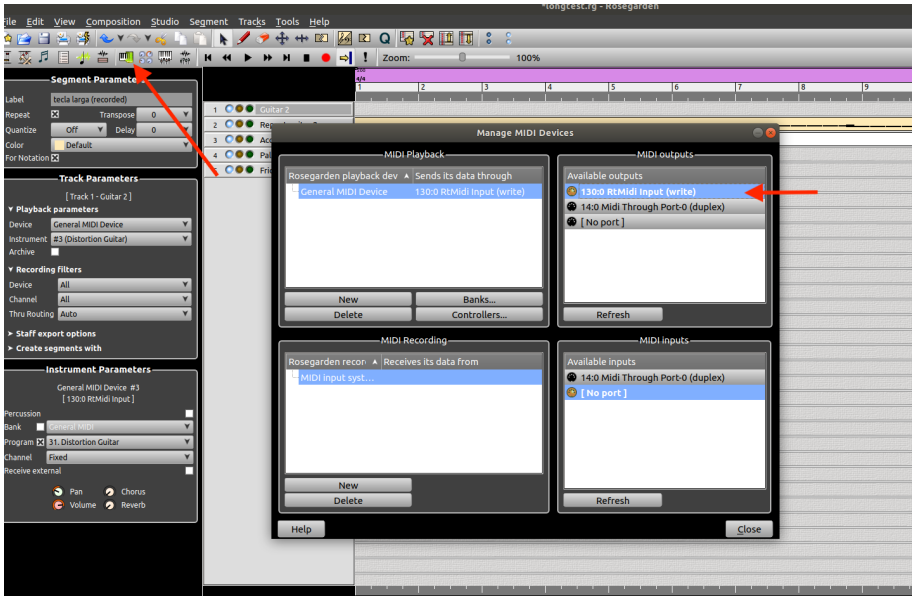
Dado que en estas experimentaciones la entrada esperada es `MIDI`, se generó un archivo con `Rosegarden` que consiste en un bucle infinito con tres operaciones principales:

- *cuerda*: Representa el toque de una o varias cuerdas por diferentes lapsos de tiempo. Éste es un parámetro de toque ya que requiere tener un impacto de baja latencia.
- *palanca*: Representa el uso de la palanca mientras se tocan las cuerdas. Éste es un escalar que modifica la tensión de las cuerdas tocadas. Es un claro ejemplo de parámetros de ajuste y la latencia puede ser media.
- *fricción*: Éste representa un cambio en la fricción de alguna de las cuerdas. Es un meta-parámetro ya que cada vez que hay un cambio se debe calcular la masa de todos los nodos de las cuerdas. Similarmente al anterior, es un parámetro de ajuste pero la latencia esperada puede ser mayor (dado su costo computacional y la baja frecuencia con la que el usuario interactuaría con este valor mientras toca).

Es importante notar que estas tres operaciones representan el conjunto de parámetros con los cuales queremos trabajar por lo que todas las experimentaciones se basarán en estos tres.

Los pasos que se siguieron para realizar las experimentaciones son los siguientes:

1. Inicializar `Jack` con los siguientes parámetros:
   a) *Driver*: Alsa
   b) *Sample Rate*: 44 100 Hz
   c) *Frames/Period*: 64
   d) *Periods/Buffer*: 2
2. Inicializar `Rosegarden` abriendo el archivo de prueba
3. Inicializar el simulador (*make run* en terminal)
4. Conectar `Rosegarden` al sistema



5. Presionar Play

Una vez comenzada la reproducción, el programa generará un archivo de salida donde cada línea representa una iteración con información sobre los tiempos (en microsegundos) de cada iteración.

Finalmente este archivo es usado como entrada de un *script* en `Python` para generar los gráficos que se verán en los siguientes apartados. Dicho *script* elimina los valores atípicos (*outliers*), aclarando en la distribución cuántos valores fueron ignorados, ya que los mismos son casos aislados que provienen de la utilización de un sistema operativo no optimizado (que está realizando varios procesos al mismo tiempo). Es importante destacar que estos *outliers* no afectan negativamente al sistema ya que la biblioteca utilizada (`STK`) implementa *buffers* internos que logran atenuar su impacto, haciendo así posible la finalización de todas las pruebas satisfactoriamente.

3.3.2. Buffer de sonido

Habiendo dado la metodología para realizar las experimentaciones, pasaremos a analizar los distintos resultados obtenidos. Como primer paso, veamos qué sucede al medir el tiempo total de cada iteración en el programa original (sin modificaciones):

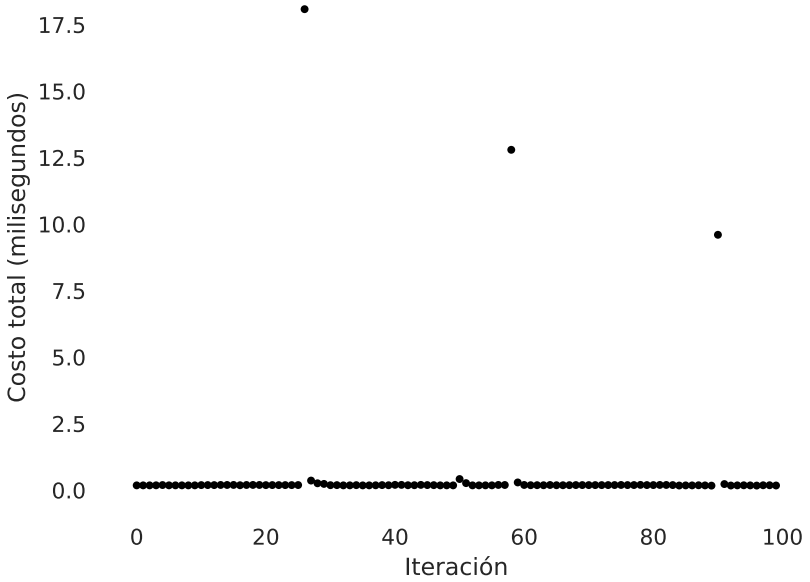


Fig. 3.4: Recorte de 100 iteraciones (puntos negros) donde se midió el tiempo desde que se inicia hasta que termina la iteración. Se pueden notar tres puntos (iteraciones) que sobresalen notablemente por sobre el resto.

Para poder explicar el comportamiento presentado en la figura 3.4, debemos observar parte de los datos obtenidos[4]:

iteración	1	2	3	4	...	34	35	36	...	66	67
microsegundos	182	**19168**	288	177	...	**18123**	215	158	...	**19154**	232

Tanto en el gráfico, como en los datos podemos observar que cada 32 iteraciones hay un salto temporal considerable (outliers en el gráfico, valores en negrita en los datos), que al ser valores tan grandes en comparación con el costo de las otras iteraciones, hacen parecer a estas últimas como despreciables (se asemejan a una línea recta en cero).

Esto se debe a que en la medición estamos incluyendo el tiempo de escritura en los *buffers* de sonido (particularmente en los *buffers* de `STK`). Como se describió en la sección 2.1, `STK` posee sus propios *buffers* que va escribiendo y una vez completos, espera a que los mismos sean consumidos por la siguiente capa (en nuestro caso `Jack`). Esta espera es bloqueante, y es la causante de los saltos que se observan en el gráfico.

Es por esto, que para el análisis realizaremos las mediciones justo antes de escribir en `STK`. Ignorando así el tiempo de escritura en *buffer* y dejando abierto el análisis del uso de `STK` en sistemas de baja latencia para futuros trabajos.

A continuación veremos el mismo gráfico sin tener en cuenta el tiempo de escritura:

[4] Tanto el gráfico como los datos son un subconjunto de los tiempos obtenidos

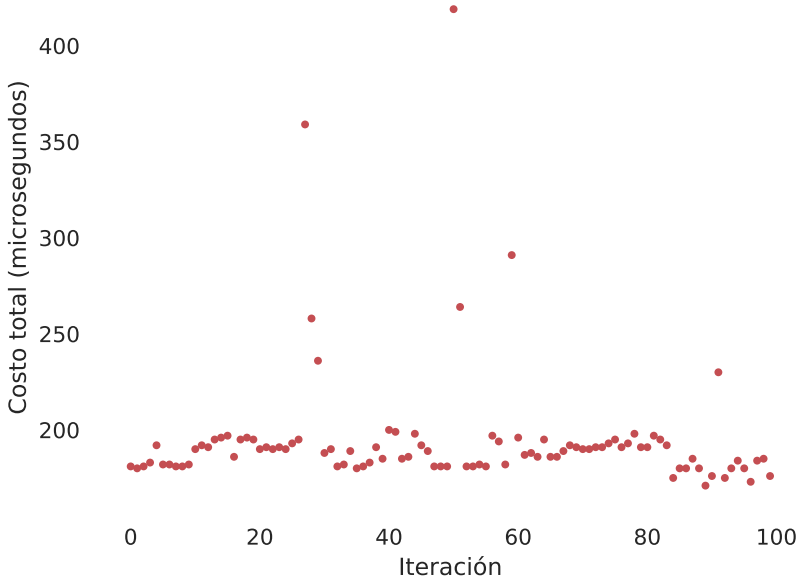


Fig. 3.5: Recorte de 1000 iteraciones donde cada punto representa el tiempo de la misma en microsegundos. Se puede observar varianza en los datos pero todos se mantienen en el mismo orden (microsegundos).

Esto nos da una representación más realista del costo de las operaciones realizadas ya que el tiempo de procesamiento no es "tapado"por la llamada bloqueante del *buffer*. Más allá de las fluctuaciones (que son esperables dado que cada iteración procesa diferente información), se pueden observar claramente los saltos y es importante notar que ahora el orden es de microsegundos (µs), mientras que antes eran milisegundos (ms). Así establecemos el marco para realizar las mediciones, comenzando por entender los tiempos originales del sistema, es decir, los tiempos del *Simulador sin cambios*.

3.3.3. Simulador sin cambios

Ya con un mejor entendimiento de las mediciones a realizar, observaremos el promedio y la distribución de los datos del sistema original (sin cambios) para diferentes interacciones (cuerda, palanca y fricción).

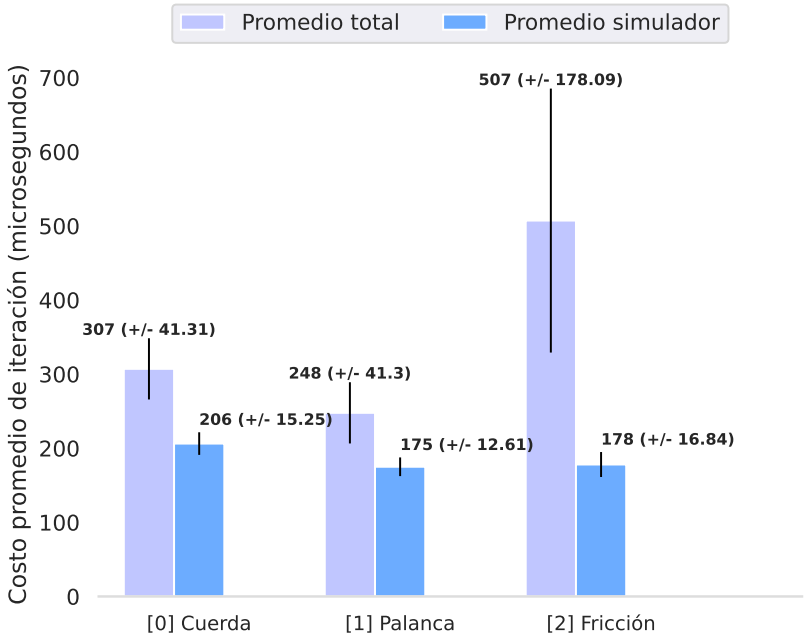
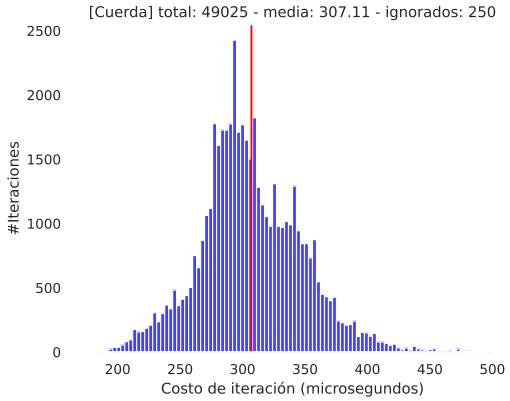
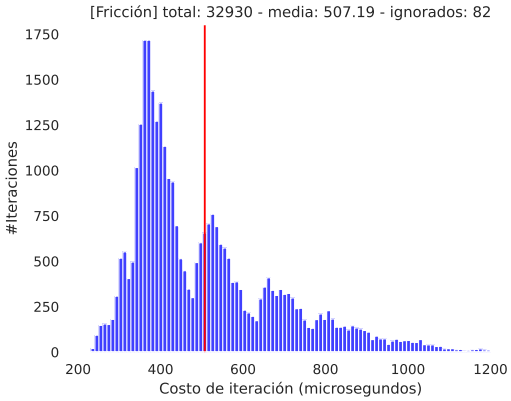


Fig. 3.6: Simulador original. Aquí podemos observar el tiempo promedio de las iteraciones cuando se tocan las cuerdas, se mueve la palanca o se cambia la fricción. También se observa el costo temporal de realizar las operaciones en el simulador junto con sus varianzas.

Interacción	#iteraciones	#iteraciones ignoradas (outliers)
Cuerda	49025	250
Palanca	57274	279
Fricción	32930	82

por otro lado, la distribuciones de los datos para el tiempo total de las iteraciones es:





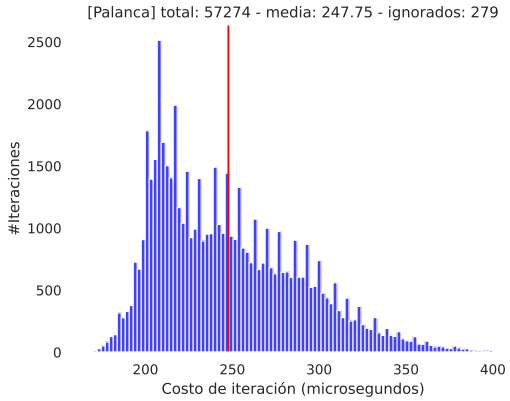


Fig. 3.8: Distribución del tiempo de las iteraciones, donde cada barra representa la cantidad de iteraciones (eje de ordenadas) que tardaron un determinado tiempo (eje de abscisas). La línea vertical roja es el promedio de dicha distribución.

El objetivo principal de estas mediciones es realizar una ponderación temporal del sistema previo a las modificaciones que introduciremos en la arquitectura, para así tener con qué comparar futuros cambios.

Los tiempos obtenidos están en el orden de los microsegundos, lo que hace posible que muchas variables puedan impactar en las mediciones (como pueden ser otros procesos del sistema), para minimizar esta interferencia todas las mediciones fueron hechas bajo contextos y recursos similares.

3.3.4. Simulador con State

Ya habiendo obtenido los resultados sobre el sistema sin cambios y definidos los parámetros a experimentar[5] (ver sección 3.3.3), hemos realizado las siguientes mediciones sobre el sistema implementando el `State` (definido en la sección 3.1) y realizando la actualización del mismo a partir de una archivo de texto al inicio del programa (notar que previamente los mismos estaban en variables en el código). Los resultados fueron:

[5] Cuerda, palanca y fricción

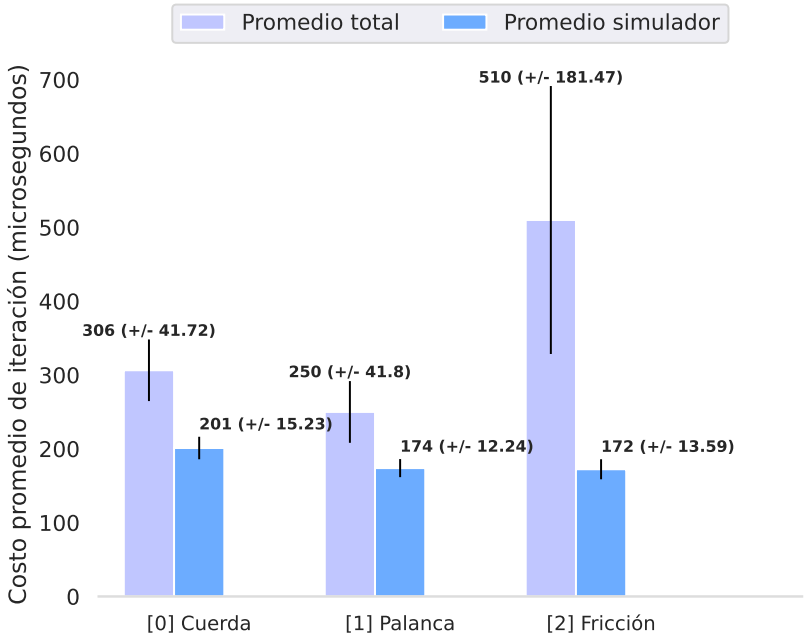
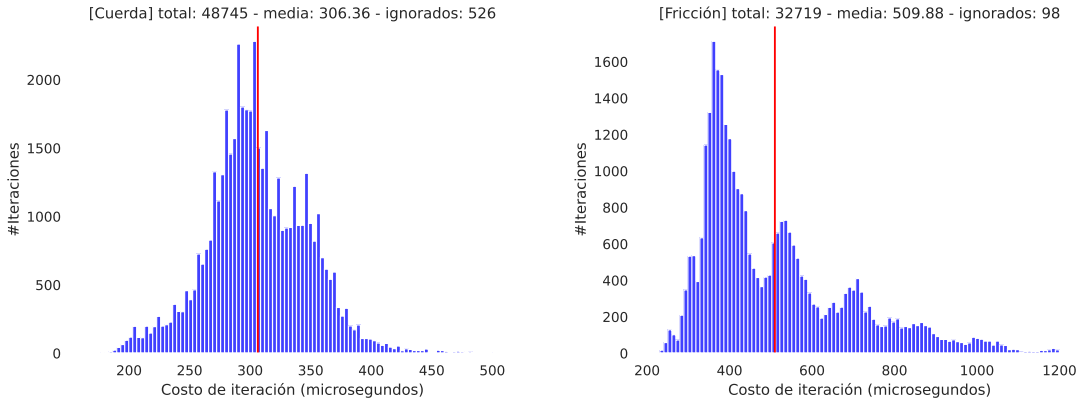


Fig. 3.9: Simulador con `State`. Se puede observar como los valores continúan siendo similares a los valores originales [Fig. 3.6]

Interacción	#iteraciones	#iteraciones ignoradas (*outliers*)
Cuerda	48745	526
Palanca	57615	235
Fricción	32719	98

Con una distribución del costo total de las iteraciones:



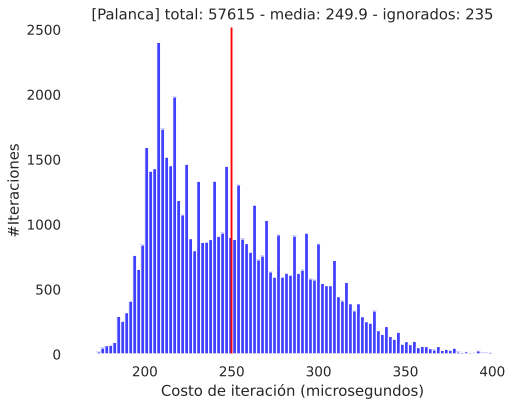


Fig. 3.11: Distribución de tiempos por número de iteraciones. La misma se mantiene consistente con los datos obtenidos en la medición del sistema original.

Podemos observar que los costos se mantienen muy similares a los valores originales, esto es esperable ya que seguimos manteniendo el acceso directo a las variables a través del `State`. Sin embargo, aun no logramos interactuar con los parámetros de forma externa, siendo éste uno de los objetivos principales de los cambios. Nuestra primera aproximación será utilizar un archivo de texto como interfaz principal entre el usuario y el sistema.

3.3.5. Carga de archivo

Hasta este punto realizamos la carga del archivo con los parámetros solo una vez al inicio del sistema, por lo que los únicos parámetros beneficiados con este cambio pueden ser los de configuración. El objetivo ahora es agregar los parámetros de ajuste, por lo que necesitamos que el sistema lea el archivo de configuración más dinámicamente. Esto lo haremos siguiendo las dos propuestas detalladas en la Sección 3.1.1.

Cada N-iteraciones

La primer propuesta consiste en calcular el tiempo promedio sobre la lectura de los parámetros cada un número (N) de iteraciones. Los resultados obtenidos fueron:

Frecuencia de actualización [N]	Promedio de iteración	Máximo	Resultado
14 ms [10]	-	-	Error al empezar
72 ms [50]	-	-	Error luego de pocas iteraciones
87 ms [60]	-	-	Error a los 2 s
108 ms [75]	541 µs	54 955 µs	Error a los 3 s
145 ms [100]	358 µs	32 116 µs	Error a los 13 s
362 ms [250]	243 µs	31 009 µs	Error a los 47 s

Siendo el siguiente el costo promedio de lectura del archivo de configuración utilizando la biblioteca `libconfig`:

Tiempo	Desviación Estándar
10.12 ms	2 ms

Es así que podemos observar que el costo promedio de lectura de archivos es muy alto, pero el promedio general de las corridas es bajo. Lo que sucede es que el costo grande se produce cada N iteraciones y, teniendo en cuenta que cada promedio contiene al menos 60 mil iteraciones, hace que los costos grandes se vean tapados por tantos pequeños.

Esto lo podemos ver claro en la columna "máximo"[6] que representa el costo de la iteración que más tarda (junto con el desvío estándar). Nos permite ver que el tiempo sobrepasa por mucho el buscado (726 µs [ver detalles en la sección 1.3.3]) produciendo el siguiente error en la biblioteca utilizada:

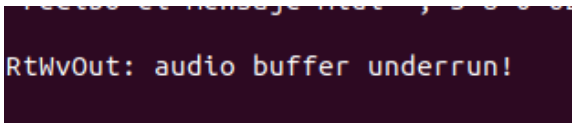


Fig. 3.12: Error producido por la biblioteca `Rtwvout` cuando no se llega a llenar los *buffers* de `STK` a tiempo. En estos casos el sonido empieza a sufrir distorsiones

Indicando que no se logró cargar el *buffer* de sonido lo suficientemente rápido para mantener el tiempo real, causando una interrupción catastrófica del flujo.

No obstante vemos que simplemente no es viable leer un archivo de configuraciones (que puede ser arbitrariamente grande) dentro del flujo principal del programa, ya que no podemos garantizar la fluidez del sonido sabiendo que cada N iteraciones vamos a tener picos en el tiempo de procesamiento. Por esto surge la necesidad de crear una entidad que se encargue de realizar este tipo de tareas costosas en un hilo aparte.

3.3.6. Administrador de estados

Requisitos y Configuración - Archivo

Hasta ahora, hemos realizamos los experimentos utilizando `Rosegarden` con tres tipo de operaciones (cuerda, palanca y fricción). Ahora necesitamos ver qué sucede al cambiar la fricción desde el archivo de texto (que contiene los parámetros) para así poder comparar la implementación con `stateManager` contra `MIDI`. En este caso solo nos limitamos a la experimentación de la fricción porque la palanca no tiene sentido cambiarla desde un archivo (ya que es un valor dinámico que siempre vuelve a cero cuando se suelta y esta interacción no puede ser representada en un archivo).

Fue así que creamos un nuevo archivo en `Rosegarden` que solo contiene toques de cuerda y a su vez elaboramos un *script* que cada medio segundo (500 ms) cambia el valor de la fricción de las cuerdas en el archivo con los parámetros.

De esta manera, los nuevos pasos son:

1. Inicializar `Jack`
2. Inicializar `Rosegarden` abriendo el archivo de prueba
3. Inicializar el simulador (*make run* en terminal)
4. Conectar `Rosegarden` al sistema

[6] Aquellas columnas sin datos corresponden a corridas sin muestras suficientes para calcular el tiempo promedio

5. Presionar Play

6. Iniciar el script (*make tocar* en terminal)

Así, el archivo generado contendrá el mismo formato descripto en 3.3.1 y se generaran los gráficos correspondientes.

Experimentación

Con el aliciente de realizar operaciones costosas sin afectar el flujo del sistema, aquí realizaremos experimentaciones utilizando el administrador de estado (ver sección 3.2) con la carga de archivos en un *thread* aparte, es decir, que cada un tiempo determinado (M) el archivo se carga nuevamente.

El siguiente gráfico muestra los tiempos promedios para la lectura de los parámetros cada 500 ms ($M = 500$), valor que es aceptable para parámetros que no son de toque.

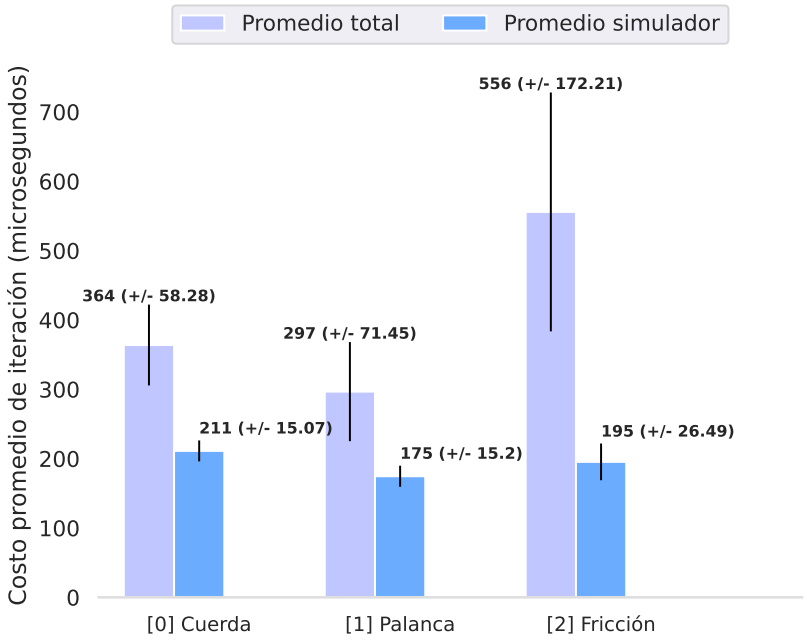


Fig. 3.13: Simulador con Administrados de Estado. Aquí observamos que el costo promedio del simulador no es afectado aunque se ve un pequeño incremento de los costos totales respecto al original

A continuación se muestra la distribución de los datos anteriores:

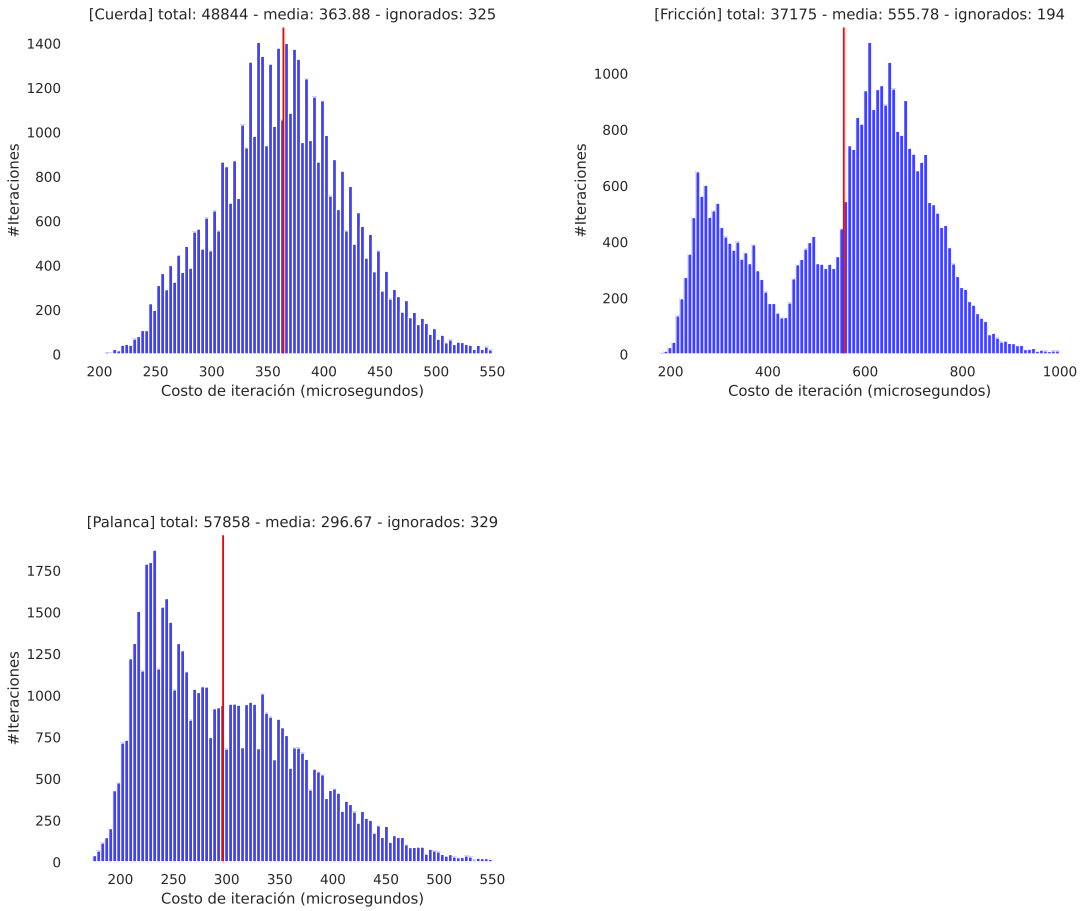


Fig. 3.15: Distribución de tiempos por #iteraciones utilizando el `StateManager` y múltiples hilos de ejecución. En todos estos casos es utilizado `MIDI`

Interacción	#iteraciones	#iteraciones ignoradas (outliers)
Cuerda	48844	345
Palanca	57858	329
Fricción	37175	194

A diferencia del caso anterior 3.3.5, no tenemos picos cada n iteraciones, sino que los tiempos se ven mucho más similares a las primeras experimentaciones (que no poseían lectura de archivo).

También podemos observar pequeñas variaciones en cuanto a los costos, especialmente para la fricción. Esto nos permite observar una desventaja en la utilización de *pooling* sobre archivos. Estamos escribiendo el `State` a través del `StateManager` utilizando `MIDI`, pero a su vez, el `StateManager` está leyendo un archivo (que contiene todos los parámetros) cada 500 ms, lo que hace que sobre-escriba el valor cargado por `MIDI`. Esto, además de generar sobre-procesamiento y latencia, hace que los cambios efectuados por `MIDI` no puedan ser apreciados correctamente.

Sin embargo, siendo solo de carácter experimental (ya que en la práctica solo se utilizará una interfaz para la actualización de un mismo parámetro), los valores obtenidos siguen dentro del rango esperado, por lo que nos permite ver que esta arquitectura no

introduce latencia significativa en el simulador (gracias a la utilización de múltiples hilos de ejecución).

Por su parte, en la experimentación previa, los cambios son realizados utilizando `MIDI` (ya que la idea era poder comparar con las experimentaciones anteriores) pero lo interesante a observar es qué sucede cuando se introduce un cambio directamente en el archivo de configuraciones (en vez de `MIDI`).

A continuación analizaremos los costos de cada iteración donde se actualizó el metaparámetro "fricción por cuerda" utilizando el archivo de configuraciones en vez de `MIDI`:

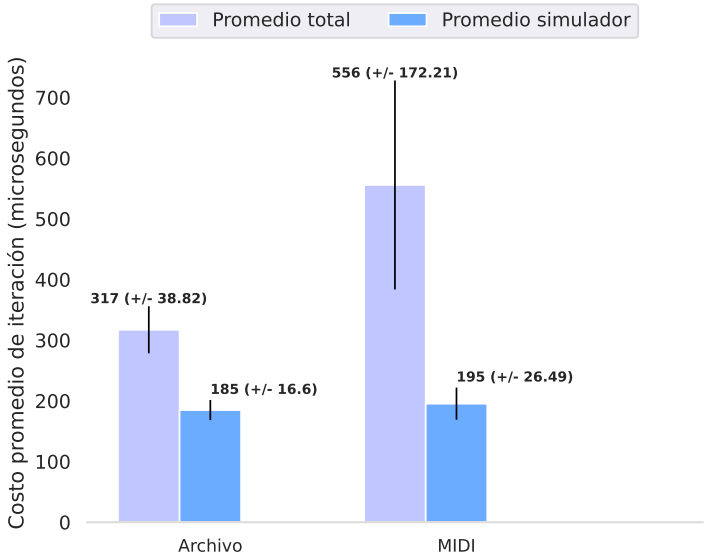


Fig. 3.16: Comparación de interacción con `MIDI` contra archivo. Aquí se compara utilizando la misma metodología utilizada hasta el momento para las pruebas con `MIDI` mientras que para el caso del archivo, se utiliza un script que modifica el archivo de configuraciones cada 500 ms [3.3.6], cambiando la fricción de las cuerdas.

Comparamos las distribuciones entre el archivo de configuraciones y `MIDI`:

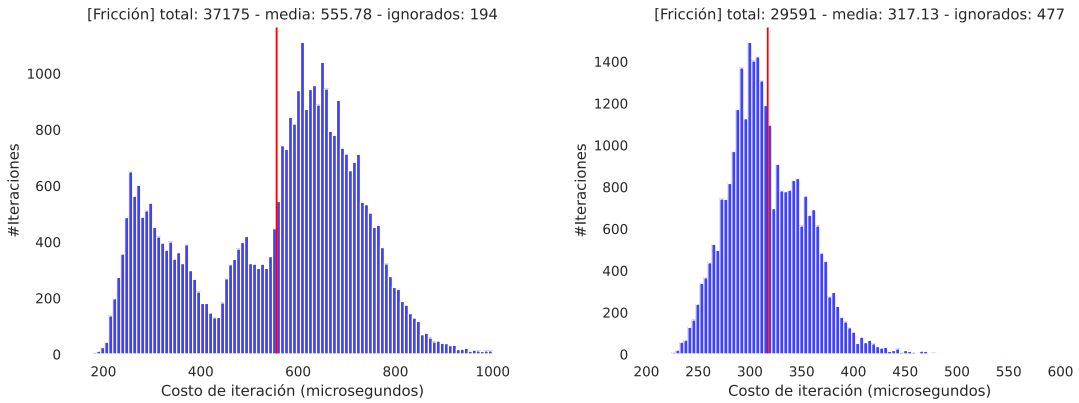


Fig. 3.17: Comparación de distribución entre `MIDI` (izquierda) y Archivo de configuraciones (derecha). Aquí se puede observar una clara mejora (40 % aprox) en los tiempos del uso del archivo de configuraciones contra `MIDI`, volviendo a los tiempos promedios originales que tenia el sistema pero utilizando un método mucho más costoso como es la lectura de archivo en disco.

Interacción	#iteraciones
Fricción MIDI	37175
Fricción Archivo	29591

La mejora del archivo contra `MIDI` es esperable ya que la escritura y generación de variables (a partir de las meta-variables) se ejecuta en el Administrador de Estado, mientras que en `MIDI` se ejecuta en el hilo principal. La "desventaja" es que el tiempo que tardan los parámetros en escucharse aplicados es el tiempo de *pooling* (500 ms o el tiempo configurado) más el tiempo que lleva la lectura y cálculo de los parámetros, que puede variar según lo modificado. Es por esto que es importante ser eficiente también en la actualización y cálculo de parámetros (mas allá que esto no afecte directamente al sistema principal); ya que la latencia percibida por el usuario puede variar basado en esto.

3.3.7. Conclusión

En este capítulo hemos introducido por partes una arquitectura que nos trae tres grandes beneficios: permite una interacción más flexible entre los parámetros y el sistema, nos posibilita la realización de operaciones costosas sin impactar en el flujo principal y es tolerante ante fallos.

La flexibilidad de la interacción es lograda mediante el `State` donde su utilización, en términos de costos temporales, es equivalente a llamar a las variables directamente (3.3.4) pero, en términos de beneficios, nos permite actualizar las variables en cualquier momento y de forma muy sencilla. Recordemos aquí la importancia de identificar correctamente los parámetros del sistema [3.1], ya que hay variables (que no son parámetros), que de modificarse podrían afectar negativamente al sistema (como punteros, por ejemplo).

Luego hemos interactuado con el `State`, y vimos que realizar operaciones costosas sobre el flujo principal (como cargar un archivo) generó retraso en algunas iteraciones que invalidaron el sistema por completo [3.3.5]. Esto nos permitió introducir y experimentar con el `StateManager`, y mostramos que su utilización no perjudicó el flujo principal (3.3.6) sino que por el contrario, generó menos impacto en lo que respecta al cálculo a partir de

meta-parámetros [3.16] (ya que muchas operaciones se realizan en un hilo aparte y los resultados son copiados al `State` una vez completadas las mismas).

Por su lado, la posibilidad de generar un sistema de backup automático nos permitió que operaciones inestables, como lo es la lectura en disco (donde el archivo puede estar siendo leído y escrito a la vez) sean realizadas de forma segura. Acá es importante resaltar que este sistema lo permite ya que en el peor de los casos[7] los últimos cambios no se verán aplicados (forzando al músico a realizarlos de nuevo) pero el instrumento seguirá en funcionamiento, mientras que en otro tipo de sistemas, volver atrás en sus parámetros puede no ser viable.

Finalmente, entendemos que actualizar los parámetros mediante *pooling* sobre un archivo de texto no es una solución óptima para el usuario, ya que muchos son dinámicos (requiriendo mayor interacción) y trae problemas cuando hay más de una fuente escribiendo en el `State`, creando una suerte de condición de carrera para la escritura de un parámetro.

Estos hechos hacen que nos planteemos la necesidad de utilizar otro tipo de implementación e interfaz para la interacción con el usuario, manteniendo la misma arquitectura ya que trajo excelentes resultados en cuanto al impacto en el flujo principal (especialmente para operaciones costosas como manejar el sistema de archivos). Dándonos una base para realizar mejoras sin afectar el programa principal. Esto será tratado en el siguiente capítulo.

[7] Cuando hay un error no capturado.

4. CONFIGURACIÓN MEDIANTE DISPOSITIVO EXTERNO

Hasta el momento logramos definir un patrón eficiente que permite abstraer el manejo del estado del simulador utilizando un archivo con los parámetros, lo que resulta conveniente como primer abstracción y prueba de concepto, pero muy poco práctico para el usuario final. Esto se debe a que el simulador no va a ser accesible directamente por el músico. Podemos pensar que el mismo va a estar corriendo en un dispositivo embebido al cual se le van a conectar diferentes artefactos de entrada y uno de salida (los parlantes).

Entre los periféricos de entrada se debe encontrar el controlador (con los sensores para las cuerdas) y un dispositivo que permita configurar los parámetros de configuración y ajuste. Este último, a su vez, liberará al sistema principal de la realización de operaciones con meta-parámetros e introducirá nuevos desafíos como es la comunicación entre dispositivos.

Es así que hemos decidido construir una aplicación `Android` (ya que en Argentina posee más del 80 % del mercado [25]) que se conecta a través de una interfaz *Bluetooth* con el sistema para permitirle al usuario más libertad a la hora de interactuar con el instrumento. Para esto hemos desarrollado y experimentado con una aplicación construida de cero para este trabajo.

4.1. Interfaz

La interfaz debe permitirle al usuario configurar parámetros de ajuste y configuración, como a su vez configuraciones propias de la aplicación (como cargar valores pre-establecidos, conectarse al *Bluetooth*, actualizar propiedades, etc).

La misma consta de cuatro pantallas (utilizando fragments [26]) junto con un menú inferior que permite el movimiento entre los mismos:

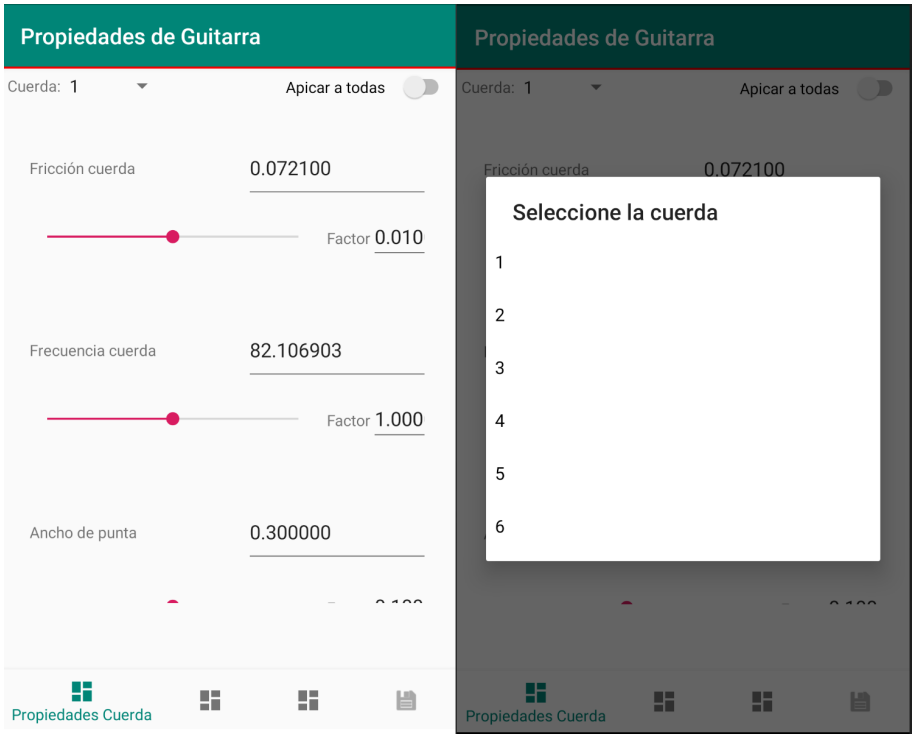


Fig. 4.1: Primera pantalla - Propiedades de cuerdas. Aquí se pueden actualizar los parámetros por cuerda o cambiar todas a la vez seleccionando “Aplicar a todas”

Esta primera pantalla [Fig. 4.1] contiene un campo desplegable (Spinner [27]) que permite seleccionar la cuerda a configurar. También se puede seleccionar el cuadro de verificación (Checkbox [28]) para aplicar los cambios a todas las cuerdas. Notar que esta pantalla contiene solo meta-variables que afectan propiedades de la cuerda. Es decir, que cada cambio hace que se calculen los parámetros reales del programa mediante algoritmos pre-establecidos [2.2.2].

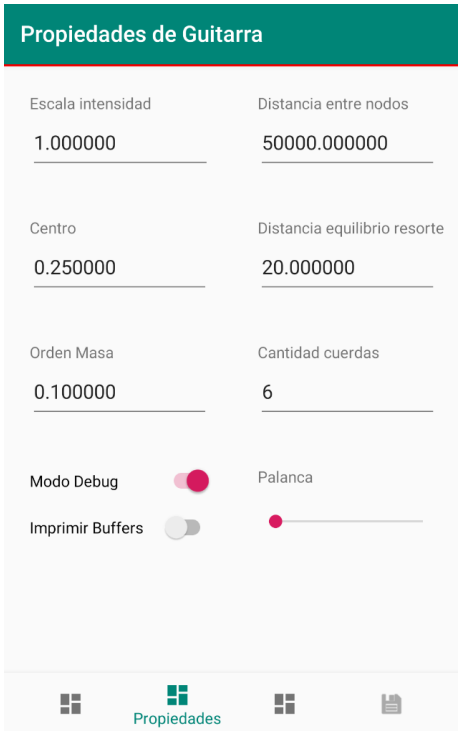
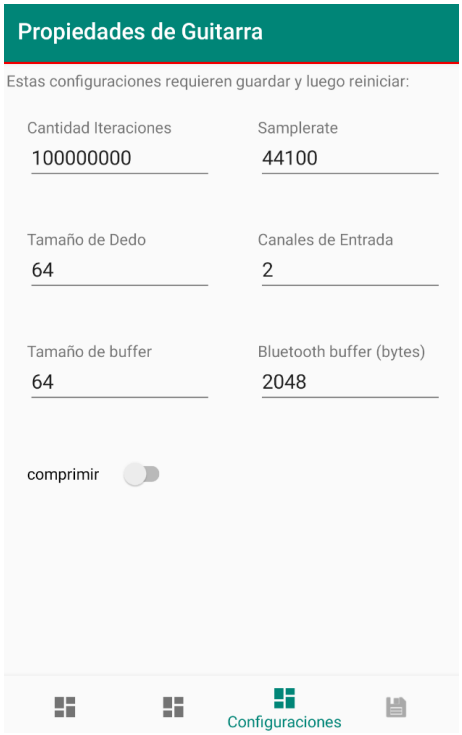


Fig. 4.2: Segunda pantalla - Parámetros de ajuste. Aquí se pueden actualizar los parámetros de ajuste. Éstos impactan al sistema de forma general, es decir, no dependen de un nodo o una cuerda.

Al igual que la pantalla anterior [Fig. 4.1], en la segunda [Fig. 4.2] se podrán configurar parámetros de ajuste. La diferencia radica en que éstos no dependen de la cuerda seleccionada. Aquí algunos son meta-parámetros y otros no, pero todos al ser de ajuste se aplican en el sistema al momento de cambiarlos.

Por su parte, la tercera pantalla [Fig. 4.3a] corresponden a los parámetros de configuración, por lo que solo se aplican cuando el simulador es reiniciado (ya que contienen información relevante para el inicio del simulador). Es por esto que los cambios efectuados aquí no son aplicados en tiempo real.



(a) Tercera pantalla - Parámetros de configuración. La mayoría de estos parámetros solo impactan al inicio de la sincronización (cuando se conecta el dispositivo con el sistema), por lo que los cambios realizados luego de esta no tendrán impacto.

(b) Cuarta pantalla - Configuración de aplicación. Aquí se le asignan las configuraciones propias del dispositivo y de experimentación.

Finalmente la cuarta pantalla [Fig. 4.3b] nos permite guardar todos los parámetros configurados con un perfil en particular, cargar los mismos y establecer la conexión *Bluetooth*, entre otros.

Todas las pantallas están pensadas para ser utilizadas sobre dispositivos móviles que soportan `Android 7.1` o superior.

4.2. Conectividad

Como se ha adelantado la conectividad la realizaremos mediante *Bluetooth*. Esto se debe a que, si bien necesitamos que los parámetros se apliquen de forma veloz, buscamos un equilibrio entre una latencia aceptable y la usabilidad. Notar que esto no aplica a los parámetros de toque ya que un mínimo retraso puede afectar la interpretación musical. El foco está puesto en el manejo de los parámetros de configuración y ajuste.

La clara ventaja que ofrece *Bluetooth* es la comodidad de no utilizar cable, permitiendo ubicar la *tablet* o celular donde resulte más cómodo. Su gran compatibilidad (todos los dispositivos móviles, junto con el embebido que utilizaremos, poseen *Bluetooth*) es otro de sus beneficios.

Lo que buscamos aquí es evaluar dicha conectividad con la arquitectura definida, obteniendo así conclusiones sobre la conveniencia de su uso en este tipo de sistemas. Adicio-

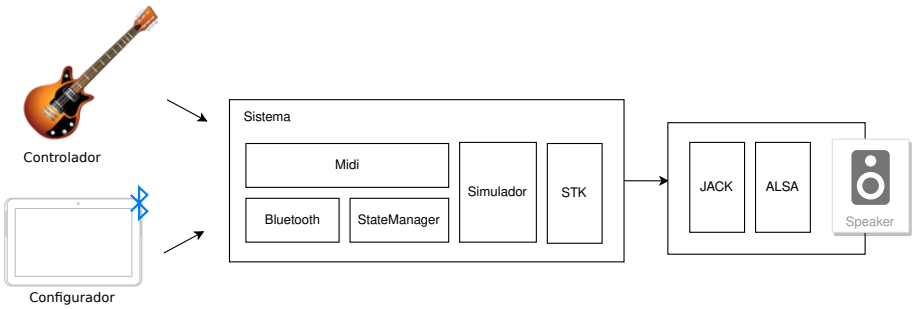


Fig. 4.4: Esquema de los principales componentes presentes en la nueva versión del simulador. Aquí observamos las diferentes interacciones que posee la información desde que ingresa al sistema mediante `MIDI` o *Bluetooth*, hasta que sale del mismo utilizando `STK`

nalmente aportamos una posible implementación con funcionalidad real, lo que convierte a las pruebas realizadas en un desarrollo en sí mismo.

4.2.1. Bluetooth

Bluetooth [29] es una tecnología inalámbrica usada para el intercambio de información entre dispositivos. Más específicamente, éste es un conjunto de protocolos muy completo que define desde la parte física, como la frecuencia de radio (cómo modular y demodular las señales), hasta las instrucciones que el programador debe realizar para establecer la comunicación.

Un protocolo importante descripto en dicha especificación es el de transporte. Cuando los dispositivos quieren establecer una conexión saliente, deben elegir el dispositivo al cual se van a comunicar y el protocolo de transporte a utilizar. Una situación similar sucede con el dispositivo que acepta la conexión, ya que debe seguir dicho protocolo.

Existen varios protocolos usados para dicha capa, pero nosotros nos centraremos en `RFCOMM` que es el utilizado en la biblioteca (SDK) de `Android`[1].

El protocolo de comunicación de radio frecuencia (`RFCOMM`) es un protocolo de flujo confiable de datos. Éste es el más usado por esta tecnología, ya que provee los mismos servicios y confiabilidad que TCP. Siendo de propósito general, se basa en realizar una emulación de un puerto serial sobre otro protocolo (`L2CAP`).

Sin ahondar en detalles, `L2CAP` puede ser comparado con UDP, ya que está basado en enviar paquetes con una política de "mejor-esfuerzo", es decir, no da garantías de que todos los paquetes lleguen, por lo que la garantía de los mensajes se debe forzar en la capa superior.

Dado el protocolo de transporte a utilizar, el manejo de la comunicación entre dispositivos se puede realizar mediante *sockets*. Ésto nos permite, mediante operaciones sencillas que detallaremos a continuación, realizar la comunicación entre dos dispositivos.

4.2.2. Servidor Bluetooth

El servidor tiene como función principal recibir los datos enviados por el cliente. El mismo fue implementado en `C++` mediante *sockets*, haciendo que establecer la conexión

[1] Disponible en `https://www.android.com`.

conste de tres pasos básicos. Primero, la aplicación debe enlazar el *socket* (identificado con un número) con el adaptador *Bluetooth* (*bind*). Luego debe poner el *socket* en modo de escucha (*listen*) para que el sistema operativo acepte conexiones entrantes desde el adaptador hasta el *socket*. Y, finalmente, se debe obtener el *socket* conectado con el fin de transferir datos (comando `accept`).

Una vez obtenido el *socket* conectado, se pueden enviar y recibir datos utilizando las instrucciones `send` y `recv` (aunque en nuestro caso solo usamos `recv`). En el caso del `recv` es una llamada bloqueante que se desbloquea solo cuando recibe datos o hay un error (en donde retorna 0 o -1 bytes). Estos datos son recibidos a través de un *buffer* (en nuestro caso de 1024 B) que se pasa como parámetro al método `recv` para que escriba en el mismo.

Para la implementación hemos decidido crear una clase encargada de abstraer el manejo del *socket* (`SocketIterator`), de forma tal que el programador solo tenga dos operaciones:

- **hasNext**: Devuelve verdadero en caso de que haya algún carácter para leer en el *buffer*. Si el *buffer* actual se leyó en su totalidad, se hace otro pedido al *socket* mediante la operación `recv` y, de haber más valores, devuelve verdadero o, en caso contrario, falso. Si ocurriera un error, se utiliza el mecanismo de excepción para indicarlo. Notar que esta llamada es bloqueante, por lo que de no haber valores, el hilo se quedará bloqueado hasta que los hubiera.

- **next**: Devuelve el carácter actual del *buffer* y avanza a la posición. Notar que esta llamada es bloqueante, por lo que de no haber valores, el hilo se quedará bloqueado hasta que los hubiera.

Así, este nuevo objeto es el encargado de manejar el *buffer* del *socket* y el programador solo tiene que pedir el siguiente valor y decodificar los datos recibidos (mensajes). Dicha decodificación será profundizada en una de las secciones subsecuentes (ver 4.2.4).

Siguiendo con el diseño, el siguiente gráfico representa el comportamiento del servidor implementado, junto con su interacción con el `StateManager`[2].

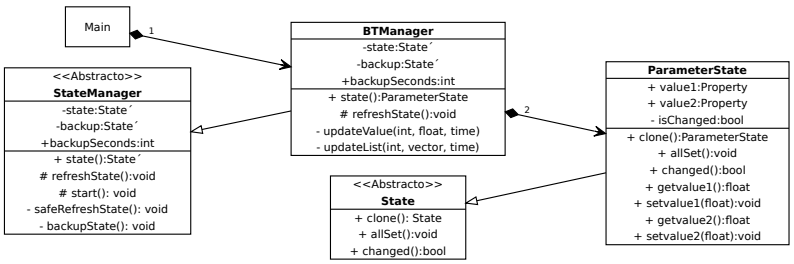


Fig. 4.5: Diseño del `StateManager` utilizando una implementación con *Bluetooth*. Aquí `BTManager` y `ParameterState` son clases concretas (implementaciones) de `StateManager` y `State` respectivamente.

Aquí podemos observar cómo el Administrador de estado (`BTManager` de tipo `StateManager`) implementa el servidor *Bluetooth* detallado previamente (dentro del método `refreshState` utilizando el `SocketIterator`). Éste, además de establecer y mantener la conexión con *Bluetooth*, se encarga de actualizar el `State` al recibir un nuevo mensaje.

De esta manera las responsabilidades quedan definidas como:

[2] Definido previamente en la sección 3.2.

- `StateManager`: Definir y mantener el ciclo de vida del `State` del programa junto con todos los objetos que lo modifiquen. A su vez establece y mantiene la comunicación *Bluetooth*.
- `State`: Contiene los parámetros junto con información de cuándo fueron actualizados.
- Flujo principal: Sigue siendo el encargado de tomar los parámetros cuando son modificados, llamar al simulador y escribir los resultados en el *buffer* de sonido.

4.2.3. Cliente Bluetooth

Para la creación del cliente, `Android` especifica en su documentación los siguientes pasos [30]:

1. Con el objeto *`Bluetooth`*`Device`, obtener el *socket* que utilizaremos para conectar el cliente con el dispositivo (se usa la función `createRfcommSocketToServiceRecord`).
2. Inicializar la conexión llamando a `connect()`. Cuando se realiza este llamado el sistema ejecuta una "búsqueda SDP"[3] para encontrar el servidor basado en el id recibido. Si se lo encuentra, se compone el canal `RFCOMM` que se usará en la conexión.

Notar que `connect()` es una llamada bloqueante, por lo que se debe realizar en un *thread* separado (caso contrario, se bloquearía la interfaz del usuario).

Este cliente lo podemos encontrar en la clase `BluetoothService` que implementa los métodos `write` (envía los datos) y `connect` (establece conexión), entre otros. Algo a destacar es que antes de llamar al método `write` los valores a enviar (mensajes) deben ser convertidos de forma tal que el servidor luego pueda interpretarlos, esta conversión la veremos en la siguiente sección.

4.2.4. Codificación de mensajes

En cualquier comunicación entre dispositivos es importante definir un protocolo para los mensajes de forma tal que el servidor pueda interpretar los datos recibidos por el cliente.

En nuestro caso hemos definido un protocolo muy sencillo que permite el envío de los parámetros mediante *Bluetooth*. Éste consiste en asignarle un valor numérico a cada parámetro según la tabla de asignaciones (ver sección 4.2.4) y luego utilizar dos puntos (":") junto con el valor del parámetro. Este valor, a su vez, puede ser un `float`, un `bool` o un vector de `float`:

- `bool`: el valor se representa con 0 para *false* y 1 para *true*
- `float`: se manda el valor directamente
- `array`: Se manda el listado de `float` separado por coma (",") encerrado por corchetes ("[]").

A su vez todo mensaje comienza y termina con un carácter especial (& y ; respectivamente). De esta manera, si queremos mandar muchos parámetros a la vez es posible identificar el comienzo y final de cada uno.

[3] Protocolo para encontrar servidor [31].

Tabla de asignaciones

La siguiente tabla muestra las asignaciones numéricas realizadas para cada parámetro definido.

Nombre	Asignación	Nombre	Asignación
nFrames	0	samplerate	1
nbufferii	2	canalesEntrada	3
npot	4	dedoSize	5
palanca	6	debugMode	7
imprimir	8	escalaIntensidad	9
distanciaEquilibrioResorte	10	distanciaEntreNodos	11
centro	12	maxp	13
expp	14	ordenMasa	15
masaPorNodo	16	friccionSinDedo	17
friccionConDedo	18	minimosYtrastes	19
cantCuerdas	20	nodos	21
frecuencia	22	maxFriccionEnPunta	23
anchoPuntas	24	distanciaCuerdaDiapason	25
distanciaCuerdaTraste	26	friccion	27

Ejemplo

El siguiente ejemplo muestra la codificación de un mensaje con tres parámetros distintos, los mismos son:

- fricción: [0.0001,0.0002,0.0003]
- imprimir: *true*
- palanca: 30

La codificación sería:

```
1 &27:[0.0001,0.0002,0.0003];&8:1;&6:30;
```

donde & y ; indican que empieza y termina el envío de un parámetro, 27, 8 y 6 representan el parámetro identificado y lo restante son los valores de cada parámetro.

Aquí no buscamos una codificación óptima sino una sencilla que nos permita realizar experimentaciones y que sea lo suficientemente eficiente (en cuanto a bytes enviados) para cumplir con las esperas aceptables a la hora de cambiar un parámetro. Sin embargo, queda para futuros trabajos la experimentación de codificaciones adaptables a cada parámetro ya que, como veremos en la siguiente sección, se pueden realizar diferentes optimizaciones que comprimen notoriamente determinados tipos de mensajes (en nuestro caso, la actualización de la fricción).

Optimizaciones

En este trabajo propondremos y evaluaremos solo una simple optimización para el envío de *arrays* ya que no es el objetivo de la misma encontrar una codificación óptima para este caso de uso.

El problema que buscamos solucionar con esta optimización es el envio de muchos valores repetidos consecutivos en los *arrays*, ya que cuando generamos parámetros a partir de meta-parámetros (especialmente los de fricción) es muy común observar este tipo de patrón. Ésto, como se mencionó en la Sección 2.2.2, se debe a que el algoritmo utilizado para el cálculo con los meta-parámetros está basado en dos gaussianas que modifican sus valores especialmente en las puntas (mientras que la mayoría de los centrales se mantienen iguales).

Es así que introduciremos un nuevo carácter "x"que nos indica la cantidad de veces que se repite un valor. Por ejemplo, mientras que antes teníamos

```
&27:[0.0001,0.0002,0.0002,0.0002,0.0002,0.0003];
```

con el nuevo carácter podemos codificar:

```
&27:[0.0001,0.0002x3,0.0003];
```

de esta manera reducimos sustancialmente la cantidad de caracteres repetidos consecutivos a enviar.

A continuación presentaremos el algoritmo utilizado:

```
String MULTIPLICADOR = 'x';
String comprimir(Float[] list) {
    // Escribo los valores en buffer
    StringBuffer temp = new StringBuffer(list.length * 12);
    int prev = 0;
    int count = 0;
    for (int i=1; i<list.length; i++) {
        if (list[i].equals(list[prev])) {
            // Sumo 1 cuando el anterior es igual al actual
            count++;
        } else {
            // Si es distinto debo escribir el anterior
            temp.append(list[prev]);
            if (count>0) {
                // Antes de escribir numero, agrego multiplicador (ya que antes hubo repetidos)
                temp.append(MULTIPLICADOR);
                temp.append(count+1);
            }
            temp.append(',');
            count = 0;
            prev = i;
        }
    }

    // Escribo ultimo valor
    temp.append(list[prev]);
    if (count>0) {
        temp.append(MULTIPLICADOR);
        temp.append(count+1);
    }
    return ":[" + temp.toString() + "]";
}
```

Listing 4.1: Pseudo-código para comprimir lista de `float`

Como se puede observar, la complejidad del algoritmo está dada por la creación del

buffer y el costo de escribir los valores en el mismo[4], donde se itera el arreglo de `float` realizando operaciones constantes sobre cada valor y, finalmente, convirtiendo el *buffer* a *string* que tiene un costo lineal sobre la entrada[5], por lo que la complejidad total es $3 * \mathcal{O}(n)$ que es equivalente a $\mathcal{O}(n)$ con n el tamaño del arreglo a enviar.

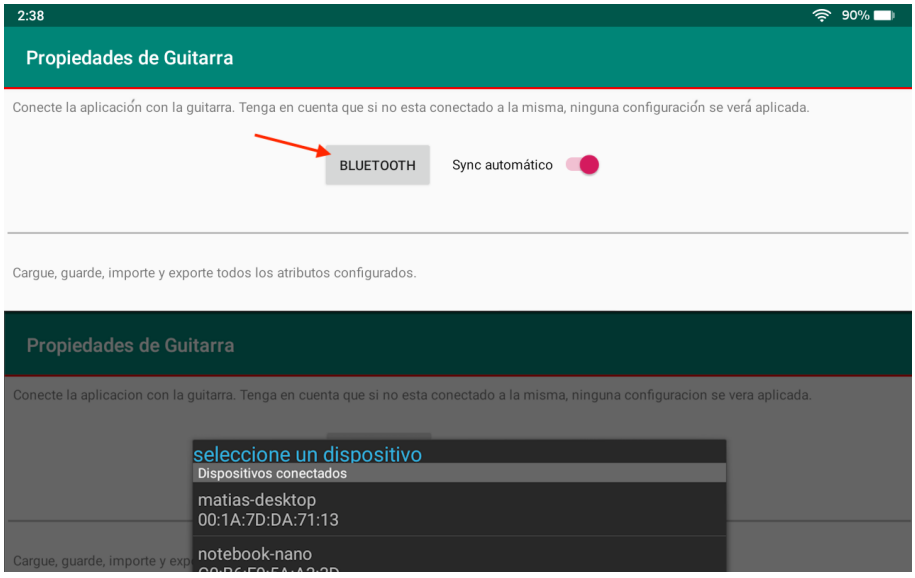
4.3. Experimentación

4.3.1. Requisitos y Configuración - *Bluetooth*

Las pruebas fueron realizadas con el mismo formato y pasos que las pruebas anteriores (ver sección 3.3.1) con la diferencia de que con `Rosegarden` solo generamos toques de cuerdas en un ciclo infinito. Ahora, la palanca y fricción es generada en la aplicación `Android` mediante un botón ("Correr Test") que va cambiando los valores de la palanca y la fricción cada 300 ms.

Así, una vez iniciado el simulador, debemos conectar la *tablet*/celular con la computadora. Para eso:

1. Conectar *tablet*/celular a la PC



2. Presionar "Correr Test".

[4] En la implementación del SDK se ve cómo `append` realiza una asignación en una posición de un *array*, operación que es constante ya que anteriormente se reservó el espacio necesario.

[5] En la implementación del SDK se ve cómo se realiza un copia del *array* interno de `char`.

3. Utilizando `Rosegarden` en PC, ejecutar el archivo que reproduce el tocado de teclas.

Esto hará que se comience a generar mensajes `MIDI` con toques de cuerdas, mientras que se va actualizando la fricción y la palanca desde `Android`.

Al finalizar la experimentación podemos presionar el botón “Stop Test” en la aplicación y nos mostrará lo siguiente:

Fig. 4.6: Información sobre costos de cambiar la fricción en el dispositivo. El promedio y la desviación corresponde a los valores enviados cada 300 ms en las pruebas. El límite de interacciones con la fricción y palanca tomado es 50 mil (para evitar *overflow* en caso que el test se corra por muchas horas).

Luego utilizando estos datos generamos los gráficos que se observaran a continuación.

4.3.2. Impacto en sistema

En esta sección mostramos el costo temporal que esta arquitectura le agrega al hilo principal. Dicha métrica la hemos usado en secciones anteriores y nos ayuda a comparar las diferentes implementaciones.

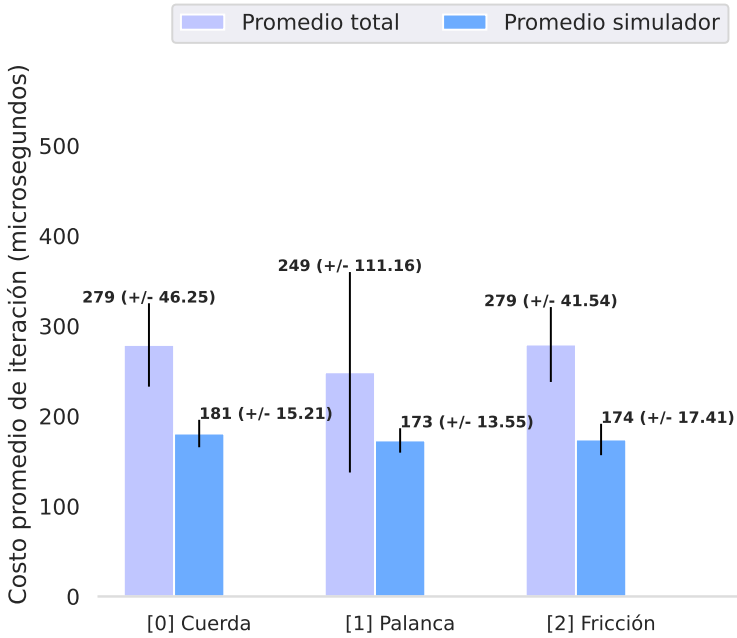
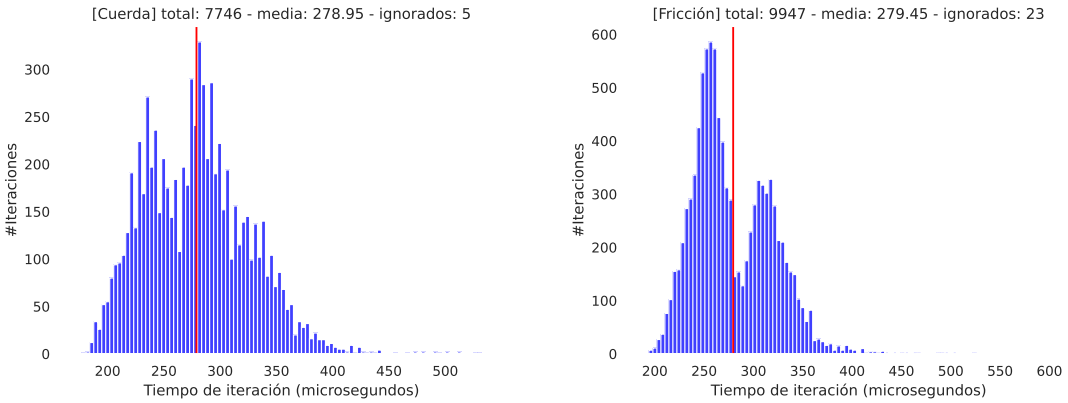


Fig. 4.7: Sistema con Administrador de Estado interactuando con un dispositivo `Android`. Tanto palanca como fricción utilizan `Android` y la cuerda sigue utilizando `MIDI`. Los tiempos obtenidos son un poco mejores a los obtenidos en el simulador sin cambios (ver sección 3.3.3) debido a que algunas de las operaciones se realizan directamente en la aplicación `Android`.

A continuación se muestra la distribución de los datos:



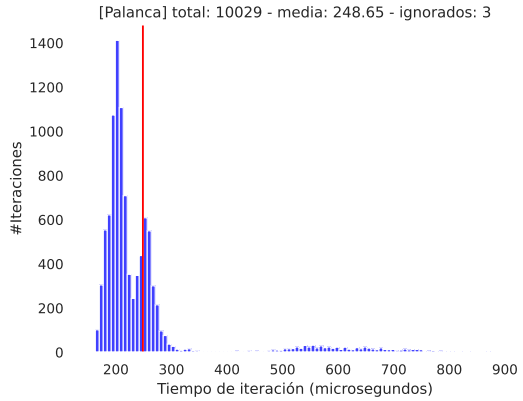


Fig. 4.9: Distribución de tiempos por número de iteraciones utilizando el `StateManager` y múltiples hilos de ejecución. La fricción y la palanca es utilizando `Android` y la cuerda `MIDI`

Recordemos que hay varios hilos que corren por separado (como la recepción de datos por *Bluetooth*, entre otros), por lo que es esperable que la latencia de la conexión no impacte en el flujo principal.

Por otro lado podemos ver que el costo de tocar una cuerda y mover la palanca se mantiene similar al original [6] mientras que el de modificar la fricción, junto con la palanca (que ahora se aplican desde el dispositivo `Android`) tienen un impacto mucho menor en el sistema.

Al igual que en el `StateManager` usando el archivo de configuraciones [3.3.6], esta arquitectura nos permite realizar el cálculo de meta-parámetros fuera del flujo principal (en este caso, en el dispositivo), reduciendo mucho el impacto de los mismos, pero en contrapartida, los cambios no son aplicados en la misma iteración que se producen, es decir, que desde que se cambia la fricción hasta que se escuchan los cambios pueden pasar varios millisegundos. Esta diferencia será tomada en cuenta en la latencia total descripta en la siguiente sección.

4.3.3. Latencia en cambios

Luego de evaluar el impacto en el sistema, nos queda medir lo que realmente tarda un parámetro en aplicarse. A esto lo llamamos *latencia en cambios*.

Realizar esta medición exacta es difícil ya que se requieren herramientas externas que capturen el momento exacto donde se toca la pantalla de la tablet, junto con el momento donde se escucha el cambio aplicado (sin mencionar la dificultad de la detección del cambio en si, ya que a veces es sutil). Esto, a su vez, debemos repetirlo miles de veces para sacar un promedio razonable.

Es así que decidimos realizar el cálculo de esta latencia en base a la suma de diferentes momentos por separado (tanto del dispositivo como del sistema) que intervienen en el cambio de un parámetro. Los mismos resultaron en la sumatoria de:

- *Tiempo de cálculo en dispositivo*: Incluye actualización de variables y cálculo sobre meta-parámetros en el dispositivo.

[6] Ver sección 3.3.6.

- *Tiempo de transporte*: Siendo que el mismo esta dado por:

$$ts_p - tb_p \tag{4.1}$$

donde ts_p = tiempo en el que se guardó el parámetro p en el `State`, tb_p = tiempo en el que leemos el primer valor de p. Como se especificó anteriormente en la sección 4.2.2, los llenados del *buffer* son llamadas bloqueantes, por lo que cuando calculamos tb_p el valor no incluye el tiempo de transporte del primer *buffer*. Esto se debe a que cuando empezamos a medir, el primer *buffer* ya está lleno (caso contrario el hilo estaría bloqueado). Es por esto que las mediciones son realizadas para la fricción, que requiere llenar decenas de *buffers*[7] para transportar los 2400 float, y no un parámetro que contenga un valor solo porque solo requeriría un *buffer* y el tiempo de transporte sería cercano a cero (ya que empezamos a medir una vez que el valor ya se encuentra en memoria).

- *Tiempo desde que se guarda un parámetro en el `State` hasta que se aplica en el simulador*: Este valor resulta de la diferencia entre el momento que se guarda el/los parámetro/s en el `State` y el momento en el que es utilizado por el simulador (cuando se copia a la GPU utilizando `cudaMemcopy`)).

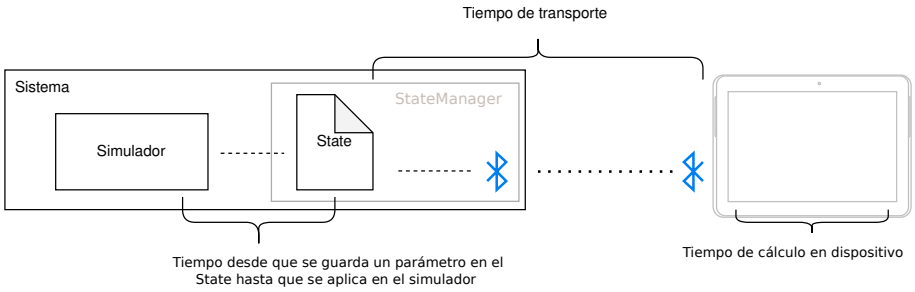


Fig. 4.10: Tres partes principales involucradas en el cálculo de la latencia total desde que se cambia un parámetro en el dispositivo hasta que se aplica en el simulador. La suma de éstas resulta en la aproximación de la latencia total.

El siguiente gráfico muestra la latencia total de la fricción (tomamos ese valor porque consiste en un vector, a diferencia de palanca que es solo un float, y como fue mencionado necesitamos llenar varios *buffers* para hacer posible el cálculo del tiempo de transporte):

[7] Con un tamaño de *buffer* de 1024 MB, 2400 floats requieren 23,3 *buffers*.

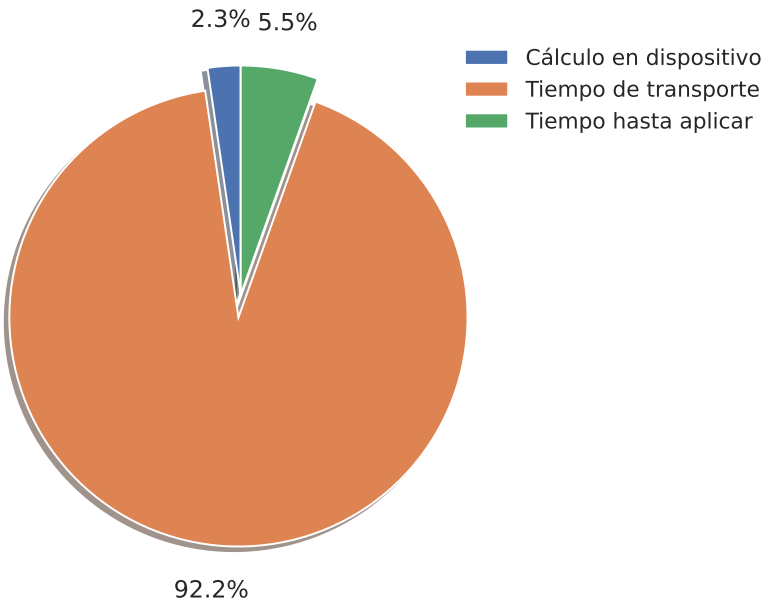


Fig. 4.11: Distribución de los tiempos involucrados al cambiar un parámetro. Promedio de más de 9 mil cambios en la fricción, donde cada cambio en la fricción toma ≈ 108 ms en aplicarse en el sistema.

Descripción	Promedio	Desvío estándar
Cálculo en dispositivo	2.61 ms	1.22 ms
Tiempo de transporte	96.09 ms	82.10 ms
Tiempo hasta aplicar	5.95 ms	5.62 ms

Este valor nos permite dar una idea de lo que tarda un parámetro en ser aplicado. Como podemos observar la latencia de cambiar la fricción está en el orden de los 100 ms, donde más de un 90 % del tiempo corresponde al transporte (*Bluetooth*).

En nuestro caso, y dado que trabajamos sobre parámetros de ajuste y configuración, 100 ms es un valor aceptable por los beneficios que nos trae utilizar *Bluetooth*, pero de utilizar parámetros de toque deberíamos buscar otro protocolo más eficiente ya que necesitaríamos una latencia menor a los 20 ms. Quedando para futuros trabajos la utilización de la arquitectura presentada utilizando otros protocolos de transmisión con parámetros de toque.

Continuando con el análisis, podemos observar que el cálculo de los meta-parámetros en el dispositivo es notoriamente más costoso comparando con anteriores análisis[8] pasando de menos de un milisegundo a casi tres. Ésto se debe a que las mediciones también incluyen el costo de realizar los cambios en cada cuerda (recordemos que actualizamos la fricción en cada una de ellas) y a que estamos ejecutando estos algoritmos junto con otros procesos que están ejecutando en recursos limitados como son los del dispositivo (menos de la mitad de frecuencia e hilos en procesador, tal como se muestra en la tabla de características del sistema en la sección 7.2 en la página 62).

[8] Refiere a cuando se realizaban los cálculos con meta-parámetros dentro del flujo principal mostrado en la sección 3.6.

Una solución al aumento del tiempo de cálculo con meta-parámetros puede ser operar directamente en el servidor (en vez del dispositivo), pero esto trae dos problemas. El primero es que realizar cambios en los algoritmos utilizados sería mucho más complicado, mientras que en el dispositivo alcanzaría con actualizar la aplicación para cambiar los mismos. Y el segundo, es que estas experimentaciones están ejecutando en una PC de escritorio con determinadas características y éstas no tienen por qué ser iguales en un embebido[9] (de hecho, probablemente sean mucho menores). Por lo que aquí planteamos la discusión pero la elección de dónde realizar los cálculos con meta-parámetros está más ligado a los recursos disponibles que a la arquitectura propuesta.

Por otro lado, el último valor a analizar es el tiempo desde que se guarda en el `State` hasta que es utilizado por el simulador. Éste es un valor interesante porque, a diferencia de los dos anteriores, es el más atado a la arquitectura propuesta. El mismo posee mucha varianza y va desde 6 ms a casi 12 ms. Aquí intervienen muchos factores, ya que en cada iteración el hilo principal le pregunta al stateManager si hubo algún cambio, por lo que el impacto de estos dependen exclusivamente del hilo principal y de cuánto tarda cada iteración.

Con respecto a esto último, recordemos que hasta aquí las mediciones realizas en este trabajo no incluyen el costo del llenado del *buffer* de sonido ya que la biblioteca utilizada agrega tiempos de espera para estar sincronizado con `Jack` [3.3.2]. El problema en estas nuevas mediciones es que estamos incluyendo el llenado del *buffer* y la escritura del archivo con tiempos, ya que estamos tomando la diferencia entre el momento en el que se actualiza el `State` y el instante en el que se utiliza el valor en cuestión. Por lo que mientras que antes era suficiente medir cada iteración hasta antes que se llenara el *buffer*, en este caso, desde que se actualiza el `State` hasta que se aplican los cambios, pueden pasar varias iteraciones, obligándonos así a sumar el tiempo total de cada iteración (incluyendo las esperas de `STK` y los tiempos de escritura de las mediciones). Es así que el valor obtenido representa el percibido pero la varianza está muy atada a los tiempos de la biblioteca, lo que dificulta la comparación con los anteriores resultados.

Entre las posibles soluciones se encuentra la remoción de `STK` para las experimentaciones, pero de hacerlo no podríamos escuchar los resultados obtenidos (haciendo más complicado saber si los mismos son correctos).

Finalmente, sabiendo que el tiempo de transporte es el más sobresaliente, vamos a experimentar utilizando una codificación más eficiente entendiendo el tipo de parámetro que manejamos.

4.3.4. Tiempo de transporte con optimización

Como se explicó en la sección previa 4.2.4, vamos a repetir la experimentación anterior utilizando un cambio en la codificación de los mensajes. Recordemos que el objetivo es reducir el tamaño de los mensajes a enviar cuando tenemos listas con valores consecutivos repetidos.

[9] Recordando que el objetivo del proyecto es utilizar un sistema embebido 1.2.

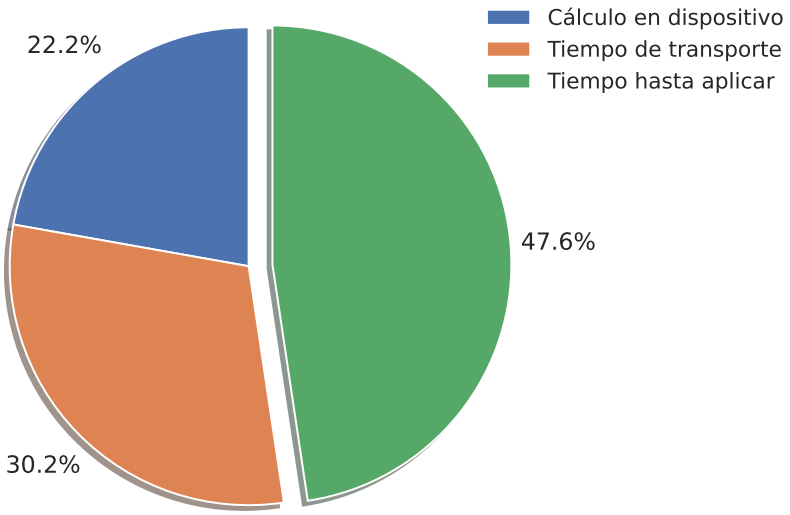


Fig. 4.12: Distribución de los tiempos involucrados al cambiar la fricción en todas las cuerdas. Promedio de más de 9 mil cambios, donde cada cambio en la fricción toma 13 ms promedio (contra ≈ 108 ms de la anterior experimentación [Fig. 4.11]) en aplicarse en el sistema gracias a la optimización en los mensajes

Descripción	Promedio	Desvío estándar
Cálculo en dispositivo	2.52 ms	1.05 ms
Tiempo de transporte	2.31 ms	6.72 ms
Tiempo hasta aplicar	6.09 ms	5.69 ms

Aquí podemos observar una gran mejora en los tiempos de transporte, reduciendo notoriamente los tiempos totales de aplicación de los parámetros. Este resultado es esperable, ya que como se explicó en la sección 2.2.2 el algoritmo utilizado para modificar la fricción realiza modificaciones en las puntas de las cuerdas pero los valores del centro se mantienen iguales. De esta manera, al comprimir los mensajes, se reduce cerca de 30 veces el tamaño del mismo (para el caso de la actualización de la fricción).

Por supuesto cada meta-parámetro genera diferentes tipos de arreglos, con diferentes patrones, pero esta optimización muestra que si buscamos codificaciones que se adapten a estos patrones, podemos reducir muchísimo los tiempos de aplicación. Notemos que algo a tener en cuenta también, es que determinadas codificaciones pueden ser costosas de decodificar para el servidor, por lo que se debe buscar un equilibrio entre una codificación eficiente y una decodificación poco costosa.

4.3.5. Conclusión

En este capítulo continuamos trabajando sobre la arquitectura propuesta permitiendo configurar parámetros de forma eficiente en un sistema real-time. La misma se puede resumir en el siguiente diagrama:

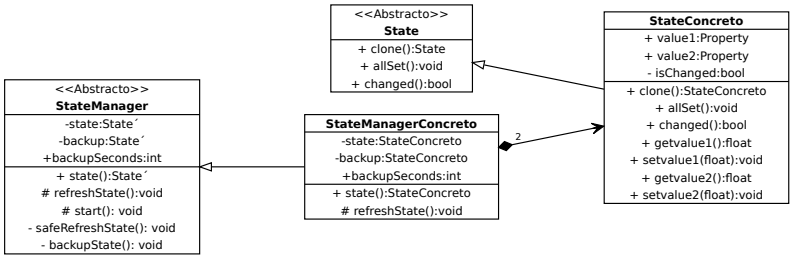


Fig. 4.13: Generalización del diseño del `StateManager` con el `State` presentado en el trabajo. Notar que `State'` refiere a algún tipo que implemente `State`

Vimos como en un sistema *multi-thread*[10] logramos realizar operaciones costosas sin afectar el flujo principal del sistema, a cambio de aceptar una mayor latencia a la hora de aplicar los parámetros. Por supuesto, dicha latencia se encuentra atada a la frecuencia con la que el `State` es consultado por nuevos cambios y a las operaciones realizadas al actualizar cada valor. Este ejemplo lo vimos al realizar un cambio en la fricción de cada cuerda (con 400 nodos) donde llegó tardar menos de 12 ms[11] en aplicarse los cambios, realizando algunas optimizaciones en los mensajes enviados (ver sección 4.2.4).

Ésto muestra que el patrón propuesto no garantiza velocidad a la hora de aplicar los cambios sino que garantiza un bajo impacto en el flujo principal donde corren las operaciones en tiempo real, ya que para garantizar lo primero debemos asegurarnos que el flujo principal consulta por cambios frecuentemente y eso no depende del `StateManager`.

Por otro lado, observamos que el hecho de utilizar estas abstracciones (`StateManager` y `State`) nos permite cambiar totalmente la lógica con la que se actualiza el `State` de forma transparente. Fue así como cambiamos de la lectura de archivos, al servidor *Bluetooth* sin cambio alguno en el flujo principal o simulador.

A su vez, vimos la factibilidad de utilizar dispositivos externos mediante *Bluetooth* en este tipo de sistemas, entendiendo que las mediciones se realizaron en ambientes no controlados (por los que los desvíos con respecto a la media fueron grandes) y sin embargo los resultados obtenidos se ajustan a los esperados en parámetros de ajuste y configuración, permitiendo así a futuros trabajos la experimentación de esta arquitectura utilizando parámetros de toque.

[10] Con múltiples hilos por núcleo de procesador.
[11] Valor promedio, en el peor caso puede llegar a 25 ms.

5. CONCLUSIÓN GENERAL

El objetivo de este trabajo fue elaborar una arquitectura para la parametrización de atributos en un sistema de tiempo real. Ésto lo logramos realizando un abordaje incremental de distintos enfoques que solucionaban problemas específicos. Partiendo desde la identificación de parámetros y su abstracción, hasta la creación de una entidad que los administre de forma independiente (sin afectar el flujo principal).

Este patrón surgió de un objetivo concreto, priorizándose así determinadas características como es el bajo impacto sobre el flujo principal (que debe ser real-time), tanto desde el punto de vista temporal como el funcional (una falla en los cambios propuestos no deberían invalidar al sistema) y la flexibilidad para cambiar de implementación (aquí utilizamos sistema de archivos y *Bluetooth*, pero podría utilizarse cualquier otro tipo, como USB por ejemplo). Lo interesante a observar es que el patrón podría ser adaptado a cada necesidad, agregando funcionalidades nuevas siguiendo la esencia del manejo de parámetros en un hilo aparte.

A su vez, observamos que al compartir un objeto entre diferentes hilos tenemos que tener especial cuidado con las operaciones de lectura y escritura en dicho objeto. Las mismas tienen que ser atómicas, garantizando que en todo momento el objeto se encuentre en estado consistente para su lectura, ya que uno de los principios de la arquitectura propuesta es que una entidad escribe un objeto mientras que otro lo lee, y las operaciones son totalmente asincrónicas (para no afectar el tiempo real) por lo que la lectura no puede realizarse sobre un objeto a medio escribir.

Por su parte, la experimentación mostró sus dificultades a la hora de medir las latencias. Realizamos las mediciones bajo las mismas condiciones, pero al trabajar con un sistema operativo no optimizado, la varianza en los resultados fue notable. Algo similar sucedió con el caso de *Bluetooth*, donde a su vez influyeron factores externos como la distancia entre la PC y el dispositivo, u otros periféricos haciendo interferencia, que resultaron en mediciones con gran varianza respecto a la media. Sin embargo, los resultados mostraron un tiempo de ejecución medio muy por debajo de lo perceptible (entre $\approx 556\,\mu s$ [Fig. 3.13] y $\approx 11\,ms$ [Fig. 4.12] dependiendo de la implementación) y aún en el peor de los casos ($\approx 248\,ms$ [Fig. 4.11]) el flujo no fue interrumpido, evitando cualquier problema que se pudiera causar en el mismo.

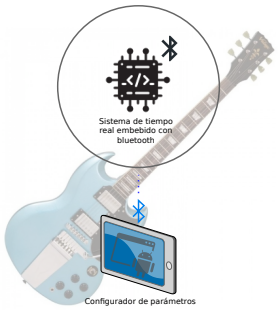


Fig. 5.1: Esquema con el foco en el resultado obtenido en base al objetivo final del proyecto. En este trabajo definimos una arquitectura que le permite al dispositivo embebido conectarse al dispositivo `Android` de forma eficiente, con bajo impacto en el sistema y alta flexibilidad en la configuración de nuevos parámetros

De esta manera, como el presente trabajo se encuadró dentro de un proyecto Fig. 1.1 de mayor alcance, los resultados obtenidos logran alcanzar los objetivos propuestos que incluyen darle al usuario una experiencia más enriquecedora a la hora de experimentar con el instrumento, abriendo así las puertas a innumerables configuraciones y sonidos distintos para un mismo instrumento desde la palma de la mano Fig. 5.1.

6. TRABAJOS FUTUROS

Como trabajo futuro se propone utilizar esta arquitectura para parámetros que requieren tiempo real duro (como la posición de los dedos en la cuerda), donde el tiempo desde que se realiza un cambio hasta que se aplica tiene que ser menor a 20 ms. Para esto se deberán evaluar alternativas y optimizaciones en la utilización de *Bluetooth* dada su gran varianza en los tiempos de comunicación. Aquí los mensajes entre el dispositivo externo y el sistema juegan un rol fundamental. Distintas codificaciones pueden impactar enormemente en los tiempos ya que, como se observó con la fricción Fig.4.12, hay parámetros que podrían seguir determinados patrones y una buena codificación resultaría en la reducción sustancial del tamaño del mensaje.

A su vez, en este trabajo observamos cómo el *StateManager* interactúa siempre con el *State* en base a un estímulo externo (ya sea desde archivo o un dispositivo *Android*), sin embargo se pueden dar situaciones donde la interacción con un parámetro se realice desde distintos puntos. Un ejemplo es la palanca, que podría ser manejada desde un dispositivo *MIDI* externo y, a la vez, ser usada desde una aplicación *Android*. Esto requeriría la definición de prioridades ante la concurrencia de distintos dispositivos sobre un mismo parámetro.

Por su parte, los resultados y herramientas generadas en este trabajo abren la puerta al desarrollo de algoritmos que mediante la variación de parámetros, puedan automatizar la búsqueda de sonidos específicos. Generando nuevas clasificaciones en los resultados sonoros en base a los parámetros utilizados.

7. ANEXO

7.1. Lenguaje Unificado para Modelado (UML)

Muchos de los gráficos utilizados a lo largo del trabajo están basados en diagramas estructurales (clases) descriptos en `UML 2.5.1` [32].

7.1.1. Diagramas de Clases

Los diagramas de clases permiten describir la estructura de un sistema mostrando las clases del mismo, sus variables, operaciones y relaciones. Aquí las clases son representadas con rectángulos:

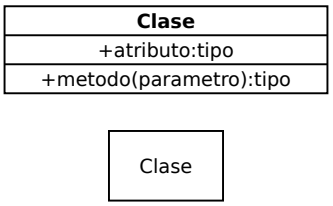


Fig. 7.1: Clases en UML 2

Notar que estos pueden contener o no atributos u operaciones con distintos modificadores de visibilidad:

+	publico	Se puede acceder desde otra clase
-	privado	Solo se puede acceder desde la misma clase
#	protegido	Solo las clases que heredan pueden acceder

y a su vez, puede contener la notación «*Abstracto*» cuando la clase es abstracta (no se puede instanciar).

Por otro lado algunas de las interacciones que se pueden dar son:

1. Herencia
2. Agregación
3. Composición

que son las que más utilizaremos en este trabajo.

Herencia

Representa una relación de "es un". Las subclases poseen el comportamiento de la clase padre.

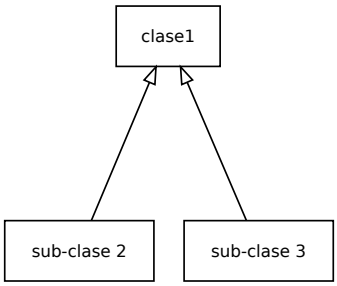


Fig. 7.2: Relación de herencia en Diagrama de Clases

En `C++` se utilizan los dos puntos (":") para indicar herencia.

Va a ser muy común encontrarse en este trabajo con que siempre hablamos de la clase padre para no referirnos a implementaciones especificas, es decir, que cuando hablamos de *clase* 1 nos referimos también a $sub - clase$ 2 y 3 siguiendo el ejemplo de la figura. 7.2.

Agregación

Muestra una relación de "es parte de".

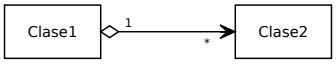


Fig. 7.3: Relación de Agregación en Diagrama de Clases

En `C++` la *clase* 1 posee un atributo de tipo *clase* 2, siguiendo el ejemplo de la figura7.3.

Composición

Como la agregación, es una relación de "es parte de" pero la diferencia es que el ciclo de vida de una clase depende de la otra, es decir, cuando un objeto muere (*clase* 1) el otro muere con el (*clase* 2).

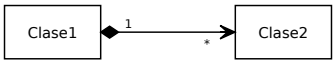


Fig. 7.4: Relación de Composición en Diagrama de Clases

En `C++` la implementación es igual a la agregación pero el destructor de la *clase* 1 también destruye la *clase* 2, siguiendo el ejemplo de la Fig. 7.4.

Un abuso de notación que usaremos es que cuando no aclaramos el multiplicador en la relación significa que es uno.

7.2. Características de hardware utilizado

Para las experimentaciones se utilizaron los siguientes componentes:

- **PC**: Procesador AMD Ryzen 5 2400G (3.6 GHz), 16 GB memoria RAM, ASUS GeForce GTX 1050 2 GB GDDR5, Conector *Bluetooth* (4.0) por USB
- **Tablet**: Amazon Fire HD 8, Procesador Quad-Core Cortex (1.3 GHz) A-53, 1.5 GB Memoria RAM, Multi-touch, Resolución 1280px x 800px

7.3. Agregar nuevo parámetro

Aquí describiremos cómo agregar nuevos parámetros en la implementación que utiliza `Android`. Los pasos fundamentales son:

- Agregar una nueva variable en el `State`
- Implementar el copiado del elemento en la operación *clone* del `State`
- Asignarte un identificador numérico único al nuevo parámetro y agregarlo en el servidor y cliente bluetooh.
- Implementar la asignación del valor en el servidor *Bluetooth*
- Agregar el nuevo parámetro en el estado del dispositivo
- Crear el componente en la UI de `Android`
- Asociar el componente creado con el valor correspondiente en el estado del dispositivo

veamos estos pasos con un ejemplo para la variable "palanca" de tipo `float`:

7.3.1. Servidor

```
...
class ParameterState : public State {
  public:
    PROPERTY(float, palanca);
    ...
}
...
```

Listing 7.1: lib/headers/parameterState.h - Define la nueva variable en el `State`.

```
...
void ParameterState::allSet() {
  setChangedpalanca();
    ...
}
...
ParameterState* ParameterState::clone() const {
    ...
    COPY(palanca);
    ...
}
```

Listing 7.2: `lib/parameterState.cpp` - Implementa la copia de la nueva variable dentro de `clone` y se agrega el flag en `allSet` para que cuando se llama a `allSet` se marque la variable como leída.

```
...
void BTManager::updateValue(int key, float value, std::chrono::steady_clock
    ::time_point startRead) {
    ...
    CASE(6, float, palanca, value, startRead)
    ...
}
...
```

Listing 7.3: `lib/btManager.cpp` - Como la `palanca` es un `float`, se agrega el condicional (`CASE`) dentro de `updateValue`. El `6` corresponde al valor numérico asignado para la codificación de los mensajes *Bluetooth*.

Luego para poder utilizar el nuevo parámetro en el hilo principal podemos hacer:

```
...
void iim2(StateManager<ParameterState>* parameterManager) {
    ...
    if (parameterManager -> state() -> hasChangedpalanca()) {
        palanca = parameterManager -> state() -> getpalanca();
        parameterManager -> state() -> setChangedpalanca();
    }
    ...
}
```

Listing 7.4: `iim2.cu` - Define la nueva variable en el `State`.

7.3.2. Cliente

```
public class State implements Serializable {
    ...
    public Float palanca = 1f;
    ...
}
```

Listing 7.5: `State.java` - Define la variable en el `State` del cliente.

```
...
<SeekBar
    android:id="@+id/palancaBar"
    android:layout_width="165dp"
    android:layout_height="42dp"
    android:layout_marginRight="10dp"
    android:max="100"
    android:progress="0"
/>
...
```

Listing 7.6: `fragment_propiedades.xml` - Define la interfaz de usuario (UI), en este caso usamos un `SeekBar` pero podría variar según el parámetro.

```
public class PropiedadesFragment extends Fragment {
...
    private SeekBar palancaBar;
    ...
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        ...
        palancaBar = (SeekBar) getView().findViewById(R.id.palancaBar);
```

```
8           palancaBar.setOnSeekBarChangeListener(new SeekBar.
    OnSeekBarChangeListener() {
9               @Override
10              public void onProgressChanged(SeekBar seekBar, int progress,
    boolean fromUser) {
11                  if (fromUser) {
12                      stateUtil.state.palanca = (float) progress;
13                      stateUtil.sendValueByBluetooth("palanca", stateUtil.
    state.palanca);
14                  }
15              }
16
17              @Override public void onStartTrackingTouch(SeekBar seekBar) {}
18
19              @Override
20              public void onStopTrackingTouch(SeekBar seekBar) {
21                  seekBar.setProgress(1);
22                  stateUtil.state.palanca = 1f;
23                  stateUtil.sendValueByBluetooth("palanca", stateUtil.state.
    palanca);
24              }
25          });
26          ...
27      }
28  }
```

Listing 7.7: `PropiedadesFragment.java` - Conecta el componente de la UI con la lógica. Aquí en cada cambio se llama a `sendValueByBluetooth` y cuando se suelta vuelve a 1 (`setProgress(1)`)

```
1  public class StateFormater {
2      ...
3      static {
4          ...
5          numberAssignator.put("palanca", "6");
6          ...
7      }
8      ...
9      public static String compressAll(State state) {
10         ...
11         generateValue("palanca", state.palanca) +
12         ...
13     }
14     ...
15     public static Spanned prettyPrint(State state) {
16         ...
17         printValue("palanca", state.palanca) + "\n" +
18         ...
19     }
20     ...
21
22 }
```

Listing 7.8: `StateFormater.java` - Define como se escribe el parámetro en la UI y en los mensajes *Bluetooth*.

En caso de ser un meta-parámetro también hay que agregar el algoritmo en: `MetaAlgorithms.java` y en `StateUtil.java` hay que modificar `sendValueByBluetooth`.

Bibliografía

[1] J. R. Alves, The History of the Guitar: Its Origins and Evolution, Marshall Digital Scholar, 2015.

[2] S. Howell, The lost art of sampling: Part 1 (2005).
URL https://www.soundonsound.com/techniques/lost-art-sampling-part-1

[3] L. Hiller, P. Ruiz, Synthesizing musical sounds by solving the wave equation for vibrating objects: Part 1, J Audio Eng Soc 19 (6) (1971) 462–470.
URL http://www.aes.org/e-lib/browse.cfm?elib=2164

[4] K. Karplus, A. Strong, Digital synthesis of plucked-string and drum timbres, Comput Music J 7 (2) (1983) 43–55. doi:10.2307/3680062.

[5] J. O. Smith, Physical modelling using digital waveguides, Comput Music J 16 (4) (1992) 74–91. doi:10.2307/3680470.

[6] M. Karjalainen, V. Välimäki, T. Tolonen, Plucked-string models: From the karplus-strong algorithm to digital waveguides and beyond, Computer Music Journal 22 (1998) 17–32. doi:10.2307/3681155.

[7] S. Bilbao, C. Desvages, M. Ducceschi, B. Hamilton, R. Harrison-Harsley, A. Torin, C. Webb, Physical modeling, algorithms, and sound synthesis: The ness project, Comput Music J 43 (2-3) (2020) 15–30. doi:10.1162/comj_a_00516.

[8] P. A. Laplante, Real-Time Systems Design and Analysis, John Wiley & Sons, Ltd, 2004. doi:10.1002/0471648299.

[9] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, 2nd Edition, Springer, Boston, MA, 2011. doi:10.1007/978-1-4419-8237-7.

[10] J. Davis, Y.-H. Hsieh, H.-C. Lee, Humans perceive flicker artifacts at 500 hz, Sci Rep 5 (1) (2015) 7861. doi:10.1038/srep07861.

[11] M. Czeiszperger, J. Rothstein, Midi: A comprehensive introduction, Comput Music J 17 (1992) 75.

[12] M. M. Chris Cannam, T. Felix, Rosegarden (1993).
URL https://www.rosegardenmusic.com

[13] R. H. Walden, Analog-to-digital converter survey and analysis, IEEE Journal on Selected Areas in Communications 17 (4) (1999) 539–550. doi:10.1109/49.761034.

[14] C. E. Shannon, Communication in the presence of noise, Proceedings of the Institute of Radio Engineers (IRE) 37 (1) (1949) 10–21. doi:10.1109/jrproc.1949.232969.

[15] W. Kester, J. Bryant, Data Conversion Handbook, Newnes, Burlington, 2005. doi:https://doi.org/10.1016/B978-075067841-4/50016-6.

[16] P. R. Cook, G. P. Scavone, The synthesis toolkit (stk), International Computer Music Association, 1999, pp. 164–166.

[17] S. L. Paul Davis, Jack audio connection kit (2016).
URL https://jackaudio.org

[18] Advanced linux sound architecture (2019).
URL https://www.alsa-project.org/wiki/Main_Page

[19] K. N. Perrott, David R & Williams, Auditory temporal resolution: Gap detection as a function of interpulse frequency disparity, Psychon Sci 25 (1971) 73–74. doi:10.3758/BF03335851.

[20] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, Cg: A system for programming graphics hardware in a c-like language, ACM Trans. Graph. 22 (3) (2003) 896–907. doi:10.1145/882262.882362.

[21] N. Wilt, The CUDA Handbook: A comprehensive guide to GPU Programming, Pearson Education, 2013.

[22] Nvidia, Cuda programming guide, Tech. rep., Nvidia (2015).

[23] M. Lindner, Libconfig (2018).
URL https://hyperrealm.github.io/libconfig/

[24] M. Ernst, G. J. Badros, D. Notkin, An empirical analysis of c preprocessor use, IEEE T Software Eng 28 (2002) 1146–1170. doi:10.1109/TSE.2002.1158288.

[25] Statcounter, Os market share in argentina (2020).
URL https://gs.statcounter.com/os-market-share/all/argentina

[26] Google, Android fragments.
URL https://developer.android.com/guide/components/fragments

[27] Google, Android spinner.
URL https://developer.android.com/guide/topics/ui/controls/spinner

[28] Google, Android checkbox.
URL https://developer.android.com/reference/android/widget/CheckBox

[29] J. C. Haartsen, The bluetooth radio system, IEEE Pers Commun 7 (1) (2000) 28–36. doi:10.1109/98.824570.

[30] Google, Android bluetooth client.
URL https://developer.android.com/guide/topics/connectivity/bluetooth.html#ConnectAsAClient

[31] A. S. Huang, L. Rudolph, Bluetooth Essentials for Programmers, Cambridge University Press, 2007. doi:10.1017/CBO9780511546976.

[32] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, 2nd Edition, Addison-Wesley Object Technology Series, Addison-Wesley Professional, 2005.