P. A. Kurta
Bonch-Bruevich Saint-Petersburg State University of Telecommunications

INTERACTION OF THE USER WITH THE INFORMATION SYSTEM.
PART 3. EFFICIENCY EVALUATION

The article is the third in a series devoted to the issues of user interaction with the information system. The problem of calculating and evaluating the effectiveness of interactions is considered. For this, it is proposed to classify the group of compared information services (implementation of the interaction interface and information system) to a certain class for which the interaction model has been built. Then, in terms of the model, it is possible to calculate performance indicators – potency, as a guarantee of the user getting the result; operativeness, like the speed of entering and receiving data; resource preservation, as the moderation of the cognitive load on a person during work. The resulting metric values can be directly used to compare interactions. The article identifies 3 classes of common information services: request, control and gaming. For the first of them, as simple enough for research, an analytical model of interaction is built, and 3 implementation schemes are proposed, which differ in the density of filling the interface forms with input/output fields. Formulas for calculating indicators are displayed. An experiment is carried out to calculate the indicators of the effectiveness of interactions in each of the schemes for a different number of interface elements. The values of the indicators, as well as the root-mean-square efficiency for them and their pairs, are given in tabular and graphical form; also, the ranking of the efficiency of schemes by Pareto is considered. The analysis of the results obtained is carried out and conclusions are drawn that confirm the validity of the formulas obtained. Directions for further research are indicated.

Information system, interface, efficiency, indicators, model

УДК 004.42

Е. Е. Кошелев, С. В. Букунов
Санкт-Петербургский государственный архитектурно-строительный университет

Плагин-синтезатор для цифровой рабочей станции

Приведено описание аудиосинтезатора с собственным графическим интерфейсом пользователя. В плагине реализованы два осциллятора, ADSR-огибающая, фильтр нижних частот, фильтр верхних частот и эффект низкочастотной модуляции. Используемые осцилляторы генерируют волны четырех типов: синусоидальную, квадратную, треугольную и пилообразную. Реализована возможность смешения волн двух осцилляторов в любой пропорции. Для реализации каждой составляющей плагина были разработаны необходимые алгоритмы. Для удобства работы с плагином был создан графический пользовательский интерфейс в виде фортепианной клавиатуры, клавиши которой срабатывают по нажатию на левую кнопку мыши. Интерфейс содержит ручки и кнопки для изменения параметров обработки сигнала, а также ручку для регулировки громкости выходного сигнала. Разработанный плагин имеет формат VSTi и может использоваться практически в любой современной цифровой звуковой рабочей станции. Плагин реализован на языке C++ с использованием методологии объектно-ориентированного программирования. Для создания плагина использовалась среда разработки Microsoft Visual Studio и библиотека WDL-OL. В качестве цифровой звуковой рабочей станции для загрузки готового плагина использовалась цифровая аудиостанция Reaper.

Цифровая обработка звука, плагин-синтезатор, цифровая рабочая станция, объектно-ориентированное программирование, графический интерфейс пользователя

В последнее время для обработки и генерации звука стали широко применяться компьютеры, постепенно вытесняя используемые в этих целях физические устройства (микшеры, процессоры эффектов и др.) [1], [2]. Именно компьютерная обработка звука лежит в основе работы современных цифровых звуковых рабочих станций, которые представляют собой специализированное программное обеспечение, предназначенное для записи, хранения, обработки и воспроизведения

цифрового звука [3]. В основе работы любой цифровой рабочей станции лежит возможность подключения специальных модулей обработки звука – плагинов. Плагин представляет собой независимо компилируемый программный модуль, подключаемый к основной программе для расширения ее возможностей [4]. Плагины, предназначенные для работы со звуком, принято называть аудиоплагинами.

Появление плагинов позволило многочисленным студиям звукозаписи существенно удешевить процесс обработки звука, заменив многочисленные дорогостоящие, часто выходящие из строя и занимающие много места физические устройства на их дешевые, надежные и компактные программные аналоги.

Все аудиоплагины делятся на три основные группы: плагины-эффекты [5], [6], плагины-анализаторы [7] и плагины-инструменты [8], [9]. Каждая категория плагинов выполняет свои определенные функции.

Синтезатор – это электронный музыкальный инструмент, способный генерировать аудиосигналы, которые могут быть преобразованы в звук. Они могут имитировать традиционные музыкальные инструменты – фортепиано, гитару и др., а также создавать собственные тембры в зависимости от встроенных в них алгоритмов генерации сигналов.

Плагины-синтезаторы – это программные модули, подключаемые к цифровой звуковой рабочей станции для создания различных звуковых сигналов. Плагины-синтезаторы работают с набором MIDI-команд [8], [9]. Технология MIDI (*англ*. Musical Instrument Digital Interface – цифровой интерфейс музыкальных инструментов) позволяет передавать посредством синтезатора или MIDI-клавиатуры определенные команды, содержащие информацию о нажатой клавише, длительности ее нажатия, скорости и других параметрах.

Главная задача любого синтезатора – это генерация звуковых волн с помощью осциллятора. Современный синтезатор может иметь несколько осцилляторов, каждый из которых имеет свой параметр громкости, что позволяет смешивать их друг с другом в любой пропорции [10]. Осцилляторы могут генерировать волны различных форм.

Вторым по значимости элементом синтезатора является огибающая функция, также именуемая ADSR-огибающей (от *англ*. Attack-Decay-Sustain-Release – четыре разные стадии огибающей).

И, наконец, синтезированный плагином сигнал может быть подвергнут обработке для создания того или иного звукового эффекта. В [6] авторами был разработан аудиоплагин-эффект и описаны основные заложенные в него алгоритмы.

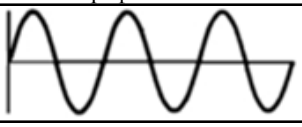
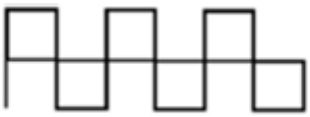
Цель разработки. Разработкой аудиоплагинов и, в частности, плагинов-синтезаторов занимается большое число компаний по всему миру. Однако отечественные разработки в данной сфере практически не ведутся, что представляет большую проблему, поскольку практически любая область, связанная со звуком, а в особенности творческая (музыка, кинематограф, видеоигры и т. д.), не обходится без обработки звука с помощью зарубежных программных комплексов. Создание отечественных программных продуктов могло бы существенно снизить материальные затраты на их приобретение в сравнении с затратами на зарубежные аналоги. Помимо этого, сфера обработки и синтеза звука имеет немалую творческую составляющую, что может поспособствовать появлению новых эффектов обработки или звуковых тембров при интерпретации алгоритмов зарубежных разработчиков. Поэтому исследование области обработки и генерации цифровых звуковых сигналов, изучение способов разработки программного обеспечения для работы со звуком и реализация полученных знаний в виде конечных программных продуктов несомненно являются важнейшими задачами при проектировании программных комплексов для работы с цифровым звуком.

Цель данной статьи заключается в создании собственного плагина-синтезатора с различными эффектами обработки звукового сигнала, алгоритмы и программная реализация которых представлены в [6].

Основные элементы синтезатора. При разработке плагина-синтезатора необходимо определить форму звуковых сигналов, которые будут генерироваться осциллятором, параметры ADSR-огибающей и звуковые эффекты, с помощью которых можно будет обрабатывать синтезированный сигнал.

Осциллятор. В разработанном плагине-синтезаторе для генерации осциллятором звуковых волн использовались волны четырех различных форм – синусоидальной, пилообразной, квадратной и треугольной.

Синусоидальную волну часто называют «простой», потому что из таких волн складываются более сложные виды колебаний (в частности, фи-

Название формы	График волны	Пример формулы
Синусоидальная		$y = \sin(x)$
Квадратная		$y = \sin(x) + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5}$
Треугольная		$y = \sin(x) + \frac{\sin(3x)}{3^2} + \frac{\sin(5x)}{5^2}$
Пилообразная		$y = \sin(x) + \frac{\sin(2x)}{2} + \frac{\sin(3x)}{3}$

зиологическая теория слуха Гельмгольца и математическая теорема Фурье рассматривают любое звуковое колебание как результат сложения синусоидальных колебаний). Форма волны описывается, например, обычной синусоидальной функцией, а сами такие колебания также называют гармоническими.

Сигнал, имеющий волну квадратной формы, содержит нечетные гармоники, причем амплитуды этих гармоник обратно пропорциональны их частоте (относительно основного тона).

Подобно квадратной, треугольная волна содержит в своем спектре нечетные гармоники, но уменьшение амплитуды наиболее разительно. Амплитуда колебания в таком случае обратно пропорциональна квадрату данной гармоники.

Пилообразная форма волны в отличие от квадратной и треугольной имеет в своем спектре все гармоники основного сигнала, а амплитуда также обратно пропорциональна их частотам (относительно основного тона).

Основные характеристики волн этих четырех типов представлены в таблице.

Волна каждой из этих форм имеет свое характерное звучание. Одновременное смешение нескольких волн разных форм позволяет получить новый тембр звука.

ADSR-огибающая. Данный параметр позволяет работать с динамикой синтезированного звука. На любом музыкальном инструменте громкость извлеченного звука изменяется с течением времени. Например, на гитаре самый громкий звук будет наблюдаться в момент удара по струне, после чего звучание будет плавно затухать. На органе же громкость звучания клавиши остается неизменной все время, пока нажата клавиша.

Именно для создания подобных динамических эффектов и предназначена ADSR-огибающая, схематично представленная на рис. 1.

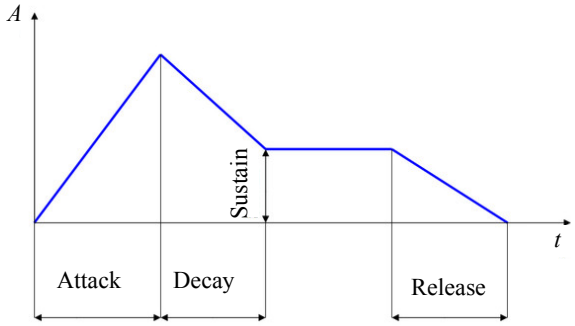


Рис. 1

По оси абсцисс на рисунке отложено время, по оси ординат – амплитуда сигнала. Аббревиатура ADSR обозначает параметры этой функции, и расшифровывается как Attack (Атака), Decay (Спад), Sustain (Поддержка) и Release (Затухание).

«Атака» – описывает время, за которое сигнал достигает своего максимального уровня громкости. Как только это время истекает, начинается стадия «Спад».

«Спад» – описывает время, за которое сигнал переходит от своего максимального уровня до уровня громкости, описываемого параметром «Поддержка».

«Поддержка» – данный параметр описывает уровень громкости, который будет иметь сигнал после прохождения стадий «Атаки» и «Спада», при условии, что клавиша синтезатора нажата. Следует отметить, что данный параметр иногда ошибочно рассматривается как временной параметр, который определяет время звучания опреде-

ленного сигнала. Однако в контексте огибающей «Поддержка» – это именно значение уровня громкости.

«Затухание» – параметр, описывающий время, за которое сигнал затухает, когда клавиша синтезатора отпускается.

За исключением параметра «Поддержка», который лишь обозначает уровень громкости, остальные параметры могут быть нелинейными, что позволяет создавать более разнообразные динамические эффекты.

Эффекты обработки сигнала. Для создания различных эффектов обработки синтезируемого звукового сигнала в компьютерной обработке звука используются амплитудная и частотная обработка сигнала, а также изменение его временных характеристик. Математическое описание используемых в данной статье эффектов и особенности их программной реализации представлены в работе [6].

Используемые технологии. Для разработки плагина была выбрана библиотека WDL-OL. Данная библиотека позволяет создавать кроссплатформенные плагины – т. е. плагины различных форматов (VST, AU, AXX). Библиотека содержит большое количество классов и функций, необходимых для создания аудиоплагинов. К ее достоинствам можно также отнести наличие разрешительной лицензии, позволяющей использовать данную библиотеку при разработке собственных плагинов даже в коммерческих целях. Кроме того, библиотека предоставляет средства для создания собственного интерфейса пользователя. Характерную особенность библиотеки представляет необходимость сборки интерфейса непосредственно в коде, т. е. все необходимые графические ресурсы нужно самостоятельно прописывать в файле ресурсов (rc-файле) проекта. С одной стороны, это может считаться недостатком, но, с другой стороны, позволяет разработчику создавать свои собственные графические изображения элементов пользовательского интерфейса (ручки регулировки, переключатели и т. д.) в виде соответствующих bmp-файлов.

Для создания плагина использовалась среда разработки программного обеспечения Microsoft Visual Studio.

В качестве языка программирования был выбран язык C++, поскольку именно этот высокопроизводительный язык используется во всех основных фреймворках, предназначенных для решения задач подобного рода [4], [8].

В качестве формата плагина был выбран формат VST (Visual Studio Technology) – самый популярный и широко используемый формат аудиоплагинов. Кроме того, данный формат работает на всех основных операционных системах, в том числе Windows.

В качестве цифровой звуковой рабочей станции, в которую будет загружаться готовый плагин, использовалась станция Reaper. Эта бесплатная станция имеет портативную версию и часто применяется для работы с аудиоматериалом.

Программная реализация. В операционной системе Windows аудиоплагины в формате VST представлены в виде динамически подключаемых библиотек и имеют расширение .dll. Поэтому для работы в Visual Studio с библиотекой WDL-OL необходимо подключить ее к рабочему проекту, указав в свойствах проекта путь к файлам этой библиотеки. Саму библиотеку можно скачать на официальной странице разработчика (например, на портале GitHub).

При разработке плагина применялся объектно-ориентированный подход, когда плагин представляет собой совокупность классов, написанных на языке C++. Диаграмма классов плагина представлена на рис. 2. Для отображения диаграммы классов использовался графический язык

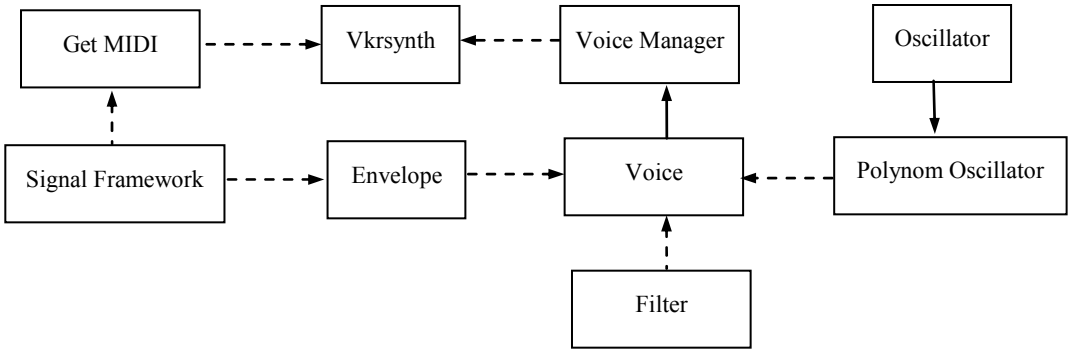


Рис. 2

```
#include "IPlug_include_in_plug_hdr.h"

class vkrSynth : public IPlug
{
public:
    vkrSynth(IPlugInstanceInfo instanceInfo);
    ~vkrSynth();

    void Reset();
    void OnParamChange(int paramIdx);
    void ProcessDoubleReplacing(double** inputs, double** outputs, int nFrames);
};
```

Рис. 3

UML. В соответствии с терминологией этого языка сплошные стрелки на диаграмме отображают механизм наследования, а штриховые – механизм включения.

Основной класс плагина – vkrSynth, наследник класса IPlug библиотеки WDL-OL. Данный класс содержит следующие методы:

– конструктор класса;

– деструктор класса;

– метод Reset(), который сбрасывает предустановленные значения параметров плагина;

– метод OnParamChange(), который будет вызываться при изменении параметров плагина (ручек, фейдеров, переключателей);

– метод ProcessDoubleReplacing(), который будет обрабатывать входящий сигнал.

На рис. 3 представлен программный код класса плагина.

Реализация синтеза сигнала. Для программной реализации синтеза сигналов был создан соответствующий класс Oscillator. Для работы с разными типами волн в этом классе было создано перечисление Type, код которого представлен на рис. 4.

```
class Oscillator
{
public:
    enum Type
    {
        Sine,
        Saw,
        Square,
        Triangle,
        pCount
    };
```

Рис. 4

В данном перечислении переменная Sine соответствует синусоидальной волне, переменная Saw – пилообразной волне, переменная Square – квадратной волне и переменная Triangle – треугольной волне. Последняя переменная pCount хранит количество типов волн.

Кроме того, для реализации алгоритма синтеза сигналов были использованы еще несколько переменных и констант. В частности, для хранения фазы была создана переменная Phase. Значение этой переменной изменяется в каждом семпле и показывает, в какой фазе цикла находится осциллятор при синтезировании формы волны [11].

Для того чтобы знать, насколько изменять фазу, была введена переменная PhaseRaise, значение которой будет зависеть от частоты ноты, которую необходимо получить. Для хранения частоты также была создана переменная Frequency.

Для работы с числом π создаются константные переменные PI и twoPI, участвующие в алгоритме.

Генерация волн реализована в функции synthesize() типа void. В качестве аргументов она принимает количество семплов nFrames и переменную, обозначающую значение выходящего сигнала (вначале оно будет равным нулю). В эту переменную в дальнейшем записывается результат синтезирования. Для реализации переключения типа волны была использована конструкция Switch-Case.

Программная реализация синтеза синусоидальной волны представлена на рис. 5.

```
case Sine:
    for (int i = 0; i < nFrames; i++)
    {
        out[i] = sin(Phase);
        Phase += PhaseRaise;
    }
```

Рис. 5

На первом шаге значение out[i] равно нулю. При первом заходе в цикл значение первого семпла будет равно синусу от переменной Phase, обозначающей фазу, которая в начальный момент равна нулю [1], после чего значение фазы увеличивается на значение переменной PhaseRaise.

Программный код синтеза пилообразной волны представлен на рис. 6.

```
case Saw:
    for (int i = 0; i < nFrames; i++)
    {
        out[i] = 1 - (2 * Phase / twoPI);
        Phase += PhaseRaise;
        while (Phase >= twoPI)
        {
            Phase -= twoPI;
        }
    }
    break;
```

Рис. 6

В начальный момент времени фаза равна нулю, поэтому выходящий семпл будет равен единице. Значение фазы Phase возрастает с каждым семплом, а значит, значение выражения 2.0 * Phase / twoPI возрастает с 0 до 2, пока значение фазы не достигнет значения 2π. До этого момента значение выходящего семпла меняется от 1 до –1, создавая нисходящую пилообразную волну.

Программный код синтеза квадратной волны представлен на рис. 7.

```
case Square:
    for (int i = 0; i < nFrames; i++)
    {
        if (Phase <= PI)
        {
            out[i] = 1;
        }
        else
        {
            out[i] = -1;
        }
        Phase += PhaseRaise;
        while (Phase >= twoPI)
        {
            Phase -= twoPI;
        }
    }
    break;
```

Рис. 7

Знакомая нижняя часть кода говорит о том, что цикл также имеет длину 2π. Условный оператор разбивает длину цикла на две части. Пока значение фазы Phase меньше или равно π, значение выходящего семпла равно 1. Как только значение фазы превышает π, значение выходящего семпла становится равно –1, отчего форма волны приобретает резкий скачок, и в результате формируется квадратная волна.

Программный код синтеза треугольной волны представлен на рис. 8.

Цикл также имеет длину 2π. Для расчета выходящего значения сначала берется абсолютное значение (fabs) синтеза восходящей пилы, из которого вычитается 0.5, чтобы выровнять форму волны относительно нуля, иначе волна будет представлена только положительными значениями и эффект будет совершенно другим. Затем полученное выражение умножается на 2, чтобы получить волну в нужных пределах от 1 до –1. Таким образом генерируется треугольная волна.

Реализация ADSR-огибающей. Параметры огибающей: Attack (Атака), Decay (Спад), Sustain (Поддержка) и Release (Затухание) [12].

По своей сути огибающая работает с амплитудой сигнала во времени, т. е. каждый параметр либо изменяет амплитуду за определенный промежуток времени, либо поддерживает эту амплитуду на каком-то временном интервале [8].

Параметры огибающей точнее называть ее стадиями, а сам звуковой сигнал будет переходить из одной стадии в другую, находясь в каждый момент времени только в одной стадии. В теории алгоритмов такая математическая абстракция называется конечным автоматом [7]. Помимо четырех вышеперечисленных стадий необходимо наличие пятой стадии, отвечающей за то состояние, когда сигнал не находится ни в одной из этих четырех стадий. Такая ситуация возникает, когда синтезатор еще не начал генерировать сигнал или когда он закончил генерацию сигнала, т. е. тогда, когда клавиши либо еще не нажимались, либо перестали нажиматься, и стадия Release (последняя стадия) завершилась. Такое состояние обычно называют Off (выключено).

```
case Triangle:
    for (int i = 0; i < nFrames; i++)
    {
        out[i] = 2 * (fabs(-1 + (2 * Phase / twoPI)) - 0.5);
        Phase += PhaseRaise;
        while (Phase >= twoPI)
        {
            Phase -= twoPI;
        }
    }
    break;
```

Рис. 8

Огибающая будет принимать значения типа double от 0 до 1. Каждый семпл синтезированного сигнала будет умножаться на значение огибающей. На стадиях Attack, Decay и Release значение огибающей будет меняться в течение определенного времени, поскольку это временные параметры. За временной интервал как раз и отвечают ручки плагина, позволяющие менять интервал от минимального к максимальному, определенному разработчиком плагина.

Например, когда время параметра Attack минимально, то звуковой сигнал слышен сразу со всей его громкостью. Если увеличить время атаки, то первый семпл сигнала будет умножаться на нулевое значение огибающей. Затем с течением времени значение огибающей будет расти, и будет слышно плавное нарастание звука. Значение огибающей нарастает от 0 до 1 за то время, которое определено пользователем синтезатора с помощью ручки плагина. Как только это время закончилось, сигнал переходит в следующую стадию – Decay.

Состояния Sustain и Off могут длиться неограниченное время, пока не будет вызвана функция смены состояния – например, отпусканием или нажатием клавиши [5].

Для программной реализации был создан класс огибающей, названный Envelope. В этом классе создано перечисление стадий, в которых будет находиться звуковой сигнал. Код перечисления представлен на рис. 9. Последний элемент перечисления StageCount хранит в себе количество стадий.

```
class Envelope
{
public:
    enum Stages
    {
        Off,
        Attack,
        Decay,
        Sustain,
        Release,
        StageCount
    };
}
```

Рис. 9

Далее была создана переменная MinVolume типа double, отвечающая за минимальный уровень сигнала. Ее нужно сразу инициализировать значением, близким к нулю, например 0/0001, так как с абсолютным нулем некоторые вычисления работать не будут.

Необходимо наличие переменной типа Stage, отвечающей за то, в каком состоянии находится звуковой сигнал. Изначально данная переменная должна иметь состояние Off. С помощью переменной SampleNow огибающая будет определять, в каком моменте времени она находится. Переменная VolumeNow типа double будет отвечать за то, на какой громкости находится конкретный звуковой семпл.

Переход из одной стадии в другую будет происходить по достижении определенного времени (не на всех стадиях). Так как время измеряется в количестве семплов, то для хранения количества семплов была создана переменная SampleChangeStage типа double. Для определения времени работы каждой стадии был создан массив StageTime[StageCount] типа double из пяти элементов, каждый из которых хранит значение времени работы своей стадии. Значение, на которое будет умножаться текущий уровень громкости VolumeNow, будет называться Level и иметь тип double.

Необходимо сразу проинициализировать созданные переменные. Код инициализации представлен на рис. 10.

```
Envelope() :
MinVolume(0.0001),
StageNow(Off),
VolumeNow(MinVolume),
Level(1.0),
SampleNow(0),
SampleChangeStage(0)
{
    StageTime[Off] = 0.0;
    StageTime[Attack] = 0.01;
    StageTime[Decay] = 0.5;
    StageTime[Sustain] = 0.1;
    StageTime[Release] = 1.0;
};
```

Рис. 10

Результат инициализации:

– параметр Off имеет нулевое значение амплитуды сигнала;

– параметр Attack длится сотую долю секунды;

– параметр Decay длится полсекунды;

– параметр Sustain обозначает уровень сигнала, в данном случае – одна десятая от общего уровня, следовательно, после перехода из состояния Decay сигнал станет заметно тише;

– параметр Release длится одну секунду.

Чтобы получить время работы стадии в семплах, нужно перевести значение работы стадии в секунды, а затем умножить его на частоту дискретизации проекта [2].

```
double Envelope::CountSample()
{
    if (StageNow != Off &&
        StageNow != Sustain)
    {
        if (SampleNow == SampleChangeStage)
        {
            Stages newStage = static_cast<Stages>((StageNow + 1) % StageCount);
            ChangeStage(newStage);
        }
        VolumeNow = VolumeNow * Level;
        SampleNow++;
    }
    return VolumeNow;
}
```

Рис. 11

```
void Envelope::CountLevel(double LevelStart, double LevelEnd,
    unsigned long long SampleLength)
{
    Level = 1.0 + (log(LevelEnd) - log(LevelStart)) / (SampleLength);
}
```

Рис. 12

```
switch (newStage)
{
    case Off:
        VolumeNow = 0.0;
        Level = 1.0;
        break;
    case Attack:
        VolumeNow = MinVolume;
        CountLevel(VolumeNow, 1.0, SampleChangeStage);
        break;
    case Decay:
        VolumeNow = 1.0;
        CountLevel(VolumeNow, fmax(StageTime[Sustain], MinVolume),
                           SampleChangeStage);
        break;
    case Sustain:
        VolumeNow = StageTime[Sustain];
        Level = 1.0;
        break;
    case Release:
        CountLevel(VolumeNow, MinVolume, SampleChangeStage);
        break;
    default:
        break;
}
```

Рис. 13

Условно стадии можно разделить на два типа: те, которые изменяют амплитуду сигнала во времени, и те, которые ее не изменяют. Программный код функции по расчету амплитуды определенного семпла CountSample() на разных стадиях представлен на рис. 11.

На стадиях Off и Sustain просто возвращается та громкость, которая была установлена изначально (VolumeNow), так как эти состояния не меняют амплитуду. Когда SampleNow достигает определенного значения (времени, заданного ручкой плагина и переведенного в количество семплов), то огибающая переключается на следующую стадию (StageNow+1). Деление с остатком на количество стадий необходимо для того, чтобы на переходе с четвертой стадии (Release) номер стадии становился не 5, а 0 и начиналась стадия Off. После условного оператора идет увеличение текущего семпла SampleNow на единицу.

Текущее значение амплитуды семпла VolumeNow определяется как текущее значение амплитуды, умноженное на переменную Level. Для вычисления этой переменной создана функция CountLevel(), код которой представлен на рис. 12.

Это – простая реализация алгоритма Кристиана Шонебека для экспоненциальной огибающей [3]. Функция принимает в качестве аргументов начальное значение громкости, конечное значе-

ние громкости и количество семплов, из которых можно получить множитель для текущей огибающей. Так как человеческое ухо от природы воспринимает громкость в логарифмическом виде, то изменение громкости должно быть экспоненциальным, чтобы оно казалось линейным на слух [13].

На рис. 13 представлен программный код функции ChangeStages(), которая отвечает за переключение стадий.

Данная функция описывает, что конкретно нужно делать с сигналом при попадании на определенную стадию.

При попадании на стадию Off текущая громкость становится равной нулю, а множитель Level – равным единице.

Как только сигнал переходит на стадию Attack, громкость сигнала становится равной MinLevel (0.0001 – практически нулевой), и начинает свою работу функция CountLevel(), которая меняет значение громкости с близкого к нулю до единицы за время, которое выбрано ручкой на плагине. За это время отвечает переменная SampleChangeStage.

Затем сигнал переходит на стадию Decay, на ней он опускается до уровня, который задается параметром Sustane. Функция fmax() не позволяет сигналу опуститься ниже нулевого уровня.

Когда сигнал попадает на стадию Sustain, то становится равным переменной StageTime[Sustain], которая конкретно для данного параметра означает не время его работы, а уровень сигнала. Значение множителя Level при этом равно единице. При этом значении он никак не влияет на громкость сигнала.

Реализация эффектов обработки сигнала. Для обработки синтезированного плагином сигнала разработано достаточно большое количество различных эффектов. Алгоритмы и программные реализации некоторых из них подробно описаны в [6].

В данном плагине на основе разработанных в [6] алгоритмов были реализованы фильтры нижних и верхних частот, а также низкочастотная модуляция.

Фильтр нижних частот (*англ.* low-pass filter) – эффект, изменяющий амплитудно-частотную характеристику сигнала, пропуская частотный спектр сигнала ниже частоты среза и подавляя частоты сигнала выше этой частоты [14]. Т. е. в зависимости от того, какая частота фильтрации выбрана, фильтр будет подавлять все частоты, находящиеся выше выбранной частоты. Фильтр верхних частот, в свою очередь, пропускает частотный спектр выше частоты среза.

Для создания фильтра нижних и верхних частот был создан класс Filter, содержащий две функции: HighCut() – фильтр нижних частот, и LowCut() – фильтр верхних частот. Семплы сигнала имеют тип double. Поскольку функции работают с семплами сигнала, то они будут принимать и возвращать значения типа double.

Поскольку фильтров два, то и частоты среза две, и поэтому нужно создать переменные типа double, например HighCutFreq и LowCutFreq. Данные переменные будут получать значения при повороте соответствующих ручек плагина на графическом пользовательском интерфейсе.

В данном плагине реализован фильтр с бесконечной импульсной характеристикой. Он подразумевает использование выхода в качестве входа, т. е. в данном случае выходящий семпл зависит от значения предыдущего выходящего семпла [15]. Для этого необходимо создать буфер – переменную типа double, хранящий предыдущее значение. Фильтр линейный, поскольку к входящему сигналу будет применен линейный оператор (частота среза).

Для получения эффекта фильтрации верхних частот достаточно повторно реализовать фильтр нижних частот, а затем из исходного сигнала вычесть полученную низкочастотную фильтрацию – получится фильтрация верхних частот. Алгоритм один – результата два.

Программный код функций двух фильтров представлен на рис. 14.

```
double Filter::HighCut(double inputValue)
{
        if (inputValue == 0.0) return inputValue;
        b0 += HighCutFreq * (inputValue - b0);
        b1 += HighCutFreq * (b0 - b1);
        b2 += HighCutFreq * (b1 - b2);
        b3 += HighCutFreq * (b2 - b3);
        return b3;
}

double Filter::LowCut(double inputValue)
{
        if (inputValue == 0.0) return inputValue;
        b10 += LowCutFreq * (inputValue - b10);
        b11 += LowCutFreq * (b10 - b11);
        b12 += LowCutFreq * (b11 - b12);
        b13 += LowCutFreq * (b12 - b13);
        return inputValue - b13;
}
```

Рис. 14

Данный фильтр представляет собой четыре последовательных соединения фильтра нижних частот первого порядка. Это значит, что каждую октаву амплитуда составляющих сигнала над частотой среза становится в два раза ниже. Т. е. громкость сигнала падала бы на 6 дБ, если бы был использован один такой фильтр, а в данной программе такой фильтр реализован последовательно четыре раза, что дает понижение громкости на 24 дБ/окт, и, значит, в отфильтрованном сигнале будет меньше верхних частот.

В качестве еще одного эффекта в плагине реализована простая низкочастотная модуляция сигнала по амплитуде. Эффект несколько схож с приемом вибрато при игре на музыкальных инструментах.

Для низкочастотной модуляции необходим осциллятор, генерирующий волну низкой частоты, менее 20 Гц [16]. Такие колебания не слышны человеческому уху, но результат работы осциллятора и не будет подаваться на выход плагина. Результат работы данного осциллятора будет влиять на сгенерированный синтезатором сигнал.

Программная реализация модуляции сигнала представлена на рис. 15.

В данном случае в ранее созданном классе Filter была создана функция Mod(), которая в качестве аргумента принимает входящий семпл типа double. В этой функции реализован простой осциллятор синусоидальной волны, который был описан ранее. Переменная ModMix отвечает за то, насколько сильно будет применен данный эффект, и получает свое значение при изменении положения соответствующей ручки на интерфейсе плагина.

При этом выходящий сигнал складывается из суммы входящего сигнала, умноженного на результат работы осциллятора (sin(Phase)) и на переменную ModMix, и входящего сигнала, умноженного на разность (1-ModMox).

Параметры плагина и графический интерфейс пользователя. Параметрами плагина служат элементы графического интерфейса (ручки, фейдеры, переключатели и т. д.), взаимодействие с которыми влияет на входящий сигнал или выпол-

```
double Filter::Mod(double inputValue)
{
    inputValue = inputValue * sin(Phase) * ModMix + inputValue * (1 - ModMix);
    Phase += PhaseIncrease;
    while (Phase >= twoPI)
    {
        Phase -= twoPI;
    }
    return inputValue;
}
```

Рис. 15

Рис. 16

няет другие определенные действия, задуманные разработчиком (например, открывает дополнительное меню или переходит по ссылке) [17].

Современный аудиоплагин имеет свой графический интерфейс, включающий в себя различные регуляторы, фейдеры, переключатели и многое другое. Для создания пользовательского интерфейса использовались возможности библиотеки WDL-OL. Был разработан дизайн графического интерфейса, с помощью программы Adobe Photoshop нарисованы необходимые графические изображения (ручка, изменяющая параметры, кнопка изменения волны, черная и белая клавиша фортепианной клавиатуры) и написаны функции, осуществляющие связь графических параметров с переменными, участвующими в синтезе и обработке сигнала.

Внешний вид графического пользовательского интерфейса плагина в цифровой рабочей станции Reaper представлен на рис. 16.

С помощью библиотеки WDL-OL и языка программирования C++ разработан и реализован плагин-синтезатор, позволяющий синтезировать звуковые волны, а также производить цифровую обработку звука с помощью различных эффектов.

На этапе программной реализации синтеза сигнала и огибающей были созданы функции синтеза волн четырех основных типов (синусоидальная, пилообразная, квадратная и треугольная), а также огибающая функция в виде четырех параметров сигнала (Attack, Release, Decay и Sustain).

На этапе программной реализации эффектов обработки сигнала были созданы функции фильтрации нижних и верхних частот сигнала, а также реализована низкочастотная модуляция амплитуды сигнала.

Для удобства работы с плагином был спроектирован и реализован графический интерфейс пользователя.

Разработанный аудиоплагин имеет формат VSTi, поскольку является инструментом, что позволяет подключать его практически ко всем цифровым звуковым рабочим станциям.

В качестве цифровой звуковой рабочей станции, в которую загружался разработанный плагин, использовалась цифровая аудиостанция Reaper.

Полный код плагина размещен на портале GitHub [18].

СПИСОК ЛИТЕРАТУРЫ

1. Загуменнов А. П. Компьютерная обработка звука. М.: ДМК Пресс, 2017.
2. Gazi O. Understanding digital signal processing. Singapore: Springer, 2018.
3. Zolzer U. Digital audio signal processing. Hoboken: Wiley-Blackwell, 2008.
4. Pirkle W. Designing audio effect plug-ins in C++: with digital audio signal processing theory. Oxford: Focal Press, 2012.
5. Reiss J., McPherson A. Audio effects: Theory, implementation and application. Bosa Roca: Taylor & Francis Inc, 2014.
6. Кошелев Е. Е., Букунов С. В. Разработка аудиоплагина для цифровой рабочей станции // Вестник Рос. нового ун-та. Сер.: Сложные системы: модели, анализ и управление. 2020. Вып. 4. С. 85–97.
7. Айфичер Э., Джервис Б. Цифровая обработка сигналов: практический подход: 2-е изд. М.: Вильямс, 2004.
8. Pirkle W. Designing software synthesizer plug-ins in C++: For RackAFX, VST3, and Audio Units. London: Taylor & Francis Ltd, 2015.
9. Lazzarini V. Computer music instruments: Foundations, design and development. Cham: Springer, 2017.
10. Loy G. Musimathics. Vol. 1: The mathematical foundations of music. Cambridge: The MIT Press, 2011.
11. Loy G. Musimathics, Vol. 2: The mathematical foundations of Music. Cambridge: The MIT Press, 2011.
12. Что такое ADSR. URL: https://fierymusic.ru/teoriya-zvuka/adsr (дата обращения 13.01.2021).
13. Kientzle T. A Programmer's guide to sound. Boston: Addison Wesley, 1997.
14. Zavalishin V. The art of VA filter design. URL: https://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/ VAFilterDesign_1.1.1.pdf (дата обращения 19.07.2020).
15. Coulter D. Digital audio processing. New York: R&D, 2000.
16. Модуляция. URL: https://okno-audio.ru/wiki/4853.html (дата обращения 13.01.2021).
17. Boulanger R., Lazzarini V. The audio programming book. Cambridge: The MIT Press, 2010.
18. GitHub. URL: https://github.com/egorrrkkkaaa/vkrSynth (дата обращения 09.03.2021).

E. E. Koshelev, S. V. Bukunov
Saint Petersburg State University of Architecture and Civil Engineering

PLUG-IN SYNTHESIZER FOR A DIGITAL WORKSTATION

A developed audio synthesizer with its own graphical user interface is described. The plugin implements two oscillators, an ADSR envelope, a low pass filter, a high pass filter, and a low frequency modulation effect. The oscillators used generate four types of waves: sine, square, triangular and sawtooth. The possibility of mixing waves of two oscillators in any proportion has been implemented. The necessary algorithms have been developed to implement each component of the plug-in. A graphical user interface in the form of a piano keyboard, the keys of which are triggered by pressing the left mouse button, was created for the convenience of working with the plugin. The interface contains knobs and buttons for changing signal processing parameters, as well as a knob for adjusting the output volume. The developed plug-in is in VSTi format and can be used in almost any modern digital sound workstation. The plug-in is implemented in the C ++ language using the object-oriented programming approach. To create the plug-in, the Microsoft Visual Studio development environment and the WDL-OL library were used. The digital audio station Reaper was used as a digital sound workstation for testing of the developed plug-in.

Digital audio processing, plug-in synthesizer, digital audio workstation, object-oriented programming, graphic user interface