DEVELOPMENT TOOLS FOR PWGLSYNTH

Vesa Norilo and Mikael Laurson
Sibelius Academy
CMT

ABSTRACT

This paper presents recent developments in our synthesis environment PWGLSynth. We introduce two programming tools that allow the user to extend PWGLSynth with new C++ modules. In the first one the user operates with ordinary textual C++ programming. The system also allows to reuse in the code any existing synthesis modules. In the second tool the user first defines visually a synthesis patch with the help our abstraction scheme. The abstraction is then automatically translated to a binary file that represents the new synthesis box. The Appendix gives three C++ examples that demonstrates the syntax that is used by the developer kit.

1. INTRODUCTION

PWGL [1] and its predecessor PatchWork have already a long tradition of providing interfaces for user extensions. In its simplest form the user can add Lisp code to a patch by using one of the available text-based editors. These function definitions can then be automatically be converted to visual boxes. For more advanced projects the user can employ the user-library protocol. The special case in this context has been the real time synthesizer PWGLSynth [3], being written in C++ for better floating point DSP performance. Although PWGLSynth provides a large set of kernel synthesis boxes, it has been limited by the fact that the user has not been able to define new modules. This article discusses our efforts to overcome this limitation.

The widely used Max/MSP [4] and PD [5] systems have long offered an interface for user provided externals. The idea of extending a core set of synthesis modules with user provided special case modules is well suited to a system which is used in such a wide range of artistic contexts that all user needs cannot be recognized and built into the core system.

Another example of an extendable system is the widely popular Virtual Studio Technology by Steinberg Inc. The concept is to allow commercial and other parties to provide small software processors that can be slotted into a system which is used in such a wide range of artistic contexts that all user needs cannot be recognized and built into the core system.

Yet another example of an extendable system is the widely popular Virtual Studio Technology by Steinberg Inc. The concept is to allow commercial and other parties to provide small software processors that can be slotted into several digital audio workstation software packages that comply with the standard. The concept is repeated in various formats, including DirectX filters by Microsoft Corporation and Audio Units in Apple Macintosh.

Recently, work has been done to accommodate third party extensions to PWGLSynth. There are currently two principal methods for this, Extension Developer Kit and Visual Patch Compiler.

The rest of this paper is organized as follows. We start with our first extension tool that requires traditional textual C++ programming skills. (Section 2). Next we discuss practical case studies (Section 3) that are complemented by C++ code samples found in the Appendices. Section 4 gives some important ideas dealing with the special feature that allows to reuse modules that are already existing in the system. This idea is evolved in Section 5 that describes the second extension scheme in our system. Here the user can define new synthesis boxes visually, thus opening the extension system also to users who don’t master C++ programming.

2. EXTENSION DEVELOPER KIT

The developer kit enables interested parties to program synthesis modules in C++. A C++ base class is provided to the third party developer, along with facilities for exporting the newly provided synthesizer module in a way compatible with PWGLSynth. Using the developer kit, the developer can compile MacOS X bundles or Windows DLLs and distribute any synthesis modules in binary format. PWGLSynth boxes are C++ classes with a standardized virtual function table, accomplished by a specified abstract base class. Thus the interface for exporting and importing boxes to PWGLSynth can be kept lightweight and simple, with just a single function call requesting the instantiation of a synthesis module and the memory reference to the instance created.

This non-wrapped interface offers the benefits of low overhead function calls, but requires virtual function table compatibility (if any alterations made to the pwsBox base class on the synthesizer side may break any existing extensions. The class has been mature for several years now, and extensions to it can be made transparently in a way not disrupting the virtual function table.

Another issue is the compiler compatibility, since a C++ compiler can conceivably lay out the virtual function tables of classes in an implementation dependant way. Fortunately all C++ compilers we are aware of produce compatible virtual function tables. Further, Xcode and Visual Studio are such de facto standards on their respective platforms that the issue is not considered troubling.

55
3. PRACTICAL USE OF THE DEVELOPER KIT

We provide a template C++ project to interested third parties, containing an example project in Xcode and Visual Studio formats that can be used to build a PWGLSynth extension. The task of writing an extension is then accomplished by designing a C++ class inherited from the base class provided in the developer kit. By overriding different members of the base class the developer can specify how the box responds to several system events. In the following section we briefly describe different member functions the developer can override.

3.1. Constructor

The class constructor is used to define the box configuration. The base class provides member functions for setting the box name, category, I/O configuration and facilities for providing other user interface data for the inputs of the resulting synthesis box. As defined by the C++ language, the constructor is called whenever an instance of the class is created.

3.2. Prepare

Prepare is a member function that gets called before a synthesis run. This member allows the box to prepare for audio streaming, zeroing any feedback buffers and allocating resources necessary.

3.3. Refresh

Refresh is our method of providing unified audio and control signals. Refresh is called when the system is aware that the input signal changes. This makes it possible to provide coarse grained control rate calculations that support the audio stream. It is possible to mark inputs as control inputs that require refresh, and the system transparently provides coarse refresh calls when they are connected to audio outputs. An example of a typical a Refresh call would be to read high level parameters from the box inputs and translate them to an internal representation, such as filter frequency and bandwidth to feedforward and feedback coefficients.

3.4. Process

This is the main member of a pwsBox class, providing the actual DSP calculation that is used in the main synthesis loop. The pwsBox base class takes care of setting up connection buses between boxes. These buses are defined as memory locations storing a single frame of the signal at a time. The box input memory locations can be accessed by the inputBoxBuf pointers and the box should store its own final result to the memory location specified by the pointer outputBuf.

3.5. Trigger

Trigger is a method for providing one-off trigger events to a synthesis box. PWGL will look for inputs with the name trig and provides a push-button instead of a regular value input leaf. When the user presses the button, this member function of the corresponding class is called. A typical triggered box would be a sample player that starts streaming a sampled waveform anew whenever a trigger event is sent.

The Appendices contain three characteristic box examples ranging from a simple DC blocking filter demonstrating the use of Prepare and Process (8.1), to a box emitting a pseudo dirac impulse when triggered using the Trigger method (8.2), to a vectorized example, a summation operation capable of adding together multichannel signals or vectors of values (8.3).

4. BUILDING COMPOUND BOXES

Often it's possible to make use of synthesis modules already extant within the system. Many DSP algorithms are merely variations and combinations of several well known building blocks.

The synthesizer extension developer kit provides access to all other synthesizer modules visible to the system. User code can request the system kernel for an instance of another synthesis box, identified via an alphanumeric name by which it is also accessed in Lisp. PWGLSynth contains a dynamic manufacturing system that can instantiate synthesis boxes from symbol names during runtime.

The manufacturing system is exposed to external box code via a simple function call.

Using the manufacturing system, an external, binary format synthesis module can be built of other modules, either in the PWGLSynth kernel or even other external binary extensions. We are currently investigating methods of providing feedback about possible module dependency chains being broken.

5. VISUAL PATCH COMPILER

The Visual Patch Compiler (VPC) can be thought of as a visual interface to the extension developer kit. There is an automated mechanism that can process and analyze a visually constructed PWGLSynth patch and produce C++ code based on the same scheduling algorithms that are used for the traditional real time processing. Making use of the facility for accessing all system synthesis boxes from binary extensions, the compiler produces a standalone synthesis module bundle from a visual patch that refers some parts of the processing to the modules used within the patch. The compiler makes use of the box manufacturing system to build an extension with extant synthesis boxes, producing some glue code to allow the boxes inside a binary extension to interact with the outside patch. Even some GUI data can be encoded inside a binary extension, as the system provides means for the user
to provide box documentation and information on suitable parameter ranges for inputs.

Basically all synthesis parameters can be replicated and built into the box, but as the PWGL patch preceding the synthesis boxes is evaluated for values during compilation, any dynamic patches or Lisp code becomes a static snapshot.

VPC is realized using the standard PWGL abstraction mechanism. An abstraction box was a natural choice for us as it contains inputs and outputs just like an ordinary synthesis box. Furthermore, an abstraction contains a sub-window that is used here to define visually the actual DSP patch that in turn is translated automatically to C++ code by the underlying VCP system. In Figure 1 the left-side part shows an abstraction box (labeled with an 'A' under the low-right corner) having the name 'stringmodel3'. It has three inputs that contain default values that will be used later by the resulting synthesis box. The user can also define suitable range values for each individual input using a standard PWGL input-box editor.

The patch to the right of Figure 1 gives the contents of the abstraction box. The upper row of three small abstraction-in boxes refer to the inputs of the abstraction box. They are indexed (the index number is given above the box). The index refers to the input position of the abstraction box, thus ’1’ is mapped to the first input, ’2’ to the second input, and so on. The abstraction-in boxes have extra labels (’id’, ’trig’, and ’freq’) which are used to name the inputs of the resulting synthesis box. The rest of the patch consists of ordinary synthesis boxes (labeled with ’S’) that define a simple string model.

After the abstraction is finished the user can start the VSP process to translate the visual patch definition to a binary extension. This is done by choosing from the dedicated abstraction box popup menu the option ‘compile synth patch’. A dialog will appear asking for box name, synthesis menu name, and box documentation. After this a background process will start that first creates a C++ text file based on the abstraction box information. Then the C compiler is evoked which in turn creates the final binary file. If this part of the process succeeds, the resulting extension is placed in the synthesis part of the PWGL-User folder. This sub-folder will always be loaded at PWGL startup making the new extension available.

Figure 2 shows an example where two instances of the ’stringmodel3’ synth box are used within an ordinary synthesis patch. Note that the boxes are now labeled with an ’S’ which means that they have the same status as all other synthesis boxes.

6. ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland (SA 105557 and SA 114116).
7. REFERENCES


8. APPENDICES

8.1. Simple box example

```cpp
pwsDCBox::pwsDCBox()
{
    // name of the box in PWGLSynth
    setBoxName("DC");
    // for menu hierarchy
    setBoxCategory("Filters/basic");
    // number of inputs
    setNumInputs(1);
    // input number '0' is called 'sig'
    setInputName(0,"sig");
}

void pwsDCBox::Prepare()
{
    // zero feedback and feedforward buffers
    y1=x1=0;
    // calibrate coefficient to sampling rate
    R=1-(190./_pws_sampleRate);
}

void pwsDCBox::Process()
{
    // expect floating point signal
    float ** ptr = (float **)inputBoxBuf;
    // perform the actual filtering
    y1=ptr[0]-x1+R*y1;
    // store x(t-1)
    x1=*ptr[0];
    // write the output
    *(float *)outputBuf)=y1;
}
```

8.2. Trigger box example

```cpp
pwsImpulseTrigBox::pwsImpulseTrigBox()
{
    // name of the box in PWGLSynth
    setBoxName("impulse-trigger");
    // for menu hierarchy
    setBoxCategory("control/trig");
    // number of inputs
    setNumInputs(2);
    // no input index given:
    // same name+slider info for all inputs
    setInputName("vector");
    setSliderInfo(-1,1,0,1);
    // end user documentation
    boxCaps.docString="Calculates the sum of two vectors.";
}

pwsImpulseTrigBox::Process()
{
    // obtain vectors connected to inputs
    float *vec1=(float *)inputBoxBuf[0];
    float *vec2=(float *)inputBoxBuf[1];
    float *sum=(float *)outputBuf;
    // outputBufLen is provided by
    // the base class
    int n = outputBufLen;
    // calculate each output element
    while(n--) *sum++ =(*vec1++)+vec2++;
}
```

8.3. Vector box example

```cpp
pwsAddVector::pwsAddVector()
{
    // set name and category
    setBoxName("add-vector");
    setBoxCategory("vector/1-basic");
    // two inputs
    setNumInputs(2);
    // no input index given:
    // same name+slider info for all inputs
    setInputName("vector");
    setSliderInfo(-1,1,0,1);
    // end user documentation
    boxCaps.docString="Calculates the sum of two vectors.";
}

pwsAddVector::Process()
{
    // obtain vectors connected to inputs
    float *vec1=(float *)inputBoxBuf[0];
    float *vec2=(float *)inputBoxBuf[1];
    float *sum=(float *)outputBuf;
    // outputBufLen is provided by
    // the base class
    int n = outputBufLen;
    // calculate each output element
    while(n--) *sum++ =(*vec1++)+vec2++;
```