NGS Performance Measure Sample User's Guide

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

About This Document

The NGS audio engine runs code on the main ARM processor as well as the Codec Engine. It supports a range of different configuration options, which makes it difficult to provide a specific voice cost for all combinations. The NGS performance sample is designed to give developers the opportunity to analyze the performance of different NGS configurations when designing their game engine.

The application can be used with little knowledge of the underlying source code to measure initial performance costs. Source code is provided so that the user may gain a deeper understanding of the NGS configurations used and also so that it can be modified for more specific use cases.

The first section of this document explains the application and how to configure and understand it. The second section gives an overview of the source code.

1 NGS Profiler Application

Overview

The application measures the timing around two core aspects of NGS usage:

(1) The main update cycle including `sceNgsSystemUpdate()` and `sceNgsVoiceGetStateData()`.

(2) The parameter update functions including `sceNgsPatchGetInfo()` and `sceNgsVoicePatchSetVolumesMatrix()` for volumes; and `sceNgsVoiceLockParams()` and `sceNgsVoiceUnlockParams()` for module parameter updates.

For the ARM processor, the timing information is reported as the absolute elapsed time and the actual percentage of a single core used during that time.

The application also reports the Codec Engine usage as a percentage during the update cycle. Codec Engine usage is not recorded during parameter updates as this only incurs cost on the ARM processor. Codec Engine performance is measured using `sceCodecEnginePmonGetProcessorLoad()`.

The performance figures are given statistically and also shown over time in a performance graph.

The following items may be modified, while running, to monitor the effect on performance in different areas:

- The number of voices to update per frame (use the L and R buttons)
- Whether voices are stopped or playing (X to start/stop)

The application is designed to ensure that all voices in the configuration file are in constant use. This means that the performance measurements are under peak-use conditions.

Configuration

The configuration file `config.txt` is designed to give some flexibility in designing different voice types to suit your game configuration. By editing the configuration file you can create different voice types which might be representative of your game's needs.

If you wish to customize the application to suit more specific configurations, see Chapter 2, NGS Profiler Source Code Overview.

Using unmodified source code, the application creates a single master buss and limiter. All voices specified in the user configuration file are connected to the limiter which is, in turn, connected to the master buss. The configuration file specifies the different voice types as described in the following sections.

The sample data for the different voice types is included with the application and selected based on the voice type.

The configuration file is effectively a comma-separated values (CSV) file which supports blank lines, and comments that are prefixed with the # symbol. In the sample configuration file, column headings are specified using a comment for easy display in a spreadsheet. Each column is described below in the order they are used within the file:

Number of Voices

This is the number of voices that should be instanced according to the specification of this line in the configuration file. If this number is 0, the line is ignored.

Voice Type

The voice can be one of three types: PCM, VAG and AT9. PCM and VAG voices use the “Simple Player” voice definition in `simple_voice.h` or “Voice Template 1”, defined in `voice_template_1.h` (depending on the value of the flag `VOICE_DEF_COMPLEXITY`, defined in `ngs_scenario.h`); whereas AT9 voices use the “AT9 simple voice” definition in `simple_voice_at9.h` or “Voice Template AT9”, defined in `voice_template_at9.h`.

Number of Channels

This specifies the number of channels in the audio data. Only mono or stereo voices are supported.

Minimum and Maximum Sample Rate

These values allow you to randomize the sample rate used during voice playback. If you expect particular voice types to use a fixed sample rate the minimum and maximum values should be identical.

Update Flag

This can be set to either 0 or 1. If it is 1, parameter updates are applied to these voices to simulate the effect of changing values such as playback pitch, volume or filter coefficients. For some voices, such as ambient music, it may be reasonable to expect that they do not require parameter updates and so this value should be set to 0 for simulation purposes.

Reverb Flag

This flag indicates whether reverb should be applied to these voice types. Please remember that a reverb DSP is instantiated for every voice of this type, so if “Number of Voices” is set to 4 and the reverb flag is active, then four reverb DSPs will be instantiated. Please consider this carefully when profiling your configuration; for example, if your game is only likely to use 1 reverb, then ensure that the “Number of voices” is set to 1 for that configuration line.

Uncached Memory Flag

The uncached memory flag allows you to specify whether the voice type should use cached or uncached memory. Uncached memory generally leads to faster NGS performance.

Loop Flag

This flag specifies whether the voices should loop. Voices which do not loop are restarted in the application when they reach the end of playback.

Streaming Flag

This flag allows you to simulate streamed voices which queue the buffer data.

Bypass Flags

This is a bit field which holds the LSB at the left and MSB at the right. Each bit describes which modules of the voice definition should be bypassed in the order the modules appear in the voice template file. The size of the bit field is dependent on the voice type. For the “Simple Voice” and “AT9 simple voice” definition, there are six values; and for the “Voice Template 1” and “Voice Template AT9” there are 10. To bypass a module, the bit value should be set to 1. It is not necessary to specify the full bit field, unspecified bits default to 0 for unbypassed.

Recommended Usage

The recommended procedure for using this application is:

(1) Create a baseline configuration file (see example enclosed with the sample) with the maximum number of expected voices.

(2) Run the application.

(3) Stop the voices to indicate the baseline performance figures with no voices playing.

(4) Restart the voices to measure performance usage with all voices playing.

(5) Adjust the number of parameter updates to assess the impact on performance.

(6) Edit the configuration file to assess different configurations or other use cases (average expected load, etc).

Performance Analysis

The top performance graph shows the ARM performance during the NGS update cycle. The middle graph shows the value reported by the Codec Engine Performance Library (for further details see the Codec Engine documentation, *libcodecengine Overview*). The bottom graph shows the update performance and timing.

For all graphs, statistical values are shown in colored text and a performance line is shown on the graph in a matching color.

Performance Units

The "NGS Update" (top) and "NGS Param Update" (bottom) graphs show two performance figures. The second column simply shows the elapsed time (in milliseconds) during the measurement, this is based on the global system timer. The first column shows the percentage of a single ARM core that is used by the NGS engine during the timing operation. This is calculated as follows:

$$\frac{\left. \text{ticks used by this thread} \middle/ \text{ticks per second} \right.}{\text{elapsed time}} \text{ X } 100$$

The "Codec Engine Usage" shows the figure reported by the Codec Engine Performance Library (see Codec Engine documentation for further details).

Statistical Values

The statistical values are described below:

- **Scale:** This is the maximum value that can be displayed on the graph and is based on the long-term maximum value from the data. This value is used for the graph scale. The value can be reset to the current maximum value by touching the screen over the graph or statistical data.
- **Max:** This is the current maximum value shown on the graph. Touching the statistical values or the graph sets "Scale" to this value.
- **Avg:** This is the average of all values shown on the graph.
- **Cur:** This is the most recently plotted value on the graph.

Interface

- Δ: Show/Hide Debug Information
- **O:** View a configuration summary in the display and TTY.
- □: Reload the configuration file.
- **X:** Start/ Stop all voices.
- **L:** Reduce the number of voices to update.

- **R:** Increase the number of voices to update.
- **Select:** Capture screenshot (to `app0:screenshot.bmp`).

Touch each graph to rescale.

2 NGS Profiler Source Code Overview

The Profiler source code is split into a number of modules which are described below.

NGS Profiler Main Code

The application entry point is in `main.cpp`. The class `NgsProfiler` (derived from `SampleSkeleton`) contains the high-level application code (`ngs_profiler.h` and `ngs_profiler.cpp`). Key functions are described in the following sections.

main()

Main is the application entry point. This function initializes the application using the `init()` function, then calls the `update()` and `render()` functions, which contain the main application code and finally calls `shutdown()`.

init()

This function sets up the application by loading necessary system modules. `SampleSkeleton::init()` initializes the main graphic display components, while `gxmSpriteInitialise()` initializes the sample specific graphics sprites. The touch screen interface is initialized using `touchInitialise()`.

The method `initNgsGuiObjSystem()`, is used to setup and layout a number of graphical objects used in the application front-end.

The audio files used in the application for test data are loaded using `loadAudioFiles()` and the low level audio system is initialized using `prepareAudioOut()`. `perfInit()` is called to initialize the performance measurement.

Finally, the user configuration file which defines the NGS configuration is loaded by calling `loadConfig()`.

update()

The main processing consists of a main application loop which is terminated by pressing the PS key. The loop uses system timers to maintain an approximate frame rate of 60 fps. The update function performs the following tasks:

(1) Process buttons and touch screen input.
(2) Updates parameters on selected NGS voices.
(3) Updates the GUI.

render()

The render function is also called in the main processing loop and it is responsible for the graphics rendering.

shutdown()

This function shuts down the application by unloading the relevant system modules. `SampleSkeleton::shutdown()` deinitializes the main graphic display components, while `gxmSpriteShutdown()` shuts down the sample specific graphics sprites. The touch screen interface is terminated using `touchTerminate()`.

loadConfig()

This function is called during initialization and whenever the configuration file is reloaded. The function performs the following actions:

(1) Reads configuration file and initializes the NGS engine using `initNGS()` function from `ngs_scenario.h`.

(2) Configures the NGS engine using `configureNGS()` function from `ngs_scenario.h`.

(3) Launches the NGS update thread.

(4) Starts playback of all voices.

unloadConfig()

`unloadConfig()` is called before reloading the configuration file. This function performs the following actions:

(1) Stops all active voices.

(2) Kills the audio update thread.

(3) Releases the NGS engine.

(4) Frees all buffers allocated to NGS by calling the `freeBuffers()` function from `ngs_scenario.h`.

Configuration Module

The configuration module is encapsulated by `ngs_config.cpp` and `ngs_config.h`. This is a simple module which parses the `config.txt` file.

The header contains the global configuration options (such as NGS granularity), structures to hold configuration options, and one function: `readConfigData()`.

`readConfigData()` takes a pointer to an unused `ConfigData` pointer. The function effectively gives the caller a pointer to the `ConfigData` structure which the caller can then parse in code. The function returns zero if there is no error.

This function is called by the `initNgs()` function in `loadConfig()`.

Performance Module

The performance module is encapsulated by `ngs_perform.cpp` and `ngs_perform.h` and abstracts performance measurement code from the main application.

Performance measurement is handled by separate functions, which start and stop the performance measurement for the parameter updates and the main NGS update cycle. The main functions are described below:

- `perfInit()`: Sets up the ARM performance counters used by the application.
- `perfUpdateTimingBegin()`: Starts the NGS update timer.
- `perfUpdateTimingEndAndReport()`: Stops the NGS update timer and stores results in global variables.
- `perfParamTimingBegin()`: Starts the param update timer.
- `perfParamTimingEndAndReport()`: Stops the param update timer and stores the result in global variables.

The parameter timing functions are called during the parameter update function in `updateVoiceParams()` and the update timing macros are called by the NGS update thread around the NGS update cycle calls.

This module makes use of the PlayStation®Vita performance library and the PlayStation®Vita Codec Engine performance library.

The module also implements the following private functions:

(1) `perfStoreUpdateTiming`: Updates the "NGS Update" performance graph on the GUI. This is used in the function `perfUpdateTimingEndAndReport()`.

(2) `perfStoreParamTiming`: Updates the parameter performance graph on the GUI with the latest measurements. This is used in `perfParamTimingEndAndReport()`.

NGS Scenario Module

This module is encapsulated by the `ngs_scenario.cpp` and `ngs_scenario.h` files. This module is responsible for the initialization and configuration of the NGS engine according to the user-specified configuration file.

Various game-style configuration options are emulated by the scenario module. Here are some key points describing its operation:

- Different audio data files are used depending on the combination of number of channels, format (VAG, AT9 or PCM), and whether the data is streaming or not.
- For each of the above combinations, the audio data is duplicated into different memory locations (up to `MEM_BLOCK_SIZE)` to emulate different in-memory audio files and provide conditions for cache-misses that might be expected to occur in-game during audio playback.
- Audio data is copied into cached and uncached memory to allow performance measurement using both types of memory.
- For streaming files memory resident buffers are cycled for each combination above. The sample does not emulate disk access during streaming as this does not affect NGS performance.
- Stream buffers are not duplicated, except to copy into cached and uncached memory. Separate parts of the stream buffer are passed to NGS player buffers in the `SceNgsAT9Params` and `SceNgsPlayerParams` structures.
- The parameter update function updates the volume of each voice by setting the patch info. The function also emulates updates to the player modules and one EQ module. The emulation is performed by unlocking and locking the params buffers. Although no parameters are changed, the lock/unlock cycle causes parameters to be updated in the NGS update cycle.
- There are also some flags in `ngs_scenario.h` that can be used to configure the system:
  - `PARAM_UPDATE_TIME_STEP`: controls the parameter update rate (in seconds). By default is set to 60 fps.
  - `MODULE_PARAM_CHECK`: enables/disables parameter validation, which has an important impact on performance during parameter updates. By default is set to 1, to match the real system if the flag "`SCE_NGS_SYSTEM_FLAG_NO_MODULE_PARAM_CHECKING`" is not set manually.
  - `VOICE_DEF_COMPLEXITY`: Voice definition templates complexity. If the flag is set to 0, simple voices are used in the sample (`simple_voice.h` template for PCM/VAG, `simple_voice_at9.h` template for AT9). If the flag is 1 T1 voices are used (`voice_template_1.h` template for PCM/VAG, `voice_template_at9.h` template for AT9). The default value is 0.

The following key functions are exposed by this module:

initNgs();

This function initializes the NGS engine and allocates rack memory according to the configuration described in the user configuration file.

configureNgs()

This function configures the NGS engine by creating all required racks, voices and patches according the user configuration file.

shutdownNgs()

This function shuts down the NGS engine and releases rack memory allocated during `initNgs()`.

loadAudioFiles()

This function loads the audio files from disk into RAM and duplicates the data for the different scenario tests. This function is only called once during the lifecycle of the application as the same audio data buffers are used regardless of configuration type.

unloadAudioFiles()

This function clears up all memory allocated by the `loadAudioFiles()` function.

updateVoiceParams()

This function is responsible for emulating parameter updates for active voices that are marked for update in the configuration file. It selects the correct modules to update depending on the voice definition for the selected voice type.

stopAllVoices()

Stops all NGS voices by cycling through `s_voiceList`.

startAllVoices()

Starts all NGS voices by cycling through voices in `s_voiceList`.

NGS GUI Module

The NGS GUI helper class is a simple drawing module to enable display of performance figures using the PlayStation®Vita Development Kit. It is not intended as sample code for displaying graphics using the PlayStation®Vita system and will not be described further.