Document serial number: 000004892117

Sample Utilities Overview

000004892117

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

© 2013 Sony Computer Entertainment Inc.
All Rights Reserved.
SCE Confidential

Document serial number: 000004892117

Document serial number: 000004892117

Table of Contents

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

000004892117

Document serial number: 000004892117

©SCEI

Document serial number: 000004892117

Document serial number: 000004892117

About This Document

The purpose of this guide is to describe the sample utilities.

Conventions

The typographical conventions used in this guide are explained in this section.

Hyperlinks

Hyperlinks are used to help you to navigate around the document. To allow you to return to where you clicked a hyperlink, select **View > Toolbars > More Tools…** from the Adobe Reader main menu, and then enable the **Previous View** and **Next View** buttons.

Notes

Additional advice or related information is presented as a ‘note’ surrounded by a box. For example:

Note: This is an additional note.

Text

- Named application windows or GUI features are formatted in a bold sans-serif font. For example, the **Build** button.
- Names of keyboard functions or keys are formatted in a bold serif font. For example, **Ctrl**, **Delete**, **F9**.
- File names, source code and command-line text are formatted in a fixed-width font. For example:

```
SDK_DIR/samples/common/sample_utilities
```

Errata

Any updates or amendments to this guide can be found in the release notes that accompany the release.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

1 Introduction

This chapter provides an introduction to the sample utilities.

Overview

The sample utilities consist of a set of functions that encapsulate lower level API calls for initialization and shutdown, and some higher level runtime functionality where appropriate. The utilities provide default initialization functions for the lower level SDK libraries, but also allow this functionality to be customized for edge cases. The sample utility architecture is designed to allow the utilities to integrate easily into a sample application.

Purpose

The purpose of the sample utilities is:

- To initialise commonly used SDK features such as controllers, timers, graphics, and menus. This aids sample development by reducing the amount of code required to set up an application.
- To increase sample code clarity by encapsulating these common elements into included functions.
- To reduce duplication of startup and commonly wrapped functionality within the SDK.
- To create a quick-startup style set of functions, allowing newcomers to get started quickly and increasing productivity.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

2 Software Architecture

This chapter describes the sample utility architecture.

Overview

The sample utilities are a set of functions and associated structures, organized under a single `#include` file. There are two main types of utility:

- Those that perform initialization only
- Those that perform initialization and runtime functions

Initialization Utilities

Initialization Utilities act mainly to encapsulate the common initialization code required by most sample developers within their application. The goal is to reduce the amount of code that populates a particular sample. The general features of Initialization Utilities are:

- An initialization structure that contains variables common to an API's initialization requirements
- Some functions to initialize and shut down the API in question

In many cases, a sample initializes an SDK library with default options. To facilitate this, the sample utilities provide a function that initializes the common structure with such default variables. Another function is provided that performs the API initialization, taking this structure as an argument. By providing a separate function to set the default variables before actual initialization, the utilities allow the sample developer to modify variables if required for a particular sample.

For example, if a graphics sample requires all values to be initialized at their defaults except for a particular buffer size, the code may initialize the structure with the default values, but then modify the buffer size structure independently before initializing the API.

In this way, the quantity of initialization code in a sample is reduced for readability purposes, and these initialization sections can be encapsulated for use by other samples that require this functionality.

The MeshLoader Utility provides initialization and data buffer setup functionality. A Heap Utility, which provides custom memory block initialization and allocation functionalities is also included.

Runtime Utilities

Initialization and Runtime Utilities provide extra functionality in addition to initialization. These utilities wrap common functionality that is generally added on top of an underlying API.

One example is the Controller Utility, which provides extra functionality to manage the button pressed and released events. These functions are not part of the standard low-level controller API, though they are almost always required and have to be built on top separately.

By providing these and other similar functions, the Initialization and Runtime Utilities aim to reduce development time and increase productivity.

The Debug Menu Utility, Timer Utility, Graphics Utility, Font Utility, and Controller Utility all provide runtime functionality in addition to initialization.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

3 Including the Sample Utilities in your Project

This chapter describes how to use the sample utilities in sample code development.

Overview

For maximum simplicity, the sample utilities are designed to be used in sample code development as a single include and link operation.

To include the sample utilities in a C or C++ project:

(1) Include the `<sample_utilities.h>` file.

(2) Link with `libSampleUtilities.a`.

(3) In addition, you must link with any library that the sample utilities rely on, as the utilities do not replace, for example, the controller and graphics libraries.

Note: The MeshLoader Utility is currently usable with C++ applications only.

Compilation

To build the sample utilities library separately, open the sample utilities project file from `SDK_DIR/target/samples/sample_code/common/source/sample_utilities` in Visual Studio, and click **Build**.

Using the Sample Utilities

After you include the sample utilities header and set the link script within your project, the next step is to select which utility or set of utilities you wish to use. For C++ application development, you can derive a class from the `SampleSkeleton` base class provided in:

```
SDK_DIR/target/samples/sample_code/common/include/sample_skeleton/
sample_skeleton_base.h
```

This class includes minimal code for initialization, update, and shutdown of the basic sample utilities required for running an application.

Utility Types

The sample utilities are:

- **Controller Utility** – provides initialization, shutdown, and higher-level runtime functionality to manage controller state and provide useful event information.
- **Graphics Utility** – provides simple initialization and shutdown functions to reduce code duplication.
- **Timer Utility** – provides an interface to the low-level hardware timer, with useful runtime functions such as `update()`, `getTotalTime()`, and `getTimeDelta()`.
- **Debug Menu Utility** – provides a simple on-screen menu with controller input to modify application variables at run time.
- **MeshLoader Utility** – provides functions to load a simple COLLADA mesh and store data in an interleaved buffer, together with index and texture data.
- **Font Utility** – provides functions to initialise fonts and create an image buffer containing information to render unicode characters.
- **Heap Utility** – provides functions to initialise and terminate a heap of memory and to allocate and free memory blocks within the heap.

The utilities are described in detail in the following chapters.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

4 Controller Utility

This chapter describes the Controller Utility and how to use it.

Overview

The Controller Utility provides a helper library to simplify usage of the low-level controller API and to add useful higher-level functionality.

The Controller Utility effectively wraps the Controller API and provides functions to maintain and query the state of the controller using an intuitive interface. There is a controller structure that contains the low level API structures and extra information used to maintain controller state between updates.

The Controller Utility provides the following:

- Management of controller button states and analog stick values.
- Management of analog stick deadzones.
- Representation of button 'pressed' and 'released' events.
- Button held 'key-repeat' functionality.

Using the Controller Utility

The `CtrlUtilData` structure is used to initialize the Controller Utility and to contain controller state at run time.

```
typedef struct CtrlUtilData{
    SceCtrlData currentCtrlData;    ///< Current frame controller data
    uint32_t   pressedButtonData;   ///< 'Pressed' button event data
    uint32_t   releasedButtonData;  ///< 'Released' button event data
    float      deadZone;            ///< Controller deadzone variable
    uint32_t   buttonRepeatDelay;   ///< Cycle delay between button repeats
    uint32_t   port;                ///< Controller port number
                                         (used internally)
    uint32_t   numBufs;             ///< The number of buffers that will
                                         receive controller data (1 to 64).
    float      leftStickXYValues[2];  ///< Left analog X, Y values
    float      rightStickXYValues[2]; ///< Right analog X, Y values
}CtrlUtilData;
```

(1) To initialize `CtrlUtilData` with the default values, pass a pointer to the structure to the `controllerUtilInitDefaults()` function. If any variables within the structure need to be modified before initializing the API, they can be modified directly before passing the structure into the `controllerUtilInit()` function.

(2) At run time during application processing, retrieve the current state of the controller by calling `controllerUtilUpdateState()`, passing a pointer to the `CtrlUtilData` as a parameter. This calls the low level controller API, reads the current state of the controller, and performs some simple tasks to help with controller state.

Button Up/Down State and Pressed/Released Events

After the `controllerUtilUpdateState()` function has been called, the controller state can be queried using the following functions:

```
controllerUtilIsButtonPressed()  // checks for button pressed events
controllerUtilIsButtonReleased() // checks for button released events
controllerUtilGetLeftStickX()    // retrieves left analog stick X value
controllerUtilGetLeftStickY()    // retrieves left analog stick Y value
controllerUtilGetRightStickX     // retrieves right analog stick X value
controllerUtilGetRightStickY     // retrieves right analog stick Y value
```

©SCEI

Document serial number: 000004892117

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

```
controllerUtilIsButtonDown()      // checks whether the button state is down
controllerUtilIsButtonUp()        // checks whether the button state is up
```

For more information about these functions and the parameters they take, see the companion *Sample Utilities Reference* documentation.

Key Repeat

- Key repeat is available on a per-button basis. To initialize key repeat, call `controllerUtilSetButtonRepeat()` with the specific button flag to set the repeat function and a Boolean flag to set or unset the repeat function.
- The key repeat functionality is automatically managed in the update function by the `controllerUtilUpdateState()` function. If the button is continuously held down, this functionality appears as true values returned repeatedly by `controllerUtilIsButtonPressed()` over a series of frames.
- You can set the key repeat delay using the `CtrlUtilData` structure, on a per-cycle basis. This delay is set to a default value in the `controllerUtilInitDefaults()` function; however, you can override this value by providing a user value at any time.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

5 Graphics Utility

This chapter describes the Graphics Utility and how to use it.

Overview

The Graphics Utility provides general purpose initialization and shutdown of the `libgxm` graphics API and the API's commonly used structures. It is designed to simplify a sample by reducing the amount of code required to set up the API, allowing the developer to focus on the main aspects of the sample.

The Graphics Utility also provides some useful and commonly used functions to clear the screen and set up default display buffers, if required.

Data Structures, Initialization, and Shutdown

There are two data structures in the Graphics Utility:

- The `GraphicsUtilConfigParams` structure, which stores data used for initializing the libgxm graphics API.
- The `GraphicsUtilContextData` structure, which stores data structures related to context handling and display services used in graphics processing.

```
typedef struct GraphicsUtilConfigParams{
    SceGxmContextParams                ctxParams;
    SceGxmShaderPatcherParams          patcherParams;
    SceGxmInitializeParams             initializeParams;
    GraphicsUtilHeapSizes              usageHeapSizes;
    uint32_t                           usageFlags;

    SceGxmColorFormat                  displayColorFormat;

    ...

    SceGxmMultisampleMode              msaaMode;
    SceGxmOutputRegisterSize           gxmOutputRegSize;
    ...
    uint32_t                           displayStrideInPixels;
};

typedef struct GraphicsUtilContextData {
    SceGxmContext                      *pContext;
    SceGxmShaderPatcher                *pShaderPatcher;

        ...
        // Display queue data
        SceGxmDisplayQueue             *pDisplayQueue;
        SceGxmColorSurface
        displaySurface[GRAPHICS_UTIL_MAX_DISPLAY_BUFFER_COUNT];

        // Depth buffer for display surface
        void        *pDisplayDepthBufferData;
        SceUID      displayDepthBufferUid;
        SceGxmDepthStencilSurface   displayDepthSurface;

        ...
        GraphicsUtilConfigParams    configParams;

} GraphicsUtilContextData;
```

Document serial number: 000004892117

000004892117

Document serial number: 000004892117

You can initialize `GraphicsUtilContextData` structures with a set of common defaults by passing pointers to the structure into the `graphicsUtilInit()` function. This function performs the main API initialization and creates instances of the `SceGxmContext` and `SceGxmShaderPatcher` structures, which are required for rendering. The memory management for different resources within the Graphics Utility is handled by the Heap Utility. The sizes for different types of memory is set using an instance of the `GraphicsUtilHeapSizes` structure.

The `graphicsUtilShutdown()` function takes a pointer to `GraphicsUtilContextData`, shuts down the `libgxm` graphics API, and de-allocates any memory that was previously allocated in the `graphicsUtilInit()` function.

If any variables require modification or have usage requirements outside of the default parameters, you can initialize `GraphicsUtilConfigParams` with defaults by calling `graphicsUtilSetDefaultParams()`, modifying the structure variables as required and passing a pointer to the structure into the `graphicsUtilInit()` function.

Runtime Functionality

The Graphics Utility provides runtime functionality to aid productivity in the form of display buffer and clear screen methods. These methods are described in the following sections.

Display Buffers

The Graphics Utility provides functionality to initialize, swap, and shut down a set of default display buffers, if the application requires them. These functions are included in the utility for general use when an application does not require more functionality than a standard swap chain, and the choice of whether the utility allocates memory for these buffers is entirely up to the user.

Initialization

Initializing the display buffers is performed by simply setting a flag within the `GraphicsUtilConfigParams` structure. By default, this flag is set when the `graphicsUtilSetDefaultParams()` function is called. After this method is called, the `graphicsUtilInit()` function reads the flag data in the structure and initializes the buffers accordingly. If you wish to create and manage your own swap chain, use the following procedure to stop the Graphics Utility creating the buffers:

(1) Initialize the `GraphicsUtilConfigParams` structure with a set of common defaults by passing a pointer to the structure into the `graphicsUtilSetDefaultParams()` function.

(2) After this function returns but prior to calling the `graphicsUtilInit()`, make any changes to the configuration parameters if required, for example the display buffer flag, vertex and fragment ring buffer sizes, output register size, and any others in the `GraphicsUtilConfigParams` structure.

To clear the display buffer flag in a `GraphicsUtilConfigParams` structure named `dataStruct`, use the following:

```
dataStruct.usageFlags &= GRAPHICS_UTIL_USE_DISPLAY_BUFFERS;
```

To change the output register size to store 64 bits per pixel, use the following:

```
dataStruct.gxmOutputRegSize = SCE_GXM_OUTPUT_REGISTER_SIZE_64BIT;
```

(3) Finally, call `graphicsUtilInit()` to finish buffer initialization.

Using the Display Buffers at Runtime

The following code demonstrates usage of the default display buffers at runtime:

```
// Access data within the GraphicsUtilContextData structure for use with libgxm.
uint32 index = graphicsData.displayBackBufferIndex;
SceGxmSyncObject  *pDisplayBufSync = graphicsData.pDisplayBufferSync[index];
```

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

```
SceGxmColorSurface *pColorSurface =
&graphicsData.displaySurface[backBufferIndex];
SceGxmDepthStencilSurface    *pDepthSurface = &displayDepthSurface;
SceGxmRenderTarget* pMainRenderTarget =
                    graphicsUtilCreateRenderTarget(graphicsData.pContext,
                    width, height, graphicsData.msaaMode);

// Used to set up the SceGxmScene structure.
sceGxmBeginScene(pContext,
                 0,
                 pMainRenderTarget,
                 NULL,
                 NULL,
                 pDisplayBufSync,
                 pColorSurface,
                 pDepthSurface);

   // Issue draw commands...

sceGxmEndScene(graphicsData.pContext, NULL, NULL );

//  Set Display Data for the callback function.
DisplayData  displayData;
displayData.pAddress = graphicsData.pDisplayBufferData[backBufferIndex];

// Call this function to cycle the display buffers
// prior to the next frame.

graphicsUtilUpdateDisplayQueue(&graphicsData, &displayData);
```

Clear Screen Functionality

In addition to the display buffer functionality, the Graphics Utility provides methods to perform a default clear-screen function.

Initialization

The clear screen functionality is initialized by default; however, if you wish to prevent this behavior, you can stop the Graphics Utility creating this data. To do this, use the same procedure as that to avoid using the default display buffers, but instead of clearing the `GRAPHICS_UTIL_USE_DISPLAY_BUFFERS` flag in the `usageFlags` variable of the `GraphicsUtilConfigParams` structure, use the `GRAPHICS_UTIL_USE_CLEAR` flag.

Clear Screen Runtime Functionality

The Graphics Utility provides a clear screen function call that takes a color parameter encoded as an unsigned integer:

```
int32_t graphicsUtilClearScreen(GraphicsUtilContextData *pData,
uint32_t color);
```

If a 64 bit surface is used it can be cleared by calling:

```
int32_t graphicsUtilClear64BitTarget(GraphicsUtilContextData *pData,
uint32_t color);
```

These functions can be called at any time for any render target as long as it remains within a `sceGxmBeginScene/sceGxmEndScene` pair.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Create Render Target Functionality

The Graphics Utility also provides a method to create a default render target on which the scene renders. The function takes in the width, size and the multisample mode for the surface which then internally allocates memory for the target and sets up a `SceGxmRenderTargetParams` parameter which is used to create the render target. You can call it as:

```
SceGxmRenderTarget* graphicsUtilCreateRenderTarget(
                                GraphicsUtilContextData *pData,
                                uint32_t width, uint32_t height,
                                SceGxmMultisampleMode msaaMode);
```

Once the render target is not required it can be destroyed using the following call:

```
int32_t graphicsUtilDestroyRenderTarget(GraphicsUtilContextData *pData,
SceGxmRenderTarget  *pRenderTarget);
```

Capture Screen Functionality

The Graphics Utility also provides a function to easily capture the main display and save it as a `.bmp` file. The function takes the file path as input and saves the current display buffer into the specified file name, as follows:

```
int32_t graphicsUtilSaveDisplayAsBmp(GraphicsUtilContextData *pData, const
char *path);
```

You can link the function to a controller button and then call the function at run time by pressing the button when you wish to capture the screen.

Shutdown

The library handles all resources allocated by the Graphics Utility for use with the default display buffer and clear screen methods. All resources are released automatically by the utility when the user calls the `graphicsUtilShutdown()` function.

Allocating and De-allocating Graphics Memory

The Graphics Utility provides functions to allocate and free memory for the GPU. These functions are `graphicsUtilAlloc()` and `graphicsUtilFree()`, respectively. You can find complete prototypes for these functions in the `<graphics_utility.h>` header file.

The following code provides an example of how to use these functions:

```
int32_t vertexAllocSize = numVerts*sizeof(MyVertex);
int32_t alignment = 4;

// Allocation
MyVertex* pWaterVertices = (MyVertex*)graphicsUtilAlloc(pContextData,
                                             memoryBlockHeapType,
                                             vertexAllocMemSize,
                                             alignment);
// ...
```

where `memoryBlockHeapType` is one of the `GRAPHICS_UTIL_HEAP_TYPE_*` heaps types.

```
// De-allocation
graphicsUtilFree(pContextData, pWaterVertices);
```

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

6 Timer Utility

This chapter describes the Timer Utility and how to use it.

Overview

The Timer Utility provides simple multi-resolution timing functionality, wrapping the low level system timer. It provides methods to start, stop, reset, and update a timer, and functions for retrieving the total time since start and the time delta elapsed since the last update. The timer supports retrieval of these values both as single or double precision float.

Using the Timer Utility

General use cases of the Timer Utility are as follows:

```
// Initialization
TimerUtilData timer;
timerUtilInit(&timer);

...

// Update function

timerUtilUpdate(&timer);

...

// Value retrieval
float totalTime, timeDelta;

// Get total time
totalTime = timerUtilGetTotalTimeFloat(&timer,
                                       TIMER_UTIL_RESOLUTION_SECONDS);

// foo(totalTime);


// Get time since last update
timeDelta = timerUtilGetTimeDeltaFloat(&timer,
                                       TIMER_UTIL_RESOLUTION_MILLISECONDS);

// bar(timeDelta);
```

000004892117

Document serial number: 000004892117

©SCEI

Document serial number: 000004892117

Document serial number: 000004892117

7 MeshLoader Utility

This chapter describes the MeshLoader Utility and how to use it.

Overview

The MeshLoader Utility has been provided to load simple COLLADA files and make available the vertex and index data for processing and rendering the meshes. The utility is designed to simplify the loading process of a mesh.

The user must pass in the COLLADA mesh file; this file is internally parsed and data is converted to an interleaved buffer based on attributes required by, and made available to, the user for further processing. Currently loading of triangles and polygons type geometry elements is allowed. Loading multiple meshes is also supported.

Data Structure, Initialization, and Shutdown

The main data structure utilized in the MeshLoader Utility is the `MeshLoaderUtilData` structure. This structure contains all the required data and structures to initialize and set up the mesh vertex and index data, the texture to be mapped, and bounding box information.

```
typedef struct MeshLoaderUtilData {
    int           meshHandle;
    int           meshSize;
    int           numVerts;
    float         boundingSphereRadius;
    float         boundingOrigin[3];
    float         boundingBoxMin[3];
    float         boundingBoxMax[3];
    float         *pMeshVertices;
    uint16_t      *pMeshIndices;

    //Texture Data
    char              textureName[MAX_STRING_LENGTH];
    uint8_t           *pMeshTextureData;
    SceGxmTexture     meshTexture;

} MeshLoaderUtilData;
```

This structure is initially filled in with default values by using the `meshLoaderUtilInitDefaults()` function. When the COLLADA mesh file is passed into the `meshLoaderUtilInit()` function, the file is parsed and the specific data is updated and stored for usage. The parsed data supports the 8 source streams available in the COLLADA mesh, which are represented by the following enumeration:

```
typedef enum MeshLoaderUtilSourceMap
{
    SOURCEMAPPING_VERTEX,      ///< Represents Vertex position type stream.
     SOURCEMAPPING_NORMAL,     ///< Represents Normal type stream.
     SOURCEMAPPING_TEXCOORD,   ///< Represents Texture coordinate type stream.
     SOURCEMAPPING_COLOR,      ///< Represents Color type stream.
     SOURCEMAPPING_TANGENT,    ///< Represents Tangent stream.
     SOURCEMAPPING_BINORMAL,   ///< Represents Binormal stream.
     SOURCEMAPPING_TEXTANGENT, ///< Represents texture space tangent stream.
     SOURCEMAPPING_TEXBINORMAL ///< Represents texture space binormal stream.

} MeshLoaderUtilSourceMap;
```

The next step is to create an interleaved data stream dependent on the developer attribute requirements, which can be checked for availability using the `meshLoaderUtilCheckAttributeAvailability()`

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

function. The available streams are then stored in an interleaved vertex data buffer `pMeshVertices`; the index data array `pMeshIndices` is also filled when the developer calls `meshLoaderUtilSetDataBuffer()`. If a texture is attached to the mesh, the pointer to the texture data `pMeshTextureData` is also stored.

There are no runtime functions in the MeshLoader Utility, and handling of all the mesh data buffers is expected to be done by the user. Users must write their own functions for processing the mesh, attaching shaders, and rendering. The only other function available to the user is `meshLoaderUtilShutdown()`, which de-allocates any memory that has been previously allocated in earlier `meshLoaderUtil` functions.

Sample Code

For a simple demonstration of how to set up and use the MeshLoader Utility, see the `tutorial_meshloader_utility.cpp` file in:

```
SDK/target/samples/sample_code/system/tutorial_sample_utilities/
meshloader_utility
```

Note: The MeshLoader Utility is currently usable with C++ applications only. It provides limited functionalities and supports geometries and images libraries only. Materials, Animation, Physics and Shader data in COLLADA files are not currently supported.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

8 Font Utility

This chapter descries the Font Utility and how to use it.

Overview

The Font Utility is a library used to create an image for which fonts are simply rendered using libpgf.

Strings specified by the game application will be rendered to the image buffer.

How to Use the Font Utility

General usages of the Font Utility are as follows:

```
// Initialization of the Font utility
FontUtilFontProperty  fontProperty;
fontUtilInit( &fontProperty,
            NULL, // Uses the internal fonts when NULL is specified
            SCE_FONT_LANGUAGE_J,
            SCE_FONT_FILEBASEDSTREAM,
            NULL, // Omitted user data specification
            cb_alloc_func,
            cb_realloc_func,
            cb_free_func );


//Creation of image buffer info
FontUtilImageBuffer  fontImage;
fontUtilCreateImageBuffer( &fontImage, fontMem,
                        width, height, byteStride,
                        FONTUTIL_IMAGEBUFFER_TYPE_GRAY );


// Specification of font size
fontUtilSetFontSize( &fontProperty, 12.0f );


// Specification of font color
// At the time of execution of fontUtilCreateImageBuffer
// Enabled only when FONTUTIL_IMAGEBUFFER_TYPE_COLOR_ABGR is specified
fontUtilSetFontColor( &fontProperty, 0xff00A5ff );

// Acquisition of frame info
fontUtilGetPrintSizeUtf8( &fontProperty, &fontImage, stringUtf8,
                        textLength, &fontWriteWidth, &fontWriteHeight );

// Rendering of strings
fontUtilPrintUtf8( &fontProperty, &fontImage, stringUtf8, textLength, width,
height );

...

// Termination
// Release of image buffer info
fontUtilFreeImageBuffer( &fontImage );

// Disabling the Font utility
fontUtilShutdown( &fontProperty );
```

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Sample Code

A simple demonstration sample of how to setup and use the Font Utility can be found in:

```
SDK/target/samples/sample_code/system/tutorial_sample_utilities/
font_utility
```

000004892117

Document serial number: 000004892117

©SCEI

Document serial number: 000004892117

Document serial number: 000004892117

9 Sound Utility

This chapter describes the Sound Utility and how to use it.

Overview

The Sound Utility is a library to allow the game application to simply play a sound using libngs and provides the following features.

- Voice playback of VAG, Wav, or ATRAC9™ files
- Streaming playback of a BGM
- Envelope and reverb buss settings
- Playback from BGM port

Features of Audio Codec Supported by Sound Utility

Sound Utility can play VAG, Wav and ATRAC9™ data formats. Please select the play data format taking into account the following features:

VAG

VAG is a format which enables high-speed decode. It is suitable for playing effect sounds. In the shooting game sample, which complies with TRC, the VAG format is used for effect sounds.

Wav

Wav is an uncompressed audio data format. Although it provides the highest quality sound, because it is uncompressed, streaming play processing will be overloaded with file I/O accesses. Use Wav format data as little as possible for playing streaming sounds in Sound Utility.

ATRAC9™

The ATRAC9™ format is a game-optimized, high-compressive audio format. With high compressibility, high-quality sound, and low CPU load during playing, it is suitable for use of both effective sound and BGM. In the shooting game sample, which complies with TRC, the ATRAC9™ format is used for BGM.

How to Use the Sound Utility

This section describes a simplified usage of the Sound Utility.

Initialization

```
//E Set the initialization parameters of the voice
SoundUtilVoiceInitParams    voiceParams;
voiceParams.monoVoiceNum    = MONO_VOICE_NUM;    //E Maximum number of monaural
                                                    voices
voiceParams.stereoVoiceNum  = STEREO_VOICE_NUM; //E Maximum number of stereo
                                                    voices

//E Set the initialization parameters of the buss
SoundUtilBussInitParams     bussParams;
bussParams.mixerBussNum     = MIXER_BUSS_NUM;  //E Maximum number of mixer
                                                   busses
bussParams.reverbBussNum    = REVERB_BUSS_NUM; //E Maximum number of reverb
                                                   busses
```

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

```
//E Set the initialization parameters of the update thread
SoundUtilUpdateThreadInfo    threadInfo;
threadInfo.priority          = SCE_KERNEL_HIGHEST_PRIORITY_USER;
threadInfo.stackSize         = 128*1024;
threadInfo.userFunction      = soundUtilUpdateThread;
threadInfo.userAttr          = (void*)&resources;
threadInfo.cpuAffinityMask   = SCE_KERNEL_CPU_MASK_USER_0;

//E Set parameters of the sound to be used
SoundUtilSoundInfo           soundInfo;
Int                          streamId;
SoundUtilStreamingInfo       streamInfo;

//E Initialization
SoundUtilResources           resources;
ret = soundUtilInit(&resources, &voiceParams, &bussParams, &threadInfo);
```

Bus Setting

```
//E Open the reverb bus
Int      reverbID;
reverbID = soundUtilBussCreateReverb(&resources, 2);
```

SFX Playback

```
//E Load data
ret = soundUtilLoadData(&resources, filepath, SOUND_UTIL_SOUND_TYPE_ADPCM,
&soundInfo);


//E Open a voice
m_sfxId = soundUtilVoiceOpen(&resources, SOUND_UTIL_DATATYPE_SOUND, &SoundInfo,
NULL);

//E Set an envelope
SoundUtilEnvelopeInfo        envelopeInfo;
envelopeInfo.envelopePoints[0].uMsecsToNextPoint = 50;
envelopeInfo.envelopePoints[0].fAmplitude = 0.0f;
envelopeInfo.envelopePoints[0].eCurveType = SOUND_UTIL_ENVELOPE_LINEAR;
envelopeInfo.envelopePoints[1].uMsecsToNextPoint = 50;
envelopeInfo.envelopePoints[1].fAmplitude = 2.0f;
envelopeInfo.envelopePoints[1].eCurveType = SOUND_UTIL_ENVELOPE_CURVED;
envelopeInfo.envelopePoints[2].uMsecsToNextPoint = 50;
envelopeInfo.envelopePoints[2].fAmplitude = 0.5f;
envelopeInfo.envelopePoints[2].eCurveType = SOUND_UTIL_ENVELOPE_LINEAR;
envelopeInfo.envelopePoints[2].uMsecsToNextPoint = 50;
envelopeInfo.envelopePoints[2].fAmplitude = 1.0f;
envelopeInfo.envelopePoints[2].eCurveType = SCE_NGS_ENVELOPE_CURVED;

envelopeInfo.uNumPoints = 4;
envelopeInfo.uLoopStart = 0;
envelopeInfo.nLoopEnd = 3;
envelopeInfo.uReleaseMsecs = 5000;

soundUtilVoiceSetEnvelope(&resources, m_BGMID, &envelopeInfo);

//E Connect to the buss
soundUtilVoiceSetBuss(&resources, m_sfxId, 0, m_reverbID, NULL);

//E Playback
soundUtilVoicePlay(&resources, m_sfxId, SOUND_UTIL_VOICE_AUTO_CLOSE);
```

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

BGM Playback

```
//E Process a wave file
SoundUtilWavInfo wavInfo;
soundUtilGetWavInfo(&resources, wav_filepath, &wavInfo);
m_gameBgmInfo.nNumChannels = wavInfo.numChannels;
m_gameBgmInfo.nSampleRate  = wavInfo.sampleRate;
m_gameBgmInfo.nType        = SOUND_UTIL_SOUND_TYPE_WAV;
m_gameBgmInfo.filePath     = at9_filepath;
m_gameBgmId = soundUtilVoiceOpen(&resources,
SOUND_UTIL_DATATYPE_STREAMING_SOUND, &m_gameBgmInfo, NULL);

//E Playback
soundUtilVoicePlay(&resources, m_gameBgmId, SOUND_UTIL_VOICE_NONCLOSE);

//E Update
soundUtilUpdateThread(void* buffer, void* userAttr)
{
   soundUtilProcess((SoundUtilResources*)userAttr, buffer);
}
```

Termination Processing

```
// Termination
soundUtilVoiceClose(&resources, m_gameBgmId);
soundUtilUnloadData(&resources, &m_SoundInfo);
soundUtilExit(&resources);
```

Sample Code

For a simple demonstration of how to use the Sound Utility, see `sound_manager.h` and `sound_manager.cpp` files in:

```
SDK/target/samples/sample_code/system/tutorial_shooting_game_trc_compliant
```

000004892117

Document serial number: 000004892117

©SCEI

Document serial number: 000004892117

Document serial number: 000004892117

10 Heap Utility

This chapter describes the Heap Utility and how to use it.

Overview

The Heap Utility has been provided to initialize a memory heap to which free memory blocks can be added and allocated as required. When an allocated memory block is no longer being used, it is freed back to the heap, and the heap can be destroyed when it is no longer required.

Data Structure, Initialization, and Shutdown

The main data structure utilized in the Heap Utility is the `HeapUtilMemBlock` structure. This structure contains all the required information regarding the memory block in the heap including the size and type of the memory.

```
struct HeapUtilMemBlock {
    HeapUtilMemBlock *next;
    int32_t      type;
    uintptr_t    base;
    uint32_t     offset;
    uint32_t     size;
};
```

The other structure that is used is the `HeapUtilContext` structure, which contains the list of allocated and free memory blocks in the heap.

```
struct HeapUtilContext {
    HeapUtilMemBlock *allocList;
    HeapUtilMemBlock *freeList
};
```

The first step is to initialize a `HeapUtilContext` structure by calling `heapUtilInitialize()`. This creates an empty heap. Blocks of memory can then be added to the context's free list by calling `heapUtilExtend()` with specific parameters (type, address, offset and size) passed to it.

Memory can then be allocated from these blocks as required by calling `heapUtilAlloc()`. Depending on the type of allocation required, the memory blocks are searched, and memory of the required size and alignment is allocated in the heap. If memory needs to be allocated with an offset, `heapUtilAllocWithOffset()` can be called with an additional offset in order to track the block. These allocated blocks of memory are tracked from the heap context's `allocList`.

Once the allocated memory is not required, it can be freed back to the heap by calling `heapUtilFree()` and passing in the base address of the memory to be freed. If the memory blocks in this heap are no more used, the heap context can be destroyed by calling `heapUtilTerminate()`.

Sample Code

Sample code is supplied that provides a demonstration of how to use the Heap Utility to initialise a memory heap, and how to use it to extend and allocate memory blocks in the heap. It can be seen in the Graphics and MeshLoader Utilities in `graphics_utility.c` and `loader_utility.cpp` files in:

```
SDK/target/samples/sample_code/common/source/sample_utilities
```

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

11 Debug Menu Utility

This chapter describes the Debug Menu Utility and how to use it.

Overview

The Debug Menu Utility provides helper functions to output a text-based on-screen debug menu, as well as to modify the menu item values and toggle features and effects at run time. This utility uses libdbgfont to set up and output the debug text for the menu.

The Debug Menu Utility provides the following:

- Initialization of Text Debug Menu Utility with default settings (libdbgfont dependent).
- User-defined setup for text output on-screen. The user can define:
  - Options for outputting labels, floats, integers, and bool type menu items.
  - An option for setting up a menu item as changeable or non-changeable.
  - An option for setting up callback function-based menu items.
- Traversing changeable menu items.
- Changing values of selected menu items between user-specified ranges at run time.
- Enabling/disabling the state of a selected menu item.

000004892117

Using the Debug Menu Utility

The Debug Menu Utility is a self-managed library that manages all memory allocations. It works on an object handle-based system as follows:

(1) To create a menu, you call the `debugMenuUtilCreateMenu()` function. If successful, this function returns an integer handle to the newly created menu item.

(2) To initialize a menu and related values, you call `debugMenuUtilInitDefaults()`, passing color and text attribute information with the menu handle as parameters.

(3) After a menu has been successfully initialized, you can add menu items to it using `debugMenuUtilAddItem`, passing in relevant menu item data with the menu handle.

Currently, the following menu item value types are defined:

- `DEBUG_MENU_UTIL_ITEM_TYPE_LABEL`
- `DEBUG_MENU_UTIL_ITEM_TYPE_FLOAT`
- `DEBUG_MENU_UTIL_ITEM_TYPE_INTEGER`
- `DEBUG_MENU_UTIL_ITEM_TYPE_BOOL`
- `DEBUG_MENU_UTIL_ITEM_TYPE_CALLBACK`

Data for each type is managed by the `DebugMenuUtilItemValueX` structures shown in Table 1.

Note: Label types are managed slightly differently. For label types, you first create a label menu item, and then add individual string value pairs that will be selected when modifying the menu. For more information, see Label Type Menu Items.

Table 1 Menu Item Structures

Structure	Date Type of Menu Items	Values
`DebugMenuUtilItemValueFloat`	For items showing float type values	The menu item's initial values, min-max range, and step values can be set.

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Structure	Date Type of Menu Items	Values
`DebugMenuUtilItemValueInt`	For items showing integer type values	The menu item's initial values, min-max range, and step values can be set.
`DebugMenuUtilItemValueBool`	For items showing bool type values	The menu item's current value (true or false) can be set.
`DebugMenuUtilItemValueCallback`	For items with a set of user defined callback functions	The user can define up to two callback functions per menu item that will be executed by the Debug Menu Utility.

To add a menu item, call the `debugMenuUtilAddItem()` function, as follows:

```
debugMenuUtilAddItem
(
    int32_t menuHandle,
    const char *itemLabel,
    void *pItemValue,
    MenuUtilItemType itemType,
    bool changeProperty
)
```

This function returns the menu item handle for a newly created menu item. This handle is required when adding label type menu items (see Label Type Menu Items below).

- `itemLabel` is the label you wish to give to this menu item.
- `pItemValue` pointer is a pointer to the relevant `DebugMenuUtilItemValueX` structure depending upon which `DebugMenuUtilItemType` is passed in as the fourth parameter.
- `changeProperty` is a Boolean to determine whether this menu item is modifiable or not.

Initializing and Using the Menu

To initialize and use the menu:

(1) After the items have been registered, initialize the cursor position to the first changeable item to be updated using `debugMenuUtilInitCursorPosition()`.

(2) To allow interaction with the menu items, you can set buttons for traversing up the menu, down the menu, and to increment and decrement the value of the selected item. Depending on the action you wish to achieve, call the following functions:

```
debugMenuUtilSelectUpItem()    //to move to the previous changeable menu item
debugMenuUtilSelectDownItem() //to move to the next changeable menu item
debugMenuUtilIncrementItemValue()  //to increment/toggle the value/status
debugMenuUtilDecrementItemValue()  //to decrement/toggle the value/status
```

(3) After all the menu items have been registered and their values updated, they can be flushed out on-screen using `debugMenuUtilRenderMenu()`. This function traverses through all the registered items and outputs each menu item on-screen using its settings and data values. A menu item that is selected for change is highlighted with a preset color and preceded by ">>".

(4) When finished, remember to call `debugMenuUtilShutdown()` to shut down the menu system and free its allocations.

000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Document serial number: 000004892117

Label Type Menu Items

Label menu items, as mentioned previously, require some extra initialization to set up. A label menu item displays a set of user-defined labels and pairs an integer value with each label that can be selected by the user.

To set up a label menu item:

(1) Create a label menu item using the `debugMenuUtilAddItem()` function as for the other types, passing in `DEBUG_MENU_UTIL_ITEM_TYPE_LABEL` as the `DebugMenuUtilItemType` parameter.

In this case, there is no `DebugMenuUtilItemValueLabel` type to pass in as the `pItemValue` pointer, as there is for other types. Instead, you pass in a pointer to an integer value that you want the menu to modify when the user selects different labels within the menu item. The handle returned by this function is then used to refer to the label menu item when you want to add string/value pairs to it.

(2) After the label item has been created, you can add string/value pairs to the label item by calling `debugMenuUtilAddStringValueToItem()`, passing in the menu handle, the label handle, and the string and integer value pair you wish to add to the label.

Sample Code

For a simple demonstration of how to set up the Debug Menu Utility, see the `tutorial_sample_utilities.c` file in:

```
SDK/target/samples/sample_code/system/tutorial_sample_utilities/simple
```

000004892117

Document serial number: 000004892117

Document serial number: 000004892117