Cross-Play Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

About This Document

This document explains the design and the techniques used in the “Cross-Play demo” sample code. The sample code is located in the PlayStation®Vita SDK package at:

```
%SCE_PSP2_SAMPLE_DIR%/sample_code/network/demo_crossplay
```

Typographic Conventions

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

Hints

A GUI shortcut or other useful tip for gaining maximum use from the software is presented as a “Hint” surrounded by a box. For example:

Hint: This hint provides a shortcut or tip.

Notes

Additional advice or related information is presented as a “Note” surrounded by a box. For example:

Note: This note provides additional information.

Text

- 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:

```
m_targetBufferData[idx]);     // pointer to the surface data
```

Related Documentation

- *NP Toolkit Library Overview* – provides details about using NP Toolkit to handle PSN℠ services.
- *Network Overview* – provides details about using network functions.
- *NP Signaling Library Overview* – provides details about the signaling process.

1 Introduction

The Cross-Play demo sample demonstrates how to create a simple "shoot 'em up" (SEU) game exploiting the Cross-Play features of PlayStation®Vita and PlayStation®3.

"Cross-Play" is an umbrella term indicating both competitive/cooperative play and a cross-platform experience between PlayStation®Vita and PlayStation®3 systems. The Cross-Play demo allows PlayStation®3 and PlayStation®Vita users to share a seamless gaming experience through many different PSN℠ services.

Both PlayStation®3 and PlayStation®Vita SDKs contain a rich set of platform-specific network libraries to interface titles with the PSN℠ features. These libraries are quite low-level so a high-level cross-platform library called the *NP Toolkit* is provided with the SDK for convenience. This sample addresses important aspects of the development of a cross-platform title using the NP Toolkit, including portability, networking, endianness handling and assets management.

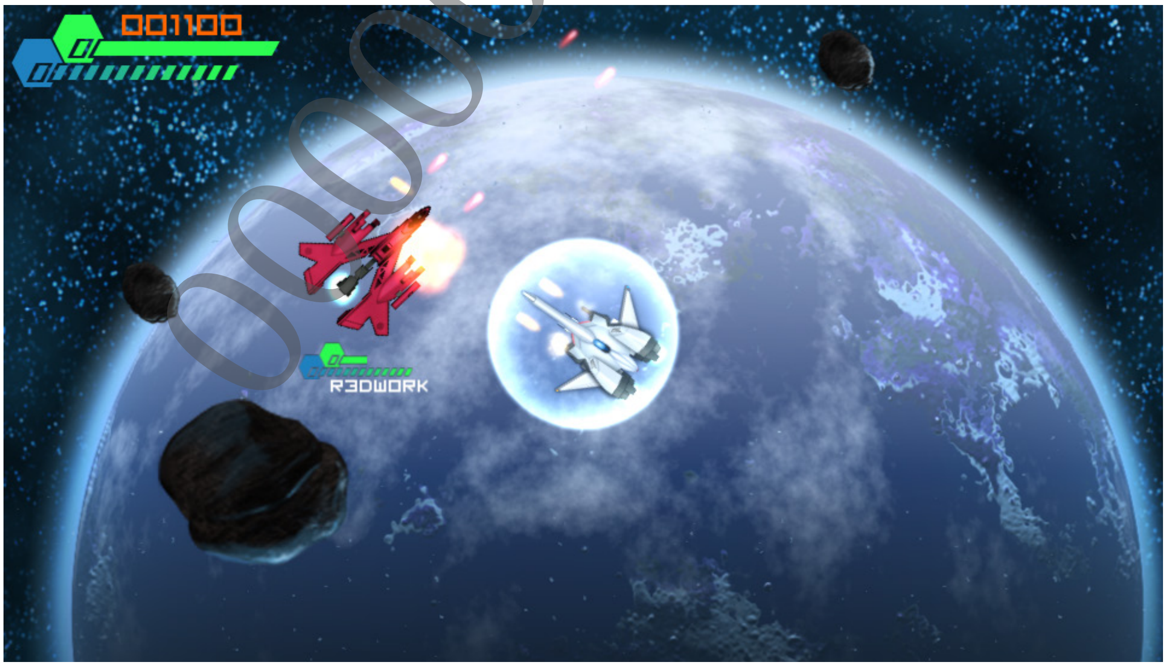
When the Cross-Play application is started, all the PSN℠ services are initialized using the NP Toolkit. The following operations are performed:

- The user is logged into PSN℠.
- The NP ID of the user is retrieved.
- The trophy set is retrieved from the application's package.
- The unlock state of each level is retrieved by querying the relative TUS variable on PSN℠.

PlayStation®3 and PlayStation®Vita have different hardware and APIs; therefore, in order to simplify development, the sample is split into two projects:

- The *Games Toolkit* (*Gt*): a library which handles the low-level, platform-specific aspects of the development.
- A cross-platform application, built on top of the Games Toolkit and NP Toolkit libraries, which implements the game logic for the SEU Cross-Play demo.

Figure 1 The Cross-Play Demo



PlayStation®3 Version

A PlayStation®3 solution for the Cross-Play demo is provided with the PlayStation®Vita SDK. It is located at:

```
%SCE_PSP2_SAMPLE_DIR%/sample_code/network/demo_crossplay/project/ps3
```

The following software is required to build the Cross-Play demo on PlayStation®3:

- PlayStation®3 SDK 430 (or newer)
- NP Toolkit 4.30 (or newer)

The NP Toolkit library for PlayStation®3 can be downloaded using the SDK Manager or from the Network Platform SDK page on DevNet: https://ps3.scedev.net/projects/np_sdk.

2 Design of the Cross-Play Demo

This chapter introduces the game design of the Cross-Play demo.

Overview

The Cross-Play demo is a simple multi-player "shoot 'em up" (SEU) game in which each player controls a space ship and has to defeat the other players by shooting at them.

Sessions

The game is arranged in *sessions* in which up to four players can play together. A session lasts until only one player survives. Players can create a new session or join an existing one.

Levels

The game has three different levels in which sessions can be created. For simplicity, there are no differences between the levels in terms of gameplay. When the player plays the game for the first time, only the first level is available. Winning a session in the first level unlocks the second level, and winning a session in the second level unlocks the third level.

Gameplay

The goal of the game is to defeat all the other players in the session.

At the beginning of each session, each player is spawned at a random position in the level. Players can move across the outer space, rotate and shoot bullets which can damage the enemy players.

Each player has an energy value which decreases each time the ship collides with a bullet shot by an enemy. When a player's energy value reaches zero, his/her ship is destroyed and the player cannot play any more in that session.

Players can use a shield which protects them from the enemy bullets. The shield lasts only a few seconds but can be recharged by not using it for a certain amount of time.

Score

Players have a score which is calculated according to the number of their bullets that hit an enemy.

Each time a bullet hits an enemy 50 points are added to the score of the player who spawned the bullet. If a player hits an enemy quickly enough the score is multiplied by a factor which increases for each hit.

When a session is completed, the scores of all players are updated to the PSN℠ scoreboard for the current level.

Trophies

When a session is completed, trophies are analyzed and unlocked according to the status of the player. The following trophies can be achieved:

- Lazy boy (bronze trophy) – awarded to players who lose a Cross-Play session without hitting anyone.
- Winner (silver trophy) – awarded to the winner of a Cross-Play session.
- Invincible (gold trophy) – awarded to a player who wins a Cross-Play session without suffering any damage.

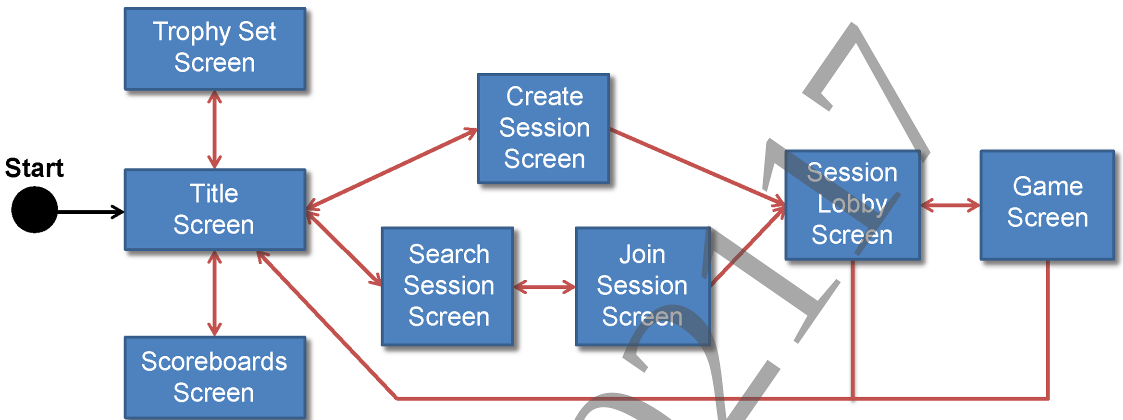
3 Screens

This chapter describes the logical phases of the game (also called *screens* for simplicity) and their transition flow.

Screen Transitions Flow

Figure 2 shows the screens which compose the Cross-Play demo and their dependencies.

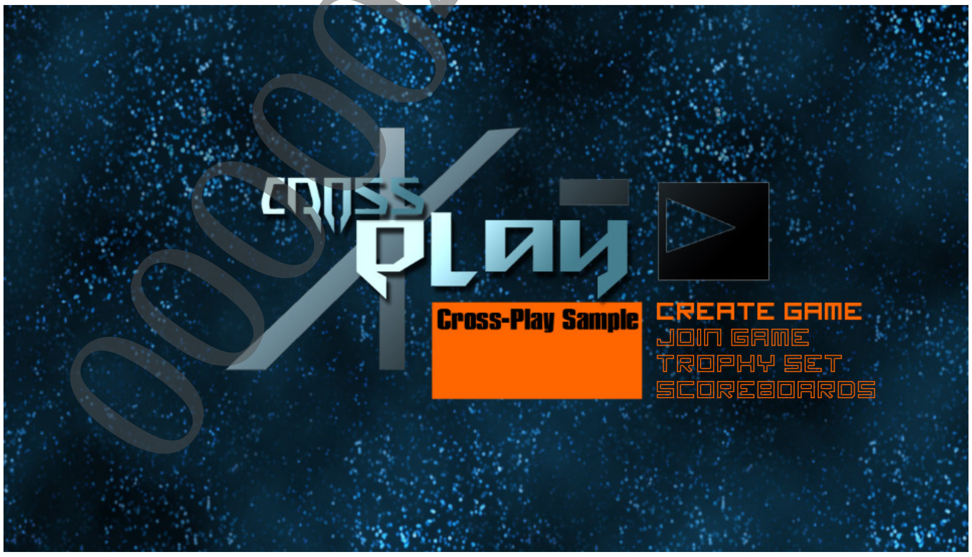
Figure 2 Screen Transitions Flow of the Cross-Play Demo



Title Screen

The **Title** screen is the main screen of the Cross-Play demo. It allows players to access all the other screens of the game quickly.

Figure 3 The Title Screen



Functionality

To create a new Cross-Play gaming session, select **CREATE GAME**.

To join an existing gaming session, select **JOIN GAME**.

To view the available trophies and the game progress, select **TROPHY SET**.

To access the scoreboards, select **SCOREBOARDS**.

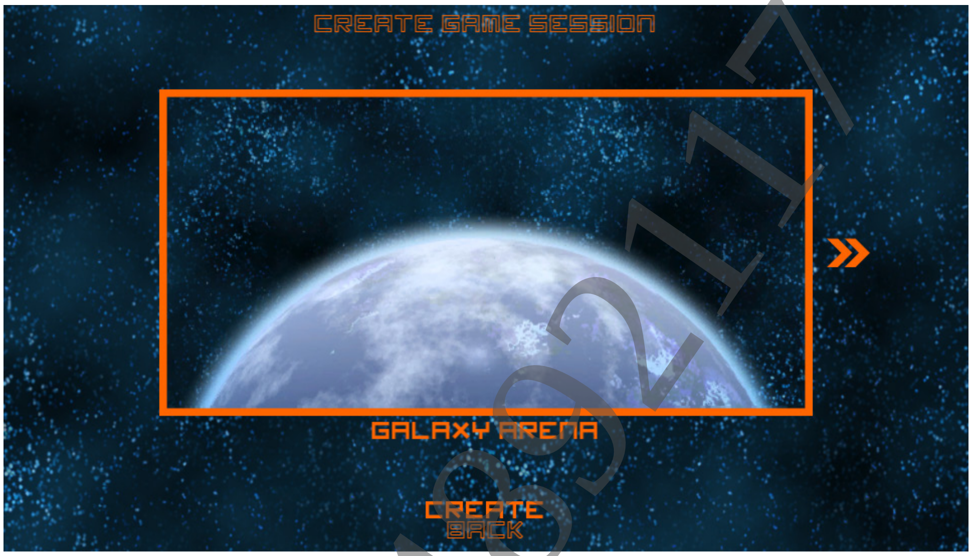
Table 1 Input Controls for the Title Screen

Input Control	Action
D-Pad up	Highlights the previous menu option
D-Pad down	Highlights the next menu option
X button	Selects the current menu option

Create Session Screen

The **Create Session** screen is displayed when a player selects **CREATE GAME** in the **Title** screen. The **Create Session** screen allows the player to select a level and create a new Cross-Play session.

Figure 4 The Create Session Screen



Functionality

To create a new session:

(1) Use the left and right D-Pad buttons to select the level in which the session should be set.

(2) Select the **CREATE** menu option. The **Session Lobby** screen is displayed.

Note: Levels can be selected only after they have been unlocked; that is, if the previous level has been won. For more information, see the Levels section.

Table 2 Input Controls for the Create Session Screen

Input Control	Action
D-Pad left	Selects the previous level
D-Pad right	Selects the next level
D-Pad up	Highlights the previous menu option
D-Pad down	Highlights the next menu option
X button	Selects the current menu option
O button	Goes back to the previous screen

NP Toolkit Usage

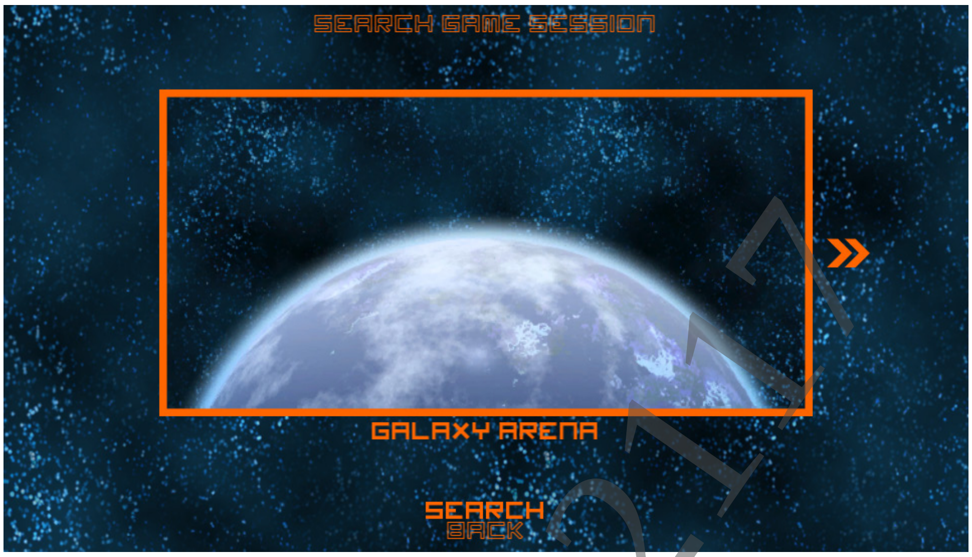
The **Create Session** screen uses the NP Toolkit to initialize a public matching session. The selected level is set as a searchable attribute of the session. The NP Toolkit is also responsible for initializing the session signaling module, which is responsible for retrieving the network addresses of the session members.

Each matching session has a unique name. The Cross-Play demo uses the NP ID of the session owner as a name, so users can easily identify sessions.

Search Session Screen

The **Search Session** screen is displayed when a player selects **JOIN GAME** in the **Title** screen. The **Search Session** screen allows the player to search for an existing Cross-Play session set in the selected level.

Figure 5 The Search Session Screen



Functionality

To join a game:

(1) Select the session level that you wish to join using the left and right D-Pad buttons.

(2) Select the **SEARCH** menu option. If sessions are found in the selected level, the game moves to the **Join Session** screen. If no sessions are available for the selected level, an error message is displayed and the game moves back to the **Title** screen.

Note: Levels can be selected only after they have been unlocked; that is, if the previous level has been won. For more information, see Levels.

Table 3 Input Controls for the Search Session Screen

Input Control	Action
D-Pad left	Selects the previous level
D-Pad right	Selects the next level
D-Pad up	Highlights the previous menu option
D-Pad down	Highlights the next menu option
X button	Selects the current menu option
O button	Goes back to the previous screen

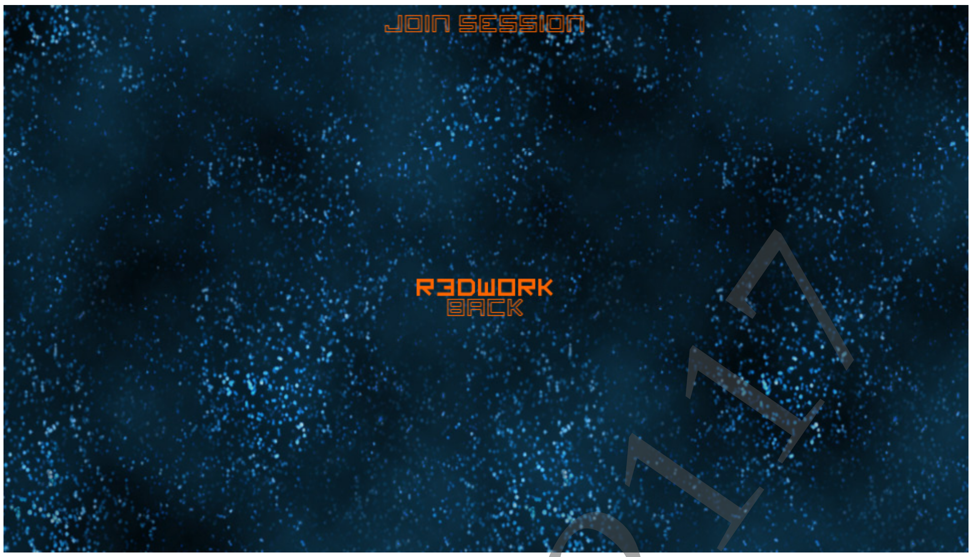
NP Toolkit Usage

The **Search Session** screen uses the NP Toolkit to search for a public matching session. The search request contains the currently selected level as a searchable attribute of the session.

Join Session Screen

The **Join Session** Screen allows the player to join an existing Cross-Play session which matches the level criterion specified in the **Search Session** screen.

Figure 6 The Join Session Screen



Functionality

To join a session, select a session from the list. The **Session Lobby** screen is displayed.

Table 4 Input Controls for the Join Session Screen

Input Control	Action
D-Pad up	Highlights the previous menu option
D-Pad down	Highlights the next menu option
X button	Selects the current menu option
O button	Goes back to the previous screen

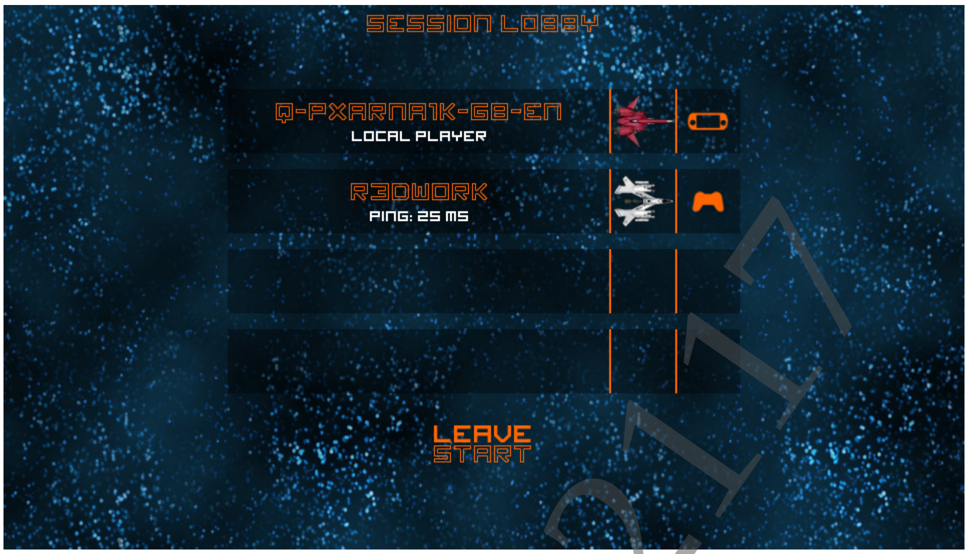
NP Toolkit Usage

This screen uses the NP Toolkit to request that the user be allowed to join the selected session.

Session Lobby Screen

The **Session Lobby** screen is displayed when a player has joined or created a session. It allows the player to choose a spaceship, view information about the other members of the session, and either start the session (session owner only), start playing (all other players) or leave the session (any player).

Figure 7 The Session Lobby Screen



Layout

Each session slot contains the following information about a session member:

- NP ID
- Ping time in milliseconds
- Platform
- Currently selected ship

The first slot is occupied by the session owner; that is, the player who acts as the central coordinator during the game phase.

Functionality

To start the session (session owner only), select the **START** menu option.

To start playing (all non-owner players), select the **START** menu option.

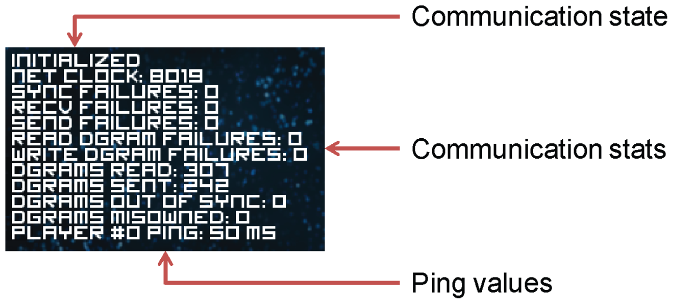
To leave the session (any player), select the **LEAVE** menu option. If the session owner leaves the session, the session is terminated.

Table 5 Input Controls for the Session Lobby Screen

Input Control	Action
D-Pad left	Selects the previous ship
D-Pad right	Selects the next ship
D-Pad up	Highlights the previous menu option
D-Pad down	Highlights the next menu option
X button	Selects the current menu option
O button	Goes back to the previous screen
L button	Toggles the communication statistics panel
R button	Resets the communication statistics

A debug panel containing the communication statistics can be toggled using the L button. Figure 8 shows the debug panel.

Figure 8 Debug Panel



```
INITIALIZED
NET CLOCK: 8019
SYNC FAILURES: 0
RECV FAILURES: 0
SEND FAILURES: 0
READ DGRAM FAILURES: 0
WRITE DGRAM FAILURES: 0
DGRAMS READ: 307
DGRAMS SENT: 242
DGRAMS OUT OF SYNC: 0
DGRAMS MISOWNED: 0
PLAYER #0 PING: 50 MS
```

NP Toolkit Usage

The **Session Lobby** screen uses the NP Toolkit to retrieve the current state of the matching session. The matching interface of the NP Toolkit allows the Cross-Play demo to access the NP ID and network addresses of all the session members through a process called *signaling*.

For more information, see Matching in Chapter 5, NP Toolkit Usage in the Cross-Play Demo.

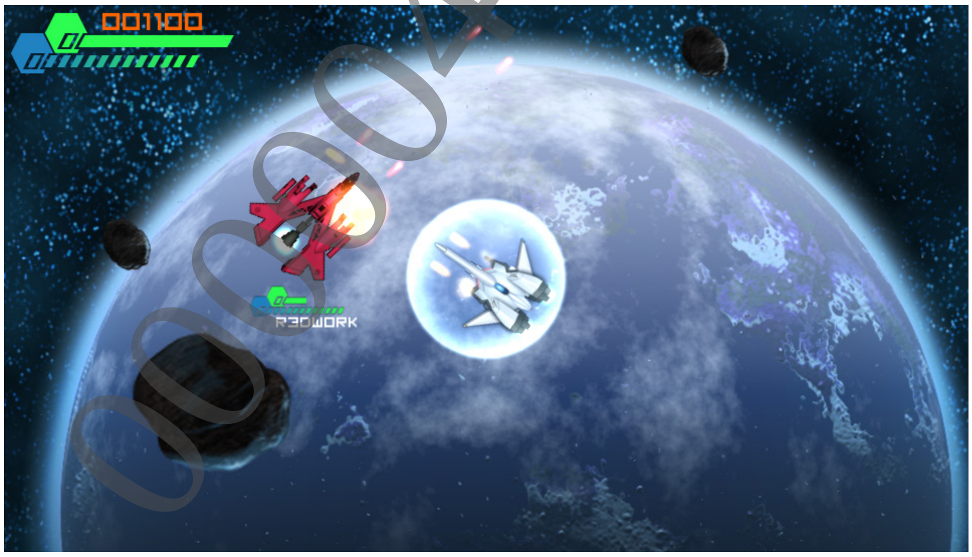
The ping time between the local player and the other session members is also retrieved using the features of the session signaling API; however, this operation is not currently wrapped by the NP Toolkit.

When the signaling is complete, a real-time network communication between the session owner and all the other session members is established. For more information, see Chapter 6, Network Communication.

Game Screen

The **Game** screen implements the simple game mechanics defined in Gameplay in Chapter 2, Design of the Cross-Play Demo.

Figure 9 The Game Screen



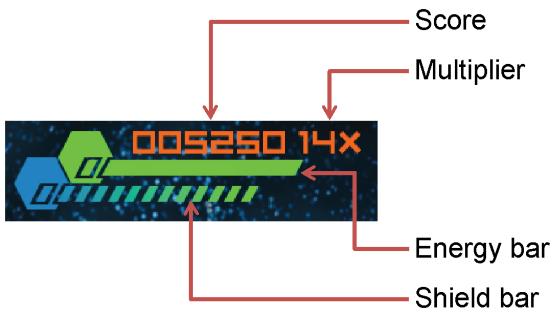
Layout

The screen is divided as follows:

- The level area contains the level itself and the ships of the session members.
- The HUD area is located at the top-left corner and contains a graphical overview of the player status.

Figure 10 shows further information about the HUD elements.

Figure 10 The HUD Elements in the Game Screen



The level area contains also a small HUD for each active enemy player. These HUDs follow the on-screen position of the enemy players but are clamped to the screen area. This allows the player to use the HUDs of the enemies who are out of screen as a kind of radar.

Functionality

To pause the game, press the **START** button.

To **RESUME** the game or **LEAVE** the session, select the relative option on the **Pause** menu.

At the beginning of the gaming session each player is spawned at a random location in the level. The energy and shield values are set to the maximum levels and the score is reset.

The goal of each player is to defeat his/her opponents. Players can move and rotate, shoot bullets and use a special shield which protects them for a limited time from the enemy bullets. Table 6 shows the input controls for the **Game** screen:

Table 6 Input Controls for the Game Screen

Input Control	Action
Left analog stick	Moves the spaceship
X button	Shoots a bullet
O button	Holds down to enable the shield
START button	Pauses the game
L button	Toggles the communication statistics panel
R button	Resets the communication statistics

The score of a player is increased each time a bullet spawned by the player damages an enemy. A single hit results in 50 points. Each time a player scores, an internal bonus multiplier is increased. The bonus multiplier is reset to 1 if the player does not damage any opponent within one second of the previous strike. The actual assigned points are multiplied by the bonus factor. Both the score value and the bonus multiplier are shown in the top left HUD area.

The level area contains also a small HUD for each active enemy player. These HUDs follow the position on screen of the enemy players but are clamped to the screen area. This allows the player to use the HUDs of the enemies which are out of screen as a kind of radar.

The game can be paused with the **START** button, and players can then **RESUME** the game or **LEAVE** the session by selecting the relative entries of the pause menu.

A debug panel containing the communication statistics can be toggled by pressing the **L** button. Communication statistics can be reset using the **R** button.

NP Toolkit Usage

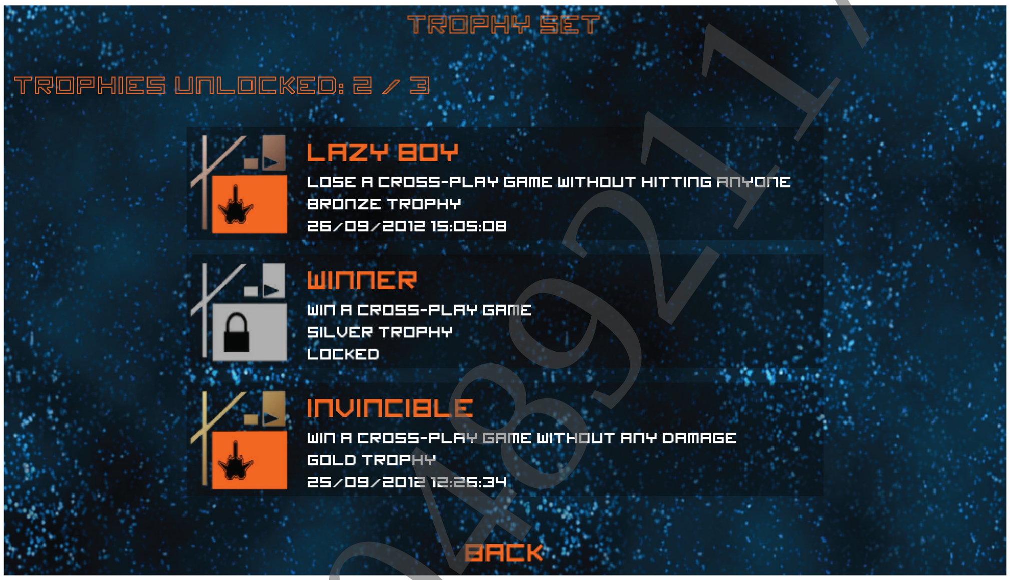
This screen uses the NP Toolkit to update the player profile at the end of each match. The following operations are performed:

- The trophy set is processed to unlock the achieved trophies for the local player.
- The score of the local player is uploaded to the PSN℠ scoreboard of the current level.
- If the local player won the match the next level is unlocked and the TUS variable containing the unlock state of the levels is updated.

Trophy Set Screen

The Trophy Set Screen allows the player to see the trophy list and the current game progress.

Figure 11 The Trophy Set Screen



TROPHY SET

TROPHIES UNLOCKED: 2 / 3

LAZY BOY
LOSE A CROSS-PLAY GAME WITHOUT HITTING ANYONE
BRONZE TROPHY
26/09/2012 15:05:08

WINNER
WIN A CROSS-PLAY GAME
SILVER TROPHY
LOCKED

INVINCIBLE
WIN A CROSS-PLAY GAME WITHOUT ANY DAMAGE
GOLD TROPHY
25/09/2012 12:26:34

Functionality

Each trophy can be in a locked or unlocked state. When a trophy is unlocked, the time it was unlocked is displayed next to the trophy.

Table 7 Input Controls for the Trophy Set Screen

Input Control	Action
X button	Goes back to the Title Screen
O button	Goes back to the Title Screen

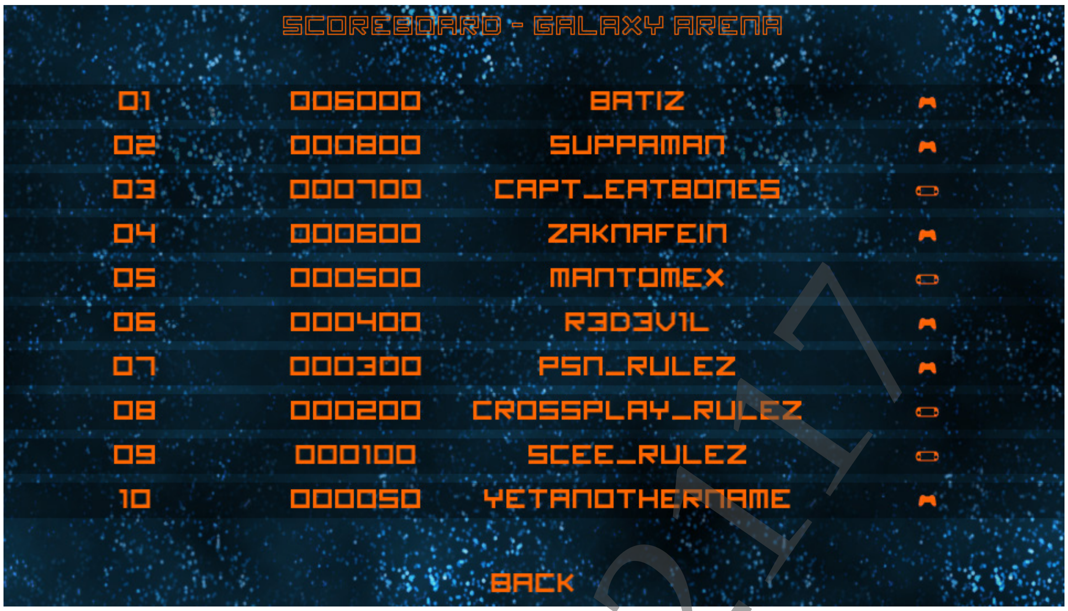
NP Toolkit Usage

This screen uses the NP Toolkit to retrieve the trophy set which has been registered with the application package. For each registered trophy, its name, description, unlock state and unlock date are provided to the application.

Scoreboards Screen

The **Scoreboards** screen allows the player to see the ranking scoreboard of each level.

Figure 12 The Scoreboards Screen



SCOREBOARD - GALAXY ARENA

01	006000	BATIZ
02	000800	SUPPAMAN
03	000700	CAPT_EATBONES
04	000600	ZAKNAFEIN
05	000500	MANTOMEX
06	000400	R3D3V1L
07	000300	PSN_RULEZ
08	000200	CROSSPLAY_RULEZ
09	000100	SCEE_RULEZ
10	000050	YETANOTHERNAME

Functionality

Each level has its own scoreboard. Each entry contains the ranking number, the best score for the current level, the NP ID of the player who registered the score and an icon indicating the player's platform.

To scroll up and down the scoreboard, use the D-Pad buttons.

To select a level, press the left and right D-Pad buttons.

Table 8 Input Controls for the Scoreboards Screen

Input Control	Action
D-Pad left	Selects the scoreboard of the previous level
D-Pad right	Selects the scoreboard of the next level
D-Pad up	Scrolls to the previous 10 rankings
D-Pad down	Scrolls to the next 10 rankings
O button	Goes back to the Title Screen

NP Toolkit Usage

This screen uses the NP Toolkit to access to the PSN℠ scoreboards and navigate in the ranking entries.

4 Cross-Platform Development

Implementing a cross-platform application can be challenging, especially if porting to new platforms is not planned at the very beginning of the development process. This chapter briefly analyzes the most important considerations to simplify the development task.

High-Level Approach to Cross-Platform Development

There are at least two approaches to cross-platform development. One approach is to create multiple versions of the same application using completely different code bases. While this might seem an intuitive and easy approach, it is usually much more expensive in terms of development time. Maintainability is also an issue, as every change or bug fix usually needs to be integrated in different code bases.

A second approach is to use the same code base for each platform and isolate the platform-specific parts. This is based on the observation that typically in games only the low-level subsystems require platform-specific implementations; the gameplay code tends to be *platform agnostic* (or at least it should be).

This approach requires the developers to build the application in a modular way, so that all the platform-specific aspects are "wrapped" into the low-level modules and the gameplay is built on top of them. The benefits are that development time is heavily reduced and maintainability is easier because the entire code base is shared among all the target platforms.

A potential drawback is that developers are often restricted to using the lowest common denominator subset of features which are available on all platforms. Depending on how the low-level modules are implemented, this could prohibit developers from using the most advanced features of the different platforms.

The Cross-Play demo uses the second approach to tackle the portability issues. The project has been split into the following:

- The *Games Toolkit* library which handles the low-level, platform-specific aspects of the development.
- The entirely cross-platform *Demo* module which contains the gameplay of the Cross-Play demo and is shared between the PlayStation®3 and the PlayStation®Vita versions of the game.

Code Portability

Language standards should guarantee that all the conformant compilers behave in exactly the same way. In practice, however, different compilers tend to produce code in different ways. Although the PlayStation®3 and PlayStation®Vita compilers are very similar in terms of behavior and custom keywords, there are some portability issues.

Compiler Extensions

Many compilers extend the standards-defined feature set by adding proprietary keywords to the language. These keywords usually control very useful aspects of C/C++ programming which are not directly specified by the language standard; for example, function inlining, warnings suppression, data padding, and data alignment. These extensions usually define new keywords which are prefixed and suffixed with a double underscore; for example, the `__attribute__` declaration specifier.

Using compiler-specific extensions in cross-platform code leads to portability issues because the compiler used on another platform might use a different keyword or might not even support all the extensions used. The PlayStation®3 and PlayStation®Vita compilers have a very similar set of extensions but, as a good practice, they are never directly referenced in the Cross-Play code. Instead, the Games Toolkit module defines a set of cross-platform preprocessor macros which wrap the non-standard keywords.

Size of the short, int and long Built-in Types

The size of these types is compiler-dependent. Only a few rules are defined by the C/C++ standards:

- `short`s must be at least 16 bits wide
- `int`s must be at least as wide as `shorts`
- `long`s must be at least as wide as `ints`

The rules above give compilers some degree of freedom in their implementation. For example, both of the following scenarios are valid:

- `short`s are 16 bits wide, `int`s are 32 bits wide and `long`s are 64 bits wide
- `short`s are 16 bits wide, `int`s are 32 bits wide and `long`s are 32 bits wide

Ignoring the type size differences can lead to communication issues in networked applications such as the Cross-Play demo whereby the format of the data structures sent across the network cannot match on the destination platform.

The type size can also affect bitwise operators: errors can be introduced by assuming the size of the built-in types being manipulated.

Developers can handle these inconsistencies in different ways; for example, by using the *standard integer types* defined in `stdint.h` which have well-defined sizes. The Cross-Play demo uses custom `typedef`s which map onto the standard integer types for all the types that can be serialized over the network.

Data Structure Padding

Modern CPUs read and write data in word-sized chunks at addresses which are multiples of the word size. Memory operations which are not aligned can generate more memory accesses, thus degrading performance.

By default the compiler allocates structure members on aligned boundaries, but sometimes structures can have members with different alignment requirements. In these cases the compiler inserts meaningless padding bytes between the members of the data structures. Padding is only inserted when a member of a structure is followed by a member with a larger alignment requirement, or at the end of the structure. Because that alignment depends on the word size of the target CPU, the amount of padding bytes inserted can be different on different platforms. The default behavior of the compiler can produce data structures which have the same members but a different memory layout on different architectures. The networking code must take these differences into consideration to prevent communication issues.

There are various ways to tackle this problem but the most efficient one, considering that network bandwidth is usually a scarce resource, is to prevent the compiler from inserting padding. Different compilers usually have proprietary extensions to deal with alignment and padding.

Although the PlayStation®3 has a 64-bit processor memory, operations can be 32-bit aligned; therefore the compiler aligns structures to a 32-bit boundary, which is the same as that of the PlayStation®Vita compiler. As a good practice, the Cross-Play demo turns off data padding by using the `__attribute__((packed))` declaration specifier for structures. To maintain a good performance level, data structures are manually aligned at a 32-bit boundary using standard integer types.

The example below declares two structures:

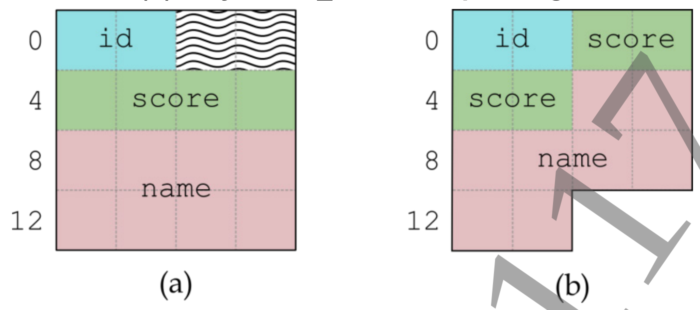
- `PlayerData_A` in which the compiler adds padding.
- `PlayerData_B` in which the padding insertion is inhibited with the `__attribute__((packed))` keyword.

```
struct PlayerData_A
{
   GtUInt16 id;
   GtUInt32 score;
   GtChar8 name[8];
};
```

```
struct __attribute__((packed)) PlayerData_B
{
   GtUInt16 id;
   GtUInt32 score;
   GtChar8 name[8];
};
```

The memory layout of the two structures above, assuming a 32-bit alignment, can be represented as in Figure 13. The 2 bytes after the `id` attribute on the left are meaningless padding bytes.

Figure 13 Memory Layout of (a) PlayerData_A structure with padding and (b) PlayerData_B without padding



Endianness Considerations

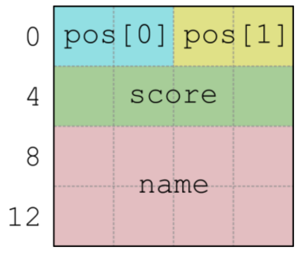
The term "endianness" refers to the way in which multi-byte types are stored in memory. A big endian machine stores the most significant byte first; a little endian machine stores the least significant byte first. The CELL processor of the PlayStation®3 is a big endian machine. The Cortex A9 processor of the PlayStation®Vita supports mixed endianness, which means that it can be configured to work either as a little endian or big endian machine. All the system libraries, however, are built using the little endian byte ordering and therefore the PlayStation®Vita is referred to as a little endian machine.

The endianness difference between PlayStation®3 and PlayStation®Vita requires Cross-Play titles to handle the shared data structures in an endian-aware way, which means that the title should swap the bytes according to the memory layout of the shared data structures. For example, consider the following structure:

```
struct PlayerData
{
   GtUInt16 pos[2];
   GtUInt32 score;
   GtChar8 name[8];
};
```

The memory layout of the `PlayerData` structure, assuming a 32-bits alignment, can be represented as shown in Figure 14.

Figure 14 Memory Layout for the PlayerData Structure

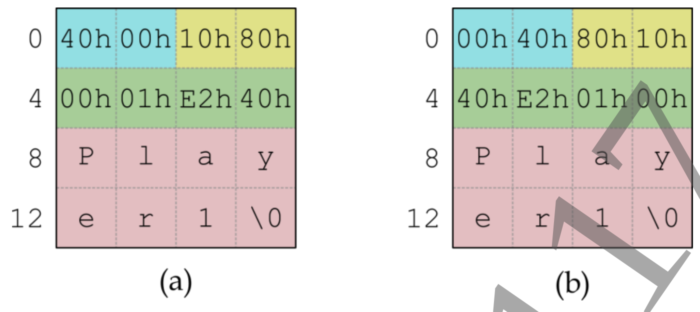


The order of the attributes of `PlayerData` structure is preserved, while the order of the bytes belonging to each attribute is swapped. Assume an instance of `PlayerData` is initialized by running the following code:

```
PlayerData data;
data.pos[0] = 0x0400;
data.pos[1] = 0x1080;
data.score = 0x001E240;
strncpy(data.name, "Player1", 8);
```

Then the memory layout for a big endian and a little endian machine would be as represented in Figure 15.

Figure 15 Byte Ordering for the PlayerData Structure with (a) Big Endian Byte Ordering and (b) Little Endian Byte Ordering



Sharing Endian-aware Data with Gt::NetworkDatagram

The Games Toolkit module has some built-in byte-order swapping functionality that can easily be used by the network communication protocols and is encapsulated by the `Gt::NetworkDatagram` class. This class contains an internal memory buffer in which the contents of a single UDP datagram can be written and read in an endian-aware fashion.

The `readStruct` and `writeStruct` methods are used respectively to read and write endian-aware structures in the datagram. Both methods require an array of `Gt::NetworkDatagramField` structures as a parameter. Each entry of this array defines the memory layout of one attribute of the serialized structure. This array of field descriptors is used internally by the `swapBytes` method to swap bytes belonging to a single structure attribute, if necessary.

The `Gt::NetworkDatagramField` structure is defined as follows:

```
struct Gt::NetworkDatagramField
{
    GtUInt size;
    GtUInt count;
};
```

The `size` attribute should be set as the type size of a single element of the relative structure attribute. The `count` attribute should be set as the number of elements belonging to the attribute, which is useful to support arrays in a compact way. Referring to the code sample above, a PlayerData structure could be serialized in an endian-aware way with the following code:

```
const Gt::NetworkDatagramField fields[4] =
{
    {2, 2},     // GtUInt16 pos[2]
    {4, 1},     // GtUInt32 score
    {1, 8},     // GtChar8 name[8]
};

Gt::NetworkDatagram datagram;
PlayerData data;

// ...

// Sender (e.g: PS3)
datagram.writeStruct(&data, sizeof(data), fields, 4);
```

```
// ...

// Receiver(e.g: PS Vita)
kDatagram.readStruct(&data, sizeof(data), fields, 4);
```

The `swapBytes` method, which is called internally by the `readStruct` and `writeStruct` methods, is based on a set of inlined functions called `GtEndianSwap`. These functions, where possible, are heavily based on intrinsics increasing the swapping performances. The entire infrastructure required to perform the swapping, however, has a non-null cost in terms of performance. To completely remove this overhead on some specific platforms, the swapping behavior of `Gt::NetworkDatagram` can be statically set by defining the preprocessor macro `GT_NETWORK_ENDIANNESS` with the `GT_LITTLE_ENDIAN` or the `GT_BIG_ENDIAN` values. In other words, the `Gt::NetworkDatagram` class performs the bytes swapping only when the network endianness is different from the native endianness. The default implementation sets the reference endianness as little endian, because this saves some computation time required to swap the network data on PlayStation®Vita, which is not as performant as PlayStation®3.

5 NP Toolkit Usage in the Cross-Play Demo

This chapter gives a brief overview of how the NP Toolkit is used to implement the Cross-Play features of the Cross-Play demo.

For more information about the NP Toolkit library, refer to the *NP Toolkit Library Overview*.

Matching

The main feature on which the Cross-Play demo is based is the ability to create multi-platform matching sessions through the facilities offered by the NP Toolkit. The matching API is responsible for creating a session room on the PSN℠ server, setting the attributes of this room, hosting the members of the session and retrieving their network addresses. The matching API can also be used to retrieve connectivity information about the members of a session. The Cross-Play demo uses the matching API to retrieve the ping time between the local player and all the remote players.

Signaling

The signaling process is responsible for retrieving the network addresses of all the members of a session. The NP Toolkit handles this complex process automatically.

Note: The NP Toolkit uses NP Signaling to perform NAT traversal process. It is recommended that you connect the device to a network with NAT Type 1 or NAT Type 2. Connectivity issues may be experienced when traversing NAT type 3.

For more information about the signaling process, refer to the *NP Signaling Library Overview*.

Visibility

By default, matching sessions are visible to all players unless they are filtered out by search results or flagged as hidden. The Cross-Play demo dynamically sets the visibility of matching sessions to prevent remote players from joining active sessions. Right before starting the game (that is, before setting the session as ACTIVE) the session is set as hidden. The visibility is restored when the session owner returns to the **Session Lobby** screen.

The session visibility can be dynamically changed by using the `modifySession` method of `NP::Matching::Interface`. Refer to the `Application::setSessionVisibility` method for further details.

Ranking

The NP Toolkit allows both PlayStation®3 and PlayStation®Vita players to upload a score to a set of unified scoreboards. The Cross-Play demo uses this feature to populate a per-level scoreboard at the end of each game. The NP Toolkit also allows players to query the scoreboards to retrieve the sorted rankings for each level.

Trophy Sets

The NP Toolkit allows both PlayStation®3 and PlayStation®Vita players to share the same trophy set and their locked/unlocked status. Each player's progress is saved automatically on the PSN℠ server, which can then be accessed from both PlayStation®3 and PlayStation®Vita. This allows a player using the same PSN℠ profile on PlayStation®3 and PlayStation®Vita to start the game on one platform and continue seamlessly on the other.

Title User Storage

The NP Toolkit allows both PlayStation®3 and PlayStation®Vita players to share the Title User Storage (TUS) data, which is a block of memory of 1 MiB which is uploaded to the PSN℠ servers. The Cross-Play demo uses TUS data to store the locked/unlocked status of the levels. This allows a player using the same PSN℠ profile on PlayStation®3 and PlayStation®Vita to start the game on one platform and continue seamlessly on the other.

NP Toolkit Usage Tips

The NP Toolkit is a tool that allows the application to interface with the PSN℠ services at a higher level which makes implementation of these services easier. Moreover, the NP Toolkit library is also cross-platform on both PlayStation®3 and PlayStation®Vita. A complete walkthrough of the library is out of the scope of this document, but full details can be found in the *NP Toolkit Library Overview*. This section, however, briefly describes how the NP Toolkit is integrated into the Cross-Play demo and illustrates a few gotchas relating to the library.

Asynchronous Model and Future Objects

The network operations implemented by the NP Toolkit are asynchronous in nature.

Note: NP Toolkit also supports synchronous behavior for all the operations, but it is recommended that you use the asynchronous model as it prevents the calling thread from being blocked until the completion of each NP operation.
Migrating from a synchronous to an asynchronous programming model could be a non-trivial change in an application. For this reason, applications should be architected to work with an asynchronous model from the very beginning of development.

The NP Toolkit introduces the concept of *future object*, which is a proxy object which can be queried to retrieve the state of each asynchronous operation. The processes involved in the network operation of NP services using the NP Toolkit are split into two phases:

- Request phase: the request is triggered by passing a future object as an argument with a set of additional parameters.
- Wait/Callback phase: the future object can be queried until it returns the result or an error, or the application can wait for the callback. The callback is registered during the initialization of the NP Toolkit library.

Note: Each future object can be used for only one request at a time. The application has to check whether a future object is busy or not before kicking a new request.
Attempting to trigger a new request while the future object is still waiting for another request will result in an error.

For more information about future objects, refer to the *NP Toolkit Library Overview*.

Events and Threading

The NP Toolkit runs on a dedicated thread which is responsible for scheduling the NP operations. The communication of the NP Toolkit thread with the application is based on an event model in which a user callback is invoked in the context of the dedicated thread. To prevent synchronization issues, the Cross-Play demo implements a thread-safe queue of NP events which is filled up by the user callback provided to the NP Toolkit and processed each frame from the main thread.

The only exception to this model relates to the update of a matching session, which needs to be executed on the dedicated thread of NP Toolkit. The Cross-Play demo provides a mutex object which synchronizes accesses to the matching session data structure so that this operation is performed in a thread-safe way.

Socket and Matching Session Lifecycle

The UDPP2P socket created during the matching session should not have a life span outside the session. The Cross-Play demo initializes a socket only after the signalling process of a matching session has been successfully completed. The socket should be closed and shut down when the matching session is left. This is important for the following reasons:

- The signalling process could open different virtual ports for different sessions.
- Datagrams belonging to old sessions could interfere with a newer session.

Common Dialog Update

Many high-level functions needed by the NP Toolkit, such as PSN℠ login and trophy set initialization, require the Common Dialog library. The Common Dialog library behaves in different ways on PlayStation®Vita and PlayStation®3.

Table 9 summarizes the steps required for the Common Dialog library to work properly:

Table 9 Common Dialog Operations

	PlayStation®Vita	PlayStation®3
Initialization	Nothing required since SDK 2.100.	Call `cellSysutilRegisterCallback`.
Update	Call `sceCommonDialogUpdate` each frame during the execution of a common dialog.	Call `cellSysutilCheckCallback` each frame.
Shutdown	Nothing required.	Call `cellSysutilUnregisterCallback`.

6 Network Communication

This chapter analyzes the techniques used in the Cross-Play demo to implement a real-time multi-player gaming experience.

Introduction

The NP Toolkit provides interface APIs to set up a room on a NP Matching2 server and retrieve the network addresses of the session members. However, the actual network communication needs to be handled by the game itself, which is responsible for establishing a network connection and implementing a data protocol suitable for synchronizing the game state across all the session members in real time.

Architecture

Real-time networking in games is a wide and complex topic and an exhaustive walkthrough is beyond the scope of this document. The Cross-Play demo implements a simple client-server architecture based on UDPP2P. The network synchronization is implemented using a naïve approach where the clients send a lightweight input state to the server, which updates the scene and sends a state datagram back to the clients. Complex techniques such as lag compensation and input prediction are not implemented.

The Cross-Play demo implements both the server and the client on the same machine. The player who created the gaming session (the owner) assumes the role of the server, while all the other players act as clients. PlayStation®3 and PlayStation®Vita can both act as servers or clients.

The UDPP2P Protocol

The Cross-Play demo uses the UDPP2P protocol to send and receive datagrams across the network. This protocol allows many connections to be tunneled to the same port by introducing the concept of a "virtual port".

Note: The UDP port must be set as 3658 (`SCE_NP_PORT`). The virtual port can be an arbitrary port in the range 1-32767.
For more details about the UDPP2P protocol, refer to the *Network Overview*.

Network Communication States

The communication process in the Cross-Play demo is driven by a *finite state machine* which has the following states:

Table 10 States of Finite State Machine

State	Description
Stopped	The initial state of the communication. No data communication is performed. No session role is defined.
Initialized	The communication has been initialized. The session role (either client or server) is defined at this point. The data communication is performed between all the members for whom the signaling process has been completed.
Started	The communication is initialized and the server requested to start the game. The server can start a game only if the signaling process has been completed for all the members of the session. This state is used to signal to the clients that they need to load the level and start the game.
Active	The communication is active and the game is running.

The state machine is updated by different game screens according to the session role. Figure 16 shows the state machine for the server role. Figure 17 shows the state machine for the client role.

Figure 16 State Machine for the Server

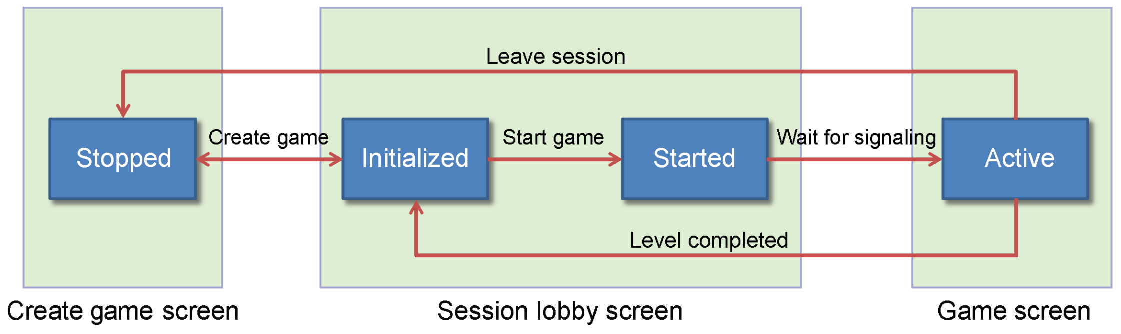
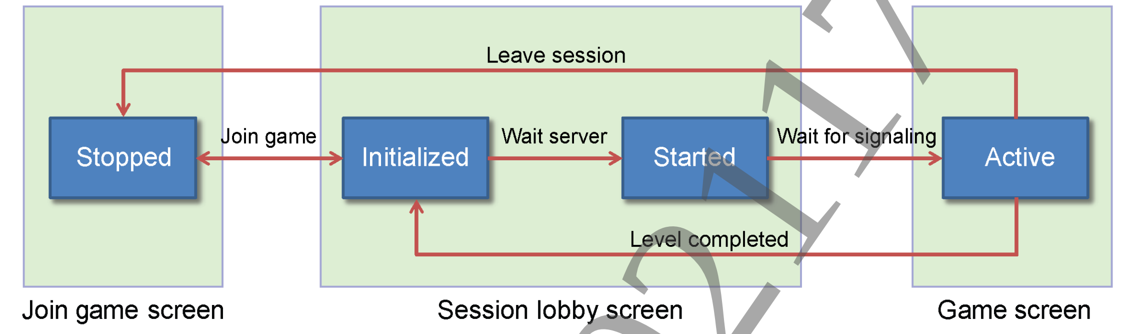


Figure 17 State Machine for the Client

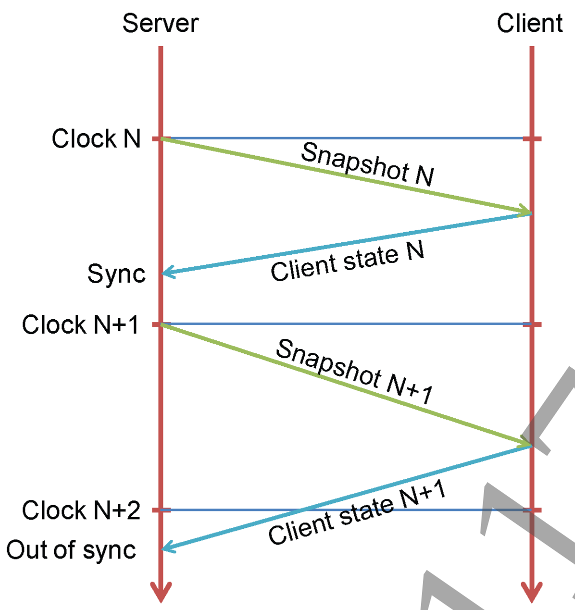


Game State Synchronization

The communication manager shares a synchronized state of the game between the server and the clients. The game state is thus split into two different macro structures:

- The Client State contains only that information which the server requires at each network cycle to update the game level, such as the input state. The client state is also used by the clients to communicate session-specific data to the server. For example, the ship type, which is a parameter that the server needs to know in order to compute the game logic, is sent as part of the client state.
- The Server State contains a snapshot of all the gameplay-critical elements of the level, that is, players and bullets. The positions and all the physical parameters required by the clients to interpolate the game state as well as the gameplay parameters such as energy, shield and score values are sent for each player. The position, speed and owner of each bullet are sent too.

Figure 18 The Client-Server Synchronization Protocol



Bandwidth Considerations

It is important to decide which elements of the level need to be serialized into the server snapshots. Ideally all the active elements of the game logic should be serialized; however, this would generate a snapshot of several KiB in size. Considering that each snapshot is sent 20 times per second, it is easy to saturate the available upload bandwidth. In general, titles need to work properly with a bandwidth of 128 Kbps so it is important to consider the minimum bandwidth requirements when designing a communication protocol.

Another aspect to consider is the datagram size. The UDPP2P protocol supports datagrams with a maximum payload of 9216 bytes, which means that bigger snapshots must be split into multiple datagrams, increasing the total overhead.

Because network bandwidth is a scarce resource for the Cross-Play demo, only the level objects which affect the gameplay are serialized. Cosmetic effects are not synchronized. To update the game logic properly, it is absolutely necessary to synchronize the state of each player and each active bullet. Effects like explosions, muzzle flashes and engine flares are not strictly necessary and therefore are not synchronized. This may produce wrong results in terms of visualization but at least the gameplay will be coherent.

Network Clock

The server maintains an internal clock which runs at 20 Hz and is used to determine the network cycles. Each network datagram contains the value of the network clock that the sender had when the datagram was sent. This value is used by both server and clients to understand whether a datagram contains fresh data and if it is synchronized with the stream of other datagrams.

From now on the term *network cycle* will refer to the timespan of 50 milliseconds in which the network clock is constant.

Shadow Copy of the Game State

The networking process is asynchronous in nature. Both server and client contain a shadow copy of the game state that is used to perform network operations while the game updates the real copy of the game state. At the end of each network cycle, the shadow copy and the real copy of the game state are synchronized. Depending on the session role (client or server), this synchronization happens in a different way. The following sections describe the synchronization process for both server and client.

Server Behavior

The server is responsible for receiving the client states, updating the game level and finally propagating the updated snapshot to the clients. The algorithm is split in two phases, a receive phase and a send phase.

Receive Phase

(1) Read all the datagrams queued into the UDP socket. For each received datagram:

(a) Check whether it belongs to a registered client or not.

(b) Check the network clock to understand if it contains a more recent client state or not.

(c) If the client state is newer:

(i.) Update the relative client state in the shadow state.

(ii.) Flag the client as synchronized.

(2) Copy the state of the remote clients from the shadow state to the real state.

(3) Update the input state relative to the local player in the real state.

Send Phase

The send phase is triggered by an internal timer which indicates when the network clock must be increased.

(1) Copy the game state from the real copy to the shadow copy.

(2) Send a snapshot datagram to all the remote clients.

If, at the end of a network cycle, all the clients are synchronized, then the game state has been updated successfully and at least the server side state of the game should be synchronized.

Client Behavior

The client is responsible for receiving the server snapshots, sending back the current state of the input and finally updating the game level. The algorithm is split in two phases, a receive phase and a send phase.

Receive Phase

(1) Read all the datagrams queued into the UDP socket. For each received datagram:

(a) Check whether it belongs to the server or not.

(b) Check the network clock to understand if it contains a more recent snapshot or not.

(c) If the snapshot is newer:

(i.) Update the network clock.

(ii) Copy the snapshot onto the real game state.

Send Phase

The send phase is triggered when the network clock changes. In other words the clients do not send an updated input packet as long as they do not receive a snapshot from the server.

(1) Copy the state the local client from the real state to the shadow state.

(2) Send a snapshot datagram to the server.

Game State Interpolation

Both clients and server can interpolate the game state during a network cycle.

Clients interpolate the game state using the last valid snapshot as a starting point. As soon as a new snapshot is received, the level is updated using the new data. Unfortunately, this produces a glitch in the gameplay if the state of the level calculated by the client differs too much from the one calculated by the server. For simplicity reasons, the Cross-Play demo does not implement any form of snapshot interpolation but a ping time of less than 50ms (that is, the duration of a network cycle) allows the game to run smoothly.

Connection Statistics

The connection manager contains some internal counters which are useful to troubleshoot the communication. Table 11 briefly describes the most important parameters tracked.

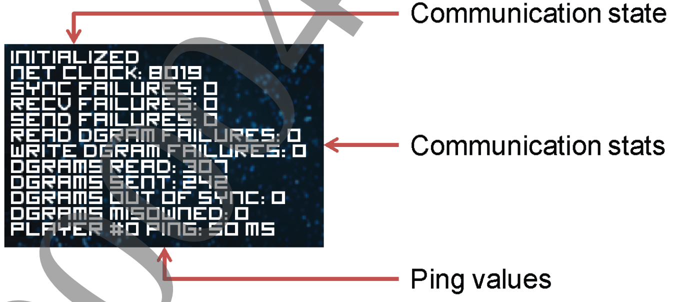
Table 11 Connection Statistics

Counter Name	Description
Net clock	The network clock.
Sync failures	The number of network cycles in which the server could not retrieve a synchronized state with all the clients. A high number of failures indicates a poor connection with some of the clients.
Recv failures	The number of network errors on datagrams receive operations. Failures could be related to a networking issue; for example, socket receive buffer full.
Send failures	The number of network errors on datagrams send operations. Failures could be related to a networking issue; for example, socket send buffer full.
Dgrams out of sync	The number of out-of-order datagrams received.
Dgrams misowned	The number of datagrams received which do not belong to any of the allowed senders.

These counters are displayed in a debug panel which is accessible in the **Session Lobby** screen and the **Game** screen by pressing the **L** button, as shown in Figure 19. The internal counters can be reset by pressing the **R** button.

The state of the communication is reported at the top of the panel. The ping time of each remote player is shown at the bottom.

Figure 19 The Communication Stats Panel



```
INITIALIZED
NET CLOCK: 8019
SYNC FAILURES: 0
RECV FAILURES: 0
SEND FAILURES: 0
READ DGRAM FAILURES: 0
WRITE DGRAM FAILURES: 0
DGRAMS READ: 307
DGRAMS SENT: 242
DGRAMS OUT OF SYNC: 0
DGRAMS MISOWNED: 0
PLAYER #0 PING: 50 MS
```

7 Gt: Games Toolkit

This chapter describes the Games Toolkit library, a cross-platform library supporting both PlayStation®3 and PlayStation®Vita.

Overview

The Cross-Play demo is implemented using a unified code base for both the PlayStation®3 and PlayStation®Vita versions. However, these platforms are different in terms of hardware and APIs. The Games Toolkit library abstracts such differences and offers a very basic layer of low level functionalities such as:

- System initialization and shutdown
- Renderer management
- Graphical assets management
- Input management
- File streams management
- Sockets management
- High resolution timers
- Endianness conversion
- Settings management

The Cross-Play demo has been implemented on the top of the Gt library, which means that the game's code is entirely cross-platform.

System Initialization and Shutdown

The `Gt::System` class handles system services initialization and shutdown in a highly customizable way. It exposes two static methods, `initialize` and `shutdown`, which perform the initialization and de-initialization of the required system services respectively. The `initialize` method accepts an instance of `Gt::System::InitParams` as a parameter and returns `GT_TRUE` on success or `GT_FALSE` on failure.

Renderer Management

The `Gt::Renderer` singleton class encapsulates the very basic set of rendering functionalities supported by the Games Toolkit module in a completely cross-platform way:

- Initialization/de-initialization of the rendering device
- Viewport management
- Screen clearing operations
- 2D camera management
- 2D bitmap rendering
- GPU memory management
- GPU synchronization

Initialization Parameters

The `Gt::Renderer::InitParams` class contains the initialization parameters for the renderer. The default constructor sets up a default configuration which is suitable in most cases. Table 12 describes the most important initialization parameters.

Table 12 The Renderer Initialization Parameters

Parameter	Description
`canvasWidth`	The width of the canvas (viewport) in pixels.
`canvasHeight`	The height of the canvas (viewport) in pixels.
`maxBitmapInstanceCountPerBuffer`	The maximum number of bitmap instances that can be batched together in a single instance buffer, i.e.: in a single draw call.
`initialBitmapInstanceBuffersCount`	The number of instance buffers which are allocated by the renderer at initialization time.
`gpuDebuggerEnabled`	Indicates whether or not Razor or GPAD can be attached to the application.

Viewport Management

The `Gt::Renderer` class automatically initializes the display mode to the system resolution. The renderer, however, allows the specification of a viewport rectangle which can be used to set up a 2D reference frame which is consistent across different platforms. The Cross-Play demo uses this feature to abstract pixel coordinates to *canvas coordinates* to make sure that PlayStation®3 and PlayStation®Vita users see the same portion of the level.

Camera Management

The `Gt::Renderer` class allows the specification of a simple 2D offset which is used internally to translate the scene. Camera rotations are not supported for simplicity reasons as they are not needed by the Cross-Play demo.

Bitmap Rendering

The `Gt::Bitmap` class represents a 2D bitmap in the Games Toolkit module.

Bitmaps have a platform-specific internal implementation which handles the life-cycle of the underlying texture object. Bilinear texture filtering is enabled by default on every `Gt::Bitmap` object and mipmapping is also supported.

Bitmaps are graphical resources managed by the `Gt::ResourcePackage` class. For more information about resource packages, see Assets Management.

The main rendering feature required by the Cross-Play demo is 2D bitmaps rendering. The `Gt::Renderer` class exposes some overloaded `drawBitmap` methods which implement bitmap rendering supporting rotation, zooming, and source and destination region selection. Rotation is internally implemented by fetching a look-up table of pre-rotated quad offsets; this method is more convenient than performing the rotation operation in the vertex shader.

Bitmap Blending

The `Gt::Renderer` class supports simple color blending between the source and destination surfaces. When a bitmap (called *source*) is rendered upon the frame buffer (called *destination*), the source and destination color can be combined using different blending modes. Table 13 shows all the supported blending modes:

Table 13 Supported Blending Modes

Blending Mode	Description
`BM_NORMAL`	Default blending mode. The source color overwrites the destination color. The source alpha is used to modulate the source color and the inverse of the source alpha is used to modulate the destination color.
`BM_ADD`	The source color is added to the destination color. Useful to brighten the destination surface. In the Cross-Play demo this mode is used to draw glowing sprites such as bullets and explosions.

Blending Mode	Description
BM_MULTIPLY	The source color is multiplied by the destination color. Useful to darken the destination surface.

Bitmap Instance Batching

The `Gt::Renderer` class tries to reduce the amount of draw calls by batching together the `drawBitmap` calls which share the same rendering state. A change of bitmap object or blending mode produces a flush of the current batch of primitives internally.

The helper `Gt::BitmapInstanceBuffer` class is used internally to store the vertex and index data for the independent bitmap instances. Those buffers are managed by the renderer using a free-list so that each time a new buffer is needed the free-list is accessed. When a buffer is discarded, the renderer waits until the GPU has finished with it before returning it to the free-list.

The maximum number of bitmap instances that can be batched together in a draw call is defined by the `maxBitmapInstanceCountPerBuffer` attribute of the `Gt::Renderer::InitParams` class. Note that each instance buffer can store exactly `maxBitmapInstanceCountPerBuffer` bitmap instances; therefore a good trade-off between batch size and memory should be found. If the application uses many different bitmap images, more draw calls are likely to cause a performance loss and a high memory footprint, because each image will be assigned to a different instance buffer. The suggested strategy to keep the draw calls count and the memory usage low is to use the source region clipping functionalities in `Gt::Renderer` to group different bitmaps into a single atlas. For more information, see the Bitmap Atlas section.

The initial amount of pre-allocated instance buffers can be specified with the `initialBitmapInstanceBuffersCount` parameter. The application should try to minimize per-frame allocations as much as possible because they can easily degrade performance. Therefore it is strongly recommended that you find a reasonable value for this parameter.

Bitmap Atlas

The `Gt::BitmapAtlas` class, based on the source region selection feature of `Gt::Renderer`, encapsulates bitmap atlasing functionalities. This class is widely used by some other convenience classes such as `Gt::BitmapFont` and `Gt::TiledMap`. Grouping different images together is not only useful from a logical point of view but also leads to performance and memory improvements, as described in the Bitmap Instance Batching section.

Bitmap Fonts

The `Gt::BitmapFont` class can be used to draw and measure text on the screen using raster fonts.

The `drawText` method draws a text string at a specific screen position. The `alignment` parameter is an OR-ed combination of the values defined by the `Alignment` enumeration and can be used to align the text around the argument position. The convenience `formatAndDrawText` method accepts a *printf*-like argument string with a variable number of arguments and can be used to format text strings dynamically.

The `Gt::BitmapFont` class also exposes some text measurement methods such as `measureTextExtents`, which retrieves the width and height of the passed string, and `formatAndMeasureTextExtents`, which accepts a *print*-like argument string as a parameter. The `getLineHeight` method returns the line height of the font in pixels.

Bitmap fonts are graphical resources managed by the `Gt::ResourcePackage` class. For more information about resource packages, see Assets Management.

Tiled Maps

The `Gt::TiledMap` class can be used to draw a layered *tile*-based bitmap. Tiled maps can be used to produce very big virtual bitmaps, keeping the memory requirements very low. Each map is made of multiple background and foreground layers, which can be rendered respectively before and after the level objects. In the Cross-Play demo, for instance, the level's background is rendered first, then the game area is rendered and finally the foreground layers containing the rocks which are close to the camera are rendered.

The `Gt::TileMap` supports per-layer parallax levels to achieve an old-style *depth* effect.

Tiled maps are graphical resources managed by the `Gt::ResourcePackage` class. For more information about resource packages, see Assets Management.

Assets Management

The graphical resources which are used by the Game Toolkit to render bitmaps, fonts and maps usually require a fairly high amount of memory. Also, bitmaps are mapped internally to texture objects which must be formatted in a GPU-dependent format. PlayStation®3 and PlayStation®Vita have different native formats for textures. To simplify the life cycle management of the graphical assets and reduce the loading time, a `Gt::ResourcePackage` class has been implemented. When a resource package is loaded, all the contained resources are set up and, upon a successful operation, they can be referenced safely by the application. Resource life-cycle is managed by the package, which is responsible for deallocating the resources during its destruction.

Input Management

The `Gt::InputManager` singleton class encapsulates all the functionalities required to handle user input. For simplicity and to provide a uniform gaming experience across the supported platforms, only a subset of the input peripherals available for PlayStation®3 and PlayStation®Vita are supported.

- The `Gt::InputManager` singleton supports only game pad input events, that is, digital buttons, analog sticks and shoulder triggers.
- The `Gt::InputManager::Button` enumeration defines the set of all the supported digital buttons.
- The `Gt::InputManager::Stick` enumeration defines a set of the supported analog axis.

Table 14 and Table 15 show the supported input events and a mapping for the specific platforms.

Table 14 Gt::InputManager::Button Input Events

Gt::InputManager::Button	PlayStation®3 Implementation	PlayStation®Vita Implementation
BTN_UP	D-Pad up – DUALSHOCK®3 wireless controller	D-Pad up
BTN_DOWN	D-Pad down – DUALSHOCK®3 wireless controller	D-Pad down
BTN_LEFT	D-Pad left – DUALSHOCK®3 wireless controller	D-Pad left
BTN_RIGHT	D-Pad right – DUALSHOCK®3 wireless controller	D-Pad right
BTN_CROSS	Cross button – DUALSHOCK®3 wireless controller	Cross button
BTN_CIRCLE	Circle button – DUALSHOCK®3 wireless controller	Circle button
BTN_TRIANGLE	Triangle button – DUALSHOCK®3 wireless controller	Triangle button
BTN_SQUARE	Square button – DUALSHOCK®3 wireless controller	Square button
BTN_L	L1 – DUALSHOCK®3 wireless controller	L
BTN_R	R1 – DUALSHOCK®3 wireless controller	R

Gt::InputManager::Button	PlayStation®3 Implementation	PlayStation®Vita Implementation
`BTN_SELECT`	Select – DUALSHOCK®3 wireless controller	Select
`BTN_START`	Start – DUALSHOCK®3 wireless controller	Start

Table 15 Gt::InputManager::Stick Input Events

Gt::InputManager::Stick	PlayStation®3 Implementation	PlayStation®Vita Implementation
`STICK_LEFT`	Analog left stick – DUALSHOCK®3 wireless controller	Analog left stick
`STICK_RIGHT`	Analog right stick – DUALSHOCK®3 wireless controller	Analog right stick

The boolean state of the digital buttons can be queried using the `isButtonPressed`, `isButtonReleased`, `isButtonUp` and `isButtonDown` methods of the `Gt::InputManager` class. `isButtonPressed` and `isButtonReleased` return `GT_TRUE` only in the frame in which the specific action occurs, that is, when a button is pressed or released, respectively. The `isButtonUp` and `isButtonDown` methods return the instantaneous state of each digital button.

The `getEnterButton` and `getBackButton` methods can be used to provide a region-coherent behavior of the application in terms of menu navigation. On PlayStation®3, the behavior of the X and O buttons is related to the release region of the console: in the USA region the X button is used to confirm and the O button is used to cancel; in the Japan region the opposite applies. Using the `getEnterButton` and `getBackButton` methods, the application can abstract the region-specific behavior and provide a gaming experience which is coherent to the other system menus.

The `getStickValueX` and `getStickValueY` methods return the analog state of the required analog axis in the [-1, 1] range. An absolute dead zone value can be specified in the initialization parameters of `Gt::InputManager` to filter the output around the center position of the analog sticks.

File Streams Management

Handling file streams is a very common operation and both PlayStation®3 and PlayStation®Vita have a very similar and intuitive API for it. The naming convention and some parameters, however, are slightly different. In addition, the path name conventions used to access the file system are different. The `Gt::File` class of the Games Toolkit module encapsulates some very basic file access functionalities in a completely cross-platform way.

Path names define where a specific resource is located in the storage areas of the target consoles. PlayStation®3 uses a UNIX-like path system; PlayStation®Vita supports absolute paths using drive specifiers. The Games Toolkit module defines a cross-platform, drive-based path name convention in which path names must be defined as absolute paths which are then mapped to the platform-specific paths. Table 16 shows how the Gt paths are mapped for the supported platforms.

Table 16 Gt Path Mapping

Games Toolkit Drive	PlayStation®3 Mapping	PlayStation®Vita Mapping
`media:`	`/app_home`	`app0:`
`debug:`	`/host_root`	`host0:`

The `Gt::File` class manages a single file stream using the low-level API of the target platform internally. The `toNativePath` static method of `Gt::File` performs the Games Toolkit pathname translation to a platform-specific path name.

Networking Management

The Gt::UdpSocket class implements the common features of a UDP socket.

On PlayStation®Vita, the network stack can be reset upon a system suspension and therefore all the sockets need to be recreated when the connectivity is resumed. The Gt::UdpSocket class maintains an internal list of all the active sockets. This list is used by the resumeAll method to re-initialize all the sockets upon request. The Cross-Play demo is responsible for invoking the resumeAll method when the network layer is resumed.

The networking service needs to be initialized through the Gt::System::Initialize method. For more information, see System Initialization and Shutdown.

High Resolution Timers

The Gt::Timer class encompasses high resolution timing functionalities. All the platform-specific implementations of this class are based on the hardware clock of the CPU and provide a microsecond-accurate time interval measurement.

The start method resets the internal base time; the set of queryXxx methods retrieve the time elapsed since the last start call in different formats and time units. Floating point, 32 bits and 64 bits formats are supported. The Gt::Timer class stores a 64 bits wide value internally to perform its operations.

Settings Management

The Gt::Settings class is a data-driven settings manager which can be used to read the configuration parameters of the game logic from a text file.

Settings are loaded from a file using the load method which accepts the pathname of the settings file as a parameter and returns a boolean value indicating whether the operation was successful or not.

For simplicity, settings are text-based and can be queried using the getValue method, which accepts the setting name as a parameter and returns a boolean value indicating whether the setting has been found or not in the file. Settings can also be set using the setValue method. Some other convenience methods such as getIntValue, getUIntValue and getFloatValue are available for querying settings in commonly used formats.

Settings File Format

Settings are specified in a text file with the following format:

```
setting_name=value # Comment
```

The example below defines the string_value, int_value and float_value settings.

```
# String value
string_ value=My test string

# Integer value
int_value=10

# Float value
float_value=-123.567
```