TRC Compliant Shooting Game Creation Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Introduction

About This Tutorial

The technical requirements to be complied with by applications are written up in a TRC (Technical Requirements Checklist). This tutorial is intended as a reference for application development and as such provides a sample program that complies with TRC requirements.

Tutorial Location

The sample program of this tutorial is located in the following directory:

- %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant

Build Configurations in This Tutorial

When this tutorial is built, the features enabled will vary based on build configuration.

Debug Build

All features will be enabled, including the "near" system. All the features of this tutorial can be debugged using this build configuration.

Release Build

Features other than the "near" system will be enabled. The self file created using this build configuration will be included in the application package files. Moreover, if the Publishing Tools are installed, with this build configuration, application packages will be created under %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\package.

Patch Build

All features will be enabled, including the "near" system. The self file created using this build configuration will be included in the patch package files.

Start-up Method of This Tutorial

In addition to startup from Neighborhood for PlayStation®Vita or Visual Studio, which is supported by regular samples, a file for LiveArea™ is also provided, allowing this tutorial to be started up from ★APP_HOME on the home screen. Start up the program from ★APP_HOME with the following procedure.

(1) Use Visual Studio to build the following solution file using the "Patch Build".
%SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\tutorial_simple_shooting_game.sln

(2) eboot.bin file is created to the following path
%SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\app\eboot.bin

(3) In order to use a Development Kit (DevKit) to start up a program from the system software's ★APP_HOME icon,
set %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data in host0. For details on the application start-up from ★APP_HOME, refer to the "Starting Application from ★APP_HOME" chapter of the "System Software Overview" document.

(4) Tap the ★APP_HOME icon to display LiveArea™ of the tutorial. For details on LiveArea™, refer to the "LiveArea™ Specifications" document.

(5) Tap Gate in LiveArea™ to start up the tutorial.

Procedure for the Creation and Installation of Patch Package Files

(1) With Publishing Tools installed, build %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\tutorial_simple_shooting_game.sln with "Release Build". The pkg file prior to adding the patch (%SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\package\IV0002-NPXS09292_00-THELASTGALLERYFP.pkg) will be generated.

(2) Build %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\tutorial_simple_shooting_game.sln with "Patch Build". The file eboot.bin, necessary for the package, will be generated.

(3) Double click on %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\thelastgallery_patch.gp4p. Package Generator will start up.

(4) Click on the **Package** button that is in Package Generator. The IV0002-NPXS09292_00-THELASTGALLERYFP.pkg file will be created at the specified location

(5) First, install the pkg file prior to adding the patch (IV0002-NPXS09292_00-THELASTGALLERYFP.pkg, created in (1)) on the DevKit. Next, by installing the IV0002-NPXS09292_00-THELASTGALLERYFP.pkg file created in (4), the patch will be added to the package that was installed first. Refer to the "Application Development Process Overview" document for details on how to install pkg files.

2 Precautions regarding Implementation Taking into Consideration TRC

Below is an explanation of points requiring special consideration when an application performs implementation, and of how these were implemented in this tutorial.

Processing Related to Loading, Blackout Screen, Screen Stopped State, and Non-response State

Based on TRC [R3019], it must be confirmed that the time required for loading does not exceed 60 seconds. Based on TRC [R3020], it must be confirmed that the blacked out, suspended and unresponsive screen display durations do not exceed 15 seconds. In particular, check that when loading large assets, the continuous display time of the loading screen does not exceed 60 seconds.

In this tutorial, shortened loading time is achieved by loading the assets on the backend while displaying the startup logo. If the loading time may exceed 60 seconds owing to insufficient loading speed, consider taking measures such as compressing the data using PSARC of FIOS2 in order to increase the loading speed.

Error Processing

The error processing to be implemented by applications can be classified into the two main types as follows.

(a) When an error occurs, the application performs processing for that error and automatically restores the regular processing without notifying the user

(b) When an error occurs, a message indicating that an error has occurred and the recovery method is displayed to the user, and the user is requested to take the appropriate action according to the error occurrence status

Errors that require handling method (b) are stipulated as errors that must be notified to the user upon occurrence. One of the typical errors that require the handling method (b) is capacity shortage of save data file system. In this tutorial, according to the TRC requirements, an error message is displayed through the error code display mode of the Save Data dialog, and the error message will be repeatedly shown until the file system capacity shortage is resolved.

Save Data Implementation Method

Based on TRC [R3022], broken status should not be applied to save data for reasons other than causes that cannot be avoided by the application, such as the direct removal of the memory card while save data is being written. The processing of `sceAppUtilSaveDataDataSave()` completes even if an application is suspended while this function is being called. However, because there is no guarantee that the save data will not be broken, separately follow TRC [R3070] and implement measures against broken save data. In this tutorial, necessary save data updates are performed by calling `sceAppUtilSaveDataDataSave()` once. If `sceAppUtilSaveDataDataSave()` needs to be called more than once, implement simplified transaction processing whereby a flag file indicating that writing is in progress is created at the first writing and then deleted once the last writing is complete, so that the application may detect that the processing sequence has been completed normally, and also consider implementing processing enabling the detection of application termination during writing.

Based on TRC[R3070], when save data processing within an application does not complete normally because of a bug or broken data, it is prohibited to continue the application or to delete save data without notifying the user. When save data is in the broken status upon load, this shooting game is implemented so that the user is asked whether to delete save data and save data is deleted only when the user permits it.

For the concrete implementation method in this tutorial, refer to the Chapter 6 "Using the Save Data Utility".

Calling the Network Check Dialog

Based on TRC [R3090], calling the Network Check dialog is required when beginning use of the network. In the tutorial, this is implemented as follows.

First, when the tutorial is started up the Network Check dialog will be launched, requesting sign-in to PSN℠. Signing in here will enable sending activities in the game by using the NP Activity library. If this sign-in is cancelled, sending activities will not be possible until sign-in to PSN℠ is performed again in the tutorial. The Network Check dialog will also be called when moving from the Top Menu Scene to the Online Ranking Scene, requesting sign-in to PSN℠. In this tutorial, the youngest playable age is set to 12 and the Network Check dialog is called with 12 specified to the *`defaultAgeRestriction`* member of the *`param`* argument (`SceNetCheckDialogParam` structure).

3 COLLADA File Creation and Rendering

This tutorial sample uses Maya for model data creation with the `sce::SampleUtil::Graphics::Collada` class provided by the SampleUtil library for rendering in applications. The `sce::SampleUtil::Graphics::Collada` class is a library that supports the rendering of COLLADA scenes. This chapter explains the procedures, implementations, and notes from model data creation to rendering in applications.

The versions of the software used in this chapter are as follows.

- Autodesk Maya 2012
- PhyreEngine™ 3.2.5.0

COLLADA File Exporting

The following is an explanation of the basic procedure up to saving a model created with Maya in COLLADA format for handling with the `sce::SampleUtil::Graphics::Collada` class.

(1) Install the COLLADA export plug-in
(2) Allocate materials to model data
(3) Create the COLLADA file

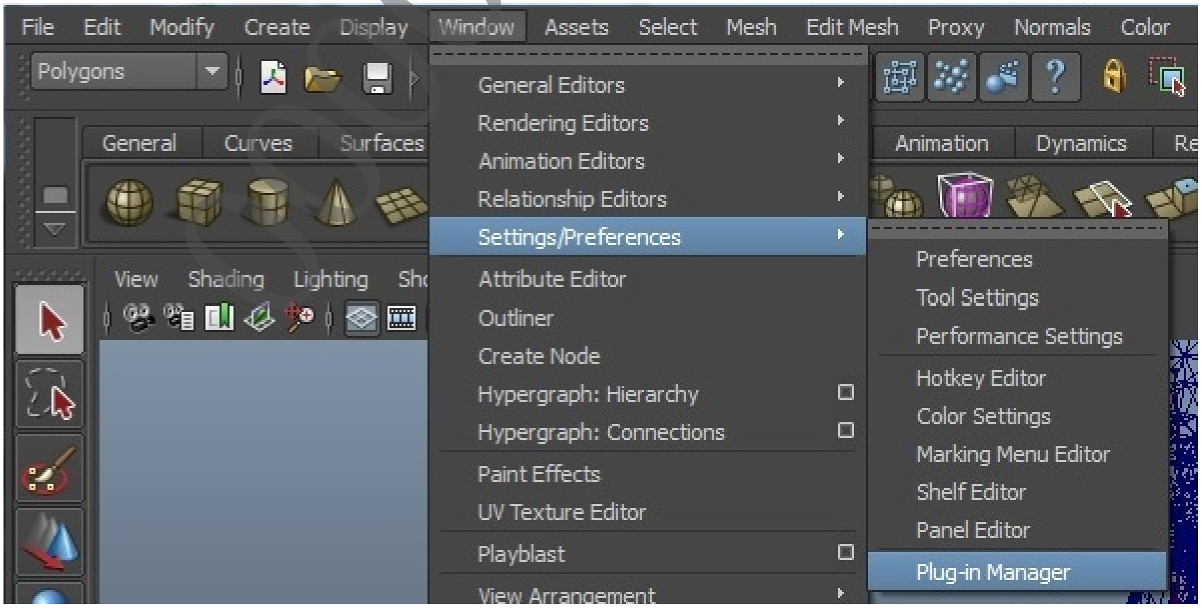
(1) Install the COLLADA export plug-in

PhyreEngine™ includes a COLLADA export plug-in for Maya, so place a check next to **Engines** in SDK Manager and download PhyreEngine™. When installing PhyreEngine™, place a check next to **Exporters Install.** After installing, start Maya.

After Maya has started, by performing the following settings it will be possible to export COLLADA files (explained later) and select materials that load proprietary shaders. By performing these settings, **COLLADA exporter** will be selectable in "(3) Create the COLLADA file" for **Files of type**, and **Colladafx Shader** will be selectable for the model data material type.

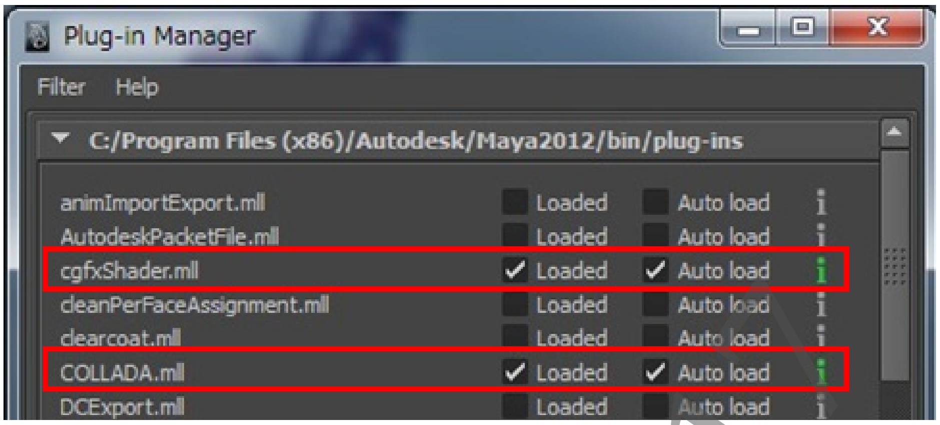
Select Window -> **Settings/Preferences** -> **Plug-in Manager**, and open the **Plug-in Manager** screen.

Figure 1 Window Menu



Place a check next to both **Loaded** and **Auto load** for **COLLADA.mll** and **cgfxShader.mll** in the **Plug-in Manager** screen.

Figure 2 Plug-in Manager Screen



animImportExport.mll	Loaded	Auto load
AutodeskPacketFile.mll	Loaded	Auto load
cgfxShader.mll	✓ Loaded	✓ Auto load
cleanPerFaceAssignment.mll	Loaded	Auto load
clearcoat.mll	Loaded	Auto load
COLLADA.mll	✓ Loaded	✓ Auto load
DCExport.mll	Loaded	Auto load

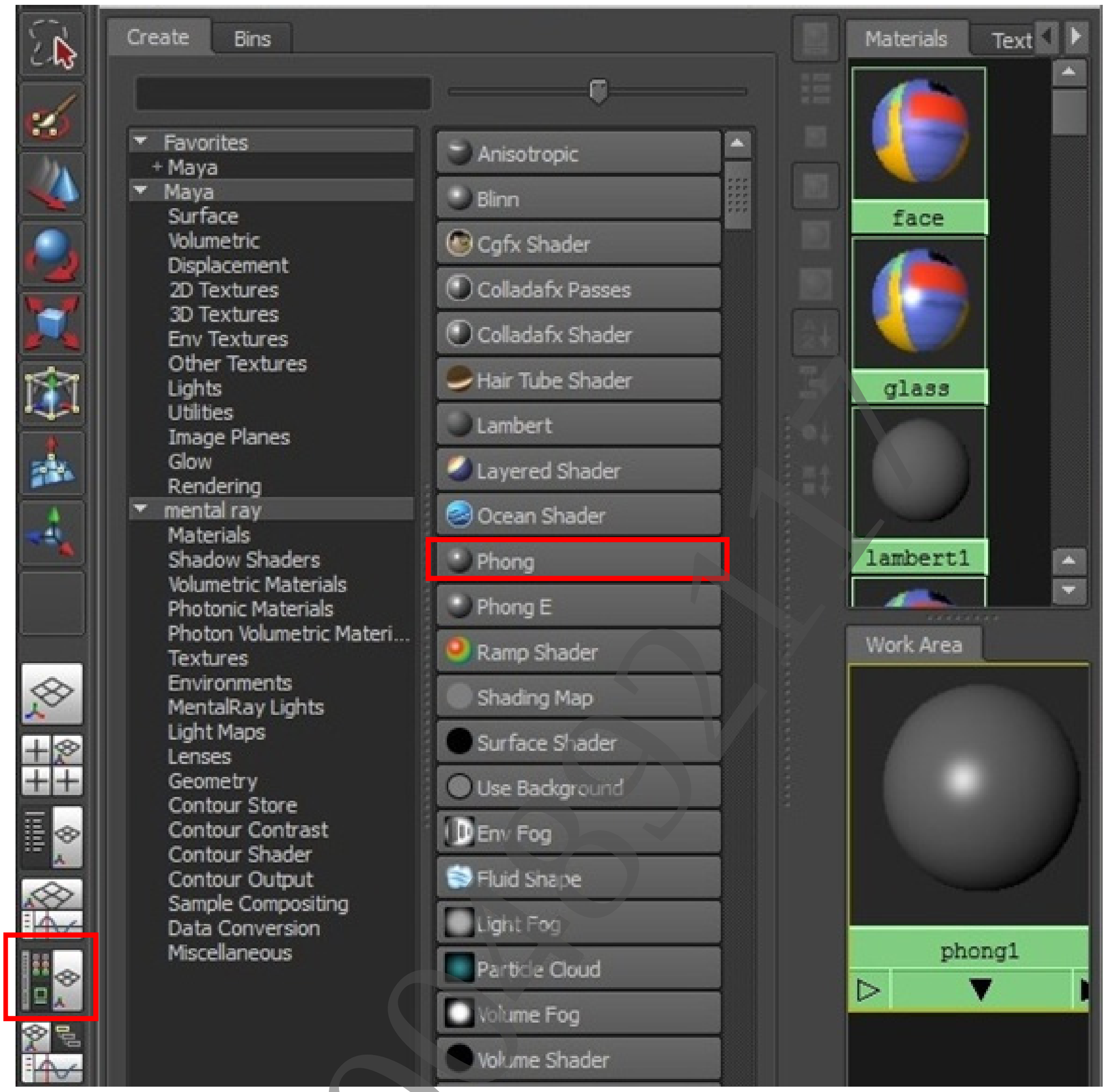
(2) Allocate materials to model data

The following is an explanation of the method for allocating materials to model data. Only the Phong material and Colladafx Shader material are compatible with the `sce::SampleUtil::Graphics::Collada` class.

The Phong material is a material that possesses reflection highlights that use three characteristics: diffusion, specular reflection, and gloss. First, the method for allocating the Phong material will be explained. For the Colladafx Shader material, refer to the "Method for Using Proprietary Shaders" item later in the document.

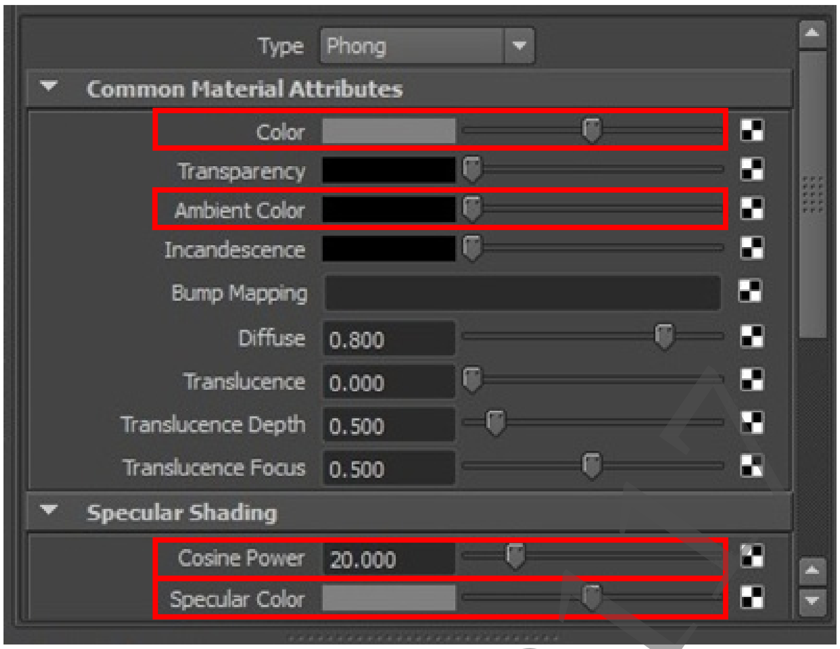
Since polygon meshes are the only geometry that can be displayed, create model data using polygon meshes. Select **Phong** in **Hypershade**, then allocate it to the created polygon mesh.

Figure 3 Selection of the Phong Material for Hypershade



Next, the settings will be performed for the Phong material using the Attribute Editor. The Phong material parameters that are compatible with the SampleUtil library are **Color**, **Ambient Color**, **Specular Color**, and **Cosine Power**, and a single color or 2D texture can be specified.

Figure 4 Phong Material Settings Compatible with the SampleUtil Library



Type: Phong

Common Material Attributes

- Color
- Transparency
- Ambient Color
- Incandescence
- Bump Mapping
- Diffuse: 0.800
- Translucence: 0.000
- Translucence Depth: 0.500
- Translucence Focus: 0.500

Specular Shading

- Cosine Power: 20.000
- Specular Color

(3) Create the COLLADA file

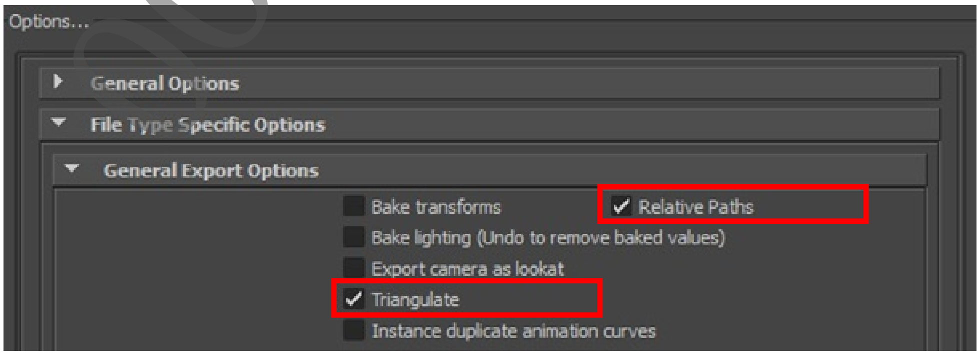
Perform the option setting when creating COLLADA files. Select **Window** -> **Export All** and open the **Export All** screen.

Currently the following data is compatible with the `sce::SampleUtil::Graphics::Collada` class; if other data is checked, it will not be used even if exported.

- Materials (Phong and Colladafx Shader only)
- Polygon meshes (triangles only)
- Joints and skins
- Scenes
- Animations (object transformation only)

In addition, since only triangle polygon meshes are supported, place a check next to **Triangulate** in **General Export Options**. If textures or proprietary shaders are loaded, development host computer absolute paths cannot be read, so place a check next to **Relative Paths** in **General Export Options** to use relative paths.

Figure 5 Export All Screen



Options...

General Options

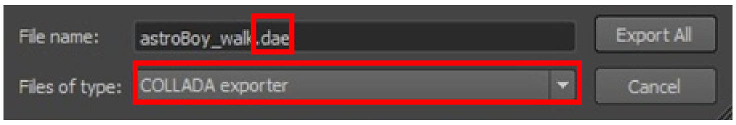
File Type Specific Options

General Export Options

- [ ] Bake transforms
- [✓] Relative Paths
- [ ] Bake lighting (Undo to remove baked values)
- [ ] Export camera as lookat
- [✓] Triangulate
- [ ] Instance duplicate animation curves

Set the extension to .dae for **File name**. Change **Files of type** to **COLLADA exporter,** and select **Export All**.

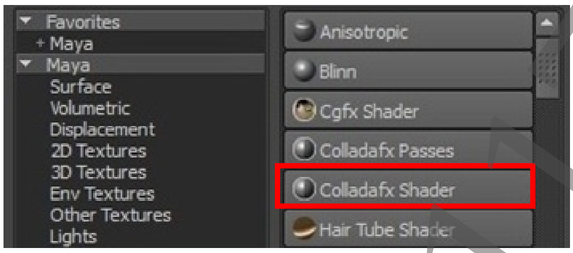
Figure 6 File Name Description and File Type Selection



Method for Using Proprietary Shaders

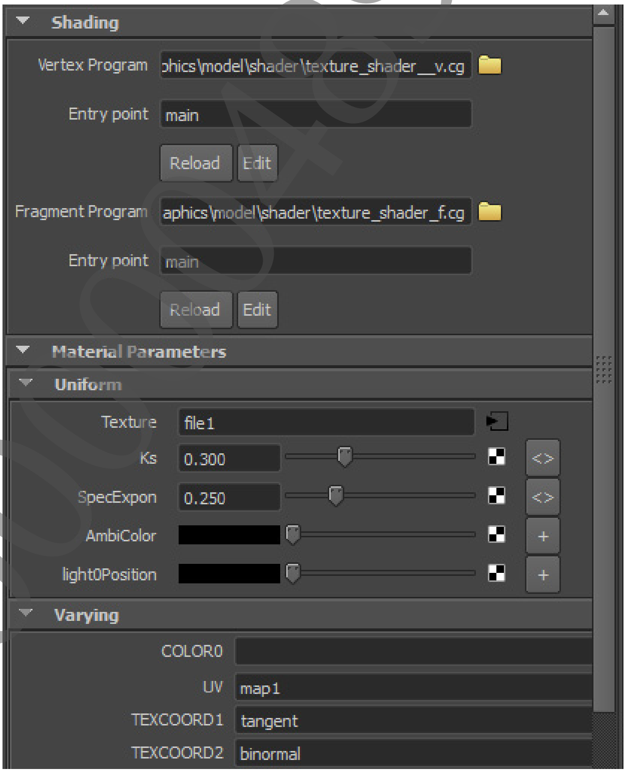
The following is an explanation for when vertex shader files and fragment shader files are proprietarily used. Select **Colladafx Shader** in **Hypershade**.

Figure 7 Colladafx Shader Material Selection in Hypershade



Perform the settings for the Colladafx Shader material using the Attribute Editor. Set the vertex shader file/fragment shader file paths and entry points, and set the values for the parameters to pass to the shaders.

Figure 8 Colladafx Shader Material Settings



Vertex Program: ɔhics\model\shader\texture_shader__v.cg
Entry point: main
Fragment Program: aphics\model\shader\texture_shader_f.cg
Entry point: main
Texture: file1
Ks: 0.300
SpecExpon: 0.250
AmbiColor
light0Position
COLOR0
UV: map1
TEXCOORD1: tangent
TEXCOORD2: binormal

In order for the runtime to load the shader during execution, already compiled GXP files must exist in the same directory as the Cg files. In the example shown in Figure 8, the path for the shader in the COLLADA file is ./shader/texture_shader_v.cg, so an already compiled ./shader/texture_shader_v.gxp shader file must be prepared in advance for this file. At the same time, ./shader/texture_shader_f.gxp must also be prepared for ./shader/texture_shader_f.cg. In this tutorial, Cg files are converted to GXP files using the batch file called build.bat. Refer to %PSP2_SDK_SAMPLES%/data/graphics/model/shade/build.bat.

Confirm the Path Described in the Created COLLADA File

If texture files or shader files are loaded, there will be paths similar to the following inside the output COLLADA file. The paths in the COLLADA file must be usable upon executing on DevKit and the Testing Kit (TestKit). Specify absolute paths that can be read by the DevKit/TestKit or relative paths from the COLLADA file. The following is an example of a path descriptor inside the COLLADA file.

Texture file path description example

```
<library_images>
<image id="file1" name="file1">
<init_from>./texture/duckCM.bmp</init_from>
```

Shader file path description example

```
<profile_CG platform="PC-OGL">
<include sid="include" url="./shader/texture_shader_v.cg"/>
<include sid="include2" url="./shader/texture_shader_f.cg"/>
```

Reference Material

Refer to the following website for the COLLADA specifications.

- COLLADA - 3D Asset Exchange Schema
  http://www.khronos.org/collada/

(The above reference destination has been confirmed as of February 2, 2015. Note that pages may have been subsequently moved or its contents modified.)

Procedure for Passing a Model to the Renderer and Displaying

Here, the basic procedure for the rendering process using the `sce::SampleUtil::Graphics::Collada` class will be explained with the following process flow.

(1) Initialize the `sce::SampleUtil::Graphics::Collada` class
(2) Load the COLLADA file
(3) Create a scene instance
(4) Obtain the scene
(5) Specify the animation time
(6) Render the scene instance
(7) Perform termination processing for the animation
(8) Discard the scene instance
(9) Discard the loaded COLLADA file
(10) Perform termination processing for the `sce::SampleUtil::Graphics::Collada` class

(1) Initialize the `sce::SampleUtil::Graphics::Collada` class

Call `sce::SampleUtil::Graphics::Collada::ColladaLoader::initialize()` and initialize the `sce::SampleUtil::Graphics::Collada::ColladaLoader` class.

```
sce::SampleUtil::Graphics::GraphicsContext *graphicContext;
sce::SampleUtil::Graphics::Collada::ColladaLoader loader;
loader.initialize(graphicContext);
```

(2) Load the COLLADA file

Call `sce::SampleUtil::Graphics::Collada::ColladaLoader::load()`, load the COLLADA file, and save it in the `sce::SampleUtil::Graphics::Collada::Collada` class.

```
sce::SampleUtil::Graphics::Collada::Collada colladaFile;
loader.load(colladaFile, colladaBoyPath);
```

(3) Create a scene instance

Call `sce::SampleUtil::Graphics::Collada::InstanceVisualScene::initialize()` and create a scene instance.

```
sce::SampleUtil::Graphics::Collada::InstanceVisualScene instanceVisualScene;
instanceVisualScene.initialize(colladaFile.getVisualScene());
```

(4) Obtain the scene

Call `sce::SampleUtil::Graphics::Collada::AnimationPlayer::initialize()` and obtain the scene.

```
sce::SampleUtil::Graphics::Collada::AnimationPlayer      animPlayer;
animPlayer.initialize(&instanceVisualScene, colladaFile.getAnimation());
```

(5) Specify the animation time

Call `sce::SampleUtil::Graphics::Collada::AnimationPlayer::setTime()` and specify the animation time for the scene instance. By calling this function, the scene will be reproduced for the specified time.

```
animPlayer.setTime(0.1f);
```

(6) Render the scene instance

Call `sce::SampleUtil::Graphics::Collada::InstanceVisualScene:draw()` and render the scene instance.

```
//Set projection matrix and view matrix
Vectormath::Aos::Matrix4 projectionMatrix = …;
Vectormath::Aos::Matrix4 viewMatrix = …;

loader.getDefaultParams()->setProjectionMatrix(projectionMatrix);
loader.getDefaultParams()->setViewMatrix(viewMatrix);

//Set point light position and color
Vectormath::Aos::Vector3 lightPosition = …;
Vectormath::Aos::Vector3 lightColor = …;

loader.getDefaultParams()->setLight(lightPosition, lightColor);

//Set matrix for converting the coordinate system of the InstanceVisualScene to
be rendered and its scene to world coordinates
sce::Vectormath::Simd::Aos::Matrix4 sceneLocalToWorld = …;
```

```
instanceVisualScene.draw(graphicContext, sceneLocalToWorld);
```

(7) Perform termination processing for the animation

Call `sce::SampleUtil::Graphics::Collada::AnimationPlayer::finalize()` and perform termination processing for the animation.

```
animPlayer.finalize();
```

(8) Discard the scene instance

When the scene instance is no longer needed, call `sce::SampleUtil::Graphics::Collada::InstanceVisualScene::finalize()` and discard the scene instance.

```
instanceVisualScene.finalize();
```

(9) Discard the loaded COLLADA file

When the loaded COLLADA file is no longer needed, call `sce::SampleUtil::Graphics::Collada::Collada::finalize()` and discard the COLLADA file. Before discarding the COLLADA file, all `InstanceVisualScenes` created from this COLLADA file must be discarded.

```
colladaFile.finalize();
```

(10) Perform termination processing for the `sce::SampleUtil::Graphics::Collada` class

Call `sce::SampleUtil::Graphics::Collada::ColladaLoader::finalize()` and perform termination processing. Before performing termination processing, all of the COLLADA files created with this `SimpleRenderer` must be discarded.

```
loader.finalize();
```

4 Design of the Shooting Game

This tutorial is a first-person shooting game where the player fights off pursuing monsters in a three-dimensional space by shooting them with a gun. This tutorial consists of several scenes. The scene transitions in this tutorial and the detail of the scenes are explained below.

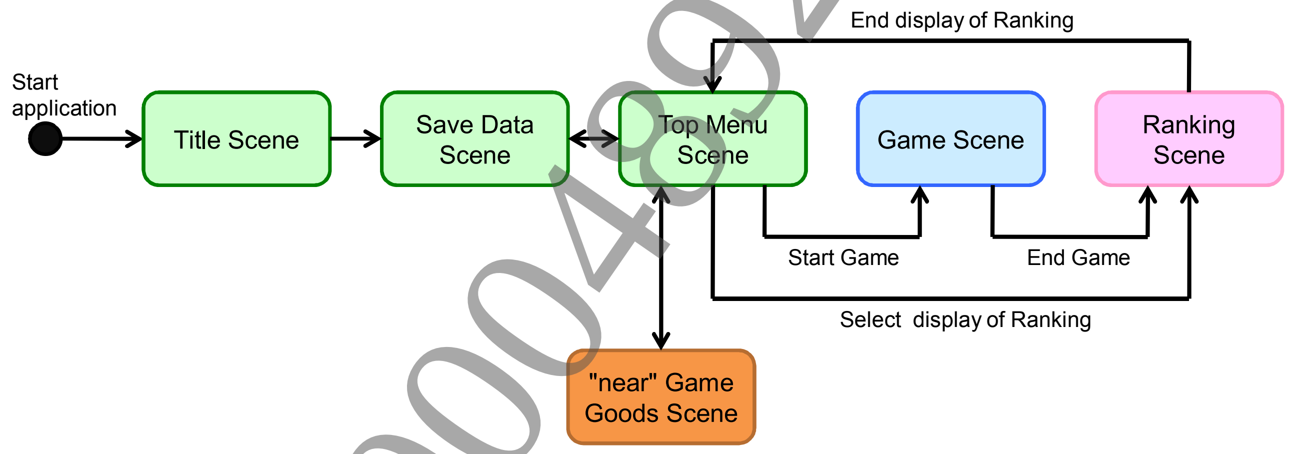
Scene Transitions

This tutorial consists of the following categories of scenes.

- Title Scene
- Save Data Scene
- Top Menu Scene
- Game Scene
- Local Ranking Scene
- "near" Game Goods Scene
- Online Ranking Scene

The overall sequence of scenes is as follows.

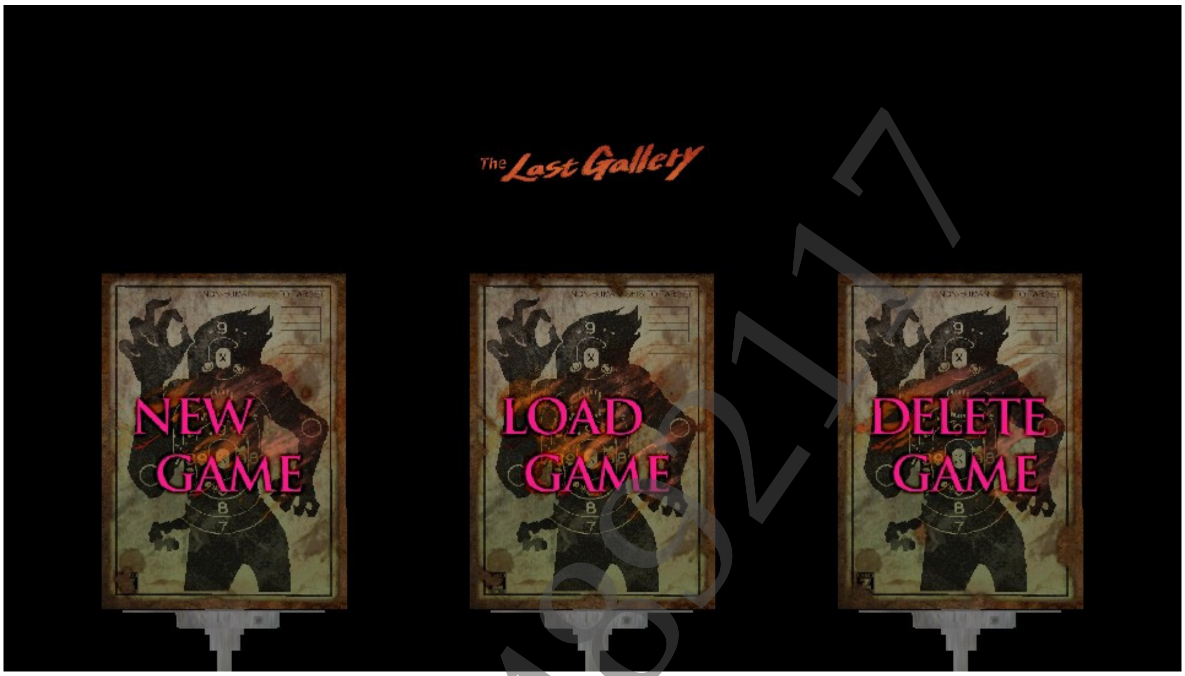
Figure 9 Transitions of Scenes of TRC Compliant Shooting Game Tutorial



Title Scene, Save Data Scene and Top Menu Scene

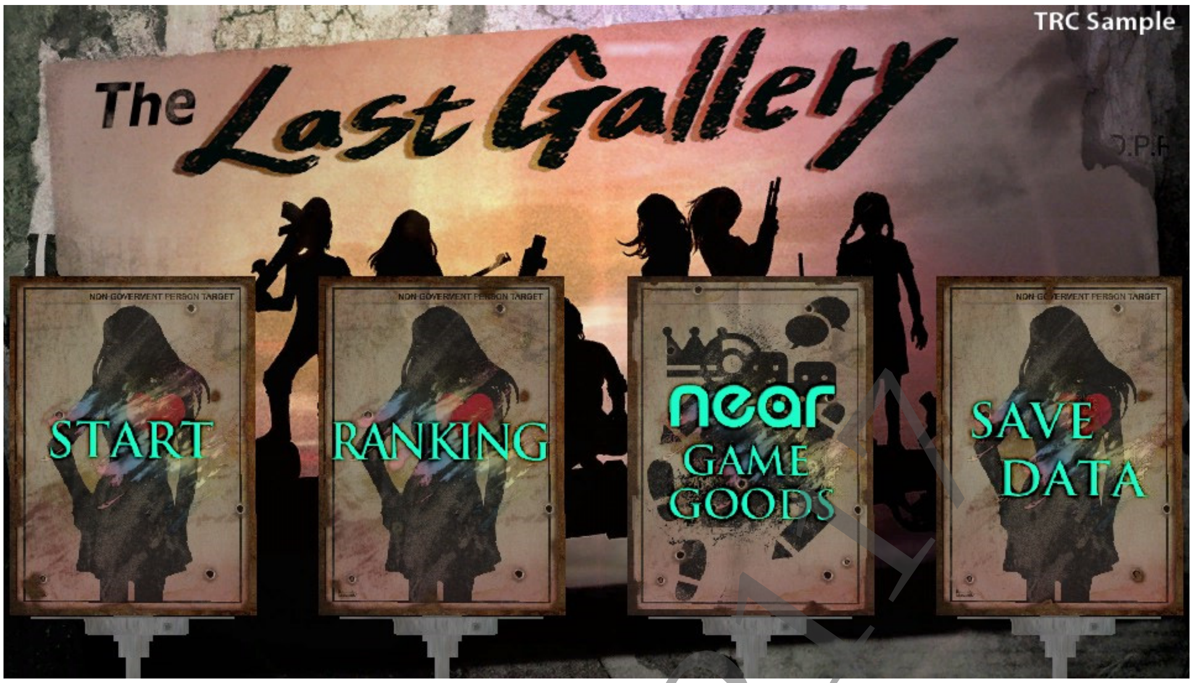
Following the transition from the Title Scene to the Save Data Scene, a screen for selecting which save data to use in this tutorial is displayed. The **LOAD GAME**, **NEW GAME** and **DELETE GAME** choices appear. To load already existing save data, select **LOAD GAME**, and to create new save data, select **NEW GAME**. Select **DELETE GAME** when deleting existing save data.

Figure 10 Save Data Scene



In this tutorial, once one save data that will be used is set, this save data will be used until different save data is selected/created. For the detailed specifications of the save data of this tutorial, refer to the Chapter 6 "Using the Save Data Utility". Once the save data to be used has been set, the tutorial moves to the following Top Menu Scene depicted below.

Figure 11 Top Menu Scene

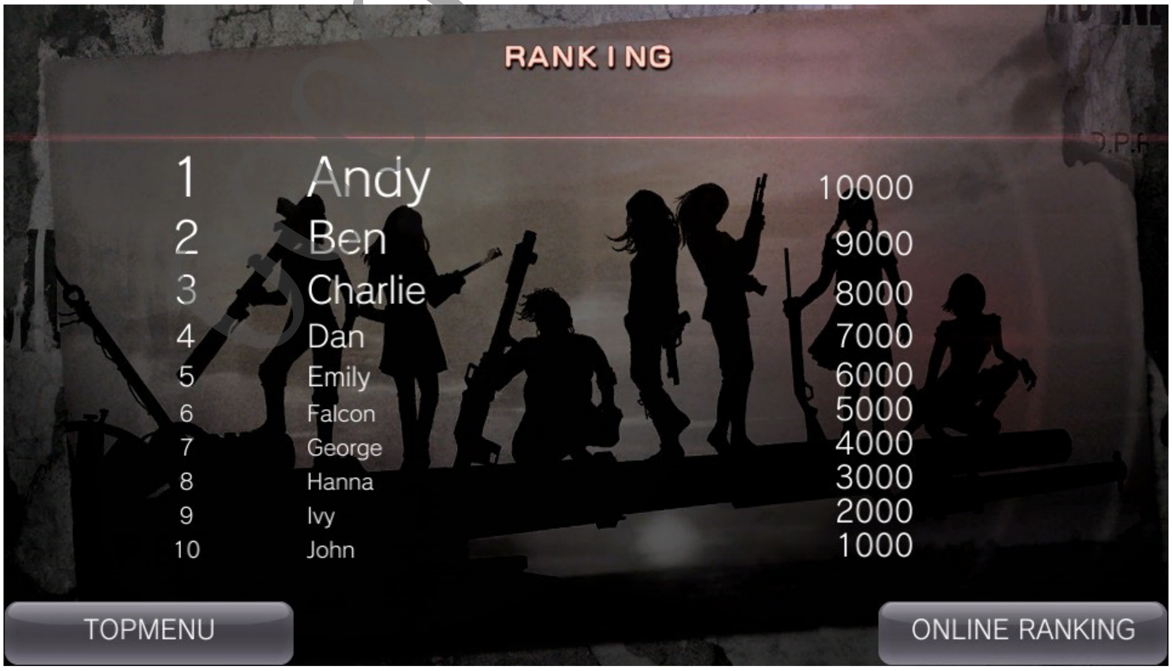


The **START**, **RANKING**, **"near" GAME GOODS** and **SAVE DATA** choices appear. If **START** is selected, the tutorial moves to the Game Scene, and if **RANKING** is selected, it moves to the Local Ranking Scene. If **SAVE DATA** is selected, the tutorial moves to the Save Data Scene. **"near" GAME GOODS** will be grayed out until the game is played; moving to "near" Game Goods Scene will be possible once the game is played.

Local Ranking Scene

In this tutorial, only ranking information is saved to the save data. The ranking information saved to the save data selected in the Save Data Scene is displayed as follows in the Local Ranking Scene.

Figure 12 Local Ranking Scene



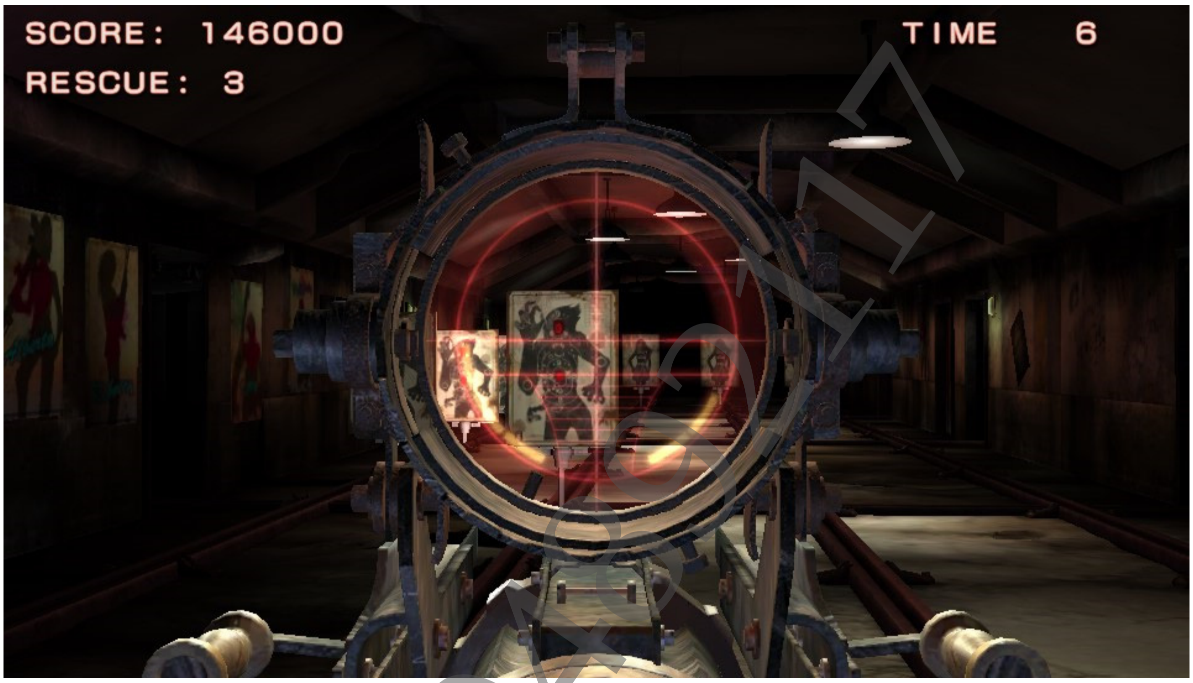
1	Andy	10000
2	Ben	9000
3	Charlie	8000
4	Dan	7000
5	Emily	6000
6	Falcon	5000
7	George	4000
8	Hanna	3000
9	Ivy	2000
10	John	1000

From the Local Ranking Scene, move to the Top Menu Scene by tapping on **Top Menu**, and move to the Online Ranking Scene by tapping on **Online Ranking**.

Game Scene

Following the transition to the Game Scene, the shooting game begins in which the player fights off pursuing monsters in a three-dimensional space by shooting them with a gun. The player can use the target sight on the screen to aim and shoot at target objects. Since head shots give extremely high scores, the player can get more points by using the zoom to take better aim at the target.

Figure 13 Game Scene



The main control methods for the controller in the Game Scene are as follows.

- R button: Shoots bullets
- L button: Reloads bullets
- Left stick: Moves the target sight
- Rear touch pad: Adjusts the zoom amount of the target sight by Drug of finger on the left side of the rear touch pad

In addition, the motion sensor is used for the movement of the target sight. Through this, control adjustments with the controller are possible where the motion sensor movement moves the target sight.

Reset the position of the target sight with the triangle button.

Also, for debugging purposes, the L button, R button, and square button can be pressed simultaneously which will reread the Lua file where the Game Scene settings are recorded and restart the Game Scene with all settings reset. This feature can be used to easily make changes such as adjusting the appearance pattern of the target object.

"RESCUE" points are accumulated by avoiding hitting girls asking for help with the bullets and only shooting the monsters. These "RESCUE" points can be used in the "near" Game Goods Scene described later.

If Game Goods have been downloaded in the "near" Game Goods Scene, these Game Goods will be used when the Game Scene begins, and treasure boxes will appear in the Game Scene. The player can get bonuses by hitting the treasure boxes with his/her bullets.

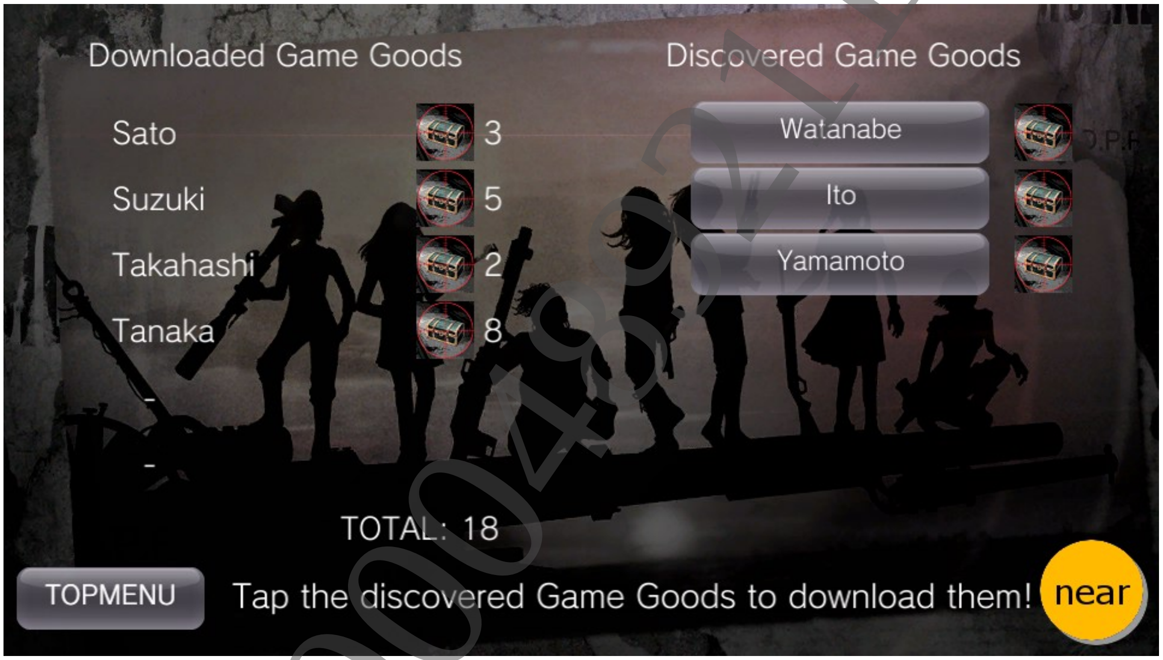
For the implementation/specification details, refer to the 5. "Design and Configuration of Game Scenes" chapter. If, when the time becomes 0 and the shooting game part completes, the final score is among the top 10, the IME dialog starts up. The character string input in the IME dialog is used as the user name in the local rankings. Following the IME dialog input, the tutorial moves to the Local Ranking Scene.

"near" Game Goods Scene

In the "near" Game Goods Scene, players can check downloaded Game Goods and discovered Game Goods.

"Discovered Game Goods" are downloadable Game Goods discovered by starting up the "near" application; by tapping each Game Goods item, it is possible to start up the "near" application and to download the Game Goods. Downloaded Game Goods are used in the Game Scene, and the number of treasure boxes appearing in the game scene is determined by the treasure boxes contained in each Game Goods item. For details on implementation/specifications, refer to the 8. "Specifications and Implementation of Features Using the "near" System" chapter.

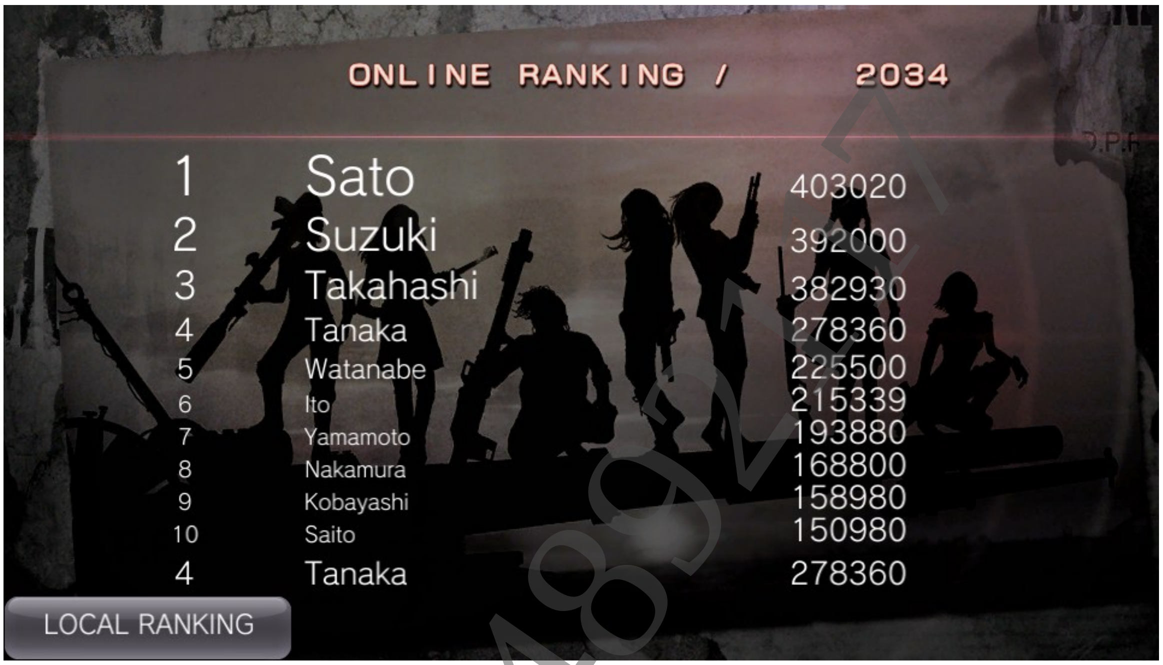
Figure 14 "near" Game Goods Scene



Online Ranking Scene

The Online Ranking Scene allows checking the online ranking implemented by using the NP ScoreRanking library. When moving to the Online Ranking Scene, players will be asked whether they wish to register their current score in the online ranking; by answering **Yes** here, players will be able to register their score in the online ranking. Registration in the online ranking is performed by using Sony Entertainment Network account names.

Figure 15 Online Ranking Scene



ONLINE RANKING / 2034

1	Sato	403020
2	Suzuki	392000
3	Takahashi	382930
4	Tanaka	278360
5	Watanabe	225500
6	Ito	215339
7	Yamamoto	193880
8	Nakamura	168800
9	Kobayashi	158980
10	Saito	150980
4	Tanaka	278360

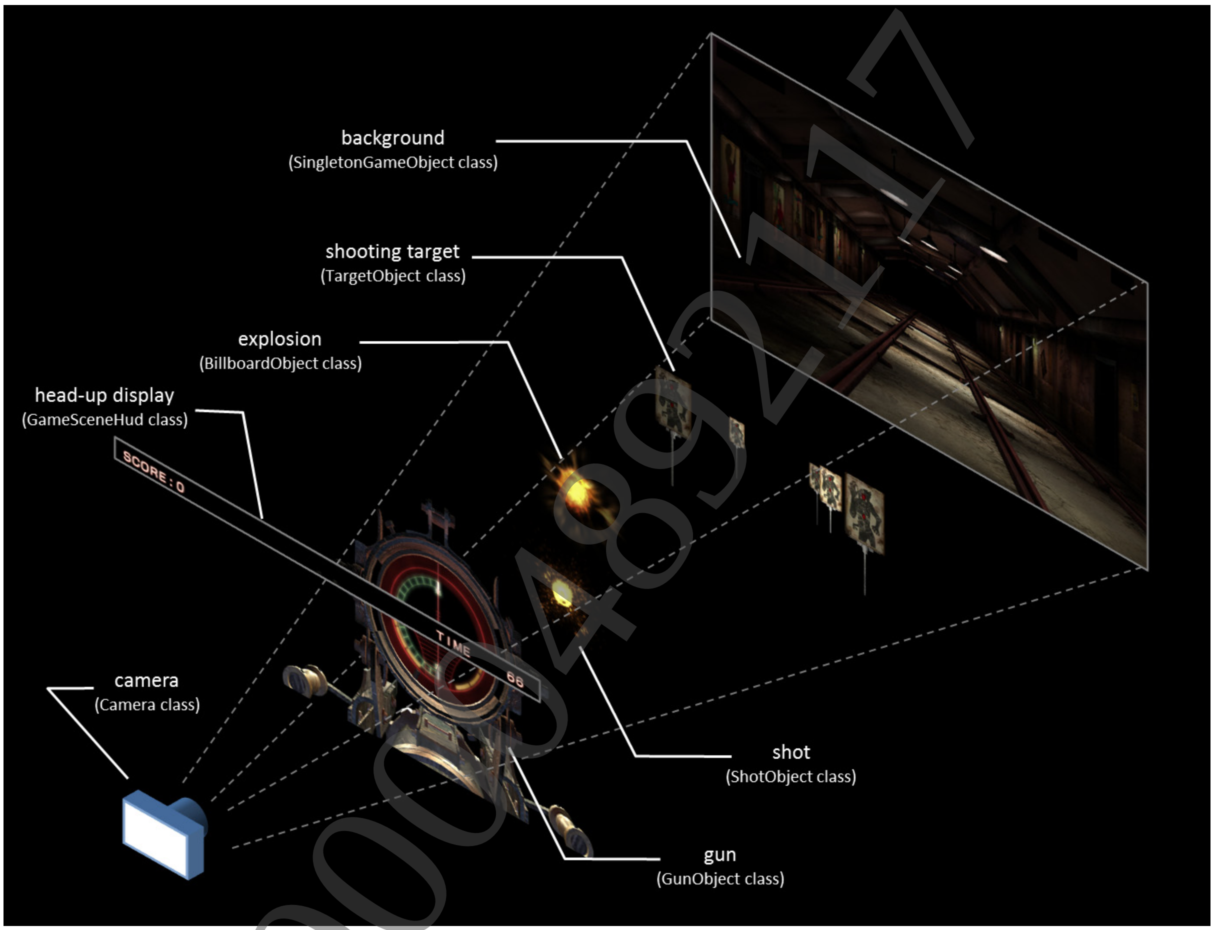
5 Design and Configuration of Game Scenes

The scene with the most complex configuration in the shooting game is the Game Scene. This chapter explains the elements comprising the Game Scene and how classes were designed for those elements.

Objects and Classes Comprising the Game Scene

Figure 16 shows the objects and classes comprising the Game Scene as viewed from the game screen.

Figure 16 Objects and Classes Comprising the Game Scene as Viewed from the Game Screen

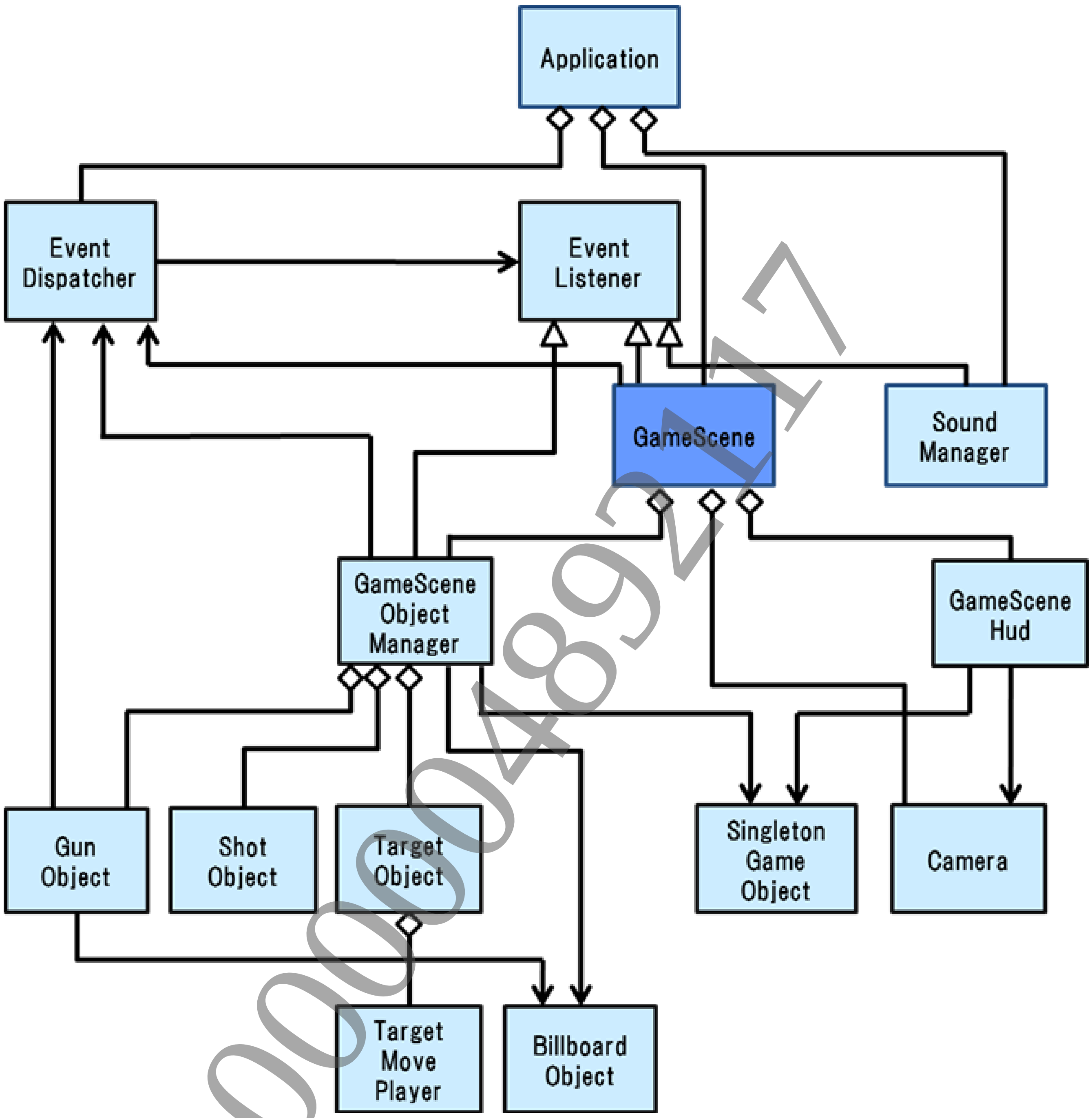


The main elements and classes shown in Figure 16 and that are introduced in the Game Scene are shown below.

- Gun: `GunObject` class
- Shooting target: `TargetObject` class
- Bullet: `ShotObject` class
- Explosion: `BillboardObject` class
- Display of time, score, etc.: `GameSceneHud` class
- Viewpoint: `Camera` class
- Background: `SingletonGameObject` class

Figure 17 shows the above classes and how they are related to the Game Scene.

Figure 17 `GameScene` Class Configuration



The Game Scene is implemented as a `GameScene` class, which manages various resources and game state transitions in the Game Scene.

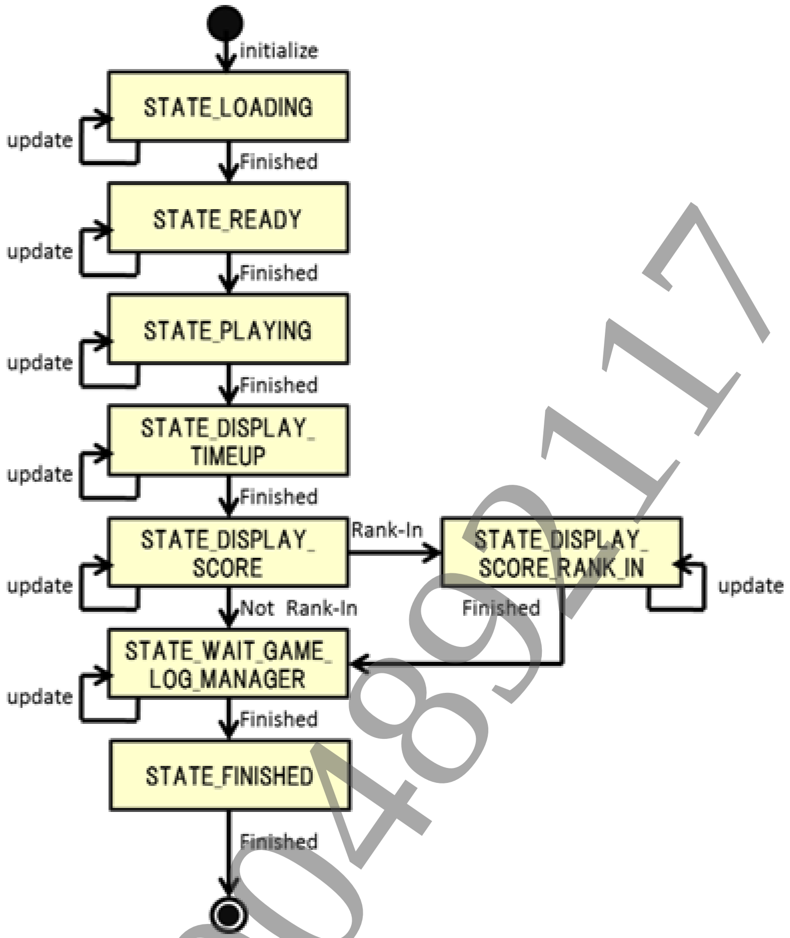
The `EventListener` class and `EvendDispatcher` class are the base classes, and they currently receive events that generate sounds, the trophy feature, and game log. Refer to the "Audio Processing Design and Implementation" section in this chapter for more information. The `Camera` class manages the viewpoint. `SingletonGameObject` is a class for handling the drawing object of which only one can be present on the game screen such as a background. `BillboardObject` is used to paste and render effects - such as, an explosion - on a single board onscreen.

Details of other classes are presented below.

GameScene Class State Transitions

Figure 18 shows the state transitions of the `GameScene` class.

Figure 18 `GameScene` State Transition Diagram



The `GameScene` class has the following eight states.

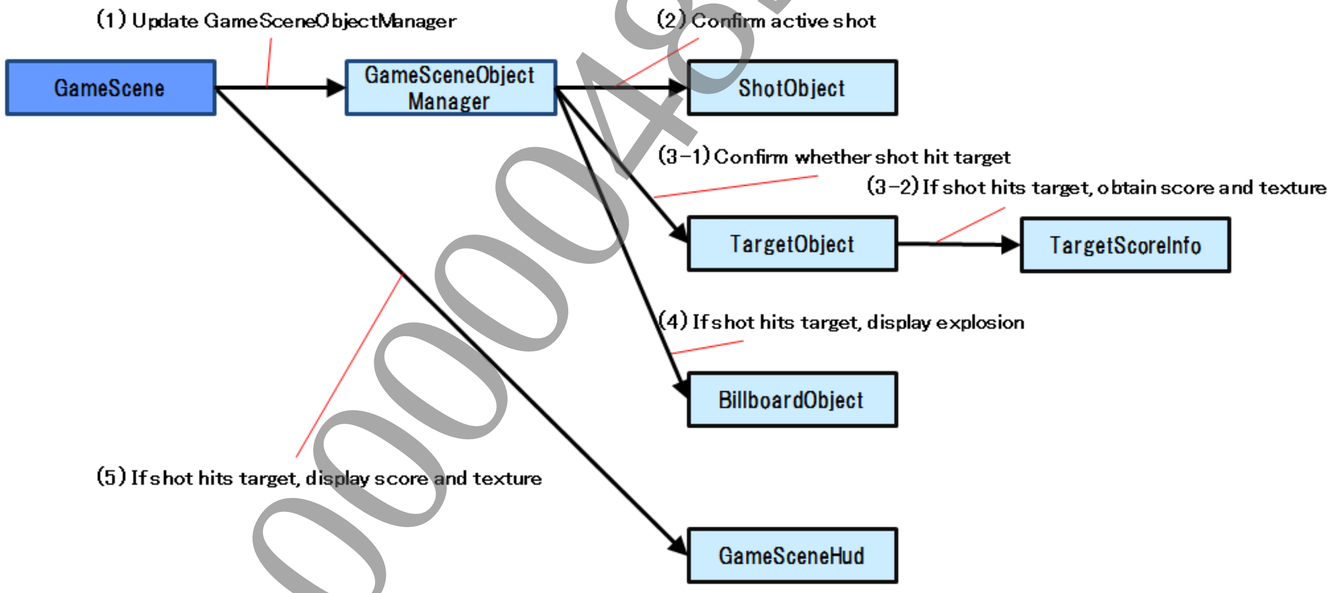
- STATE_LOADING
  When `GameScene::initialize()` is called, the `GameScene` class enters STATE_LOADING state. The resources that are used by the game are loaded in STATE_LOADING state. The NOW LOADING image is displayed on-screen at this time. The class automatically transitions to STATE_READY state when the loading of resources has completed.
- STATE_READY
  After the transition to STATE_READY state, the READY? image is displayed on-screen. After a fixed amount of time has elapsed, the class automatically transitions to STATE_PLAYING state, and the shooting game begins.
- STATE_PLAYING
  After the transition to STATE_PLAYING state, the class will remain in this state throughout the rest of the shooting game. When the remaining game time becomes 0, the class will automatically transition to STATE_DISPLAY_TIMEUP state.
- STATE_DISPLAY_TIMEUP
  After the transition to STATE_DISPLAY_TIMEUP state, the TIME UP! image is displayed on-screen and the shooting game ends. After a fixed amount of time has elapsed, the class will automatically transition to STATE_DISPLAY_SCORE state.

- STATE_DISPLAY_SCORE
  After the transition to STATE_DISPLAY_SCORE state, the game score is displayed on-screen. The score display is ended by pressing the START button. If the score is among the top 10 rankings, the class transitions to STATE_DISPLAY_SCORE_RANK_IN state. Otherwise, it transitions to STATE_WAIT_GAME_LOG_MANAGER state.
- STATE_DISPLAY_SCORE_RANK_IN
  After the transition to STATE_DISPLAY_SCORE_RANK_IN state, the IME dialog is called. After the player has input the name, the class will transition to STATE_WAIT_GAME_LOG_MANAGER state.
- STATE_WAIT_GAME_LOG_MANAGER
  After the transition to STATE_WAIT_GAME_LOG_MANAGER state, trophies are checked, LiveArea™ is updated, and save processing is carried out, in that order. After completing processing, the class will transition to STATE_FINISHED state.
- STATE_FINISHED
  Whether the state should transition to finished state is determined by calling `GameScene::isFinished()`. After the game is finished, `GameScene::finalize()` is called to perform finalization processing.

GunObject, TargetObject, ShotObject and BillboardObject Interactions

The sequence of interactions among the `GunObject`, `TargetObject`, `ShotObject` and `BillboardObject`, which comprise the main elements of the game screen, result in a bullet being fired from a gun, the bullet impacting the shooting target, and an explosion occurring. The complicated part of this processing sequence is collision detection. Figure 19 shows how collision detection processing is performed.

Figure 19 Collision Detection Processing Flow


(1) Update GameSceneObjectManager
(2) Confirm active shot
GameScene
GameSceneObject Manager
ShotObject
(3-1) Confirm whether shot hit target
(3-2) If shot hits target, obtain score and texture
TargetObject
TargetScoreInfo
(4) If shot hits target, display explosion
BillboardObject
(5) If shot hits target, display score and texture
GameSceneHud

The following is the detailed processing flow of collision detection.

(1) `GameScene` updates `GameSceneObjectManager`.

(2) For every `ShotObject` saved by `GameSceneObjectManager`, use `ShotObject::m_isActive()` to determine whether its state is Active.

(3) If a `ShotObject` in Active state is found, call `TargetObject::isHit()` for every `TargetObject` that `GameScene` has. The decision of whether a `TargetObject` is hit is made based on its own position and the position of the `ShotObject` specified by the argument of `TargetObject::isHit()`. If it is determined that `TargetObject` was hit, get the score and texture for displaying it on-screen from `TargetScoreInfo` and pass them to `GameScene`. At this time, the `TargetObject` itself starts the animation in which the shooting target explodes.

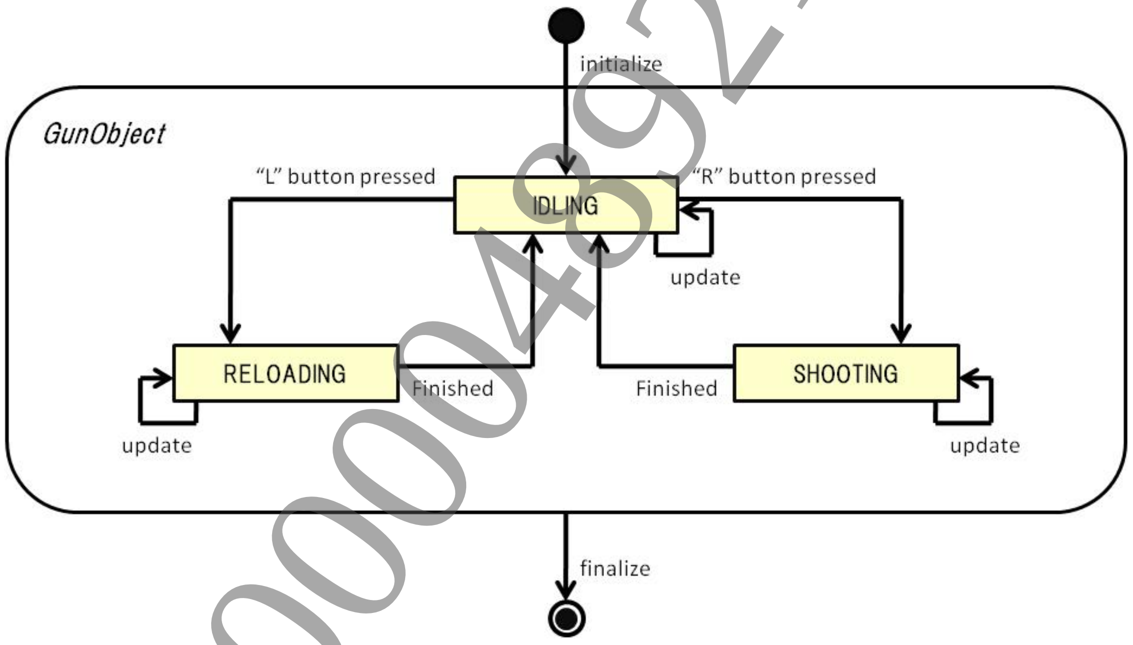
(4) If a hit was determined in step (3), `GameSceneObjectManager` generates a `BillboardObject` and calls `startExplosion()`. The position where the explosion occurs can be specified as an argument of `startExplosion()` at this time. As a result, the animation of the explosion will start at the specified position.

(5) Following the processing in step (4), `GameScene` calls `GameSceneHud::onHit()` for `GameSceneHud`. The position, score and texture can be specified as arguments of `onHit()`. `GameSceneHud` starts the display of the score and texture at the specified position.

GunObject Class State Transitions

`GunObject` corresponds to a gun on the game screen. It manages the zoom state and number of remaining shots and performs operations such as zooming, shooting and reloading. Figure 20 shows the state transitions of the `GunObject` class.

Figure 20 `GunObject` State Transitions



`GunObject` has the following three states.

- IDLING
  After `GunObject` is created, `GunObject::initialize()` is called and the class enters IDLING state. The class transitions through IDLING state on the way to performing each operation. Operations such as zooming in and out, shooting bullets and reloading bullets can be performed when `GunObject` is in IDLING state. All of these operations are performed within `GunObject::update()`. After the game is finished, `GunObject::finalize()` is called to perform `GunObject` finalization processing.
- RELOADING
  If the L button is pressed or the R button is pressed when the remaining number of bullets is 0, the class transitions to RELOADING state and bullets are reloaded. After reloading is completed, the class returns to IDLING state.

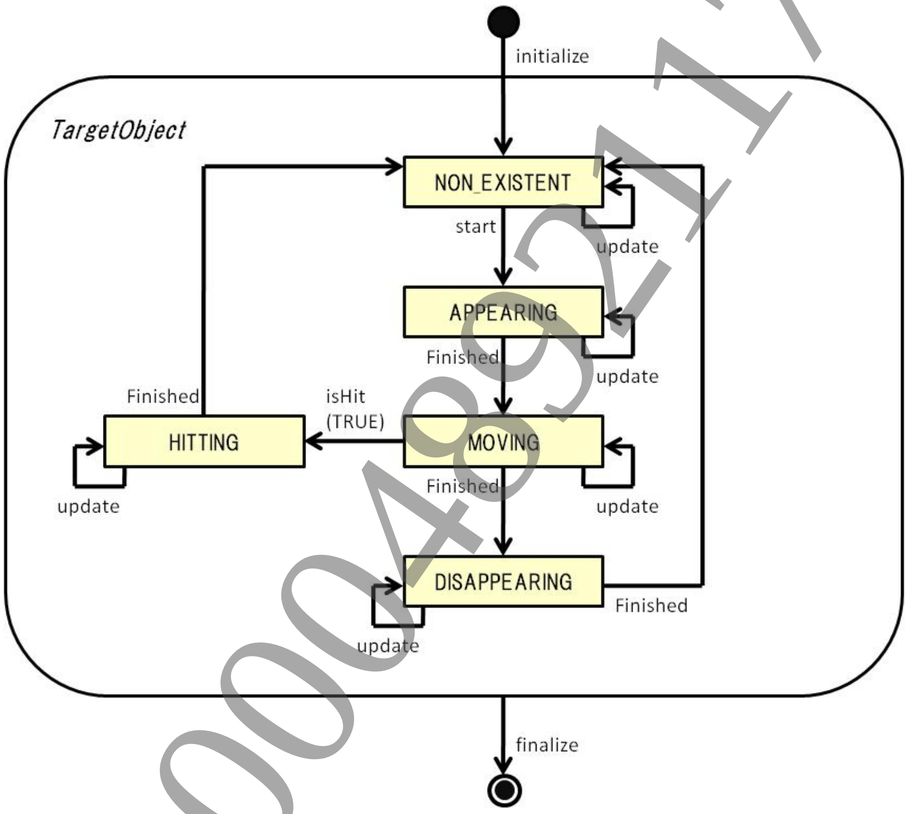
- SHOOTING
  The transition of `GunObject` to SHOOTING state is triggered by pressing the R button. Following the transition to SHOOTING state, bullets are shot and the animation showing the muzzle of the gun emitting fire begins. For details about this processing, refer to `ShotObject`. After bullets are shot, the class transitions to IDLING state after the time period set in Config has elapsed.

TargetObject Class State Transitions

`TargetObject` corresponds to a shooting target on the game screen. It manages the position and operational state of the shooting target and performs processing such as animation, movement and shot collision detection. `TargetMovePlayer` is a class that performs processing for moving the target.

Figure 21 shows the state transitions of the `TargetObject` class.

Figure 21 `TargetObject` State Transitions



`TargetObject` performs processing in the following five states.

- NON_EXISTENT
  After `TargetObject` is created, `TargetObject::initialize()` is called, internal initialization is performed, and `TargetObject` enters NON_EXISTENT state. During NON_EXISTENT state, the shooting target has not appeared yet. Calling `TargetObject::start()` on a `TargetObject` in NON_EXISTENT state will cause `TargetObject` to transition to APPEARING state which starts the animation in which the target makes its entrance onto the game screen.
- APPEARING
  Following the transition to APPEARING state, the animation starts in which the shooting target appears on the game screen. After the animation ends, `TargetObject` automatically transitions to MOVING state.

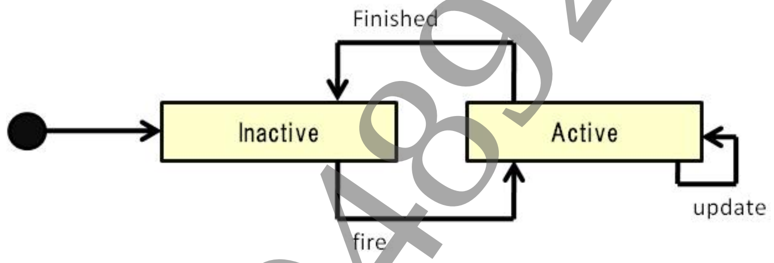
- MOVING
  `TargetObject` will stay in MOVING state while the shooting target is moving on the screen. `TargetObject::isHit()` can be called to perform collision detection only from MOVING state. For details about collision detection, refer to the "GunObject, TargetObject, ShotObject and BillboardObject Interactions" section. If a collision is detected by `TargetObject::isHit()`, `TargetObject` will transition to HITTING state. After a fixed period of time has elapsed, `TargetObject` will transition from MOVING state to DISAPPEARING state.
- HITTING
  Following the transition to HITTING state, the animation starts in which the target explodes. When the animation ends, `TargetObject` automatically transitions to NON_EXISTENT state.
- DISAPPEARING
  Following the transition to DISAPPEARING state, the animation starts in which the shooting target disappears from the game screen. When the animation ends, `TargetObject` will transition from DISAPPEARING state to NON_EXISTENT state.

ShotObject Class State Transitions

`ShotObject` corresponds to a bullet that was fired from a gun on the game screen. It manages the bullet position, speed and elapsed time.

Figure 22 shows `ShotObject` state transitions.

Figure 22 `ShotObject` State Transitions



`ShotObject` performs processing in the following two states.

- Inactive
  When `ShotObject` is created, internal initialization is performed, and then `ShotObject` transitions to Inactive state. When `ShotObject::fire()` is called, `ShotObject` transitions to Active state.
- Active
  Following the transition to Active state, `ShotObject` returns to Inactive state if the bullet strikes a target or wall or after a fixed amount of time has elapsed.

GameSceneHud Class

`GameSceneHud` is a class for handling scores, time remaining, and other information that is displayed on the game screen in a unified manner.

`GameSceneHud::initialize()` is called to perform internal initialization. Afterwards, `GameSceneHud::update()` is called to specify the score and time remaining and then `GameSceneHud::render()` is called to display text information such as the score and time at the top of the screen.

If `GameSceneHud::renderDisplayScore()` is called after a call of `GameSceneHud::update()`, the score string will be displayed in the middle of the screen. This is used to display the score string after the shooting game ends.

If `GameSceneHud::renderDisplayPause()` is called after a call of `GameSceneHud::update()`, a character string `"PAUSE"` will be displayed in the middle of the screen. This is used when the Game Scene is suspended.

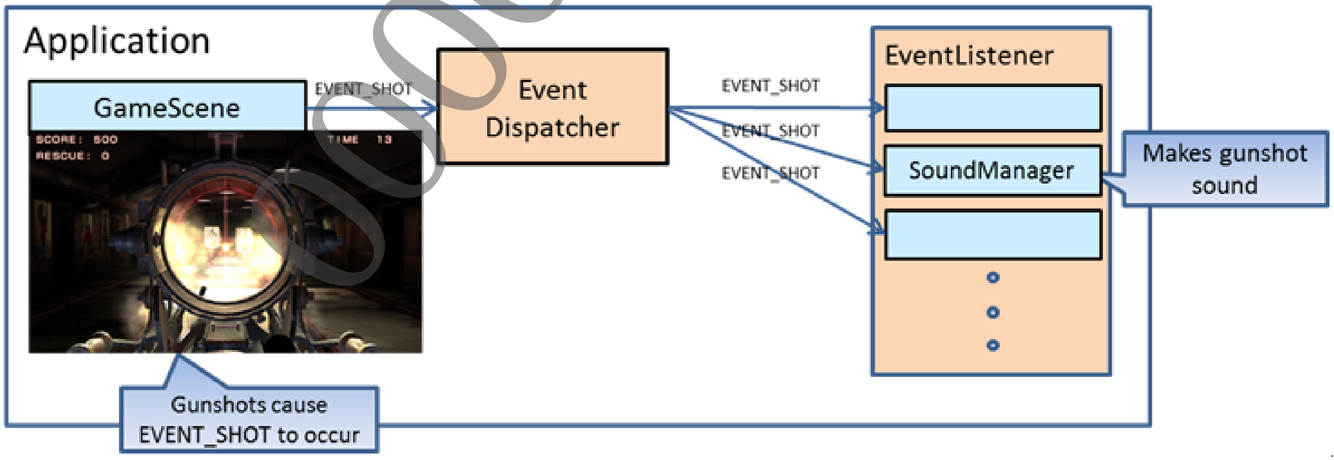
In addition, by calling `GameSceneHud::onHit()` the score will be displayed when a bullet hits the shooting target.

Audio Processing Design and Implementation

It is possible for audio processing to be handled for all scenes rather than just the Game Scene, but the implementation of similar codes related to audio processing for all scenes has the cost of code maintenance. Therefore, in this tutorial a combination of `EventListener` and `EventDispatcher` were used to implement entirely integrated audio processing for all scenes. For similar reasons, this combination was also used for the trophy feature and game log.

Each process performed when an arbitrary event occurs will be registered in advance for the `EventListener` class. The `EventDispatcher` class receives the event that occurs and sorts it to the `EventListener` class. The `SoundManager` class is a utility class registered to the `EventListener` class for handling audio processing. Obtaining, setting, and terminating audio data is possible. It can make a sound using an event that makes a sound. Figure 23 shows a process where an event called `EVENT_SHOT` occurs with the `GameScene` class, it is received by the `EventDispatcher` class, sorted to the `EventListener` class, and the `SoundManager` makes the sound corresponding to `EVENT_SHOT`.

Figure 23 Process for Making a Sound



Locations in the Game Scene Where Lua is Used

In this tutorial, the following parts are controlled using Lua scripts.

- Specification of the path of the collada file to be loaded by the program
- Specification of the path of the texture file to be loaded by the program
- Specification of the path of the sound file to be generated by the program
- Frame control for the animation of each model
- Appearance patterns and motion control of the objects that appear in the Game Scene
- Items you wishes to set from an external file, such as the touch panel operation settings

A Lua script exists as data/config.lua, and reading of the Lua script is controlled by config.h and config.cpp. In this tutorial, reloading of the Lua script is done by pressing simultaneously the L button, the R button, and the square button. This will reset the time in the Game Scene and start the shooting game over from the beginning.

By enabling control of the above parts with Lua scripts, it is possible to change behavior in the program without having to recompile the program, for example, it is possible for designers to change the textures loaded without having to recompile the program, making game creation more efficient.

Further, in this tutorial, the appearance pattern of the monsters in the game, the stage configurations, and so on, are defined using Lua scripts. By allowing control of the monster appearance pattern with a Lua script, adjustment of the game balance, etc., is possible without recompiling.

Monitoring Amounts of Memory Usage

By pressing the square button in this tutorial, four bars will be displayed for the current memory amount usage using red, white, and blue bars. The following is an explanation of the calculation method for these bars and how to read each bar.

How to monitor amounts of memory usage

The heap information can be monitored with the heap management in the LPDDR replaced with an implementation that uses memory management function `mspace` of the C and C++ Standard Libraries. Management and monitoring is similarly performed using `mspace` for CDRAM as well.

For details on the memory management function replacement and `mspace`, refer to the "Memory Management Function Replacements of the C and C++ Standard Libraries: Reference" document and mspace.h in the "C and C++ Standard Libraries Overview and Reference" document.

Output memory information

These bars show the following memory information. The information is described from top to bottom

- Heap information for the data in the CDRAM
- Heap information for the LPDDR uncached areas
- Heap information for the USSE code in the CDRAM
- Heap information for the LPDDR cached areas

Red is the current heap usage amount, red + blue is the heap usage amount at the instant where the most heaps were used, red + blue + white is the total amount of kernel memory blocks allocated to the heaps. For details on the implementation, refer to heapallocator.h and heapallocator.cpp in the %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_complia nt directory.

6 Using the Save Data Utility

The specifications of the save data used in this tutorial and the implementation details are described below.

Implementation of Save Data Processing

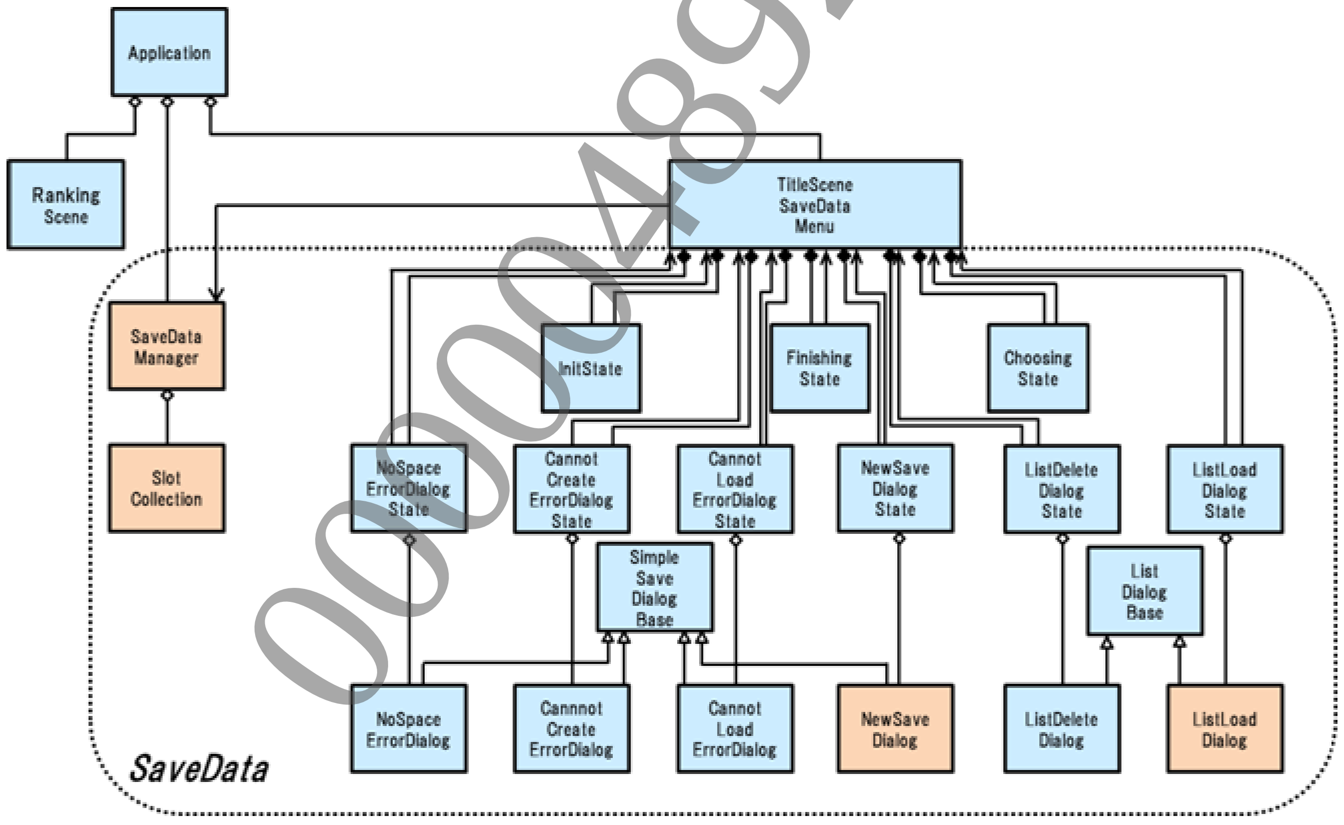
The save data feature offered in PlayStation®Vita consists of the Save Data dialog, which is the GUI part, and the application utility part, which performs save data slot creation, deletion, parameter updates, and so on.

For the Save Data dialog, refer to the "Save Data Dialog Overview" and "Save Data Dialog Reference" documents and for the application utility library, refer to the "Application Utility Overview" and "Application Utility Reference" documents.

The following four classes are explained in detail here.

- `systemService::saveData::SlotCollection` class
- `systemService::saveData::SaveDataManager` class
- `dialog::NewSaveDialog` class
- `dialog::ListLoadDialog` class

Figure 24 `SaveData` Related Class Configuration



systemService::saveData::SlotCollection Class

This class is for managing the save data slots where individual save data are stored. Save data slots can be created, deleted, and managed, and parameters can be updated using the `systemService::saveData::SlotCollection` class.

Upon the creation of a slot, a save data file (`systemService::saveData::SaveDataFileImage` structure) is created simultaneously. `systemService::saveData::SaveDataFileImage` structure is for handling a save data file, which is created for each slot. It is possible to obtain the save data file from `systemService::saveData::SlotCollection` by specifying the slot ID. The file can be loaded and saved, and the information can be obtained and set. Use this class to hold various kinds of game data.

systemService::saveData::SaveDataManager Class

This is a utility class for handling save data slots and files in applications.

This class can be used to create a new save data, load and save the save data files, and obtain and set various kinds of recorded game data such as the ranking data.

dialog::ListLoadDialog Class

This is a wrapper class making it easy to use the "Save Data List Display Mode" of the Save Data dialog. By specifying an instance of the `systemService::saveData::SlotCollection` class as the argument of the member function `start()`, the save data slots saved in the instance of the `systemService::saveData::SlotCollection` class are displayed in the list form in the "Save Data List Display Mode" of the Save Data dialog. By combining usage of the `systemService::saveData::SlotCollection` class and the `dialog::ListDialogBase` class, the save data feature can easily be implemented.

dialog::NewSaveDialogState Class

This is a wrapper class making it easy to use the "User Specified Message Display Mode" of the Save Data dialog. This class is implemented specifically for checking the creation of new save data slots with the player.

Use of SafeMemory

When `save()` of the `systemService::saveData::SaveDataFileImage` structure is called, data is temporarily stored in `SafeMemory`. (This is the same for load/save from the `systemService::saveData::SaveDataManager` class) Only the data for which `save()` is called most recently can be stored in `SafeMemory`.

It is required to call `checkSafeMemory()` of the `systemService::saveData::SaveDataManager` class to write data into a file from `SafeMemory`. After the execution of `checkSafeMemory()` completes, no valid data will exist in `SafeMemory`. If `SafeMemory` does not store any valid data, no processing is done even if `checkSafeMemory()` is called.

In the current tutorial `checkSafeMemory()` is executed in the following timings:

- Immediately after the creation of new save data
- Immediately after the transition to the Save Data Scene

7 Trophy Feature

The specification of the trophy feature adopted in this tutorial and the details on its implementation methods are described below.

Types of Trophy and Conditions for Acquisition

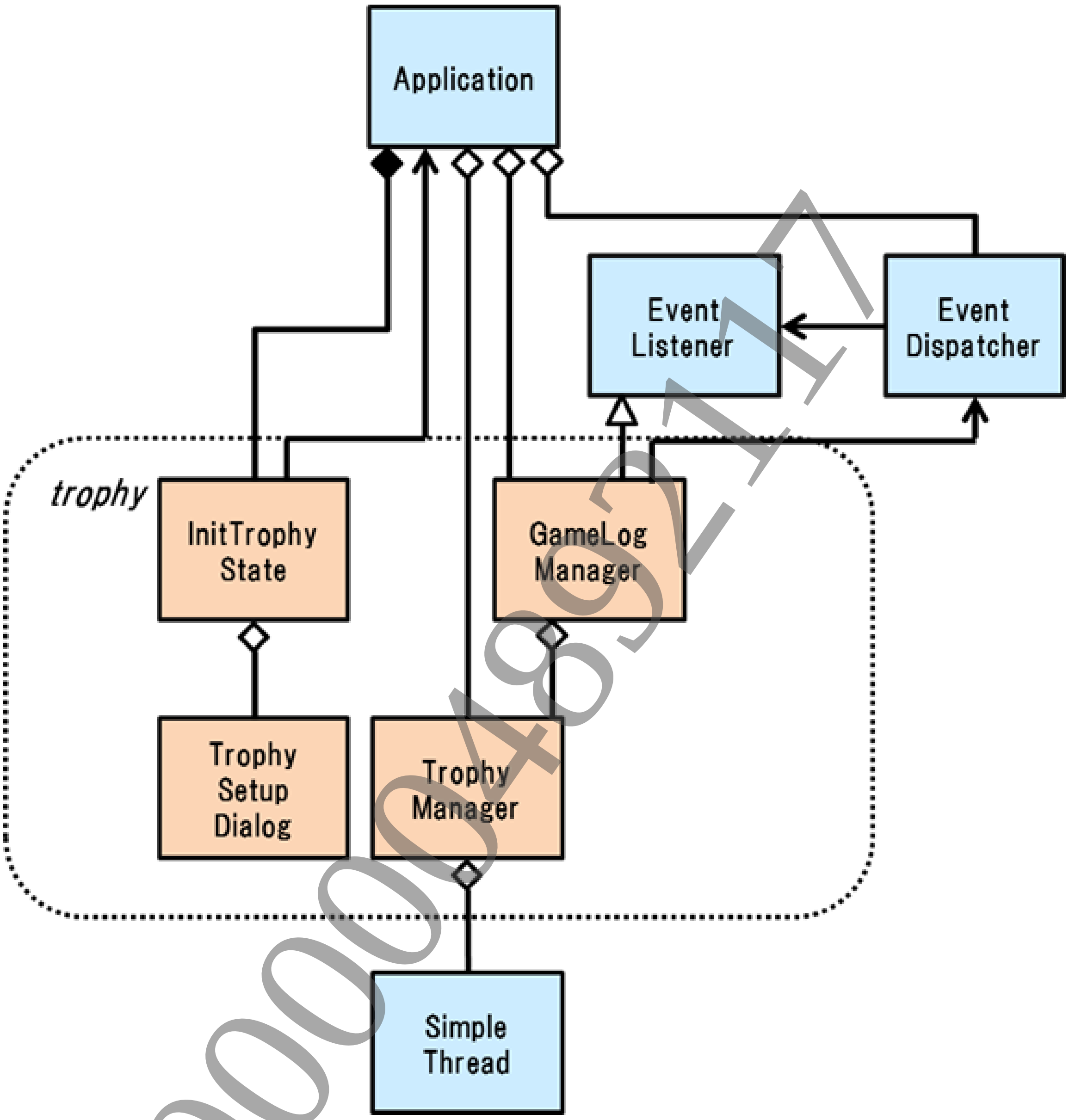
In this tutorial, 12 types of trophies are implemented as an application in the small scope game category. Title Allocation Points are 300 points. Refer to the following table for the types of trophy and the conditions for acquisition.

Trophy Name	Type	Points	Conditions for Acquisition
LG Master	Gold	90	Acquired all trophies
Perfect	Silver	30	All target objects are shot and defeated in one game and no points are taken off
Head Shot Ace	Silver	30	Head shots are successfully done twenty times in one game *Score-down targets are not included
LG Ace	Silver	30	Achieved 600,000 scores per game
100 Head Shots	Bronze	15	Total of 100 head shots are successfully done *Score-down targets are not included
100 Targets	Bronze	15	Total of 100 target objects are shot and defeated *Score-down targets are not included
Head Shot Rookie	Bronze	15	Head shots are successfully done ten times in one game *Score-down targets are not included
LG Rookie	Bronze	15	Achieved 300,000 scores per game
LG Fan	Bronze	15	Played five games
Mercy	Bronze	15	No points are taken off in one game
Seeker	Bronze	15	Performed the zoom operation ten times in one game
Ranker	Bronze	15	First ranked in

Implementation of Trophy Feature

The trophy feature is implemented with four classes as shown in Figure 25.

Figure 25 Trophy Related Class Configuration



The trophy feature is implemented mainly through the following two processing: the setup processing, which is executed after the game starts up, and the unlock processing, which is performed when the conditions for trophy acquisition are satisfied. In the setup processing, the NP Trophy library is set up, and the trophy information is obtained. In the unlock processing, unlocking of the NP Trophy library is performed for the trophies that have yet been acquired.

GameLogManager Class

This class manages various game data and cumulative data based on the information of events issued in the Game Scene and judges the conditions for trophy acquisition. If the conditions are satisfied, `GameLogManager` calls `unlock()` for `TrophyManager`. Refer to the following table for the data managed by `GameLogManager` and the related trophy.

Type of Log	Related Trophy
Cumulative number of games	LG Fan
Cumulative number of hits	100 Targets
Cumulative number of head shots	100 Head Shots
Number of targets per game	Perfect
Number of score-down targets per game	-
Number of hits per game	Perfect
Number of head shots per game	Head Shot Rookie, Head Shot Ace
Number of score-down hits per game	Mercy, Perfect
Number of times of zoom operation per game	Seeker

application::InitTrophyState Class

This class is for transitioning the application to the trophy system's setup state; it is also used within the setup state. It creates the `systemService::trophy::TrophySetupDialog` class.

systemService::trophy::TrophySetupDialog Class

This is a wrapper class to easily use "Trophy Setup Dialog" of the NP Trophy library. This class is the dialog displayed during trophy setup.

systemService::trophy::TrophyManager Class

This class sets up the trophy system and manages the trophy information. By specifying the trophy ID, the trophy can be unlocked, and the unlock state can be checked. This class holds the trophy's unlock information obtained from the NP Trophy library at the time of setup processing.

8 Specifications and Implementation of Features Using the "near" System

Below is an explanation of the specifications and implementation of the features provided in this tutorial by using the "near" system. For details on the "near" system, refer to the "near System Overview" and the "near Utility Overview" documents.

Specifications of the Features Using the "near" System

In this tutorial, the following two characteristics are implemented using the features of the "near" system:

(a) Better gifts can be distributed based on game performance

(b) By receiving better gifts, players can gain an edge in the game

The adoption of specification (a) will add a further gaming target. Moreover, users will actively engage in (b) in order to gain an edge in the game. As a result, increased interest in the game can be expected using game changes through the network. For gifts to be received, they must first be discovered; when an update operation with the "near" application is performed for this purpose, gifts will be distributed and discovered automatically. Specific features are as follows:

Distributing Gift

In this tutorial, we have added a system whereby "RESCUE" points are accumulated by killing the monsters only. When the Game Scene ends, gift distribution settings will be performed in accordance with the results. The higher the "Rescue" points, the greater will be the number of treasure boxes distributed as gifts.

Receiving Gifts

Gifts are received in the "near" Game Goods Scene, and treasure boxes will appear during the Game Scene. Treasure box targets will appear in the course of the Game Scene in accordance with the number of treasure boxes contained in the gifts received; by shooting the treasure box targets, it will be possible to obtain bonuses.

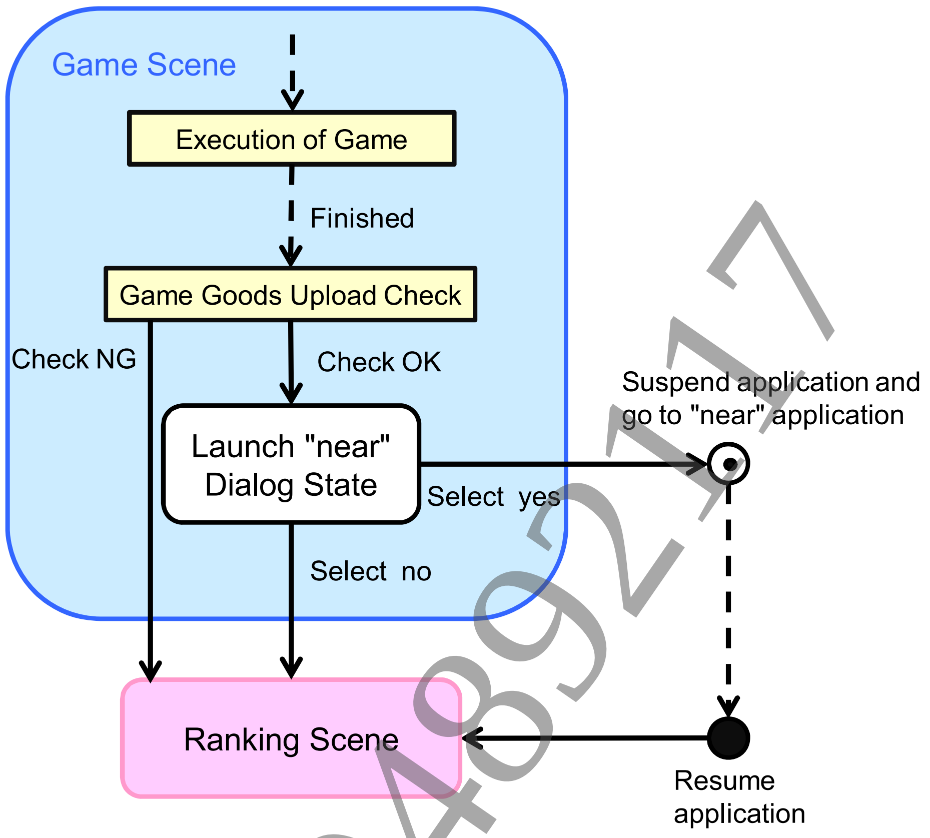
"near" Application Start-Up

In order to prompt gift distribution/receipt, users will be shown a confirmation dialog asking whether to start up the "near" application. If the users agree, the "near" application will start from this tutorial.

Details of Scene Transitions

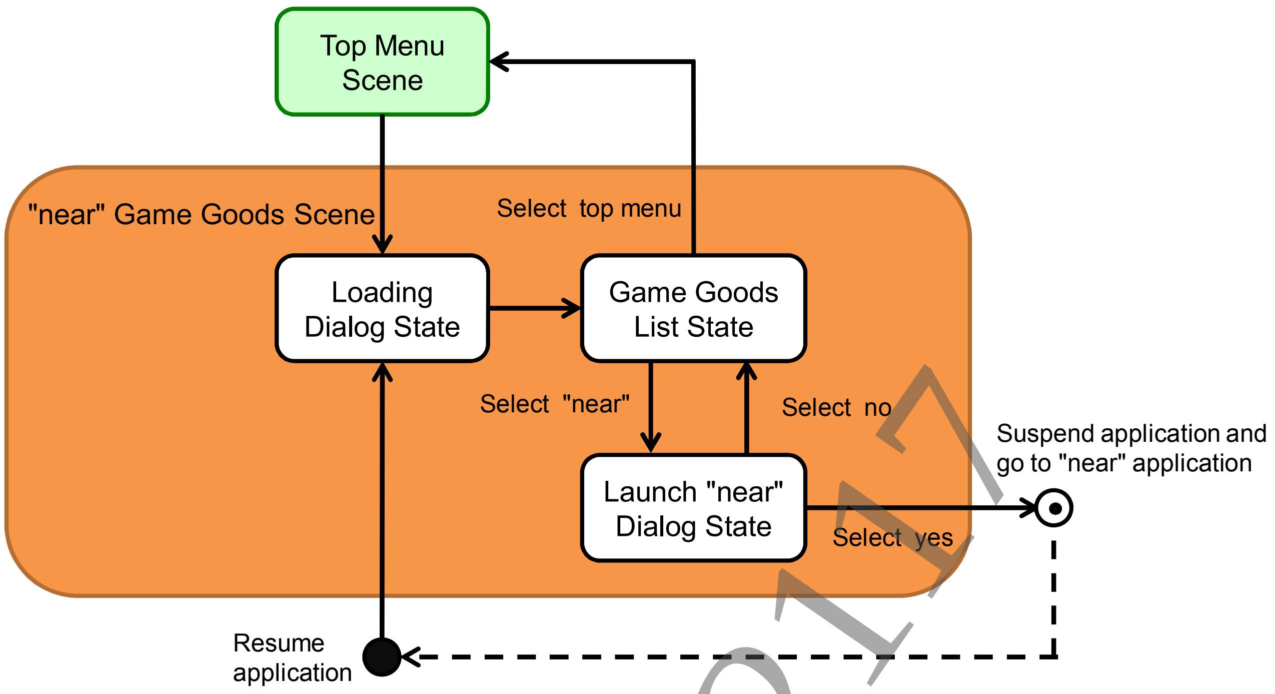
Transitions between the Game Scene and the "near" application are as shown below:

Figure 26 Diagram of Transitions Between the Game Scene and the "near" Application



After the game ends, the dialog for confirming the startup of the "near" application will appear, and the "near" application's gift distribution settings will be performed once the "Game Goods Upload Check" is **OK**. Transitions between the "near" Game Goods Scene and the "near" application are as shown below.

Figure 27 "near" Game Goods Scene Transition Diagram



"near" utility gift data is received in "Loading Dialog State", and recorded in the save data.

Classes Providing Features Using the "near" System

This tutorial's features using the "near" system are implemented with the following classes:

- `systemService::NearManager` class
- `application::NearScene` class

systemService::NearManager Class

This is a class for calling APIs provided by the "near" utility. It performs library initialization, set-up of gifts for distribution, received gift operations, start-up of the "near" application, and library termination.

application::NearScene Class

This is a class for GUI using "near" game service features. It displays dialogs for starting up the "near" application to prompt the distribution of gifts, and the "near" Game Goods Scene for receiving gifts.

Details of the Features Using the "near" System

Below is a detailed description of each of the features implemented by using the two classes above.

Gift Distribution

This function sets data such as gift images, quantity of treasure boxes, etc., and performs gift distribution settings by using `sceNearSetGift()`.

In the tutorial, the fixed value 0xc0092921 is used for `SceNearGiftId` when `sceNearSetGift()` is called. The value 0xc0000000 utilizes the upper 8 bit setting value indicated in the "near Utility Reference" document. For 0x00092920, the last five figures of this tutorial's title ID (09292) were set; this value, however, does not have a particular significance. 0x00000001 is a value indicating the first type of gift in the tutorial. Since gift ID is fixed, we have set a limited value for the quantity of gifts to be distributed in order to prevent Game Goods from being obtained unlimitedly.

Gift Receipt

Lists of discovered gifts and received gifts are obtained by using `sceNearGetDiscoveredGifts()` and `sceNearGetDiscoveredGiftStatus()`. After obtaining data by using `sceNearOpenReceivedGiftData()`, `sceNearReadReceivedGiftData()` and `sceNearCloseReceivedGiftData()`, received gifts are deleted by using `sceNearDeleteDiscoveredGift()`. Obtained gift data is recorded in the save data and used in the Game Scene.

Discovered gifts that have not been received are displayed to users as a "near" application start-up button list in the "near" Game Goods Scene, prompting the receipt of gifts. Also, the online IDs of the Game Goods' senders are obtained from `sceNearGetDiscoveredGiftSender()` as information shown to the user.

"near" Application Start-Up

Using `sceNearLaunchNearAppForUpdate()` or `sceNearLaunchNearAppForDownload()`, start up the "near" application. Use `sceNearLaunchNearAppForUpdate()` for distributing and exchanging gifts, and use `sceNearLaunchNearAppForDownload()` for receiving gifts.

When the game is resumed, latest information of the "near" application is obtained again using `sceNearRefresh()`.

9 Online Ranking Feature

Below is a detailed description of the specifications and implementation of the online ranking features adopted in this tutorial.

Refer to the "NP Library Overview" document for information on all NP features, to the "NP ScoreRanking Library Overview" and "NP ScoreRanking Library Reference" documents for each online ranking feature, and to the "libnetctl Reference" document for the Network Check dialog.

How to Use Online Ranking

The scores achieved in the shooting game are registered in the online ranking, allowing users to compete with each other.

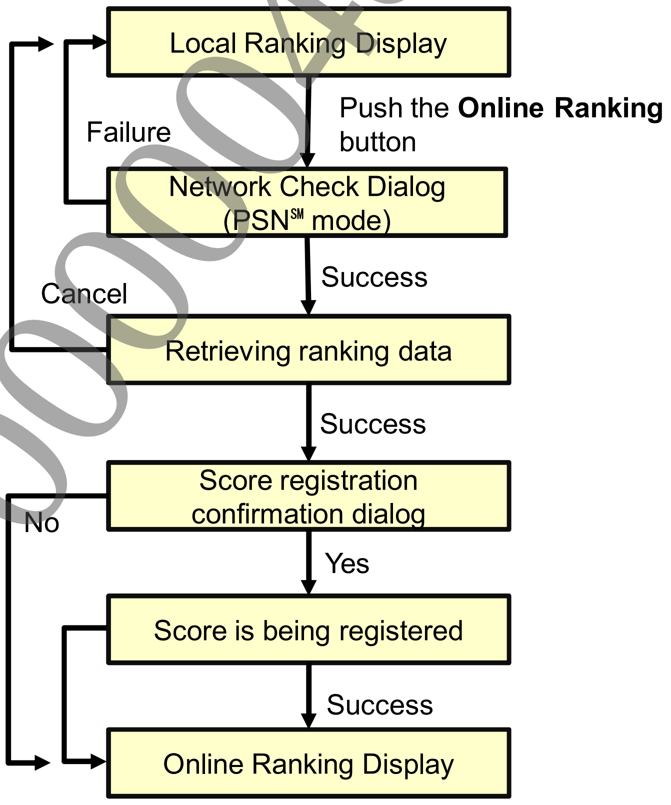
Requesting an NP Communication ID

In order to use the NP ScoreRanking feature, it is necessary to newly register a product on the PlayStation®Vita Developer Network and set the services for this product. By doing this, the NP Communication ID required for online ranking can be issued. For details on product and various ID registration/requesting, refer to "PSN℠ Service Configuration on VITA DevNet" (https://psvita.scedev.net/docs/psn_form_quick_guide/).

How Online Ranking is Implemented

Choose **RANKING** from the Top Menu Scene to display the Local Ranking Scene. Tap on the **Online Ranking** button when this screen is displayed and the application will transition to the Online Ranking Scene. The processing flow when transitioning to online ranking is as shown below:

Figure 28 Online Ranking Processing Flow


Local Ranking Display
Push the Online Ranking button
Failure
Network Check Dialog (PSN℠ mode)
Success
Cancel
Retrieving ranking data
Success
Score registration confirmation dialog
No
Yes
Score is being registered
Success
Online Ranking Display

First, the user's signed-in state is verified using the Network Check dialog in accordance with the "Service States and Usable Functionalities" section in the "NP Library Overview" document.

```
SceNetCheckDialogParam netCheckDialogParam;
sceNetCheckDialogParamInit(&netCheckDialogParam);
netCheckDialogParam.mode = SCE_NETCHECK_DIALOG_MODE_PSN;
ret = sceNetCheckDialogInit(&netCheckDialogParam);
SAMPLE_UTIL_ASSERT_MSG(SCE_OK==ret, "ret=%#x", ret);
```

If verification is successful, obtain the top ten data using `sceNpScoreGetRankingByRange()`.

```
SceNpScoreBoardId boardId = 0;
SceNpScoreRankNumber startSerialRank = 1;
SceNpScoreRankData rankData[10];
SceRtcTick lastSortDate;
SceNpScoreRankNumber totalRecord;

int nRank =sceNpScoreGetRankingByRange(
        requestId,
        boardId,
        startSerialRank,
        rankData,
        (10 * sizeof(SceNpScoreRankData)),
        NULL, 0, // comment
        NULL, 0, // game info
        10,
        &lastSortDate,
        &totalRecord,
        NULL
        );
```

If the user has unregistered data, register the user data using `sceNpScoreRecordScore()`.

```
SceNpScoreBoardId boardId = 0;
ret = sceNpScoreRecordScore(requestId, boardId, score, NULL, NULL, NULL, NULL,
NULL);
```

Ranking data acquisition and data registration may take some time. During this time, in the sample program a dialog with a **Cancel** button will be displayed, allowing the user to abort processing.

When this button is pressed, abort the ongoing processing using `sceNpScoreAbortRequest()`.

```
int ret = sceNpScoreAbortRequest(m_currentRequestId);
SAMPLE_UTIL_ASSERT_MSG(ret == SCE_OK, "ret=%#x", ret);
```

Precautions When Implementing Online Ranking

When conducting testing during development, bear in mind that data registered in the online ranking will not be reflected immediately on the server.

At the point in time when a score is saved in the save data, that score is not associated with its Sony Entertainment Network account. In order to prevent a score from being registered from multiple accounts, each score can only be registered in the ranking once.

10 Package Creation

Below is an explanation of the procedure for creating and installing the shooting game created in this tutorial as a package and a patch package. The following three types are available in this tutorial:

- [Type1]
  This type generates a package including all features.
- [Type2]
  This type generates the package prior to adding the patch. Some of the features are excluded.
- [Type3]
  This type generates the patch package for [Type2]. By adding the patch, "near" game service features will be added to [Type2].

param.sfo Creation

Create param.sfo for inclusion in the package. Start the Param File Editor from **All Programs** -> **SCE - PS Vita** -> **Publishing Tools** -> **Param File Editor,** set the parameters as shown below, and save it as the following file.

- [Type1]
  %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\app\sce_sys\param.sfo
- [Type2]
  %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\app_for_patch\sce_sys\param.sfo
- [Type3]
  %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\patch\sce_sys\param.sfo

Core

In the Core category, set the parameters as follows.

[Type1]

Parameter	Setting
Category	PS Vita Application
Content ID	IV0002-NPXS09292_00-THELASTGALLERYAL
VERSION	01.00
APP_VER	01.00
Parental Lock Level	1

[Type2]

Parameter	Setting
Category	PS Vita Application
Content ID	IV0002-NPXS09292_00-THELASTGALLERYFP
VERSION	01.00
APP_VER	01.00
Parental Lock Level	1

[Type3]

Parameter	Setting
Category	PS Vita Application Patch
Content ID	IV0002-NPXS09292_00-THELASTGALLERYFP
VERSION	01.00
APP_VER	01.01
Parental Lock Level	1

Patch Setting

In the Patch Setting category, set the parameters as follows ([Type3] only).

Parameter	Setting
Patch Type	Cumulative Patch (or not a patch)

App Setting

In the App Setting category, set the parameters as follows.

Parameter	Setting
Save Data Quota	10240
liblocation	No check

Languages

In Languages, select Text(English) in addition to Text (Default Language), which is selected by default .Text (Default Language), Text (English)

Text (Default Language), Text (English)

The parameters are set as follows.

Parameter	Setting
App Title	The Last Gallery
App Short Title	The Last Gallery

Bootable Message

The parameters are set as follows.

Parameter	Setting
Bootable Message	No check

gp4p File Creation

Create the gp4p file for inclusion in the package.

Start the Package Generator from **All Programs** -> **SCE - PS Vita** -> **Publishing Tools** -> **Package Generator,** set the parameters as shown below, and save it as the following file.

- [Type1]
  %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\ thelastgallery_app_all.gp4p
- [Type2]
  %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\ thelastgallery_app_for_patch.gp4p
- [Type3]
  %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\ thelastgallery_patch.gp4p

Setting

In Setting, set the following parameters.

[Type1]

Parameter	Setting
Project Type	PS Vita Application
General - Content ID	IV0002-NPXS09292_00-THELASTGALLERY00
General - Master Version	01.00
Storage - Distribution Type	VC-MC (VC-MC/MC-MC or VC-MC/No MC)
Storage - Media Capacity	VC 2GB (R/O:1792MiB, R/W:On MC)
DRM - DRM Type	Local
DRM - Passcode (32 chars)	Auto-generate

[Type2]

Parameter	Setting
Project Type	PS Vita Application
General - Content ID	IV0002-NPXS09292_00-THELASTGALLERYFP
General – Master Version	01.00
Storage – Distribution Type	VC-MC (VC-MC/MC-MC or VC-MC/No MC)
Storage – Media Capacity	VC 2GB (R/O:1792MiB, R/W:On MC)
DRM – DRM Type	Local
DRM – Passcode (32 chars)	Auto-generate

[Type3]

Parameter	Setting
Project Type	PS Vita Application Patch
General - Content ID	IV0002-NPXS09292_00-THELASTGALLERY FP
General – Master Version	01.00
DRM – DRM Type	Local
DRM – Passcode (32 chars)	Set to the same as [Type2]

File Configuration

[Type1]

The following files and directories that are under %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\app are stored directly under Root.

- eboot.bin
- sce_sys
- game_data
- sce_module

However, since the files that are in the following directories are already included in game_data\game_data.psarc, they are not included in the package.

- game_data\image
- game_data\model
- game_data\sound

[Type2]

The following files under %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\app_for_patch will be stored under Root with the same directory configuration:

- sce_sys\param.sfo
- eboot.bin

The following files and directories under %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\app, excluding sce_sys\param.sfo and game_data\gift_data, will be stored as they are under Root.

- sce_sys
- game_data
- sce_module

However, since the files that are in the following directories are already included in game_data\game_data.psarc, they are not included in the package.

- game_data\image
- game_data\model
- game_data\sound

[Type3]

The following files are the minimum necessary for creating patches. The "Patch Overview" document contains detailed explanations concerning patches.

- The package prior to adding the patch
- sce_sys\param.sfo
- sce_sys\changeinfo\changeinfo.xml

In this tutorial's patch creation, game_data\gift_data and eboot.bin for providing the "near" system features to be added are allocated in addition to the minimum necessary files above. Also, the package prior to adding the patch is generated automatically by building %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\tutorial_simple_shooting_game.sln with "Release Build", and is specified as package\IV0002-NPXS09292_00-THELASTGALLERYFP.pkg. The following files under %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\app are stored under Root with the same directory configuration:

- game_data\gift_data

The following files under %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\data\patch are stored under Root with the same directory configuration.

- sce_sys\param.sfo
- sce_sys\changeinfo\changeinfo.xml
- eboot.bin

pkg File Creation and Installation

The following procedure can be used to create and install the pkg file.

Procedure for the Creation and Installation of Application Package Including All the Features

(1) Build %SCE_PSP2_SDK_DIR%\target\samples\sample_code\system\tutorial_shooting_game_trc_compliant\tutorial_simple_shooting_game.sln with "Patch Build". Files that are required in the package such as eboot.bin and libc.suprx will be created.

(2) With Publishing Tools installed, double click on thelastgallery.gp4p that was created. Package Generator will start up.

(3) Click on the [Package] button that is in Package Generator. The IV0002-NPXS09292_00-THELASTGALLERYAL.pkg file will be created at the specified location.

(4) Install the IV0002-NPXS09292_00-THELASTGALLERYAL.pkg file that was created. For details about how to install a pkg file, refer to the "Application Development Process Overview" document.

How to Create and Install Packages not Including "near" System Features, and Subsequently Create and Install Patch Packages

Refer to the "Procedure for the Creation and Installation of Patch Package Files" section in the 1."Introduction" chapter.