Game Orientated Camera Demo

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

About This Document

This document explains techniques used in the "Game Orientated Camera Demo" sample code. The sample code is located in the PSP2 SDK package at:

```
%SCE_PSP2_SAMPLE_DIR%/sample_code/input_output_devices/demo_game
_orientated_camera
```

Typographic Conventions

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

Hyperlinks

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

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

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

1 Introduction

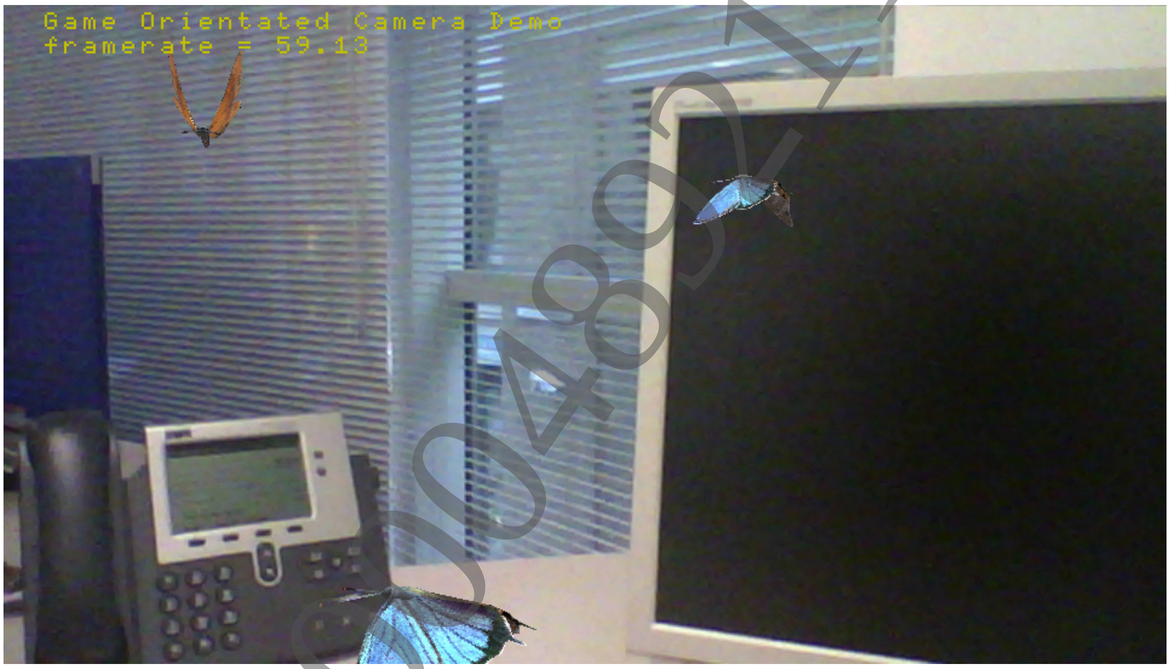
The “Game Orientated Camera Demo” sample program demonstrates the usage of the camera device library and motion library.

You can use the PSP2 cameras in-game to achieve various effects. This sample displays the back camera device input and augments it using a very basic method: The orientation or projection of the 3D scene is determined by the orientation of the PSP2 console, and the animated model reacts to console movement and scene changes.

The sample allows switching between various camera device options such as encoding formats and buffer input and processing method. These were used to investigate the differences in performance of various settings and methods, although the final specifications for certain features are still to be confirmed at the time of writing this document.

This document provides an explanation of the simple camera handling code, the motion controls implemented, and performance considerations when using the PSP2 camera device library.

Figure 1 Screenshot of Game Orientated Camera Demo



```
Game Orientated Camera Demo
framerate = 59.13
```

2 Implementation

This chapter provides an overview of this sample's implementation. It is focused on the implementation of the Camera Device class, motion controls, and image processing.

Camera Device Class

This class wraps the SDK camera library so that initialization data is correctly set up when different image formats and read buffer methods are used. The class also provides the captured image data in a form appropriate for display as a gxm texture, which the sample renders as a textured quad.

To cycle through formats, press the **R** button and check console output for information on the current setting. RAW format was not fully supported in the library at the time of development so it is not implemented in this sample. Additionally, the `YUV_PLANE` format is not directly supported as a gxm texture data format; therefore, this sample displays only the luminance values as a gray scale texture.

This sample program uses a textured quad to display the camera image because such a display method is quite flexible. An alternative is to read camera input directly into the display buffer. Such a display method restricts you to matching the size and image format to a compatible display format, in fact, only `SCE_CAMERA_FORMAT_YUV422_TO_ABGR` can be used. However, an advantage of reading into the display buffer is that you do not need extra read buffers for the camera data. In this sample, at least two read buffers are required so that the camera can read into one buffer while the other is being displayed.

In either display method, the read timing needs to be synchronized with scene rendering and draw buffer flips. This prevents the camera image data from being overwritten while the data is being used in rendering. In the sample it is achieved by using gxm scene notifications (for details, see the *libgxm Overview*).

Motion Controls

The motion library is used to obtain the orientation of the console. Then in the scene rendering, the scene view matrix is calculated so that the projection of 3D models matches console orientation. The result is a rough match partly because several assumptions are made by the sample in calculating the view matrix:

- When mapping between different coordinate systems assume that the camera, motion sensors, and the pivot point of the controller are all at the same point.
- Also assume that the area where the sample is operated has enough open space for the butterflies.

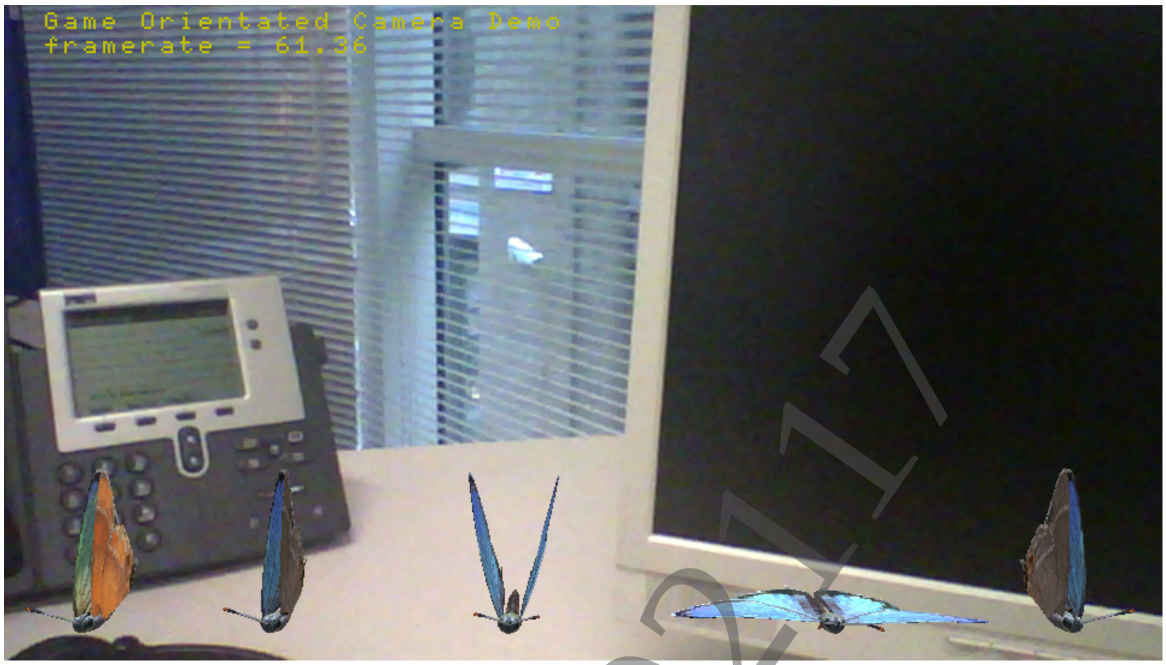
The effect may be improved by changing these environment dependent parameters; for example, the pivot point of the console could be considered to be approximately an arm's distance to the user. Or the dimensions of the 3D scene can be adjusted to fit in a more confined space or a larger room size.

Additionally, there are inaccuracies in motion-detected orientation, and it is possible for errors to accumulate (for details, see the Motion Library SDK documentation). In such cases, the motion library needs to be reset. The sample allows the user to reset when the square button is pressed.

The sample design allows for some inaccuracy in mapping because the butterfly is always flying when the controller is moving a great amount. The state of the motion controller is set to be stationary when the change in orientation is within a defined threshold. The butterflies only try to land when the controller is stationary. Then the position they land on is calculated to coincide with the bottom edge of the screen (Figure 2).

The butterflies remain standing on the screen edge if the controller is maintained in a stationary state, although in practice the controller could be moving slightly. Therefore, the landed butterflies' positions are recalculated according to the view matrix at each frame to give a fixed screen coordinate position. An exception to this behavior is made if the camera direction is nearly parallel to the y axis; this exception is mainly due to a limitation in model rendering where only y rotation is handled. However, it is also quite interesting to see the effect produced when the view matrix is updated with motion orientation while butterflies stay on their world coordinate landing position (the controller needs to be moved slowly to see this; otherwise the butterflies will fly away). In such a scenario, the inaccuracies explained above can be observed quite easily.

Figure 2 Butterflies Landed on Screen Edge



Camera Image Processing

With the camera device input displayed and only orientation provided by the motion library, the 3D scene interaction is somewhat limited. The next step to enhance interaction between the camera image and scene could be to analyze the image for features that scene objects can react to. Image processing or computer vision methods can be used to analyze the camera input, such as filtering, edge and feature detection, and movement detection. A very basic example is implemented in the sample, mainly to allow for simple performance testing. A full implementation with optimization would be quite application dependent.

In the sample, the Y-plane (luminance values) was chosen as source data; therefore the image processing is enabled for `YUV420` and `YUV422` planar formats only. Switching between these formats with difference processing enabled shows the contrast between the performance of each format. For this case, the processing time is not as significant as the time taken to read the camera device input in `YUV422` planar format.

The `processImage` class sets up multiple source and result buffers, then copies data from camera device buffers to processing buffers. Using only a subset of data, it calculates pixel value differences between the current and previous frame. The processing is only done when the camera read buffer has been updated.

The result buffer was displayed as a texture to produce the image in Figure 3. Only every eighth pixel is differenced; then the absolute value is set in the result buffer.

A measure of the difference recorded in the result buffer is calculated by comparing the accumulated difference values with a threshold value to determine if a significant change was detected over the entire image. This threshold is set quite high in the sample, so only a large change in image will start butterflies flying, such as putting fingers in front of camera. The difference detection result is only used when the controller motion is considered stationary, because at such a time the butterflies are potentially landing.

Figure 3

3 Performance Issues

To improve performance, several issues could be considered; some of these are quite application dependent. The default settings in this sample are listed below. They were chosen because they offered the best overall performance for this particular application.

- `SCE_CAMERA_FORMAT_YUV420_PLANE` was chosen as the default format because it is the fastest format for device data transfer and decoding (closely followed by ARGB/ABGR formats). The decode and transfer is handled by the camera device driver when `sceCameraRead()` is called. The driver transfers via DMA, which makes decoded formats much lighter on CPU resources and faster than native `YUV422` based formats that do not use DMA.
- Because read times may be variable, the camera read function should be called in a separate thread. Therefore in the sample the display and frame rate for 3D model rendering is 60 fps whereas the camera update rate is variable.
- Read buffers are set at read time by initializing the camera with user buffer setting of `SCE_CAMERA_BUFFER_SETBYREAD`, because this allows multiple user buffers to be set. Multiple user buffers allow maximum flexibility for later processing and can prevent buffer contention when frame times are long. If the user buffer setting method is `SCE_CAMERA_BUFFER_SETBYOPEN`, it is not possible to have multiple read buffers, because the read buffer is set when device is started. The two different read methods did not cause significant change in the performance of `sceCameraRead()`.
- The optimum number of buffers depends on the relative frame rates of the camera and application and on the number of frames taken from update to receiving notification that a buffer is free. Because the sample program uses a triple-buffered display, three buffers are used for camera data, and the camera frame rate is set to `SCE_CAMERA_FRAMERATE_60`.
- The Y-plane of the YUV read buffer is copied to cached memory before image processing because read buffers for `YUV420` are uncached. The overhead of extra `memcpy()` was found to be much less than any increased time when accessing uncached buffers directly during processing.
- In-place sub-sampling at the image processing stage, from camera resolution to smaller processing resolution, improved performance dramatically. **Note:** Memory footprint could also be improved if `memcpy()` was replaced by true image down sampling.

The last two points are influenced by the very basic nature of the differencing processing implemented, so it should be noted that copying the read buffer before processing begins will probably not be the best case with a more optimized process. Ideally, the uncached memory should be accessed as few times as possible, but if you must do one pass through the data, then additional processing that can be rolled into the same pass would be most efficient. However the simple image processing algorithm used in this sample did many more memory accesses per pixel than the straight `memcpy()`; therefore it was better to move all the data into a cached buffer first.

In addition, when the format is `YUV422`, buffers can be created in cached memory. In this case the read buffers are better used in-place. However the read buffer update in `sceCameraRead()` is still much slower compared to the decoded formats, so the advantage of having cached read buffers for processing in this sample does not result in an overall improvement.

Table 1 shows a summary of some of the pros and cons for each format.

Table 1 Summary of Camera Device Formats and Specifications

Camera Image Format	Texture Format	Fast Read	Memory Type	Buffer Size
`SCE_CAMERA_FORMAT_YUV422_TO_ABGR`	Yes	Yes	Uncached	Large
`SCE_CAMERA_FORMAT_YUV422_TO_ARGB`	Yes	Yes	Uncached	Large
`SCE_CAMERA_FORMAT_YUV420_PLANE`	Yes	Yes	Uncached	Smallest*
`SCE_CAMERA_FORMAT_YUV422_PLANE`	No	No	Any	Small
`SCE_CAMERA_FORMAT_YUV422_PACKED`	Yes	No	Any	Small

* At the default camera resolution of this sample, `SCE_CAMERA_FORMAT_YUV420_PLANE` would use the smallest buffer size. However, if lower resolution data is used `YUV422` formats may not require pitch settings. For additional details, refer to the sample code and the applicable SDK reference document.

Because the image processing in this sample is very minimal, the processing in a more advanced application is likely to take significant resources. In this case, there are further improvements and optimizations that would be worth considering:

- Process images in a separate thread, independent of frame rendering and camera device capture.
- Use GPU for processing; however this decision is highly application dependent. The advantage is that the GPU is possibly fastest at image processing, and uncached memory for texture data is also not an issue. However, the disadvantages are that there will be no gain if the game is already GPU bound. Also, there would be at least one frame delay to access texture buffer processing results and additional synchronization code might be necessary.
- Optimize on the CPU using the NEON instruction set. This should produce good performance and may be a simpler option.