RenderWare Graphics

User Guide

PlayStation 2 Specific

Contact Us

Criterion Software Ltd.

For general information about RenderWare Graphics e-mail info@csl.com.

Developer Relations

For information regarding Support please email devrels@csl.com.

Sales

For sales information contact: rw-sales@csl.com

Contributors

RenderWare Graphics development and documentation teams.

Table of Contents

Foreword

About the PlayStation 2 Specific Guide

In this release of the User Guide, we have documented the following features available only on PlayStation 2:

- PS2All Pipe Overview
- Pipeline Delivery System (PDS)

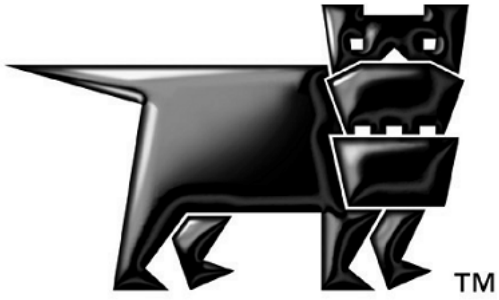
Please let us know what you think and feel free to offer any suggestions.

Regards,

The RenderWare Graphics Team

Chapter 1

PS2All Pipe Overview



1.1 Introduction

Before reading this chapter, you should first read the chapters *PowerPipe Overview* and *Pipeline Nodes*, as this chapter refers to concepts introduced therein.

1.1.1 What PS2All Provides

PS2All is a collection of RenderWare Graphics functionality for the PS2 platform. PS2 is a very different architecture from prior consoles and so requires specialized software to drive it (as demonstrated by the "ppvu" examples). The purpose of PS2All is to allow the construction of PowerPipe pipelines that can render objects in a highly efficient and yet flexible manner.

1.1.2 The PS2All Nodes

PS2All provides two pipeline nodes:

- PS2All, which is used to construct object pipelines;
- PS2AllMat, which is used to construct material pipelines.

It should be noted that, in this case, the division of object and material pipelines does *not* incur the execution overhead incurred in generic pipelines. Internally, PS2All functions quite differently from generic pipelines and pipeline nodes (more details are available in the API reference for **`RxNodeDefinitionGetPS2All()`** – documentation relating to PS2All can be found in: Modules → PowerPipe → World Extensions → PS2 All). It is convenient, and close to the truth, to think of the PS2AllMat node as merely encapsulating data used by the PS2All node for each **`RpMesh`** in the current object. PS2All-based object pipelines will only work with PS2AllMat-based material pipelines, and vice versa.

1.1.3 The Benefits of PS2All

PS2All drives PS2 rendering by transferring geometry through the DMA engine to the VU1 vector processor, which performs all transformation and lighting processing (as well as further optional rendering effects). PS2All allows overloading of the microcode used on VU1 and the CPU-side processing performed by PS2All (usually comprising of mainly data instancing and render-state setup) is overloadable via many callbacks, which are configurable through a pipeline-construction-time API. It innately and efficiently supports multi-material objects. The callbacks can be optimized using application specific knowledge to increase performance significantly.

1.1.4 The PS2All Example

The PS2All Example is in the **`examples/ps2all/`** folder. This provides example code that demonstrates the construction of a pipeline based on PS2All, as well as the use of PS2All to maximize rendering performance on the basis of application-specific knowledge.

1.1.5 Other Documents

For further information on PS2All, you should refer to the API reference documentation section Modules → PowerPipe → World Extensions → PS2 All.

There is a white paper, *PS2 Vector Code*, which covers the process of creating custom VU1 vector code for use with RenderWare.

The "PPVU2" example also contains documents, pertaining to PS2 development, which may be useful.

An introduction to PowerPipe can be found in the chapters *PowerPipe Overview* and *Pipeline Nodes* in the User Guide.

1.2 PS2All Pipelines

This section will cover the construction of PS2All-based object pipelines and PS2AllMat-based material pipelines.

1.2.1 Construction of PS2All Pipelines

Pipeline Structure

PS2All-based object pipelines should be constructed solely from the PS2All node. It is possible to put nodes before PS2All in the pipeline, though it will not accept packets from any such nodes and so their effect will be limited to doing independent processing and optionally terminating the pipeline before PS2All is executed. An example of where this might still be useful is the PVS node (see the chapter entitled *Potentially Visible Sets*). If placed at the head of an object pipeline, the PVS node may prematurely terminate the pipeline on the basis of visibility data (i.e. preventing the rendering of occluded objects by preventing the execution of PS2All).

Pipeline Node Setup API

The two API functions used for setting up the PS2All node, during pipeline construction, are:

- **RxPipelineNodePS2AllSetCallBack()**
- **RxPipelineNodePS2AllGroupMeshes()**
- **RxPipelineNodePS2AllSetLightBufferOffset()**

RxPipelineNodePS2AllSetCallBack()

RxPipelineNodePS2AllSetCallBack() allows the user to set the callbacks called by the PS2All node at different points during its execution. This function should be called after the pipeline has been unlocked. There are two callback types supported by the PS2All node, the **RxPipelineNodePS2AllObjectSetupCallBack** and then **RxPipelineNodePS2AllObjectFinalizeCallBack**. These are described in the following section, *PS2All Customization*.

RxPipelineNodePS2AllGroupMeshes()

RxPipelineNodePS2AllGroupMeshes() may optionally be used to make the PS2All node *always* use a specific PS2AllMat-based material pipeline, rather than follow the default behavior of retrieving material pipelines from the **RpMaterial**s attached to object **RpMesh**s.

RxPipelineNodePS2AllSetLightBufferOffset()

RxPipelineNodePS2AllSetLightBufferOffset() may optionally be used to set the location of the lighting data buffer in VU1 memory. This is set on a per-object basis, since lighting is usually done once per object. Changing the location of the lighting buffer can be used to shrink the buffer (so that more space is available for input/output geometry buffers) or to grow it (so that more lighting data or static data may be uploaded). The *PS2 Vector Code* white paper covers the process of developing custom VU1 vector code for use with RenderWare, and this contains further related information.

Further detail can be found in the API reference documentation for the PS2All callback types and API functions.

Example Code

To summarize, the construction of a PS2All-based object pipeline proceeds as follows:

1. Create a pipeline;
2. Lock the pipeline for editing;
3. Add a fragment to the pipeline, with the PS2All node at its end;
4. Unlock the pipeline;
5. Call post-unlock PS2All node setup API functions (**RxPipelineNodePS2AllSetCallBack()** for all callback types and optionally **RxPipelineNodePS2AllGroupMeshes()**);

Some example code demonstrating this process follows:

```
RxPipeline *
CreateMyPS2AllPipeline(void)
{
    RxPipeline *newPipe;

    newPipe = RxPipelineCreate();

    if (NULL != newPipe)
    {
        RxLockedPipe *lockedPipe;

        lockedPipe = RxPipelineLock(newPipe);
```

```
if (NULL != lockedPipe)
{
    RxNodeDefinition *ps2All;
    RxPipeline       *result;

    ps2All = RxNodeDefinitionGetPS2All();
    assert(NULL != ps2All);

    /* PS2All-based object pipelines are trivial,
     * (usually) containing just the PS2All node */
    lockedPipe = RxLockedPipeAddFragment(lockedPipe,
                                         NULL,
                                         ps2All,
                                         NULL);
    assert(NULL != lockedPipe);

    /* Unlock the pipeline */
    result = RxLockedPipeUnlock(lockedPipe);

    if (result != NULL)
    {
        RxPipelineNode   *plNode;

        /* Get a pointer to the PS2All node
         * in the unlocked pipeline */
        plNode = RxPipelineFindNodeByName(
                     result, ps2All->name, NULL, NULL);

        /* Set up the necessary callbacks */
        RxPipelineNodePS2AllSetCallBack(
            plNode,
            rxPS2ALLCALLBACKOBJECTSETUP,
            MyPS2AllObjectSetupCB);
        RxPipelineNodePS2AllSetCallBack(
            plNode,
            rxPS2ALLCALLBACKOBJECTFINALIZE,
            MyPS2AllObjectFinalizeCB);

        /* Here we can optionally make this pipeline always
         * use a specific PS2AllMat-based material pipeline
         * (ignoring RpMaterial pipeline pointers) */
        /* RxPipelineNodePS2AllGroupMeshes(
         *     plNode, myPS2AllMatPipe); */

        return(result);
    }
}

RxPipelineDestroy(newPipe);
```

```
    }

    return(NULL);
}
```

Execution Order

In brief, the "object setup" callback is called once for each object, at the beginning of PS2All pipeline execution. After this, the appropriate PS2AllMat-based material pipeline is executed for each **RpMesh** in the object (either the pipeline associated with each **RpMesh**'s **RpMaterial**, or the pipeline specified in a call to **RxPipelineNodePS2AllGroupMeshes()**). Finally, the "object finalize" callback is called once for each object, at the end of pipeline execution.

The PS2AllMat pipeline and the related node setup API will be described in the next section.

1.2.2 Construction of PS2AllMat Pipelines

Pipeline Structure

PS2AllMat-based material pipelines should be constructed solely from the PS2AllMat node. There are no exceptions to this rule and it will be tested with asserts in the debug builds of the RenderWare Graphics libraries.

Pipeline Node Setup API

The API functions used for setting up the PS2AllMat node, during pipeline construction, are:

- **RxPipelineNodePS2AllMatSetCallBack()**
- **RxPipelineNodePS2AllMatGenerateCluster()**
- **RxPipelineNodePS2AllMatSetVU1CodeArray()**
- **RxPipelineNodePS2AllMatSetVUBufferSizes()**
- **RxPipelineNodePS2AllMatSetPointListVUBufferSize()**
- **RxPipelineNodePS2AllMatSetVIFOffset()**

RxPipelineNodePS2AllMatSetCallBack()

RxPipelineNodePS2AllMatSetCallBack() allows the user to set the callbacks called by the PS2AllMat node at different points during its execution. This function should be called after the pipeline has been unlocked. There are five callback types supported by the PS2AllMat node, as follows:

- **RxPipelineNodePS2AllMatMeshInstanceTestCallBack**
- **RxPipelineNodePS2AllMatResEntryAllocCallBack**
- **RxPipelineNodePS2AllMatInstanceCallBack**
- **RxPipelineNodePS2AllMatBridgeCallBack**
- **RxPipelineNodePS2AllMatPostMeshCallBack**

These callbacks execute, for each **RpMesh**, in the order listed above. They are described in the following section, *PS2All Customization*.

RxPipelineNodePS2AllMatGenerateCluster()

Because PS2All and PS2AllMat are optimized in a highly platform-specific manner, they do not use packets at all and use only a small set of modified, PS2-specific clusters (specified by the enumeration type **RxPS2ClusterType**). **RxPipelineNodePS2AllMatGenerateCluster()** allows the user to specify the "clusters" which are to be instanced and processed, for each **RpMesh** rendered by the current PS2AllMat pipeline. PS2-specific attributes (specified by the enumeration types **RxPS2ClusterAttrib** and **RxPS2ClusterFormatAttrib**) are attached to these clusters to define their format for DMA transfer to VU1 and how their data should be instanced. **RxPipelineNodePS2AllMatGenerateCluster()** should be called before the pipeline has been unlocked.

The API reference documentation for **RxPipelineNodePS2AllMatGenerateCluster()**, **RxPS2ClusterType**, **RxPS2ClusterAttrib** and **RxPS2ClusterFormatAttrib** may be consulted for further details on the use of clusters in PS2AllMat.

RxPipelineNodePS2AllMatSetVU1CodeArray()

RxPipelineNodePS2AllMatSetVU1CodeArray() allows the user to attach an array of pointers to VU1 microcode chunks to the current PS2AllMat pipeline. Each chunk is a segment of code, conforming to standards described elsewhere (see the *PS2 Vector Code* white paper), which is capable of performing transformation and lighting (and other arbitrary rendering effects) of incoming geometric data before submitting 2D primitives to the Graphics Synthesizer (or GS) for rasterization. An array of such code chunks is provided with RenderWare Graphics and is used by default. The appropriate code chunk is uploaded to VU1 by the bridge callback (if it is not there already!) according to an index set up in the object setup callback (see the object setup callback sub-section in the following *PS2All Customization* section). **RxPipelineNodePS2AllMatGetVU1CodeArray()** can be used to retrieve the VU1 code array from an existing PS2AllMat material pipeline. Both these functions should be called after the pipeline has been unlocked.

RxPipelineNodePS2AllMatSetTriangleVUBufferSizes(), etc

RxPipelineNodePS2AllMatSetTriangleVUBufferSizes() is used to define the input buffer sizes for geometry transferred to VU1 by the DMA engine (which uses a double-buffering scheme). This function should be used if the current pipeline is to be used to render triangle-based objects (tri-strips or tri-lists – note that tri-fans are converted into tri-lists at instance time as they are not supported by RenderWare's VU1 vector code).

If you wish to render line-based objects (line-lists or poly-lines) then use the function **RxPipelineNodePS2AllMatSetLineVUBufferSizes()** to set up your buffer sizes. If you wish to render point based objects (point-lists) then use **RxPipelineNodePS2AllMatSetPointListVUBufferSize()**.

Note that point-lists require custom VU1 vector code. Point-list vertices may be interpreted in arbitrary ways, there is no implicit topology – for example, vertices could be interpreted as being the control points of Bézier patches.

A PS2AllMat material pipeline can only render one of triangles, lines or points, not a mixture (though point-lists could be interpreted by custom VU1 vector code as being any or all of these types). A triangle pipeline can render tri-lists, tri-strips or tri-fans (emulated, as mentioned above) and a line pipeline can render line-lists or poly-lines. This may change in the future.

RxPipelineNodePS2AllMatGetVUBatchSize() can be used, after the pipeline has been unlocked, to determine the final size of the batches (calculated during pipeline unlock from the input buffer sizes) in which the geometry data will be transferred to VU1. This size applies to the primitive type passed in (it will likely be different for tri-strips and tri-lists or poly-lines and line-lists) and may have been rounded down from the value you initially specified.

RxPipelineNodePS2AllMatSetVIFOffset()

RxPipelineNodePS2AllMatSetVIFOffset() is used to set the location of the second geometry input buffer in VU1 memory.

Further detail can be found in the API reference documentation for the PS2All callback types and API functions.

Example Code

To summarize, the construction of a PS2AllMat-based material pipeline proceeds as follows:

1. Create a pipeline;
2. Lock the pipeline for editing;
3. Add a fragment to the pipeline, containing *only* the PS2AllMat node;
4. Call pre-unlock PS2AllMat node setup API functions: (**RxPipelineNodePS2AllMatGenerateCluster()** for all desired clusters and **RxPipelineNodePS2AllMatSetVUBufferSizes()** or **RxPipelineNodePS2AllMatSetPointListVUBufferSize()**);
5. Unlock the pipeline;
6. Call post-unlock PS2AllMat node setup API functions (**RxPipelineNodePS2AllMatSetVIFOffset()**, **RxPipelineNodePS2AllMatSetVU1CodeArray()** and **RxPipelineNodePS2AllMatSetCallBack()** for all callback types);

Some example code demonstrating this process follows:

```
RxPipeline *
CreateMyPS2AllMatPipeline(void)
{
    RxPipeline *newPipe;

    newPipe = RxPipelineCreate();
    if (NULL != newPipe)
    {
        RxLockedPipe *lockedPipe;

        lockedPipe = RxPipelineLock(newPipe);
        if (NULL != lockedPipe)
        {
            RxNodeDefinition *ps2AllMat;
            RxPipelineNode   *plNode;
```

```
RxPipeline       *result;

ps2AllMat = RxNodeDefinitionGetPS2AllMat();
assert(NULL != ps2AllMat);

/* PS2AllMat-based material pipelines are trivial,
 * (always) containing just the PS2AllMat node */
lockedPipe = RxLockedPipeAddFragment(lockedPipe,
                                     NULL,
                                     ps2AllMat,
                                     NULL);
assert(NULL != lockedPipe);

/* Get a pointer to the PS2AllMat
 * node in the locked pipeline */
plNode = RxPipelineFindNodeByName(lockedPipe,
                                  ps2AllMat->name,
                                  NULL,
                                  NULL);
assert(NULL != plNode);

/* Set up the PS2AllMat node to generate clusters
 * "XYZ", "UV", "RGBA" and "NORMAL" */
RxPipelineNodePS2AllMatGenerateCluster(plNode,
                                       &RxClPS2xyz,
                                        CL_XYZ);
RxPipelineNodePS2AllMatGenerateCluster(plNode,
                                       &RxClPS2uv,
                                        CL_UV);
RxPipelineNodePS2AllMatGenerateCluster(plNode,
                                       &RxClPS2rgba,
                                        CL_RGBA);
RxPipelineNodePS2AllMatGenerateCluster(plNode,
                                       &RxClPS2normal,
                                        CL_NORMAL);


/* Set up VU1 input buffer sizes (standard values) */
RxPipelineNodePS2AllMatSetVUBufferSizes(
    plNode,
    _rwskyStrideOfInputCluster,
    _rwskyTSVertexCount,
    _rwskyTLTriCount);

/* Unlock the pipeline */
result = RxLockedPipeUnlock(lockedPipe);

if (NULL != result)
{
```

```
                    /* Get a pointer to the PS2AllMat
                     * node in the unlocked pipeline */
                    plNode = RxPipelineFindNodeByName(
                        lockedPipe, ps2AllMat->name, NULL, NULL);
                    assert(NULL != plNode);

                    /* Set up the VU1 VIFOffset (standard value) */
                    RxPipelineNodePS2AllMatSetVIFOffset(
                        plNode, _rwskyVIFOffset);

                    /* Overload the array of VU1 microcode chunks */
                    RxPipelineNodePS2AllMatSetVU1CodeArray(
                        plNode, myCodeArray, myCodeArrayLength);

                    /* Set up the necessary callbacks */
                    RxPipelineNodePS2AllMatSetCallBack(
                        plNode,
                        rxPS2ALLMATCALLBACKMESHINSTANCETEST,
                        MyPS2AllMatMeshInstanceTestCallBack);
                    RxPipelineNodePS2AllMatSetCallBack(
                        plNode,
                        rxPS2ALLMATCALLBACKRESENTRYALLOC,
                        MyPS2AllMatResEntryAllocCallBack);
                    RxPipelineNodePS2AllMatSetCallBack(
                        plNode,
                        rxPS2ALLMATCALLBACKINSTANCE,
                        MyPS2AllMatInstanceCallBack);
                    RxPipelineNodePS2AllMatSetCallBack(
                        plNode,
                        rxPS2ALLMATCALLBACKBRIDGE,
                        MyPS2AllMatBridgeCallBack);
                    RxPipelineNodePS2AllMatSetCallBack(
                        plNode,
                        rxPS2ALLMATCALLBACKPOSTMESH,
                        MyPS2AllMatPostMeshCallBack);

                    return(result);
                }
            }

            RxPipelineDestroy(newPipe);
        }

        return(FALSE);
    }
```

The following section describes each of the callback types used by the PS2All and PS2AllMat nodes in greater detail and considers their potential for customization and optimization.

1.3 PS2All Customization

1.3.1 Overview

This section will give a brief overview of each callback type used by PS2All and PS2AllMat. As part of this, it will discuss the possibilities for customization and optimization of each callback. Here is a list of all the callback types:

- **RxPipelineNodePS2AllObjectSetupCallBack**
- **RxPipelineNodePS2AllObjectFinalizeCallBack**
- **RxPipelineNodePS2AllMatMeshInstanceTestCallBack**
- **RxPipelineNodePS2AllMatResEntryAllocCallBack**
- **RxPipelineNodePS2AllMatInstanceCallBack**
- **RxPipelineNodePS2AllMatBridgeCallBack**
- **RxPipelineNodePS2AllMatPostMeshCallBack**

The `RxPS2AllPipeData` structure

The **`RxPS2AllPipeData`** structure is used throughout PS2All and PS2AllMat execution to pass essential information between callbacks. It is initialized to a blank or default state at the beginning of the pipeline and its members are set up as necessary as it is passed from callback to callback. The structure is shown here:

```
struct RxPS2AllPipeData
{
    /* Per-object stuff */
    struct rxNodePS2AllPvtData    *objPvtData;
    struct rxNodePS2AllMatPvtData *matPvtData;
    void                          *sourceObject;
    RpMeshHeader                  *meshHeader;
    RwMeshCache                   *meshCache;
    RxInstanceFlags                objInstance;
    RwUInt32                       objIdentifier;
    RwReal                         spExtra;
    RwInt32                        numMorphTargets;
    RwUInt32                       fastMorphing;
    RwUInt8                        transType;
    RwUInt8                        primType;
    RwUInt8                        matModulate;

    /* The following change per-mesh */
    RwUInt8                        vu1CodeIndex;
    const RpMesh                  *mesh;
    RwResEntry                   **cacheEntryRef;
    RxInstanceFlags                meshInstance;
    RwUInt32                       meshIdentifier;
    RwSurfaceProperties           *surfProps;
    RwTexture                     *texture;
    RwRGBA                         matCol;
};
```

A brief description of each of the members of this structure follows. The relationship of each one to the various PS2All and PS2AllMat callback types will be described in the following sub-sections dealing with the callback types.

The following members will generally be specified once per object and persist for all **`RpMesh`**s in the object (though exceptions can be made to this rule):

- The **`objPvtData`** member holds a pointer to the current PS2All node's private data (this is valid through PS2AllMat execution). matPvtData holds a pointer to the private data of the currently executing PS2AllMat node. This changes (potentially) for each **`RpMesh`** in the object and is only valid once **`RpMesh`**s processing has begun.

- The **sourceObject** member holds a void pointer to the "source object", that which is being rendered (e.g. an **RpAtomic** or an **RpWorldSector**). The meshHeader member holds a pointer to the **RpMeshHeader** from within the source object. The meshCache member holds a pointer to the **RwMeshCache** associated with the source object (this holds a pointer to instance data for each of the **RpMesh**s in the object).

- The **objInstance** member is of type **RxInstanceFlags** and specifies to what degree instancing should be performed at the per-object level (i.e. for all **RpMesh**s in the object). See the API reference documentation for the function **RxPipelineNodePS2AllSetCallBack()** for further details.

- The **objIdentifier** member is an arbitrary "identifier" for the current object. It is stored with instance data and changes in it are used to determine whether the object has been modified to the extent that it requires reinstancing.

- **spExtra** is an **RwReal** value that is uploaded to VU1 in the spare **w** component of the quad-word describing surface properties. This can be used as a parameter for custom VU1 code (for example, VU1 code used by the **RpMorph** plugin uses this value as the interpolant between key-frames).

- **numMorphTargets** and **fastMorphing** are both used internally and should not be modified. They will probably be removed at a later date.

- The **transType** member is of type **RxSkyTransTypeFlags** (truncated to eight bits to save space). This is used to specify the type of VU1 code to use for processing the **RpMesh**s in the current object (e.g. perspective-projected triangles with fogging and back-face culling, or parallel-projected lines with view frustum clipping).

- The **primType** member is an eight-bit code for the GS primitive type (e.g. tri-strips or line-lists) which the VU1 code will submit to the GS for rasterization. These primitive types are described in the GS manual.

- The **matModulate** member is a boolean which specifies whether or not the color of the **RpMaterial** of each **RpMesh** in the current object should modulate the color of the object's vertices (which are calculated on VU1 as the sum of pre-lighting colors, if present, and any contributions from light-sources).

The following members will generally change on a per-**RpMesh** basis (though exceptions can be made to this rule):

- The **vu1CodeIndex** member specifies the index of the VU1 code chunk to be used to render the current **RpMesh** (this index selects the code from the array specified by **RxPipelineNodePS2AllMatSetVU1CodeArray()**). This will be based on the **transType** member, on knowledge of the items in the VU1 code array and on the properties of the current **RpMesh**.

- The **mesh** member simply points to the current **RpMesh**. The **cacheEntryRef** member is a doubly indirected pointer to the **RwResEntry** holding the instance data for the current mesh. As will be explained in the description of the "res-entry-alloc" callback below, this need not necessarily lie within the resources arena.

- The **meshInstance** member complements the **objInstance** member. It must specify at least the same degree of instancing as the **objInstance** member, but may specify a higher degree. See the API reference documentation for the function **RxPipelineNodePS2AllSetCallBack()** for further details.

- The **meshIdentifier** member is similar to the **objIdentifier** member. It is an arbitrary "identifier" for the current **RpMesh**. It is stored with the **RpMesh**'s instance data and changes in it are used to determine whether the **RpMesh** has been modified to the extent that it requires reinstancing.

- The **surfProps** member is simply a pointer to an **RwSurfaceProperties** structure describing the surface lighting properties of the current **RpMesh**. The **texture** member is a pointer to an **RwTexture** to be used for the current **RpMesh** and the matCol member is an **RwRGBA** structure describing the material color of the current **RpMesh**.

Each of the PS2All and PS2AllMat callback types will now be described in turn.

1.3.2 The ObjectSetup CallBack

The **RxPipelineNodePS2AllObjectSetupCallBack** is called once per object, at the beginning of pipeline execution. Its purpose is to perform per-object processing and to set up some members of the **RxPS2AllPipeData** structure for use in subsequently executed material pipelines. This callback is required for the pipeline to function.

Here follows the prototype for the **RxPipelineNodePS2AllObjectSetupCallBack**:

```
typedef RwBool (*RxPipelineNodePS2AllObjectSetupCallBack)
    ( RxPS2AllPipeData *ps2AllPipeData,
      RwMatrix **transform,
      RxWorldApplyLightFunc lightingFunc );
```

The **transform** parameter contains a doubly indirected pointer to the transformation matrix to be used to transform the object from object-space to camera-space. The **lightingFunc** parameter is a pointer to a function that will add a light to the lighting data buffer that will be uploaded to VU1 after the completion of the callback. The return value of the callback is a boolean – if the callback returns **FALSE**, then the pipeline will be prematurely terminated and the object will not be rendered.

The object setup callbacks used in the default PS2All-based RenderWare Graphics pipelines perform a number of standard tasks. Given application-specific knowledge that a developer may have, they may create an object setup callback that performs significantly less work than the default callbacks. Custom object setup callbacks may be constructed from the same helper macros that are used in the default callbacks. The default object setup callbacks and the macros that they use are described in detail in the API reference documentation for:

- **RpAtomicPS2AllObjectSetupCallBack**
- **RpWorldSectorPS2AllObjectSetupCallBack**
- **RwIm3DPS2AllObjectSetupCallBack**

Here follows an overview of some of the standard tasks performed by the default object setup callbacks (in execution order), as well as possibilities for optimization of these tasks in custom object setup callbacks:

1. 1. All object setup callbacks *must* set up the **meshHeader** and **meshCache** members of the **RxPS2AllPipeData** struct in order for the rest of the pipeline to function.

2. Object-level instance testing may then be performed in order to set up the **objInstance** member of **RxPS2AllPipeData**. This may be skipped in custom object setup callbacks if it is known that objects are never modified at run-time and so will never require reinstancing.

3. The **RwMatrix** used to transform the object into camera-space should then be set up using the **transform** parameter. If the pointer referenced by this parameter is set to **NULL** then the **RwMatrix** currently in VU1 memory will be allowed to persist. This is more efficient in the case of, for example, **RpWorldSector**s, all of which in an **RpWorld** will require exactly the same transformation matrix (since the geometry of an **RpWorldSector** is specified in world-space).

4. The **transType** and **primType** members of **RxPS2AllPipeData** should next be set up. This may involve a frustum test and inspection of the **RpMeshHeaderFlags** of the current object. However, with specialized knowledge about the object, setting up these members may very simple indeed. For example, it might be known that the object being rendered is always composed of tri-strips and will always be fully onscreen and so will never require view frustum clipping.

5. The **matModulate** member of **RxPS2AllPipeData** may be set up on the basis of flags in the object or with specialized knowledge it may be set unconditionally.

6. Setting up the lighting data to be sent to VU1 for the current object involves determining which lights affect the current object, adding them to the lighting data buffer and setting up the inverse lighting matrix (this is used on VU1 to transform lights into object-space). It is possible, however, to allow existing lighting data to persist in VU1 memory, with the optional update of the inverse lighting matrix. For further details on this, see the API reference documentation for the helper macro **RpAtomicPS2AllLightingPersist**.

Further detailed information on the responsibilities and functioning of the object setup callback and on the default **RpAtomic**, **RpWorldSector** and **RwIm3D** render pipeline object setup callbacks can be found in the API reference documentation for, respectively:

- **RxPipelineNodePS2AllObjectSetupCallBack**
- **RpAtomicPS2AllObjectSetupCallBack**
- **RpWorldSectorPS2AllObjectSetupCallBack**
- **RwIm3DPS2AllObjectSetupCallBack**

1.3.3 The ObjectFinalize CallBack

The **RxPipelineNodePS2AllObjectFinalizeCallBack** is called at the end of pipeline execution and its purpose is to perform any necessary per-object tasks which must take place after the processing, by material pipelines, of all **RpMesh**s in the object. This callback is optional and may be set to NULL. It is in fact not used by any of the default PS2All-based RenderWare Graphics pipelines.

Here follows the prototype for the **RxPipelineNodePS2AllObjectFinalizeCallBack**:

```
typedef RwBool (*RxPipelineNodePS2AllObjectFinalizeCallBack)
    ( RxPS2AllPipeData *ps2AllPipeData );
```

1.3.4 The MeshInstanceTest CallBack

The **RxPipelineNodePS2AllMatMeshInstanceTestCallBack** is used to determine whether the current **RpMesh** has been edited – if so, its data may have to be reinstanced. For further information on the relationship between object-level and mesh-level instancing and the different degrees of reinstancing, see the API reference documentation for **RxPipelineNodePS2AllSetCallBack()** and **RxPipelineNodePS2AllMatMeshInstanceTestCallBack()**.

Here follows the prototype for the **RxPipelineNodePS2AllMatMeshInstanceTestCallBack**:

```
typedef RwBool (*RxPipelineNodePS2AllMatMeshInstanceTestCallBack)
    ( RxPS2AllPipeData *ps2AllPipeData );
```

The **RxPipelineNodePS2AllMatMeshInstanceTestCallBack** is optional. If it is known that source mesh data is not edited at run-time and so will not need re-instancing, it will improve pipeline performance to set this callback to NULL.

1.3.5 The ResEntryAlloc CallBack

The **RxPipelineNodePS2AllMatResEntryAllocCallBack** is used to allocate memory for instance data. The default callback, **RpAtomicPS2AllResEntryAllocCallBack()**, allocates memory from the RenderWare Graphics resources arena. You may replace this callback in order to allocate instance data memory elsewhere (for instance, there is rudimentary support in RenderWare Graphics for "persistent" instance data, which is created once, or streamed from disk at load-time, and never modified again. The resource arena should never throw this data away, so it can be allocated outside the arena by a custom "res-entry-alloc" callback). If instancing occurs then this callback will be required (in the case of persistent instance data loaded from disk, it may be omitted).

Here follows the prototype for the **RxPipelineNodePS2AllMatResEntryAllocCallBack**:

```
typedef RwResEntry *(*RxPipelineNodePS2AllMatResEntryAllocCallBack)
    ( RxPS2AllPipeData *ps2AllPipeData,
      RwResEntry **repEntry,
      RwUInt32 size,
      RwResEntryDestroyNotify destroyNotify );
```

The **repEntry** parameter is a pointer to the pointer, in the **RwMeshCache** of the current object, to the **RwResEntry** for the current **RpMesh**. It should be set up by the callback once it has allocated and initialized an **RwResEntry**. The **size** parameter specifies the size of the data to be allocated within the **RwResEntry**. The **destroyNotify** parameter is a pointer to an **RwResEntryDestroyNotify** function, which should be called before the **RwResEntry** is destroyed to ensure that all DMA transfers involving the contained data have been completed before the memory is freed.

This callback does not affect performance since it is only called when instancing occurs (and if instancing occurs too often then performance will be poor anyway).

Further detailed information on the responsibilities and functioning of the **RxPipelineNodePS2AllMatResEntryAllocCallBack** and on the default callback can be found in the API reference documentation for, respectively: **RxPipelineNodePS2AllMatResEntryAllocCallBack** and **RpMeshPS2AllResEntryAllocCallBack**.

1.3.6 The Instance CallBack

The **RxPipelineNodePS2AllMatInstanceCallBack** is used to instance the data of an **RpMesh** into a resources arena entry (allocated by the prior **RxPipelineNodePS2AllMatResEntryAllocCallBack**). If the source object contains non-standard data (such as per-vertex weights used in skinning) then code must be put in this callback to instance this data into the appropriate format. Instancing is a complex process (it involves laying out data in the form expected by the DMA engine) and is dealt with in detail in the API reference documentation for the **RxPipelineNodePS2AllMatInstanceCallBack** callback type. Also see the notes on the **RxPipelineNodePS2AllMatGenerateCluster()** node API function above. If instancing occurs then this callback will be required (in the case of persistent instance data loaded from disk, as introduced in the previous section, it may be omitted).

This callback does not affect performance since it is only called when instancing occurs (and if instancing occurs too often then performance will be poor anyway).

1.3.7 The Bridge CallBack

The **RxPipelineNodePS2AllMatBridgeCallBack** is used to upload various pieces of information (controlling render state, texture uploads and the VU1 code to be used, in addition to extra data potentially required by custom VU1 code) to VU1, including, lastly, the mesh's instance data. This process is quite involved so refer to the API reference documentation for the **RxPipelineNodePS2AllMatBridgeCallBack** callback type for details. This callback is required for the pipeline to function.

1.3.8 The PostMesh CallBack

The **RxPipelineNodePS2AllMatPostMeshCallBack** is called after an **RpMesh**'s data has been dispatched to VU1 for processing. It may be used to perform tasks necessary after the rendering of a mesh. This callback is optional and may be set to NULL. It is in fact only used by the default PS2All-based RenderWare Graphics pipelines in a metrics build of the libraries, and in this case it is used to gather information on the number of polygons rendered.

Here follows the prototype for the **RxPipelineNodePS2AllMatPostMeshCallBack**:

```
typedef RwBool (*RxPipelineNodePS2AllMatPostMeshCallBack)
    ( RxPS2AllPipeData *ps2AllPipeData );
```

1.4 Default PS2All Pipelines

This section describes the default PS2All-based pipelines provided with RenderWare Graphics for PS2.

1.4.1 RwIm3D

Because there is no default PS2 pipeline for **RwIm3D** to use and PS2All is a part of the **RpWorld** library, **RwIm3D** requires that the **RpWorld** library is attached to your application. This may change in future revisions of RenderWare Graphics.

RwIm3D Pipelines

As detailed in the chapter *PowerPipe Overview*, there are two types of pipeline used in **RwIm3D** rendering:

1. Firstly, **RwIm3DTransform()** uses a pipeline to transform vertices from world-space (or object-space now that there is an optional **RwMatrix** parameter to this function) into screen-space;

2. Secondly, **RwIm3DRenderPrimitive()** and **RwIm3DRenderIndexedPrimitive()** both submit triangles, made from the transformed vertices, to the rasterization API.

On PS2, **RwIm3DTransform()** does very little work, so PS2All and PS2AllMat pipelines are used only by the **RwIm3D** render functions.

RpMaterials are not used by **RwIm3D**. Because of this, the PS2All-based **RwIm3D** pipeline used is set up, through its **RxPipelineNodePS2AllGroupMeshes()** construction-time API function, to always use the same PS2AllMat-based **RwIm3D** material pipeline.

Temporary Limitations

Due to a memory-allocation limitation in the current PS2 RenderWare Graphics core library, additional code is used in the PS2All-based RwIm3D pipeline to split up primitive above a certain size. This is the reason for the presence of the "internal" callback types in rpworld.h:

- **RxPipelineNodePS2AllMatIm3DPreMeshCallBack**
- **RxPipelineNodePS2AllMatIm3DPostMeshCallBack**
- **RxPipelineNodePS2AllMatIm3DMeshSplitCallBack**

This limitation, and hence the additional code, will be removed in a future revision of the library.

PS2All and PS2AllMat CallBacks for RwIm3D

The callbacks used by the PS2All and PS2AllMat nodes in the **RwIm3D** render pipelines are:

- **RwIm3DPS2AllObjectSetupCallBack()**
- **RwIm3DPS2AllIm3DPreMeshCallBack()**
- **RwIm3DPS2AllResEntryAllocCallBack()**
- **RwIm3DPS2AllInstanceCallBack()**
- **RwIm3DPS2AllBridgeCallBack()**
- **RwIm3DPS2AllIm3DPostMeshCallBack()**
- **RwIm3DPS2AllMeshSplitCallBack()**
- **RwIm3DPS2AllPostMeshCallBack()**

The API reference documentation for these callbacks contains further detailed information on their functioning and the macros from which they are composed (and which can be used in constructing callbacks for custom **RwIm3D** pipelines).

Here follows a brief overview of how the functionality of each of these callbacks is specific to **RwIm3D**. The callbacks used only to facilitate the splitting of large primitives, as mentioned above, are omitted.

RwIm3DPS2AllObjectSetupCallBack()

In **RwIm3DPS2AllObjectSetupCallBack(),** there is no real **RpMeshHeader** or **RwMeshCache** to retrieve from the **RwIm3D** "object" being rendered, so fake ones are used. The **RwIm3D** "object" always needs instancing (since its instance data is not cached from frame to frame) so the appropriate value is set in the **objInstance RxPS2AllPipeData** member. The transformation matrix is set up from the (optional) matrix passed as a parameter to **RwIm3DTransform()**. View frustum testing is based only on the clipping hint passed as a parameter to **RwIm3DTransform()**.By default, no lighting is performed for **RwIm3D**, though a custom **RwIm3D** object setup callback could perform lighting.

RwIm3DPS2AllResEntryAllocCallBack()

RwIm3DPS2AllResEntryAllocCallBack() uses a custom allocator to allocate instance data for **RwIm3D** rendering. This is necessary because this data is thrown away immediately after use (rather than being cached from frame to frame as occurs for retained mode rendering).

RwIm3DPS2AllInstanceCallBack()

RwIm3DPS2AllInstanceCallBack() creates instance data from the **RwIm3DVertex** array passed to **RwIm3DTransform()** and the **RwImVertexIndex**s passed to the **RwIm3D** render function.

RwIm3DPS2AllBridgeCallBack()

RwIm3DPS2AllBridgeCallBack() performs no texture upload and no texture-related render state setup, since render state persists through **Im3D** rendering.

RwIm3DPS2AllPostMeshCallBack()

RwIm3DPS2AllPostMeshCallBack() performs some work related to splitting large **RwIm3D** primitives, but it also gathers metrics information in a metrics build of the libraries.

1.4.2 RpAtomic

The default PS2All **RpAtomic** and **RpWorldSector** object pipelines use the same default PS2AllMat material pipeline, so that will be described in a subsequent section.

The only callback used by the default PS2All **RpAtomic** object pipeline is **RpAtomicPS2AllObjectSetupCallBack()**. The API reference documentation for this callback type contains further detailed information on its functioning and the macros from which it is composed (and which can be used in constructing callbacks for custom **RpAtomic** pipelines).

Here follows a brief overview of how the functionality of **RpAtomicPS2AllObjectSetupCallBack()** is specific to **RpAtomic**s:

1. Morphing information is set up if the current **RpAtomic** is morph animated;

2. The object is tested for reinstancing requirements, by inspection of its **RpGeometry**, **RpMeshHeader** and **RpInterpolator;**

3. Dirty flags are cleared in the **RpGeometry** and **RpInterpolator;**

4. The object's transform is next set up, as the concatenation of its Local Transformation Matrix (or LTM) and the current camera's view matrix;

5. The object's bounding sphere is used to test it for intersection and culling by the current camera's view frustum;

6. Finally, lighting data is set up. The inverse lighting matrix is set up as the inverse of the object's LTM and then all global lights and local lights whose regions of influence overlap the **RpWorldSector**(s) in which the current object lie are added to the lighting data buffer.

1.4.3 RpWorldSector

The only callback used by the default PS2All **RpWorldSector** object pipeline is **RpWorldSectorPS2AllObjectSetupCallBack()**. The API reference documentation for this callback type contains further detailed information on its functioning and the macros from which it is composed (and which can be used in constructing callbacks for custom **RpWorldSector** pipelines).

Here follows a brief overview of how the functionality of **RpWorldSectorPS2AllObjectSetupCallBack()** is specific to **RpWorldSector**s:

1. Firstly, the object is tested for reinstancing requirements, by inspection of its **RpMeshHeader**;
2. The object's transform is set up as the current camera's view matrix (since **RpWorldSector** geometry is specified in world-space);
3. The object's bounding box is used to test it for intersection and culling by the current camera's view frustum;
4. Finally, lighting data is set up. The inverse lighting matrix is set up as the identity matrix and then all global lights and lights whose regions of influence overlap the **RpWorldSector** are added to the lighting data buffer.

1.4.4 RpMaterial

As mentioned above, the default PS2AllMat material pipeline is used by both the **RpAtomic** and **RpWorldSector** default PS2All object pipelines. This is for convenience and while it involves some extra predication being added to this material pipeline, it is expected that custom material pipelines constructed by developers will be specific to the objects they are known to be applied to. One of the difficulties in writing a general-purpose library is that it will be used in many different circumstances, each of which might, in isolation, benefit from *different* optimizations. This is why RenderWare Graphics has been made so customizable, allowing developers to take advantage of the knowledge they have of the specific circumstances found in their application.

Default RpMaterial PS2AllMat CallBacks

The callbacks used by the PS2AllMat node in the default material pipeline are:

- **RpMeshPS2AllMeshInstanceTestCallBack()**
- **RpMeshPS2AllResEntryAllocCallBack()**
- **RpMeshPS2AllInstanceCallBack()**
- **RpMeshPS2AllBridgeCallBack()**
- **RpMeshPS2AllPostMeshCallBack()**

The API reference documentation for these callbacks contains further detailed information on their functioning and the macros from which they are composed (and which can be used in constructing callbacks for custom material pipelines).

RpMeshPS2AllMeshInstanceTestCallBack()

RpMeshPS2AllMeshInstanceTestCallBack() tests whether the number of vertices in the current mesh has changed to determine if it needs reinstancing.

RpMeshPS2AllResEntryAllocCallBack()

RpMeshPS2AllResEntryAllocCallBack() determines the type of the current object (is it an **RpAtomic** or **RpWorldSector**?) and on this basis allocates an **RwResEntry** in the appropriate manner for the instance data of the current **RpMesh**. The allocation for **RpAtomic**s predicates on whether or not the object is morph animated (if so, then instance data is attached to the **RpAtomic** and not the **RpGeometry**, since several **RpAtomic**s referencing the same **RpGeometry** could be in different animation states at the same time) or, if not, whether its **RpGeometry** is to have persistent instance data (as mentioned in the section above on the res-entry-alloc callback). The **RwResEntry** for an **RpWorldSector** is always allocated in the resources arena in the standard manner.

RpMeshPS2AllInstanceCallBack()

RpMeshPS2AllInstanceCallBack() also determines the type of the current object and on this basis creates instance data for the current **RpMesh**, instancing just the standard clusters which are present in the current object (see the API reference documentation for **RxPipelineNodePS2AllMatInstanceCallBack** and **RxPipelineNodePS2AllMatGenerateCluster()** for further details).

RpMeshPS2AllBridgeCallBack()

RpMeshPS2AllBridgeCallBack() performs all of the standard tasks for a bridge callback. It uploads the texture for the current **RpMesh** if it is not already resident in GS memory, and sets up render state for that texture. It uploads a GIF tag to the GS containing the primType member of **RxPS2AllPipeData**. It uploads the material color and surface properties of the current **RpMesh**'s **RpMaterial** (the latter containing an extra **RwReal** of plugin data such as the **RpMorph** key-frame interpolant). It uploads clipping information used to toggle the behavior of VU1 code and to facilitate clipping where necessary. It selects and (if it is not already resident in VU1 memory) uploads the appropriate VU1 code chunk, based on the vu1CodeIndex member of **RxPS2AllPipeData**. Finally, it kicks off the transfer of geometry data to VU1.

RpMeshPS2AllPostMeshCallBack()

RpMeshPS2AllPostMeshCallBack() merely gathers metrics information in a metrics build of the libraries.

1.5 Common Traps and Pitfalls

When constructing PS2All or PS2AllMat pipelines and custom callbacks for use therewith, it is highly recommended that you read the API reference documentation pertaining to the relevant API functions and callback types (constructing instance and bridge callbacks is a particularly technical task). PS2 is an unforgiving platform and crashes are often difficult to debug, especially where DMA transfer is involved. Whilst this chapter gives a useful, top-down overview of PS2All, it is a complement rather than a substitute for the API reference documentation, which is detailed and comprehensive.

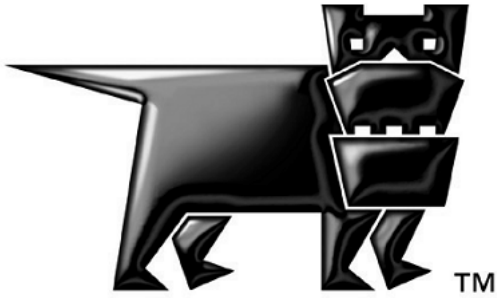
When optimizing pipeline code for performance on PS2, bear in mind that the main factor influencing performance (this applies to the performance of CPU-side code execution, not VU1 execution nor GS rasterization) is the CPU cache, with I-Cache usually being more significant than D-Cache. When system bus usage is high (if lots of geometry is being pushed through VU1 or much texture data is being uploaded to the GS each frame), CPU cache misses will cost more. Keeping code small and program execution localized (reduce the number of function calls taken) will improve performance, as will making data access more coherent (use local variables instead of globals where possible, avoid switch statements, etc).

1.6 Summary

This chapter has provided an overview of the PS2All rendering pipeline architecture for RenderWare Graphics on PS2. It has explained the purpose and benefits of this architecture. It has covered the construction of PS2All-based object pipelines and PS2AllMat-based material pipelines, describing the construction-time API functions used during this process. It has given descriptions of the callback types used within PS2All and PS2AllMat pipelines and has offered advice on the possibilities for optimization of custom callbacks on the basis of application-specific knowledge. Finally, it has described the pertinent aspects of the default PS2All-based and PS2AllMat-based pipelines supplied with RenderWare Graphics for PS2. Hopefully this has provided a useful top-down introduction to PS2All, in the context of which the copious details in the API reference documentation will be more comprehensible and more easily applied in the use of PS2All.

Chapter 2

Pipeline Delivery System



2.1 Introduction

2.1.1 What is the Pipeline Delivery System (PDS)?

The Pipeline Delivery System (PDS) has been designed to maintain and control RenderWare Graphics' growing number of rendering pipelines on the PlayStation 2. Initially it might not be obvious why the PDS is necessary. However we believe it will be essential to games developers who want to have complete control over RenderWare Graphics and obtain the highest possible performance from their target platform.

2.1.2 Why use the PDS?

RenderWare Graphics generic pipelines

The generic rendering pipelines have been moved out of the core and the world, and into the RenderWare Graphics PDS. We have implemented the PDS in a plugin called **RpPDS**. This means that any application that uses the platform specific generic rendering and platform specific immediate mode pipelines from the world must also attach the **RpPDS** plugin.

RenderWare Graphics plugin pipelines

Most of the plugin rendering pipelines have also been moved out of their respective plugin and into the PDS. The following plugins have moved their pipelines into the PDS: **RpWorld**, **RpMatFX**, **RpSkin**, **RpPatch**, **RpLtMap**, **RpToon**, **RpTeam** and **Rt2d**. Any application using pipelines from these plugins must also attach the **RpPDS** plugin. See later references for exceptions to this rule.

Reduce final binary size

Previously if a developer attached a plugin that contained rendering pipelines they would be created automatically when the plugin was opened. This meant that, even if the pipelines weren't used, there was a two-fold cost to the application. Firstly the pipelines created used valuable runtime memory and secondly the application contained the extra instruction code. These may seem like two trivial facts to worry about but when we start considering the growing number of pipelines within the RenderWare Graphics plugins and the limited RAM available on the target platform the problem becomes evident.

For example, there are now well over 64 vector code pipelines in the PDS. These micro code programs can each be up to **16Kb** in size. That is a total of **1024Kb** of micro code. As we can see this is already starting to weigh down the application with potentially unused code.

Hence the next section, which neatly circumvents the problem of unwanted rendering pipelines.

Control link dependency

The PDS has been designed to be linker friendly. Previously there was no way for a linker to drop any unused rendering code. The linker had no way of knowing that an application wasn't using the code, and RenderWare Graphics itself was generating a dependency while creating the pipeline.

Hence it is now the application's task to select those rendering pipelines that should be contained in the final binary. The **RpPDS** library has been designed so that there are no link dependencies between RenderWare Graphics and the plugin. Also, the plugin's constituent objects have been designed to allow the linker to drop any unused rendering pipelines and their associated code.

Introduce new custom pipelines

The RenderWare Graphics core and plugins used to contain a full suite of generic rendering pipelines. All the pipelines supported a wide range of features, including true clipping, triangle culling, back and front face culling, fogging and generic lighting. Adding all these features made the pipelines flexible but sometimes at the cost of performance.

The PDS allows us to develop custom versions of these pipelines that support a reduced feature set with the benefit of superior performance. By reducing the feature set of a pipeline we're able to write tighter more efficient rendering code. These custom pipelines can easily be added to the **RpPDS** plugin without effecting any of the previous pipelines, hence making them easily available for the developer.

This release of RenderWare Graphics contains our first set of custom pipelines. We've initially concentrated on writing a set of custom lighting pipelines. These are detailed later in the chapter.

Unified pipelines code

All the pipelines have been rationalized to share as much pipeline code as possible. This has become possible since all the rendering pipelines were updated to use the latest **PS2All** and **PS2AllMat** pipeline nodes. This has reduced the amount of replicated code and in turn will have a positive effect on instruction cache usage.

Adding user pipelines

It is also possible to add external user pipelines to the PDS. This feature may be useful to developers wishing to share and expose their pipelines as if they were default RenderWare Graphics pipelines.

Substitute pipelines

When pipelines are registered by the application the PDS allows the pipeline to be registered with an alternative pipeline identifier. This allows the gross substitution of a pipeline with another. This feature is intended to allow a developer to substitute the generic rendering pipelines with the new custom rendering pipelines.

Customize rendering pipelines

The pipeline registration process is also fully exposed to allow an application to customize any of the rendering pipelines. Instead of forcing the developer to replicate a pipeline to alter it, it's possible to alter the original creation of the pipeline. This is an advanced feature and isn't necessary unless the developer has a need for it.

2.1.3 When not to use the PDS?

While any application using any of the world or plugin rendering pipelines *must* use the PDS to get access to the pipelines, it isn't the only way to create custom pipelines.

Developer pipelines can still be outside

The developer can still develop pipelines outside of the PDS. There is little need to convert any old pipelines into the new PDS structure. The application still has full access to the PS2All nodes and pipeline creation functions.

Not all pipelines are in the PDS system

Contrary to the preceding text not *all* of the RenderWare Graphics pipelines are currently in the PDS. The following plugins have not had their pipelines moved: **`RpCrowd`** and **`RpPTank`**. This is mainly due to their extremely custom nature. They may be integrated into the PDS in the future.

2.1.4 Other documents

API reference

The API Reference provides technical details for the functions and data structures of the **`RpPDS`** plugin, as well providing details for the pipelines and their registration macros.

In the API reference: *Modules→PowerPipe→RpPDS*

PS2All documentation

The pipelines within the PDS are all built off the **`PS2All`** and **`PS2AllMat`** pipeline nodes. For detailed documentation about PS2All and its use see the User Guide chapter PS2All Pipe Overview.

2.2 PDS functionality overview

All the pipelines contained within the PDS have a similar structure. Before we detail an example pipeline let us examine the objects and structures the PDS exposes.

2.2.1 PDS related data objects

Pipeline identifiers

All pipelines within the PDS have a unique identifier. These are defined by the enumeration **RpPDSPipeID**. This identifier is used to tag the pipeline when it is created and retrieve the pipeline before it is attached to a rendering object. The pipeline identifiers are 32-bit numbers, and are constructed with two 16-bit numbers. The upper 16 bits groups the pipelines, whilst the lower 16 bits defines the index in the group. Currently the **RpPDSPipeID** is split into three groups:

Core rendering pipelines

These pipelines were originally in **RpWorld**. These pipelines are RenderWare Graphics' *generic* rendering pipelines. They support all rendering states and can be used to render any object.

Plugin rendering pipelines

These pipelines were originally in the extension plugins. These pipelines perform the *advanced* rendering defined by the plugins. These pipelines support as many of the rendering states as possible. However there are exceptions where it is not practical to support a pipeline feature.

World rendering pipelines

These pipelines are the new custom rendering pipelines. At the release of RenderWare Graphics 3.4 they contain a collection of custom lighting pipelines. These pipelines are substantially quicker than the core and plugin pipelines, at the expense of uncommon features.

The pipeline identifies have a strict naming convention:

`rwPDS_[pipename]_[pipetype]PipeID`

Where **pipename** specifies the unique pipeline name that details briefly it's rendering features and the **pipetype** specifies the pipelines attachment object.

The **RpPDS** API reference currently contains a full detailed list of the pipeline names, identifiers and their features.

Pipeline registration

Pipelines are registered with the PDS through a uniform interface. The interface defines the pipeline's *type* and it's construction definition structure.

Pipeline type

The pipeline's type is defined by the **RpPDSPipeType** enumeration. Currently it's possible to create object and material pipelines. This maybe extended in the future to support new types of rendering pipelines.

There are currently two pipeline types:

1. **rpPDSMATPIPE**: **RpMaterial** rendering pipelines,

2. **rpPDSOBJPIPE**: Object pipelines.

The object pipelines can be attached to either **RpAtomic** or **RpWorldSector** objects depending on the pipeline's features.

There is an exception here, as immediate mode pipelines are attached to neither atomic nor world sector objects. In this instance the pipelines are just considered *Object* pipelines.

Pipeline register

The pipeline's registration is defined by the **RpPDSRegister** structure. This defines the pipeline's type, identification and construction definition. If the application is registering the pipelines within the **RpPDS** plugin without modification, then a set of registration macros are defined to simplify the process.

The pipeline registration macros have a strict naming convention:

```
rwPDS_[pipename]_[pipetype]PipeRegister()
```

Where **pipename** and **pipetype** specifies the pipeline as before.

The **RpPDS** API reference contains a full detailed list of the pipeline registration macros.

Pipeline construction definition

A pipeline construction is defined by its construction definition structure. Each pipeline has a definition structure that is used by the generic pipeline construction functions to create the pipeline. These are platform specific structures but are typedef'd to the platform independent **RpPDSMatTemplate** and **RpPDSObjTemplate**.

Object pipeline template

The object pipeline template is **RpPDSSkyObjTemplate** this structure defines everything a **PS2All** node object pipeline needs.

Material pipeline template

The material pipeline template is **RpPDSSkyMatTemplate** this structure defines everything a **PS2AllMat** node pipeline needs. The material template also contains a pointer to an external **RpPDSSkyVU1CodeTemplate** construction structure. This defines the vector code array size and vector code array elements. It is disconnected from the base material definition structure to allow the vector code to be customized.

The pipeline definition structures have a strict naming convention:

rwPDS_[**pipename**]_[**pipetype**]Pipe

The material pipeline VU1 code definition structures have a strict naming convention:

rwPDS_[**pipename**]_VU1Code
rwPDS_[**pipename**]_VU1Transforms
rwPDS_[**pipename**]_CodeArraySize

Where **pipename** and **pipetype** specifies the pipeline as before.

The **RpPDS** API reference contains a full detailed list of the pipeline definition structures.

2.2.2 PS2All pipelines

Currently all the pipelines within the PDS are constructed of the PS2All pipelines object and material nodes.

Object pipelines

Any pipeline registered with the type **rpPDSOBJPIPE** will be constructed with the **RpPDSSkyObjPipeCreate** function from its **RpPDSSkyObjTemplate.** This function creates an **RxPipeline** from the PS2All pipeline node (see **RxNodeDefinitionGetPS2All**) and the information in the pipeline definition structure.

Material rendering pipelines

Any pipeline registered with the type **rpPDSMATPIPE** will be constructed with the **RpPDSSkyMatPipeCreate** function from its **RpPDSSkyMatTemplate**. This function creates an **RxPipeline** from the **PS2AllMat** pipeline node (see **RxNodeDefinitionGetPS2AllMatNode**) and the information in the pipeline definition structure.

2.3 Basic use of the PDS plugin

Let us first look at the basic usage of the PDS. Below, we detail the minimum requirements to use the **RpPDS** plugin.

2.3.1 Setting up the plugin

Before any other plugins are attached the **RpPDS** plugin must be registered with **RpPDSPluginAttach**. The **RpPDS** plugin needs to be attached before any other plugin that might want access to a rendering pipeline. This is because during the **RwEngineStart** process, all the registered plugins are opened. Hence any plugin registered after the **RpPDS** plugin will be able to make calls into the PDS to retrieve pipelines.

While attaching the **RpPDS** plugin the application must specify the number of pipelines in order to reserve sufficient space for their registration and creation.

The following code example attaches the **RpPDS** plugin and reserves space for the original **RpWorld** pipelines and a new custom lighting pipeline:

```
/* Attach the RpPDS plugin reserving space for pipelines. */
RpPDSPluginAttach(RpWorldNumPipes + 2);
```

2.3.2 Registering pipelines

Once the **RpPDS** plugin has been attached the application can start registering pipelines. These pipelines can be registered up to the point of starting the engine with **RwEngineStart**. Any pipelines registered before the engine start will be created during the **RpPDS** plugin open in **RwEngineStart**.

Pipelines can still be registered after the engine has started; in this case the pipelines will be created instantly, and are available once the registration function returns.

The function **RpPDSRegisterPipe** should be used to register pipelines. However all the default PDS pipelines have a registration macro to simplify the process.

The following code example registered the original **RpWorld** pipelines and an example new custom lighting pipeline, before calling **RpWorldPluginAttach()**:

```
/* Register custom pipeline. */
rwPDS_G3x_APL_MatPipeRegister();
rwPDS_G3x_Generic_AtmPipeRegister();

/* Register original world pipelines and attach plugin. */
RpWorldPipesAttach();
RpWorldPluginAttach();
```

Switching from 3.3 pipeline functionality to 3.4

If an application was developed with RenderWare Graphics 3.3, the easiest update path to 3.4 is to use the plugin pipeline macros provided. These macros mirror the pipeline-attach functions and return the pipeline functionality that was available previously in 3.3.

For example the following code burst registers the original skin pipeline and the skinning plugin:

```
/* Register original skinning pipelines. */
RpSkinPipesAttach();
/* Register skinning plugin. */
RpSkinPluginAttach();
```

The pipeline attach macros do not cover a full set of the 3.4 rendering pipeline. Nearly all the pipelines have been updated for the latest release and we encourage people to ultimately select and specify the pipelines their application needs.

2.3.3 Retrieving pipelines

Once the engine has been started the pipelines within the PDS are available for rendering. The function **RpPDSGetPipe()** will return the **RxPipeline** created for a registered pipeline.

The following code example retrieves the two custom lighting pipelines, and attaches it to an atomic and it's geometry's materials:

```
RxPipeline *matPipe;
RxPipeline *atmPipe;

/* Retrieve the custom lighting pipeline. */
matPipe = RpPDSGetPipe(rwPDS_G3x_APL_MatPipeID);
atmPipe = RpPDSGetPipe(rwPDS_G3x_Generic_AtmPipeID);

/* Attach the object pipeline to the atomic. */
RpAtomicSetPipeline(<atomic>, atmPipe);

/* Attach the material pipeline to the materials. */
<geometry> = RpAtomicGetGeometry(<atomic>);
for ( i = 0; i < RpGeometryGetNumMaterials(<geometry>); i++ )
{
  <material> = RpGeometryGetMaterial(<geometry>, i);
  RpMaterialSetPipeline(<material>, matPipe);
}
```

2.4 PDS example

The RenderWare Graphics example **RW\Graphics\examples\pds** demonstrates using the **RpPDS** plugin to register and retrieve pipelines.

The PDS example uses the following code in **AttachPlugins()** (**main.c**) to register the default PDS pipelines it uses (this occurs after the RenderWare example skeleton has attached the **RpPDS** plugin):

```
/* Register custom pipelines. */
rwPDS_G3x_APL_MatPipeRegister();
rwPDS_G3x_ADL_MatPipeRegister();
rwPDS_G3x_A4D_MatPipeRegister();
rwPDS_G3x_ADLClone_MatPipeRegister();
rwPDS_G3x_OPLClone_MatPipeRegister();

rwPDS_G3x_Generic_AtmPipeRegister();
rwPDS_G3x_ADLClone_AtmPipeRegister();
rwPDS_G3x_OPLClone_AtmPipeRegister();

/* Register new style world pipelines. */
rwPDS_G3_Generic_AtmPipeRegister();
rwPDS_G3_Generic_MatPipeRegister();
```

The following code in **PS2AllOptimizedPipelinesCreate()** (**userpipe.c**) registers the custom PDS pipelines:

```
RpPDSRegisterMatPipe(&UserOpt1_MatPipe, User1MatID);
RpPDSRegisterMatPipe(&UserOpt2_MatPipe, User2MatID);
RpPDSRegisterObjPipe(&UserOpt1_AtmPipe, User1AtmID);
RpPDSRegisterObjPipe(&UserOpt2_AtmPipe, User2AtmID);
```

These pipelines are retrieved in **ps2AllStateCB()** (**ps2all.c**) with calls to **RpPDSGetPipe()** and set as the default atomic and material pipelines.

The example has menu options to allow the performance (EE core, VU1 or a mixture) of the pipelines to be compared by testing the number of atomics that can be rendered with each pipeline without dropping frames.

2.5 Summary

This chapter has covered the benefits of introducing the **RpPDS** plugin, registering and retrieving pipelines and the PDS example. The section on registering pipelines describes the modifications necessary to make a RenderWare Graphics 3.3 application work with RenderWare Graphics 3.4.