RenderWare Graphics

White Paper

PS2 VCL Pipelines

Contact Us

Criterion Software Ltd.

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

Contributors

RenderWare Graphics development and documentation teams.

Table of Contents

1. Introduction

This document assumes knowledge of VCL and the preprocessors it uses (GCC and GASP). These tools and documentation are available for download from the PS2 DevNet website https://www.ps2-pro.com/projects/vcl. EE-GASP is included in the VCL preprocessing sample.

This document describes the generic lighting VCL micro-programs and the custom lighting VCL micro-programs, which may be a useful starting place for creation of new micro-programs. Generic lighting micro-programs can process any combinations of dynamic lights that will fit in the lighting block in VU1 memory. Custom lighting micro-programs can only process the dynamic lights they are hard-coded to work with. This reduction in flexibility allows the micro-program to be substantially faster.

Generic lighting VCL pipelines (the pipeline IDs are `rwPDS_<Name>_MatPipeID`):

PIPELINE NAME	DESCRIPTION
`VCL_Uva`	Single pass UV transform.
`VCL_DupUva`	Dual pass UV transform.
`VCL_SkinUva`	Skinned single pass UV transform.
`VCL_SkinDupUva`	Skinned dual pass UV transform.

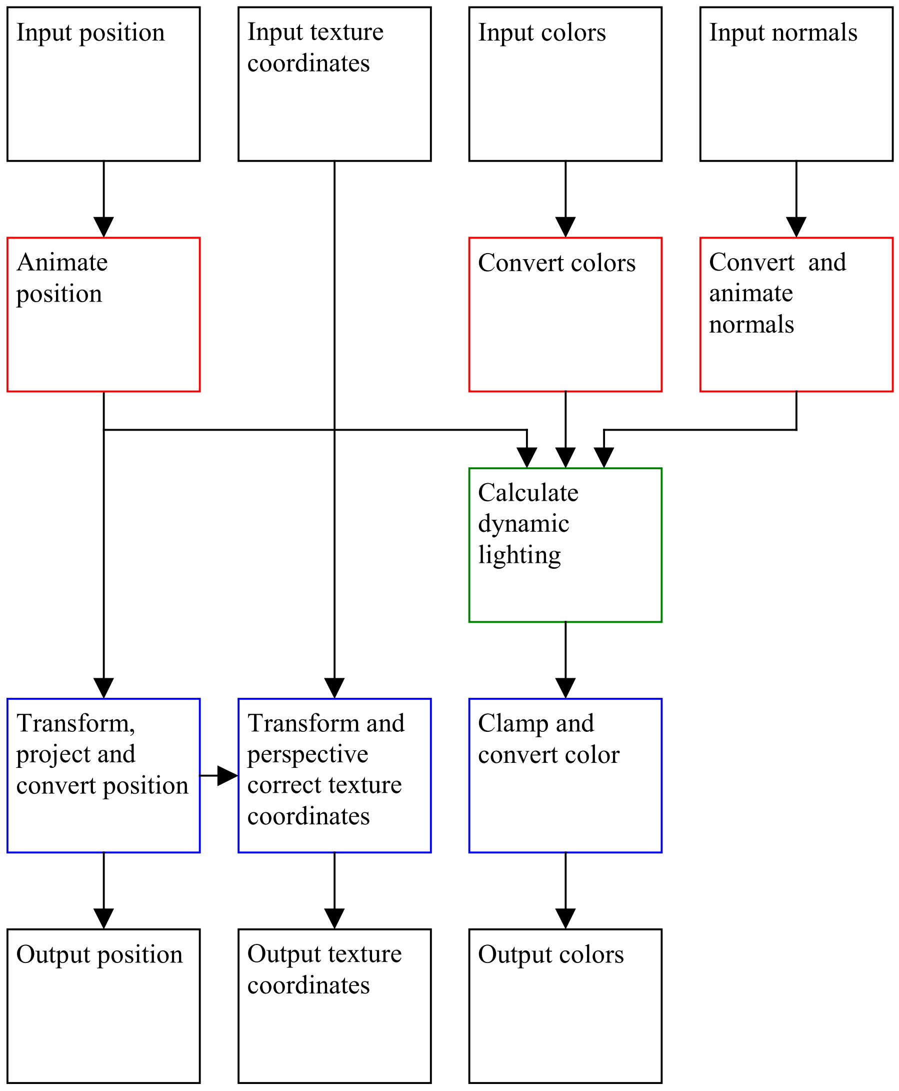
Custom lighting VCL pipelines (the pipeline IDs are `rwPDS_<Name>_MatPipeID`):

PIPELINE NAME	DESCRIPTION
`VCL_ADLDot3`	Directional light dot3 bump mapping.
`VCL_APLDot3`	Point light dot3 bump mapping.
`VCL_ADLSkinDot3`	Skinned directional light dot3 bump mapping.
`VCL_APLSkinDot3`	Skinned point light dot3 bump mapping.
`VCL_ADLSpec`	Ambient + specular directional light rendering.
`VCL_APLSpec`	Ambient + specular point light rendering.
`VCL_ADLSkinSpec`	Skinned ambient + specular directional light rendering.
`VCL_APLSkinSpec`	Skinned ambient + specular point light rendering.

2. Generic Lighting VCL Pipelines

These pipelines use micro-programs that are functionally equivalent to G3 micro-programs except they do not support tri-list clipping.

The generic pipelines consist of three parts as follows: pre-processing (in red on the second row), lighting (in green on the third row) and transforming (in blue on the fourth row).


Input position
Input texture coordinates
Input colors
Input normals
Animate position
Convert colors
Convert and animate normals
Calculate dynamic lighting
Transform, project and convert position
Transform and perspective correct texture coordinates
Clamp and convert color
Output position
Output texture coordinates
Output colors

Pre-processing performs any processing which will affect the lighting (e.g. animation) and converts normals and colors into floating point form.

Lighting consists of calculating the contribution of each dynamic light and adding these to the input color.

The transformation stage projects the positions, perspective corrects the texture coordinates and clamps the colors output by the lighting.

3. Creating New Generic Lighting Pipelines

The easiest way to create and test a new pipeline is to copy the vclgenp example, copy a VCL pipeline directory from the PDS plugin (e.g. rwsdk/plugin/pds/sky2/VCL_Uva) into the src directory of the example and then modify the files within it as described below. Each pipeline directory contains the following files:

FILE	DESCRIPTION
`<pipe>_Node.c`	Defines the pipeline template.
`<pipe>_Data.h`	Defines the VU1 memory layout used by the pipeline.
`<pipe>_Docs.h`	This file is only used to generate the API reference.
`<pipe>_Node.h`	Defines the pipeline ID and registration macro.
`<pipe>PER.dsm`	The micro-program source (includes `<pipe>PER.vsm`).
`<pipe>PER.vcl`	The VCL source file for the micro-program (includes `<pipe>_Data.h`).
`<pipe>PER.vsm`	Generated by VCL from `<pipe>PER.vcl`.

To compile the files in an example instead of the PDS plugin you will need to make the following changes:

1. Change the include inside the .vcl file from `#include “sky2/<pipe>/<pipe>_Data.h”` to `#include “src/<pipe>/<pipe>_Data.h”`.

2. Change the include inside the .dsm file from `.include “sky2/<pipe>/<pipe>PER.vsm”` to `.include “src/<pipe>/<pipe>PER.vsm”`.

3. Remove all of the #include directives from <pipe>_Node.h.

4. Replace all of the #include directives in <pipe>_Node.c with:

```
#include <rwcore.h>
#include <rpworld.h>
#include <rppds.h>

#include "<pipe>_Node.h"
#include "<pipe>_Data.h"
```

5. In <pipe>_Node.c replace the plugin id rwID_PDSPLUGIN with an unused plugin id (the used plugin ids are defined in rwplcore.h). In <pipe>_Node.h replace the pipe id rwPDS_<pipe>_MatPipeID with an unused pipe id (the used pipe ids are defined in rppds.h).

6. Add <pipe> to the VCLPIPES variable and <pipe>/<pipe>PER (and any other micro-programs used by the pipeline) to the VCLPROGS variable in the makefile.

7. Add the following code to AttachPlugins (in main.c) to register the pipe:

   ```
   rwPDS_<pipe>_MatPipeRegister();
   ```

8. Use the pipe by setting the default material pipeline (in this example all materials use the default pipeline):

   ```
   RpMaterialSetDefaultPipeline(RpPDSGetPipe(rwPDS_<pipe>_MatPipeID));
   ```

4. Environment Mapping Generic Lighting Micro-Programs

To modify a dual pass UV transform generic lighting micro-program to generate environment map texture coordinates for the second pass the transform routines must be modified. In this case the same code modifications should be made to each routine.

Add the following code between `*:` and `*Loop:` (this code will be executed once per batch):

```
LQ.xy        EnvMapMatrix0, 0+pipeASymbEnvMapMatrix(VI00)
LQ.xy        EnvMapMatrix1, 1+pipeASymbEnvMapMatrix(VI00)
LQ.xy        EnvMapMatrix2, 2+pipeASymbEnvMapMatrix(VI00)
LQ.xy        EnvMapMatrix3, 3+pipeASymbEnvMapMatrix(VI00)
```

And replace (located after `*Loop:`):

```
; Transform texture coordinates
ADDAx.xyzw   ACC, UVMatrix3, VF00
MADDAx.xy    ACC, UVMatrix1, InputTexCoords
MADDy.xy     TransformedTexCoords1, UVMatrix2, InputTexCoords
MADDAz.zw    ACC, UVMatrix1, InputTexCoords
MADDw.zw     TransformedTexCoordsTemp, UVMatrix2, InputTexCoords
MR32.yz      TransformedTexCoordsTemp, TransformedTexCoordsTemp
MR32.xy      TransformedTexCoords2, TransformedTexCoordsTemp

; Perspective correct and store first pass texture coordinates
MULq.xyz     OutputTexCoords1, TransformedTexCoords1, Q
SQ.xyz       OutputTexCoords1, 0(Output1)

; Perspective correct and store second pass texture coordinates
MULq.xyz     OutputTexCoords2, TransformedTexCoords2, Q
SQ.xyz       OutputTexCoords2, 0(Output2)
```

With:

```
; Transform texture coordinates
ADDAx.xy     ACC, UVMatrix3, VF00
MADDAx.xy    ACC, UVMatrix1, InputTexCoords
MADDy.xy     TransformedTexCoords1, UVMatrix2, InputTexCoords

; Perspective correct and store first pass texture coordinates
MULq.xyz     OutputTexCoords1, TransformedTexCoords1, Q
SQ.xyz       OutputTexCoords1, 0(Output1)

; Load normal
LQ.xyz       NormalInput, 3(Input)

; Generate environment map texture coordinates
ADDAx.xy     ACC, EnvMapMatrix3, VF00
MADDAx.xy    ACC, EnvMapMatrix0, NormalInput
MADDAy.xy    ACC, EnvMapMatrix1, NormalInput
MADDz.xy     TransformedTexCoords2, EnvMapMatrix2, NormalInput

; Perspective correct and store second pass texture coordinates
MULq.xyz     OutputTexCoords2, TransformedTexCoords2, Q
```

```
SQ.xyz      OutputTexCoords2, 0(Output2)
```

The memory layout file (<pipe>_Data.h) needs to have the an extra define for the environment map matrix:

```
#define pipeASymbEnvMapMatrix      (10+pipeASymbMatFXBaseAddress)
```

The pipeline needs to be modified to upload this matrix by replacing the bridge callback. The easiest way of doing this (although it is faster if all the uploads are done in one packet) is to create a bridge callback that uploads the environment map matrix and then calls the dual pass uv transform bridge callback:

```
RwBool
RpPDS_G3_EnvUva_PS2AllMatBridgeCallBack(RxPS2AllPipeData
*ps2AllPipeData)
{
    RwMatrix envMapMatrix;
    RwUInt128 ltmp;
    RwUInt64 tmp1, tmp;

    /* setup environment map matrix */
    RwMatrixInvert(&envMapMatrix, RwFrameGetLTM(RpAtomicGetFrame(
        (RpAtomic *)ps2AllPipeData->sourceObject)));
    RwV3dScale(&envMapMatrix.right, &envMapMatrix.right, 0.5f);
    RwV3dScale(&envMapMatrix.up, &envMapMatrix.up, 0.5f);
    RwV3dScale(&envMapMatrix.at, &envMapMatrix.at, 0.5f);
    envMapMatrix.pos.x = 0.5f;
    envMapMatrix.pos.y = 0.5f;

    /* open packet to upload environment map matrix */
    _rwDMAOpenVIFPkt(0, 3);
    RWDMA_LOCAL_BLOCK_BEGIN();

    /* upload 2qw of environment map matrix */
    tmp = 0x10000002l; /* CNT DMA tag. QWC = 2 */
    tmp1 = (0x01000101l << 0) /* STCYCL VIF code. WL = CL = 1 */
         | ((pipeASymbEnvMapMatrix | 0x64040000l) << 32);
        /* UNPACK V2_32 VIF code. NUM = 4 */
    MAKE128(ltmp, tmp1, tmp);
    RWDMA_ADD_TO_PKT(ltmp);

    /* Environment map matrix */
    tmp = *(RwUInt64 *)&envMapMatrix.right;
    tmp1 = *(RwUInt64 *)&envMapMatrix.up;
    MAKE128(ltmp, tmp1, tmp);
    RWDMA_ADD_TO_PKT(ltmp);
    tmp = *(RwUInt64 *)&envMapMatrix.at;
    tmp1 = *(RwUInt64 *)&envMapMatrix.pos;
    MAKE128(ltmp, tmp1, tmp);
    RWDMA_ADD_TO_PKT(ltmp);

    RWDMA_LOCAL_BLOCK_END();

    return RpPDS_G3_DupUva_PS2AllMatBridgeCallBack(ps2AllPipeData);
}
```

5. Custom Lighting VCL Pipelines

The VU1 micro-programs used by the custom lighting vcl pipelines are generated from rwsdk/plugin/pds/sky2/pds_vcl/vcl_common.vcl by the following steps

1. The gcc pre-processor is used to select instructions according to the included file <microprogram>.h. The pre-processor also substitutes the values defined in the memory layout file vcl_common_data.h.

2. Gasp is used to expand the macros used by the lighting calculations.

3. Sed is used to remove the blank lines created by the first two steps. This doesn't affect the code generated but does make the resulting file <microprogram>.vcl more readable.

4. Vcl is used to compile the file <microprogram>.vcl into the file <microprogram>.vsm ready for inclusion by <microprogram>.dsm.

The custom lighting pipelines perform all calculations in one loop which does minimize the number of instructions needed especially loads and stores but can cause VCL problems to optimize.

6. Creating New Custom Lighting Pipelines

The easiest way to create and test a new pipeline is to copy the vclpipes example, copy a VCL pipeline directory from the PDS plugin (e.g. rwsdk/plugin/pds/sky2/VCL_ADLSpecular) into the src directory of the example and then modify the files within it as described below. Each pipeline directory contains the following files:

FILE	**DESCRIPTION**
`<pipe>_Node.c`	Defines the pipeline template.
`<pipe>_Docs.h`	This file is only used to generate the API reference.
`<pipe>_Node.h`	Defines the pipeline ID and registration macro.
`<microprog>.h`	Defines the input format, output format and processing to be performed by the micro-program
`<microprog>.dsm`	The micro-program source (includes `<microprog>.vsm`).
`<microprog>.vcl`	Generated by the process described in the previous chapter from `VCL_Common.vcl` and `<microprog>.h`.
`<microprog>.vsm`	Generated by VCL from `<microprog>.vcl`.

To compile the files in an example instead of the PDS plugin you will need to make the following changes:

1. Change the include inside the .dsm files from `.include "sky2/<pipe>/<microprog>.vsm"` to `.include "src/<pipe>/<microprog>.vsm"`.

2. Remove all of the #include directives from <pipe>_Node.h.

3. Replace all of the #include directives in <pipe>_Node.c with:

```
#include <rwcore.h>
#include <rpworld.h>
#include <rppds.h>

#include "<pipe>_Node.h"
#include "<pipe>_Data.h"
```

4. In <pipe>_Node.c replace the plugin id rwID_PDSPLUGIN with an unused plugin id (the used plugin ids are defined in rwplcore.h). In <pipe>_Node.h replace the pipe id rwPDS_<pipe>_MatPipeID with an unused pipe id (the used pipe ids are defined in rppds.h).

5. Add <pipe> to the VCLPIPES variable and <pipe>/<microprog> to the VCLPROGS variable in the makefile.

6. Add the following code to AttachPlugins (in main.c) to register the pipe:

```
rwPDS_<pipe>_MatPipeRegister();
```

7. Use the pipe by setting the default material pipeline (in this example all materials use the default pipeline):

```
RpMaterialSetDefaultPipeline(RpPDSGetPipe(rwPDS_<pipe>_MatPipeID));
```

7. Ambient, Two Directional and One Point Light Pipeline

To create a plain rendering pipeline which supports an ambient light, 2 directional lights and 1 point light it is easiest to start from a specular pipeline. In the <microprog>.h files change the NUMDIRECTIONAL, NUMPOINT and SPECULAR defines to:

```
#define NUMDIRECTIONAL      2
#define NUMPOINT            1
#define SPECULAR            0
```

The new pipeline will require a custom upload function adding to <pipe>_Node.c:

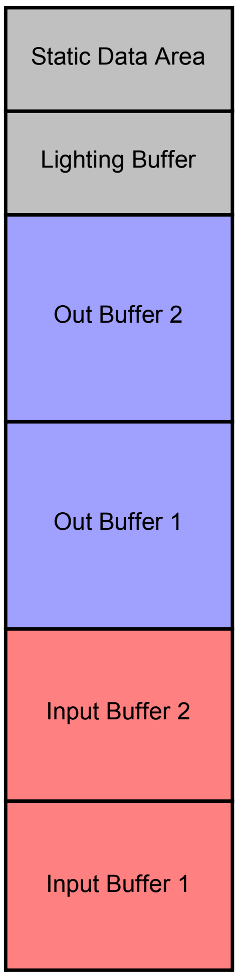
```
#include "utils.h"
#include "libgraph.h"

void
RpPDS_VCL_A2DP_LightingUpload(RpLight *ambient, RpLight
*directional1, RpLight *directional2, RpLight *point)
{
    const RwFrame *frame;

    _rwDMAOpenVIFPkt(0, 8);
    RWDMA_LOCAL_BLOCK_BEGIN();

    /* Unpack light info */
    ADDTOPKT32(0x10000007, 0, /* DMA CNT tag. QWC = 7 */
        SCE_VIF1_SET_STCYCL(1, 1, 0),
        SCE_VIF1_SET_UNPACK(4 + vuSDLightOffset, 7,
VIFCMD_UNPACKV4_32, 0));

    /* Ambient light color */
    ADDTOPKT32(
        *(const RwUInt32 *)&ambient->color.red,
        *(const RwUInt32 *)&ambient->color.green,
        *(const RwUInt32 *)&ambient->color.blue,
        *(const RwUInt32 *)&ambient->color.alpha);

    /* First directional light color */
    ADDTOPKT32(
        *(const RwUInt32 *)&directional1->color.red,
        *(const RwUInt32 *)&directional1->color.green,
        *(const RwUInt32 *)&directional1->color.blue,
        *(const RwUInt32 *)&directional1->color.alpha);

    /* First directional light direction */
    frame = RpLightGetFrame(directional1);
    ADDTOPKT32(
        *(const RwUInt32 *)&frame->ltm.at.x,
        *(const RwUInt32 *)&frame->ltm.at.y,
        *(const RwUInt32 *)&frame->ltm.at.z,
        0);

    /* Second directional light color */
    ADDTOPKT32(
        *(const RwUInt32 *)&directional2->color.red,
        *(const RwUInt32 *)&directional2->color.green,
        *(const RwUInt32 *)&directional2->color.blue,
        *(const RwUInt32 *)&directional2->color.alpha);

    /* Second directional light direction */
    frame = RpLightGetFrame(directional2);
    ADDTOPKT32(
        *(const RwUInt32 *)&frame->ltm.at.x,
        *(const RwUInt32 *)&frame->ltm.at.y,
        *(const RwUInt32 *)&frame->ltm.at.z,
        0);

    /* Point light color */
    ADDTOPKT32(
        *(const RwUInt32 *)&point->color.red,
        *(const RwUInt32 *)&point->color.green,
        *(const RwUInt32 *)&point->color.blue,
        *(const RwUInt32 *)&point->color.alpha);

    /* Point light position and radius */
    frame = RpLightGetFrame(point);
    ADDTOPKT32(
        *(const RwUInt32 *)&frame->ltm.pos.x,
        *(const RwUInt32 *)&frame->ltm.pos.y,
        *(const RwUInt32 *)&frame->ltm.pos.z,
        *(const RwUInt32 *)&point->radius);

    RWDMA_LOCAL_BLOCK_END();
}
```

8. VU1 Memory Layout

The vector code used by RenderWare Graphics operates as follows:

- Geometric input data is divided into 'batches' at instance time
- There are two Input Buffers in VU1 memory to receive these batches
- DMA writes data into one buffer while VU1 is processing the data in the other buffer, thus achieving a high degree of parallelism
- There are (at least) two Output Buffers, into which VU1 writes the post-transform 2D triangles
- The GS reads from one Output Buffer while VU1 is filling the other(s), again for parallelism

In addition to the Input and Output Buffers, VU1 contains a Lighting Buffer (containing data describing the lights to be applied to the current object) and a Static Data Area (containing common and custom parameters guiding VU1 execution). These are shown in the following diagram:



- In this diagram, VU1 memory starts at address zero at the bottom, with the highest available address (0x3FF) at the top.

The Input Buffers

The size and locations of the Input Buffers is defined by the `<pipe>_Data.h` files or the `VCL_Common_Data.h` file. Within the Input Buffer, input vertices (representing tri-lists, tri-strips or line-lists) are in a contiguous array, usually of stride four quadwords, as described in the table below:

ADDRESS	CONTENTS	X	Y	Z	W	Notes
+0	Position	X	Y	Z	-	-
+1	UV	U1	V1	U2	V2	(U2 == U1) and (V2 == V1) for objects with only one set of UVs
+2	RGBA	R	G	B	A	Integer 0:8:0 format
+3	Normal	X	Y	Z	-	Integer 1:0:7 format

["Integer x:y:z" format is a fixed point value with x sign bits, y bits above the fixed point and z bits below the fixed point]

The skinning transforms used by `RpSkin` use vertices of stride 5. Matrix weights and indices fill the fifth quadword in each vertex.

For morphing objects, a second set of positions and normals are uploaded to VU1. These are not interleaved with the per-vertex properties shown in the table above. Rather, the second set of positions follows on from the standard vertices in a contiguous array. After this comes a contiguous array comprised of the second set of normals. The format of the positions and normals is the same as in the above table.

Custom code may use vertices of a different stride. For instance, transforms that do not perform lighting need not upload normals.

Output Buffer

The extents of the Output Buffers are implicitly defined by the extents of the Input Buffers and Clipping Buffer (if present) or Lighting Buffer. Each vertex in the Output Buffers has a stride of three quadwords:

ADDRESS	CONTENTS	X	Y	Z	W	Notes
+0	STQ	S	T	1/Z	-	Z is in camera-space
+1	RGBA	R	G	B	A	Integer 0:8:0 format
+2	XYZF2	X	Y	Z	ADC/fog	Integer 0:12:4 format

Lighting Buffer

The size of the Lighting Buffer (or the Static Data Area, depending upon how much data the vector code interprets as belonging to one or the other) contains an inverse lighting matrix and data describing the lights to be applied to the current object. The matrix is used to transform these lights into object-space. Performing lighting in object-space is efficient because vertex normals need not be transformed. Here is a brief description of the data used for each light type:

- `rpLIGHTAMBIENT` – one quadword, containing color (with the light's type in the W component instead of alpha)
- `rpLIGHTDIRECTIONAL` – one quadword containing color (with the light's type in the W component instead of alpha), followed by another containing the light's direction vector (in world-space)
- `rpLIGHTPOINT` - one quadword containing color (with the light's type in the W component instead of alpha), followed by another containing the light's position (in world-space) with the light's radius in the W field
- `rpLIGHTSPOT` – one quadword containing color (with the light's type in the W component instead of alpha), followed by another containing the light's position (in world-space), with the light's radius in the W field, followed by a final quadword containing the light's direction vector (in world-space), with the spotlight's cone angle in the W field (represented as minus the cosine of the cone angle)
- `rpLIGHTSPOTSOFT` – as for `rpLIGHTSPOT`

The Lighting Buffer is filled from the bottom to the top and is terminated by a zero-valued quadword. If such a quadword directly follows the inverse lighting matrix then no lighting will be performed.

The Static Data Area

The Static Data Area contains a selection of per-object and per-mesh values used by vector code. This includes common values such as the near and far clipping plane distances, the camera matrix, material color, etc. It is possible to enlarge the Static Data Area by lowering the address of the Light Buffer (without putting any extra data into the Light Buffer). Alternatively, you may lower the address of the Clipping Buffer and write extra data in-between it and the Lighting Buffer. There is no real restriction here as long as no overlaps occur.

The default static data are as follows:

ADDRESS	NAME	MEANING
`0x3f0`	`vuSDmat0`	The first quadword of the transformation matrix
`0x3f1`	`vuSDmat1`	The second quadword of the transformation matrix
`0x3f2`	`vuSDmat2`	The third quadword of the transformation matrix
`0x3f3`	`vuSDmat3`	The fourth quadword of the transformation matrix

0x3f6	vuSDxMaxyMax	X and Y hold obsolete frustum sizes. W and Z describe a linear fog equation.
0x3f7	vuSDcamWcamHzScale	Camera frustum width and height and Z-buffer scale.
0x3f8	vuSDoffXoffYzShift	Camera raster offsets into the GS overdraw space and Z-buffer offset.
0x3f9	vuSDrealOffset	Camera raster half height and width.
0x3fa	vuSDgifTag	The `GIFtag` for submitting primitives to the GS
0x3fb	vuSDcolScale	Material color (scaled by (128.0/255.0) when a texture is in use)
0x3fc	vuSDsurfProps	Lighting surface properties (with an optional morphing interpolant in the W field)
0x3fd	vuSDClipvec1	A vector used for clipping
0x3fe	vuSDClipvec2	A vector used for clipping
0x3ff	vuSDVUSwitch	Flags determining transform behavior (fogging, culling, etc)