

PlayStation®Edge Geometry Library Quick Start

Table of Contents

About This Document	3
Purpose	3
Audience and Prerequisites	3
Related Documentation	3
Typographic Conventions	3
1 Getting Started	4
2 Vertex-Only Processing	5
Initializing the SPUs and SPURS	5
Initializing the SPURS Event Flag	5
Initializing the Output Buffer	5
Creating an Edge Geometry Job Application	6
Creating SPURS Jobs to Process the Vertexes	7
Creating a SPURS Job List	8
Creating a SPURS Command List	8
Executing the SPURS Command List	9
Waiting for the SPURS Jobs to Finish	9
3 Pre-segmented Geometry Processing	10
Integrating libedgegeomtool into an Existing Tools Pipeline	10
Initializing the SPUs and SPURS	12
Initializing the SPURS Event Flag	12
Initializing the Output Buffer	13
Creating an Edge Geometry Job Application	14
Creating SPURS Jobs to Process the Vertexes	16
Creating a SPURS Job List	18
Creating a SPURS Command List	18
Executing the SPURS Command List	19
Waiting for the SPURS Jobs to Finish	19

About This Document

Purpose

This quick start guide provides the basics you will need to know to start programming with the geometry component of the Edge library. Use this guide in combination with the Edge overview and the geometry library references listed in the “[Related Documentation](#)” section.)

Audience and Prerequisites

This document was written for PlayStation®3 developers who want to write high-performance applications for the PlayStation®3. It is assumed that such developers have familiarity with the following:

- C and C++
- PlayStation®3 hardware
- SCE standard library functions

Related Documentation

In combination with this guide, the following documents provide complete usage and reference information about the Edge library:

- *PlayStation®Edge Library Overview*
- *PlayStation®Edge Geometry Library Reference*
- *PlayStation®Edge Geometry Library for Offline Tool: Reference*
- *PlayStation®Edge Animation Library Reference*
- *PlayStation®Edge Animation Library for Offline Tool: Reference*
- *PlayStation®Edge Zlib Library Reference*
- *PlayStation®Edge LZMA Library Reference*
- *PlayStation®Edge LZO Library Reference*
- *PlayStation®Edge DXT Library Reference*
- *PlayStation®Edge Post Library Reference*

For details on using COLLADA™, including COLLADA™ FX and COLLADA™ Physics, see the *COLLADA DOM Programming Guide* and the COLLADA™ DOM reference documentation, which are located in the COLLADA_DOM/doc directory of the PS3_COLLADA-141_DOM-201.zip package.

Typographic Conventions

This document uses the following typographic conventions:

Convention	Meaning
fixed-width font	Indicates programming code and literals, such as processing instructions, register names, data types, events, and file names. Also indicates function, structure, and macro names.
blue + underlined text	Indicates a hyperlink (blue displays in color printers or online only).

1 Getting Started

There are two ways to use Edge Geometry. The first requires no tools processing, but can only work on vertexes and can take no triangle relationships into account. The second pre-processes your geometry and partitions it up into segments. With these segments in hand, the runtime can then take triangle relationships into account as well as vertexes.

Chapter 2, “[Vertex-Only Processing](#)”, will describe how to use the processing route that does not require tools pre-processing. A sample of this is provided as the `vertexes-only-sample`.

Chapter 3, “[Pre-segmented Geometry Processing](#)”, will describe how to use the processing route that requires offline tool pre-processing of your geometry data. It will cover everything from integration into the tools to integration into the runtime. A sample of this is provided as the `elephant-sample`.

2 Vertex-Only Processing

Initializing the SPU and SPURS

The first initialization step is to initialize the SPU and SPURS.

Pseudo Code for Initializing SPU and SPURS

```
int ppu_thr_prio;
sys_ppu_thread_t my_ppu_thread_id;
sys_spu_initialize(6, 0);
sys_ppu_thread_get_id(&my_ppu_thread_id);
sys_ppu_thread_get_priority(my_ppu_thread_id, &ppu_thr_prio);
cellSpursInitialize(&mSpurs, 6, 250, ppu_thr_prio, 0);
```

Initializing the SPURS Event Flag

The next initialization step is to create a SPURS event flag so that we can know when the geometry processing jobs are completed.

Pseudo Code for Creating a SPURS Event Flag

```
cellSpursEventFlagInitializeIWL (&mSpurs, &spursEventFlag,
    CELL_SPURS_EVENT_FLAG_CLEAR_AUTO,
    CELL_SPURS_EVENT_FLAG_SPU2PPU);
cellSpursEventFlagAttachLv2EventQueue (&spursEventFlag);

static CellSpursJob256 jobEnd __attribute__((__aligned__(16)));
__builtin_memset(&jobEnd, 0, sizeof(CellSpursJob256));
jobEnd.header.eaBinary = ...
jobEnd.header.sizeBinary = ...
*(uint64_t*)&jobEnd.workArea.userData[0] = &spursEventFlag;
```

Initializing the Output Buffer

The first step is to set up the memory where the processed vertexes will be stored. This is usually accomplished by allocating a 1 megabyte aligned chunk of memory and then mapping it to the RSX™.

Pseudo Code for Allocating and Mapping an Output Buffer

```
outputBuffer = memalign(1024*1024, BUFFER_SIZE);
cellGcmMapMainMemory(outputBuffer, BUFFER_SIZE, &offset);
```

The next step is to choose the output buffering type you want to use. The following output types are supported:

- Double-buffered direct output
- Single-buffered direct output
- Double buffering
- Single buffering
- Ring buffering
- Hybrid buffering

More information about each of the buffering types can be found in the overview and reference documentation (refer to the “[Related Documentation](#)” section in “[About This Document](#)”). For illustrative purposes, this guide will use *single-buffered direct output*.

When using direct output, each job's processed vertex data is written to a specific address. This is done by masking in the appropriate direct output constant into the output buffer information's effective address; the Edge Geom will then interpret the effective address as a location of pre-allocated output buffer instead of a location of an output buffer information structure, EdgeGeomOutputBufferInfo.

Creating an Edge Geometry Job Application

Write an SPU program that is to be executed as a standard SPURS job.

Pseudo Code for a Job Program

```
void cellSpursJobMain2 (CellSpursJobContext2* stInfo,
    CellSpursJob256 *job)
{
    struct __attribute__((aligned(16)))
    {
        void *vertexesA; // 0
        void *pad4; // 1
        void *pad5; // 2
    } inputs;
    cellSpursJobGetPointerList((void**)&inputs, &job->header, stInfo);

    // Generate the EdgeGeomSpuConfigInfo
    EdgeGeomSpuConfigInfo spuInfo;
    spuInfo.flagsAndUniformTableCount = job->workArea.userData[7];
    spuInfo.commandBufferHoleSize = 0;
    spuInfo.inputVertexFormatId = job->workArea.userData[4];
    spuInfo.secondaryInputVertexFormatId = 0;
    spuInfo.outputVertexFormatId = job->workArea.userData[5];
    spuInfo.vertexDeltaFormatId = 0;
    spuInfo.indexesFlavorAndSkinningFlavor = EDGE_GEOM_SKIN_NONE |
        (EDGE_GEOM_INDEXES_COMPRESSED_TRIANGLE_LIST_CCW << 4);
    spuInfo.skinningMatrixFormat = EDGE_GEOM_MATRIX_3x4_ROW_MAJOR;
    spuInfo.numVertexes = job->workArea.userData[6];
    spuInfo.numIndexes = 0;
    spuInfo.indexesOffset = 0;

    EdgeGeomSpuContext ctx;

    edgeGeomInitialize(&ctx, &spuInfo, stInfo->sBuffer, stInfo->dmaTag);
    edgeGeomDecompressVertexes(&ctx, inputs.vertexesA, 0, 0, 0);

    EdgeGeomAllocationInfo info;
    uint32_t outputEa = job->workArea.userData[3];
    uint32_t holeEa = job->workArea.userData[8];
    uint32_t allocSize = edgeGeomCalculateDefaultOutputSize(&ctx, 0);
    if(!edgeGeomAllocateOutputSpace(&ctx, outputEa, allocSize, &info,
        cellSpursGetCurrentSpuId()))
    {
        CellGcmContextData gcmCtx;
        edgeGeomBeginCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
        edgeGeomEndCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
        return;
    }

    edgeGeomCompressVertexes(&ctx);
    EdgeGeomLocation vtx;
    edgeGeomOutputVertexes(&ctx, &info, &vtx);
}
```



```

    CellGcmContextData gcmCtx;
    edgeGeomBeginCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
    edgeGeomEndCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
}

```

Creating SPURS Jobs to Process the Vertexes

You must next determine your SPU input and output vertex formats, the vertex stride of the input and output vertexes, and the number of vertex attributes in the input format. The process which uses this information is best described by the following pseudo code.

Pseudo Code for Creating a Set of Jobs to Process a Vertex Stream

```

uint32_t CreateJobs(void *vertexes, int32_t vertexCount,
    uint32_t inputVertexFormatId, uint32_t outputVertexFormatId,
    uint32_t numAttributes, uint32_t inputStride,
    uint32_t outputStride, uint32_t &outputBufferEa,
    CellSpursJob256 *jobs, CellGcmContextData *ctx)
{
    // Round the vertex count to a multiple of 8 vertexes
    vertexCount = (vertexCount + 7) & -8;
    // Compute the memory cost of a vertex
    uint32_t cost = max(inputStride, outputStride);
    cost += 16 * numAttributes;
    // Find the number of vertexes per segment
    int32_t numVertsPerSegment = AVAILABLE_SPU_SPACE / cost;
    numVertsPerSegment = min(numVertsPerSegment, 0xC000 / cost);
    numVertsPerSegment &= -16;
    // Record the starting output address for later use by LibGCM
    uint32_t startingAddr = outputBufferEa;
    uint32_t vertexesEaA = (uint32_t)vertexes;
    while(vertexCount > 0)
    {
        uint32_t numVerts = min(vertexCount, numVertsPerSegment);
        uint64_t size = (numVerts * inputStride + 15) & -16;
        uint64_t sizeA = min(size, 0x4000);
        uint64_t sizeB = min(size - sizeA, 0x4000);
        uint64_t sizeC = (size >= 0x8000) ? (size - 0x8000) : 0;
        uint32_t vertexesEaB = vertexesEaA + sizeA;
        uint32_t vertexesEaC = vertexesEaB + sizeB;

        // Jump to Self Sync
        if(ctx->current >= ctx->end)
        {
            if((*ctx->callback)(ctx, 1) != CELL_OK) return 0;
        }
        uint32_t holeEa = (uint32_t)ctx->current;
        uint32_t jumpOffset;
        cellGcmAddressToOffset(ctx->current, &jumpOffset);
        cellGcmSetJumpCommandUnsafeInline(ctx, jumpOffset);

        CellSpursJob256 *job = jobs++;
        job->workArea.dmaList[0] = vertexesEaA | (sizeA << 32);
        job->workArea.dmaList[1] = vertexesEaB | (sizeB << 32);
        job->workArea.dmaList[2] = vertexesEaC | (sizeC << 32);
        // high bit specifies that it is to output to a specific address
        job->workArea.userData[3] = outputBufferEa |
EDGE_GEOM_DIRECT_OUTPUT_TO_MAIN_MEMORY;
        job->workArea.userData[4] = inputVertexFormatId;
        job->workArea.userData[5] = outputVertexFormatId;
        job->workArea.userData[6] = numVerts;
    }
}

```



```

    job->workArea.userData[7] = numAttributes - 1;
    job->workArea.userData[8] = holeEa;
    job->header.eaBinary = (uintptr_t)_binary_spu_job_start;
    job->header.sizeBinary
        = CELL_SPURS_GET_SIZE_BINARY(_binary_spu_job_size);
    job->header.sizeDmaList = 3*8;
    job->header.eaHighInput = 0;
    job->header.useInOutBuffer = 1;
    job->header.sizeInOrInOut = 0xC000;
    job->header.sizeStack = 0;
    job->header.sizeScratch = (numVerts * numAttributes + 7) & -8;
    job->header.sizeCacheDmaList = 0;

    // Subtract the number of vertexes in this job from the total left
    vertexCount -= numVerts;
    // Increment the vertexes EA and output address to account for the
    // previous job's vertexes
    vertexesEaA += numVerts*inputStride;
    outputBufferEa += numVerts*outputStride;
}
// Round up to the next multiple of 128 bytes for the next job.
outputBufferEa = (outputBufferEa + 0x7F) & ~0x7F;

return startingAddr;
}

```

The address returned by the function is the effective address of the output vertex data that will be filled in by the SPUs. For example, when rendering the vertexes using `cellGcmSetDrawIndexArray()` or `cellGcmSetDrawArrays()`, the address output by the function should be translated to a RSX™ offset and then used in the associated `cellGcmSetVertexDataArray()` calls.

Creating a SPURS Job List

Before you can execute your jobs that were created in the previous step, you must first set up a SPURS job list.

Pseudo Code for Creating a SPURS Job List

```

static CellSpursJobList jobList;
jobList.eaJobList = (uint64_t)jobs;
jobList.numJobs = jobs - firstJob;
jobList.sizeOfJob = 256;

```

Creating a SPURS Command List

Next you must call your newly formed job list from a SPURS job command list.

Pseudo Code for Creating a SPURS Job Command List

```

static uint64_t command_list[5];
command_list[0] = CELL_SPURS_JOB_COMMAND_JOBLIST(&jobList);
command_list[1] = CELL_SPURS_JOB_COMMAND_SYNC;
command_list[2] = CELL_SPURS_JOB_COMMAND_FLUSH;
command_list[3] = CELL_SPURS_JOB_COMMAND_JOB(&jobEnd);
command_list[4] = CELL_SPURS_JOB_COMMAND_END;

```


Executing the SPURS Command List

After creating the command list, you should execute it.

Pseudo Code for Executing the SPURS Job Command List

```
static CellSpursJobChain jobChain __attribute__((aligned(128)));
static CellSpursJobChainAttribute jobChainAttributes;
static uint8_t prios[8] = {1, 1, 1, 1, 1, 1, 1, 1};
cellSpursJobChainAttributeInitialize(
    &jobChainAttributes, command_list, sizeof(CellSpursJob256),
    16, prios, 6, true, 0, 1, false, sizeof(CellSpursJob256), 6);
cellSpursCreateJobChainWithAttribute(&mSpurs, &jobChain,
    &jobChainAttributes);
cellSpursRunJobChain(&jobChain);
```

Waiting for the SPURS Jobs to Finish

Finally, you must wait for the SPURS jobs to finish so that the RSX™ can use the data output by the jobs.

Pseudo Code for Creating a SPURS Job Command List

```
uint16_t evm = 1;
cellSpursEventFlagWait(&spursEventFlag, &evm, CELL_SPURS_EVENT_FLAG_AND);
cellSpursShutdownJobChain(&jobChain);
cellSpursJoinJobChain(&jobChain);
```


3 Pre-segmented Geometry Processing

Integrating libedgegeomtool into an Existing Tools Pipeline

The tools-side library libedgegeomtool presents two different interfaces: the low-level core functions (found in `libedgegeomtool.h`) and the higher-level abstraction (found in `libedgegeomtool_wrap.h`). The high-level interface of a geometry processing tool is the simpler of the two. The steps for working with this interface are as follows:

- (1) Load studio-internal format.
- (2) Convert from custom geometry format to `EdgeGeomScene`.
- (3) Determine how you would like your geometry data formatted, storing your choices in an `EdgeGeomSegmentFormat` object.
- (4) Process through libedgegeomtool.
- (5) Store runtime structures as opaque blocks in studio-internal binary format.

Step 1 is the responsibility of each developer. Step 5 is also likely to be unique to each developer. Therefore, this chapter will discuss steps 2 through 4 in more detail, with emphasis on the simplest usable example.

First, a brief integrator's synopsis of conversion to `EdgeGeom` structures will be provided.

Explanation of Core libedgegeomtool Structures

The `EdgeGeomScene` structure is the most important input to the high-level libedgegeomtool API. It is intended to map roughly to a single 3D asset that artists would export from their content-generation tool (for example, Max® or Maya®). An `EdgeGeomScene` contains all vertex, triangle list, skinning and blend shape data, stored in full-precision, uncompressed formats.

The `EdgeGeomSegmentFormat` structure is a companion to the `EdgeGeomScene`, which describes how the scene's output data will be formatted as well as some basic data about what operations will be performed on the SPU. Users must specify vertex stream formats for SPU input vertex streams (the compressed data actually written by the Edge tools, which will be processed by the Edge SPU runtime) and the SPU output vertex streams (generated by the Edge SPU runtime and passed to the RSX™ for rendering). In addition, users can optionally specify formats of blend shape vertex delta streams (also processed by the SPUs) and RSX™-only vertex streams that are never sent through the SPUs.

The output of higher-level libedgegeomtool processing is an array of `EdgeGeomSegment` objects, each of which contains all the data required for a single Edge SPU job. The data arrays within each Segment object are opaque; in general, they should not be manipulated directly by subsequent tools code (except to write them to disk).

Converting to EdgeGeomScene

The most straightforward conversion is to walk the vertex list in your custom mesh format and generate a large flat array of floats that corresponds to the attributes per vertex (this will be the Scene's `m_vertexes` array). Note each attribute's start index and attribute ID in the `m_vertexAttributeIndexes` and `m_vertexAttributeIds` arrays, because this will be used later in the flavor description structures to tell the tools where to start reading data when generating vertex streams.

The scene's triangles are stored as a flat triangle list, with three 32-bit indexes per triangle. In addition, users must provide a separate array containing a material ID for each triangle in the scene. This material ID is required so that the Edge partitioner does not create partitions that span more than one material. Because it is never used directly by Edge, it can be anything that uniquely identifies the material within each scene.

If blend shapes are supported, these will need to be generated for whatever subset of attributes are able to animate at runtime. The format is simply:

```
[[vertex0mesh0][vertex1mesh0]...[vertexNmesh0][vertex0mesh1][vertex1mesh1]
...[vertexNmesh1]]
```

Again, these are flat tables of floats, stored in the *m_vertexDeltas* array. **Important:** The data stored in these tables need to be delta values, not the target attribute values. Depending on your studio's pipeline, you may need to perform a subtraction at this point, or you may already have deltas present. It is not necessary to include data for all attributes present in the base geometry; store whatever subset of attributes will be blended in the *m_blendedAttributeIds* and *m_blendedAttributeIndexes* arrays.

Skinning data is stored with space for exactly four influences per vertex. Unused weights should be set to 0.0, and unused matrix indexes must be set to -1. No explicit ordering within each vertex's weights/indexes is required, because all libedgegeomtool routines will walk all entries to accumulate weights and locate present matrix indexes.

Merging Vertexes

Sometimes, input scenes contain duplicated vertexes. This is an unnecessary expense and can actually cause significant slow downs at runtime due to post-transform cache misses, wasted bandwidth, and so forth. A function, *edgeGeomMergeIdenticalVertexes()*, is provided for re-indexing duplicated vertexes. This does not modify the vertex database at all, because such an operation would require re-indexing any dependent triangle lists in the tool's working memory. Rather, any triangle that references a duplicate vertex will have its index overwritten by the first vertex with the same attribute data.

Although such problems should ideally be fixed in the source data, it is highly recommended that identical vertexes be merged in all Edge scenes before further processing.

Building an EdgeGeomSegmentFormat

Once your scene is complete, you need to choose how it should be formatted by the Edge tools. The most important decision is what vertex stream format to use. You must specify formats for your SPU input and output vertex streams; if your scene uses blend shapes, you must also specify a format for the blend shape delta stream. You may also provide a format for an RSX™-only stream for attributes that will be sent directly to the RSX™ (such as texture coordinates or lighting coefficients).

There are two ways to specify vertex formats. The easiest and most efficient way is to select one of Edge's built-in formats, which are listed in *edgegeom_structs.h*. A copy of a pre-built format structure for one of the built-in formats can be retrieved using *edgeGeomGet[Spu,Rsx]VertexFormat()*. SPU input and blend shape delta streams use SPU vertex formats, while SPU output and RSX™-only streams use RSX™ vertex formats.

If you need greater flexibility than provided by the built-in formats, you can build your own custom vertex format structures from scratch. A description of your custom formats will be included in your *EdgeGeomSegment* objects, and the Edge SPU runtime will decode the description and compress/decompress the vertex stream on the fly. Although custom format processing has been highly optimized, it is still more efficient to use built-in vertex formats whenever possible.

In addition to vertex format information, the *EdgeGeomSegmentFormat* structure contains fields for index, skinning, and culling flavors and for the skinning matrix format. The index flavor is used to specify whether the triangle list for each segment should be stored in uncompressed format (16-bits per index) or in a custom highly-compressed format. The skinning flavor is used to specify whether the scene is skinned, and (if so) what type of skinning will be performed (in general, choose the most restrictive skinning type that will provide correct results for your scene). The culling flavor is used to specify which types of per-triangle culling will be performed by your Edge job. The skinning matrix format describes the format of the incoming skinning matrices.

Partitioning a Scene into Segments

After creating your `EdgeGeomScene` object and filling out an `EdgeGeomSegmentFormat` object, you are ready to begin processing. For simplicity, `libedgegeomtool` provides a helper function – `edgeGeomPartitionSceneIntoSegments()` – that performs all end-to-end processing necessary to convert a scene into an array of `EdgeGeomSegments`. This function performs the following operations:

- Partitions the scene into “batches” – collections of triangles that share the same material ID (and could thus all be rendered with a single RSX™ draw call).
- Invokes the Edge partitioner on each batch, splitting it into a collection of SPU-sized workloads.
- Processes the data for each scene into a format that the SPUs can process, according to the segment format specified earlier. This includes running the Edge `kcache` optimizer on each workload’s triangle list, and creating all runtime structures in their final, big-endian format.
- Bundling each workload’s data into an `EdgeGeomSegment` object, along with the appropriate material ID (so that the user can group segments appropriately).

Once this function has been called, the `EdgeGeomScene` and `EdgeGeomSegmentFormat` objects are no longer required and can be deleted.

An Example Application – `geomtoolsample`

The `geomtoolsample` (in the `host-common/src` directory) is a simplified tool that is not flexible, but does show the sequence of operations required to build assets for the EdgeGeom runtime. The sample is far simpler than a real production tool; it assumes that a single hard-coded input file should be represented as a single drawn object in the runtime. It also hard-codes the vertex stream formats and other configuration decisions that would be data-driven in a real production tool. It is a good starting point, however, and should probably be the basis for your studio’s first tool based on `libedgegeomtool`.

When executed, `geomtoolsample` loads a single input file (`elephant.fake`) in an artificially-contrived format meant to represent a typical studio’s scene format. The scene is converted into an `EdgeGeomScene`, is processed as described above, and then is written as a C header file (`elephant.edge.h`) that can be included in various Edge samples.

Initializing the SPUs and SPURS

The first initialization step is to initialize the SPUs and SPURS.

Pseudo Code for Initializing SPUs and SPURS

```
int ppu_thr_prio;
sys_ppu_thread_t my_ppu_thread_id;
sys_spu_initialize(6, 0);
sys_ppu_thread_get_id(&my_ppu_thread_id);
sys_ppu_thread_get_priority(my_ppu_thread_id, &ppu_thr_prio);
cellSpursInitialize(&mSpurs, 6, 250, ppu_thr_prio, 0);
```

Initializing the SPURS Event Flag

The next initialization step is to create a SPURS event flag so that we can know when the geometry processing jobs will be finished.

Pseudo Code for Creating a SPURS Event Flag

```
cellSpursEventFlagInitializeIWL (&mSpurs, &spursEventFlag,
    CELL_SPURS_EVENT_FLAG_CLEAR_AUTO,
    CELL_SPURS_EVENT_FLAG_SPU2PPU);
cellSpursEventFlagAttachLv2EventQueue (&spursEventFlag);
```

```
static CellSpursJob256 jobEnd __attribute__((__aligned__(16)));
__builtin_memset(&jobEnd, 0, sizeof(CellSpursJob256));
jobEnd.header.eaBinary = ...
jobEnd.header.sizeBinary = ... *(uint64_t*)&jobEnd.workArea.userData[0] =
&spursEventFlag;
```

Initializing the Output Buffer

The first step is to set up the memory where the processed vertexes will be stored. This is usually accomplished by allocating a one megabyte aligned chunk of memory and then mapping it to the RSX™.

Pseudo Code for Allocating and Mapping an Output Buffer

```
outputBuffer = memalign(1024*1024, BUFFER_SIZE);
cellGcmMapMainMemory(outputBuffer, BUFFER_SIZE, &offset);
```

Note: Some schemes, such as hybrid buffering, require allocating more than one output buffer.

The next step is to choose your output buffering type. The following output types are the supported:

- Double-buffered direct output
- Single-buffered direct output
- Double buffering
- Single buffering
- Ring buffering
- Hybrid buffering

More information about each of the buffering types can be found in the *Play*

Station®Edge Library Overview and in the reference documentation (refer to the “[Related Documentation](#)” section in “[About This Document](#)”). This guide will provide an example of *hybrid buffering*.

Pseudo Code for Implementing Hybrid Buffer

```
static EdgeGeomOutputBufferInfo info;
memset(&info, 0, sizeof(info));
info.sharedInfo.startEa = (uint32_t)sbuffer;
info.sharedInfo.endEa = (uint32_t)sbuffer + sizeof(sbuffer);
info.sharedInfo.currentEa = info.sharedInfo.startEa;
info.sharedInfo.locationId = CELL_GCM_LOCATION_MAIN;
cellGcmAddressToOffset(sbuffer, &info.sharedInfo.startOffset);
uint32_t labels = (uint32_t)cellGcmGetLabelAddress(64);
for(uint32_t i=0;i<6;++i)
{
    info.ringInfo[i].startEa = (uint32_t)rbuffer + i*sizeof(rbuffer)/6;
    info.ringInfo[i].endEa = info.ringInfo[i].startEa + sizeof(rbuffer)/6;
    info.ringInfo[i].currentEa = info.ringInfo[i].startEa;
    info.ringInfo[i].locationId = CELL_GCM_LOCATION_MAIN;
    info.ringInfo[i].RSXLabelEa = labels + i*16;
    cellGcmAddressToOffset(
        (void*)info.ringInfo[i].startEa,
        &info.ringInfo[i].startOffset);
    (uint32_t*)labels[i*4] = info.ringInfo[i].endEa;
}
```


Creating an Edge Geometry Job Application

Write an SPU program that is to be executed as a standard SPURS job.

Pseudo Code for a Job Program

```
void cellSpursJobMain2(CellSpursJobContext2* stInfo, CellSpursJob256 *job)
{
    struct __attribute__((aligned(16)))
    {
        EdgeGeomVertexStreamDescription *outputStreamDesc; // 0
        void *indexes; // 1
        void *pad1;
        void *skinMatrices; // 3
        void *pad2;
        void *skinIndexesAndWeights; // 5
        void *pad3;
        void *vertexesA; // 7;
        void *pad4;
        void *pad5;
        void *vertexesB; // 10
        void *pad6;
        void *pad7;
        EdgeGeomViewportInfo *viewportInfo; // 13
        EdgeGeomLocalToWorldMatrix *localToWorld; // 14
        EdgeGeomSpuConfigInfo *spuConfigInfo; // 15
        void *fixedOffsetsA; // 16
        void *fixedOffsetsB; // 17
        EdgeGeomVertexStreamDescription *inputStreamDescA; // 18
        EdgeGeomVertexStreamDescription *inputStreamDescB; // 19
    } inputs;
    cellSpursJobGetPointerList((void*)&inputs, &job->header, stInfo);

    // Extract additional values from the userData area.
    uint32_t totalMatrixCount = ((job->workArea.userData[3] >> 32)
        + (job->workArea.userData[4] >> 32)) / 48;
    uint32_t outputBufferInfoEa = job->workArea.userData[20];
    uint32_t holeEa = job->workArea.userData[21];
    uint32_t numBlendShapes = job->workArea.userData[22] >> 32;
    uint32_t blendShapeInfosEa = job->workArea.userData[23]
        & 0xFFFFFFFF;

    EdgeGeomCustomVertexFormatInfo customFormatInfo =
    {
        inputs.inputStreamDescA,
        inputs.inputStreamDescB,
        inputs.outputStreamDesc,
        0,
        0,0,0,0,0
    };

    EdgeGeomSpuContext ctx;

    edgeGeomInitialize(&ctx, inputs.spuConfigInfo, stInfo->sBuffer,
        job->header.sizeScratch << 4, stInfo->ioBuffer,
        job->header.sizeInOrInOut,
        stInfo->dmaTag, inputs.viewportInfo, inputs.localToWorld,
        &customFormatInfo);
    edgeGeomDecompressVertexes(&ctx, inputs.vertexesA,
        inputs.fixedOffsetsA, inputs.vertexesB, inputs.fixedOffsetsB);
    edgeGeomProcessBlendShapes(&ctx, numBlendShapes,
        blendShapeInfosEa);
```



```
    edgeGeomSkinVertexes(&ctx, inputs.skinMatrices,
        totalMatrixCount, inputs.skinIndexesAndWeights);
    edgeGeomDecompressIndexes(&ctx, inputs.indexes);
    uint32_t numVisibleIdxs = edgeGeomCullTriangles(
        &ctx, EDGE_GEOM_CULL_BACKFACES_AND_FRUSTUM);
    if (numVisibleIdxs == 0)
    {
        CellGcmContextData gcmCtx;
        edgeGeomBeginCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
        edgeGeomEndCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
        return;
    }

    EdgeGeomAllocationInfo info;
    uint32_t size =
        edgeGeomCalculateDefaultOutputSize(&ctx, numVisibleIdxs);
    if(!edgeGeomAllocateOutputSpace(&ctx, outputBufferInfoEa, size,
        &info, cellSpursGetCurrentSpuId()))
    {
        CellGcmContextData gcmCtx;
        edgeGeomBeginCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
        edgeGeomEndCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
        return;
    }

    EdgeGeomLocation idx;
    edgeGeomOutputIndexes(&ctx, numVisibleIdxs, &info, &idx);
    edgeGeomCompressVertexes(&ctx);
    EdgeGeomLocation vtx;
    edgeGeomOutputVertexes(&ctx, &info, &vtx);

    CellGcmContextData gcmCtx;
    edgeGeomBeginCommandBufferHole(&ctx, &gcmCtx, holeEa, &info, 1);

    edgeGeomSetVertexDataArrays(&ctx, &gcmCtx, &vtx);
    cellGcmSetDrawIndexArrayUnsafeInline(&gcmCtx,
        CELL_GCM_PRIMITIVE_TRIANGLES,
        numVisibleIdxs, CELL_GCM_DRAW_INDEX_ARRAY_TYPE_16,
        idx.location, idx.offset);
    }
    edgeGeomEndCommandBufferHole(&ctx, &gcmCtx, holeEa, &info, 1);
}
```


Creating SPURS Jobs to Process the Vertexes

Next determine your input and output flavors, the vertex stride of the input and output flavors, and the number of attributes in the input flavor. The process that uses this information is best described by the following pseudo code.

Pseudo Code for Creating a Set of Jobs to Process a Vertex Stream

```
static void CreateJob(CellGcmContextData *ctx,
    EdgeGeomPpuConfigInfo *info,
    uint32_t matricesEa, CellSpursJob256 &*jobs)
{
    CellSpursJob256 * job = jobs++;

    uint32_t outputStreamDesc = (uint32_t)info->spuOutputStreamDesc;
    uint64_t outputStreamDescSize = (uint64_t)info->spuOutputStreamDescSize;
    job->workArea.dmaList[0] = outputStreamDesc | (outputStreamDescSize << 32);

    // Indexes
    uint32_t indexesA = (uint32_t)info->indexes;
    uint64_t indexesSizeA = (uint64_t)info->indexesSizes[0];
    uint64_t indexesSizeB = (uint64_t)info->indexesSizes[1];
    uint32_t indexesB = indexesA + indexesSizeA;
    job->workArea.dmaList[1] = indexesA | (indexesSizeA << 32);
    job->workArea.dmaList[2] = indexesB | (indexesSizeB << 32);

    // Skinning Matrices
    uint64_t matricesSizeA = info->skinMatricesSizes[0];
    uint32_t matricesA = matricesEa + info->skinMatricesByteOffsets[0];
    uint64_t matricesSizeB = info->skinMatricesSizes[1];
    uint32_t matricesB = matricesEa + info->skinMatricesByteOffsets[1];
    job->workArea.dmaList[3] = matricesA | (matricesSizeA << 32);
    job->workArea.dmaList[4] = matricesB | (matricesSizeB << 32);

    // Skinning Indexes & Weights
    uint64_t iAndWSizeA = info->skinIndexesAndWeightsSizes[0];
    uint32_t iAndWA = (uint32_t)info->skinIndexesAndWeights;
    uint64_t iAndWSizeB = info->skinIndexesAndWeightsSizes[1];
    uint32_t iAndWB = iAndWA + iAndWSizeA;
    job->workArea.dmaList[5] = iAndWA | (iAndWSizeA << 32);
    job->workArea.dmaList[6] = iAndWB | (iAndWSizeB << 32);

    // Vertexes
    uint32_t vertexes1EaA = (uint32_t)info->spuVertexes[0];
    uint64_t vertexes1SizeA = info->spuVertexesSizes[0];
    uint64_t vertexes1SizeB = info->spuVertexesSizes[1];
    uint32_t vertexes1EaB = vertexes1EaA + vertexes1SizeA;
    uint64_t vertexes1SizeC = info->spuVertexesSizes[2];
    uint32_t vertexes1EaC = vertexes1EaB + vertexes1SizeB;
    job->workArea.dmaList[7] = vertexes1EaA | (vertexes1SizeA << 32);
    job->workArea.dmaList[8] = vertexes1EaB | (vertexes1SizeB << 32);
    job->workArea.dmaList[9] = vertexes1EaC | (vertexes1SizeC << 32);
    uint32_t vertexes2EaA = (uint32_t)info->spuVertexes[1];
    uint64_t vertexes2SizeA = info->spuVertexesSizes[3];
    uint64_t vertexes2SizeB = info->spuVertexesSizes[4];
    uint32_t vertexes2EaB = vertexes2EaA + vertexes2SizeA;
    uint64_t vertexes2SizeC = info->spuVertexesSizes[5];
    uint32_t vertexes2EaC = vertexes2EaB + vertexes2SizeB;
    job->workArea.dmaList[10] = vertexes2EaA | (vertexes2SizeA << 32);
    job->workArea.dmaList[11] = vertexes2EaB | (vertexes2SizeB << 32);
    job->workArea.dmaList[12] = vertexes2EaC | (vertexes2SizeC << 32);
}
```

```

// Triangle Culling Data
job->workArea.dmaList[13] = (uint32_t)&viewportInfo |
    ((uint64_t)sizeof(EdgeGeomViewportInfo) << 32);
job->workArea.dmaList[14] = (uint32_t)&localToWorldMatrix |
    ((uint64_t)sizeof(EdgeGeomLocalToWorldMatrix) << 32);

// SpuConfigInfo
job->workArea.dmaList[15] = (uint32_t)info |
    ((uint64_t)sizeof(EdgeGeomSpuConfigInfo) << 32);

// Software Fixed Point Format Integer Offsets
uint32_t fixedOffsets1 = (uint32_t)info->fixedOffsets[0];
uint64_t fixedOffsets1Size = (uint64_t)info->fixedOffsetsSize[0];
job->workArea.dmaList[16] = fixedOffsets1 | (fixedOffsets1Size << 32);
uint32_t fixedOffsets2 = (uint32_t)info->fixedOffsets[1];
uint64_t fixedOffsets2Size = (uint64_t)info->fixedOffsetsSize[1];
job->workArea.dmaList[17] = fixedOffsets2 | (fixedOffsets2Size << 32);

uint32_t inputStreamDescA = (uint32_t)info->spuInputStreamDescs[0];
uint64_t inputStreamDescSizeA =
    (uint64_t)info->spuInputStreamDescSizes[0];
job->workArea.dmaList[18] = inputStreamDescA | (inputStreamDescSizeA <<
32);
uint32_t inputStreamDescB = (uint32_t)info->spuInputStreamDescs[1];
uint64_t inputStreamDescSizeB = (uint64_t)info->spuInputStreamDescSizes[1];
job->workArea.dmaList[19] = inputStreamDescB | (inputStreamDescSizeB<<32);

// --- Dma Data Ends / User Data Begins---

// Output Buffer Info
job->workArea.userData[20] = (uint32_t)&outputBufferInfo;

// Command Buffer Hole
uint32_t holeSize = info->spuConfigInfo.commandBufferHoleSize << 4;
if(ctx->current + holeSize/4 + 3 > ctx->end)
{
    if((*ctx->callback)(ctx, holeSize/4 + 3) != CELL_OK)
        return;
}
while(((uint32_t)ctx->current & 0xF) != 0)
    *ctx->current++ = 0;
uint32_t holeEa = (uint32_t)ctx->current;
uint32_t holeEnd = holeEa + holeSize;
uint32_t jumpOffset;
cellGcmAddressToOffset(ctx->current, &jumpOffset);
cellGcmSetJumpCommandUnsafeInline(ctx, jumpOffset);
uint32_t nextJ2S = ((uint32_t)ctx->current + 0x80) & ~0x7F;
while(nextJ2S < holeEnd)
{
    ctx->current = (uint32_t*)nextJ2S;
    cellGcmAddressToOffset(ctx->current, &jumpOffset);
    cellGcmSetJumpCommandUnsafeInline(ctx, jumpOffset);
    nextJ2S = ((uint32_t)ctx->current + 0x80) & ~0x7F;
}
ctx->current = (uint32_t*)holeEnd;
job->workArea.userData[21] = holeEa;

// Blend shapes / shape count
if(info->blendShapes)
{

```



```

        job->workArea.userData[22] = (uint32_t)&shapeInfos[numShapeInfos];
        uint64_t numShapes = 0;
        for(int32_t i = 0; i < info->numBlendShapes; ++i)
        {
            if(!info->blendShapes[i] || !shapeAlphas[i])
                continue;
            uint64_t tag = (uint64_t)info->blendShapeSizes[i] << 32;
            shapeInfos[numShapeInfos].dmaTag = info->blendShapes[i] | tag;
            shapeInfos[numShapeInfos].alpha = shapeAlphas[i];
            shapeInfos[numShapeInfos].fixedOffsetsSize =
                info->shapeFixedOffsetsSize;
            numShapeInfos++;
            numShapes++;
        }
        job->workArea.userData[22] |= (numShapes << 32);
    }

    job->header.eaBinary = ...
    job->header.sizeBinary = ...
    job->header.sizeDmaList = 20*8;
    job->header.eaHighInput = 0;
    job->header.useInOutBuffer = 1;
    job->header.sizeInOrInOut = info->ioBufferSize;
    job->header.sizeStack = 0;
    job->header.sizeScratch = info->scratchSize;
    job->header.sizeCacheDmaList = 0;
}

```

In a typical game engine, this function would be called in place of a `cellGcmSetDrawIndexArray()` command.

Creating a SPURS Job List

Before you can execute your jobs that were created in the previous step, you must first set up a SPURS job list.

Pseudo Code for Creating a SPURS Job List

```

static CellSpursJobList jobList;
jobList.eaJobList = (uint64_t)jobs;
jobList.numJobs = jobs - firstJob;
jobList.sizeOfJob = 256;

```

Creating a SPURS Command List

Next you must call your newly formed job list from a SPURS job command list.

Pseudo Code for Creating a SPURS Job Command List

```

static uint64_t command_list[5];
command_list[0] = CELL_SPURS_JOB_COMMAND_JOBLIST(&jobList);
command_list[1] = CELL_SPURS_JOB_COMMAND_SYNC;
command_list[2] = CELL_SPURS_JOB_COMMAND_FLUSH;
command_list[3] = CELL_SPURS_JOB_COMMAND_JOB(&jobEnd);
command_list[4] = CELL_SPURS_JOB_COMMAND_END;

```


Executing the SPURS Command List

After creating the command list, you should execute it.

Pseudo Code for Executing the SPURS Job Command List

```
static CellSpursJobChain jobChain __attribute__((aligned(128)));
static CellSpursJobChainAttribute jobChainAttributes;
static uint8_t prios[8] = {1, 1, 1, 1, 1, 1, 1, 1};
cellSpursJobChainAttributeInitialize(
    &jobChainAttributes, command_list, sizeof(CellSpursJob256),
    16, prios, 6, true, 0, 1, false, sizeof(CellSpursJob256), 6);
cellSpursCreateJobChainWithAttribute(&mSpurs, &jobChain,
    &jobChainAttributes);
cellSpursRunJobChain(&jobChain);
```

Waiting for the SPURS Jobs to Finish

Finally, you must wait for the SPURS jobs to finish so that the RSX™ can use the data output by the jobs.

Pseudo Code for Creating a SPURS Job Command List

```
uint16_t evm = 1;
cellSpursEventFlagWait(&spursEventFlag, &evm, CELL_SPURS_EVENT_FLAG_AND);
cellSpursShutdownJobChain(&jobChain);
cellSpursJoinJobChain(&jobChain);
```