

PlayStation®Edge Library Overview

Table of Contents

| | |
|---|-----------|
| About This Document | 5 |
| Purpose | 5 |
| Audience and Prerequisites | 5 |
| Related Documentation and Other Resources | 5 |
| Typographic Conventions..... | 6 |
| 1 Library Overview | 7 |
| Characteristics..... | 7 |
| Licensing | 7 |
| Files..... | 7 |
| 2 Overview of Edge Geometry | 9 |
| Types of Geometry Processing | 9 |
| Basic Types of Data Stream..... | 9 |
| 3 Using Edge Geometry..... | 11 |
| Offline-Tool Processing | 11 |
| Runtime PPU Processing..... | 14 |
| Runtime SPU Processing..... | 14 |
| 4 Details About Edge Geometry Runtime | 17 |
| Geometry SPU-only Fixed-Point Compression..... | 17 |
| Geometry SPU-only Unit Vector Compression | 17 |
| Geometry SPU-only Indexes Compression | 18 |
| Command Buffer Hole..... | 20 |
| SPU Input DMA List | 20 |
| SPU Input User Data..... | 21 |
| SPU DMA Tag Performance Considerations | 21 |
| SPU Overall Buffer Layout..... | 21 |
| SPU Job Location Storage Management..... | 22 |
| User Functionality and Data..... | 23 |
| Runtime Output Buffer Scheme | 25 |
| Cost Estimate for Edge Geometry SPU Processing..... | 27 |
| 5 Edge Offline Geometry Tool | 28 |
| Building..... | 28 |
| Usage | 28 |
| 6 Overview of Edge Animation..... | 29 |
| Edge Animation Design | 29 |
| SPU Usage..... | 29 |
| Reference Implementation | 29 |
| Win32 Implementation | 29 |
| Animation Processing | 29 |
| Tools Processing | 30 |
| Data Structures..... | 30 |
| 7 Using Edge Animation | 32 |
| PPU – Global..... | 32 |
| SPU – For Each Job | 32 |

| | |
|---|-----------|
| 8 Details About Edge Animation Runtime | 34 |
| Pose Stack | 34 |
| Blend Tree Processing | 34 |
| Command List | 35 |
| Blending Operations and Partial Animation Logic | 35 |
| Example: Simple Blend Tree | 38 |
| Callbacks | 39 |
| Mirroring | 39 |
| Precomputed Poses | 40 |
| Animation Encoding and Compression | 41 |
| 9 Edge Animation Tools | 44 |
| Introduction | 44 |
| Usage | 44 |
| Processing Overview | 44 |
| Skeleton Processing | 45 |
| Animation Processing | 45 |
| Additive Animation Processing | 46 |
| Compression | 46 |
| 10 Using Edge Animation and Edge Geometry Together | 47 |
| Skinning | 47 |
| 11 Overview of Edge Zlib, LZMA, and LZO | 48 |
| Algorithms | 48 |
| Some Suggested Use-Cases | 48 |
| 12 Using Edge Zlib, LZMA, and LZO | 49 |
| Implementation Details | 49 |
| Procedure for Using Edge Zlib from the PPU | 50 |
| Procedure for Using Edge Zlib from the SPU | 51 |
| Common Details | 51 |
| 13 The Specifics of Edge Zlib | 54 |
| Characteristics | 54 |
| Differences Between Edge Zlib and zlib | 54 |
| Licensing | 54 |
| Headers on the DEFLATE Data | 54 |
| Segmented Versus Non-segmented Decompression | 54 |
| Building Edge Zlib | 55 |
| 14 The Specifics of Edge LZMA | 56 |
| Characteristics | 56 |
| Differences Between Edge LZMA and LZMA | 56 |
| Headers on the Compressed Data | 56 |
| The Inflate Tasks | 56 |
| 15 The Specifics of Edge LZO | 57 |
| Characteristics | 57 |
| Differences Between Edge LZO and LZO | 57 |
| Licensing | 57 |
| Headers on the Compressed Data | 57 |
| The Inflate and Deflate Tasks | 57 |

| | |
|---|-----------|
| Compression Output Buffers in Local Store | 58 |
| 16 Overview of Edge DXT | 59 |
| Introduction..... | 59 |
| Edge DXT Design..... | 59 |
| Compression Performance and Restrictions | 59 |
| 17 Using Edge DXT..... | 60 |
| Compression | 60 |
| Decompression | 60 |
| 18 Overview of Edge Post..... | 61 |
| Characteristics..... | 61 |
| SPU Usage..... | 61 |
| Data Structures..... | 61 |
| 19 Using Edge Post | 63 |
| Basic Procedure | 63 |
| 20 Edge Post Runtime..... | 64 |
| Processing Stage and the Effect Chain..... | 64 |
| Tile Size Selection | 64 |
| SPU Memory Layout | 65 |
| SPU Synchronization | 65 |
| RSX™ Synchronization..... | 65 |
| PPU Synchronization | 66 |
| SPU Pixel-Processing Functionality Provided Within the Library | 66 |
| Writing Effect Code | 67 |
| Supported Pixel Formats..... | 68 |
| Edge Post Integration and Suggested Usage..... | 69 |
| Differences Between RSX™ and SPU Implemented Post-processing Effects..... | 70 |
| Performance..... | 71 |
| 21 Edge Post Notes | 73 |
| Example Data Flow for Edge Post | 73 |
| 22 Edge Post MLAA..... | 76 |
| Overview | 76 |
| Basic Procedure | 76 |
| Placing MLAA in a Frame..... | 76 |
| Edge Detection and Tuning..... | 76 |
| Performance Characteristics..... | 77 |
| Samples | 77 |
| Restrictions..... | 77 |
| Optional Features..... | 78 |
| Making Changes to the SPURS Task | 78 |
| 23 Sample Programs | 79 |
| Edge Geometry | 79 |
| Edge Animation | 79 |
| Edge Geometry And Animation | 79 |
| Edge Zlib/LZMA/LZO | 80 |
| Edge DXT..... | 83 |
| Edge Post..... | 83 |

About This Document

Purpose

This document provides an overview of the Edge library. This library utilizes the unique PlayStation®3 architecture to achieve increased performance over a traditional PPU/RSX™ architecture.

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 and Other Resources

Edge Library

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

- *PlayStation®Edge Geometry Library: Quick Start*
- *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*

SPURS and libgcm

SPUs in Edge can be driven using SPURS. For an overview of SPURS and libgcm (a graphics command management library), see the following documents, which can be obtained from the PlayStation®3 SDK Documentation package on the PlayStation®3 Developer Network website (<https://ps3.scedev.net>):

- *libspurs Overview*
- *libgcm Overview*

The RSX™ Cache Optimizer

For information about the RSX™ Cache Optimizer, see *An Improved Vertex Caching Scheme for 3D Mesh Rendering* (<http://www.ecse.rpi.edu/~lin/K-Cache-Reorder/Lin-Yu-TVCG.pdf>).

SPA and EdgePostFilterGen Tools

The *SPU Pipelining Assembler (SPA) User's Guide*, included with the Edge package, describes the SPA, an assembly optimization tool.

The *EdgePostFilterGen User's Guide*, also included with the Edge package, describes EdgePostFilterGen, a command-line utility for generating SPA-optimized loops for simple image kernel operations (such as

Gaussian image filters). These loops can be used in Edge Post. The output is in the form of an SPA file ready to be fed to the SPA tool.

Typographic Conventions

This document uses the following typographic conventions:

| Convention | Meaning |
|--|--|
| <code>fixed-width font</code> | 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 Library Overview

Characteristics

Edge is a library that can help you achieve increased performance over a traditional PPU/RSX™ architecture. This is possible because Edge utilizes the unique PlayStation®3 architecture through the efficient use of the SPU. In comparison to the PPU/RSX™ architecture, the SPU performs tasks more quickly and uses better data compression schemes. The SPU can also perform tasks that ordinarily would be performed by the PPU.

The library provides the following utilities:

- **Edge Geometry:** The SPU performs skinning, coordinate transformation, and culling.
- **Edge Animation:** The SPU performs animation joint tree blending and transformation.
- **Edge Zlib:** The SPU performs lossless data decompression and compression.
- **Edge LZMA:** The SPU performs lossless data decompression.
- **Edge LZO:** The SPU performs lossless data decompression and compression.
- **Edge DXT:** The SPU performs image data compression and decompression.
- **Edge Post:** The SPU performs image post-processing.

All SPU code in Edge can be driven using either SPURS job, SPURS job queue, or SPURS task, but you can use any other SPU management system.

Licensing

The PlayStation®Edge library is distributed under the terms and conditions described in the following file:
cell\license\others\PlayStation_Edge_Terms_and_Conditions_e.txt.

Files

The files required to use Edge are shown in Table 1.

Table 1 Edge Files

| File Name | Description |
|--|-------------------------------------|
| common\include\edge\geom\edgegeom_structs.h | Edge Geometry common header files |
| common\include\edge\geom\edgegeom_attributes.h | |
| spu\include\edge\geom\edgegeom.h | Edge Geometry header files for SPU |
| spu\include\edge\geom\edgegeom_compress.h | |
| spu\include\edge\geom\edgegeom_decompress.h | |
| spu\src\edge\geom*.cpp | Edge Geometry source files for SPU |
| common\include\edge\anim\edgeanim_common.h | Edge Animation common header files |
| common\include\edge\anim\edgeanim_macros.h | |
| common\include\edge\anim\edgeanim_structs.h | |
| ppu\include\edge\anim\edgeanim_ppu.h | Edge Animation header file for PPU |
| spu\include\edge\anim\edgeanim_spu.h | Edge Animation header file for SPU |
| ppu\src\edge\anim*.cpp | Edge Animation source files for PPU |

| File Name | Description |
|---|---|
| spu\src\edge\anim*.cpp | Edge Animation source files for SPUs |
| spu\src\edge\anim*.s | |
| common\include\edge\edge_stdbool.h | Edge common header file |
| common\include\edge\edge_stdint.h | |
| ppu\include\edge\zlib\edgezlib_ppu.h | Edge Zlib PPU header file for library interface |
| spu\include\edge\zlib\edgezlib_spu.h | Edge Zlib SPU header file for library interface |
| common\include\edge\zlib\edgezlib_inflate_queue_element.h | Edge Zlib internal header file for PPU-to-SPU communication |
| common\include\edge\zlib\edgezlib_deflate_queue_element.h | |
| ppu\src\edge\zlib*.cpp | Edge Zlib PPU source files for the library |
| spu\src\edge\zlib*.cpp | Edge Zlib SPU source files for the library |
| spu\src\edge\zlib*.s | |
| spu\src\edge\zlib*.h | Edge Zlib SPU internal header files for the library |
| spu\src\edge\zlib-inflate-take*.cpp | Edge Zlib SPU source files for the Inflate Task |
| ppu\include\edge\lzma\edgelzma_ppu.h | Edge LZMA PPU header file for library interface |
| spu\include\edge\lzma\edgelzma_spu.h | Edge LZMA SPU header file for library interface |
| common\include\edge\lzma\edgelzma_inflate_queue_element.h | Edge LZMA internal header file for PPU-to-SPU communication |
| ppu\src\edge\lzma*.cpp | Edge LZMA PPU source files for the library |
| spu\src\edge\lzma*.cpp | Edge LZMA SPU source files for the library |
| spu\src\edge\lzma*.s | |
| spu\src\edge\lzma*.h | Edge LZMA SPU internal header files for the library |
| spu\src\edge\lzma-inflate-take*.cpp | Edge LZMA SPU source files for the Inflate Task |
| spu\src\edge\lzma-deflate-take*.cpp | Edge LZMA SPU source files for the Deflate Task |
| ppu\include\edge\lzo\edgelzo_ppu.h | Edge LZO PPU header file for library interface |
| spu\include\edge\lzo\edgelzo_spu.h | Edge LZO SPU header file for library interface |
| common\include\edge\lzo\edgelzo1x_inflate_queue_element.h | Edge LZO internal header file for PPU to SPU communication |
| common\include\edge\lzo\edgelzo1x_deflate_queue_element.h | |
| ppu\src\edge\lzo*.cpp | Edge LZO PPU source files for the library |
| spu\src\edge\lzo*.cpp | Edge LZO SPU source files for the library |
| spu\src\edge\lzo*.s | |
| spu\src\edge\lzo*.h | Edge LZO SPU internal header files for the library |
| spu\src\edge\lzo1x-inflate-take*.cpp | Edge LZO SPU source files for the LO1X Inflate Task |
| spu\include\edge\dxt\edgedxt.h | Edge DXT header file for SPU |
| spu\src\edge\dxt*.s | Edge DXT source files for SPU |
| common\include\edge\post* | Edge Post common header files |
| ppu\include\edge\post* | Edge Post header files for PPU |
| ppu\src\edge\post* | Edge Post source files for PPU |
| spu\include\edge\post* | Edge Post header files for SPUs |
| spu\src\edge\post* | Edge Post source files for SPUs |

External library dependencies:

- The Edge Animation SPU library is dependent on libdma.
- The Edge Zlib, Edge LZMA and Edge LZO PPU libraries are dependent on libspurs and libsync.
- The Edge Zlib, Edge LZMA and Edge LZO SPU libraries are dependent on libspurs, libsync, libatomic, and libdma.
- The Edge Post SPU library is dependent on libdma and libatomic.

2 Overview of Edge Geometry

Types of Geometry Processing

General operations that can be performed with Edge Geometry are described below.

Skinning and Other World-Space Conversion

Using up to four matrices per vertex, a variety of skinned vertex attributes can be computed: positions, normals, tangents, and bi-normals. Skinning can be performed for a variety of reasons. For example:

- To offload the work from the RSX™. All attributes are skinned and output in this case.
- To cull unneeded triangles (see “[Triangle Culling](#)” below). Only vertex positions need to be skinned in this case.

Triangle Culling

Vertex positions are used to generate a new index table that omits any triangles that will not pass the RSX™ setup stage:

- Frustum Culled Triangles
- Back-Facing Triangles
- Zero Area Triangles
- No Pixel Triangles

In Edge Geometry processing, objects are processed in units of “geometry segments”, each of which typically contains up to 6,000 triangles of a single material.

The triangles are ultimately displayed as “indexed triangle lists”, where each triangle is described by the indexes of its three vertexes.

Basic Types of Data Stream

Vertex Input/Output Data

The vertex data is a set of vertexes in RSX™-compatible or SPU-only attribute formats. There is no limit to the types of vertex attribute input/output (IO) data that can be introduced into the system because users can always set their own callback function to process their custom vertex data streams.

Some common input and output data types that are supported by the library are shown in Table 2 and Table 3.

Table 2 SPU Vertex Attribute Types Supported by Default

| Type | Per-component Bit Depth | Supported Number of Components |
|------------------------------------|-------------------------|--------------------------------|
| I16N | 16 | 1 to 4 |
| F32 | 32 | 1 to 4 |
| F16 | 16 | 1 to 4 |
| U8N | 8 | 1 to 4 |
| I16 | 16 | 4 |
| X11Y11Z10N ¹ | 32 | 3 |
| U8 | 8 | 4 |
| SPU-only Unit Vectors ² | 24 | 4 |
| SPU-only Fixed Point ³ | 8 | 1 to 4 |

Table 3 RSX™ Vertex Attribute Types Supported by Default

| Type | Per-component Bit Depth | Supported Number of Components |
|-------------------------|-------------------------|--------------------------------|
| I16N | 16 | 1 to 4 |
| F32 | 32 | 1 to 4 |
| F16 | 16 | 1 to 4 |
| U8N | 8 | 1 to 4 |
| I16 | 16 | 4 |
| X11Y11Z10N ¹ | 32 | 3 |
| U8 | 8 | 4 |

¹ The X11Y11Z10N format can only have three components and is always 32 bits. The X component is always 11 bits, the Y component 11 bits, and the Z component 10 bits. Each component is interpreted as a normalized value ranging from -1.0 to 1.0.

² The SPU-only Unit Vector format always has 10 bits per component for the smallest two components in the unit vector; the w component is always a single bit representing 1 or -1.

³ The SPU-only Fixed-Point format can have a maximum resolution of 31 bits per component, and the combined bit depth of all the components must be a multiple of 8.

Details about compressed SPU-only attribute formats can be found later in this document.

For efficiency of implementation, the vertex data input and output formats are specifically defined by the application in `EdgeGeomSpuVertexFormat` and `EdgeGeomRsxVertexFormat` objects. Further information about these structures can be found in the *PlayStation®Edge Geometry Library Reference*.

Index Data

The output index data is discrete triangles, three indexes per triangle. This basically is 16 bits per index, which is RSX™ compatible. However, the input index data stream could be in an SPU-only compressed format, which must be decompressed before the triangle culling can be performed.

The index data is only necessary if triangle culling is to be performed, or if application-specific code requires them.

Skinning Matrix Data

The skinning matrices are two subsets of the entire object matrices. Edge's native format for skinning matrices is 3x4 row-major, but 4x4 matrices (both row- and column-major) are also supported. These ranges are indexed locally and translated to the uploaded ranges in the tools. This data is not necessary if no skinning is to be performed.

Skinning Index/Weight Data

The skinning weights and indexes are interlaced together in unsigned 8-bit format for the indexes and normalized unsigned 8-bit format for the weights. In addition to 4-bone skinning, Edge supports single-bone skinning for rigid/mechanical objects. This data is not necessary if no skinning is to be performed.

Blend Shape Data

The blend shape data itself is very similar to vertex data. However, for each vertex with a non-zero blend shape delta, it only contains attributes such as position and normal that are needed to perform the shape blending.

3 Using Edge Geometry

This chapter contains the following major sections:

- **Offline-Tool Processing:** Describes the processing that every object goes through in the tool to be optimized for runtime to process.
- **Runtime PPU Processing:** Describes how the PPU sets up the SPU jobs for processing information from the tools. The code layout and the structures it uses can be found in the *PlayStation®Edge Geometry Library Reference* document.
- **Runtime SPU Processing:** Describes SPU processing and how it affects the application writer, the PPU processing, and the Partitioner in the tools.

There are many subsystems and data structures involved in the geometry processing pipeline. This chapter discusses the processes that occur at run time for an ordinary skinned character. Details about all of these systems are provided in Chapter 4, “[Details About Edge Geometry Runtime](#)”.

Offline-Tool Processing

Tool processing is laid out in three layered libraries. Each layer is designed for a different Edge user type:

- **Advanced users:** Contains the Partitioner and Cache-optimizer, the libedgegeomtool.
- **Intermediate users:** Contains the COLLADA™ Parser and other mid-level helper functionality.
- **Beginner users:** Contains a fully working executable for edgegeomcompiler, which can be exported from a modeling package and which can process data into a fixed binary format. For details about edgegeomcompiler, see Chapter 5, “[Edge Offline Geometry Tool](#)”.

This is a basic process flow that occurs as each object goes through the tool. This process is described in the following sections. (For details about the functions listed in this section, refer to the *PlayStation®Edge Geometry Library for Offline Tool: Reference*.)

(1) Process Art Source File

Get the index and vertex data out from the art source file.

Then, triangulate the geometry meshes if they are not.

You can use the `edgeGeomTriangulatePolygons()` function in libedgegeomtool to help you process art source files.

(2) Choose Vertex Formats, Skinning Flavor, Culling Flavor, Index List Flavor, and Skinning Matrix Format

Choose a format for the input, output, and (optionally) blend delta streams and RSX™-only streams. For each stream, either select one of Edge’s built-in vertex formats (using the `edgeGeomGet[Spu, Rsx]VertexFormat()` functions), or build a custom format. When using a custom format, it is helpful to call `EdgeGeom[Spu, Rsx]VertexFormatIsValid()` on each final format to test for a variety of common errors.

You must also choose a flavor for index lists. Options include compressed and uncompressed, in both clockwise and counterclockwise winding order.

You must also choose a flavor for skinning. Edge supports skinning matrices with non-uniform scaling (fast), uniform scaling (faster), and no scaling (fastest). In addition to 4-bone skinning (the default), there is support for single-bone skinning (for rigid objects like machines or robots) that results in significantly smaller skinning data buffers.

You must also choose a flavor for triangle culling. This flavor determines whether triangle culling is enabled, and for which tests. You could, for example, disable the test for backfacing triangles if your engine is drawing two-sided triangles.

You must also choose a format for skinning matrix data (if skinning is being performed). Edge's native format is 3x4 row-major matrices; 4x4 matrices (both row- and column-major) are supported for convenience, but they are 33% larger and lead to increased memory usage.

(3) Split Up the Vertex Data into Batches

The tool should remove redundant vertexes first. The function `edgeGeomMergeIdenticalVertexes()` can help you achieve this.

Then, the tool should split up the vertex data into batches. A batch is a block of geometry that can traditionally be rendered with a single draw call – that is, it uses the same material and render settings. The `partitionSceneIntoBatches()` function in `libedgegeomtool_wrap.cpp` demonstrates an example of this process, but it assumes that the input scene is an `EdgeGeomScene` object.

If an input scene only uses one material, this step can be skipped; the entire scene is one batch.

(4) Partition Batches into Segments

The tool should partition all geometry data into several “segments”. Each segment will represent a single workload for SPU to process.

The `edgeGeomPartitioner()` function in `libedgegeomtool` can help you achieve this.

Below is a simplified description of the `edgeGeomPartitioner()` function:

- Create vertex to face adjacency lookup tables.
- Calculate the number of uniform attribute tables required for SPU processing.
- Create the initial segment.
- Create a list of unassigned faces.
 - If compressed indexes are being used, sort the list in spatially decreasing order.
- While there are still unassigned faces.
- If compressed indexes are used:
 - Find the first face in the unassigned list (searching in reverse order) that can be added to the current segment.
- If compressed indexes formats are not used:
 - Find the closest unassigned face to the current segment's median position value that can be added to the current segment.
 - If a face to add was not found and there are no more unassigned faces, exit the loop.
 - If a face to add was not found, create a new segment and restart the while loop.
 - Add the face to the segment and remove it from the unassigned list.
 - While (added a face to the current segment):
 - Loop through all of the connected and unassigned faces and find the face that can be added to the current segment and that has the lowest memory cost if adding that particular vertex.
 - If there is a face to add, add the face to the segment and remove it from the unassigned list.
- Run the specified pre-transform cache optimizer (for example, `kcache` optimizer) on the partitioned triangle lists.
- Return the segments.

Blend shapes are taken into account in the cost function. Typically, blend shapes streams are of equal or lesser size than its original mesh counterpart, because of the sparse data storage. Thus, an inability to fit blend shapes into the IO buffer is rarely an issue because it occupies the exact same memory space as the input vertexes themselves.

(5) Generate the Final Data Buffers for Each Segment

Now that each segment knows which triangles it contains, it is necessary to generate all the final data buffers for each segment (vertex streams, compressed index streams, skinning buffers, blend shape delta streams, and so forth). From the perspective of the tools, these buffers should be considered opaque blocks of data; they are meant to be processed only by the Edge runtime.

(6) Output into File(s)

Users can output segments in their own way to best meet their needs.

About RSX™ Cache Optimizer

The RSX™ Cache Optimizer is modeled after the K-Cache vertex optimization algorithm, but with many improvements that yield better run time performance. For example, edgegeomcompiler uses a hill climbing global optimization algorithm to achieve up to 5% better results.

For more information, see *An Improved Vertex Caching Scheme for 3D Mesh Rendering*. A URL is provided in the “[Related Documentation and Other Resources](#)” section of this document.

- Find the max vertex index in the index buffer.
- Create triangle and vertex adjacency lookup tables.
- Sort indexes so that the triangle list can be searched by means of a binary search.
- While all vertexes have not been consumed
 - Find the minimum cost vertex in the cache to use next.
 - If there are non-consumed vertexes in the mini-cache:
 - Find the best vertex in the mini-cache.
 - Best is determined by minimum cost.
 - Cost is a measurement of the RSX™ cycles, the number of adjacent vertexes, and the position in the fifo.
 - Otherwise, if there are non-consumed vertexes in the post transform cache:
 - Find the best vertex in the post transform cache
 - Best is determined by minimum cost.
 - Cost is a measurement of the RSX™ cycles, the number of adjacent vertexes, and the position in the fifo.
 - Otherwise, pick any non-consumed vertex of minimum adjacency.
 - Add all adjacent triangles which have not yet been added to the output index list.
 - While there are non-consumed adjacent triangles
 - For each adjacent triangle:
 - Find the lowest cost triangle to add.
 - Cost is a measurement of the RSX™ cycles, the number of adjacent vertexes, and the position in the fifo.
 - Consume the triangle and add it to the output index list.
 - For each adjacent vertex
 - If all adjacent triangles have been consumed
 - Consume the vertex

Runtime PPU Processing

The PPU runtime process required in Edge Geometry is fairly simple, so the library does not provide any runtime PPU functions.

There are two different basic types of geometry processing that can take place:

- Standard geometry processing which can do decompression, skinning, and triangle culling.
- Streaming decompression of a vertex data array which contains SPU-only attribute formats. This type of processing requires no supplemental data from tools.

The following shows two typical pipelines that occur as each segment goes through a different process.

Create Job from Segment to do Decompression, Skinning, and Triangle Culling

- (1) Insert a 16-byte aligned hole into the command buffer of the size specified by the `EdgeGeomPpuConfigInfo` and record the address of the hole as the command buffer hole address.
- (2) At the initial location of the hole and at every 128-byte boundary the hole crosses, a “Jump to Self” must be inserted.
- (3) Convert the command buffer hole address to an RSX™ IO offset.
- (4) If using blend shapes, for each blend shape being used:
 - If the blend shape buffer's `length != 0`
 - Copy over the blend shape buffer address and size into a `shapeInfo`.
 - Copy over the alpha from the user structure.
 - Increment the `shapeInfo` list pointer.
- (5) Set up the `CellSpursJob256` structure for this segment.

Create Jobs to Decompress

- (1) Compute the maximum stride of the input and output vertex data format flavors.
- (2) Compute the maximum vertexes per segment and round down to align the vertex data output to 16-bytes.
- (3) Compute the starting output address.
- (4) While there are still vertexes left to decompress.
 - Compute the number of vertexes in this segment.
 - Compute the DMA offsets and sizes.
 - Set up the `EdgeGeomSpuConfigInfo` structure.
 - Set up the `CellSpursJob256` structure.
 - Increment the vertex data output address.
 - Increment the job pointer.

Runtime SPU Processing

The SPU processing is provided, like the animation functionality, in a library of functions for maximum user control. This allows the user, for example, to make a material wave in the wind after it is skinned and many other possible game-specific geometry modifications.

The ordering of the processes is fairly serial.

The following is a possible function ordering to perform decompression, skinning, triangle culling, output decompression, and filling the command buffer hole:

(1) Initialize

The function `edgeGeomInitialize()` initializes the internal state and working buffers for the rest of processes.

This function must be called before any other functions.

(2) Decompress Vertexes

Extract the vertex attributes (positions, normals, tangents, texture coordinates, and so forth) out from the vertex stream. This can be done by calling function `edgeGeomDecompressVertexes()`. All decompressed vertex data is stored in “uniform tables” (one table per attribute), with four single-precision F32 values per attribute per vertex.

(3) Process Blend Shapes

If blend shapes are used, perform the blending here.

This can be done by calling function `edgeGeomProcessBlendShapes()`.

(4) Skin Vertexes

Skin the object, creating new positions, normals, tangents, and bi-normals.

This can be done by calling function `edgeGeomSkinVertexes()`.

(5) Decompress Indexes

If the input index data is in an SPU-only compressed format, it must be decompressed. This is only necessary if culling is to be performed, or if index data is required by other application-specific code.

This can be done by calling functions `edgeGeomDecompressIndexes()`.

(6) Occlusion Culling

If the number of occluders is non-zero, cull any triangles behind at least one occluder in screen space, and create a new index buffer without the culled triangles in it. An occluder is a world space quad defined by four planar vertices.

This can be done by calling function `edgeGeomCullOccludedTriangles()`.

(7) Cull Triangles

If the culling flavor is not set to `EDGE_GEOM_CULL_NONE`, cull any unwanted triangles and create a new index buffer without the culled triangles in it. The culling flavor can be used to specify whether back-facing or front-facing triangles can be culled, and whether triangles outside the view frustum can be culled.

This can be done by calling function `edgeGeomCullTriangles()`.

(8) Allocate Output Space

Allocate main memory output space for the output indexes, vertexes, and command buffer hole.

This can be done by calling function `edgeGeomAllocateOutputSpace()`.

(9) Output Indexes

Output the indexes by using DMA. This is only necessary if triangle culling is used, or if other application-specific code modified the input index data.

This can be done by calling function `edgeGeomOutputIndexes()`.

(10) Compress Vertexes

Create a new output vertex stream with the new vertex attributes as specified in overwriting the input data that is no longer necessary.

This can be done by calling function `edgeGeomCompressVertexes()`.

(11) Output Vertexes

Output the vertexes by using DMA.

This can be done by calling function `edgeGeomOutputVertexes()`.

(12) Begin Command Buffer Hole

Begin writing the command buffer hole.

This can be done by calling function `edgeGeomBeginCommandBufferHole()`.

(13) Set Vertex Data Array Commands

Write attribute addresses and formats the specified context.

This can be done by calling function `edgeGeomSetVertexDataArrays()`.

(14) Set Draw Index Array Commands

Write draw index arrays command into the specified context.

This can be done by calling function `cellGcmSetDrawIndexArray()`.

(15) End Command Buffer Hole

Write any ring buffer report commands and fill the unused portion of the hole with 0's (NOP). Finally, output the command buffer hole by means of DMA.

This can be done by calling function `edgeGeomEndCommandBufferHole()`.

Note: All SPU functions *should be called in the order they appear* in the reference document to ensure correctness.

4 Details About Edge Geometry Runtime

There are three types of data compression provided by Edge Geometry: two for vertex attribute data and one for index data. This chapter provides details about each type.

Geometry SPU-only Fixed-Point Compression

The compression scheme used to compress a position-like attribute is simple bit compression where each component is an n-bit by x-bit fixed-point value. To make maximum use of the bits available, the libedgegeomtool adds an offset to each fixed-point value to ensure that all output values are positive. A corresponding offset must be applied in the runtime during decompression to bring the per-component fixed point values back into the proper original range.

- The attribute exists within the same data table as other RSX™-formatted streams where the total number of bits used by each attribute must add up to a multiple of 8, and the total number of bits used by each component is no greater than 31.
- The data uses contiguous bits to express the data for each vertex.
- Within these contiguous bits, each component has its own separate field of bits and bit counts. For example, if the 24 bits represents three components, the first component could have an 8.2 bit-field, the second component a 7.2 bit-field, and the third component a 3.2 bit-field.
- Because this is not a native RSX™ format, the data table that this compressed attribute resides in would have to be passed through the SPU, decompressed, and converted to a RSX™-compatible format.

Geometry SPU-only Unit Vector Compression

The compression scheme used to compress a unit vector attribute, for example normal and tangent vector, is the smallest 2-bit compression where the two smallest components of the unit vector are represented with 10 bits per component representing the range $-\sqrt{2}/2$ to $\sqrt{2}/2$. The largest component is then reconstructed with:

$$\text{largest} = \sqrt{1 - \text{smallestA}^2 - \text{smallestB}^2}$$

A single bit is then used to represent the fourth component as -1 or 1. The fourth component would then be used in a vertex program to reconstruct the bi-normal from the normal and tangent. Two additional bits are finally used to represent which of the two components were the smallest.

- The attribute exists within the same data table as other RSX™-formatted streams where the total number of bits used for this attribute type is always 23 bits.
- Because this is not a native RSX™ format, the data table that this compressed attribute resides in would have to be passed through the SPU, decompressed, and converted to a RSX™-compatible format.
- This format should almost never be used in blend shape vertex delta formats; recall that its magnitude is always 1.0!

The format of the data is shown in Table 4:

Table 4 Data Format for Unit Vector Compression

| 0 | 10 | 11 | 19 | 20 | 21 | 22 | 23 |
|-----------|----|-----------|----|------|----|----|----|
| SmallestA | | SmallestB | | SHUF | | W | S |

- SmallestA is the first smallest value in the original unit vector.
- SmallestB is the second smallest value in the original unit vector.
- SHUF is an index into a shuffle mask table that designates the largest value.

- If W is set, the W value of the original unit vector was 1; otherwise it was -1.
- If S is set, the sign of the largest value was positive; otherwise it was negative.

Geometry SPU-only Indexes Compression

The compression scheme used to compress an indexed triangle list is a combination of high-level and low-level compression schemes. The high-level scheme compresses from the point of view of common relationships between the current triangle's indexes and previous triangle's indexes. The low-level compression is a sequential index, bit-range and delta compression that takes advantage of the steady increasing nature of an optimized index buffer and the fact that these deltas are small numbers. The end result is about 7 to 8 bits per triangle on a typical index buffer.

A large majority of triangles are one of three different combinations of the previous triangle indexes and a new index. This information is put into a 2-bit stream, two bits for every triangle. If a triangle is not one of the previous combinations, it is stored as a full triangle. The new indexes and uncompressed triangles are stored in a separate delta compressed and bit-packed format which must be decompressed first in the runtime before decompressing the 2-bit stream.

Every triangle which has only a single new index when compared to the previous triangle can be rotated into the following form while still maintaining the same display representation.

Table 5

| Index 0 | Index 1 | Index 2 | Rotation | Index 0 | Index 1 | Index 2 |
|---------|---------|---------|--------------|---------|---------|---------|
| New | Old | Old | $\wedge > 2$ | Old | Old | New |
| Old | New | Old | $\wedge > 1$ | Old | Old | New |
| Old | Old | New | $\wedge > 0$ | Old | Old | New |

Also, a triangle which fits into that form has only three possible combinations of the previous vertex.

Table 6

| | | |
|------------------|------------------|-----------|
| Previous Index 1 | Previous Index 0 | New Index |
| Previous Index 0 | Previous Index 2 | New Index |
| Previous Index 2 | Previous Index 1 | New Index |

This would give a 2-bit stream where 0 through 3 represent compressed triangle configurations and a value of 4 would mean a non-compressed triangle.

Table 7

| | | | |
|----|------------------|------------------|-----------|
| 00 | Previous Index 1 | Previous Index 0 | New Index |
| 01 | Previous Index 0 | Previous Index 2 | New Index |
| 10 | Previous Index 2 | Previous Index 1 | New Index |
| 11 | New Index | New Index | New Index |

All the new indexes would be stored in a separate table that would have its sequential elements stripped. Next a 1-bit table will be created and then the indexes left over will be delta compressed and then bit-range compressed.

This process would start with the index buffer after the 2-bit compression and sequential elements stripped step.

Table 8 A NEW Index Buffer After the 2-bit Compression Step

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 3 | 1 | 5 |
| 6 | 3 | 5 | 7 | 8 | 4 | 9 | 10 |
| 11 | 7 | 10 | 11 | 12 | 11 | 7 | 13 |
| 12 | 13 | 7 | 14 | 15 | 5 | 16 | 6 |
| 15 | 16 | 17 | 14 | 18 | 13 | 19 | 20 |

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 3 | 1 | 5 |
| 21 | 12 | 21 | 20 | 22 | 12 | 21 | 23 |

Next, the 1-bit table would be created and the sequential indexes stripped.

Table 9 Sequential Indexes Strip Out

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 3 | 1 | 3 | 5 | 4 | 7 | 10 | 11 |
| 11 | 7 | 12 | 13 | 7 | 5 | 6 | 15 |
| 16 | 14 | 13 | 12 | 21 | 20 | 12 | 21 |

This would create a 1-bit stream with the following values:

0,0,0,0,0,1,1,0,0,1,1,0,0,1,0,0,0,1,1,1,0,1,1,0,1,1,1,0,0,1,0,1,1,1,0,1,0,1,
0,0,0,1,1,1,0,1,1,0

Next, the index values would be delta compressed. To be SPU friendly, the first 8 indexes are not compressed, and indexes after the first 8 have their delta computed with the 8th previous index.

Table 10 Delta Compressed Indexes

| | | | | | | | |
|---|---|---|----|----|----|----|----|
| 3 | 1 | 3 | 5 | 4 | 7 | 10 | 11 |
| 8 | 6 | 9 | 8 | 3 | -2 | -4 | 4 |
| 5 | 7 | 1 | -1 | 14 | 15 | 6 | 6 |

The minimum index value would then be computed, which would in this example be -4. This number would then be subtracted from all indexes.

Table 11 Minimum Value Subtracted

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 7 | 5 | 7 | 9 | 8 | 11 | 14 | 15 |
| 12 | 10 | 13 | 12 | 7 | 2 | 0 | 8 |
| 9 | 11 | 5 | 3 | 18 | 19 | 10 | 10 |

Finally, the index buffer values would be packed into the smallest bit representation possible.

This information would be contained inside a single data block for SPU consumption. Inside this block would be of the following data.

Num indexes : U16 (specifies number of indexes in index table below)
 Base value : U16
 Num 1 bits : U16 (in bytes)
 Bits per index : U8
 Pad : U8
 1 bit table : (padded to 8-bits)
 2 bit table : (padded to 8-bits)
 Index table : unique, non-sequential bit-range and delta compressed indexes

Decompression on the SPU would be implemented in five steps.

- (1) Unpack bit-range index values in place.
- (2) Decompress deltas in place.
- (3) Decompress 1-bit table to temporary memory.
- (4) Copy the 2-bit table to temporary memory.
- (5) Decompress the 2-bit table.

Because this is not a native RSX™ format the indexes would have to be passed through the SPU, decompressed, and converted to a RSX™-compatible format.

Command Buffer Hole

In the Edge Geometry, the SPU constructs the command buffer hole, which contains commands that can only be known after information is available about where the data will be stored and how many triangles will be rendered.

First, the PPU places commands in the primary command buffer for vertex formats. Then the PPU leaves a quad word-aligned hole for the vertex attribute array addresses and draw commands – the output array addresses are not known at this time. The SPU will fill in the hole as shown in Table 12.

Table 12

| | | |
|-----------------------|--------------------------|---|
| InvalidateVertexCache | (if ring buffer used) | : 32 bytes |
| SetVertexDataArray | : position | : 16 bytes |
| SetVertexDataArray | : normal | : 16 bytes |
| SetVertexDataArray | : tangent | : 16 bytes |
| SetVertexDataArray | : texture coordinate | : 16 bytes |
| SetDrawIndexArray | | : (number of Indexes + 4607) / 48 bytes |
| SetWriteTextureLabel | (1 per ring buffer used) | : number of ring buffers * 16 bytes |
| SetNopCommand | | : fill the rest of the hole with 0 |

At the initial location of the hole and every 128-byte boundary the hole crosses, a “jump-to-self” command must be inserted by using `cellGcmSetJumpCommand`. This is mainly to block RSX™ and reduce the cost for synchronization.

Because the DMA in the system is out-of-order style, it is very important to insert this jump-to-self command into every 128-byte boundary the hole crosses to prevent RSX™ from reading old data.

SPU Input DMA List

The SPU Input DMA List, like the one in the `CellSpursJob256` structure, needs a tag for each independently loaded piece of data split into 16 KB each. The ordering of the tags is important.

Table 13

| Upper 32 bits | Lower 32 bits |
|---|---------------|
| Output stream description length | address |
| Indexes A length | address |
| Indexes B length | address |
| Skinning Matrices A length | address |
| Skinning Matrices B length | address |
| Skinning Indexes/Weight A length | address |
| Skinning Indexes/Weight B length | address |
| Vertexes A1 length | address |
| Vertexes B1 length | address |
| Vertexes C1 length | address |
| Vertexes A2 length | address |
| Vertexes B2 length | address |
| Vertexes C2 length | address |
| EdgeGeomViewportInfo length | address |
| EdgeGeomLocalToWorldMatrix length | address |
| EdgeGeomSpuConfigInfo length | address |
| Input Stream Fixed Point Offsets length | address |
| Input Stream description length | address |
| Primary input stream description length | address |

SPU Input User Data

The SPU Input User Data is used for commonly varying runtime parameters. These must be set by the PPU in every geometry segment and exist in the `CellSpursJob256` structure directly after the above DMA tags.

Table 14 SPU Input User Data

| Upper 32 bits | | Lower 32 bits | |
|-------------------------|--|-----------------------------|----------------|
| | | Output Buffer Info address* | |
| | | Command Buffer Hole address | |
| BlendShapeInfos address | | # of blend shapes | Culling Flavor |
| User Data 1 | | | |
| User Data 2 | | | |

* The appropriate bit of the Output Buffer Info address parameter is set when the address is to be used as a direct output buffer address.

SPU DMA Tag Performance Considerations

Data for the SPU processing is loaded into the input/output buffer through a DMA List. Each tag specifies a main memory address (EAL only) and a length, which must be a multiple of 16 bytes.

A single tag cannot be used to load more than 16 KB, so tables larger than 16 KB must be split into multiple pieces. This is handled by the tools and it provides the PPU processing the split up sizes based in the `EdgeGeomPpuConfigInfo` structure.

DMA is inefficient if the SPU LS (Local Store) address and the main memory address do not match when logically AND-ed with 0x70. This is important because mismatched data will load double the necessary packets. This cannot be solved at tool time and must be solved at run time. In Edge, DMA uses a simple strategy to solve this problem.

In the following example, the first data is 0x20 bytes from 0x130, and the next data is 0x110 bytes from 0x470. In this case, SPURS would automatically:

- Load 0x50 bytes from main memory 0x100 to LS 0x00.
- Load 0x130 bytes from main memory 0x450 to LS 0x50.

In this case, the absolute minimum DMA is performed, and the overhead for subsequent load is a possible 0x70 bytes per tag. In other words, the overhead is 0x70 bytes to start, and 0x70 bytes for each table, large or small. However, small tables will waste a fair amount of space in the input buffer. For a case like this it could trade some memory bandwidth for space, but the library will not do this for now.

The input/output buffer in SPURS' job is already 128-byte aligned so that makes the runtime processing fairly straightforward.

SPU Overall Buffer Layout

SPU processing uses two buffers: an input/output buffer, which has a maximum size of 48 KB, and a scratch buffer that contains the per-vertex-attribute uniform tables.

Because two jobs are running at any time, the maximum available local store space, LS, for a job is always 235 KB minus 48 KB. The 48 KB is for the other job's input/output buffers, which can be inputting the next job or outputting the previous job's data.

The basic order of data flow through the buffer is as follows:

- The SPURS Job Streamer loads all data into the input/output buffer through the DMA list in the `CellSpursJob256` structure. This includes both small data tables like `EdgeGeomSpuConfigInfo` and large data tables like the vertex and index data.
- The vertex data is decompressed from the input/output buffer into the scratch buffer.

- Skinning is optionally performed on the data in the scratch buffer.
- The vertexes are optionally transformed into clipping space in the scratch buffer.
- Triangle culling and/or occlusion culling is optionally performed on the input indexes and overwrites all of the consumed data in the input/output buffer.
- The vertex data in the scratch buffer is then compressed into the input/output buffer.
- After data is ready to be output, space is allocated for the output data. Please refer to the later section, "[Runtime Output Buffer Scheme](#)" for more information.

SPU Job Location Storage Management

The memory manager for the input/output buffer in the SPU local store is very simple. A *free pointer*, which points to the "start of free space", and a pointer are maintained. Buffers are allocated at the start. The "start" can be set to a previous value, which has the effect of freeing memory.

For example, allocate A, B, C and D at the start to get:

- A
- B
- C
- D

Then save the free pointer and allocate E and F to get:

- A
- B
- C
- D
- E
- F

Finally, set the free pointer to the saved value to free the area used by E and F:

- A
- B
- C
- D

Table 15 and Table 16 show the input/output buffer contents at various stages during processing. New allocations are in **bold**. The scratch buffer's allocations are completely static throughout the SPU processing.

**Table 15 Buffer Contents at Various Stages During Processing
from Initialization to Skinning State**

| Initial Stage | Initialize | Decompress Vtx | Blend Shapes | Skinning |
|--------------------------------|--------------------------------|---------------------------|---------------------------|---------------------------|
| Output stream description | Output stream description | Output stream description | Output stream description | Output stream description |
| Indexes | Indexes | Indexes | Indexes | Indexes |
| Skin Matrices | Skin Matrices | Skin Matrices | Skin Matrices | Skin Matrices |
| Indexes/Weights | Indexes/Weights | Indexes/Weights | Indexes/Weights | Indexes/Weights |
| Vertexes | Vertexes | Vertexes | Blend Shapes | |
| Decompress Stream Descriptions | Decompress Stream Descriptions | | | |
| Viewport Info | Viewport Info | | | |
| LocalToWorld | LocalToWorld | | | |
| EdgeGeomSpuConfigInfo | EdgeGeomSpuConfigInfo | | | |
| Fixed Point Offsets | Fixed Point Offsets | | | |

**Table 16 Buffer Contents at Various Stages During Processing (cont.)
from Index Decompression to the End of Job**

| Decompress Idx | Cull | Compress | Fill Command Buffer | End of Job |
|---------------------------|---------------------------|---------------------------|----------------------------|------------|
| Output stream description | Output stream description | Output stream description | Output stream description | |
| Decompressed Indexes | Decompressed Indexes | Decompressed Indexes | Decompressed Indexes | |
| | | Compressed Vertexes | Compressed Vertexes | |
| | | | 16 bytes for VRAM readback | |
| | | | Cmd Buffer Hole | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

User Functionality and Data

Data that the user code needs can be either before or after the geometry data in the Input/Output (IO) Buffer. Either is considered acceptable.

The data must be before the IO Buffer if it will be DMA-ed in by SPURS by means of the User Tags. The data must be after the output buffer if it will be DMA-ed in at processing time by means of specifying a smaller IO Buffer area than 48 KB in the tools.

For example, suppose you have custom information that specifies a wind direction and some partial bone information, and suppose that you want to incorporate this information by using DMA.

First, you need to reserve some space for your data in the tools. Next, you need to set one of the user tags on the PPU processing, which would DMA in your data. Finally, during SPU processing you would apply a sine or cosine wave function to the vertex positions in the uniform tables after the skinning process. For example:

- Use `edgeGeomInitialize()` to initialize the Edge Geometry Library.
- Use `edgeGeomDecompressVertexes()` to decompress input vertex data stream from the IO Buffer to the uniform tables.
- Use `edgeGeomProcessBlendShapes()` to apply the blend shapes to the uniform table.
- Use `edgeGeomSkinVertexes()` to apply the skinning matrices to the uniform tables.

- Here you would call your own function that would procedurally modify the position in the uniform tables to make them appear to wave in the wind.
- Use `edgeGeomCullTriangles()` and/or `edgeGeomCullOccludedTriangles()` to overwrite and create the new index buffer.
- Now that you know how large your index buffer will be, use `edgeGeomAllocateOutputSpace()` to reserve space for the output data.
- Use `edgeGeomOutputIndexes()` to start outputting the newly created indexes to the output buffer in main memory that was just allocated.
- Use `edgeGeomCompressVertexes()` to compress the uniform tables into a RSX™ format ready for RSX™ consumption.
- Use `edgeGeomOutputVertexes()` to start outputting the newly created vertexes to the output buffer in main memory.
- Use `edgeGeomBeginCommandBufferHole()` to start creating the data for the command buffer hole.
- Use `edgeGeomSetVertexDataArrays()` to write attribute addresses and formats.
- Use `cellGcmSetDrawIndexArray()` to write drawing commands.
- Use `edgeGeomEndCommandBufferHole()` to finish the hole and start outputting it to main memory.

The other case is when you are performing custom pose space deformation and need to DMA in a relatively large amount of data in order to keep space requirements at a minimum. The procedure is as follows.

First, reserve space for your data in the tools. Next, set some of the user data in so that the SPU will know where to DMA its information. Finally, during the SPU processing, DMA in your data to the unused space after the IO Buffer space used by the geometry library and apply your PSD functions to the vertex positions in the uniform tables before the skinning process occurs. For example:

- Use `edgeGeomInitialize()` to initialize the Edge Geometry Library.
- Use `edgeGeomDecompressVertexes()` from the IO Buffer to the uniform tables.
- Use `edgeGeomProcessBlendShapes()` to apply the blend shapes to the uniform table.
- Here you would call your own function which would DMA in the PSD configuration information and then loop over the PSDs and DMA in the appropriate information for each PSD and procedurally modify the uniform tables according to your PSD function.
- Use `edgeGeomSkinVertexes()` to apply the skinning matrices to the uniform tables.
- Use `edgeGeomCullTriangles()` and/or `edgeGeomCullOccludedTriangles()` to overwrite and create the new index buffer.
- Now that you know how large your index buffer will be, use `edgeGeomAllocateOutputSpace()` to reserve space for the output data.
- Use `edgeGeomOutputIndexes()` to start outputting the newly created indexes to the output buffer in main memory that was just allocated.
- Use `edgeGeomCompressVertexes()` to compress the uniform tables into a RSX™ format ready for RSX™ consumption.
- Use `edgeGeomOutputVertexes()` to start outputting the newly created vertexes to the output buffer in main memory.
- Use `edgeGeomBeginCommandBufferHole()` to start creating the data for the command buffer hole.
- Use `edgeGeomSetVertexDataArrays()` to write attribute addresses and formats.
- Use `cellGcmSetDrawIndexArray()` to write drawing commands.
- Use `edgeGeomEndCommandBufferHole()` to finish the hole and start outputting it to main memory.

There are additional complications related to how SPURS jobs work that would affect the efficiency of incoming DMAs. Any inbound DMA traffic would need to wait for other input buffers to complete their inputting before their own input would be possible, which would make the system less efficient.

Runtime Output Buffer Scheme

In Edge geometry, there are four different output schemes to hold the output from an SPU. In general order of importance they are:

- Double buffering
- Single buffering
- Ring buffering
- Hybrid buffering

The output buffer scheme can be set using structure `EdgeGeomOutputBufferInfo`, `EdgeGeomRingBufferInfo` and `EdgeGeomSharedBufferInfo`. More information about these three structures can be found in the *PlayStation®Edge Geometry Library Reference*.

Double Buffering

A double buffer is the simplest scheme. On each frame, the data output from the SPUs are directed towards one of two buffers for use on the next frame when rendering. This requires no SPU/RSX™ synchronization, because on the next frame all the data is immediately available for RSX™ use.

Advantages:

- Simplicity. The buffering scheme is very straightforward.
- Maximum SPU efficiency. The SPUs simply process data as fast as they can without stalling.

Disadvantages:

- Significant memory footprint. The size of the output buffers must be at least as large as the space requirements for the most complex frame rendered in the game, two of them. Also, the total SPU output size can be difficult to predict for some games, leading to grossly over-sized buffers.
- Allocation failures lead to graphical errors. If the user-provided buffer is not big enough, portions of the game scene will not be drawn, resulting in very obvious graphical errors.

Single Buffering

A single buffer is very similar to the double buffer. Here, on each frame the data output from the SPUs are directed towards one buffer for use on the currently rendering frame. Unlike double buffering, this does require SPU/RSX™ synchronization because it is possible for RSX™ to out-pace the SPU.

Advantages:

- Simplicity. The buffering scheme is fairly straightforward.
- Maximum SPU efficiency. The SPUs simply process data as fast as they can without stalling.

Disadvantages:

- Significant memory footprint. The size of the output buffers must be at least as large as the space requirements for the most complex frame rendered in the game, also, the total SPU output size can be difficult to predict for some games, leading to grossly over-sized buffers.
- Allocation failures lead to graphical errors. If the provided buffer is not big enough, portions of the game scene will not be drawn, resulting in very obvious graphical errors.

Ring Buffering

A ring buffer is a more memory efficient form of single buffering. On each frame the data output from the SPUs is directed towards one buffer for use on the currently rendering frame. Like single buffering, this does require SPU/RSX™ synchronization. Because the RSX™ might have consumed a portion of memory, the SPUs are informed about it through an RSX™ label and can then re-use the memory again for later

rendering in that frame. This buffer re-use results in a drastically reduced memory footprint required for the output buffers.

Advantages:

- Memory efficiency. Overall memory usage is significantly reduced; because the capacity of the ring buffer is effectively limitless, the actual buffer size can be quite small.
- Correctness. No data will ever be discarded due to insufficient buffer space; this eliminates the possibility of graphical errors resulting from buffer overflow.

Disadvantages:

- SPU performance can degrade. If the ring buffers are too small, or if the RSX™ is processing data too slowly, the SPUs will stall while they wait for the RSX™ to catch up.
- RSX™ performance degrades by about 1.5% on average due to the RSX™ labels required to inform the SPU of what parts of the buffer have been consumed.

Hybrid Buffering

Hybrid buffering is more memory efficient than single buffering and less memory efficient than ring buffering, but it has none of the disadvantages of ring buffers. With hybrid buffering, both a single shared buffer and a per-SPU ring buffer are used. Most allocations are fulfilled by the ring buffer, with the shared buffer used as emergency overflow space if the ring buffer is temporarily full.

With hybrid buffering, the SPU's allocation logic would be as follows:

- (1) Check the ring buffer to see whether there is space for the proposed allocation. If there is, make the allocation from the ring buffer and return. If not, continue to step 2 (*do not* block and wait for the RSX™ to consume more data).
- (2) If the initial ring buffer allocation failed, check the shared buffer to see whether there is space for the proposed allocation. If there is, make the allocation from the shared buffer (by means of the usual atomic locking mechanism) and return. If not, continue to step 3.
- (3) If the shared buffer is full, check the ring buffer again. This time, block until the allocation can be fulfilled.

This approach improves SPU efficiency, because less time is spent waiting for RSX™. Also, when the shared buffer allocation path is used, the ring buffer RSX™ label is no longer necessary, saving about 1.5% overall frame time on the RSX™. On the other hand, memory usage increases slightly due to the extra buffer. It is up to the developer to determine which buffer strategy is best for their application.

Advantages:

- Memory efficiency. Overall memory usage is significantly reduced; because the capability of the ring buffer is effectively limitless, the actual buffer size can be quite small.
- Correctness. No data will ever be discarded due to insufficient buffer space; this eliminates the possibility of graphical errors resulting from buffer overflow.
- Efficiency. Can significantly reduce or eliminate the RSX™ cost for writing RSX™ labels. Reduces the likelihood of SPU stalling, because the overflow buffer would have to be entirely consumed before any SPU stalling could even possibly occur.

Disadvantage:

- Extra memory is used for the shared buffer, which could otherwise be used for larger ring buffers (or for other data).

Cost Estimate for Edge Geometry SPU Processing

Table 17 shows a list of functions and roughly how many cycles each should take.

Table 17 Functions and Their Cycle Cost Estimates

| Function | Cycle Cost Estimate |
|-------------------|--|
| Decompress | 8 cycles per attribute per vertex |
| Blend Shapes | 8+4 cycles per attribute per vertex per blend shape |
| Skinning | 16+8 cycles per attribute per vertex* |
| Occlusion Culling | (11 + (11 cycles per occluder)) per vertex plus 12 cycles per triangle |
| Triangle Culling | 11 cycles per vertex plus 16 cycles per triangle |
| Compress | 8 cycles per attribute per vertex |

* If the attribute is a normal and the skinning mode is non-uniform scaling, the estimated cost for that attribute is 17 cycles per attribute per vertex.

Note that cycle cost is highly dependent on the type of attribute. The cost estimate per attribute is broken down in Table 18 and Table 19.

Table 18

| Decompression Attribute Type | Cycles per Vertex Cost Estimate |
|------------------------------|---------------------------------|
| I16N | 5.5 cycles |
| F32 | 6 cycles |
| F16 | 9 cycles |
| U8N | 4.5 cycles |
| I16 | 5.5 cycles |
| X11Y11Z10N | 6 cycles |
| U8 | 4.5 cycles |
| SPU-only Unit Vectors | 8.5 cycles |
| SPU-only Fixed Point | 8 cycles |

Table 19

| Compression Attribute Type | Cycles per Vertex Cost Estimate |
|----------------------------|---------------------------------|
| I16N | 5.5 cycles |
| F32 | 6 cycles |
| F16 | 9 cycles |
| U8N | 5.5 cycles |
| I16 | 5.5 cycles |
| X11Y11Z10N | 7 cycles |
| U8 | 5.5 cycles |

5 Edge Offline Geometry Tool

The Edge offline geometry compiler tool, `edgegeomcompiler`, is a sample tool to demonstrate how `libedgegeomtool` can be used in an asset pipeline. Specifically, it demonstrates how to take data in COLLADA™ format and pass it through the Edge library, producing a binary file that can be loaded into the runtime sample.

Building

`edgegeomcompiler` depends on three libraries: `libedgegeomtool`, `FCollada`, and `libxml2`.

`FCollada` is a freely available library for reading and writing COLLADA™ data by Feeling Software (www.feelingsoftware.com). It can be downloaded from their website (registration is required). The version of `FCollada` required is 3.05B.

`libxml2` is a freely available library for handling data in xml format. It is used by `FCollada` to handle the low level access to the COLLADA™ files, which are xml files. A version of `libxml2` is included in the `FCollada` distribution, or it can be downloaded from <http://xmlsoft.org/>.

Usage

A simple command line tool, `edgegeomcompiler` is invoked with two arguments, the input COLLADA™ file name and the output binary file name.

```
edgegeomcompiler [OPTIONS] <input> <output>
```

Options:

- `--enable-scaled-skinning`: By default, `edgegeomcompiler` sets the skinning flavor to `EDGE_GEOM_SKIN_NO_SCALING` if skinning data is available. This will use the fastest possible skinning code in the runtime, but will produce incorrect results if the animation transforms include a scale factor. Because `edgegeomcompiler` does not know which animations will be applied to a particular scene, it provides this option to forcibly set a scene's skinning flavor to `EDGE_GEOM_SKIN_NON_UNIFORM_SCALING` if desired.
- `--inv-bind-mats-out <file>`: Specifies a file to output the array of inverse bind pose matrices to. The matrices are written out in transposed 3x4 format.

6 Overview of Edge Animation

Edge Animation Design

Edge Animation is a low-level skeletal animation system and is designed to be a lightweight and efficient SPU library, linked with your SPU code.

The system is not wrapped or abstracted behind PPU interfaces, to allow easy game-specific customizations. The user is in control of `SpuMain()` and how the different parts of the animation system, provided by Edge or by game-specific code, are glued together.

This level of flexibility and control means it is simple for the user to execute custom game code on the SPU as part of the animation job.

The Edge Animation system has been designed around a “mainline” case of ~80 joints characters with non-trivial blend trees.

SPU Usage

Samples use SPURS job and this code can be copied as a starting point, to integrate Edge Animation in your own project.

The library itself makes no assumption of any SPU programming model.

- Code is compiled position independent, but can be used in a fixed position model.
- All DMA operations performed within Edge Animation are interrupt-safe. The library can be used with SPU interrupt enabled.
- There are no SPURS/SPU Threads function calls inside the library. It can also be used on raw SPUs.

The only assumption made by the SPU library is that a `printf` handler compatible with `spu_printf()` is available on the PPU side. `spu_printf()` is not called during normal processing, but only when some critical errors are encountered.

The PPU side of Edge Animation only exposes some setup functions. All processing is meant to run on SPUs.

Reference Implementation

A cross-platform reference implementation is also provided. This source code is provided to demonstrate the features of Edge Animation and has not been optimized for performance.

Win32 Implementation

A win32 implementation is also provided. Both PPU and SPU side functions are exposed in a single library. The library may use either the reference implementation or methods optimized for a SSE2-enabled CPU.

Animation and skeleton data structures are similar to the ones in the PlayStation®3 version, but stored as little endian instead of big endian. Files must be exported in PC (little endian) mode, by using the `-pc` command line switch of `edgeanimcompiler`.

Animation Processing

Animation Pose Stack

Animation processing uses a pose stack during blend tree evaluation. The number of elements in the stack depends on how much scratch memory is passed by the user and the size of the skeleton (number of joints,

number of user channels, and so forth). The pose stack includes support to transparently use main memory (using per-SPU temporary storage) if it runs out of space in local store.

Full/Partial Animation Evaluation

Full and partial animations are supported. Partial animations are internally handled by assigning a weight of 0x00 to every joint not affected by the partial animation.

Animation storage uses non-uniform key frames, with compressed quaternion data.

Weighted Animation Blending

All blend operations work in local-space with `EdgeAnimJointTransform`, and have logic to handle partial animations through joint weights.

Additive Animations

Additive delta animations, typically used for things like hit reactions, are supported in the tools pipeline and the runtime. Additive blends specify work between base and base + delta.

Joint Transform/Matrix Conversions Functions

A comprehensive set of utility functions are exposed to perform all conversions, in any direction:

- Local-space Joint Transforms to World-Space Matrices (for either standard 4x4 or transposed 3x4)
- Local-Space Joint Transforms to World-Space Joint Transforms
- World-Space Joint Transforms to Local-Space Joint Transforms
- Joint Transforms to Matrices (for either standard 4x4 or transposed 3x4)
- Matrices (for either standard 4x4 or transposed 3x4) to Joint Transforms

Parent scale compensation can be performed when converting joints from local space to world space (and vice versa) by setting appropriate flags in the skeleton's joint hierarchy data structure.

Scalar User Channels

Animation channels representing arbitrary scalars are accessible. They can be used to drive any custom processing, such as blend shapes.

Custom User Callbacks

Complex functions expose callbacks to allow custom user processing. The "locomotion" sample shows how to handle character locomotion by means of callbacks used in `edgeAnimProcessBlendTree()`.

Tools Processing

Edge Animation is COLLADA™-compatible. It can be used at different levels, to integrate in existing pipelines:

- **High-level:** Standalone `edgeanimcompiler` executable, as-is.
- **Mid-level:** Modified `edgeanimcompiler` tool still using the COLLADA™ framework.
- **Low-level:** Custom tool using `libedgeanimtool` directly to generate the final data.

`edgeanimcompiler` uses `FCollada`. The lowest-level `libedgeanimtool` library that does the bulk of the work is independent from COLLADA™.

Data Structures

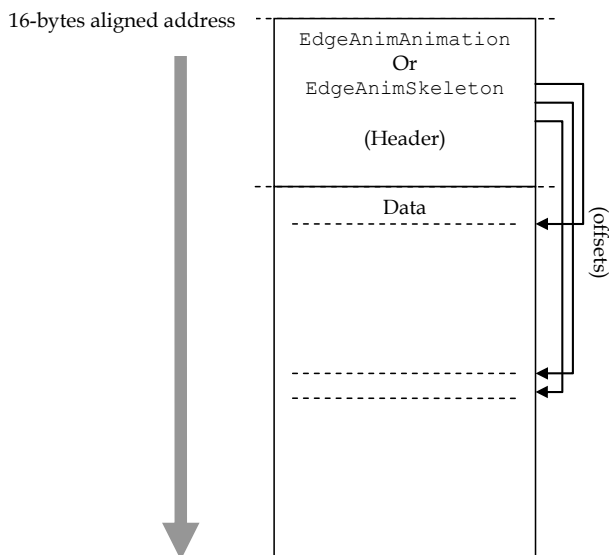
All data structures built by offline tools are designed to be loaded as a single 16 bytes-aligned block in main memory. They consist of a header, followed by variable-size chunks of data. The header uses offsets, instead of pointers, to reference data, to be position-independent.

There are two fundamental data structures built by the tools:

- **EdgeAnimSkeleton:** Contains the skeleton hierarchy, bind pose, and hashed strings for joint names, and so forth.
- **EdgeAnimAnimation:** Contains a single animation, bound to skeleton when built by the tools.

Each of these data structures also contains a 4 bytes version tag. This tag is incremented at every modification of the format.

Figure 1 Edge Animation Data



7 Using Edge Animation

This example shows how easy it is to use Edge Animation. It is similar to “[samples/edge/character-sample](#)”, found in the “[Edge Geometry And Animation](#)” section of Chapter 23, “[Sample Programs](#)”.

PPU – Global

(1) Initialization

Call `edgeAnimPpuInitialize()` to initialize the library. The `EdgeAnimPpuContext` structure will contain info about the main-memory per-SPU temporary storage.

(2) Load data

Load skeleton and animation data structures. They must be aligned to 16 bytes. Data structures written by external tools are designed to be loaded directly in memory and do not require any relocation or fix up.

(3) Begin main loop

(4) Setup animation jobs

This is user-side code, but you can follow the examples based on SPURS job provided with the SDK. In the simplest case, you need to send to the SPU job:

- A skeleton
- A blend tree
- Where to write final data

Because of the way Edge Animation is designed, everything is exposed to the user and the SPU-side library can be linked with higher level game code.

More advanced use will involve supplying to the job higher-level game data, and the blend tree will be built directly in SPU local store.

(5) Run jobs

Synchronization can be done with the usual `job_send_end`, if using SPURS job.

(6) End main loop

(7) Termination

Terminate `libedgeanim` by calling `edgeAnimPpuFinalize()`. This will release the resources allocated to the library.

SPU – For Each Job

(1) Initialization

Call `edgeAnimSpuInitialize()` to initialize the SPU library. The `EdgeAnimSpuContext` structure will then be passed to all `edgeAnim` SPU function calls. There is no static information saved internally.

`edgeAnimSpuInitialize()` does mostly memory-management operations, using the scratch buffer provided by the user.

- It configures the pose stack, according to skeleton information to use scratch space as efficiently as possible.
- It configures external storage in main memory, according to information in `EdgeAnimPpuContext` and the unique SPU ID passed by the caller.

(2) Process the blend tree

`edgeAnimProcessBlendTree()` will process the blend tree and leave the final result on top of the pose stack.

At that point, data is in local-space, stored in `EdgeAnimJointTransform` (Rotation/Translation/Scale) structures.

(3) Conversions

User code can get the pose on top of the stack, and convert it to any format needed. A typical method is as follows:

- (1) Convert to world space:

`edgeAnimLocalJointsToWorldJoints()`

- (2) Convert to matrices:

use either `edgeAnimJointsToMatrices4x4()`, compatible with the standard `Vectormath::Aos` library

or use `edgeAnimJointsToMatrices3x4()`, for consumption by Edge Geometry skinning

Alternatively, the joints can be converted directly to world matrices:

- Convert to world space matrices directly:

use either `edgeAnimLocalJointsToWorldMatrices4x4()`, compatible with the standard `Vectormath::Aos` library

or use `edgeAnimLocalJointsToWorldMatrices3x4()`, for consumption by Edge Geometry skinning

Converting directly to world space matrices does not produce an identical result to the first method because the matrices are multiplied from the already-computed parent world matrix, which ensures that the local scale orientations are preserved along the kinematic chain. This is not true for the first method, where the world scale value is in the joint's world orientation.

(4) Use results

Either DMA results out or use them directly in your SPU code.

(5) Termination

Terminate the SPU library by calling `edgeAnimSpuFinalize()`.

8 Details About Edge Animation Runtime

Pose Stack

An animation pose is defined by the information from the skeleton. It is composed of:

- An array of `EdgeAnimJointTransforms` (size aligned to a multiple of 4 joints).
- An array of joint weights, stored as fixed point `uint8_t`.
- An array of user channels, stored as scalar floats.

The pose stack uses both a local store buffer, part of the scratch memory passed to `edgeAnimSpuInitialize()`, and an optional main memory buffer allocated per SPU in `edgeAnimPpuInitialize()` to be used when the local pose stack overflows.

This external storage comes at a performance cost, and is not supposed to be used in the mainline case, but it increases flexibility. It allows processing of rare larger blend trees, and also allows the SPU code to grow in debug builds.

The number of slots available in local store/external storage is computed by `edgeAnimSpuInitialize()` and is dependent on the number of joints and user channels defined in the skeleton.

The stack is exposed through the following three functions:

- `edgeAnimPoseStackPush()`
 - Adds one pose on top of the stack (push).
 - If the system is running out of local store space, it starts an output DMA of the least recently used pose to the main memory external storage associated with this SPU. If the system runs out of main memory storage space too, the SPU runtime asserts.
- `edgeAnimPoseStackPop()`
 - Discards the pose on top of the stack (pop).
 - DMA the last pose stored in main memory external storage back into SPU local store, if any.
- `edgeAnimPoseStackGetPose()`
 - Updates the `EdgeAnimPoseInfo` structure with pointers to the different arrays of this pose.

After processing the blend tree, the stack will contain a single pose.

Blend Tree Processing

Calling `edgeAnimProcessBlendTree()` works in two stages:

- (1) The blend tree is recursively converted to a linear command list.
- (2) `edgeAnimProcessCommandList()` is then called internally to process this list. DMA latencies are hidden in a 3-stage pipeline.

A blend tree is composed of:

- Leaves, stored as an array of `EdgeAnimBlendLeaf`. A leaf is an animation evaluated at a given time; it also contains a user `uint32_t` for user processing in evaluation callbacks.
- Branches, stored as an array of `EdgeAnimBlendBranch`. A branch is a blend operation between two tree elements (either other branches or leaves); it also contains a user `uint32_t` for user processing in evaluation callbacks.

All indexes used within the blend tree must be used with a bitwise OR with one of these two flags:

- `(index | EDGE_ANIM_BLEND_TREE_INDEX_LEAF)`, if it represents a leaf.
- `(index | EDGE_ANIM_BLEND_TREE_INDEX_BRANCH)`, if it represents a branch.

In both cases index 0 is the first element of the array. The SPU runtime will assert if neither (or both) of these flags is set.

Command List

Internally, this tree will be converted to a command list. Note that individual commands store pointers to the blend tree nodes (`EdgeAnimBlendLeaf`, `EdgeAnimBlendBranch`) – they are not designed to be used directly by the user.

The possible commands are:

- `EDGE_ANIM_CMD_EVAL` (maps to a “leaf” node)
- `EDGE_ANIM_CMD_PUSH_AND_EVAL` (maps to a “leaf” node)
- `EDGE_ANIM_CMD_BLEND_AND_POP` (maps to a “branch” node)
- `EDGE_ANIM_CMD_MIRROR`
- `EDGE_ANIM_CMD_END_LIST`

The command list is processed in a three-stage pipeline, to hide DMA latencies. There is no pipelining between different command lists; it means first DMA evaluations will involve stall. This system is assuming non-trivial blend-trees, and will not be optimal with only two animations being evaluated and blended.

Table 20 Pipeline Stages

| Stage | <code>EDGE_ANIM_CMD_PUSH_AND_EVAL</code> | <code>EDGE_ANIM_CMD_BLEND_AND_POP</code> |
|--------------|--|--|
| -2 (preload) | Start DMA of animation header. | No operation |
| -1 (load) | Wait for animation header DMA completion. Find the frameset needed for the evaluation time and start DMA. | No operation |
| 0 (execute) | Wait for frameset DMA completion. Evaluate. | Blend |

Blending Operations and Partial Animation Logic

Blend Operations

Branches can perform multiple blending operations, as defined in the `EdgeAnimBlendOp` enumeration.

Additive and subtractive animations are also supported. Add and sub operations have different meanings depending on the channel type, described in Table 21.

Table 21 Additive and Subtractive Animations

| Channel Type | Add(a,b) | Sub(a,b) |
|---------------|--|--|
| Rotation | $a * b$ | $a * \text{conjugate}(b)$ |
| Translation | $a + b$ | $a - b$ |
| Scale | $a * b$ | a / b |
| User channels | Default mode: $a + b$ Minmax mode: $\max(a, b)$ | Default mode: $a - b$ Minmax mode: $\min(a, b)$ |

Each animation contains an array of `uint8_t` (unsigned 8 bits fixed point integers) used as blend weights; this array is also used to support partial animations through the whole blend tree.

A zero weight value means that this joint is undefined. This value is common when using partial animations.

Notes:

- Because joints weights are written by the blending function, each pose in the pose cache/stack must have an array of joint weights.
- To save space in the animation, if *jointWeight* is NULL, it is assumed to be a full body animation with all weights set to 1 (0xFF).

Table 22 shows all possible blend operations.

Table 22 EdgeAnimBlendOp Operations

| EdgeAnimBlendOp | Operation |
|---|--|
| EDGE_ANIM_BLENDOP_BLEND_LINEAR | Blend between: Left [$\alpha = 0.0$] Right [$\alpha = 1.0$] |
| EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_RIGHT | Blend between: Left [$\alpha = 0.0$] Add(Left,Right) [$\alpha = 1.0$] |
| EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_LEFT | Blend between: Right [$\alpha = 0.0$] Add(Right,Left) [$\alpha = 1.0$] |
| EDGE_ANIM_BLENDOP_COMPOSE_ADD | Compose: Add(Left,Right) [both defined] Otherwise Left or Right |
| EDGE_ANIM_BLENDOP_COMPOSE_SUB_RIGHT_FROM_LEFT | Compose: Sub(Left,Right) [both defined] Otherwise Left or Inverse(Right) |
| EDGE_ANIM_BLENDOP_COMPOSE_SUB_LEFT_FROM_RIGHT | Compose: Sub(Right,Left) [both defined] Otherwise Right or Inverse(Left) |

Linear Blend

Operation:

- EDGE_ANIM_BLENDOP_BLEND_LINEAR

The blend factor calculation is as follows:

if (rightWeight > leftWeight)

$$\text{blendFactor} = \left(\alpha * \frac{\text{leftWeight}}{\text{rightWeight}} \right) + \left(1 - \frac{\text{leftWeight}}{\text{rightWeight}} \right)$$

else

$$\text{blendFactor} = \left(\alpha * \frac{\text{rightWeight}}{\text{leftWeight}} \right)$$

$$\text{outputWeight} = (1 - \text{blendFactor}) \text{leftWeight} + \text{blendFactor} * \text{rightWeight}$$

Table 23 shows the behavior of partial animation logic for linear blends:

Table 23 Partial Animation Logic for EDGE_ANIM_BLENDOP_BLEND_LINEAR

| Left | Right | Output | Output Weight |
|-------------------------|-------------------------|-----------|---------------|
| Undefined (weight=0) | Undefined (weight=0) | Undefined | 0.0 (0x00) |

| Left | Right | Output | Output Weight |
|----------------------|----------------------|---------------------------------|---|
| Defined (weight>0) | Undefined (weight=0) | Left | Left weight |
| Undefined (weight=0) | Defined (weight>0) | Right | Right weight |
| Defined (weight>0) | Defined (weight>0) | Blend(Left, Right, blendFactor) | Blend(Left weight, Left weight + Right weight, blendFactor) |

Add Delta Blend

Operations:

- EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_RIGHT
- EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_LEFT

These blend operations are variations allowing blending between a base channel and this base channel plus or minus a delta channel, the operation name itself is self explanatory.

The blend factor calculation is as follows:

$$\text{blendFactor} = \alpha * \text{deltaWeight}$$

$$\text{outputWeight} = \text{baseWeight}$$

Partial animation logic requires the base channel to be defined. If only the delta channel is defined the output is undefined. The following tables show ADD_DELTA_RIGHT as an example.

Table 24 Partial Animation Logic for EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_RIGHT

| Left | Right | Output | Output Weight |
|----------------------|----------------------|---|---|
| Undefined (weight=0) | Undefined (weight=0) | Undefined | 0.0 (0x00) |
| Defined (weight>0) | Undefined (weight=0) | Left | Left weight |
| Undefined (weight=0) | Defined (weight>0) | Undefined | 0.0 (0x00) |
| Defined (weight>0) | Defined (weight>0) | Blend(Left, add(Left,Right), blendFactor) | Blend(Left weight, Left weight + Right weight, blendFactor) |

Compose

Operations:

- EDGE_ANIM_BLENDOP_COMPOSE_ADD
- EDGE_ANIM_BLENDOP_COMPOSE_SUB_LEFT_FROM_RIGHT
- EDGE_ANIM_BLENDOP_COMPOSE_SUB_RIGHT_FROM_LEFT

The “compose” behavior is different from delta blends because no blend is performed (alpha is ignored). There is no notion of base and delta, both channels are considered equal.

The blend factor calculation is as follows:

$$\text{blendFactor} = 1.0 \text{ (ignored)}$$

$$\text{outputWeight} = \text{leftWeight} + \text{rightWeight}$$

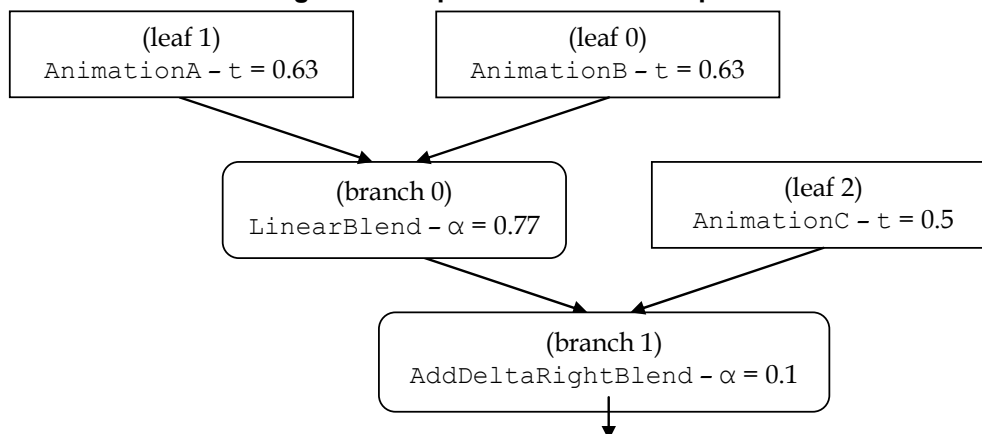
The partial animation logic differs between addition and subtraction mode, as shown in Table 25 and Table 26.

Table 25 Partial Animation Logic for EDGE_ANIM_BLENDOP_COMPOSE_ADD

| Left | Right | Output | Output Weight |
|-------------------------|-------------------------|-----------------|----------------------------|
| Undefined (weight=0) | Undefined (weight=0) | Undefined | 0.0 (0x00) |
| Defined (weight>0) | Undefined (weight=0) | Left | Left weight |
| Undefined (weight=0) | Defined (weight>0) | Right | Right weight |
| Defined (weight>0) | Defined (weight>0) | Sub(Left,Right) | Left weight + Right weight |
| Undefined (weight=0) | Undefined (weight=0) | Undefined | 0.0 (0x00) |
| Defined (weight>0) | Undefined (weight=0) | Undefined | 0.0 (0x00) |
| Undefined (weight=0) | Defined (weight>0) | Undefined | 0.0 (0x00) |
| Defined (weight>0) | Defined (weight>0) | Sub(Left,Right) | Left weight + Right weight |

Table 26 Partial Animation Logic for EDGE_ANIM_BLENDOP_COMPOSE_SUB_RIGHT_FROM_LEFT

| Left | Right | Output | Output Weight |
|-------------------------|-------------------------|-----------------|----------------------------|
| Undefined (weight=0) | Undefined (weight=0) | Undefined | 0.0 (0x00) |
| Defined (weight>0) | Undefined (weight=0) | Undefined | 0.0 (0x00) |
| Undefined (weight=0) | Defined (weight>0) | Undefined | 0.0 (0x00) |
| Defined (weight>0) | Defined (weight>0) | Sub(Left,Right) | Left weight + Right weight |

Example: Simple Blend Tree**Figure 2 Simple Blend Tree Example****Table 27 Simple Blend Tree Example: Leaf Array**

| animationHeaderEa | animationHeaderSize | evalTime | userData |
|-------------------|-----------------------|----------|----------|
| &AnimationB | AnimationB.sizeHeader | 0.63 | NULL |
| &AnimationA | AnimationA.sizeHeader | 0.63 | NULL |
| &AnimationC | AnimationC.sizeHeader | 0.50 | NULL |

Table 28 Simple Blend Tree Example: Branch Array

| Command | Left | Right | Alpha |
|---|---|---|-------|
| EDGE_ANIM_BLENDOP_ BLEND_LINEAR | 1 EDGE_ANIM_BLEND_ TREE_INDEX_LEAF | 0 EDGE_ANIM_BLEND_ TREE_INDEX_LEAF | 0.77 |
| EDGE_ANIM_BLENDOP_ BLEND_ADD_DELTA_RIGHT | 0 EDGE_ANIM_BLEND_ TREE_INDEX_BRANCH | 2 EDGE_ANIM_BLEND_ TREE_INDEX_LEAF | 0.10 |

edgeAnimProcessBlendTree takes the index of the root node (the final output) as a parameter. For this example, it is “branch 1” so the index is “1 | EDGE_ANIM_BLEND_TREE_INDEX_BRANCH”.

This blend tree example gets converted to the following command list (Table 29):

Table 29 Simple Blend Tree Example: Command List

| Command | Argument |
|-----------------------------|-----------------|
| EDGE_ANIM_CMD_PUSH_AND_EVAL | &blendLeaf[1] |
| EDGE_ANIM_CMD_PUSH_AND_EVAL | &blendLeaf[0] |
| EDGE_ANIM_CMD_BLEND_AND_POP | &blendBranch[0] |
| EDGE_ANIM_CMD_PUSH_AND_EVAL | &blendLeaf[2] |
| EDGE_ANIM_CMD_BLEND_AND_POP | &blendBranch[1] |
| EDGE_ANIM_CMD_END_LIST | NULL |

Callbacks

A user provided callback, passed to edgeAnimProcessBlendTree(), can be called for each command, after each stage of the pipeline.

For EDGE_ANIM_CMD_PUSH_AND_EVAL commands, associated with a leaf, see EdgeAnimLeafCallback.

For EDGE_ANIM_CMD_BLEND_AND_POP commands, associated with a branch, see EdgeAnimBranchCallback.

The user provided callback can determine what the current stage is by checking the pipelineStage argument (0, -1 or -2) passed to it.

Mirroring

Mirroring can optionally be applied at any stage in the blend tree by passing the EDGE_ANIM_FLAG_MIRROR flag when creating a leaf or branch. The results of the evaluated or blended pose will then get mirrored according to a predefined mirroring specification.

The mirroring specification is passed to edgeAnimProcessBlendTree() and consists of an array of EdgeAnimMirrorPair structures, each defining a mirroring operation and a pair of joint indices. The two joints have the specified operation applied to them, and their entries in the evaluated pose are exchanged. It is valid for the indices to be the same, in which case the mirroring is applied to one joint only. Any joints not defined within *spec* remain unmodified from their evaluated state.

When creating an EdgeAnimMirrorPair structure, there are three cases to consider:

- (1) The joint is not paired (for example, spine)
- (2) The joint is paired and has a non-paired parent (for example, left_shoulder & right_shoulder)
- (3) The joint is paired and has a paired parent (for example, left_arm & right_arm)

Edge Animation provides the following macros for mirroring across the $x=0$ plane, corresponding to each of the three cases:

- `EDGE_ANIM_MIRROR_SPEC_NON_PAIRED`
- `EDGE_ANIM_MIRROR_SPEC_LINK`
- `EDGE_ANIM_MIRROR_SPEC_PAIRED`

It is important that the skeleton has well defined joint orientations. In particular, these macros assume that the y -axis of each joint extends along the bone direction. If a different joint orientation or mirroring plane is required, the user must specify their own mirroring operations in place of these macros. The following table shows the format of the joint mirroring *spec*, which allows for arbitrary flips and sign inversions of the rotation and translation components.

Table 30 Joint Mirroring Specification

| 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
|----|----|----|----|----|----|----|-----|
| Rx | Tx | Ry | Ty | Rz | Tz | Rw | pad |

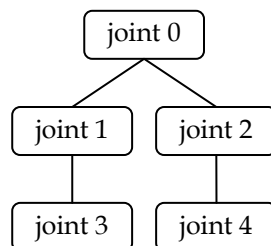
For each 4-bit block:

- Bit 0 (MSB) is the negate bit. If this bit is set, the final value will be negated.
- Bit 1 is unused.
- Bits 2 and 3 give the index of the component that the library will take the value of.

For example, `EDGE_ANIM_MIRROR_SPEC_LINK` has the value `0xB8219203` and results in the following mapping of the rotation and translation components:

```
(Rx, Ry, Rz, Rw)  ->  (-Rw, Rz, -Ry, Rx)
(Tx, Ty, Tz)      ->  (-Tx, Ty, Tz)
```

Here is an example of a simple skeleton hierarchy and associated mirror *spec*:



```
EdgeAnimMirrorPair(0, 0, EDGE_ANIM_MIRROR_SPEC_NON_PAIRED)
EdgeAnimMirrorPair(1, 2, EDGE_ANIM_MIRROR_SPEC_LINK)
EdgeAnimMirrorPair(3, 4, EDGE_ANIM_MIRROR_SPEC_PAIRED)
```

This *spec* will mirror an animation across the $x=0$ plane, and will exchange joints 1, 3 with joints 2, 4, respectively. Note that binary operations only need to be specified once (the ordering of the joint pair is not important).

For an example of mirroring, see `samples/edge/mirror-sample`.

Precomputed Poses

It is possible to specify precomputed poses in the blend tree. For example, you may want to re-use a pose that was computed during processing of a previous blend tree. To do this, pass the address of the pose to the leaf constructor and specify one of the following flags (depending on whether the pose comes from main memory or local store):

- `EDGE_ANIM_FLAG_POSE_FROM_MAIN`
- `EDGE_ANIM_FLAG_POSE_FROM_LOCAL`

To save a pose for later re-use, get the pose from the stack and write from `jointArray` onwards. For example, the following gets the pose at the top of the stack and writes it to `poseEa` in main memory:

```
EdgeAnimPoseInfo pose;  
edgeAnimPoseStackGetPose(&spuContext, &pose, 0);  
cellDmaLargePut(pose.jointArray, poseEa, spuContext.sizePose, tag, 0, 0);
```

Animation Encoding and Compression

Constant Channels

Many channels are actually constant in an individual animation. On small animations, like the ones used in a complex blend, they represent a significant portion of all the data. Edge Animation attempts to minimize the size required for them.

Constant data identical to the base pose is ignored and no information is stored in the individual animation. On a typical character, this constant data represents most of the scale and non-root translation data, but also a significant amount of rotation channels (fingers).

Channels that are different from the base pose but constant in a given animation are handled specially to reduce their impact on size.

Non-uniform Animation Data

Animations are initially sampled at a constant frequency. They then undergo an optimization pass that removes any samples that can be approximated within a given tolerance by interpolation. The remaining sparse key-frame data is then split into constant and non-constant data. Constant data that differs from the base pose is stored separately, and the non-constant data is partitioned into discrete *frame sets*.

A frame set contains the key-frames for all non-constant channels over a certain time range within the clip, and is the unit of data that is ultimately uploaded to the SPU during evaluation. The frame set memory size is constant, and hence the time range of any given frame set depends on the sparseness of the sample data at that point. In addition to the actual key-frame data, a frame set also contains bit streams which define which key-frames exist for each frame and for each channel.

The frame set is structured as follows:

- Initial joint rotations
- Initial joint translations
- Initial joint scales
- Initial user channels
- Intra-frame rotation bits
- Intra-frame translation bits
- Intra-frame scale bits
- Intra-frame user bits
- Intra-frame joint rotations
- Intra-frame joint translations
- Intra-frame joint scales
- Intra-frame user channels

The initial data contains the data of *all* non-constant channels for the first frame of the frame set. The intra-frame data contains the sparse key-frame data of the remaining frames. The intra-frame bit streams contain a bit per sample, with 1 indicating the existence of a key-frame at that sample.

Final data for all the channels is also required by the evaluator and this is taken from the initial data of the next frameset.

Table 31 shows an example data structure with four rotation channels, one translation channel, and four intra-frames.

Table 31 Example Data Structure

| | | | |
|------------|--------------|--------------|------------|
| R0 frame 0 | R1 frame 0 | R2 frame 0 | R3 frame 0 |
| T0 frame 0 | Intra R bits | Intra T bits | R0 frame 1 |
| R0 frame 2 | R0 frame 4 | R1 frame 1 | R2 frame 1 |
| R2 frame 2 | R2 frame 3 | R3 frame 2 | R3 frame 4 |
| T0 frame 1 | T0 frame 2 | | |

- Intra R bits: 1101 1000 1110 0101
- Intra T bits: 1100

Quaternion Compression

Rotation data represent the bulk of animation data. The default compression scheme used to compress a unit quaternion is smallest 3 compression, where the three smallest components of the unit quaternion are

represented with 15 bits per component representing the range $-\frac{\sqrt{2}}{2}$ to $\frac{\sqrt{2}}{2}$. The largest component is

then reconstructed with $\text{largest} = \sqrt{1 - \text{smallestA}^2 - \text{smallestB}^2 - \text{smallestC}^2}$. Two additional bits are finally used to represent which of the three components were the smallest.

The format of the data is as follows:

Table 32 Data Format of Compressed Quaternion

| 0 | 14 | 15 | 29 | 30 | 44 | 45 | 46 | 47 |
|-----------|----|-----------|----|-----------|----|------|----|----|
| SmallestA | | SmallestB | | SmallestC | | SHUF | | S |

- SmallestA is the first smallest value in the original unit quaternion.
- SmallestB is the second smallest value in the original unit quaternion.
- SmallestC is the third smallest value in the original unit quaternion.
- SHUF is an index into a shuffle mask table which designates the largest value.
- S is set when the sign of the reconstructed largest component is negative.*

* The tools currently ensure that the largest component is always positive by negating the quaternion if necessary.

Bit-Packing

Bit-packing can optionally be applied to any channel. In this mode, the channel specifies an arbitrary bit length per component, depending on a user-specified error tolerance.

The per-channel packing specification is encoded as follows:

Table 33 Bit-Packing Specification Format

| 0 | 1 | 4 | 5 | 9 | 10 | 11 | 14 | 15 | 19 | 20 | 21 | 24 | 25 | 29 | 30 | 31 |
|------|------|---|-------|---|------|------|----|-------|----|------|------|----|-------|----|-------|----|
| sgnA | expA | | mantA | | sgnB | expB | | mantB | | sgnC | expC | | mantC | | index | |

- sgnA, expA, mantA are the number of sign, exponent, and mantissa bits used to encode the first component
- sgnB, expB, mantB are the number of sign, exponent, and mantissa bits used to encode the second component
- sgnC, expC, mantC are the number of sign, exponent, and mantissa bits used to encode the third component
- index is the index of the reconstructed fourth component (quaternions only)

If the number of exponent bits is non-zero for any component, the packed values are unpacked as floats. Otherwise, they are unpacked as normalized fixed-point values.

In the case of user channels there is only one component and its spec is encoded in `sgnA`, `expA`, `mantA`.

Note: Bit-packing is more expensive to evaluate, and is SPU only – no PPU implementation is provided.

Evaluation

The following is a detailed flow of the process that happens during evaluation of a clip in `_edgeAnimEvaluate()`.

- Copy the skeleton base pose to the output pose.
- Decompress the constant channels to the output pose, using the mapping from the constant channel tables.
- Let `frameFraction` be the fractional value of the evaluation frame.
- Let `tableR` be the id table of non-constant rotation channels.
- For each non-constant rotation channel.
- Let `keyRData` be the starting address of the intra-frame compressed key-frames for this channel.
- Let `totalBits` be the number of enabled bits in the frame data's bit list for this channel.
- Let `prevBits` be the number of enabled bits previous to this frame's bit in the frame data's bit list.
- If `prevBits = 0` then
 - Let `keyA` be the initial key-frame decompressed.
- Else
 - Let `keyA` be the key-frame decompressed at `keyRData + sizeof(R) * prevBits`
- If `prevBits = totalBits` then
 - Let `keyB` be the final key-frame decompressed.
- Else
 - Let `keyB` be the key-frame decompressed at `keyRData + sizeof(R) * (prevBits+1)`
 - Let `bBits` be the number of contiguous disabled bits before, and including, this channel's bit.
 - Let `aBits` be the number of contiguous disabled bits after this channel's bit.
 - Let `alpha` be equal to $(bBits + frameFraction) / (aBits + bBits + 1)$
 - Let `interp` be the slerp between `keyA` and `keyB` with `alpha`.
 - Write `interp` to the output pose, using the mapping from `tableR`.
- Repeat for the non-constant translation, scale, and user channels.

9 Edge Animation Tools

Introduction

A sample tool, `edganimcompiler`, is provided that can be used to convert skeletons and animations stored in COLLADA™ format files into a runtime format that can be processed by Edge Animation on the SPU.

A library called `libedganimtool` is also provided that centralizes Edge Animation-specific processing. The library can be used to create tools that do not require the use of the COLLADA™ framework. The `edganimcompiler` tool is built with this library.

Usage

`edganimcompiler` is a simple command line tool that takes COLLADA™ format files as input and, depending on the arguments given, can generate skeleton, animation, or additive animation runtime files as follows:

```
edganimcompiler -skel <in Collada> <out skeleton> [-userchannels <in text>]
```

```
edganimcompiler -anim <in Collada> <in skeleton> <out anim>
```

```
edganimcompiler -additiveanim <in base Collada> <in Collada> <in skeleton>  
<out additive anim>
```

```
edganimcompiler -selfadditiveanim <in Collada> <in skeleton>  
<out additive anim>
```

Table 34 describes each animation compiler argument.

Table 34 Animation Compiler Arguments

| Argument | Description |
|---------------------|--|
| <in Collada> | Specifies a COLLADA™ format file containing skeleton and animation data. |
| <in base Collada> | Specifies a COLLADA™ format file containing base anim for creating an additive anim. |
| <in text> | Specifies a text file containing case-sensitive user channel names on each line. |
| <in/out skeleton> | Specifies a binary skeleton file generated by this tool. |
| <out anim> | Specifies a binary animation file generated by this tool. |
| <out additive anim> | Specifies a binary additive animation file generated by this tool. |

Processing Overview

As shown in the usage section, there are three types of file that can be output by the tools. Each type of output requires specific processing that is centralized in the `libedganimtool` library; however a common pattern of processing is followed by all three:

- Parse command line arguments to determine output file type.
- Extract required data from input files.
- Fill out structures exposed by `libedganimtool`.
- Call `libedganimtool` to process data and export output runtime file.

Skeleton Processing

The inputs for skeleton processing are a COLLADA™ format file containing the exported joints and an optional text file containing a list of case-sensitive user channel names, one per line. The compiler reads the COLLADA™ file and then collects all joint nodes found in the scene into a list that is used during the rest of the data extraction phase.

After validating the list of joints (there must only be a single root joint), the compiler then extracts the bind pose by evaluating all joint animation at time 0. During this step, hashes of the joint names are also calculated and stored. These hashes are used by the SPU runtime library to quickly look up joints.

If the user specified a list of user channels on the command line, the number of channels is determined and a list of their name hashes is generated to be passed later to libedgeanimtool.

The final step in extraction is to build a list of parent joint indexes for each joint collected earlier. At this point the compiler has all the data required by libedgeanimtool to process and export the skeleton binary. A Skeleton structure is filled out with this data and is then passed into the `ExportSkeleton()` function exposed by libedgeanimtool.

The `ExportSkeleton()` function takes the extracted data and performs further processing in order to condition the data into a format that is handled by the runtime SPU library. This involves partitioning the list of joints into sets of 4 so that processing can take advantage of the SIMD nature of the SPU architecture. After this additional processing, the file is written to disk into the format required by Edge Animation.

User Channels

Edge Animation supports animation of arbitrary user scalar channels. Currently the only type of user channel supported by edgeanimcompiler is blend shape weights. They can be specified at skeleton export time by passing a text file to edgeanimcompiler through the `-userchannels` argument. The text file can list the weights (one per line) using the following syntax:

```
controller.shape
```

where `controller` is the name of the controller node, and `shape` is the name of the target shape.

Hashed versions of these names get stored with the skeleton, and any user channels not listed here will be ignored when exporting an animation file.

See “[samples/edge/blendshape-anim-sample](#)”, found in the “[Edge Geometry And Animation](#)” section of Chapter 23, “[Sample Programs](#)”, for an example of exporting and rendering a mesh with animating blendshapes.

Animation Processing

The inputs required to generate an animation binary are a COLLADA™ file that contains the animation key frames and a skeleton binary file that has been previously exported by the tool.

The input skeleton file contains the “bind” skeleton that will be used in the game for animation and skinning. It is used during tools processing to calculate binding and partial animation information for the animation. The first part of the tool’s processing is to extract the same data described in the previous section on skeleton processing from both input files.

The next stage involves extracting the animation data from the COLLADA™ scene. This is achieved in two passes. The first pass examines all the key frames in all the animation curves that affect joints in the input animation and builds an ordered list of their key times. From this list, the sampling rate, start and end times are determined.

The next pass evaluates the animation of each joint at each key time in order to sample the rotation, translation and scale curves. These sampled key frames are stored for each joint and collected into a structure exposed by libedgeanimtool so that they can be passed to it for processing and export. If the skeleton specified on the command line has any user channels, these are sampled too in a similar fashion.

At this point, the animation data is ready to be passed to libedgeanimtool to be written out to the runtime format. Some additional processing is done by libedgeanimtool at this stage. Firstly any constant animation channels are removed that are the same as the base pose of the bind skeleton. Then, optionally, the animation is optimized. This reduces the size of the final output file by removing any redundant key frames that can be approximated by interpolation. Finally, the animation is partitioned into frame sets (see the section titled “[Non-uniform Animation Data](#)” in Chapter 8, “[Details About Edge Animation Runtime](#)”).

Once the framesets have been partitioned, all that remains is for the binary file to be written to disk and also to a plain text file that contains various statistics about the animation data.

Additive Animation Processing

There are two methods available for creating additive animations; either by subtracting the first frame of the input animation from the rest or by specifying another separate animation file which is subtracted from the other. Either mode can be specified from the command line with the arguments `-selfadditiveanim` and `-additiveanim`, respectively. The output is a binary animation file that contains the resultant delta animation which can be additively blended with the base animation or single pose at run time.

The processing performed by the tool is the same as for the single animation case, except that another animation or pose is extracted from the input animation and then this is subtracted from the other in the `GenerateAdditiveAnimation()` method provided in libedgeanimtool.

The method determines whether joint and user channel key frames are present in both animations before using the appropriate subtract method for each channel (rotate, translate, scale, and so forth) at the key times specified in the animation that is being subtracted from. After the animations have been subtracted, then the same optimization processing used in normal animation processing is applied to the result.

Finally, the animation is exported to the binary file as in the previous section (including Edge-specific frameset partitioning) and also a plain text file is written out containing the animation data statistics.

Compression

edgeanimcompiler has two modes of compression: default mode and bit-packed mode.

The default mode uses the smallest-three compression scheme for rotations and leaves all other channel types as uncompressed floats.

Bit-packed mode applies arbitrary bit-packing to all channels using a single specified tolerance value. This will typically give much better compression, particularly if there are many non-rotation channels. The cost is an increase in SPU processing time.

To specify bit-packing use the `-bitpacked` argument when calling edgeanimcompiler. The tolerance can be specified with the `-tolerance` argument. For example:

```
edgeanimcompiler -anim run.dae bind.dae run.anim -bitpacked -tolerance 0.001
```

Note that libedgeanimtool offers more fine-grain control over compression modes and tolerances than is currently supported by edgeanimcompiler. For details, see the *PlayStation®Edge Animation Library for Offline Tool: Reference*.

10 Using Edge Animation and Edge Geometry Together

Skinning

To render a skinned character, your code additionally needs the inverse bind matrices. When multiplied with the world space joint matrices, they provide the final skinning matrices used by Edge Geometry.

To export the inverse bind matrices from `edgegeomcompiler`, pass the `--inv-bind-mats-out` argument, specifying the output file. The resulting binary will consist of a flat array of transposed 3x4 matrices. The number of matrices is equal to the number of joints in the skeleton (any joints not involved in skinning will be given an identity matrix).

At run time, the following sequence of SPU functions can be called to create the skinning matrices from an array of local joint transforms.

```
edgeAnimLocalJointsToWorldMatrices3x4(worldMats, localJoints, ...)
edgeAnimMultiplyMatrices3x4(skinMats, worldMats, invBindMats, ...)
```

See “[samples/edge/character-sample](#)”, found in the “[Edge Geometry And Animation](#)” section of Chapter 23, “[Sample Programs](#)”, for an example of exporting, animating, and rendering a skinned character.

Note: `edgegeomcompiler` computes the inverse bind matrices as $B * M$, where M is the transform of the target mesh, and B is the inverse of the world space bind pose joint transform. Because only one inverse bind matrix is output per joint, a conflict can arise if meshes with different transforms are influenced by the same joint (`edgegeomcompiler` will warn if this is the case). To avoid this, the skinned meshes should have their transforms frozen prior to exporting.

11 Overview of Edge Zlib, LZMA, and LZO

Algorithms

Edge provides SPU-optimized implementations of three lossless decompression algorithms, as well as two SPU implementations of lossless runtime compression.

The Edge decompression algorithms are based on zlib, LZMA and LZO, and they offer the following trade-offs:

- **Edge LZMA** – provides the best compression ratio (that is, smaller files), but is also the slowest to decompress.
- **Edge LZO** – provides the fastest decompression, at the expense of the compression ratio.
- **Edge Zlib** – offers a good middle ground between the above two.

Edge Zlib and Edge LZO provide the SPU implementation of lossless runtime compression.

Table 35 and Table 36 show sample decompression/compression values from one set of test data.

Compression ratios and speeds can vary greatly depending on the input data. Your own data may show very different values.

Table 35 Edge LZO, Zlib, and LZMA Decompression: Sample Values from Test

| | Edge LZO Level 1(06) | Edge LZO Level 1(14) | Edge Zlib Level 1 | Edge Zlib Level 9 | Edge LZMA Level 1 | Edge LZMA Level 9 |
|---|-------------------------|-------------------------|----------------------|----------------------|----------------------|----------------------|
| Decompression Speed (MiB/s of output data) | 451 | 397 | 107 | 111 | 17 | 19 |
| Compression Ratio (% of master) | 71 | 61 | 54 | 52 | 53 | 44 |

Table 36 Edge LZO and Zlib Compression: Sample Values from Test

| | Edge LZO | Edge Zlib Level 1 | Edge Zlib Level 9 | Edge LZMA |
|--|----------|----------------------|----------------------|-----------|
| Compression Speed (MiB/s of input data) | 76 | 6 | 3 | N/A |
| Compression Ratio (% of master) | 61 | 58 | 57 | N/A |

Due to the limitations of Local Store, some internal buffer sizes have been decreased. For example, Edge Zlib may provide compression ratios that are not quite as good as those achieved by an offline compressor.

Some Suggested Use-Cases

If loading data from the Blu-ray Disc or HDD, Edge Zlib can decompress data faster than it is received from the drive. So Edge Zlib is a good choice for speed and compression ratio in such cases.

For downloaded games, where file size may be more important, Edge LZMA may be a good choice if the slower decompression speed is acceptable.

Edge LZO may be the best choice if runtime compression is needed or when very fast decompression is necessary.

12 Using Edge Zlib, LZMA, and LZO

The Edge Zlib, Edge LZMA, and Edge LZO interfaces are very similar. This chapter explains the procedure for performing decompression, using Edge Zlib as its example. The following chapters will specify those cases where the procedure described here is significantly different from what is required for either decompression with Edge LZMA or Edge LZO or compression with Edge Zlib or Edge LZO.

Implementation Details

Edge Zlib, Edge LZMA, and Edge LZO have a hierarchical implementation (SPU library, used by a SPURS Task, embedded in a PPU library). This allows the user to choose at what level they want to interface to the library.

SPU Library

On SPU there is a library that contains all the relevant compression/decompression code. In the example of Edge Zlib, this library (`libedgezlib.a`) provides the following functions:

- `edgeZlibInflateRawData()` – performs decompression within Local Store. It takes in pointers to two buffers in Local Store. One contains the input compressed data to be decompressed. The other is the buffer to which the output uncompressed data is to be written.
- `edgeZlibDeflateRawData()` – performs compression within Local Store. It takes in pointers to two buffers in Local Store. One contains the master input data. The other is the buffer to which the compressed output data is to be written.

SPURS Task

If the user is using SPURS, Edge provides two SPURS Tasks, one for compression (Deflate) and one for decompression (Inflate). These tasks call the processing functions in the SPU library mentioned above.

The “Inflate Task” pulls work off a queue of items to be decompressed (the “Inflate Queue”).

The “Deflate Task” pulls work off a queue of items to be compressed (the “Deflate Queue”).

Each element of the queue provides details of the source and destination buffers of the compression/decompression. The task DMAs in the input data, performs the compression/decompression, and then DMAs out the output data as necessary.

When the buffers are completely compressed/decompressed and written out, the task can atomically decrement a counter in main memory by one. This provides a way to detect when a collection of associated items is completely decompressed (because the counter was initialized to `numItems`, and has now reached zero). When the counter reaches zero, the task can notify an event flag. This provides a way for a PPU Thread to sleep while waiting for a specific set of work to complete.

If an error occurs during decompression, the top bit of the counter will be set to 1.

After an item in the queue has been completely compressed/decompressed, before looping back for more work, the task polls to determine if there is a higher priority workload to which the SPU should yield. Yielding at this point in the loop would occur after one set of data has been sent out, but before the next is read in. This means that the context save size of the SPURS Task can be kept to a minimum (2 KB of stack space). If you make any modifications to the code, you may need to save more context area than this.

PPU Library

On PPU there is a library that provides an interface to the Inflate/Deflate Tasks described above. In the example of Edge Zlib, this library (`libedgezlib.a`) is used for creating the task and attaching it to a user-specified SPURS Taskset. By creating multiple tasks (one per SPU), multiple SPUs can compress/decompress in parallel.

This library also provides functions for the creation and usage of the Inflate/Deflate Queue that the tasks pull from.

Procedure for Using Edge Zlib from the PPU

The easiest way to use Edge Zlib, Edge LZMA, and Edge LZO is by using the PPU library mentioned above, because this will handle all the SPU communication internally via the embedded Inflate/Deflate Tasks.

(1) Initialization

Create an Inflate Queue by calling `edgeZlibCreateInflateQueue()`. This returns `EdgeZlibInflateQHandle`. This Inflate Queue needs a buffer of main memory to work in. The required size for this buffer can be determined by calling `edgeZlibGetInflateQueueSize()`. The buffer should be 128 byte-aligned.

After initializing your SPURS Instance and a task set, call `edgeZlibCreateInflateTask()`. This creates an Inflate Task for decompressing. If you want to run decompression in parallel on multiple SPUs, you can call this function multiple times, once for each SPU. This function needs a 128 byte-aligned buffer of main memory to potentially page the task's context out to. The required size of this buffer can be determined by calling `edgeZlibGetInflateTaskContextSaveSize()`.

(2) Adding work

By calling `edgeZlibAddInflateQueueElement()`, you can add work to the Inflate Queue. This function takes in pointers to the input and output buffers in main memory, as well as the size of the input data and the expected size of the output data. If the output data decompresses to a size different from what is expected, then the SPU will assert.

If you want to add multiple work items, initialize a `uint32_t` counter in main memory to the number of segments you want to add. Pass the address of this counter into `edgeZlibAddInflateQueueElement()`. The counter is decremented by one when each segment completes decompressing. You can also pass in a pointer to an event flag. This event flag will be set after the counter reaches zero.

(3) SPU processing of the work

The Inflate Tasks automatically run whenever there is work on the queue to be decompressed. However, if there is no work, or if there is other higher priority work, a task yields the SPU. The yield point happens at the end of decompressing one item before starting on the next, so there is minimal context to save.

(4) When the SPU decompression work is done

Your program can poll for completion of the work by querying the value of the counter. A PPU Thread can sleep while waiting for a collection of work to decompress by waiting for the event flag to be set (`cellSpursEventFlagWait`).

The code should verify that the counter is zero. If the top bit is set, then an error occurred on the SPU during decompression.

(5) Termination

Call `edgeZlibShutdownInflateQueue()` to shutdown the Inflate Queue. After shutting down the task set, be sure to free the memory you allocated for the Inflate Queue and for the context yield space.

Procedure for Using Edge Zlib from the SPU

Edge Zlib, Edge LZMA, and Edge LZO are provided as SPU libraries, so if you have your own SPU management solution you can choose to interface to it at this level.

Note: The SPU library is PIC and interrupt-safe.

Common Details

This section discusses common details and techniques that apply to both Edge Zlib and Edge LZMA.

SPURS Tasks

Each call to `edgeZlibCreateInflateTask()` generates a SPURS Task. This SPURS Task initiates a loop that pulls work from a shared Inflate Queue and processes it. By creating just one task, the decompression work only runs on one SPU at a time. However, by creating multiple tasks, the tasks can run in parallel on different SPUs. Each SPU pulls a new item of work from the queue and therefore works on a different segment of data.

The Inflate Task runs when there is work waiting to be done. If there is no work left then the task sleeps. After the task has decompressed one item off the Inflate Queue, it polls to see if there is higher priority work to which it should yield. If there is, it yields immediately with a minimal context save. If not, it attempts to allocate more work. If there is not any work, it sleeps until work is available.

Note: If you modify the SPU code, depending on the change, the minimal context yield might not be enough and a larger context area might need to be stored.

The preceding discussion similarly applies to `edgeLzmaCreateInflateTask()`, `edgeLzmaCreateDeflateTask()`, `edgeLzmaCreateInflateTask()`, `edgeLzmaCreateDeflateTask()` or `edgeLzmaCreateDeflateTask()`.

Inflate Tasks

When performing decompression, your program needs to know in advance how big the uncompressed data is expected to be. This allows the allocation of the output buffer in main memory to be performed in advance and the SPU merely needs to fill it. The expected uncompressed size is passed into the SPU decompression code, and if the expected value turns out to be incorrect then the SPU asserts.

Applying compression to data can, in some cases, make the data larger rather than smaller. This situation works fine with Edge Zlib, Edge LZMA and Edge LZO. Of course, if applying compression makes the data larger, then compression was pointless and it would be better to use the uncompressed data instead.

Deflate Tasks

The Deflate Tasks work very similarly to the Inflate Tasks. Obviously it is not possible to know in advance how small the compressed version of the data will become. Instead, the user program passes in a large buffer for the output data to be stored into, and once the data has been written out, the SPU will communicate the size of the compressed output data by writing this value to a user specified `uint32_t` in main memory.

Applying compression can, in some cases, cause the data to grow larger than the input data was. In this case there is a choice between whether the Deflate Tasks should store out the compressed data anyway, or whether it should store out the original master data since it was smaller. The decision between always storing the compressed data or storing whichever is smaller can be chosen by an option on the work added to the Deflate Queue. The decision about which was stored is communicated back by setting the top bit of the `uint32_t` containing the output compressed size.

Sharing SPUs with Other Systems

Compression or decompression requires significant processing per byte (for example, Edge Zlib can take up to 1 ms to decompress a single 64 KB segment). During this time, if higher priority work comes up for

the specific SPU to work on, this work will not be taken until the given segment has been completely decompressed and the SPU reaches the next yield point. If this delay before taking higher priority work is a problem, you can improve the situation by segmenting the data at a size lower than 64 KB so that yield points are more frequent.

Due to the way the SPURS Job pipeline works, if each segment is queued as a SPURS Job there might actually be two jobs in the pipeline after the currently processing job, which would all need to be finished before the higher priority work could be processed. This means the delay could be as much as 3 ms for Edge Zlib segmented decompression. This is one of the reasons for implementing as SPURS Tasks instead of SPURS Jobs.

Some game teams initialize an SPU Thread Group of five SPUs and an SPU Thread Group of one SPU so that the game's main work happens on the five SPU Thread Group, which in theory is always running; however, if the operating system needs to use an SPU it pre-empted the lower-priority single SPU Thread Group. The choice of work to put on the single SPU must be managed carefully because it can be pre-empted at any time. If decompression is already waiting for loaded data, then the extra delay of the thread group having been pre-empted might not be an issue, and so teams might choose to perform decompression on this "sixth SPU".

Segmented Decompression

When performing decompression, due to the limitations of Local Store, Edge LZMA and Edge LZO limit the size of the compressed and uncompressed data so that all the data can fit in LS at once. When the PPU interface is used to queue up decompression work, the compressed and uncompressed versions of the data must not be more than 64 KB each.

To handle data greater than 64 KB, segmenting it prior to compression is recommended. This means that there is a hard barrier between segments and, thus, when they are compressed they cannot reference each other's data. Therefore, each segment is fully self-contained and can be decompressed on its own without requiring data from any other segments. If a segment fits in LS, it can be decompressed without needing to fetch any extra data from main memory.

Due to the hard-barrier between segments, the compression ratio during segmentation can give slightly inferior compression ratios in comparison to compressing the data as one large file; however, when segments are 64 KB each (the default), this difference should usually be small.

Because segments are independent, separate segments can be decompressed at the same time in parallel across multiple SPUs, so segmented decompression is parallelizable, unlike non-segmented decompression.

Similar comments apply for compression with Edge LZO of data greater than 64 KB.

Decompression In-place

An Inflate Task works on a single segment by using DMA to read in the compressed data, decompressing it in Local Store, then using DMA to write out the uncompressed data. Therefore, on a single segment that fits fully in LS (in other words, less than 64 KB input/output), decompression in place can be performed by having the output data sent out to overwrite the input data. This assumes that the input data buffer is large enough to contain the (presumably larger) output data without corrupting adjacent data.

When loading in a larger file that has been segmented and compressed offline, decompression in-place requires a little more effort. One solution is to "fragment" the memory so that after loading, the segments of input compressed data are spread out with each fragment located into its own buffer that is big enough for the uncompressed data to be written over it. In other words, after fragmentation there is a buffer (probably 64 KB) that contains the input compressed data in it, followed by padding up to the end of the buffer. This segment can then be decompressed in place on top of itself, overwriting the input compressed data and the padding. Also, other segments can similarly be decompressed in place in parallel. The sample program, "samples/edge/zlib-inflate-inplace-sample", found in the "[Edge Zlib/LZMA/LZO](#)" section of Chapter 23, "[Sample Programs](#)", shows how this can work, although the idea applies equally well to Edge LZMA and Edge LZO. The "fragmentation" task is bound by bandwidth

to main memory and does very little actual processing. It simply sets up the data in preparation for the Inflate Tasks to access.

Because the final uncompressed data presumably requires all the segments to be contiguous, the buffers where the input data was placed (in other words, input data+padding) must also be contiguous, and therefore, the compressed data *must* be less than or equal to the uncompressed data for each segment.

Another way of doing decompression in place for a large file is to limit the decompression to only run on a single SPU and to schedule the segments for decompression in an appropriate order. Using this method, the data is loaded into the bottom of the output buffer. The last segment is decompressed first, to the end of the buffer, and the remaining segments are decompressed backward from the next-to-last segment forward. If there is a header on the file (for example, `EdgeSegzipFileHeader`), then the compressed data of the first one or more segments might need copying forward instead of backward.

13 The Specifics of Edge Zlib

Characteristics

Edge Zlib is an implementation of the DEFLATE compression algorithm, which allows lossless compression and decompression. The original implementation of the library was written by Jean-loup Gailly and Mark Adler. This release contains Edge Zlib, which provides a modified and optimized implementation of zlib decompression for running on the PlayStation®3 SPUs.

Edge Zlib offers an SPU implementation of both the decompression and compression functionality. Appropriate data for decompressing may be generated offline using a tool such as gzip, or by using a library for compression such as zlib.

Edge Zlib, unlike Edge LZMA and Edge LZO, can compress or decompress data larger than can fit in Local Store at a single time by streaming it in and out of main memory as needed.

Differences Between Edge Zlib and zlib

Edge Zlib provides the following features, not found in zlib:

- Optimized for SPU instruction set and Local Store.
- A SPURS Task is provided that allows the segments for decompression or compression to be queued up from the PPU.
- Decompression or compression of separate segments can be run in parallel on multiple SPUs at once.
- Some internal buffers in the SPU code have been moved from the heap to static variables to avoid the need to `malloc()` them.

Licensing

Edge Zlib contains a modified version of zlib 1.2.3, an open source project from <http://zlib.net/>. The modified source code is distributed under the terms of the zlib license described in the file `zlib.h` and available online at http://zlib.net/zlib_license.html. This license allows its use in commercial applications.

Headers on the DEFLATE Data

The SPU decompression code (`edgeZlibInflateRawData`, `edgeZlibFetchAndInflateRawData`) only expects to receive a pointer to the raw data that it needs to decompress. Any header must be parsed off in advance. For example, zlib compression might add a 2-byte header and 4-byte footer, which is not expected. Alternatively there might be a `.gz` or `.zip` file header, say, that should be skipped before passing the pointer to `edgeZlibInflateRawData()`.

Because the pointer passed to `edgeZlibAddInflateQueueElement()` is then passed to the Inflate Task and processed by `edgeZlibInflateRawData()` or `edgeZlibFetchAndInflateRawData()`, the same requirement applies. Only a pointer to the raw data should be passed to `edgeZlibAddInflateQueueElement()`. Your program needs to deal with the header before calling this function.

Segmented Versus Non-segmented Decompression

When decompressing data larger than the constraints of Local Store, there are two approaches that may be taken with Edge Zlib:

- The data can be segmented (see “[Segmented Decompression](#)” in the previous chapter), so that during decompression the SPU only needs to work on a small amount of data that fits in LS all at once (for example, input ≤ 64 KB, output ≤ 64 KB).

- The SPU can iterate over the data in main memory, fetching input data to LS and sending output as necessary.

With 64 KB segments, the speed of segmented decompression on a single SPU versus non-segmented decompression is comparable (when timed from the PPU side, so including all transfer times). Which method is faster can differ depending on the input data. If maximum segment sizes are reduced to 2 KB, for example, then non-segmented decompression is noticeably faster.

`sample/edge/zlib-large-segmented-inflate-sample` and `samples/edge/zlib-large-unsegmented-inflate-sample` show examples of these two methods of decompression.

Similar techniques apply to Edge Zlib compression on SPU.

Building Edge Zlib

When Edge Zlib's SPU library is built, either via Makefiles or via VSI, the following is passed on the command line to the compiler:

```
-DMAX_WBITS=12 -DMAX_MEM_LEVEL=6 -DDEF_WBITS=15
```

These defines instruct Edge Zlib to use smaller buffers during compression so that they can fit in LS, while also allowing larger buffers for decompression so that Zlib is compatible with data built off-line. If you use these constants in your code, make sure they are appropriately in sync.

Edge Zlib's SPU library is always built with optimizations on, even in debug mode. This is because with optimizations off, the code can be too large and Local Store could run out of space.

14 The Specifics of Edge LZMA

Characteristics

LZMA is the lossless data compression algorithm used by 7-zip. This release contains Edge LZMA, which provides a modified and optimized implementation of LZMA decompression for running on the PlayStation®3 SPU.

Edge LZMA currently only offers an SPU implementation of the decompression functionality. Appropriate data for decompressing can be generated offline using a tool such as 7-zip, or by using a library for compression such as the LZMA SDK (available from www.7-zip.org/sdk.html).

Differences Between Edge LZMA and LZMA

Edge LZMA provides the following features, not found in LZMA:

- Optimized for SPU instruction set and Local Store.
- The probabilities buffer is static data, rather than being dynamically allocated on SPU.
- A SPURS task is provided that allows the segments for decompression to be queued up from the PPU.
- Decompression typically works in 64 KB segments. All input data must be in Local Store before the decompression runs and there must be enough space in Local Store for the output buffer.
- Decompression of separate segments can be run in parallel on multiple SPU's at once.

Headers on the Compressed Data

Edge LZMA expects to receive a pointer to the "Properties" as well as a pointer to the raw compressed data stream.

When data is compressed with `lzma.exe` or the LZMA SDK, the format it comes out in starts with 5 bytes of "Properties" data, then 8 bytes that provide the uncompressed size, and then after this header it is followed by the raw compressed data stream. If you use this data format, you should parse this data and split it into its components before passing it into Edge LZMA.

Similarly, if you use some other file format (such as `.7z`), you should parse the data and split it into its components so that you have the "Properties" and the raw data stream separately for passing into Edge LZMA.

These comments apply to the low-level function `edgeLzmaInflateRawData()` as well as to the following high-level functions:

```
edgeLzmaAddInflateQueueElement(), edgeLzmaTryAddInflateQueueElement(),
edgeLzmaAddInflateQueueElementPartialCopyOut(),
edgeLzmaTryAddInflateQueueElementPartialCopyOut().
```

The Inflate Tasks

The Inflate Tasks for Edge LZMA work exactly the same as for Edge Zlib. The only thing to be aware of is that since Edge LZMA does not support decompression larger than can fit in LS, this puts a limit on the sizes of the buffers handled by the Inflate Tasks. The input compressed data must not be more than 64 KB, and the output uncompressed data must also be not more than 64 KB. If either buffer is larger than this, then the SPU code will assert.

15 The Specifics of Edge LZO

Characteristics

LZO is a lossless data compression algorithm that is focused on speed rather than size. Edge LZO provides a modified and optimized implementation of LZO compression and decompression for running on the PlayStation®3 SPU.

LZO provides many different algorithms. Edge LZO has focused on implementing just LZO1X. Edge LZO can decompress any LZO1X data. Appropriate data for decompressing may be generated offline using the LZO SDK (available from www.oberhumer.com/opensource/lzo).

Differences Between Edge LZO and LZO

- Optimized for SPU instruction set and Local Store.
- A SPURS Task is provided that allows the segments for decompression and compression to be queued up from the PPU.
- Compression and decompression typically works in 64 KB segments. All input data must be in Local Store before the compression/decompression runs and there must be enough space in Local Store for the output buffer.
- Compression or decompression of separate segments can be run in parallel on multiple SPUs at once.
- Only one of the many LZO algorithms is implemented.

Licensing

When using Edge LZO in your application, such as middleware or a game title, you must display “Data compression by oberhumer.com” in your application’s in-game ending credits section/file (where applicable). You must also provide this text in your application’s documentation, such as any instruction manuals.

Headers on the Compressed Data

Edge LZO, like Edge Zlib and Edge LZMA, expects to receive a pointer to the raw compressed data stream. Any file header or footers should be parsed off before passing in to Edge LZO.

These comments apply to the low-level functions `edgeLzo1xInflateRawData()` and `edgeLzo1xDeflateRawData()` as well as to the following high-level functions:
`edgeLzo1xAddInflateQueueElement()`, `edgeLzo1xTryAddInflateQueueElement()`,
`edgeLzo1xAddInflateQueueElementPartialCopyOut()`,
`edgeLzo1xTryAddInflateQueueElementPartialCopyOut()`,
`edgeLzo1xAddDeflateQueueElement()`, `edgeLzo1xTryAddDeflateQueueElement()`.

The Inflate and Deflate Tasks

The Inflate and Deflate Tasks for Edge LZO work exactly the same as for Edge Zlib. Be aware, however, that the sizes of the buffers handled by the Inflate and Deflate Tasks are limited because Edge LZO does not support compression or decompression larger than can fit in Local Store. In particular, neither the input data nor the output data must be more than 64 KB. If either buffer is larger than this, the SPU code will assert.

Note: If you modify the SPU code, depending on the change, the minimal context yield by the Inflate Task or the Deflate Task might not be enough and a larger context area might need to be stored.

Compression Output Buffers in Local Store

Note: The following issue only affects `edgeLzo1xDeflateRawData()`. If your application uses the Deflate Tasks, then this issue is already handled for you.

When performing compression, LZO, and thus Edge LZO, does not check for over-writing the end of the output buffer and assumes it has enough space. Therefore, the output buffer in Local Store should be large enough to contain whatever the maximum possible size is for the output compressed data. The worst-case output block size after compression is given by:

$$\text{output_block_size} = \text{input_block_size} + (\text{input_block_size} / 16) + 64 + 3$$

So, for example, if an input buffer of 65,536 bytes is being passed in for compression, the output buffer in Local Store must be able to receive at least 69,699 bytes. Obviously, it is ideal that the output size will in fact be smaller than the 64 KB of input data, but it is important to assume the worst-case expansion.

16 Overview of Edge DXT

Introduction

Edge DXT is an SPU library that primarily allows for compression of raw image data to the CELL_GCM_TEXTURE_COMPRESSED_DXT1, CELL_GCM_TEXTURE_COMPRESSED_DXT23 and CELL_GCM_TEXTURE_COMPRESSED_DXT45 texture formats.

The library also provides functions to decompress these formats back into raw image data.

Edge DXT Design

The compression algorithms have been designed with the goal of high-compression speed. As such, many shortcuts are taken during compression. While the quality of the compressed results should be acceptable for most images, it will not be as good as images compressed offline.

Samples use SPURS job and you can copy this code as a starting point to integrate Edge DXT in your own project.

- Code is compiled position-independent, but can be used in a fixed position model.
- There are no system calls or DMAs inside the library. It can also be used on raw SPUs.

Compression Performance and Restrictions

The decompression functions in Edge DXT can handle all valid DXT data. The following restrictions apply to compression only.

DXT1

To maintain good compression performance when alpha is not required, two functions for DXT1 compression are provided.

The function `edgeDxtCompress1()` ignores the source alpha channel, producing a fully opaque output image. The compression inner loop for this function schedules to 101 cycles per DXT block.

The function `edgeDxtCompress1a()` takes source alpha into account and produces an image with 1-bit alpha. The compression inner loop for this function schedules to 120 cycles per DXT block.

DXT3

The compression inner loop schedules to 102 cycles per DXT block.

DXT5

Only the 8-value alpha block encoding is used.

The compression inner loop schedules to 126 cycles per DXT block.

17 Using Edge DXT

Compression

To achieve efficient compression, the compression functions in Edge DXT compress a contiguous row of blocks per call.

The input data must be 8-bits per channel in ARGB byte order, identical to a surface of type `CELL_GCM_SURFACE_A8R8G8B8`. Four rows of image data are required to form a row of DXT blocks. Each row must start on a 16-byte boundary, and they must be separated by constant stride.

To compress the data, call `edgeDxtCompress1()`, `edgeDxtCompress1a()`, `edgeDxtCompress3()` or `edgeDxtCompress5()` with the LS location to store the output blocks, the LS location of the first row, the stride between rows and the number of DXT blocks to produce.

For DXT1, the block output location must be 8-byte aligned. For DXT3 or DXT5, the block output location must be 16-byte aligned. If necessary, you can safely compress the data in-place – to do this, pass the same LS location for both the output blocks and the first row.

Decompression

To achieve efficient decompression, the decompression functions in Edge DXT decompress a contiguous row of blocks per call.

The decompression functions mirror the compression functions for data layout and have the same alignment restrictions.

Note: In general it is *not* safe to decompress the data in-place.

18 Overview of Edge Post

Characteristics

Edge Post can help you write SPU implementations of frame buffer post-processing effect chains and general image transformations. The library provides a framework to facilitate this, and a set of predefined effects to demonstrate example usage of the system. These predefined effects are meant to be a starting point for developing customized post-processing effects or as samples for porting their current post-processing effects from RSX™ to an SPU implementation.

Examples of effects that can be achieved using Edge Post are Motion Blur, Depth of Field, Bloom, Tone Mapping, and so forth. The provided samples demonstrate combining Motion Blur, Depth of Field, and a simple Bloom, using either LDR colors or HDR colors.

Due to the limited size of the SPU local store, processing happens per tile; tile size is chosen in advance on PPU (either manually or by using library helper functions) and depends on factors such as input image sizes, border width required for each tile, output image size, and effect code size. DMA lists are heavily used to bring tiles in and out of SPU local store.

You are responsible for writing the actual code that performs pixel processing. The way this code is provided is user specific; the library simply expects the user to provide a callback to a function that will process one tile.

The sample uses a system where code is packaged as an Edge Post Job, which basically means that code is compiled position independent and linked using a custom linker script. It is then DMAed on demand depending on the specific effect to be performed. This is done to keep overall code size to a minimum.

You can choose to use different methods – for example, SPU DLLs, overlays, or simply linking the effect code directly with the library.

The SPU tiling framework will manage double-buffered DMA input and output of tiles, synchronization with other SPUs processing the same workload, and (optionally) synchronization with RSX™. For each tile ready to be processed, the framework will call the supplied callback for specific processing.

SPU Usage

The sample uses a trivial SPURS Task, which can be copied and used as a starting point to integrate Edge Post into your project, but the library itself makes no assumptions about any SPU programming model. SPU usage details:

- Code is compiled as position independent (PIC, compiled with the `-fpic` option).
- All DMA operation are interrupt safe.
- There are no SPURS function calls inside the library.

Data Structures

The fundamental data structure is `EdgePostProcessStage`. It contains all the information required by the Edge Post tiling framework to perform an operation and can take up to a maximum of four input/output images. The input to Edge Post is an array of `EdgePostProcessStage`, and this array describes a complete “chain of effects”.

`EdgePostProcessStage` contains a large amount of information and is quite complex; however, a few PPU functions are provided to help populate its fields. These functions will try to select suitable tile sizes based on input image sizes.

`EdgePostWorkload` provides a secondary data structure that is opaque to the user. It is 16-byte sized and 16-byte aligned. It is always accessed atomically by SPUs. This structure is initialized with a pointer to

an array of `EdgePostProcessStage`, and its effective address is passed as a parameter to the main SPU Edge Post function.

Fields of this structure are used to synchronize different SPUs working on the same chain of effects.

19 Using Edge Post

Basic Procedure

(1) Create an effect chain

Create a sequential list of effects that you want to apply to a set of input images. This is achieved by creating an array of `EdgePostProcessStage` items. Each element of this array defines its own set of inputs, its output, and the operation plus operation parameters. Some PPU functions are provided in order to help fill the `EdgePostProcessStage` structure with correct data, or the structure can be filled manually.

(2) Create a workload

Declare an `EdgePostWorkload` and initialize it by calling `edgePostInitializeWorkload()` with a pointer to the effect chain.

(3) Per SPU initialization

In your SPU code (for example your own SPURS Task) initialize the Edge Post library by calling `edgePostSetSpuConfig()`. You need to pass an `EdgePostSpuConfig` structure to this function in order to tell Edge Post the size of its working area, as well as which DMA tags it is allowed to use.

(4) Start processing on SPU

Call `edgePostRunWorkload` from SPU to start (or join) execution of a workload, passing the effective address of the workload structure as a parameter.

(5) Wait for termination

On PPU you can wait for termination of a workload by calling `edgePostStallForWorkload()`.

20 Edge Post Runtime

Processing Stage and the Effect Chain

Each processing stage is described by the following information:

- Tile subdivision (`numTileX` x `numTileY`)
- Up to four tiled image descriptors:
 - Image pitch and effective address
 - Tile width x height x bpp
 - Tile border size
- Effective address of effect code
- Address and size of effect-specific parameters
- Some user data fields
- An optional RSX™ label address to be updated when the stage has finished processing

All of the information required to process a stage is contained in the `EdgePostProcessStage` structure. By creating an array of `EdgePostProcessStage` items you are effectively defining a chain of effects that need to be run in the specified sequential order.

This effect chain can be created at startup (as the samples do) or each frame to allow for dynamic variations to the chain of effects.

Once an effect chain is in place, it can be executed by creating an `EdgePostWorkload` structure and initializing it with the pointer to the effect chain; then its effective address must be passed to the SPU function `edgePostRunWorkload`.

When an effect chain is being executed (by one or more SPU) the PPU can check for termination by calling the function `edgePostIsWorkloadFinished()`, or it can stall for completion by calling `edgePostStallForWorkload()`.

Tile Size Selection

For each processing stage the tile framework needs to allocate in local store enough space to fit all image data plus the space for the DMA lists used to bring the tiles to/from main memory, and optionally the space for the effect code itself plus its own parameters.

As a consequence of this, tile size needs to be selected carefully to ensure that enough free space in local store is available. The library supplies a PPU helper function to do this. This function scans a set of tile candidates looking for a suitable match. It checks for the following:

- Total memory required is less than or equal to the amount of space available.
- Tile dimensions should be a perfect divisor of the image size.
- Calculated tile pitch should be 16-byte aligned for DMA transfer.
- Number of X and Y tiles should be a perfect divisor of image dimensions.

You do not need to use the provided function for tile size selection as long as the above requirements are all met. The SPU library will assert if it cannot fit all the required data in Local Store.

Each tile descriptor has an additional property called `multiplier` that can be used to alter the space required to store an input tile. This is useful when an input tile has a pixel format that you want to convert to a different format prior to usage.

For example, it is common to have images stored as ARGB8, because this is a common RSX™ frame buffer format, but processing has to be performed using floating-point numbers per channel. It is possible in this case to specify a `multiplier` factor of 4. The input tile will still be treated as ARGB8 in terms of DMA

size but the allocated memory for it is four times bigger and it is now possible to convert the tile (in place) to floating points.

This technique is exemplified in post-sample, specifically within the motion blur and depth of field code.

SPU Memory Layout

During each stage of processing, SPU Memory Layout is as follows:

- Effect code, 16-byte aligned
- Effect parameters, 16-byte aligned
- For each tile
 - Tile pixel data
 - Tile input DMA list

Unless a stage operates on an image that has been subdivided into only one tile, each pixel data area will be twice the size of the real tile size, to account for double buffering.

Such memory layout will be recalculated each time execution of a new stage begins.

SPU Synchronization

The synchronization mechanics that allow different SPU's to work on the same effect chain are hidden to the user but a few guidelines need to be followed:

- Workload initialization is not a one-off; `edgePostInitializeWorkload` must be called each time before the workload can be started – for example, once a frame. Once a workload has been fully executed `edgePostInitializeWorkload` needs to be called again.
- Do not call `edgePostInitializeWorkload` on a running workload.
- After a workload has been initialized, you can call `edgePostRunWorkload` on as many SPU's as you want. Each SPU will start working at the correct stage of processing, effectively joining the work. Similarly, any SPU can leave at any time if higher priority work is in the queue (and rejoin later).
- After a workload has ended, calling `edgePostRunWorkload` will have no effect.

RSX™ Synchronization

You may need to synchronize with the RSX™ in two different situations:

- Edge Post needs to be “kicked” by RSX™.
- RSX™ is waiting for some Edge Post outputs.

The second case is much simpler and will be discussed first.

A simple mechanism for synchronization with RSX™ is provided: each processing stage can optionally write a specified value into an RSX™ label as soon as the processing stage is ended.

For example, the last stage of an effect chain can write a value that is incremented each frame into a fixed RSX™ label address to signal that frame buffer post-processing has finished. The RSX™ would then pick up the result of Edge Post and continue with its work (for example, drawing the HUD and presenting the buffer to screen). This method is exemplified in the Edge Post sample.

There is no obligation to use this mechanism to synchronize with the RSX™, because Edge Post is a library and the user is in control of the main SPU program. Other methods can be applied – for example, the SPU program could clear a jump to self or directly move the RSX™ put pointer.

For the first case the library does not supply any “built-in” solutions because this can be very specific to each system.

A few examples are given:

1. Users can call `cellGcmSetUserCommand` to trigger an RSX™ callback on the PPU; from there start a SPURS task that performs Edge Post work.
2. Users can build Edge Post into a SPURS Job and use a JTS (a jump to self inside the job chain), which is cleared by the RSX™ with an inline transfer.
3. Users can build Edge Post into a custom Policy Module that does specific synchronization with RSX™.

Additional care must be taken if Edge Post needs to access pixel data from Host Memory that has been transferred by RSX™. In particular, make sure that pixel data has been fully written to Host Memory, because there are not any ways to do this from the RSX™ point of view if you are using 2D blits. More information on the subject can be found in the [“Edge Post Integration and Suggested Usage”](#) section later in this chapter.

PPU Synchronization

A method of synchronization with PPU (that is, the PPU waits for Edge Post to finish) is provided that is based on a busy wait loop. Look at `edgePostStallForWorkload` and `edgePostIsWorkloadFinished` for more information.

The busy wait method of synchronization may not be optimal in many situations, and better synchronization methods based on Signal or Mutex can be implemented directly by the user.

Because Edge Post does not depend on any OS facilities (such as a Mutex) such features have been deemed outside the scope of the library.

Synchronization with PPU is also much less relevant in the case of Edge Post, where synchronization with RSX™ is far more critical.

SPU Pixel-Processing Functionality Provided Within the Library

The Edge Post SPU library provides a set of useful functions for pixel processing that can be called from inside your effect code.

Examples are conversion between packed and floating point pixel formats, conversion from RSX™ depth buffer format to a floating point representation, simple implementations of Motion Blur, Depth of Field, and so forth. Read the Reference Manual for more detailed descriptions of these functions.

These function are written in assembly for performance reasons; some have C implementations for reference only (Motion Blur and Depth of Field).

In order to achieve good efficiency, each function in Edge Post works on rows of pixels at a time. Most of them will process more than one pixel per loop. For this reason, the row size is usually expected to be in multiples of 4 or 8 pixels; input and output pointers should always be aligned to 16 bytes.

In general, it is safe to feed the same pointer as an input and as an output to a function (to achieve in-place pixel processing) only if each pixel can be computed independently from each other. Many operations require to access a group of nearby pixels to compute one output pixel. Examples include Gaussian Filters, Motion blur, and Depth of Field. It is generally not safe to feed such functions the same input and output pointers.

When porting an existing post-processing pipeline from RSX™ to an SPU implementation, it is expected that every team will have custom fragment shaders for each effect, custom layout of information within their frame buffer (such as motion vectors), and many other subtleties. All of those are very difficult to generalize in a library. Because there is no standard solution to post-processing effects, it is not possible to provide a solution valid for everyone.

One way to mimic this programmable behavior on SPU is to not provide a fixed stage effect pipeline, but to instead provide teams with the ability to write their own “tile processing code”, similar to how they would write a fragment shader for each specific effect.

Seen from this point of view, the pixel-processing functions provided within the library are meant to be more of an example and a starting point for teams willing to:

- Port existing post-processing effect shaders which are different from the reference implementation of the library
- Create new effects

All `edgePostXXX` pixel-processing functions do not depend on any external libraries or on the tiling framework. Teams with a system in place to handle DMA in/out of pixel data can use Edge Post pixel-processing functions with no problems or dependencies on the rest of the library.

Writing Effect Code

Users of the library are in control when it comes to writing the actual effect code.

In general you will have to supply a function which follows this prototype:

```
void MyFunctionName( EdgePostTileInfo* tileInfo);
```

This function processes exactly one tile, which is the atomic unit of work. `EdgePostTileInfo` contains pointers to input and output tiles, which are filled in prior to calling this function; it also contains additional information regarding the current tile being processed, the current stage of the effect chain being processed, and a pointer to optional effect parameters.

Users need to supply one such function for each stage of an effect chain. Where this function is located is completely irrelevant to Edge Post; for example the Edge Post sample loads the function from XDR into Local Store on demand. You could choose to have the function linked in the SPURS task or have it inside an SPU DLL.

Edge Post hooks into user code by means of two specific callbacks. Pointers to such callbacks must be initialized inside the `EdgePostSpuConfig` structure.

This structure is defined as follows:

```
struct EdgePostSpuConfig
{
    void* heapStart;
    uint32_t heapSize;
    uint16_t controlDmaTag;
    uint16_t inputDmaTag;
    uint16_t outputDmaTag;
    EdgePostPollCallback pollCallback;
    EdgePostStageEnterCallback stageEnterCallback;
    EdgePostStageExitCallback stageExitCallback;
};
```

`heapStart` and `heapSize` define an area of memory the user is handing over to Edge Post. This means the library does not need to call any memory allocation function on SPU, because all the memory it needs will come from this pool.

`controlDmaTag`, `inputDmaTag` and `outputDmaTag` are DMA tags that Edge Post can use for DMA transfers.

`pollCallback` is a callback function that Edge Post uses to check if some higher priority work is requesting use of the current SPU; it is defined with the same prototype as `cellSpursTaskPoll` for simplicity.

Finally, `stageEnterCallback` and `stageExitCallback` are the most important callback functions; they are called once when the SPU starts processing on a new stage and when the SPU abandons processing on the current stage or when the current stage ends. `stageEnterCallback` will return the

pointer to the user supplied per tile callback function. Usage examples: `stageEnterCallback` could DMA, into Local Store, a piece of code containing the effect code and then return the pointer to it, or it could load an SPU DLL, query the function address via `cellSpudllSym`, and return the address of the function. `stageEnterCallback` could also output SPURS Trace information to aid performance monitoring.

Edge Post can also load binary data for you before this function is called so that you can avoid stalling for a DMA load inside the callback. For example, if you want to load dynamic code, the information on whether to load and from where is stored inside the `EdgePostProcessStage` structure.

The Edge Post samples exploit this system by using a combination of custom build steps and a linker script to generate small SPU binaries, which can be loaded position independently and executed.

The following is an “Edge Post Job” as it is used in the sample :

```
extern "C"
void edgePostMain( EdgePostTileInfo* tileInfo)
{
    EdgePostProcessStage* stage = tileInfo->stage;
    uint32_t tileSize = stage->sources[0].height *
                        stage->sources[0].width;
    uint8_t* src_address = tileInfo->tiles[1];
    uint8_t* dst_address = tileInfo->tiles[0];
    memcpy( dst_address, src_address, tileSize * 4);
}
```

`edgePostMain` is the entry function of the job. For each tile to be processed, Edge Post will call this function. This example does nothing but copy the first input tile into the output tile.

Here is an example of how to use the linker script, `edgepost_job.ld`, in samples to build the job and create a PPU object file that you can link with your main executable:

```
% spu-lv2-gcc -fpic -c job.c -o job.o
% spu-lv2-g++ -fpic -nostartfiles -Ttext=0x0 -Wl,-Tedgepost_job.ld job.o -o
job.elf
% spu_elf-to-ppu_obj job.elf job.ppu.o
```

Then you should be able to access your job in your PPU code :

```
extern EdgePostJobHeader _binary_spu_job_bin_start;
```

Supported Pixel Formats

Edge Post supports the pixel formats shown in Table 37.

Table 37 Supported Pixel Formats for Edge Post

| Format | Description |
|---------|---|
| argb8 | 32-bit size, compatible with CELL_GCM_A8R8G8B8. |
| float | Single-channel floating-point image. Can be used to store depth values in a more SPU friendly way than, for example, D24S8 RSX™ format. A function is provided to convert from an RSX™ depth buffer into this format. |
| float4 | Full-precision floating points; four channel argb images. |
| fx16 | Four channels; each channel is 16-bit fixed-point 0:5:11. |
| Luv | CIE Luv color space. L is represented as a 16-bit fixed point 0:5:11 integer; U and V are normalized 8-bit values. |
| FP16Luv | CIE Luv color space. L is represented as a 16-bit, half-precision floating-point number; U and V are normalized 8-bit values. |

| Format | Description |
|--------|--|
| LogLuv | CIE Luv color space. L is stored as an logarithm in 8:8 fixed point format; U and V are normalized 8-bit values. |
| FP16 | Four channels; each channel is a half-precision floating point stored in 16 bit. |

Many conversion functions between formats are supplied.

FP16, FP16Luv, and LogLuv are supplied to facilitate the exchange of HDR colors between RSX™ and SPU. Each format has some advantages and disadvantages. For example, FP16 is the only HDR format that can be correctly filtered/down-sampled/up-sampled by the RSX™, but it uses 8 bytes per pixel.

FP16Luv and LogLuv are 4 bytes per pixels but cannot be filtered. Moreover, LogLuv requires expensive Log/Exp operations on SPU.

Luv is provided as a format to store intermediate HDR colors between SPU stages. It is faster to convert to/from floating point than FP16Luv and LogLuv, while allowing for low memory usage.

Float4 and fx16 are the more suitable formats for doing the actual pixel processing because they are high-precision and fast for sampling operations. Fx16 has the benefit of being 8 bytes per pixel, allowing bigger tiles to fit in Local Store.

It should also be fairly easy for users to add their own pixel formats – for example a 11:11:10 32-bit RGB format.

Edge Post Integration and Suggested Usage

Although Edge Post can be used for more than frame buffer post-processing, that is its main purpose. The main reason for such a choice is to free as much RSX™ time as possible to be used for other rendering tasks.

Unfortunately, frame buffer post-processing presents a class of problems very different from geometry processing, where SPUs excel. The RSX™ would be much faster when processing raw pixels. What the RSX™ can do in 40% of a 60hz frame, one SPU will take around 150% to do! Of course you have more than one SPU.

Triple buffering may be necessary to hide such a big latency. When triple buffering you have three concurrent operations during each frame:

- PPU/SPU generates command buffer for frame N-2.
- RSX™ renders frame N-1 but displays frame N.
- SPU post-processes frame N.

There are disadvantages related to triple buffering that need to be considered. The main ones are:

- Input latency. The player perceived reaction time may be too high. This is a design decision more than a technical problem.
- More memory must be used to hold pixel data. At least three color buffers are needed, and many more in an environment using MRTs or if you need to keep your depth buffer when doing post-processing.

Another important factor that you need to carefully consider is render target location and how to present input data to Edge Post.

SPUs will need to read input pixel data from main memory because the cost of reading from RSX™ local memory is far too high. It is also likely that SPUs will write their results into main memory and have the RSX™ pull them into local memory, because the PPU/SPU write speed to local memory is still lower than the RSX™ pull speed.

RSX™ will need to present pixel data in main memory as input to SPU. There are several ways to achieve this, and although Edge Post does not care how the data ends up in main memory, this is a very important factor to consider.

The following is a brief list of possible methods to provide Edge Post with frame buffer data in main memory:

- Create render targets in main memory and render directly there.
- Render to RSX™ local memory and use a 3D blit to transfer the pixel (“draw a big quad” method).
- Render to RSX™ local memory and use a 2D blit to transfer the pixels (`cellGcmSetTransferImage`).

The first solution (rendering directly to main memory) is very problematic because RSX™ bandwidth is lower than rendering to local memory; moreover, color compression is not supported (even on tiled targets).

The second solution is comparable to the third only if the destination of the blit is tiled memory, which creates additional problems:

- An additional tiled region is needed in main memory simply for the purpose of feeding data to Edge Post.
- SPU must de-tile pixel buffers and output results into another non-tiled surface.

In practice, a 3D blit into tiled main memory is comparable in terms of performance to a 2D blit into linear main memory. However, doing a 3D blit into a linear (non-tiled) surface in main memory can be up to five times slower than doing a 2D blit using `cellGcmSetTransferImage`.

Given these results, the easiest and most effective method seems to be the third, which does a simple 2D blit into linear memory. The big advantage of this method is that you don’t need to de-tile the memory on SPU prior to usage, saving the time for the process and the space for even more temporary buffers.

`cellGcmSetTransferScaleSurface` can also be used effectively here to fold into the copy operation an initial down-sampling of the image, with the added benefit of a faster copy operation, or a final up-sample and copy of a frame buffer from main memory to local memory.

As an added bonus, `cellGcmSetTransferScaleSurface` can use point sampling as well as bilinear sampling when up-sampling and down-sampling.

Table 38 shows some test results of copying a 1280x720 image from tiled local memory to main memory using different methods:

**Table 38 Copying a 1280x720 Image from Tiled Local Memory to Main Memory:
Sample Results from Tests**

| Method Used to Transfer Data from Local Memory | Cost in Milliseconds |
|--|----------------------|
| 3D quad to tiled host memory | ~0.55 ms |
| 3D quad to linear host memory | ~2.75 ms |
| 2D blit to tiled host memory | ~0.45 ms |
| 2D blit to linear host memory | ~0.46 ms |

Differences Between RSX™ and SPU Implemented Post-processing Effects

Although you can use bilinear texture filtering nearly for free on RSX™, on SPU it is very expensive. For this reason, there is no function in Edge Post that uses or implements a bilinear filter.

Even a simple but correct “nearest filtering” can be quite expensive on SPU because it would involve a costly `floorf` function to calculate correct pixel coordinates from (negative) floating-point UV coordinates. For example, if relative UV coordinates are equal to (-2.5, 0), the RSX™ will sample the pixel at integer coordinate (-3, 0) when doing a nearest-texture lookup. This is the result of taking the floor of the UV coordinate and then converting it to integers. A simple generic floor operation on SPU can be executed in ~4 instructions, which can add up to ~6 cycles per sample taken.

In theory, this would not be a problem if UVs were always positive floating-point numbers, but in Edge Post, negative UVs are very frequent because UVs are always relative to the current pixel being processed.

If you know your limitations, you can get away without the floor altogether and just use a simple `cflts` instruction without visible quality degradation (although with slightly different results on screen). In the example above, UV coordinate (-2.5,0) will sample the pixel at (-2,0) from the current center pixel. It is very difficult to notice the difference in terms of quality, when doing a motion blur for example.

In many cases you already know which pixels you want to sample, so you do not need to have proper UV coordinates; you simply need integer offsets. This is the case for gauss filters.

Random access to textures is another difference between RSX™ and SPU that needs to be considered.

Most of the effects are, or can be, reduced to simple image filters where filter samples are distributed inside a small circle of variable radius centered at the current pixel coordinate. When this is the case (for example, gauss filters, most simple filters, depth of field, motion blur) the SPU implementation needs to bring in to Local Store more input pixels than the number of output pixels, but this is generally not a problem.

This “overfetch” of pixels will be located along the edges of each tile. This is to allow, for example, the pixel at tile coordinate (0,0) to access pixel at tile coordinate (-1,0). A tile border dimension can be specified with each tile, and its upper limit is limited only by the size of the Local Store.

When the above assumption is not valid any more (filter samples are randomly distributed on one or more textures), an SPU implementation becomes much more difficult because the only solution for texturing would be random DMA from main memory. Even using a software-cache it would probably not be the best way to use your SPU resources.

For these reasons, having an SPU implementation that perfectly matches your RSX™ implementation can be quite a challenging task.

Performance

Full frame operations (1280x720) are very expensive, so always try to work on lower-resolution images when possible. Try to fold initial down-sample and/or final up-sample operations into the RSX™ copy; a relatively simple up-sample from 640x360 to 1280x720 can take up to 2 ms on one SPU (although it will parallelize onto more SPUs, with no problems).

Table 39 provides a breakdown of Edge Post sample SPU utilization taken with SN Tuner. Note that in this case Edge Post has been constrained to run on one SPU only. Parallelized performance will scale almost linearly.

Table 39 Sample SPU Utilization by Operation (640x360 Image) – Cost in Milliseconds

| Operation | Cost in Milliseconds on One SPU |
|---------------------------------|---------------------------------|
| DOF Fuzziness | 0.9 ms |
| Depth Of Field | 5.7 ms |
| Motion Blur | 4.7 ms |
| Gauss Filters | 1.9 ms |
| Bloom | 0.2 ms |
| Internal Lens Reflection | 0.2 ms |
| Down-sampling | 0.38 ms |
| Up-sampling | 1.4 ms |
| ROP operations (such as blend) | 1.16 ms |

Table 40 provides a breakdown of average cycles per pixel for selected Edge Post Processing functions.

Table 40 Sample SPU Utilization by Function– Cost in Cycles per Pixel

| Function | Cycle Cost Estimate |
|----------------|---------------------|
| Motion Blur | 45 cycles per pixel |
| Depth of Field | 75 cycles per pixel |
| Bloom Capture | 10 cycles per pixel |
| Gaussian 7x1 | 12 cycles per pixel |
| Gaussian 1x7 | 4 cycles per pixel |

Table 41 shows a comparison between the intrinsic version and the SPA version of a Motion Blur function.

Table 41 Sample SPU Utilization for a Motion Blur Operation by Implementation Type (1280x720 Image) – Cost in Milliseconds

| Function (Intrinsic vs SPA) | Time to Process 1280x720 Image on One SPU |
|-----------------------------|---|
| Intrinsic Motion Blur | ~27.5 ms |
| SPA Motion Blur | ~17.6 ms |

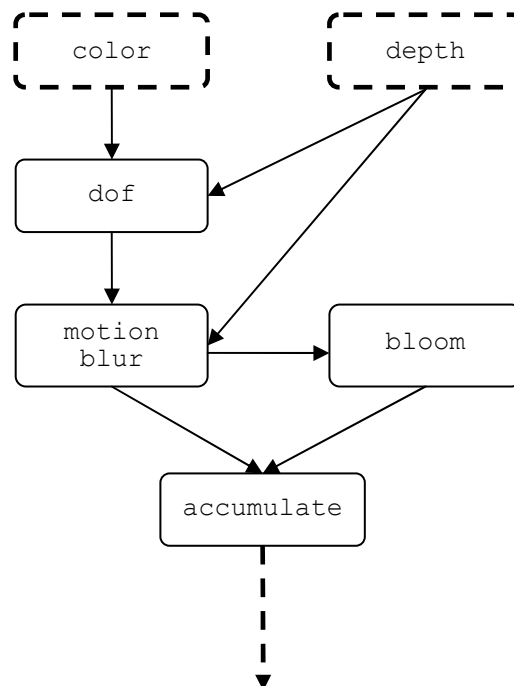
21 Edge Post Notes

Example Data Flow for Edge Post

This chapter provides a set of diagrams depicting the data flow associated with using Edge Post for post-processing as it is implemented in `post-sample` (the low dynamic range example; for more information, see “`samples/edge/post-sample`”, “`samples/edge/post-sample-hdr`”, and “`samples/edge/post-sample-mlaa`”, found in the “[Edge Post](#)” section of Chapter 23, “[Sample Programs](#)”).

Figure 3 shows the macro-visualization of the post-processing chain implemented inside `post-sample`.

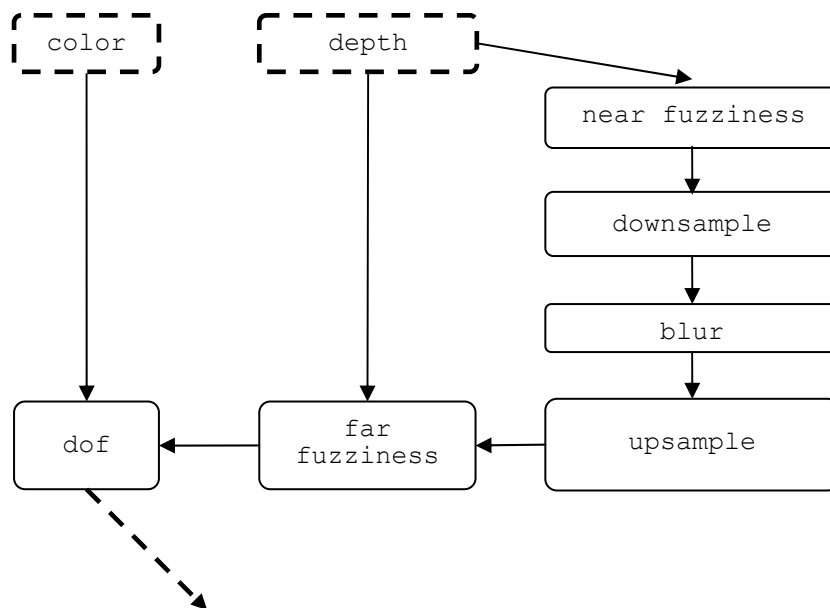
Figure 3 Macro Data Flow



In short, two inputs (color and depth/motion vectors) are provided by the RSX™. Depth of Field (`dof`) and Motion Blur are applied in series. The output of the Motion Blur is input to the Bloom, and the final result is a composition of the Motion Blur result and the Bloom result.

Figure 4 shows the data-flow details for the Depth of Field module.

Figure 4 The “Depth of Field” Component’s Data Flow



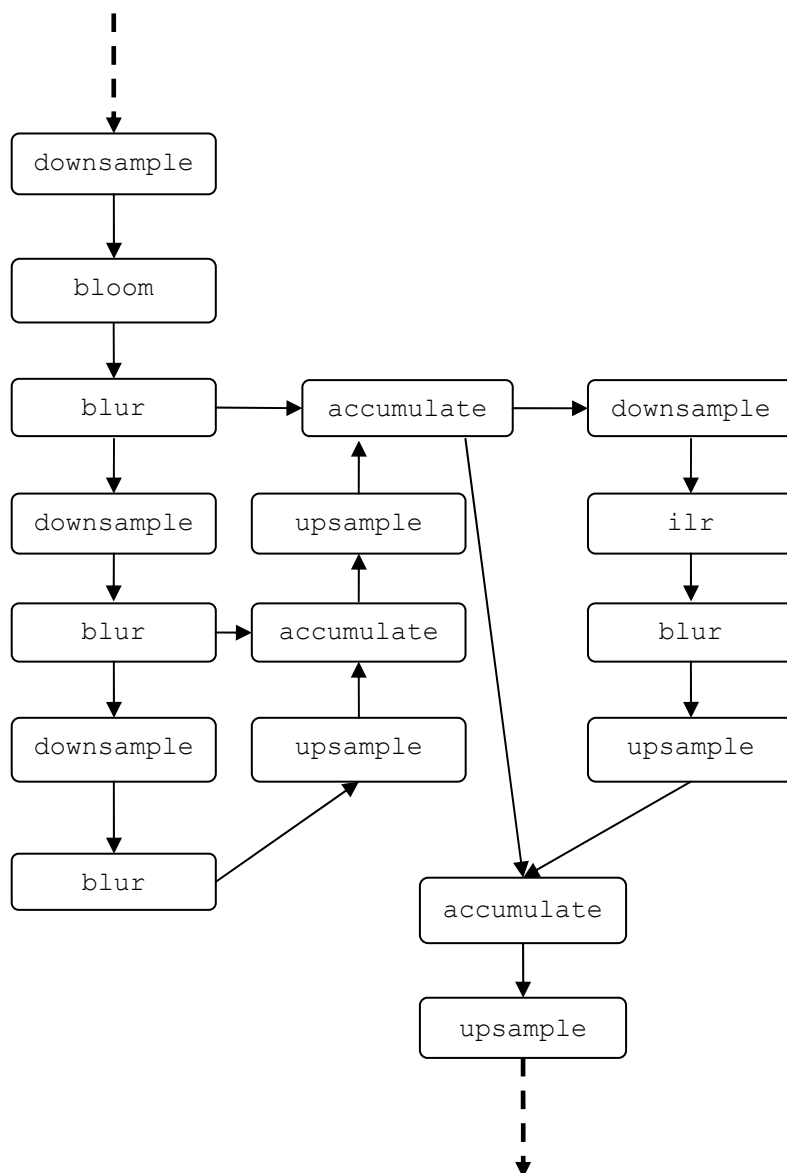
The depth buffer is transformed into a near fuzziness buffer, which is in turn down-sampled and blurred. The blurring of the near fuzziness buffer allows for smoother edges for out-of-focus foreground objects on top of an in-focus background.

A final fuzziness buffer that contains far and near fuzziness is then generated and fed into the Depth of Field module together with the original color buffer.

The output is an ARGB 32-bit image where alpha is primed with the pixel fuzziness.

Figure 5 shows the data-flow details for the Bloom module.

Figure 5 The “Bloom” Component’s Data Flow



The output of motion blur is first down-sampled and then transformed into bloom colors. This buffer is then down-sampled and blurred several times. Three differently sized bloom buffers are then accumulated together, and the result is down-sampled again to be input to the Internal Lens Reflection (ILR) module.

Finally, the ILR output and the Bloom output are added together, and the up-sampled result will be composited with the output from the Motion Blur module.

22 Edge Post MLAA

Overview

The purpose of the Edge Post MLAA (Morphological Anti-aliasing) system is to provide an alternate method to anti-aliasing for XRGB8 images. The algorithm is purely image based and intended to run on a single-sample image. It will attempt to smooth all aliased lines found, including alpha-tested geometry and textures. Because it is working on single sample buffers, it does not have the information needed to compensate for sub-pixel movement.

Basic Procedure

- (1) Create an `EdgePostMlaaContext` on the PPU and initialize it by calling `edgePostMlaaInitializeContext()`. This is the only initialization.
- (2) Set up an MLAA process by calling `edgePostMlaaPrepareWithRelativeThreshold()`. All parameters related to the source and destination buffers are set here, so they can easily be changed without creating another MLAA context.
- (3) Write source image to XDR, for example through an RSX™ image transfer.
- (4) After the source image is available, call `edgePostMlaaKickTasks()` to start processing.
- (5) Wait for completion of the MLAA operation by calling `edgePostMlaaWait()`.
- (6) Use the resulting buffer.
- (7) Repeat steps 2 - 6 for the next frame if necessary.

Placing MLAA in a Frame

An important question is when to perform MLAA. In principle, placing the MLAA immediately before drawing the user interface will ensure that all visible edges are considered and might seem like the obvious choice. However, the algorithm works better if it is employed before any blurring effects like Bloom, Depth of Field, or Motion Blur. We expect the best visual quality on images that contain both opaque and transparent geometry and that are in the final brightness (that is, tone-mapping is applied). Because geometry information is not considered, even reflections, refractions, and shadow-edges will be processed. In most cases, this will make it difficult to find tasks for RSX™ to work on while MLAA is busy, unless frame processing is interleaved. Good candidates for work are shadow maps and depth-only passes of the following frame.

Edge Detection and Tuning

Both the performance and image quality depend on the edge detection algorithm. If too few edges are detected, aliasing remains visible. If too many edges are detected, the image will get blurred and noisy. The latter may be unexpected, but MLAA is not a blurring algorithm and will not apply a uniform smoothing to an image if too many edges are detected.

It is therefore important to ensure that the parameters of the edge detection system are adapted to your particular game. If different game levels have different levels of contrast, it may be beneficial to change the parameters between levels. However, be aware that changing them during gameplay may cause popping artifacts.

At present, the only edge detection algorithm provided uses thresholding on the absolute difference of the color-channels of adjacent pixels. It is controlled by two parameters, *base* and *scale*, of `edgePostMlaaPrepareWithRelativeThreshold()`.

Given a pixel p , the algorithm first computes a pixel threshold $t(p) := \max(\text{base}, \max(p.r, p.g, p.b) * \text{scale})$.

When testing if two pixels $p0$ and $p1$ form an edge, the maximum absolute difference $a(p0, p1) := \max(|p0.r - p1.r|, |p0.g - p1.g|, |p0.b - p1.b|)$ is computed and compared to the minimum of the pixel-thresholds of $p0$ and $p1$. The final predicate is $edge(p0, p1) := \min(t(p0), t(p1)) \leq a(p0, p1)$.

A few general guidelines:

- If high-luminance features seem overly blurred, increase the scale.
- If too few edges are detected, decrease the scale.
- If dark features are missed, decrease the base.
- If there is excessive bleeding between bright and dark features, try increasing the base. Most likely, the scale is too low, however.

The scale parameter is encoded as a 0.8 fixed-point number. We have had good results with a base of 0xA and a scale of 0x59. These should only be considered to be starting points.

Performance Characteristics

When processing a 720p image on 5 SPUs, the average total SPU time appears to be approximately 10 ms, with a range from approximately 7 ms to 14 ms. The code can run on any number of SPUs, but will not release an SPU fast enough to be integrated with the system workload. According to our measurements, the increase in total SPU time when going from 1 SPU to 5 SPUs is approximately 600 μ s.

As the image is simply split into strips of equal size, some tasks will idle while waiting for others to finish. During these times the SPUs are released and other work can be scheduled. The tasks of the MLAA workload do not need to be on physical SPUs at the same time, but this is recommended for optimal latency.

During two parts of about 400 μ s each, the system is capable of fully saturating XDR bandwidth, if used on more than 1 SPU. If this is an issue (and it most likely will not be an issue), the `EDGE_POST_MODE_SINGLE_SPU_TRANSPOSE` mode can be used to limit peak XDR load.

The algorithm is designed to work in-place and only requires one image buffer in XDR memory. Independently of whether it operates in-place or not, the target buffer must be large enough to contain the source image with its height increased to the nearest multiple of 128.

If performance is not an issue, the restriction can be relaxed by using the `EDGE_POST_MLAA_MODE_TRANSPOSE_64` mode, which works with multiples of 64. This should be considered only if the increased memory usage of the default mode is a severe issue, because it costs approximately 2 ms and may increase the load on the XDR memory system.

Samples

Edge Post MLAA is integrated into two samples: `post-sample` and `post-sample-mlaa` (see the “[Edge Post](#)” section of Chapter 23, “[Sample Programs](#)”). `samples/edge/post-sample` shows how MLAA performs in motion and how to integrate MLAA with a render loop and the rest of Edge Post. `samples/edge/post-sample-mlaa` performs MLAA on 720p textures and can be used to profile the performance on different test images.

Restrictions

- Both dimensions of the source image must be no more than 1280 pixels.
- The image width must be divisible by 128, or 64 if `EDGE_POST_MLAA_MODE_TRANSPOSE_64` is set.
- The target buffer must be able to contain an image with its height increased to the nearest multiple of 128, or 64 if `EDGE_POST_MLAA_MODE_TRANSPOSE_64` is set.
- Images must be 8-bit ARGB or XRGB.
- The contents of the alpha channel are undefined after the operation finishes. This can be controlled through an optional feature as described in the next section.

Optional Features

There are several optional parts to the MLAA system that can be used to ease development and/or change its behavior. These are controlled by several preprocessor constants and require the library to be rebuilt.

- **edgepost_mlaa.cpp: `ENABLE_DEBUG_FEATURES`.** Code is added that allows the visualization of the edges detected or the separation lines found. The visualization method is selected by defining `DEBUG_MODE` to be either `SHOW_EDGES` or `SHOW_SEPARATION_LINES`. In both modes, horizontal lines are shown in green and vertical lines in red, with pixels that contain both types being yellow. By default, edge visualization support is enabled because it has practically no performance impact and is very helpful for debugging. Separation-line visualization mode marks the start of a separation line with 0xff, inner pixels with 0x80, and the end pixel with 0xd0. Enabling support for `SHOW_SEPARATION_LINES` impacts performance.
- **edgepost_mlaa.cpp: `CLEAR_ALPHA`.** If it is undesirable that the alpha channel is in an undefined state after processing, this flag enables the clearing of the channel. It is not possible to preserve the alpha channel during processing. Clearing costs about 160µs and the value written is defined by `CLEAR_ALPHA_VALUE`.
- **edgepost_mlaa_blend.spa: `WRITE_BLENDED`.** Setting this to 1 will mark all pixels touched by the blend function red (for Z shapes) or green (for U shapes). This will look different from the edge visualization, because not all edges that were found actually get blended and because the blend code will touch more pixels than just those on an edge.
- **edgepost_mlaa_compress_seplines.spa: `BLEND_DISCARDED`.** Single pixel features are discarded at this point of the pipeline for performance reasons. To maintain image quality, especially of diagonals, they are blended directly here. On the other hand, this causes an artifact near the step of an edge because the step is a single pixel feature orthogonal to the edge. Not using `BLEND_DISCARDED` will give perfectly smooth edges as long as these edges are at least 2 pixels long. In general, the artifact is hard to see and the improved smoothness of diagonals and temporal coherency is visually more pleasant.
- **edgepost_mlaa_write_threshold.spa: `SCALE_FROM_ALPHA`.** Using a global scale parameter for the edge detection may not be ideal for all situations. This flag allows per-pixel scale parameters to be fetched from the alpha channel of the source image. Because a scale of 255 is not sufficient to completely disable edge detection by itself, a second option `DISABLE_ON_255` is provided, which allows this behavior. Each option adds approximately 60 µs to the processing time.

Making Changes to the SPURS Task

MLAA is intended to be used through the provided handler library, which contains a SPURS task. At certain times, specifically when no MLAA process is running, or when individual SPUs no longer have work, the tasks will release the SPU so that other work can be performed. To conserve memory, only 2 KB of stack is saved in these situations. If changes are made that require other data to be saved (for example, global variables or additional stack space), the handler code must be changed accordingly.

23 Sample Programs

Edge Geometry

| Program | Description |
|---|--|
| <code>samples/edge/collision-sample</code> | An example showing an extension of Edge Geometry to do ray-triangle collision. |
| <code>samples/edge/conditional-sample</code> | An example demonstrating the conditional rendering of Edge jobs based on the results of an occlusion query. |
| <code>samples/edge/occluders-sample</code> | An example showing the usage of quadrilateral occluders with Edge Geometry to cull hidden triangles. |
| <code>samples/edge/runtime-partitioning-sample</code> | An example showing how to run the libedgegeomtool partitioner at run time on the PPU. |
| <code>samples/edge/tunable-sample</code> | An example that allows users to observe the effects of various parameters on Edge Geometry performance. |
| <code>samples/edge/vertexes-only-sample</code> | An example showing the basic usage of Edge Geometry to work on unpartitioned vertex data only, without offline pre-processing. |
| <code>samples/edge/writeback-sample</code> | An example of writing data computed by Edge Geometry back to main memory for use in subsequent frames, demonstrated in the form of a simple particle system. |

Edge Animation

| Program | Description |
|---|--|
| <code>samples/edge/anim-sample</code> | An example showing the basic usage of Edge Animation. |
| <code>samples/edge/locomotion-sample</code> | An example showing the basic usage of Edge Animation to do character locomotion. Callbacks are used to process user data at each stage of blend tree processing. |
| <code>samples/edge/mirror-sample</code> | An example showing the basic usage of Edge Animation to do animation mirroring. |

Edge Geometry And Animation

| Program | Description |
|--|---|
| <code>samples/edge/geom-sample</code> | An example showing the basic usage of Edge Geometry with data output from edgegeomcompiler. |
| <code>samples/edge/elephant-sample</code> | An example showing the basic usage of Edge Geometry applied to a non-trivial skinned character, processed by edgegeomcompiler. |
| <code>samples/edge/blendshape-anim-sample</code> | An advanced example showing how to use Edge Animation and Edge Geometry together to draw a mesh with animated blendshapes. |
| <code>samples/edge/character-sample</code> | An advanced example showing how to use Edge Animation and Edge Geometry together to draw several independently animated characters. |
| <code>samples/edge/fragment-patch-sample</code> | An example showing how to correctly and efficiently patch fragment program constants on the SPU while using jump-to-self synchronization. |

Edge Zlib/LZMA/LZO

Sample programs using Edge Zlib/LZMA/LZO are shown in following table. None of these samples show anything on-screen. Output is entirely over TTY.

| Program | Description |
|---|--|
| <code>samples/edge/zlib-basic-inflate-sample</code> | An example showing the basic usage of Edge Zlib. It loads a file in the .gz format, decompresses it on an SPU, and then checks it against the master version to prove that the decompression is correct. The data is necessarily no more than 64 KB so that the input data and output data can both fit in LS at once during decompression. |
| <code>samples/edge/zlib-basic-deflate-inflate-sample</code> | An example of how Edge Zlib can be used to first compress a buffer and then decompress it again back to the original master data. The PPU loads in a master file and puts this item of work on the Deflate Queue to be compressed by an SPU. An SPU takes this work, and compresses it, then sends the compressed data out to the specified Effective Address. Edge Zlib is then used a second time to re-decompress data and verify that the final data is the same as the original master version. |
| <code>samples/edge/zlib-large-unsegmented-inflate-sample</code> | An example of how a file greater than 64 KB in size can be decompressed with Edge Zlib by letting the SPU fetch the data from main memory as it needs it. The PPU loads the .gz file and puts this item of work on the Inflate Queue to be decompressed by an SPU. An SPU pulls the work off the Inflate Queue, iterates over the input data to decompress it, and sends out the decompressed data to the specified Effective Address. Finally, the decompressed output is compared against the master version to prove it is correct. |
| <code>samples/edge/zlib-large-segmented-inflate-sample</code> | An example of how a file greater than 64 KB in size can be decompressed with Edge Zlib in parallel on multiple SPUs. The PPU loads the .segs file and queues up all the segments onto the Inflate Queue for decompression by the SPUs. The SPUs pull work off the Inflate Queue, decompress the segments in parallel, and send out their results. Finally, the decompressed output is compared against the master version to prove it is correct. |
| <code>samples/edge/zlib-inflate-inplace-sample</code> | Much like " samples/edge/zlib-large-segmented-inflate-sample ", this sample shows the decompression of a large file in segments on multiple SPUs. However in this case, the compressed data is loaded directly into the output buffer to which it will be decompressed. This removes the need for a separate buffer for input and output data. See " Decompression In-place " in Chapter 12, " Using Edge Zlib, LZMA, and LZO " for more information about in-place decompression. |

| Program | Description |
|---|--|
| <code>samples/edge/zlib-segcomp-sample</code> | <p>An example showing how an offline tool for segmenting and compressing data might work. The master file is first split into (64 KB) segments. Then each segment is compressed by calling zlib in such a way that it does not output the 2-byte header and 4-byte footer. The compressed data is then collected together into a container file (" .segs"). The samples above use the output created by this sample tool.</p> <p>Applying compression to a file can, in some cases, make it larger. In such cases, the "compressed" version is not stored in the output, choosing instead to store the raw uncompressed version because this is more efficient.</p> <p>This sample only shows how a tool might work.</p> |
| <code>samples/edge/zlib-streaming-inflate-sample</code> | <p>An example of how to use streaming to decompress multiple large files with Edge Zlib. A PPU streaming thread will load and decompress one file at a time. For each segment in the file, it loads the segment from disk into the destination and decompresses it in-place via an SPU Inflate Task. After each file is decompressed, it will call a user callback, which is used to verify the decompressed version against the master version of the file.</p> |
| <code>samples/edge/zlib-localmemory-streaming-inflate-sample</code> | <p>This program is a modification of the zlib-streaming-sample, in which the destination is local memory (VRAM) instead of main memory. Because it is very slow to read from local memory, each compressed segment is loaded into one of two temporary buffers in main memory. The Inflate Task decompresses from these main memory buffers, and then the uncompressed output is stored to local memory.</p> |
| <code>samples/edge/zlib-inflate-retry-sample</code> | <p>An example showing a (faked) disc-read error that results in invalid data. Edge Zlib attempts to decompress the data on SPU, but detects an error and communicates this fact via the top bit of the <code>workToDoCounter</code>. The PPU detects this error, re-reads the data, and retries the decompression.</p> |
| <code>samples/edge/lzma-basic-inflate-sample</code> | <p>An example showing the basic usage of Edge LZMA. It loads a compressed file, decompresses it on an SPU, and then checks it against the master version to prove that the decompression is correct. The data is necessarily no more than 64 KB so that the input data and output data can both fit in Local Store at once during decompression.</p> |
| <code>samples/edge/lzma-inflate-retry-sample</code> | <p>An example showing a (faked) disc-read error that results in invalid data. Edge LZMA attempts to decompress the data on SPU, but detects an error and communicates this fact back by using the top bit of the <code>m_eaWorkToDoCounter</code>. The PPU detects this error, re-reads the data, and re-tries the decompression.</p> |

| Program | Description |
|--|---|
| <code>samples/edge/lzma-segcomp-sample</code> | <p>An example showing how an offline tool for segmenting and compressing data might work. The master file is first split into (64 KB) segments. Then each segment is compressed by calling LZMA. The compressed data is then collected together into a container file (“<code>.segs</code>”). The samples above use the output created by this sample tool.</p> <p>Applying compression to a file can, in some cases, make it larger. In these cases, the “compressed” version is not stored in the output, choosing instead to store the raw uncompressed version because this is more efficient.</p> <p>This sample only shows how a tool might work.</p> |
| <code>samples/edge/lzma-large-segmented-inflate-sample</code> | <p>An example of how a file larger than 64 KB can be decompressed with Edge LZMA in parallel on multiple SPUs. The PPU loads the “<code>.segs</code>” file and queues up all the segments onto the Inflate Queue for decompression by the SPUs. The SPUs pull work off the Inflate Queue, decompress the segments in parallel and send out their results. Finally the decompressed output is compared against the master version to prove it is correct.</p> |
| <code>samples/edge/lzo1x-basic-inflate-sample</code> | <p>An example showing the basic usage of Edge LZO. It loads a compressed file, decompresses it on an SPU, and then checks it against the master version to prove that the decompression is correct. The data is necessarily no more than 64 KB so that the input data and output data can both fit in Local Store at once during decompression.</p> |
| <code>samples/edge/lzo1x-basic-deflate-sample</code> | <p>An example of how Edge LZO can be used to compress a buffer. The PPU loads in a master file and puts this item of work on the Deflate Queue to be compressed by an SPU. An SPU takes this work, and compresses it, then sends the compressed data out to the specified Effective Address. This buffer is then saved out to a file and may be used as input data to “samples/edge/lzo1x-basic-inflate-sample”.</p> |
| <code>samples/edge/lzo1x-inflate-retry-sample</code> | <p>An example showing a (faked) disc-read error that results in invalid data. Edge LZO attempts to decompress the data on SPU, but detects an error and communicates this fact back via the top bit of the <code>workToDoCounter</code>. The PPU detects this error, re-reads the data, and re-tries the decompression.</p> |
| <code>samples/edge/lzo1x-large-segmented-inflate-sample</code> | <p>An example of how a file greater than 64 KB can be decompressed with Edge LZO in parallel on multiple SPUs. The PPU loads the <code>.segs</code> file and queues up all the segments onto the Inflate Queue for decompression by the SPUs. The SPUs pull work off the Inflate Queue, decompress the segments in parallel, and send out their results. Finally, the decompressed output is compared against the master version to prove that it is correct.</p> |

| Program | Description |
|--|---|
| <code>samples/edge/lzo1x-segcomp-sample</code> | <p>An example of how an offline tool for segmenting and compressing data might work. The master file is first split into 64-KB segments. Each segment is then compressed by calling LZO, and the compressed data is collected together into a container file (“ . segs”). The samples above use the output created by this sample tool.</p> <p>Applying compression to a file could make it larger. In such cases, the raw uncompressed version is stored in the output instead of the compressed version, because this is more efficient.</p> <p>This sample only shows how a tool might work.</p> |

Edge DXT

| Program | Description |
|--------------------------------------|--|
| <code>samples/edge/dxt-sample</code> | <p>This program shows a steady state system that generates some image data on the RSX™, compresses the results to one of the DXT formats on SPU, and then uses the compressed results as a texture on the RSX™, all within the same frame. For debugging purposes, the compressed results are also decompressed on SPU back to raw image data and displayed as a texture using the RSX™.</p> |

Edge Post

| Program | Description |
|--|--|
| <code>samples/edge/post-sample</code> | <p>An example showing the basic usage of Edge Post. It loads a simple scene and applies a set of post-processing effects to the rendered image. The effects applied are Depth of Field, Motion Blur, and Bloom. For comparison purposes, the sample also provides a reference RSX™ implementation of such effects.</p> <p>MLAA is also integrated into this sample. It shows how MLAA performs in motion and how to integrate MLAA with a render loop and the rest of Edge Post.</p> |
| <code>samples/edge/post-sample-hdr</code> | <p>An example showing the basic usage of Edge Post. It loads a simple scene and applies a set of post-processing effects to the rendered image. The effects applied are Depth of Field, Motion Blur, Bloom, and average luminance calculation. The entire post-processing pipeline is done using HDR colors.</p> |
| <code>samples/edge/post-sample-mlaa</code> | <p>This program performs MLAA on 720p textures and can be used to profile the performance on different test images.</p> |