

PlayStation®Edgeライブラリ 概要

現在の仕様は暫定的なものであり、このドキュメントに記載されている内容は予告なく変更される場合があります。ご了承ください。

© 2010 Sony Computer Entertainment Inc.
All Rights Reserved.
SCE Confidential

目次

1 このドキュメントについて	6
目的	6
対象読者と前提条件	6
関連ドキュメントとその他のリソース	6
表記法	7
1 ライブラリ概要	8
特徴	8
ライセンス	8
ファイル	8
2 Edgeジオメトリの概要	11
ジオメトリ処理の種類	11
データストリームの基本型	11
3 Edgeジオメトリの利用	14
オフラインのツール処理	14
実行時PPU処理	17
実行時SPU処理	18
4 Edgeジオメトリランタイムの詳細	21
ジオメトリSPUのみの固定小数点圧縮	21
ジオメトリSPUのみの単位ベクトル圧縮	21
ジオメトリSPUのみのインデックス圧縮	22
コマンドバッファホール	24
SPU入力DMAリスト	24
SPU入力ユーザデータ	25
SPU DMAタグのパフォーマンスに関する注意事項	25
SPU全体のバッファレイアウト	26
SPUジョブのローカルストア管理	26
ユーザ機能およびデータ	27
ランタイム出力バッファの仕組み	29
EdgeジオメトリのSPU処理コストの見積り	31
5 Edgeオフラインジオメトリツール	33
ビルド	33
使用法	33
6 Edgeアニメーションの概要	34
Edgeアニメーションの設計	34
SPUの使い方	34
リファレンス実装	34
Win32の実装	35
アニメーション処理	35
ツール処理	37

データ構造体	37
7 Edgeアニメーションの利用	38
PPUーグローバル	38
SPUー各ジョブ用	39
8 Edgeアニメーションのランタイムの詳細	41
ポーズスタック	41
ブレンドツリーの処理	41
コマンドリスト	42
ブレンディング操作と部分的なアニメーションのロジック	43
例：簡単なブレンドツリー	46
コールバック	47
ミラーリング	47
計算済みのポーズ	48
アニメーションのエンコードおよび圧縮	49
9 Edgeアニメーションのツール	53
はじめに	53
使用法	53
処理概要	53
スケルトン処理	54
アニメーション処理	54
付加的なアニメーション処理	55
圧縮	56
10 EdgeアニメーションとEdgeジオメトリの併用	57
スキニング	57
11 Edge Zlib、LZMA、およびLZOの概要	58
アルゴリズム	58
推奨ユースケース	58
12 Edge Zlib、LZMA、およびLZOの使用	59
実装の詳細	59
Edge ZlibをPPUから利用する手順	60
Edge ZlibをSPUから利用する手順	61
共通の詳細	61
13 Edge Zlibの詳細	64
特徴	64
Edge Zlibとzlibの違い	64
ライセンス	64
DEFLATEデータのヘッダ	64
セグメント化された伸長とされていない伸長	65
Edge Zlibのビルド	65
14 Edge LZMAの詳細	66
特徴	66
Edge LZMAとLZMAの違い	66

圧縮データのヘッダ	66
伸長タスク	66
15 Edge LZOの詳細.....	68
特徴	68
Edge LZOとLZOの違い	68
ライセンス	68
圧縮データのヘッダ	68
伸長・圧縮タスク	68
ローカルストア中の圧縮出力バッファ	69
16 Edge DXTの概要.....	70
概要	70
Edge DXTの設計	70
圧縮のパフォーマンスおよび制限	70
17 Edge DXTの利用.....	71
圧縮	71
解凍	71
18 Edge Postの概要.....	72
特徴	72
SPUの使い方	72
データ構造体	73
19 Edge Postの利用.....	74
基本的な手順	74
20 Edge Postランタイム.....	75
処理ステージとエフェクトチェイン	75
タイルサイズの選択	75
SPUのメモリレイアウト	76
SPU同期	76
RSX®同期	77
PPU同期	77
ライブラリの提供するSPUピクセル処理機能	78
エフェクトコードを書く	79
サポートされているピクセルフォーマット	80
Edge Postの組み込み、および推薦される使用法	81
後処理エフェクトのRSX®実装とSPU実装の違い	83
パフォーマンス	84
21 Edge Post備考.....	85
Edge Postのデータフローの例	85
22 Edge Post MLAA	88
概要	88
基本的な手順	88
MLAAのフレーム中の配置	88
エッジ検出およびチューニング	88

性能特性	89
サンプル	89
制限	90
オプションの機能	90
SPURSタスクの変更	91
23 サンプルプログラム	92
Edgeジオメトリ	92
Edgeアニメーション	92
Edgeジオメトリおよびアニメーション	92
Edge Zlib・LZMA・LZO	93
Edge DXT	96
Edge Post	96

1 このドキュメントについて

目的

このドキュメントには、Edge ライブラリの概要が記載されています。このライブラリは、PlayStation®3 のユニークなアーキテクチャを使用して、従来の PPU/RSX®アーキテクチャを超えるパフォーマンスを実現するためのライブラリです。

対象読者と前提条件

このドキュメントは、PlayStation®3 用の高性能アプリケーションを書こうとしている PlayStation®3 デベロッパのため書かれたものです。デベロッパは、以下の点に関して熟知していることを前提としています。

- C と C++
- PlayStation®3 のハードウェア
- SCE の標準ライブラリ関数

関連ドキュメントとその他のリソース

Edge ライブラリ

この概要と以下のドキュメントを併用することにより、Edge ライブラリの使用法やリファレンスについての完全な情報を得ることができます。

- 「PlayStation®Edge ジオメトリライブラリ クイックスタート」
- 「PlayStation®Edge ジオメトリライブラリ リファレンス」
- 「PlayStation®Edge オフラインツール用ジオメトリライブラリ リファレンス」
- 「PlayStation®Edge アニメーションライブラリ リファレンス」
- 「PlayStation®Edge オフラインツール用アニメーションライブラリ リファレンス」
- 「PlayStation®Edge Zlib ライブラリ リファレンス」
- 「PlayStation®Edge LZMA ライブラリ リファレンス」
- 「PlayStation®Edge LZO ライブラリ リファレンス」
- 「PlayStation®Edge DXT ライブラリ リファレンス」
- 「PlayStation®Edge Post ライブラリ リファレンス」

SPURS および libgcm

Edgeの中では、SPURSを使ってSPUを駆動することができます。SPURS、libgcm（グラフィックスコマンド管理ライブラリ）の概要については、PlayStation®3 Developer Networkのウェブサイト

(<https://ps3.scedev.net>) 上のPlayStation®3 SDKドキュメントパッケージから入手できる次のドキュメントを参照してください。

- 「libspurs 概要」
- 「libgcm 概要」

RSX®キャッシュオプティマイザ

RSX®キャッシュオプティマイザについての詳細は、「三次元メッシュレンダリングのための頂点キャッシュ方式の改良 (An Improved Vertex Caching Scheme for 3D Mesh Rendering)」

(<http://www.ecse.rpi.edu/~lin/K-Cache-Reorder/Lin-Yu-TVCG.pdf>) を参照してください。

SPA、および EdgePostFilterGen ツール

Edge パッケージに含まれている、「SPU パイプライン化アセンブラ (SPA) ユーザガイド」は、アセンブリ最適化ツールである SPA について説明しています。

Edge パッケージに含まれている、「EdgePostFilterGen ユーザガイド」は、簡単な画像カーネルの操作（ガウス画像フィルタなど）用の SPA に最適化されたループを生成するためのコマンドラインユーティリティである、EdgePostFilterGen について説明しています。このループは、Edge Post の中で利用できます。出力は、そのまま SPA ツールに渡すことのできる SPA ファイルの形式になります。

表記法

このドキュメントでは、以下のような印刷上の表記法を使います。

記法	意味
等幅フォント	プログラミングコード、処理命令、レジスタ名、データ型、イベント、ファイル名、その他のリテラルを表します。また、関数、構造体、マクロの名前を表すこともあります。
<u>青字 + 下線</u>	ハイパーリンクを表します（青く表示されるのは、カラー印刷もしくはオンラインの場合だけです）。

1 ライブラリ概要

特徴

Edge は、従来の PPU/RSX®アーキテクチャ上でさらなるパフォーマンスを実現するのに役立つライブラリです。これが可能なのは、Edge が SPU を効率的に使うことにより、PlayStation®3 固有のアーキテクチャを使用しているからです。SPU は、PPU/RSX®アーキテクチャよりも高速に計算を行い、優れたデータ圧縮方式を利用しています。また SPU は、通常 PPU によって実行される処理を行うこともできます。

このライブラリは、以下のユーティリティを提供します。

- **Edge ジオメトリ**：SPU により、スキニング、座標変換、カリングを実行します。
- **Edge アニメーション**：SPU により、アニメーションジョイントツリーのブレンディングや変換を実行します。
- **Edge Zlib**：SPU により、可逆データの解凍・圧縮を実行します。
- **Edge LZMA**：SPU により、可逆データの解凍を実行します。
- **Edge LZO**：SPU により、可逆データの解凍・圧縮を実行します。
- **Edge DXT**：SPU により、画像データの圧縮・解凍を実行します。
- **Edge Post**：SPU により、画像の後処理を実行します。

Edge においては、あらゆる SPU コードを SPURS ジョブ、SPURS ジョブキュー、あるいは SPURS タスクを使って動かすことができますが、それ以外の SPU マネジメントシステムを使うこともできます。

ライセンス

PlayStation®Edge ライブラリは、license¥others¥PlayStation_Edge_Terms_and_Conditions_j.txt に明記されている諸条件の基で配布されています。

ファイル

Edgeを利用するために必要なファイルは、表 1 の通りです。

表 1 Edge のファイル

ファイル名	解説
common¥include¥edge¥geom¥edgegeom_structs.h	Edge ジオメトリ 共通のヘッダファイル
common¥include¥edge¥geom¥edgegeom_attributes.h	
spu¥include¥edge¥geom¥edgegeom.h	Edge ジオメトリのヘッダファイル (SPU 用)
spu¥include¥edge¥geom¥edgegeom_compress.h	
spu¥include¥edge¥geom¥edgegeom_decompress.h	
spu¥src¥edge¥geom¥*.cpp	Edge ジオメトリのソースファイル (SPU 用)
common¥include¥edge¥anim¥edgeanim_common.h	Edge アニメーション 共通のヘッダファイル
common¥include¥edge¥anim¥edgeanim_macros.h	
common¥include¥edge¥anim¥edgeanim_structs.h	

ファイル名	解説
ppu¥include¥edge¥anim¥edgeanim_ppu.h	Edge アニメーションのヘッダファイル (PPU 用)
spu¥include¥edge¥anim¥edgeanim_spu.h	Edge アニメーションのヘッダファイル (SPU 用)
ppu¥src¥edge¥anim¥*.cpp	Edge アニメーションのソースファイル (PPU 用)
spu¥src¥edge¥anim¥*.cpp	Edge アニメーションのソースファイル (SPU 用)
spu¥src¥edge¥anim¥*.s	
common¥include¥edge¥edge_stdbool.h	Edge 共通のヘッダファイル
common¥include¥edge¥edge_stdint.h	
ppu¥include¥edge¥zlib¥edgezlib_ppu.h	Edge Zlib PPU ライブラリインタフェース用のヘッダファイル
spu¥include¥edge¥zlib¥edgezlib_spu.h	Edge Zlib SPU ライブラリインタフェース用のヘッダファイル
common¥include¥edge¥zlib¥edgezlib_inflate_queue_element.h	SPU・PPU 間通信用の Edge Zlib 内部ヘッダファイル
common¥include¥edge¥zlib¥edgezlib_deflate_queue_element.h	
ppu¥src¥edge¥zlib¥*.cpp	Edge Zlib PPU ライブラリ用のソースファイル
spu¥src¥edge¥zlib¥*.cpp	Edge Zlib SPU ライブラリ用のソースファイル
spu¥src¥edge¥zlib¥*.s	
spu¥src¥edge¥zlib¥*.h	Edge Zlib SPU ライブラリ用の内部ヘッダファイル
spu¥src¥edge¥zlib-inflate-take¥*.cpp	Edge Zlib SPU 伸長タスクのソースファイル
ppu¥include¥edge¥lzma¥edgelzma_ppu.h	Edge LZMA PPU ライブラリインタフェース用のヘッダファイル
spu¥include¥edge¥lzma¥edgelzma_spu.h	Edge LZMA SPU ライブラリインタフェース用のヘッダファイル
common¥include¥edge¥lzma¥edgelzma_inflate_queue_element.h	SPU・PPU 間通信用の Edge LZMA 内部ヘッダファイル
ppu¥src¥edge¥lzma¥*.cpp	Edge LZMA PPU ライブラリ用のソースファイル
spu¥src¥edge¥lzma¥*.cpp	Edge LZMA SPU ライブラリ用のソースファイル
spu¥src¥edge¥lzma¥*.s	
spu¥src¥edge¥lzma¥*.h	Edge LZMA SPU ライブラリ用の内部ヘッダファイル
spu¥src¥edge¥lzma-inflate-take¥*.cpp	Edge LZMA SPU 伸長タスクのソースファイル
spu¥src¥edge¥lzma-deflate-take¥*.cpp	Edge LZMA SPU 圧縮タスクのソースファイル
ppu¥include¥edge¥lzo¥edgelzo_ppu.h	Edge LZ0 PPU ライブラリインタフェース用のヘッダファイル
spu¥include¥edge¥lzo¥edgelzo_spu.h	Edge LZ0 SPU ライブラリインタフェース用のヘッダファイル
common¥include¥edge¥lzo¥edgelzo_inflate_queue_element.h	SPU・PPU 間通信用の Edge LZ0 内部ヘッダファイル
common¥include¥edge¥lzo¥edgelzo_deflate_queue_element.h	
ppu¥src¥edge¥lzo¥*.cpp	Edge LZ0 PPU ライブラリ用のソースファイル
ppu¥src¥edge¥lzo¥*.s	Edge LZ0 SPU ライブラリ用のソースファイル
ppu¥src¥edge¥lzo¥*.h	
spu¥src¥edge¥lzo¥*.h	Edge LZ0 SPU ライブラリ用の内部ヘッダファイル
spu¥src¥edge¥lzo_inflate-take¥*.cpp	Edge LZ0 SPU ソースファイル (L01X 伸長タスク用)
spu¥include¥edge¥dxt¥edgedxt.h	Edge DXT ヘッダファイル (SPU 用)
spu¥src¥edge¥dxt¥*.s	Edge DXT ソースファイル (SPU 用)
common¥include¥edge¥post¥*	Edge Post 共通のヘッダファイル
ppu¥include¥edge¥post¥*	Edge Post のヘッダファイル (PPU 用)
ppu¥src¥edge¥post¥*	Edge Post のソースファイル (PPU 用)
spu¥include¥edge¥post¥*	Edge Post のヘッダファイル (SPU 用)
spu¥src¥edge¥post¥*	Edge Post のソースファイル (SPU 用)

外部ライブラリ依存関係

- Edge アニメーション SPU ライブラリは、libdma に依存しています。
- Edge Zlib、Edge LZMA、Edge LZ0 PPU ライブラリは、libspurs と libsync に依存しています。
- Edge Zlib、Edge LZMA、Edge LZ0 SPU ライブラリは、libspurs、libsync、libatomic、および libdma に依存しています。
- Edge Post SPU ライブラリは、libdma と libatomic に依存しています。

2 Edgeジオメトリの概要

ジオメトリ処理の種類

次に、Edge ジオメトリで実行できる一般的な演算について説明します。

スキニングその他のワールド空間変換

1つの頂点に対して最大で4つの行列を使用して、位置、法線、接線、従法線など、さまざまな頂点属性をスキニングしたものを計算することができます。スキニングを実行する理由には、さまざまなものがあります。以下に例を挙げます。

- RSX®の負荷を減らすため。この場合には、あらゆる属性がスキニングされて出力されます。
- 不要な三角形をカリングするため（下記の [トライアングルカリング](#) を参照）。この場合にスキニングする必要があるのは頂点位置だけです。

トライアングルカリング

頂点位置を使って、以下のような、RSX®のセットアップ段階を通過しない三角形を除外した新しいインデックステーブルを生成します。

- フラストムによってカリングされた三角形
- 後ろ向きの三角形
- 面積ゼロの三角形
- ピクセルのない三角形

Edge ジオメトリ処理においては、オブジェクトは「ジオメトリ セグメント」単位で処理されます。各「ジオメトリセグメント」には、通常、同一のマテリアルの三角形が最高 6,000 個まで含まれています。

三角形は最終的に「インデックス化された三角形リスト」として表されます。このリストの中では、各三角形が、その3つの頂点のインデックスによって記述されます。

データストリームの基本型

頂点入出力データ

頂点データは、RSX®互換、もしくはSPUのみの属性形式による頂点の集合です。カスタム頂点データストリームを処理するために、ユーザはいつでも独自のコールバック関数を設定できるため、システムに導入できる頂点属性入出力（IO）データの種類の制限はありません。

ライブラリがサポートする一般的な入出力データ型の一部を示したのが、表2および表3です。

表2 デフォルトでサポートされている SPU 頂点属性型

型	各要素のビット深度	サポートする成分の数
I16N	16	1～4
F32	32	1～4
F16	16	1～4
U8N	8	1～4
I16	16	4
X11Y11Z10N ¹	32	3
U8	8	4

型	各要素のビット深度	サポートする成分の数
SPU のみの単位ベクトル ²	24	4
SPU のみの固定小数点 ³	8	1～4

表 3 デフォルトでサポートされている RSX®頂点属性

型	各要素のビット深度	サポートする成分の数
I16N	16	1～4
F32	32	1～4
F16	16	1～4
U8N	8	1～4
I16	16	4
X11Y11Z10N*	32	3
U8	8	4

¹ X11Y11Z10N フォーマットが持てる成分の数は3つだけで、常に 32 ビットです。X 成分は常に 11 ビット、Y 成分は 11 ビット、Z 成分は 10 ビットです。各コンポーネントは-1.0～1.0 の範囲に正規化された値として解釈されます。

² SPU のみの単位ベクトルのフォーマットでは、単位ベクトル中の最小の 2 成分に対しては、常に、1 成分につき 10 ビットが割り当てられます。w 成分は常に 1 ビットで、1 もしくは-1 を表します。

³ SPU のみの固定小数点フォーマットの各成分は、最高 31 ビットの分解能を持つことができますが、全成分を組み合わせたデータのビット深度は 8 の倍数である必要があります。

このような圧縮 SPU のみの属性フォーマットの詳細は、このドキュメントの後で述べます。

実装効率のため、頂点データの各種入出力フォーマットは、アプリケーションによって

EdgeGeomSpuVertexFormat と EdgeGeomRsxVertexFormat オブジェクトで特別に定義されます。

これらの構造体についての詳細は、「PlayStation®Edge ジオメトリライブラリリファレンス」に記載されています。

インデックスデータ

出力インデックスデータは、個別の三角形の集まりです（1 つの三角形に対して 3 つのインデックス）。各インデックスは基本的に 16 ビットで、RSX®と互換性があります。ただし、入力インデックスデータストリームは、SPU のみの圧縮されたフォーマットであることもあり、この場合には、トライアングルカリングを実行する前に解凍を行う必要があります。

インデックスデータが必要なのは、トライアングルカリングが行われる場合、もしくはアプリケーション固有のコードがそれを必要とする場合だけです。

スキニング行列データ

スキニング行列は、オブジェクト行列全体の 2 つの部分集合です。Edge のネイティブのスキニング行列のフォーマットは 3×4 行優先ですが、4×4 行列（行優先もしくは列優先）もサポートされます。これらの範囲には、ローカルにインデックスが付けられ、ツールの中でアップロードされた範囲に変換されます。このデータは、スキニングを実行しない場合には不要です。

スキニングインデックス/ウェイトデータ

スキニング重みとインデックスは、互いにインタレース化されて配置されており、インデックスは符号なし 8 ビットフォーマット、重みは正規化された符号なし 8 ビットフォーマットになっています。Edge は、4 ボーンスキニングに加えて、剛体・機械的物体の単一のボーンスキニングをサポートします。このデータは、スキニングを実行しない場合には不要です。

ブレンド形状データ

ブレンド形状のデータ自体は、頂点データに非常によく似ています。ただし、ブレンド形状がゼロでないそれぞれの頂点に関しては、このデータは、形状ブレンディングを実行するために必要な位置や法線などの属性しか含んでいません。

3 Edgeジオメトリの利用

この章には、主に以下のようなセクションがあります。

- オフラインツール処理：実行時処理用の最適化のために、各オブジェクトに対してツール内で行われる処理について説明します。
- PPU ランタイム処理：PPU がツールからの情報を処理するための SPU ジョブを設定する方法について説明します。ここで使われているコードレイアウトや構造は、「PlayStation®Edge ジオメトリライブラリリファレンス」ドキュメントに記載されています。
- SPU ランタイム処理：SPU 処理、およびこの処理がアプリケーション製作者、PPU 処理、オフラインツールであるパーティショナーにどのような影響を与えるかについて説明します。

ジオメトリ処理パイプラインに必要なサブシステムやデータ構造は数多くあります。この章では、通常のスキニングされたキャラクタに対して、実行時に行われる処理について説明します。これらのシステムすべての詳細については、第4章「[Edgeジオメトリランタイムの詳細](#)」で説明します。

オフラインのツール処理

ツール処理は3層のライブラリから構成されます。各層は、それぞれ異なる種類の Edge ユーザを想定して設計されています。

- 上級ユーザ：パーティショナー、キャッシュオブティマイザ、libedgegeomtool が含まれています。
- 中級ユーザ：COLLADA™パーサーなど、中級レベルのヘルパー機能が含まれています。
- 初級ユーザ：モデリングパッケージからエクスポートして、データを固定されたバイナリフォーマットに変換することのできる、edgegeomcompilerのフル機能の実行可能ファイルが含まれています。edgegeomcompilerの詳細については5章「[Edgeオフラインジオメトリツール](#)」の章を参照してください。

これは、各オブジェクトがツールを通過する際に行われる基本的なプロセスのフローです。このプロセスについては、後のセクションで説明します（このセクションに記載された機能の詳細については、「PlayStation®Edge オフラインツール用ジオメトリライブラリ リファレンス」を参照してください）。

アートソースファイルを処理する

アートソースファイルからインデックスと頂点データを取得します。さらに、ジオメトリメッシュが三角形に分割されてない場合には分割します。libedgegeomtool の中の edgeGeomTriangulatePolygons() 関数を使うと、アートソースファイルを処理するのに便利です。

(2) 頂点フォーマット、スキニングフレーバー、カリングフレーバー、インデックスリストフレーバー、スキニング行列フォーマットの選択

入力、出力、および（オプションで）ブレンドデルタストリームと RSX®だけのストリームのフォーマットを選択します。ストリームごとに、Edge の組み込み頂点フォーマットの1つを選択する

（edgeGeomGet[Spu,Rsx]VertexFormat() 関数を使う）か、もしくはカスタムフォーマットを構築します。カスタムフォーマットを使う場合には、それぞれの最終的なフォーマットに対して edgeGeom[Spu,Rsx]VertexFormatIsValid() を呼び出せば、一般的なエラーをテストするのに役立ちます。

また、インデックスリスト用のフレーバーも選択する必要があります。オプションとしては、時計回りおよび反時計回りのワインディング順序による圧縮と非解凍があります。

また、スキニング用のフレーバーも選択する必要があります。Edge では、ユニフォームではないスケーリング（高速）とユニフォームなスケーリング（さらに高速）、さらにスケーリングなし（最高速）の3種類のスケーリングを含むスキニング行列をサポートしています。また、4 ボーンスキニング（デフォルト）に加えて、スキニングデータバッファを大幅に縮小する単一のボーンスキニング（機械やロボットのような剛体用）をサポートしています。

また、三角形カリング用のフレーバーも選択する必要があります。このフレーバーは、三角形カリングが有効化されているか、およびどのテストに対してかを決定します。たとえば、エンジンが2 辺の三角形を描画している場合、バックフェースの三角形のテストを無効にすることができます。

また、(スキニングを実行する場合)にはスキニング行列データのフォーマットを選択する必要があります。Edge のネイティブのフォーマットは、3×4 の行優先行列です。利便性のため、4×4 行列（行優先および列優先）もサポートされていますが、この行列は3×4 行列より 33%大きいので、メモリ使用量がより増えることになります。

(3) 頂点データをバッチに分類する

ツールは、まず、冗長な頂点を取り除きます。この処理を実現するには、`edgeGeomMergeIdenticalVertexes()` という関数が便利です。

さらに、ツールは頂点データをバッチに分割する必要があります。バッチというのは、1 回の描画呼び出しでレンダリングできるジオメトリ、つまり、同一のマテリアルおよびレンダリング設定を使っているジオメトリのブロックです。`libedgegeomtool_wrap.cpp` 中の `partitionSceneIntoBatches()` 関数は、この処理の一例ですが、ここでは入力シーンが `EdgeGeomScene` オブジェクトであると仮定しています。

入力シーンで使われているマテリアルが1 つだけの場合、この手順は省略することができ、シーン全体が1 つのバッチになります。

(4) バッチをセグメントに分割する

ツールは、あらゆるジオメトリデータを複数の「セグメント」に分割する必要があります。各セグメントは、SPU が処理する単一の作業負荷を表しています。

これを実現するには、`libedgegeomtool` 中にある `edgeGeomPartitioner()` という関数が便利です。以下は、`edgeGeomPartitioner()` 関数の簡単な説明です。

- ポリゴン・頂点間隣接ルックアップテーブルを生成します。
- SPU の処理に必要なユニフォームテーブルの数を計算します。
- 初期セグメントを生成します。
- 未アサインのポリゴンのリストを生成します。
 - 圧縮されたインデックスが使われている場合には、リストを空間的に降順になるようにソートします。
- 未割当のポリゴンが残っている間、以下を繰り返します。
- 圧縮されたインデックスが使われている場合、
 - 未アサインリストから、現在のセグメントに追加できる最初のポリゴンを（逆順で）探します。

- 圧縮されたインデックスが使われていない場合、
 - ・現在のセグメントに追加できる未アサインのポリゴンのうち、現在のセグメントの中央位置の値に最も近いものを探します。
 - ・追加できるポリゴンが見つからなくて、なおかつ、未アサインのポリゴンがなくなった場合には、ループを終了します。
 - ・追加できるポリゴンが見つからない場合、新しいセグメントを生成して、while ループを再開します
 - ・ポリゴンをセグメントに追加して、未アサインリストから削除します。
 - ・現在のセグメントに追加するポリゴンがある間、以下を行います。
 - 接続済みおよび未アサインのポリゴンすべてをループし、現在のセグメントに追加でき、かつその頂点を追加する場合に最もコストが低くなるポリゴンを検索します。
 - 追加できるポリゴンが存在する場合には、そのポリゴンをセグメントに追加して、未アサインリストから削除します。
- 分割された三角形リストに対して、指定された変換前キャッシュオブティマイザ (kcache オプティマイザなど) を実行します。
- セグメントを返します。

このコスト評価関数では、ブレンド形状も勘定に入れられます。通常は、ブレンド形状ストリームのサイズは保存データがまばらなため、対応する元のメッシュのサイズ以下になります。したがって、ブレンド形状の占有するメモリ空間は、入力頂点とまったく等しいので、ブレンド形状が入出力バッファに入りきらないという問題が発生することはめったにありません。

(5) 各セグメント用の最終的なデータバッファを生成する

各セグメントにどの三角形が含まれるかがわかったので、各セグメント用の最終的なデータバッファのすべて（頂点ストリーム、圧縮インデックスストリーム、スキニングバッファ、ブレンド形状デルタストリームなど）を生成する必要があります。これらのバッファは、ツールから見たときには、不透明なデータブロックであると思える必要があります。これらのバッファを処理すると想定されているのは、Edge ランタイムだけです。

(6) ファイルに出力する

ユーザは、自分のニーズに最もふさわしい独自の方法でセグメントを出力することができます。

RSX®キャッシュオブティマイザについて

RSX®キャッシュオブティマイザは、K キャッシュ頂点最適化アルゴリズムをモデルにしていますが、よりよい実行時パフォーマンスを得るためにさまざまな改善が施されています。たとえば、edgegeomcompiler は、山登り法による大域的最適化アルゴリズムを利用して、結果を最大で 5%改善することに成功しています。

詳細については、「三次元メッシュレンダリングのための頂点キャッシュ法の改良 (An Improved Vertex Caching Scheme for 3D Mesh Rendering)」を参照してください。URLは、このドキュメントの「[関連ドキュメントとその他のリソース](#)」のセクションに記載されています。

- インデックスバッファ中の最大の頂点インデックスを探します。
- 三角形・頂点間隣接ルックアップテーブルを生成します。
- インデックスをソートして、三角形リストを二分検索によって検索できるようにします。

- 頂点が残っている間、以下を実行します。
 - 次の方法で、キャッシュの中から、最小コストの頂点を探します。
 - ミニキャッシュの中に頂点がある場合
 - ミニキャッシュ中で最善の頂点を探します。
 - ・ 最善というのは、最小コストによって決定します。
 - ・ コストは、RSX®サイクル、隣接する頂点の数、FIFO の中での位置による尺度です。
 - それ以外の場合で、変換後のキャッシュの中に頂点がある場合、
 - 変換後のキャッシュ中で最善の頂点を探します。
 - ・ 最善というのは、最小コストによって判断します。
 - ・ コストは、RSX®サイクル、隣接する頂点の数、FIFO の中での位置による尺度です。
 - それ以外の場合には、使われていない任意の頂点の中から、最近傍のものを選びます。
 - まだ出力インデックスリストに追加されていない隣接三角形のすべてを追加します。
 - 隣接する三角形が残っている間、以下を実行します。
 - 隣接する三角形のそれぞれについて、
 - ・ 追加する最もコストの低い三角形を探します。
 - ・ コストは、RSX®サイクル、隣接する頂点の数、FIFO の中での位置による尺度です。
 - 三角形を取り出して、出力インデックスリストに追加します。
 - 隣接する頂点のそれぞれについて、
 - 隣接した三角形のすべてが取り出された場合、
 - 頂点を取り出します。

実行時PPU処理

Edge ジオメトリに必要な PPU の実行時処理はかなり単純なので、ライブラリではランタイムの PPU 関数を提供していません。

実行できるジオメトリ処理には、次の 2 つの基本的な種類があります。

- 解凍、スキニング、およびトライアングルカリングが可能な、標準ジオメトリ処理。
- SPU のみの属性フォーマットを含む頂点データ配列の解凍結果をストリーミングする。この種類の処理には、ツールからの追加データは不要です。

以下は、各セグメントに対して異なる処理を行う際に生じる、一般的な 2 つのパイプラインを示しています。

解凍、スキニング、トライアングルカリングの対象となるセグメントからジョブを生成する

- (1) EdgePpuConfigInfo によって指定されたサイズのコマンドバッファに、16 バイト境界でアラインメントされた空白部分を挿入して、その空白部分のアドレスをコマンドバッファホールアドレスとして記録します。
- (2) ホールの先頭位置、およびホールが 128 バイト境界を超えるたびにその位置に、「自分へのジャンプ」を挿入する必要があります。
- (3) コマンドバッファホールアドレスを RSX®入出力オフセットに変換します。
- (4) ブレンド形状を使っている場合、使っている各ブレンド形状について、以下の処理を行います。

ブレンド形状のバッファの長さ != 0

–ブレンド形状バッファのアドレスとサイズを `shapeInfo` にコピーします。

–ユーザ構造体からアルファをコピーします。

–`shapeInfo` リストのポインタをインクリメントします。

(5) このセグメント用の `CellSpursJob256` 構造体を設定します。

解凍のためのジョブを生成する

(1) 入出力頂点データフォーマットフレーバーの最大ストライドを計算します。

(2) セグメント当たりの最大頂点数を計算し、頂点データ出力が 16 バイトにアラインメントされるように丸めます。

(3) 出力アドレスの先頭を計算します。

(4) 解凍対象の頂点が残っている間、以下を実行します。

- このセグメントの頂点の数を計算します。
- DMA オフセットおよびサイズを計算します。
- `EdgeSpuConfigInfo` 構造体を設定します。
- `CellSpursJob256` 構造体を設定します。
- 頂点データ出力アドレスをインクリメントします。
- ジョブポインタをインクリメントします。

実行時SPU処理

SPU 処理は、ユーザによる最大限の制御を可能にするために、たとえば独自のアニメーション処理を組み込むなど、ライブラリは一連の関数として提供されます。これにより、スキニング後のマテリアルを風の中で揺らせるなど、ジオメトリに対するゲーム固有のさまざまな変更を行うことができます。

この処理の順序は、ほぼ直列です。

以下は、各セグメントに対して異なる処理を行う際に生じる、一般的な 2 つのパイプラインを示しています。

(1) 初期化

関数 `edgeGeomInitialize()` は、他の処理のために、内部状態や作業用バッファを初期化します。

この関数は、他のいかなる関数よりも先に呼び出される必要があります。

(2) 頂点を解凍する

頂点ストリームから頂点属性（位置、法線、接線、テクスチャ座標、その他）を抽出します。これは、`edgeGeomDecompressVertexes()` 関数を呼出すことによって実行できます。解凍された頂点データは、すべて、頂点の属性ごとに 4 つの単精度 F32 値を持つ「ユニフォームテーブル」（属性ごとに一個のテーブル）に保存されます。

(3) ブレンド形状を処理する

ブレンド形状が使われている場合には、ここでブレンディングを実行します。

これは、`edgeGeomProcessBlendShapes()` 関数を呼び出すことによって実行できます。

(4) 頂点をスキニングする

オブジェクトをスキニングして、新しい位置、法線、接線、従法線を生成します。
これは、`edgeGeomSkinVertexes()` 関数を呼び出すことによって実行できます。

(5) インデックスを解凍する

入力インデックスデータが SPU のみの圧縮されたフォーマットである場合には、解凍する必要があります。
この処理が必要なのは、カリングが実行される場合、もしくはアプリケーション固有のコードがインデックスデータを必要とする場合だけです。
これは、`edgeGeomDecompressIndexes()` 関数を呼び出すことによって実行できます。

(6) オクルージョンカリリング

オクルーダーの数がゼロでない場合には、スクリーン空間内で 1 つ以上のオクルーダーの背後にある三角形をすべてカリリングし、カリリングされた三角形を除いた新しいインデックスバッファを作成します。オクルーダーは、4 つの平面頂点によって定義されるワールド空間クワッドです。
これは、`edgeGeomCullOccludedTriangles()` 関数を呼び出すことによって実行できます。

(7) 三角形をカリリングする

カリリングフレイバーが `EDGE_GEOM_CULL_NONE` に設定されていない場合には、不必要な三角形をすべてカリリングして、カリリングされた三角形を含まない新しいインデックスバッファを作成します。カリリングフレイバーは、裏向きと表向きの三角形のどちらがカリリング可能か、および視錐台の外にある三角形はカリリング可能かどうか指定するのに利用できます。
これは、`edgeGeomCullTriangles()` 関数を呼び出すことによって実行できます。

(8) 出力領域を割り当てる

メインメモリに、インデックス、頂点、コマンドバッファホールの出力用の出力領域を割り当てます。これは、`edgeGeomedgeGeomAllocateOutputSpace()` 関数を呼び出すことによって実行できます。

(9) インデックスを出力する

インデックスを DMA を使って出力します。この処理が必要なのは、トライアングルカリリングが使われている場合、もしくはアプリケーション固有のコードが入力インデックスデータを変更する場合だけです。
これは、`edgeGeomOutputIndexes()` 関数を呼び出すことによって実行できます。

(10) 頂点を圧縮する

不要になった入力データを上書きする方式で、新たに指定された頂点属性を含んだ新しい出力頂点ストリームを生成します。
これは、`edgeGeomCompressVertexes()` 関数を呼び出すことによって実行できます。

(11) 頂点を出力する

頂点を DMA を使って出力します。
これは、`edgeGeomOutputVertexes()` 関数を呼び出すことによって実行できます。

(12) コマンドバッファを開始する

コマンドバッファホールの書き込みを開始します。

これは、`edgeGeomBeginCommandBufferHole()` 関数を呼び出すことによって実行できます。

(13) 頂点データ配列コマンドを設定する

指定されたコンテキストに属性アドレスとフォーマットを書き込みます。これは、`edgeCullTriangles()` 関数を呼び出すことによって実行できます。

(14) 描画インデックスアレイコマンドを設定する

指定されたコンテキストに描画インデックスアレイコマンドを書き込みます。
これは、`cellGcmSetDrawIndexArray()` 関数を呼び出すことによって実行できます。

(15) コマンドバッファホールを終了する

リングバッファレポートコマンドを書き込んで、ホールの使われていない部分に 0 (NOP) を書き込みます。
最後に、コマンドバッファホールを DMA によって出力します。
これは、`edgeGeomEndCommandBufferHole()` 関数を呼び出すことによって実行できます。

注意： 正確さを保証するために、あらゆる SPU 関数は、リファレンスドキュメントに記載された通りの順序で呼び出すようにしてください。

4 Edgeジオメトリランタイムの詳細

Edge ジオメトリによって提供されるデータ圧縮には、頂点属性データ用が2つと、インデックスデータ用が1つの3種類があります。この章では、それぞれの詳細について説明します。

ジオメトリSPUのみの固定小数点圧縮

頂点の位置属性などを圧縮するために使われる圧縮方式は、各成分が n/x ビットの固定小数点値であるような単純なビット圧縮です。libedgegeomtool では、ビット数を最大限に活用するために、各固定小数点値にオフセットを加算して、出力値がすべて正になることを保証します。実行時の解凍の際には、各成分の固定小数点値を適切に元の範囲に戻すために、対応するオフセットを適用する必要があります。

- 属性は他の RSX®フォーマットのストリームと同じデータテーブル中に存在するので、各属性の使用ビット数の合計は8の倍数、各成分の使用ビット数の合計は31以下である必要があります。
- このデータでは、各頂点のデータを表すのに連続するビットを使います。
- 連続するビットの中で、各成分は固有のビットフィールドとビット数を持ちます。たとえば、3つの成分を24ビットで表す場合、1番目の成分が8.2ビットのフィールド、2番目の成分が7.2ビットのフィールド、3番目の成分が3.2ビットのフィールドというように割り当てることができます。
- RSX®ネイティブのフォーマットではないため、このように圧縮された属性が存在するデータテーブルは、SPUの中で解凍して、RSX®互換のフォーマットに変換する必要があります。

ジオメトリSPUのみの単位ベクトル圧縮

法線や接ベクトルのような単位ベクトル属性を圧縮するために使われる圧縮方式は、単位ベクトルの最小の2つの成分が $-\sqrt{2}/2 \sim \sqrt{2}/2$ の範囲を表す10ビットで表現される最小2ビット圧縮です。最大成分は、次の式によって復元されます。

$$\text{最大成分} = \sqrt{1 - \text{最小成分 A}^2 - \text{最小成分 B}^2}$$

さらに、4番目の成分が-1か1かを表すのに1ビットが使われます。この4番目の成分は、後で頂点プログラムにおいて、法線と接線から従法線を復元するために使われます。最後に、2つの成分のうちどちらが最小かを表すために、2つの追加ビットが使われます。

- 属性は、他の RSX®フォーマットのストリームと同じデータテーブルの中に存在するので、この属性型のに使われるビット数の合計は、常に23ビットである必要があります。
- これは、RSX®ネイティブのフォーマットでないため、このように圧縮された属性が存在するデータテーブルは、SPUの中で解凍して、RSX®互換フォーマットに変換する必要があります。
- このフォーマットは決してブレンド形状頂点差分フォーマットとして使用すべきではありません。大きさは常に1.0であることを思い出してください。

このデータのフォーマットは、表4の通りです。

表4 単位ベクトル圧縮用のデータフォーマット

0	10	11	19	20	21	22	23
最小成分 A		最小成分 B		SHUF		W	S

- 最小成分 A は、元の単位ベクトルの成分の中で最も小さい値です。
- 最小成分 B は、元の単位ベクトルの成分の中で2番目に小さい値です。

- SHUF は、最大値を指定するシャッフルマスクテーブルへのインデックスです
- W がセットされている場合には、元の単位ベクトルの W の値は 1、セットされていなければ -1 です。
- S がセットされている場合には、最大値の符号は正、セットされていなければ負です。

ジオメトリSPUのみのインデックス圧縮

インデックス化された三角形リストを圧縮するために使われる圧縮方式は、高レベルの圧縮方式と低レベル圧縮方式を組合せたものです。高レベルの方式では、現在の三角形のインデックスと、その前の三角形のインデックスとの間の共通関係を利用して圧縮します。低レベルの圧縮は、最適化されたインデックスバッファがもつ、安定して増加するという性質と、その差分が小さい数であるという事実を利用する、連続インデックス圧縮、ビット範囲圧縮、および差分圧縮です。圧縮結果は、典型的なインデックスバッファの場合、1つの三角形が約 7～8 ビットになります。

三角形の大多数は、その前の三角形のインデックスと、新しいインデックスとの、3つの異なる組合せのうちの1つです。この情報は、2ビットのストリーム（各三角形ごとに2ビット）に収まります。三角形が、前の三角形のインデックスの組合せのいずれかになっていない場合には、三角形の全インデックスが保存されます。新しいインデックスや圧縮されていない三角形は、差分圧縮されビットパックされた形式で独立に保存され、実行時には、2ビットストリームを解凍する前に、まずこれを解凍する必要があります。

その前の三角形と比べて新しいインデックスが1つしかない三角形は、以下の形で回転させても、画面上での表現は変わりません。

表 5

インデックス 0	インデックス 1	インデックス 2	回転	インデックス 0	インデックス 1	インデックス 2
新	旧	旧	$\wedge > 2$	旧	旧	新
旧	新	旧	$\wedge > 1$	旧	旧	新
旧	旧	新	$\wedge > 0$	旧	旧	新

また、この形に変換した三角形のその前の頂点組み合わせとして可能なのは、次の3通りだけです。

表 6

前のインデックス 1	前のインデックス 0	新しいインデックス
前のインデックス 0	前のインデックス 2	新しいインデックス
前のインデックス 2	前のインデックス 1	新しいインデックス

これにより、0～3 が圧縮された三角形の構成を表し、4 の値が圧縮されていない三角形を意味する、2 ビットのストリームが得られます。

表 7

00	前のインデックス 1	前のインデックス 0	新しいインデックス
01	前のインデックス 0	前のインデックス 2	新しいインデックス
10	前のインデックス 2	前のインデックス 1	新しいインデックス
11	新しいインデックス	新しいインデックス	新しいインデックス

新しいインデックスは、連続した要素が取り除かれた、別のテーブルに格納されます。次に、1ビットテーブルが生成され、残りのインデックスが差分圧縮され、さらにビット範囲圧縮されます。

この処理は、2ビット圧縮と連続した要素の除去を行った後のインデックスバッファから始まります。

表 8 2 ビット圧縮処理後の新しいインデックスバッファ

0	1	2	3	4	3	1	5
6	3	5	7	8	4	9	10

0	1	2	3	4	3	1	5
11	7	10	11	12	11	7	13
12	13	7	14	15	5	16	6
15	16	17	14	18	13	19	20
21	12	21	20	22	12	21	23

次に、1ビットのテーブルが生成され、連続するインデックスが取り除かれます。

表 9 連続インデックスの除去

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

ここから、以下のような値をもつ1ビットのストリームが生成されます。

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

次に、インデックス値が差分圧縮されます。SPU との相性をよくするために、最初の8つのインデックスは圧縮されず、その後のインデックスは、それぞれ8つ前のインデックスとの差分が計算されます。

表 10 差分圧縮されたインデックス

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

さらに、インデックスの最小値が計算されます。この例では-4になります。そして、この数が、あらゆるインデックスから差し引かれます。

表 11 最小値を差し引いた後

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

最後に、このインデックスバッファの値は、可能な限り最小のビット表現にパックされます。

この情報は、SPU で利用するために、単一のデータブロックに収められます。このブロックの中には、以下のようなデータが設定されます。

インデックス数	: U16 (下記のインデックステーブル中のインデックスの数を指定します)
ベース値	: U16
1ビットテーブルの数	: U16 (バイト)
1インデックス当たりのビット数	: U8
パディング	: U8
1ビットテーブル	: (8ビットにパディングされる)
2ビットテーブル	: (8ビットにパディングされる)
インデックステーブル	: ビット範囲圧縮および差分圧縮された、一意で不連続のインデックス

SPU 上での解凍は、以下の5段階で実装されます。

- (1) ビット範囲インデックス値をそのメモリ上で上書きしながらアンパックします。
- (2) 差分圧縮をそのメモリ上で上書きしながら解凍します。
- (3) 1ビットテーブルを一時メモリに解凍します。
- (4) 2ビットテーブルを一時メモリにコピーします。

(5) 2 ビットテーブルを解凍します。

インデックスは RSX®ネイティブのフォーマットではないので、SPU に渡して解凍し、RSX®互換フォーマットに変換する必要があります。

コマンドバッファホール

Edge ジオメトリの中で、SPU はコマンドバッファホールを生成し、データの場所やレンダリングする三角形の数に関する情報が得られないと決まらないコマンドが含まれています。

まず、PPU が、頂点フォーマット用の一次コマンドバッファにコマンドを配置します。それから PPU は、頂点属性配列アドレスおよび描画コマンド用に、4 ワードにアラインメントされたホールを残します。この時点では、出力配列のアドレスはわかっていません。SPU は、このホールを表 12 に示すように埋めます。

表 12

InvalidateVertexCache	(リングバッファが使われている場合)	: 32 バイト
SetVertexDataArray	: 位置	: 16 バイト
SetVertexDataArray	: 法線	: 16 バイト
SetVertexDataArray	: 接線	: 16 バイト
SetVertexDataArray	: テクスチャ座標	: 16 バイト
SetDrawIndexArray		: (インデックスの数 + 4607) / 48 バイト
SetWriteTextureLabel	(使用されているリングバッファごとに 1 つ)	: リングバッファの数 * 16 バイト
SetNopCommand		: ホールの残りの部分に 0 を書き込む

ホールの先頭位置、およびホールが 128 バイト境界を超えるたびにその位置に、cellGcmSetJumpCommand を使って「自分へのジャンプ」コマンドを挿入する必要があります。これは主に、RSX®をブロックして同期のコストを削減するためです。

このシステムの DMA は、アウトオブオーダー方式なので、RSX®が古いデータを読まないようにするためには、ホールが 128 バイト境界を越えるたびに「自分へのジャンプ」コマンドを挿入することは、きわめて重要です。

SPU入力DMAリスト

(CellSpursJob256 構造体の中にあるような) SPU 入力 DMA リストには、16KB ごとに分割されて個別にロードされるデータのそれぞれについてタグが必要です。タグの順序は重要です。

表 13

上位 32 ビット	下位 32 ビット
出力ストリーム記述の長さ	アドレス
インデックス A 長さ	アドレス
インデックス B 長さ	アドレス
スキニング行列 A 長さ	アドレス
スキニング行列 B 長さ	アドレス
スキニングインデックス/重み A 長さ	アドレス
スキニングインデックス/重み B 長さ	アドレス
頂点 A1 長さ	アドレス
頂点 B1 長さ	アドレス
頂点 C1 長さ	アドレス
頂点 A2 長さ	アドレス

上位 32 ビット	下位 32 ビット
頂点 B2 長さ	アドレス
頂点 C2 長さ	アドレス
EdgeGeomViewportInfo 長さ	アドレス
EdgeGeomLocalToWorldMatrix 長さ	アドレス
EdgeGeomSpuConfigInfo 長さ	アドレス
入力ストリーム固定小数点オフセットの長さ	アドレス
入力ストリーム記述の長さ	アドレス
一次入力ストリーム記述の長さ	アドレス

SPU 入カユーザデータ

SPU 入カユーザデータは、一般に変化する実行時パラメータに使われます。このデータは、PPU により各ジオメトリセグメントの中に設定され、CellSpursJob256 構造体中の上の DMA タグの直後に存在する必要があります。

表 14 SPU 入カユーザデータ

上位 32 ビット	下位 32 ビット	
	出力バッファ情報アドレス*	
	コマンドバッファホールアドレス	
BlendShapeInfos アドレス	blendshapes の数	カリングフレイバー
ユーザデータ 1		
ユーザデータ 2		

* このアドレスが直接出力バッファアドレスとして使われる場合には、出力バッファ情報アドレスパラメータの適切なビットが設定されます

SPU DMA タグのパフォーマンスに関する注意事項

SPU 処理用のデータは、DMA リストを介して入出力バッファにロードされます。各タグは、メインメモリのアドレス（EAL のみ）と長さを指定しますが、それは 16 バイトの倍数である必要があります。

単一のタグを使って 16KB 以上のデータをロードすることはできないので、16KB を越えるテーブルは、複数の部分に分割する必要があります。これはツールによって処理され、EdgeGeomPpuConfigInfo 構造体をベースにした分割サイズを、PPU 処理に提供します。

SPU LS(ローカルストア)のアドレスとメインメモリのアドレスに対して 0x70 と AND をとった結果が一致しない場合、DMA は効率的ではなくなります。このアドレスが一致していないデータをロードすると、必要なパケットの 2 倍のデータをロードしてしまうので、このことは重要です。これは、ツール使用時には解決できないので、実行時に解決する必要があります。Edge では、DMA はこの問題を解決するのに、ある単純な戦略を使っています。

次の例では、最初のデータが 0x130 から 0x20 バイトで、その次のデータが 0x470 から 0x110 バイトになっています。この場合、SPURS は自動的に以下の処理を行います。

- メインメモリの 0x100 から LS の 0x00 に 0x50 バイトをロードします。
- メインメモリの 0x450 から LS の 0x50 に 0x130 バイトをロードします。

この場合、絶対最小限の DMA が実行され、以後のロード時のオーバーヘッドは、タグ 1 つ当たり 0x70 バイトになる可能性があります。言い換えれば、オーバーヘッドは 0x70 バイトから始まり、どのテーブルが大きだろうが小さだろうが、テーブルごとに 0x70 バイトになります。けれども、小さいテーブルは、入力バッファ中のかなりの量の領域を無駄にします。このような場合、メモリバンド幅の一部と引き換えに領域を節約することも可能ですが、現在のライブラリではそれを行っていません。

SPURS ジョブの入出力バッファは 128 バイト境界にアラインメント済みなので、実行時の処理はきわめて単純です。

SPU全体のバッファレイアウト

SPU 処理では、2 つのバッファを使います。最大サイズ 48KB の入出力バッファと、頂点ごとの属性ユニフォームテーブルを含むスクラッチバッファです。

SPURS ジョブストリーマがパイプラインストールすることなく動作するには、着目するジョブの目のジョブの出力 DMA と、次のジョブの入力 DMA のためのバッファをローカルストア上に確保されている必要があります。その結果、特定のジョブが利用できる LS サイズは、前後のジョブの入出力バッファ用の 48KB を 235KB から引いた 186KB になります。

このバッファを通過するデータフローの基本的な順序は、以下のようになります。

- SPURS ジョブストリーマは、CellSpursJob256 構造体の中の DMA リストを介して、入出力バッファにあらゆるデータをロードします。このデータの中には、EdgeGeomSpuConfigInfo のような小さいデータテーブルも、頂点やインデックスデータのような大きなデータテーブルもあります
- 入出力バッファからスクラッチバッファに頂点データが解凍されます。
- スクラッチバッファの中のデータに対し、オプションでスキニングが実行されます。
- スクラッチバッファの頂点は、オプションでクリッピング空間に変換されます。
- 入力インデックスに対し、オプションでトライアングルカリング、またはオクルージョンカリングが実行され、入出力バッファ中の使用されたデータのすべてが上書きされます。
- そして、スクラッチバッファ中の頂点データが、入出力バッファに圧縮されます。
- データ出力の準備が済むと、出力データ用の領域が割り当てられます。詳細については、後のセクションにある「[ランタイム出力バッファの仕組み](#)」を参照してください。

SPUジョブのローカルストア管理

SPU ローカルストアの中の入出力バッファ用のメモリマネージャは、非常に単純です。常に「空き領域の先頭」へのポインタを意味するフリーポインタと、もう 1 つのポインタが保持されます。バッファは、開始時に割り当てられます。この「先頭」が以前の値に設定されると、メモリが解放されることになります。たとえば、まず以下のように A、B、C、D を割り当てたとします。

- A
- B
- C
- D

続いて、「空き領域の先頭」ポインタを保存し、以下のように E、F を割り当てます。

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

最後に、「空き領域の先頭」ポインタを保存された値に設定すると、E、Fによって使われていた領域が解放されます。

- A
- B
- C
- D

表 15 および 表 16 は、処理中のさまざまな段階における入出力バッファの内容です。新たに割り当てられる領域は、**太字**になっています。SPU処理中のスクラッチバッファの割り当ては、すべて完全に静的です。

表 15 処理中のさまざまな段階におけるバッファの内容
初期化からスキニング状態まで

出力ストリーム記述	出力ストリーム記述	出力ストリーム記述	出力ストリーム記述	出力ストリーム記述
圧縮コード	圧縮コード	圧縮コード	圧縮コード	圧縮コード
インデックス	インデックス	インデックス	インデックス	インデックス
スキン行列	スキン行列	スキン行列	スキン行列	スキン行列
インデックス/重み	インデックス/重み	インデックス/重み	インデックス/重み	インデックス/重み
頂点	頂点	頂点	形状ブレンド	
解凍ストリーム記述	解凍ストリーム記述			
ビューポート情報	ビューポート情報			
LocalToWorld	LocalToWorld			
EdgeGeomSpuConfigInfo	EdgeGeomSpuConfigInfo			
固定小数点オフセット	固定小数点オフセット			

表 16 処理中のさまざまな段階におけるバッファの内容（続き）
インデックス解凍からジョブの終了まで

インデックスの解凍	カリング	圧縮	コマンドバッファの書き込み	ジョブの終了
出力ストリーム記述	出力ストリーム記述	出力ストリーム記述	出力ストリーム記述	
解凍されたインデックス	解凍されたインデックス	解凍されたインデックス	解凍されたインデックス	
		圧縮された頂点	圧縮された頂点	
			VRAM リードバック用の 16 バイト	
			コマンドバッファホール	

ユーザ機能およびデータ

ユーザコードが必要なデータは、入出力 (IO) バッファ中のジオメトリデータの前後のどちらにあってもかまいません。どちらの方法でも使用可能です。

データは、SPURS によってユーザタグにより DMA を行う場合には入出力バッファより前におく必要があります。ツールの中で 48KB 未満の入出力バッファエリアを指定することにより、処理中に DMA を行う場合には、出力バッファより後に、データをおく必要があります。

たとえば、風向きや部分的なボーン情報を指定するカスタム情報を、DMA を使って取り込みたいとします。この場合、まず、データ用の領域をツールの中に予約する必要があります。次に、データに対して DMA を行う PPU 処理上で、ユーザタグの 1 つを設定する必要があります。最後に、SPU 処理時のスキニング処理の後で、ユニフォームテーブルの頂点位置に対して正弦・余弦関数を適用します。例：

- `edgeGeomInitialize()` を使って、Edge ジオメトリライブラリを初期化します。
- `edgeGeomDecompressVertexes()` を使って、入力頂点データストリームを I0 バッファからユニフォームテーブルに解凍します。
- `edgeGeomProcessBlendShapes()` を使って、ユニフォームテーブルにブレンド形状を適用します。
- `edgeGeomSkinVertexes()` を使って、ユニフォームテーブルにスキニング行列を適用します。
- ここで、位置が風の中で揺れ動いているように見えるようにするために、ユニフォームテーブル中の位置を手続的に変更する、ユーザ独自の関数を呼び出します。
- `edgeGeomCullTriangles()` そして/または `edgeGeomCullOccludedTriangles()` を使って、インデックスバッファを上書きし、新しいインデックスバッファを生成します。
- インデックスバッファがどのくらいの大きさになるかがわかったので、`edgeGeomAllocateOutputSpace()` を実行して、出力データ用の領域を予約します
- `edgeGeomOutputIndexes()` を使って、先程割り当てたメインメモリ中の出力バッファに対し、新たに生成されたインデックスの出力を開始します。
- `edgeGeomCompressVertexes()` を使って、ユニフォームテーブルを圧縮して RSX®フォーマットに変換し、RSX®で利用できるようにします
- `edgeGeomOutputVertexes()` を使って、先程割り当てたメインメモリ中の出力バッファに対し、新たに生成された頂点の出力を開始します。
- `edgeGeomBeginCommandBufferHole()` を使って、コマンドバッファホール用のデータの生成を開始します。
- `edgeGeomSetVertexDataArrays()` を使って、属性アドレスとフォーマットを書き込みます。
- `cellGcmSetDrawIndexArray()` を使って、描画コマンドを書き込みます。
- `edgeGeomEndCommandBufferHole()` を使って、書き込み終わったホールを、メインメモリに出力します。

これ以外の例として、必要スペースを最小に維持するために、カスタムポーズ領域の変更を行い、比較的大量のデータを DMA 転送する必要のある場合もあります。その手順は以下のとおりです。

まず、データ用の領域をツールの中に予約します。次に、ユーザデータの一部を設定して、情報を DMA 転送する場所を SPU に知らせます。最後に、SPU 処理中のスキニング処理の前に、ジオメトリライブラリが使用している入出力バッファ空間の後の未使用の領域にデータを DMA 転送し、ユニフォームテーブル中の頂点位置に PSD 関数を適用します。例：

- `edgeGeomInitialize()` を使って、Edge ジオメトリライブラリを初期化します。
- `edgeGeomDecompressVertexes()` を使って、入出力バッファからユニフォームテーブルに頂点を解凍します
- `edgeGeomProcessBlendShapes()` を使って、ユニフォームテーブルにブレンド形状を適用します。
- ここで、PSD 設定情報を DMA 転送して、PSD についてループを行い、各 PSD に該当する情報を DMA 転送し、PSD 関数にしたがってユニフォームテーブルを手続的に変更する、独自の関数を呼び出します。
- `edgeGeomSkinVertexes()` を使って、ユニフォームテーブルにスキニング行列を適用します。
- `edgeGeomCullTriangles()` そして/または `edgeGeomCullOccludedTriangles()` を使って、インデックスバッファを上書きし、新しいインデックスバッファを生成します。
- インデックスバッファがどのくらいの大きさになるかがわかったので、`edgeGeomAllocateOutputSpace()` を実行して、出力データ用の領域を予約します

- `edgeGeomOutputIndexes()` を使って、先程割り当てたメインメモリ中の出力バッファに対し、新たに生成されたインデックスの出力を開始します。
- `edgeGeomCompressVertexes()` を使って、ユニフォームテーブルを圧縮して RSX®フォーマットに変換し、RSX®で利用できるようにします
- `edgeGeomOutputVertexes()` を使って、先程割り当てたメインメモリ中の出力バッファに対し、新たに生成された頂点の出力を開始します。
- `edgeGeomBeginCommandBufferHole()` を使って、コマンドバッファホール用のデータの生成を開始します。
- `edgeGeomSetVertexDataArrays()` を使って、属性アドレスとフォーマットを書き込みます。
- `cellGcmSetDrawIndexArray()` を使って、描画コマンドを書き込みます。
- `edgeGeomEndCommandBufferHole()` を使って、書き込み終わったホールを、メインメモリに出力開始します。

さらに、SPURS ジョブの動作に関連して、インバウンド DMA の効率性に影響する問題があります。任意のインバウンド DMA トラフィックの入力が可能になるためには、他の入力バッファが入力を完了するのを待つ必要があり、これが、システムが最大限の効率を発揮することを阻んでいます。

ランタイム出力バッファの仕組み

Edge ジオメトリでは、SPU からの出力を保持するために、4 つの異なる出力方式があります。その方式を、一般的な重要性の順に並べると、以下のようになります。

- ダブルバッファリング
- シングルバッファリング
- リングバッファリング
- ハイブリッドバッファリング

出力バッファ方式は、構造体 `EdgeGeomOutputBufferInfo`、`EdgeGeomRingBufferInfo`、および `EdgeGeomSharedBufferInfo` を使って設定することができます。この 3 つの構造体に関する詳しい情報は、「PlayStation®Edge ジオメトリライブラリリファレンス」ドキュメントにあります。

ダブルバッファリング

ダブルバッファは、最も単純な方式です。SPU からのデータ出力は、フレームごとに、2 つのバッファのうちの 1 つに転送され、次のフレームのレンダリング時に使用されます。この方式では、RSX®は次のフレームでただちに全データを利用できるので、SPU/RSX®間の同期は不要です。

長所：

- 単純さ。このバッファ方式は、非常に簡単です。
- SPU の効率を最大化。SPU は、ストールすることなく、可能な限り高速にデータを処理します。

短所：

- 著しいメモリ使用量。出力バッファのサイズは、2 つとも、ゲーム中にレンダリングされる最も複雑なフレームに必要な領域と、最低でも同じくらいの大きさが必要です。また、一部のゲームでは、SPU 出力サイズの合計を予測するのが難しいため、バッファのサイズが著しく過大になってしまうことがあります。

- 割り当てに失敗すると、グラフィカルエラーが発生します。ユーザが提供したバッファの大きさが不足していると、ゲームシーンの一部が描画されないことになるので、誰が見ても明らかなグラフィカルエラーが発生します。

シングルバッファリング

シングルバッファは、ダブルバッファによく似ています。シングルバッファでは、SPU からのデータ出力は、フレームごとに、1つのバッファに転送され、現在レンダリング中のフレームで使用されます。この場合、ダブルバッファリングとは異なり、RSX®がSPUの処理を追い越す可能性があるため、SPUとRSX®間の同期が必要です。

長所：

- 単純さ。このバッファ方式は、非常に簡単です。
- SPUの効率を最大化。SPUは、ストールすることなく、可能な限り高速にデータを処理します。

短所：

- 著しいメモリ使用量。出力バッファのサイズは、少なくとも、ゲーム中にレンダリングされる最も複雑なフレームに必要な領域と同じの大きさが必要です。また、一部のゲームでは、SPU出力サイズの合計を予測するのが難しいため、バッファのサイズが著しく過大になってしまうことがあります。
- 割り当てに失敗すると、グラフィカルエラーが発生します。提供されたバッファの大きさが不足していると、ゲームシーンの一部が描画されないことになるので、誰が見ても明らかなグラフィカルエラーが発生します。

リングバッファリング

リングバッファは、シングルバッファリングのメモリ効率をより高くした形式です。リングバッファでは、SPUからのデータ出力は、フレームごとに、1つのバッファに転送され、現在レンダリング中のフレームで使用されます。シングルバッファリングと同じように、SPU・RSX®間の同期が必要です。リングバッファでも、RSX®がメモリの一部を使用した可能性があるため、SPUはそれをRSX®ラベルを通じて知らされ、後でそのメモリをそのフレームのレンダリングに再利用することが可能です。このようなバッファの再利用は、出力バッファに必要なメモリ使用量の大幅な削減に結びつきます。

長所：

- メモリ効率。全体的なメモリ使用量が著しく削減されます。リングバッファの容量は事実上無限なので、実際のバッファサイズを大幅に小さくすることができます。
- 正確さ。バッファ領域の不足のせいでデータが破棄されることはありません。したがって、バッファオーバーフローに起因するグラフィカルエラーの可能性はなくなります。

短所：

- SPUのパフォーマンスが低下する可能性があります。リングバッファが小さすぎるか、RSX®のデータ処理が遅すぎる場合、RSX®の処理を待つ間、SPUはストールすることになります。
- バッファのどの部分が使われたかをSPUに知らせるために必要なRSX®ラベルのため、RSX®のパフォーマンスは、平均で約1.5%低下します。

ハイブリッドバッファリング

ハイブリッドバッファリングは、シングルバッファリングよりはメモリ効率がよく、リングバッファリングよりはメモリ効率が悪いのですが、リングバッファの持ついかなる短所も持っていません。ハイブリッ

ドバッファリングでは、単一のシングル共有バッファと、SPU ごとのリングバッファが使われます。ほとんどの割り当ては、リングバッファによって実現され、共有バッファは、リングバッファが一時的に一杯になった際の、緊急オーバーフロー領域として使われます。

ハイブリッドバッファリングにおける、SPU の割り当てロジックは以下のようになります。

- (1) リングバッファをチェックして、指定された割り当てを行うための領域が存在するかどうか調べます。存在する場合には、リングバッファから割り当てを行い、リターンします。存在しない場合には、手順 2 に進みます (ブロックして、RSX®がさらにデータを消費するのを待たないでください)。
- (2) 最初のリングバッファからの割り当てに失敗した場合には、共有バッファをチェックして、指定された割り当てのための領域が存在するかどうかを調べます。存在する場合には、(通常のアトミックなロック機構によって) 共有バッファから割り当てを行い、リターンします。存在しない場合には、手順 3 に進みます。
- (3) 共有バッファが一杯の場合には、再度リングバッファをチェックします。今度は、割り当てが実現されるまで、ブロックします。

この方法により、RSX®の待ち時間として使われる時間が減るので、SPU の効率が改善されます。また、共有バッファの割り当てパスが使われた場合、リングバッファの RSX®ラベルは不要になるので、RSX®上でのフレーム時間全体の約 1.5%が節約されます。他方、余分なバッファのため、メモリの使用量は微妙に増加します。どのバッファ戦略がアプリケーションに最適かの判断は、ディベロッパ次第です。

長所：

- メモリ効率。全体的なメモリ使用量が著しく削減されます。リングバッファの容量は事実上無限なので、実際のバッファサイズを大幅に小さくすることができます。
- 正確さ。バッファ領域の不足のせいでデータが破棄されることはありません。したがって、バッファオーバーフローに起因するグラフィカルエラーの可能性はなくなります。
- 効率性。RSX®ラベルを書き込むための RSX®コストを、著しく削減もしくは排除することができます。SPU のストールは、オーバーフローバッファがすべて使い果たされない限り発生しないので、SPU ストールの確率が低下します。

短所：

- 共有バッファ用に、大きなリングバッファ (やその他のデータ) 用の追加のメモリが使われます。

EdgeジオメトリのSPU処理コストの見積り

表 17 は、各機能におよそ何サイクルかかるかの一覧です。

表 17 機能およびその推定サイクルコスト

機能	推定サイクルコスト
解凍	1 頂点の 1 属性当たり 8 サイクル
形状ブレンド	1 ブレンド形状の 1 頂点の 1 属性当たり 8+4 サイクル
スキニング	1 頂点の 1 属性当たり 16+8 サイクル
オクルージョンカリング	(11 + (11 サイクル/オクルーダー)) / 頂点 + 12 サイクル/三角形
トライアングルカリング	11 サイクル/三角形 + 16 サイクル/頂点
圧縮	1 頂点の 1 属性当たり 8 サイクル

*属性が法線であり、スキニングモードが不均一なスケーリングである場合には、その属性に対する推定コストは、1 頂点につき 1 属性につき 17 サイクルです。

サイクルコストは、属性の型に強く依存していることに注意してください。属性ごとのコスト見積りの内訳は、表 18、表 19 のようになります。

表 18

解凍する属性の型	頂点ごとのサイクルコスト見積り
I16N	5.5 サイクル
F32	6 サイクル
F16	9 サイクル
U8N	4.5 サイクル
I16	5.5 サイクル
X11Y11Z10N	6 サイクル
U8	4.5 サイクル
SPU のみの単位ベクトル	8.5 サイクル
SPU のみの固定小数点	8 サイクル

表 19

圧縮する属性の型	頂点ごとのサイクルコスト見積り
I16N	5.5 サイクル
F32	6 サイクル
F16	9 サイクル
U8N	5.5 サイクル
I16	5.5 サイクル
X11Y11Z10N	7 サイクル
U8	5.5 サイクル

5 Edgeオフラインジオメトリツール

Edge オフラインジオメトリコンパイラツール (edgegeomcompiler) は、libedgegeomtool をアセットパイプラインの中でどのように利用できるかを示すサンプルツールです。具体的には、COLLADA™フォーマットのデータを取り込んで、Edge ライブラリに渡し、ランタイムサンプルにロードできるバイナリファイルを作成する方法を示します。

ビルド

edgegeomcompiler は、libedgegeomtool、FCollada、libxml2 の 3 つのライブラリに依存しています。

FCollada は、Feeling Software (www.feelingsoftware.com) から無料で入手できる、COLLADA™データを読み書きするためのライブラリです。このライブラリは、Feeling Software のウェブサイトからダウンロードできます (要登録)。必要な FCollada のバージョンは 3.05B です。

libxml2 は、無料で入手できる xml フォーマットのデータを処理するためのライブラリです。このライブラリは、FCollada により、XML ファイルである COLLADA™ファイルへの低水準アクセスを処理するために使われます。libxml2 (の特定のバージョン) は、FCollada ディストリビューションにも含まれており、<http://xmlsoft.org/> からダウンロードすることもできます。

使用法

edgegeomcompiler は、単純なコマンドラインツールで、入力 COLLADA™ファイル名、および出力バイナリファイル名という 2 つの引数とともに呼び出されます。

```
edgegeomcompiler [オプション] <入力> <出力>
```

オプション:

- `--enable-scaled-skinning`: edgegeomcompiler は、スキニングデータが利用できる場合、スキニングフレイバーをデフォルトで `EDGE_GEOM_SKIN_NO_SCALING` に設定します。この設定では、ランタイムの中でも最高速のスキニングコードが使われますが、アニメーション変換にスケールファクターが含まれている場合には、誤った結果が生成されます。特定のシーンにどのアニメーションが適用されるかは、edgegeomcompiler にはわからないので、シーンのスキニングフレイバーを必要に応じて強制的に `EDGE_SKIN_NON_UNIFORM_SCALING` に設定するために、このオプションが用意されています。
- `--inv-bind-mats-out <file>`: 逆バインドポーズ行列の配列を出力するファイルを指定します。この行列は、3x4 の転置形式で書き出されます。

6 Edgeアニメーションの概要

Edgeアニメーションの設計

Edge アニメーションは、低水準のスケルトンアニメーションシステムで、SPU コードとリンクするための、軽くて効率的な SPU ライブラリとして設計されています。

このシステムは、PPU インタフェースの背後にラップ、もしくは、抽象化されていないので、容易にゲーム固有のカスタマイズを行うことができます。SpuMain() や、Edge もしくはゲーム固有のコードによって提供されるさまざまなアニメーションシステムがどのように統合されるかは、ユーザが制御することができます。

このような柔軟性および制御機能は、ユーザが SPU の上で、アニメーションジョブの一部としてカスタムゲームコードを簡単に実行できることを意味します。

Edge アニメーションシステムは、ジョイント数 80 程度の、単純でないブレンドツリーをもつ一般的なキャラクターを想定して設計されています。

SPUの使い方

サンプルでは SPURS ジョブを利用しており、このコードをコピーして、Edge アニメーションを自分のプロジェクトに組み込むための叩き台にすることができます。

このライブラリ自体は、いかなる SPU プログラミングモデルをも前提としていません。

- コードは、位置とは独立にコンパイルされますが、固定位置モデルで使うこともできます。
- Edge アニメーションの中で実行される DMA 操作は、すべて割り込み安全です。したがって、このライブラリは、SPU の割り込みを有効にした状態で利用できます。
- このライブラリ中には、SPURS/SPU スレッド関数呼出しは存在しません。また、このライブラリは、Raw SPU 上でも利用できます。

SPU ライブラリ側で要求する唯一の前提条件は、PPU 側で、spu_printf() と互換性のある printf ハンドラが利用できるようなっていることです。spu_printf() は、通常の処理中に呼び出されることはなく、呼び出されるのは、クリティカルなエラーが発生したときだけです。

Edge アニメーションの PPU 側で実行されるのは、一部のセットアップ関数だけです。あらゆる処理は、SPU 上で実行されるようになっています。

リファレンス実装

クロスプラットフォームのリファレンス実装も用意されています。このソースコードは、Edge アニメーションの機能をデモするためのもので、パフォーマンスのためには最適化されていません。

Win32 の実装

また、win32 用の実装も提供されます。PPU 側および SPU 側の両方の関数は、単一のライブラリで公開されます。ライブラリは、リファレンス実装、または、SSE2 が有効な CPU に最適化されたメソッドのいずれかを利用できます。

アニメーションとスケルトンのデータ構造は、PlayStation®3 バージョンのものに似ていますが、ビッグエンディアンではなくリトルエンディアンで保存されます。ファイルは、edgeanimcompiler の -pc コマンドラインスイッチを使って、PC（リトルエンディアン）モードでエクスポートする必要があります。

アニメーション処理

アニメーションポーズスタック

アニメーション処理では、ブレンドツリー評価の際に、ポーズスタックを利用します。スタックの要素数は、ユーザに渡されたスクラッチメモリの量や、スケルトンのサイズ（ジョイントの数、ユーザチャンネルの数、その他）によって異なります。このポーズスタックには、ローカルストア中の領域を使い果たした場合に、(SPU ごとの一時記憶を使って) メインメモリを透過的に利用するためのサポートが含まれています。

アニメーションの全体的・部分的な評価

アニメーションについては、全体アニメーションと部分的なアニメーションがサポートされています。部分的なアニメーションは、部分的なアニメーションに影響を受けないジョイントに重み 0x00 を割り当てることにより、内部的に処理されます。

アニメーションストレージでは、圧縮クォータニオンデータを持つ不均一なキーフレームを使います。

重み付きアニメーションブレンディング

あらゆるブレンド演算は、EdgeAnimJointTransform によりローカル空間で動作し、ジョイント重みを介して部分的なアニメーションを処理するロジックを持っています。

付加的なアニメーション

付加的な差分アニメーション（通常、物体の衝突反応などで使われる）が、ツールパイプライン、およびランタイムでサポートされます。付加的なブレンド指定は、ベース、およびベース+デルタの間で機能します。

ジョイント変換・行列変換関数

以下のような、任意の方向へのあらゆる変換を実行する、ユーティリティ関数の包括的セットが公開されています。

- ローカル空間ジョイント変換からワールド空間行列（標準の 4×4 行列、もしくは 3×4 の転置行列）へ
- ローカル空間ジョイント変換からワールド空間ジョイント変換へ
- ワールド空間ジョイント変換からローカル空間ジョイント変換へ
- ジョイント変換から行列（標準の 4×4 行列、もしくは 3×4 の転置行列）へ
- 行列（標準の 4×4 行列、もしくは 3×4 の転置行列）からジョイント変換へ

親スケールの補償は、スケルトンのジョイント階層データ構造に適切なフラグを設定することにより、ジョイントをローカル空間からワールド空間（およびその逆）に変換する際に実行できます。

スカラーユーザチャンネル

ユーザは、任意のスカラーを表すアニメーションチャンネルにアクセスすることができます。スカラーチャンネルは、(形状ブレンドのような) 任意のカスタム処理を実行するために利用できます。

カスタムユーザコールバック

複合関数は、カスタムユーザ処理を可能にするコールバックを公開しています。「locomotion」のサンプルでは、キャラクタの移動を、`edgeAnimProcessBlendTree()` の中で使われているコールバックによって処理する方法を示しています。

ツール処理

Edge アニメーションは、COLLADA™と互換性があります。Edge アニメーションは、既存のパイプラインに組込むため、以下のようなさまざまなレベルで利用できるようになっています。

- 高レベル：スタンドアロンの実行可能ファイル `edgeanimcompiler` のまま
- 中レベル：COLLADA™フレームワークを利用する修正された `edgeanimcompiler` ツール
- 低レベル：直接 `libedgeanimtool` を使って最終データを生成するカスタムツール

`edgeanimcompiler` は、FCollada を利用します。大部分の作業を行う最下位レベルの `libedgeanimtool` ライブラリは、COLLADA™に依存していません。

データ構造体

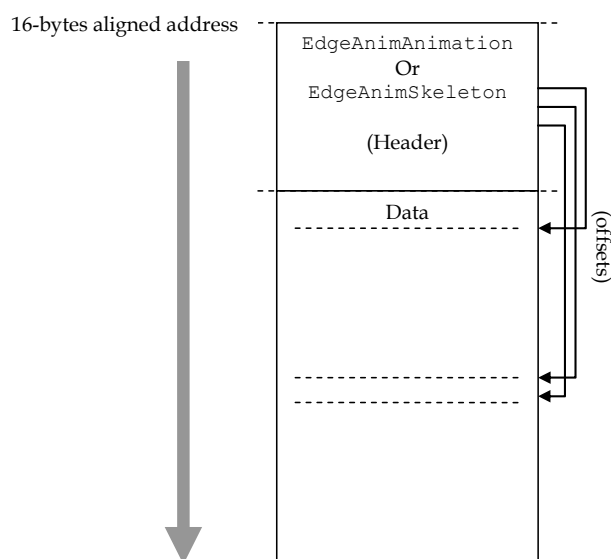
オフラインツールでビルドされるデータ構造は、すべて、メインメモリ上では 16 バイト境界でアラインメントされた単一のブロックとしてロードされるように設計されています。このようなデータ構造は、ヘッダと、その後続く可変サイズのデータチャンクから構成されています。このヘッダでは、位置に依存せずにデータを参照するために、ポインタではなくオフセットを使います。

ツールで構築される基本的なデータ構造には、次の 2 つがあります。

- `EdgeAnimSkeleton`: スケルトン階層、バインドポーズ、それにジョイント名用のハッシュされた文字列、その他が含まれます
- `EdgeAnimAnimation`: ツールでのビルド時にスケルトンにバインドされた単一のアニメーションが含まれます。

また、これらのデータ構造には、それぞれ 4 バイトのバージョンタグも含まれています。このタグは、フォーマットが変更されるたびにインクリメントされます。

図 1 Edge アニメーションのデータ



7 Edgeアニメーションの利用

この例は、Edgeアニメーションの使用がいかに簡単なものであるかを示しており、第23章「[サンプルプログラム](#)」の「[Edgeジオメトリおよびアニメーション](#)」セクションの「`samples/edge/character-sample`」に似ています。

PPUーグローバル

(1) 初期化

`edgeAnimPpuInitialize()` を呼び出してライブラリを初期化すること。EdgeAnimPpuContext 構造体には、メインメモリ上の各 SPU 用一時記憶領域に関する情報が含まれています。

(2) データのロード

スケルトンおよびアニメーションデータの構造体をロードします。この構造体は、16 バイト境界にアラインメントされている必要があります。外部ツールで書き込まれたデータ構造はメモリに直接ロードできるように設計されており、リロケーションや修正は不要です。

(3) メインループの開始

(4) アニメーションジョブのセットアップ

これはユーザ側のコードですが、SDK に付属する、SPURS ジョブに基づいたサンプルを参考にすることができます。最も単純な場合、SPU ジョブに送信する必要があるのは、以下の通りです。

- スケルトン
- ブレンドツリー
- 最終データの書き込み先

Edge アニメーションの設計上、すべてがユーザに公開され、SPU 側のライブラリは、高レベルのゲームコードとリンクさせることができます。

より高度な利用法では、ジョブに高レベルのゲームデータを指定して、SPU のローカルストア内に直接ブレンドツリーを構築する必要があります。

(5) ジョブの実行

SPURS ジョブを使う場合、通常の `job_send_end` によって同期を行うことができます。

(6) メインループの終了

(7) 終了処理

`edgeAnimPpuFinalize()` を呼び出して、`libedgeanim` を終了します。これにより、ライブラリに割り当てられたリソースも解放されます。

SPUー各ジョブ用

(1) 初期化

`edgeAnimSpuInitializ()` を呼び出して、SPU ライブラリを初期化します。この `EdgeAnimSpuContext` 構造体は、以後の `edgeAnimSPU` 関数呼出しのすべてに渡されます。ライブラリ内部で静的に保存される情報ははありません。

`edgeAnimSpuInitialize()` は、メモリ管理操作のほとんどを、ユーザによって提供されるスクラッチバッファを使って行います。

- この関数は、スクラッチ領域をできるだけ効率的に利用するために、スケルトン情報にしたがってポーズスタックを設定します。
- `edgeAnimSpuInitialize()` は、`EdgeAnimPpuContext` 中の情報、および呼出し元によって渡された一意の SPU ID に従って、メインメモリ内の外部記憶を設定します。

(2) ブレンドツリーの処理

`edgeAnimProcessBlendTree()` は、ブレンドツリーを処理して、その最終結果をポーズスタックの一番上に置きます。

この時点では、データはローカル空間内にあり、`EdgeAnimJointTransform` (回転/変換/スケール) 構造体に保存されます。

(3) 変換

ユーザコードは、スタックの最上位にあるポーズを取得して、必要な形式に変換することができます。典型的な方法は、以下の通りです。

- (1) ワールド空間に変換します

```
edgeAnimLocalJointsToWorldJoints()
```

- (2) 行列に変換します

標準の `Vectormath::Aos` ライブラリと互換性のある `edgeAnimJointsToMatrices4x4()` を使うか、もしくは Edge ジオメトリスキニングで使う場合には `edgeAnimJointsToMatrices3x4()` を使う

また、ジョイントを直接ワールド行列に変換することもできます。

- 直接ワールド空間行列に変換するには、次のようにします。

```
標準 Vectormath::Aos ライブラリと互換性がある
edgeAnimLocalJointsToWorldMatrices4x4() を使う
もしくは、Edge ジオメトリスキニングで使う場合には
edgeAnimLocalJointsToWorldMatrices3x4() を使う
```

直接ワールド空間行列に変換した場合、その行列は、計算済みの親ワールド行列から掛け算されるので、最初の方法と同じ結果をもたらしません。この方法では、キネマティックスチェインの中でローカルなスケールの方向が維持されることが保証されます。このことは、最初の方法には当てはまりません。最初の方法では、ワールドスケールの値がジョイントのワールド方向にあるからです。

(4) 結果の利用

結果を DMA 転送するか、もしくは、ユーザの SPU コード内で直接利用します。

(5) 終了処理

`edgeAnimSpuFinalize()` を呼び出して、SPU ライブラリを終了します。

8 Edgeアニメーションのランタイムの詳細

ポーズスタック

アニメーションポーズは、スケルトンからの情報によって定義されます。アニメーションポーズは、以下のデータから構成されます。

- `EdgeAnimJointTransforms` の配列（サイズはジョイント4つの倍数にアラインメント）
- ジョイント重みの配列。 `uint8_t` に固定小数点数として格納。
- ユーザチャンネルの配列。スカラー浮動小数点数として格納。

このポーズスタックでは、ローカルポーズスタックがオーバーフローしたときに、ローカルストアバッファ、`edgeAnimSpuInitialize()` に渡されたスクラッチメモリの一部、および `edgeAnimPpuInitialize()` の中で SPU ごとに割り当てられたオプションのメインメモリバッファを使います。

このような外部記憶は、性能コストを犠牲にするので、多用することは想定されていませんが、柔軟性を増すためには役立ちます。このような外部記憶は、通常より大きなブレンドツリーの処理や、デバッグビルド用に SPU コードを増やすことを可能にします。

ローカルストア/外部記憶で利用できるスロットの数は `edgeAnimSpuInitialize()` によって計算され、スケルトンの中で定義されたジョイントやユーザチャンネルの数に依存します。

このスタックは、以下の3つの関数を経由して公開されます。

- `edgeAnimPoseStackPush()`
 - スタックの上に、ポーズを1つ追加（プッシュ）します。
 - システムがローカルストア領域を使い果たしそうな場合には、最も最近使われていない（LRU）ポーズを、SPUに関連付けられたメインメモリ上の外部記憶に、DMA 出力します。システムがメインメモリ上のローカルストア領域も使い果たした場合には、SPU のランタイムがアサートが発生します。
- `edgeAnimPoseStackPop()`
 - スタックの上のポーズを、1つ破棄（ポップ）します。
 - メインメモリ上の外部記憶に保存された最後のポーズがあれば、SPU ローカルストアに DMA 転送します。
- `edgeAnimPoseStackGetPose()`
 - `EdgeAnimPoseInfo` 構造体を、ポーズの別の配列へのポインタで更新します。

ブレンドツリーの処理後にスタックに残るポーズは1つだけです。

ブレンドツリーの処理

`edgeAnimProcessBlendTree()` の呼び出しは、次の2つの段階で処理されます。

- (1) ブレンドツリーが、線形のコマンドリストに再帰的に変換されます。
- (2) そして、このリストを処理するために、内部的に `edgeAnimProcessCommandList()` が呼び出されます。DMA 待ち時間は、3段階のパイプラインに隠されます。

ブレンドツリーは、以下のデータから構成されます。

- リーフ。EdgeAnimBlendLeaf の配列として保存される。リーフは、特定の時刻に評価されるアニメーションです。また、その中には、評価コールバック内のユーザ処理用のユーザ uint32_t も含まれています。
- ブランチ。EdgeAnimBlendBranch の配列として格納されます。ブランチは、2 つのツリー要素（他のブランチやリーフ）の間のブレンド操作です。また、その中には、評価コールバック内のユーザ処理用のユーザ uint32_t も含まれています。

ブレンドツリーの中で使われるインデックスは、すべて次の 2 つのフラグのいずれかとのビット OR をとって使う必要があります。

- リーフを表す場合：(index | EDGE_ANIM_BLEND_TREE_INDEX_LEAF)
- ブランチを表す場合：(index | EDGE_ANIM_BLEND_TREE_INDEX_BRANCH)

どちらの場合も、インデックス 0 は配列の最初の要素です。どちらのフラグ（もしくは両方）もセットされていない場合には、SPU のランタイムでアサートが発生します。

コマンドリスト

このツリーは、内部的に、コマンドリストに変換されます。各コマンドにはブレンドツリーノード（EdgeAnimBlendLeaf、EdgeAnimBlendBranch）へのポインタが格納されていますが、このポインタは、ユーザが直接利用するようには設計されていないことに注意してください。

可能なコマンドは次のものです。

- EDGE_ANIM_CMD_EVAL（「リーフ」ノードへのマップ）
- EDGE_ANIM_CMD_PUSH_AND_EVAL（「リーフ」ノードへのマップ）
- EDGE_ANIM_CMD_BLEND_AND_POP（「ブランチ」ノードへのマップ）
- EDGE_ANIM_CMD_MIRROR
- EDGE_ANIM_CMD_END_LIST

このコマンドリストは、DMA 待ち時間を隠すために、3 つのステージのパイプラインで処理されます。異なるコマンドリストの間でのパイプライン化はありません。つまり、最初の DMA 評価がストールを伴うということです。このシステムは、複雑なブレンドツリーを想定しているので、評価・ブレンドの対象となるアニメーションが 2 つしかない場合には最適ではありません。

表 20 パイプラインのステージ

ステージ	EDGE_ANIM_CMD_PUSH_AND_EVAL	EDGE_ANIM_CMD_BLEND_AND_POP
-2（プレロード）	アニメーションヘッダの DMA を開始します。	ノーオペレーション
-1（ロード）	アニメーションヘッダの DMA が完了するのを待ちます。 評価時に必要なフレームセットを発見して、DMA を開始します。	ノーオペレーション
0（実行）	フレームセットの DMA が完了するのを待ちます。 評価します。	ブレンドします。

ブレンディング操作と部分的なアニメーションのロジック

ブレンド操作

ブランチは、EdgeAnimBlendOp 列挙型で定義された複数のブレンディング操作を実行することができます。

また、加減算アニメーションもサポートされます。加減算操作は、表 21 に記載されているように、チャンネルタイプによって異なる意味を持ちます。

表 21 加減算アニメーション

チャンネルタイプ	Add (a, b)	Sub (a, b)
回転	$a * b$	$a * \text{conjugate}(b)$
平行移動	$a + b$	$a - b$
スケール	$a * b$	a / b
ユーザチャンネル	デフォルトモード: $a + b$ 最小最大モード: $\max(a, b)$	デフォルトモード: $a - b$ 最小最大モード: $\min(a, b)$

また、各アニメーションには、ブレンド加重値として使われる uint8_t (符号なし 8 ビット固定小数点整数) の配列も含まれています。この配列は、ブレンドツリー全体で部分的なアニメーションをサポートするためにも使われます。

重み値ゼロは、そのジョイントが未定義であることを意味します。この値は、部分的なアニメーションを利用する際には一般的です。

注意:

- ジョイント重みはブレンディング関数によって書き込まれるので、ポーズキャッシュやスタック中の各ポーズは、ジョイント重みの配列を持っている必要があります。
- アニメーション中の領域を節約するために、jointWeight が NULL の場合には、すべての重みが 1 (0xFF) に設定されたフルボディアニメーションであると仮定されます。

表 22 は、可能なブレンド操作のすべてを示しています。

表 22 EdgeAnimBlendOp 操作

EdgeAnimBlendOp	操作
EDGE_ANIM_BLENDOP_BLEND_LINEAR	以下のブレンド: 左 [$\alpha = 0.0$] 右 [$\alpha = 1.0$]
EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_RIGHT	以下のブレンド: 左 [$\alpha = 0.0$] Add(左、右) [$\alpha = 1.0$]
EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_LEFT	以下のブレンド: 右 [$\alpha = 0.0$] Add(右、左) [$\alpha = 1.0$]
EDGE_ANIM_BLENDOP_COMPOSE_ADD	合成: Add (左、右) [両方が指定された場合] それ以外の場合は、左もしくは右
EDGE_ANIM_BLENDOP_COMPOSE_SUB_RIGHT_FROM_LEFT	合成: Sub (左、右) [両方が指定された場合] それ以外の場合は、左もしくは Inverse(右)

EdgeAnimBlendOp	操作
EDGE_ANIM_BLENDOP_COMPOSE_SUB_LEFT_FROM_RIGHT	合成： Sub（右、左）[両方が指定された場合] それ以外の場合は、右もしくは Inverse（左）

線形ブレンド

操作：

- EDGE_ANIM_BLENDOP_BLEND_LINEAR

ブレンド係数の計算は、以下のように行われます。

if (rightWeight > leftWeight)

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

else

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

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

表 23 は、線形ブレンド用の部分的なアニメーションロジックの動作を示しています。

表 23 EDGE_ANIM_BLENDOP_BLEND_LINEAR のための部分的なアニメーションロジック

左	右	出力	出力重み
未定義 (重み= 0)	未定義 (重み= 0)	未定義	0.0 (0x00)
定義済 (重み>0)	未定義 (重み= 0)	左	左重み
未定義 (重み= 0)	定義済 (重み>0)	右	右重み
定義済 (重み>0)	定義済 (重み>0)	Blend (左、右、ブレンド係数)	Blend (左重み、左重み+ 右重み、ブレンド係数)

加算デルタブレンド

操作：

- EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_RIGHT
- EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_LEFT

これらのブレンド操作は、操作名自体で明らかなように、ベースチャンネル、およびこのベースチャンネル±デルタチャンネルの間のブレンド操作を可能にします。

ブレンド係数の計算は、以下のように行われます。

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

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

部分的なアニメーションロジックは、ベースチャンネルの定義を必要とします。デルタチャンネルだけが定義されている場合の出力は未定義です。

以下の表では、例として ADD_DELTA_RIGHT を示します。

表 24 EDGE_ANIM_BLENDOP_BLEND_ADD_DELTA_RIGHT のための部分的なアニメーションロジック

左	右	出力	出力重み
未定義 (重み= 0)	未定義 (重み= 0)	未定義	0.0 (0x00)
定義済 (重み>0)	未定義 (重み= 0)	左	左重み
未定義 (重み= 0)	定義済 (重み>0)	未定義	0.0 (0x00)
定義済 (重み>0)	定義済 (重み>0)	Blend (左、add (左、右)、 ブレンド係数)	Blend (左重み、左重み+ 右重み、ブレンド係数)

合成

操作：

- EDGE_ANIM_BLENDOP_COMPOSE_ADD
- EDGE_ANIM_BLENDOP_COMPOSE_SUB_LEFT_FROM_RIGHT
- EDGE_ANIM_BLENDOP_COMPOSE_SUB_RIGHT_FROM_LEFT

「合成」操作ではブレンドが行われない (α は無視される) ので、デルタブレンドとは異なります。また、どちらのチャンネルも同等に扱われるので、ベースやデルタという概念也没有せん。ブレンド係数の計算は、以下のように行われます。

$$\begin{aligned}\text{blendFactor} &= 1.0 \text{ (ignored)} \\ \text{outputWeight} &= \text{leftWeight} + \text{rightWeight}\end{aligned}$$

表 25、表 26 に示されているように、部分的なアニメーションロジックは、加算モードと減算モードでは異なります。

表 25 EDGE_ANIM_BLENDOP_COMPOSE_ADD の部分的なアニメーションロジック

左	右	出力	出力重み
未定義 (重み= 0)	未定義 (重み= 0)	未定義	0.0 (0x00)
定義 (重み>0)	未定義 (重み= 0)	左	左重み
未定義 (重み= 0)	定義 (重み>0)	右	右重み
定義 (重み>0)	定義 (重み>0)	Sub (左、右)	左重み+右重み
未定義 (重み= 0)	未定義 (重み= 0)	未定義	0.0 (0x00)
定義 (重み>0)	未定義 (重み= 0)	未定義	0.0 (0x00)
未定義 (重み= 0)	定義 (重み>0)	未定義	0.0 (0x00)
定義 (重み>0)	定義 (重み>0)	Sub (左、右)	左重み+右重み

表 26 EDGE_ANIM_BLENDOP_COMPOSE_SUB_RIGHT_FROM_LEFT のための部分的なアニメーションロジック

左	右	出力	出力重み
未定義 (重み= 0)	未定義 (重み= 0)	未定義	0.0 (0x00)
定義済 (重み>0)	未定義 (重み= 0)	未定義	0.0 (0x00)
未定義 (重み= 0)	定義済 (重み>0)	未定義	0.0 (0x00)
定義済 (重み>0)	定義済 (重み>0)	Sub (左、右)	左重み+右重み

例：簡単なブレンドツリー

図 2 簡単なブレンドツリーの例

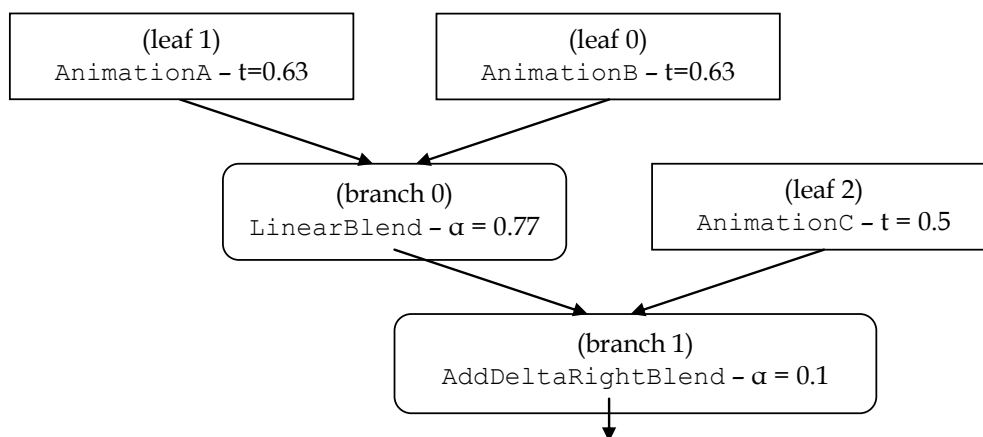


表 27 簡単なブレンドツリーの例：リーフ配列

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

表 28 簡単なブレンドツリーの例：ブランチ配列

コマンド	左	右	アルファ
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 は、パラメータとして、ルートノード（最終的な出力）のインデックスをとります。この例なら、ルートノードは「ブランチ 1」なので、そのインデックスは「1 | EDGE_ANIM_BLEND_TREE_INDEX_BRANCH」になります。

このブレンドツリーの例は、以下のようなコマンドリストに変換されます(表 29)。

表 29 簡単なブレンドツリーの例 : コマンドリスト

コマンド	引数
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

コールバック

edgeAnimProcessBlendTree() に渡された、ユーザの提供するコールバックは、パイプラインの各ステージの後で、各コマンドについて呼び出すことができます。

リーフに結び付けられる EDGE_ANIM_CMD_PUSH_AND_EVAL コマンドについては、EdgeAnimLeafCallback を参照してください。

ブランチに結び付けられる EDGE_ANIM_CMD_BLEND_AND_POP コマンドについては、EdgeAnimBranchCallback を参照してください。

ユーザ定義のコールバックでは、pipelineStage に渡された引数 (0, -1, -2) をチェックすることによって、現在のステージを求めることができます。

ミラーリング

リーフまたはブランチの作成時に EDGE_ANIM_FLAG_MIRROR フラグを渡すことで、ブレンドツリーのどの段階においてもミラーリングをオプションで適用することができます。評価された、またはブレンドされたポーズの結果は、あらかじめ定義されたミラーリング指定に基づいてミラーリングされます。

ミラーリング指定とはミラーリング操作とジョイントインデックスの組を定義する

EdgeAnimMirrorPair 構造体の配列で構成されており、この構造体の配列が

edgeAnimProcessBlendTree() に渡されます。この 2 つのジョイントには指定された操作が適用され、その結果評価されたポーズに含まれるエントリが交換されます。インデックスが同一であってもよく、この場合には単一のジョイントに対してのみミラーリングが適用されます。ミラーリング指定で定義されていないジョイントは、評価後の状態から変更されないままとなります。

EdgeAnimMirrorPair 構造体を作成する際には、3 つのケースを考慮する必要があります。

- (1) ジョイントは対ではない (たとえば背骨)
- (2) ジョイントは対であり、親は対ではない (たとえば left_shoulder と right_shoulder)
- (3) ジョイントは対であり、親は対である (たとえば left_arm と right_arm)

Edge Animation では、x=0 のプレーンに対してミラーリングを行う以下のマクロが提供されています。それぞれ 3 つのケースに対応しています。

- EDGE_ANIM_MIRROR_SPEC_NON_PAIRED
- EDGE_ANIM_MIRROR_SPEC_LINK
- EDGE_ANIM_MIRROR_SPEC_PAIRED

スケルトンではジョイントの向きが適切に定義されていることが重要です。特に、これらのマクロは、ボーン方向に沿って各ジョイントの y 軸が伸びていることを前提としています。これとは異なるジョイントの向きやミラーリングプレーンが必要な場合には、これらのマクロは使用せず、ユーザが自分でミラーリング処理を記述しなければなりません。ジョイントミラーリングの指定のフォーマットを次の表に示しま

す。これらを用いて、回転および平行移動コンポーネントを任意に反転および符号変換することができます。

表 30 ジョイントミラーリング指定

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

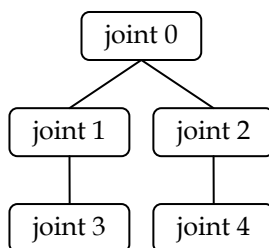
各 4 ビットブロックの内容は以下のとおりです。

- ビット 0 (MSB) はネグートビットです。このビットが設定されている場合、最終結果がネグートされます。
- ビット 1 は未使用です。
- ビット 2 およびビット 3 は、ライブラリが値を取得する成分のインデックスです。

たとえば、EDGE_ANIM_MIRROR_SPEC_LINK の値が 0xB8219203 であり、回転および平行移動コンポーネントに対して以下に示すマッピングが行われます。

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

また、シンプルなスケルトン構造と、関連付けられたミラー指定の例を以下に示します。



```

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)
  
```

この指定では、x=0 のプレーンに対してアニメーションのミラーリングを行い、ジョイント 1 および 3 がそれぞれジョイント 2 および 4 と交換されます。バイナリ操作が必要な指定のは 1 度だけである点に注意してください (ジョイントの対の順序は重要ではありません)。

ミラーリングのサンプルについては、「samples/edge/mirror-sample」を参照してください。

計算済みのポーズ

ブレンドツリーの中では、計算済みのポーズを指定することができます。たとえば、以前のブレンドツリーの処理の際に計算したポーズを再利用したいことがあります。そうするには、ポーズのアドレスを葉のコンストラクタに渡して、(ポーズがメインメモリとローカルストアのどちらにあるかに応じて) 以下のフラグのいずれかを指定します:

- EDGE_ANIM_FLAG_POSE_FROM_MAIN
- EDGE_ANIM_FLAG_POSE_FROM_LOCAL

ポーズを保存して後で再利用するには、ポーズをスタックから取得して、jointArray から書き込みます。たとえば、以下のコードは、スタックの一番上からポーズを取得して、それをメインメモリ中の poseEa に書き込みます。


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

アニメーションのエンコードおよび圧縮

定数チャンネル

1つのアニメーションに含まれた多くのチャンネルは実際、定数で構成されております。複雑なブレンドで使われような小さいアニメーションでは、定数が全データの大部分を占めます。Edge アニメーションでは、定数に必要なサイズを最小化しようとしています。

ベースポーズと同じ定数データは無視され、アニメーションには何の情報も格納されません。一般的なキャラクタでは、この定数データが、スケールやルート以外の平行移動のデータの大部分と、回転チャンネル（指）のかなりの部分を占めます。

ベースポーズとは異なるが、指定されたアニメーション中では定数であるチャンネルも、サイズへの影響を減らすため、特別に処理されます。

均でないアニメーションデータ

アニメーションは、まず一定周波数でサンプリングされます。その後、最適化パスを実行し、補間により指定された許容範囲内で近似できるサンプルをすべて取り除きます。散在する残りのキーフレームのデータは、定数と定数でないデータに分けられます。ベースポーズと異なる定数データは別に格納され、定数でないデータは別々のフレームセットに分割されます。

フレームセットは、クリップ内の特定の時間範囲の定数でないチャンネルすべてのキーフレームを含む、評価のために最終的に SPU にアップロードされるデータの単位です。フレームセットのメモリサイズは一定なので、特定のフレームセットの時間範囲は、その時のサンプルデータのまばらさによって異なります。また、フレームセットには、実際のキーフレームデータの他に、どのキーフレームが各フレームや各チャンネル用に存在するかを定義するビットストリームが含まれています。

フレームセットは、以下のような構造をしています。

- 初期ジョイント回転
- 初期ジョイント平行移動
- 初期ジョイントスケール
- 初期ユーザチャンネル
- フレーム内回転ビット
- フレーム内平行移動ビット
- フレーム内スケールビット
- フレーム内ユーザビット
- フレーム内ジョイント回転
- フレーム内ジョイント平行移動
- フレーム内ジョイントスケール
- フレーム内ユーザチャンネル

初期データには、フレームセットの最初のフレームの、定数以外のチャンネルすべてのデータが含まれています。フレーム内データには、残りのフレームの散在するキーフレームデータが含まれています。フレ

ーム内ビットストリームには、1 サンプル当たりのビット数が含まれています。これが 1 の場合、そのサンプルにキーフレームがあることを示しています。

また、エバリュエータには全チャンネルの最終データが必要で、このデータは次のフレームセットの初期データから取得されます。

表 31 は、回転チャンネルが 4 つ、平行移動チャンネルが 1 つ、フレーム内が 4 つのデータ構造の例です。

表 31 データ構造の例

R0 フレーム 0	R1 フレーム 0	R2 フレーム 0	R3 フレーム 0
T0 フレーム 0	イントラ R ビット	イントラ T ビット	R0 フレーム 1
R0 フレーム 2	R0 フレーム 4	R1 フレーム 1	R2 フレーム 1
R2 フレーム 2	R2 フレーム 3	R3 フレーム 2	R3 フレーム 4
T0 フレーム 1	T0 フレーム 2		

- イントラ R ビット： 1101 1000 1110 0101
- イントラ T ビット： 1100

クォータニオン圧縮

回転データは、アニメーションデータの大部分を占めています。単位クォータニオンを圧縮するために使われるデフォルトの圧縮方式は、表の最小 3 成分圧縮です。この方式では、単位クォータニオンの小さい

方の 3 成分が、1 成分当たり 15 ビットで表され、それぞれ $-\frac{\sqrt{2}}{2} \sim \frac{\sqrt{2}}{2}$ の範囲を表します。そして、最大

の成分は、最大成分 $= \sqrt{1 - \text{smallestA}^2 - \text{smallestB}^2 - \text{smallestC}^2}$ によって復元されます。最後に、3 つの成分のうちどれが最小かを表すために、さらに 2 ビットが使われます。

このデータのフォーマットは、以下の通りです。

表 32 圧縮クォータニオンのデータ形式

0	14	15	29	30	44	45	46	47
最小成分 A		最小成分 B		最小成分 C		SHUF		S

- 最小成分 A は、元の単位クォータニオンの成分の中で最も小さい値です。
- 最小成分 B は、元の単位クォータニオンの成分の中で 2 番目に小さい値です。
- 最小成分 C は、元の単位クォータニオンの成分の中で 3 番目に小さい値です。
- SHUF は、最大値を指定するシャッフルマスクテーブルへのインデックスです。
- S は、復元された最大成分の符号が負の場合にセットされます*1。

*1 現在、このツールでは、必要に応じてクォータニオンの符号を反転させることにより、最大成分が常に正であることを保証しています。

ビットパック

ビットパックは、オプションで任意のチャンネルに適用できます。このモードでは、チャンネルは、コンポーネント当たりの任意のビット長を指定します。この値は、ユーザ指定のエラー許容範囲に依存します。チャンネルごとのパック指定は、以下の通りにエンコードされます。

表 33 ビットパック指定のフォーマット

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 は、1 番目の成分をエンコードするために使われる符号、指数、仮数ビットの数です。
- sgnB、expB、mantB は、2 番目の成分をエンコードするために使われる符号、指数、仮数ビットの数です。
- sgnC、expC、mantC は、3 番目の成分をエンコードするために使われる符号、指数、仮数ビットの数です。
- index は、復元された 4 番目の成分のインデックスです（クォータニオンのみ）。

どの成分でも指数ビットの数がゼロでない場合には、パックされた値は float としてアンパックされます。それ以外の場合には、正規化された固定小数点値としてアンパックされます。

ユーザチャンネルの場合、成分は 1 つだけなので、その指定は sgnA、expA、mantA の中にエンコードされます。

注意: ビットパックは評価するのにコストがかかり、SPU でしか実装されていません。PPU 用の実装は提供されていません。

評価

以下は、_edgeAnimEvaluate() 内でのクリップの評価の際に起こる処理の詳細フローです。

- スケルトンベースポーズを出力ポーズにコピーします。
- 定数チャンネルテーブルのマッピングを使って、定数チャンネルを出力ポーズに解凍します。
- frameFraction を評価フレームの小数值とします。
- tableR を非定数回転チャンネルの ID テーブルとします。
- 非定数回転チャンネルのそれぞれについて、以下を実行します。
 - keyRData を、このチャンネルのフレーム内圧縮キーフレームの開始アドレスとします。
 - totalBits を、このチャンネルのフレームデータのビットリスト中の有効ビットの数とします。
 - フレームデータのビットリスト中の、このフレームのビットより前にある有効ビットの数を prevBits とします。
 - prevBits = 0 の場合、keyA を解凍する最初のキーフレームとします。
 - それ以外の場合、keyA を $\text{keyRData} + \text{sizeof(R)} * \text{prevBits}$ で解凍するキーフレームとします。
 - prevBits = totalBits の場合、keyB を解凍する最後のキーフレームとします。
 - それ以外の場合、

keyB を $\text{keyRData} + \text{sizeof}(\text{R}) * (\text{prevBits} + 1)$ で解凍するキーフレームとします。

- bBits を、このチャンネルのビット以前（このチャンネルのビットも含む）の連続する無効ビットの数とします。
- aBits をこのチャンネルのビットの後にある連続する無効ビットの数とします。
- alpha を $(\text{bBits} + \text{frameFraction}) / (\text{aBits} + \text{bBits} + 1)$ とします
- interp を keyA と keyB の間の alpha による球面線形補間とします。
- tableR のマッピングを使って、interp を出力ポーズに書き込みます。
- 以上を、非定数の平行移動、スケール、ユーザチャンネルに対して繰り返します。

9 Edgeアニメーションのツール

はじめに

COLLADA™フォーマットのファイルに格納されたスケルトンやアニメーションを、Edge アニメーションにより SPU 上で処理できるランタイムフォーマットに変換するために利用できる「edgeanimcompiler」サンプルツールが提供されています。

また、Edge アニメーション固有の処理を集中化する libedgeanimtool と呼ばれるライブラリも提供されています。このライブラリを使うと、COLLADA™フレームワークの使用を必要としないツールを生成することができます。edgeanimcompiler ツールは、このライブラリと一緒にビルドされます。

使用法

edgeanimcompiler は、以下のように、入力として COLLADA™フォーマットのファイルをとる簡単なコマンドラインツールであり、指定された引数に応じて、スケルトンやアニメーションや付加的なアニメーションのランタイムファイルを生成することができます。

```
edgeanimcompiler -skel <入力 COLLADA> <出力スケルトン> [-userchannels <入力テキスト>]
edgeanimcompiler -anim <入力 COLLADA> <入力スケルトン> <出力アニメーション>
edgeanimcompiler -additiveanim <入力ベース COLLADA> <入力 COLLADA> <入力スケルトン> <出力付加的アニメーション>
edgeanimcompiler -selfadditiveanim <入力 COLLADA> <入力スケルトン> <出力付加的アニメーション>
```

表 34 は、アニメーションコンパイラの各引数を説明したものです。

表 34 アニメーションコンパイラの引数

引数	解説
<入力 COLLADA>	スケルトンやアニメーションデータを含む COLLADA™フォーマットのファイルを指定します。
<入力ベース COLLADA>	付加的なアニメーションを生成するためのベースアニメーションを含む COLLADA™フォーマットファイルを指定します。
<入力テキスト>	ユーザチャンネル名（大文字と小文字の区別あり）を各行に含むテキストファイルを指定します。
<入力／出力スケルトン>	ツールで生成されるバイナリスケルトンファイルを指定します。
<出力アニメーション>	ツールで生成されるバイナリアニメーションファイルを指定します。
<出力付加的アニメーション>	ツールで生成されるバイナリ付加的アニメーションファイルを指定します。

処理概要

使用法のセクションで示したように、このツールから出力できるファイルは 3 種類あります。各種類の出力には、libedgeanimtool ライブラリに集約される固有の処理が必要ですが、以下の共通の処理のパターンは 3 種類すべてに適用されます。

- コマンドライン引数を解析して出力ファイルの種類を求めます。
- 入力ファイルから必要なデータを抽出します。
- libedgeanimtool によって公開されている構造体を書き込みます

- libedgeanimtool を呼び出してデータを処理し、出力ランタイムファイルをエクスポートします。

スケルトン処理

スケルトン処理の入力は、エクスポートされたジョイントを含む COLLADA™フォーマットのファイル、および（オブションの）各行にユーザチャンネル名（大文字と小文字の区別あり）の一覧を含むテキストファイルです。コンパイラは、COLLADA™ファイルを読み込み、シーンの中にあるジョイントノードのすべてを収集して、リストを生成します。このリストは、以後のデータ抽出フェーズで使われます。

コンパイラは、このジョイントのリスト（ルートジョイントは1つだけであるはず）を検証してから、時刻0におけるジョイントアニメーションのすべてを評価することにより、バインドポーズを抽出します。また、この手順の際に、ジョイント名のハッシュも計算され、格納されます。このハッシュは、ジョイントを素早く検索するために、SPU ランタイムライブラリで使われます。

ユーザがコマンドラインでユーザチャンネルのリストを指定した場合には、チャンネルの数が決定され、後で libedgeanimtool に渡される名前ハッシュのリストが生成されます。

抽出の最後の手順は、先に収集したジョイントのそれぞれについて、親ジョイントインデックスのリストをビルドすることです。これで、コンパイラには libedgeanimtool がスケルトンバイナリを処理し、エクスポートするために必要な、あらゆるデータが揃います。このデータはスケルトン構造体書き込まれ、libedgeanimtool によって公開されている ExportSkeleton() 関数に渡されます。

ExportSkeleton() 関数は、抽出されたデータを取得して、そのデータをランタイム SPU ライブラリが扱うフォーマットに合わせるために処理を実行します。この処理には、SPU アーキテクチャの SIMD の特性を処理に利用できるようにするために、ジョイントのリストを4つの集合に分割することが含まれます。この追加処理の後、Edge アニメーションが必要とするフォーマットのファイルが、ディスクに書き込まれます。

ユーザチャンネル

Edge アニメーションは、任意のユーザスカラーチャンネルのアニメーションをサポートします。今のところ、edgeanimcompiler がサポートするユーザチャンネルの種類は、ブレンド形状重みだけです。ユーザチャンネルは、スケルトンエクスポート時に、-userchannels 引数によって edgeanimcompiler にテキストファイルを渡すことにより、指定できます。このテキストファイルには、以下の構文を使って、（1行に1つずつ）重みを記載することができます。

```
controller.shaper
```

ただし、「controller」はコントローラノードの名前、「shape」はターゲット形状の名前です。これらの名前をハッシュ化した値は、スケルトンといっしょに格納され、その中に記載されていないユーザチャンネルはすべて、アニメーションファイルをエクスポートする際に無視されます。

ブレンド形状のアニメーションを含むメッシュをエクスポートしたりレンダリングしたりする例としては、第23章「[サンプルプログラム](#)」の「[Edgeジオメトリおよびアニメーション](#)」セクションの「samples/edge/blendshape-anim-sample」を参照してください。

アニメーション処理

アニメーションバイナリを生成するために必要な入力は、アニメーションキーフレームを含む COLLADA™ファイル、および先程ツールでエクスポートしたスケルトンバイナリファイルです。

この入力スケルトンファイルには、ゲーム中でアニメーションやスキニングのために使われる「バインドされた」スケルトンが含まれています。このスケルトンは、ツール処理の際に、アニメーション用のバインドや部分的なアニメーション情報を計算するために使われます。ツール処理の第一段階は、両方の入力ファイルから、先にスケルトン処理についてのセクションで説明したのと同じデータを抽出することです。次の段階では、COLLADA™シーンからアニメーションデータを抽出する必要があります。これは2つのパスで実現されます。最初のパスでは、入力アニメーション中のジョイントに影響する、あらゆるアニメーション曲線中のであらゆるキーフレームを調べて、キー時間の順序リストを作成します。このリストから、サンプリングレート、開始時間、および終了時間が求められます。

次のパスでは、各ジョイントの各キー時間におけるアニメーションを評価して、回転、平行移動、およびスケール曲線をサンプリングします。このサンプリングされたキーフレームは、ジョイントごとに格納され、処理やエクスポートのために edgeAnimLib に渡せるように、libedgeanimtool の公開している構造体に収集されます。コマンドラインで指定されたスケルトンがユーザチャンネルを持っている場合には、そのユーザチャンネルも同じようにサンプリングされます。

この段階で、アニメーションデータは、libedgeanimtool に渡して実行時フォーマットに書き込める状態になっています。ここで、libedgeanimtool によって、さらにいくつかの処理が行われます。まず、バインドスケルトンのベースポーズと変わらない定数アニメーションチャンネルは、すべて削除されます。それから、オプションでアニメーションの最適化が行われます。最適化では、補間によって近似できる冗長なキーフレームを取り除くことによって、最終的な出力ファイルのサイズを減らします。最後に、アニメーションはフレームセットに分割されます（第8章「[Edgeアニメーションのランタイムの詳細](#)」の「[均一でないアニメーションデータ](#)」というセクションを参照）。

フレームセットへの分割が済むと後に残るのは、ディスクに書き込まれるバイナリファイルと、アニメーションデータに関するさまざまな統計を含むプレーンテキストファイルだけです。

付加的なアニメーション処理

付加的なアニメーションの生成に利用できる方法には、入力アニメーションの最初のフレームを残りのフレームから差し引く方法と、別個の独立したアニメーションファイルを指定してファイルから差し引く方法の2つがあります。これらのモードは、それぞれ、コマンドラインから引数 `-selfadditiveanim`、そして `-additiveanim` によって指定することができます。出力されるのは、実行時にベースアニメーションや特定のポーズに加算的にブレンドすることができる差分アニメーションを含む、バイナリアニメーションファイルです。

ツールで実行される処理は、libedgeanimtool で提供される `GenerateAdditiveAnimation()` メソッドの中で、入力アニメーションから別のアニメーションやポーズが抽出され、元のアニメーションから差し引かれることを除けば、単一のアニメーションと同じです。

このメソッドでは、各チャンネル（回転、平行移動、スケーリング、その他）に対して該当する減算メソッドを使用する前に、差し引かれるアニメーションの中で指定されたキー時間において、ジョイントやユーザチャンネルのキーフレームがどちらのアニメーションにも存在するかどうかを判定します。アニメーションの減算が済むと、その結果に対し、通常のアニメーション処理で使われるのと同じ最適化処理が適用されます。

最後に、先のセクション（Edge 固有のフレームセットパーティショニングを含む）と同じように、アニメーションがバイナリファイルにエクスポートされ、さらに、アニメーションデータ統計を含むプレーンテキストファイルが書き出されます。

圧縮

edgeanimcompiler には、デフォルトモードとビットパックモードの 2 種類の圧縮モードがあります。

デフォルトモードでは、回転に対して最小 3 成分圧縮方式を使用し、それ以外のチャンネルタイプはすべて圧縮しない float のままにします。

ビットパックモードでは、指定された特定の許容範囲値を使って、全チャンネルにビットパックを適用します。このモードでは、特に回転以外のチャンネルがたくさんある場合に、より効果的な圧縮が行われます。そのためにかかるコストは、SPU 処理時間の増加です。

ビットパックを指定するには、edgeanimcompiler を呼び出す際に、-bitpacked 引数を使います。許容範囲は、-tolerance 引数で指定できます。次に例を示します。

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

libedgeanimtool では、現在 edgeanimcompiler でサポートしているものよりも細かい圧縮モードや許容範囲の制御を提供していることに注意してください。詳しくは、「PlayStation®Edge オフラインツール用アニメーションライブラリ リファレンス」を参照してください。

10 EdgeアニメーションとEdgeジオメトリの併用

スキニング

スキニングされたキャラクタをレンダリングするために、コードには逆バインド行列も必要です。この行列にワールド空間ジョイント行列をかけると、最終的に Edge ジオメトリが使用するスキニング行列が提供されます。

edgegeomcompiler から逆バインド行列をエクスポートするには、出力ファイルを `-inv-bind-mats-out` 引数に指定します。エクスポートされるバイナリは、3x4 の転置行列のフラット化された配列によって構成されています。行列の数は、スケルトン中のジョイントの数に等しくなります（スキニングに関係ないジョイントには、単位行列が割り当てられます）。

実行時に、以下のような SPU 関数を順次呼び出すと、ローカルジョイント変換の配列のからスキニング行列を生成することができます。

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

スキニングされたキャラクタに対し、エクスポート、アニメーション、レンダリングを実行する例については、第 23 章「[サンプルプログラム](#)」の「[Edgeジオメトリおよびアニメーション](#)」セクションの「[samples/edge/character-sample](#)」を参照してください。

注意：edgegeomcompiler は、逆バインド行列を $B \cdot M$ という式によって計算します。ただし、 M は対象メッシュの変換、 B はワールド空間バインドポーズジョイント変換の逆です。各ジョイントに対して出力される逆バインドマトリックスは 1 つだけなので、同じジョイントによって異なる変換をもつメッシュが影響を受けている場合には、衝突が起こる可能性があります（この場合、edgegeomcompiler は警告を出力します）。この事態を避けるため、エクスポートの前に、スキニングされたメッシュの変換を凍結しておく必要があります。

11 Edge Zlib、LZMA、およびLZOの概要

アルゴリズム

Edge は、SPU に対して最適化された 3 種類の可逆伸長アルゴリズムの実装、および 2 種類の可逆実行時圧縮の SPU 実装を提供します。

Edge の伸長アルゴリズムには、zlib ベースのもの、LZMA ベースのもの、LZO ベースのものがあり、それぞれには以下のようなトレードオフがあります。

- **Edge LZMA** は、圧縮率をもっとも優れていますが（つまりファイルがより小さくなる）、伸長にはもっとも長い時間がかかります。
- **Edge LZO** は、圧縮率を犠牲にして、最も高速な伸長を提供します。
- **Edge Zlib** は、上記 2 つをうまく折衷した方法を提供します。

Edge Zlib と Edge LZO は、可逆実行時圧縮の SPU 実装を提供します。

表 35 と表 36 は、特定のテストデータのセットから得られた、伸長/圧縮値の例を示しています。圧縮率や速度は、入力データによって大きく異なる可能性があります。したがって、ユーザのデータが示す値は、これとは大きく異なる可能性があります。

表 35 Edge LZO、Zlib、および LZMA の伸長:テスト結果値の例

	Edge LZO レベル 1 (06)	Edge LZO レベル 1 (14)	Edge Zlib レベル 1	Edge Zlib レベル 9	Edge LZMA レベル 1	Edge LZMA レベル 9
伸長速度 (出力データの MiB/s)	451	397	107	111	17	19
圧縮率 (マスタに対する%)	71	61	54	52	53	44

表 36 Edge LZO と Zlib 圧縮:テスト結果値の例

	Edge LZO	Edge Zlib レベル 1	Edge Zlib レベル 9	Edge LZMA
圧縮速度 (入力データの MiB/s)	76	6	3	該当なし
圧縮率 (マスタに対する%)	61	58	57	該当なし

Edge Zlib では、ローカルストアの制限により、一部の内部バッファのサイズが削減されています。たとえば、オフラインのコンプレッサで取得される場合ほどは圧縮率がよくないことがあります。

推奨ユースケース

Edge Zlib は、Blu-ray Disc や HDD からデータをロードした場合、ドライブからデータを受け取るスピードよりも高速にデータを伸張することができます。したがって、このような場合には、速度と圧縮率の両面で Edge Zlib を使うのが良い選択です。

ダウンロードしたゲームの場合、ファイルサイズの方が重要である可能性があるので、伸長速度の遅さを許容できるのであれば、Edge LZMA が良い選択であることもあります。

Edge LZO は、実行時の圧縮が必要な場合、あるいは、極めて高速な伸長が必要な場合に、最適である可能性があります。

12 Edge Zlib、LZMA、およびLZOの使用

Edge Zlib、Edge LZMA、および Edge LZ0 のインタフェースは非常によく似ています。この章では、Edge Zlib を例に使うて伸長を実行するための手順を説明します。その後の章では、Edge LZMA または Edge LZ0 での伸長や Edge Zlib または Edge LZ0 での圧縮に必要とされるものが、この章で説明した手順と著しく異なる場合について指摘します。

実装の詳細

Edge Zlib、Edge LZMA、および Edge LZ0 は、階層的に実装されています（SPU ライブラリは SPURS タスクに利用され、PPU ライブラリに組み込まれている）。このような実装により、ユーザは、ライブラリに対するインタフェースのレベルを選択することができます。

SPU ライブラリ

SPU には、関連する圧縮/伸長コードのすべてを含むライブラリがあります。このライブラリ（libedgezlib.a）は、Edge Zlib の例の中では、以下のような関数を提供しています。

- edgeZlibInflateRawData() - ローカルストア内で伸長を実行します。この関数は、ローカルストア内の 2 つのバッファのポインタを受け取ります。1 つは、伸張すべき入力圧縮データを含むバッファです。もう 1 つは、出力伸張データを書き込むバッファです。
- edgeZlibDeflateRawData() - ローカルストア内で圧縮を実行します。この関数は、ローカルストア内の 2 つのバッファのポインタを受け取ります。1 つは、マスタ入力データを含むバッファです。もう 1 つは、圧縮出力データを書き込むバッファです。

SPURS タスク

SPURS を使用する場合、Edge は、圧縮用と伸長用の 2 つの SPURS タスクを提供します。これらのタスクは、上で言及した SPU ライブラリの中の処理関数を呼び出します。

- 伸長タスクは、キュー（「伸長キュー」）から伸長作業を取り出します。
- 圧縮タスクは、キュー（「圧縮キュー」）から圧縮作業を取り出します。

これらのキューの各要素は、圧縮・伸長元および圧縮・伸長先のバッファの詳細を提供します。このタスクは、入力データを DMA 転送し、圧縮・伸長を実行してから、出力データを必要に応じて DMA 転送します。バッファの圧縮・伸長が完全に完了して書き出されると、そのタスクは、メインメモリ上のカウンタをアトミックに 1 デクリメントすることができます。このカウンタは、関連する要素の集合が完全に伸張されるのはいつかを検出する方法を提供します（numItems に初期化されたカウンタがゼロになるので）。カウンタがゼロになった時点で、タスクはイベントフラグを通知することができます。これは、特定の作業セットが完了するのを待ちながら、PPU スレッドをスリープさせる方法を提供します。

伸長の際にエラーが発生した場合には、カウンタの最上位ビットが 1 に設定されます。

タスクは、キュー要素の圧縮・伸長が完全に完了すると、ループの先頭に戻って次の作業を行う前に、SPU のリソースを明け渡す必要のある、自分より優先順位の高いワークロードがあるかどうかを判断するためのポーリングを行います。ループ中のこの点でのリソース解放は、1 セットのデータが送り出された後、しかし次のデータが読み込まれる前までの間に発生します。これにより、SPURS タスクのコンテキスト保存サイズを最小（2KB のスタック領域）に維持することができます。コードに変更を行った場合には、これより大きなコンテキスト領域を保存する必要がある可能性があります。

PPU ライブラリ

PPU 上には、上述の伸長/圧縮タスクにインタフェースを提供するライブラリが存在します。Edge Zlib の例では、このライブラリ (libedgezlib.a) が、タスクを生成してユーザ指定の SPURS タスクセットにアタッチするために使われます。複数のタスク (SPU ごとに) を生成することにより、複数の SPU が圧縮・伸長を並列処理することができます。

このライブラリも、タスクが取り出される圧縮・伸長キューの生成や使用のための関数を用意しています。

Edge ZlibをPPUから利用する手順

Edge Zlib/Edge LZMA/Edge LZ0 を使う最も簡単な方法は、上記の PPU ライブラリを使うことです。このライブラリは、組み込まれた伸長/圧縮タスクを通じて、あらゆる SPU コミュニケーションを内部的に処理します。

(1) 初期化

edgeZlibCreateInflateQueue() を呼び出して伸長キューを生成します。この関数は、EdgeZlibInflateQHandle を返します。この伸長キューは、作業用にメインメモリのバッファを必要とします。このバッファに必要なサイズは、edgeZlibGetInflateQueueSize() を呼び出すことにより求めることができます。このバッファは、128 バイトにアラインメントされている必要があります。SPURS インスタンスおよびタスクセットの初期化が済んだら、edgeZlibCreateInflateTask() を呼び出します。これにより、伸長用の伸長タスクが 1 つ生成されます。複数の SPU で伸長の並列処理を行いたい場合には、1 つの SPU について一度ずつ、複数回呼び出します。この関数は、タスクのコンテキストをページアウトする可能性があるため、メインメモリ上に 128 バイトにアラインメントされたバッファを必要とします。このバッファに必要なサイズは、edgeZlibGetInflateTaskContextSaveSize() を呼び出すことにより求めることができます。

(2) 作業の追加

edgeZlibAddInflateQueueElement() を呼び出すことにより、伸長キューに作業を追加することができます。この関数は、引数として、メインメモリ中の入力バッファおよび出力バッファへのポインタ、入力データのサイズと出力データの予想サイズを受け取ります。伸張後の出力データが予想サイズと異なる場合には、SPU はアサーションが発生します。

複数の作業アイテムを追加したい場合には、メインメモリ中の uint32_t 型のカウンタを、追加したいセグメントの数に初期化します。そして、このカウンタのアドレスを edgeZlibAddInflateQueueElement() に渡します。このカウンタは、各セグメントの伸長が済むたびに、1 デクリメントされます。また、この関数にはイベントフラグへのポインタを渡すこともできます。このイベントフラグは、カウンタがゼロになった後、設定されます。

(3) 作業の SPU 処理

キューに伸張すべき作業があると、自動的に伸長タスクが実行されます。ただし、作業がない場合や、より優先順位の高いワークロードがある場合には、タスクは SPU のリソースを解放します。リソースが解放されるのは、特定の要素の伸長が終わった後、次の伸長が始まる前なので、保存されるコンテキストは最小限で済みます。

(4) SPU 伸長作業が完了したとき

プログラムは、カウンタの値をクエリーすることによって、作業の完了をポーリングすることができます。PPU スレッドは、イベントフラグ (cellSpursEventFlagWait) がセットされるのを待つことにより、特定の作業の集まりに対する伸長が済むのを待っている間スリープすることができます。

コードは、カウンタがゼロであることを確認する必要があります。最上位ビットが設定されている場合、伸長の際に SPU 上でエラーが発生しています。

(5) 終了処理

伸長キューを終了するには、edgeZlibShutdownInflateQueue() を呼び出します。タスクセットを終了したら、伸長キューやコンテキスト退避領域として割り当てたメモリをかならず解放してください。

Edge ZlibをSPUから利用する手順

Edge Zlib、Edge LZMA、および Edge LZ0 は、SPU ライブラリとして提供されるので、ユーザ独自の SPU 管理ソリューションがある場合には、このレベルのインタフェースを利用することもできます。

注意: SPU ライブラリは PIC や割り込みに対して安全です。

共通の詳細

このセクションでは、Edge Zlib と Edge LZMA の両方に当てはまる、共通な詳細やテクニックについて説明します。

SPURS タスク

edgeZlibCreateInflateTask() は、呼び出されるたびに SPURS タスクを生成します。SPURS タスクは、ループの中で共有された伸長キューから作業をとりだして処理するループを開始します。生成するタスクが 1 つだけだと、伸長作業は 1 つの SPU 上で実行されるだけです。が、複数のタスクを生成すると、そのタスクを異なる SPU 上で同時に起動し、並列に実行することができます。各 SPU は、それぞれキューから新しい要素を取り出し、異なるデータセグメントを処理します。

伸長タスクは、行うべき作業があつて待っていると、自動的に実行されます。作業がなくなると、タスクは自動的にスリープ状態に移行します。タスクは、伸長キューの要素を取り出して伸張した後、SPU リソースを渡さなくてはならないより優先順位の高いワークロードがあるかどうかを調べるためにポーリングを行います。もし、そのようなワークロードがある場合には、タスクは最小限のコンテキストを保存して退避し、SPU リソースを渡します。そのような作業が存在しない場合には、タスクはさらに作業を割り当てようとします。割り当てる作業がない場合には、作業が発生するまでスリープ状態に入ります。

注意: SPU コードを変更した場合、その変更によっては、最小限のコンテキストだけを退避したのでは不十分で、より大きなコンテキスト領域を格納しなくてはならない可能性があります。

先の議論は、edgeLzmaCreateInflateTask()、edgeLzolxCreateInflateTask()、edgeZlibCreateDeflateTask() または edgeLzolxCreateDeflateTask() にも同じように当てはまります。

伸長タスク

伸長を行う間、プログラムは伸張後のデータがどのぐらいの大きさになるかは、あらかじめわかっている必要があります。これにより、メインメモリ中の出力バッファの割り当てを事前に行って、SPU はそこに

書き込むだけに行うことができます。SPU 伸長コードには、伸張後の予想サイズが渡され、その予想値が間違っていることがわかった場合には、SPU はアサートを実行します。

データを圧縮すると、元のデータより小さくなるどころか、より大きくなることもあります。このような状況になっても、Edge Zlib、Edge LZMA、および Edge LZ0 は問題なく動作します。もちろん、圧縮を適用することでデータがさらに大きくなってしまえば、そもそも圧縮すること自体が無意味で、圧縮されていないデータをそのまま使った方がよいでしょう。

圧縮タスク

圧縮タスクは、伸長タスクとほとんど同じように働きます。圧縮バージョンのデータがどれくらい小さくなるかを、事前に知ることは明らかに不可能です。その代わり、ユーザのプログラムは出力データを格納するために大きなバッファを渡し、データが書き込まれたら、SPU はユーザの指定したメインメモリ中の `uint32_t` に圧縮出力データのサイズを書き込むことにより、サイズを通知します。

圧縮を適用すると、場合によっては、データが入力データより大きくなることもあります。この場合には、圧縮タスクが圧縮データを格納するか、それとも、より小さい元のマスタデータを格納するかを選ぶことができます。常に圧縮データを格納するか、それとも、小さい方を格納するかの指定は、圧縮キューに追加された作業のオプションによって選択できます。どちらを格納するかという指定は、出力圧縮サイズを含む `uint32_t` の最上位ビットを設定することにより、通知されます。

SPU を他のシステムと共有する

圧縮・伸長に必要なバイト当たりの計算量は莫大です（Edge Zlib が 64KB のセグメントを 1 つ伸張するには、最高で 1ms かかる可能性があります）。この間に、特定の SPU で処理すべきより優先順位の高い作業が生じたとしても、指定されたセグメントの伸長が完全に完了して、SPU が次の結果点に来るまでは、処理されません。このような、より優先順位の高い作業を処理する前の遅延が問題な場合、データを 64KB 未満のサイズにセグメント化して、SPU 結果点の頻度を増やせば、改善を図ることができます。

各セグメントが SPURS ジョブとしてキューに入れられた場合、SPURS ジョブパイプラインの仕組み上、実際のパイプラインの中には、現在処理中のジョブの後に 2 つのジョブがある可能性があり、これらのジョブをすべて処理しないと、より優先順位の高い作業を処理することはできません。つまり、Edge Zlib のセグメント化された伸長では、遅延は最高で 3ms になる可能性があるということです。これが、SPURS ジョブでなく SPURS タスクとして実装する理由の 1 つです。

ゲームチームによっては、ゲームの主な処理が、理論上は常に稼働している SPU 5 つのスレッドグループで実行されるように、SPU 5 つの SPU スレッドグループ 1 つと、SPU 1 つの SPU スレッドグループ 1 つを初期化することもあるようですが、オペレーティングシステムが SPU を使う必要に迫られた場合には、優先順位の低い SPU 1 つのスレッドグループから制御を奪おうとします。単一の SPU はいつ制御を奪われるかわからないので、作業を単一の SPU に渡そうという選択は、慎重に行う必要があります。伸長処理がデータのロードを待つ場合には、スレッドグループの制御が奪われることによる余分な遅れは問題にならない可能性もあり、したがって、この「6 番目の SPU」上で伸長を実行することを選ぶチームもあるかもしれません。

セグメント化伸長

Edge LZMA と Edge LZ0 は、伸長の実行時に、ローカルストアの制限のため、圧縮・伸張後のデータのサイズを制限し、あらゆるデータが LS の中に収まるようにします。PPU インタフェースを使って伸長作業をキューに入れる場合、圧縮前後のデータは 64k 以下である必要があります。

64KB を超えるデータを処理する際には、データを圧縮前にセグメント化することをお勧めします。これは、セグメントの間には強固な障壁があって、圧縮後は、セグメント間で互いにデータを参照できなくなることの意味しています。したがって、各セグメントは完全に独立していて、他のセグメントからのデータを必要とすることなく、単独で伸張することができます。LS に読み込まれたセグメントは、メインメモリから追加のデータを読み込まなくても伸張することができます。

このセグメント間の強固な障壁のため、セグメント化時の圧縮比は、データ全体を大きな 1 つのファイルとして圧縮した場合と比べると、わずかに劣る場合があります。ただし、この差は、各セグメントが 64KB（デフォルト）ならば、それほど大きくなることはありません。

セグメントが独立しているので、各セグメントは複数の SPU 上で同時並列的に伸張できます。したがって、セグメント化された伸長は、セグメント化されていない伸長と違って、並列化可能です。

同じようなコメントは、64KB を超えるデータの Edge LZ0 による圧縮にも当てはまります。

その場での伸長

伸長タスクは単一のセグメントに作用し、圧縮済みデータを DMA 転送で読み込み、ローカルストア内で伸張して、伸張後のデータを DMA 転送して書き込みます。したがって、LS の中にすべて収まる単一のセグメント（言い換えれば、入出力 64KB 未満）上では、転送した出力データを入力データに上書きすることにより、その場での伸長を行うことができます。もちろん、この処理では、入力データ用のバッファが、隣接するデータを破損することなく、出力データを格納することができる（より大きいのが理想）ということを前提としています。

オフラインでセグメント化され圧縮されたより大きなファイルをロードした場合、その場での伸長には多少の努力が必要です。1 つの解決法としては、ロードされた入力圧縮済みデータのセグメントをばらばらにして、伸張後のデータを上書きできるだけの大きさのバッファに各断片を書き込めるように、メモリを「断片化」する方法があります。つまり、断片化の後、入力圧縮データの後ろにパディングが含まれたバッファ（おそらく 64KB）があるということです。このセグメントは、その後、入力圧縮データとパディングを上書きするように、その場で伸張することができます。他のセグメントも同じように、並列処理によってその場で伸張することができます。第 23 章「[サンプルプログラム](#)」の「[Edge Zlib・LZMA・LZ0](#)」セクション中の「`samples/edge/zlib-inflate-inplace-sample`」サンプルプログラムは、この方法がどのように働くかを示しています。ただし、このアイデアは、Edge LZMA と Edge LZ0 にも同じように応用できます。この「断片化」タスクは、メインメモリとのバンド幅によって制約され、実際に行われる処理はごくわずかです。その処理とは、単に伸長タスクからアクセスするデータを用意することだけです。

おそらく、最終的な伸張後のデータでは、全セグメントが連続している必要があるので、入力データが配置されたバッファ（つまり、入力データ+パディング）も、連続している必要があります。したがって、どのセグメントの圧縮データのサイズも伸張後のデータサイズ以下である必要があります。

大きなファイルをその場で伸張するもう 1 つの方法は、伸長の実行を単一の SPU 上だけに限定し、複数セグメントの伸長を適切な順番でスケジューリングすることです。この方法を使う際には、データは出力バッファの最後にロードします。まず、最後のセグメントをバッファの最後に伸張してから、最後から 2 番目のセグメントから逆方向に残りのセグメントが伸張されます。ファイルにヘッダがある

（EdgeSegzipFileHeader など）場合、最初の数セグメントの圧縮データは、後方ではなく前方にコピーする必要があるかもしれません。

13 Edge Zlibの詳細

特徴

Edge Zlib は、可逆圧縮・伸長を可能にする DEFLATE 圧縮アルゴリズムを実装したものです。このライブラリの元の実装は、Jean-loup Gailly および Mark Adler によって書かれたものです。この Edge Zlib のリリースには、PlayStation®3 の SPU 上で実行できるように修正・最適化された zlib 伸長の実装が含まれています。

Edge Zlib は、伸長・圧縮機能双方の SPU 実装を提供します。伸張用のデータは、gzip のようなツール、もしくは、zlib のような圧縮用のライブラリを使って、オフラインで生成することができます。

Edge Zlib は、Edge LZMA や Edge LZ0 とは異なり、ローカルストア中に収まりきらない大きなデータを必要に応じてメインメモリにストリーミング入出力することで、1 回で圧縮・伸張することができます。

Edge Zlib と zlib の違い

Edge Zlib は、zlib にはない以下のような機能を提供します。

- SPU の命令セットやローカルストア向けに最適化されている。
- 圧縮・伸長用のセグメントを PPU からキューに入れることのできる SPURS タスクが用意されている。
- 複数セグメントの圧縮・伸長を、複数の SPU 上で並列に実行できる。
- `malloc()` に対する要求を回避するために、SPU コードの内部バッファの一部が、ヒープから静的変数に移動されている。

ライセンス

Edge Zlib には、オープンソースプロジェクトである zlib 1.2.3 (<http://zlib.net/>) の修正バージョンが含まれています。この修正されたソースコードは、http://zlib.net/zlib_license.html においてオンラインで公開されているファイル `zlib.h` に記載された、zlib ライセンスの条件の下に配布されます。このライセンスは、商用アプリケーションでの利用を認めています。

DEFLATE データのヘッダ

SPU 伸長コード (`edgeZlibInflateRawData`, `edgeZlibFetchAndInflateRawData`) が受け取れるのは、伸張すべき生データへのポインタだけです。ヘッダは前もって解析して取り除く必要があります。たとえば、zlib 圧縮は、予期せず 2 バイトのヘッダと 4 バイトのフッタを追加することがあります。また、たとえば、`.gz` や `.zip` ファイルヘッダが存在することもあります。このようなヘッダは、ポインタを `edgeZlibInflateRawData()` に渡す前に読み飛ばす必要があります。

`edgeZlibAddInflateQueueElement()` に渡されたポインタは、その後伸長タスクに渡されて、`edgeZlibInflateRawData()` もしくは `edgeZlibFetchAndInflateRawData()` によって処理されるので、同じ要件が適用されます。`edgeZlibAddInflateQueueElement()` に渡すのは、生データへのポインタだけにする必要があります。ヘッダの処理は、この関数を呼び出す前に行う必要があります。

セグメント化された伸長とされていない伸長

Edge Zlib において、ローカルストアの制約より大きなデータを伸張する際にとることのできる方法は、大きく分けて 2 つあります。

- データをセグメント化して（前章の「[セグメント化伸長](#)」を参照）、伸長の際に、LSの中に同時に収まるような少量のデータ（入力≤64KB、出力≤64KB）だけをSPUが処理すればすむようにする。
- SPU がメインメモリ中のデータを繰り返し走査し、入力データを LS にフェッチして、必要に応じて出力を送信する。

64KB のセグメントの場合、単一の SPU 上でのセグメント化された伸長の速度は、セグメント化されていない伸長の速度に匹敵します（PPU 側から測定した場合。したがって、転送時間もすべて含まれる）。どちらの方法がより高速かは、入力データによって異なります。たとえば、最大セグメントサイズが 2KB まで減った場合、セグメント化されていない伸長の方が著しく高速になります。

「[sample/edge/zlib-large-segmented-inflate-sample](#)」および

「[samples/edge/zlib-large-unsegmented-inflate-sample](#)」は、この 2 つの伸長法の例を示しています。同じようなテクニックは、SPU 上での Edge Zlib 圧縮にも適用されます。

Edge Zlibのビルド

Edge Zlib の SPU ライブラリをビルドする際には、メイクファイルや VSI を通して、コマンドラインからコンパイラに次の定義を渡します。

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

これらの定義は、Edge Zlib に対して、圧縮中に LS に収まるようなより小さなバッファを利用することを指示します。また、伸長中により大きなバッファを利用して、Zlib がオフラインデータとの互換性を維持できるようにします。これらの定数をコード中で利用する際には、これらが適切に同期されていることを確認するようにしてください。

Edge Zlib の SPU ライブラリは、デバッグモードのときでも、常に最適化付きでビルドされます。最適化をオフにすると、コードが大きくなりすぎて、ローカルストアに入りきらなくなるからです。

14 Edge LZMAの詳細

特徴

LZMA は、7-zip で使われているロスレスデータ圧縮アルゴリズムです。このリリースは Edge LZMA を含み、PlayStation®3 の SPU 上で実行できるように修正・最適化された LZMA 伸長の実装が含まれています。現在 Edge LZMA が提供するののは、解凍機能の SPU 実装だけです。解凍用のデータは、7-zip のようなツール、もしくは LZMA SDK のような圧縮用のライブラリを使って、オフラインで生成することができます (<http://www.7-zip.org/sdk.html> より入手可)。

Edge LZMA と LZMA の違い

Edge LZMA は、LZMA にはない以下のような機能を提供します。

- SPU の命令セットやローカルストア向けに最適化されている。
- 確率バッファは、静的データであって、SPU 上で動的に割り当てられるのではない。
- 解凍用のセグメントを PPU からキューに入れることのできる SPURS タスクが用意されている。
- 解凍は通常、64KB のセグメントの中で動作する。解凍を実行する前の入力データはローカルストアに存在する必要がある、ローカルストアには、出力バッファのために十分な領域が存在する必要があります。
- 複数セグメントの解凍を複数の SPU 上で並列に実行できる。

圧縮データのヘッダ

Edge LZMA は、「属性」へのポインタ、および生の圧縮データストリームへのポインタを受け取ることを想定します。

データが `lzma.exe` や LZMA SDK で圧縮されている場合、フォーマットは 5 バイトの「属性」データと 8 バイトの伸長後サイズで始まり、このヘッダの後に、生の圧縮データストリームが続きます。このデータ形式を利用する場合には、データを Edge LZMA に渡す前に、解析してコンポーネントに分割する必要があります。

同様に、それ以外の一部のファイルフォーマット（.7z など）を使う際も、データを解析してコンポーネントに分割し、「属性」と生のデータストリームを別々に Edge LZMA に渡せるようにする必要があります。

これらのコメントは、`edgeLzmaInflateRawData()` のような低水準関数、および以下のような高水準関数にあてはまります。

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

伸長タスク

Edge LZMA 用の伸長タスクは、Edge Zlib のものとまったく同じように動作します。唯一注意すべきことは、Edge LZMA は LS に収まりきらない大きさの伸長をサポートしないため、これにより伸長タスクが処理する

バッファのサイズが制限されることです。伸張前の入力データは 64KB 以下、伸張後の出力データも 64KB 以下である必要があります。どちらかのバッファがこのサイズを超えると、SPU コードはアサートが発生します。

15 Edge LZOの詳細

特徴

LZO は、サイズよりも速度を重視した可逆データ圧縮アルゴリズムです。Edge LZO は、LZO 圧縮・伸長を、PlayStation®3 の SPU 上で実行するために、修正・最適化した実装を提供します。

LZOには、さまざまなアルゴリズムが用意されています。Edge LZOでは、LZO1Xの実装だけを重視しました。Edge LZOは、任意のLZO1Xデータを伸長することができます。伸長に適したデータは、LZO SDK (www.oberhumer.com/opensource/lzoから入手可) を使って、オフラインで生成することができます。

Edge LZOとLZOの違い

- SPU の命令セットやローカルストア向けに最適化されている。
- 圧縮・伸長用のセグメントをPPUからキューに入れることのできる SPURS タスクが用意されている。
- 圧縮・伸長は通常、64KB のセグメントの中で動作する。圧縮・伸長を実行する前の入力データはローカルストアに存在する必要がある、ローカルストアには、出力バッファのために十分な領域が存在する必要があります。
- 複数セグメントの圧縮・伸長を、個々の SPU 上で並列に実行できる。
- 実装されているのは、さまざまな LZO アルゴリズムのうちの 1 つだけです。

ライセンス

Edge LZO を、ミドルウェアやゲームタイトルなどのアプリケーション内で利用する際には、そのアプリケーションのゲームエンディングクレジットのセクション/ファイルの中に「Data compression by oberhumer.com」を表示する必要があります (該当する場合)。また、アプリケーションの取扱説明書などのドキュメントの中にも記載する必要があります。

圧縮データのヘッダ

Edge LZO は、Edge Zlib や Edge LZMA と同じく、生の圧縮データストリームのポインタを受け取ることを想定しています。ファイルのヘッダやフッタは、Edge LZO に渡す前に、解析して取り除く必要があります。このことは、`edgeLzolxInflateRawData()` や `edgeLzolxDeflateRawData()` のような低水準関数だけでなく、以下のような高水準関数にも当てはまります。

```
edgeLzolxAddInflateQueueElement(), edgeLzolxTryAddInflateQueueElement(),
edgeLzolxAddInflateQueueElementPartialCopyOut(),
edgeLzolxTryAddInflateQueueElementPartialCopyOut(),
edgeLzolxAddDeflateQueueElement(), edgeLzolxTryAddDeflateQueueElement().
```

伸長・圧縮タスク

Edge LZO 用の伸長・圧縮タスクは、Edge Zlib のものとまったく同じように動作します。ただし、Edge LZO では、ローカルストアに収まらないような大きなサイズの圧縮・伸長はサポートしていないので、伸長・圧縮タスクが処理できるバッファのサイズが制限されることを知っておいてください。特に、入出力デー

タは、64KB を超えないようにする必要があります。どちらかのバッファがこのサイズを超えると、SPU コードはアサートを発生します。

注意：SPU コードを変更した場合、その変更によっては、圧縮・伸長タスクの最小限のコンテキストだけを退避したのでは不十分で、より大きなコンテキスト領域を格納しなくてはならない可能性があります。

ローカルストア中の圧縮出力バッファ

注意：以下の問題が関係するのは、`edgeLzo1xDeflateRawData()` だけです。アプリケーションで使っているのが伸長タスクである場合には、この問題はすでに対処済みです。

LZO および Edge LZO は、圧縮を実行する際に、出力バッファには十分な領域があると仮定し、書き込み先がバッファの最後を越えているかどうかをチェックしません。したがって、ローカルストアの出力バッファには、出力圧縮データの可能な最大サイズを収容できるだけの大きさが必要です。最悪の場合の圧縮後の出力ブロックサイズは、次の式で計算されます。

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

したがって、たとえば、65,536 バイトの入力バッファが圧縮用に渡された場合、ローカルストア中の出力バッファは、最低 69,699 バイトを受け取れる必要があります。出力サイズは、入力データの 64KB より小さいことが理想的であるのは明らかですが、最高に増えた場合を前提とすることが重要です。

16 Edge DXTの概要

概要

Edge DXT は基本的に、生の画像データを、CELL_GCM_TEXTURE_COMPRESSED_DXT1、CELL_GCM_TEXTURE_COMPRESSED_DXT23、CELL_GCM_TEXTURE_COMPRESSED_DXT45 テクスチャフォーマットに圧縮することを可能にする SPU ライブラリです。

このライブラリは、これらのフォーマットを生の画像データに解凍し直す関数も提供しています。

Edge DXTの設計

この圧縮アルゴリズムは、圧縮速度の速さを目標に設計されています。したがって、圧縮の際にはさまざまな近道を行っています。圧縮結果の品質は、ほとんどの画像で許容できる範囲であるはずですが、オフラインで圧縮された画像ほど高品質ではありません。

サンプルでは SPURS ジョブを利用しており、このコードをコピーして、Edge DXT を自分のプロジェクトに組み込むための叩き台にすることができます。

- コードは、位置とは独立にコンパイルされますが、固定位置モデルで使うこともできます。
- このライブラリの中では、システムコールや DMA は行っていません。また、このライブラリは、生の SPU 上でも利用できます。

圧縮のパフォーマンスおよび制限

Edge DXT の解凍関数は、有効な DXT データのすべてを処理することができます。一方、圧縮には以下の制限が適用されます。

DXT1

アルファが不要な場合にも優れた圧縮パフォーマンスを維持するために、DXT1 圧縮には 2 つの関数が用意されています。

`edgeDxtCompress1()` 関数は、ソースのアルファチャンネルを無視し、完全に不透明な出力画像を生成します。この関数の圧縮用の内部ループは、DXT ブロックあたり 101 サイクルにスケジューリングされます。

`edgeDxtCompress1a()` 関数は、ソースのアルファを計算に入れて、1 ビットのアルファ付き画像を生成します。この関数の圧縮用の内部ループは、DXT ブロックあたり 120 サイクルにスケジューリングされます。

DXT3

圧縮用の内部ループは、DXT ブロックあたり 102 サイクルにスケジューリングされます。

DXT5

8 値アルファブロックエンコーディングだけが、使われます。

圧縮用の内部ループは、DXT ブロックあたり 126 サイクルにスケジューリングされます。

17 Edge DXTの利用

圧縮

Edge DXT の圧縮関数は、効率的な圧縮を実現するために、1 回の呼び出しごとにブロックの連続した行を 1 つ圧縮します。

入力データは、1 チャンネル当たり 8 ビット、ARGB のバイト順である必要があります。これは、CELL_GCM_SURFACE_A8R8G8B8 型の面と同じです。DXT ブロックの行 1 行には、画像データの行が 4 つ必要です。各行は 16 バイト境界から始まっていて、一定のストライドだけ離れている必要があります。

データを圧縮するには、edgeDxtCompress1()、edgeDxtCompress1a()、edgeDxtCompress3()、または edgeDxtCompress5() を呼び出して、LS 中の出力ブロックを格納する場所、LS 中の最初の行の場所、行間のストライド、および生成する DXT ブロックの数を渡します。

DXT1 の場合、ブロック出力位置は、8 バイト境界にアラインメントされている必要があります。DXT3 および DXT5 の場合、ブロック出力位置は、16 バイト境界にアラインメントされている必要があります。これを実現するために、必要に応じてデータを圧縮し、出力ブロックと最初の行に同じ LS 中の場所を渡してもかまいません。

解凍

Edge DXT の解凍関数は、効率的な解凍を実現するために、1 回の呼び出しごとにブロックの連続した行を 1 つ解凍します。

解凍関数のデータ配置は圧縮関数のそれを反映しており、圧縮関数と同じアラインメント制限があります。

注意：解凍の場合、その場でのデータ解凍は一般に安全ではありません。

18 Edge Postの概要

特徴

Edge Post は、フレームバッファの後処理エフェクトチェーンや、汎用イメージ変換の SPU 実装を書く際に役立ちます。このライブラリでは、このような実装を容易にするフレームワーク、およびこのシステムの利用例を示す一連の定義済みエフェクトを提供します。この定義済みエフェクトは、カスタマイズされた後処理エフェクトを開発するための叩き台にしたり、現在の後処理エフェクトを RSX®実装から SPU 実装に移植する例として使ったりするためのものです。

Edge Post を使って実現できるエフェクトの例としては、モーションブラー、被写界深度、ブルーム、トーンマッピングなどがあります。提供された例は、LDR カラーと HDR カラーのいずれかを利用して、モーションブラー、被写界深度、簡単なブルームを組み合わせたエフェクトを示しています。

SPU ローカルストアのサイズに制限があるので、処理はタイルごとに行われます。タイルのサイズは、PPU 上で（手動もしくはライブラリのヘルパー関数を使って）事前に選択され、入力画像の寸法、各タイルに必要な境界線の幅、出力画像の寸法、エフェクトのコードサイズなどの要因に依存します。SPU ローカルストアとの間のタイル転送には、DMA リストが頻繁に使われます。

実際にピクセル処理を実行するコードを書く責任は、ユーザ側にあります。このコードの提供方法は、ユーザが決めることができます。ライブラリがユーザに期待するのは、タイル 1 個を処理する関数のコールバックを提供することだけです。

サンプルが利用するシステムでは、コードが Edge Post ジョブとしてパッケージ化されます。つまり、コードは基本的に位置独立としてコンパイルされ、カスタムリンカスクリプトを使ってリンクされるということです。コードは、実行される特定のエフェクトにより、必要に応じて DMA 転送されます。これは、コード全体のサイズを最低限に維持するためです。

使用する方法としては、SPU DLL、オーバーレイ、エフェクトのコードを直接ライブラリにリンクするなど、さまざまな方法を選択することができます。

SPU タイルフレームワークは、ダブルバッファリングされたタイルの DMA 入出力、および同じワークロードを処理する他の SPU との同期、RSX®との同期（オプション）を管理します。フレームワークは、各タイルの処理準備ができるたびに、特定の処理用の指定されたコールバックを呼び出します。

SPUの使い方

サンプルでは、簡単な SPURS タスクを利用しています。このサンプルは、Edge Post をユーザプロジェクトに組み込むための叩き台として、コピーして利用することができますが、ライブラリ自体は、いかなる SPU プログラミングモデルも前提にしません。

SPU 利用法の詳細：

- コードは位置独立（PIC。-fpic オプション付きでコンパイル）としてコンパイルされます。
- DMA 操作は、すべて割り込みセーフです。
- このライブラリ中には、SPURS 関数呼出しは存在しません。

データ構造体

基本的なデータ構造は、EdgePostProcessStage です。この構造体には、Edge Post タイルフレームワークが操作を実行するために必要なあらゆる情報が含まれており、最高 4 つの入出力画像を保存することができます。Edge Post の入力 は EdgePostProcessStage の配列であり、この配列により、完全な「エフェクトチェーン」が記述されます。

EdgePostProcessStage には大量の情報量が含まれていて、非常に複雑ですが、そのフィールドを設定するのを支援する PPU 関数がいくつか用意されています。これらの関数は、入力画像のサイズに基づいて、適切なタイルサイズを選択しようとしています。

EdgePostWorkload は、ユーザからは見えない、2 次的なデータ構造を提供します。この構造体のサイズは 16 バイトで、16 バイトにアラインメントされています。この構造体は、SPU によって常にアトミックにアクセスされます。この構造体は、EdgePostProcessStage の配列へのポインタで初期化され、その実効アドレスは、メイン SPU Edge Post 関数にパラメータとして渡されます。

この構造体のフィールドは、同じエフェクトチェーンの処理をしている複数の SPU を同期するために使われます。

19 Edge Postの利用

基本的な手順

(1) エフェクトチェーンを作成する

一連の入力画像に適用したいエフェクトを、順番に並べたリストを作成します。この処理は、EdgePostProcessStage の配列を作成することによって実現されます。この配列の各要素は、固有の入力、出力、操作、操作パラメータの集合を定義します。EdgePostProcessStage 構造体は、正しいデータを設定するのに便利な PPU 関数も用意されていますし、手作業で設定することもできます。

(2) ワークロードを作成する

EdgePostWorkload を宣言して、edgePostInitializeWorkload() を呼び出し、エフェクトチェーンのポインタを渡して初期化します。

(3) SPU 初期化時

SPU コード (SPURS タスクなど) の中で edgePostSetSpuConfig() を呼び出して、Edge Post ライブラリを初期化します。この関数には、EdgePostSpuConfig 構造体を渡して、作業領域のサイズや利用できる DMA タグを Edge Post に知らせる必要があります。

(4) SPU 上での処理を開始する

SPU から edgePostRunWorkload を呼び出して、パラメータとしてワークロード構造体の実効アドレスを渡し、ワークロードの実行を開始 (またはそれに参加) します。

(5) 終了を待つ

PPU の上で edgePostStallForWorkload() を呼び出すことにより、ワークロードの終了を待つことができます。

20 Edge Postランタイム

処理ステージとエフェクトチェイン

各処理ステージは、以下の情報によって記述されます。

- タイルのサブディビジョン (numTileX x numTileY)
- 最高 4 つまでのタイル画像記述子:
 - 画像ピッチ、および実効アドレス
 - タイルの幅×高さ×bpp
 - タイルの境界線サイズ
- エフェクトコードの実効アドレス
- エフェクト固有パラメータのアドレスおよびサイズ
- 若干のユーザデータフィールド
- ステージの処理が完了したときに更新される、オプションの RSX®ラベルアドレス

ステージの処理に必要な情報は、すべて、EdgePostProcessStage 構造体に含まれています。

EdgePostProcessStage の配列を作成することによって、実質的に、指定された順序で実行する必要のあるエフェクトチェインを定義したことになります。

このエフェクトチェインは、(サンプルのように) 起動時に作成することもできるし、フレームごとに作成してエフェクトチェインを動的に変化させることもできます。

作成の済んだエフェクトチェインは、EdgePostWorkload 構造体を作成してから、エフェクトチェインのポインタで初期化し、さらに、この構造体の実効アドレスを SPU 関数 edgePostRunWorkload に渡すことにより、実行できます。

(SPU 上での) エフェクトチェインの実行中に、PPU は edgePostIsWorkloadFinished() 関数を呼び出して終了をチェックしたり、edgePostStallForWorkload() を呼び出して完了までストールしたりすることができます。

タイルサイズを選択

タイルフレームワークは、処理ステージごとに、全画像データを格納できるだけの領域、およびメインメモリとの間でタイルを転送するために使われる DMA リストのための領域、さらにオプションでエフェクトコード自体およびそのパラメータのための領域を、ローカルストアに割り当てる必要があります。

したがって、ローカルストアに十分な空き領域が確保できるようにするために、タイルサイズは注意深く選択する必要があります。ライブラリには、このための PPU ヘルパー関数が用意されています。この関数は、タイル候補の集合をスキャンして、適切なタイルを探します。この関数は、以下の事項をチェックします。

- 必要なメモリの総量が、利用できる領域の総量以下である。
- タイルサイズが、画像のサイズの約数である。
- DMA 転送のため、計算されたタイルピッチは、16 バイトにアラインメントされている必要がある。
- タイルの X、Y の数が、画像サイズの約数である必要がある。

上記の要件がすべて満たされている場合には、タイルサイズ選択用に用意された関数を使う必要はありません。必要なデータのすべてをローカルストアに収納できない場合、SPU ライブラリはアサーションを発行します。

各タイル記述子には、入力タイルの保存に必要な領域を変更するために利用できる、乗数と呼ばれる追加プロパティがあります。このプロパティは、入力タイルのピクセルフォーマットを、使用前に別のフォーマットに変換したい場合に便利です。

たとえば、ARGB8 は、RSX®フレームバッファのフォーマットとして一般的なので、この形式で画像が保存されることは珍しくありませんが、各チャンネルの処理は浮動小数点数を使って行う必要があります。このような場合、乗数として 4 を指定することができます。入力タイルは、DMA サイズとしてはそのまま ARGB8 として処理されますが、そのために割り当てられるメモリは 4 倍になり、タイルを（その場で）浮動小数点に変換できるようになります。

このテクニックの例は、特に、サンプリング後のモーションブラーや被写界深度のコード内にあります。

SPUのメモリレイアウト

各処理ステージにおける、SPU のメモリレイアウトは以下の通りです。

- エフェクトのコード。16 バイトアラインメント
- エフェクトのパラメータ。16 バイトアラインメント
- 各タイルのデータごと
 - タイルのピクセルデータ
 - タイル入力 DMA リスト

ステージが処理する画像が複数のタイルに再分割されている場合、ダブルバッファリングのため、各ピクセルデータの領域は、実際のタイルサイズの 2 倍になります。

そのようなメモリレイアウトは、新しいステージの実行が始まるたびに、再計算されます。

SPU同期

異なる SPU が同じエフェクトチェーンを処理する際の同期のメカニズムは、ユーザからは見えなくなっていますが、以下のようなガイドラインには従う必要があります。

- ワークロードの初期化は、1 回だけでは済みません。ワークロードを開始する前（たとえば、1 フレームに 1 回）には、毎回 `edgePostInitializeWorkload` を呼び出す必要があります。ワークロードの実行が完全に済んだら、再度 `edgePostInitializeWorkload` を呼び出す必要があります。
- 実行中ワークロードに対して `edgePostInitializeWorkload` を呼び出すのはやめてください。
- ワークロードの初期化が済めば、さまざまな SPU から好きなだけ `edgePostRunWorkload` を呼び出すことができます。各 SPU は、適切な処理ステージで処理を開始し、効果的に処理に参加します。同じように、より優先順位の高い作業がキューの中にある場合、SPU はいつでも処理をやめる（そして、後でまた参加する）ことができます。
- ワークロードが終了した後で `edgePostRunWorkload` を呼び出しても、何の効果もありません。

RSX®同期

RSX®と同期する必要がある状況には、異なる 2 種類の状況があります。

- RSX®が、Edge Post の処理を開始する必要がある。
- RSX®が、何らかの Edge Post 出力を待っている。

2 番目の状況の方がはるかに簡単なので、こちらからまず説明します。

RSX®との同期のための簡単なメカニズムが用意されています。各処理ステージは、オプションで、終了直後に、指定された値を RSX®ラベルに書き込むことができます。

たとえば、エフェクトチェーンの最後のステージは、フレームごとにインクリメントした値を、固定された RSX®ラベルアドレスに書き込むことにより、そのフレームバッファの後処理が完了したことを知らせることができます。その後、RSX®は、Edge Post の結果を取得して、処理（HUD を描画して、バッファを画面に表示するなど）を継続します。この方法の例は、Edge Post サンプルの中にあります。

Edge Post はライブラリであり、メイン SPU プログラムの制御をしているのはユーザ側なので、このメカニズムを RSX®との同期に使う義務はありません。それ以外の方法を適用すること、たとえば、SPU プログラムが、自分へのジャンプをクリアしたり、RSX®出力ポインタを直接移動したりすることも可能です。前者の方法は、システムに強く依存するので、ライブラリではいかなる「組み込み」ソリューションも用意していません。

以下は若干の例です。

1. ユーザは、cellGcmSetUserCommand を呼び出して、PPU 上の RSX®コールバックを起動することができます。そこから、Edge Post の処理を実行する SPURS タスクが開始されます。
2. ユーザは、SPURS ジョブの中に Edge Post を組み込んで、JTS（ジョブチェーン中の自分へのジャンプ）を利用することができます。この JTS は、RSX®のインライン転送によってクリアされます。
3. ユーザは、RSX®との特定の同期を行うカスタムポリシーモジュールに、Edge Post を組み込むことができます。

Edge Postが、RSX®によって転送されたホストメモリ上のピクセルデータにアクセスする必要がある場合には、さらなる注意が必要です。特に、二次元ビットマップ転送を使っている場合、RSX®側からは、ピクセルデータがホストメモリに完全に書き込まれたことを確認する方法がないので、確実に書き込むようにしてください。この話題に関する詳しい情報は、この章の後ほど「[Edge Postの組み込み、および推薦される使用法](#)」のセクションにあります。

PPU同期

ビジーウェイトループに基づく PPU との同期（つまり、PPU が Edge Post の完了を待つ）の方法が用意されています。詳しくは、edgePostStallForWorkload、および edgePostIsWorkloadFinished の項目を参照してください。

ビジーウェイトによる同期方式は、状況によっては最適でないこともあり、シグナルやミューテックスに基づくより優れた同期方式を、ユーザが直接実装することもできます。

Edge Post はいかなる OS の機能（ミューテックスなど）にも依存していないので、そのような機能は、ライブラリの範囲外であると見なされてきました。

また、Edge Post の場合、RSX®との同期の方がはるかに重要なので、PPU との同期はあまり重視されていません。

ライブラリの提供するSPUピクセル処理機能

Edge Post SPU ライブラリは、エフェクトコードの内部から呼び出すことができ、ピクセル処理に役立つ一連の関数を提供します。

そのような関数の例としては、パックされたピクセル形式と浮動小数点ピクセル形式との間の変換、RSX®デプスバッファ形式から浮動小数点表現への変換、モーションブラー、被写界深度などの簡単な実装などがあります。このような関数のより詳しい解説は、リファレンスマニュアルをお読みください。

これらの関数は、パフォーマンスを向上させるため、アセンブリで書かれています。ただし、リファレンス実装として、C による実装が存在する関数もあります（モーションブラーおよび被写界深度）。

Edge Post の各関数は、優れた効率性を実現するために、複数列のピクセルを同時に処理します。また、ほとんどの関数は、1 回のループで複数のピクセルを処理します。このため、通常、ライン方向のサイズは、4 もしくは 8 ピクセルの倍数であることが想定されています。また、入出力ポインタは、常に、16 バイトにアラインメントされている必要があります。

通常は、(その場でのピクセル処理を実現するために) 関数の入力と出力に同じポインタを渡しても安全ですが、それは、各ピクセルを互いに独立に計算できる場合に限りです。多くの処理では、特定の出力ピクセルを計算するために、近傍の複数のピクセルにアクセスする必要があります。そのような処理の例としては、ガウスフィルタ、モーションブラー、被写界深度などがあります。このような関数の入出力として同じポインタを渡すことは、通常安全ではありません。

既存の後処理パイプラインを RSX®実装から SPU 実装に移植する際には、各エフェクト用のカスタムフラグメントシェーダ、フレームバッファ中の情報（運動ベクトルなど）のカスタムレイアウトなど、チームによって実装が異なることが予想されます。これらのすべてを、ライブラリの中で一般化するのは極めて困難です。後処理エフェクトには標準的なソリューションがないので、万人に対して有効なソリューションを提供することは不可能です。

このプログラマブルな動作を SPU 上で模倣する 1 つの方法は、固定されたステージ効果パイプラインを提供する代わりに、チーム独自のエフェクト用フラグメントシェーダを書くのと同じような方法で、チーム独自の「タイル処理コード」を書ける機能を提供することです。

このような観点から見ると、ライブラリの中で提供されるピクセル処理関数は、むしろ、以下のようなときの、サンプルないし叩き台としての用途が想定されています。

- ライブラリの参照実装と異なる既存の後処理エフェクトシェーダを移植する
- 新しいエフェクトを作成する

edgePost.XXX ピクセル処理関数は、どれも、いかなる外部ライブラリやタイリングフレームワークにも依存していません。ピクセルデータの DMA 転送を処理するシステムをもつチームは、他のライブラリに依存せずに、なんの問題もなく Edge Post ピクセル処理関数を利用することができます。

エフェクトコードを書く

実際のエフェクトコードを書く際に主導権を持つのは、ライブラリのユーザです。

一般に、ユーザの提供する関数は以下のプロトタイプに従っている必要があります。

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

この関数は、ちょうど1個のタイルを処理します。これはアトミックな処理単位です。EdgePostTileInfoには、この関数を呼び出す前に書き込まれた、入出力タイルへのポインタが含まれています。また、現在処理中のタイル、および現在処理中のエフェクトチェーンのステージ、オプションのエフェクトパラメータのポインタに関する追加情報も含まれています。

ユーザは、このような関数を、エフェクトチェーンの各ステージに提供する必要があります。この関数は、どこに存在しても、Edge Post にとってはまったく問題ありません。たとえば、この Edge Post のサンプルは、要求に応じて XDR からローカルストアに関数をロードしています。この関数は、SPURS タスクにリンクすることも、SPU DLL の内部に持つこともできます。

Edge Post は、2つのコールバックによってユーザコードにフックします。このコールバックのポインタは、EdgePostSpuConfig 構造体の中で初期化する必要があります。

この構造体は、以下のように定義されます。

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

heapStart および heapSize は、ユーザが Edge Post に渡すメモリの領域を定義します。つまり、ライブラリが必要とするメモリはすべてこのプールから取得されるので、ライブラリは SPU 上のいかなるメモリ割り当て関数をも呼び出す必要はないということです。

controlDmaTag、inputDmaTag、outputDmaTag は、Edge Post が DMA 転送に使う DMA タグです。

pollCallback は、現在の SPU の使用をより優先順位の高い処理が要求しているかどうかを、Edge Post が確認するために使うコールバック関数です。簡単にするため、このコールバック関数も cellSpursTaskPoll と同じプロトタイプによって定義されます。

最後になりましたが、stageEnterCallback および stageExitCallback は、最も重要なコールバック関数です。これらのコールバック関数は、SPU が新しいステージの処理を開始するとき、および SPU が現在のステージの処理を終了するときや現在のステージが終了するときに、1回ずつ呼び出されます。stageEnterCallback は、ユーザの渡したタイル単位のコールバック関数のポインタを返します。たとえば、stageEnterCallback は、エフェクトコードを含むコードをローカルストアに DMA 転送して、そのポインタを返すことができますし、SPU DLL をロードして、cellSpudllSym を使って関数アドレスを

取得して、その関数アドレスを返すこともできます。また、stageEnterCallback は、パフォーマンスの監視を助ける SPURS トレース情報を出力することもできます。

また、Edge Post は、この関数が呼び出される前にバイナリデータをロードして、コールバック内の DMA ロードがストールすることを回避することもできます。たとえば、動的なコードをロードしたい場合には、ロードするかどうかや、どこからロードするかに関する情報は、EdgePostProcessStage 構造体に格納されます。

Edge Post サンプルは、カスタムビルド手順とリンカースクリプトの組合せを利用して、位置独立にロード・実行可能な小さい SPU バイナリを生成するために、このシステムを活用します。

以下は、サンプルの中で使われている「Edge Post ジョブ」そのものです。

```
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 は、このジョブのエントリ関数です。Edge Post は、各タイルを処理するたびに、この関数を呼び出します。このサンプルは、単に、最初の入力タイルを出力タイルにコピーするだけです。

以下は、サンプル付属のリンカースクリプト edgepost_job.ld を利用して、ジョブをビルドし、メインの実行可能ファイルとリンク可能な PPU オブジェクトファイルを作成する方法の例です。

```
% 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
```

さらに、PPU コードの中でジョブにアクセスできるようにする必要があります。

```
extern EdgePostJobHeader _binary_spu_job_bin_start;
```

サポートされているピクセルフォーマット

Edge Postがサポートするピクセルフォーマットを示したのが、表 37 です。

表 37 Edge Post でサポートされているピクセルフォーマット

フォーマット	解説
argb8	32 ビットサイズ、CELL_GCM_A8R8G8B8 と互換。
float	単一チャンネルの浮動小数点画像。デプス値を、D24S8 RSX フォーマットなどより SPU に適した方法で格納するために利用できます。RSX@デプスバッファからこのフォーマットに変換する関数が用意されています。
float4	完全精度の浮動小数点数。4 チャンネル argb 画像。
fx16	4 チャンネル。各チャンネルは、16 ビットで 0:5:11 の固定小数点数です。
Luv	CIE Luv 色空間。L は、16 ビットで 0:5:11 の固定小数点数で表されます。U、V は、正規化された 8 ビット値です。
FP16Luv	CIE Luv 色空間。L は、16 ビットの半精度浮動小数点数で表されます。U、V は、正規化された 8 ビット値です。

フォーマット	解説
LogLuv	CIELuv 色空間。L は、8:8 の固定小数点数に、対数として格納されます。U、V は、正規化された 8 ビット値です。
FP16	4 チャンネル。各チャンネルは、16 ビットに格納された半精度浮動小数点です。

フォーマットの間の変換関数としては、さまざまなものが用意されています。

FP16、FP16Luv、LogLuv は、RSX®と SPU の間の HDR カラーの交換を容易にするために提供されたものです。各フォーマットには、それぞれ、長所と短所があります。たとえば、FP16 は、RSX®が正しくフィルタリング/ダウンサンプリング/アップサンプリングすることのできる唯一の HDR フォーマットですが、1 ピクセル当たり 8 バイトを使用します。

FP16Luv、LogLuv は、1 ピクセル当たり 4 バイトですが、フィルタリングはできません。さらに、LogLuv は、SPU 上で計算量の多い指数・対数計算を必要とします。

Luv は、SPU ステージ間の中間 HDR カラーを格納するためのフォーマットとして提供されています。Luv は、メモリ使用量を低く抑えながら、浮動小数点との間の変換を FP16Luv や LogLuv より高速に行うことができます。

Float4 および fx16 は、高精度でなおかつサンプリング操作が高速なので、実際のピクセル処理を行う際により適したフォーマットです。Fx16、1 ピクセル当たり 8 バイトなので、より大きなタイルをローカルストアに格納することができます。

また、11:11:10 の 32 ビット RGB フォーマットのような、ユーザ独自のピクセルフォーマットを追加することも極めて簡単です。

Edge Postの組み込み、および推薦される使用法

Edge Post は、フレームバッファ後処理以外にも利用できますが、それが主な用途です。そのような選択をする主な理由は、RSX®時間をできるだけ多く解放して、他のレンダリングタスクで使えるようにすることです。

フレームバッファ後処理は、残念ながら、SPU が得意なジオメトリ処理とはまったく異なる種類の問題を提示します。生のピクセルの処理ならば、RSX®の方がはるかに高速です。RSX®が 60Hz フレームの 40% で実行できることを、SPU がやると 150%近くかかります。もちろん、SPU は複数個あります。

そのような長い遅延を隠すには、トリプルバッファリングが必要になることがあります。トリプルバッファリングの際には、フレームごとに 3 つの並列操作が存在します。

- PPU/SPU は、フレーム N-2 のためのコマンドバッファを生成します。
- RSX®はフレーム N-1 をレンダリングしますが、フレーム N を表示します。
- SPU は、フレーム N の後処理を行います。

トリプルバッファリングに関しては、考慮すべきいくつかの短所があります。主なものは以下の通りです。

- 入力遅延。プレイヤーに認識される反応時間が遅くなりすぎることがあります。これは、技術的問題というより設計時の選択です。
- ピクセルデータを保持するために、より多くのメモリを使う必要があります。カラーバッファは、最低でも 3 つ必要で、MRT を使っている環境や、後処理中にデプスバッファを保持する必要がある場合には、さらに多くなります。

慎重に考える必要のあるもう 1 つの重要な要素は、レンダーターゲットの場所、および入力データを Edge Post に渡す方法です。

RSX®ローカルメモリからの読み取りコストはきわめて高いので、SPU は、メインメモリから入力ピクセルデータを読み込む必要があります。また、PPU/SPU のローカルメモリへの書き込み速度は、RSX®の取り込み速度より遅いので、SPU は結果をメインメモリに書き込んで、それを RSX®によってローカルメモリに取り込ませるのが普通です。

RSX®は、SPU への入力として、ピクセルデータをメインメモリに配置する必要があります。これを実現するためにはいくつかの方法があります。そして、データをメインメモリにどのようにして配置するかは、Edge Post にとってはどうでもいいことですが、考慮すべき重要な要素です。

以下は、フレームバッファデータをメインメモリ経由で Edge Post に渡す可能な方法の簡単な一覧です。

- メインメモリ中にレンダーターゲットを作成して、そこに直接レンダリングする。
- RSX®ローカルメモリにレンダリングし、三次元ビットマップ転送を使ってピクセルを転送します（「巨大クワッド描画」法）。
- RSX®ローカルメモリにレンダリングして、二次元ビットマップ転送を使ってピクセルを転送します（`cellGcmSetTransferImage`）。

RSX®のバンド幅は、ローカルメモリへのレンダリングより低いので、1 番目のソリューション（メインメモリに直接レンダリング）にはかなり問題があります。しかも、（タイリングされたターゲット上でさえ）色圧縮もサポートされていません。

2 番目のソリューションは、ビットマップ転送の転送先がタイリングされたメモリである場合にだけ、3 番目のソリューションに匹敵しますが、これはさらなる問題を生み出します。

- つまり、単に Edge Post にデータを渡すだけのために、メインメモリ中に追加のタイル領域が必要になるということです。
- SPU は、ピクセルバッファのタイリングを解除して、タイリングされていない別の表面に結果を出力する必要があります。

実際、パフォーマンスに関しては、タイリングされたメインメモリへの三次元ビットマップ転送は、リニアなメインメモリへの二次元ビットマップ転送に匹敵します。けれども、メインメモリ中のリニアな（タイリングされていない）表面に三次元ビットマップ転送を行うと、`cellGcmSetTransferImage` を使って二次元ビットマップ転送を行うより、最悪 5 倍まで遅くなる可能性があります。

このような結果を考えると、最も簡単かつ効果的な方法は、3 番目のソリューション、つまり、リニアなメモリに単純な二次元ビットマップ転送を行う方法であるようです。この方法の大きな利点は、メモリの使用前に SPU 上でタイリングを解除する必要がないので、処理時間やさらなる一時バッファのための領域の節約になるということです。

また、ここで `cellGcmSetTransferScaleSurface` を使うと、最初の画像のダウンサンプリングにコピー操作を組み込むことができ、コピー操作や、最後のアップサンプリングとメインメモリからローカルメモリへのフレームバッファのコピーがより高速になるという利点があります。

さらなる利点として、`cellGcmSetTransferScaleSurface` では、アップサンプリングやダウンサンプリングの際に、点サンプリングだけでなくバイリニアサンプリングも利用できます。

表 38 は、異なる方法を使って、タイリングされたローカルメモリからメインメモリに 1280x720 の画像をコピーするテストの結果の一部を示しています。

表 38 タイリングされたローカルメモリからメインメモリに 1280x720 の画像をコピーするテスト結果の例

ローカルメモリからのデータ転送に使われた方法	コスト (ミリ秒)
タイリングされたホストメモリへの三次元クワッド	~0.55 ms
リニアなホストメモリへの三次元クワッド	~2.75 ms
タイリングされたホストメモリへの二次元ビットマップ転送	~0.45 ms
リニアなホストメモリへの二次元ビットマップ転送	~0.46 ms

後処理エフェクトのRSX®実装とSPU実装の違い

バイリニアテクスチャフィルタリングは、RSX®上ではほとんどただ同然で使うことができますが、SPU 上では、かなり計算コストがかかります。このため、バイリニアフィルタを使用もしくは実装した関数は、Edge Post にはありません。

簡単かつ正確な「最近傍フィルタリング」でさえ、(負の) 浮動小数点 UV 座標から正しいピクセル座標を計算するために、計算コストの多い `floorf` 関数が必要なので、SPU 上では計算コストが高くつく可能性があります。たとえば、相対 UV 座標が (-2.5, 0) に等しい場合、RSX®は、最近傍テクスチャ探索を行う際に、整数座標 (-3, 0) のピクセルをサンプリングします。これは、UV 座標の `floor` (床関数) を取得して、それを整数に変換した結果です。一般的な単純な `floor` 演算は、SPU 上では最高 4 命令で実行でき、取得した 1 サンプルにつき最高 6 サイクルかかります。

このことは、UV が常に正の浮動小数点数であれば、理論的に問題になりませんが、Edge Post では、UV は常に現在処理中のピクセルからの相対座標なので、UV が負になることは珍しくありません。

制限がわかっている場合には、`floor` をまったく使わずに、簡単な `cf1ts` 命令だけを使っても、目に見える品質悪化を伴わずに済ますことができます (画面上の結果は微妙に異なりますが)。上記の例では、UV 座標 (-2.5, 0) は、現在の中心ピクセルから (-2, 0) の位置にあるピクセルをサンプリングします。たとえば、モーションブラーなどを行う場合には、品質の差はほとんどわかりません。

多くの場合には、サンプリングしたいピクセルがどれかはあらかじめわかっているので、正しい UV 座標を持つ必要はありません。必要なのは、整数オフセットだけです。このことは、ガウスフィルタなどに当てはまります。

RSX®と SPU の違いでもう 1 つ考慮する必要があるのは、テクスチャのランダムアクセスです。

エフェクトの多くは、現在のピクセル座標を中心とする可変の半径の小円内部にフィルタサンプルが分布する、簡単な画像フィルタに還元されます (もしくは還元可能です)。この場合 (ガウスフィルタ、簡単なフィルタのほとんど、被写界深度、モーションブラーなど)、SPU 実装は、出力ピクセルの数より多くの入力ピクセルをローカルストアに取り込む必要がありますが、このことは一般には問題になりません。

このピクセルの「オーバーフェッチ」が行われるのは、各タイルの境界線に沿った場所です。これにより、たとえば、タイル座標 (0, 0) のピクセルから、タイル座標 (-1, 0) ピクセルにアクセスすることができます。各タイルにはタイル境界線のサイズを指定することができ、その上限を制限するのはローカルストアのサイズだけです。

上記の仮定があてはまらない (フィルタサンプルが 1 つ以上のテクスチャ上にランダムに分布する) 場合、テクスチャリングの唯一のソリューションはメインメモリからのランダムな DMA になるので、SPU 実装ははるかに困難になります。おそらく、ソフトウェアキャッシュを活用したとしても、SPU リソースを使う方法は最善ではありません。

このような理由から、RSX®実装と完全に一致する SPU 実装を書くことは、きわめて骨の折れる作業になる可能性があります。

パフォーマンス

フレーム全体 (1280×720) の操作は極めて計算コストが高くつくので、常にできるだけ低解像度の画像で作業するようにしてください。最初のダウンサンプリングや最後のアップサンプリングの操作を、RSX®コピーに組み込むようにしてください。640×360 から 1280×720 の比較的簡単なアップサンプリングは、1 個の SPU 上で、最高 2ms かかります (複数の SPU 上で並列処理しても問題はありませんが)。

表 39 には、SN Tunerで取得した、Edge PostサンプルSPU利用率の内訳が記載されています。ここでは、Edge Postを実行するSPUは 1 個だけに制限されていることに注意してください。並列処理を行うと、パフォーマンスは、ほぼ線形にスケールアップされます。

表 39 各操作 (640x360 画像) のサンプル SPU 利用率 - コスト (ミリ秒)

操作	1 個の SPU 上のコスト (ミリ秒)
被写界深度によるピンぼけ	0.9 ms
被写界深度	5.7 ms
モーションブラー	4.7 ms
ガウスフィルタ	1.9 ms
ブルーム	0.2 ms
レンズ内面反射	0.2 ms
ダウンサンプリング	0.38 ms
アップサンプリング	1.4 ms
ラスタ演算 (ブレンドなど)	1.16 ms

表 40 には、選択されたEdge Post処理機能のピクセル当たりの平均サイクルの内訳が記載されています。

表 40 各機能のピクセル当たりのサイクル数 - ピクセル当たりのサイクル数によるコスト

機能	推定サイクルコスト
モーションブラー	45 サイクル/ピクセル
被写界深度	75 サイクル/ピクセル
ブルームキャプチャ	10 サイクル/ピクセル
ガウス 7x1	12 サイクル/ピクセル
ガウス 1x7	4 サイクル/ピクセル

表 41 は、モーションブラー機能の、組み込みバージョンとSPAバージョンの間の比較を示しています。

表 41 各実装 (1280x720 画像) のモーションブラー操作のサンプル SPU 利用率 - ミリ秒コスト

機能 (組み込み対 SPA)	SPU 上で 1280x720 画像を処理する時間
組み込みモーションブラー	~27.5 ms
SPA モーションブラー	~17.6 ms

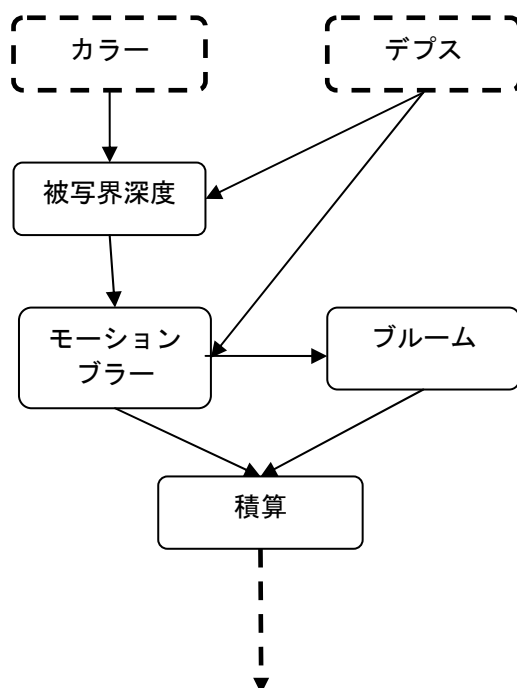
21 Edge Post備考

Edge Postのデータフローの例

この章には、Edge Postを利用してpost-sampleとして実装された後処理に関するデータフローを表す一連の図が記載されています（低ダイナミックレンジのサンプル。詳しくは、第23章「[Edge Post](#)」セクションの「samples/edge/post-sample」、「samples/edge/post-sample-hdr」、「samples/edge/post-sample-mlaa」を参照してください）。

図3は、post-sampleの中に実装された後処理チェーンをマクロ的に視覚化したものを示しています。

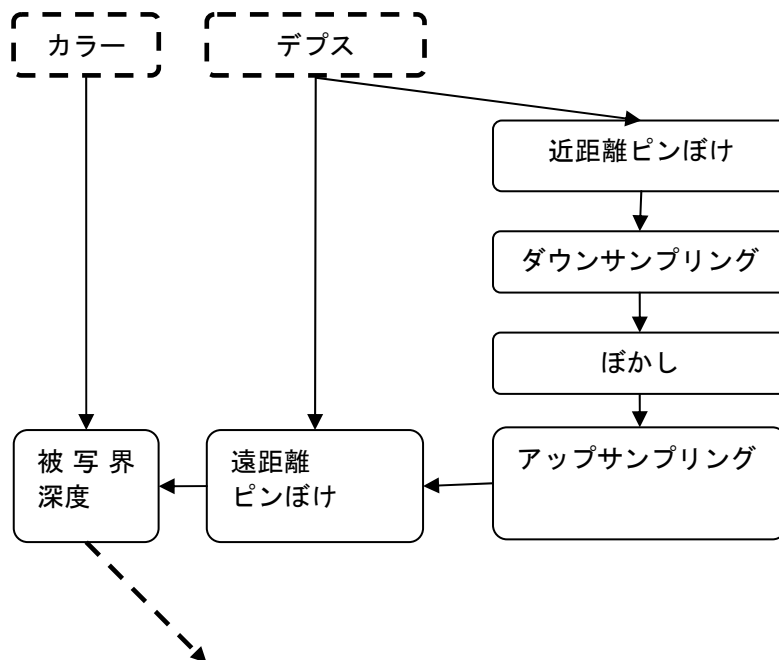
図3 データフロー（マクロ的な視点）



簡単にいえば、RSX®によって2つの入力（カラー・デプス/運動ベクトル）が渡されます。被写界深度とモーションブラーは、直列的に適用されます。モーションブラーの出力はブルームの入力であり、最終結果はモーションブラーの結果とブルームの結果を組み合わせたものになります。

図4は、被写界深度モジュールのデータフローの詳細を示しています。

図4 「被写界深度」コンポーネントのデータフロー



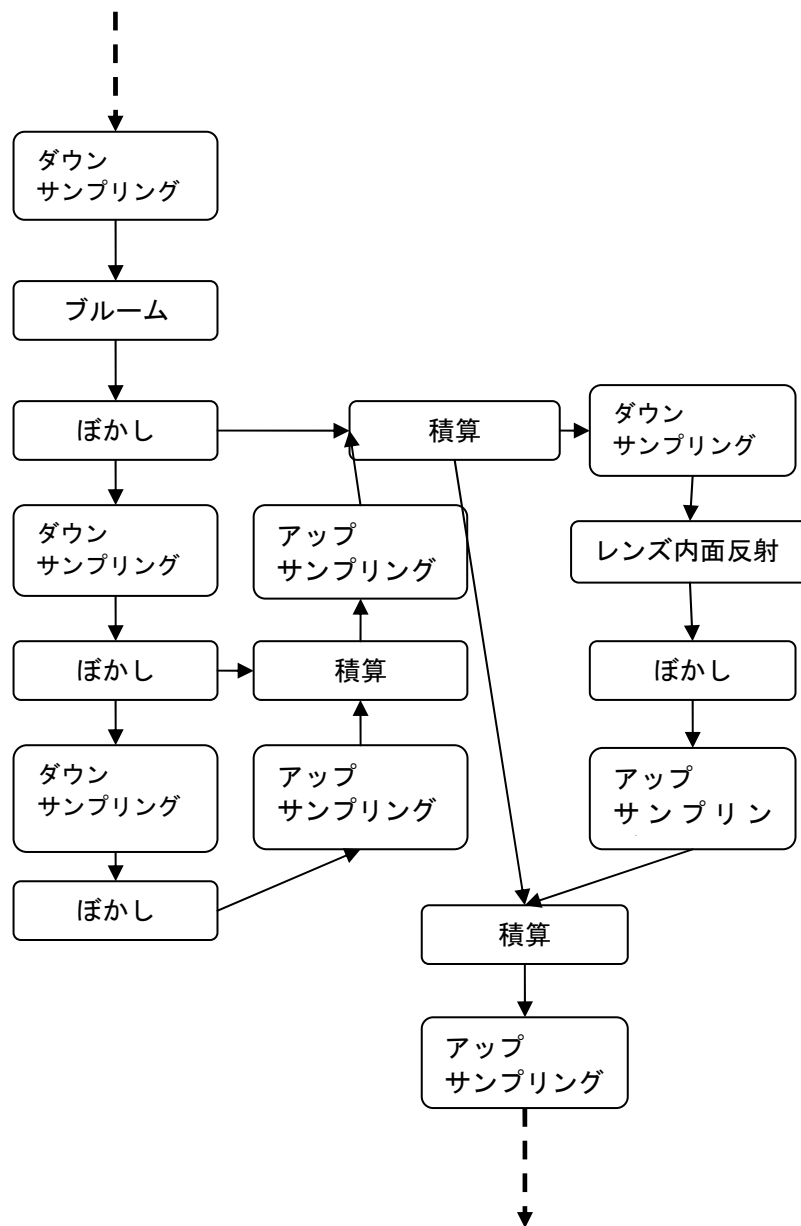
デプスバッファは、近距離ピンぼけバッファに変換され、さらにダウンサンプリングされて、ぼかされます。近距離ピンぼけバッファのぼかしは、焦点合った背景の前にある焦点の合わない前景物体の輪郭をよりスムーズにすることができます。

それから、遠距離・近距離のピンぼけを含む最終的なピンぼけバッファが生成され、元のカラーバッファと共に被写界深度モジュールに渡されます。

出力は、ピクセルのピンぼけがアルファに設定された ARGB 32 ビット画像です。

図5は、ブルームモジュールのためにデータフロー詳細を表示します。

図5 「ブルーム」コンポーネントのデータフロー



モーションブラーの出力は、まずダウンサンプリングされてから、ブルーム色に変換されます。さらに、このバッファはダウンサンプリングされ、ぼかしを複数回かけられます。そして、3つの異なるサイズのブルームバッファが積算され、その結果は再度ダウンサンプリングされて、レンズ内面反射（ILR）モジュールに入力されます。

最後に、ILR 出力とブルーム出力が加算され、アップサンプリングされた結果が、モーションブラーモジュールの出力と合成されます。

22 Edge Post MLAA

概要

Edge Post MLAA（形態学的アンチエイリアス）システムの目的は、XRGB8 画像にアンチエイリアスを行うもう一つの方法を提供することです。このアルゴリズムは、完全に画像ベースで、シングルサンプル画像に対して実行することを想定しています。このアルゴリズムは、アルファテストされたジオメトリやテクスチャを含む、エイリアス後の直線を発見して平滑化させます。このアルゴリズムが処理するのはシングルサンプルのバッファなので、サブピクセルの動きを補償するために必要な情報は含まれていません。

基本的な手順

- (1) EdgePostMlaaContext を PPU 上で作成して、edgePostMlaaInitializeContext () を呼び出して初期化します。これが唯一の初期化です。
- (2) edgePostMlaaPrepareWithRelativeThreshold () を呼び出すことにより、MLAA 処理を設定します。ここでソースとデスティネーションバッファに関連する全パラメータが設定されるので、他の MLAA コンテキストを作成することなく簡単に変更できます。
- (3) RSX®画像転送などを経由してソース画像を XDR に書き込みます。
- (4) ソース画像の準備ができれば、edgePostMlaaKickTasks () を呼び出して処理を開始します。
- (5) edgePostMlaaWait () を呼び出して、MLAA 処理が完了するのを待ちます。
- (6) 処理後のバッファを使います。
- (7) 必要があれば次のフレームでも 2~6 の手順を繰り返します。

MLAAのフレーム中の配置

ここで重要な点は、MLAA をいつ行うべきかということです。MLAA をユーザインタフェースを描画する直前に配置すると、原則として目に見えるエッジのすべてが計算に入れられることが保証されるので、これが当然の選択のように見えるかもしれません。けれども、このアルゴリズムは、ブルーム、被写界深度、モーションブラーなどのぼかしのエフェクトを施す前に適用した方がよく機能します。最高の視覚品質を期待できるのは、不透明なジオメトリと透明なジオメトリの両方を含み、最終的な輝度を持つ（つまり、トーンマッピングが適用された）画像です。ジオメトリ情報は計算に入れないので、反射や屈折やシャドウエッジさえも処理されます。ほとんどの場合、フレーム処理をインタリーブしない限り、MLAA の処理中に RSX®が実行するタスクを見つけることは難しくなります。そのような作業のよい例としては、次のフレームのシャドウマップやデプスのみのパスがあります。

エッジ検出およびチューニング

パフォーマンスや画質は、エッジ検出アルゴリズムに依存します。検出されたエッジが少なすぎると、認識可能な折り返し歪みが残ります。検出されたエッジが多すぎると、画像がぼやけノイズが多くなります。後者は予想外かもしれませんが、MLAA はぼかしアルゴリズムではないので、検出されたエッジが多すぎる場合、画像に様な平滑化を適用しません。

したがって、エッジ検出システムのパラメータが特定のゲームに適しているかを確認することが重要です。ゲームレベルによってコントラストのレベルも異なる場合、レベルに応じてパラメータを変更するのも有益かもしれません。ただし、このパラメータをゲームプレイ中に変更すると、ポッピングアーティファクトを引き起こす可能性があることを忘れないでください。

現時点で提供されている唯一のエッジ検出アルゴリズムは、隣接するピクセルのカラーチャンネルの差の絶対値のしきい値を使います。このアルゴリズムは、`edgePostMlaaPrepareWithRelativeThreshold()` の `base` と `scale` という 2 つのパラメータによって制御します。

最初のアルゴリズムは、指定されたピクセル p に対する、ピクセルしきい値 $t(p) := \max(\text{base}, \max(p.r, p.g, p.b) * \text{scale})$ を計算します。

2 ピクセル $p0, p1$ がエッジを構成するかどうかテストする際には、最大絶対差 $a(p0, p1) := \max(|p0.r - p1.r|, |p0.g - p1.g|, |p0.b - p1.b|)$ を計算して、 $p0, p1$ のピクセルしきい値の最小値と比較します。最終的な述語は、 $\text{edge}(p0, p1) := \min(t(p0), t(p1)) \leq a(p0, p1)$ になります。

以下は全般的なガイドラインです。

- 高輝度の部分がぼやけ過ぎる場合には、`scale` を大きくします。
- 検出されたエッジが少なすぎる場合には、`scale` を小さくします。
- 暗い部分が失われる場合には、`base` を小さくします。
- 明るい部分と暗い部分の間のブリーディングが多すぎる場合には、`base` を増やしてみてください。ただし、`scale` が小さすぎる可能性が高いです。

`scale` パラメータは 0.8 固定小数点数としてエンコードされます。弊社では、`base` が 0xA、`scale` が 0x59 のときにより結果が出ています。ただし、この結果はあくまで叩き台として考えてください。

性能特性

5 つの SPU で 720p の画像を処理した場合、全 SPU 時間の平均は約 10ms、範囲はおよそ 7ms～14ms です。コードは任意の数の SPU 上で実行できますが、SPU の解放は、システムの負荷を処理するために十分な速さでは行われません。弊社の測定によれば、SPU の数を 1 個から 5 個に増やした際の全 SPU 時間の増加は、約 600 μ s です。

画像は同じ大きさのストリップに分割されるので、タスクの一部は他のタスクが完了するまでアイドル状態になります。このアイドル状態の間に SPU は解放されるので、他の処理をスケジューリングすることができます。MLAA 処理のタスクは、複数の物理的な SPU 上に同時に存在する必要はありませんが、レイテンシを最適化するためにはそうすることをお勧めします。

約 400 μ s の部分が 2 つあって、1 個以上の SPU 上で使われた場合、システムは XDR バンド幅を完全に飽和させる可能性があります。これが問題になる場合（問題にならない可能性の方が高いですが）には、`EDGE_POST_MODE_SINGLE_SPU_TRANSPOSE` モードを使うと、XDR の負荷のピークを制限することができます。

このアルゴリズムはその場で処理できるように設計されていて、XDR メモリ中に必要な画像バッファは 1 つだけです。その場で処理するかどうかにかかわらず、ターゲットバッファは、ソース画像の高さを最も近い 128 の倍数まで拡大したソース画像が収まるだけの大きさが必要です。

この制限は、パフォーマンスが問題でない場合には、`EDGE_POST_MLAA_MODE_TRANSPOSE_64` モードを利用すれば緩和できます。このモードは 64 の倍数で動作します。このモードでは、約 2ms が必要であり、なおかつ、XDR メモリシステムへの負荷も増える可能性があるため、このモードの利用を検討するのは、デフォルトモードでメモリの使用量が増えることが重大な問題になる場合だけにしてください。

サンプル

Edge Post MLAA は、`post-sample` および `post-sample-mlaa` の 2 つのサンプルに組み込まれています（第 23 章「[Edge Post](#)」の「[サンプルプログラム](#)」のセクションを参照）。`samples/edge/post-sample` は、MLAA が実際に動作する様子、および MLAA をレンダリンググループや Edge Post のほかの部分と統合する方

法を示します。samples/edge/post-sample-mlaaはMLAAを 720pテクスチャに対して実行します。このサンプルは、別のテスト画像のパフォーマンスをプロファイリングする際にも利用できます。

制限

- ソース画像の寸法は、どちらの辺も 1280 ピクセル以下である必要があります。
- 画像の幅は 128 (EDGE_POST_MLAA_MODE_TRANSPOSE_64 が設定されている場合には 64) で割り切れる必要があります。
- ターゲットバッファは、高さを最も近い 128 (EDGE_POST_MLAA_MODE_TRANSPOSE_64 が設定されている場合は 64) の倍数に拡大した画像を収容できる必要があります。
- 画像は、8 ビット ARGB または XRGB である必要があります。
- 操作が完了後のアルファチャンネルの内容は未定義です。この内容は、次のセクションで説明するオプション機能を通じて制御できます。

オプションの機能

MLAA システムには、開発を容易にしたり、動作を変更したりするために利用できる、オプション部品がいくつかあります。このオプション部分は、プリプロセッサ定数によって制御されており、ライブラリをビルドし直す必要があります。

- **edgepost_mlaa.cpp** : `ENABLE_DEBUG_FEATURES`。検出されたエッジや発見された分離線の視覚化を可能にするコードが追加されます。視覚化の方法は、`DEBUG_MODE` を `SHOW_EDGES` または `SHOW_SEPARATION_LINES` のいずれかに定義することにより選択できます。どちらのモードでも、水平線は赤色、垂直線は緑色、両方の線を含むピクセルは黄色で表示されます。エッジ視覚化のサポートは、パフォーマンスにはほとんど影響がなく、デバッグには極めて便利なので、デフォルトで有効になっています。分離線の視覚化モードでは、分離線の先端を `0xff`、内部のピクセルを `0x80`、終端のピクセルを `0xd0` でマークします。`SHOW_SEPARATION_LINES` のサポートを有効にすると、パフォーマンスに影響があります。
- **edgepost_mlaa.cpp** : `CLEAR_ALPHA`。このフラグは、処理後のアルファチャンネルが未定義状態になることが望ましくない場合に、チャンネルのクリアを可能にします。処理中にアルファチャンネルを維持することはできません。クリアには約 160 μ s が必要であり、書き込まれる値は `CLEAR_ALPHA_VALUE` によって定義されます。
- **edgepost_mlaa_blend.spa** : `WRITE_BLENDED`。これを 1 に設定すると、ブレンド関数によって操作された全ピクセルを赤 (Z 形状のもの) または緑 (U 形状のもの) にマークします。発見されたエッジが実際にはすべてブレンドされるわけではなく、また、ブレンドコードが操作するのはエッジ上のピクセルだけではないので、この結果はエッジの視覚化とは異なります。
- **edgepost_mlaa_compress_seplines.spa** : `BLEND_DISCARDED`。パフォーマンスを維持するため、パイプラインのこの段階で、単一ピクセルの特徴を廃棄します。そのような特徴は、特に対角線の画質を維持するため、ここで直接ブレンドされます。一方、エッジの段はエッジに直交する単一ピクセルの特徴なので、このオプションはエッジの段のそばにアーティファクトが発生する原因になります。エッジの長さが最低 2 ピクセル以上あれば、`BLEND_DISCARDED` を使わなくても完全にスムーズなエッジが得られます。一般に、このアーティファクトはほとんど目立たず、対角線のスムーズさや時間コヒーレンスが向上することにより視覚的により快くなります。
- **edgepost_mlaa_write_threshold.spa** : `SCALE_FROM_ALPHA`。エッジ検出にグローバルな `scale` パラメータを使うことが、あらゆる状況で理想的であるとは限りません。このフラグを使うと、`scale` パラメータをソース画像のアルファチャンネルからピクセル単位で取得することができます。エッジ検出を完全に無効にするには、`scale` を 255 にしただけでは不十分なので、これを可能にする第 2 のオプション `DISABLE_ON_255` が用意されています。各オプションは、処理時間を約 60 μ s 増加します。

SPURSタスクの変更

MLAA は、付属のハンドライブラリ経由で利用するように設計されています。このライブラリには SPURS タスクが含まれています。決まったタイミング、具体的には、MLAA 処理が実行されていないときや、SPU が処理する作業がなくなったときには、タスクは SPU を解放して他の作業を実行できるようにします。こういった場合には、メモリを維持するため、スタック 2KB 分だけが保存されます。これ以外のデータ（グローバル変数や追加のスタック領域）を保存する必要のある変更が行われた場合には、ハンドラコードもそれに応じて変更する必要があります。

23 サンプルプログラム

Edgeジオメトリ

プログラム	解説
<code>samples/edge/collision-sample</code>	光線・三角形の衝突処理を行うための Edge ジオメトリの拡張を示したサンプル
<code>samples/edge/conditional-sample</code>	オクルージョンクエリーの結果に基づく Edge ジョブの条件付きレンダリングを示したサンプル
<code>samples/edge/occluders-sample</code>	Edge ジオメトリで四辺形オクルーダーを使って隠れた三角形をカリングする方法を示したサンプル
<code>samples/edge/runtime-partitioning-sample</code>	実行時に PPU 上で libedgegeomtool パーティショナーを実行する方法を示したサンプル
<code>samples/edge/tunable-sample</code>	Edge ジオメトリのパフォーマンスに対するさまざまなパラメータのエフェクトをユーザに示すサンプル
<code>samples/edge/vertexes-only-sample</code>	オフラインの前処理なしで、パーティションに分割されていない頂点データだけを処理するための、Edge ジオメトリの基本的な使用法を示したサンプル
<code>samples/edge/writeback-sample</code>	Edge ジオメトリによって計算されたデータを、以降のフレームで利用するために、メインメモリに書き戻す方法を、単純なパーティクルシステムの形で示したサンプル

Edgeアニメーション

プログラム	解説
<code>samples/edge/anim-sample</code>	Edge アニメーションの基本的な使用法を示したサンプル
<code>samples/edge/locomotion-sample</code>	キャラクターの移動運動に対する Edge アニメーションの基本的な使用法を示したサンプル。ブレンドツリー処理のさまざまな段階でユーザデータを処理する際には、コールバック関数が使われます。
<code>samples/edge/mirror-sample</code>	Edge Animation を用いてアニメーションミラーリングを行う基本的な方法を示したサンプル

Edgeジオメトリおよびアニメーション

プログラム	解説
<code>samples/edge/geom-sample</code>	edgegeomcompiler の出力データに対する Edge ジオメトリの基本的な使用法を示したサンプル
<code>samples/edge/elephant-sample</code>	Edge ジオメトリがスキニングされた複雑なキャラクターに適用され、edgegeomcompiler によって処理されるような基本的な使用法を示したサンプル
<code>samples/edge/blendshape-anim-sample</code>	Edge アニメーションと Edge ジオメトリを同時に利用して、アニメーションするブレンド形状を含むメッシュを描画する方法を示す高度なサンプル
<code>samples/edge/character-sample</code>	Edge アニメーションと Edge ジオメトリを一緒に使って、

プログラム	解説
	別個にアニメートされた複数のキャラクタを描画する方法を示す高度なサンプル
<code>samples/edge/fragment-patch-sample</code>	SPU 上のフラグメント・プログラムの定数に対し、「自分へのジャンプ」方式の同期を利用して、正確かつ効率的にパッチを当てる方法を示したサンプル

Edge Zlib・LZMA・LZO

以下の表は、Edge Zlib・LZMA・LZO を利用したサンプルプログラムの一覧です。これらのサンプルでは、画面に何も表示されないことに注意してください。出力はすべてキャラクタ端末上に出力されます。

プログラム	解説
<code>samples/edge/zlib-basic-inflate-sample</code>	Edge Zlib の基本的な使用法を示したサンプル。このサンプルは、「.gz」フォーマットのファイルをロードして、SPU 上で伸張し、さらに、正しく伸張されたことを証明するために、マスタバージョンと照合します。伸張の際に、入力データと出力データの両方がともに LS の中に収まるように、このデータは常に 64KB 以下である必要があります。
<code>samples/edge/zlib-basic-deflate-inflate-sample</code>	Edge Zlib を使って、まずバッファを圧縮し、その後、再び伸張して、元のマスタデータに戻すサンプル。PPU は、マスタファイルをロードし、その作業を SPU によって圧縮するために、圧縮キューに配置します。SPU はこの作業を受け取って、圧縮し、圧縮後のデータを指定された実効アドレスに送り出します。それから、もう一度 Edge Zlib を使ってデータを再伸張し、最終データが元のマスタバージョンと同じであることを確認します。
<code>samples/edge/zlib-large-unsegmented-inflate-sample</code>	サイズが 64KB より大きなファイルを、SPU が必要になった時点でメインメモリからデータをフェッチすることによって、Edge Zlib で伸張する方法のサンプル。PPU は、.gz ファイルをロードし、その作業を SPU によって伸張するために、伸長キューに配置します。SPU は伸長キューから作業を取り出して、入力データを繰り返し走査しながら伸長を行い、伸張後のデータを指定された実効アドレスに送信します。最後に、伸張後の出力がマスタバージョンと比較され、正しいことが確認されます。
<code>samples/edge/zlib-large-segmented-inflate-sample</code>	Edge Zlib を利用した、サイズが 64KB を越えるファイルを複数の SPU 上の並列処理により伸張する方法のサンプル。PPU は、「.segs」ファイルをロードして、その全セグメントを SPU で伸張するために、伸長キューに入れます。SPU は、伸長キューからタスクを取り出して、セグメントを並列処理により伸長し、結果を送信します。最後に、伸張後の出力がマスタバージョンと比較され、正しいことが確認されます。

プログラム	解説
<code>samples/edge/zlib-inflate-inplace-sample</code>	<p>このサンプルは、 「samples/edge/zlib-large-segmented-inflate-sample」と同じように、大きなファイルの伸長を、複数のSPUの上でセグメントに分けて行う方法を示します。ただし、このサンプルでは、圧縮済みデータは、伸張先の出力バッファに直接ロードされます。これにより、入力データ用と出力データ用のバッファを別にもつ必要がなくなります。その場で行う伸長の詳細については、第 12 章「Edge Zlib、LZMA、および LZ0 の使用」の「共通の詳細」を参照してください。</p>
<code>samples/edge/zlib-segcomp-sample</code>	<p>データのセグメント化と圧縮を行うオフラインツールの使用方法を示したサンプルです。マスタファイルは、最初に（64KB の）セグメントに分割されます。その後、各セグメントは zlib 呼び出しによって、2 バイトヘッダおよび 4 バイトフッタの出力を行わない形で圧縮されます。圧縮されたデータはコンテナファイル（「.segs」）へ集積されていきます。上記のサンプルでは、このサンプルツールによって作成された出力を使用しています。</p> <p>ファイルを圧縮すると、元のファイルより大きくなることもあります。このような場合、出力に「圧縮」バージョンを格納する代わりに、より効率的な圧縮されていない生バージョンを格納することが選択されます。</p> <p>このサンプルは、単にツールの仕組みを示すためのものです。</p>
<code>samples/edge/zlib-streaming-inflate-sample</code>	<p>Edge Zlib でストリーミングを使ってサイズの大きい複数のファイルをどうやって伸張するかというサンプル。1 つの PPU ストリーミングスレッドは、一時に 1 つずつファイルをロードして伸張します。ファイルの各セグメントについて、それはディスクからセグメントをロードして宛先に格納し、SPU 伸長タスクを利用してその場で伸張します。各ファイルが伸張された後、ユーザコールバックを呼び出して、ファイルのマスタバージョンに対して伸長を行ったバージョンの検証をします。</p>
<code>samples/edge/zlib-localmemory-streaming-inflate-sample</code>	<p>このプログラムは zlib-streaming-sample を変更したもので、宛先がメインメモリではなくローカルメモリ（VRAM）となっています。ローカルメモリからの読み出しは非常に低速であるため、圧縮済みの各セグメントはメインメモリ上の 2 つの一時バッファのいずれかにロードされます。伸長タスクはこのメインメモリバッファを伸張し、伸張された出力がローカルメモリに格納されます。</p>
<code>samples/edge/zlib-inflate-retry-sample</code>	<p>このサンプルは、(人工的な) ディスク読取りエラーによって結果的に無効なデータが出力される例を示しています。Edge Zlib は SPU 上でデータを伸張しようとしますが、エラーを検出して、その事実を <code>workToDoCounter</code> の最上位ビットを通じて伝えます。PPU はこのエラーを検出して、データを読み直し、再度伸長を試みます。</p>

プログラム	解説
<code>samples/edge/lzma-basicinflate-sample</code>	Edge LZMA の基本的な使用法を示したサンプル。このサンプルは、圧縮ファイルをロードして、SPU 上で伸長し、さらに、正しく伸長されたことを証明するために、マスタバージョンと照合します。伸長の際に、入力データと出力データの両方が同時にローカルストアの中に収まるように、このデータは常に 64KB 以下である必要があります
<code>samples/edge/lzma-inflate-retry-sample</code>	このサンプルは、(人工的な) ディスク読取りエラーによって結果的に無効なデータが出力される例を示しています。Edge LZMA は SPU 上でデータを解凍しようとしますが、エラーを検出して、その事実を <code>m_eaWorkToDoCounter</code> の最上位ビットを通じて伝えます。PPU はこのエラーを検出して、データを読み直し、再度伸長を試みます。
<code>samples/edge/lzma-segcomp-sample</code>	データのセグメント化と圧縮を行うオフラインツールの使用法を示したサンプルです。マスタファイルは、最初に (64KB の) セグメントに分割されます。それから、LZMA を呼び出すことにより、各セグメントが圧縮されます。圧縮されたデータはコンテナファイル (「 <code>.segs</code> 」) へ集積されていきます。上記のサンプルでは、このサンプルツールによって作成された出力を使用しています。 ファイルを圧縮すると、元のファイルより大きくなることもあります。このような場合、出力に「圧縮」バージョンを格納する代わりに、より効率的な圧縮されていない生バージョンを格納することが選択されます。 このサンプルは、単にツールの仕組みを示すためのものです。
<code>samples/edge/lzma-large-segmented-inflate-sample</code>	Edge LZMA を利用して、64KB より大きなファイルを複数の SPU 上の並列処理により解凍する方法のサンプル。PPU は、「 <code>.segs</code> 」ファイルをロードして、その全セグメントを SPU で伸張するために、伸長キューに入れます。SPU は、解凍キューからタスクを取り出して、セグメントを並列処理により解凍し、結果を送信します。最後に、解凍後の出力が、マスタバージョンと比較され、正しいことが確認されます。
<code>samples/edge/lz0lx-basic-inflate-sample</code>	Edge LZ0 の基本的な使用法を示したサンプル。このサンプルは、圧縮ファイルをロードして、SPU 上で伸長し、さらに、正しく伸長されたことを証明するために、マスタバージョンと照合します。伸長の際に、入力データと出力データの両方が同時にローカルストアの中に収まるように、このデータは常に 64KB 以下である必要があります。
<code>samples/edge/lz0lx-basic-deflate-sample</code>	Edge LZ0 をバッファの圧縮に利用する方法のサンプル。PPU は、マスタファイルをロードし、その作業を SPU によって圧縮するために、圧縮キューに配置します。SPU はこの作業を受け取って、圧縮し、圧縮後のデータを指定された実効アドレスに送り出します。このバッファは、後でファイルに保存して、 「samples/edge/lz0lx-basic-inflate-sample」 の入力データとして利用することができます。

プログラム	解説
<code>samples/edge/lzo1x-inflate-retry-sample</code>	このサンプルは、(人工的な) ディスク読取りエラーによって結果的に無効なデータが出力される例を示しています。Edge LZ0 は SPU 上でデータを伸長しようとしませんが、エラーを検出して、その事実を <code>workToDoCounter</code> の最上位ビットを通じて伝えます。PPU はこのエラーを検出して、データを読み直し、再度伸長を試みます。
<code>samples/edge/lzo1x-large-segmented-inflate-sample</code>	Edge LZ0 を利用した、64KB より大きなファイルを複数の SPU 上の並列処理により伸張する方法のサンプルです。PPU は、 <code>.segs</code> ファイルをロードして、その全セグメントを SPU で伸張するために、伸長キューに入れます。SPU は、伸長キューからタスクを取り出して、セグメントを並列処理により伸張し、結果を送信します。最後に、伸長後の出力が、マスタバージョンと比較され、正しいことが確認されます。
<code>samples/edge/lzo1x-segcomp-sample</code>	データのセグメント化と圧縮を行うオフラインツールの使用方法のサンプルです。マスタファイルは、最初に 64KB のセグメントに分割されます。それから、各セグメントは LZ0 を呼び出すことにより圧縮され、圧縮されたデータはコンテナファイル (「 <code>.segs</code> 」) に収集されます。上記のサンプルでは、このサンプルツールによって作成された出力を使用しています。 圧縮を適用されたファイルは、元のファイルより大きくなることもあります。そのような場合には、圧縮後バージョンのかわりに、生の圧縮前バージョンを保存した方が効率的なので、こちらが出力に格納されます。 このサンプルは、単にツールの仕組みを示すためのものです。

Edge DXT

プログラム	解説
<code>samples/edge/dxt-sample</code>	このプログラムは、RSX®上で若干の画像データを生成し、SPU 上でその結果を DXT フォーマットの 1 つに圧縮し、さらに、RSX®上でその圧縮結果をテクスチャとして利用する、定常状態システムを示しています。この処理はすべて同一フレーム内で行われます。またデバッグのため、圧縮結果は SPU 上で生の画像データに解凍し直して、RSX®を使ってテクスチャとして表示されます。

Edge Post

プログラム	解説
<code>samples/edge/post-sample</code>	Edge Post の基本的な使用法を示したサンプル。このプログラムは、単純なシーンをロードして、レンダリングされた画像に後処理エフェクトのセットを適用します。適用されるエフェクトは、被写界深度、およびモーションブラー、ブルームです。また、このサンプルでは、参照のために、これらのエフェクトの基準となる RSX®実装も提供しています。 また、このサンプルには MLAA も組み込まれています。こ

プログラム	解説
	のサンプルは、MLAA が実際に動作する様子、および MLAA をレンダリンググループや Edge Post のほかの部分と統合する方法を示します。
<code>samples/edge/post-sample-hdr</code>	Edge Post の基本的な使用法を示したサンプル。このプログラムは、単純なシーンをロードして、レンダリングされた画像に後処理エフェクトのセットを適用します。適用されるエフェクトは、被写界深度、およびモーションブラー、ブルーム、平均輝度の計算です。後処理パイプラインはすべて、HDR カラーを使って実行されます。
<code>samples/edge/post-sample-mlaa</code>	このプログラムは、MLAA を 720p テクスチャに対して実行します。異なったテスト画像のパフォーマンスをプロファイリングする際にも利用できます。