

PlayStation®Edgeジオメトリライブラリ クイックスタート

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

目次

このドキュメントについて	3
目的	3
対象読者および前提条件	3
関連ドキュメント	3
表記法	3
1 はじめに	4
2 頂点のみの処理	5
SPU および SPURS の初期化	5
SPURS イベントフラグの初期化	5
出力バッファの初期化	5
Edge ジオメトリジョブアプリケーションの作成	6
頂点を処理するための SPURS ジョブを作成する	7
SPURS ジョブリストの作成	8
SPURS コマンドリストの作成	8
SPURS コマンドリストの実行	9
SPURS ジョブの完了を待つ	9
3 セグメント化されたジオメトリの処理	10
既存のツールパイプラインと libedgegeomtool の統合	10
SPU および SPURS の初期化	13
SPURS イベントフラグの初期化	13
出力バッファの初期化	13
Edge ジオメトリジョブアプリケーションの作成	14
頂点を処理するための SPURS ジョブを作成する	16
SPURS ジョブリストの作成	18
SPURS コマンドリストの作成	19
SPURS コマンドリストの実行	19
SPURS ジョブの完了を待つ	19

このドキュメントについて

目的

このクイックスタートガイドは、Edgeライブラリのジオメトリコンポーネントを使ったプログラミングを始めるために知る必要のある、基本的な知識を提供します。このガイドは、「関連ドキュメント」セクションに挙げた、Edgeの概要やジオメトリライブラリリファレンスと併せてご利用ください。

対象読者および前提条件

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開発しようとしているPlayStation®3ディベロッパのため書かれたものです。ディベロッパは、次の点を熟知していることを前提にしています。

- 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

このドキュメントと以下のドキュメントを併せて参照することにより、Edge の詳しい使用法およびリファレンス情報を得ることができます。

- PlayStation® Edge ライブラリ概要
- 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

COLLADA™ FX や COLLADA™ Physics を含む COLLADA™の使用法の詳細については、「COLLADA™ DOM プログラミングガイド」と「COLLADA™ DOM 1.4.0 リファレンスドキュメント」を参照してください。どちらも、PS3_COLLADA-141_DOM-201.zip パッケージの COLLADA_DOM/doc ディレクトリにあります。

表記法

このドキュメントでは、以下のような表記法を使います。

記法	意味
等幅フォント	プログラミングコードおよびリテラル（処理命令、レジスタ名、データ型、イベント、ファイル名など）を表します。また、関数、構造体、マクロなどの名前を表すこともあります。
青色+下線のテキスト	ハイパーリンクを表します（青色で表示されるのはカラープリンタやオンラインの場合だけです）。

1 はじめに

Edge ジオメトリには、2通りの使い方があります。1番目の使い方では、ツール処理は不要ですが、処理できるのは頂点だけで、三角形の関係を計算に入れることはできません。2番目の使い方では、ジオメトリに対して前処理を行って、セグメントに分割します。セグメント分割を行うと、ランタイムは頂点だけでなく三角形の関係を計算に入れることができるようになります。

[第2章の「頂点のみの処理」](#)では、ツールによる前処理を必要としない処理ルートを使う方法を説明します。この方法のサンプルは、vertexes-only-sampleとして提供されています。

[第3章の「セグメント化されたジオメトリの処理」](#)では、オフラインツールによるジオメトリデータに対する前処理を必要とする処理ルートの使用方法を説明します。この章では、ツールとの統合からランタイムとの統合までの、すべてを扱います。この方法のサンプルは、elephant-sampleとして提供されています。

2 頂点のみの処理

SPUおよびSPURSの初期化

最初の初期化手順は、SPU と SPURS を初期化することです。

SPU と SPURS を初期化するための擬似コード

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

SPURSイベントフラグの初期化

次の初期化手順は、ジオメトリ処理ジョブがいつ完了するかがわかるように、SPURS イベントフラグを作成することです。

SPURS イベントフラグを作成するための擬似コード

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

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

出力バッファの初期化

最初の手順は、処理された頂点を格納するメモリを用意することです。これは、通常、1 メガバイトにアラインメントされたメモリチャンクを割り当ててから、RSX®にマップすることによって実現されます。

出力バッファの割り当てとマッピングを行うための擬似コード

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

その次の手順は、使用する出力バッファの種類を選択することです。以下の種類の出力がサポートされています。

- ダブルバッファ付き直接出力
- シングルバッファ付き直接出力
- 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

バッファリングの各種類についての詳しい情報は、概要およびリファレンスドキュメントに記載されています（「このドキュメントについて」の「関連ドキュメント」セクションを参照）。説明の都合上、このドキュメントでは、シングルバッファ付き直接出力を使います。

直接出力を使った場合、各ジョブの処理済み頂点データは特定のアドレスに書き込まれます。これは、出力バッファ情報の実効アドレスを、適切な直接出力定数でマスクすることによって実現されます。EdgeGeomは、この実効アドレスを、出力バッファ情報構造体である EdgeGeomOutputBufferInfo の場所ではなく、割当て済みの出力バッファの場所として解釈します。

Edgeジオメトリジョブアプリケーションの作成

標準の SPURS ジョブとして実行される SPU プログラムを書いてください。

ジョブプログラムのための擬似コード

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

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

    EdgeGeomSpuContext ctx;

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

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

```

    }

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

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

```

頂点を処理するためのSPURSジョブを作成する

今度は、SPU 入出力頂点フォーマット、入出力頂点の頂点ストライド、入力フォーマット中の頂点属性の数を求める必要があります。このような情報を利用する処理を最もよく示すのは、以下の擬似コードです。

頂点ストリームを処理するジョブのセットを作成するための擬似コード

```

uint32_t CreateJobs(void *vertexes, int32_t vertexCount,
uint32_t inputVertexFormatId, uint32_t outputVertexFormatId,
uint32_t numAttributes, uint32_t inputStride,
uint32_t outputStride, uint32_t &outputBufferEa,
CellSpursJob256 &*jobs, CellGcmContextData *ctx)
{
    // 頂点の数を 8 の倍数に丸める
    vertexCount = (vertexCount + 7) & ~8;
    // 頂点のメモリコストを計算する
    uint32_t cost = max(inputStride, outputStride);
    cost += 16 * numAttributes;
    // セグメント当たりの頂点数を見つける
    int32_t numVertsPerSegment = AVAILABLE_SPU_SPACE / cost;
    numVertsPerSegment = min(numVertsPerSegment, 0xC000 / cost);
    numVertsPerSegment &= ~16;
    // 後で LibGCM が使用する出力アドレスの先頭を記録する
    uint32_t startingAddr = outputBufferEa;
    uint32_t vertexesEaA = (uint32_t)vertexes;
    while(vertexCount > 0)
    {
        uint32_t numVerts = min(vertexCount, numVertsPerSegment);
        uint64_t size = (numVerts * inputStride + 15) & ~16;
        uint64_t sizeA = min(size, 0x4000);
        uint64_t sizeB = min(size - sizeA, 0x4000);
        uint64_t sizeC = (size >= 0x8000) ? (size - 0x8000) : 0;
        uint32_t vertexesEaB = vertexesEaA + sizeA;
        uint32_t vertexesEaC = vertexesEaB + sizeB;

        // 「自分へのジャンプ」による同期
        if(ctx->current >= ctx->end)
        {
            if((*ctx->callback)(ctx, 1) != CELL_OK) return 0;
        }
        uint32_t holeEa = (uint32_t)ctx->current;
        uint32_t jumpOffset;
        cellGcmAddressToOffset(ctx->current, &jumpOffset);
        cellGcmSetJumpCommandUnsafeInline(ctx, jumpOffset);

        CellSpursJob256 *job = jobs++;
        job->workArea.dmaList[0] = vertexesEaA | (sizeA << 32);
        job->workArea.dmaList[1] = vertexesEaB | (sizeB << 32);
    }
}

```

```

    job->workArea.dmaList[2] = vertexesEaC | (sizeC << 32);
    // 最上位ビットは、出力先が特定のアドレスであることを示す
    job->workArea.userData[3] = outputBufferEa |
        EDGE_GEOM_DIRECT_OUTPUT_TO_MAIN_MEMORY;
    job->workArea.userData[4] = inputVertexFormatId;
    job->workArea.userData[5] = outputVertexFormatId;
    job->workArea.userData[6] = numVerts;
    job->workArea.userData[7] = numAttributes - 1;
    job->workArea.userData[8] = holeEa;
    job->header.eaBinary = (uintptr_t)_binary_spu_job_start;
    job->header.sizeBinary
        = CELL_SPURS_GET_SIZE_BINARY(_binary_spu_job_size);
    job->header.sizeDmaList = 3*8;
    job->header.eaHighInput = 0;
    job->header.useInOutBuffer = 1;
    job->header.sizeInOrInOut = 0xC000;
    job->header.sizeStack = 0;
    job->header.sizeScratch = (numVerts * numAttributes + 7) & -8;
    job->header.sizeCacheDmaList = 0;

    // 残る頂点の合計からこのジョブ中の頂点の数を引く
    vertexCount -= numVerts;
    // 前のジョブ中の頂点の分だけ
    // 頂点の実効アドレスおよび出力アドレスを増やす
    vertexesEaA += numVerts*inputStride;
    outputBufferEa += numVerts*outputStride;
}
// 次のジョブに備えて、次の 128 バイトの倍数まで切上げる。
outputBufferEa = (outputBufferEa + 0x7F) & ~0x7F;

return startingAddr;
}

```

関数によって返されるアドレスは、SPU によって書き込まれる出力頂点データの実効アドレスです。たとえば、`cellGcmSetDrawIndexArray()` や `cellGcmSetDrawArrays()` を使って頂点をレンダリングする際には、この関数によって出力されたアドレスを RSX® オフセットに変換してから、関連する `cellGcmSetVertexDataArray()` 呼び出しで利用する必要があります。

SPURS ジョブリストの作成

先の手順で生成したジョブを実行する前に、まず、SPURS ジョブリストを用意する必要があります。

SPURS ジョブリストを作成するための擬似コード

```

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

```

SPURS コマンドリストの作成

次は、新たに作られたジョブリストを、SPURS ジョブコマンドリストから呼び出す必要があります。

SPURS ジョブコマンドリストを作成するための擬似コード

```

static uint64_t command_list[5];
command_list[0] = CELL_SPURS_JOB_COMMAND_JOBLIST(&jobList);

```



```
command_list[1] = CELL_SPURS_JOB_COMMAND_SYNC;  
command_list[2] = CELL_SPURS_JOB_COMMAND_FLUSH;  
command_list[3] = CELL_SPURS_JOB_COMMAND_JOB(&jobEnd);  
command_list[4] = CELL_SPURS_JOB_COMMAND_END;
```

SPURSコマンドリストの実行

コマンドリストの作成が済んだら、それを実行する必要があります。

SPURS ジョブコマンドリストを実行するための擬似コード

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

SPURSジョブの完了を待つ

最後に、ジョブによって出力されたデータを RSX®が利用できるように、SPURS ジョブの完了を待つ必要があります。

SPURS ジョブコマンドリストを作成するための擬似コード

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

3 セグメント化されたジオメトリの処理

既存のツールパイプラインとlibedgegeomtoolの統合

libedgegeomtool は、低水準のコア関数 (libedgegeomtool.h で宣言)、および高水準に抽象化されたインタフェース (libedgegeomtool_wrap.h で宣言) の 2 種類のインタフェースをツール側に提供しています。この 2 つの中では、高水準のジオメトリ処理ツールのインタフェースの方が簡単で、次の操作で利用することができます。

- (1) スタジオ内部のフォーマットをロードします。
- (2) カスタムジオメトリ形式から EdgeGeomScene に変換します。
- (3) ジオメトリデータの望ましいフォーマット方法を決めて、その選択を EdgeGeomSegmentFormat オブジェクトに格納します。
- (4) libedgegeomtool で処理します。
- (5) スタジオ内のバイナリフォーマットでは、実行時構造を不透明体なブロックとして格納します。

手順 1 は、ディベロッパ側の責任です。また、手順 5 も、ディベロッパ独自の手順になる可能性が高いです。したがって、この章では、手順 2 から 4 について、実際に利用できる簡単な例を使ってさらに詳しく説明します。

まず、統合する人向けに、EdgeGeom 構造体への変換の概要を説明します。

libedgegeomtool の主な構造体の説明

EdgeGeomScene 構造体は、libedgegeomtool の高水準 API における最も重要な入力データです。この構造体は、アーティストが (Max®や Maya®のような) コンテンツ生成ツールからエクスポートした特定の三次元アセットに割り当てることを目的として作られています。EdgeGeomScene には、完全な精度を持つ圧縮されていないフォーマットで格納された、頂点、三角形リスト、スキニング、ブレンド形状データが含まれています。

EdgeGeomSegmentFormat 構造体は EdgeGeomScene の仲間で、シーンの出力データのフォーマット方法や、SPU 上で実行される処理についての基本的なデータを記述します。ユーザは、SPU 入力頂点ストリーム (実際に Edge ツールによって書き込まれ、Edge SPU ランタイムによって処理される圧縮データ) および SPU 出力頂点ストリーム (Edge SPU ランタイムによって生成され、レンダリングのために RSX®に渡される) の頂点ストリームフォーマットを指定する必要があります。さらにユーザは、ブレンド形状の (SPU によって処理される) 頂点差分ストリームのフォーマットと、SPU には送信されない RSX®のみの頂点ストリームのフォーマットをオプションで指定することができます。

高水準の libedgegeomtool 処理の出力は、EdgeGeomSegment オブジェクトの配列で、その要素のそれぞれに、特定の Edge SPU ジョブに必要なデータのすべてが含まれています。各セグメントオブジェクト中のデータ配列は、可視的ではありません。一般に、以後のツールコードでは、このデータ配列を直接操作すべきではありません (ただし、ディスクへの書き込みは除く)。

EdgeGeomScene への変換

最も簡単な変換方法は、ユーザ独自のメッシュフォーマットの中の頂点リストを走査して、各頂点の属性に対応する大きな float のフラット配列を生成することです (これが、シーンの `m_vertexes` 配列になり

ます)。 `m_vertexAttributeIndexes` 配列および `m_vertexAttributeIds` 配列中での各属性の先頭インデックスおよび属性 ID は、後にフレーバー記述構造体の中で、頂点ストリームを生成する際にデータをどこから読み始めるかをツールに教えるために使われるので、注意してください。

シーン中の三角形は、三角形ごとに三つの 32 ビットのインデックスをもつフラットな三角形リストとして保存されます。さらに、ユーザは、シーン中の各三角形のマテリアル ID を含む配列を別個に提供する必要があります。このマテリアル ID が必要なのは、Edge パーティショナーが複数のマテリアルにまたがったパーティションを作成しないためです。この ID は、Edge が直接利用することはないので、各シーン中のマテリアルを一意に識別できる値ならなんでもかまいません。

この配列は、ブレンド形状がサポートされている場合、実行時に属性の任意のサブセットをアニメーション可能にするためにも、生成する必要があります。そのフォーマットは、次のような単純なものです。

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

繰り返しますが、これは `m_vertexDeltas` に配列として格納されている float のフラットテーブルです。

重要：これらのテーブルに保存するデータは、対象となる属性値ではなく、その差分値である必要があります。スタジオのパイプラインによっては、この時点で引算を行う必要があるかもしれないし、すでに差分を計算済みである場合もあるでしょう。この配列には、ベースジオメトリ中に存在する全属性のデータを含んでいる必要はありません。 `m_blendedAttributeIds`、および `m_blendedAttributeIndexes` 配列中のブレンド対象となる任意の属性の部分集合を格納してください。

スキニングデータは、ちょうど 1 頂点当たり 4 インフルエンスの領域に格納されます。使われていない重みは、0.0 に設定する必要があります。また、使われていないマトリックスインデックスは、-1 に設定する必要があります。 `libedgegeomtool` のルーチンは、重みの合計を計算し、現在のマトリックスインデックスを特定するために、全入力を走査するので、それぞれの頂点の重み/インデックス内の明示的な順位付けは不要です。

頂点のマージ

入力シーンは複製された頂点を含むことがあります。この処理は不必要なコストで、実際、変換後のキャッシュミスやバンド幅の無駄使いなどにより、実行時の著しいパフォーマンス悪化の原因になる可能性があります。複製された頂点のインデックスをつけ直すために、 `edgeGeomMergeIdenticalVertexes()` という関数が用意されています。そのような操作は、ツールの作業メモリ中の依存する三角形リストのインデックスをつけ直すことを必要とするので、頂点データベースを変更することはありません。むしろ、重複する頂点を参照する任意の三角形のインデックスを、同じ属性データを持つ最初の頂点のインデックスに書き換えます。

このような問題は、理想的には、ソースデータの中で修正するべきですが、全 Edge シーン中の同一の頂点は、さらなる処理を行う前にマージしておくことをお勧めします。

EdgeGeomSegmentFormat を構築する

シーンが完成したら、Edge ツールがシーンをフォーマットする方法を選択する必要があります。最も重要なのは、使用する頂点ストリームフォーマットの選択です。フォーマットは、SPU 入力および出力頂点ストリームに対して指定する必要があります。また、シーンでブレンド形状が使われている場合には、ブレンド形状デルタストリームのフォーマットも指定する必要があります。RSX®へ直接送信される属性のため、RSX®のみのストリームのフォーマットを提供することも可能です（テクスチャ座標やライティング係数など）。

頂点フォーマットを指定するには、2 つの方法があります。最も簡単かつ効率的な方法は、 `edgegeom_structs.h` の中に定義されている Edge の組み込みフォーマットの 1 つを選択することです。

組み込みフォーマットの構築済みフォーマット構造体のコピーは、`edgeGeomGet[Spu,Rsx]VertexFormat()` を使って取得することができます。SPU 入力およびブレンド形状差分ストリームは SPU 頂点フォーマットを使用しますが、SPU 出力および RSX®のみのストリームは RSX®頂点フォーマットを使用します。

組み込みフォーマットが提供するものより幅広い柔軟性が必要な場合には、ユーザ独自のカスタム頂点フォーマット構造体を一から構築することもできます。カスタムフォーマットの記述は、`EdgeGeomSegment` オブジェクトに含まれています。`Edge SPU ランタイム`は、この記述を自動的にデコードして、頂点ストリームをその場で圧縮/解凍します。カスタムフォーマット処理は高度に最適化されていますが、それでも、可能な限り、組み込み頂点フォーマットを使ったほうが効率的です。

`EdgeGeomSegmentFormat` 構造体には、頂点フォーマット情報に加えて、インデックス、スキニング、カリングフレーバー用、およびスキニング行列フォーマット用のフィールドも含まれています。インデックスフレーバーは、各セグメント用の三角形リストを、圧縮しないフォーマット（1 インデックス当たり 16 ビット）で格納するか、圧縮率の高いカスタムフォーマットで格納するかを指定するために使われます。スキニングフレーバーは、シーンをスキニングするかどうか、および（スキニングを行うとすれば）どの種類のスキニングを行うかを指定するために使われます（通常は、シーンに適した結果を得られるスキニングのうち、もっとも制限の厳しい種類のスキニングを選んでください）。カリングフレーバーは、`Edgeジョブ`がどの種類の各三角形カリングを実行するかを指定するために使われます。スキニング行列フォーマットは、入力スキニング行列のフォーマットを記述します。

シーンをセグメントに分割する

`EdgeGeomScene` オブジェクトの作成、および `EdgeGeomSegmentFormat` オブジェクトへの書き込みが済んだら、処理を開始することができます。`libedgegeomtool` では、作業を簡単にするために、シーンを `EdgeGeomSegment` の配列に変換するために必要なあらゆるエンドツーエンドの処理を実行する `edgeGeomPartitionSceneIntoSegments()` というヘルパー関数を提供しています。この関数は、以下のような処理を行います。

- シーンを「バッチ」に分割します。バッチというのは、同じマテリアル ID を共有する（したがって、一回の RSX®描画呼び出しですべてレンダリングできる）三角形の集まりです。
- 各バッチに対して `Edge パーティショナー` を呼び出して、バッチを SPU サイズの作業負荷の集合に分割します。
- 各シーンのデータを、先に指定されたセグメントフォーマットにしたがって、SPU によって処理可能なフォーマットに変換します。この処理には、各作業負荷の三角形リストに対して `Edge kcache` オプティマイザを実行することや、あらゆるランタイム構造体を最終的なビッグエンディアン形式で作成することなどが含まれます。
- 各作業負荷のデータを、（セグメントを適切にグループ化できるように）`EdgeGeomSegment` オブジェクト、および適切なマテリアル ID と関連付けます。

この関数の呼び出しが終われば、`EdgeGeomScene` オブジェクト、および `EdgeGeomSegmentFormat` オブジェクトは不要になるので、削除することができます。

サンプルアプリケーション- `geomtoolsample`

（`host-common/src` ディレクトリ内の）`geomtoolsample` は、柔軟性のない単純なツールですが、`EdgeGeom` ランタイム用のアセットをビルドするために必要な操作手順を示しています。このサンプルは、実際の生産用ツールよりはるかに単純です。このツールでは、実行時に、特定の入力ファイルによって、単一の描画オブジェクトが表されると仮定します。また、実際の生産用ツールではデータ駆動になるであ

ろう頂点ストリームフォーマットその他の設定もハードコードされています。けれども、このサンプルは出発点としては優れており、おそらく、libedgegeomtool を基礎にした自分で作成する最初のツールとなることでしょう。

実行すると、geomtoolsample は、典型的なスタジオのシーンフォーマットを表す人工的なフォーマットで、単一の入力ファイル (elephant.fake) をロードします。シーンは、EdgeGeomScene に変換されて、上で説明した処理されて、さまざまな Edge サンプルにインクルード可能な C ヘッダファイル (elephant.edge.h) として書き出されます。

SPUおよびSPURSの初期化

最初の初期化手順は、SPU と SPURS を初期化することです。

SPU と SPURS を初期化するための擬似コード

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

SPURSイベントフラグの初期化

次の初期化手順は、ジオメトリ処理ジョブがいつ完了したかがわかるように、SPURS イベントフラグを作成することです。

SPURS イベントフラグを作成するための擬似コード

```
cellSpurseEventFlagInitializeIWL (&mSpurs, &spursEventFlag,
    CELL_SPURS_EVENT_FLAG_CLEAR_AUTO,
    CELL_SPURS_EVENT_FLAG_SPU2PPU);
cellSpurseEventFlagAttachLv2EventQueue (&spursEventFlag);

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

出力バッファの初期化

最初の手順は、処理された頂点を格納するメモリを用意することです。これは、通常、1 メガバイトにアラインメントされたメモリチャンクを割り当ててから、RSX®にマップすることによって実現されます。

出力バッファの割り当てとマッピングを行うための擬似コード

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

注：ハイブリッドバッファリングのような一部の方式では、複数の出力バッファを割り当てる必要があります。

次の手順は、出力バッファの種類を選択することです。以下の種類の出力がサポートされています。

- ダブルバッファ付き直接出力

- シングルバッファ付き直接出力
- 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

各バッファタイプについての詳しい情報は、「PlayStation®Edge ライブラリ概要」やリファレンスドキュメントに記載されています（「このドキュメントについて」の「関連ドキュメント」セクションを参照）。このドキュメントでは、ハイブリッドバッファリングの例について説明します。

ハイブリッドバッファを実装するための擬似コード

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

Edgeジオメトリジョブアプリケーションの作成

標準の SPURS ジョブとして実行される SPU プログラムを書いてください。

ジョブプログラムのための擬似コード

```
void cellSpursJobMain2(CellSpursJobContext2* stInfo, CellSpursJob256 *job)
{
    struct __attribute__((aligned(16)))
    {
        EdgeGeomVertexStreamDescription *outputStreamDesc; // 0
        void *indexes; // 1
        void *pad1;
        void *skinMatrices; // 3
        void *pad2;
        void *skinIndexesAndWeights; // 5
        void *pad3;
        void *vertexesA; // 7;
        void *pad4;
        void *pad5;
        void *vertexesB; // 10
        void *pad6;
        void *pad7;
        EdgeGeomViewportInfo *viewportInfo; // 13
    }
```

```

    EdgeGeomLocalToWorldMatrix *localToWorld; // 14
    EdgeGeomSpuConfigInfo *spuConfigInfo; // 15
    void *fixedOffsetsA; // 16
    void *fixedOffsetsB; // 17
    EdgeGeomVertexStreamDescription *inputStreamDescA; // 18
    EdgeGeomVertexStreamDescription *inputStreamDescB; // 19
} inputs;
cellSpursJobGetPointerList((void*)&inputs, &job->header, stInfo);

// userData 領域から追加の値を抽出する。
uint32_t totalMatrixCount = ((job->workArea.userData[3] >> 32)
    + (job->workArea.userData[4] >> 32)) / 48;
uint32_t outputBufferInfoEa = job->workArea.userData[20];
uint32_t holeEa = job->workArea.userData[21];
uint32_t numBlendShapes = job->workArea.userData[22] >> 32;
uint32_t blendShapeInfosEa = job->workArea.userData[23]
    & 0xFFFFFFFF;

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

EdgeGeomSpuContext ctx;

edgeGeomInitialize(&ctx, inputs.spuConfigInfo, stInfo->sBuffer,
    job->header.sizeScratch << 4, stInfo->ioBuffer,
    job->header.sizeInOrInOut,
    stInfo->dmaTag, inputs.viewportInfo, inputs.localToWorld,
    &customFormatInfo);
edgeGeomDecompressVertexes(&ctx, inputs.vertexesA,
    inputs.fixedOffsetsA, inputs.vertexesB, inputs.fixedOffsetsB);
edgeGeomProcessBlendShapes(&ctx, numBlendShapes,
    blendShapeInfosEa);
edgeGeomSkinVertexes(&ctx, inputs.skinMatrices,
    totalMatrixCount, inputs.skinIndexesAndWeights);
edgeGeomDecompressIndexes(&ctx, inputs.indexes);
uint32_t numVisibleIdxs = edgeGeomCullTriangles(
    &ctx, EDGE_GEOM_CULL_BACKFACES_AND_FRUSTUM);
if (numVisibleIdxs == 0)
{
    CellGcmContextData gcmCtx;
    edgeGeomBeginCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
    edgeGeomEndCommandBufferHole(&ctx, &gcmCtx, holeEa, 0, 0);
    return;
}

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

```

```

    }

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

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

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

```

頂点を処理するためのSPURSジョブを作成する

今度は、入出力フレイバー、入出力フレイバーの頂点ストライド、入力フレイバー中の属性の数を求める必要があります。この情報を使った処理は、以下の擬似コードで説明するのが最もわかりやすいでしょう。

頂点ストリームを処理するジョブのセットを作成するための擬似コード

```

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

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

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

    // スキニング行列
    uint64_t matricesSizeA = info->skinMatricesSizes[0];
    uint32_t matricesA = matricesEa + info->skinMatricesByteOffsets[0];
    uint64_t matricesSizeB = info->skinMatricesSizes[1];
    uint32_t matricesB = matricesEa + info->skinMatricesByteOffsets[1];
    job->workArea.dmaList[3] = matricesA | (matricesSizeA << 32);
    job->workArea.dmaList[4] = matricesB | (matricesSizeB << 32);

    // スキニング行列、および重み
    uint64_t iAndWSizeA = info->skinIndexesAndWeightsSizes[0];
    uint32_t iAndWA = (uint32_t)info->skinIndexesAndWeights;
    uint64_t iAndWSizeB = info->skinIndexesAndWeightsSizes[1];
    uint32_t iAndWB = iAndWA + iAndWSizeA;
    job->workArea.dmaList[5] = iAndWA | (iAndWSizeA << 32);
}

```



```

job->workArea.dmaList[6] = iAndWB | (iAndWSizeB << 32);

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

// 三角形カリングデータ
job->workArea.dmaList[13] = (uint32_t)&viewportInfo |
    ((uint64_t)sizeof(EdgeGeomViewportInfo) << 32);
job->workArea.dmaList[14] = (uint32_t)&localToWorldMatrix |
    ((uint64_t)sizeof(EdgeGeomLocalToWorldMatrix) << 32);

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

// ソフトウェア固定小数点形式の整数オフセット
uint32_t fixedOffsets1 = (uint32_t)info->fixedOffsets[0];
uint64_t fixedOffsets1Size = (uint64_t)info->fixedOffsetsSize[0];
job->workArea.dmaList[16] = fixedOffsets1 | (fixedOffsets1Size << 32);
uint32_t fixedOffsets2 = (uint32_t)info->fixedOffsets[1];
uint64_t fixedOffsets2Size = (uint64_t)info->fixedOffsetsSize[1];
job->workArea.dmaList[17] = fixedOffsets2 | (fixedOffsets2Size << 32);

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

// --- Dma データ終了 / ユーザデータ開始---

// 出力バッファ情報
job->workArea.userData[20] = (uint32_t)&outputBufferInfo;

// コマンドバッファホール
uint32_t holeSize = info->spuConfigInfo.commandBufferHoleSize << 4;
if(ctx->current + holeSize/4 + 3 > ctx->end)
{
    if((*ctx->callback)(ctx, holeSize/4 + 3) != CELL_OK)
        return;
}

```

```

while(((uint32_t)ctx->current & 0xF) != 0)
    *ctx->current++ = 0;
uint32_t holeEa = (uint32_t)ctx->current;
uint32_t holeEnd = holeEa + holeSize;
uint32_t jumpOffset;
cellGcmAddressToOffset(ctx->current, &jumpOffset);
cellGcmSetJumpCommandUnsafeInline(ctx, jumpOffset);
uint32_t nextJ2S = ((uint32_t)ctx->current + 0x80) & ~0x7F;
while(nextJ2S < holeEnd)
{
    ctx->current = (uint32_t*)nextJ2S;
    cellGcmAddressToOffset(ctx->current, &jumpOffset);
    cellGcmSetJumpCommandUnsafeInline(ctx, jumpOffset);
    nextJ2S = ((uint32_t)ctx->current + 0x80) & ~0x7F;
}
ctx->current = (uint32_t*)holeEnd;
job->workArea.userData[21] = holeEa;

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

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

```

一般的なゲームエンジンでは、cellGcmSetDrawIndexArray() コマンドの代わりに、この関数が呼び出されます。

SPURS ジョブリストの作成

先の手順で生成したジョブを実行する前に、まず SPURS ジョブリストを設定する必要があります。

SPURS ジョブリストを作成するための擬似コード

```

static CellSpursJobList jobList;
jobList.eaJobList = (uint64_t)jobs;

```

```
jobList.numJobs = jobs - firstJob;  
jobList.sizeOfJob = 256;
```

SPURSコマンドリストの作成

次は、新たに作られたジョブリストを、SPURS ジョブコマンドリストから呼び出す必要があります。

SPURS ジョブコマンドリストを作成するための擬似コード

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

SPURSコマンドリストの実行

コマンドリストの作成が済んだら、それを実行する必要があります。

SPURS ジョブコマンドリストを実行するための擬似コード

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

SPURSジョブの完了を待つ

最後に、ジョブによって出力されたデータを RSX®が利用できるように、SPURS ジョブの完了を待つ必要があります。

SPURS ジョブコマンドリストを作成するための擬似コード

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

Filename: Edge_Geometry_Library-Quick_Start_j.doc
Directory: C:\Documents and Settings\MTESHIMA\My Documents\Working
Files\7-19-10\Edge 1.2.0
Template: C:\sony\yoshi\templates\overview_V1.1_j.dot
Title: Edge_Geometry_Library-Quick_Start_j
Subject:
Author: SCE Document Group
Keywords:
Comments:
Creation Date: 7/19/2010 2:36:00 PM
Change Number: 3
Last Saved On: 7/19/2010 2:36:00 PM
Last Saved By: mteshima
Total Editing Time: 3 Minutes
Last Printed On: 7/19/2010 2:38:00 PM
As of Last Complete Printing
Number of Pages: 19
Number of Words: 8,623 (approx.)
Number of Characters: 22,250 (approx.)