CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

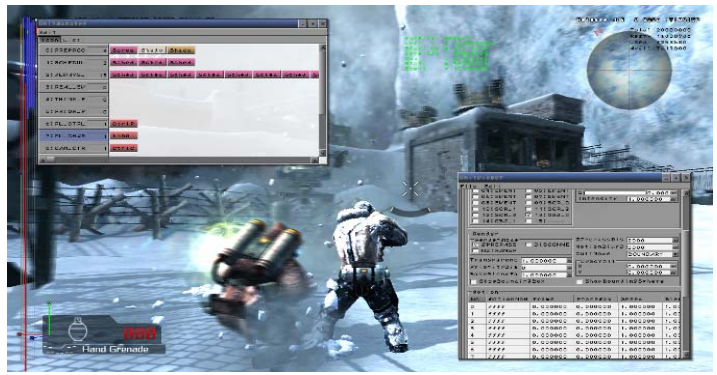
次世代機に向けた ゲームエンジンの設計

株式会社カプコン
第二制作部ソフトウェア制作室
石田智史

CAPCOM

フレームワーク

- 自社で開発しているゲームエンジン
- 主要な次世代機向けのタイトルが採用
  - (Dead Rising ,Lost Planet , BH5 , DMC4 ,MH3…)
- マルチプラットフォームに対応
  - (XBOX360, PC(VISTA) , PS3)
- マルチコアに最適化された設計
- ツールベースアーキテクチャ



アジェンダ

- マルチスレッドの活用
- レンダリングテクニック
- ツールアーキテクチャ

マルチスレッドの活用

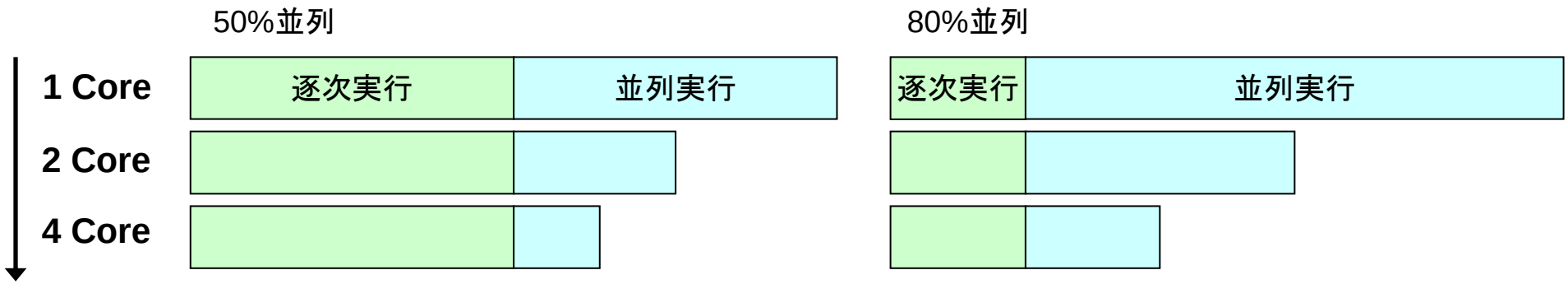
次世代機のCPU

- CPUのマルチコア化
  - CPU周波数の向上は頭打ち
    TDPの問題
  - CPUのトレンドが変化
    周波数の向上から、コア数の増加へ
- CPUのコア数
  - XBOX360
    3Core x 2 SMT
  - PS3
    1Core x 2 SMT + SPU x 7
  - PC
    1Core ~ Many-Core

→ゲームプログラムのマルチコア/スレッド対応は必須

アムダールの法則

- 並列化できない部分に、全体の実行時間が制約される
  - 例えば50%並列化できない部分があったとすると、どんなに並列度が向上しても2倍以上高速化できない



→並列実行できる割合を増やすことが重要

何を並列化するか

- モジュール単位の並列化
  - レンダリング、サウンド、コリジョン、モーション、物理シミュレーション、AIなどをモジュール化し、並列に処理
- 利点
  - 依存関係を最小化できる場合、高い並列度が得られる
- 欠点
  - 一部を除きモジュールの分離は困難

    ゲームプログラムは、各モジュールが密接に依存しあって構成されているため
  - SMTを利用する場合、キャッシュ効率が悪化

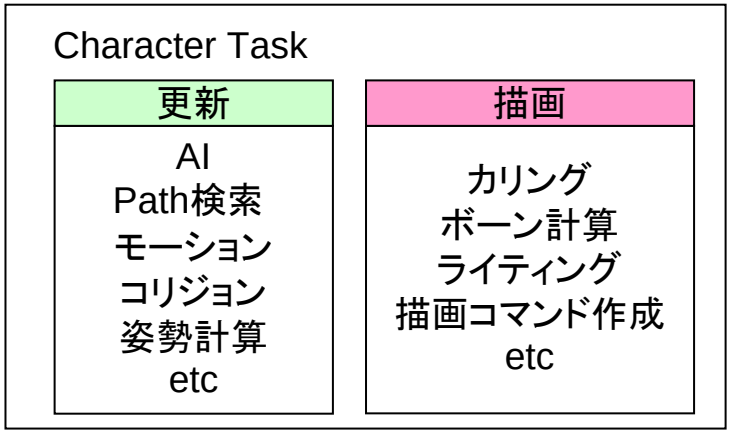
    まったく異なるコードが動作するため

何を並列化するか

- ループ単位の並列化
  - 負荷の高い依存関係のないループを分散して並列に処理
- 利点
  - SMTとの相性が非常に良い
    同じコードが動作するため
- 欠点
  - ゲームプログラムの数%を占める依存関係のないループは多くない
  - 処理量の少ないループでは、同期のオーバーヘッドが上回る

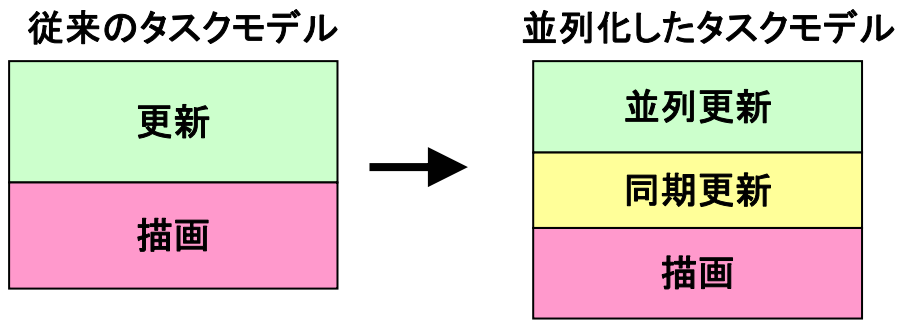
何を並列化するか

- タスク単位の並列化
  - ゲームプログラムにおけるタスク
    毎フレーム更新と描画を行う単位
    プレイヤー、敵、弾、カメラ、効果ジェネレーターetc
  - 依存関係のないタスクを並列に実行
- 利点
  - 依存関係は限定できる
    他のタスクの情報が必要な場合は限られている
  - 同じ内容のタスクを実行する場合、SMTの効率は高くなる
- 欠点
  - 依存関係のないタスクがない場合は並列に実行できない



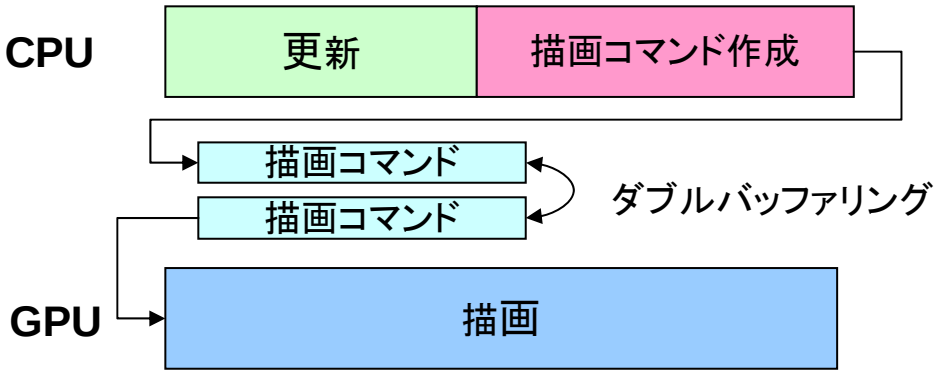
並列化のアプローチ

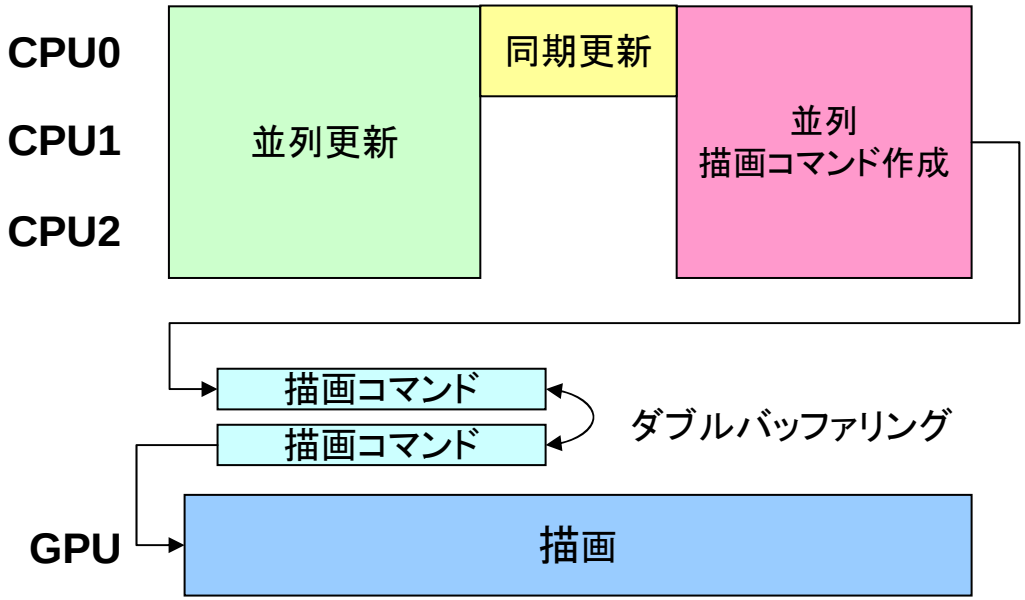
- モジュール単位の並列化
  - レンダリング、サウンド、リソースロード
- ループ単位の並列化
  - 一部のソート処理など
- タスク単位の並列化
  - 全てのゲームオブジェクト
    (プレイヤー、敵、エフェクト、ライト、カメラetc.)
  - 毎フレームのタスクの更新を、**並列更新**と**同期更新**に分け、並列更新時のタスク間の依存関係を減らす



ゲームプログラムの流れ

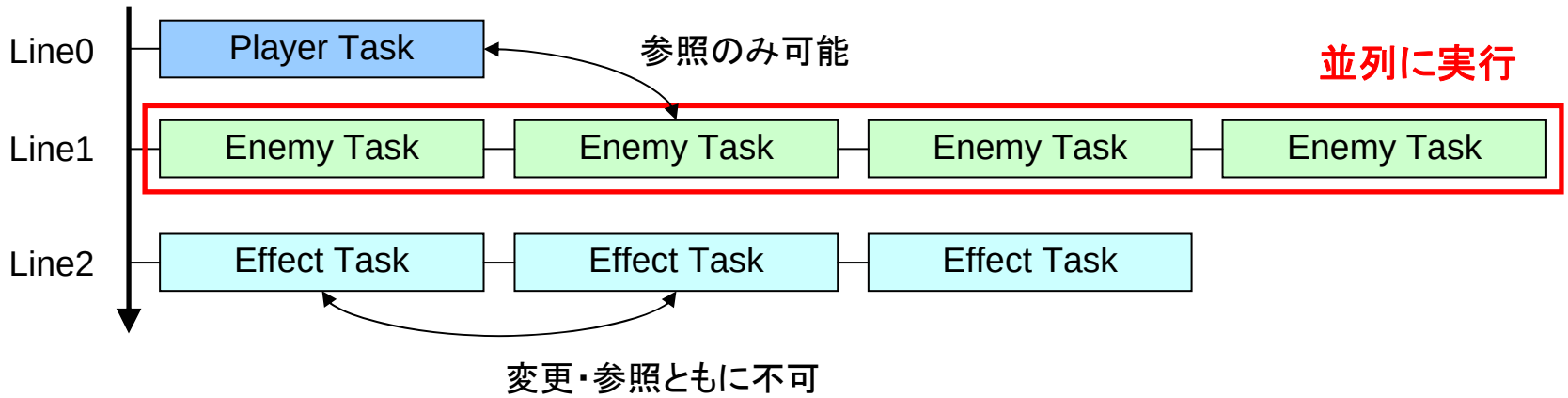
- 従来のゲームプログラムの流れ
  - CPU とGPUが並列に動作
- 並列化されたゲームプログラムの流れ
  - 複数CPUとGPUが並列に動作





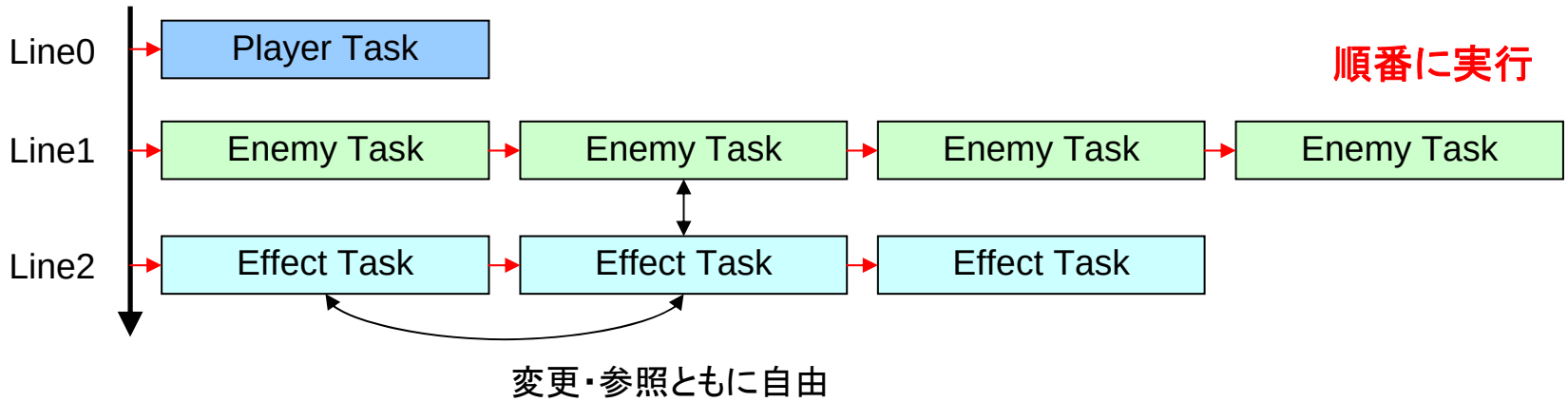
並列更新

- タスクの依存関係をラインで管理
  - 依存関係のないタスクは同じラインに登録
  - 依存関係のあるタスクは別のラインに登録
- マルチスレッドでライン単位に並列更新
  - 同一ラインのタスクは変更・参照ともに行うことはできない
  - 異なるラインのタスクは参照のみ可能



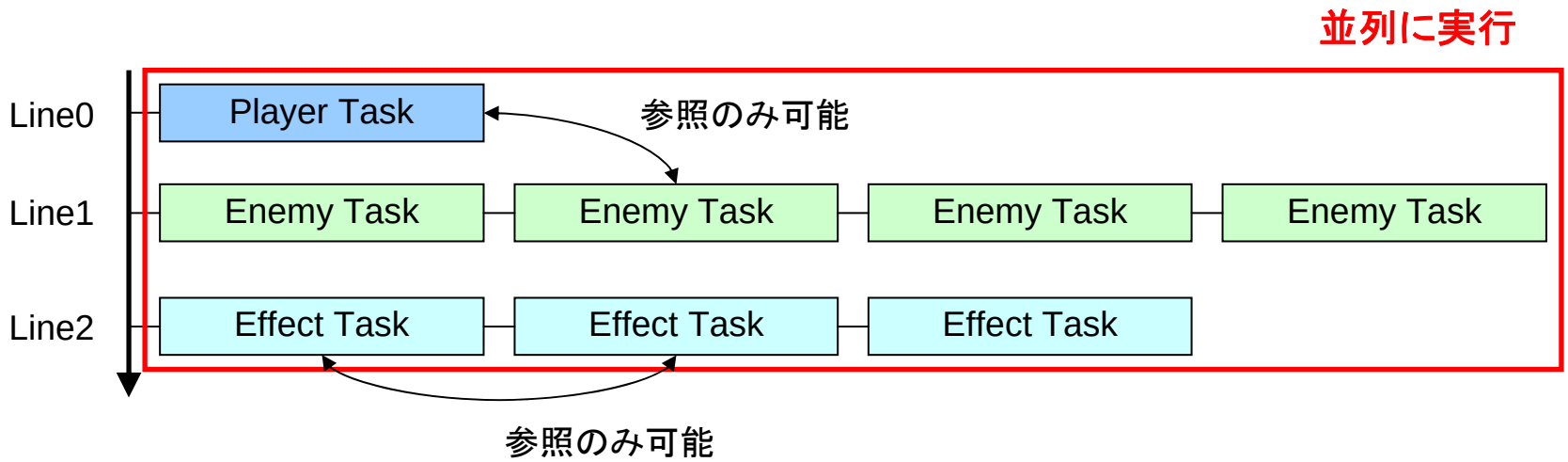
同期更新

- シングルスレッドでタスクを逐次更新
  - 他のタスクへの参照や変更を自由に行える
- 最短時間で更新
  - キャッシュとメモリ帯域を一つのスレッドに占有させる
  - 同期更新時は不要な同期オブジェクトを無効化する
- 処理は最小限にする
  - 可能な場合、値の取得だけを行い、計算は次の並列更新時に行う



並列描画コマンド作成

- マルチスレッドで全てのタスクを並列に実行
  - 自分自身の状態は参照のみで**変更は不可**
  - 他のタスクは参照のみ可能
  - スレッドごとに独立したコマンドバッファへの登録のみ可能



並列描画コマンド作成の問題

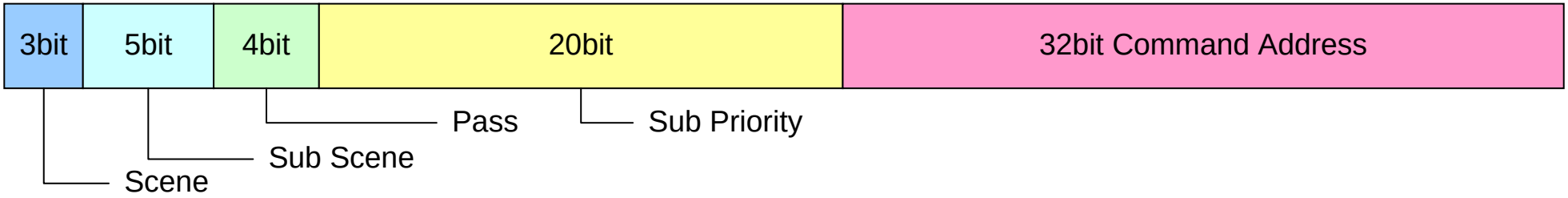
- 描画コマンドの登録順が安定しない
  - タスク内での登録順は安定
  - タスク間での登録順は不定

- 登録順序に依存せず、実行順序を安定させる
  - 描画コマンドにプライオリティを持たせる
  - 最後にプライオリティでソートする

    順序を保持するソートアルゴリズムを用いる
    →マージソート

    同一プライオリティの場合、タスク内の登録順序に依存する
    →安定

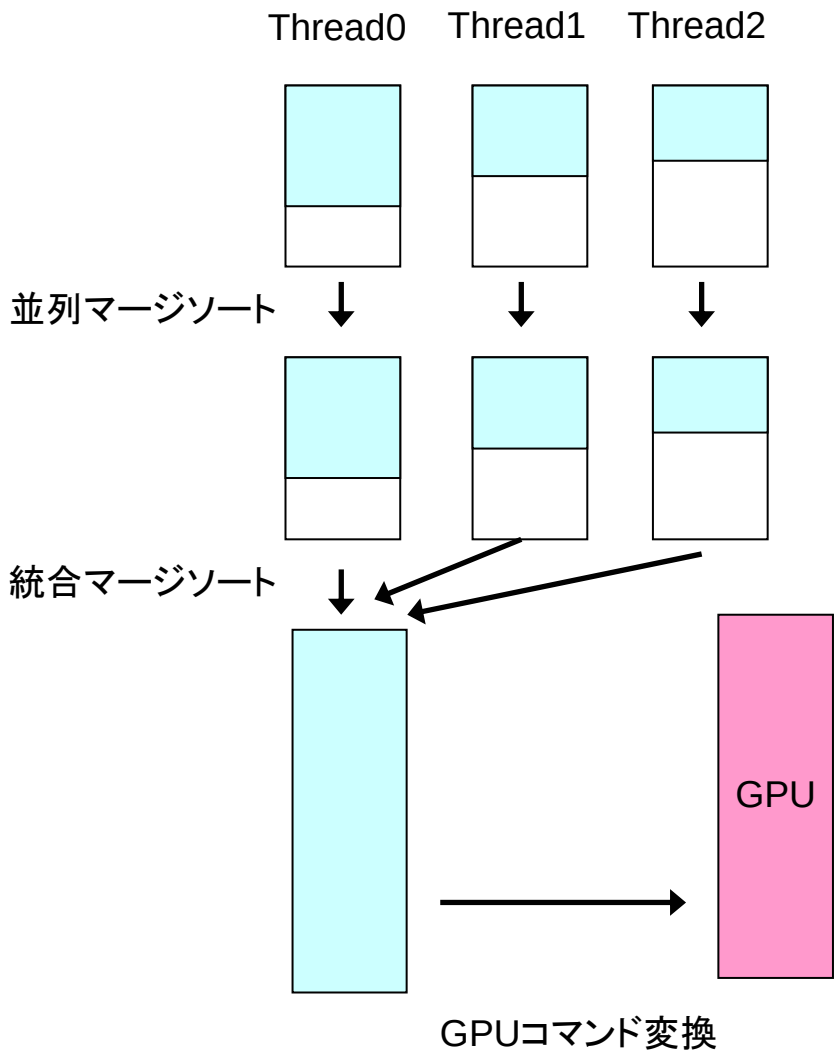
中間描画コマンド



- 描画コマンドを64bitのタグの配列として記述
  - 32bitのプライオリティと、32bitのコマンドへのアドレスから構成
- プライオリティの内訳
  - Scene
    完全に独立したシーンを定義（画面分割など）
  - Sub Scene
    シーン描画時に必要な別のシーン（シャドウマップや、リフレクションマップなど）を定義
  - Pass
    レンダリングパス(Zプレパス、不透明パス、半透明パス、フィルタパス)などを定義
  - Sub Priority
    パス内での優先度。
    例えば不透明パスではマテリアルを優先度とし、マテリアル切り替えが最小になるように制御。
    半透明パスでは視点からの距離を優先度とし、奥から手前へ描画されるように制御。

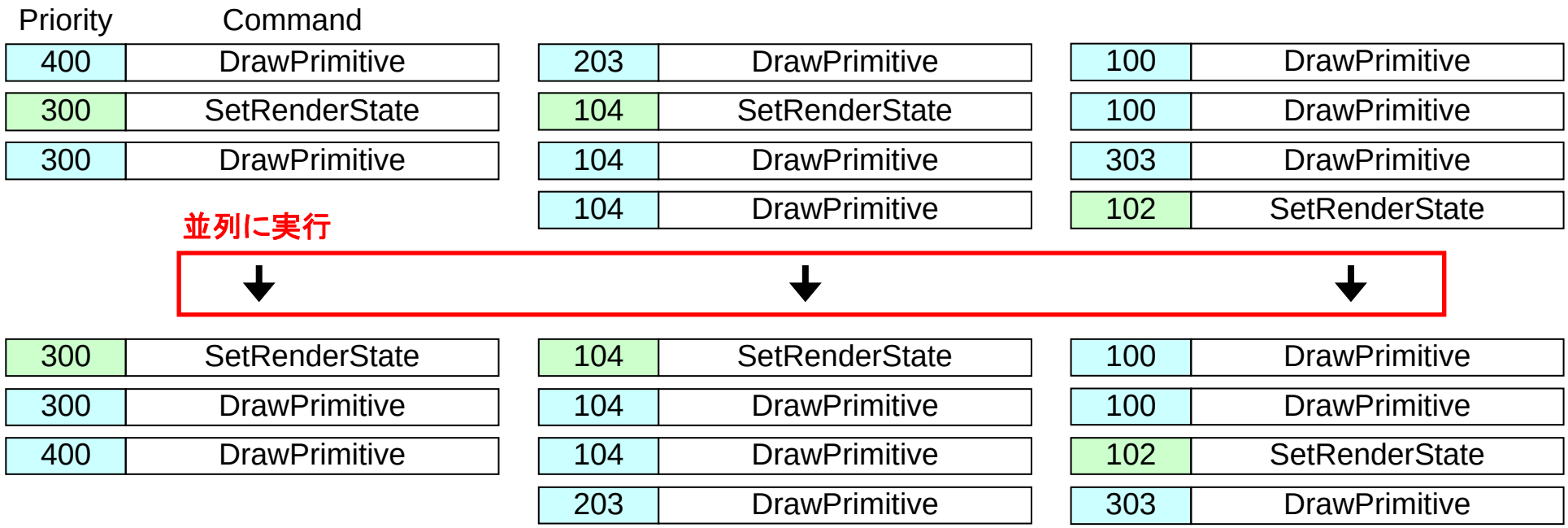
描画コマンド作成の流れ

1. スレッドごとに独立したコマンドバッファに、中間描画コマンドを複数スレッドで並列に作成
2. コマンドバッファ単位にコマンドのプライオリティで並列マージソート
3. 複数のコマンドバッファを一つのコマンドバッファにプライオリティで統合マージソート
4. 一本化された中間描画コマンドをネイティブの描画コマンドに変換し実行



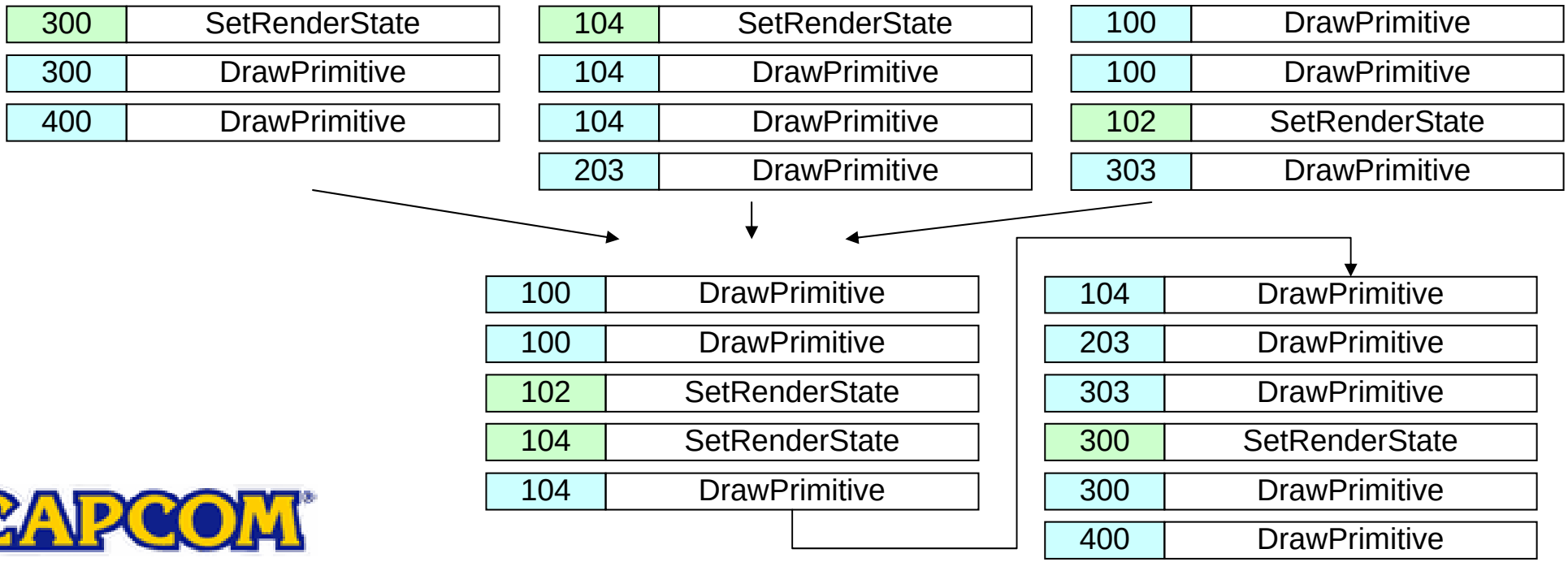
並列マージソート

- コマンドバッファ単位にマージソート
  - スレッドごとに独立したコマンドバッファ単位でプライオリティ順に並列でマージソート
  - 同一プライオリティでの順序は保障される



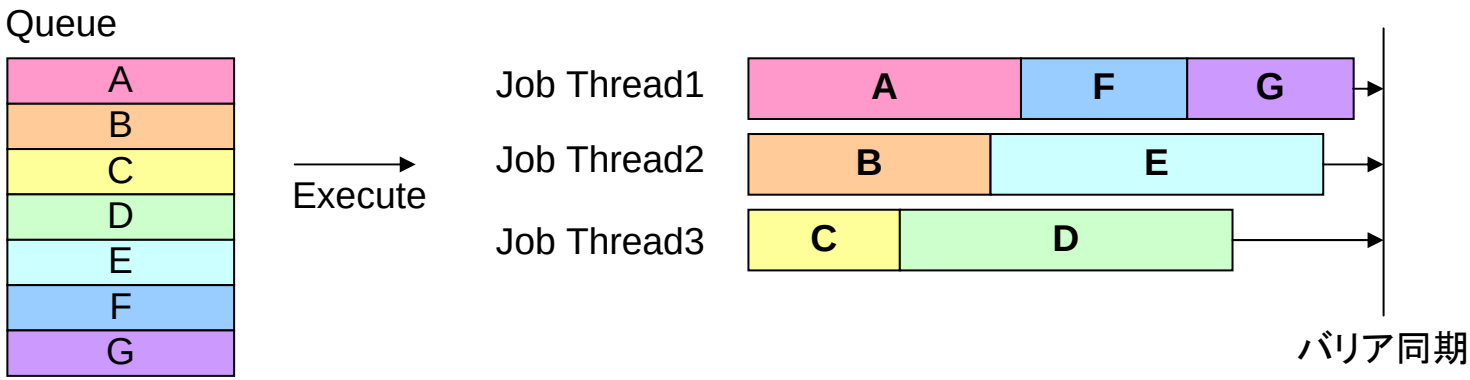
統合マージソート

- コマンドバッファを一本化
  - スレッドごとに独立したソート済みのコマンドバッファをプライオリティ順に一つのコマンドバッファにまとめる
  - 通常マージソートは、2要素のマージだが、任意要素をマージできるようにアルゴリズムを拡張



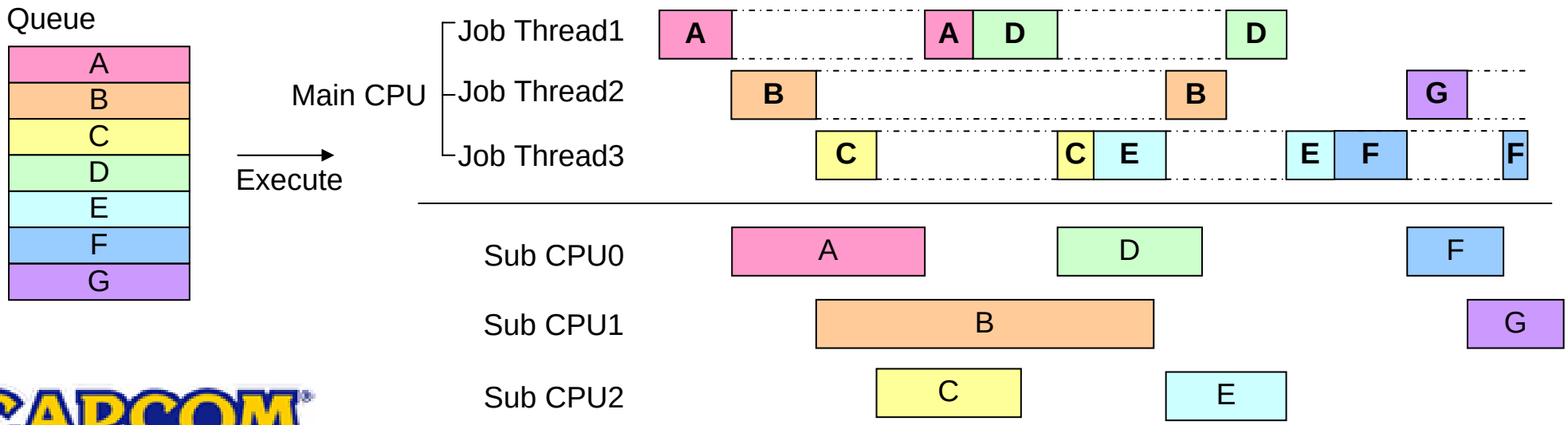
対称コアにおける並列処理

- ジョブ・キュー
  1. 並列に実行したい関数のポインタをキューに積む
  2. 複数のジョブスレッドで並列に関数を実行する
  3. 全ての関数の実行が終了するまで待つ。



非対称コアにおける並列処理

- ジョブ・キュー
  - ジョブスレッドは、ソフトウェアスレッドとして実装
- ジョブ関数内
  - 一部の処理をコプロセッサに投げる
  - 処理が終わるまで別のソフトウェアスレッドにスイッチ
- コプロセッサ内
  - DMAでデータを取得
  - データを処理
  - DMAでデータをストア



実例：デッドライジング

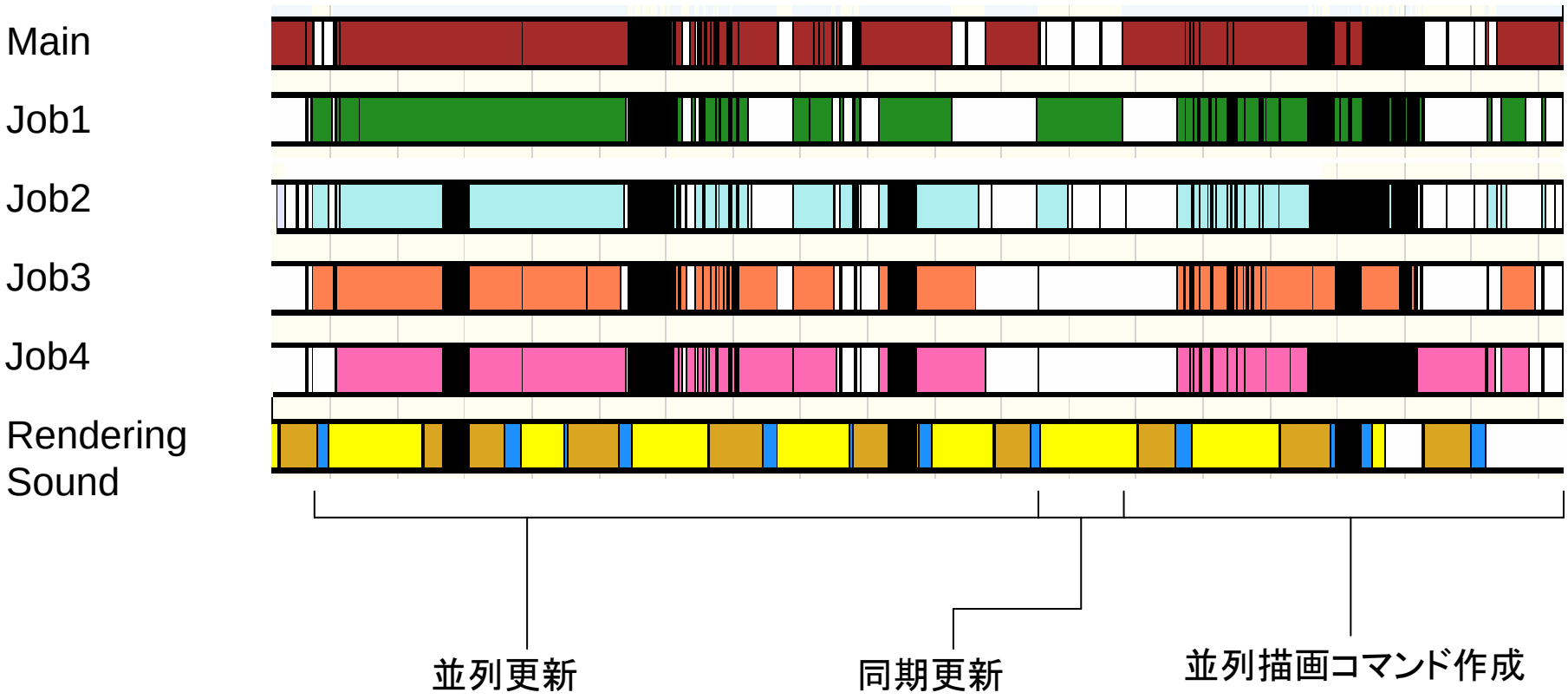
- テストケース
  - 2,840,000 Vertex
  - 5148 Batch
  - 3200 Particle
  - 300 Zombies
  - 303 Item
  - 98 Generator

CAPCOM

実例：デッドライジング

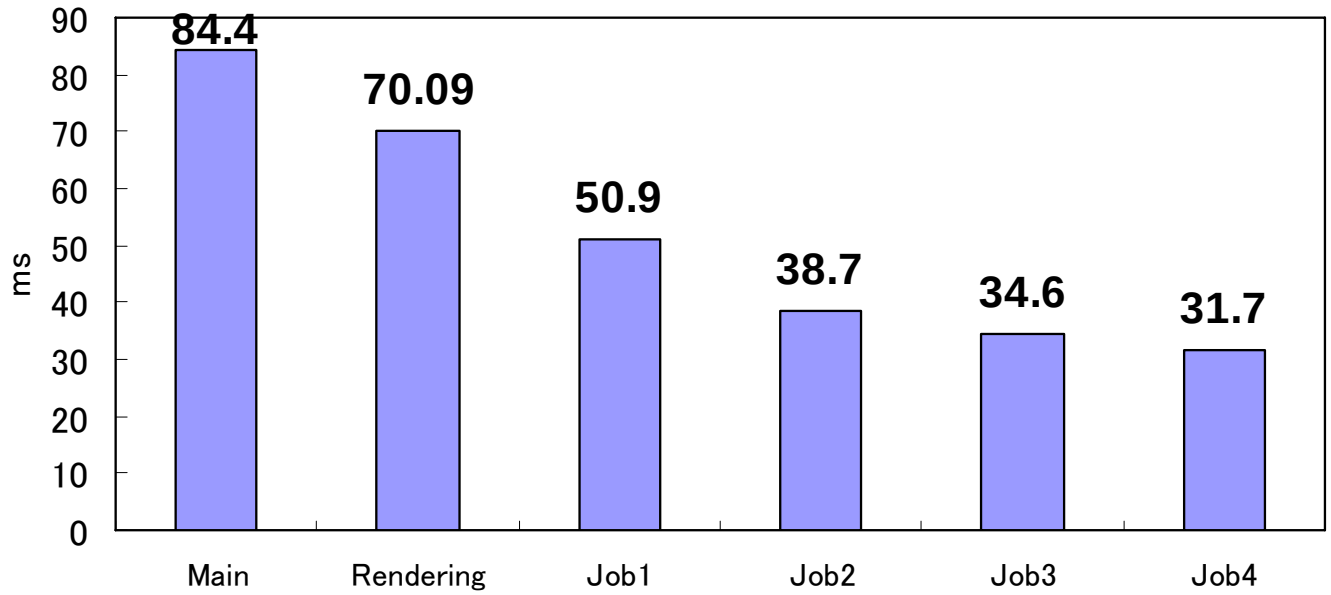
- スレッドの利用率

空白の領域がアイドルタイム



実例：デッドライジング

- スレッド数と処理速度
  - 6スレッドを利用した場合、1スレッドしか利用しなかった場合に比べて、約**2.6**倍の速度向上



実例:ロストプラネット

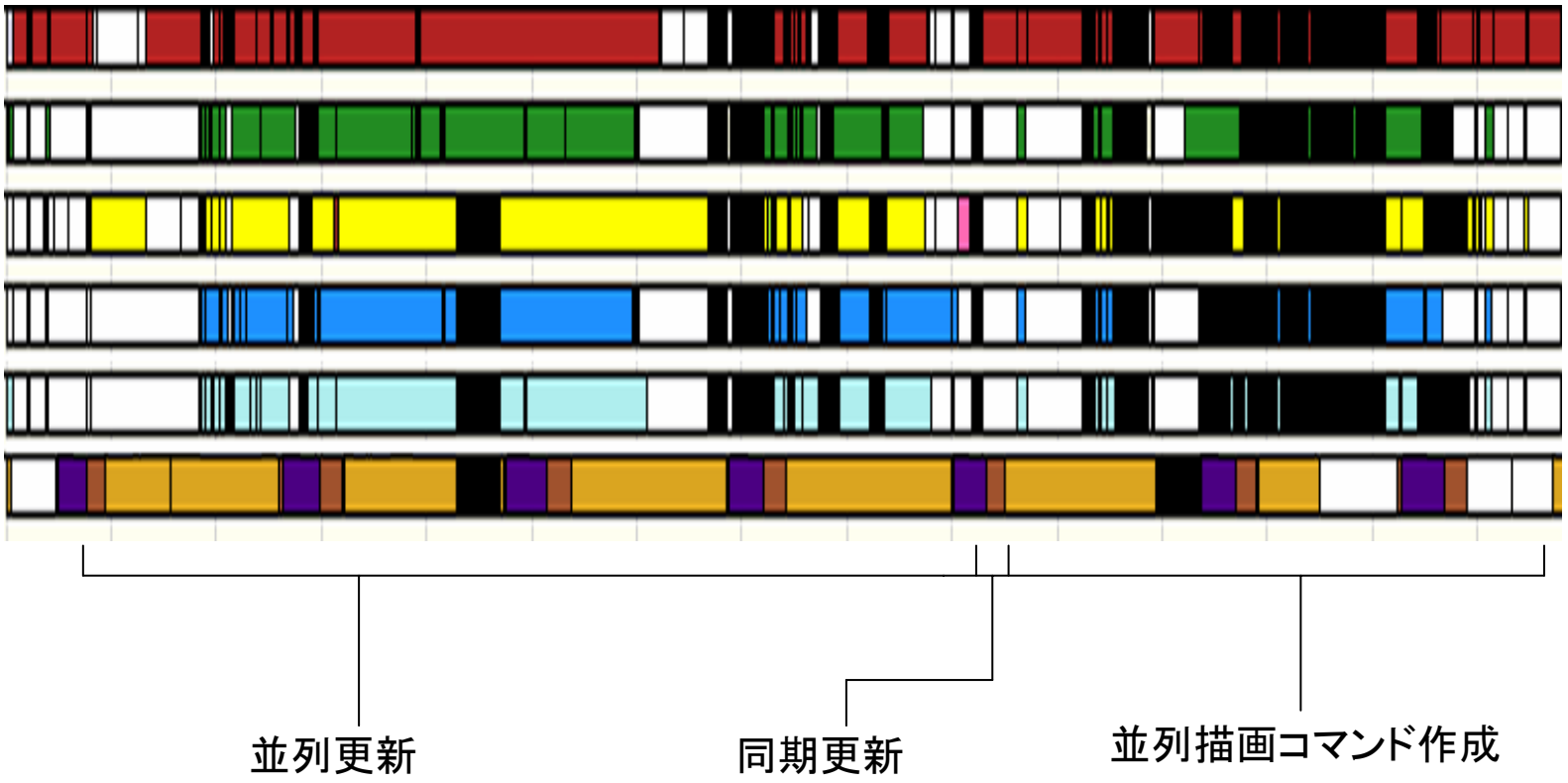
- テストケース
  - 810,000 Vertex
  - 892 Batch
  - 12000 Particle
  - 26 Character
  - 130 Object
  - 119 Generator

CAPCOM

実例：ロストプラネット

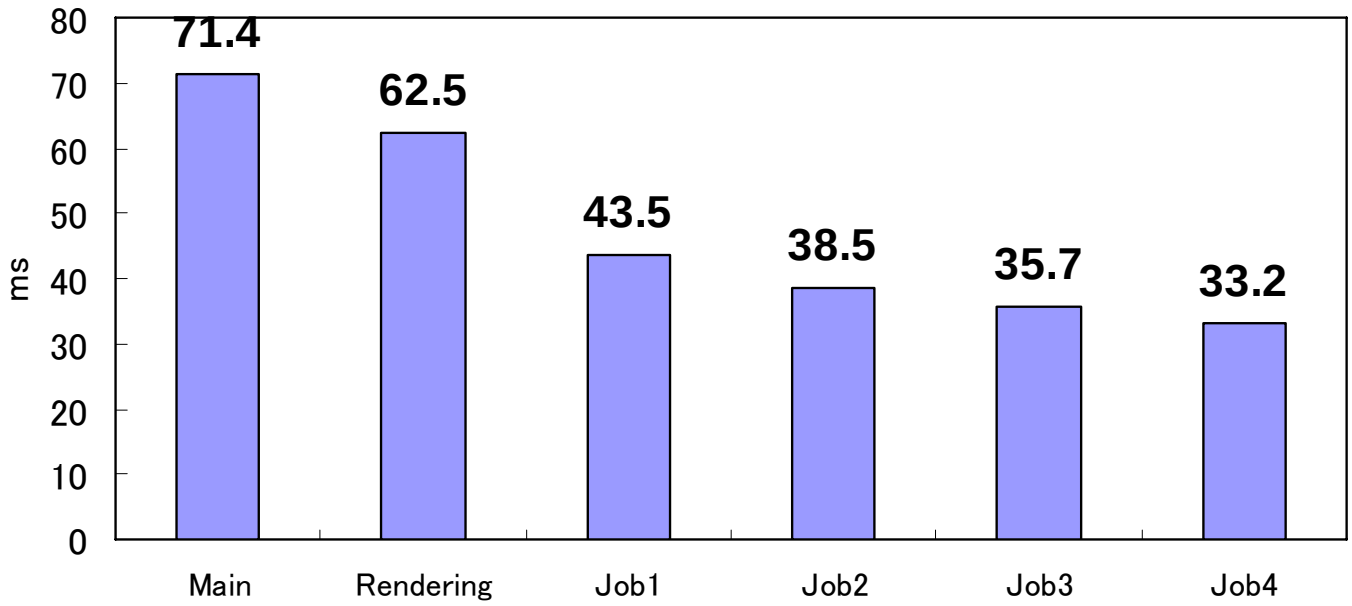
- スレッドの利用率

空白の領域がアイドルタイム



実例：ロストプラネット

- スレッド数と処理速度
  - 6スレッドを利用した場合、1スレッドしか利用しなかった場合に比べて、約**2.15**倍の速度向上



並列化による問題

- 並列化によるバグ
  - 変数への同時書き込み
  - デッドロック

  →再現性が低く、特定が困難

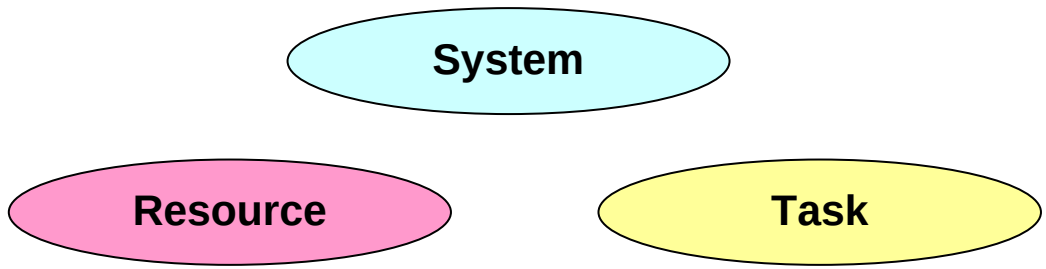
- 並列化によるパフォーマンスの低下
  - 不必要に同期オブジェクトを多用する

→システム設計が重要

設計をシンプルにする

- ゲームプログラム内のクラスを3つに分類
  - System
    唯一のインスタンスを持つシングルトンオブジェクト
    　　パッドなどデバイスへのアクセスクラスや、マネージャークラスが相当
  - Resource
    インスタンスが作成された時点から、一切状態が変化しないオブジェクト
    　　モデルデータ、モーションデータなどデータクラスが相当
  - Task
    毎フレーム状態の更新と描画コマンド作成を行うオブジェクト
    　　プレイヤー、敵、ライト、カメラ、エフェクトクラスなどが相当

→全てのゲーム中のクラスは3種類のクラスを継承する、もしくは3種類に内包されるクラスとしてデザイン



ルールを明確にする

- System
  - グローバル変数は用いずシステムのメンバ変数とし、取得・設定はシステムの関数を介して行う
    複数スレッドからのアクセスを同期オブジェクトで保護
- Resource
  - 作成されてから一切状態を変更してはならない
- Task
  - 並列更新時
    同一ライン上のタスクの変更参照ともに禁止
    異なるライン上のタスクは参照のみ可能
  - 同期更新時
    全てのタスクの変更・参照ともに可能
  - 並列描画コマンド作成時
    自分自身も含め、状態の変更は一切してはならない
    全てのタスクの参照のみ可能

同期を最小限にする

- System
  - 並列実行中でない場合は、同期オブジェクトを無効化する
    例えば、同期更新中など
- Resource
  - 同期は一切不要
    状態が変化しないことが保障されているため
- Task
  - 同期が必要な処理は、同期更新中に行う
    同期オブジェクトは一切使用しない

並列化によるバグ

- デッドライジングのケース
  - バグチェック期間に入ってからの、並列化が原因のバグ
    数件のみ

- ケース1：モデルの回転が安定しない
  - ✓ 同じライン上のタスクのパラメータを並列更新中に書き換えていた
    並列更新中は、同一ライン上のタスクの変更参照ともに禁止

- ケース2：メモリリーク
  - ✓ リソースの状態を動的に書き換えていた
    リソースは作成されてから一切状態を変更してはならない

→アプリケーション全体を並列化しても深刻なバグ地獄にはつながらない。

まとめ

- ゲームプログラムは並列化しやすい
  - タスクという観点から見た場合、依存関係は最小化できる
- 並列化することで大幅なパフォーマンスの向上が得られる
  - 実際のタイトルで約2~3倍程度の向上
- 並列化によるバグは深刻ではない
  - シンプルな設計と明確なルールが重要

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

レンダリングテクニック

CAPCOM

次世代機のレンダリングの問題

- フレームレート
- フィルレート
- テクスチャ品質と容量

フレームレートの問題

- 次世代機の絵作りに求められるもの
  - HD解像度(1280x720以上)、アンチエイリアシング
  - 複雑なマテリアル
  - 高品質な影
  - 何重ものポストフィルタ

- 60fps(16.6ms)で全てを満たすことは困難
  - 次世代機ではピクセルコスト が大幅に増加
  - ピクセルコストの削減は困難

→ビジュアルに大きく影響

モーションブラー

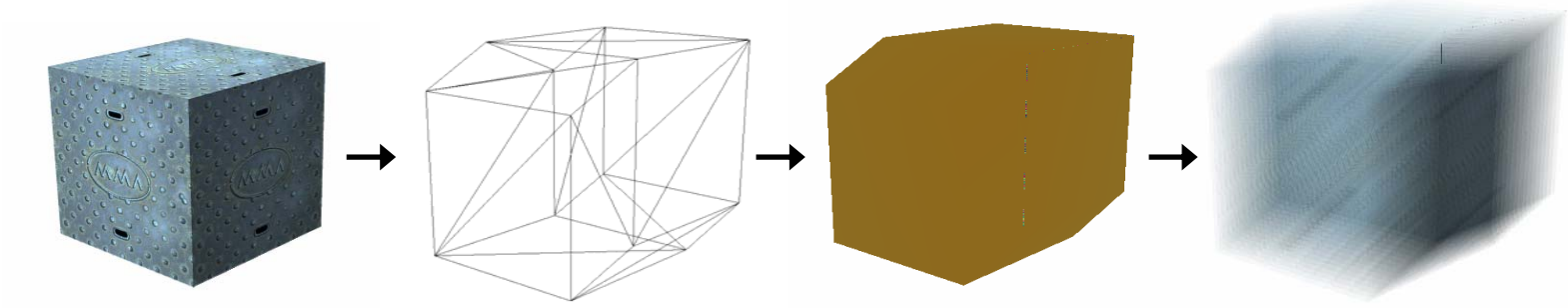
- 60fps ≒ 30fps + モーションブラー
  - うまく実装されたモーションブラーは60fps並みに滑らかに見える
  - +シネマティックな映像効果

→スーパーサンプリングは現実的ではない

2.5Dモーションブラー

- GDC2003
  - NVIDIA OpenGL Shader Tricks

- ジオメトリ処理とイメージベースの処理を組み合わせる
  1. 法線方向と速度ベクトル(ベロシティ)に基づいてモデルを引き伸ばし、ベロシティマップを描画
  2. ベロシティマップに基づいてシーンをブラー処理



Source Image　Stretch　Velocity　Final Image

2.5Dモーションブラーの問題

- ストレッチの破綻
  - 同じ頂点座標で法線方向が異なるモデルで発生

    共有頂点でない頂点を含むモデル。

    例えば箱のような形状をストレッチした場合

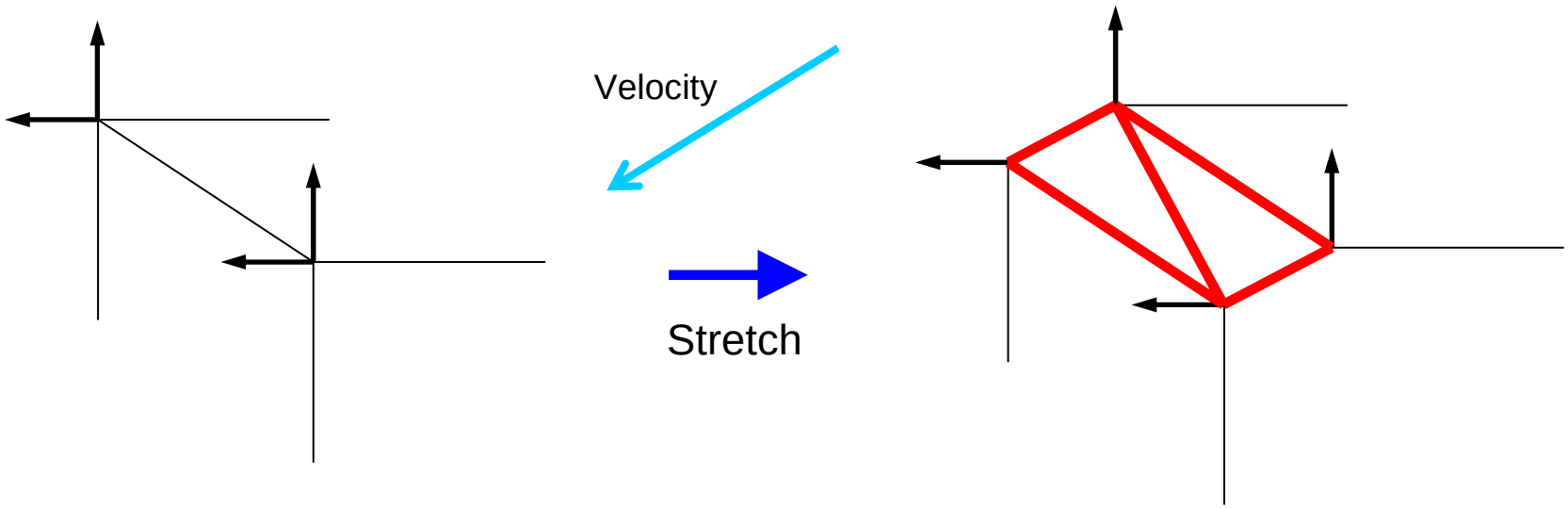
- モデル境界のアーティファクト
  - ベロシティが大きく異なる境界で発生

- ジオメトリコストの増加
  - ベロシティ描画のために追加でモデルを2回レンダリング

    ストレッチあり＋ストレッチなし

ストレッチの破綻回避

- 同じ辺を共有し、かつ法線方向の異なる辺に対してポリゴンを事前に生成
- ストレッチ時のみ追加で描画



モデル境界のアーティファクト軽減

- ブラー時のサンプリングに深度情報を考慮
  - ベロシティと共に深度情報を書き込む
    ベロシティテクスチャのB値に線形の深度情報
    シーンテクスチャのA値に線形の深度情報
  - 対象ピクセルの深度値より手前のピクセルはサンプリングしない

Scene Texture
ARGB8888

Velocity Texture
ARGB2101010

モデル境界のアーティファクト軽減

- 深度情報を考慮したサンプリング

```
float4 MotionBlur(INPUT_V I, uniform const int nsamples) : COLOR
{
        float3 v        = tex2D(VelocitySampler, I.uv);     //ベロシティマップからフェッチ
        float2 velocity  = v.xy - float2(0.5,0.5);          //ベロシティを取得
        float   depth    = v.z;                             //ピクセルの線形深度を取得
        float4 color = float4(0,0,0,0);             //アキュムレーター
        float  n = 0;                                       //サンプリング回数

        for (int i=0;i<nsamples;i++){
            //ベロシティ方向にUVをずらしてシーンテクスチャからフェッチ
            float4 c  = tex2D(SceneSampler, I.uv + velocity * ((float)i / ((float)nsamples-1.0)));
            if (c.a >= depth){
                        //奥のピクセルであればサンプリング
                        color += c;
                        n ++;
            }
        }
        color /= n;                                         //サンプリング回数で割る
        return color;
}
```

CAPCOM

モデル境界のアーティファクト軽減

通常のサンプリング

深度情報を考慮したサンプリング

CAPCOM

ジオメトリコストの削減

- 全てのモデルを三種類に分類
  - 静的モデル
    背景などワールド座標が変化しないモデル
  - 遠景モデル
    遠景のワールド座標が変化するモデル
    遠景のためワールド空間の移動量は相対的に少なくなる
  - 近景モデル
    近景のワールド座標が変化するモデル

ジオメトリコストの削減

- 近景モデル
  - ワールド空間の移動量の影響が大きい
    モデルをストレッチしベロシティを描画
- 静的モデル、遠景モデル
  - ワールド空間の移動がない
  - またはワールド空間の移動量の影響が小さい

→Zバッファの情報を元にベロシティを描画

ジオメトリコストの削減

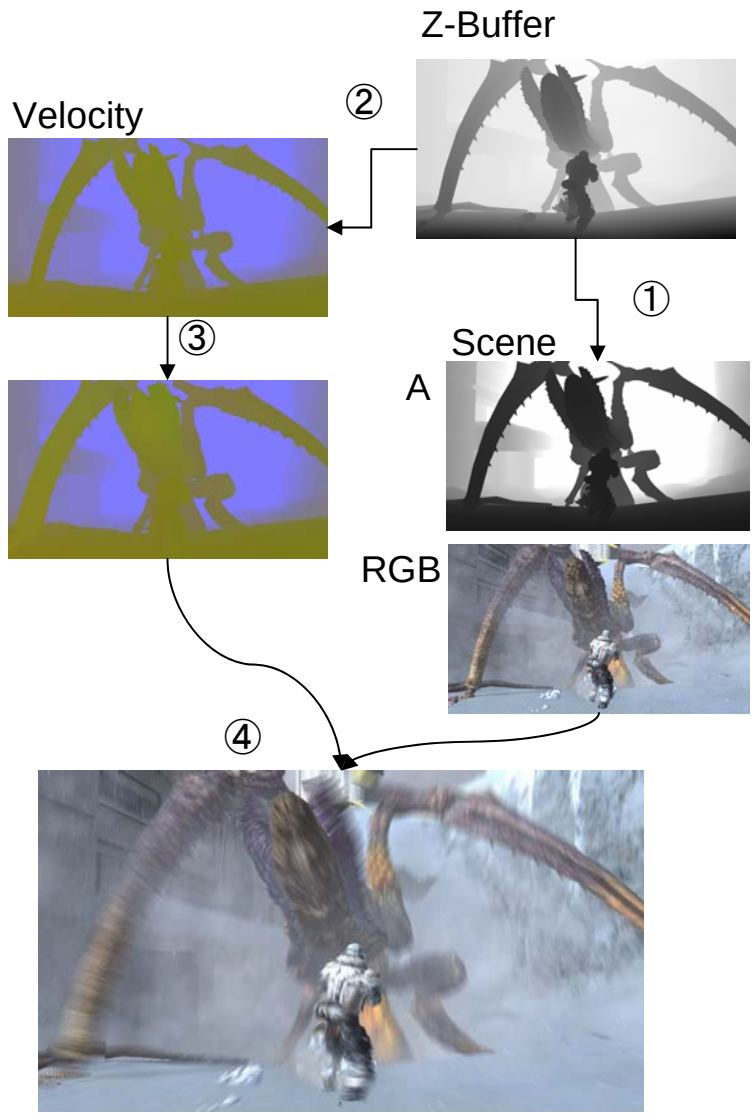
- Zバッファを元にしたベロシティの描画
  - ワールド空間の移動量を考慮しない場合、ベロシティはカメラの動きのみに依存する
    1. スクリーン座標とZバッファの値から、ピクセルのワールド座標を逆算
    2. ワールド座標を一フレーム前のビュープロジェクション行列で変換し、一フレーム前のスクリーン座標を計算
    3. 現在のスクリーン座標と、一フレーム前のスクリーン座標からベロシティを計算

Z-Buffer

Velocity

フルシーン2.5Dモーションブラー

- アルゴリズム
  - ① Zバッファの値を線形のZ値に変換し、シーンのアルファチャンネル(8bit)に書き込む
  - ② Zバッファの値からベロシティを計算し、R,Gチャンネルに描画、同時に線形のZ値を計算し、Bチャンネルに書き込む
  - ③ 近景モデルをストレッチし、ベロシティをR,Gチャンネルに描画、同時に線形のZ値をBチャンネルに書き込む
  - ④ ベロシティと深度情報に基づいてサンプリング



フルシーン2.5Dモーションブラー

Super-Sampling Motion-Blur

Full-Scene 2.5D Motion-Blur

CAPCOM

フルシーン2.5Dモーションブラー

Super-Sampling Motion-Blur

Full-Scene 2.5D Motion-Blur

CAPCOM

フルシーン2.5Dモーションブラー

- 利点
  - ジオメトリの複雑さに依存しにくい
    近景モデルのみが変動要素、あとは一定コスト
  - ユニファイドシェーダーに適したアプローチ
    各パスごとに、ALUをピクセル/頂点処理に特化できる
- 欠点
  - 静止画としての質は高くない
    動画として見る分には大きな問題はない
  - MSAAとの相性が悪い
    回避方法はあるが、完全ではない
    輪郭を少し外側に引き伸ばす（簡易対応）

フィルレートの問題

- 次世代機ではHD解像度が必須
  - 最低でも1280x720解像度、従来の約4倍の面積
  - GPUのフィルレートの向上は4倍程度

    統合的に見るとフィルレートは従来と変わらない

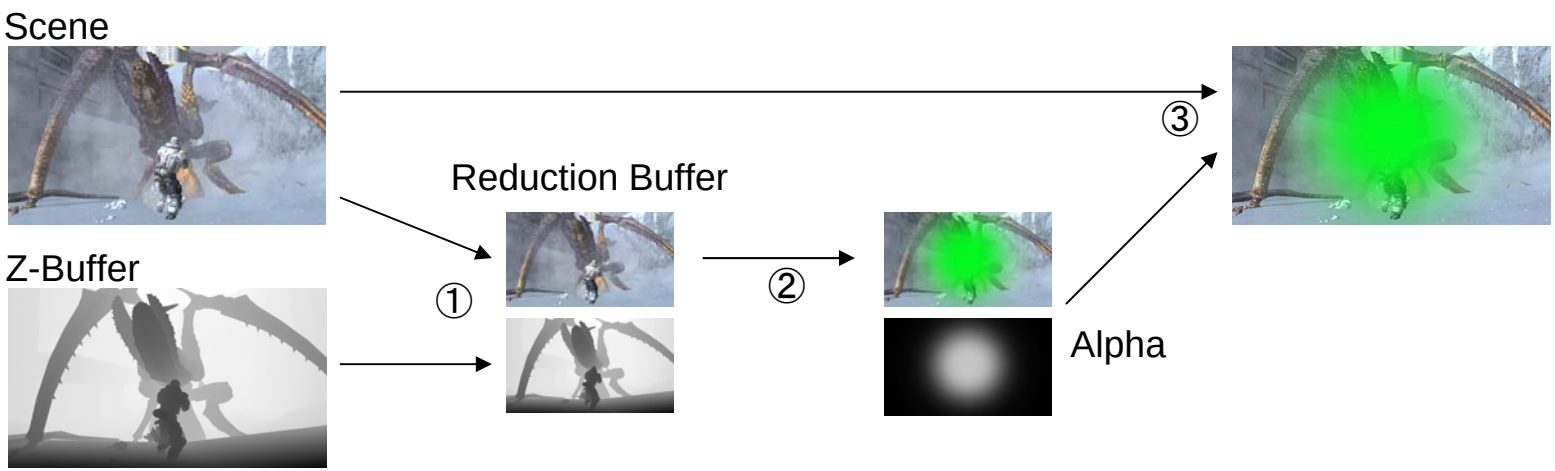
- 次世代機で求められるエフェクト
  - より派手で、よりボリューム感のあるエフェクト

→フィルレートの不足

縮小バッファ

- CEDEC2002
  - DOUBLE-S.T.E.A.L techniques

- エフェクトを縮小したバッファに描画し合成する
  ① シーンのカラーとZを4分の1以下に縮小して、縮小バッファに描画
  ② エフェクトを縮小バッファにアルファに透明度情報を書き込みつつ描画
  ③ 縮小バッファを拡大してアルファ値を元にシーンに合成



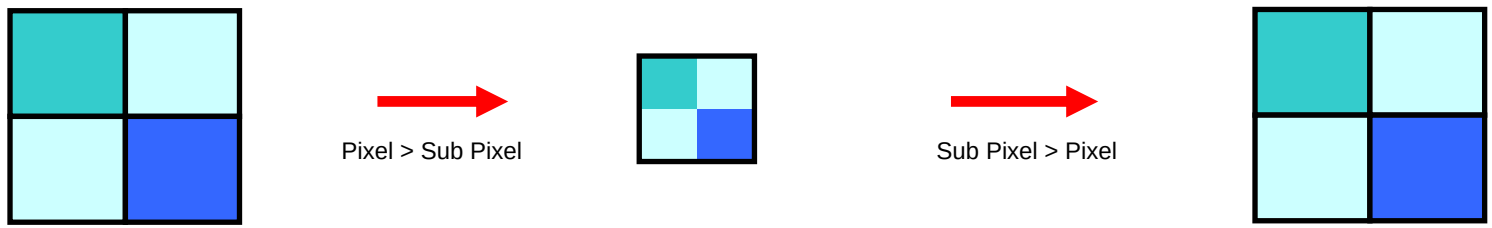
CAPCOM

縮小バッファの問題

- 境界のエイリアス
  - Zテストが正しくおこなわれない
  - オブジェクトとの境界にアーティファクト
- 不自然なぼやけ
  - エフェクトの合成部分がぼやける
- 不自然な合成
  - エフェクトだけが浮いたようなイメージ
  - まったく同じブレンド計算にすることは不可能

MSAAを利用した縮小バッファ

- MSAAのサブピクセルを活用
  1. フレームバッファを¼サイズのMSAA 4倍のフレームバッファに変換
     ピクセルとサブピクセルのメモリマップが等しくなるようにピクセルを並び替え
  2. エフェクトを描画
     カラーは2x2サブピクセル単位に決定
     ブレンドとZテストはサブピクセル単位に行われる
  3. ¼サイズのMSAA 4倍のフレームバッファから等倍のフレームバッファに変換
- MSAAが無償のGPUではフィルレートが4倍に向上



2x2 Pixel　　1 Pixel(2x2 Sub-Pixel)　　2x2 Pixel

MSAAを利用した縮小バッファ

通常（24.2fps)

MSAA縮小バッファを利用 (32.1fps)

CAPCOM

MSAAを利用した縮小バッファ

- 利点
  - 合成の問題がない
    解像度以外の色味などの変化は一切ない
  - Zのエイリアスとボヤケの問題がない
    ピクセルの情報は一切失われない、Zテストはピクセル単位で正確に行われる
  - 8bitの出力先アルファチャンネルを必要としない
    FP10など8bit精度以上のアルファが利用できないフォーマットでも適用可能
- 欠点
  - ブロック化が目立つ
    バイリニア補間が利用できない
  - ハードウェア仕様に依存する
    PCでは利用できない
    MSAAのコストが大きいGPUでは意味がない

ブロック化の軽減

- 遠景のエフェクトに目立ちやすい
  - 一定距離より手前のエフェクトを自動的に縮小バッファに描画。それ以外は、通常描画
- ブラー系のフィルタと併用
  - 被写界深度、モーションブラー、ラジアルブラーなどと組み合わせる

テクスチャ容量と品質の問題

- 容量の問題
  - 次世代機では高解像度テクスチャが必須
  - 法線マップ、マスクマップ、ライトマップなどベースマップ以外のテクスチャが必要になる。
- 帯域の問題
  - 複雑なシェーダーではテクスチャフェッチの量が増大
  - メモリ帯域を圧迫

→DXT(S3TC)圧縮が必須

DXT圧縮の問題

- HDRテクスチャの圧縮ができない
- 法線マップの圧縮に向かない
- 明度の低いテクスチャでマッハバンドが出易い
  - CLUTとは異なり、精度がRGB565に固定
- 一定以上品質を向上させることができない
  - DXT圧縮か非圧縮かの選択肢しかない

DXT圧縮の拡張

- 次世代機のGPU
  - ALUは強力
  - 帯域がボトルネック
- 常に高品質テクスチャが必要とは限らない
  - 容量(解像度)を重視したい場合も多い
- シェーダーに簡易的なテクスチャのデコードを組み込む
  - DXT5のアルファを拡張情報に利用
  - DXT1のアルファが1.0という特性を活用

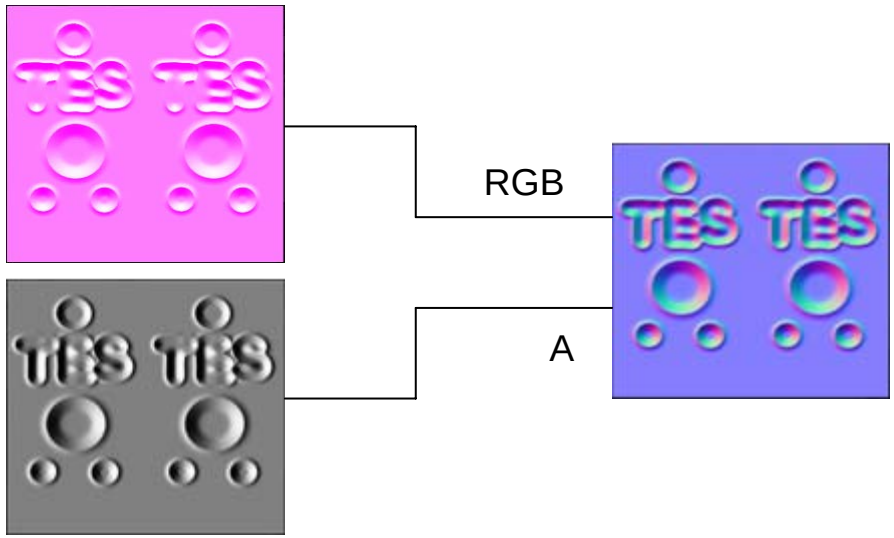
    アルファ1.0がデフォルトとなるようにエンコード

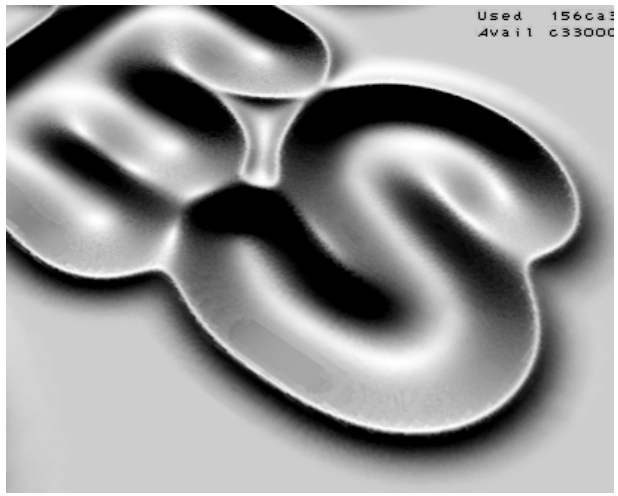
法線テクスチャの圧縮

- エンコード
  - 法線のXをAlpha成分にYをG成分に分離して格納
  - R成分は1.0にしておく
- デコード
  - $X=R \times A$, $Y=G$, $Z=\mathrm{sqrt}(1.0 - X^2 - Y^2)$
- 用途
  - 法線マップ、ディティールマップ
- 特徴
  - XとY成分で独立した線形補間が行えるため品質が向上
  - 専用の法線圧縮フォーマットが利用できる場合は、そちらを利用

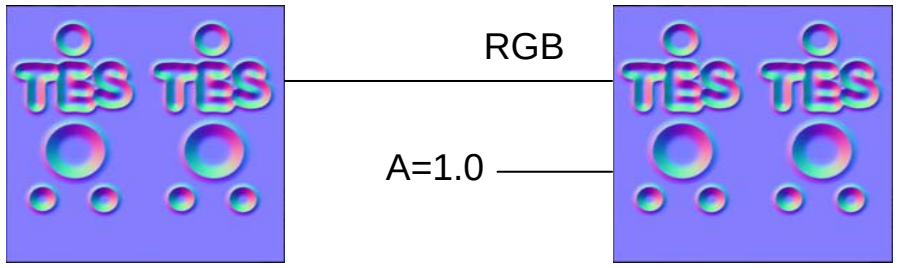
法線テクスチャの圧縮

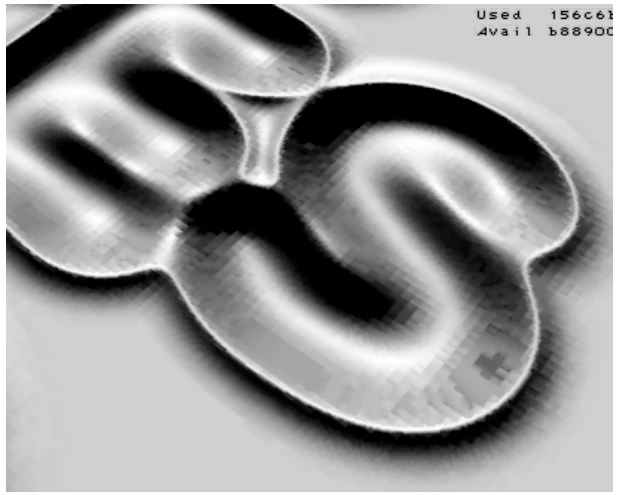
DXT5





DXT1





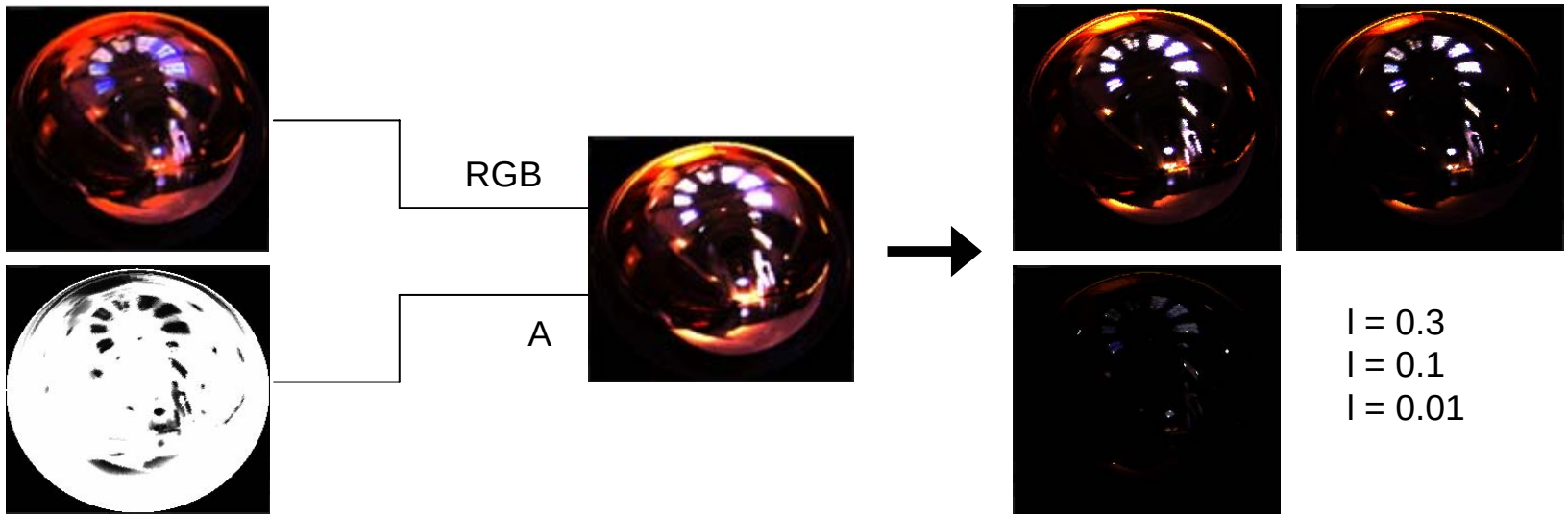
$X=R \times A$, $Y=G$, $Z=\mathrm{sqrt}(1.0 - X^2 - Y^2)$

HDRテクスチャの圧縮

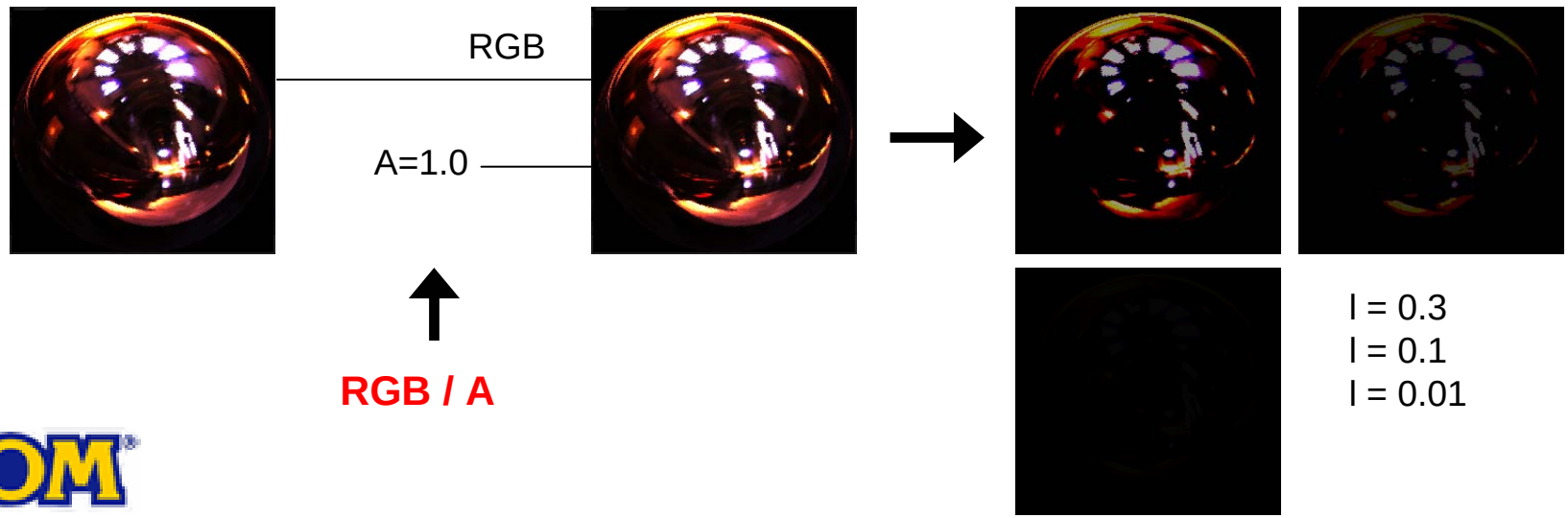
- エンコード
  - {R,G,B,1.0}の最大値の逆数をAlphaに格納
  - RGBを{R,G,B,1.0}の最大値で除算
- デコード
  - RGB / A
- 用途
  - 環境マップ
- 特徴
  - 低いレンジの時により高い階調が得られる
  - RGBEと比べてバイリニア補間時に破綻しない

HDRテクスチャの圧縮

DXT5



DXT1

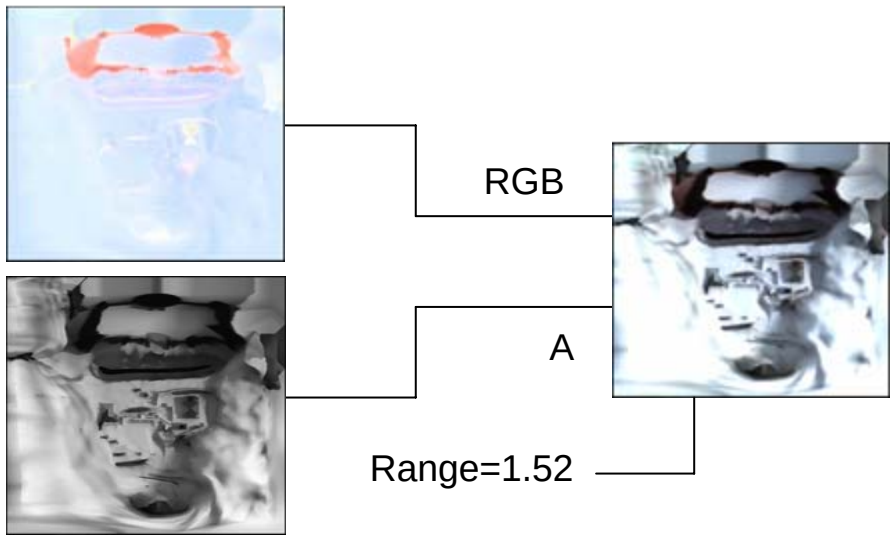


高階調テクスチャの圧縮

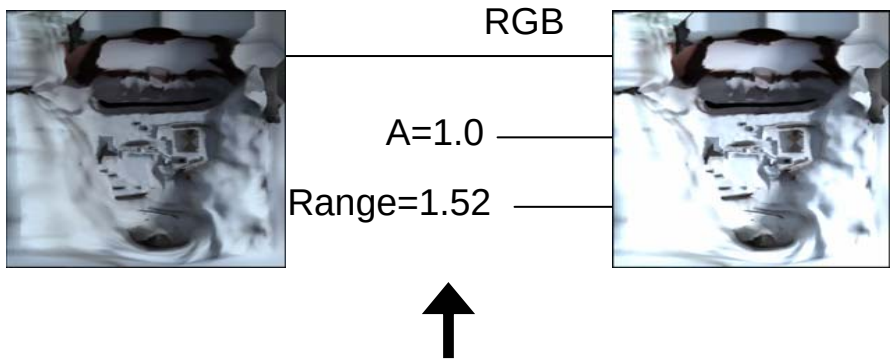
- エンコード
  - テクスチャ全体のRGBの最大値をRangeとし、テクスチャ全体をRangeで除算
  - RGBの長さをAlphaに格納し、RGBは正規化
- デコード
  - RGB × A × Range
- 用途
  - ベースマップ、ライトマップ、一枚絵
- 特徴
  - 全体的に暗いテクスチャは階調が向上
  - 2~4倍程度のHDRテクスチャにも対応可能
    HDRライトマップに最適
  - 高い階調と精度
    DXT5のアルファ値の精度は4x4ブロック辺り8色(DXT1は4色)

高階調テクスチャの圧縮

DXT5



DXT1



RGB × A × Range

テクスチャ圧縮の拡張

- 利点
  - 共通のシェーダーで透過的に利用可能
  - アーティスト側で品質(DXT5)と容量(DXT1)のバランスを柔軟に選択可能
  - 非圧縮テクスチャや、HDR(FP16)テクスチャも共通のシェーダーで利用可能

    動的テクスチャへの対応
- 欠点
  - 余分なシェーダー命令の増加
  - アルファチャンネルを利用するテクスチャには利用できない

まとめ

- フレームレート
  - モーションブラーを使用
- フィルレート
  - 縮小バッファを活用
- テクスチャ品質と容量
  - テクスチャ圧縮を拡張

ご質問は？