Imagire Day

最新世代機に関わる技術トラック

内製ミドルウェア開発にみた 前世代機から次世代機への移り変わり

株式会社バンダイナムコゲームス
コンテンツ制作本部
技術部 技術サポート課
冨澤 茂樹

0.はじめに

- 家庭用据置機の現行機って何？
  - いまいち、あいまい

- 本日の発表では
- 前世代機
  - PlayStation2, GAMECUBE, Xbox
- 次世代機
  - Xbox360, PLAYSTATION3, Wii

……とします

0.はじめに

- シェーダテクニック等は、ここまでのセッションで、お腹いっぱいでは…
- 本セッションでは GPU 側よりも CPU 側にフォーカス

目次

1. 弊社内製ミドルウェア誕生の時代的背景
2. 弊社内製ミドルウェアの概要
3. 次世代機プログラムのあれこれ
4. マルチプラットフォーム対応のあれこれ
5. 異プラットフォーム間の移植のあれこれ
6. マルチコアプロセッサのあれこれ
7. 最後に

目次

1. 弊社内製ミドルウェア誕生の時代的背景

世界を震撼させた言葉

- “The Free Lunch Is Over”
  Herb Sutter (2005)
  - 「タダ飯食いは終わった」
  - 遅いプログラムを書いても CPU の高クロック化に頼っていた時代は終わりだ

- ちょっと待てよ…
  我々はタダ飯を食べてきたのだろうか？

2D 時代

- 業務用ハードウェア・各社ボード
- 家庭用ハードウェア・ファミコン等
  - ハードウェアエンジニア、プログラマ共に 1 枚でも多くのスプライトを表示するために全身全霊を尽くしていた
  - スプライトレース

3D 時代

- まずは業務用
  - ハードウェアエンジニア、プログラマ共に 1 枚でも多くのポリゴンを高フレームレートで表示するために全身全霊を尽くした
  - ジオメトリエンジン、ジオメタライザー
  - ラスタライザー
  - ポリゴンレースの始まり

3D 時代 (続き)

- そして家庭用になだれこむ
  - 固定されたハードウェア
  - ハードウェアベンダーがミドルウェア的部分を提供していても、プログラマはそれを使わず 1 枚でも多くのポリゴンを表示するためにアセンブリ言語等を駆使し尽くした
  - リフレクションなど
  - ポリゴンレース激化

3D 時代 (続き)

- PlayStation2 の登場
  - ハードウェアベンダーがミドルウェア的な部分を提供しないハードウェア
  - しかし旧ナムコを含め数社は歓迎
  - プログラマのウデ試し
  - 1 枚でも多くのポリゴン！
  - 少しでも違う映像表現！
- ポリゴングランプリ！

弊社社内での動き

- 「ライブラリ」はいくつかのプロジェクトで作られていた
- それらは総て、GS パケット作成支援、DMAC 制御支援ライブラリ
- そのライブラリを使うためには「力」あるプログラマを必要とした
  - VU プログラムを書く
  - それに合わせたモデルフォーマットを作る
  - そのモデルフォーマットを出力するコンバータ

そこで

- そこまでの「力」を必要としない、いわゆる「ミドルウェア」ライブラリを提案した

- ここに本ミドルウェア・ライブラリの誕生

折しも

- 世間でもミドルウェアブーム
- GAMECUBE, Xbox の登場
- マルチプラットフォームの要求
- ハードウェア隠蔽化の要求

- これらが後押しとなってくれた
- そして今に至る

目次

2. 弊社内製ミドルウェアの概要

本ライブラリの概要

- ライブラリファミリ
  - コア
    - グラフィックエンジン
    - ファイルシステム
    - コントローラ入力
    - デバッグ支援
  - サウンド
  - モーション
  - フレームワーク
- 本日はコアライブラリにフォーカス

対応機種

- 前世代機版

- PlayStation2
- GAMECUBE
- Xbox
- PlayStation Portable
- 業務用基板

対応機種

- 次世代機版

- Xbox360
- PLAYSTATION3
- Wii
- 業務用基板

PC プラットフォームは？

- 次世代機と PC のアーキテクチャ
  - マルチコアCPU
  - プログラマブルシェーダGPU
- 一見似ているようだが、実は全く異なるアーキテクチャ

- Xbox360GPU ≠ RADEON
- RSX ≠ GeForce
- Hollywood ≠ RADEON

PC プラットフォームは？

- Xbox360CPU ≠ x86, x86-64
- Cell ≠ x86, x86-64
- Broadway ≠ x86, x86-64

- 「ついで」に作ることが可能なものではない
- 別のひとつのプラットフォームと考える
- ビューワ, レベルエディタ目的に使用することは考えていない

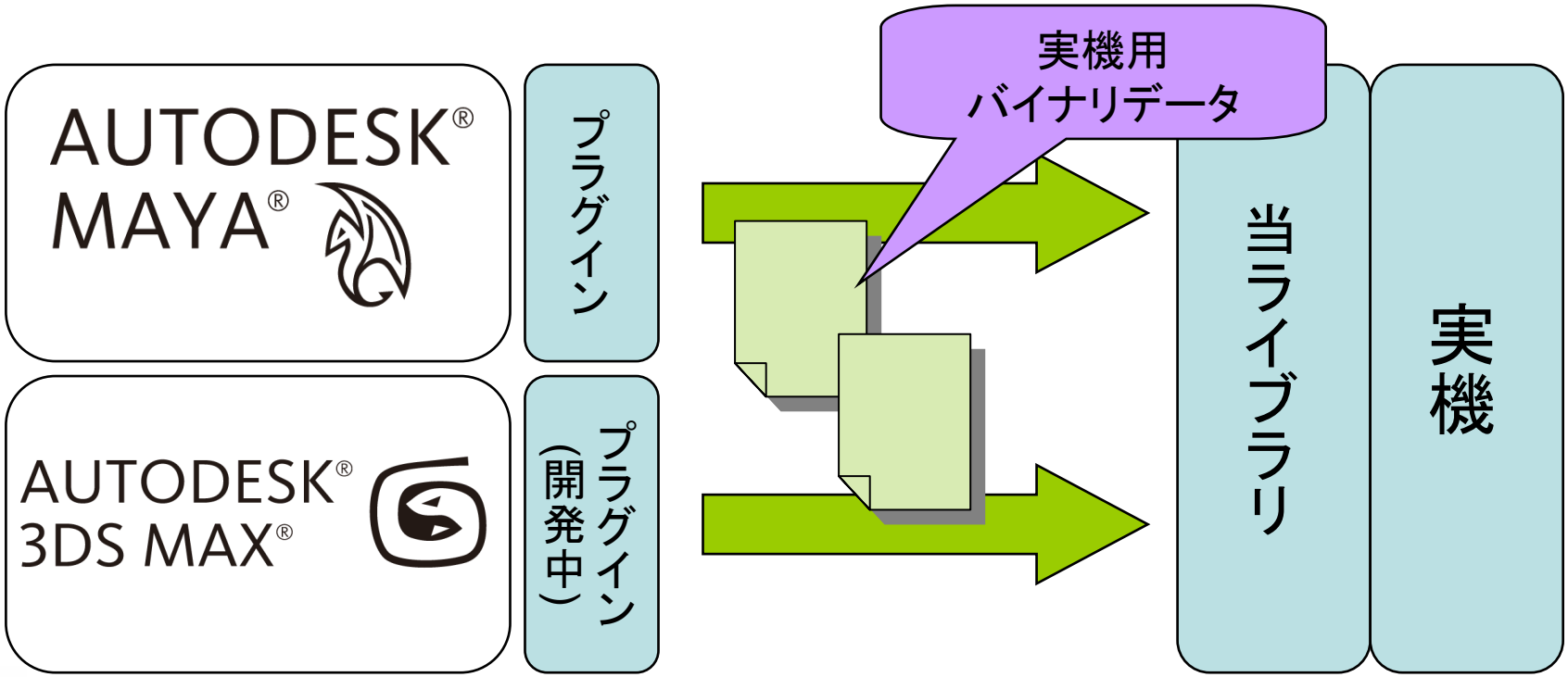
それでは

- ビューワ, レベルエディタはどうする？

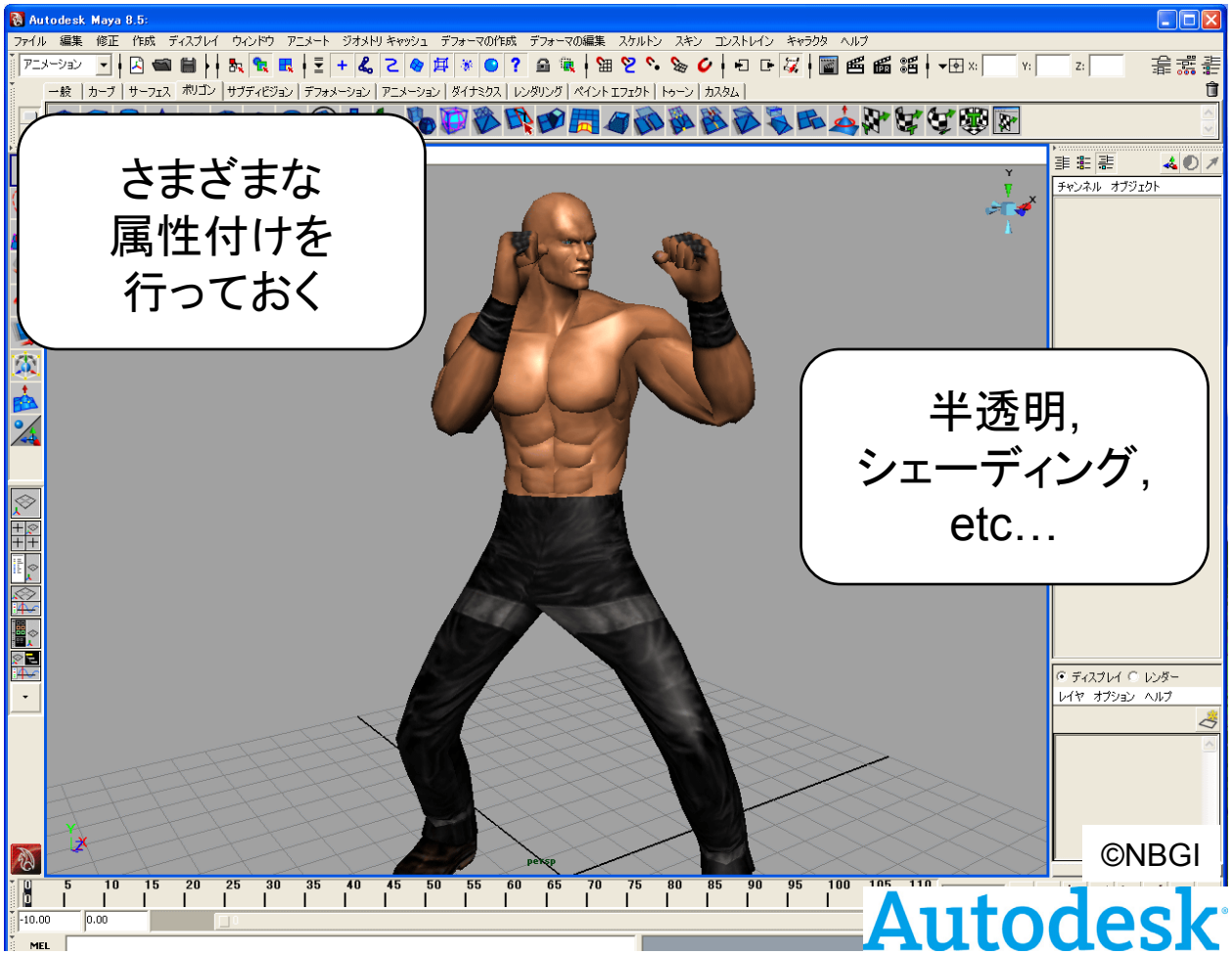
- PC ⇔ 実機の通信環境を提供

データドリブンアーキテクチャ

- Maya 上で全ての属性を設定
- プラグインから実機用バイナリをエクスポート



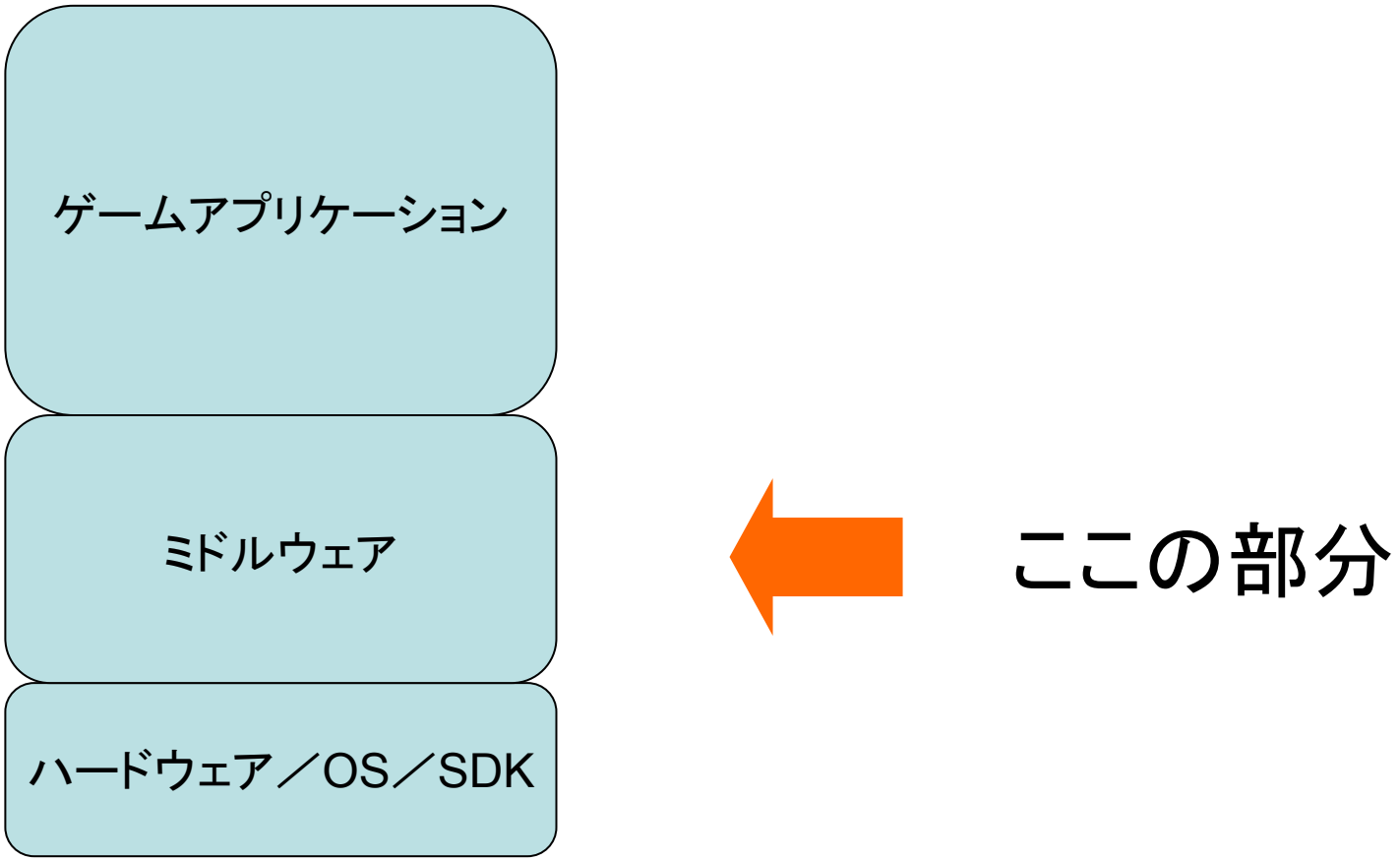
Maya からエクスポート



デモンストレーション

階層構造でみる位置付け

- 前世代機や当ミドルウェアの場合



開発者の実情

- ゲームメイン(あるいはジョブマネージャ、タスクマネージャ、フレームワークと呼ばれる部分)は、デリケートな部分のひとつ

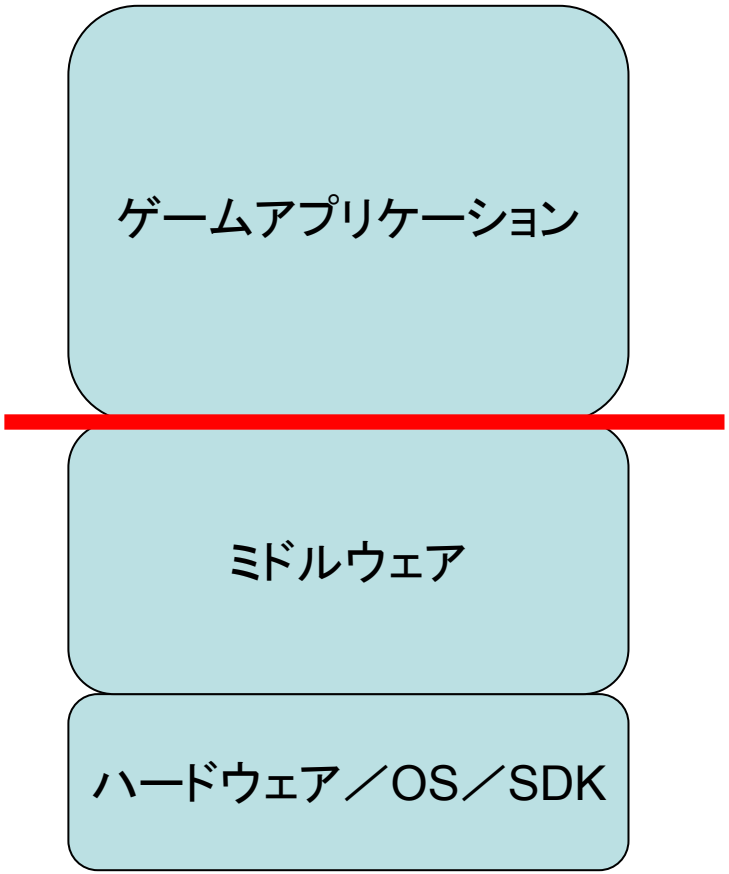
- あまり踏み込まないほうが良い場合もある

微妙な領域

- フレームワーク層
- カメラ制御ライブラリ
  - プログラマ各人、プロジェクトごと、に自前のカメラ制御ルーチンがある
- 数値演算
  - C 風か？ C++ か？ 演算子を使うか？ …

もう一度階層構造

- 難しい…

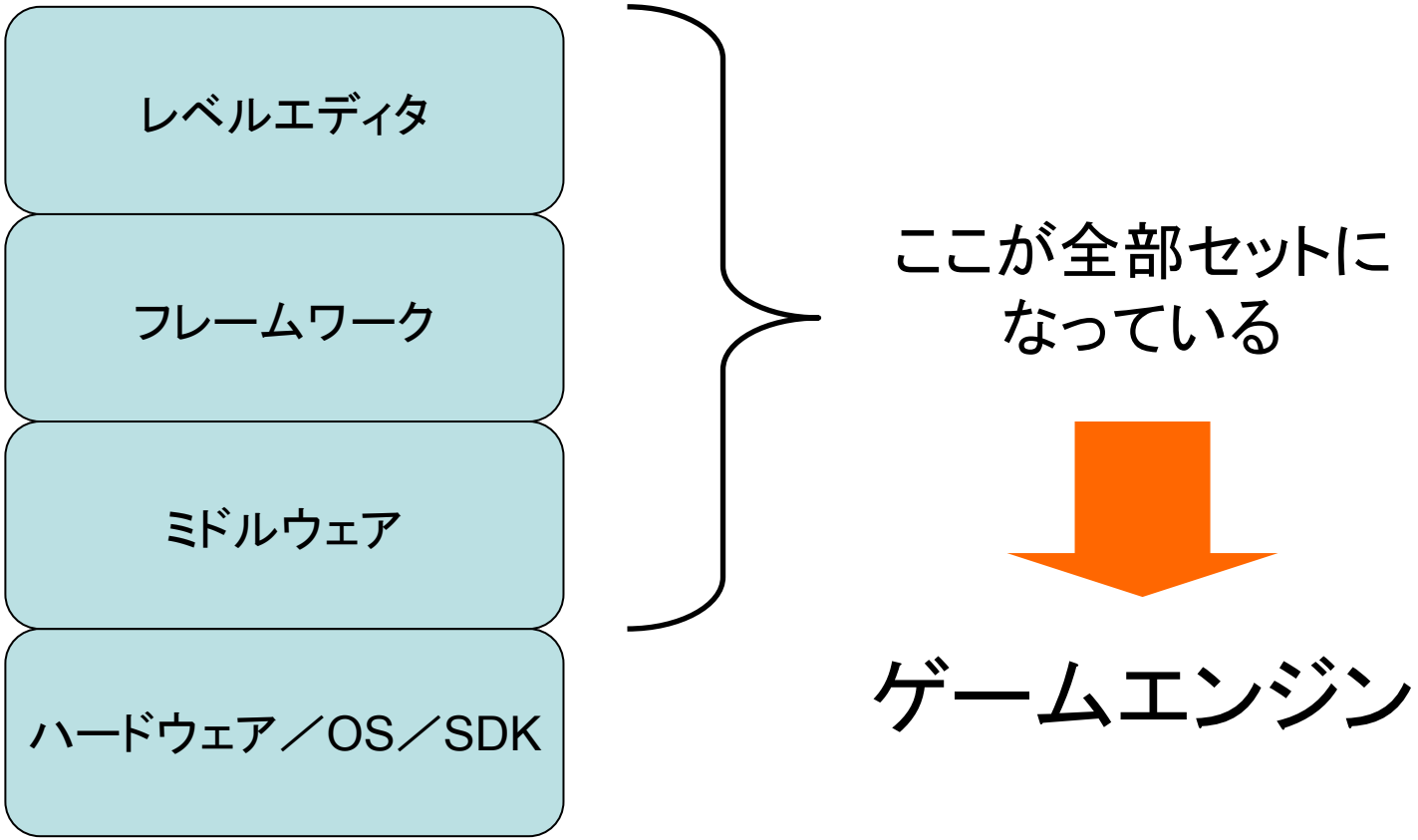


ここの線引きが難しい！

- ハッキリと線を
  引かないほうが
  良いかもしれない

階層構造でみる位置付け

- 昨今のトレンド(特に海外製)の場合

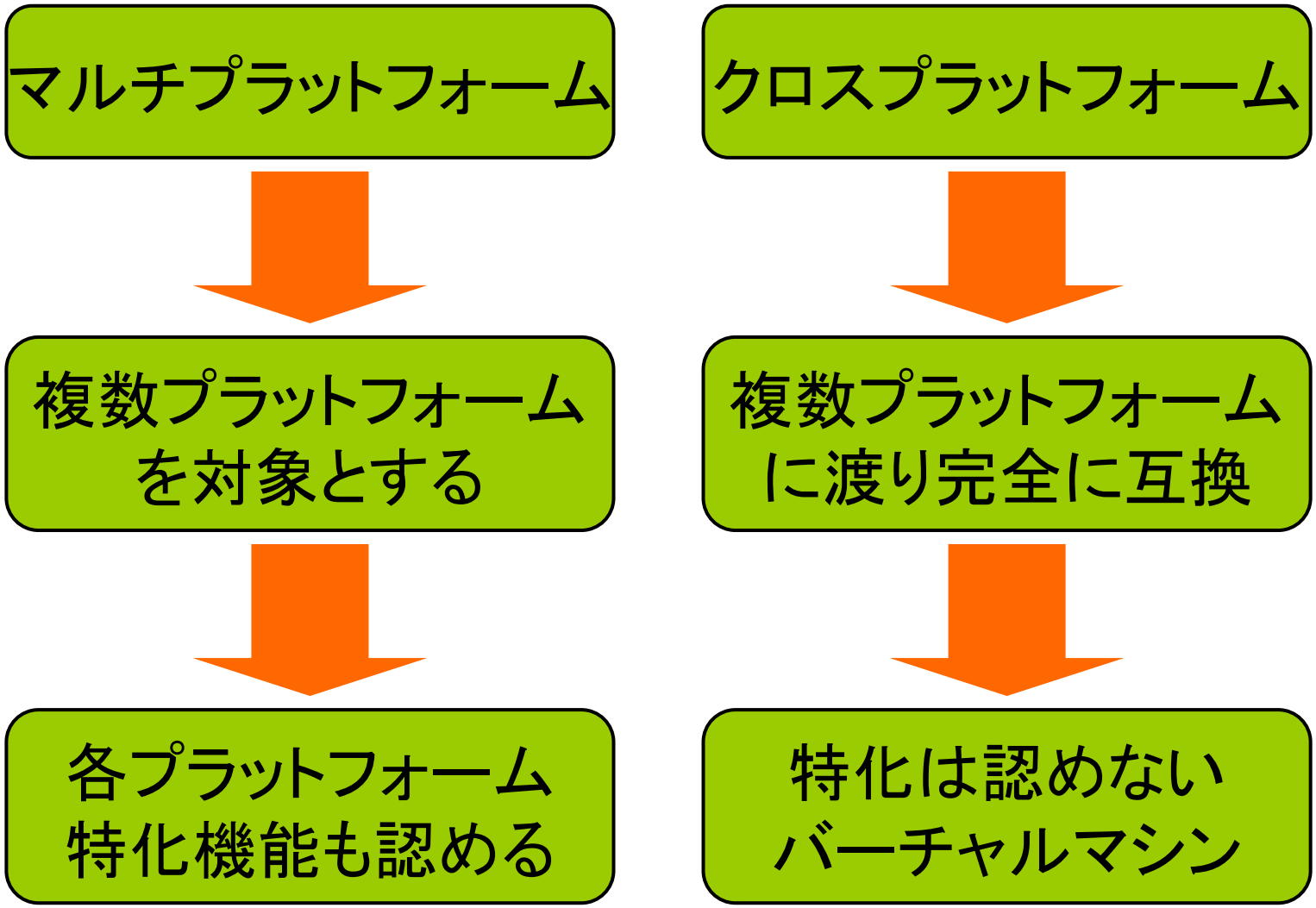


ゲームエンジン

- 制作するゲームとゲームエンジンとが完全にマッチしていて、制作方法も完全にマッチしていれば非常に有用
- だが、おのずとジャンルが限定されてくる

- 本ライブラリは、この路線をとらなかった

マルチ？クロス？


マルチプラットフォーム
複数プラットフォーム
を対象とする
各プラットフォーム
特化機能も認める
クロスプラットフォーム
複数プラットフォーム
に渡り完全に互換
特化は認めない
バーチャルマシン

マルチ？クロス？ (続き)

- 本ライブラリは
「マルチプラットフォーム・ライブラリ」
の路を選択
- そのため、マルチプラットフォーム展開を考えたタイトルだけでなく、単独プラットフォームで発売するタイトルにも数多く採用された
- むしろ単機種タイトルのほうが多い…

前世代機版採用タイトル

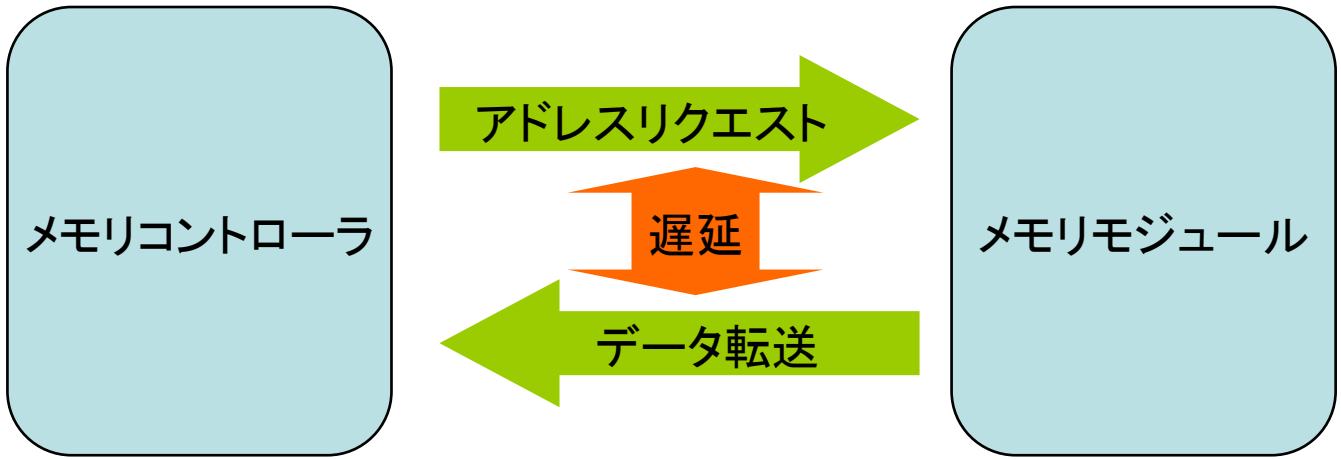
目次

3. 次世代機プログラムのあれこれ

最初に言っておく
メモリは
か～～～～な～～～～り
遅い

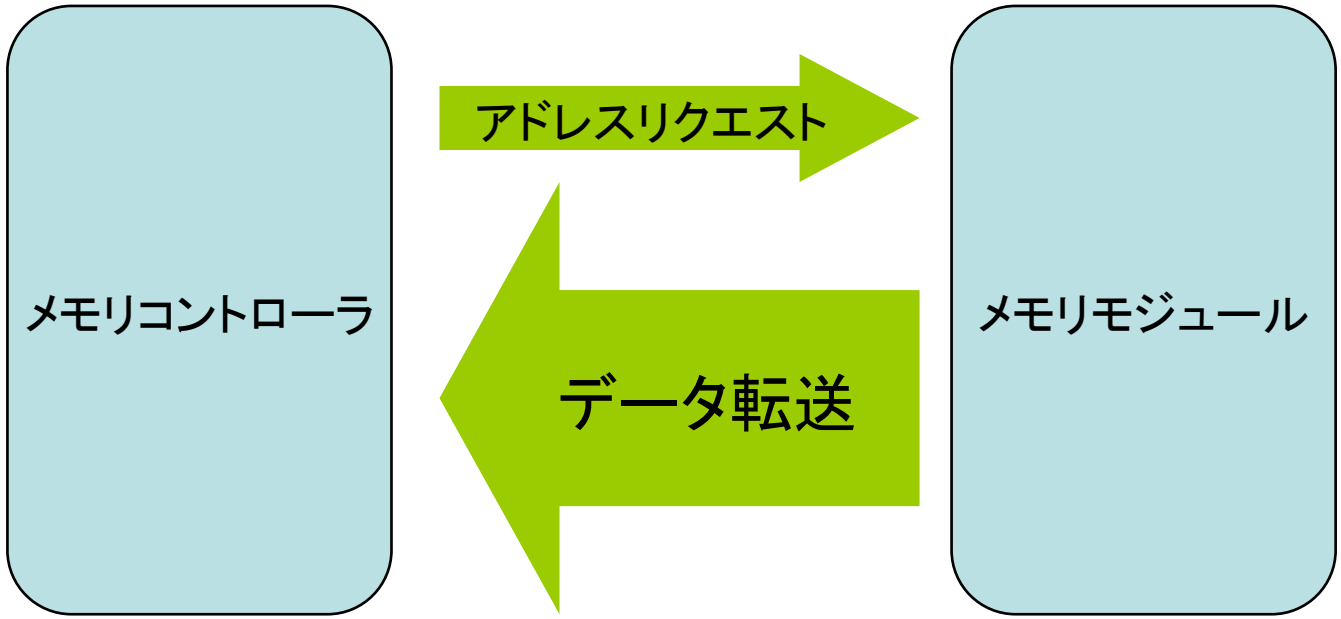
現在のメモリの特徴

- バンド幅は広い(数十GB/s)
- ただし、リクエストからリード／ライトまでの遅延(レイテンシ)が大きい
  - ランダムなアクセスは苦手
  - 小さい単位のアクセスは苦手



バースト転送

- 1 回のアドレスリクエストに対して、一定サイズのデータを転送する
  - キャッシュラインサイズ
  - DMA 転送サイズ



PowerPC コアの注意点

- Xbox360CPU, Cell の PowerPC コアの共通した特徴
  - ハードウェア 2 スレッド
  - 長～～～～～～～いパイプライン
  - インオーダー実行
  - 決して大きくないキャッシュ
  - MMU 搭載
  - VMX 搭載 (Xbox360CPU では拡張)

長いパイプライン

- インラインアセンブリとお別れ
  - インオーダー実行の長いパイプラインを予測して、手動でアセンブリ言語でプログラムするのは、もはや不可能
- コンパイラの最適化に頼る
  - コンパイラとうまく付き合う必要あり
- どうしてもアセンブリを書かなければいけないときは、コンパイラの組み込み関数を使う
  - VMX 命令、キャッシュ命令など

MMU

- Memory Management Unit：メモリ管理ユニット
  - 仮想記憶 (HDDへのSWAPなど)
  - セキュリティ (読み／書き／実行 など)
  - 物理メモリの断片化防止
    - 仮想メモリ→物理メモリのアドレス変換
  - 複数プロセス

かなりの情報量

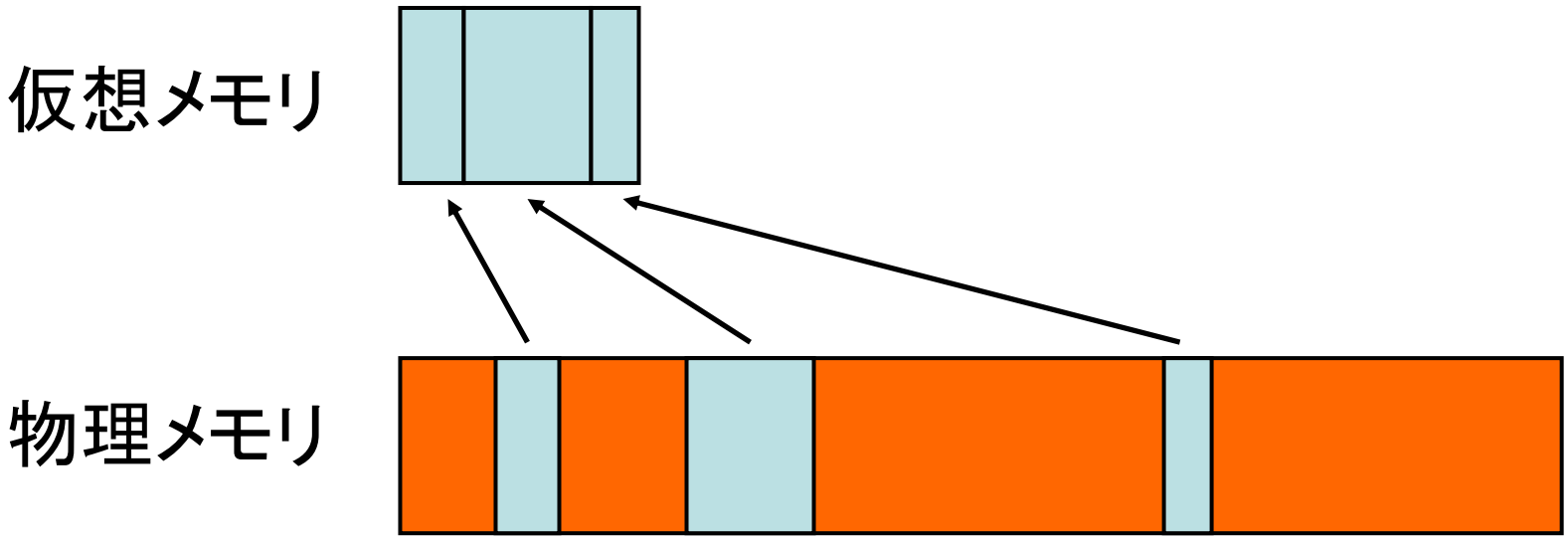
MMU (続き)

- すべてはページ単位で行われる
  - 4kB, ???B, ???B
  - 2 の N 乗サイズ
- 各情報は 1ページ、1 エントリとして、物理メモリ上にテーブルとして置かれる

PTE (Page Table Entry : ページテーブルエントリ)

MMU (続き)

- 仮想メモリ→実効アドレス空間 (Effective Address)
- 物理メモリ→実アドレス空間 (Real Address)
- EA から RA へのアドレス変換が行われる



TLB

- メモリアクセスのたびに物理メモリ上の PTE を参照しにいっていると多大な時間がかかる

- そこでオンチップの
  TLB (Translation Look-aside Buffer : アドレス変換緩衝機構)
  ATC (Address Translation Cache : アドレス変換キャッシュ)
  と呼ばれるキャッシュが載っている

TLB ミス

- 前述の通り TLB は、あくまで「キャッシュ」

- アドレス変換キャッシュミス、すなわち「TLB ミス」を、なるべく起こさないようにする

- なるべく大きなページを使用し、PTE 使用量自体を減らす

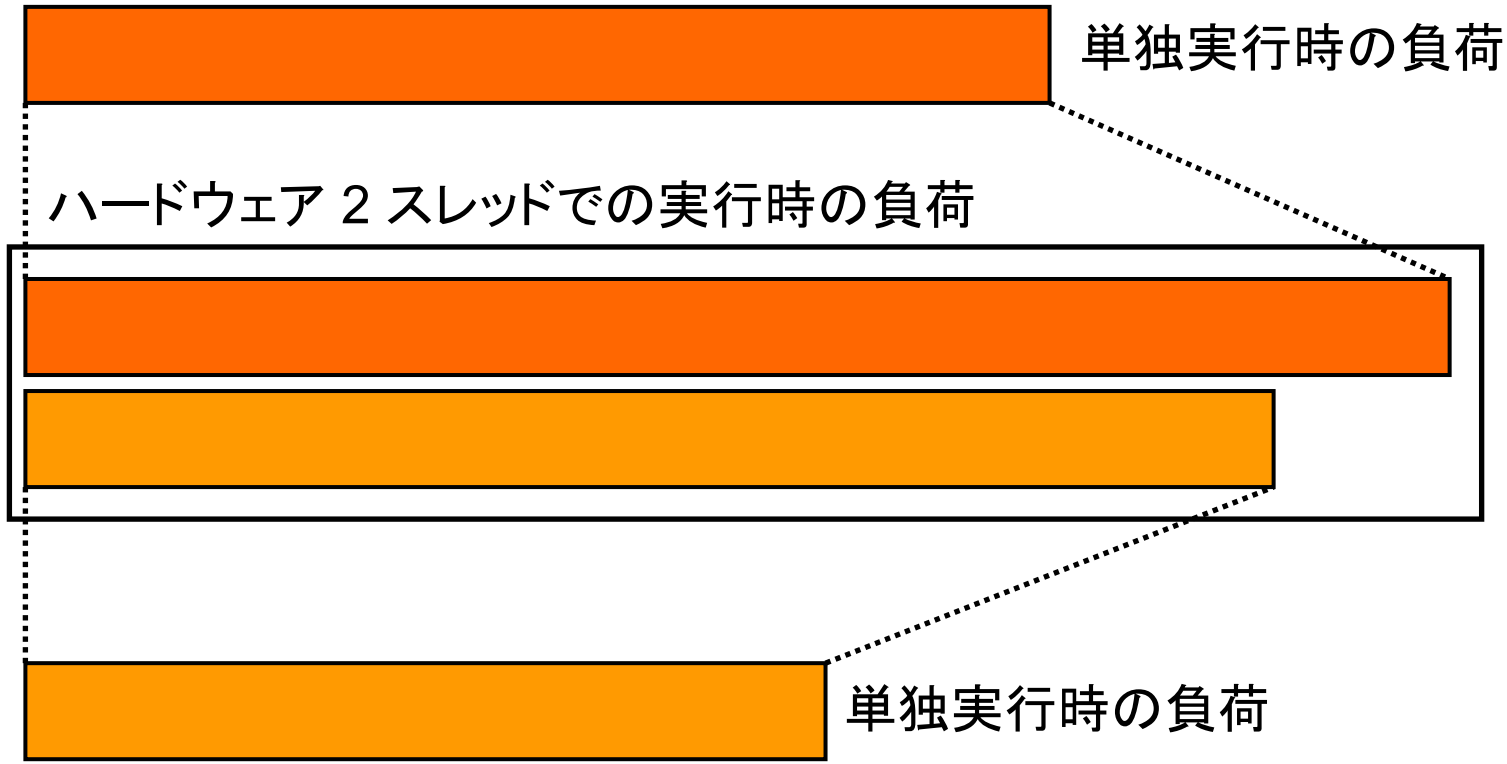
ハードウェア 2 スレッド

- レジスタ以外は、ほぼ共有
- 共有したユニットを両スレッドで競合すると足を引っ張り合うことになる
- カリカリにチューニングしたプログラム同士を走らせるとかえって遅くなる

200% にはならないことを
覚悟しておきましょう！

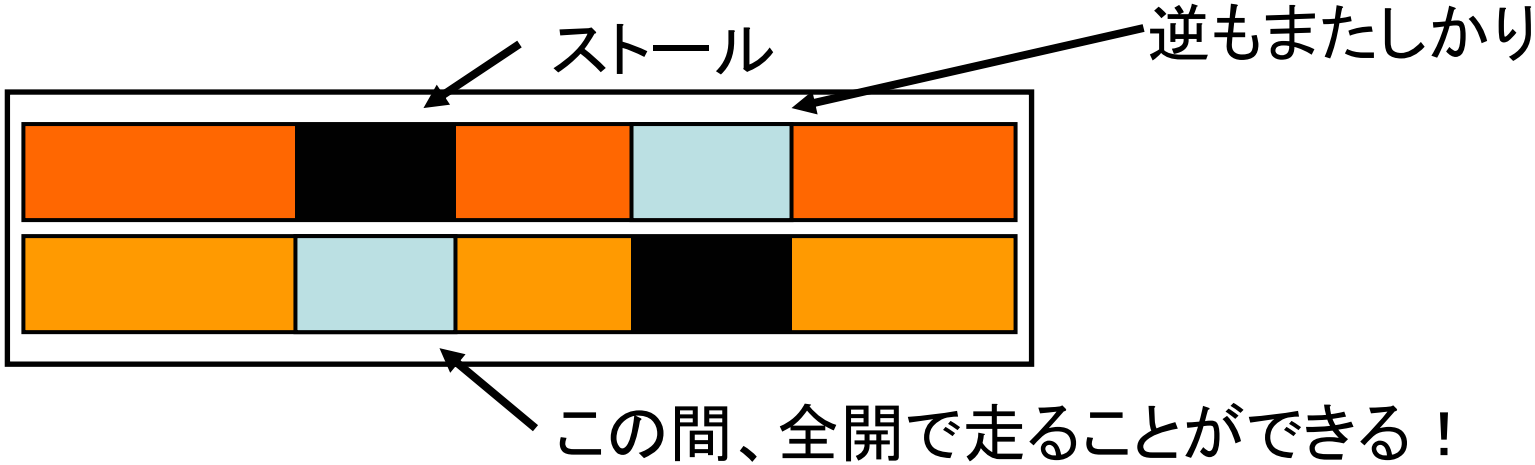
ハードウェア 2 スレッド (続き)

- 単独実行時よりも遅くなる…



効率の良い使い方

- キャッシュミス、TLB ミス、分岐ミス等、パイプラインストールが多いコード
- 片方のスレッドでストールが起こっている間、もう片方のスレッドが走る
- ストールを隠蔽できる!!!



その他

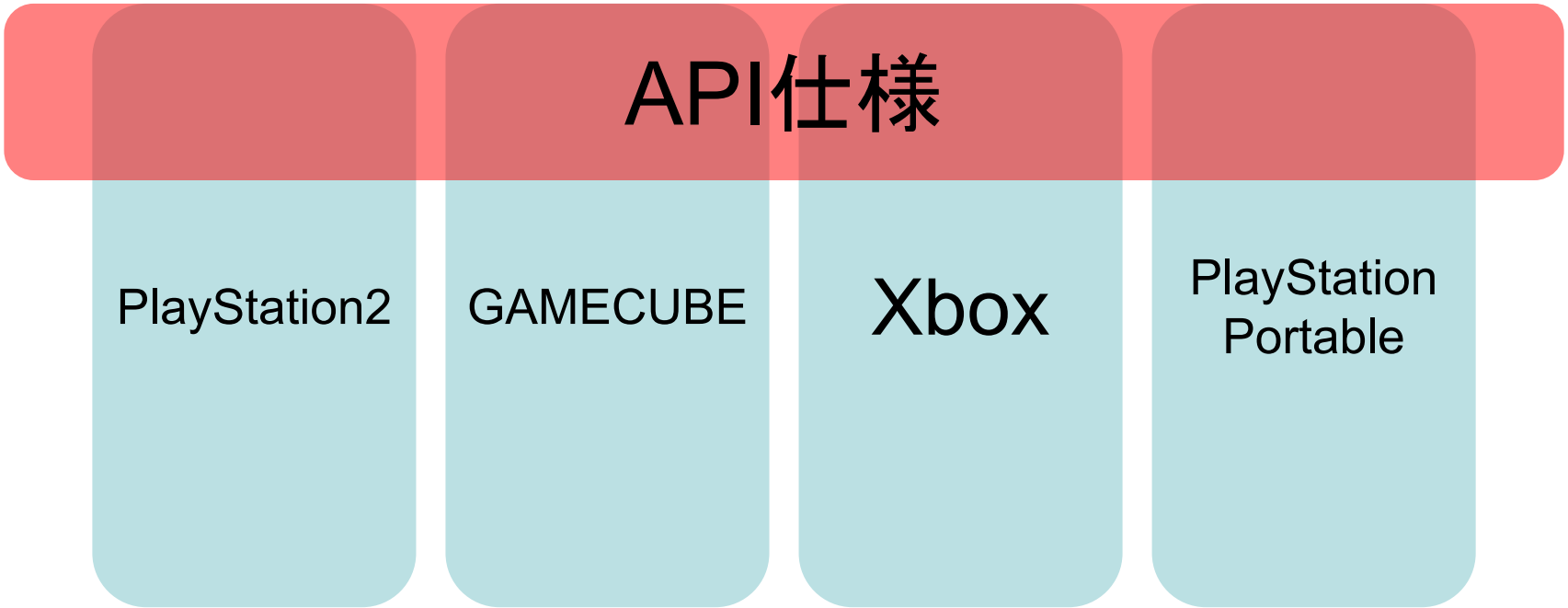
- プラットフォームベンダーさんからの注意
  - ロード・ヒット・ストア
  - ポインタのエイリアシング
  - 整数、浮動小数、VMX 各レジスタ間の移動
  - 浮動小数比較, ベクトル比較
  - リーフ関数の使用
  - etc…
- 正しくプロファイリングし、ボトルネックを特定してから最適化することが重要

目次

4. マルチプラットフォーム対応のあれこれ

前世代機版アーキテクチャ

- API仕様だけを決めて各プラットフォーム担当が実装



理由と見直すきっかけ

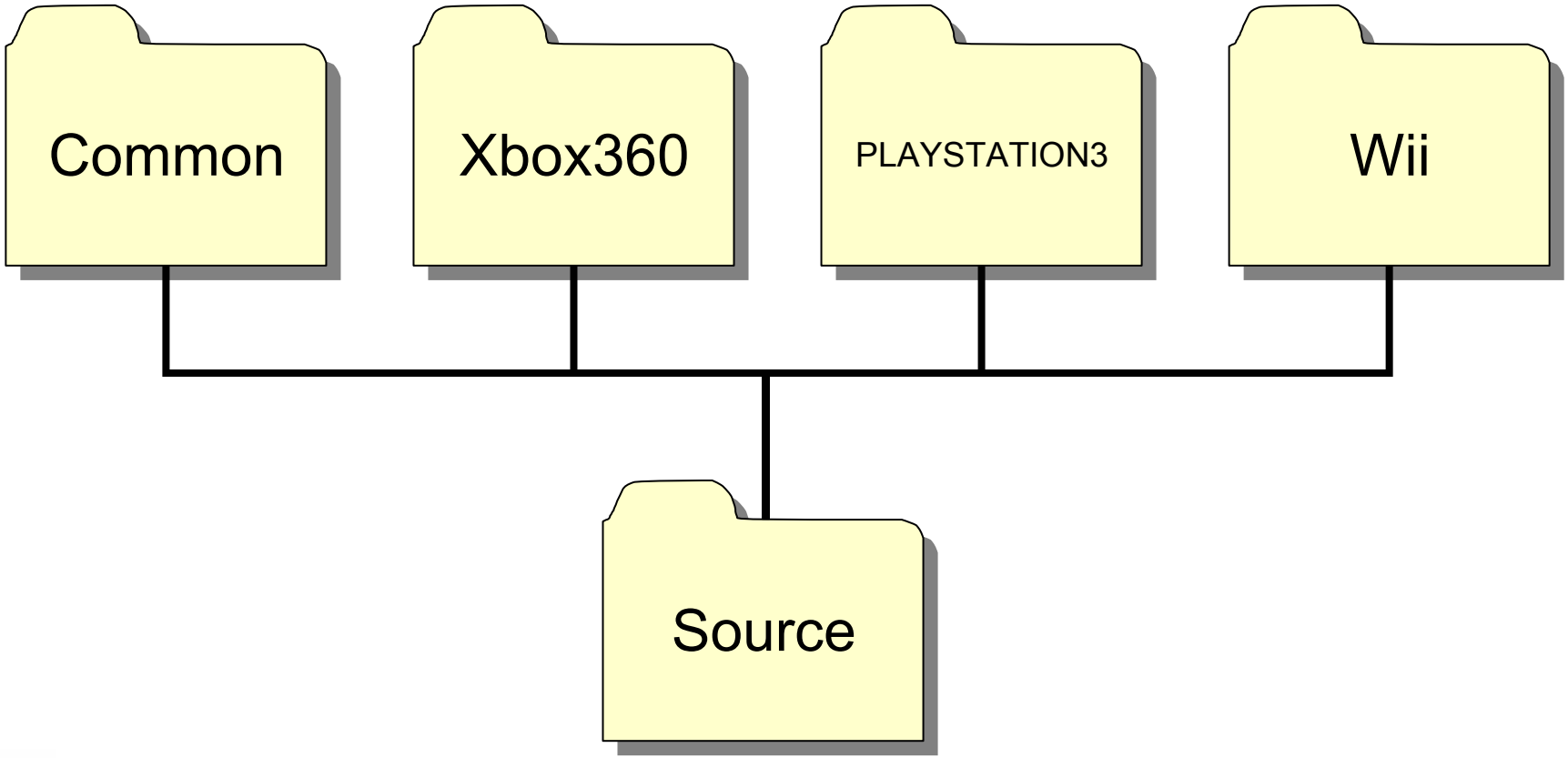
- 理由
  - PlayStation2 版がかなり先行していたため、GAMECUBE 版, Xbox 版は「移植」という形になった
  - PlayStation2 版に追いついた後も引き続き
- 見直すきっかけ
  - メンテナンスがしづらい
  - それぞれ、ばらばらの方向へ

次世代機版アーキテクチャ

- なるべく共通ソースを使うように
  アーキテクチャを見直した

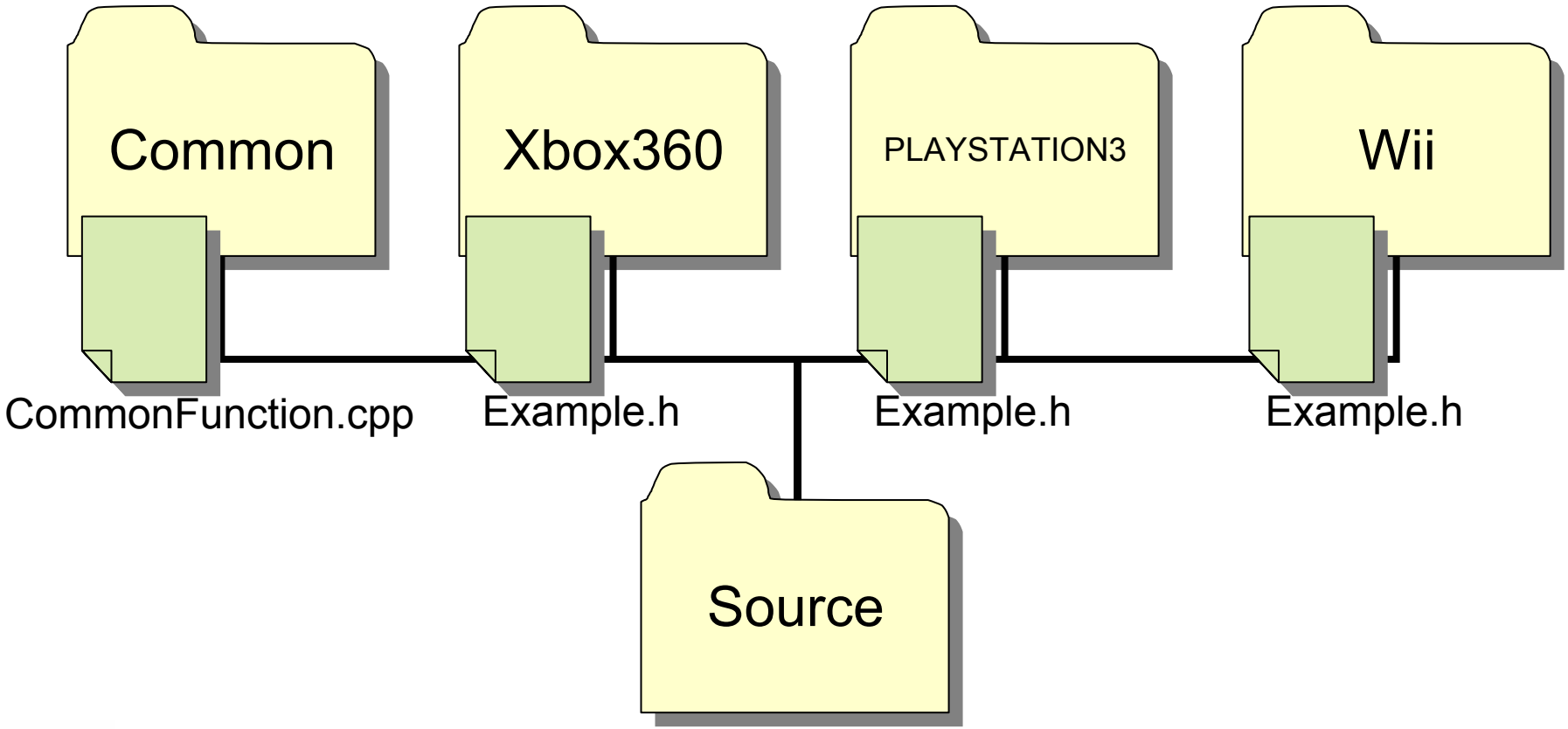
共通コード化

- ディレクトリ構造



共通コード化 (続き)

- インクルードパスのギミック



インクルードパス

- ビルド時、”Common” と自分の機種のディレクトリにインクルードパスを通しておく
- 共通コードに

```
#include　“Example.h”
```

と書いておけば、自動的に自分の機種のヘッダをインクルードしてくれる

Interface を使う

- 教科書に載っている例

```
#define    interface    struct
#define    PURE    = 0

interface IExample {
    virtual void Function1() PURE;
    virtual void Function2() PURE;
};
```

IExample.h

```
#include    “IExample.h”

void CommonFunction( IExample *pIEx )
{
    pIEx->Funtion1();
    pIEx->Funtion2();
}
```

CommonFunction.cpp

Interface を使う (続き)

- 機種別の実装

```
#include    “IExample.h”
class Example : public IExample {
    MachineSpecial    m_machineSpecial;
public:
    virtual void Function1();
    virtual void Function2();
};
```

Machine¥Example.h

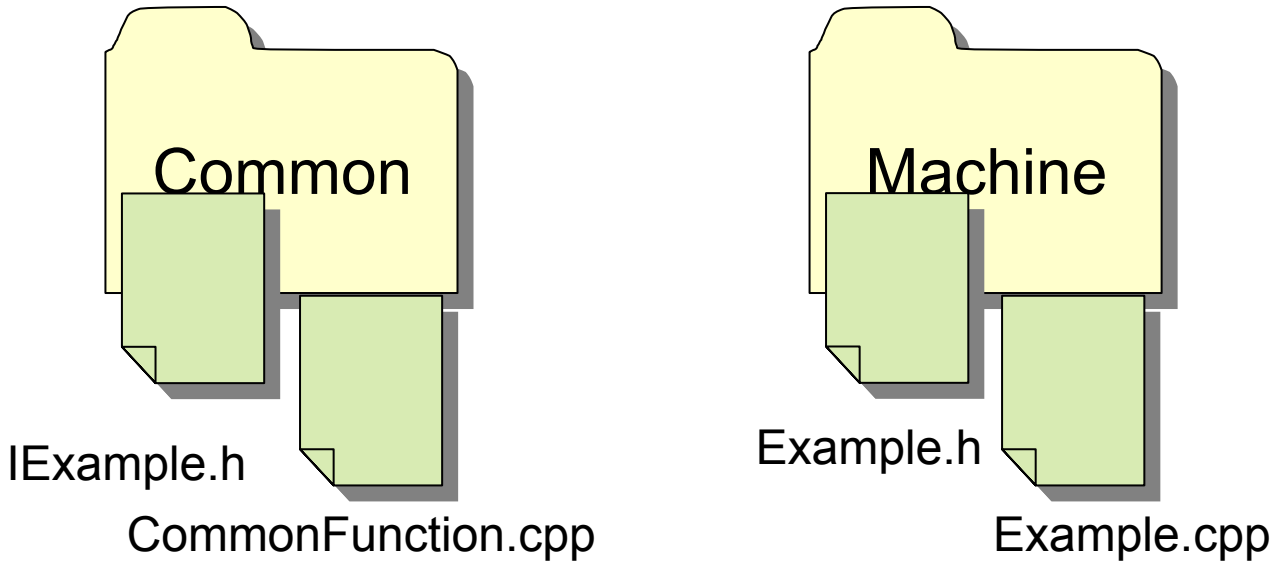
```
#include    “Example.h”

void Example::Function1()
{
    // 機種別コード.
    …
}
```

Machine¥Example.cpp

Interface の利点

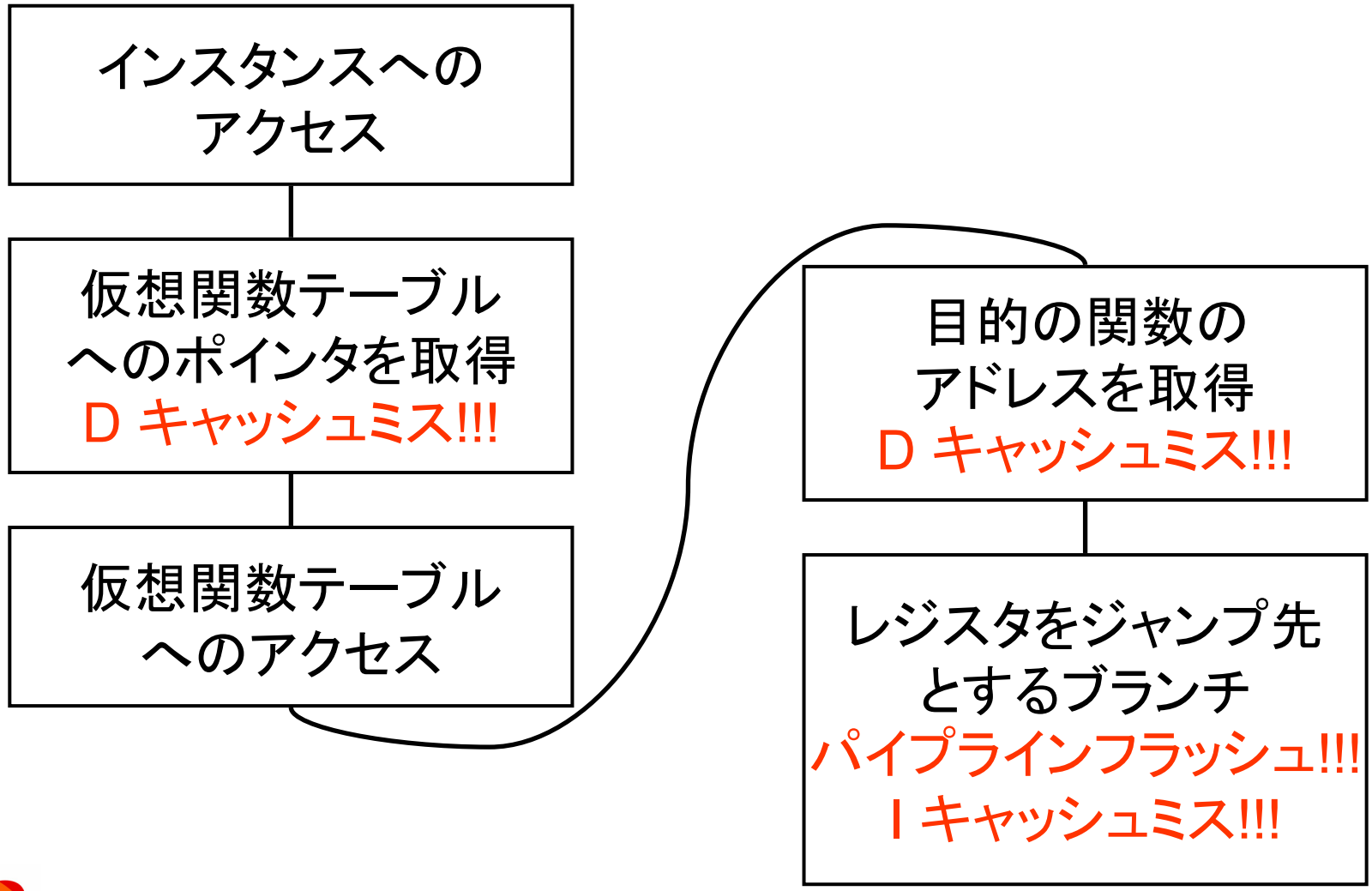
- 共通コードを安全に正しく書くことができる
  - IExample.h, CommonFunction.cpp は総ての機種にわたり、そのまま使用できる
  - Machine¥Example.{h,cpp} のように機種別のコードを書けば、正しく動作する



Interface の欠点

- 仮想関数を使う!!!
- 仮想関数は大きなペナルティ!!!
- しかも本来、仮想関数を使う必要がない状況
  - 本来はランタイムに挙動を変える場合に使う
  - 今回はビルド時に挙動(コード)は決まっている

仮想関数のペナルティ


インスタンスへの
アクセス
仮想関数テーブル
へのポインタを取得
D キャッシュミス!!!
仮想関数テーブル
へのアクセス
目的の関数の
アドレスを取得
D キャッシュミス!!!
レジスタをジャンプ先
とするブランチ
パイプラインフラッシュ!!!
I キャッシュミス!!!

Interface を使わない

- そこで、あまり美しくはないですが…
- Common ディレクトリ

```
void Function1();
void Function2();
```

ExampleFunction.h

```
#include    “Example.h”

void CommonFunction( Example *pEx )
{
    pEx->Funtion1();
    pEx->Funtion2();
}
```

CommonFunction.cpp

Interface を使わない (続き)

- 機種別ディレクトリ

```
class Example {
    MachineSpecial    m_machineSpecial;
public:

#include    “ExampleFunction.h”

};
```

Machine¥Example.h

```
#include    “Example.h”

void Example::Function1()
{
    // 機種別コード.
    …
}
```

Machine¥Example.cpp

目次

5. 異プラットフォーム間の移植のあれこれ

ケーススタディ 1

- テイルズオブシンフォニア PlayStation2 版

テイルズオブシンフォニア

- ナムコテイルズスタジオ制作
- 元々 GAMECUBE で販売することしか考えていなかった
  - 任天堂さんのミドルウェアを使用
- そのためデータ、プログラム、サウンド等すべてが GAMECUBE に特化
- ところが、お客様からの強い要望が寄せられ PlayStation2 版の発売が決定

PlayStation2 への移植

- GAMECUBE → PlayStation2 の移植

- ミドルウェア to ミドルウェアの移植
  - プログラムの修正箇所は少なくてすんだ

- とはいえ与えられたのは 6 ヶ月という短期間
- 限られたスタッフ
  - プログラマ 3 + 3 名
  - 企画は追加仕様やデバッグに

作業内容

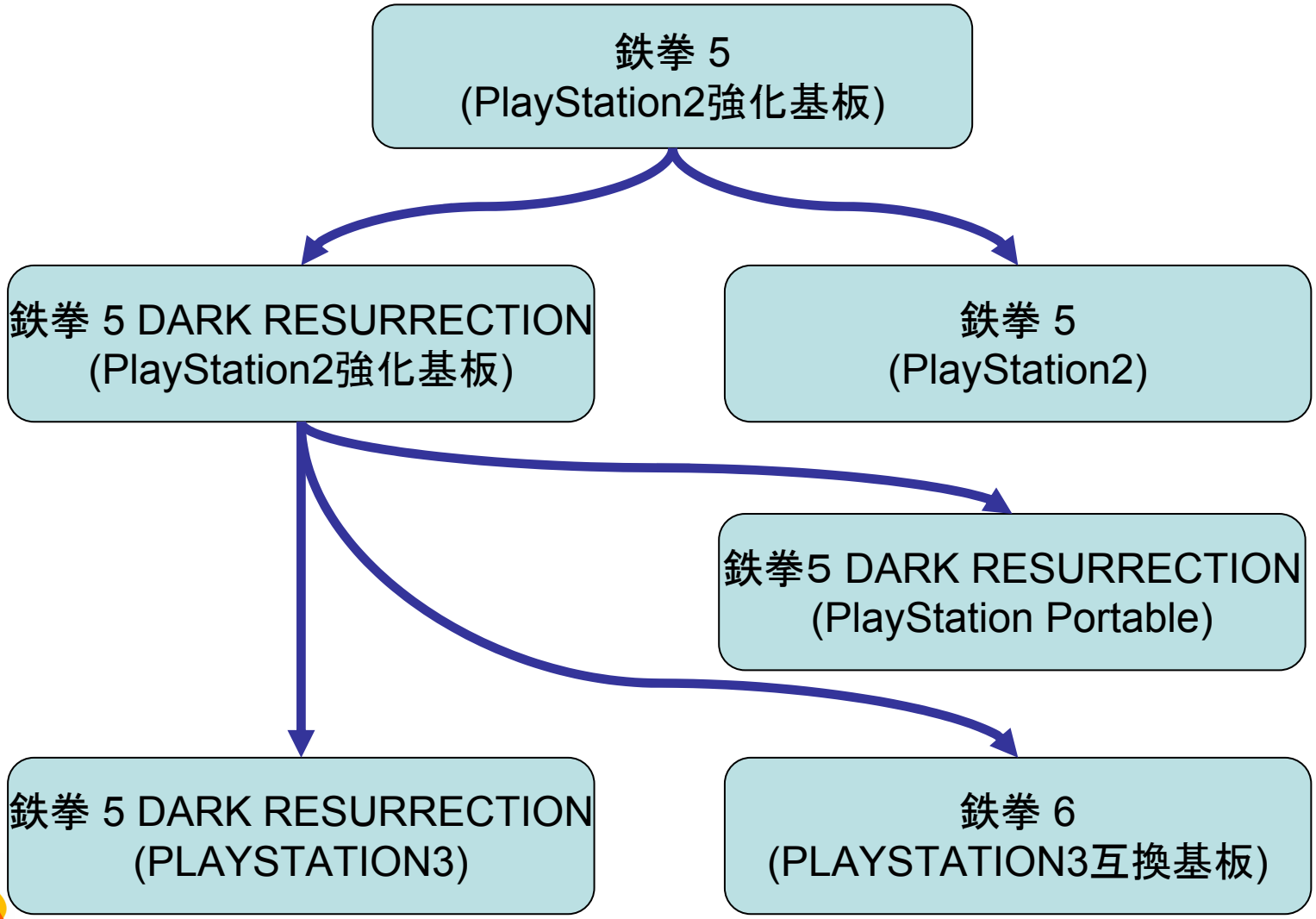
- 性能差
  - 単純な性能比較はできませんが…
  - Gekko 485MHz vs EE 294MHz
- データはグラフィックデザイナの手を借りず、修正なく同じソースから再コンバート
  あるいはプログラマ側での修正

- シビアなチューニング
- Flipper 特化な表現の再現

ケーススタディ 2

- 鉄拳 5

鉄拳の系図


鉄拳 5
(PlayStation2強化基板)
鉄拳 5 DARK RESURRECTION
(PlayStation2強化基板)
鉄拳 5
(PlayStation2)
鉄拳5 DARK RESURRECTION
(PlayStation Portable)
鉄拳 5 DARK RESURRECTION
(PLAYSTATION3)
鉄拳 6
(PLAYSTATION3互換基板)

ケーススタディ 2.1

- 鉄拳 5
- PlayStation2強化基板 → PlayStation2
- 性能差
  - スペックダウン
- プログラマによるチューニング
- グラフィックデザイナによる見栄えを変えずにデータを軽くする

ケーススタディ 2.2

- 鉄拳 5 DARK RESURRECTION
- PlayStation2強化基板 → PlayStationPortable
- 性能差
  - MIPS 294MHz → MIPS 222MHz
- 画面アスペクト比 4:3 → 16:9
- PS2 特化部の移植
  - GS を使ったポストエフェクトなど
  - インラインアセンブリを使用した部分など
- シビアなチューニング

ケーススタディ 2.3

- 鉄拳 5 DARK RESURRECTION
- PlayStation2強化基板 → PLAYSTATION3
- 性能差
  - スペックアップ
- 4:3 → 16:9, PS2 特化部の移植
  - PSP 移植のノウハウが活かされた
- とはいえ作業期間は約 2 ヶ月という短期間

目次

6. マルチコアプロセッサのあれこれ

マルチコアプロセッサ

- 他社さんの選択
  - フレームワーク層まで取り込み、ゲームエンジンとしてマルチコアにジョブを振っていく
  - ミドルウェアとしてはマルチコアを使わない
    ゲーム側で自由に使ってください
  - etc…

マルチコアプロセッサ

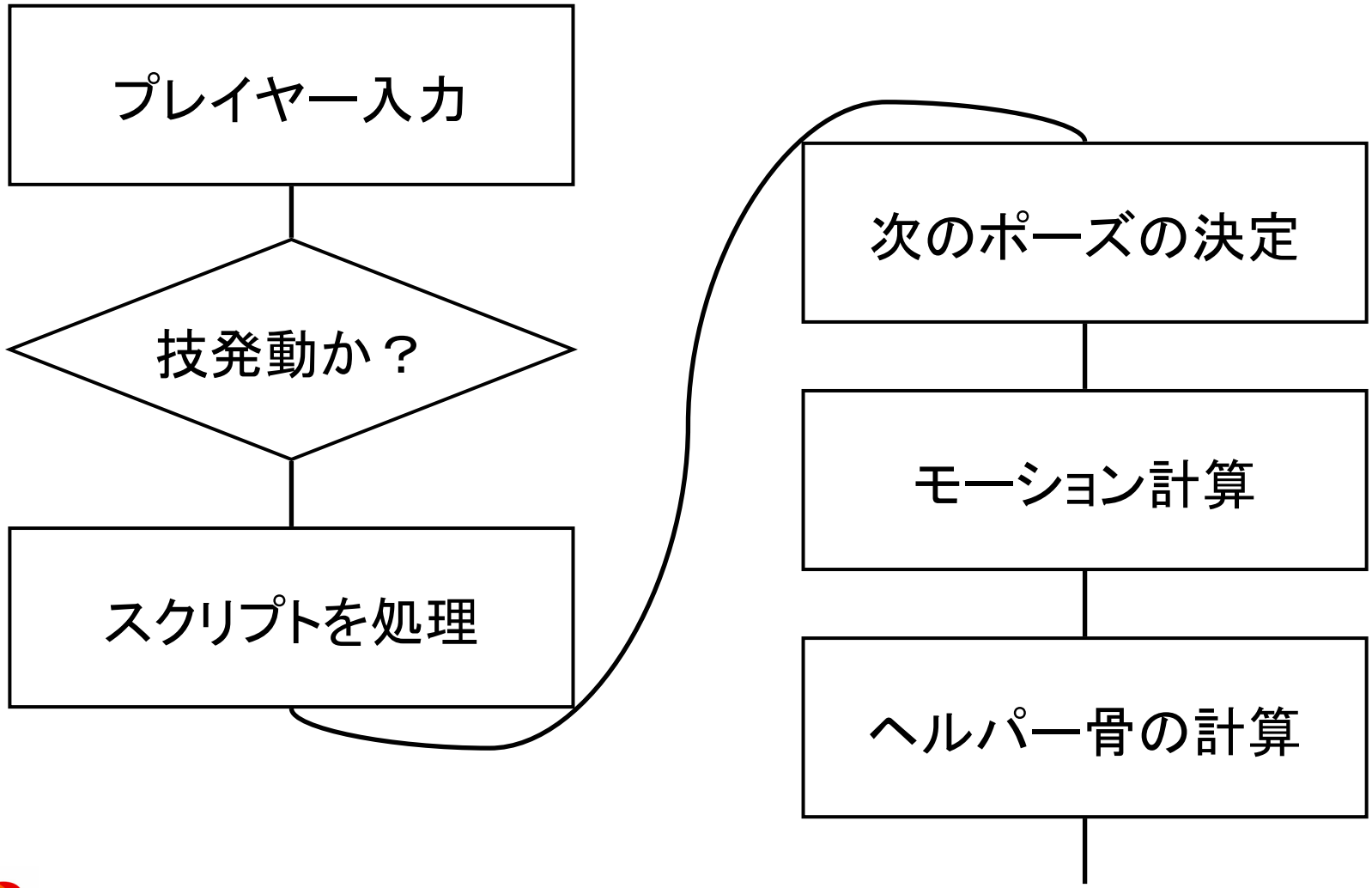
- 当ライブラリの選択
  - 当ライブラリがなるべくマルチコアにジョブを振っていく
  - ゲーム制作側は、いままでのようにシングルコア、シングルスレッドのように制作してかまいません

並列化可能？

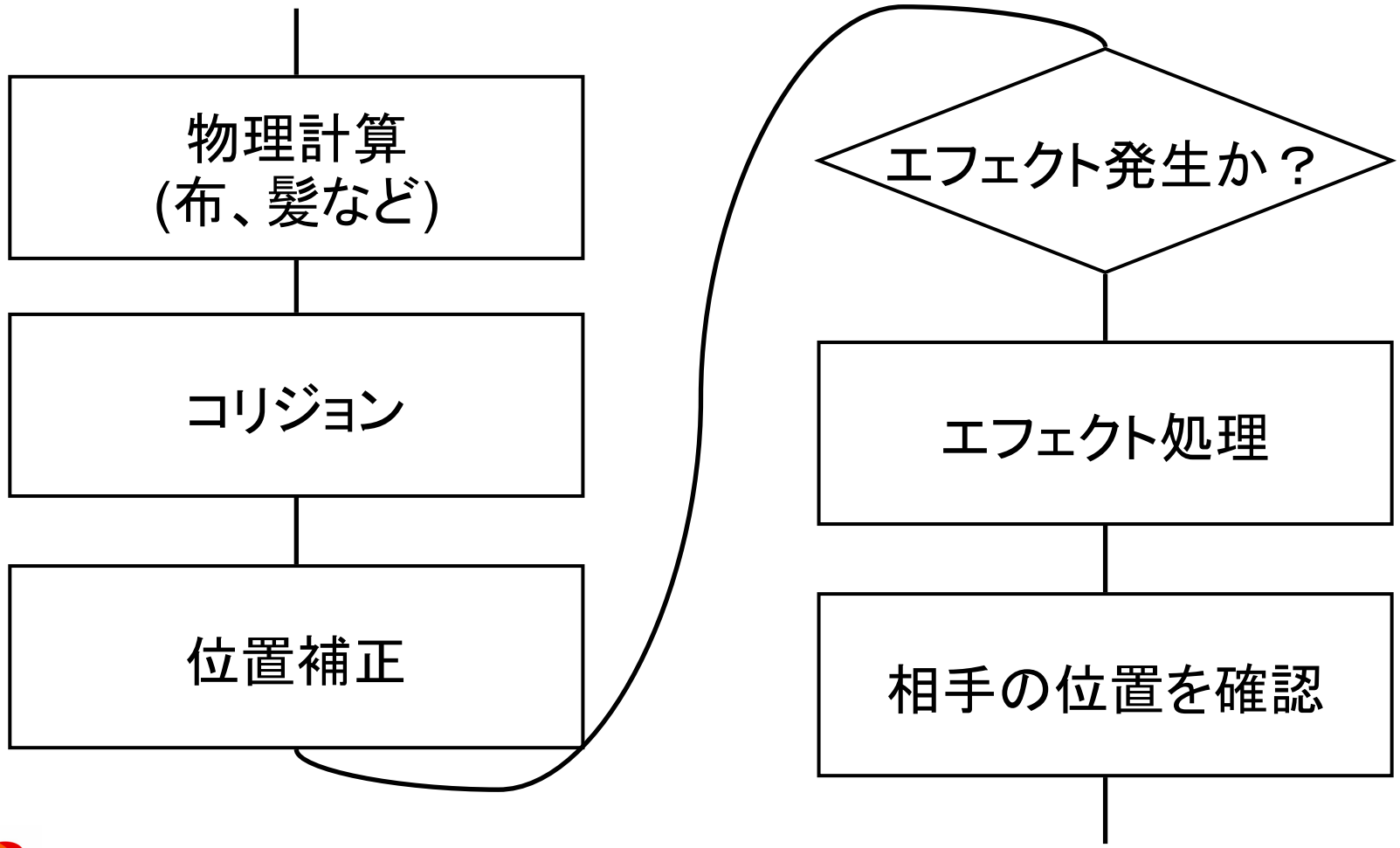
- ゲームジョブを並列化できるか？

- ゲームによっては並列化が難しいものも少なくない

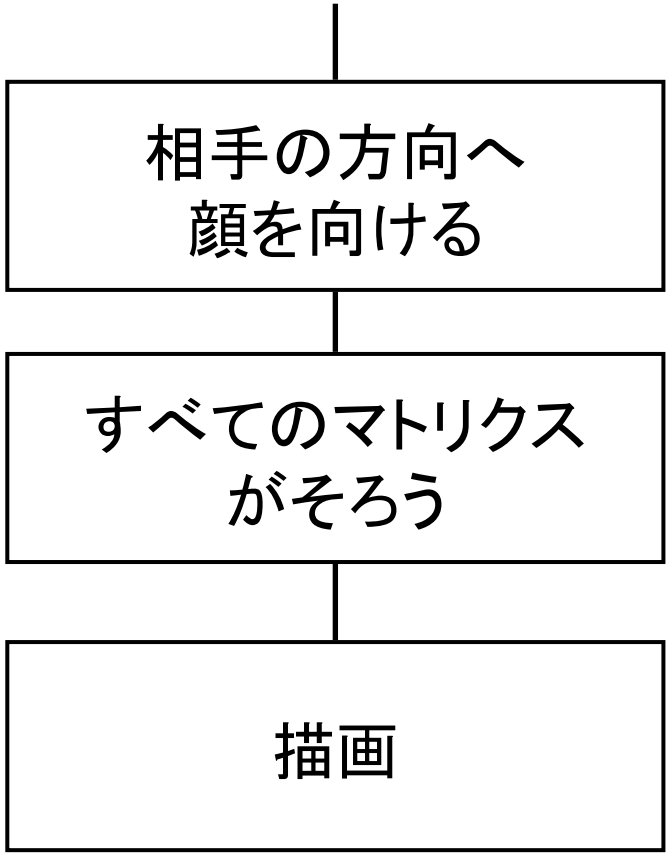
とあるゲームの例



とあるゲームの例 (続き)



とあるゲームの例 (続き)



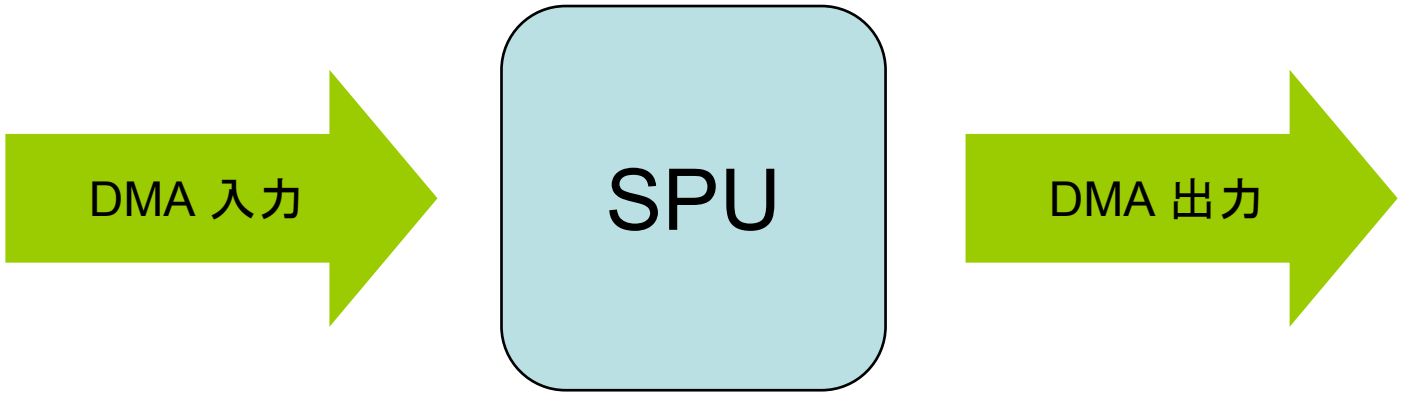
- ほぼ総てがシーケンシャル
  - 前の処理の結果が必要
- 各処理の中では並列処理が可能だが…
- 「並列処理→バリアで総ての終了を待つ」を繰り返すことになる

並列化は良く考えて

- ジョブを振っていく方法が簡単
- ただし、前後に依存関係があるときは注意が必要
- キャッシュミス, キャッシュスラッシングが起こらないように気をつける
- 頂点処理を振るのは良いアイデアのひとつ (GDC2007)

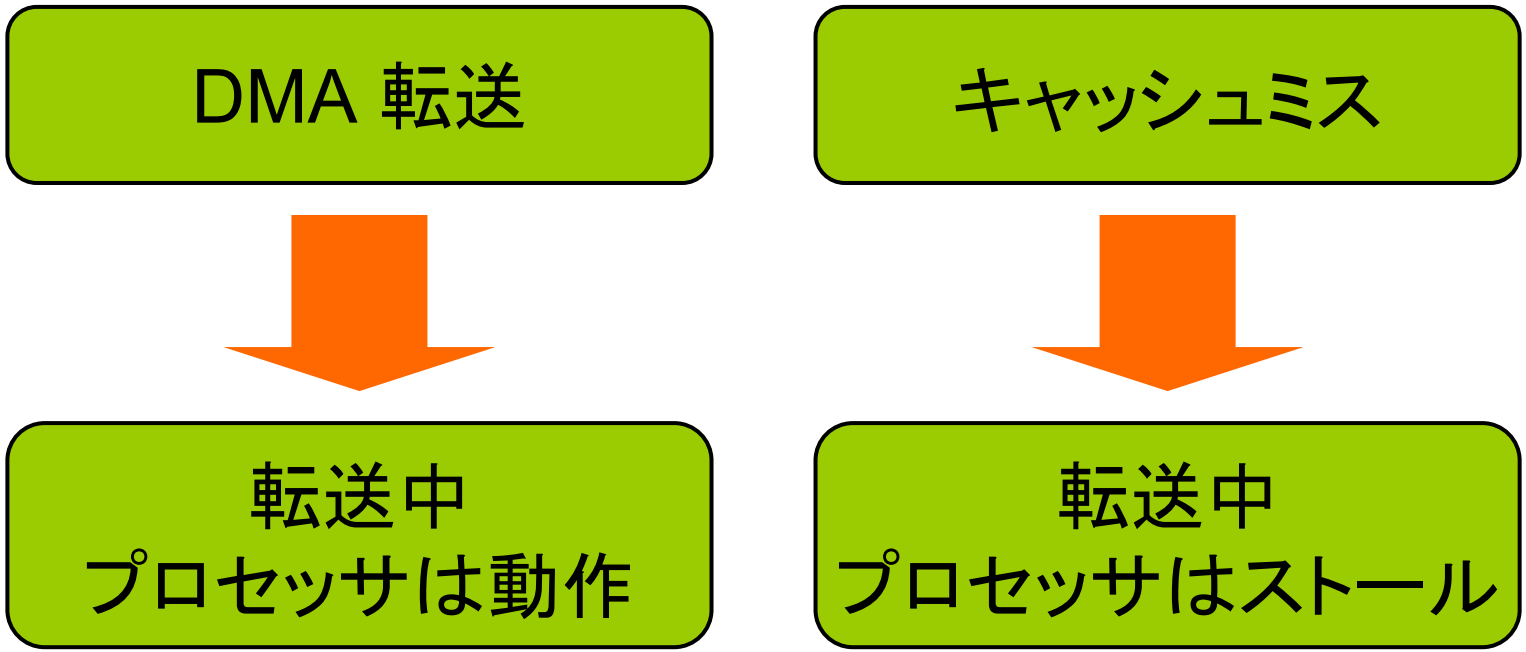
SPU

- VMX に似たベクトルエンジン
- LS (Local Storage) という高速メモリ
- MFC (Memory Flow Controller) 搭載
  MMU に対応した DMA 転送
- ストリーミング向き



SPU (続き)

- カリカリにチューニングして DMA 転送を絶え間なく行うと…
- PPU の足を引っ張る可能性も…



SPU (続き)

- ベクトル特化とはいえ、C, C++ でプログラム可能な汎用プロセッサ
- 高速なメモリ
- PPU は力不足
- SPU に振っていく必要あり

- ベクトルにこだわらず汎用的に使うのも手

グローバル変数

- 危険!!!
- 昔から、その危険性は指摘されてきた
- マルチコアになったらなおさら

- Singleton パターンを使う

Singleton パターン

- 教科書に載っている例

```
class Singleton {
    static Singleton *m_pTheInstance;
public:
    static Singleton *GetInstance();
};
```

```
Singleton *Singleton::m_pTheInstance = 0;

Singleton *Singleton::GetInstance()
{
    if ( m_pTheInstance == 0 ) {
        m_pTheInstance = new Singleton;
    }
    return m_pTheInstance;
}
```

この場合の欠点

- Singleton::GetInstance() が重い
  - Singleton::m_pTheInstance をアクセス
    - D キャッシュミス!!! 小さなアクセス!!!
  - Singleton::m_pTheInstance が 0 か判定
    - 分岐予測ミスの可能性!!!
      - パイプラインフラッシュ!!!
      - I キャッシュミス!!!
  - operator new の実行
    - スレッドセーフでない
    - ユーザは勝手に malloc されるのを嫌う
      - operator new をオーバーロードする手もあるが

Singleton パターン 2

- class Singleton の実体を持たせる

```
class Singleton {
    static Singleton m_theInstance;
public:
    static Singleton *GetInstance()
    {
        return &m_theInstance;
    }
};
```

```
Singleton Singleton::m_theInstance;
```

この場合の利点

- Singleton::GetInstance() が高速
  - 通常、1 命令か 2 命令程度
  - D キャッシュミスを起こさず、this を取得
- operator new が実行されない
  - ユーザに怒られずにすむ
  - 都合の良い位置にインスタンスを置くことができる

この場合の欠点

- グローバルコンストラクタ、グローバルデストラクタが実行される可能性
  - グローバルコンストラクタ、グローバルデストラクタは非常に危険!!!
  - 実行されるタイミングが処理系によって違う
  - 実行されない可能性もある
- operator new が実行されないのは危険!!!
  - 仮想関数テーブルポインタ、RTTI ポインタはコンストラクタの中で設定される

Singleton パターン 3

- placement new/delete を使う
  - 明示的に初期化・終了関数を呼び出す

```
class Singleton {
    static Singleton m_theInstance;
    static void *operator new( size_t, void *pPlace ) { return pPlace; }
    static void operator delete( void *, void * ) {}
    static void operator delete( void * ) {}
    static void operator delete( void *, size_t ) {}
public:
    static Singleton *CreateInstance() {
        return new (&m_theInstance) Singleton;
    }
    static void DestroyInstance() {
        delete (&m_theInstance);
    }
};
```

さらに安全に

- コンストラクタ／デストラクタを空にする
  - グローバルコンストラクタ／デストラクタ対策
- そのかわり初期化・終了関数を作る

```
public:
    static Singleton *CreateInstance();
    static void DestroyInstance();

protected:
    Singleton() {}
    ~Singleton() {}

    void Initialize();
    void Destroy();
```

さらに安全に (続き)


```
Singleton *Singleton::CreateInstance()
{
    Singleton *p = new (&m_theInstance) Singleton;
    p->Initialize();
    return p;
}
void Singleton::DestroyInstance()
{
    m_theInstance.Destroy();
    delete (&m_theInstance);
}
void Singleton::Initialize()
{
    //初期化処理.
}
void Singleton::Destroy()
{
    //終了処理.
}
```

目次

7. 最後に

次世代機版採用タイトル

ご清聴ありがとうございました

- 質問はございませんか?