TOSHIBA

Cell グラフィックスプログラミング

Syoyo Fujita
Takahiro Saito

2006/09/01

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

Agenda

- Motivation
- Cell プロセッサの仕組み
- Cell ポリゴンレンダリング
- Cell リアルタイムレイトレーシング
- 参考文献

Agenda

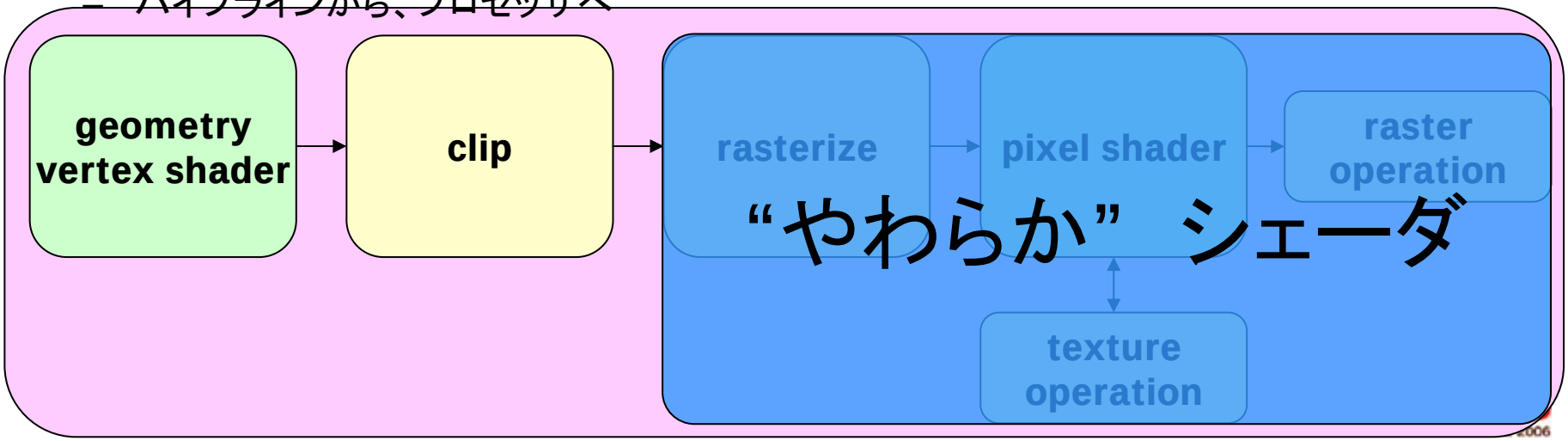
- Motivation
- Cell プロセッサの仕組み
- Cell ポリゴンレンダリング
- Cell リアルタイムレイトレーシング
- 参考文献

グラフィックスのアプローチ

- ポリゴンベース
  - 固定パイプライン
  - プログラマブルシェーダ
  - プロセッサベースでの、よりやわらかいパイプライン
    - ただし、今回はそのベースとなる従来のパイプラインの実装を解説します。
- ピクセルベース
  - レイトレーシング
    - 近年、インタラクティブレイトレーシングの学会が開かれたり、動的シーンのレイトレ手法、レイトレチップが提案され始めているなど**大きな発展**が見られる

ポリゴンレンダリングのこれから

- より高度な表現へ…
  - モーションブラー、ブラーフィルタ、グレアフィルタ、etc…
  - 隣接ピクセル情報の利用
- 現在のシェーダアーキティクチャでは…
  - ピクセル独立
  - 周りのピクセルを利用とすると、テクスチャに書いて、それをリードして、またテクスチャに書いて…
    - 帯域の無駄
    - GPGPU 的なアプローチには制約が多い(書き込み先に制約があるなど)
- “やわらか” シェーダ
  - ピクセルレベルの処理は統合され、もっとやわらかいものに
  - まとまったピクセルを一度に処理
  - パイプラインから、プロセッサへ



レイトレーシングの最近

- リアルタイムレイトレーシングが見えてきた
  - アルゴリズムの発展が近年大きい
  - 動的シーンのリアルタイムレイトレーシングが提案されてきている。

Agenda

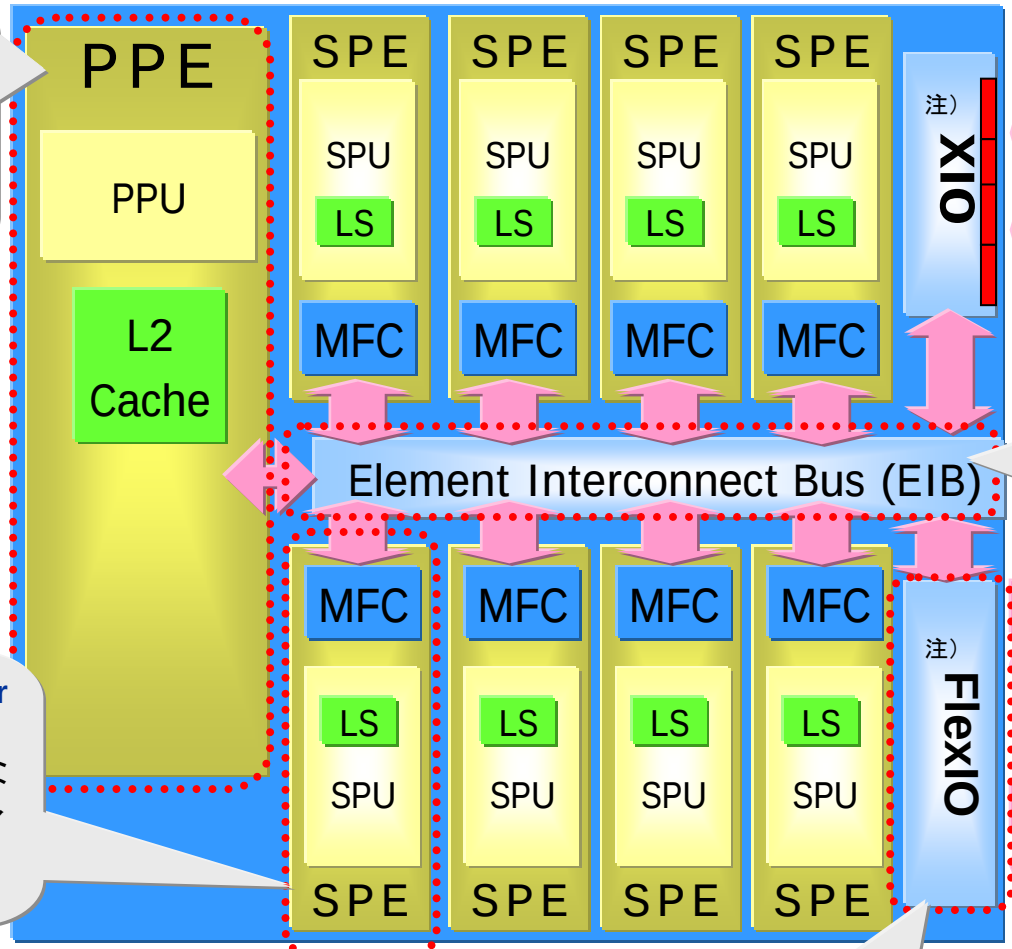
- Motivation
- Cell プロセッサの仕組み
- Cell ポリゴンレンダリング
- Cell リアルタイムレイトレーシング
- 参考文献

Cellの基本構成

TOSHIBA

Power Processor Element (PPE)
汎用処理に向いたPowerPCアーキテクチャのCPUコア

Synergistic Processor Element (SPE)
SIMD型のシンプルなプロセッサコア（メディア処理に向いた）



高速メモリ（XDR）
XDR I/F 2Channel

注）
XDR DRAM
25.6GB/s

Element Interconnect Bus (EIB)
高速広帯域な内部リングバスでCell内部の各ブロックを接続

Cell / GPU 等
76.8GB/s
Super Companion Chip
I/O

PPE : Power Processor Element
SPE: Synergistic Processor Element
LS : Local Storage

Flex IO
広帯域2Chanelの外部インターフェイス

注）Rambus社の商標です

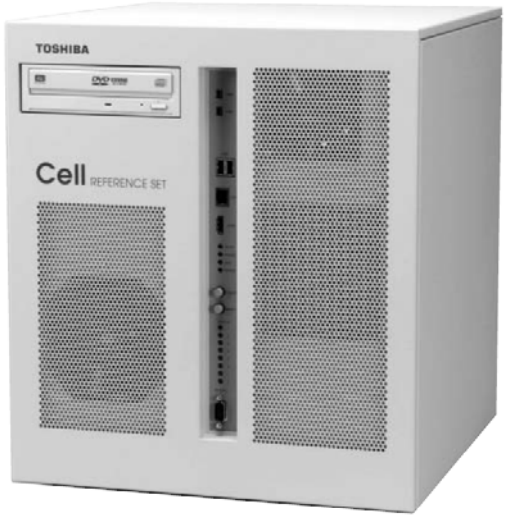
CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

- 周波数
  - 2.4 – 4.0 GHz(*)
- SPE
  - SIMD 演算コア。256 KB の作業メモリ(LS) 。8 個装備。
  - 7 個も悪くない。
    - **“素数”**は1と自分でしか割ることのできない孤独な数字… わたしに勇気を与えてくれる。[Pucci 2011]
- PPE
  - 制御コア。
- XDR
  - メモリ。
- EIB
  - 内部バス。

(*) 4.0 GHz は実験室データ

実装開発環境

- 東芝製 Cell リファレンスセット
  - OS: linux
  - コンパイラ: gcc

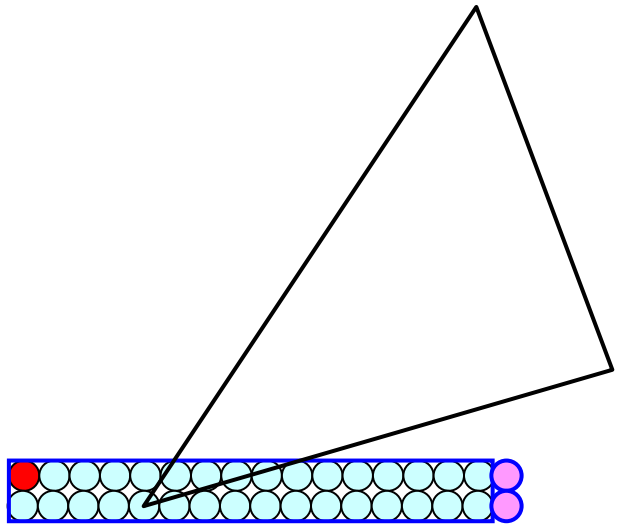


Agenda

- Motivation
- Cell プロセッサの仕組み
- Cell ポリゴンレンダリング
- Cell リアルタイムレイトレーシング
- 参考文献

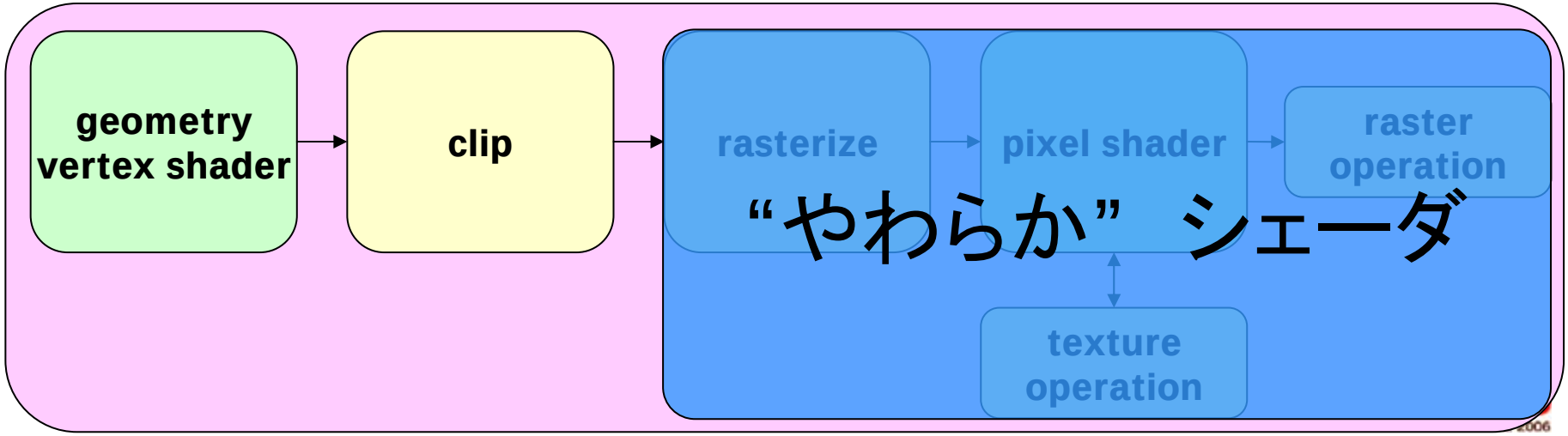
Cell ポリゴンレンダリング

- 概要
- ラスタライズ
- シェーディング
- 結果と統計
- まとめ



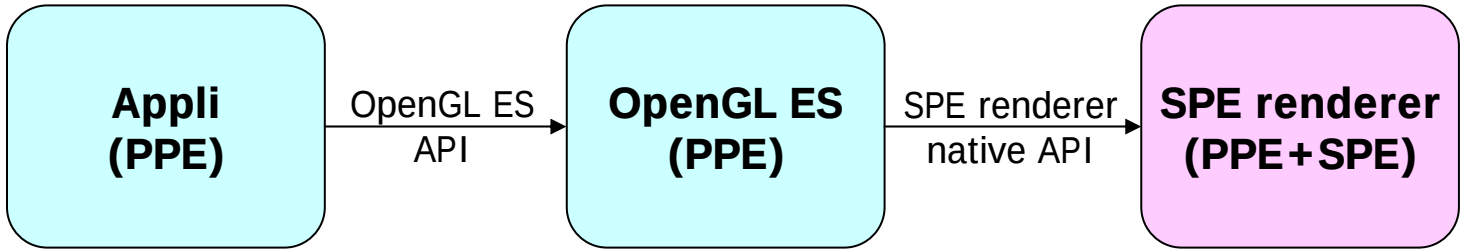
ポリゴンレンダリングのこれから

- より高度な表現へ…
  - モーションブラー、ブラーフィルタ、グレアフィルタ、etc…
  - 隣接ピクセル情報の利用
- 現在のシェーダアーキティクチャでは…
  - ピクセル独立
  - 周りのピクセルを利用とすると、テクスチャに書いて、リードして、またテクスチャに書いて…
    - 帯域の無駄
    - GPGPU 的なアプローチには制約が多い(書き込み先に制約があるなど)
- “やわらか” シェーダ
  - ピクセルレベルの処理は統合され、もっとやわらかいものに
  - まとまったピクセルを一度に処理
  - パイプラインから、プロセッサへ

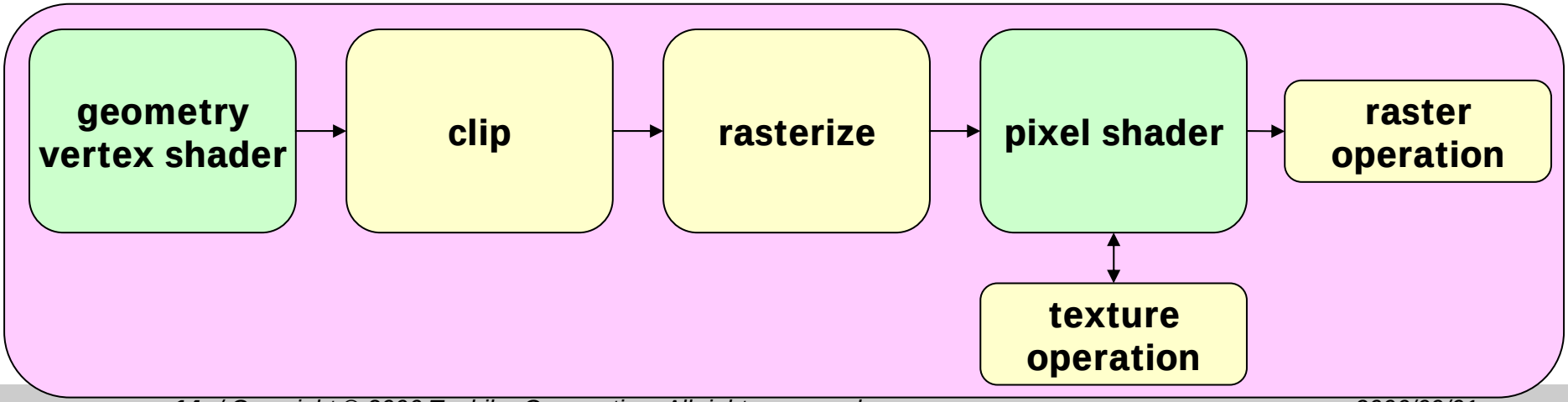


今回は、基本的なパイプラインを実装

- SPE SWで3Dグラフィックスの描画を行う
- PPE で **OpenGL ES** のwrapperを実装

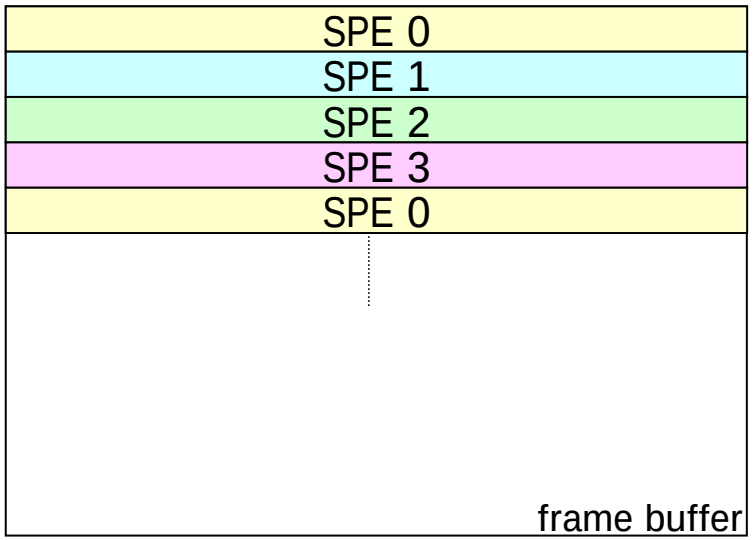


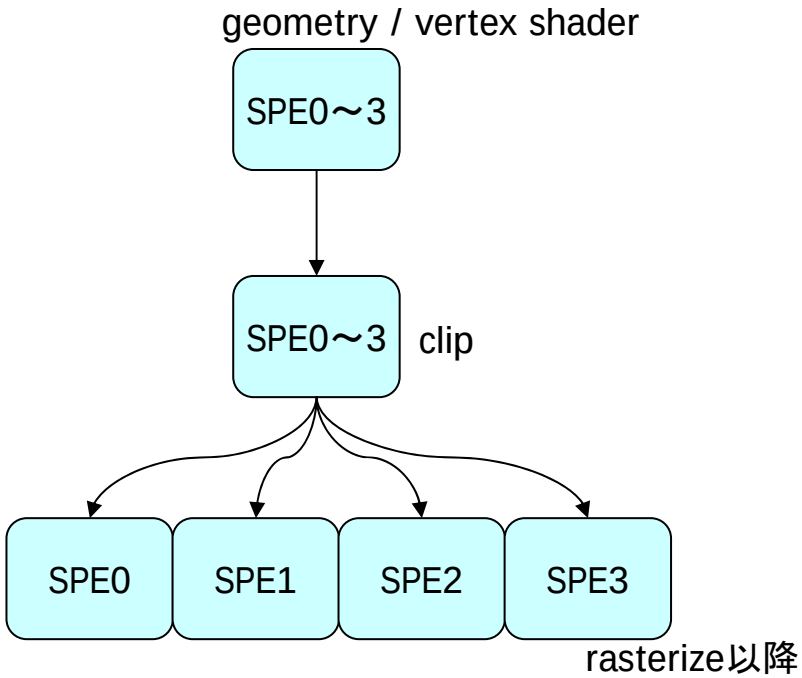
- 以下の処理ブロックに分割
  - 緑：user programmable
  - 黄：fixed function



SPEへの処理の割り当て(1/2)

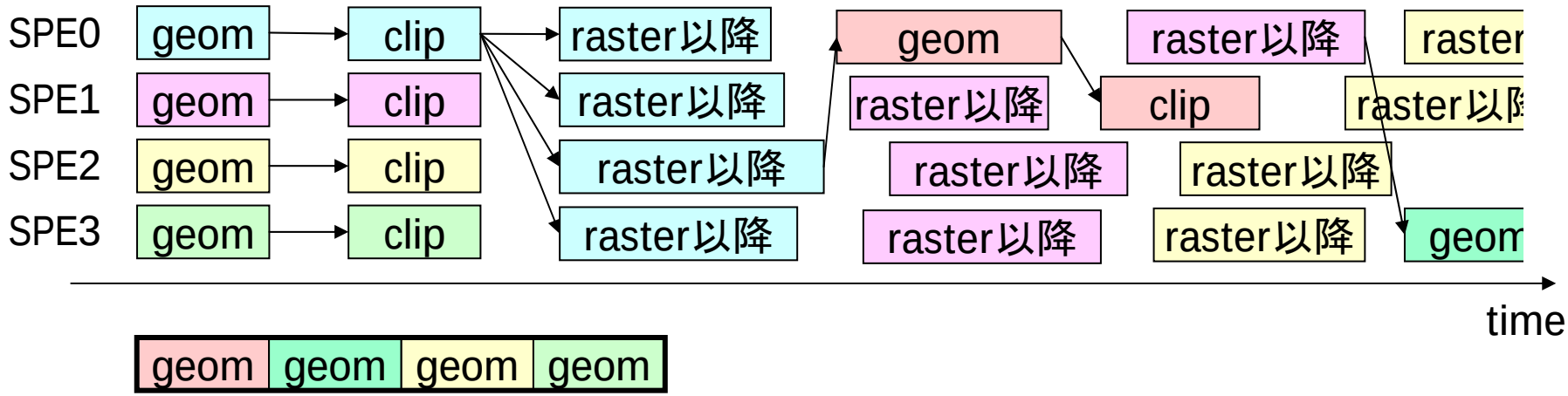
- 負荷の重いラスタライズ以降を、複数個の SPE で担当
  - アドレス計算などを簡単にするために、2のべき乗個のSPE (=4SPE) に処理を割り当て
    - 描画画面の位置で担当SPEを静的に割り当てる
    - rasterizeのlineごとに分割
- geometry/vertex shader は、上記の処理の合間に空いてるSPEにて処理





SPEへの処理の割り当て(2/2)

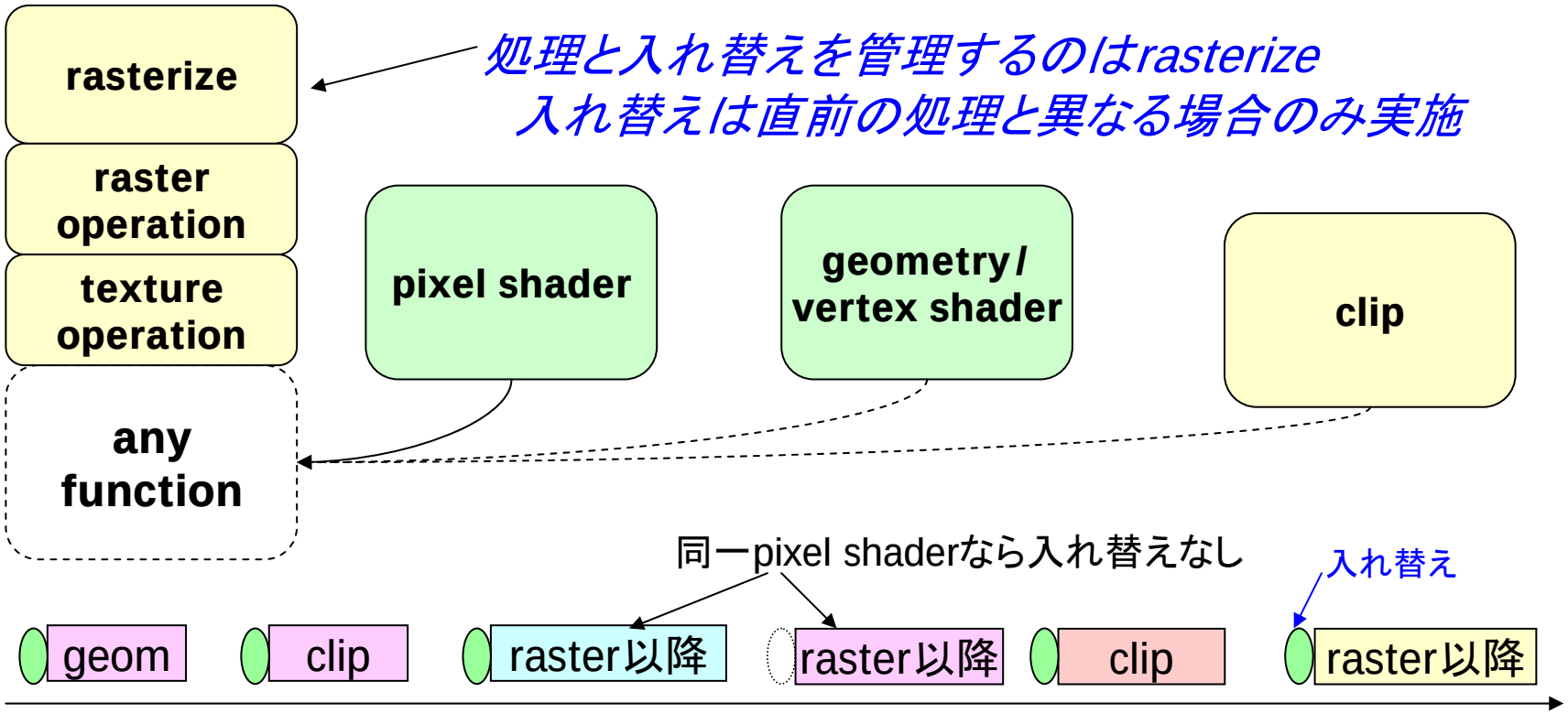
- 動作イメージ



1. geometry/clip処理は依存関係がないので、次々に処理
   - geometry用のバッファがある限り優先して処理
2. geometry用のバッファがいっぱいになったら、Rasterize以降の処理
   - 全SPEで分担して処理し、全SPEで処理が完了したら、geometry用のバッファを開放
3. 次のgeometryの処理へ

SPE Local Storage

- 全機能をLocal Storage(LS)に入れるのは困難
  - 常駐する機能
    - rasterize, raster operation, texture operation
  - 非常駐の機能で、使用する際に入れ替える
    - clip, pixel shader, geometry/vertex shader

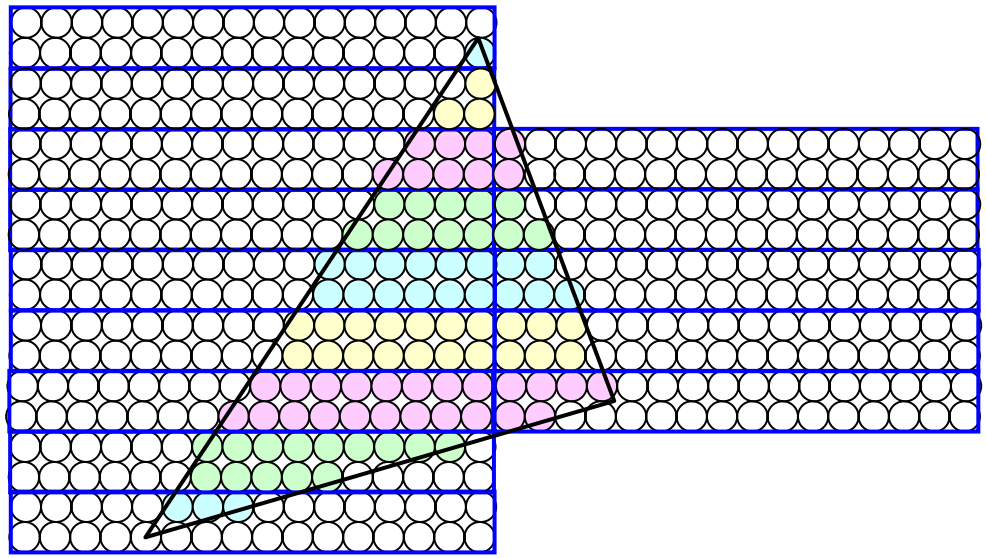


Cell ポリゴンレンダリング

- 概要
- ラスタライズ
- シェーディング
- 結果と統計
- まとめ

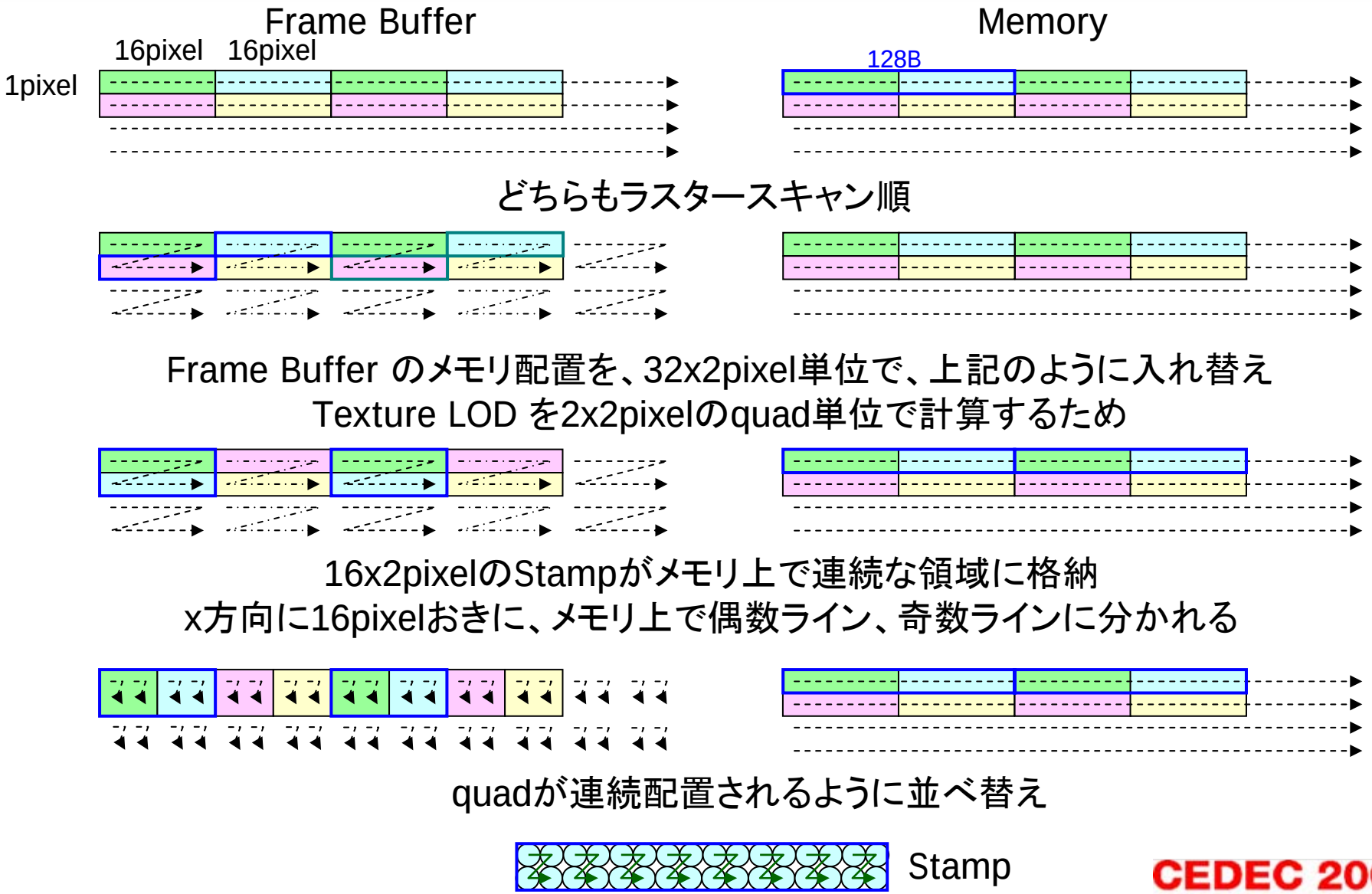
各SPEでのRasterize

DDA



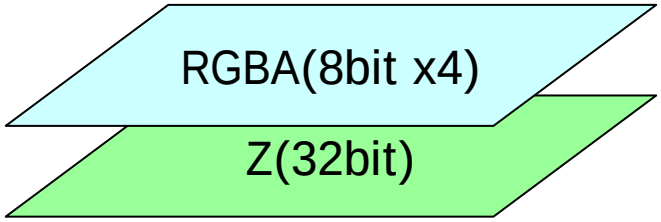
- 16x2 pixel ごとのStampを単位として処理
  - DMAは128B単位が効率よい → 128B単位で処理
  - 128B = 32pixel x 8bit x 4component
  - → 16x2pixel を処理単位

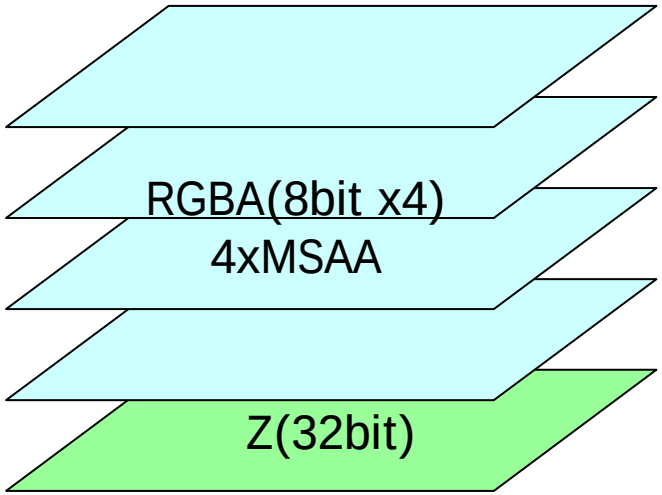
Frame Buffer のメモリ配置

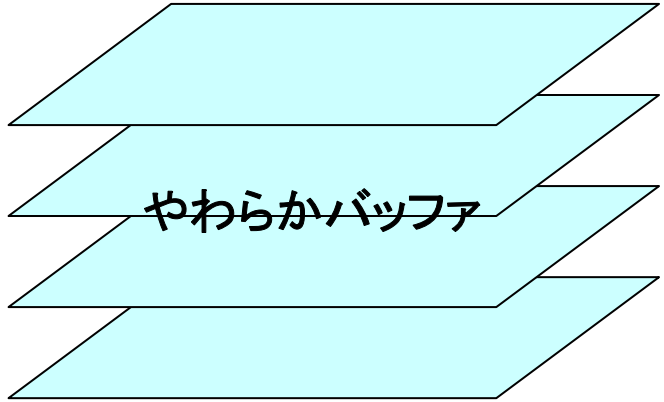


各種buffer

- ソフトウェアゆえ、任意の Layer を定義可能
- 複数Layerの使用で、HDR、AAにも対応





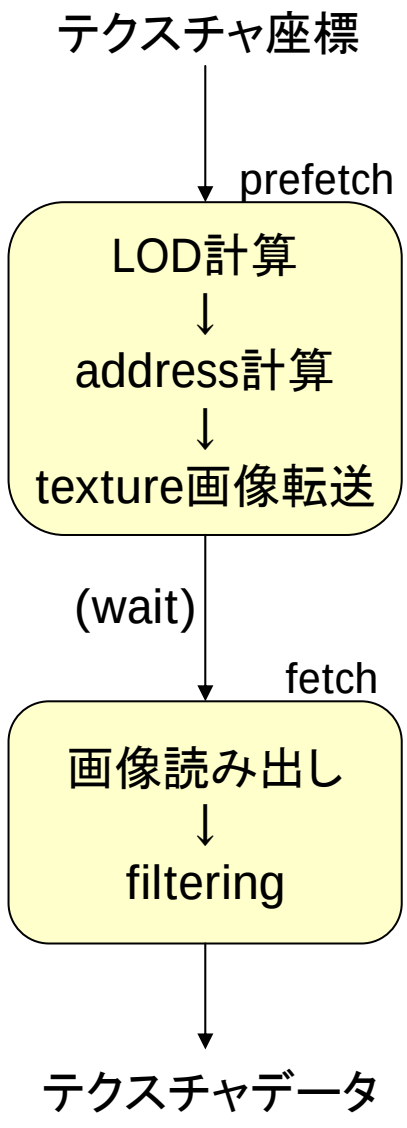


Cell ポリゴンレンダリング

- 概要
- ラスタライズ
- シェーディング
- 結果と統計
- まとめ

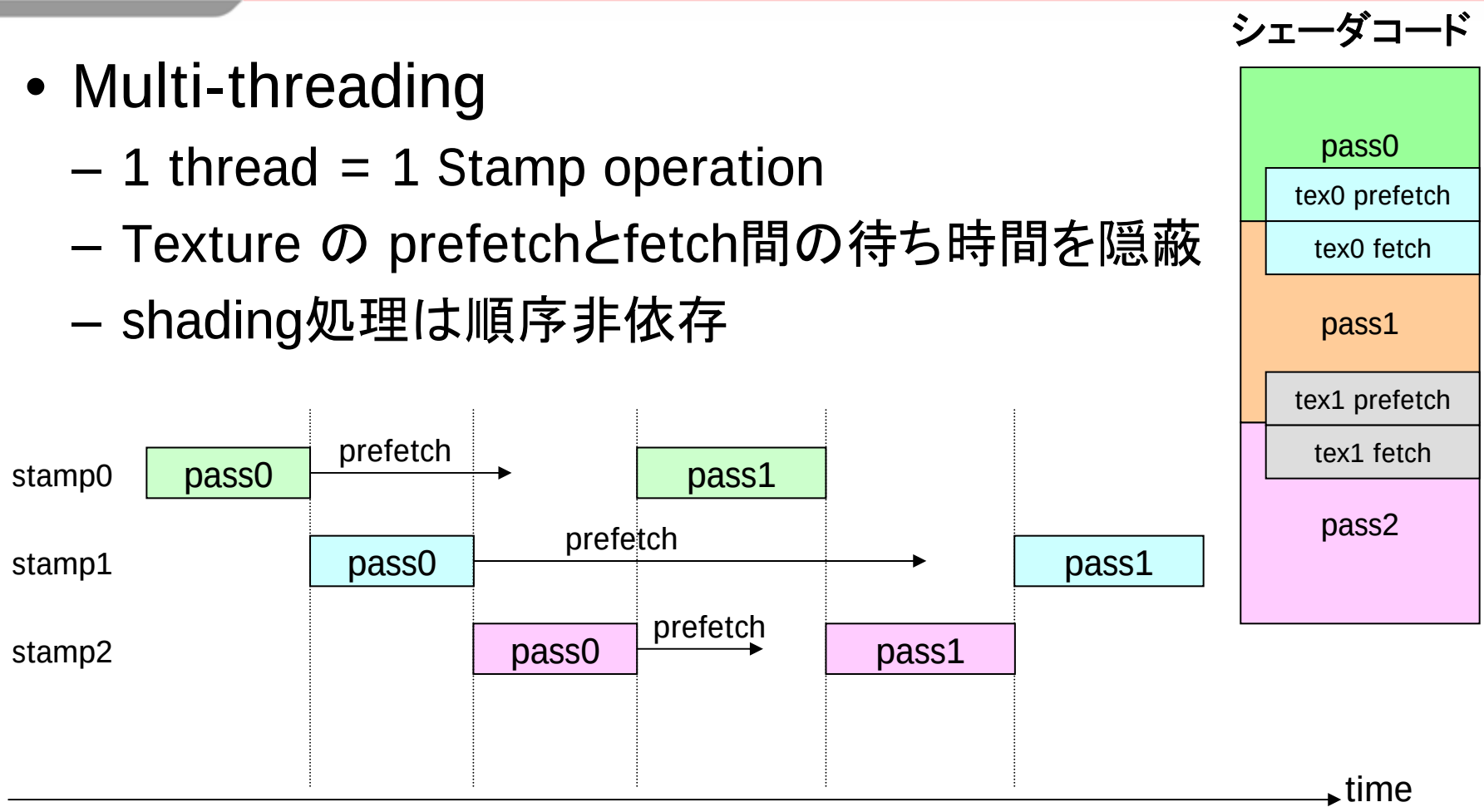
Texture fetch

- テクスチャ転送のレイテンシを隠蔽するために、以下の2つの処理に分割
- **prefetch**
  - 座標からアドレスを計算し、テクスチャイメージの一部をLSに転送
    - 転送待ちの間に、他のstampの処理を実施
  - 出力は、(u_{offset}, v_{offset}, lod, filter mode)
    - offset はテクスチャの転送単位のブロック内での座標。fetch時に使用
      - ブロックは8x4pixel (frame buffer で 16x2pixelだったように128B単位)
    - lod はLOD値の計算結果。trilinearの補間係数に使用
    - filter mode はmin/magの選択結果。fetchでのfilteringで使用
- **fetch**
  - LS内のテクスチャイメージを補間計算して出力
    - prefetchの出力パラメータを使用
- prefetchとfetchの間は、LSへの転送が終了するまで待つ



Pixel Shading

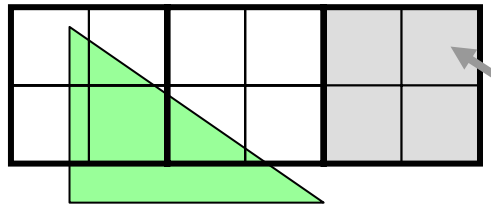
- Multi-threading
  - 1 thread = 1 Stamp operation
  - Texture の prefetchとfetch間の待ち時間を隠蔽
  - shading処理は順序非依存



prefetch が終わったthread(Stamp)から処理

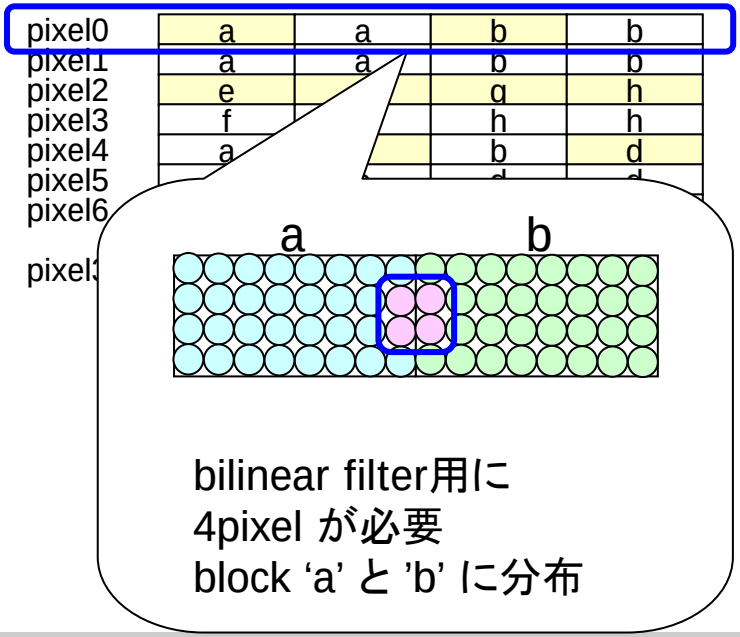
prefetch

1. stamp内の有効な全pixelが必要とするテクスチャイメージのアドレスを計算 (quad単位で有効/無効の判定)
2. 同一アドレスをマージして、転送回数を削減
   - 各fetchで必要なLS上の場所を生成
3. 必要なデータをまとめて転送

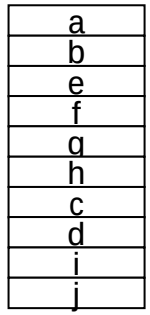


無効な quad

1) アドレス計算



2) アドレスマージ



3) DMA転送

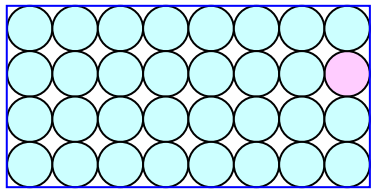
SW cache?
tag checkによる性能低下
最悪のケースに対応するcacheは高コスト

fetch以降

- fetch
  - prefetchでの転送が終了するまでwait
  - prefetch出力のoffsetとfilter modeから計算し出力

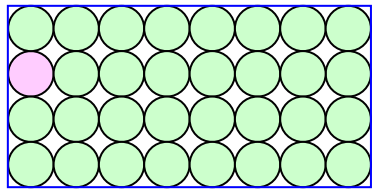
ex) bilinear filtering

(U0,V0)



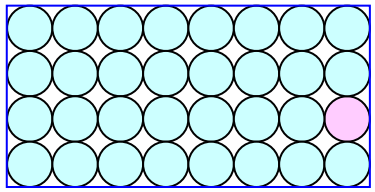
offset
=(7,1)

(U0+8,V0)



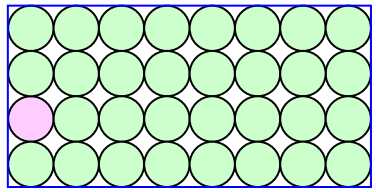
offset
=(0,1)

(U0,V0)



offset
=(7,2)

(U0+8,V0)

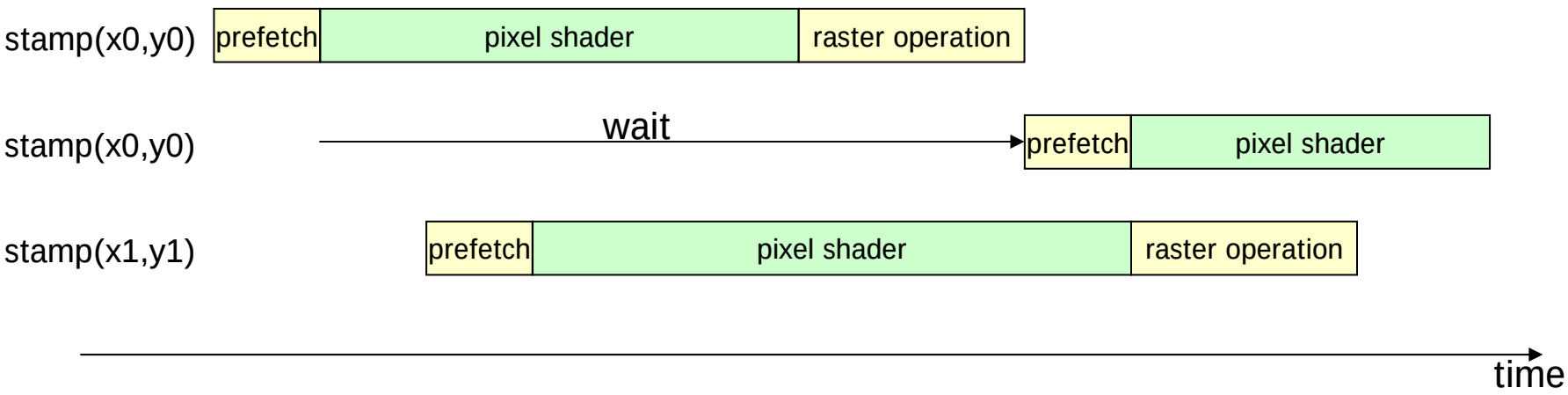


offset
=(0,2)

- trilinearをやるには？
  - tmp0 = prefetch (texcoord, pixel_valid, tex_id, lod_{off}=0.0)
  - col_L = fetch(tmp0, pixel_valid, tex_id)
  - tmp1 = prefetch (texcoord, pixel_valid, tex_id, lod_{off}=1.0)
  - col_H = fetch(tmp1, pixel_valid, tex_id)
  - blend = fraction(tmp0.lod)
  - col = col_L(1.0-blend) + colH*blend

Raster Operation

- pixel shader実行前にframe bufferのprefetch
  - 同一位置のstampの実行は処理終了までwait
  - 同じ stamp 位置では、フレームバッファデータの排他処理が必要なため

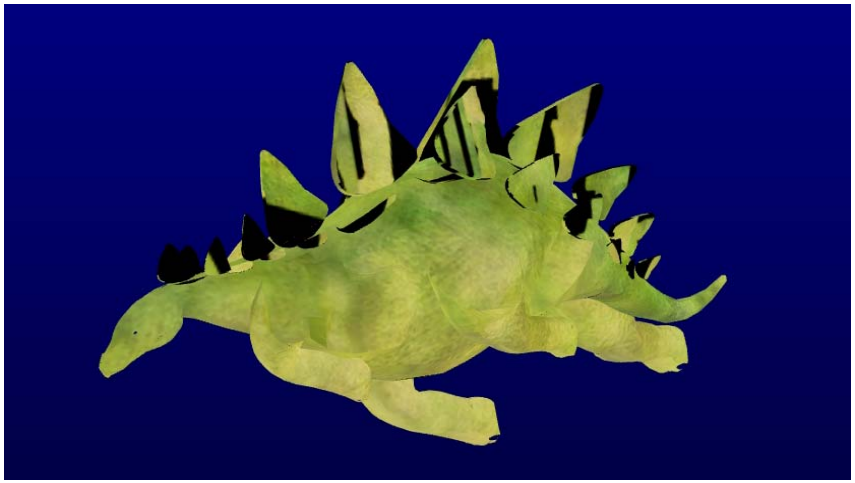


Cell ポリゴンレンダリング

- 概要
- ラスタライズ
- テクスチャリング
- 結果と統計
- まとめ

サンプル

Stegosaur



13380 Triangles

	max	min
zoom in (870Kpix)	21 Mpix/s 18 fps	12 Mpix/s 14 fps
middle (280Kpix)	6.0 Mpix/s 24 fps	3.1 Mpix/s 20 fps
zoom out (16Kpix)	0.48 Mpix/s 33 fps	0.26 Mpix/s 29 fps

画面サイズ: 1280x720
shader: 1 texture

zoom in/middle/zoom out
視点の条件: ズームイン、中間、ズームアウトしたとき

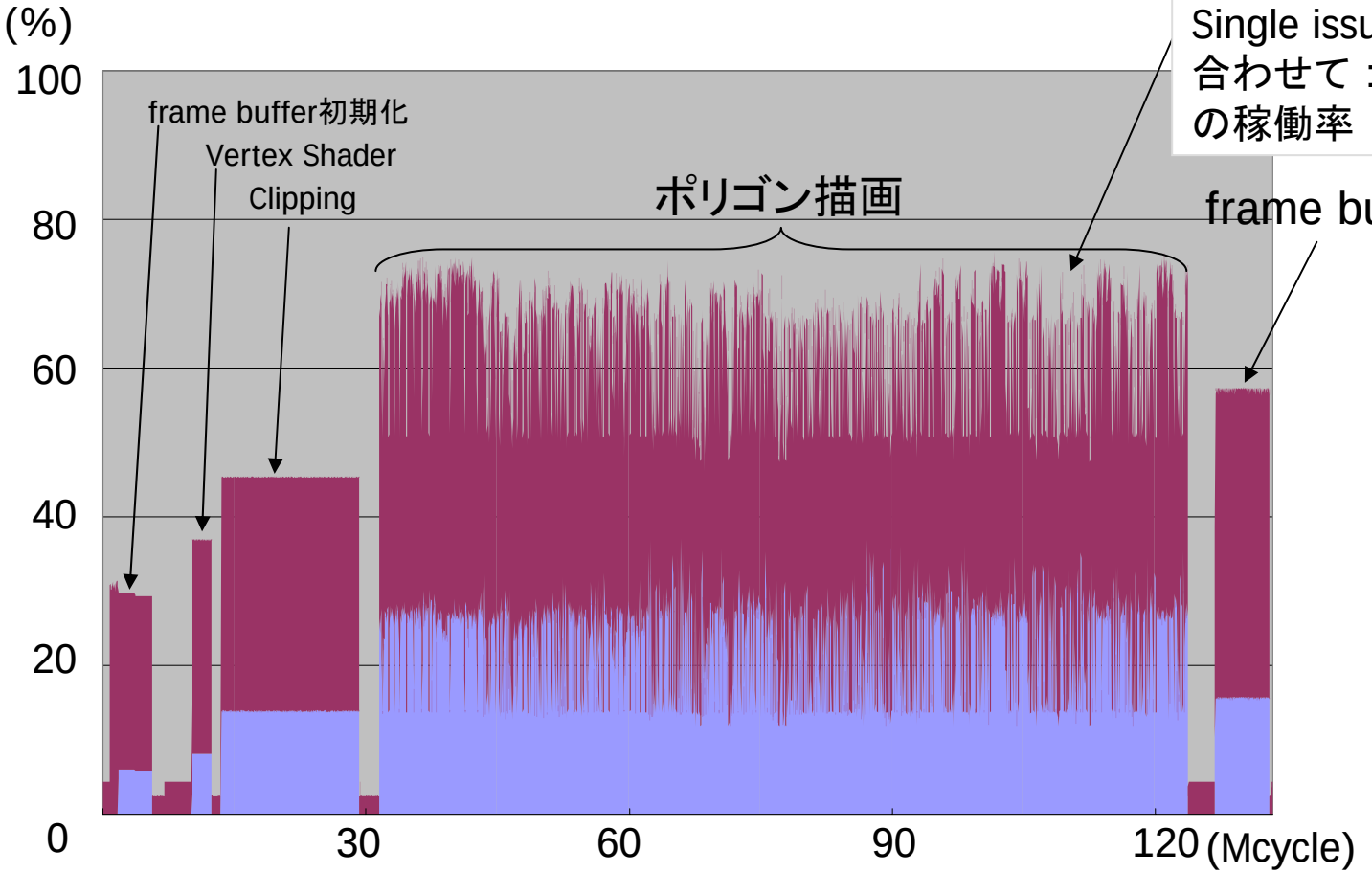
Kpix: 描画されたおおよその pixel 数
Mpix/s: 秒間で実際に処理したピクセル数
max/min: 視点を変えてアニメーションさせたときの最大・最小

※ 性能は、4 SPE @ 2.8GHz での実測値

性能解析：Cell performance monitor

TOSHIBA

Dual issue：15～30%
Single issue：35～40%
合わせて：50～70%
の稼働率



Cell ポリゴンレンダリング

- 概要
- ラスタライズ
- テクスチャリング
- 結果と統計
- まとめ

まとめとこれから

- 基礎的なパイプラインの実装方法について解説しました。
  - スタート地点としての従来型パイプラインの実装
- 今後の課題
  - これをベースに、やわらかシェーダのあり方について考えていきたい
    - shader + texture + rop ひとまとめでの効率的な処理方法

Agenda

- Motivation
- Cell プロセッサの仕組み
- Cell ポリゴンレンダリング
- Cell リアルタイムレイトレーシング
- 参考文献

Cell レイトレーシング

- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめと改善案

レイトレーシング

- リアルタイムレイトレーシングが見えてきた
  - アルゴリズムの発展が近年大きい
  - 動的シーンも扱えるようになってきている。
  - 現在は、レイキャスティング、一次レイ+シャドウレベルがリアルタイムで処理できて来ている。
- レイトレーシングのアルゴリズム
  - 並列処理に適したアルゴリズム
  - Cell の複数 SPE の恩恵を受けられる
- 基礎となるリアルタイム向けレイトレーシングの Cell への実装方法を解説します。

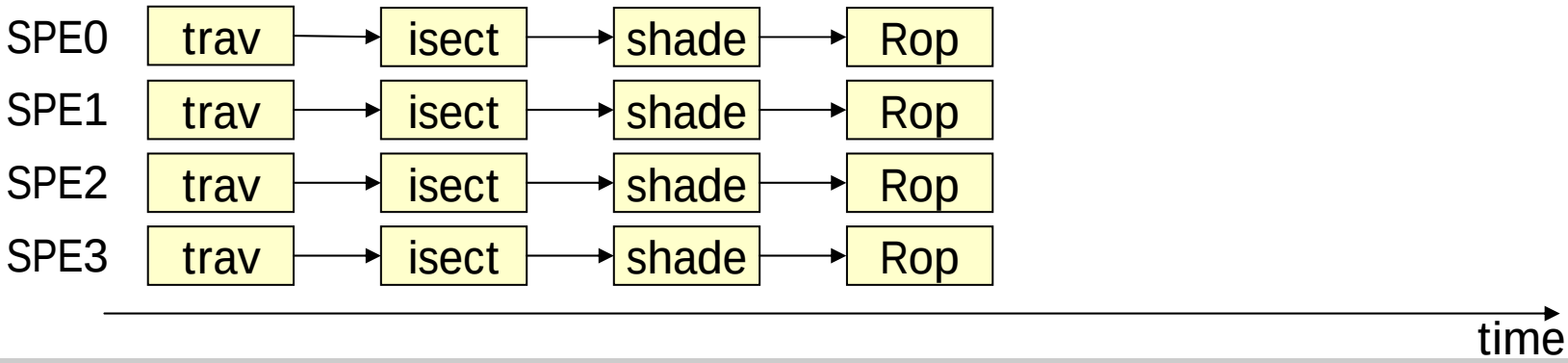
Cell レイトレーシング

- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめと改善案

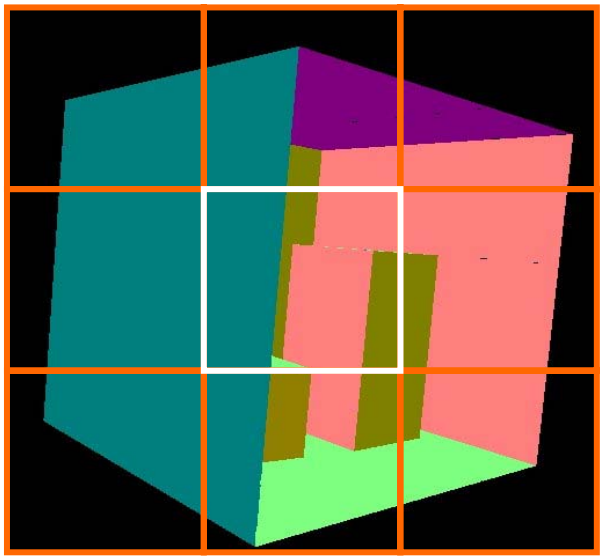
SPEへの処理の割り当て

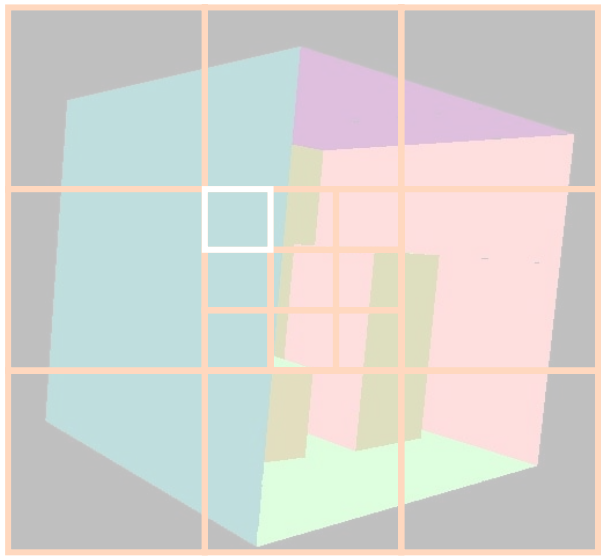
- 画像分割モデルを採用
  - すべての SPE がトラバース・交差判定・シェーディングを行う。
  - 異なるのは処理する画面位置
  - パイプラインモデルにしなかった理由:
    - モジュール間のロードバランスが困難
    - プログラミングのコスト
    - レイトレではバックトラック処理がある

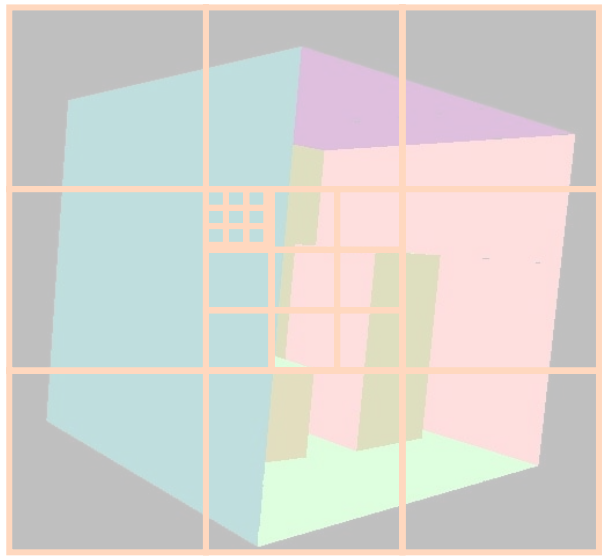
- 画面分割モデル
  - 全SPEが全機能を実行する



レイトレ処理の並列化の階層







タイル
SPE 間のワーク処理単位
(e.g. 32x32 pixel)

パケット
レイをまとめてトラバースするときの単位。ソフトウェアスレッディングの単位
(e.g. 4x4 ray)

バンドル
レイと三角形との交差判定を行うときの単位
(e.g. 2x2 ray)

SPE のお仕事

while (タイルデータが無くなるまで) {

タイルカウンタのリード
タイルカウンタのインクリメント
} 排他アクセス区間

タイルデータを DMA 転送で取得
レンダリング
タイルの結果を返す

}

排他アクセスの実現

MFC アトミック命令を使って
セマフォ処理を行う

```
int status;
do { // PUTLLC が成功するまで繰り返す
        // XDR 上のアドレス ea からアトミックに 128byte をロード
        spu_mfcdma32(data, ea, 128, 0, MFC_GETLLAR_CMD);
        spu_readch(27); // DMA 待ち
        // (... data を料理する... )
        // XDR 上のアドレス ea へアトミックに 128byte をストア
        spu_mfcdma32(data, ea, 128, 0, MFC_PUTLLC_CMD);
        status = spu_readch(27); // PUTLLC の status 取得
} while(status != 0)
```


タイルデータ

現在処理中のタイル番号

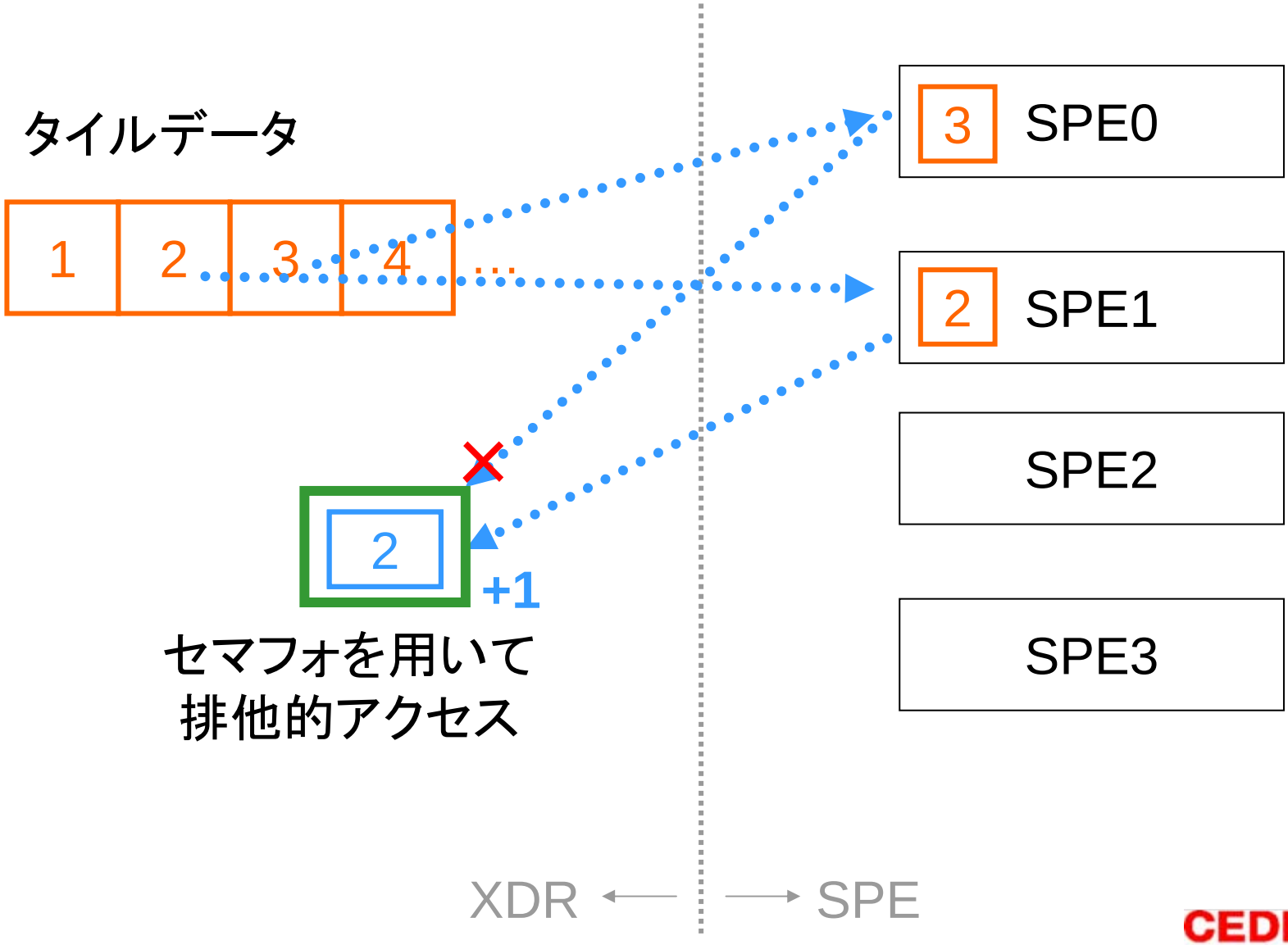
SPE0

SPE1

SPE2

SPE3

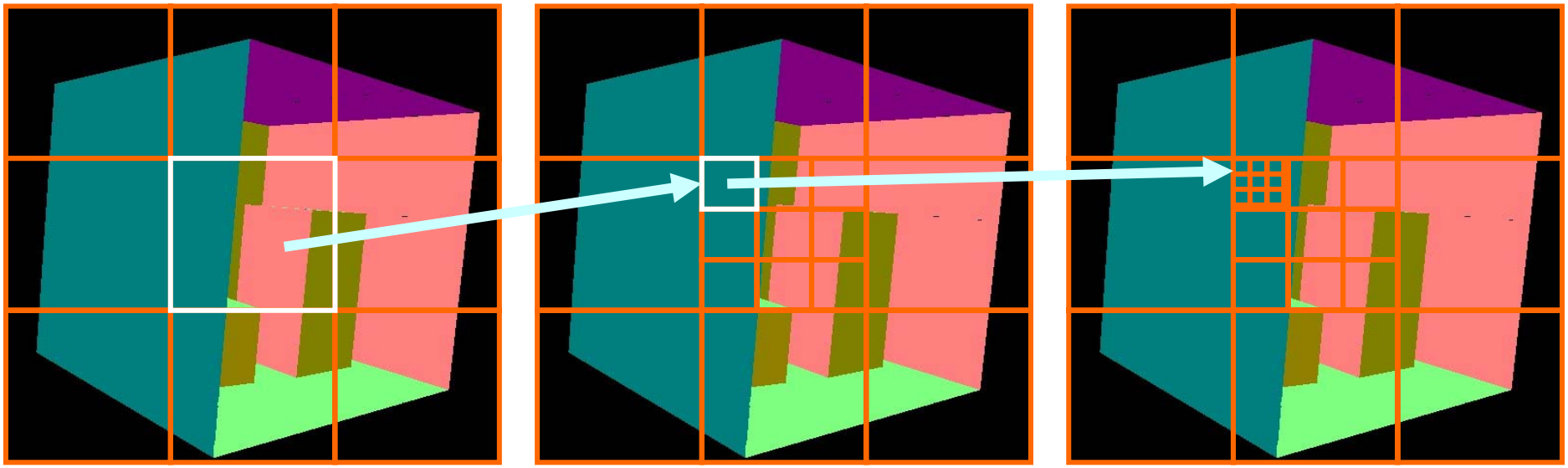
XDR ← → SPE



Cell レイトレーシング

- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめと改善案

レイトレ処理の並列化の階層

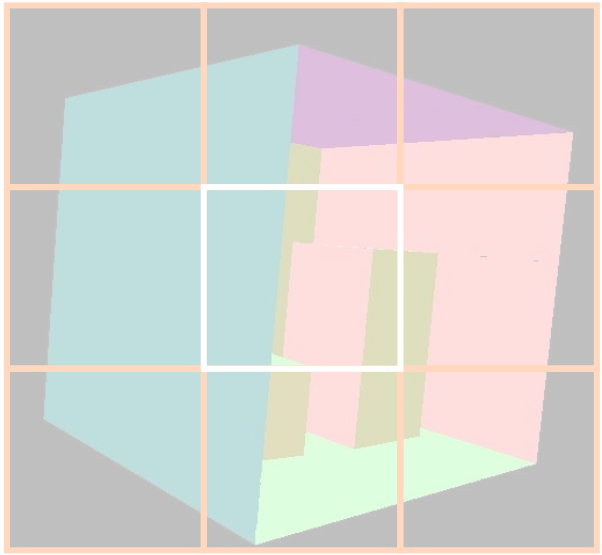


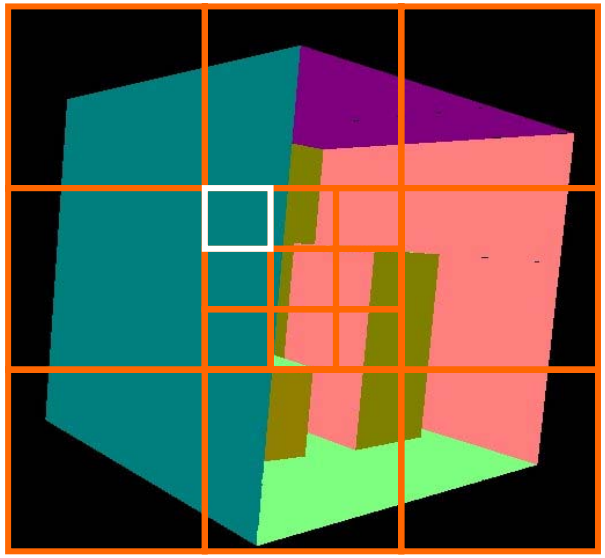
タイル
SPE 間のワーク処理単位
(e.g. 32x32 pixel)

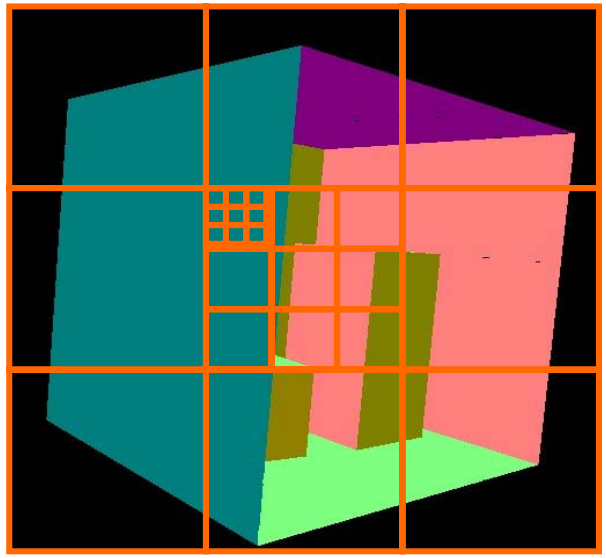
パケット
レイをまとめてトラバースするときの単位。ソフトウェアスレッディングの単位
(e.g. 4x4 ray)

バンドル
レイと三角形との交差判定を行うときの単位
(e.g. 2x2 ray)

レイトレ処理の並列化の階層







タイル

SPE 間のワーク処理単位
(e.g. 32x32 pixel)

パケット

レイをまとめてトラバースするときの単位。ソフトウェアスレッディングの単位
(e.g. 4x4 ray)

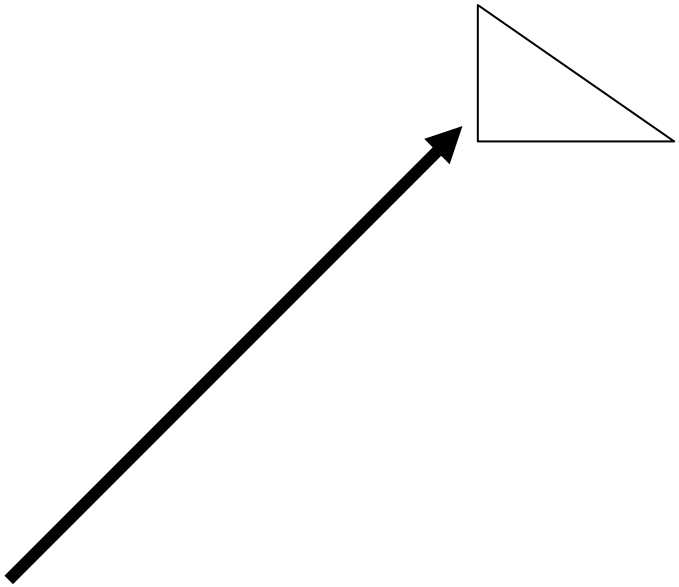
バンドル

レイと三角形との交差判定を行うときの単位
(e.g. 2x2 ray)

Cell レイトレーシング

- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめと改善案

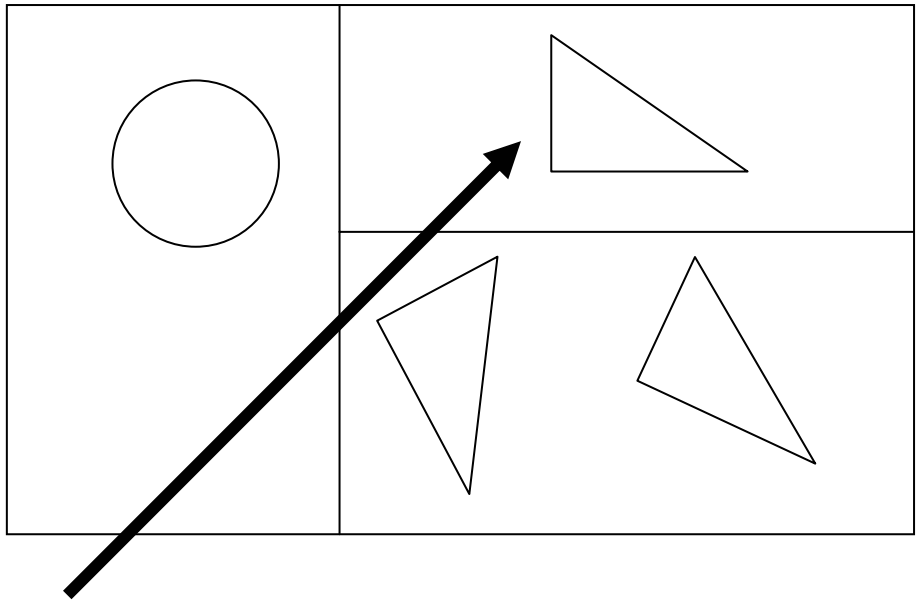
SPE 内での処理



交差判定

実際にレイと三角形とが
交差するかどうかテストする。
交差する場合は、その交点を求める。

定型的な処理。SPE での実装に最適。



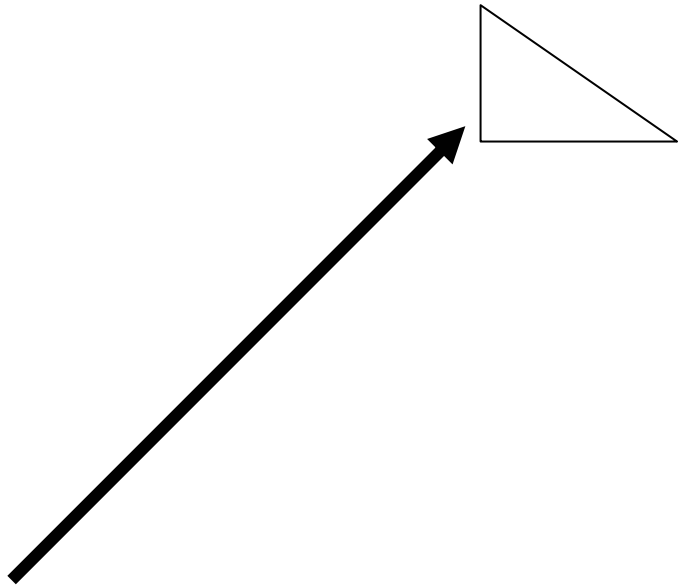
トラバース

空間データ構造にアクセスし、
レイとヒットしそうなポリゴンを
絞り込む
(e.g. kd-tree, uniform grid)

各種アルゴリズムがある。

Cell レイトレーシング

- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめと改善案

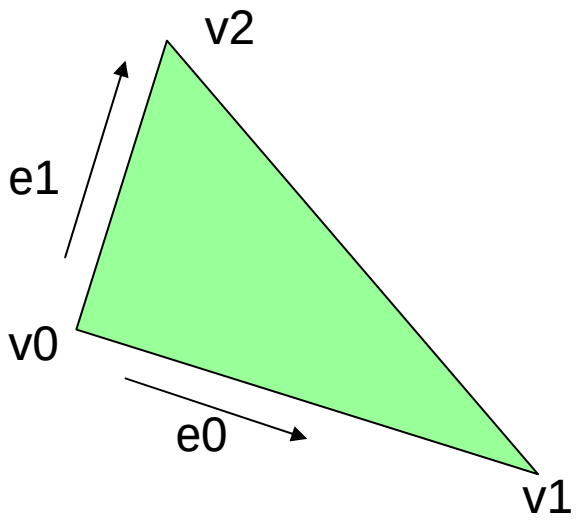


交差判定ルーチン(1/3)

- SIMD 演算を使って 4ray – 4 triangle の同時テスト
- 三角形リスト
  - LS に乗り切らない可能性
  - Double buffering を使って DMA 転送
  - 法線やテクスチャ UV もデータとして持つ
    - すこし冗長になるが、シェーディング時に再度法線や UV を DMA 転送する必要が無くなる
    - Smooth shading を行えるように、各頂点に法線・UV を持たせている
- 交差判定アルゴリズムは [Mueller 1997]
  - SIMD 化して SPE 向けに最適化

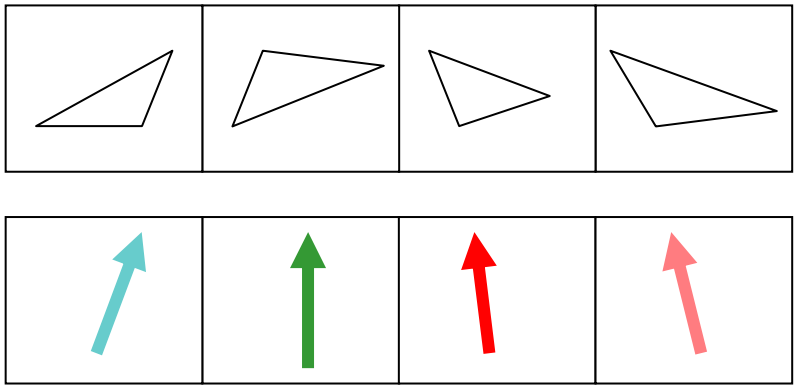
交差判定ルーチン(2/3)

- エッジベクトルを前計算
  - e0 = v1 – v0, e1 = v2 – v0
  - (v0, v1, v2) ではなく、(v0, e0, e1) を三角形データとする
  - v1, v2 は交差判定では必要としない。
    - 必要であれば、v1 = v0 + e0, v2 = v0 + e1 で求められる。
  - 3 sub x 2 = 6 sub op の節約



交差判定ルーチン(3/3)

- 4 レイ x 4 三角形 @ 221 cycles
  - 161 single-issue cycles, 60 dual-issue cycles, 0 stall cycles.
  - 1 交差判定あたり **13.81 cycles**.
    - x86 は 20-25 cycles [Wald Ph.D thesis]
    - ループアンロール + 豊富なレジスタ数のおかげ
    - 原点が同じだとすれば(一次レイの場合)、さらに削減の余地あり



三角形データ構造詳細

384 bytes

P	N	ST

t0v0x	t1v0x	t2v0x	t3v0x
t0v0y	t1v0y	t2v0y	t3v0y
t0v0z	t1v0z	t2v0z	t3v0z
t0e0x	t1e0x	t2e0x	t3e0x
t0e0y	t1e0y	t2e0y	t3e0y
t0e0z	t1e0z	t2e0z	t3e0z
t0e1x	t1e1x	t2e1x	t3e1x
t0e1y	t1e1y	t2e1y	t3e1y
t0e1z	t1e1z	t2e1z	t3e1z

36 floats(SIMD friendly)

t0n0x	t0n0y	t0n0z
t0n1x	t0n1y	t0n1z
t0n2x	t0n2y	t0n2z

...

36 floats

Scalar format

t0st0s	t0st0t
t0st1s	t0st1t
t0st2s	t0st2t

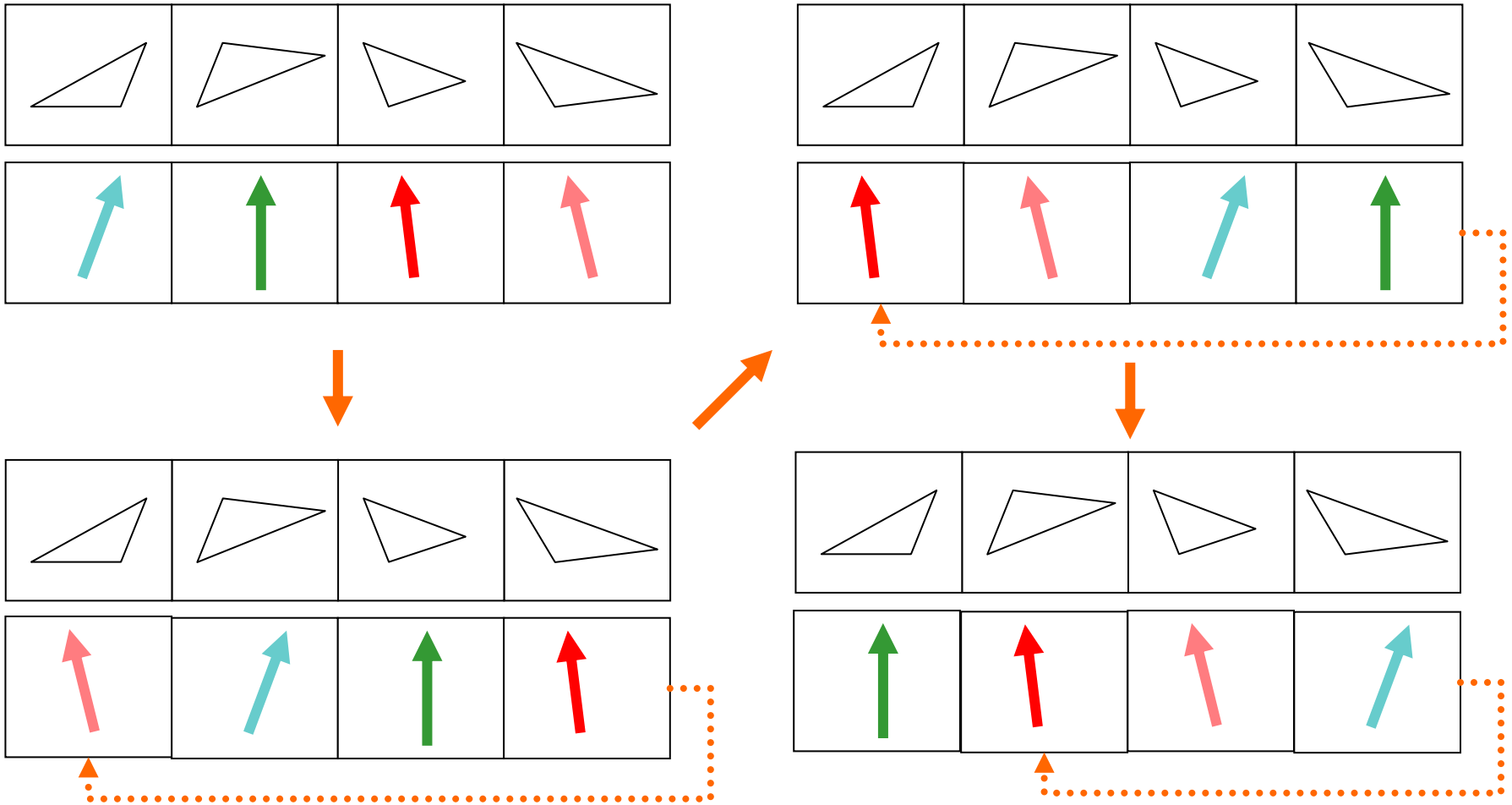
...

24 floats

Scalar format

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

SIMD レイ-三角形交差判定



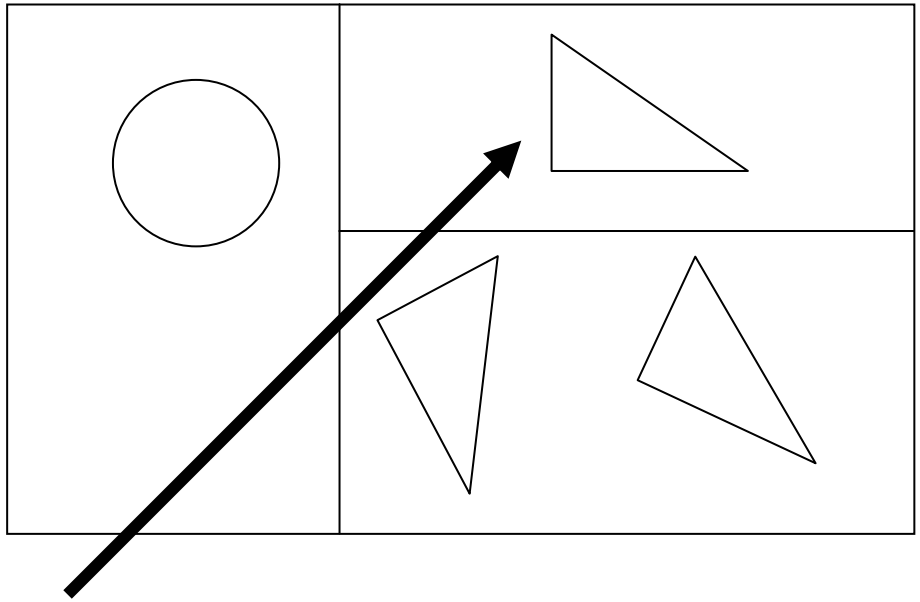
レイのデータを rotate することで、4 個の SIMD 三角形と
4 個の SIMD レイとの交差判定を行う(Appendix 参照)

性能見積もり

- x86: 20 - 25 サイクル/三角形 [Wald Ph.D thesis]
- SPE: 14 サイクル/三角形
- 1024x1024 スクリーンサイズ, 30 fps
  - 3.2(GHz) / (1024 * 1024[pixels] * 30[fps] * 14[cycle]) = **7.26** 三角形/ピクセル
  - SPE 1 個につき、ピクセルあたり 7.26 個の三角形との交差判定しか余裕が無い
    - SPE 8 個だと 58 三角形
    - これは視点レイのみ。シャドウレイをトレースするとここから半減。多重反射をするとさらに減少…
  - [Reshetov 2005] では、交差判定に 1/3, トラバースに 2/3 のコストモデルを提案
    - 当てはめると、ピクセルあたり 19 三角形 @ 30 fps

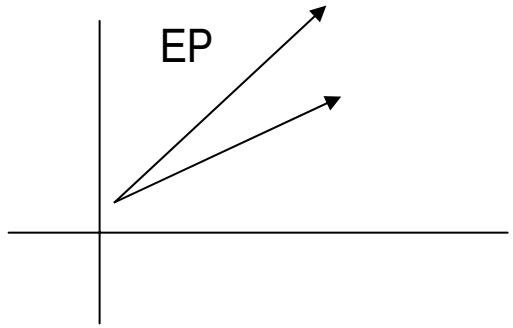
Cell レイトレーシング

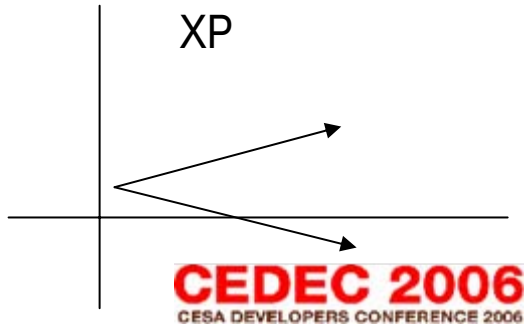
- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめと改善案



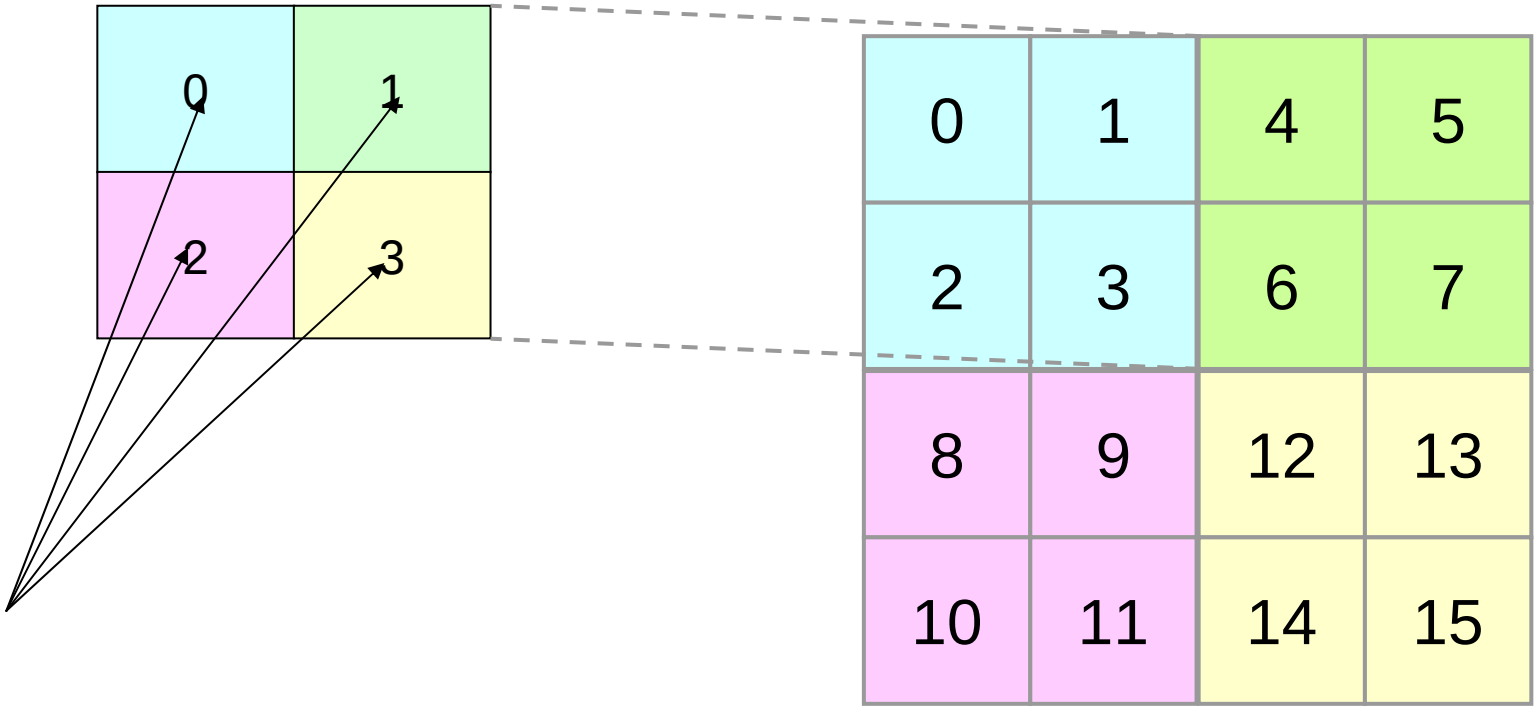
トラバースアルゴリズム

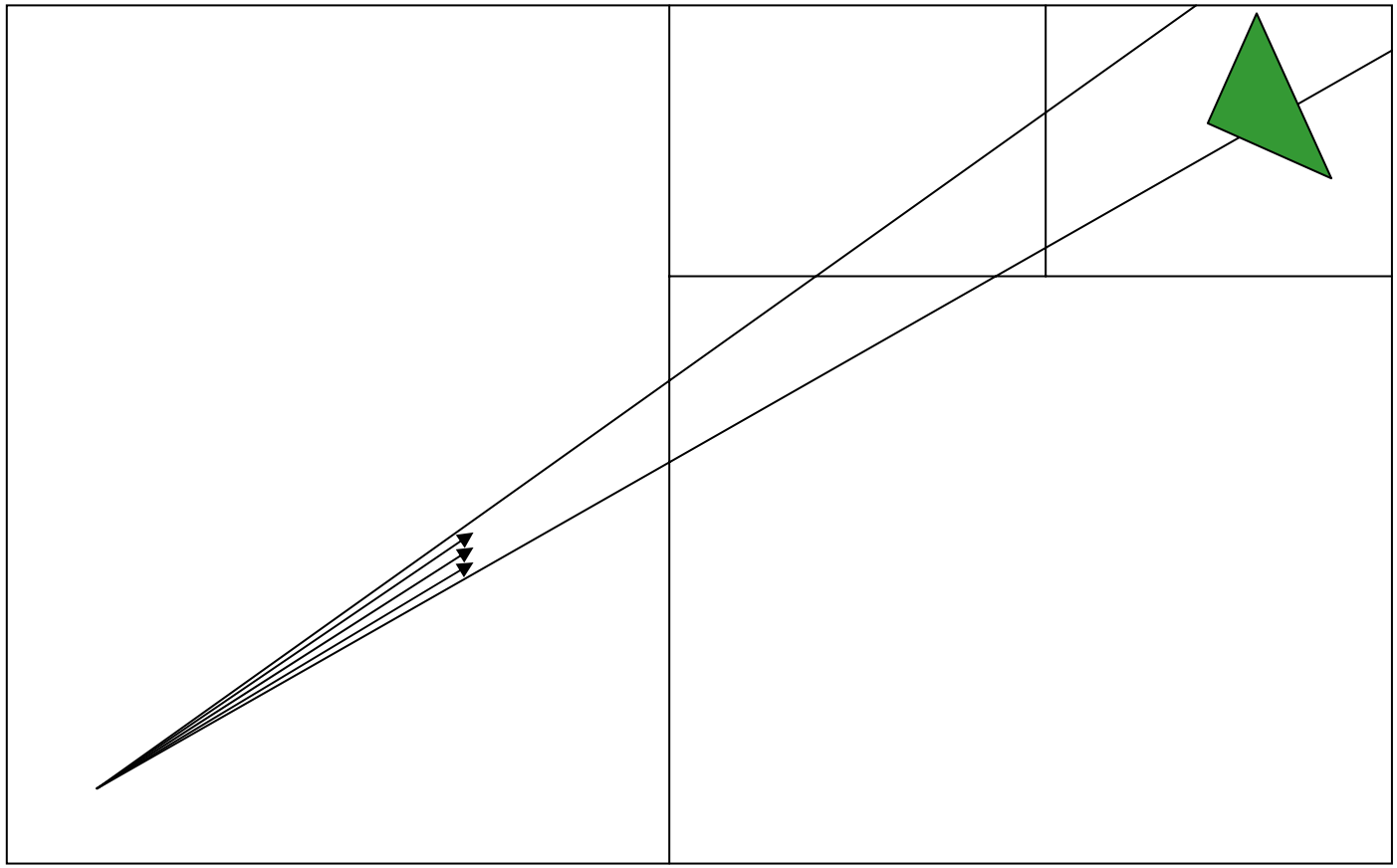
- kd-tree を利用
  - ノードデータ全部は LS に入りきらない
    - LS に常駐できるサイズにノードデータを制限
      - 1 ノード 4 byte のデータ構造を採用
      - 8Kから16K個までのノード数に制限される(32KB~64KB in LS)
    - その代わり葉ノードあたりの三角形数が増えるので、交差判定数が多くなる
- パケットトラバース
  - パケット単位でレイをまとめてトラバース
  - MLRTA[Reshetov 2005] ベースのアルゴリズム
  - EP 探索と XP 探索
    - EP 探索
      - パケットトラバースが可能な探索
      - パケット内のレイがすべて同じ方向を向いている
        - かどうかの判定には、単純にレイの方向の符号をチェックすればよい
    - XP 探索
      - パケットトラバースが不可能な探索
        - パケット内のレイの方向が異なる
      - 細分割を試み、ダメなら1 本 1 本のレイを別々にトラバースする
- バンドルトレーシング
  - バンドル単位でレイをトラバース

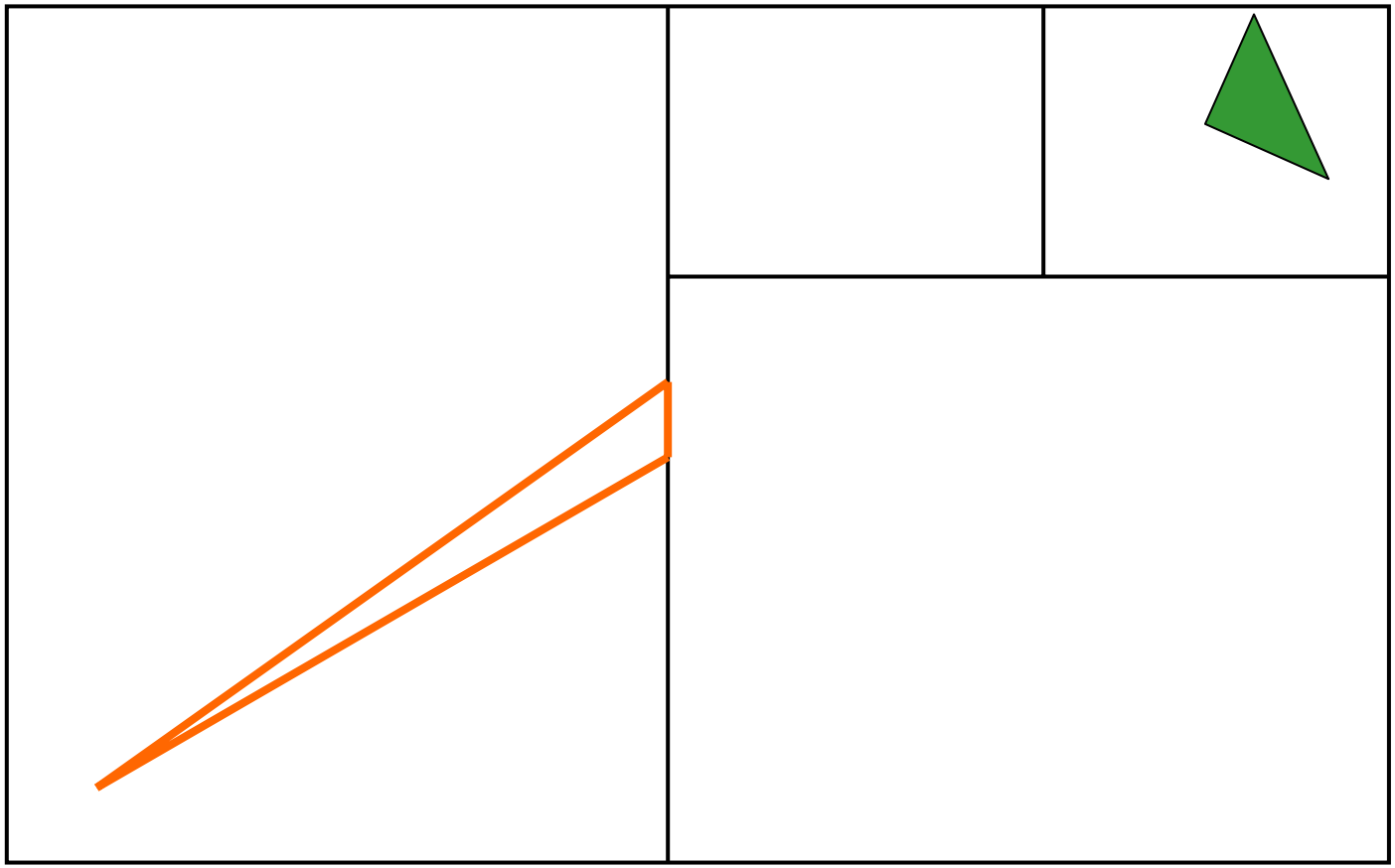




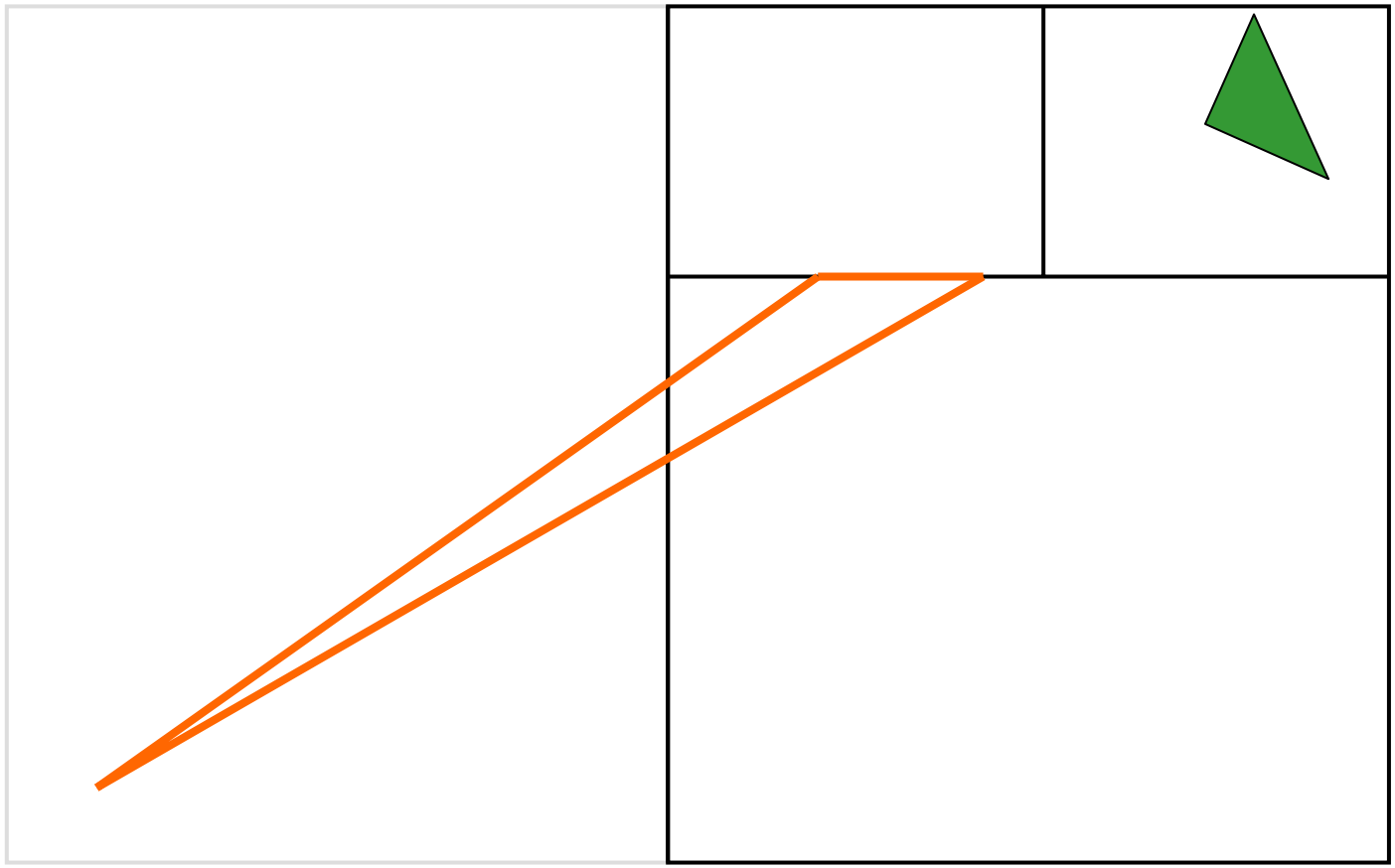
- バンドルトレーシング
  (2x2 pixel を SIMD処理)
- パケットトラバース
  (4x4 pixel 単位)



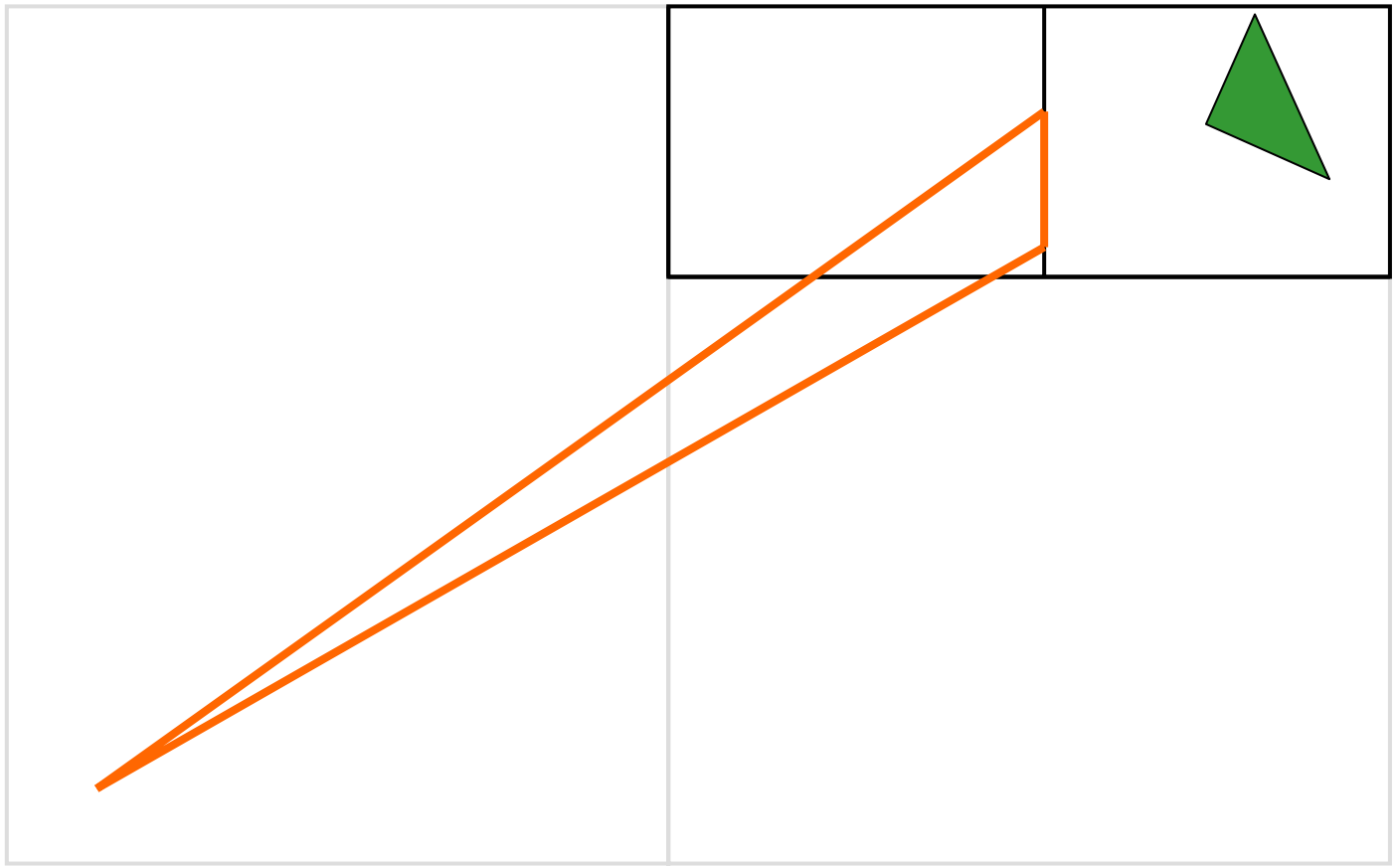




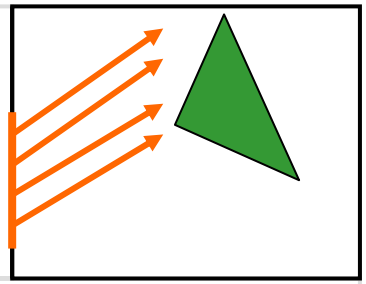
パケットトラバース



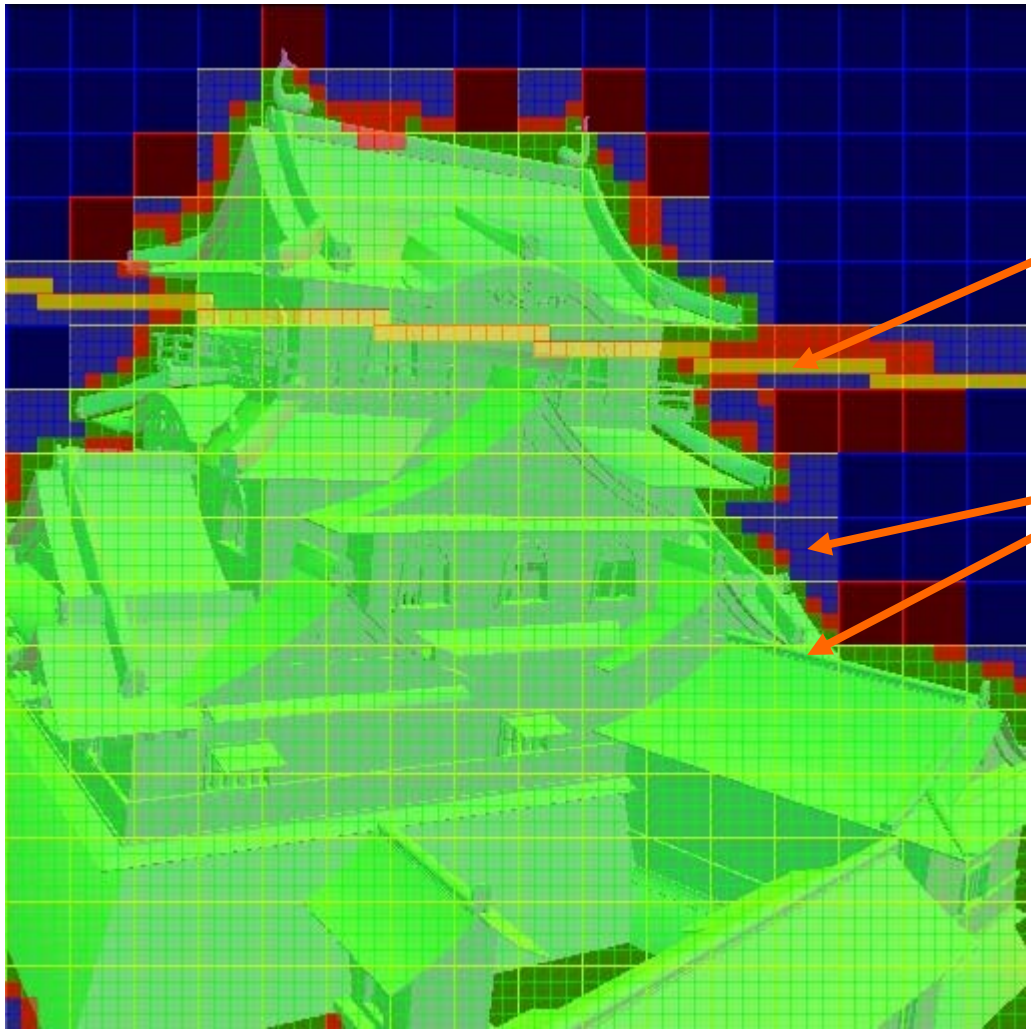
パケットトラバース



まだまだパケットトラバース



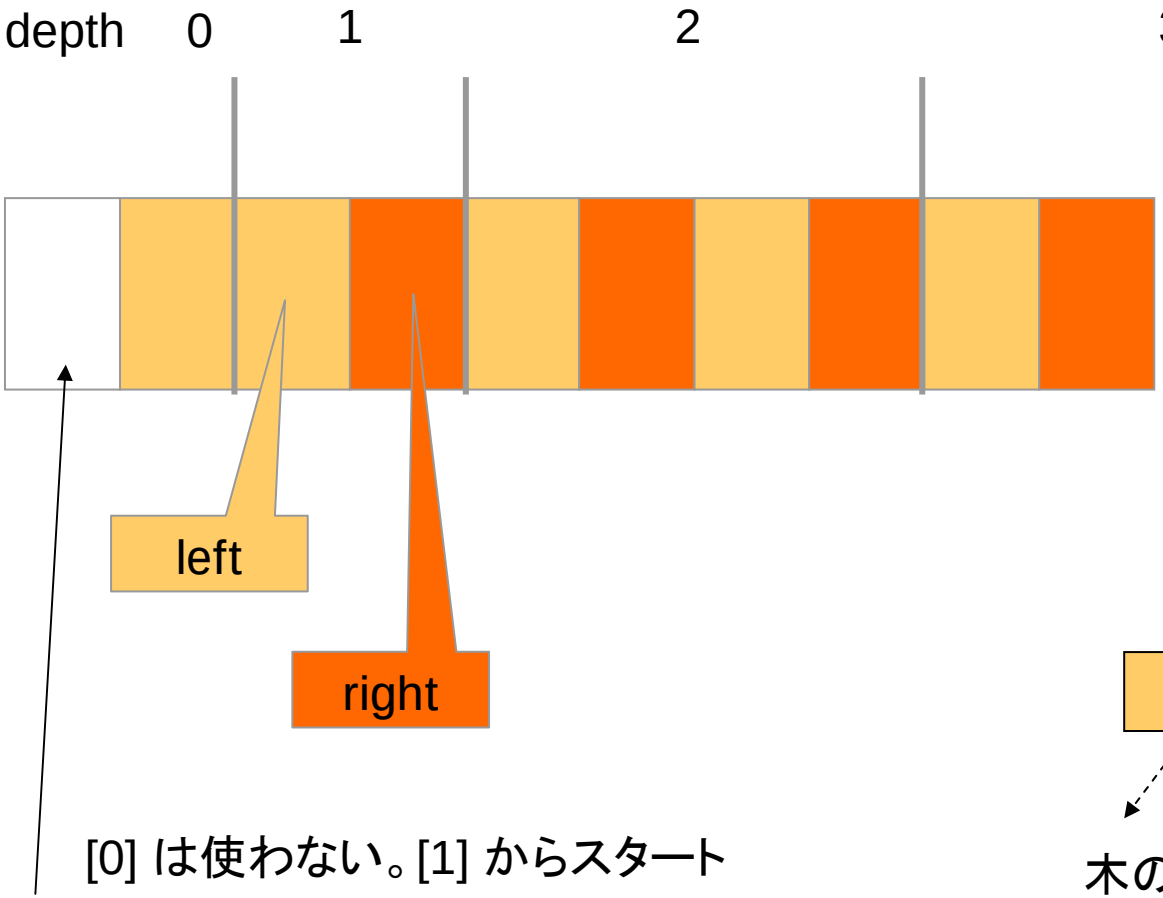
バンドルトレーシング



XP 探索部分
(黄色)

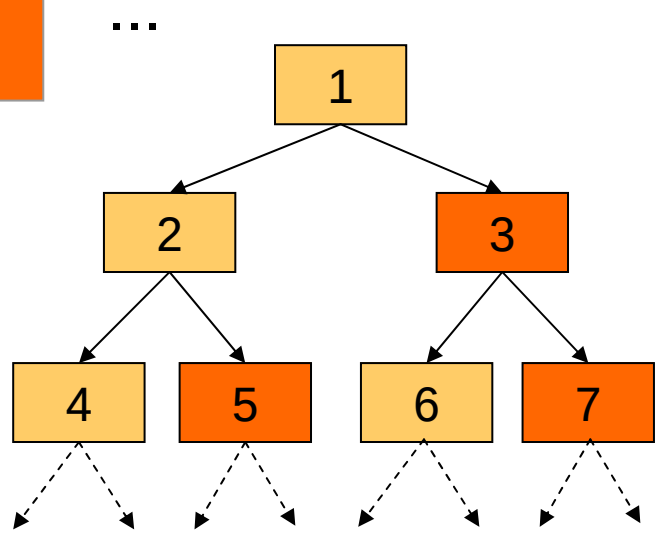
EP 探索部分

kd-tree の木構造の配列表現



[0] は使わない。[1] からスタート

left = curr_index << 1
right = curr_index << 1 + 1

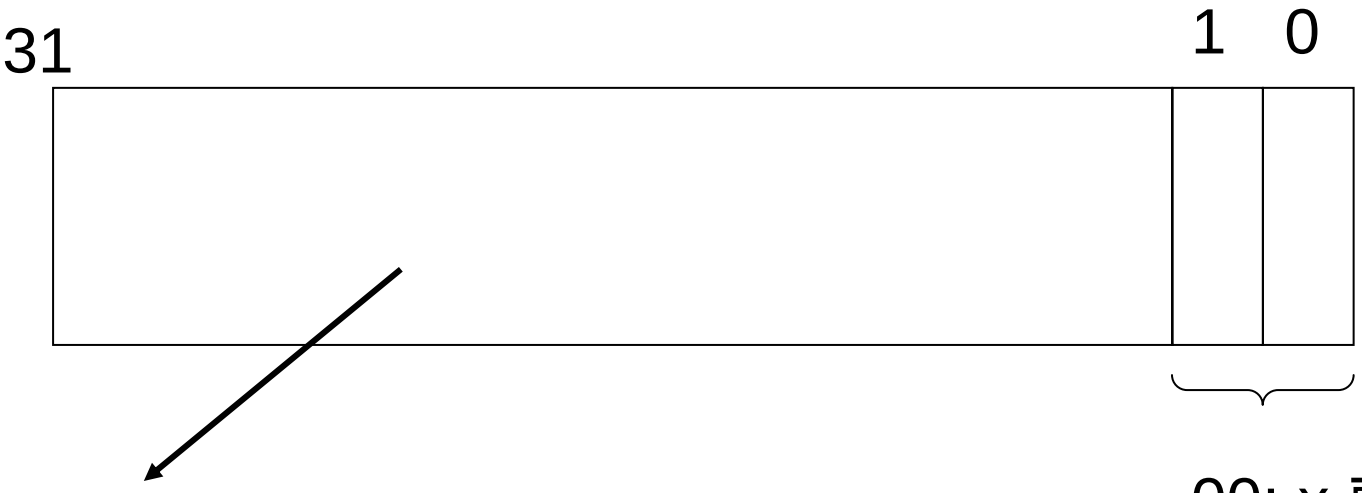


木の最大の深さを depth とすると、
2^depth ぶんのノード数が必要

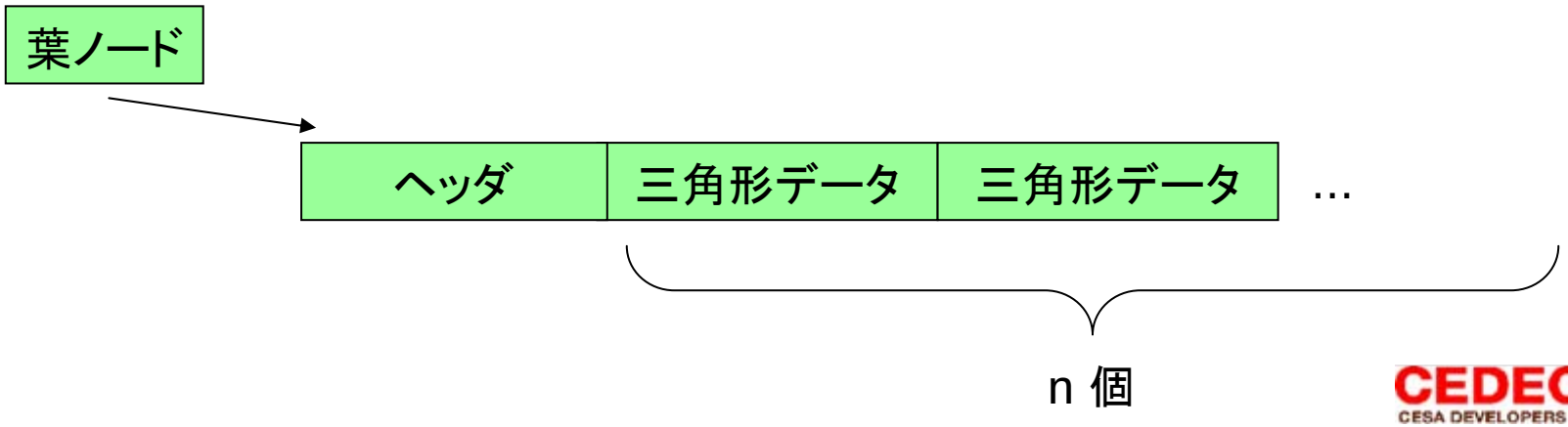
CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

ノードのデータ構造

- コンパクトノードフォーマット[Ericson 03]
  - 2bit を葉ノードか中間ノードかの判定に使う
  - 30bit で分割位置を表現する(X * 2^(-19) の精度. X はシーンの最大座標値)
    - [Graphics Programming Methods]



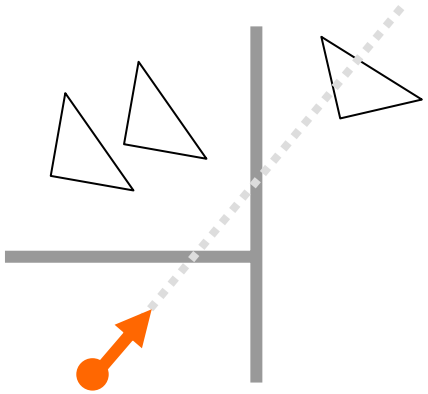
- 葉ノードは三角形リストへのアドレスを持つ
- リストのヘッダ
  - 三角形数と三角形データのバウンディングボックスを持つ
  - 交差判定の前に三角形データのバウンディングボックスでカリングを行うのに使う
  - DMA 転送を容易にするためにヘッダサイズは 128byte
  - 三角形データのヘッダをキャッシュ
    - 葉ノードに到達するたびにアクセスされるため
    - ダイレクトマップ、16 エントリ
    - スレッドごとにテーブルを保持
    - Cache hit
      - コード単体: 20 cycle
      - インライン展開されれば 10 cycle 程度



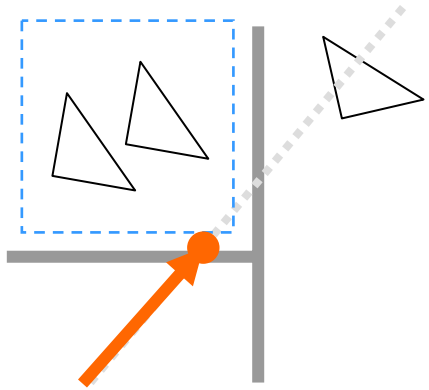
スレッド化トラバース(1/3)

- パケット単位のソフトウェアスレッディング
  - カリング fail 後、交差判定を行うため三角形データの最初の chunk を DMA 転送している間にスレッド切り替え
  - sw スレッディングはレイトレのように処理の並列度が高く、データ依存関係が少ない(スレッドストレージが少なくて済む)場合に有効。
    - 完全なコンテキストスイッチの実装は、レジスタの退避などコストがかかる
    - パケットのサイズ 約 3KB
      - トラバース用スタック: 600B
      - レイのデータやその他: 2KB
  - スレッド数は最大で 4.

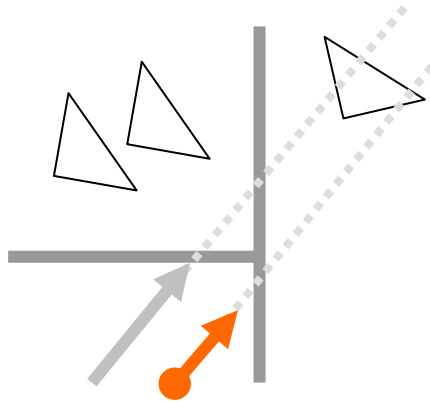
スレッド化トラバース(2/3)



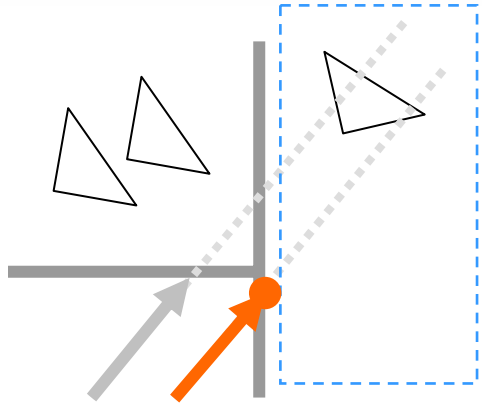
Ray1 のトラバースを開始



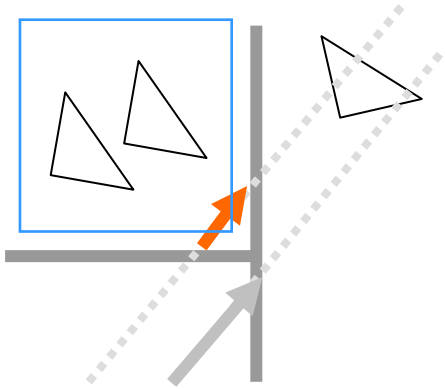
Ray1 が Leaf1 に入る
- Leaf1 の triangle を DMA 転送開始
- Thread 切り替え



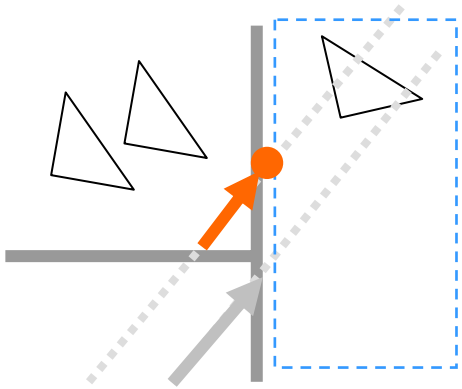
Ray2 のトラバースを開始



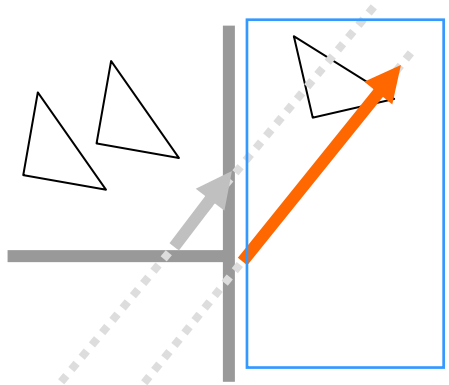
Ray2 が Leaf に入る
- Leaf2 の triangle を DMA 転送開始
- Thread 切り替え



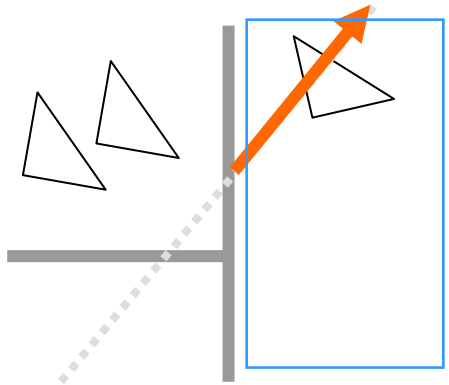
Ray1 と Leaf1 の交差判定を行う。



交点は見つからない
- Ray1 のトラバースを続ける
- Ray1 が Leaf2 に入る
- Leaf2 の DMA 転送開始
- thread 切り替え

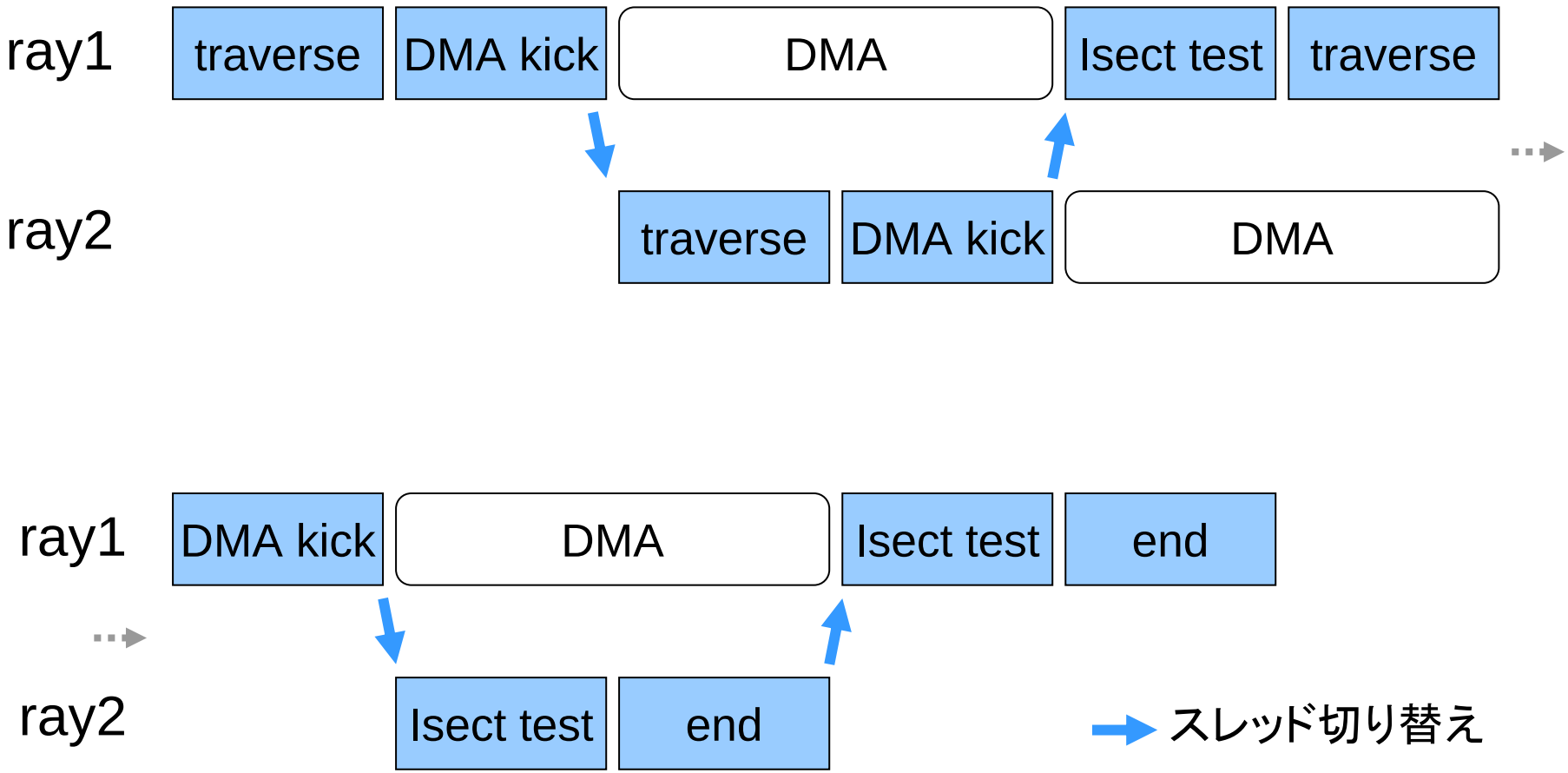


Ray2 と Leaf2 の交差判定
- 交点が見つかる
- Ray2 のトラバース終了
- thread 切り替え

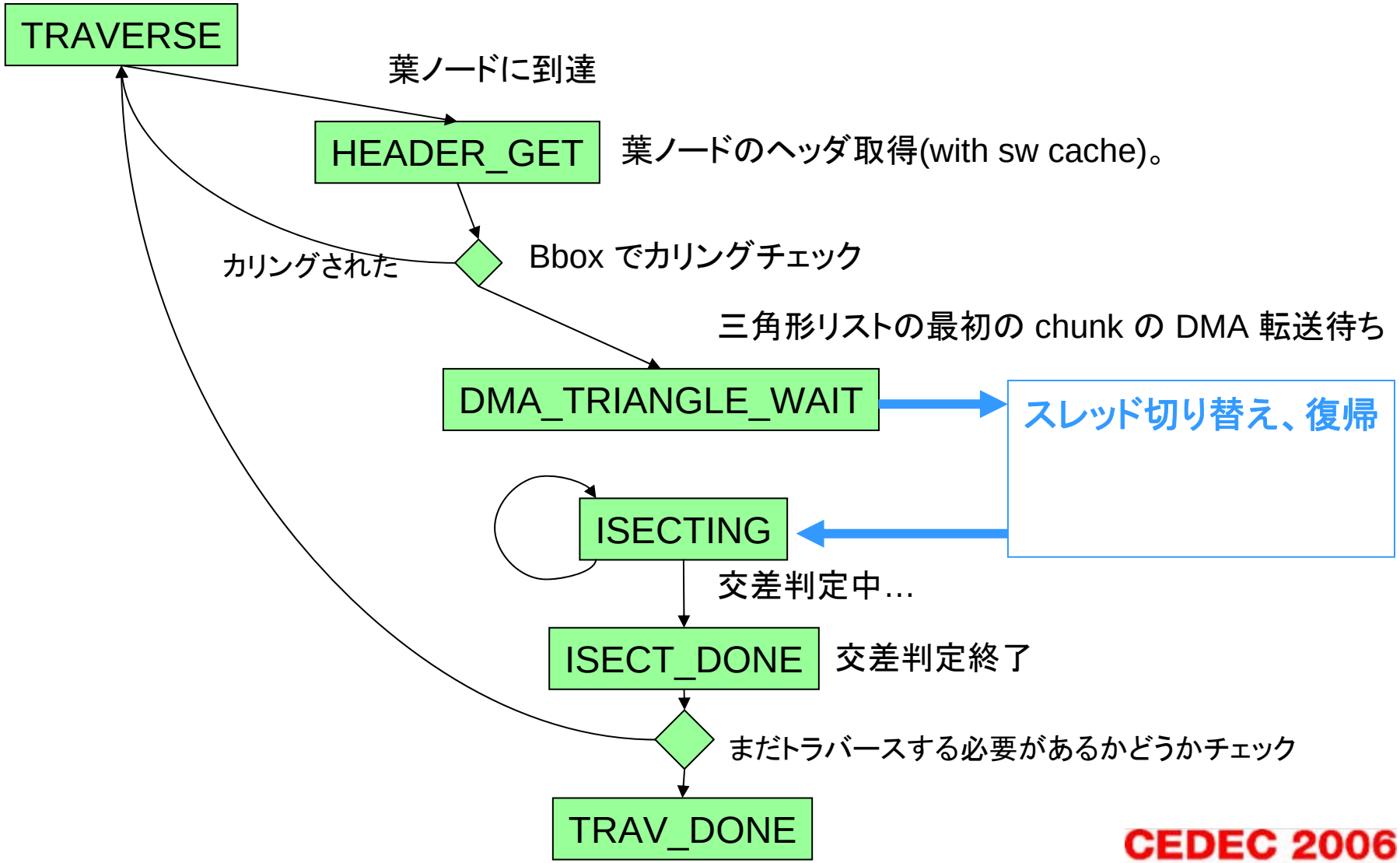


Ray1 と Leaf2 の交差判定
- 交点が見つかる
- Ray1 のトラバース終了
- 全 ray のトラバース完了

スレッド化トラバース(3/3)

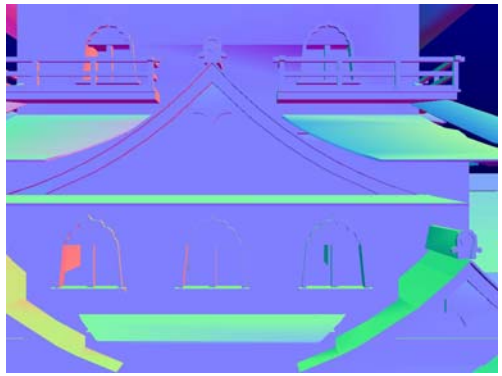


トラバースのフロー



Cell レイトレーシング

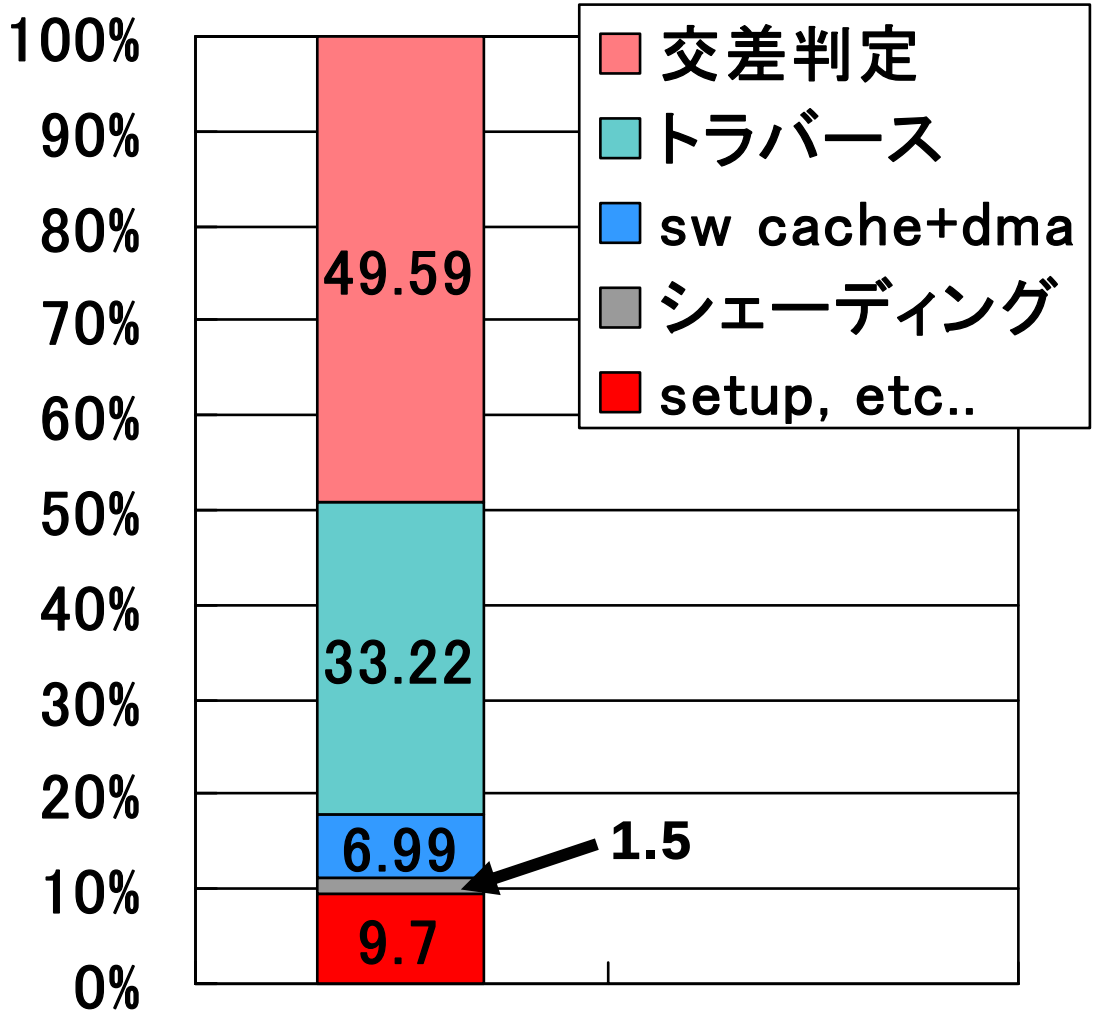
- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめ



	城	Stegosaur
トラバーサル数	77.34 ノード/ パケット(4x4 rays)	90.0 ノード/パケット(4x4 rays)
交差判定数	195 三角形/ray	46 三角形/ray
SPE レンダリング時間	281 msec	184 msec

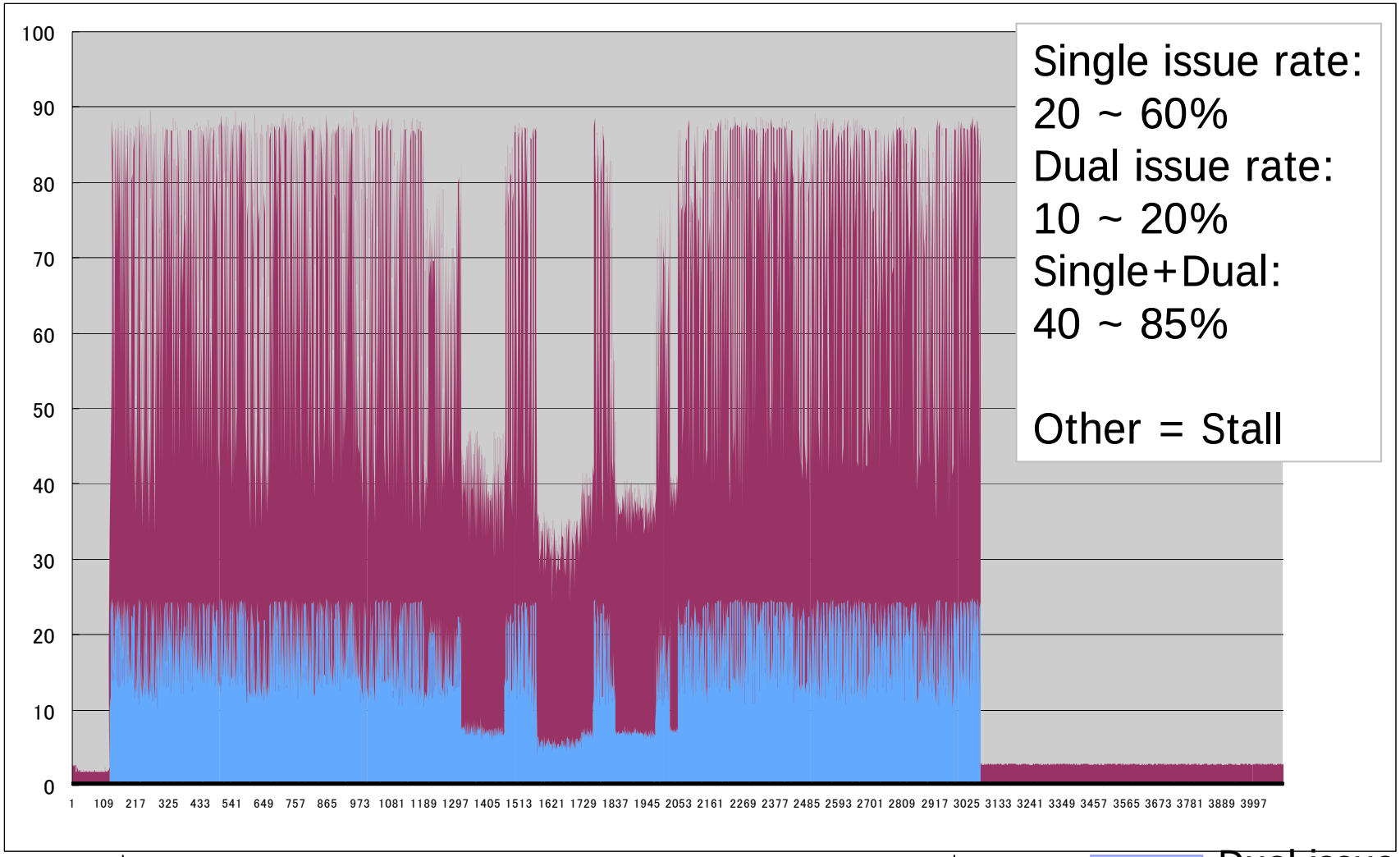
1024x768, 2.4GHz x 8 SPEs, シェーディング: 法線の補間

城シーンの処理内訳



- sw cache + dma
  - 三角形データヘッダの sw cache 計算 + cache miss stall
  - 三角形データの DMA 転送にからむ処理 (DMA stall 含む)

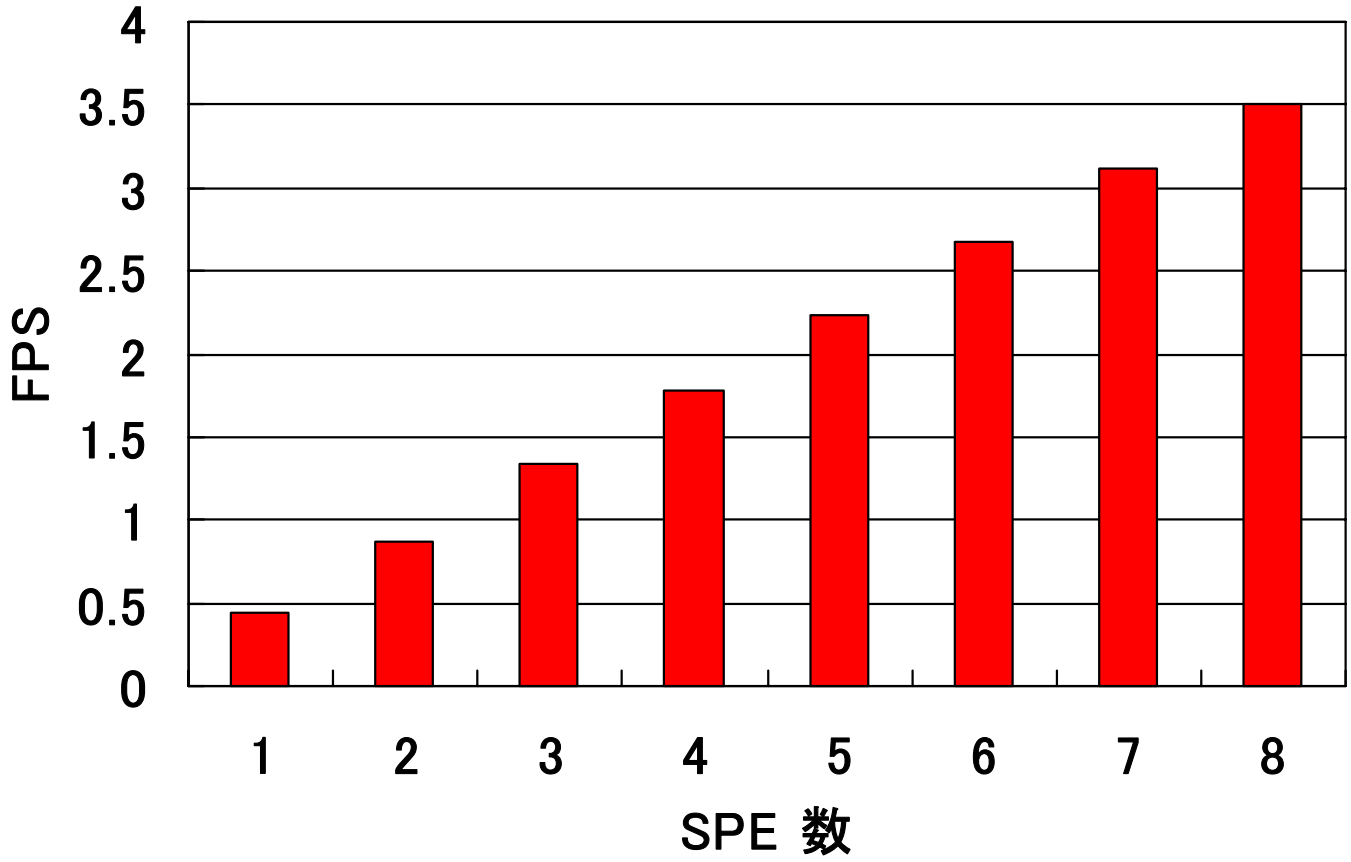
時系列解析



交差判定検証

- 1 交差判定 13.81 サイクル on SPE
- 理論上限
  - 2.4GHz x 8 / (1024x768x13.81) = 1768 三角形/ray(秒間)
- 城シーン
  - 195 三角形/ray x (1000 / (281 msec x 49.59%)) = 1400 三角形/ray(秒間)
- 1400/1768 = 理論値の 80%
  - プログラムのプロセッサ利用率 が最大 85-90%
  - 城シーンの処理内訳の交差判定部分(49.59%)にはシェーディング用データのアップデート(t, u, v, 法線)など extra な処理も含まれている
  - ゆえに理論値の 80% というのは妥当なところ

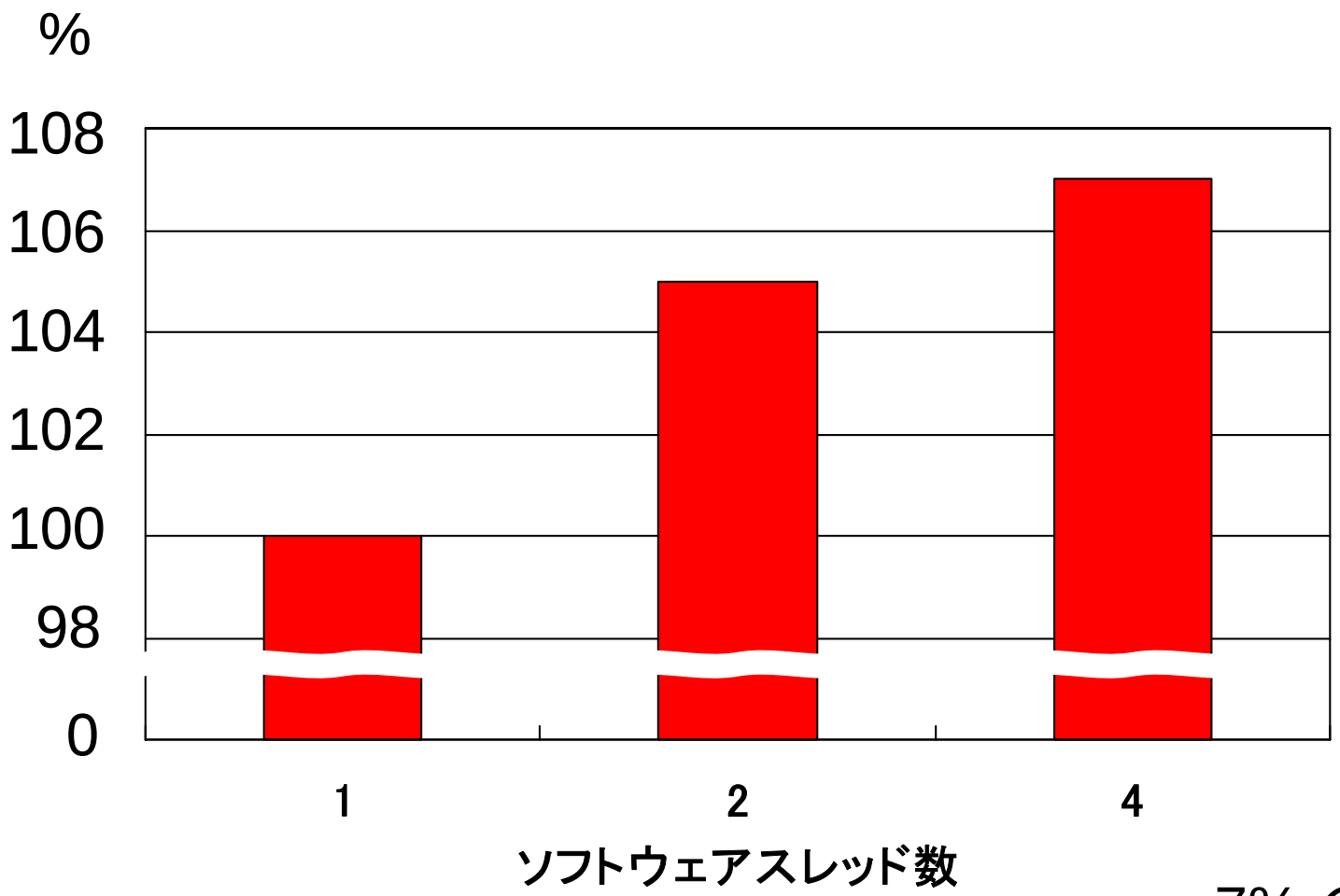
SPE 数の効果



城のシーン

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

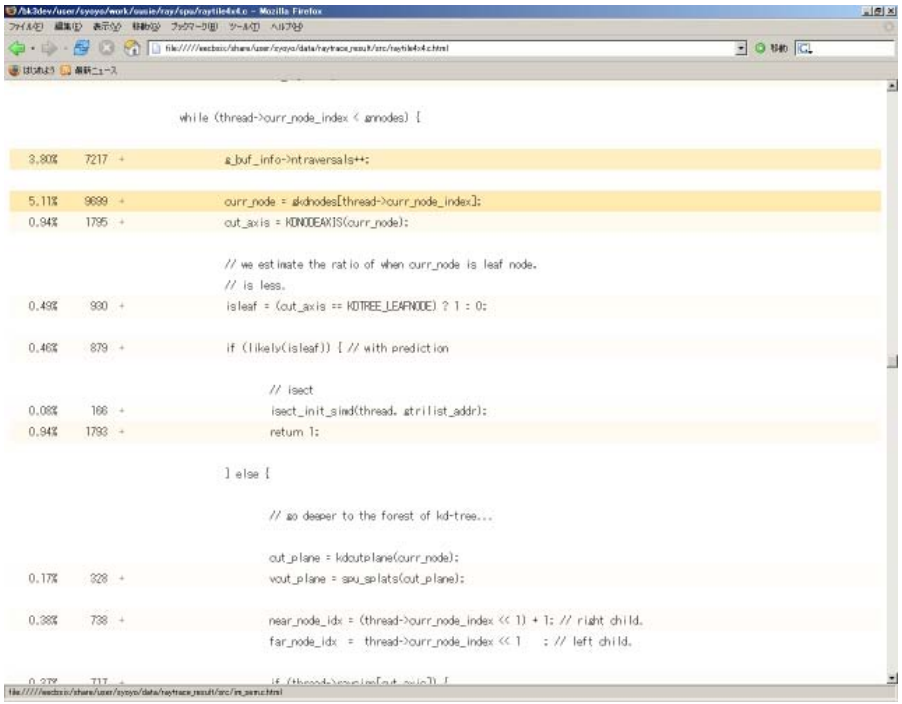
ソフトウェアスレッドの効果

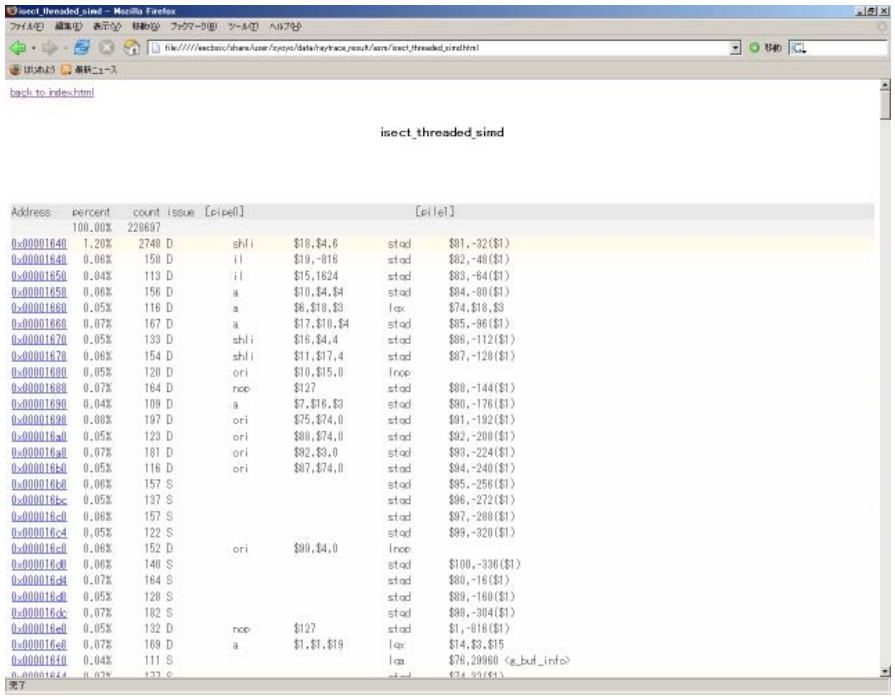


~7% の向上

最適化: オンラインプロファイラの活用

- インハウスオンラインプロファイラの活用で、最適化してあると思っているコードでもさらに 1.2 – 2.0 倍の高速化を実現。
- プロファイル結果でカバレッジの高い結果が出たコード部分を集中的に最適化、とくに交差判定コード部分は無ストールに。





CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

Cell レイトレーシング

- 概要
- SPE 間での負荷分散
- SPE 内での処理
  - レイと三角形の交差判定
  - レイのトラバース
- 結果と統計
- まとめと改善案

まとめ

- リアルタイムレイトレーシングの基礎的な Cell 実装を解説しました
- 交差判定処理は Cell のピーク性能に近い処理性能を達成することができる
- Kd-tree ノードは LS に常駐
  - 8Kから16K個までのノード数に制限される(32KB~64KB in LS)
- 葉ノードでの三角形リストは double bufferring で取得
- パケット単位でのソフトウェアスレッディング
- SPE 間でのワークの同期処理
  - アトミック命令によるセマフォの利用
  - PPE は介在しない
- SPE プログラムサイズ
  - プログラムコード(text): 41 KB
  - データ(data, bss) : 150 KB

問題点と改善案(1/2)

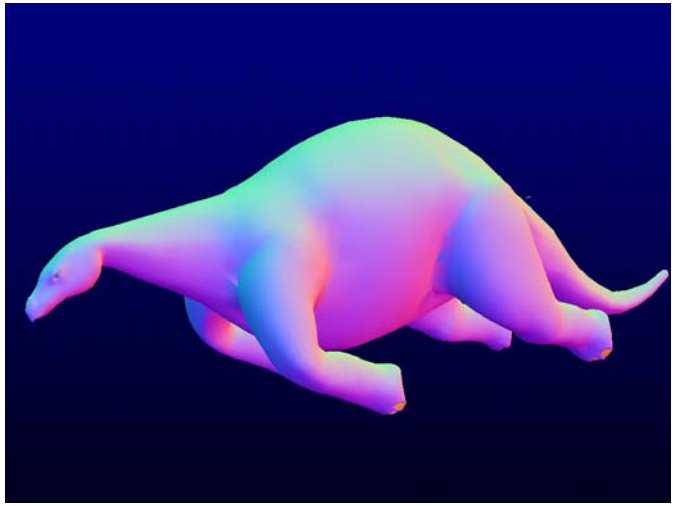
- 1 ray 数十から数百の三角形との交差判定は多すぎ
  - LS にノードを常駐させるため、少ないノード数にした代償に、多くの三角形を葉ノードが持ってしまう。
  - 世界のリアルタイムレイトレ研究者は 1 ray につき 1~10 三角形までにカリカリチューンしている。
- Cell に向いたトラバースアルゴリズムを考える必要がある
- 他の交差判定手法も試してみる
  - Projection 法[Wald, Ph.D thesis], Pluecker 座標法, etc.
    - SPE に向いた交差判定があるかも

問題点と改善案(2/2)

- ノードごとに bounding box を保持すると LS に収まらない
  - Float bmin[3], bmax[3] で 24byte 消費
  - E.x. 8192 nodes: 32KB で収まっていたのが...
    - 8192*24 = 192KB の増加!!
  - [David, JGT] では node あたり 12byte の bounding box 手法を提案している。
- ヒープによる暗示的なノード階層
  - 実際にはほとんどが empty node.
  - LS は in キャッシュなので、PC raytracing のように L1 キャッシュミスのコストは無い。
  - 明示的に子へのポインタ(ノードあたり ushort x 2の増加)を持つほうがノード情報全体としては少なくなることもある。
- 1 シーン 1 kd-tree
  - LS に常駐するだけの階層に制限される。
  - ポータルなどを利用して、シーン全体で kd-tree を構築するのではなく、部屋ごとに kd-tree を構築するとよいかも。
- ノード情報は LS に常駐
  - より大きなノード階層は、プリフェッチさせることで扱うことができる。
  - プリフェッチしても使わなかった場合のコストとの兼ね合い。
  - もしくは毎度ソフトウェアキャッシュを通したノードへのアクセスにする
    - キャッシュミスした場合はレイのスレッドを切り替え

Future work

- 全部をレイトレでやらずに、アルゴリズムの適材適所
  - 一次レイはラスタライズで、反射などの二次レイをレイトレが担う
  - “やわらか”シェーダの中で、レイトレを呼び出す
- RealTime Global Illumination？
  - レイトレのさらに 100 倍以上の性能が必要(レイトレで GI を実現するなら)
- 新しいレイトレのアルゴリズムの探索
  - 特に二次レイ以降の扱い



Future?

Agenda

- Cell プロセッサの仕組み
- Cell ポリゴンレンダリング
- Cell リアルタイムレイトレーシング
- 参考文献

参考文献(Cell 一般)

- Cell 公開文書
  - http://cell.scei.co.jp/index_j.html
  - アーキティクチャ、言語仕様など
- Cell SDK
  - http://www-128.ibm.com/developerworks/power/cell/index.html
  - コンパイラ、シミュレータ環境など

参考文献(リアルタイムレイトレーシング)

- Ingo Wald, **Realtime Ray Tracing and Interactive Global Illumination**. PhD thesis, Saarland University, 2004
  - レイトレ野郎は必読
- Ingo Wald, Philipp Slusallek, Carsten Benthin, Markus Wagner. **Interactive Rendering with Coherent Ray Tracing**. EUROGRAPHICS 2001, pp 153-164, 20(3), Manchester, United Kingdom, Sept. 2001
- Carr, Nathan A., Jesse D. Hall and John C. Hart. **The Ray Engine**, Proc. Graphics Hardware 2002, Sep. 2002.
- Timothy J. Purcell and Ian Buck and William R. Mark and Pat Hanrahan. **Ray Tracing on Programmable Graphics Hardware**, SIGGRAPH 2002.
- David Cline, Kevin Steele and Parris Egbert. **Lightweight Bounding Volumes for Ray Tracing**, Journal of Graphics Tools(to appear)
  - ノードあたり 12byte のバウンディングボックス
- Laszlo Szecsi, **An Effective Implementation of the k-D Tree**, Graphics Programming Methods, Charles River Media. 2003
  - コンパクトなノード保持方法について
- Crister Ericson, **Memory Optimization**, GDC 2003.
- Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha, **R-LODs: Fast LOD-Based Ray Tracing of Massive Models**, Tech. Report, TR06-009, Univ. of North Carolina at Chapel Hill, 2006
  - LOD を使った巨大シーンのレイトレ. 2-3 fps @ Dual Xeon.
- Tomas Mueller and Ben Trumbore, **Fast, minumum storage ray-triangle intersection**. Journal of Graphics Tools, 2(1) pp.21-28, 1997
- Alexander Reshetov, Alexei Soupikov and Jim Hurley, **Multi-level ray tracing algorithm**. SIGGRAPH 2005, pp. 1176-1185, 2005
  - MLRTA
- Carsten Benthin, Ingo Wald, Michael Scherbaum and Heiko Friedrich, **Ray Tracing on the CELL processor**. Technical Report, inTrace Realtime Ray Tracing GmhB, No inTrace-2006-001, 2006.
  - 世界初の明文化された Cell レイトレ?
- **http://ompf.org/forum**
  - 世界のリアルタイムレイトレ野郎が集まる場所
- **The 2006 IEEE Symposium on Interactive Ray Tracing**
  - **http://www.sci.utah.edu/RT06/**
  - 世界初のリアルタイムレイトレ学会

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

?

Contact: Syoyo Fujita
masahiro1.fujita@toshiba.co.jp

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

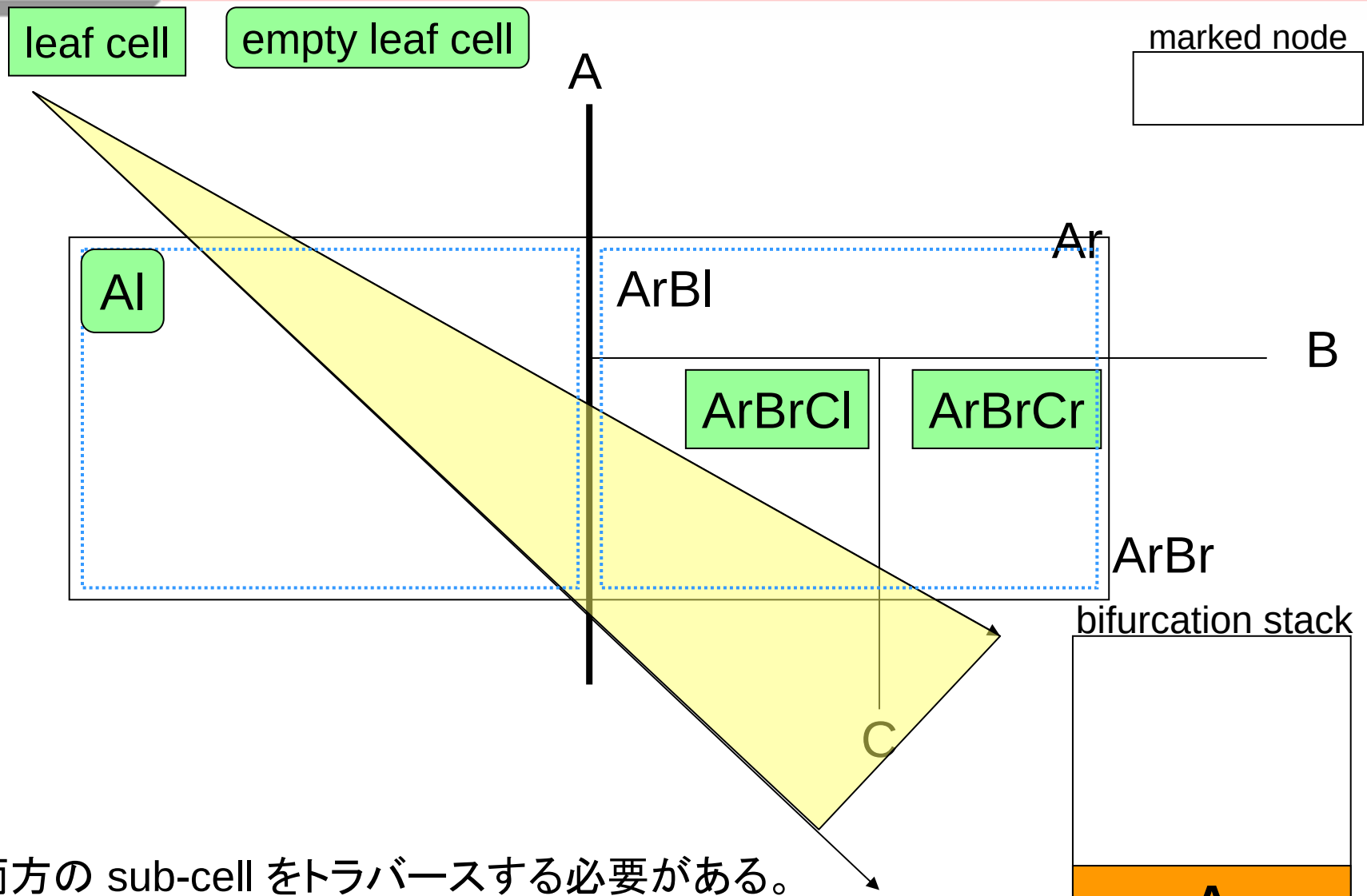
Thank you!

Appendix

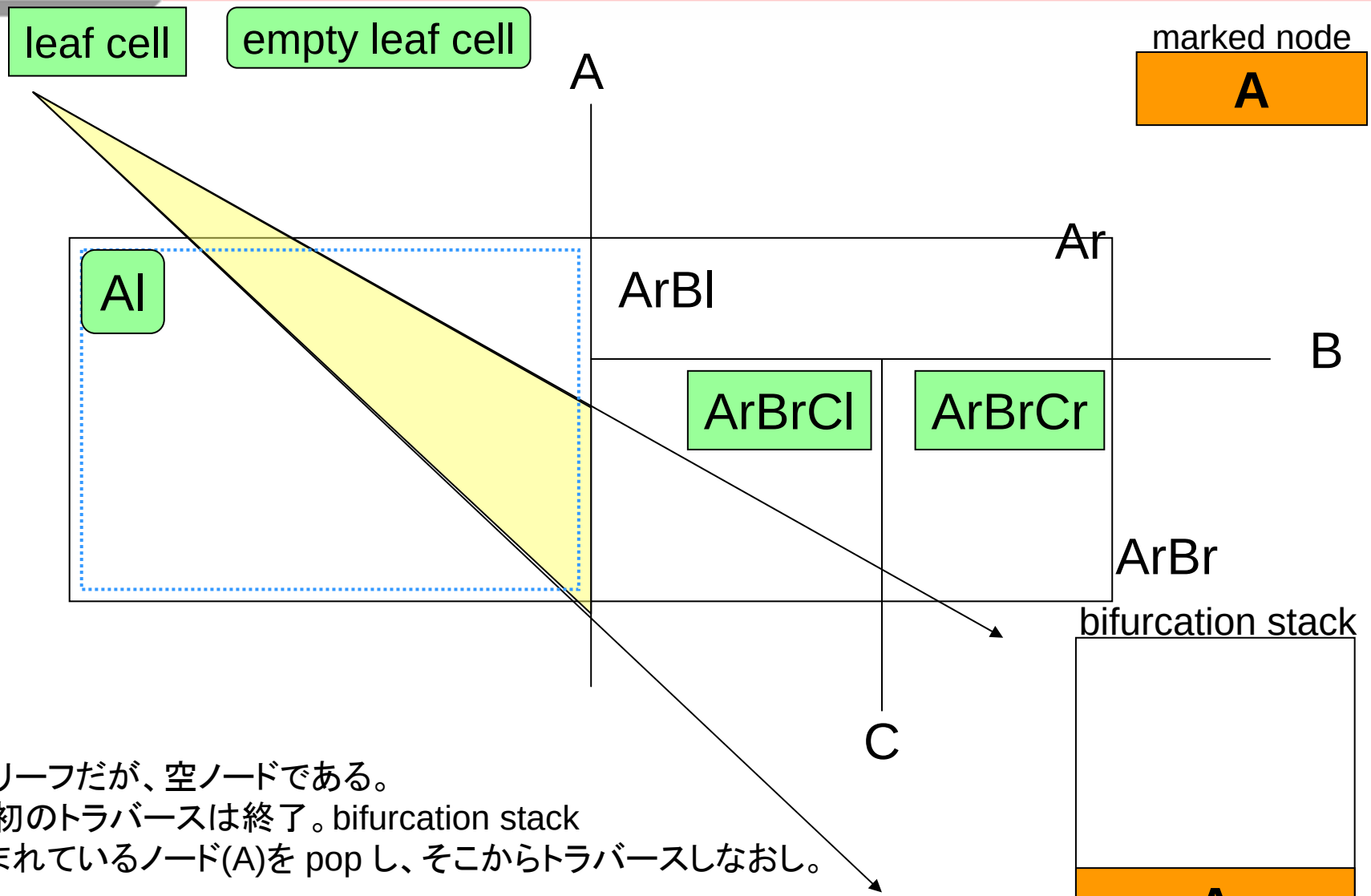
SPE プログラミング Tips

- なるべく処理をまとめる(粒度を上げる、コードパスをひとつにする)
  - SIMD 化
    - スカラデータに対する演算は苦手。スカラ演算をする場合、面倒でも一度 SIMD データにする
      - spu_splats() (スカラ -> ベクトル), spu_extract() (ベクトル -> スカラ) 命令を活用
    - データの変換(SoA <-> AoS など)
      - spu_shuffle() (シャッフル) 命令を活用
  - ブランチの除去
    - 演算であれば spu_cmpgt() + spu_sel() 命令を活用
    - case 文は関数ポインタ化(リーフ関数での case 文の関数ポインタ化は、関数呼び出しとのコストとの兼ね合い)
- ソフトウェアスレッディング + ソフトウェアキャッシュ + ダブルバッファリングでほとんどの DMA レイテンシは隠蔽できる
  - e.g. キャッシュミスしたらスレッド切り替えを使う
  - ただしそれでもソフトウェアキャッシュを通したアクセスは(ヒットするしないにかかわらず)一回につき数十サイクルはかかるので多用は厳禁
- インストラクションレベルの最適化をする
  - インオーダーなので、インストラクションの依存によるストールをなるべく減らす
    - spu-gcc_timing, simulator などを使う
    - コンパイラが出力するアセンブラと仲良くなる。
    - 豊富なレジスタ数を生かし、ループ展開やソフトウェアパイプライン化を行う
  - コンテキストスイッチによる割り込みや L1 cache miss などの不確定要素は無いので、性能の見積もりは行いやすい。
- 適切なコーディングを行えば、コンパイラが十分に最適化を行ってくれる
  - アセンブラで書くよりも多くの場合最適なアセンブリコードが得られる
  - 適宜生成されるアセンブリを確認しながらコンパイラの振る舞いを理解していくとよい

MLRTA EP search(Entry Point search)

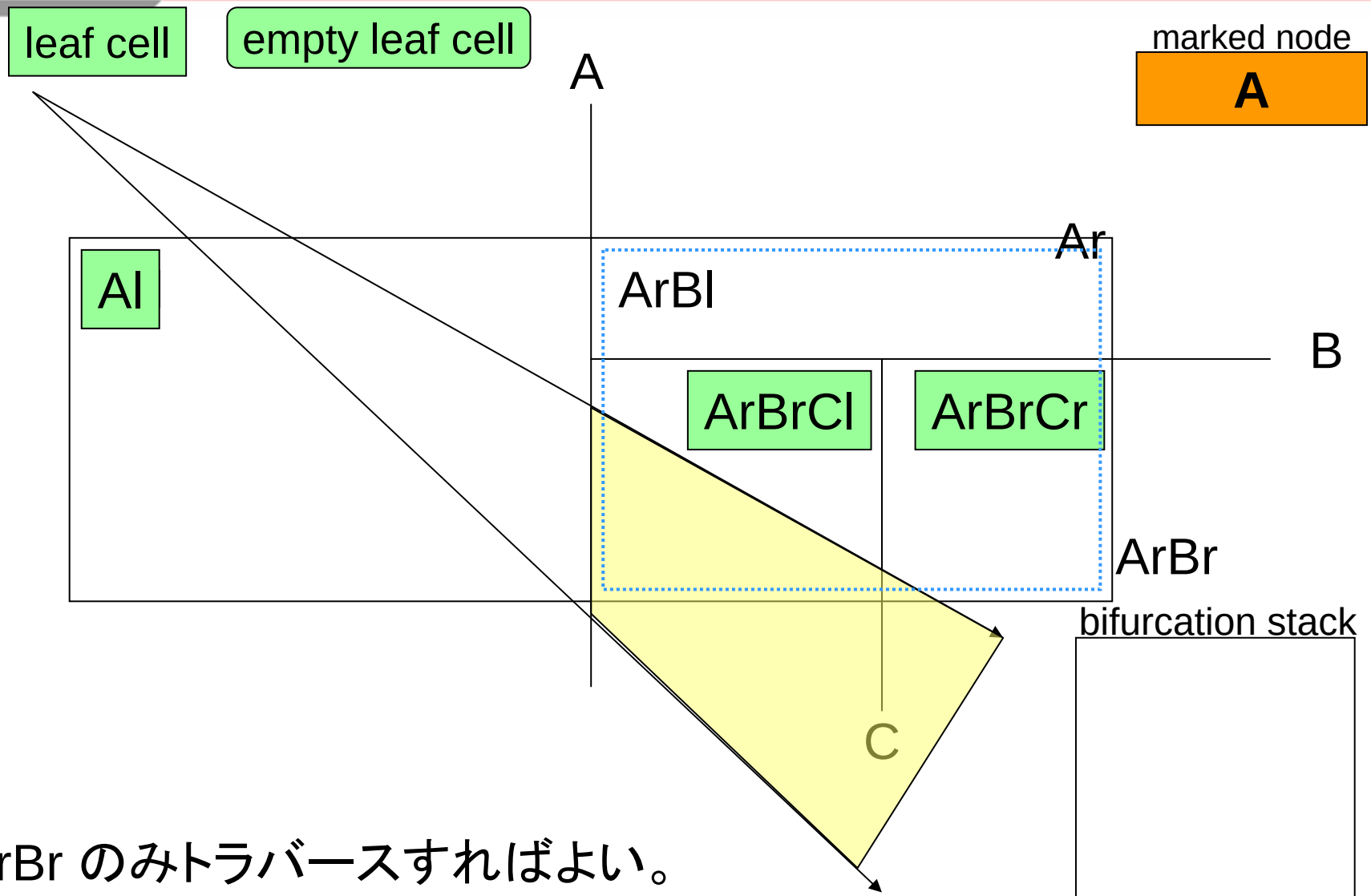


A: 両方の sub-cell をトラバースする必要がある。
-> A を bifurcation stack に積み次へ。

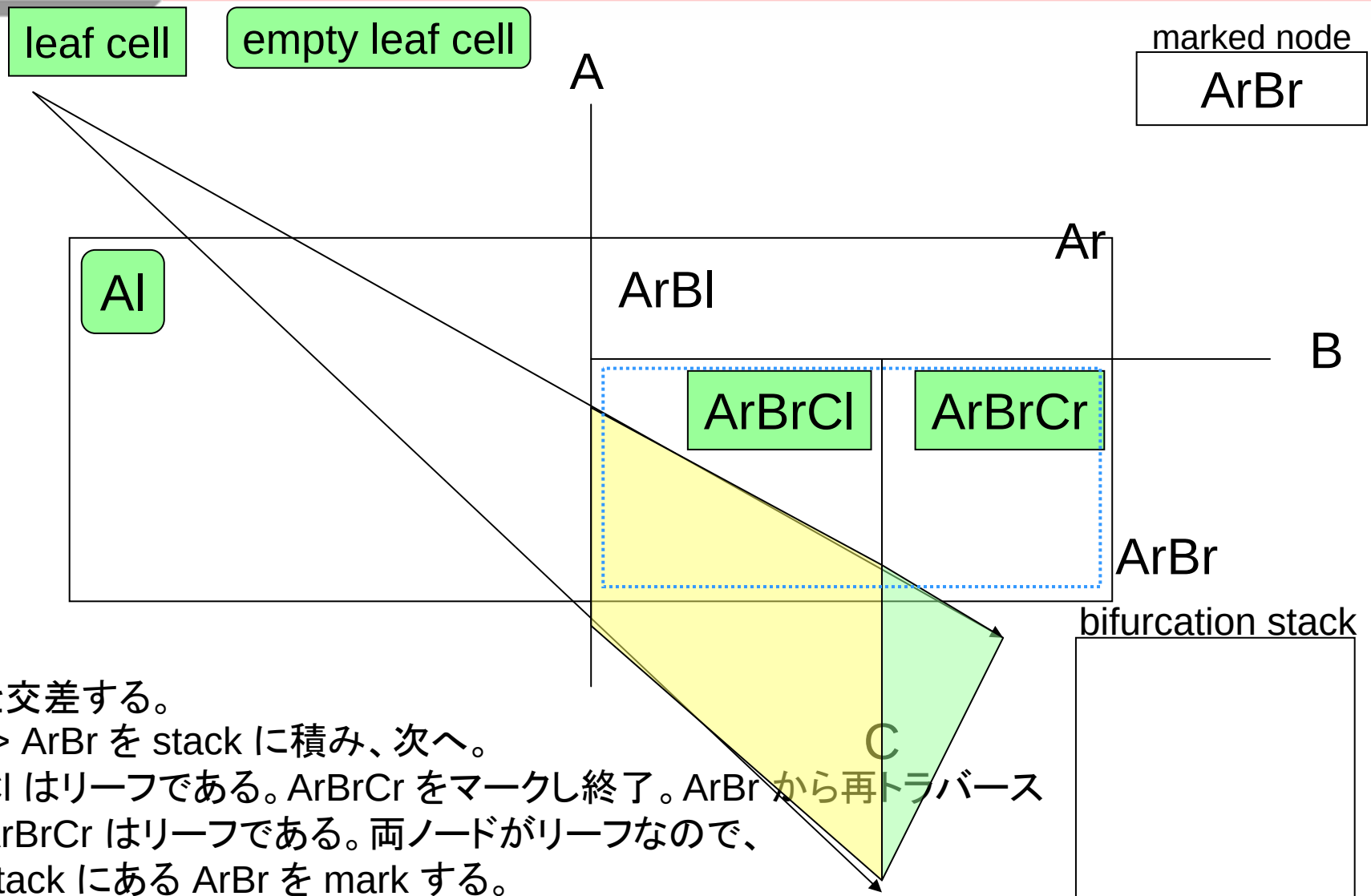


Al はリーフだが、空ノードである。
-> 最初のトラバースは終了。bifurcation stack
に積まれているノード(A)を pop し、そこからトラバースしなおし。

A を marked node とする。



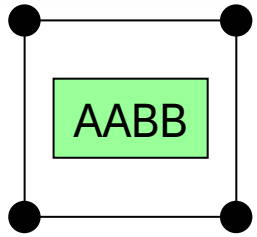
ArBr のみトラバースすればよい。
-> 次へ。



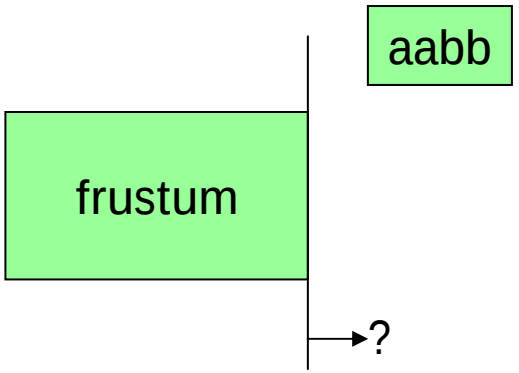
- 両方と交差する。
  - -> ArBr を stack に積み、次へ。
- ArBrCl はリーフである。ArBrCr をマークし終了。ArBr から再トラバース
  - ArBrCr はリーフである。両ノードがリーフなので、
  - stack にある ArBr を mark する。
- stack が空なので終了。現在 mark されている ArBr を Entrypoint とする。

Frustum-AABB culling

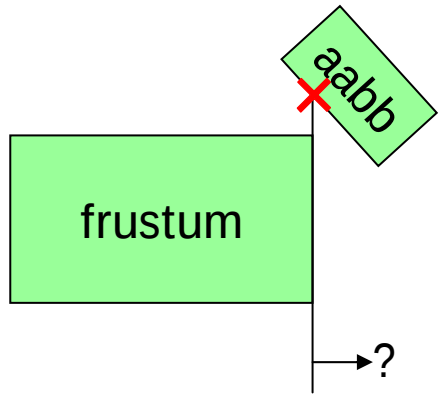
- frustum と aabb の交差判定
  - frustum = レイのパケットを覆う錐体
  - 結構やっかい
- 正確にやるにはクリッピングが必要
  - 処理量が大きい
- frustum を構成する平面の半平面でチェック
  - 処理量が小さいが、失敗するケースも



単純な 4 点テスト
で失敗するケース



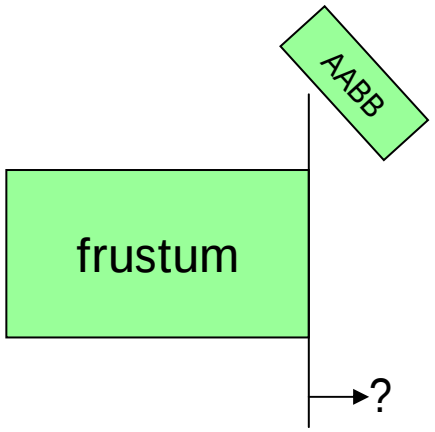
frustum の平面でチェック



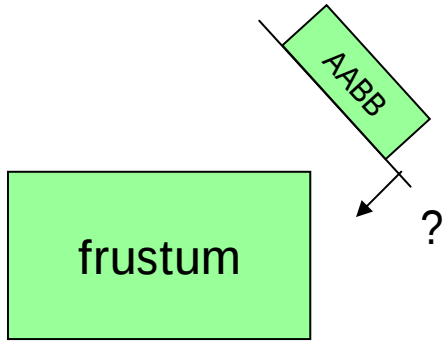
失敗するケース

Reverse frustum-AABB test

- Reverse frustum-AABB test
  - [Reshetov 2005]
  - frustum を AABB とみなし、AABB を frustum とみなす。
    - AABB の平面でチェック
    - 通常 frustum の方が AABB に比べて小さいので、失敗するケースが少なくなる。
    - トラバースオーダーの決定に有益(2D での問題に帰結されるので)



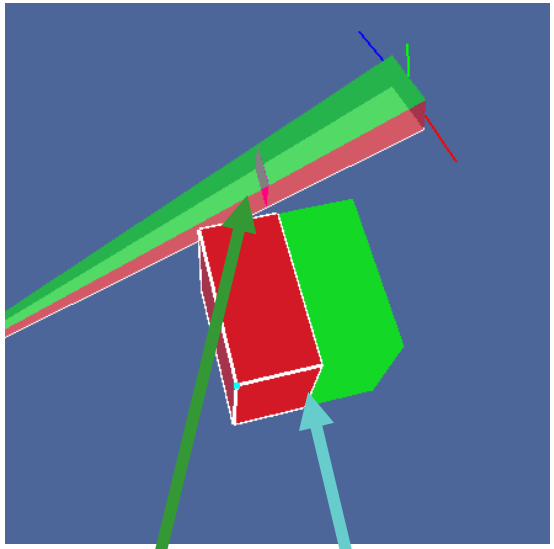
失敗するケース



reverse AABB-frustum test

トラバースオーダーの決定

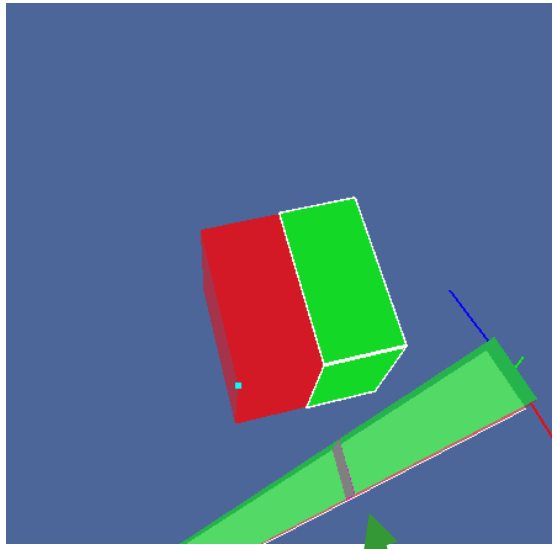
near



frustum
の射影面

kdtree
分割面

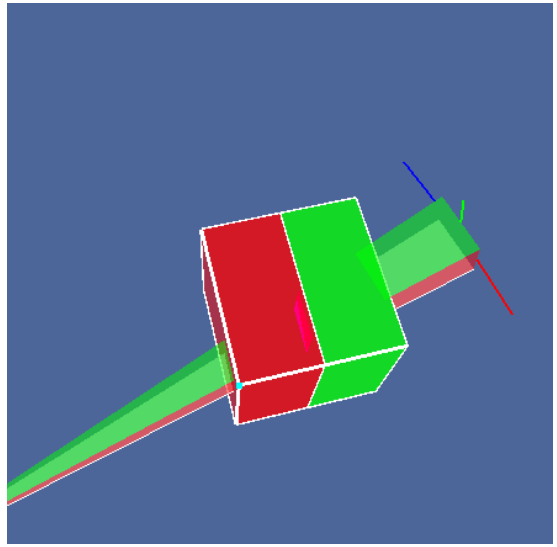
far



kdtree
分割面

frustum
の射影面

both



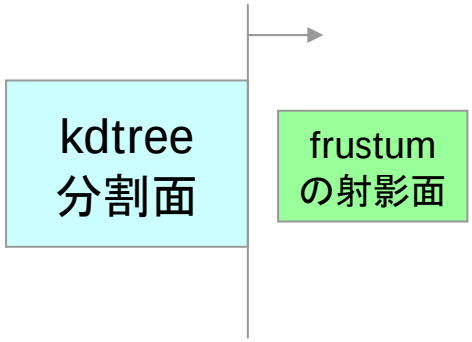
frustum
の射影面

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

- kd-tree の分割面の矩形と、その平面への frustum の射影矩形の位置関係で、どちらの子をたどればよいかが分かる

重なりがある = both

重なりが無い場合

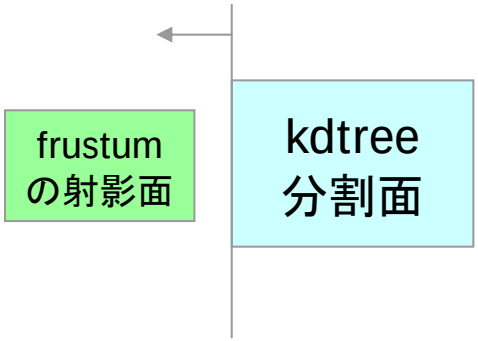


sign(u)	+	-
traverse	near	far

sign(v)	+	-
traverse	near	far

sign(i): frustum の方向ベクトルの i 軸の符号

u = (射影軸 + 1) % 3
v = (射影軸 + 2) % 3

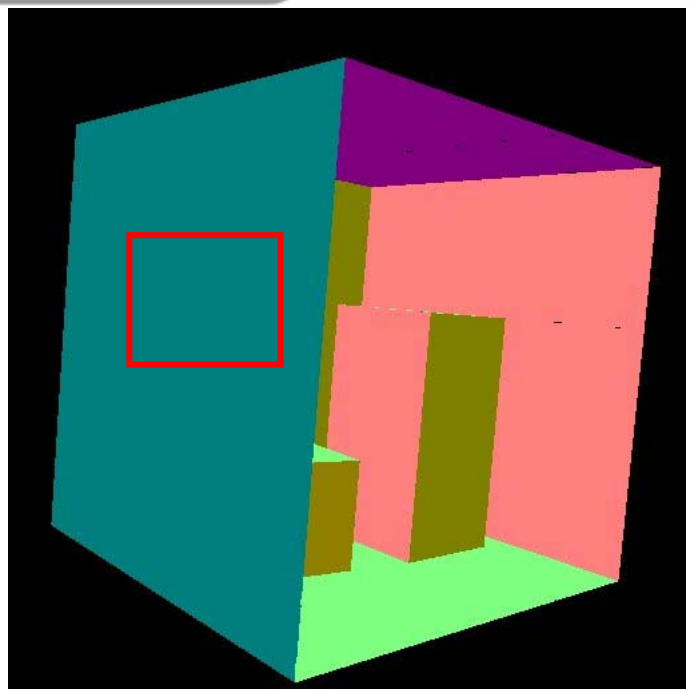


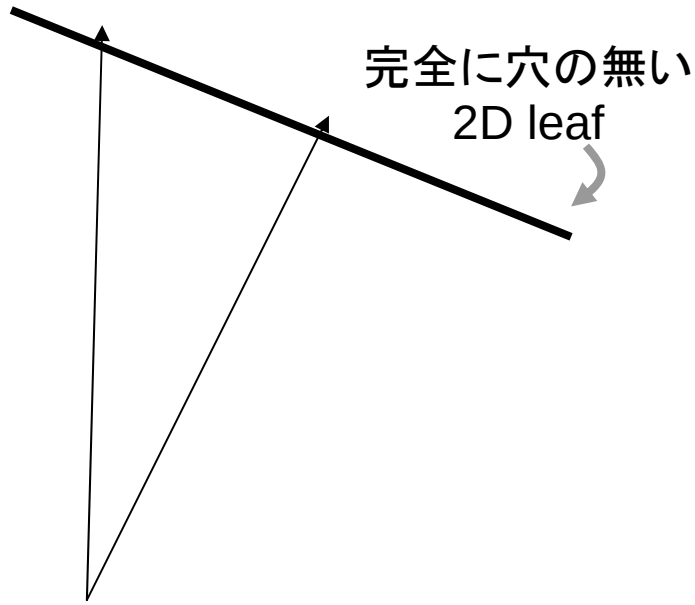
sign(u)	+	-
traverse	far	near

sign(v)	+	-
traverse	far	near

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

EP search entering a 2D leaf





完全に穴の無い 2D leaf に
EP seach が最初にヒットするなら、
必ずレイはこの壁(2D leaf)にヒットし、
他の遮蔽物が無いことが保障される

穴の無い 2D leaf かどうかのテストには、
General Polygon Clipper [GPC] などが使える

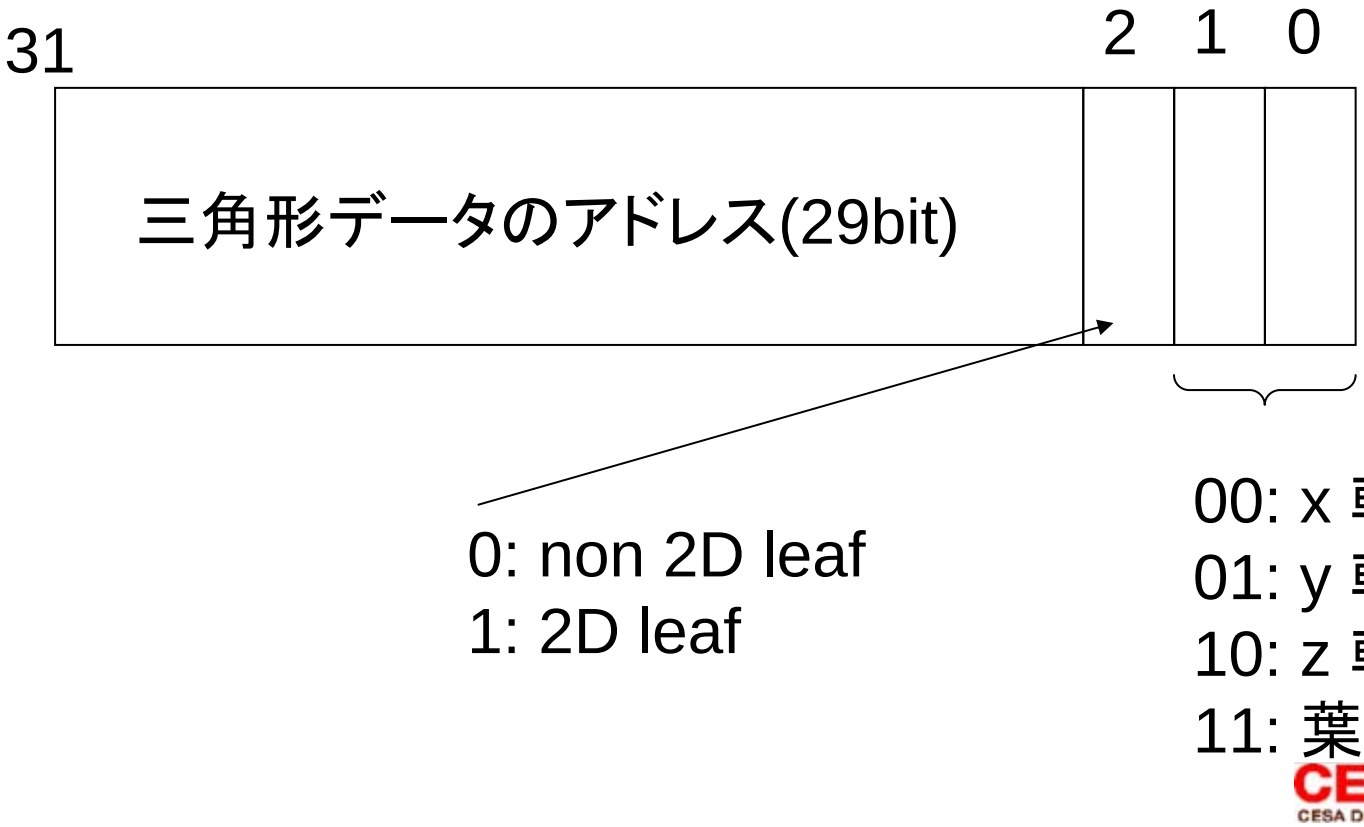
→

錐体の 4 つのカドでのみ
レイをトレースし、結果を
補間することで高速化が図れる
(とはいえ補間もそれなりにコストが
かかるよ by Reshetov[private
communication])

CEDEC 2006
CESA DEVELOPERS CONFERENCE 2006

ノードデータ構造の変更

- さらなるコンパクトノードフォーマット
  - 1bit を完全に遮蔽されている 2D ノードかどうかの判定に使う



ノードデータの取得

- ノード情報の取得

```
#define KDNODEAXIS(node) ((node) & 0x3)
```

- 三角形データのアドレスの取得

```
#define KDOBJINDEX(node) ((node) >> 3)
```

- 分割面の取得

```
inline float kdcutplane (unsigned int node)
{
    union
    {
        unsigned int    i;
        float           f;
    } d;

    unsigned int tmp;
    float           ret;

    tmp = node & 0xfffffffc;

    d.i = tmp;
    ret = d.f;

    return ret;
}
```


交差判定コード例

```
hitrec := (t, u, v, tid); hitinfo := (t, u, v, tid, normal[3], st[3])
rox, roy, roz:= ray の原点; rdx, rdy, rdz:= ray の方向;
validmask:= 有効なレイかどうか; tid_step = (4, 4, 4, 4);

for (k = 0; k = nSIMDtriangles / bufsize; k++) {// bufsize = DMA で一度に取得する三角形データの個数
    tridata = ダブルバッファで三角形データを LS に DMA 転送
    for (m = 0; m < nSIMDrays; m++) {
        triptr = tridata;
        hitrec[0..3] = 初期値に設定;
        rox = ray4[m].rox; roy = ray4[m].roy; roz = ray4[m].roz;
        rdx = ray4[m].rdx; rdy = ray4[m].rdy; rdz = ray4[m].rdz;
        validmask = ray4[m].validmask;
        tid = (1, 2, 3, 4);

        for (i = 0; i < bufsize; i++) {
            for (j = 0; j < 4; j++) { // 実際には look unroll します
                isect(hitrec[j], rox, roy, roz, rdx, rdy, rdz, triptr, tid, validmask); // IBM のサイトにある Cell
                    SDK 参照
                // レイデータを rotate
                rox = spu_rlqwbyte(rox, 4); roy = spu_rlqwbyte(roy, 4);
                roz = spu_rlqwbyte(roz, 4); rdx = spu_rlqwbyte(rdx, 4);
                rdy = spu_rlqwbyte(rdy, 4); rdz = spu_rlqwbyte(rdz, 4);
            }

            triptr += TRIANGLE_SIZE;
            tid = spu_add(tid, tid_step);
        }
        // 現在のレイの交点よりも、hitrec[0..3] の交点のいずれかが近い場合は、その近い交点にアップデートする。
        // また交点をアップデートする場合は、法線と UV 座標もアップデートする
        ray4[m].curr_hitinfo = update_hitrecord(ray4[m]. curr_hitinfo, hitrec[0], hitrec[1], hitrec[2], hitrec[3],
            tridata);
    }
}
```


Traversal routine.

```
#define likely(x)     __builtin_expect((x), 1)
#define KDNODEAXIS(node) ((node) & 0x3)

int
traverse_threaded_simd(raythread_t *thread) {
    const vector float veps = spu_convtf(spu_splats(1), 16);
    unsigned int curr_node, unsigned int near_node_idx, far_node_idx;
    vector float t;
    vector float vorgpos, vinvdirpos, vcut_plane;
    vector unsigned int vsign, vnear_node_idx, vfar_node_idx;
    unsigned int  cut_axis;
    float         cut_plane;
    int           isleaf = 0, hit;

    while (thread->curr_node_index < gnnodes) {
        curr_node = gkdnodes[thread->curr_node_index];
        cut_axis = KDNODEAXIS(curr_node);
        isleaf = (cut_axis == KDTREE_LEAFNODE) ? 1 : 0;
        if (likely(isleaf)) {   // 分岐予測付き
            // スレッドのステートを交差判定の準備へ
            isect_init_simd(thread);
            return 1;
        } else {
            // kd-木をさらにトラバース

            cut_plane = kdcutplane(curr_node);
            vcut_plane = spu_splats(cut_plane);
            vorgpos    = thread->vrayorg[cut_axis];
            vinvdirpos = thread->vrayinvdir[cut_axis];
```

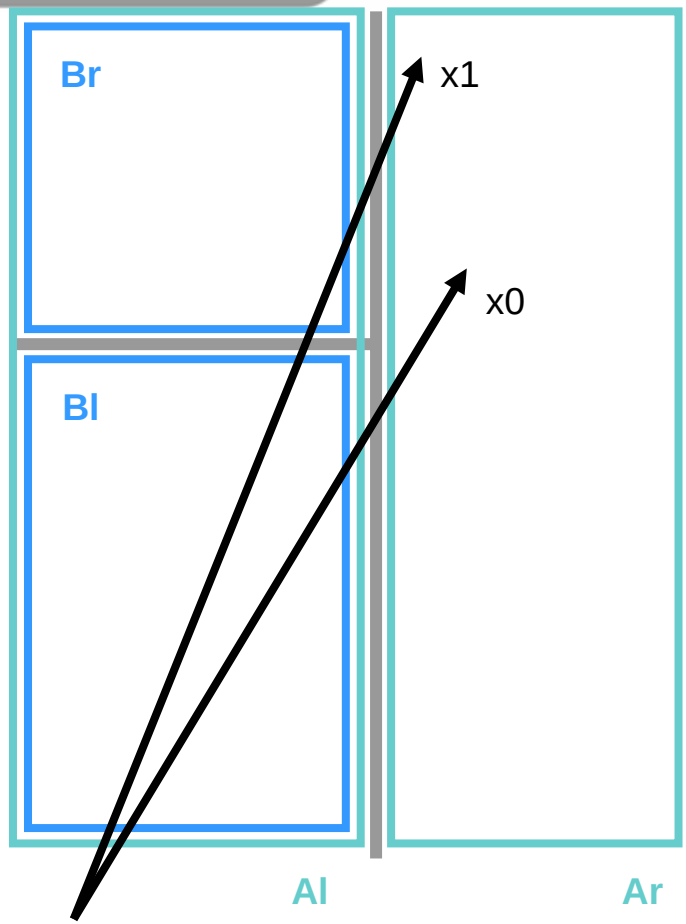


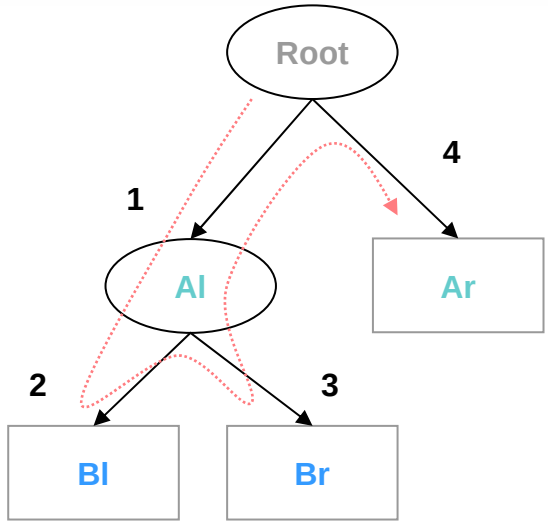
```
// t = (cut_plane - orgpos) * invdirpos;
t = spu_mul(spu_sub(vcut_plane, vorgpos), vinvdirpos);

near_node_idx = (thread->curr_node_index << 1) + 1; // right child.
far_node_idx  =  thread->curr_node_index << 1    ; // left child.

// acitve[i] = (t_near[i] < t_far[i])
thread->vactive   = spu_cmpgt(thread->vmaxt,
                              thread->vmint);

vector unsigned int vdeactive;
vdeactive = spu_xor(thread->vactive,
                    spu_splats(0xffffffff));

vector float org_plus_eps;
vector float plane_plus_eps;
vector unsigned int cond0, cond1, cond2;
vector float vmint_minus_eps;
vector float vmaxt_plus_eps;
unsigned int ret0, ret1;

org_plus_eps   = spu_add(vorgpos, veps);
plane_plus_eps = spu_add(vcut_plane, veps);

if (thread->raysign[cut_axis]) {
    near_node_idx = thread->curr_node_index << 1;
    far_node_idx  = (thread->curr_node_index << 1) + 1;
}

vmint_minus_eps = spu_sub(thread->vmint, veps);
vmaxt_plus_eps = spu_add(thread->vmaxt, veps);
```



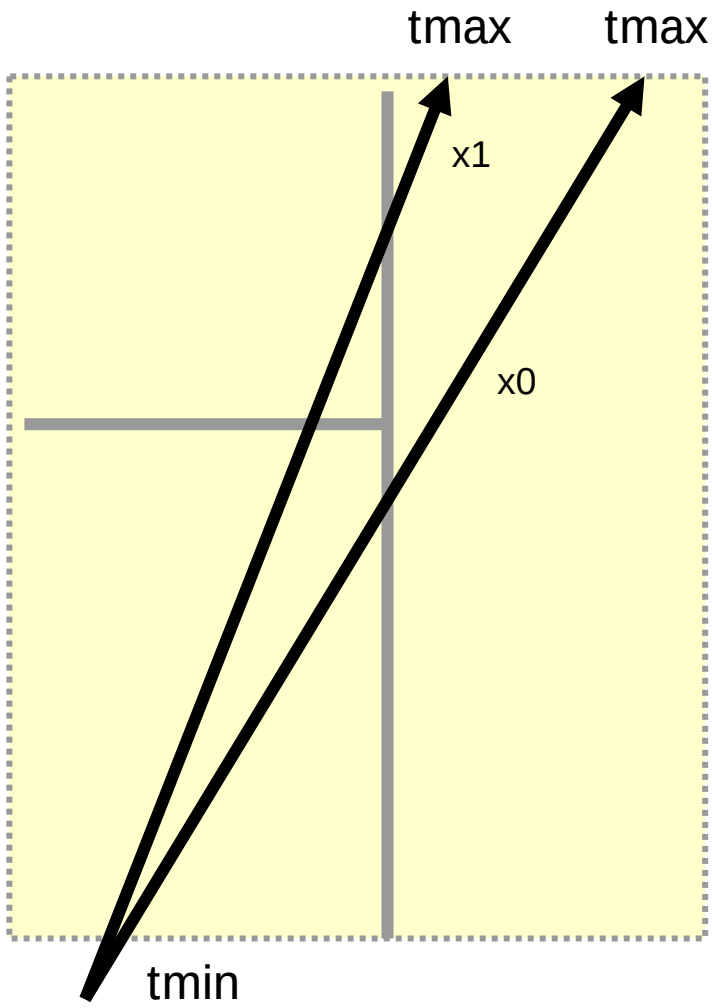
```
            cond0 = ~spu_cmpgt(t, vmint_minus_eps);
            cond1 = ~spu_cmpgt(vmaxt_plus_eps, t);
            cond2 = spu_cmpgt(thread->vmint, t);

            // ret0 = count(cond0 || !active).
            ret0 = spu_extract(spu_gather(spu_or(cond0, vdeactive)), 0);

            // ret1 = count(cond1 || !active).
            ret1 = spu_extract(spu_gather(spu_or(cond1, vdeactive)), 0);

            if (ret0 == 0xf) {
                thread->curr_node_index = far_node_idx;
            } else if (ret1 == 0xf) {
                thread->curr_node_index = near_node_idx;
            } else {
                thread->stackptr++;
                thread->stack[thread->stackptr].vrayMinT =
                    spu_sel(t, thread->vmint, cond2);
                thread->stack[thread->stackptr].vrayMaxT = thread->vmaxt;
                thread->stack[thread->stackptr].nodeidx = far_node_idx;

                thread->curr_node_index = near_node_idx;
                thread->vmaxt = spu_sel(t, thread->vmint, cond2);
            }
        }
    }

    thread->state = RAY_THREAD_STAT_TRAV_DONE;
    return 0;
}
```

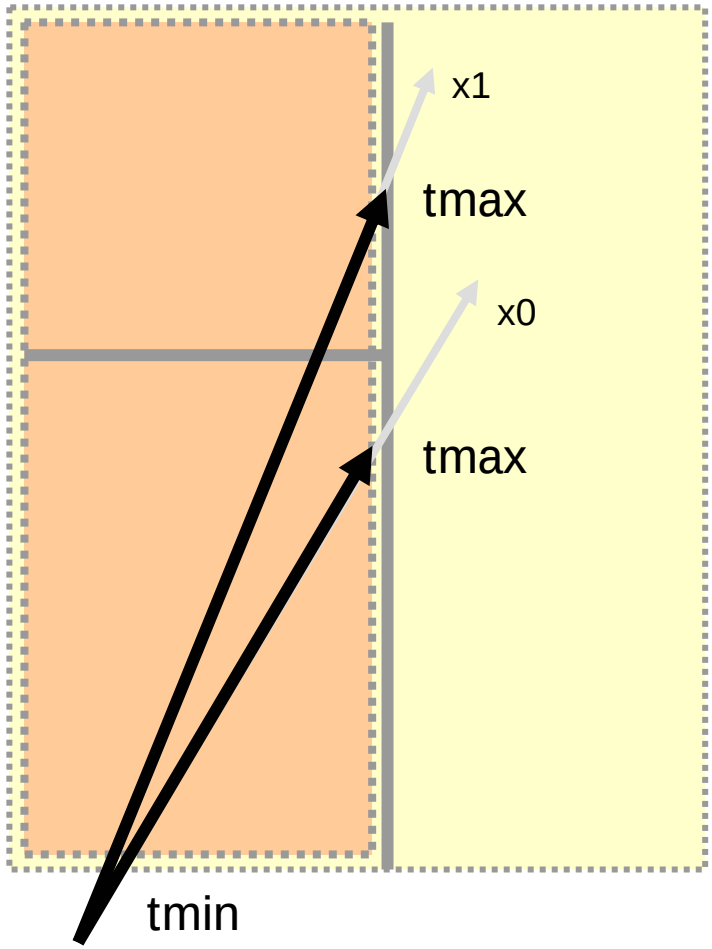

マスク付トラバース

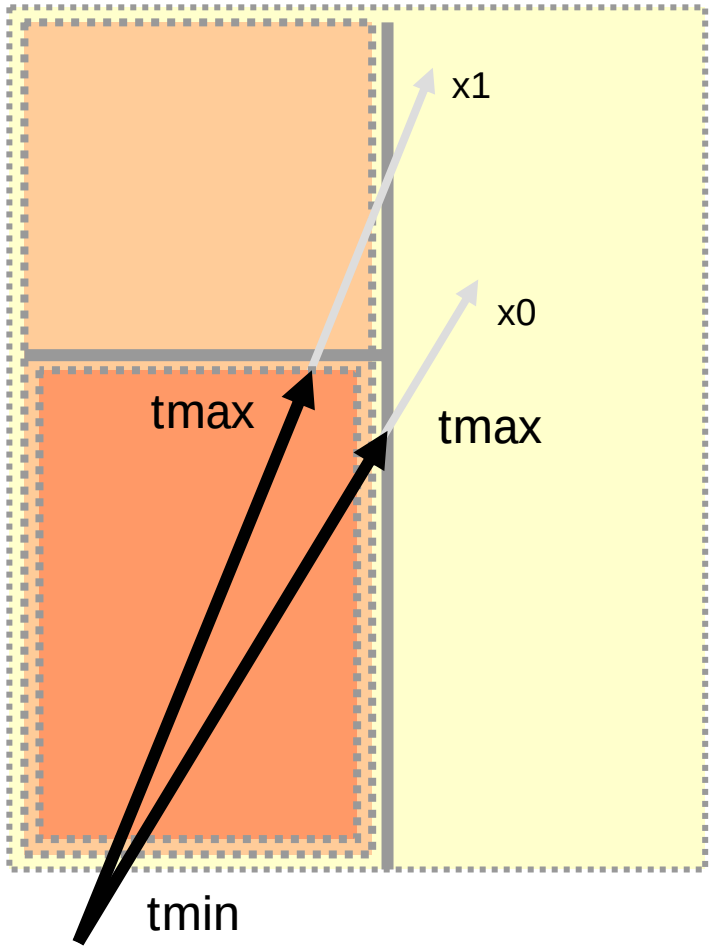


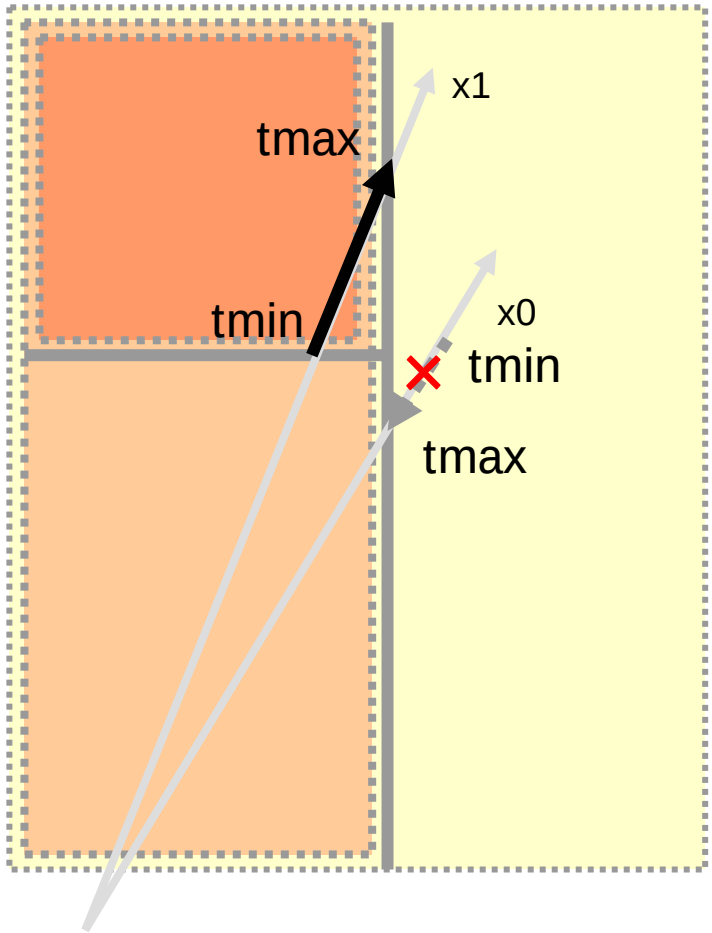


Order	Mask (x0, x1)
root	(valid, valid)
1(Al)	(valid, valid)
2(Bl)	(valid, valid)
3(Br)	(**invalid**, valid)
4(Ar)	(valid, valid)









レイ x0 は
無効な区間　(tmin > tmax)
にあるので、invalid にする。