エンジニアリング

『GRANBLUE FANTASY: Relink』ソフトウェアスタライザによる実践的なオクルージョンカリング

株式会社Cygames　コンシューマー ゲームエンジニア / 大河原 昭

Cygames

自己紹介

コンシューマー

大河原 昭

前職にてミドルウェア開発に従事した後、2017年にCygamesに合流。
以後、グラフィックスエンジニアとしてゲーム開発に従事。

本セッションで伝えたいこと

- Masked Software Occlusion Cullingの仕組み
- 組み込み時のノウハウ
- パフォーマンスと活用事例

アジェンダ

1. オクルージョンカリングとは？
2. 既存手法の紹介
3. Masked Software Occlusion Cullingの紹介
4. Relinkでの導入と最適化
5. オクルージョンカリングの効果と負荷
6. まとめ

オクルージョンカリングとは？

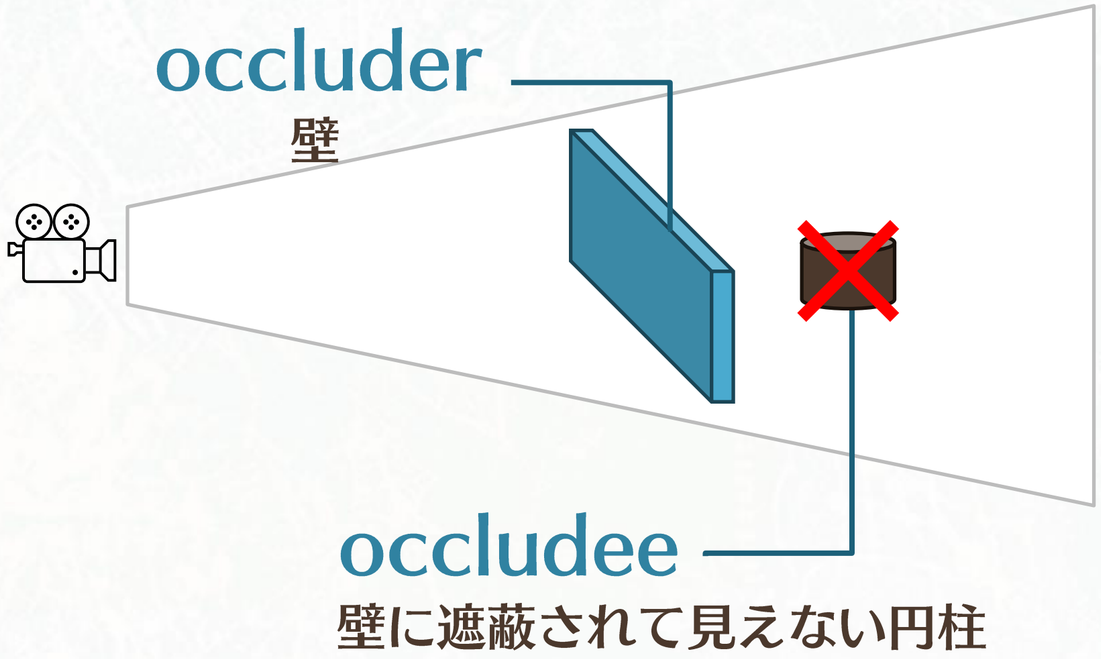
遮蔽されたオブジェクト描画を省く

occluder

- 遮蔽するオブジェクト

occludee

- 遮蔽されているか
  チェックするオブジェクト



Occludeeがoccluderで遮蔽されているかチェックする処理

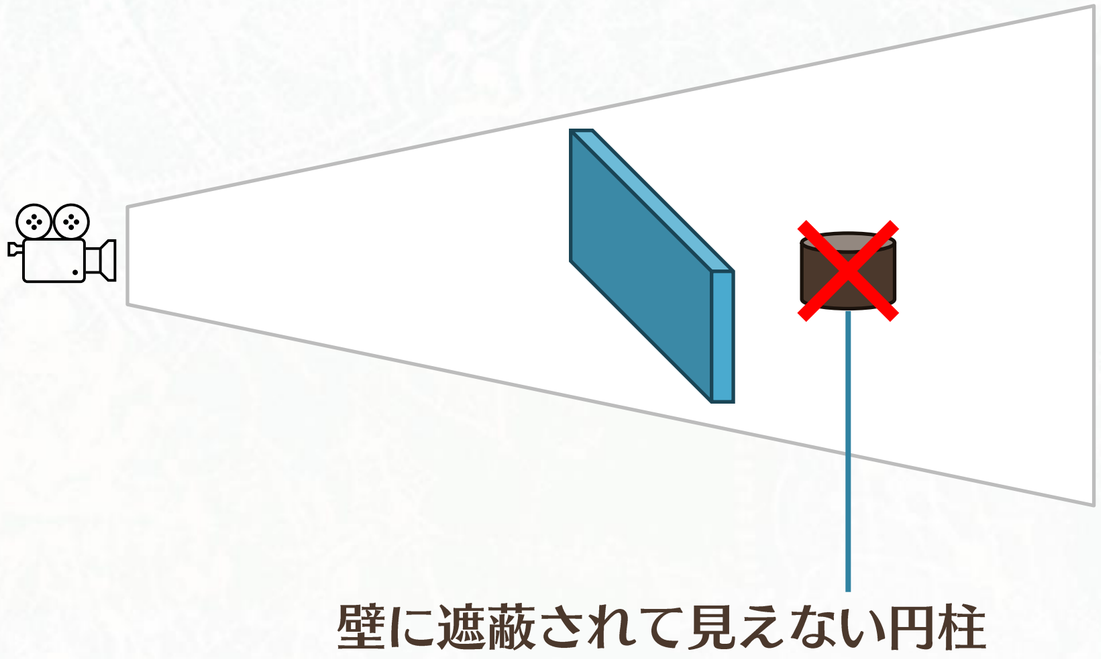
オクルージョンカリングの効果

GPU

- 不要なメッシュの処理を抑制

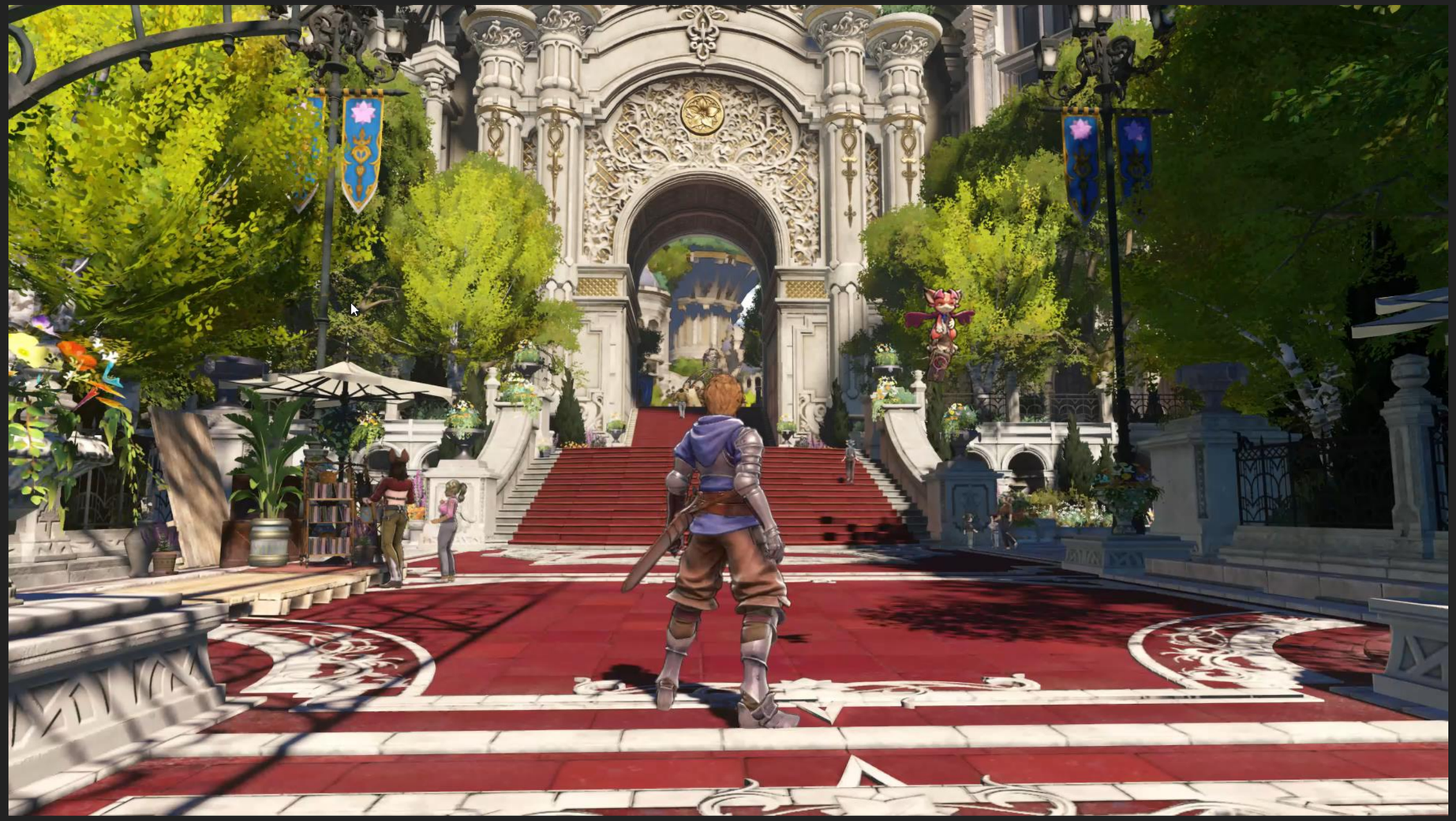
CPU

- ドローコールの抑制
- 不要なリソース更新抑制



画面に関係ない処理を省いて最適化

実際のオクルージョンカリングの様子



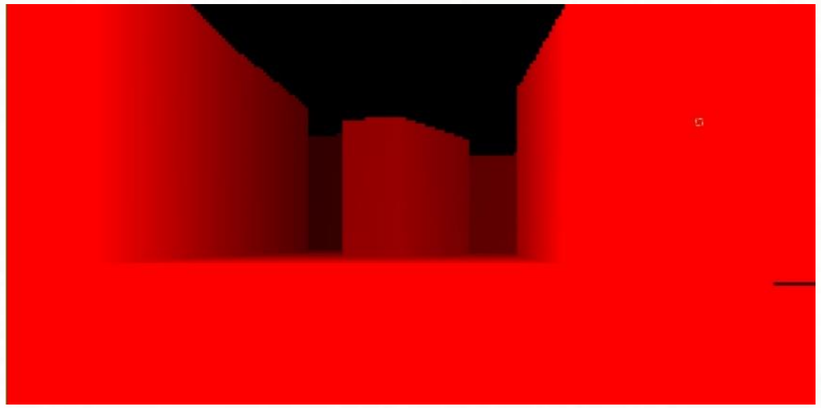
アジェンダ

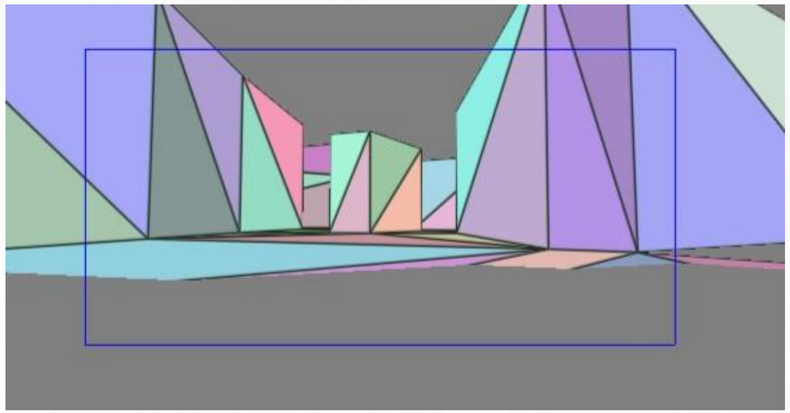
1. オクルージョンカリングとは？
2. 既存手法の紹介
3. Masked Software Occlusion Cullingの紹介
4. Relinkでの導入と最適化
5. オクルージョンカリングの効果と負荷
6. まとめ

既存手法の紹介

GPUでオクルージョンカリング

- 前フレームのdepth bufferを利用[Mat21]
  - 新しいリソース計算なしでもできる
  - ポッピング対策が必要
- 縮小z bufferにラスタライズしてテスト[三嶋 2018]
  - 追加でGPUに負荷





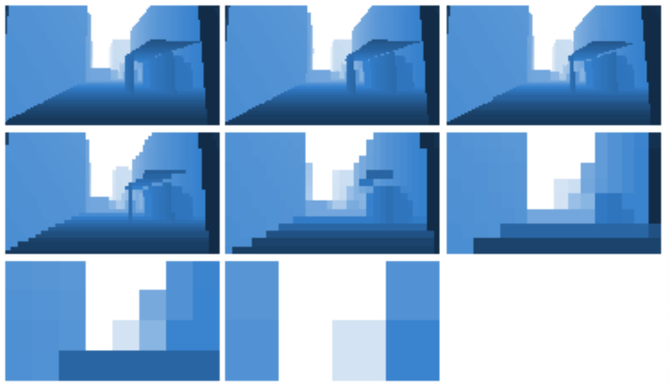
CPUでオクルージョンカリング

幾何学的な手法

- 実装し易い
- 板ポリやボックスなどの
  単純な形状に限る

ソフトウェアラスタライザ

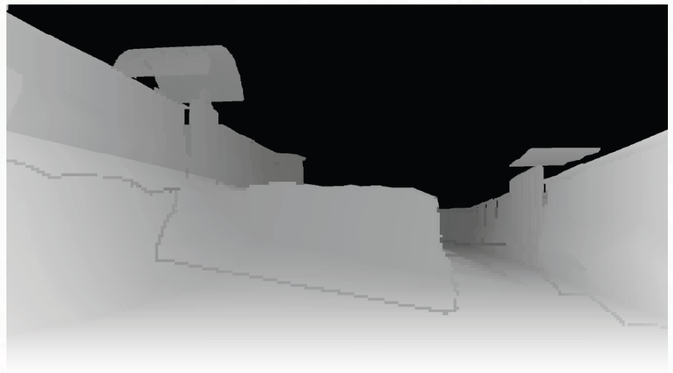
- Bounding boxなど
  メッシュをラスタライズしてテスト
- 比較的重い
- HiZによる高速化
  - Depth bufferからmaxでミップを作る[Ste11]



Relinkのオクルージョンカリング

Masked Software Occlusion Culling[HAAM16]

- CPUで完結
  - GPUの負荷と結果待ち無
  - カリング結果を用いて他のCPU処理を高速化
- 高速なソフトウェアラスタライザ
  - 更新負荷が軽い depth bufferフォーマット
  - 計算量的に有利
    - occluderの数*occludeeの数に比例しない
- 組み込みが簡単
  - Intelがソースコードを公開



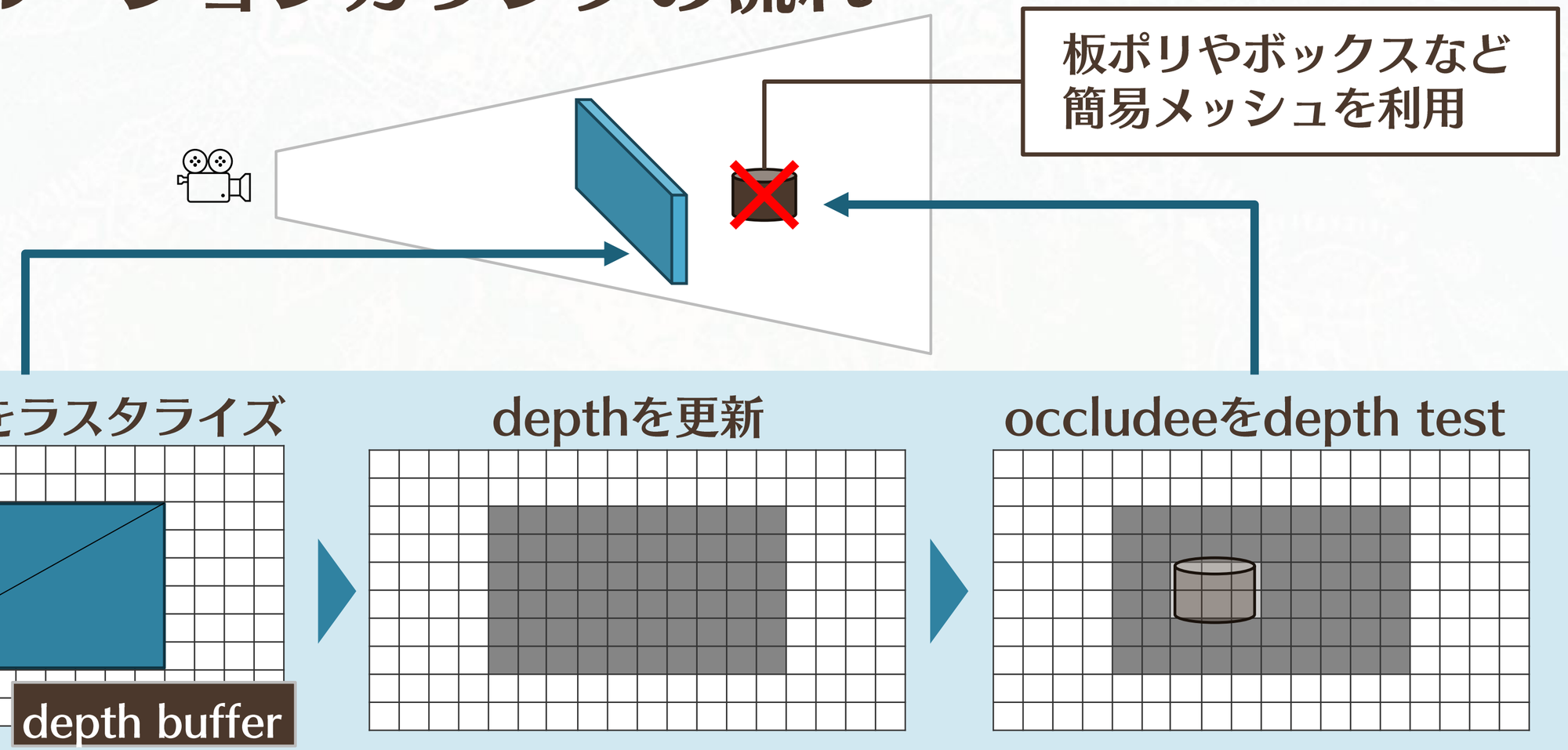
アジェンダ

1. オクルージョンカリングとは？
2. 既存手法の紹介
3. Masked Software Occlusion Cullingの紹介
4. Relinkでの導入と最適化
5. オクルージョンカリングの効果と負荷
6. まとめ

Masked Software Occlusion Cullingの紹介

ソフトウェアラスタライザ

オクルージョンカリングの流れ



本手法の特徴

特徴

フル解像度 coverage mask
Hierarchical depth buffer

↓

利点

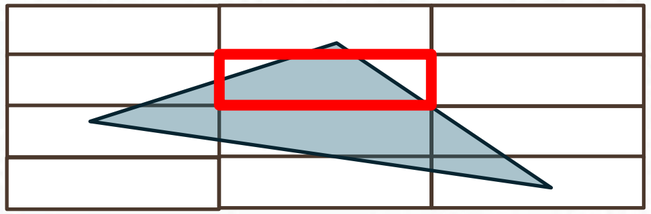
メモリのオーバーヘッドが低い
高精度なオクルージョンカリング
SIMD命令で実装しやすい

SIMD命令

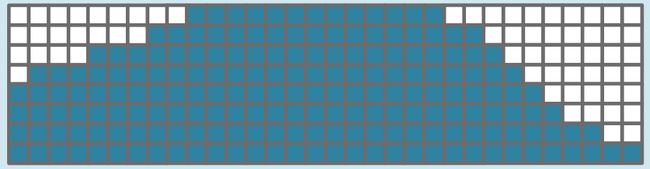
- １回で複数データの演算を行える
- ベクトル演算やタイルごとの処理と相性が良い
- パックした8つのfloatを扱う組込関数の例

演算	関数
乗算	__m256 _mm256_mul_ps(__mm256 m1, __mm256 m2);
最上位ビット抽出	int _mm256_movemask_ps(__mm256 a);
要素の入れ替え	__mm256 _mm256_suffle_ps(__mm256 m1, __mm256 m2, const int select);

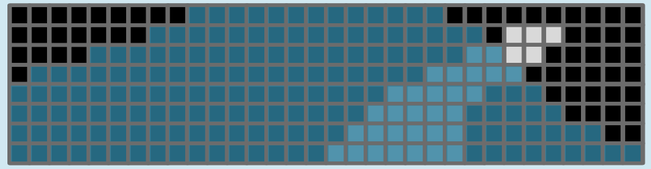
Depth test/updateの流れ



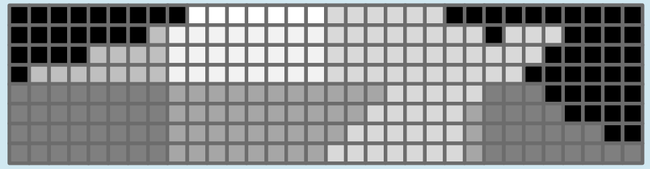
頂点位置からタイルを決定



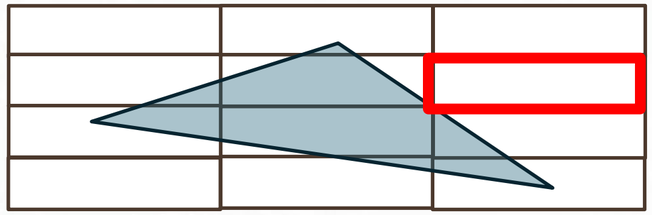
Coverage maskを計算



Coverage maskを用いて
Depth test/update



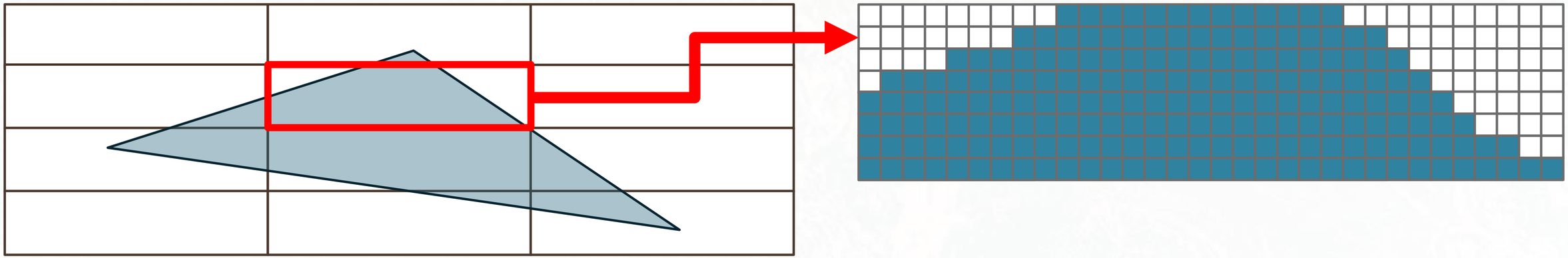
updateの場合は、
hierarchical depth bufferを更新



次のタイルへ

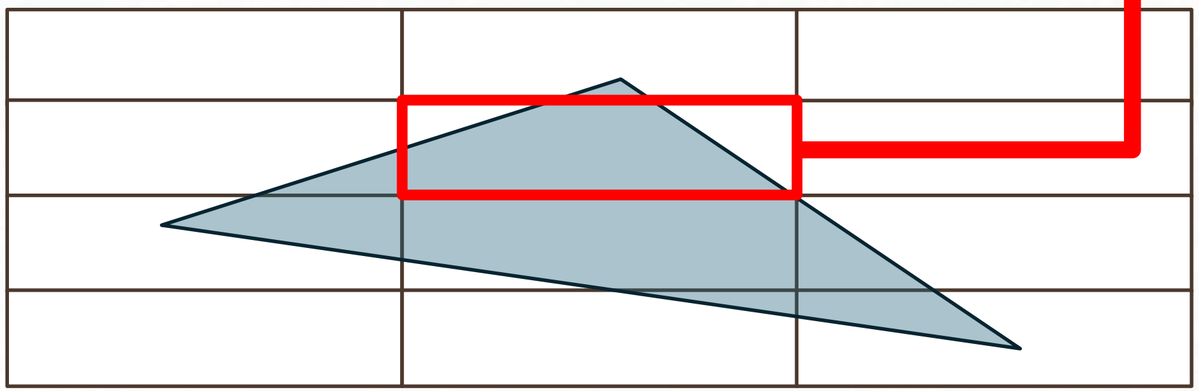
フル解像度 Coverage mask

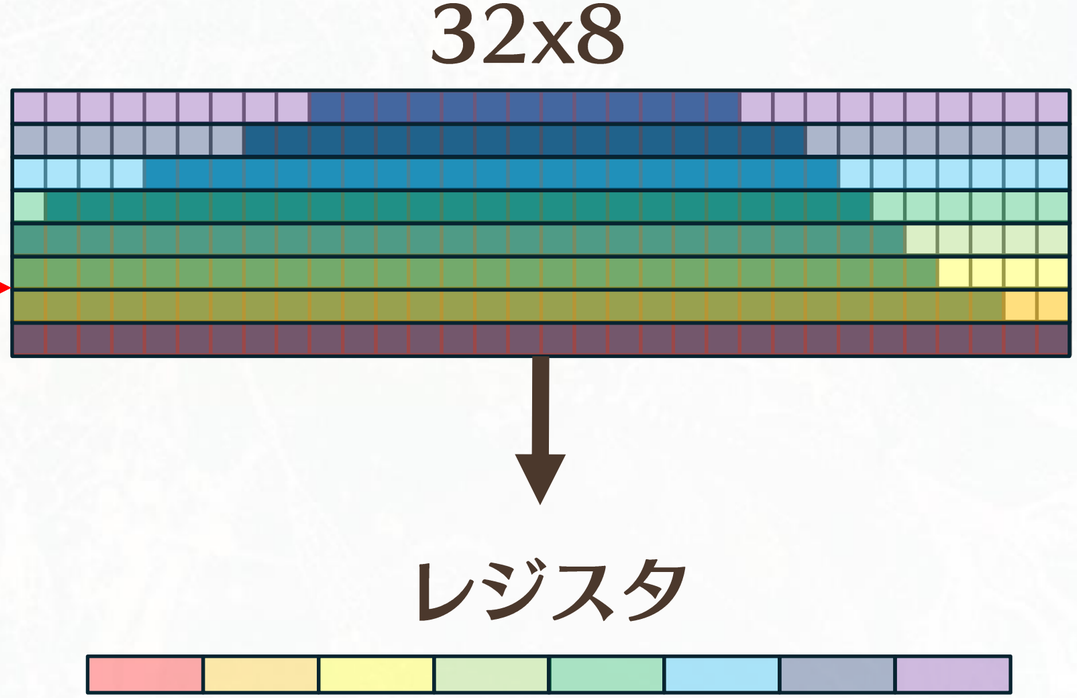
- ポリゴンがラスタライズされたか
  - １ピクセル 1bit, スクリーンと同じ分解能
  - タイル分割
- ピクセル単位のテストが可能
  - 高精度なオクルージョンカリング



Coverage maskとSIMD命令

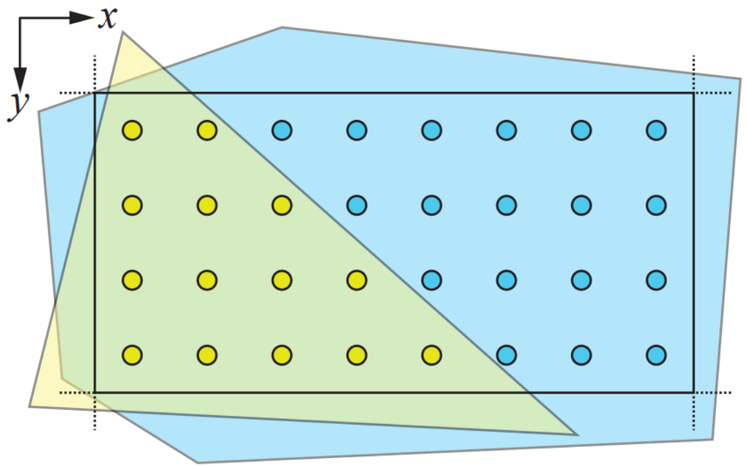
- タイル分割して管理
  - AVX2なら32x8ピクセルが1タイル→256bitマスク
- 1命令で1タイル処理
  - 後段の処理でマスクとして活用

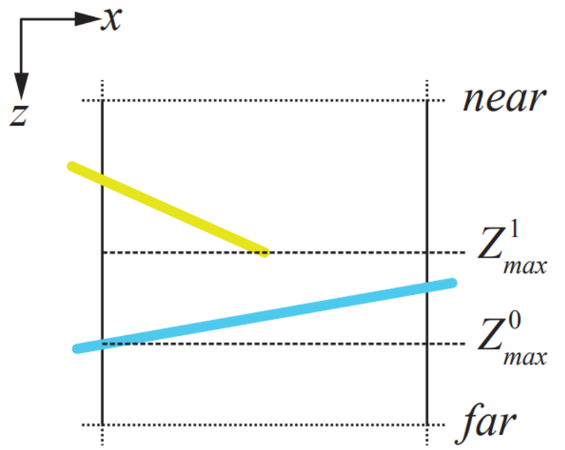




Hierarchical depth buffer

- 8x4ピクセルが 1 tile
  - ポリゴン２枚分の情報
  - Z maxが２つ
    - Z maxはポリゴン中の最深度
    - ポリゴン２枚分(Z0, Z1)
  - Z0/Z1マスク
    - Z0かZ1か
    - 1bit
    - Coverage maskと同じく
      フル解像度





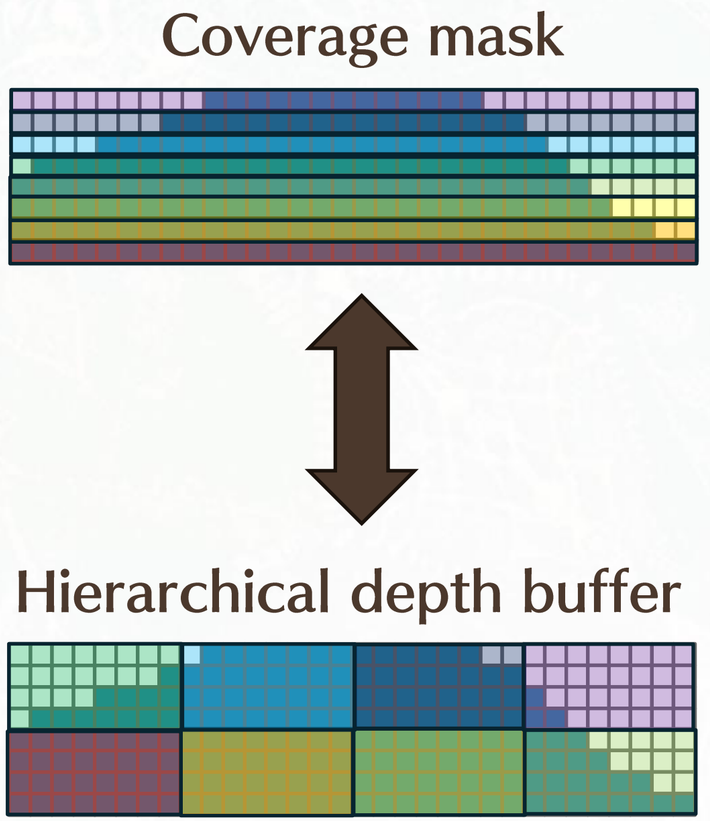
Hierarchical depth bufferとSIMD命令

◆256bit長命令と相性が良い

- それぞれ32bit
  - Z0max, Z1max, mask(1bit 8x4)
- それぞれ8個揃えると256bit
  - SoAとして扱う

◆Coverage maskとは異なる

- 処理する際は
  インデックスのシャッフルが必要

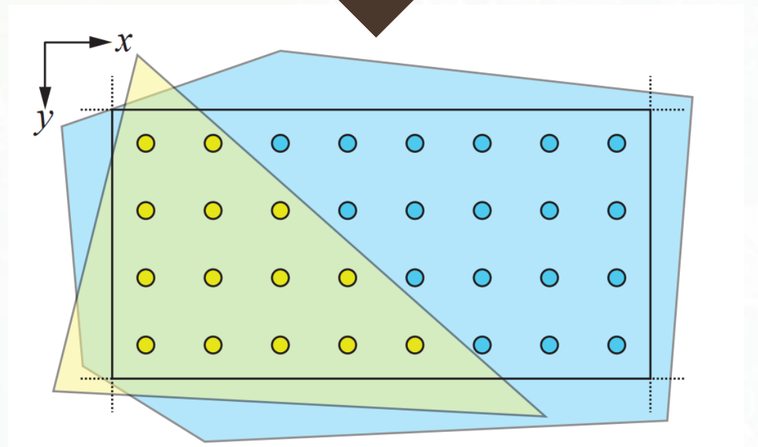


Depth test/depth buffer update

- タイルごとスキップ
  - 不要な処理をスキップ
- タイルの２段階Z max
  - Depth testの精度向上
- バッファの更新
  - 低負荷でヒューリスティックな処理
  - ポリゴンとバッファの内容をマージ
    - Z0/Z1マスクとcoverage mask
    - Z maxとポリゴンのdepth

HiZとの比較

- バッファサイズを小さくできる
  - メモリのオーバーヘッドを小さくできる
- Depthのmipが必要ない
  - フル解像度depthから生成するのは重い
  - ラスタライズして更新するので更に重い



HiZと比べて高速に処理できる

簡単な組み込み

- Github上にソースコード有り
  Intelが管理、ソースファイルは6つ
- Apache 2.0ライセンス
  ライセンス表記すれば、製品に組み込むことが可能

本手法の特徴

- ソフトウェアラスタライザ
- フル解像度 coverage mask
- Hierarchical depth buffer フォーマット
- SIMD命令による最適化
- 簡単な組み込み

アジェンダ

1. オクルージョンカリングとは？
2. 既存手法の紹介
3. Masked Software Occlusion Cullingの紹介
4. Relinkでの導入と最適化
5. オクルージョンカリングの効果と負荷
6. まとめ

GRANDBLUE FANTASY: Relinkでの導入と最適化

最適化の方針

CPUのカリング負荷を極力減らす

PS4のCPU負荷対策

カリング精度が
犠牲になっても構わない

- 3つの章に分けて説明
  - occluder
  - occludee
  - その他

occluder

occluder

◆アーティストが配置

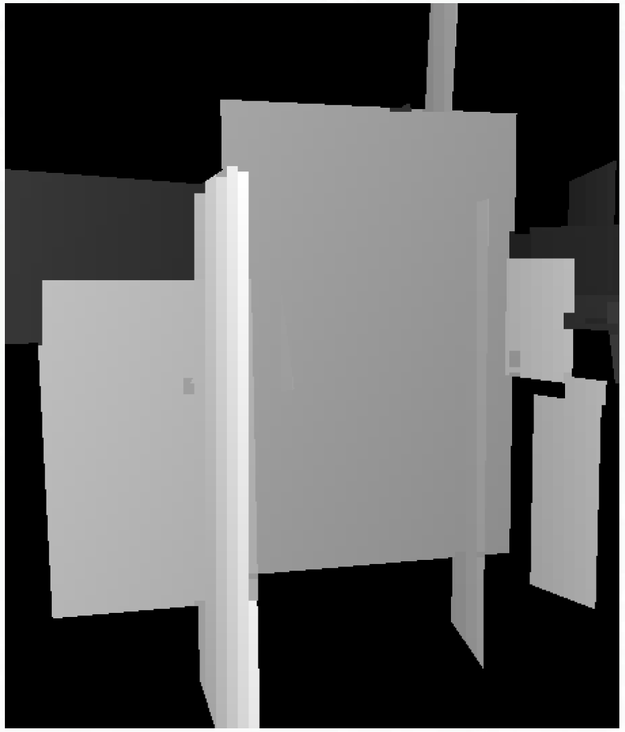
- 長方形の板ポリ
- 極力少ない枚数で
- 各ステージ20~300枚

◆ラスタライズして depth bufferを更新

- occludeeのdepth test で利用

板ポリの配置

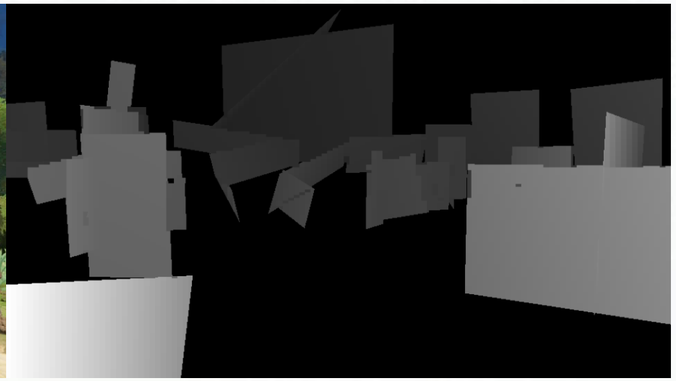
depth buffer



occluderの配置

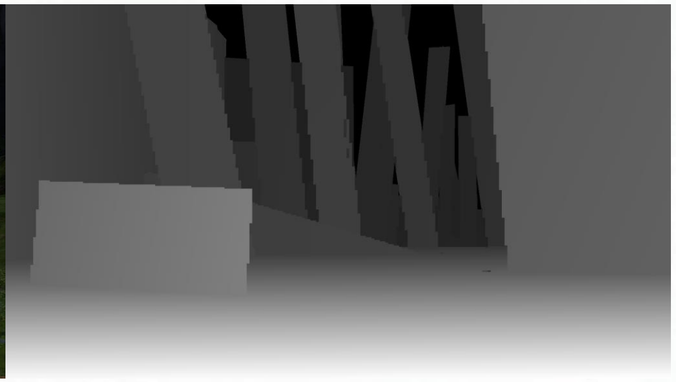
◆開けた場所

- 建物や坂に垂直に配置
  - occluderを減らすため



◆入り組んだ一本道

- 地面と壁にべったり配置
  - 遠方のメッシュもカリング
  - 自由なカメラの向き



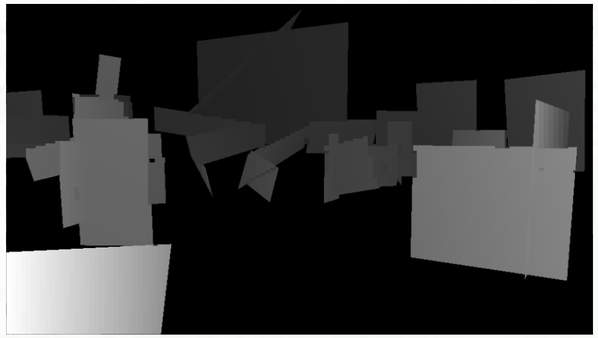
開けた場所のoccluder

入り組んだ一本道のoccluder

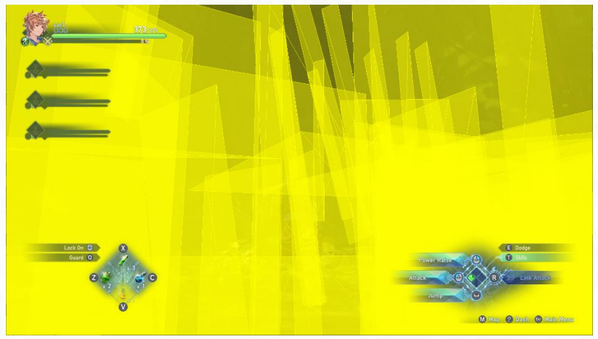
occluderの負荷が大きくなる配置

- スクリーンを覆い隠す板ポリ
- 複数枚重なっているとさらに重い





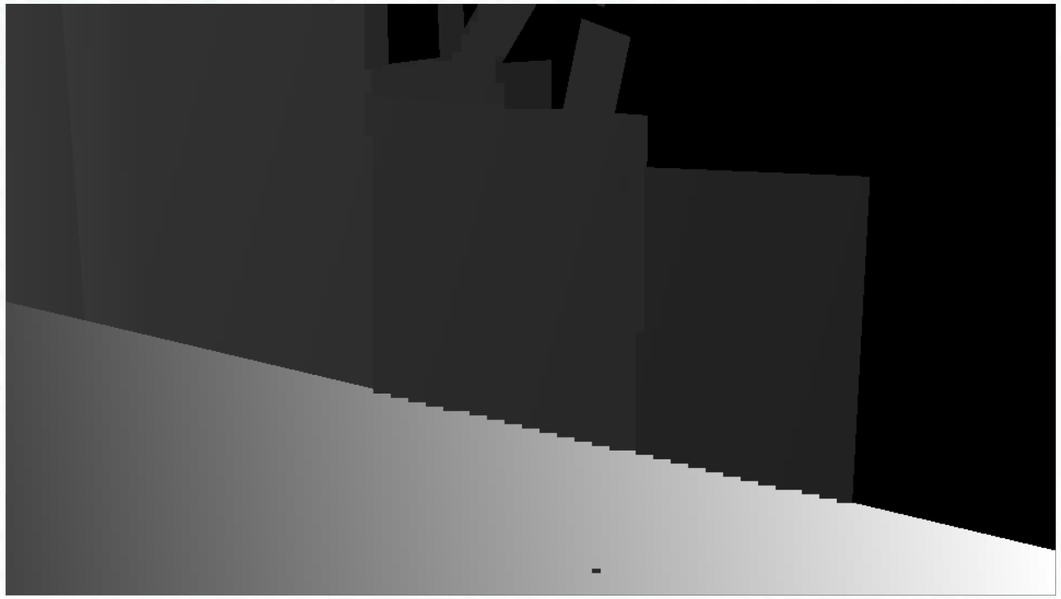
0.4ms


2.0ms

裏に板ポリがたくさん

バッファ解像度を半分にして高速化



	フル	半分
occluder	3.03ms	1.46ms
カリング判定	0.98ms	0.72ms
カリングされたメッシュ数	2953	2926

※PS4の場合　全メッシュ数は16834

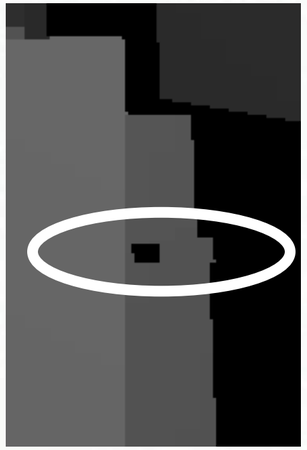
occluderの
ラスタライズで
効果が大きい

どうしてもoccluderが重い場合

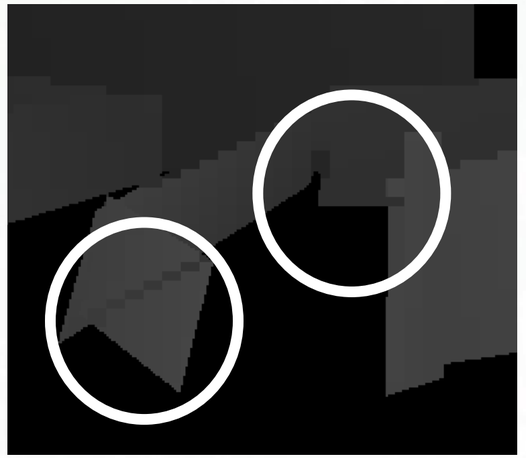
- occluder自体のカリング
  - 距離、フラスタム、空間分割
  - 不要なoccluderの削除
- occluderのソート
  - カメラ距離によるソート
  - Depth bufferの二度書きを防ぐため

Depth bufferのアーティファクト

- depth値がおかしくなることがある
  - ポリゴンの端や、重なり



- これはアルゴリズムの仕様
  - 不正にカリングされることはない
  - あまり酷いとカリング結果に影響あり

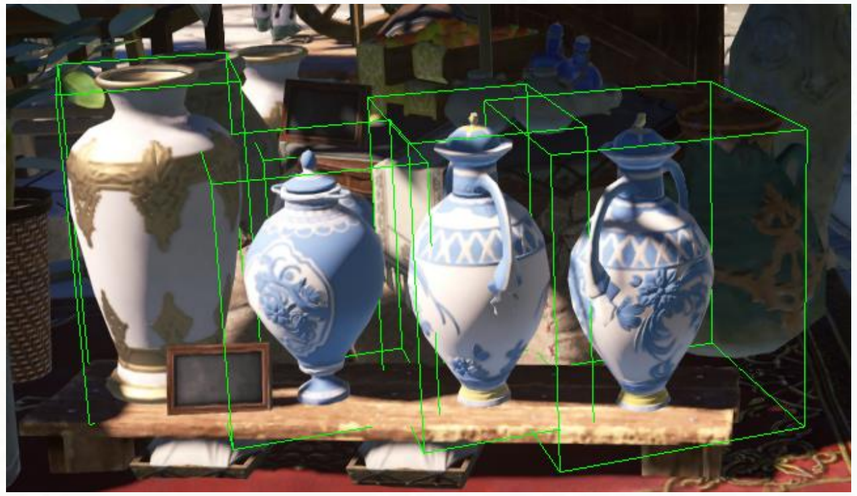


Occluderまとめ

- 板ポリをアーティストが配置
- バッファ解像度を半分に

occludee

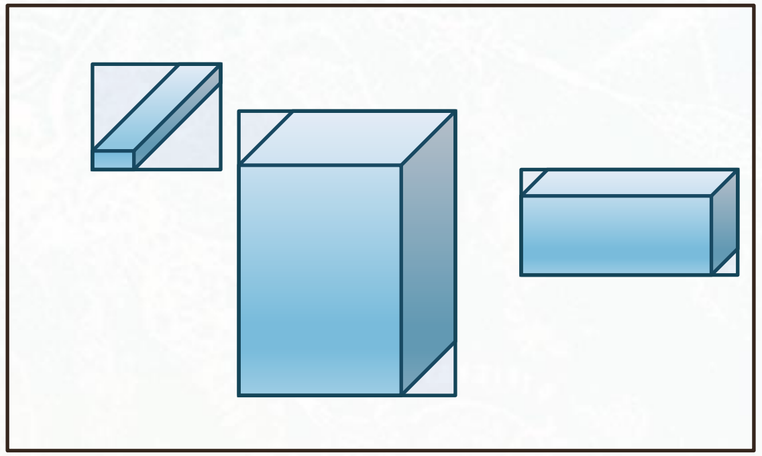
occludee



- メッシュのAABBは?
  - PlayStation 4では重い
- ポリゴン数は抑えたい
  - ポリゴン数に依存して処理時間も長くなる
  - 極力低ポリで

Screen space bounding rectangle

- AABBをNDC座標系に投影して min/maxするだけ
- 最適化された高速な実装が既にある
- 精度は落ちるが高速化



	AABB	提案手法
PS4 FHD	21.8ms	7.07ms
PS5 4k	4.68ms	1.61ms

1コア1スレッドの場合

Screen space bounding rectangleの座標計算

MVP行列を使って計算すると重い

- 4x4行列と4列ベクトルの積
- AABBは8頂点なので8回

命令数を減らす

- 行列計算やmin/maxを分解
- 同じ計算をまとめてSIMD命令化
- サンプルコードに実装あり

	MVP行列	SIMD化
PS4 FHD	7.07ms	4.52ms
PS5 4k	1.61ms	1.36ms

1コア1スレッドの場合

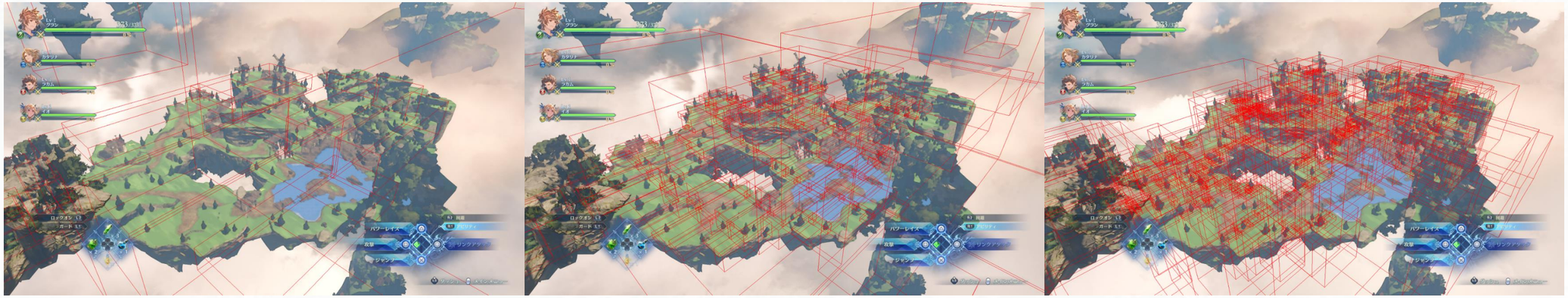
Occludeeを空間分割木で管理

◆各メッシュのAABBから構築

- ツリーノードのAABBは、包含するメッシュのAABBから計算
- 簡易的な八分木。XYZ軸に沿って、立方体になるように分割するだけ

◆ツリーノードのAABBを用いてカリング

- カリングされたら、配下にあるメッシュは全てスキップ
- オクルージョンカリングはscreen space bounding rectangle



空間分割木の工夫

◆距離カリング用の空間分割木

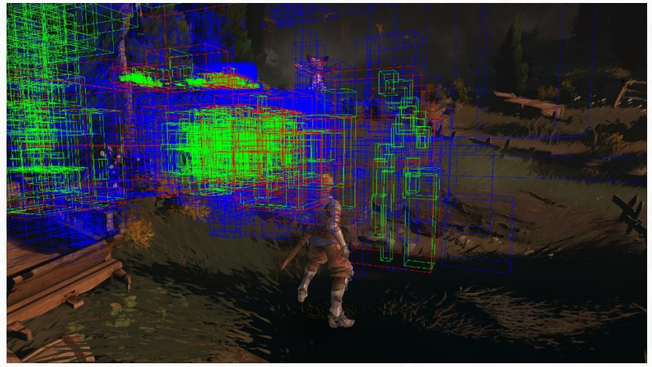
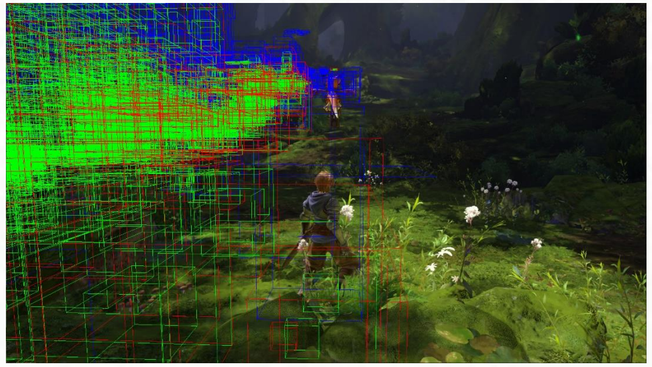
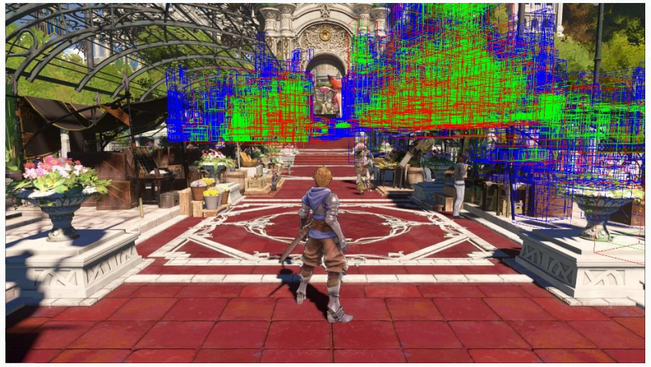
- 距離カリングは小さな草などで利用
- 綺麗に空間分割するために、大きいものは除外

◆動的メッシュは除外

- キャラクタやスキニングメッシュなど
- 移動、変形時の空間分割木の再構築の負荷

空間分割木の効果

緑のボックスはツリーノードごとカリングされたメッシュ
青のボックスは個々にカリングされたメッシュ

	なし→あり		
PS4 FullHD	4.34ms→3.87ms	5.78ms→3.31ms	6.00ms→4.52ms
PS5 4k	1.41ms→1.32ms	1.64ms→1.03ms	1.71ms→1.36ms

1コア1スレッドの場合

occluderがスクリーンを覆い隠していると、効果が大きい

occludeeまとめ

- Screen space bounding rectangle
  - 低ポリゴン、専用実装
  - 座標計算を最適化
- 空間分割木

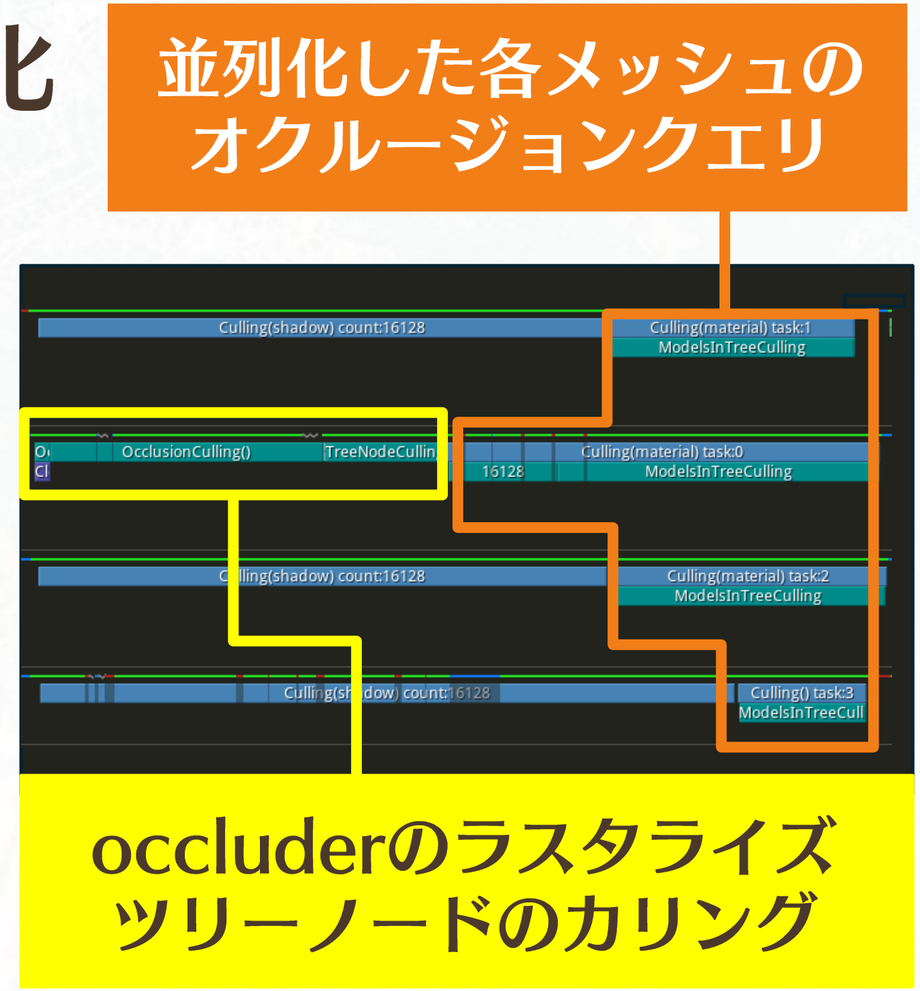
その他

SIMD命令による最適化

- 4つの実装が存在
  - 利用するCPUの世代によって、実装を切り替える
    - SSE 2(128bit長), SSE4.1(128bit長)
    - AVX2(256bit長), AVX512(512bit長)
- PlayStation4はSSE4.2, AVXに対応
  - SSE 4.1(128bit長)を選択
- PlayStation5はAVX2に対応
  - AVX2(256bit長)を選択

並列処理の活用

- オクルージョンクエリは並列化
  - リーフノードがもつメッシュによる膨大なクエリ数
- 並列化していない処理
  - occluderのラスタライズ
  - ツリーノードのカリング
- 他の処理と並列に
  - シャドウのカリング
  - 並列化してない処理は、うまく隠せた



導入と最適化のまとめ

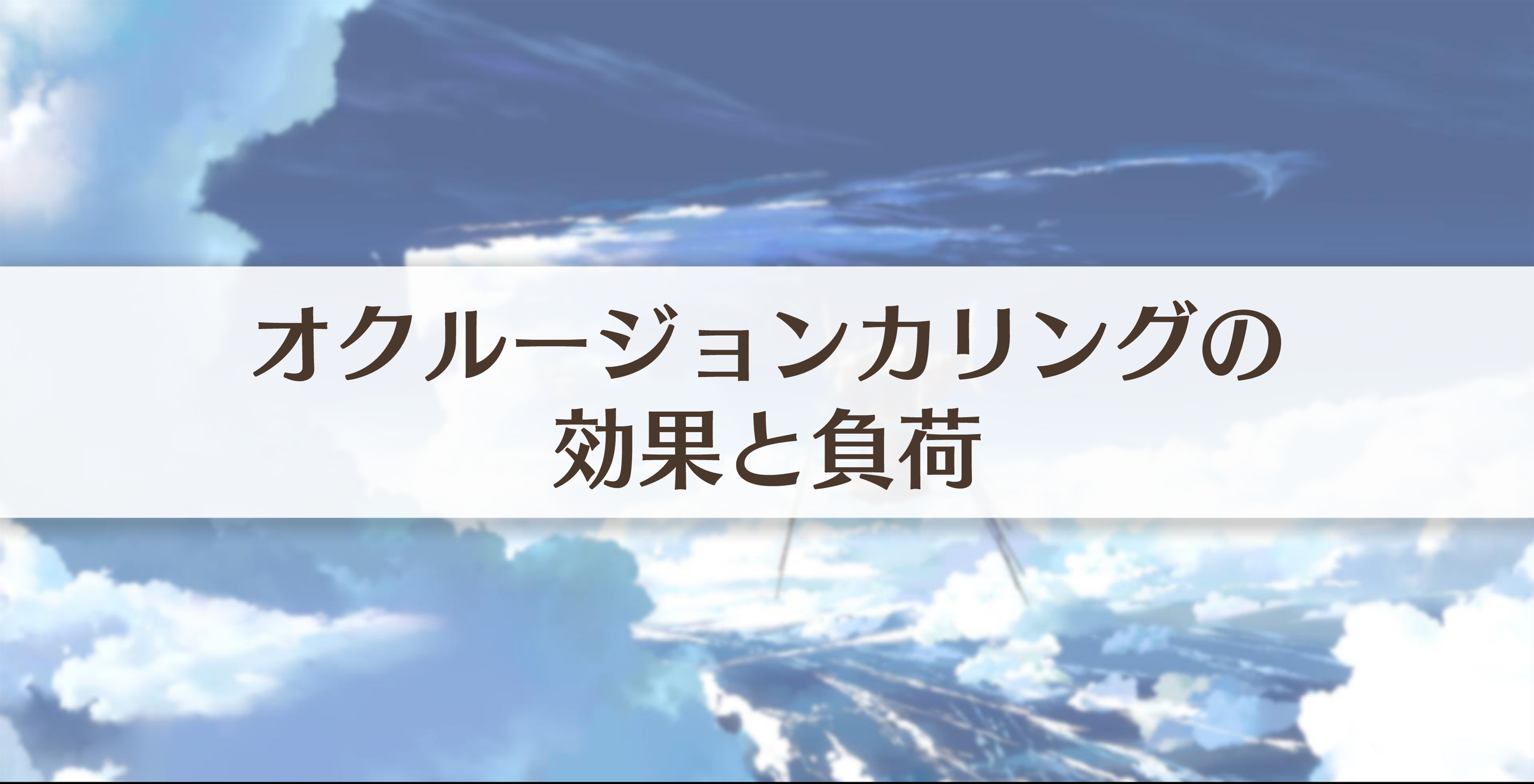
- occluder
  - アーティストが配置する板ポリ, バッファ解像度半分
- occludee
  - Screen space bounding rectangle, 空間分割木で管理
- プラットフォーム毎に適切な実装を選択
  - PlayStation 4: SSE4.1実装, PlayStation 5: AVX2実装
- オクルージョンクエリの並列化

CPUのカリング負荷を極力減らす

ピクセル単位で見た場合のカリング精度は落ちた

アジェンダ

1. オクルージョンカリングとは？
2. 既存手法の紹介
3. Masked Software Occlusion Cullingの紹介
4. Relinkでの導入と最適化
5. オクルージョンカリングの効果と負荷
6. まとめ

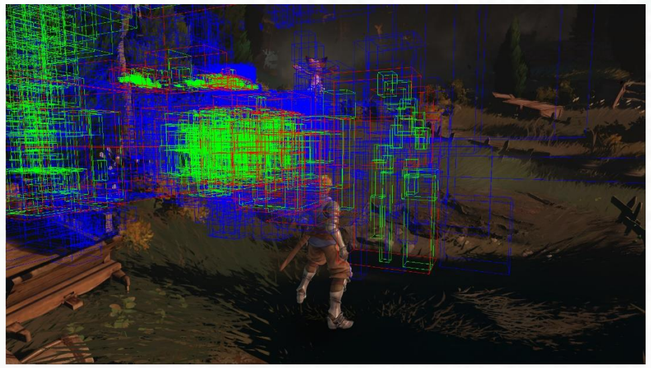
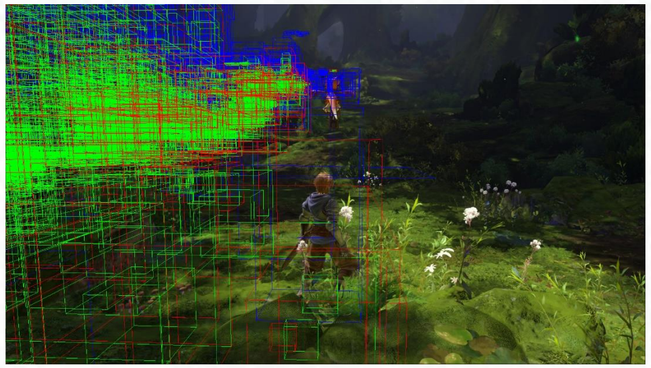
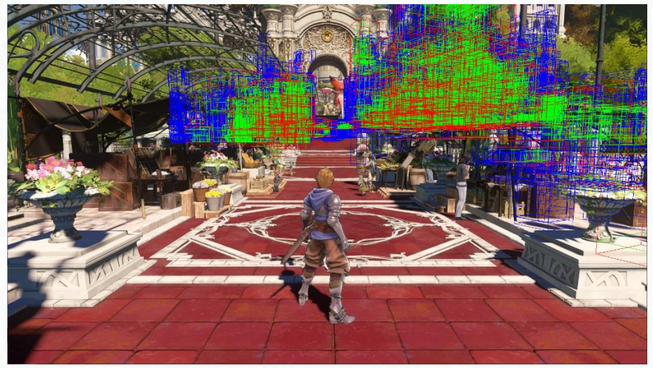
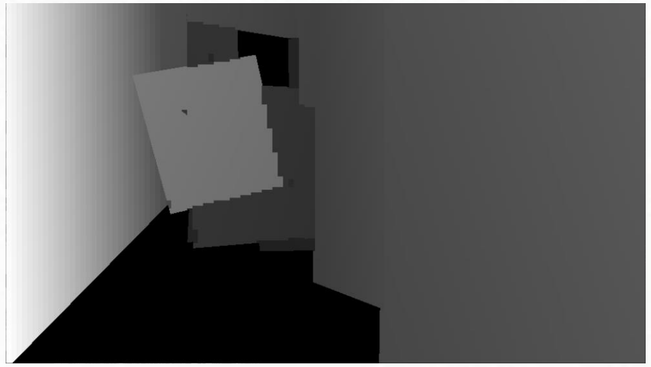
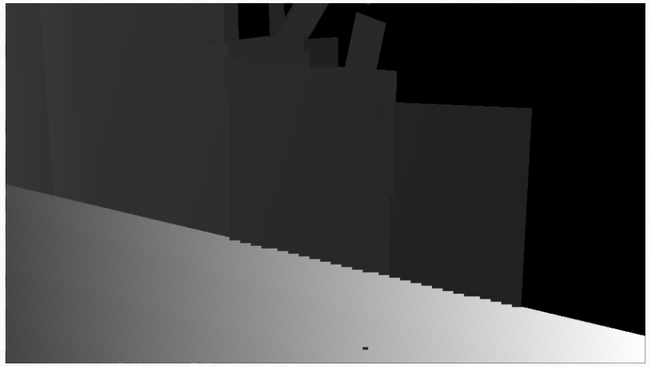
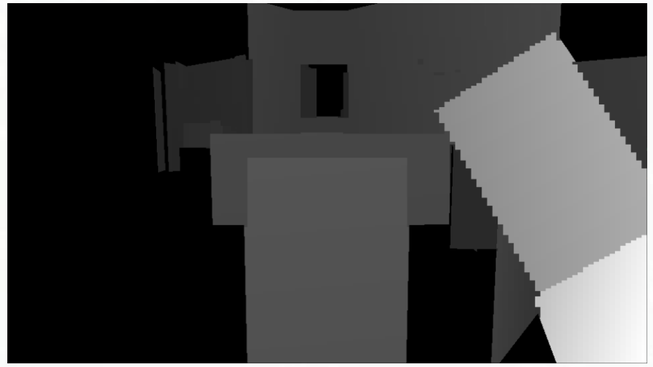


オクルージョンカリングの効果と負荷

測定環境と並列処理

- プラットフォームの設定
  - PlayStation4: FullHD, PlayStation5: 4k
  - 30fps目標
- 並列化
  - PlayStation4: 1クラスタ4コア, PlayStation5: 1クラスタ6コア
- シャドウのカリングと混ぜて並列化
  - 物理コア活用のため
- フラスタムカリングと一緒に測定

測定したステージ

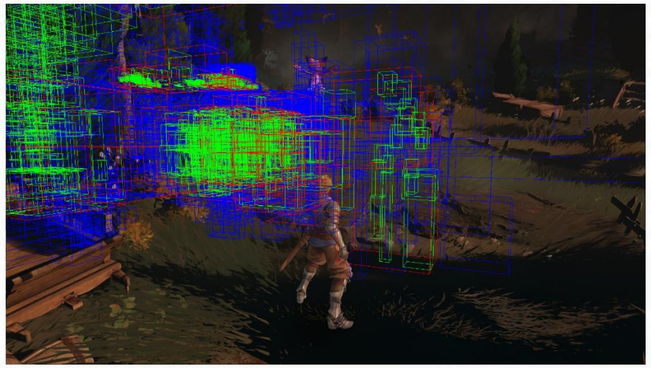
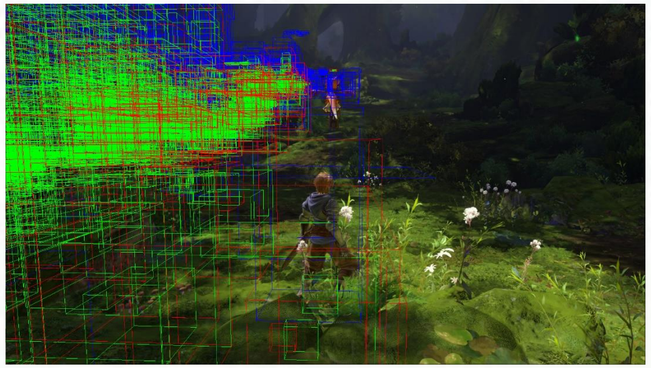
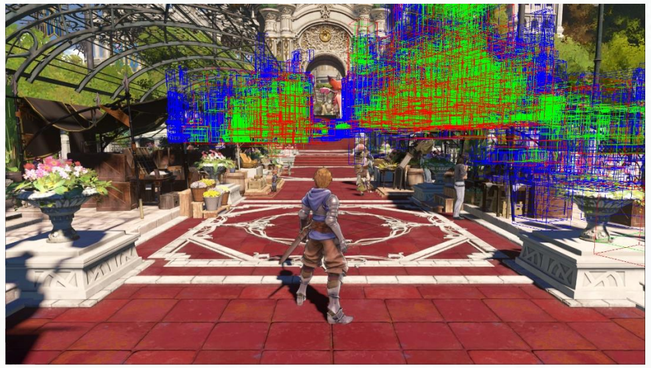







メッシュ数	12558	16890	16075
occluder数	26	141	46

負荷の大半は背景メッシュのカリング処理

実際の処理時間

※ 青と緑のボックスはカリングされたメッシュのAABB

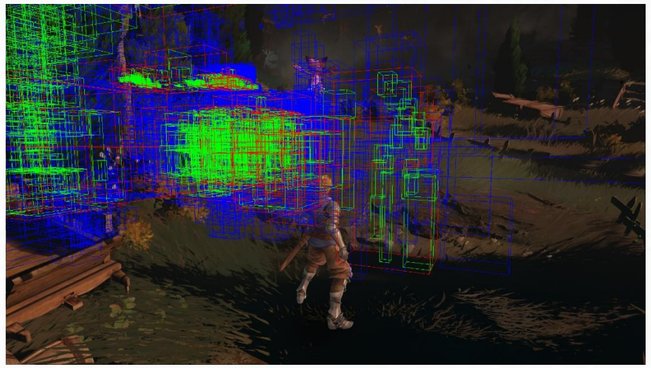
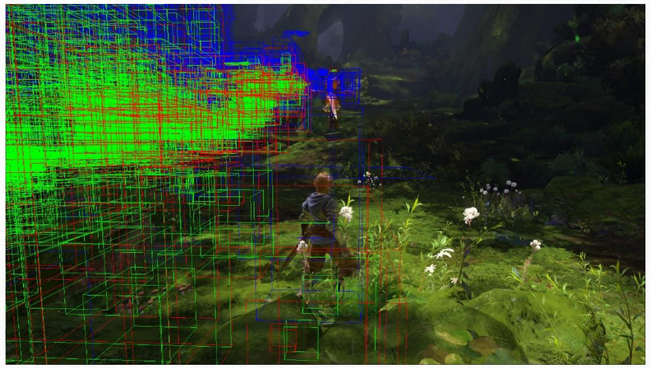
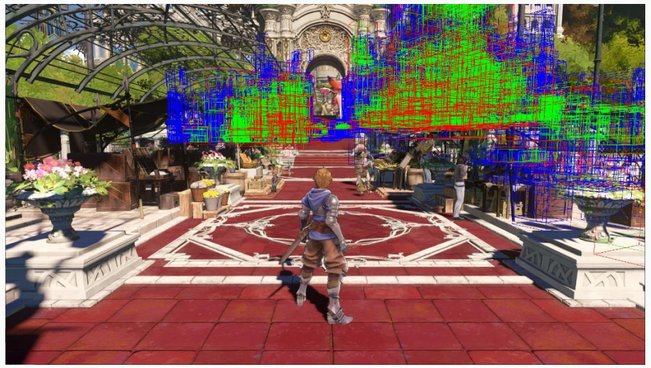
PS4 FullHD	2.88ms	2.95ms	3.53ms
PS5 4k	1.13ms	0.86ms	1.07ms

- PlayStation 4: 2.88~3.53ms
- PlayStation 5: 1.0ms前後

1コア1スレッドの場合

◆純粋なカリング処理のみ

※ 青と緑のボックスはカリングされたメッシュのAABB

PS4 FullHD	4.34ms	3.32ms	5.06ms
PS5 4k	1.71ms	1.18ms	1.62ms

効果と負荷について細かく紹介

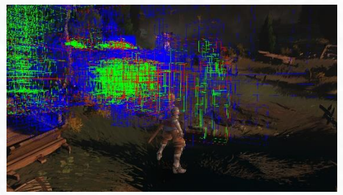
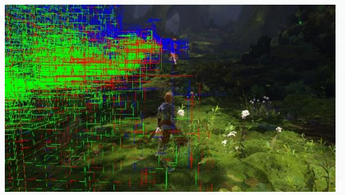
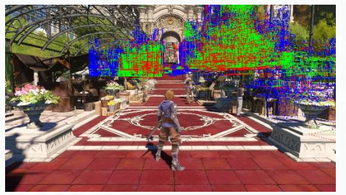
- まずはPlayStation 4に注力して紹介
  PS4優先でカリング処理を最適化したため
- PlayStation 5についても軽く紹介
  最適化ではあまり意識せず

カリング負荷の傾向

- 処理時間はクエリ数に強く依存
- 個々のメッシュのカリング処理が半分以上を占める
- occluderのラスタライズ負荷が大きいことも

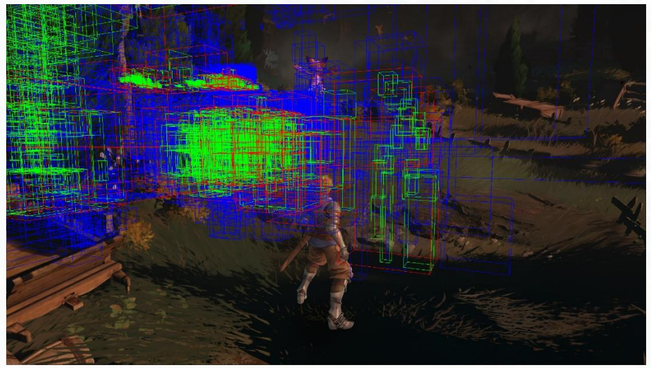
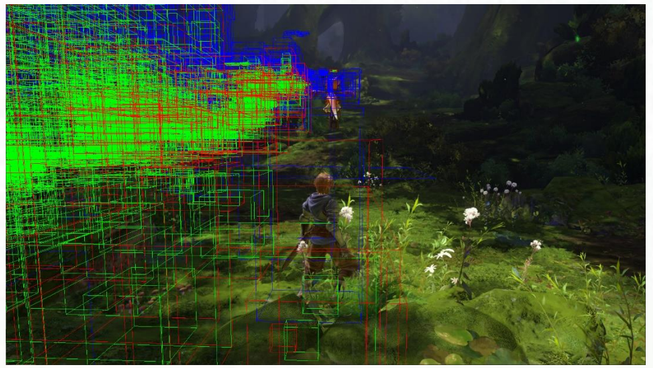
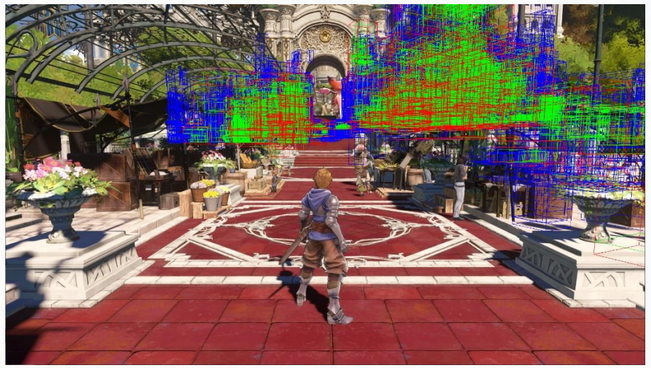
PS4での処理時間の構成

1コア1スレッドの場合

occluderのラスタライズ			0.45ms	1.19ms	1.03ms
ツリーノードのカリング	クエリ数	フラスタム	531	422	672
		オクルージョン	469	317	608
	処理時間		0.42ms	0.33ms	0.44ms
個々のメッシュのカリング	クエリ数	フラスタム	7931	3969	8341
		オクルージョン	6989	3192	7846
	処理時間		3.39ms	1.81ms	3.49ms
カリング全体の処理時間			4.27ms	3.34ms	4.97ms

PS4のオクルージョンカリングとGPUの負荷

※ 青と緑のボックスはカリングされたメッシュのAABB

	オクルージョンカリング なし→あり		
カリング	2.0ms→2.81ms	1.92ms→2.89ms	2.77ms→3.43ms
GPU	22.4ms→21.8ms	22.0ms→21.0ms	25.1ms→24.2ms

オクルージョンカリングの貢献

- PS4 GPUだと1ms前後の効果になりがち
  - エリアごとのカリングも利用
  - 最適化された遠景のアセットと実装

フレームレートの安定化に貢献

- GPUの負荷が突発的に増えるとき
  - 大きな遮蔽物と描画物が多い箇所
- 少しでもGPUの負荷を減らしたいとき
  - バトル中
- 手軽にカリングを利用したいとき
  - カットシーン中

大きな遮蔽物と描画物が多い箇所

- 大きな壁に囲われ、曲がりくねった1本道
- カメラを振ったときに、負荷が増えがち

オクルージョンカリング	PS4 GPU
なし	24.3ms
あり	22.1ms

大きな壁で、
多くのメッシュが遮蔽される

バトル中の負荷

◆少しでも負荷を減らしたい

ボス戦におけるオクルージョンカリング

- 多数のエフェクトや敵が描画
- 動的解像度により
  解像度が落ちてしまう

カットシーンでの効果

- キャラクタ関連で、大きな追加コスト
- 板ポリoccluderを別途追加して、負荷を減らす
  - ステージにある扉、大きな移動物などでも利用

オクルージョンカリング	PS4 GPU
なし	38.3ms
あり	34.9ms

カメラに映らない背景をoccluderで無理矢理カリング

CPU負荷の低減

- カリング結果を次フレームで使用
  - 更新頻度の低下(アニメーション、エフェクトなど)
- 最終的なカリング結果を適用
  - オクルージョンカリングだけではない
  - フラスタムカリングなどを含む

PlayStation 5の処理時間

- カリング処理が1.0ms前後
  - 4k/30fps, FullHD/60fps, 共に問題ない処理時間
  - 開発中はあまりケアせず
- CPUが高速
  - Jaguar → Zen2
  - AVX2対応なので最適化された実装を利用できる

PlayStation 5のパフォーマンス

- FullHD/60fpsの場合
  - 少しでも負荷を減らしておきたい
  - スパイクは避けたい
- 4k/30fpsの場合
  - GPUネック

フレームレートは安定するので導入すべき

導入メリットのまとめ

- フレームレートの安定化に貢献
  - 大幅にGPU負荷が減ることは稀
- occluderを配置すれば簡単に利用できる
  - カットシーン時だけに配置
  - ドアや巨大な移動物に配置
- 負荷低減策は複数あるほうが良い
  - オクルージョンカリング、エリアごとのカリング、遠景の最適化
  - 状況に応じて選択できる

アジェンダ

1. オクルージョンカリングとは？
2. 既存手法の紹介
3. Masked Software Occlusion Cullingの紹介
4. Relinkでの導入と最適化
5. オクルージョンカリングの効果と負荷
6. まとめ

まとめ

まとめ

- Masked Software Occlusion Cullingを導入
- PlayStation 4/5で現実的な処理時間に収まった
- 組み込みしやすい

開発の現状と工数を考慮して、
適切な技術や仕様を採用することが大切

参考文献

1. [HAAM16], “Masked software occlusion culling”, In High Performance Graphics, pages 23-31, 2016
2. [Ste11], Stephen H.: Practical, Dynamic Visibility for Games, self shadow (blog), 2011
3. [Mat21], Matthew P.: Samurai Landscapes: Building and Rendering Tsushima Island on PS4, Game Developer’s Conference (presentation), 2021
4. 三嶋　仁, 最新タイトルのグラフィックス最適化事例, CEDEC 2018
5. Masked Software Occlusion Culling
https://www.intel.com/content/www/us/en/developer/articles/technical/masked-software-occlusion-culling.html