Cascaded Voxel Cone Tracing in The Tomorrow Children

トゥモロー　チルドレンで採用した
カスケード・ボクセル・コーントレースライティング技術について

James McLaren

Q-Games Ltd.

Who am I?

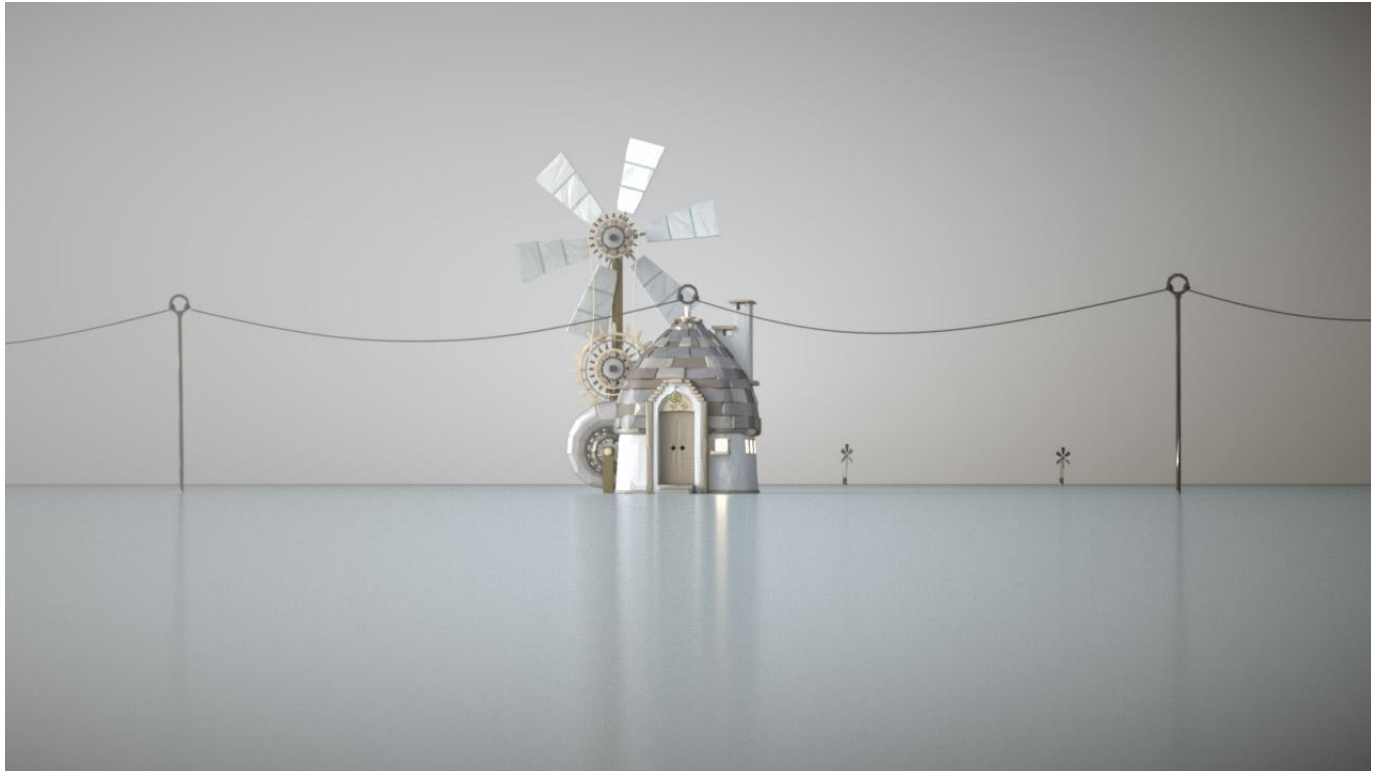
- Director of Engine Technology @ Q-Games.
- 19 years in the industry.
- PS3 OS Graphics

・キュー・ゲームスのエンジン・テクノロジディレクター
・19年のゲーム開発経験
・PlayStation3　OSグラフィクスや、ヴィジュアライザを開発

Game Video

Concept Art

Concept Art



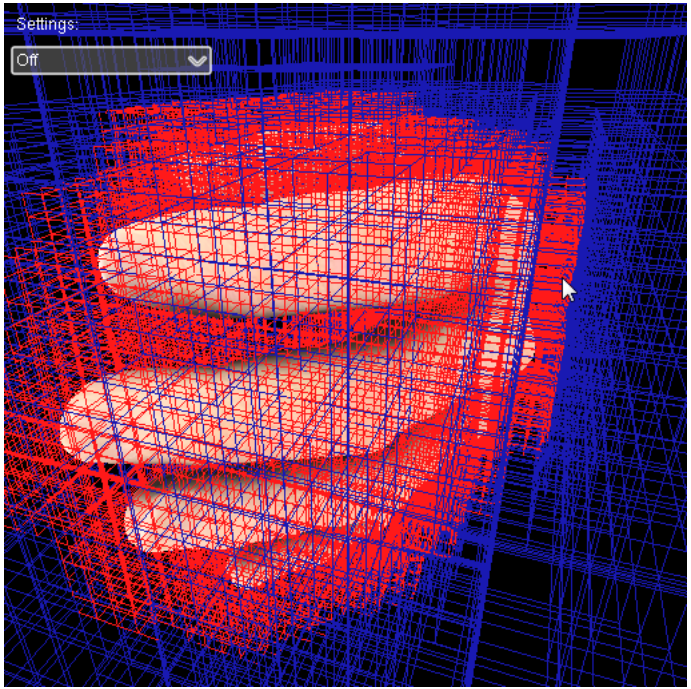
Inspiration – Voxel Cone Tracing Talk@Siggraph 2011

Inspiration – Unreal Elemental Demo

Voxel Cone Tracingとは？

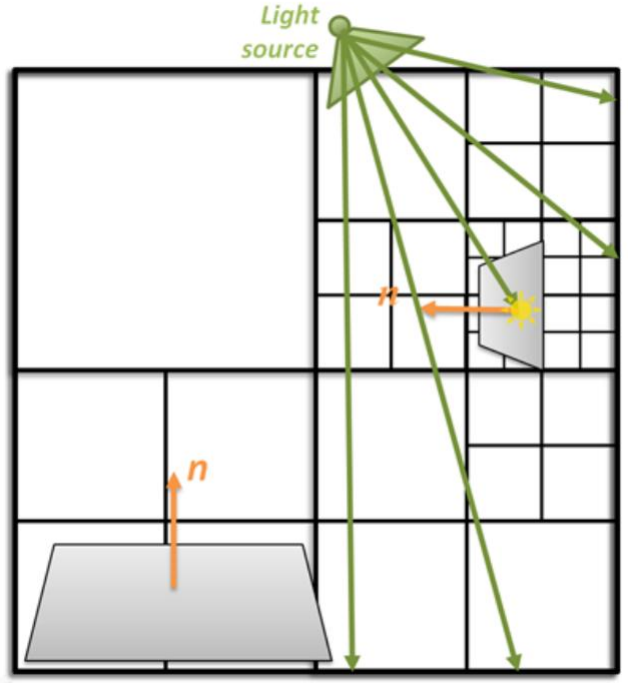
- Voxelize the geometry to build Sparse Voxel Octree.

- Sparse Voxel Octreeを構築するために、ジオメトリをボクセライズする



Voxel Cone Tracingの概要

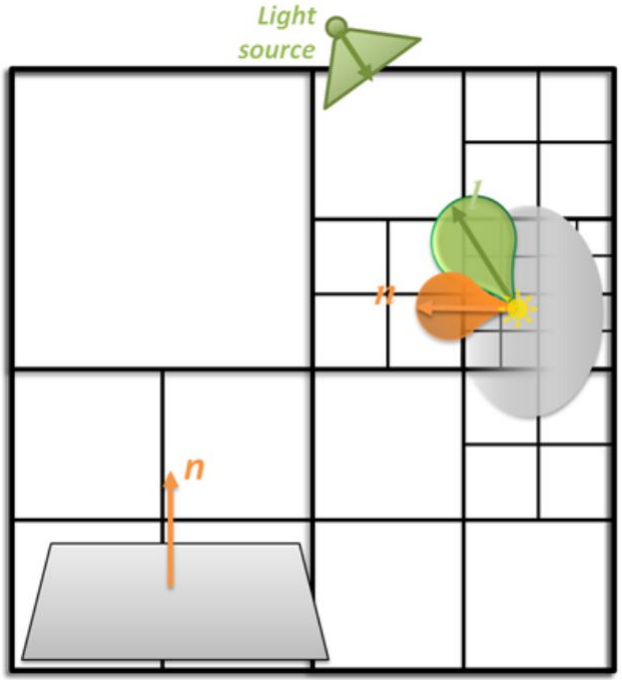
- Voxelize the geometry to build Sparse Voxel Octree.
- Inject Irradiance.



- SVOを構築するために、ジオメトリをボクセライズする
- 直接光の反射情報を加える

Voxel Cone Tracingの概要

- Voxelize the geometry to build Sparse Voxel Octree.
- Inject Irradiance.
- Filter the Irradiance up the SVO.

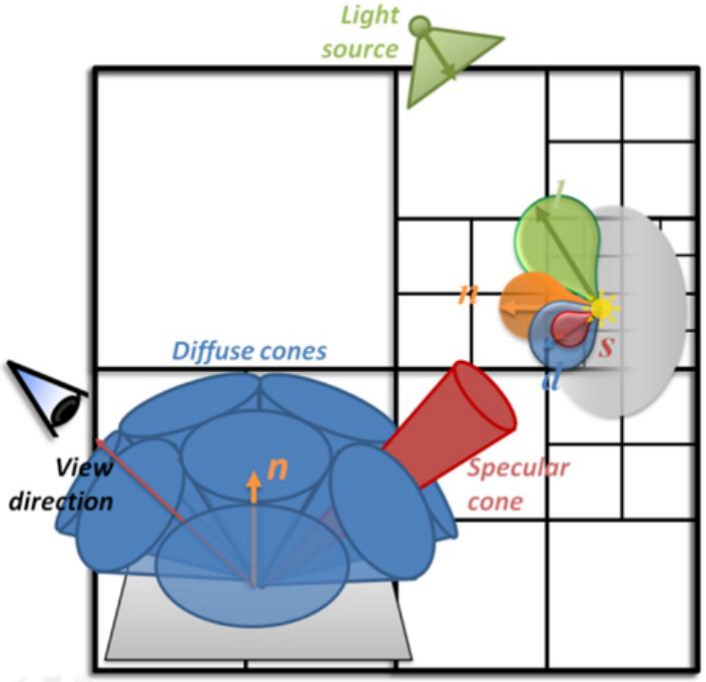


- SVOを構築するために、ジオメトリをボクセライズする
- 直接光の反射情報を加える
- SVO光量をフィルタリングする

Voxel Cone Tracingの概要

- Voxelize the geometry to build Sparse Voxel Octree.
- Inject Irradiance.
- Filter the Irradiance up the SVO.
- Trace cones per pixel through the SVO.

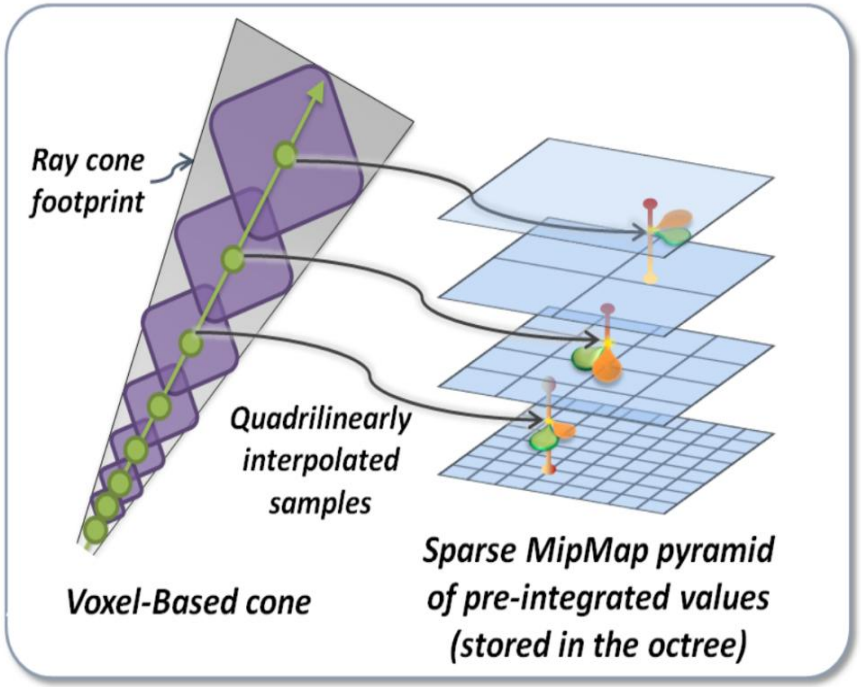
- SVOを構築するために、ジオメトリをボクセライズする
- 直接光の反射情報を加える
- SVO光量をフィルタリングする
- SVOを通してピクセルごとにコーントレースをする



Voxel Cone Tracingの概要

- Voxelize the geometry to build Sparse Voxel Octree.
- Inject Irradiance.
- Filter the Irradiance up the SVO.
- Trace cones per pixel through the SVO.
- Accumulate Irradiance.

- SVOを構築するために、ジオメトリをボクセライズする
- 放射照度を加える
- SVO光量をフィルタリングする
- SVOを通してピクセルごとにコーントレースをする
- 直接光の反射情報を蓄積する



Voxel Cone Tracingの実装

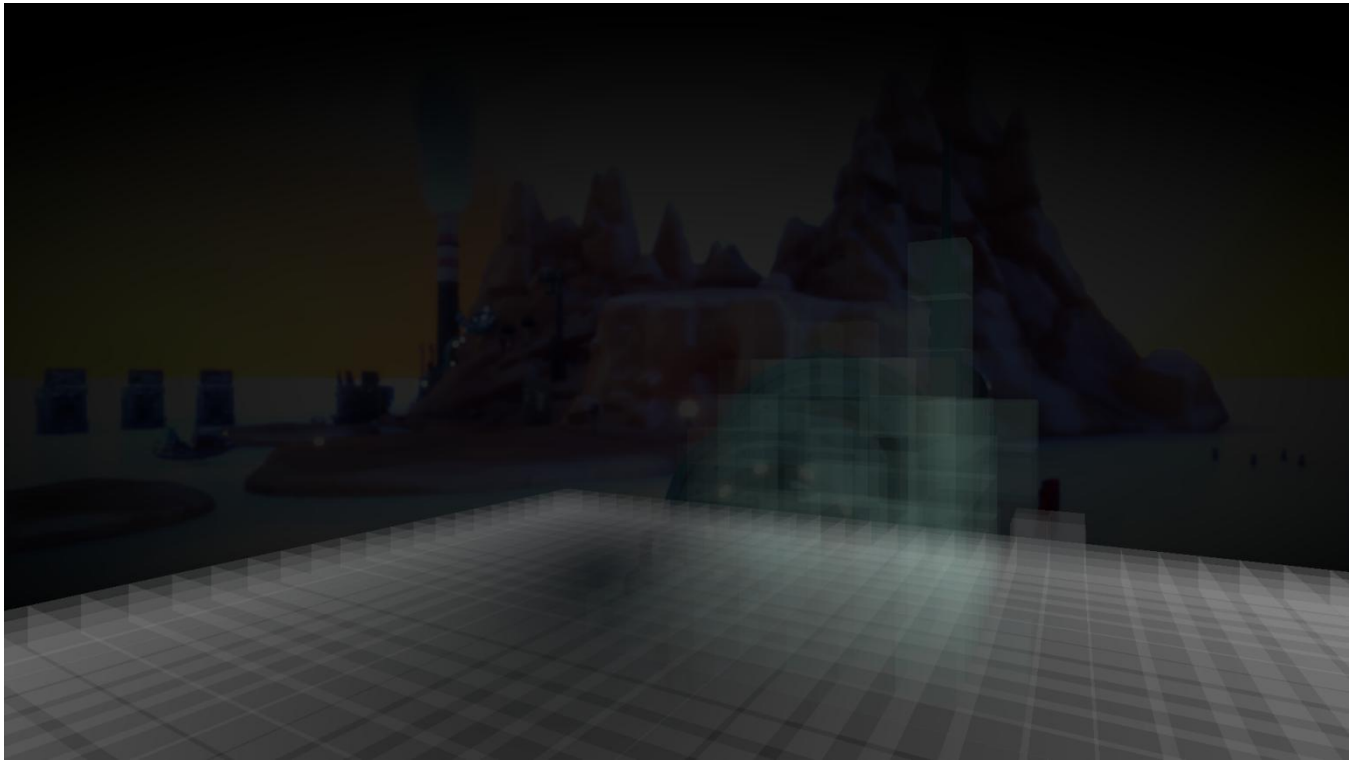
- First experiment on DX11 & early PS4 SDK
- Worked but quite slow (30+ms)
- Finally returned to utterly rotten code 1 year later.
- Decided to K.I.S.S. for 2nd attempt
- Just use a 3D texture!
- DX11 & 初期のPS4 SDK での実装テスト
- 動いたが非常に遅い(30+ms)
- 一年後には使い物にならないコードになった
- なので2回めはシンプルな方法を使うことにした
- 3D textureを使おう！

What we ended up with

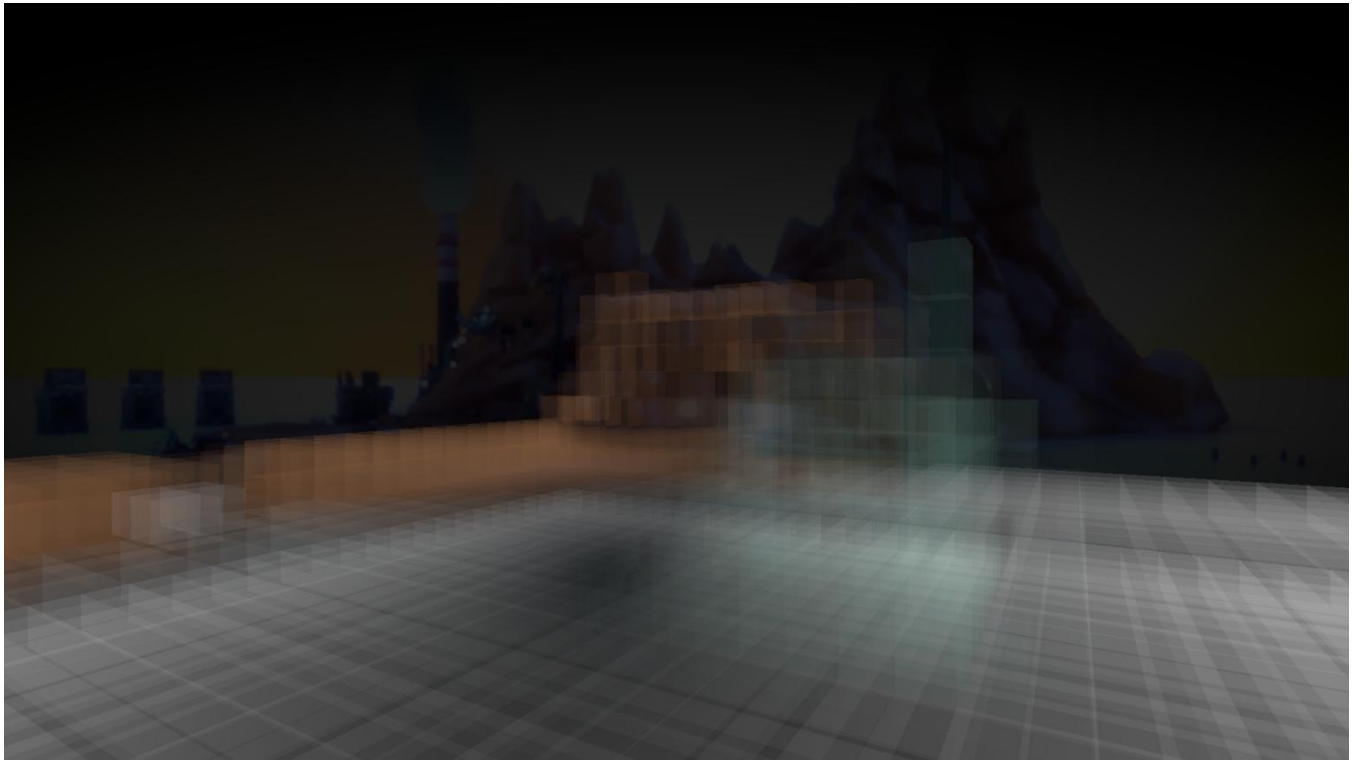
- 6 Cascades of 32^3 voxels.
- Anisotropic Voxels (6 facing directions)
- Packed into a single 3D texture for each attribute ((6*32)x(6*32)x32).
- Trace in 16 directions (spherical t-design)
- 2-3 Bounce Diffuse Indirect Lighting
- Also handles Direct Lighting.
- We don’t use any shadow maps.
- ６のカスケードと32^3のボクセル
- 異方性ボクセル（6面方向）
- 各属性を（（6*32）×（6* 32）x32）の単一の3Dテクスチャに詰め込む
- 16方向のトレース（球状T形状）
- 2-3回のバウンス拡散間接照明
- 直接光の処理
- シャドウマップは使用しない

Voxel Cascades

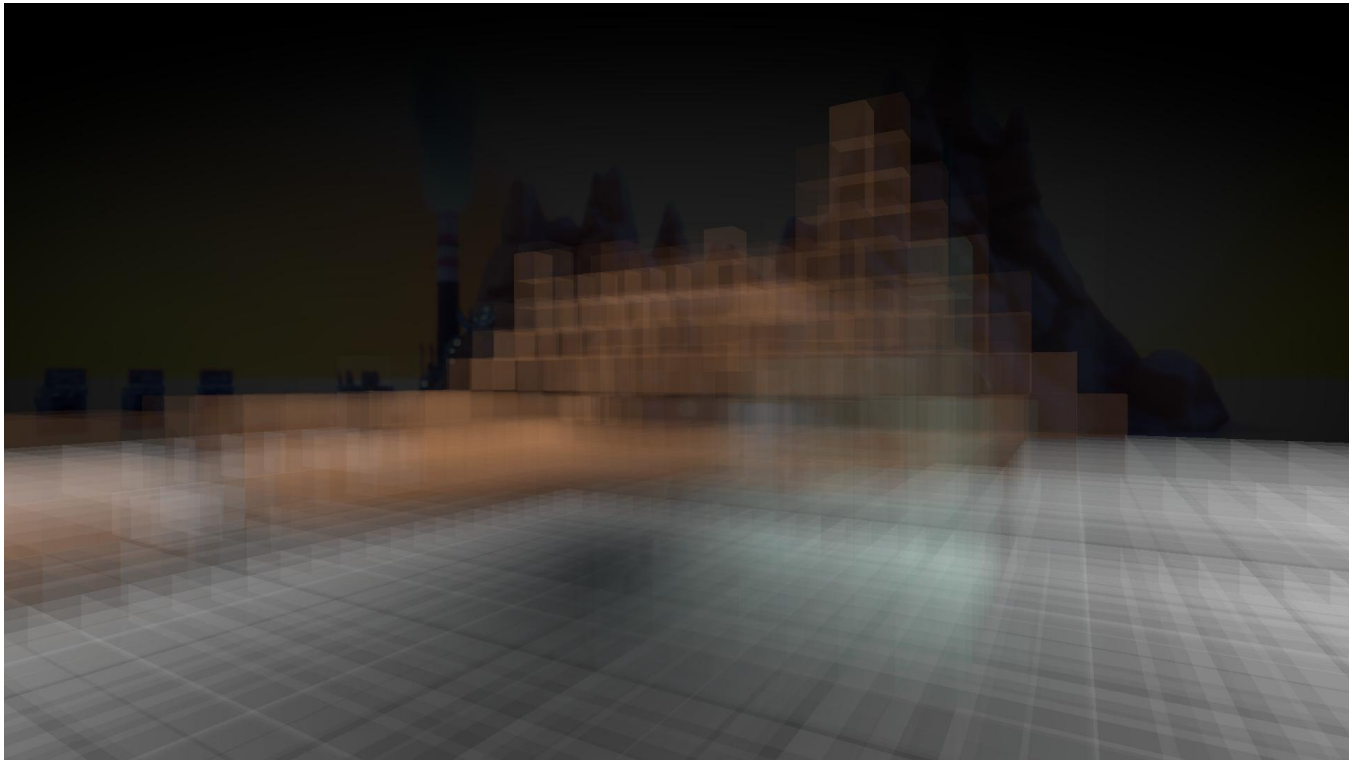
Voxel Cascades



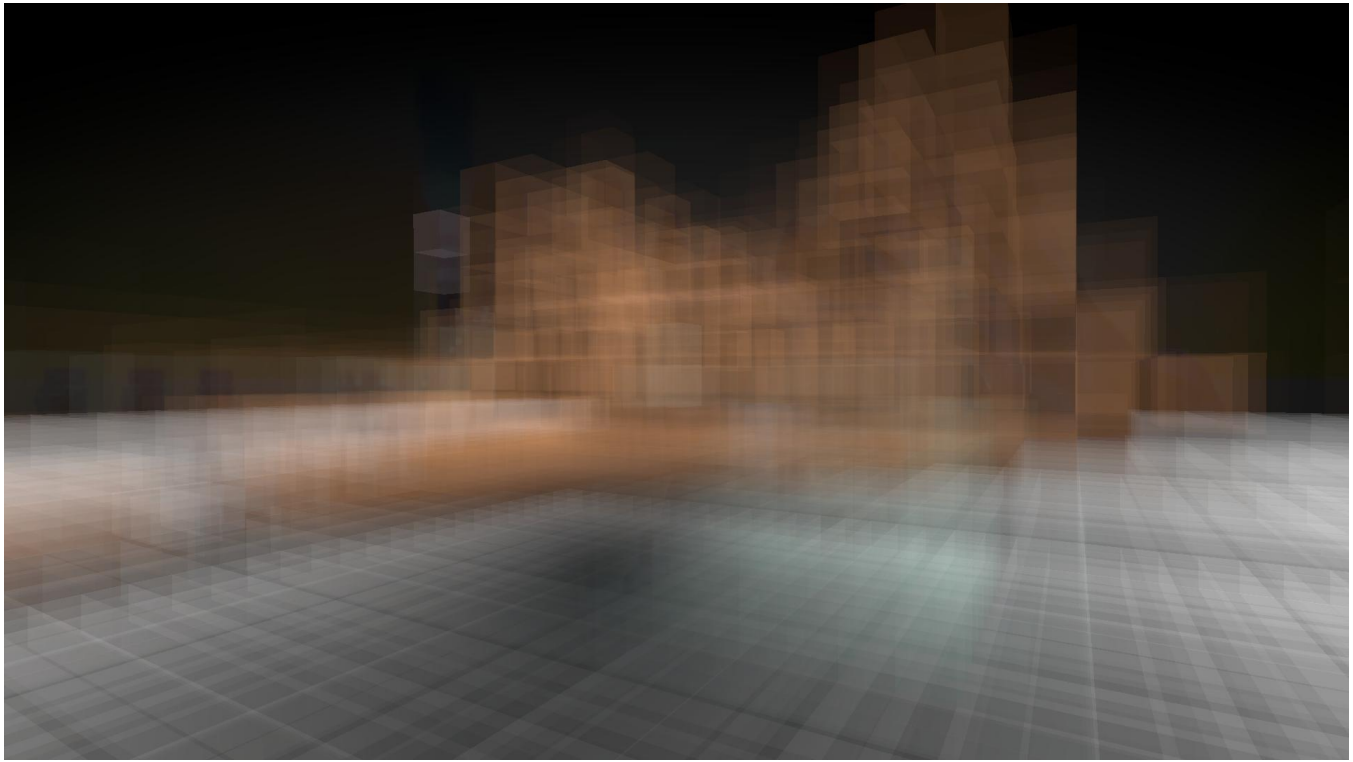
Voxel Cascades



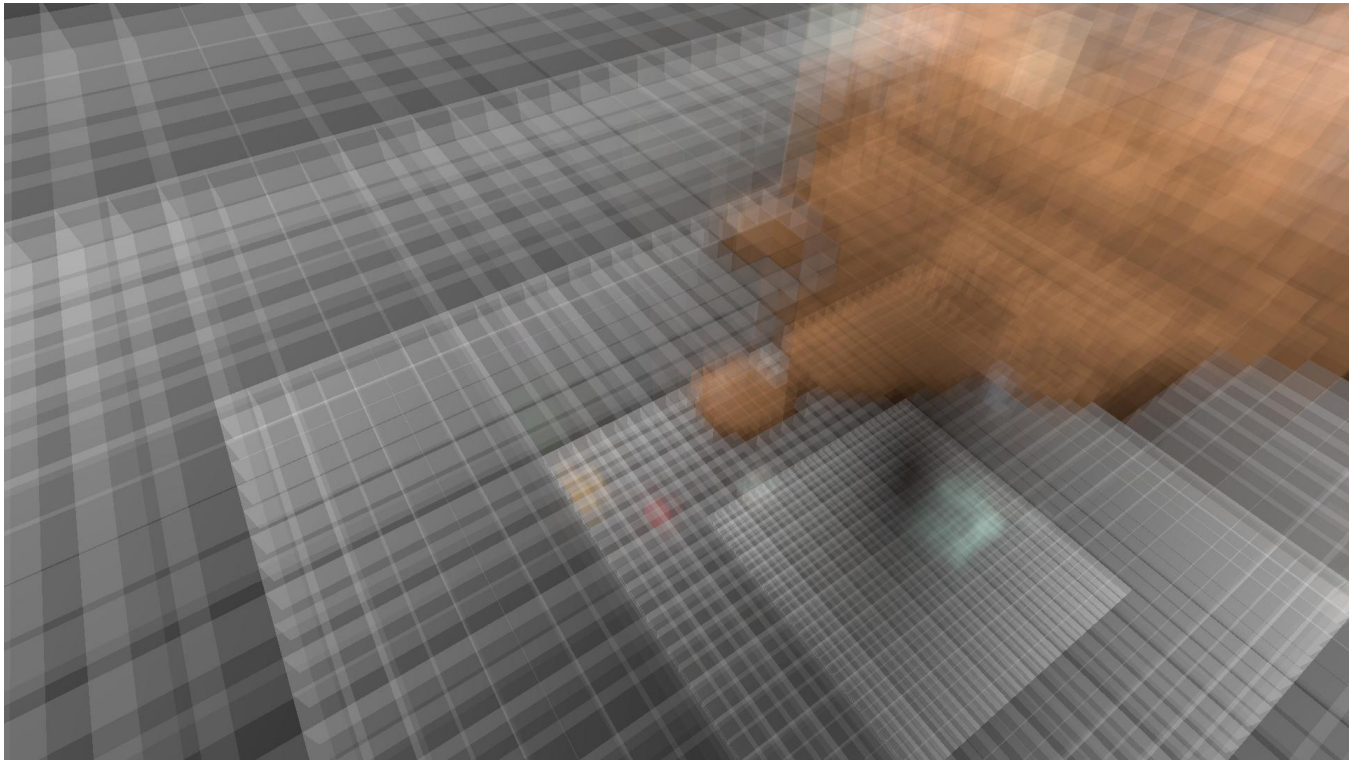
Voxel Cascades



Voxel Cascades



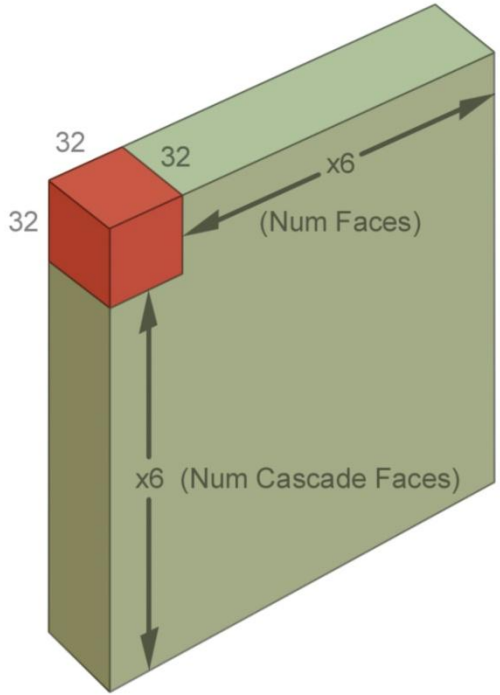
Voxel Cascades



3D Textureへのデータ格納

- Cascades and faces all packed into a single texture per attribute.
- Attribute textures for albedo, normal, occlusion and emission.

- カスケードと面はすべてアトリビュートごとにシングルテクスチャーになっている
- アトリビュートテクスチャーはアルベド、法線、オクルージョン、エミッションに使われている



Algorithm Overview

- Algorithm split into two phases:

1. Internal update of cascades
2. Screen space cone trace.

- アルゴリズムは2つ:

1. カスケードの内部更新
2. スクリーンスペースでのコーントレース

Cascade Update

- We only voxelize/update one level of the cascade per frame.
- Use simple incrementing counter and find the lowest set bit.
- `CountTrailingZeroes(((count++))&~((1<<(kVoxelCubeGILevels-1))-1));`
- Each cascade updated twice as frequently as the next.

- 一レベルのボクセライズ/アップデートを毎フレーム行う
- 単純な増分カウンタを使用して、最下位セットビットを見つける
- 各カスケードは、頻繁に2回更新する

Cascade Update

- Calculate new cascade center.
- Scroll cascade data if we have moved.
- Voxelize to update any geometry that has changed, if necessary.
- “Surface” voxels are identified during voxelization.
- We then propagate illumination through the cascade via cone tracing (in 16 directions), starting at these voxels.

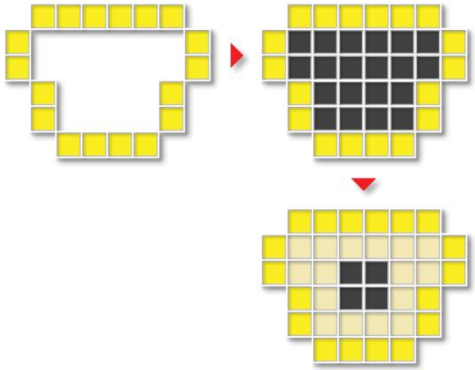
- 新しいカスケードセンターを計算する
- 移動していた場合、カスケードデータをスクロールする
- 必要に応じて、変更されている任意のジオメトリをボクセライズし更新する
- 「サーフェイス」のボクセルは、ボクセライズ中に識別する
- これらのボクセルからスタートし、コーントレース（16方向）を経由してカスケードを通して照明を伝播する

Voxelization

- Only voxelize static geometry.
- Voxelize at each cascade at 128x128x32 for each axis to get 16x super sampling.
- Landscape uses a LDC, and is thus trivial to voxelize.
- Objects use a block based cache.
- 静的ジオメトリのみをボクセライズする
- 128x128x32のカスケードと、それぞれの軸で16xのスーパーサンプリングを得るためにボクセライズ
- 地形はLDCを使っているのでボクセライズは簡単です
- オブジェクトにはブロックベースのキャッシュを使っている

Voxelization

- Important to be a solid voxelization.
- Fill spans between voxelized surfaces with black opaque voxels.
- Propagate surface attributes to first inner layer of voxels.



- 重要なのはソリッドなボクセル化
- 黒い不明瞭なボクセルとボクセル化した表面の間を埋める
- ボクセルの第一階層へサーフェイスアトリビュートを伝播する

Multi-Bounce Cone Trace

- Want to get multiple bounces of light propagated in our cascades.
- Also need to inject direct lighting from lights and from the sky.

- カスケードに伝搬する光の、複数回のバウンスを取得したい
- また、ライトや天球から直接光を注入する必要がある

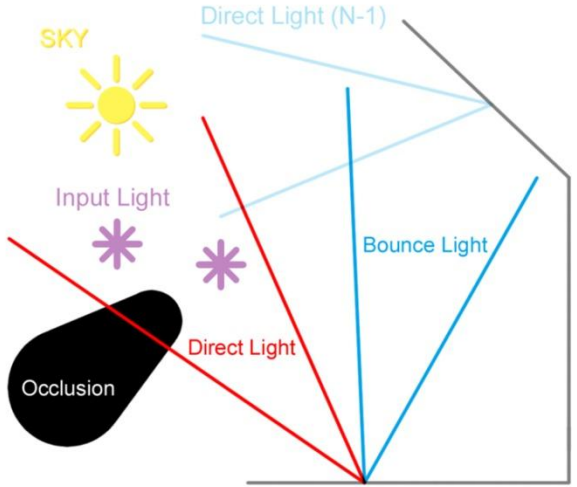
Multi-Bounce Cone Trace

- For a given voxel + direction, a cone trace to determine direct illumination, and one for 1st or 2nd bounce light, all touch the same voxels, and accumulate the same occlusion information.
- Just need to provide extra textures as input to the cone trace,and return multiple results.
- Have to be careful about how we accumulate to ensure we don’t get feedback!

- 与えられたボクセルと方向、直接照明を決めるコーントレース、1，または2番目の反射光、同じボクセルに触れるすべて、そして同じ遮蔽情報の蓄積
- コーン･トレースへの入力として余分なテクスチャを提供し、複数の結果を返す必要がある
- 注意する点は、データーパイプラインが欲しいということ、フィードバックは欲しくない

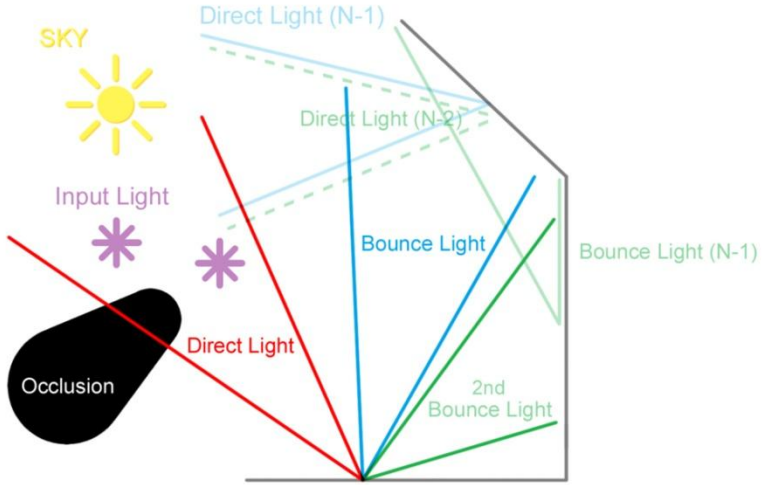
Cone Trace Inputs

- Occlusion cascade.
- Input Light cascade– radiance from point lights.
- Direct Light cascade – all direct lighting at a voxel on the last step.
- Bounce Light cascade – Light that bounced on the last step
- オクルージョンカスケード
- ライトテクスチャからの入力 - ポイントライトからの発光
- 直接光のカスケード - 最後のステップ上のボクセルは全て、直接照明
- ダイレクトライトカスケード　最後のステップ上のボクセルにあるすべてのダイレクトライト
- バウンスライトカスケード - 最後のステップのバウンスライト



Cone Trace Outputs

- Direct Light cascade
- Bounce Light cascade
- Bounce Bounce Light Cascade – Direct Light + 1st bounce light + 2nd bounce light + extra magic. Used by Screen Space Cone Trace.
- ダイレクトライトカスケード
- バウンスライトカスケード
- バウンス、バウンスライトカスケード
- Direct Light + 1st bounce light + 2nd bounce light + extra magic. スクリーンスペースコーントレースを使用



Extra Magic

- Two bounces at voxel granularity is good, but would like more.
- Fake more by looking for places that received more second bounce of light than first bounce, and boost them slightly.

- 粒状ボクセルでの2回の反射は良い結果だが、もっと良くしたい
- 一回目より、二回目の反射光を受けた場所を探して、それらを少しブーストさせる

```
float3 bounce_diff = min(10.f*max(second_bounce -
        bounce,0.f), second_bounce*0.5f);
```

Propagation

- Propagate irradiance up our cascade levels.
- Do this for all 3 irradiance textures, Direct, Bounce and Bounce Bounce.
- These textures will be scrolled to as we move around.
- Missing edge info taken from next cascade up.

- カスケードレベルの放射照度をアップし伝播する
- 3つの照度テクスチャ、方向、バウンスとバウンスバウンスを行う
- これらのテクスチャは、私たちが移動すると、スクロールを行う
- 次のカスケードアップから欠落したエッジ情報を取得する

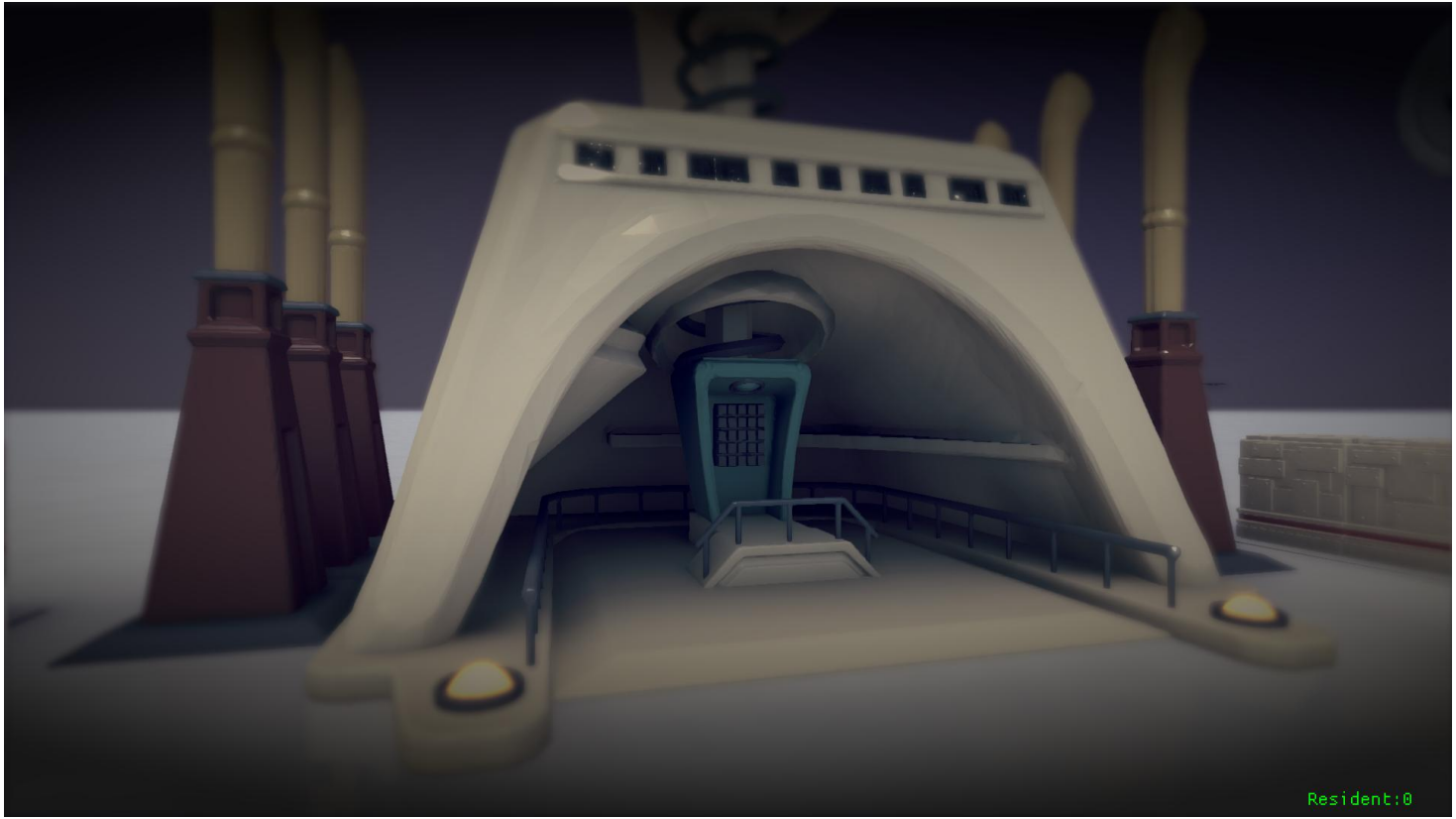
Screen Space Cone Trace

- Final pass, handles *all* per pixel diffuse lighting, both direct and indirect.
- Trace at 1/4 dimensions, and intelligently upscale. (16 directions again)
- Build up append buffer of “fail case” pixels as we upscale, and use dispatchIndirect() to do extra cone traces.
- Blend starting cascade for trace based on distance.

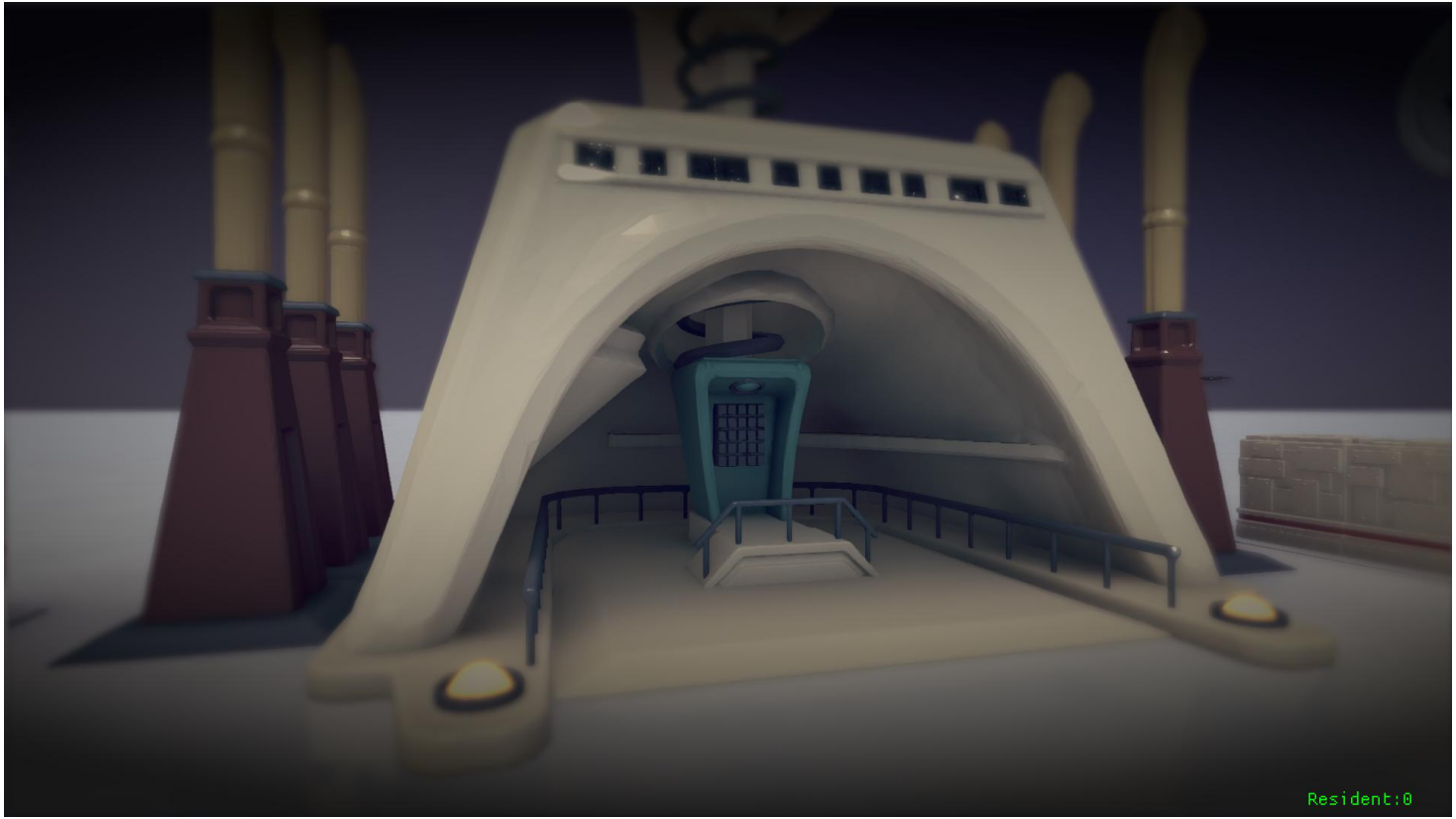
- 最後のパスは、直接、間接を問わず、すべてのピクセルあたりの拡散照明を、処理する
- 16分の1の面積でトレースし、賢くアップスケールする（再び16方向で）
- スクリーンすべてをアップスケールをしながら情報が足りない場合、登録をした「失敗ケース」ピクセルのアペンドバッファを構築する　dispatchIndirect()
- 距離に基づいて、トレースの開始カスケードをブレンドする

Direct Only

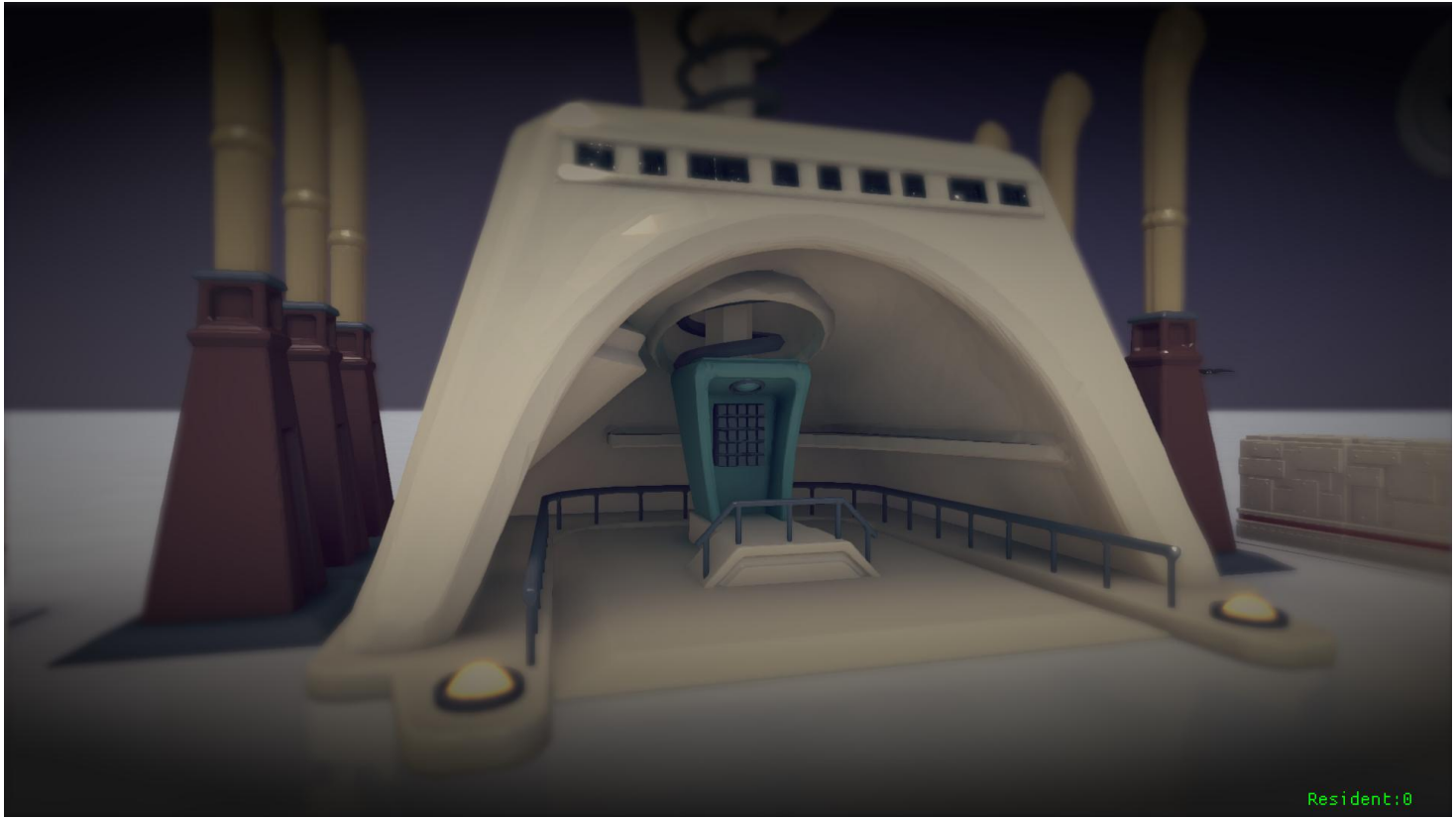
One Bounce



Two Bounces



Three Bounces



最適化について

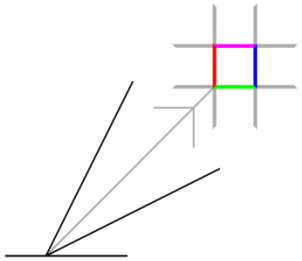
- Cone Tracing is still slow with even with a 3D texture cascade.
- 10’s of ms for final screen space traces.
- Way too many texture lookups.

- コーントレースは3Dテクスチャカスケードを使っても遅い
- 最終的なスクリーンスペースのトレースのために10ミリ秒かかる
- テクスチャルックアップが非常に多い

Pre-Combine Anisotropic Voxels

- For each Cone Trace step, we must interpolate between values from 3 voxel faces.
- Weighting is determined by the direction we trace.

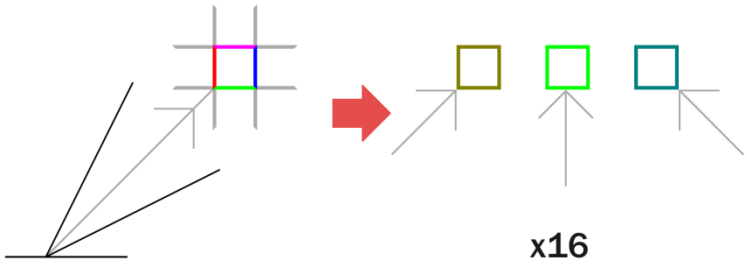
- 各コーントレースステップのために、私たちは3つのボクセル面からの値を補間する必要がある
- トレースの方向によって、重み付けが決定される



Pre-Combine Anisotropic Voxels

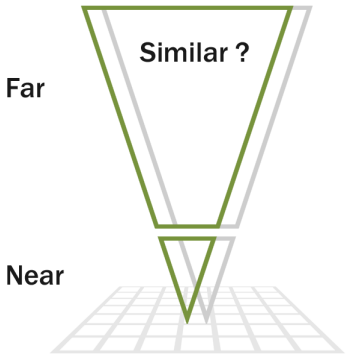
- But our directions are fixed.
- Pre-combine and store for each of our 16 directions (in a (16*32)x(6*32)x32 texture!)
- 1/3 the texture cost.

- しかし、私たちの方向は固定されている
- 16方向のれぞれについて予め結合して保存を行う（（16 *32）×（6*32）x32のテクスチャ！）
- 3分の1はテクスチャコスト



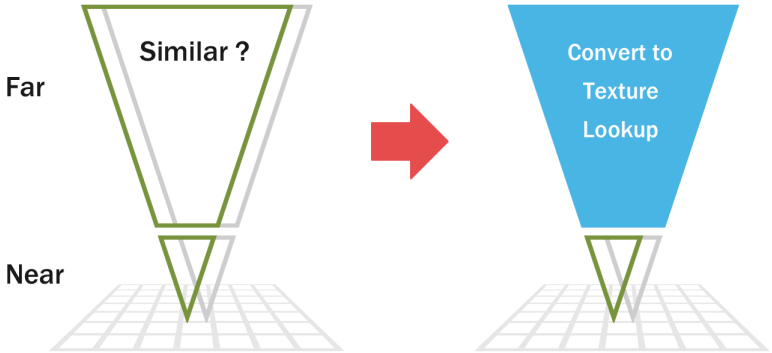
Split Cone Trace

- Think of two cones traced in the same direction that are close in world space.
- The samples we take as we trace each cone will become increasingly similar the further down the cone we get.
- This work is redundant.
- ワールドスペースで同じ方向にトレースされているコーンが2つある場合
- それぞれのコーンをトレースする度に、サンプルはコーンの下に行くほどどんどん似てくる
- この作業は冗長です



Split Cone Trace

- Build another texture cascade with the “far” cone data, for each of our 16 directions.
- Per-pixel, only trace the “near” half of the cone (using our pre-combined cascade).
- Interpolate the “far” data from our “far” texture, and combine.
- 16方向の遠くにあるコーンデータを持つテクスチャカスケードを作る
- ピクセルごとに、コーンに近い半分をトレースする（事前に融合したカスケードを使用）
- 遠くのテクスチャから、「遠い」データを補間し、組み合わせる



Fine Detail

- Cone tracing gives us a lot of large scale lighting detail
- But our smallest voxels are only 0.4 meters.
- Still need augment with fine detail computed in Screen Space.

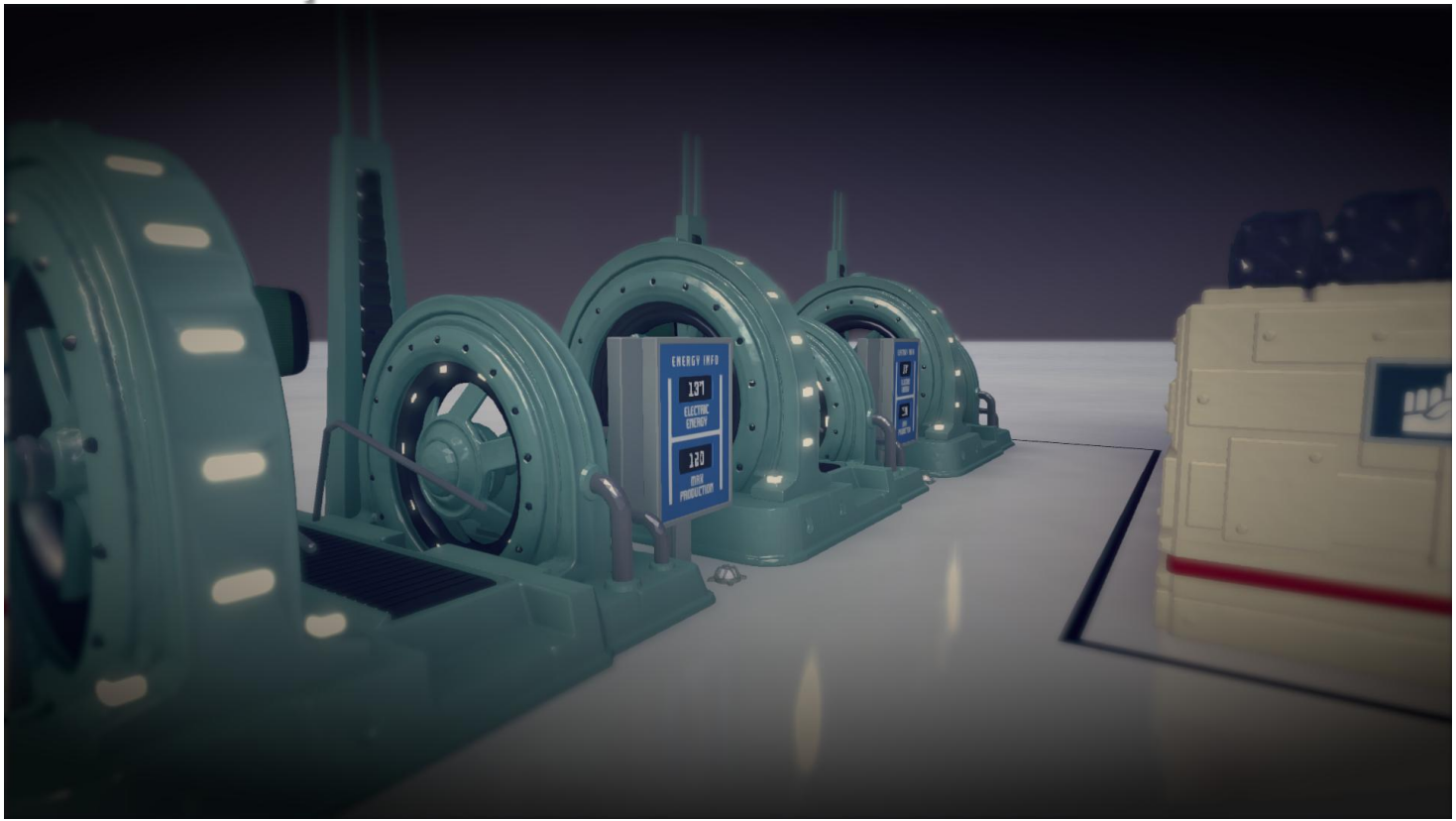
- コーントレースは私たちに、大きなスケールライティングのディテールを与えます
- しかし、この世界で扱う最も小さいボクセルは最小0.4メートルです
- さらにスクリーンスペースで細かく計算されたディテールを補強する必要がある

Screen Space Directional Occlusion

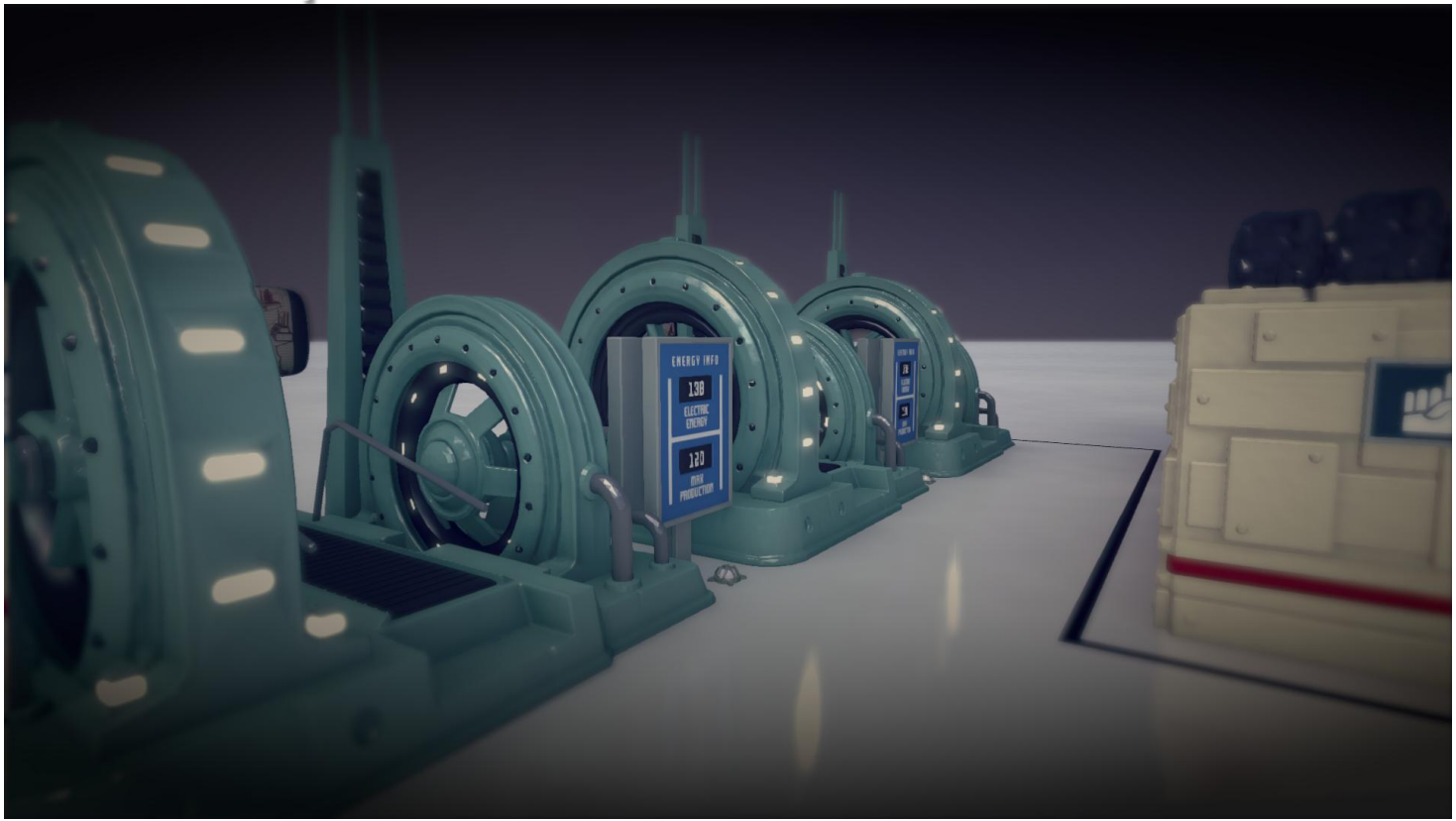
- Integrate 2 band SH, rather than a scalar occlusion value.
- Easy to convert this into a visibility cone.
- Intersect the visibility cone with cones for each of the directions we trace, and modulate the incoming light accordingly.
- See “Ambient Aperture Lighting” (Chris Oat) for Cone-Cone overlap approximation details.

- スケーラーオクルージョンバリューではなく、２バンドSHを実装した
- ビジビリティコーンを２バンドSHにコンバートするのは易しい
- トレース方向のそれぞれのコーンと交差するビジビリティコーン、
- それぞれトレースする方向で、ビジビリティーコーンをコーンで横切り、それに応じて入射光を変調する
- コーンとコーンのオーバーラップの近似についてはこちらを参照してください。“Ambient Aperture Lighting” (Chris Oat)

Screen Space Occlusion - Off



Screen Space Occlusion - On



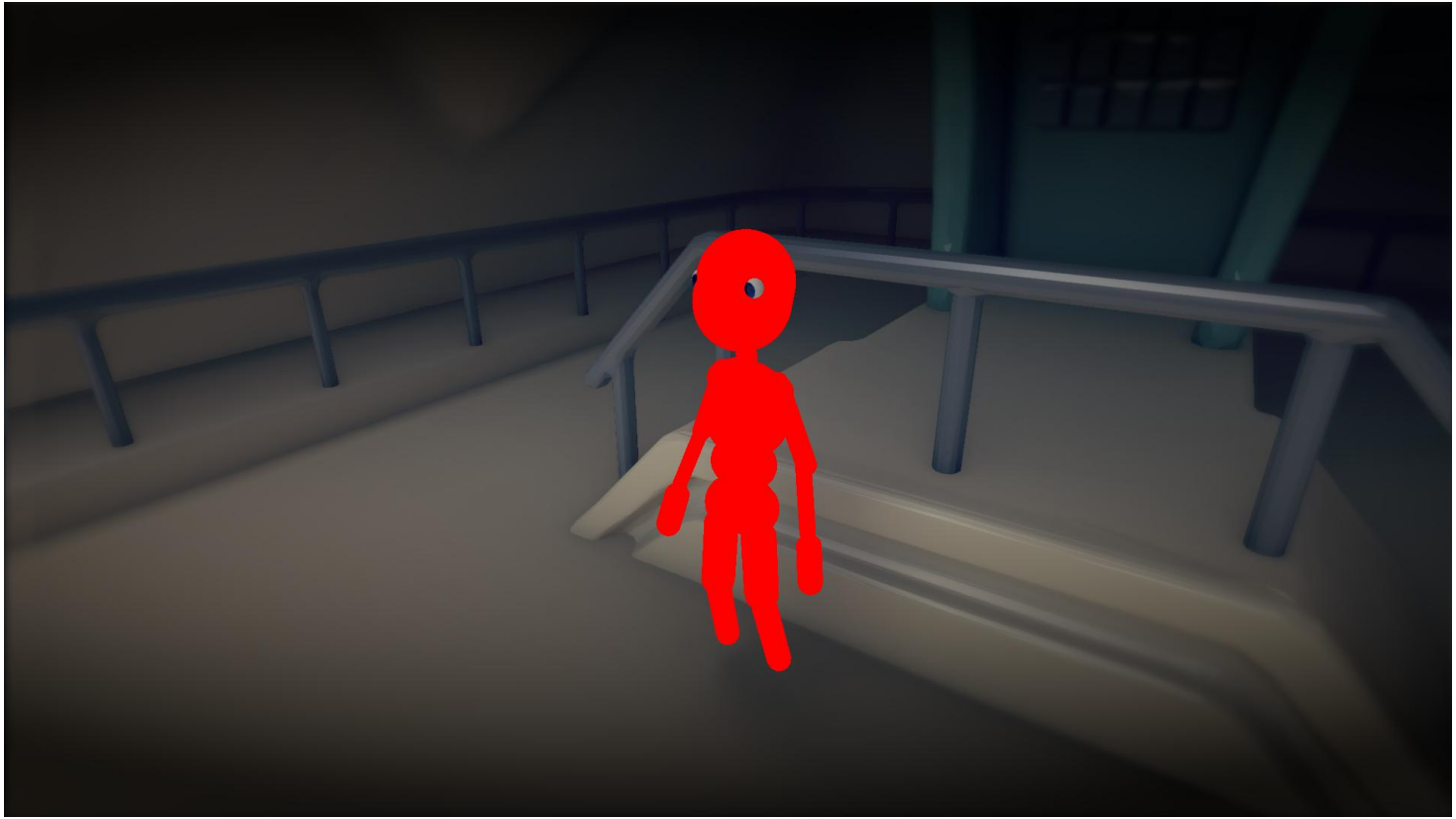
キャラクターのシャドウ

キャラクターのシャドウ

- Characters are not voxelized, due to size.
- Would also cause extra voxelization overhead.
- Use collision capsules, perform cone occlusion tests instead.
- Similar to “Lighting Technology of Last of Us”, but in 16 directions, rather than just 1.
- Fills a 16 deep screen space texture array.

- キャラクターはサイズが小さく、ボクセライズに含まれない
- また、余分なボクセル化のオーバーヘッドの原因になる
- かわりにコリジョン用カプセルをつかって、コーンオクルージョンテストのかわりにする
- “Last of Us”のライティング技術に似ていますが、こちらは1方向ではなく16方向
- １６のディープスクリーンスペーステクスチャーアレイを埋めます

キャラクターのシャドウ



Characters - On

Characters -Off

乗り物のシャドウ

- Not easy to define with capsules.
- Integrate visibility into 2 band SH and store in a 3D texture.
- Easy to intersect again with cones in our 16 directions.
- Apply to same screen space texture array as capsules.

- カプセルに適していない形状の場合
- ビジビリティを2バンドSHと3Dテクスチャーに蓄える
- 16の方向とにコーンを再び交差するのは簡単
- カプセルと同じスクリーン空間テクスチャ配列に適用される

Vehicles - Off

Vehicles - On

パーティクルのシャドウ、オクルージョン

- 16 Cone Traces per pixel per particle – too expensive!
- Use a simplified 2 band SH Cascaded Texture, like simple irradiance probes.
- Tessellate, and sample per vertex.
- Particles also fill Dynamic Occlusion texture.
- Feeds into cone trace. Provides self occlusion and shadowing.

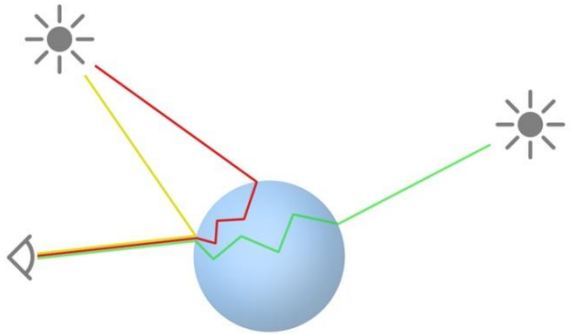
- １６方向のコーンとレースをパーティクル単位で行うのは高価すぎ！
- 単純な照度探索のようなシンプルな2バンドSHカスケードテクスチャをつかう
- テッセレーションと頂点単位のサンプリング
- パーティクルはダイナミック･オクルージョンのテクスチャを埋める
- それをコーントレースに与える。セルフオクルージョンとシャドウを提供する。

Particles – No Occlusion

Particles with Dynamic Occlusion

サブサーフェーススキャッタリング

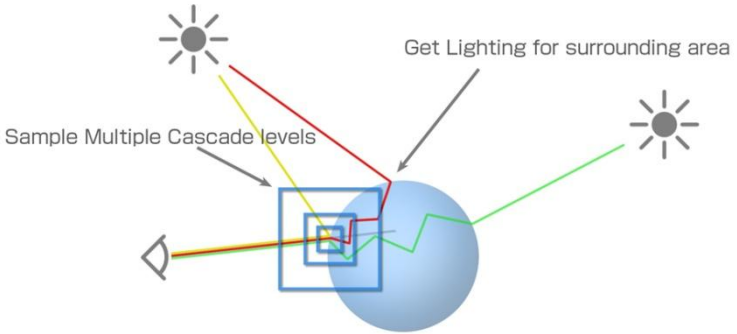
- To give a SS look we need to simulate light that has bounced inside the material.
- Possible to work in texture space or screen space and blur.
- Doesn’t necessarily deal with light bleeding from behind the object.
- SSの見た目を与えるには、マテリアルの内部に光がバウンスするシュミレートが必要
- テクスチャースペースもしくはスクリーンスペースとブラーで動作するように
- オブジェクトの背後から光がある場合は正しく計算されない可能性がある



サブサーフェーススキャッタリング

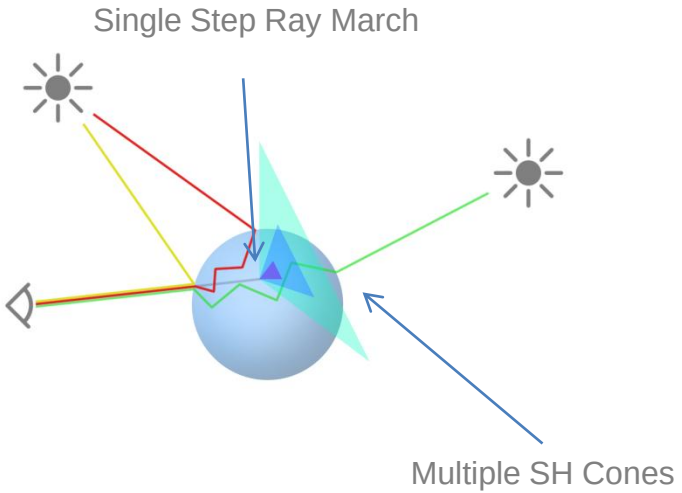
- SH Texture for particle lighting can quickly give us the irradiance at any point.
- Moving up cascade levels gives us the average over a wider area.

- 粒子照明用のSHテクスチャは任意の点での放射照度を与えることができる
- パーティクルライティングのSHテクスチャーは任意の点での照度を与える
- カスケードレベルの上に行くほど、光量のデータより、広い地域のデータが取れる

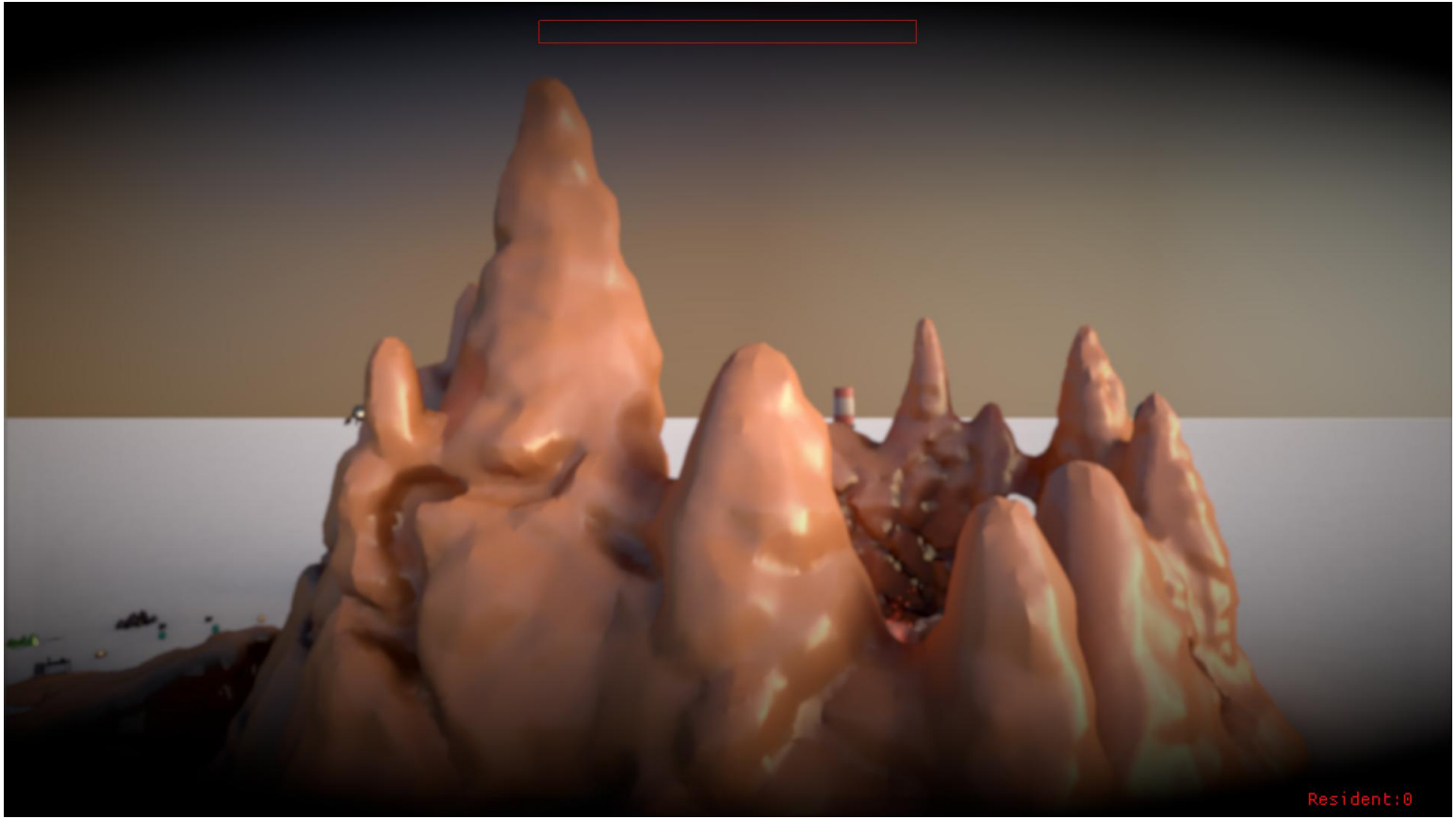


サブサーフェーススキャッタリング

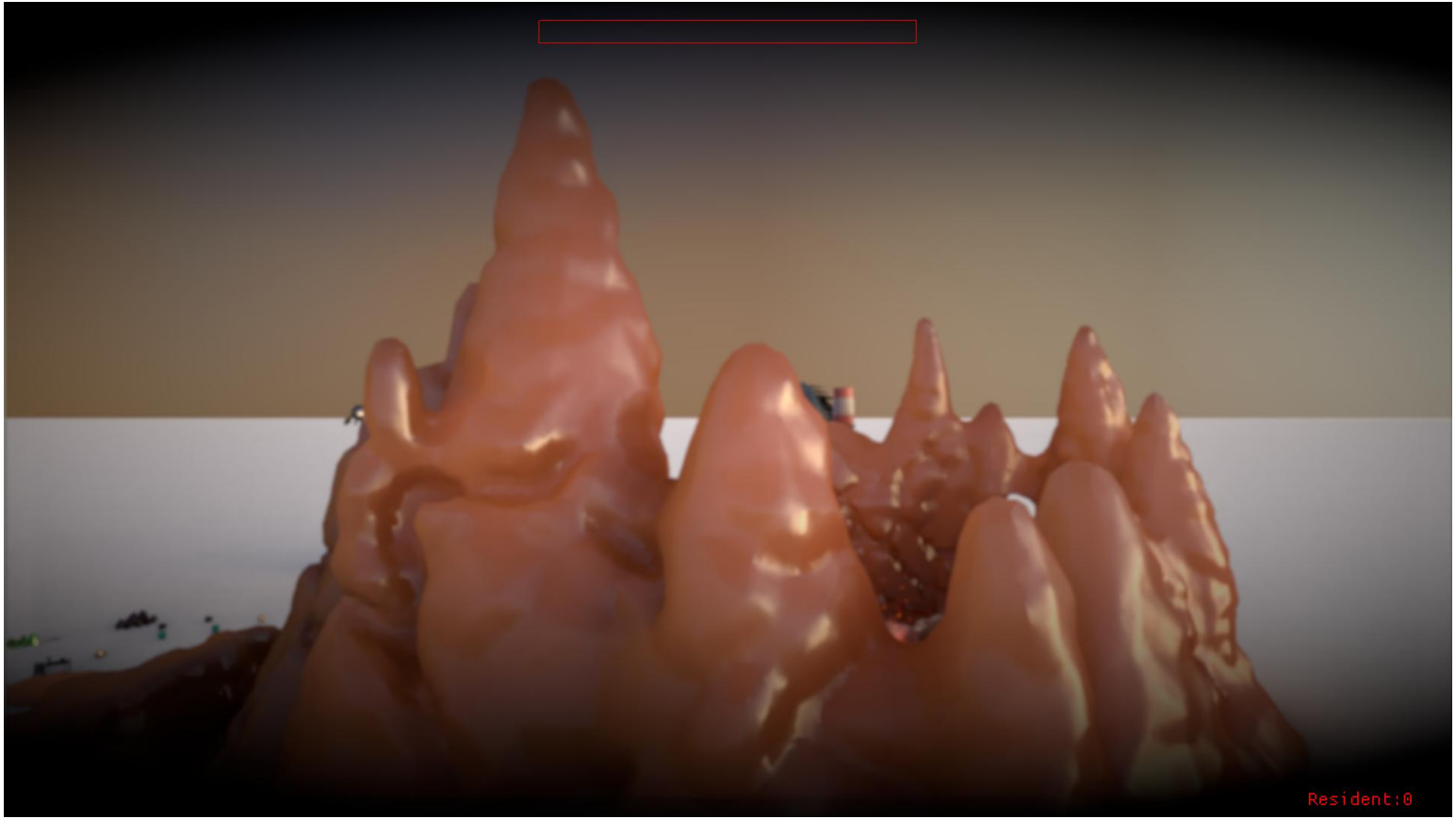
- Also allows us to raymarch away from the eye.
- Gathering light falling on the back of the object.
- Change SSDO calc to provide screen space thickness. Used to “frost” thin objects.
- プレイヤー視点からレイマッチすることができる
- ギャザリングライトは被写体の後に当たる
- スクリーン空間の厚さを提供するために、SSDOの計算値を変更する。”フロスティ”のオブジェクトに使用



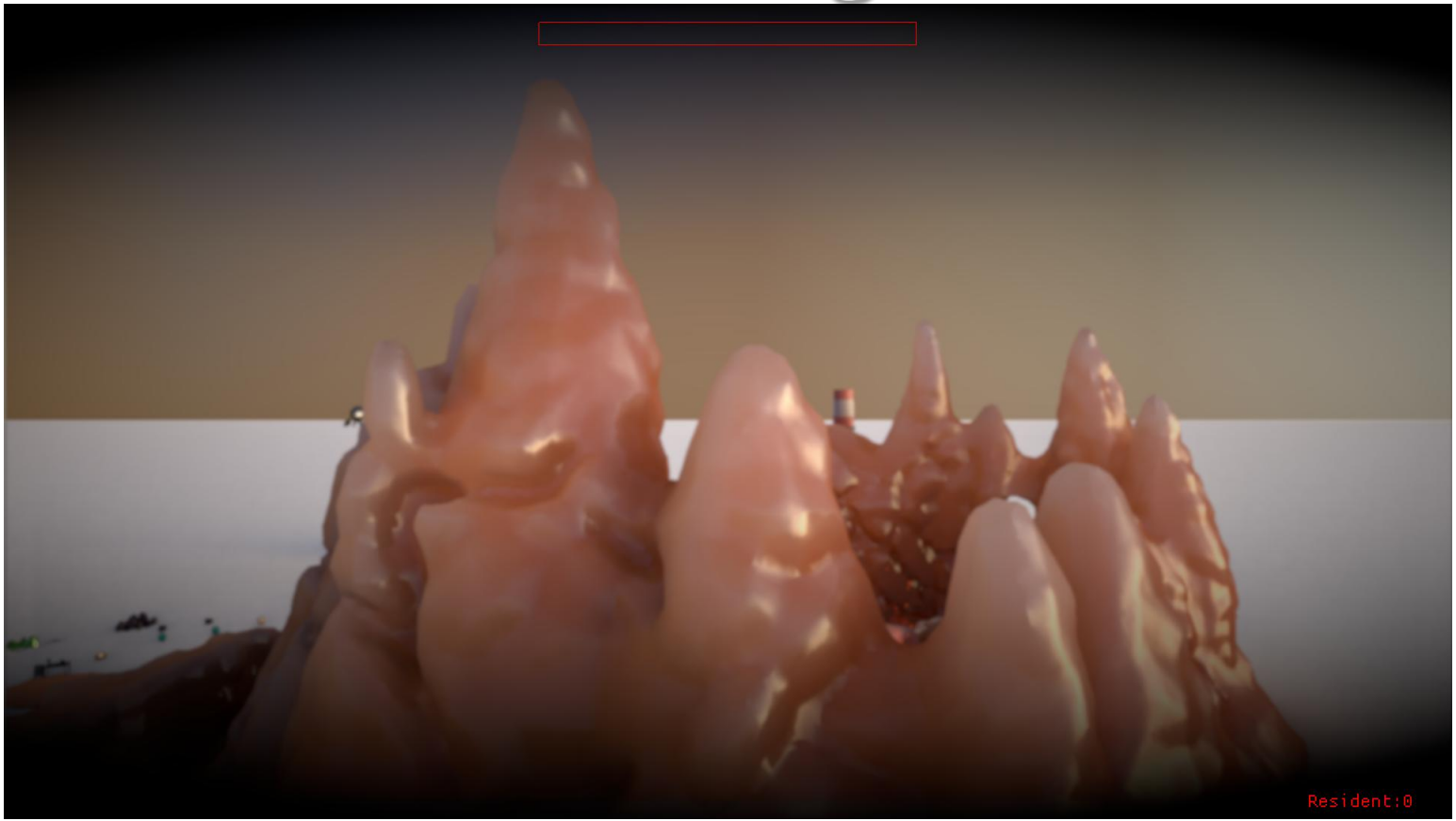
Subsurface - Off



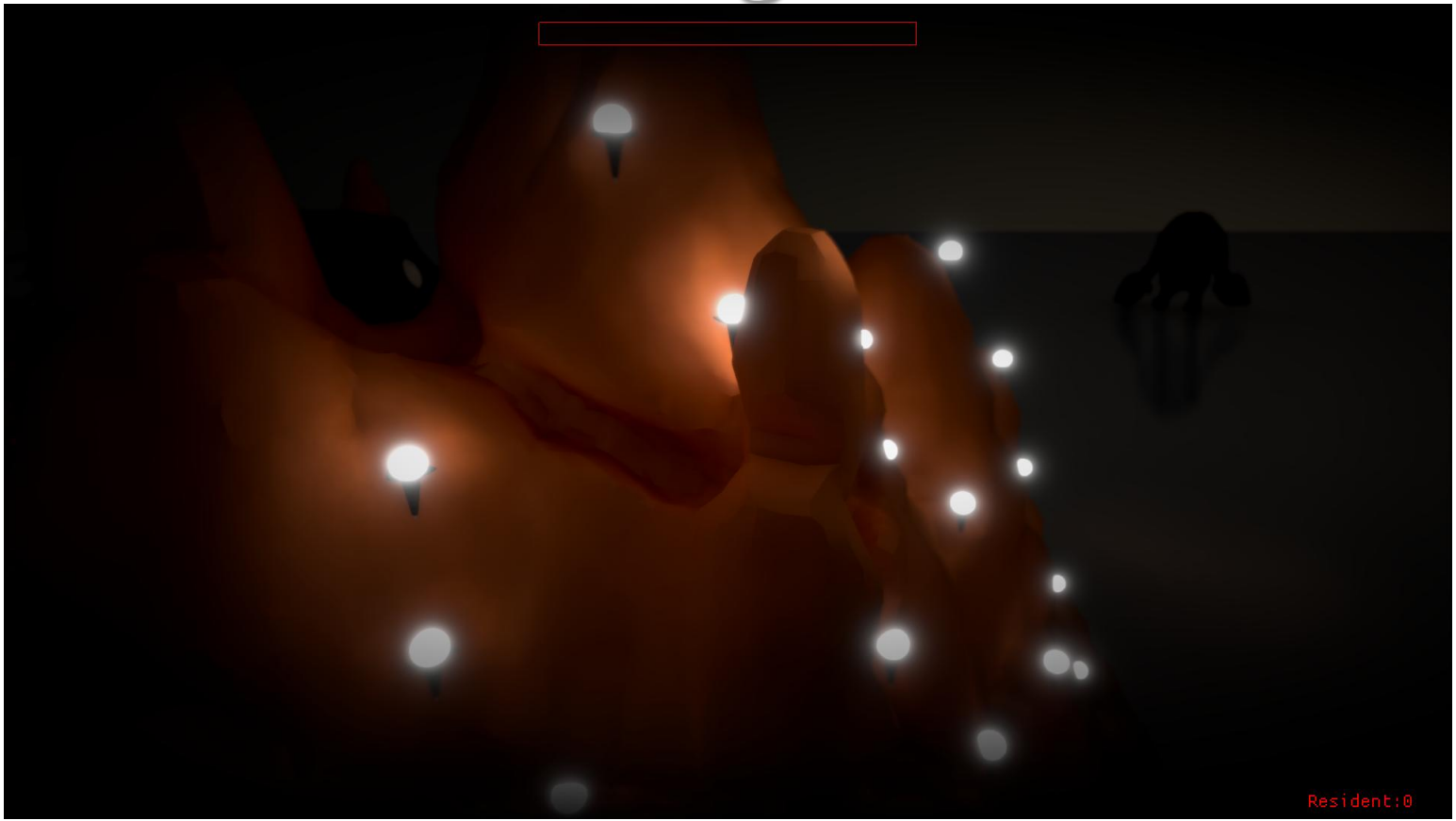
Subsurface - On



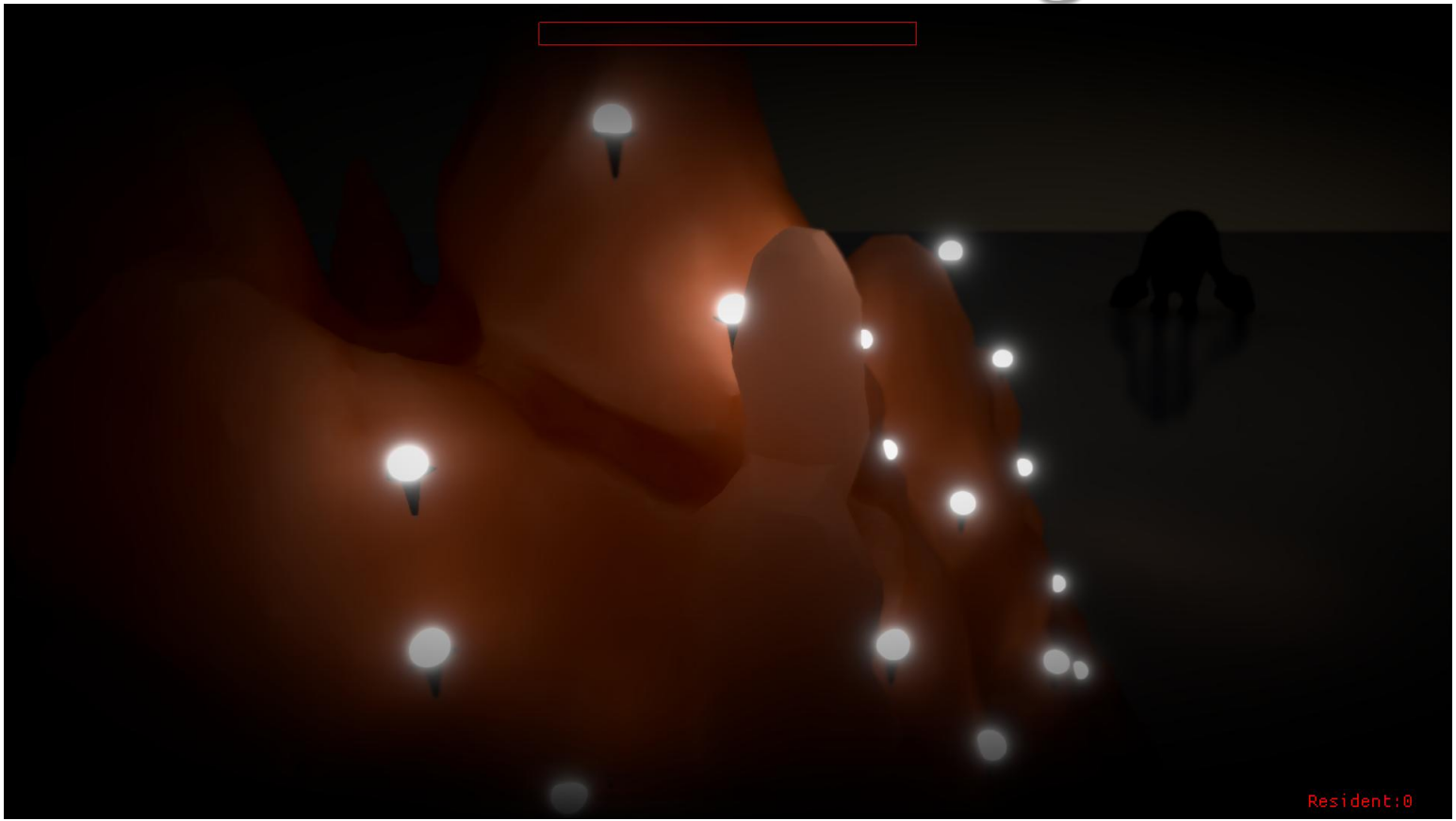
Frosting



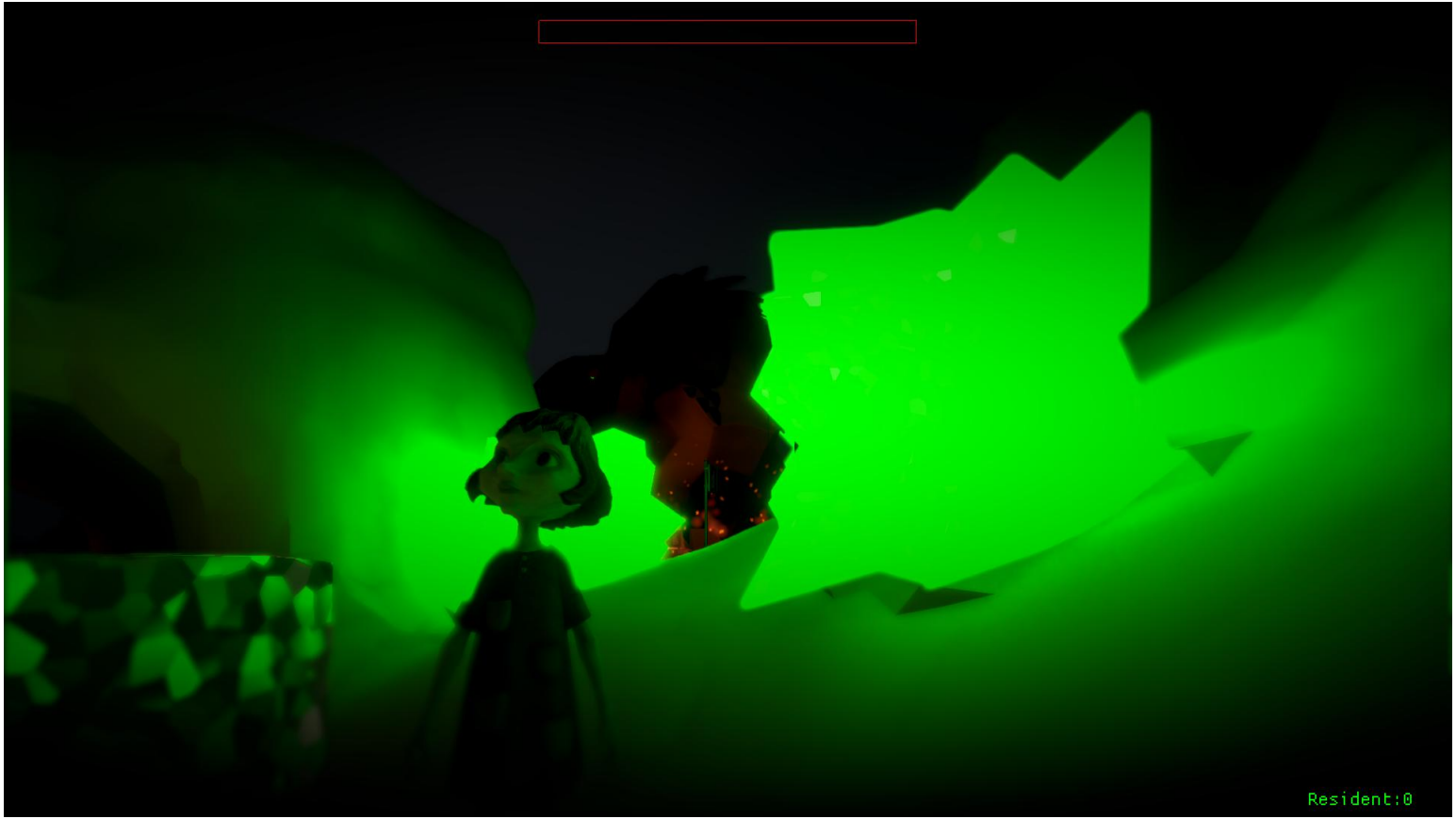
At Night



Subsurface – At Night



Emissive Materials

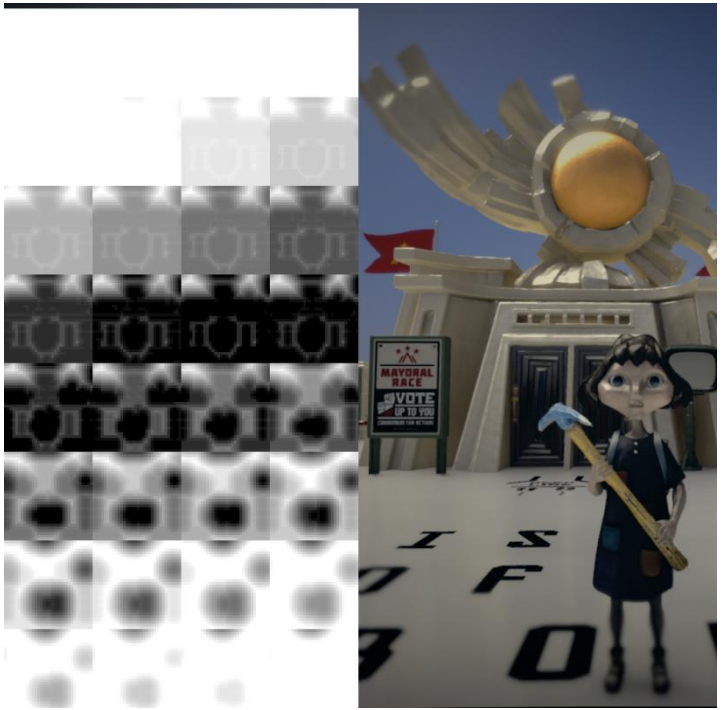


Ray Marched Reflections

Signed Distance Fields

- Fast to build.
- Uses Jump Flooding.
- One for landscape and objects, and one for dynamic lights.
- Can be extended to work with Voxel Cascades.

- 素早い構築
- ジャンプフルーディングを使用
- 1つは風景やオブジェクトのため、もうひとつはダイナミックライトの為
- ボクセルカスケードで動作するように拡張することも可能



レイマーチによる反射表現

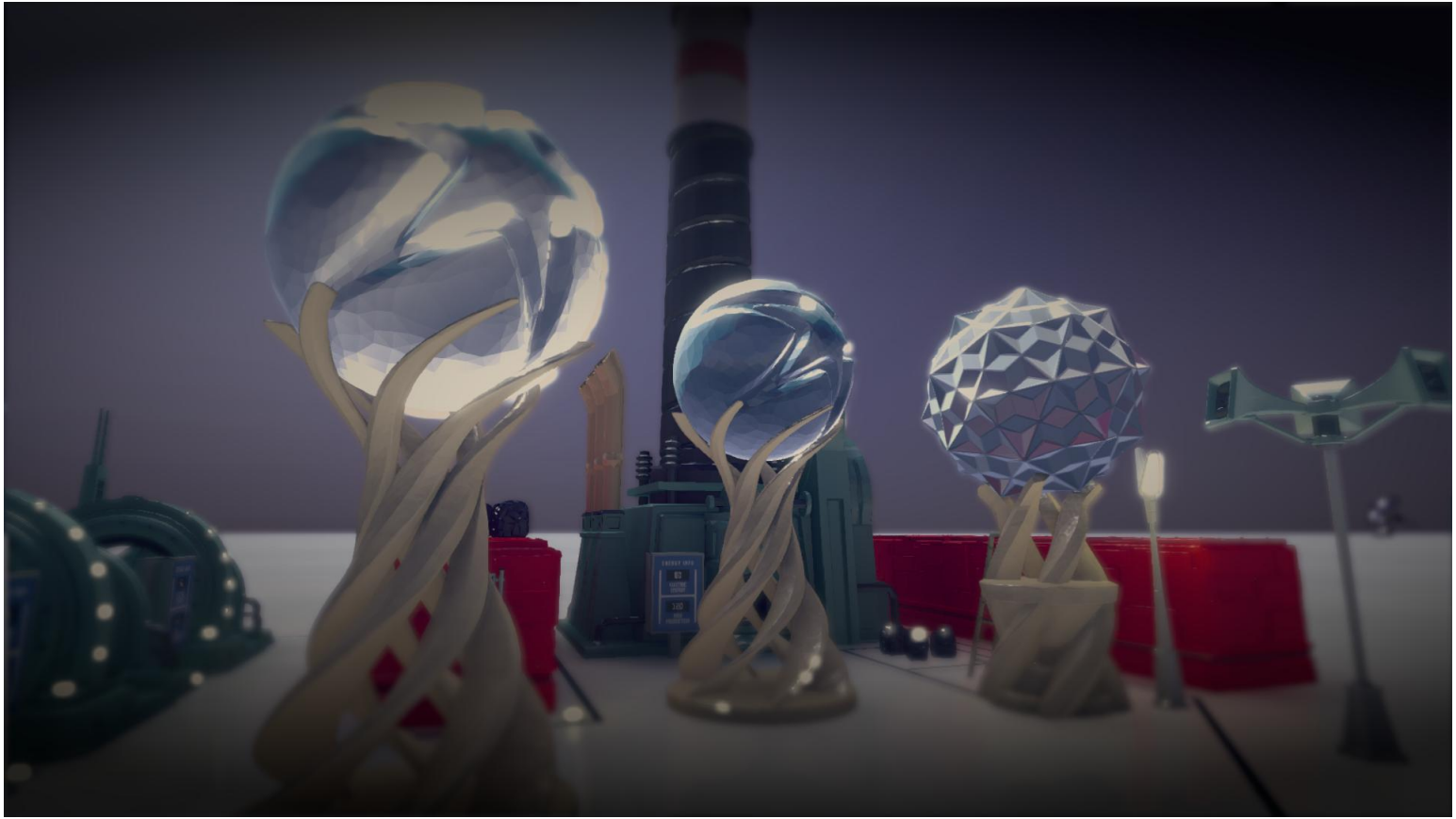
- Ray march through these distance fields.
- Sample from SH cascades to accumulate radiance.
- Provides very rough view of the world.
- But not dependent on screen space.
- レイマーチはディスタンスフィールドを通る
- SHカスケードからのサンプリングは、輝きを蓄積する
- ワールドの非常に大まかなビューを提供する
- しかしスクリーンスペースには依存しない

レイマーチによる反射表現

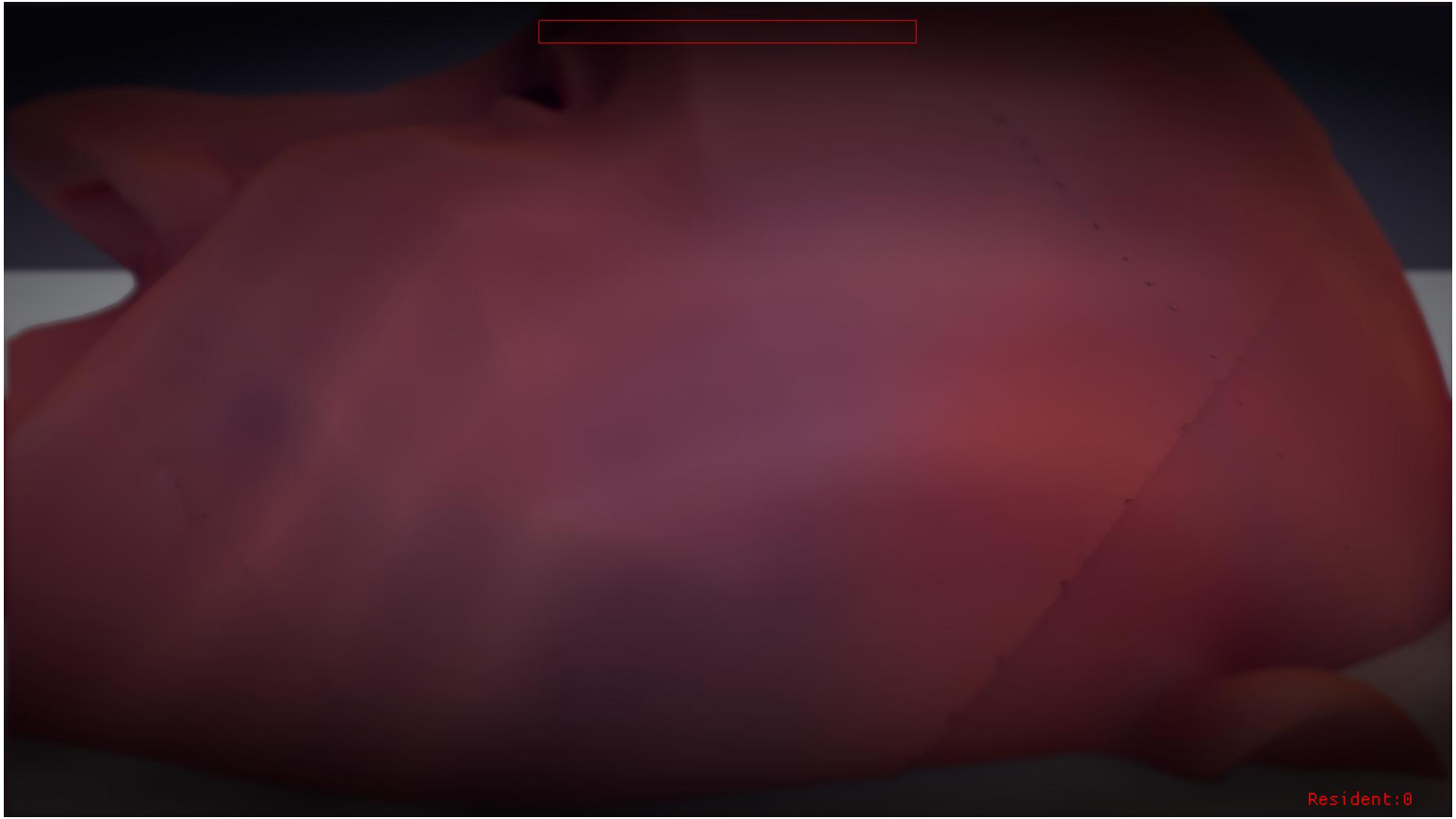
Ray Marched Reflections - On

Ray Marched Reflections - Off

屈折を表現したマテリアル



サンプリングによる問題点

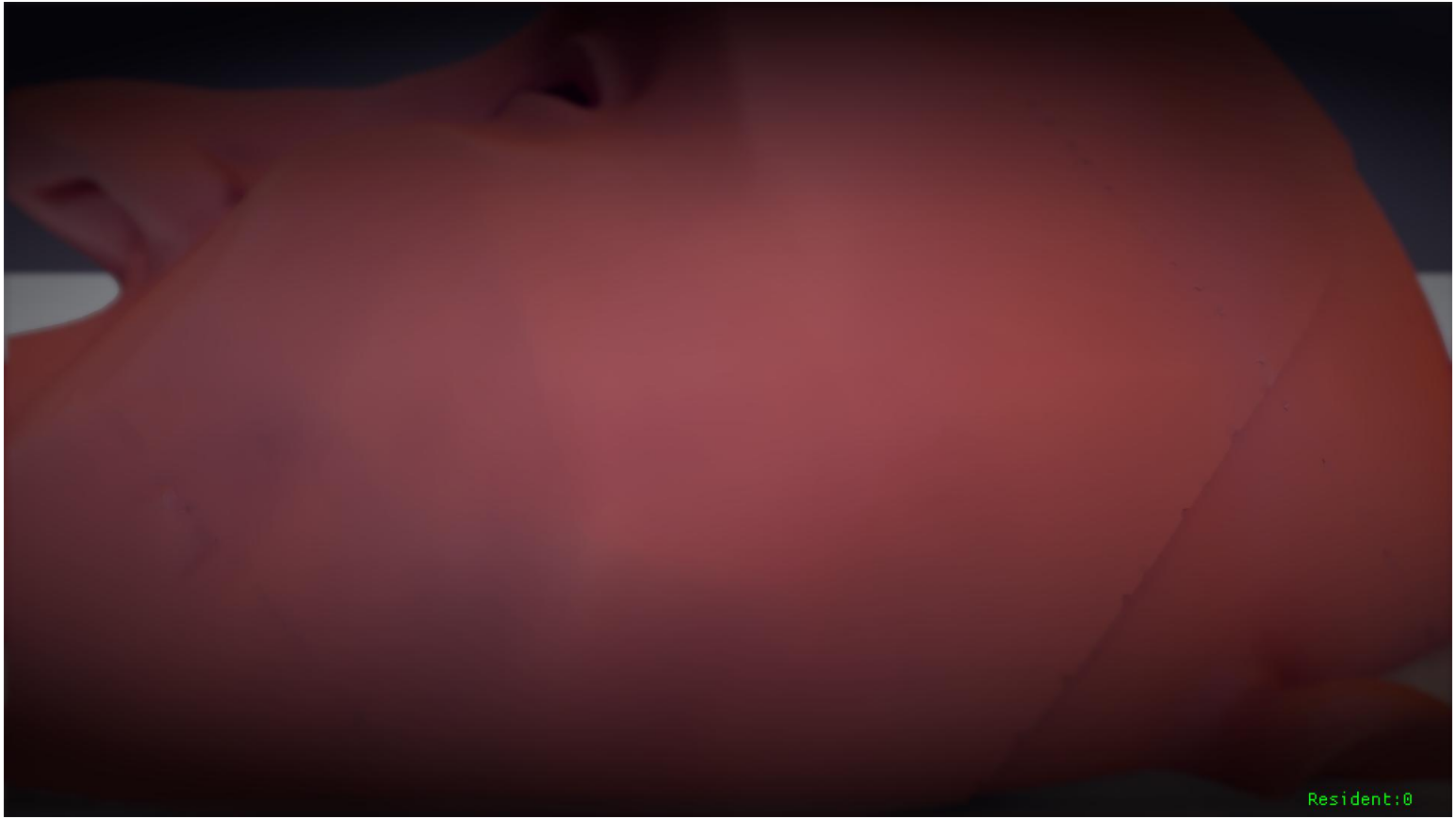


サンプリングによる問題点

- Typically bias Cone Tracing away from the surface by ~0.5 a voxel to avoid self occlusion.
- Still face subtle aliasing issues on planar and smoothly curving surfaces.
- Abuse SSDO again, to get a screen space metric for curvature.
- Increase bias in low curvature areas.
- 通常、バイアスコーンは、表面から～0.5ボクセルでセルフオクルージョンを避けるために、離れている
- 平面上の微妙なエイリアシングの問題や、スムーズに湾曲したサーフェイス問題がある
- 歪のためのスクリーンスペースを得るために、再びSSDOを酷使する
- 低曲率の領域の偏りを増やす

Screen Space Curvature

Modified Bias



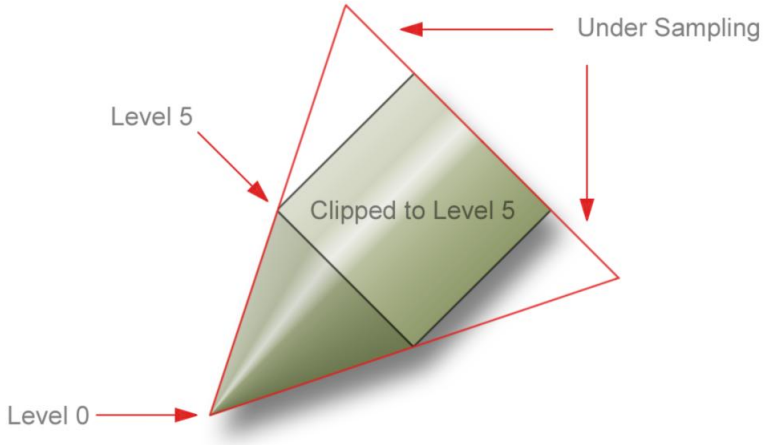
Wide Cones + Direct Lighting

- Only have 6 levels of cascade.
- Want to trace cones further than we have data for (our landscape objects are huge!).
- A Clipmap doesn't fit naturally with our texture addressing.
- Data tends to propagate up mipmaps faster than our cone trace ascends them.
- The top MIP voxel (nearly) always ends up semi opaque. Not good for direct lighting.

- 私たちは6レベルのカスケードしか持っていない
- トレースしたいコーンの広さがゲームのデータを上回る(私たちの地形はとても巨大!).
- クリップマップは、テクスチャアドレッシングに自然にフィットしない
- 私たちのコーン･トレースはそれらが上昇するよりも速くミップマップを伝播する傾向がある
- トップMIPボクセルは常に半不透明である。これは直接照明のためによくない

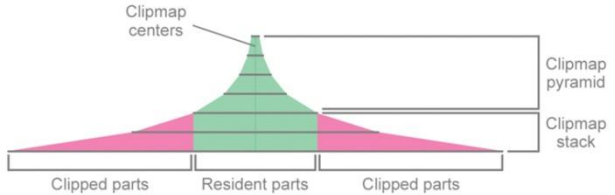
現在の解決方法

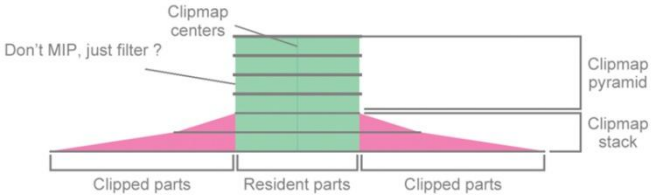
- Currently we just clamp to the top cascade level.
- Not an ideal solution.
- Causes Undersampling.
- Sometimes visible on shadows of large objects.
- 現状ではトップのカスケードレベルをクランプしている
- 理想的な解決法ではない
- アンダーサンプリングを引き起こす
- 巨大なオブジェクトの影などで顕著



将来的な解決法?

- Extra cascade levels
- Similar to Clipmap, but voxel resolution stays the same (Not a MIP)
- Prefilter our precombined direction voxels in the plane perpendicular to the direction?
- エクストラカスケードレベル
- クリップマップに近いが、ボクセルの解像度は（MIPではない）同じまま
- 16方向ごとにコーン方向と垂直する方向に事前に準備したカスケードをフィルタします





メモリとパフォーマンスについて

- ~3ms to update our cascades (only one level done a frame), more if we have to voxelize.
- ~3ms to do our screen space cone tracing
- ~3.5ms for specular ray march.
- ~2.5ms to do our final upscale, and combination with our various occlusion textures.
- 600mb+ of textures for voxel data and the like.

- ～3ms　カスケードのアップデートに掛かる時間（フレームに一つのレベル）　ボクセライズしなければならない場合
- ～3ms　スクリーンスペースのコーントレースに掛かる時間
- ～3.5ms スペキュラーのレイマッチングに掛かる時間
- ～2.5ms 最終的なアップスケールと様々なオクルージョンテクスチャーの組み合わせに掛かる時間
- 600mb+　ボクセルデータやそれに関するテクスチャ容量

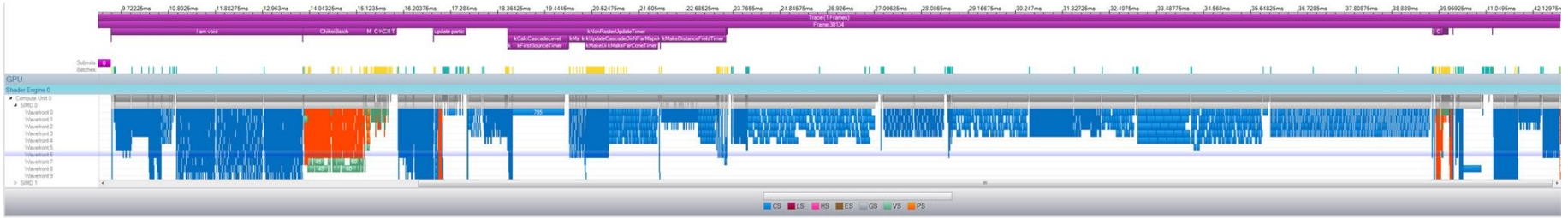
Async Compute

- Most of our Screen Space (and Voxel Space) shaders have been moved to Compute.
- Frame is pipelined. Post processing overlaps Gbuffer fill for the next frame.
- Massive win compared to just graphics pipe.
- ~5ms back on a 33ms frame from using Async Compute.
- Everyone should do this!

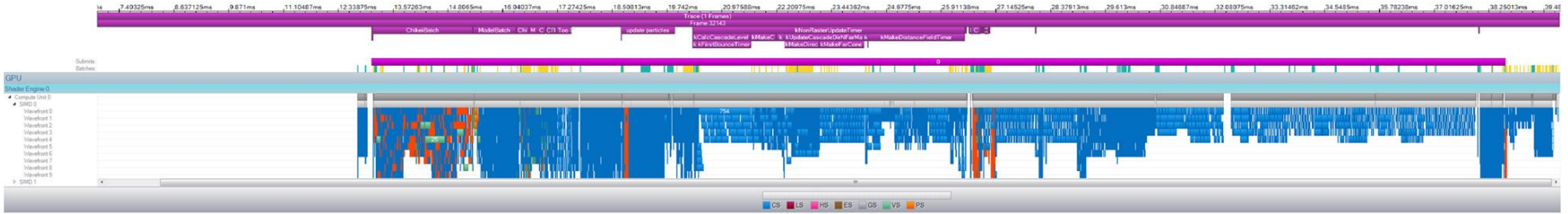
- スクリーン･スペース（およびボクセルスペース）シェーダのほとんどは、コンピュートシェーダに移した
- フレーム処理はパイプライン化する。ポストプロセスは次のフレームとGバッファのフィルが重なる
- GPUをグラフィックだけでなく、色々なことに使うのが効果的
- Asyncコンピュートの33msから約5ms稼げる
- 皆さんこれを使いましょう！

Async Compute

Graphics Pipe: ~ 33ms



Async Compute: ~ 27ms



将来に向けて・・・

- Higher Frequency Shadows.
  - Possibly Voxel Soft Shadows.
- Higher Resolution Grid.
  - Investigate using a brick map.
- Bounce from Characters.
  - Some form of limited injection.
- Improve Material Model.
  - Currently not very physically correct.

- 高解像度のシャドウ
  - 出来ればボクセルソフトシャドウも
- 高解像度のグリッド
  - ブリックマップの効果を調べる
- キャラクタからのライトのバウンス
  - 限定されたインジェクションから
- マテリアルモデルの改善
  - 現在は物理的に正しくない


Thankyou!
ご清聴ありがとうございます
• james@q-games.com

We're Hiring!

Q-Gamesで一緒にゲームを作りませんか？

ゲーム作りに情熱あふれるプログラマから、業界歴XX年のベテランプログラマまで幅広く募集しています。

歴史ある京都で楽しくゲームを開発しましょう！！

recruit@q-games.com
http://www.q-games.com/

Q-Games