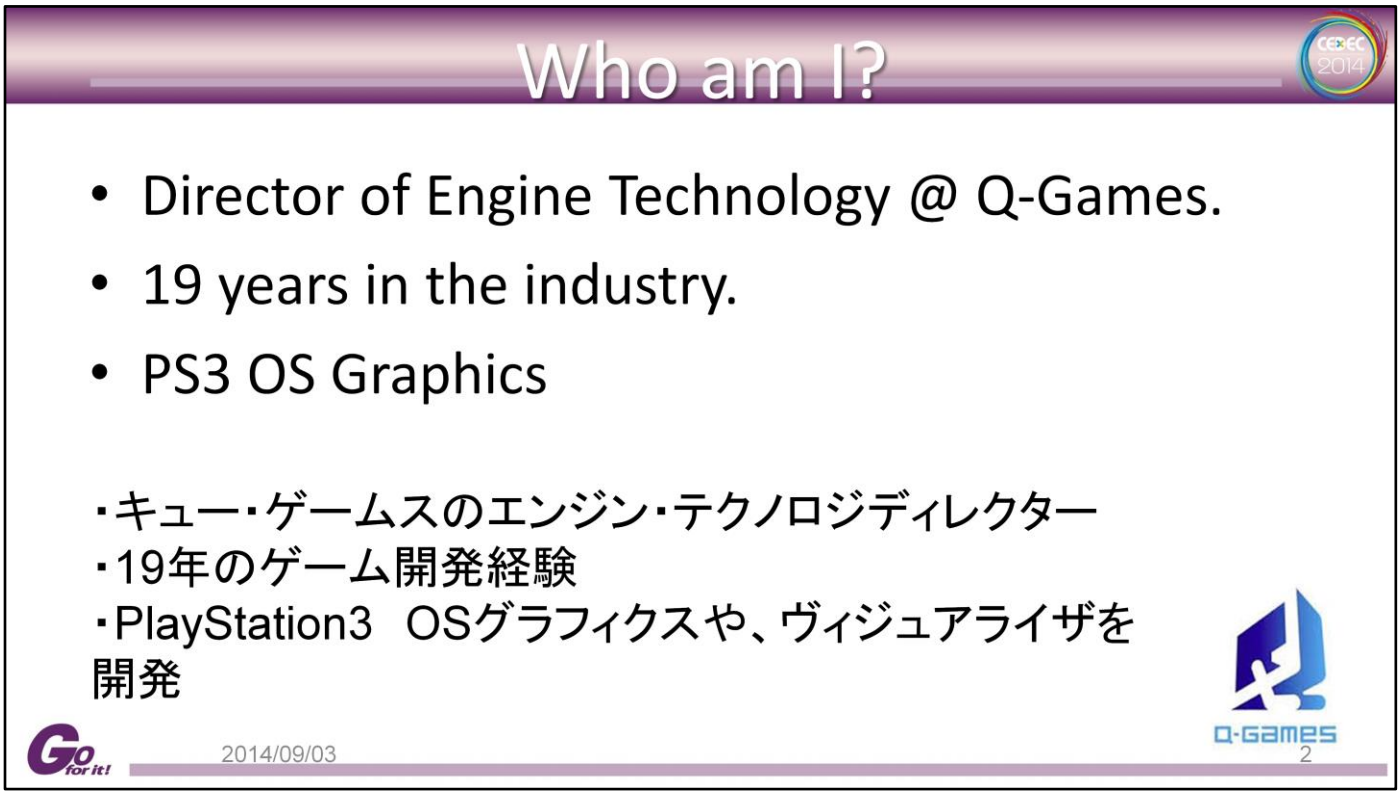
Cascaded Voxel Cone Tracing in The Tomorrow Children

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.



こんにちは、ジェームスともうします。実は日本語が少し出来ますけれども、一時間ほどのプレゼンぐらいの自信がないので助手の吉田さんが日本語の説明をします。

Before I start I'd just like to say thanks for inviting me to give a talk at CEDEC.

For those of you who don't know me,

Hi, I'm James McLaren, the Director of Engine Technology at Q-Games, out in Kyoto, Japan.

I've been doing this whole game dev gig for about 19 years now

And my main claim to fame is that I was lucky enough to be in the right place at the right time to

End up being part of the small 3 man team from Q that worked with Sony on the OS graphics and visualizers on the PS3.

最初にCEDECで皆さんにお話ができることに感謝します。

簡単に自己紹介をいたします。

私はジェームス　マクラーレンです。京都にオフィスを構えるQ-Gamesでテクノロジーエンジニアディレクターとして働いています。

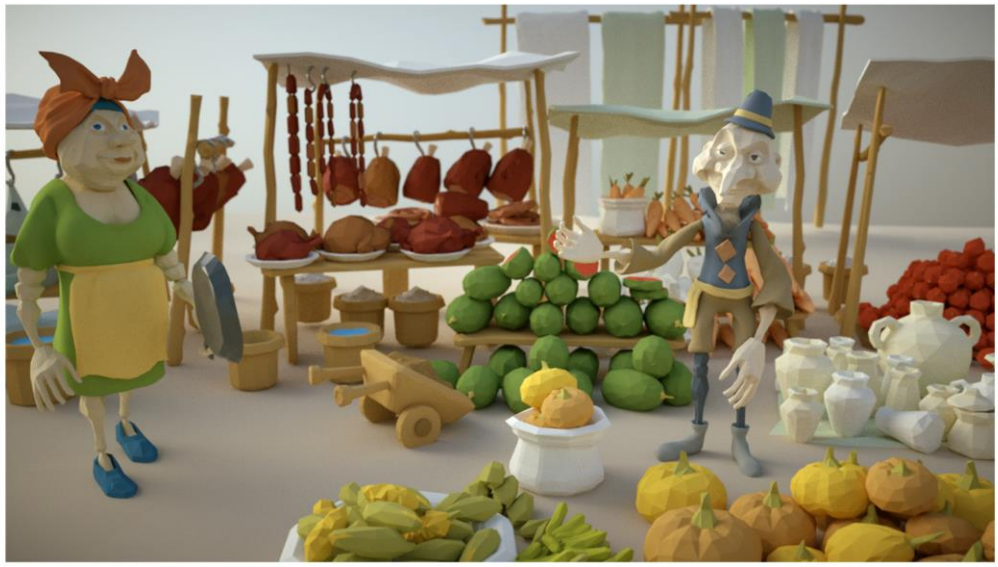
これまでに19年間ゲーム表業界でプログラマをしてきました。

代表的な作品には、Qのテクノロジーチーム（3人の小規模なチームです）で手がけ

た、PS3のOSグラフィクスやビジュアライザのプログラムです。

So hopefully everyone here has seen our game, but in case you haven't here is a our trailer to give you a hint of what we've been making.

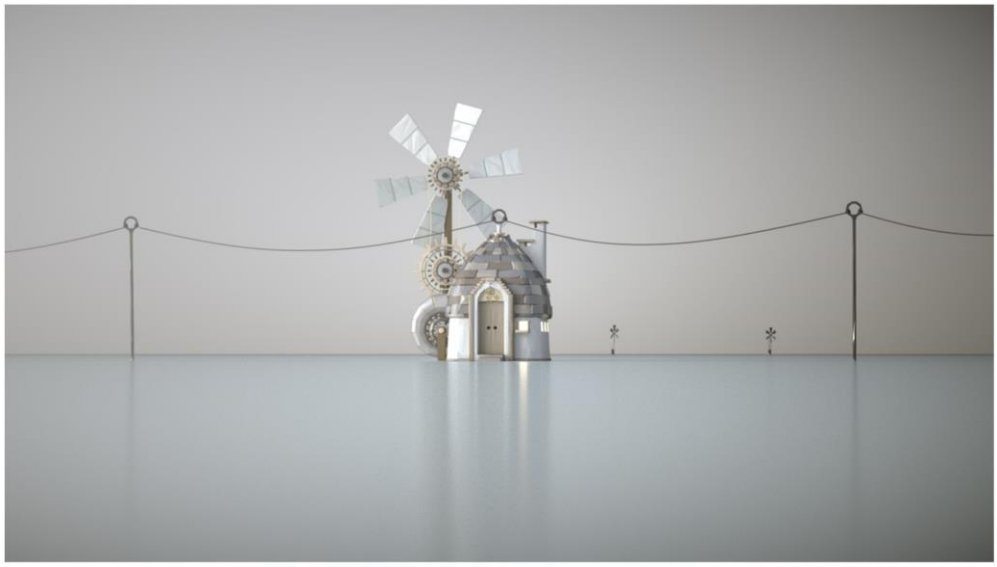
Concept Art



When I first joined the project in early 2012, there was already a good deal of concept art created for the game, showing this very soft look.

私がこのプロジェクトに入ったのは2012年のことです。すでに柔らかいライティングで表現されたコンセプトアートがありました。

Concept Art



The art team was producing all of this with Octane Render, a GPU raytracer

And it became clear very quickly that going for a normal deferred plus shadows rendering solution was not going to cut it

Especially as the world was dynamic.

アートチームはOctane RenderというGPUレイトレーサーを使っていました。

通常のシャドウやデファードレンダリングというアプローチは今回実現したいビジュアルに足りていないと判断しました。

ダイナミック環境だからこそ、上記のアプローチは不適切でした。

Inspiration – Voxel Cone Tracing Talk@Siggraph 2011

Thankfully, the previous year there had been a really nice talk at Siggraph by Cyril Crassin about using Voxel Cone Tracing for Indirect Lighting.

Which seemed like it might be able to help us achieve this soft diffuse look.

幸いに、前の年のシーグラフでシロ クラッシンのボクセルコーントレースを使用した間接照明についての良い話がありました。

私たちが実現したいソフトライティングの見た目にこれは使えるだろうと思えた。

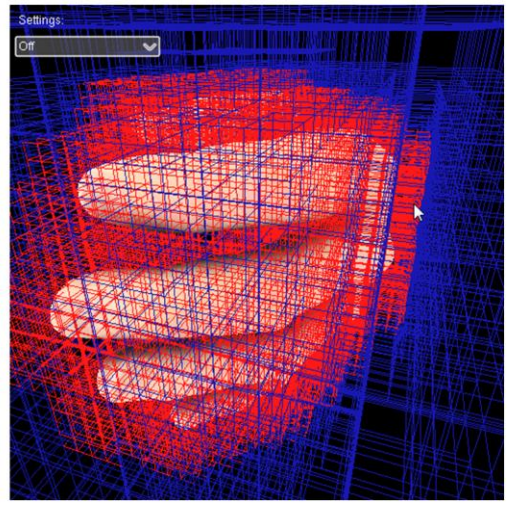
Inspiration – Unreal Elemental Demo

And later that year we got some positive reinforcement from Epics elemental demo that perhaps this was practical for next gen.

その後、Epicのエレメンタルデモからもボクセルコーンライティングが次世代機で実用化出来るという後押しを得られた。
結果的には入っていませんが。

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



So, how does Voxel Cone Tracing work?

I'll just run through it quickly for anyone who isn't familiar with the idea

In the original paper they start by voxelizing their scene into a Sparse Voxel Octree

ボクセルコーントレーシングはどんなものなのか？ボクセルのことを知らない人のために軽く説明をします。

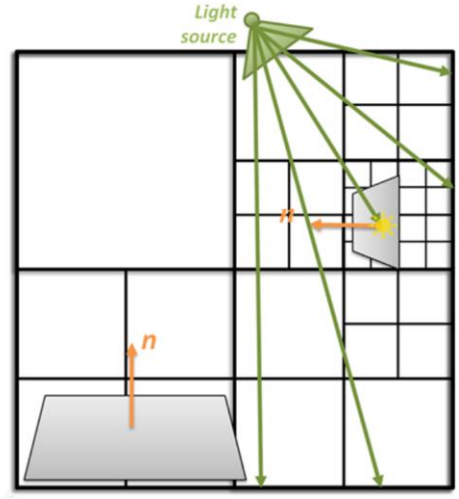
オリジナル論文に書かれている、彼らのスタートはボクセライズをしたいシーンをSVOに入れることです。

ボクセルコーントレースの説明　最初に使った方法だけど試したところ、オークツリーを使うのをやめた。

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



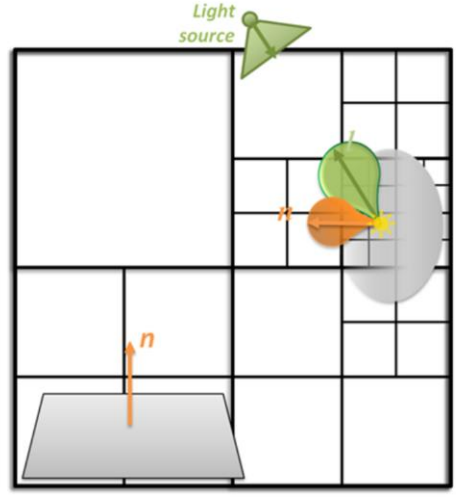
Then Injecting Irradiance values into the SVO.
Usually using something like Reflective Shadow Maps.

SVOへ光量値をインプットする。通常は反射シャドウマップのようなものを使用。

ライトからサーフェースにどのようなイルミネーションが必要かを計算する
描画するシーンを構成するすべての情報を階層化した8分木構造で扱う
Octreeは各要素を最大でも8分割して階層ごとに構成する

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

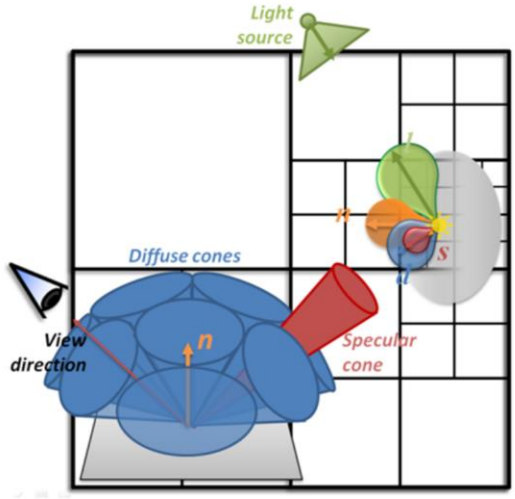
After injection they filter the irradiance up the octree, to get themselves mipmapped versions of the Irradiance data.

彼らのアプローチは、光量値をインプットした後に、オークツリーの光量のミップマップを作る。

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



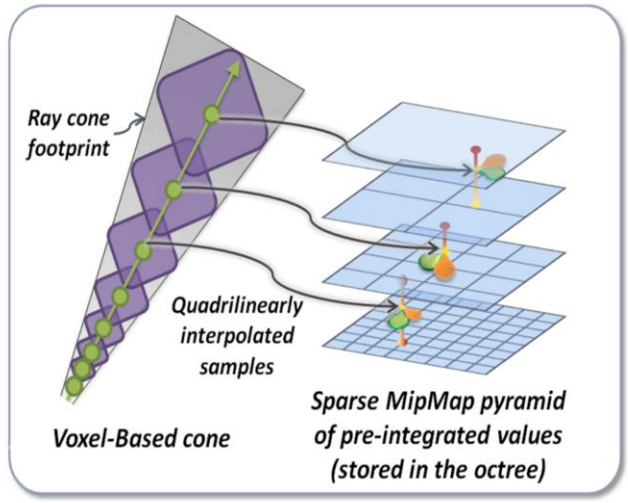
After which they trace a set of cones through the SVO per pixel.

自分から跳ね返った光が、回りのオブジェクトをどのように照らすのかを知るため、ピクセルごとにSVOを行う。

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



And take quadilinear samples at ever increasing levels up the SVO as they trace along the cone to accumulate the Irradiance for the pixel

Explain quadrilinear.

Voxel Cone Tracingによって下の階層から反射した光で照らされるオブジェクトを探していく。

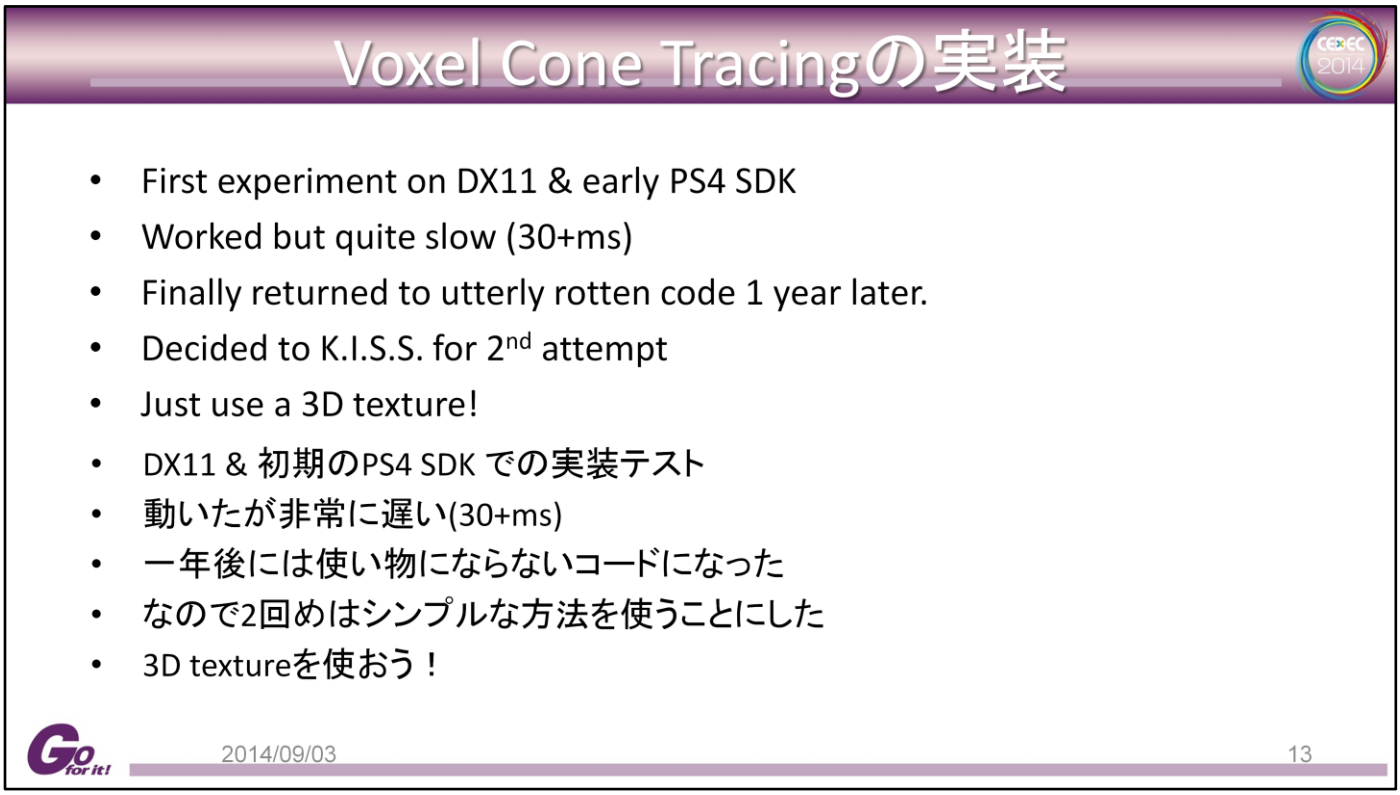
オブジェクトの探索ではConeの半径を徐々に大きくしていくが、それに応じて探すOctreeの階層も上げていく。

bilinearは 2D Textureのinterpolation

trilinear -> 3D Texture

quadriliniear -> 4D (interpolating between two trilinear interpolations)

3D texture A のtrilinear sample + 3D texture B のtrilinear sampleをinterpolationすることです。



So,

To cut a long story short.

I did some prototyping and implemented something similar to what the paper did in early 2011

Got something working, on an early PS4 SDK

But then got diverted to take care of a whole load of other engine work that needed to happen

And when I finally got a chunk of time to look at the lighting again a year later

It didn't run anymore, and I was scared about sinking time into reviving it, especially as it hadn't been that fast in the first place

I was always a little unsure how hiddeous walking the octree was going to be performance-wise on final hardware

So for my second attempt I decided to go with something simpler

And just get rid of the octree entirely and use a 3D texture instead.

No complicated traversal, simple!

私はPS4の早期のSDKの上でいくつかのSVO試作をして、

そして2011年にテストの結果が出た。そこからはほかの作業が入ってしまってSVOのテストを止めていたところ1年後にはそれは動かなくなっていた。

オークツリーはパフォーマンス的に望みが薄いことを感じて、別のシンプルな方法を選んだ。それが3Dテクスチャを使うことだった。

オークツリーのデータパスをせずに、一つの3Dエレメントとして簡略化した。

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

Obviously just a single 3D texture would never be large enough cover our scene, and fit in memory

So we extended this to use cascades.

We kept the anisotropic voxels from the origin paper, as this proved to be quite important for getting good results

And so we ended up pack all 6 faces and 6 cascades into a single 3D texture per attribute

We trace in 16 fixed directions , in a similar way to how Epic traced 9 directions for the Elemental demo

And this gets us 2 to 3 bounces of Indirect lighting

As well as our direct illumination

単一の3Dテクスチャで私たちのシーンに必要なサイズをカバーした場合、メモリに収まる大きさでないだろう。

次で説明するカスケードを使用するためにこれらを拡張した。

この異方性ボクセルは非常に大事なものと証明されています。それは良い結果が出るためです。

属性ごとに単一の3Dテクスチャに6面と6階層を詰め込んだ。

EPICがエレメンタルデモで9方向にトレースしたように、16の固定方向にトレースをする。

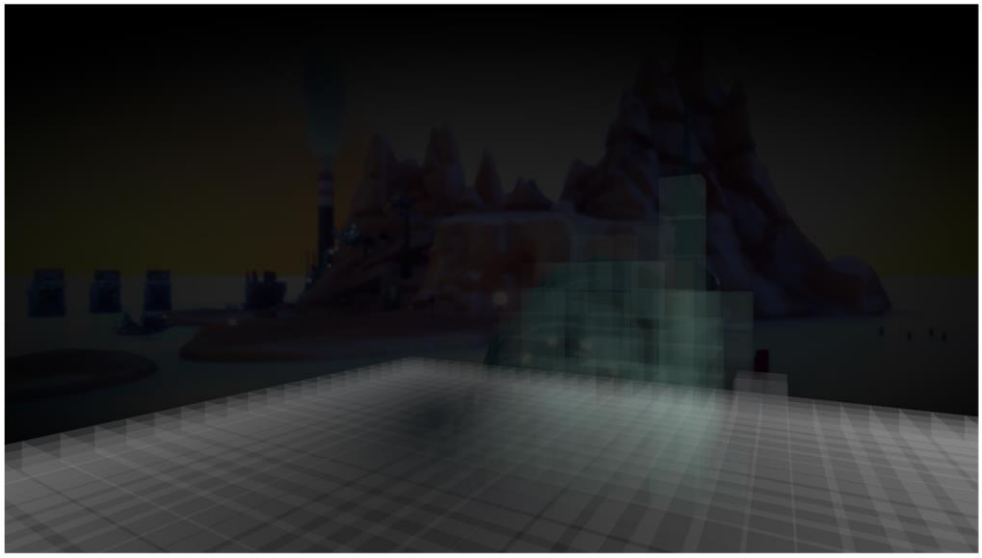
2から3バウンスの間接照明を得る、同様に直接照明を得る

Voxel Cascades

So, for anyone, who doesn’t know what I mean by cascades.
Here’s a quick look at a debug view on this test level.

ここでデバック表示を使ってカスケードをお見せします。

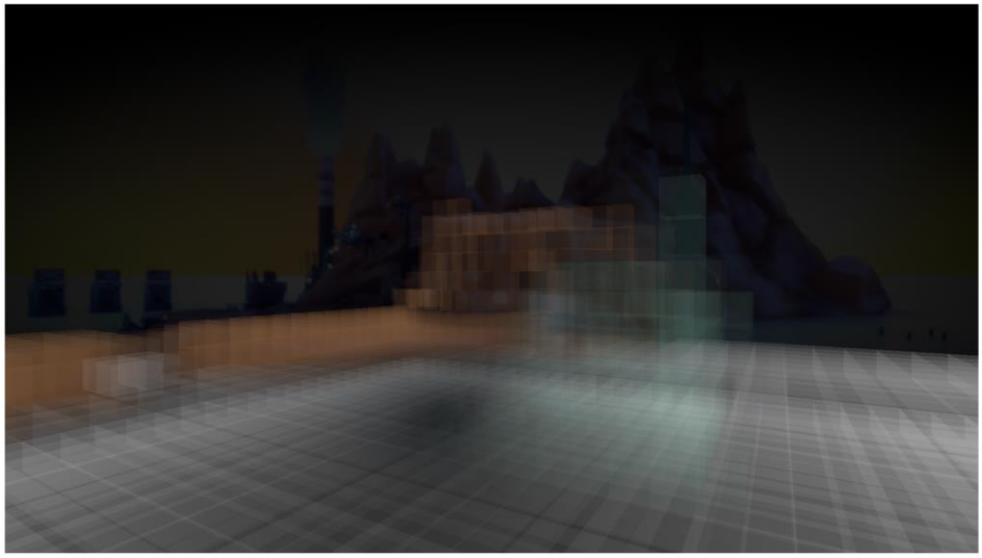
Voxel Cascades



Here’s what’s represented in the first 32x32x32 cascade level.

32x32x32レベルで表現さているものです。

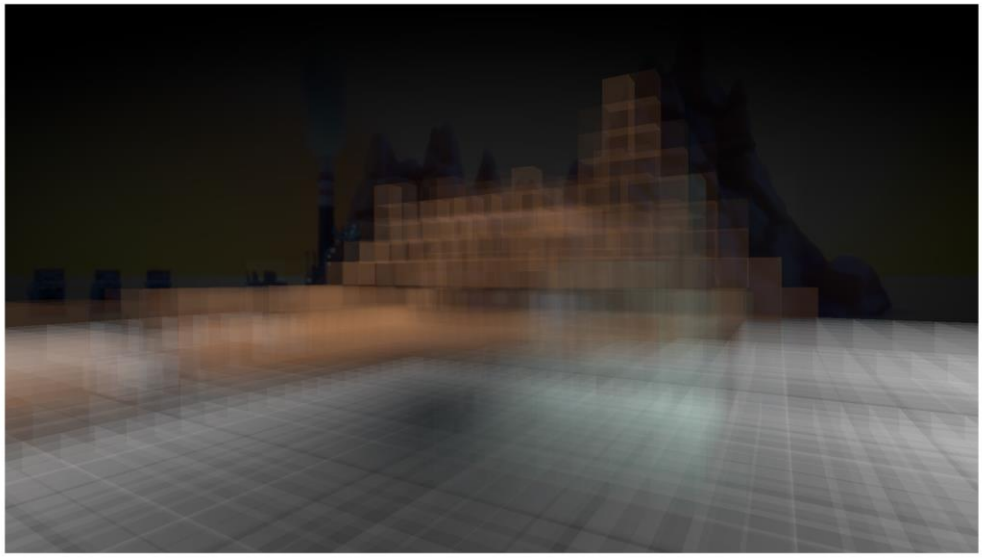
Voxel Cascades



And and as we add more and more cascades

更にカスケードを追加する

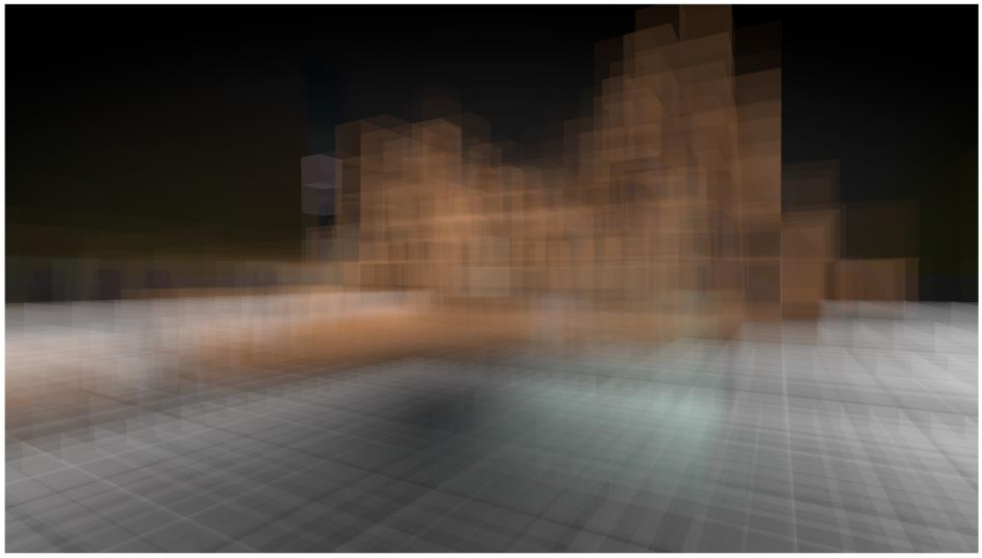
Voxel Cascades



And you can see as I add more cascades

更に増えているのが見えるでしょう

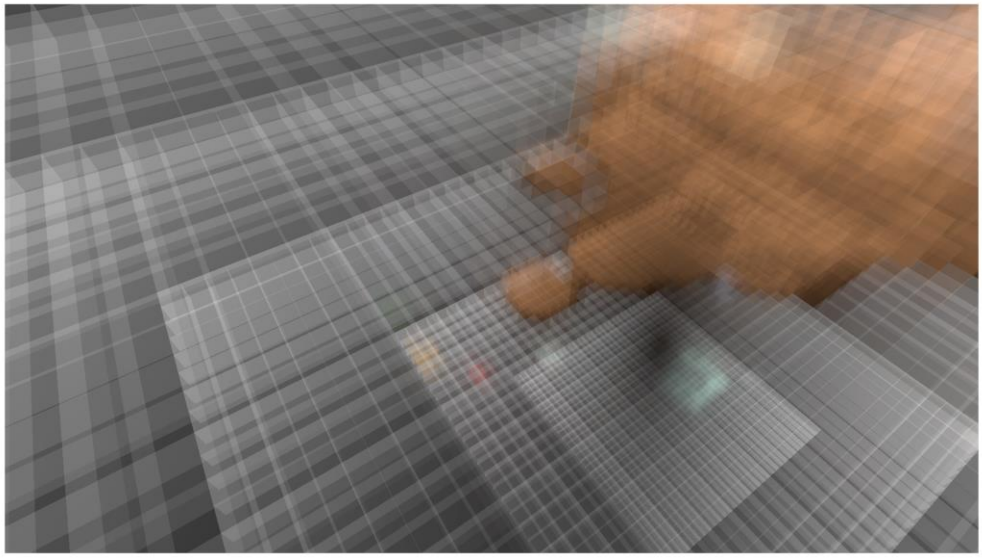
Voxel Cascades



You can see that we get something that sort of approximates the original scene

元のシーンがぼんやり見えてきました

Voxel Cascades



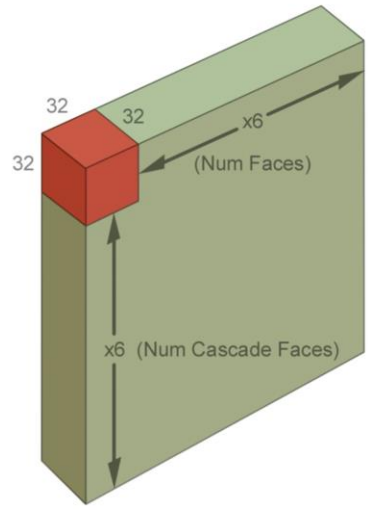
Looking from above we can see the concentric pattern of the cascades
that will be familiar to anyone who's implemented Clip Maps, or Light Propagation Volumes.

上から見るとカスケードが同心円状になっている。
これはクリップマップやライトプロパゲーションボリュームなどで知られています。

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



A quick word about how we store our textures.

We pack everything into a single 3D texture per attribute.

We need repeated blocks of 32^3 texels for each face, as our voxels are ansiotropic, and we tile these in X.

We also need to tile them in Y for our cascade levels.

This setup allows us to easily do trilinear interpolation between our voxels,

but we do have to be careful when we sample to always clamp to the edge of our cascades to avoid bleeding from other faces or cascades.

テクスチャへの格納方法

属性ごと（アルベド、法線、オクルージョン、エミッション等）に3Dテクスチャにすべてを詰め込む。

私たちのボクセルは異方性なので、各面のために32^3テクセルのブロックを繰り返すことが必要でした。

そしてこれらXにタイルする。また、カスケードレベルのためにこのブロックをYでタイルすることも必要となっています。

このやり方のおかげで私たちのボクセルの間にトライリニアの補完をもっと簡単に作ることができます。

しかし漏れを避けるためにカススケードの端をいつも固定するように注意しなければならない。

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

So our implementation is split into two main parts.

In the first we update our internal voxel data structures, voxelizing anything new that comes into view, and injecting and bouncing light.

In the second phase we trace cones in screen space to get local irradiance information which will then be used to drive our per pixel lighting.

主に2つの実装があります

・内部ボクセル構造を更新、変更があれば直接光と間接光を得る。

・ピクセルごとのライティングのために、ピクセルの周りの光量データをとり、スクリーンスペースでコーントレースを行う。

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

We made the decision early on to only update one cascade level per frame.

This works well because indirect lighting doesn't need to be updated at a particularly high frequency for users to find it convincing.

A simple way to implement this, would be to just cycle through our cascade levels,

But this means our nearest cascade would only be updated every 6 frames, which is a bit too slow.

So instead we have an incrementing counter, that we mask off, and count the number of trailing zeros on.

This now gets us a situation where our closest cascade is updated every 2 frames, and our next closest every 4, and so on.

Which perceptually is a much better balance.

1フレームあたり、1カスケードしか更新しないようにした。それは開発の早い段階で決めた。

ユーザーの感覚ではそれほど高頻度で間接光を更新する必要はない。これはうまく機能しています。

シンプルな実装方法はカスケードレベルでサイクルすること。しかし最寄りのカスケードをアップデートするのに6フレーム必要で、少し重い。

プレイヤーの視点から一番近い風景が一番大事なので、それを優先して、2フレームごとに最も近いものをアップデートする。

カスケード1は2フレーム、カスケード2は4フレーム、カスケード3は8フレームといった形で。これらは見た感じに良いバランスになっています。

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

The first thing we need to do is calculate the new center of our cascade if the viewer has moved.

It’s important to note that in order to make mip-mapping easy later on, we have to lock this to a grid that is half the resolution of our cascade.

We must then scroll any data we have in our cascade level if we have moved,

And voxelize any new geometry at the edges of our cascade, or geometry for objects that have just appeared or been changed.

At the end of our voxelization process we scan for voxels that are on the surface of our geometry, and write these our to an RW_Buffer.

These are the voxels that we will trace from to update the direct and bounce lighting for our cascade.

視点が移動した場合、まずカスケードの新しい中心を計算します。

大事なポイントは、あとでミップマップをもっと簡単にするように、グリッドにミップマップを固定しないとならない。

そのグリッドはカスケードの解像度の半分です。

そして動いた場合、カスケードにあるすべてのデータをスクロールさせないといけない。

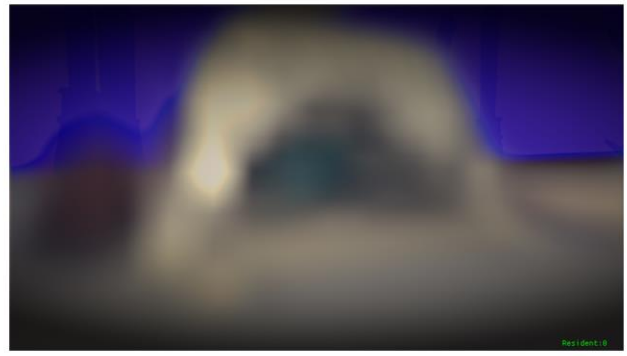
また、変わったばかりか現れたばかりのオブジェクトためのジオメトリと、カスケードの端にあるすべてのジオメトリをボクセライズしなければならない。

ボクセル化プロセスの最後に、2つのことをします。まずボクセルにあるジオメトリのサーフェースをスキャンしてRW_Bufferにいれます。

カスケードのために、そのボクセルについてはこのボクセルをトレースしたら直接光と間接光を次のステップのためにアップデートする。

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



Because voxelization can be slow, we only voxelize static objects.

Characters and dynamic objects are dealt with separately, which I will describe later.

Our voxels, even at the finest level tend to be quite large, and so some degree of antialiasing when voxelizing is very important.

For this purpose we voxelize into 128x128x32 textures for each axis, before applying a final resolve, to give us 16x AA.

Our landscape is stored as a Layered Depth Cube, which is essentially a set of span lists in each axis, and is thus very easy to voxelize

Objects however have to be voxelized via the hardware rasterizer, with a custom pixel shader that we use to export a list of voxels, on each axis.

These are then sorted in a compute shader to ensure correct depth ordering.

This process is relatively expensive, and can be a bottleneck when a large amount of objects suddenly enter our cascade.

In order to amortize some of this cost, we voxelize objects into 8x8x8 blocks, that are stored in a simple cache, that covers an area slightly larger than our cascade.level

In this way we can voxelize objects that will be needed in the near future over a number of frames, without causing any undue frame spikes.

ボクセル化に時間がかかるため、スタティックなオブジェクトのみをボクセライズす

る。キャラクターや動的なオブジェクトは別扱い。

すぐれたレベルのボクセルでもすべてのボクセルは、かなり大きい傾向があって、ボクセライジングするときに、ある程度アンチエリアエリアシングすることが重要です。

各軸の128x128x32テクスチャにvoxelizeする。最終結果を適用する前に、16倍のAAを与える。

私たちの地形はレイヤードデプスキューブに作られています（カスケードとは別）。レイヤードデプスキューブには各アクシスで色々なスパンリスト（ピクセルのイン、アウト情報）があります。

そのおかげでボクセライズをするのが簡単です。

しかしハードウェアのラスタライザを通じてオブジェクトはボクセライズされます。そうするために各アクシスにボクセルのリストを書き出すのにカスタムピクセルシェーダを使っていた。

正しい深さの順序を保証するために、コンピュートシェーダでソートします。

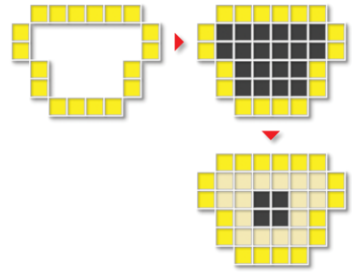
このプロセスはかなり処理が重い。オブジェクトが大量にカスケードに追加された場合ボトルネックになる。

このコストの問題を軽くするために、オブジェクトのボクセルキャッシュに入っているオブジェクトを、8x8x8にボクセライズします。（私たちのカスケードより少しだけ大きいエリアを覆う）

数フレームにプレイヤーの動きによって必要だと予想される領域は、オブジェクトキャッシュを利用して読み込みます。そうするとフレームスパイクを起こさないようにできる。

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

One thing that we found important for robustness was to ensure that the voxelization was solid.

So as a post step after voxelization we fill spans between voxels with opaque black voxels.

We also propagate our surface attributes inward to the first subsurface layer of voxels to ensure we can't trace past important information.

2つのステップがあります。中のボクセルをスペースなしで黒く塗りつぶす。

次は表面の近くのサンプリングが黒いボクセルの影響を受けないように、表面を厚くする。

ボクセル化後の後工程として、私たちは不透明なバックボクセルとボクセル間のスパンを埋める。

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

So, now we have the updated attribute data for our scene.

We need to inject lighting information and propagate bounce light within out voxel data structure.

今私たちはシーンの為に、更新された属性データを持っています。

私たちは、ライティング情報を注入し、アウトボクセルデータ構造内にバウンス光を伝搬する必要があります。

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

Because of the look we were going for, we decided not use shadow maps, and to use cone tracing for our direct lighting as well.

As our tracing directions are fixed, this means that for our surface voxels, the cones we will trace for direct lighting , are the exact same cones we need to trace for bounce lighting.

So we can fold everything into one cone trace for each of our 16 directions, and just feed it with multiple input textures and get multiple results back.

We do have to be careful about how we organize this though, as we want a pipeline of data flowing through the system, we don't want feedback.

私たちは、シャドウマップを使用しないことを決定した。直接照明にコーントレースを使用してシャドウを作る。

私たちのトレースする方向は固定されています。なので直接照明にトレースするコーンは、関節照明にトレースするのとまったく同じコーンである。

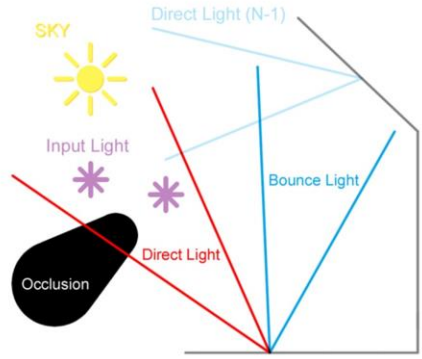
直接照明、間接照明、2回目のバウンスは一つのコーントレースでカバーできます。マルチプルのテクスチャを入力すれば、マルチプルの結果を得られます。

私たちはデーターパイプラインが欲しいのです。フィードバックは欲しくない。

（いくつかのバウンスがある。そのバウンスは気をつけなくてはいけない時がある。）

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



So, our cone trace, takes as input an occlusion cascade telling us which voxels are occupied.

An input light cascade, which contains information about radiance from any point lights we have in our scene.

A Direct Light cascade, which is the direct lighting we accumulated last time this cascade was updated.

And a Bounce Light cascade, which is the first bounce indirect light that we accumulated on our last update.

Note, there is no direct light added into the bounce light cascade otherwise we will get feedback.

オクルージョンデータがインプットデータです。そのインプットデータによって、どのボクセルが入っているかが判定できます。

ライトカスケードのインプット-　任意のポイントライトから光量に関する情報を読み込みます。

ダイレクトライトカスケードが必要です。ダイレクトライトカスケードは前回のカスケードが更新されたときのダイレクトライティングのデータです。

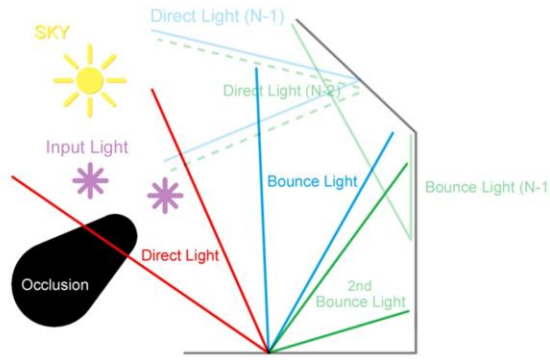
バウンスライトカスケードも必要です。バウンスライトカスケードは前回のカスケードが更新されたときの第一バウンスライティングのデータです。

バウンスライトカスケードに、ダイレクトライティングのデータはなし。（あるならフィー

ドバックになってしまうため。）

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



Once we have the results of our cone traces in the 16 directions from each voxel, we can light each of its faces with the light we have gathered.

And spit out a new direct light and bounce light cascade,

as well as what I like to call the Bounce Bounce Light cascade,

which is the Final result texture containing our direct lighting, 2 bounce of indirect, and a tiny bit of extra special magic, to give it an extra kick.

あらためて、私たちはそれぞれのボクセルに対して16方向のコーントレースの結果を持っています。私たちはそれぞれの面にたいして拡散のライティングができる新たなダイレクトライトと間接光を吐き出す。

2回のバウンスの反射光と小さなマジックを使って、追加の効果を得た。

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);
```


Two bounces of light at our voxel granularity are nice, but we would like to have even more.

So, we try to fake just a little bit more by looking for places that received more second bounce light than first bounce light,

and surmising that they would probably get more illumination from a third bounce.

We add this extrapolated extra bounce to our final bounce bounce result.

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

最初の反射光より、2回目の反射を受けた場所を探して、その場所はおそらく3回目の反射から多くの照明を得るのでは無いだろうか？

結果を少し偽装する。

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

Finally we propagate the irradiance up the cascades with a compute shader.

We need to do this for all 3 irradiance textures that we have.

It's also worth noting that we also have to scroll these textures as our cascades move around.

With data that we don't have at the edge of a cascade pulled from the next cascade up, to give us a reasonable starting point.

最後に、コンピュートシェーダーでカスケードを光量をアップして、このプロセスでミップマップのレベルを作っている。

ダイレクト、バウンス、2度目のバウンスという3つのテクスチャ、それぞれにこのプロセスを適用する。

プレイヤーの視点がが移動すると、テクスチャをスクロールさせる必要がある。

カスケードのエッジデータがないときは、一つ上のカスケードデータを読み込みます。

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

Now that we have updated the data that we have in our voxel cascades, we can now go about starting to get this data into screen space.

Because we are dealing with very soft lighting, we can afford to cone trace at a much lower resolution than 1080p

So, we use a ¼ dimension buffer

and intelligently upscale it, fixing up any pixels that really need it as we go.

Again this is traced in the same 16 directions that we used to trace from the voxels,

And we output to a 16 layer deep, screen space texture array.

ボクセルカスケードの中にあるデータの準備ができた。

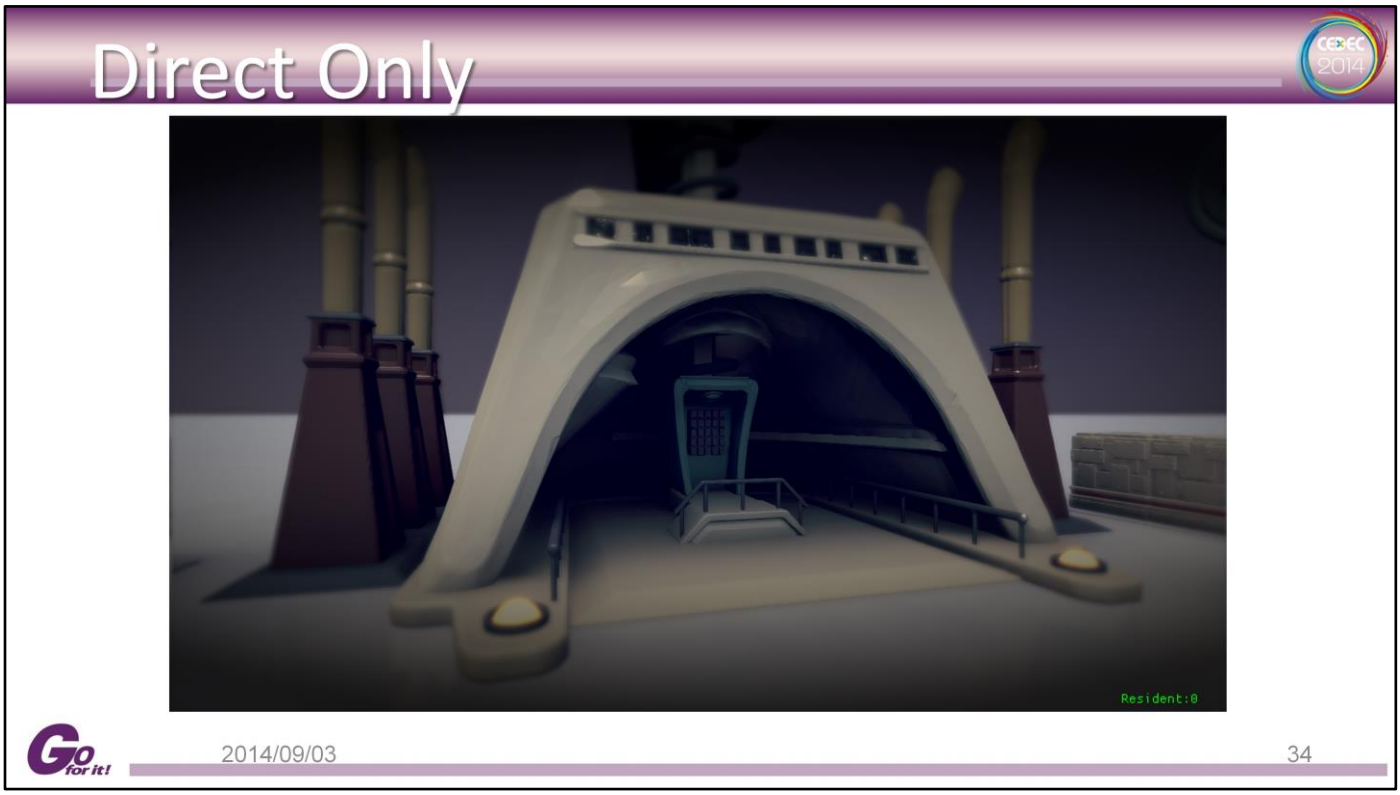
次のステップはスクリーン空間に、このデータを使うことです。

今回、非常に柔らかな照明を扱っている。コーントレースのレベルは 1080pよりもはるかに低い解像度をつかえます。

そこで、4分の1バッファを使用する。

本当に必要なあらゆるピクセルを固定して、賢くアップスケールする。

スクリーンスペースのテクスチャーは16層を持っていて、それにアウトプットする。



So you can begin to see what this looks like.

Here is our scene with just direct cone traced illumination.

BTW, a this point it's probably worth noting that we project our sky into 16 SRBFs, one for each direction we trace.

As you can see, this all gives us a very soft look to our shadows.

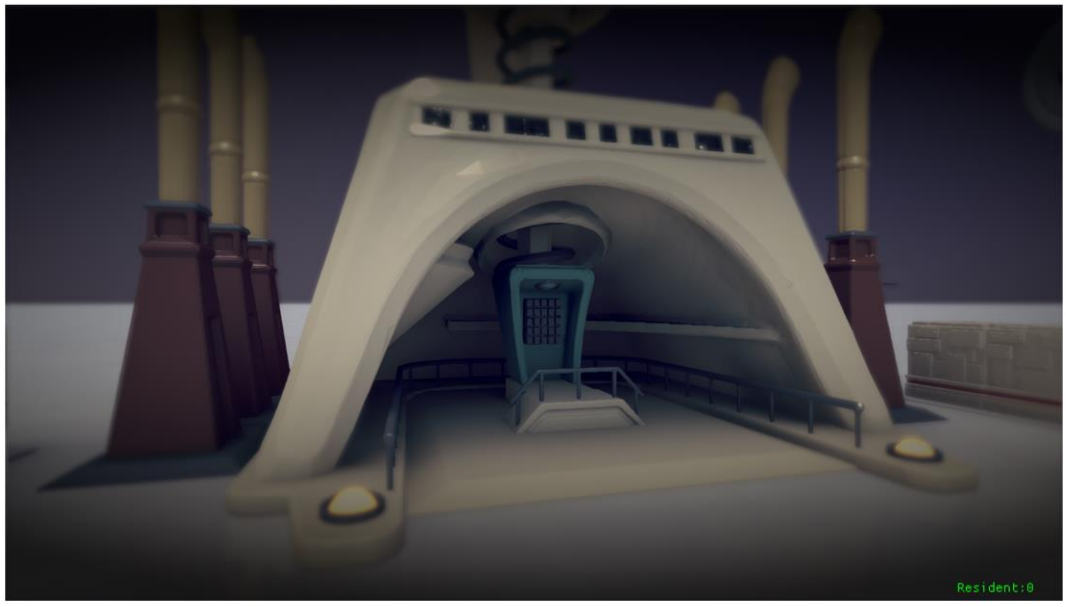
実際にお見せします。

これは直接のコーントレース照明のみの結果です。

私たちがトレースする方向それぞれに、16 SRBFs（球面放射基底関数）に天球を投影する 。

これは私たちの影に非常に優しい表情を与えます。

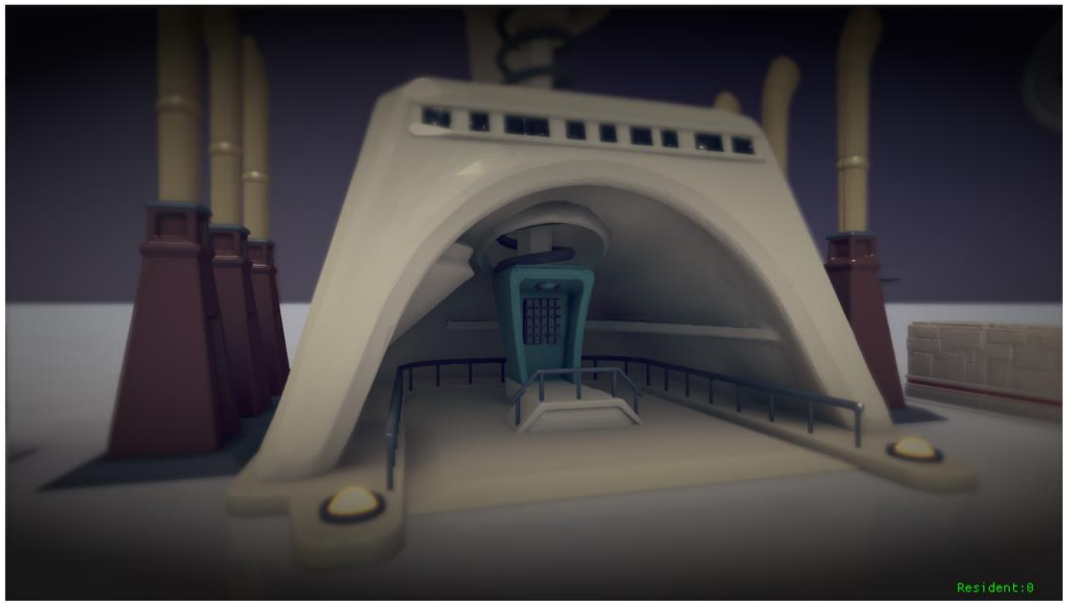
One Bounce



And you can see what starts happening as we add in the indirect lighting.
Here we have just one bouce.

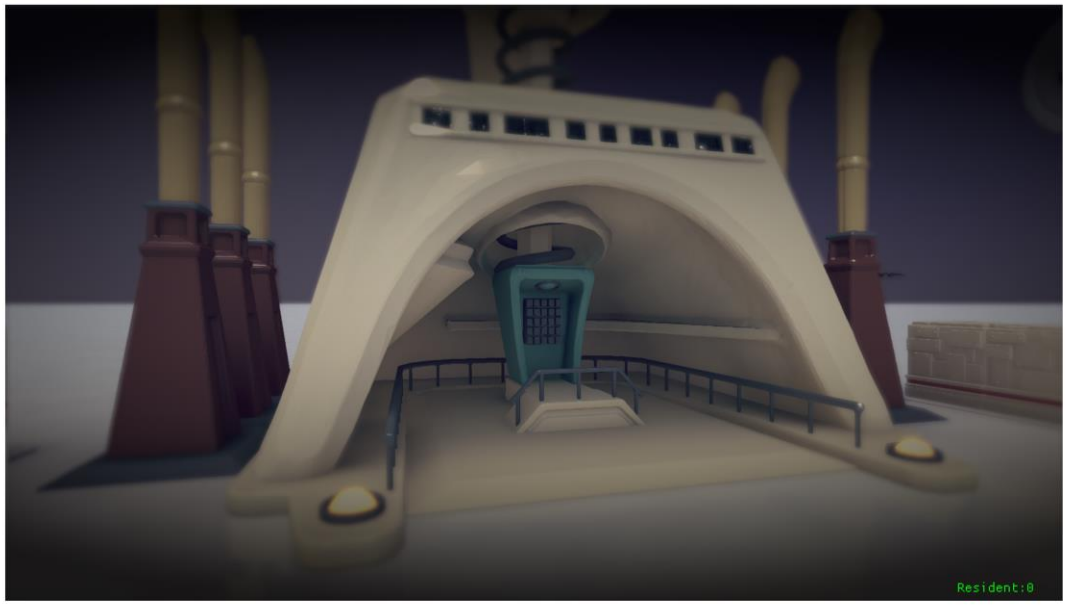
間接照明の効果を見ることができます。
これはまだ1つの反射です。

Two Bounces



And now two.

Three Bounces



And three.

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

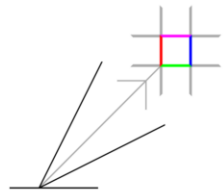
So all of this works, but is still a little too slow.
So we have had to cut a few more corner to get to a workable speed.

これらのすべては動いていますが、まだ少し遅い。
そこで、現実的な速度を満たすために、更にいくつかの角を落とす必要があった。

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



The first thing we do is note that because our voxels are anisotropic,

as we trace, we are constantly having to do an interpolation between the values of 3 face voxels.

This is quite costly in terms of the number of texture lookups.

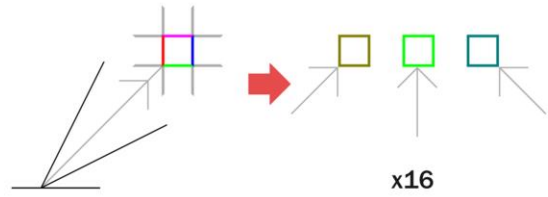
私たちのボクセルは異方性なのでトレースしながら、3面のボクセル値を常に直線補完している。

これは、テクスチャルックアップの数の点で非常にコスト高である。

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



But because we have fixed the directions we will trace, we can actually just pre-combine these values for each of our 16 directions.

And store the whole thing in another texture.

There is a slight overhead for doing this, but it is relatively cheap, and it reduces the number of texture lookups we need for the

Cone tracing steps by a factor of 3.

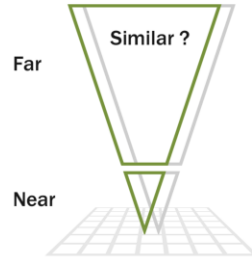
トレースの方向は固定されている。実際には16方向のそれぞれについて、これらの値を事前に組み合わせることができます。

そして別のテクスチャにすべて保管する。

これを行うには若干のオーバーヘッドがあります。しかし比較的安価で、私たちはコーンが3倍にステップをトレースするために必要なテクスチャルックアップの数を減らすことができます。

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



The second big optimization we make is simply to take advantage of parallax.

If we trace two cones in the same direction from a similar point in space,

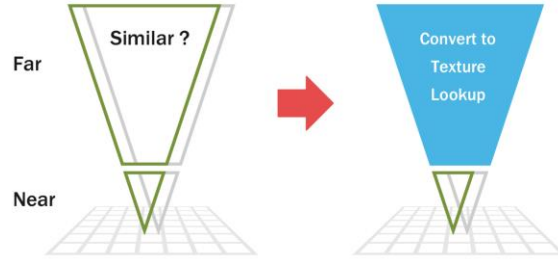
the voxel data we access becomes increasingly similar as we move towards the far end of the cones.

2番目の大きなオプティマイズはシンプルに視差を利用することです。

2つのコーンは近いところから、同じ方向にトレースします。コーンの遠い方に近づくと、アクセスしているボクセルデータはどんどん同じようになってきます。

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



So instead of tracing this data repeatedly, we trace this “far” cone data once, from the center of each voxel in a cascade,

And store this in another texture, which we can then trilinearly interpolate from in the future, to reconstruct this data.

Then we only need to trace the “near” part of the cone, and use our texture for the “far” part.

Where the “far” cone trace starts can be tuned for the best balance between quality and speed.

だから、毎回このデータをトレースするより、カスケードの中のすべてのボクセルの真ん中から、遠いデータを一回だけトレースする。

そしてこのデータが、もう一つのテクスチャに保存される。テクスチャで保管したデータをいつか再現するためにトライリニアインタポレートが必要です。

コーンの「近く」の部分だけをトレースする必要がある。“遠い”部分のためにはテクスチャを使用する必要があります。

遠いトレースのセンターは動かせる。クオリティーが必要なら遠くに、スピードが必要なら近くに変更できる。

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

Cone tracing is great, but our implementations smallest voxel size is only 0.4m, so we need to augment this with extra detail computed in screen space.

コーントレースは素晴らしいのですが、私たちの実装では、最小のボクセルサイズは0.4メートルです。
スクリーンスペースを使い、追加のディテールでこれを補強する必要があります。

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)

In order to do that need some screen space occlusion.

Note, that I left out the word ambient there,

We do something similar to Screen-Space Bent cones from GPU Gems 3

And integrate 2 band SH rather than a scalar occlusion value.

Which gets us a visibility cone at each pixel, that we can then intersect with the cones from our 16 trace directions

Keeping this occlusion directional rather than just a single scalar value really helps, especially when we deal with specular.

私たちはいくつかのスクリーンスペースオクルージョンが必要でしょう。アンビエントオクルージョンとは考えていない。

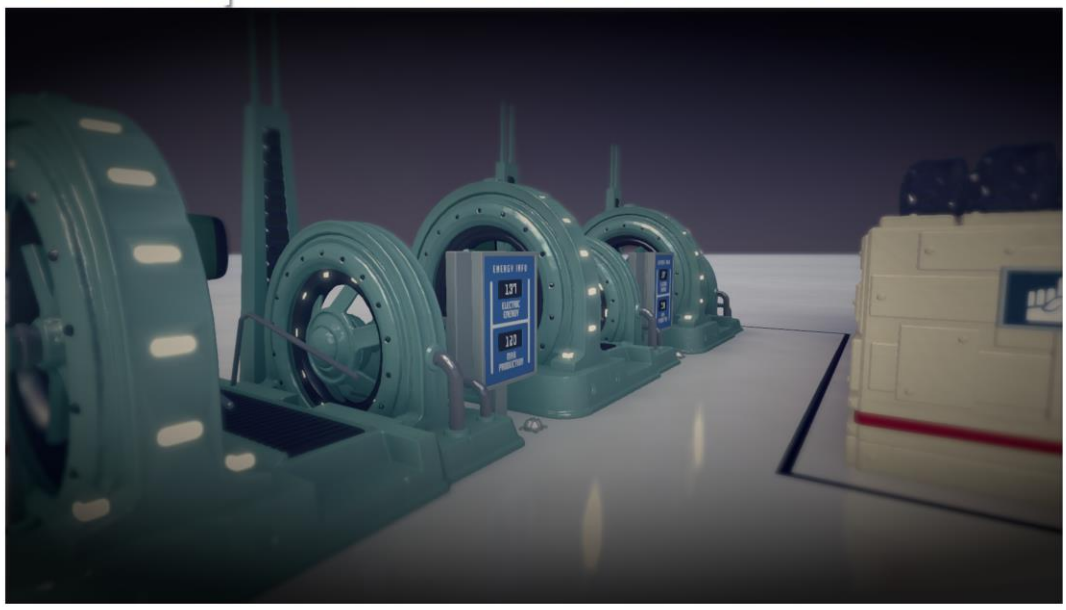
私たちは、GPU Gems 3からスクリーンスペースベントコーンと同様なことをする。そしてスケーラーオクルージョンバリューではなく、2バンドSHを実装した

各ピクセルにビジブルなコーンがある。それで16のトレース方向のコーンにオーバーラップする。

この閉塞はちょうど単一のスカラー値ではなく、用方向保つことは本当に私たちは、鏡面を扱う場合は特に、役立ちます。

今回のオクルージョンは、一つのスケーラーの値より方向性があったほうが有利である。特にスペキュラーで有利です。

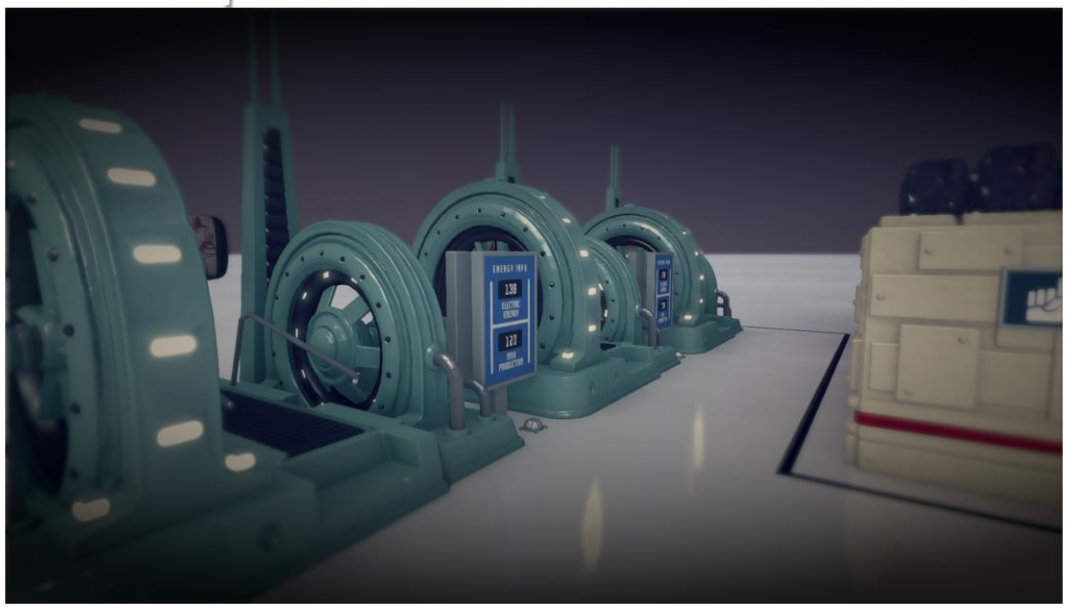
Screen Space Occlusion - Off



So, here is an example scene without screen space occlusion

スクリーンスペースオクルージョン　オフ

Screen Space Occlusion - On



And now with, you can see what a big difference that makes.

スクリーンスペースオクルージョンの効果を見ることが出来ます。

キャラクターのシャドウ

As I said earlier Characters are not voxelized due to the size of our voxels,

And the extra overhead it would cause

But we still want nice soft shadows from them

最初にふれたように、キャラクターはサイズの問題でボクセル化されていません。

それは余分なオーバーヘッドになる。しかし彼女にいい感じのソフトシャドウを付けたい。

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

So we use the characters collision volumes, to generate occlusion in a very similar way to what was done in The Last of Us.

Except that we have to do a cone overlap test in each of our 16 cone tracing directions, rather than just a single primary direction.

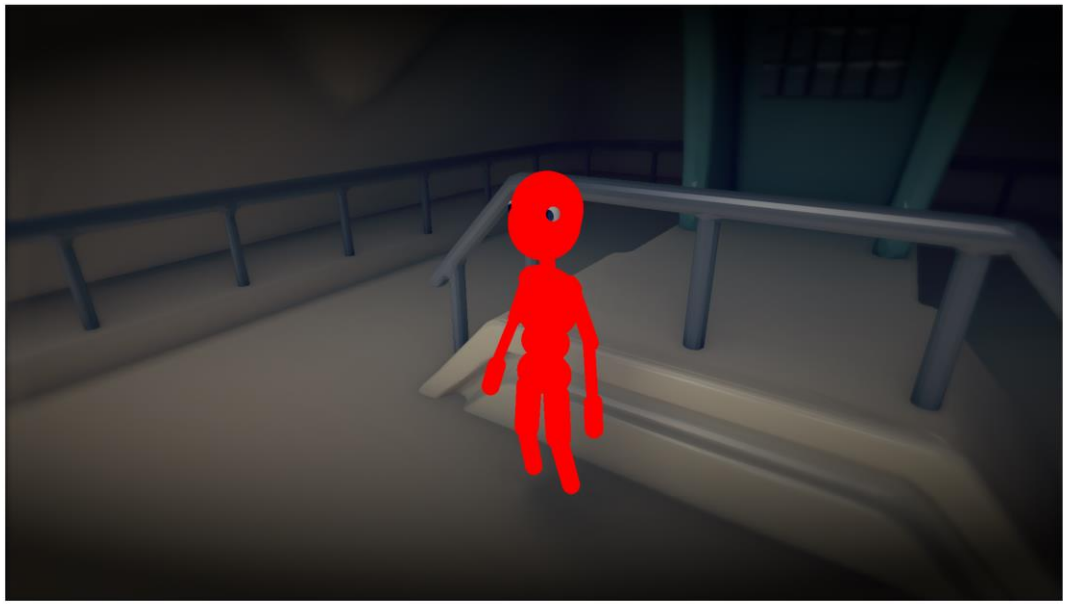
The result of this is a another 16 deep screen space texture array.

私たちは“ラストオブアス”で使われた方法に近いオクルージョンを行うために、キャラクターコリジョンボリュームを使いました。

これは基本の1方向だけでなく、私たちの16方向のコーントレースダイレクションに対してコーンオーバーラップテストをしなければいけなかった、

この結果は別の16のスクリーンスペーステクスチャ配列になる。

キャラクターのシャドウ



Here you can see the volumes we are using for the main character.

キャラクターのボリュームが見えます。

Characters - On

As you can see, this helps the character feel much more rooted in the world.

御覧のようにキャラクターがよりリアルに存在するように見えます。

Characters -Off

I'll just flick between those two so you can see it more easily.

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

So that takes care of characters, but capsules are not a very good fit for some of the other dynamic things we have in the game, namely Vehicles.

For these we instead build a 3D texture containing 2 bands of SH coefficients that describe a visibility fn which we can then use to intersect with the cones from our 16 directions.

Again, this outputs to the same 16 deep texture array as the characters.

乗り物等の形状にはカプセルを使ったオクルージョンは向いていない

代わりにビジビリティ関数を記述したSH係数の2バンドを含む3Dテクスチャを作成します。それは私達の16方向からのコーンとオーバーラップするように使用することができます

これは、キャラクターと同一の16ディープテクスチャ配列に出力します

Vehicles - Off

So here you can see our bus, without an occlusions.

Vehicles - On

And now with.

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。

So, we also have to deal with lighting transparent objects such as particles.

But the potential cost of doing 16 cone traces per pixel per particle, are just too expensive to really be viable.

What we do instead (and this might be getting familiar to you now) is build another texture!

While we are tracing our “far” cones, and storing that per voxel, we also actually trace the “near” cone per voxel too,

And store the composited cone in another texture.

This gives us a texture where for any point in space we can query and get a rough approximation of the incoming light in all 16 directions.

But 16 texture lookups is still way too expensive, so we then encode this texture as 2 band SH, which gets us down to just 3 texture lookups.

We then tessellate our particles, and sample from this texture per vertex.

私達はパーティクルのような透明な物体もライティングに対応させる必要があった。

しかし、パーティクルのピクセルあたり16コーンのトレースを行うためのコストは高すぎる。

私達が行ったのは別のテクスチャを使うことです。（これはみなさんが使っている方法と近いかもしれません）

私たちは「遠い」コーンをトレースし、ボクセルごとにそれをトレースする間に、よりボ

クセル毎に「近い」コーンをトレースします。

遠いトレースと近いトレースはもう一つのテクスチャに保管されます。

世界の任意の点で、私達は16方向の入射光全てに大まかな近似を得ることが出来ます。これは私達にテクスチャを与えます。

しかし、16のテクスチャルックアップは、まだあまりにも高価です。3つのテクスチャルックアップを得て、2バンドSHとしてこのテクスチャをエンコードする。

次に、頂点ごとに、テクスチャから私たちの粒子、およびサンプルをテッセレーションする。

Particles – No Occlusion

We also have the particles fill a dynamic occlusion texture, which is fed back into the cone tracing,

コーントレースにフィードバックされたパーティクルはダイナミックオクルージョンテクスチャを塗りつぶす。

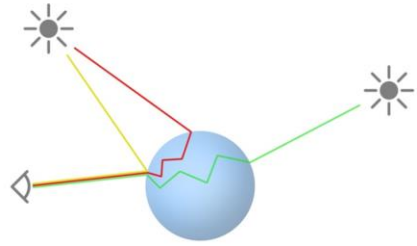
Particles with Dynamic Occlusion

And you can see how this allows things like smoke to have a volumetric feel
And a sense of presence.

ボリューメトリック感をもった煙になりました。存在感が出ています。

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



The nice thing about having this SH texture is that we can use it for other things.

Typically Subsurface scattering is a difficult thing to simulate in realtime.

We need to simulate light entering an object at multiple different points, and bouncing around inside the material before exiting at the point seen by the viewer.

One possible approach that has been tried is to blur the lighting information in either texture space, or in a geometry aware way in screen space.

This works to some degree, but doesn't help us that much with lights that are behind the object.

So we have some chance of getting some good results for the red ray above, but not the green one.

このSHテクスチャーの良い所は他の用途にも使えることです。

通常、サブサーフェーススキャッタリングをリアルタイムにシミュレートすることは困難なことです。

光がいろいろなポイントでオブジェクトに作られたマテリアルに反射されて、視点から出てくるということをシミュレートしなければならない。

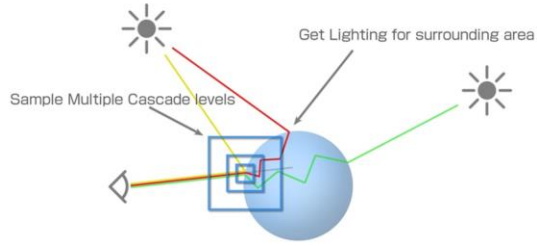
一つの方法はライティングデータを動かす方法があります。テクスチャースペースでぼかす、あるいはスクリーンスペースで形状から出ないようなぼかしのアプローチもできる。

光がオブジェクトの後ろじゃなければこのアプローチはOKです。

図の赤い光はOK、緑の光だと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



Thankfully the SH texture we built for particles gives us another way to tackle this problem.

The texture we have build is effectively a light field for the scene that we can sample at any point in space,

And can quickly give us the irradiance at any point we would like to sample.

Going up the cascade levels also gives us the information about the incident lighting over a wider and wider region of space.

So we can simply take a weighted average of the irradiance from different cascade levels, and use that to gather light that doesn't directly hit our viewpoint.

In our implementation we sample over the whole sphere, effectively just taking the 1st SH band, to accumulate this lighting, which we call the "static SS" shading.

幸いなことに、パーティクルで使ったSHテクスチャーは私たちに、SHに対処する別の方法を提供します。

構築している物は効果的な空間内のライトフィールド。空間内の任意の場所でサンプリングすることが出来る。

ライトフィールドで任意のポイントでのサンプリングができて、ポイントの光量のデータが計算できます。

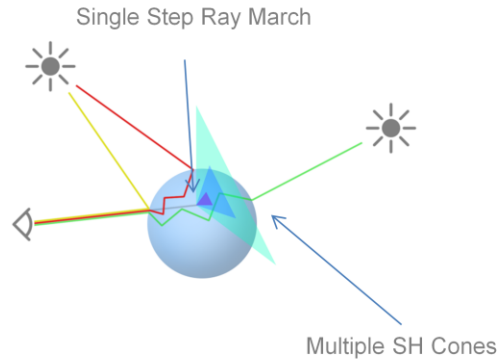
そしてカスケードレベルの上に行くほど、光量のデータより、広い地域のデータが取れます。ということで、カスケードレベルごとから光量の平均価値を取って、プレイヤ

一視点に直接この光のデータを作ることができます。

このアプローチで第一SHバンドのみ、を利用して名づけて“スタティックSSシェーディング”であります。

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



We can also give a directional effect by ray marching through the object away from the viewer, sampling from different cascade levels as we go, gathering light using a projected SH cone in the direction of the ray march.

But in order to take as few samples as possible we only take one raymarch step, and take multiple samples from our cascade at this position,

Using progressively wider SH cones as we ascend our cascade levels.

This we call our “directional SS” term.

To get a rich range of material looks, we interpolate between the normal diffuse lighting we get from our cone tracing, and these two sub surface lighting results.

In order to provide even more of a sub surface look, we also add a parameter for “frosting”, which uses a screen space local thickness parameter we calculate along with our SSDO to modulate the albedo of the material.

もう一つのアプローチでディレクショナルエフェクトを実現できます。それはレイマーチをプレイヤーの目線よりオブジェクトのポジションから遠くなる位置から増分的にサンプリングします。

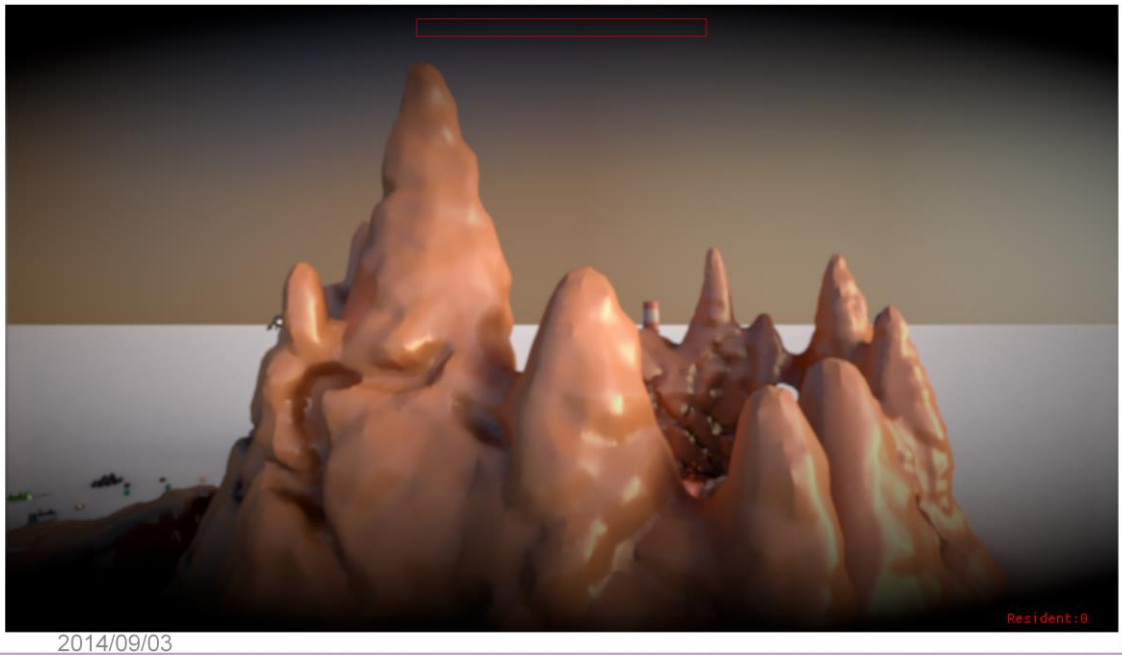
レイマーチの同じ方向のSHコーンから光のデータを取っています。ただし、サンプルを少なくするため、実質的に一つのレイマーチステップのみをし、そのポジションからいくつのサンプルを取ります。カスケードレベルを上がりながらSHコーンはどんどん広くなっています。

名づけてダイレクションSSというアプローチ。

いろんなマテリアルの見た目を実現するため、コーントレースで取る通常ディフューズライティングとダイレクションのSSやスタティックSSを補完します。

よりリアルなサブサーフェースの見た目を実現するため、フロスティングというパラメータを追加します。SSDOを計算しながら、フロスティングがスクリーンスペースのローカル厚さパラメータを利用して、マテリアルのアルベド（太陽の光を地球が反射する割合）を変更します。

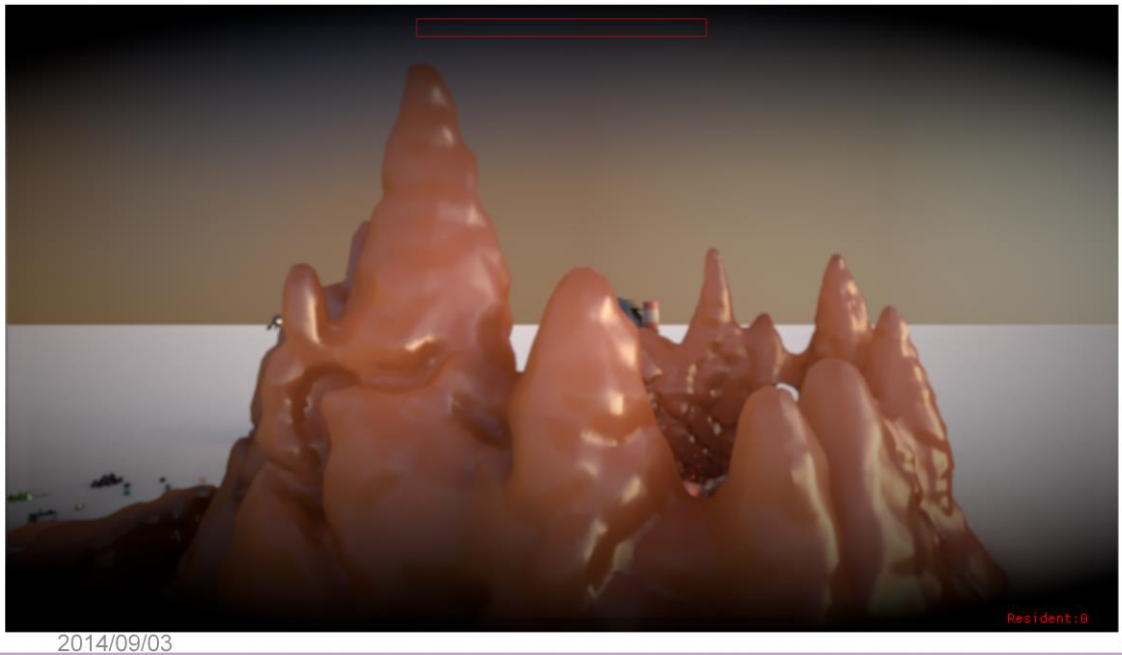
Subsurface - Off



So here you can see some slightly glossy mountains that are light with our vanilla cone trace lighting.

プレーンなコーントレース照明の光と少し光沢のある山を見ることができます。

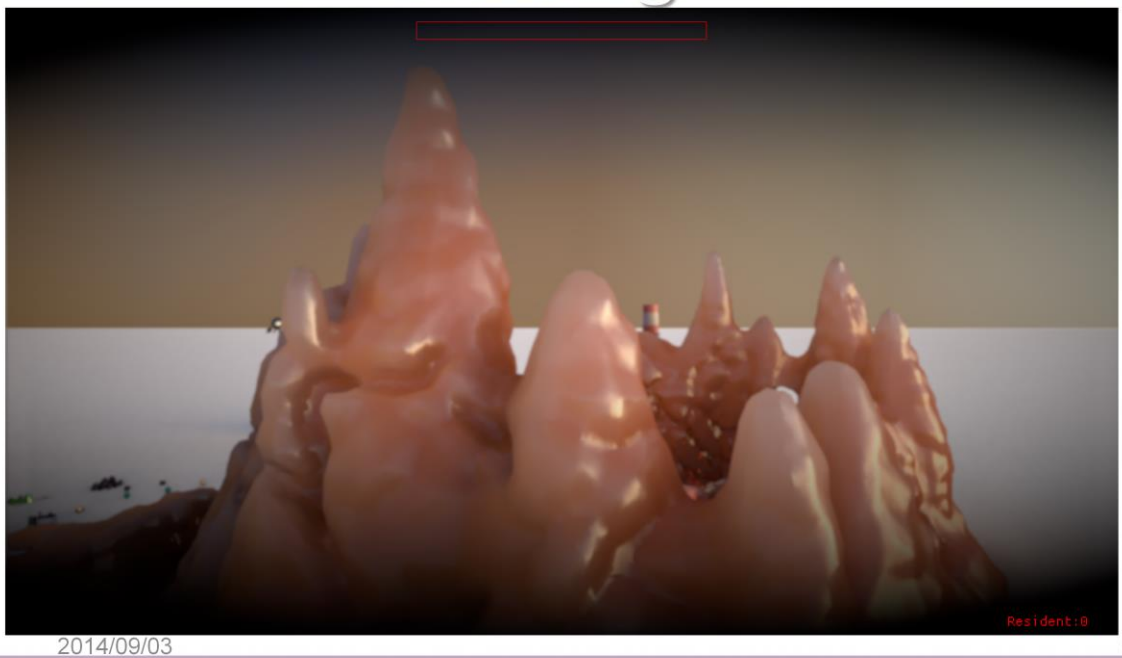
Subsurface - On



And you can see how we can turn on sub surface scattering and give these mountains a nice waxy, semi translucent appearance.

サブサーフェーススキャッタリングをオンにし、これらの山の素敵なワックス状の半透明の外観を与えることができるかを見ることができます。

Frosting



And then we can add the frosting effect to acccentuate the thin regions, and we start to have something that looks quite believable.

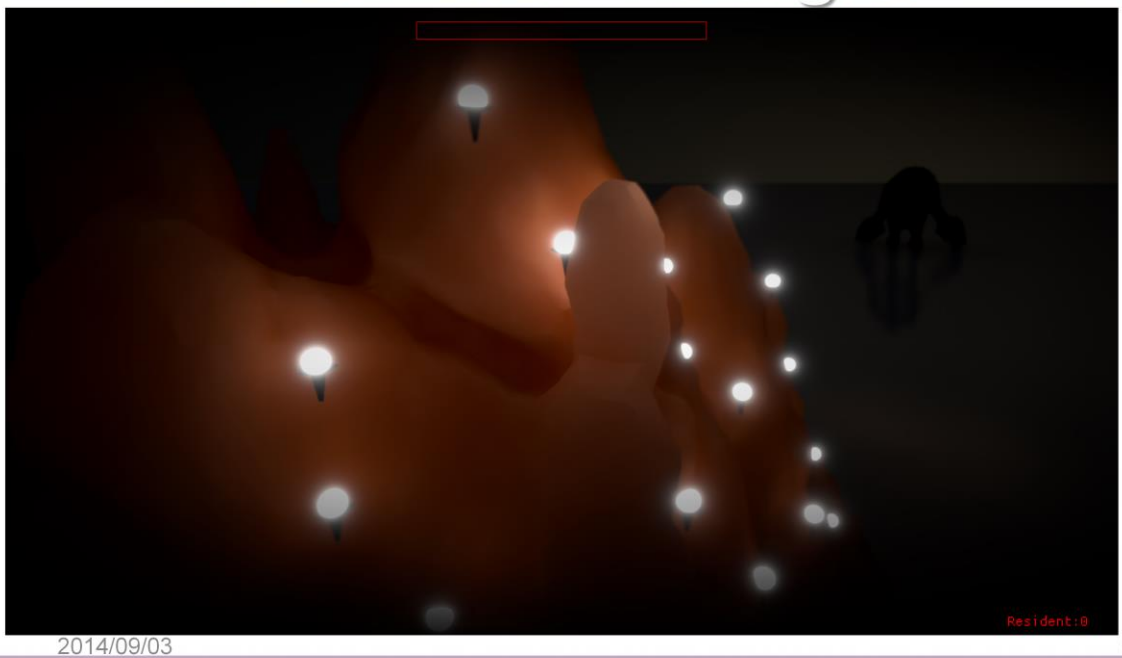
そして、私たちは薄い領域を強調するためにフロスティング効果を追加することができて、よりリアルなインパクトがあるでしょう。

At Night

If we take a look at this same material at night

私たちは夜にこの同じ材料を見ると

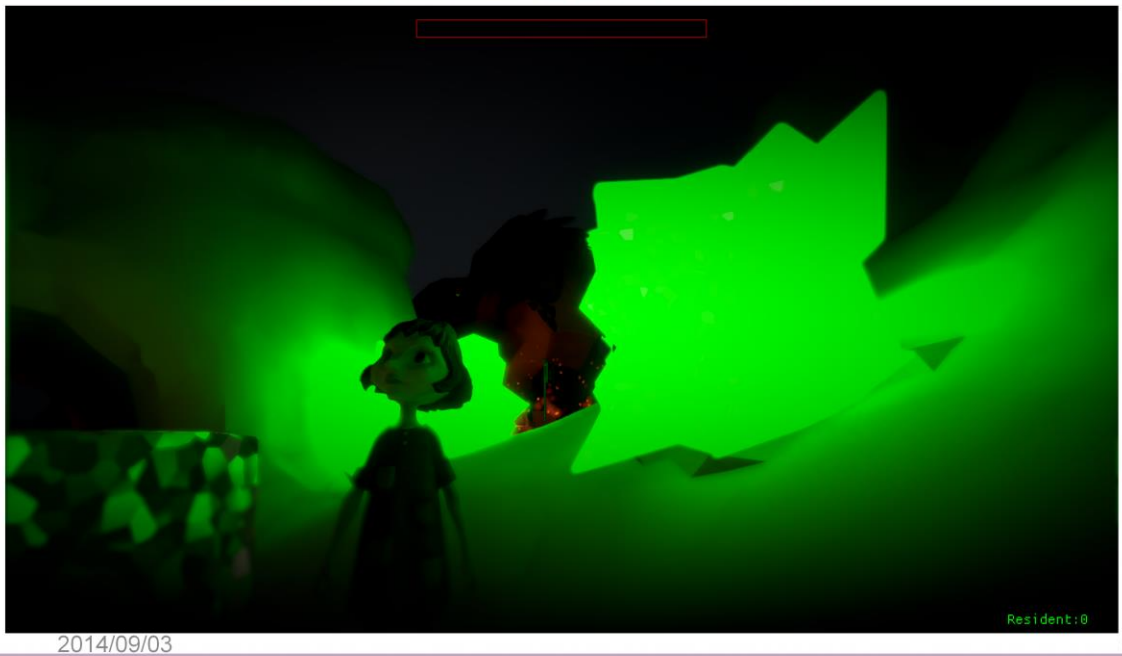
Subsurface – At Night



We can see more clearly how this SSS sampling scheme allows light to bleed through the landscape.

このSSSサンプリング方式は、光が景色の中に光がこぼれるのを見ることができます。

Emissive Materials



Whilst I'm showing you glowy things. I should probably add, its also very easy to support emissive materials in our engine, as they fit very naturally into cone tracing.

We simply need to inject the radiance on the surface of the material into the voxel grid, and we're away.

コーントレースに自然にフィットするように発光マテリアルを実装することは簡単でした。

ボクセルグリッドのサーフェースマテリアルに光量を注入するだけ。

Ray Marched Reflections

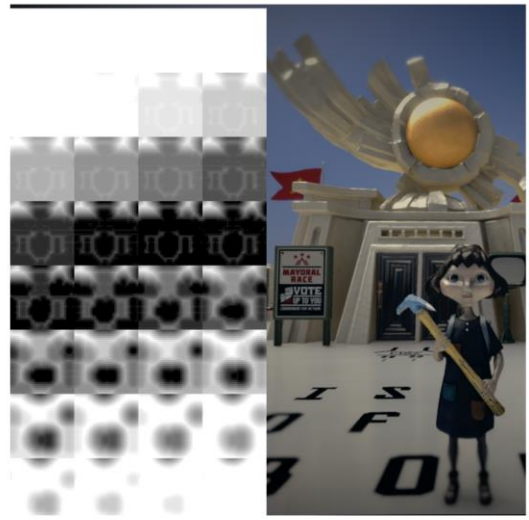
The simplified SH texture, is also very useful for other effects, like reflections.

簡略化されたSHテクスチャーは、リフレクションのような他の効果にも使えます。

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



It's very fast to build a Signed Distance Field for our cascades with Jump Flooding, even though they are 3D.

We generate one for our landscape and objects, and one for our lights.

Once we have these they can be used to accelerate a ray march, through our voxel data.

ジャンプフルーディングを使えば、カスケードのディスタンスフィールドを構築するのはとても高速におこなえます。

ランドスケープオブジェクトは一つ、光量も一つ。

それを獲得したら、ボクセルデータを通るレイマーチの動きが実現できる。

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

We can then sample from our SH cascade texture when we get close to a surface or a light, and accumulate the results.

You can see from my picture here that this gives us a very course view of the world, but it's good enough for glossy specular reflections

And has the advantage that we can reflect objects even when they are off screen.

サーフェースやライトに近い時には、SHカスケードテクスチャからサンプリングすることが出来ます。そして結果を蓄積します。

この画像から見るとこのワールドビューは非常にラフです。ですが、グロッシー・スペキュラー・リフレクション反射の表現には十分です。

リフレクトオブジェクトがオフスクリーンの時にオブジェクトの反射をすることが出来る。

レイマーチによる反射表現

So in this scene with our burning town hall. We can see the reflection of the fire on the floor, and if we look down…

これは火事になっている建物です。床に反射している炎が見えますし、そして下を見たら・・・

Ray Marched Reflections - On

You can see that this still give us a nice, relatively sharp image of our surroundings,
Notice how we can see the fire and the hole in the roof, even when the are off screen.

これは周囲の比較的鮮明なイメージを提供しています。
火災の明かりや屋根に空いた穴が映り込みます、オフスクリーンのイメージなのに反射が見えている。

Ray Marched Reflections - Off

And you can see all the detail we lose if I turn the effect off

効果をオフにした場合に何が失われたかわかるでしょうか

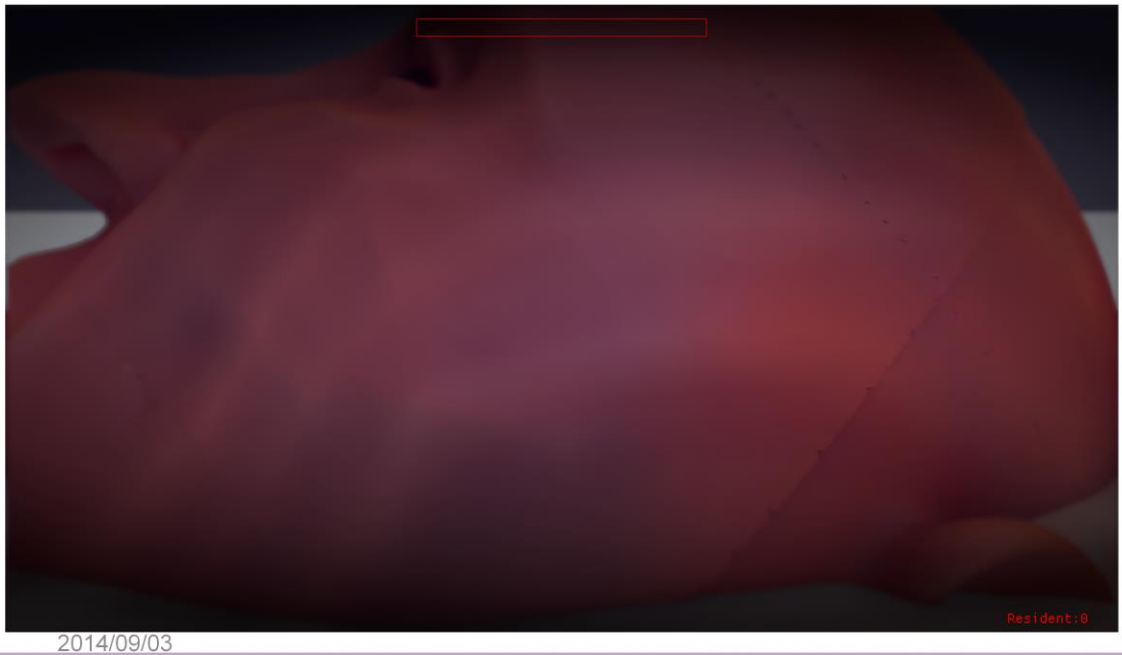
屈折を表現したマテリアル

And of course, we can also extend this approach to allow us to have objects that appear to exhibit glossy refraction.

Like these monuments.

グロッシー・リフラクションのような、オブジェクト表現の為にアプローチを拡張することができます。

サンプリングによる問題点



So, it’s not all sunshine and light in our new voxel world.

There are a few issues that have caused us trouble.

One of them is sampling.

As you can see from the picture above, if we are not careful, then once our voxels get large, significant sampling artifacts can start to show themselves.

ただし、新しいボクセルワールドはどこにでも太陽やライトの光があるわけではありません。

トラブルを起こしている幾つかの問題があります。ひとつはサンプリングです。

気を付けないとボクセルが大きくなるほど、サンプリングのエリアシングが出てきます。

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

Typically we've been solving this by biasing our cone trace so that it starts half a voxel away from the surface of our object.

This works well enough in a lot of cases, but fails in cases like the giant head where we have planar or very smoothly curving surfaces.

Our solution to this has been to do something similar to what we did for the frosting in for the SSS, and add another output to our SSDO, this time one that

Gives us a measure of curvature, both concave and convex.

With this we can choose to bias the cone trace out further in low curvature areas, whilst still retaining the detail we want when we have complex objects.

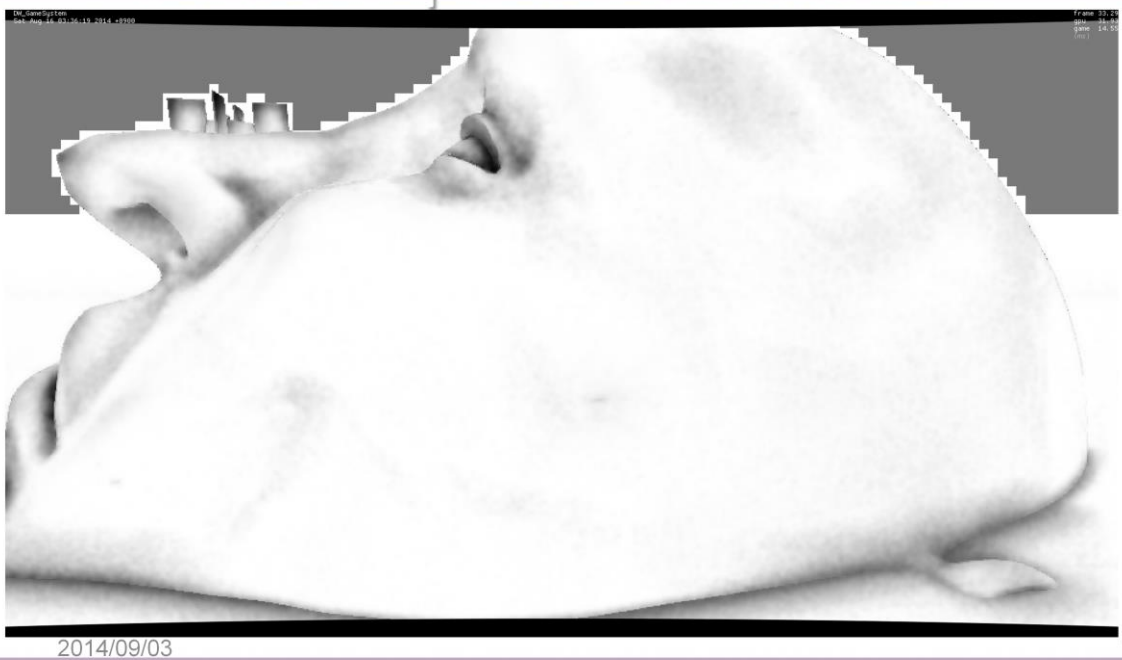
サンプリングによる問題を回避するために、コーントレースに、バイアスさせて、オブジェクトの表面から半ボクセルの距離をスタート地点にしています。

これでほとんどのケースが解決しますが、残念ながら巨大な地形の場合で、とてもスムーズな広い局面がある場合にまだまだ問題があります。

ですので、もう一つの解決方法は、先ほど申し上げたようにフロスティングSSのようなアプローチでSSDOにもう一つのアウトプットを足すことです。

このアウトプットは、曲面の曲率の価値が計算できます。この計算の結果に基づいて、緩いカーブのときにコーントレースバイアスの距離を変更して、より複雑なオブジェクトを細かく読み込みできます。

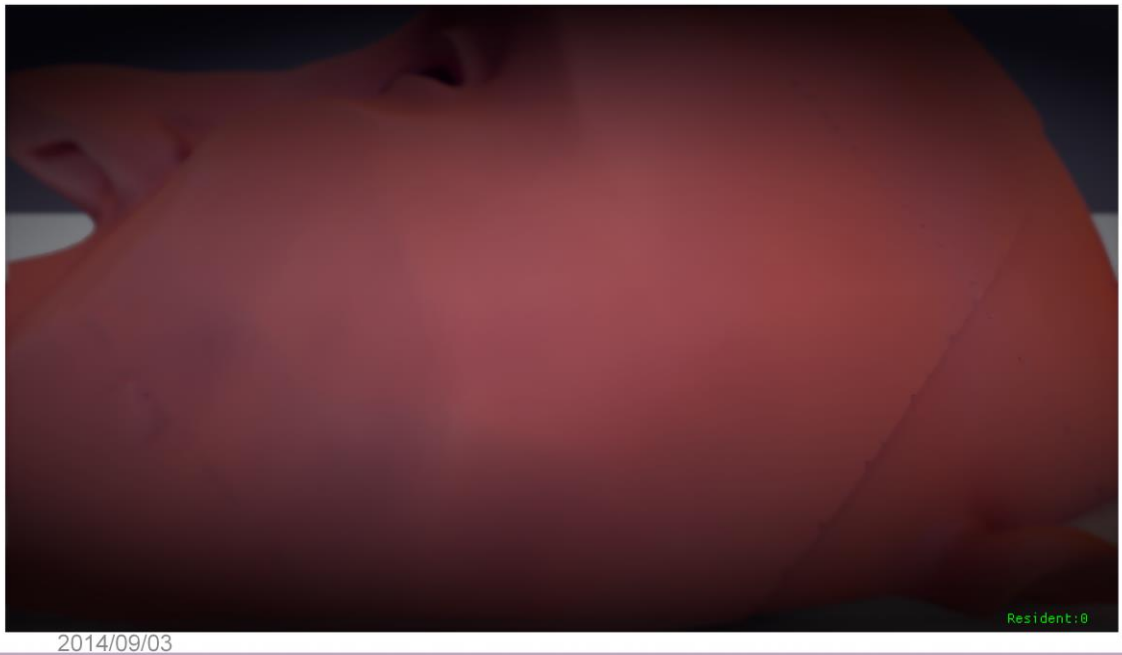
Screen Space Curvature



So this is the sort of output we get from our screen space curvature.

これは私達のスクリーンスペースの曲率から得られるアウトプットのようなものです。

Modified Bias



And you can see that this has a dramatic effect on the alias issues that we were seeing.

これは私たちが見ていたエリアシング問題に劇的な効果を持っていることがわかります。

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

The other problem we have hit, is that our we only have 6 levels for our cascades,

but we want to potentially trace for quite large distances, requiring our cones to get even wider than we have data for, as some of our objects are huge.

We could use a Clipmap, but it doesn't fit in very naturally with out texture addressing scheme.

We could also add more cascade levels which is a more natural fix, but both this idea and the clipmap idea suffer from some very similar issues.

Because we filter our data in 3 dimensions when we generate mipmaps,

Occlusion data from courser levels has a tendency to travel up our cascades faster than our cone trace actually ascends them.

This manifests itself as the problem that typically, the top MIP of our voxel chain is always semi opaque, which means that we always end up occluded

This is not a very good situation for direct lighting.

私たちのもう一つの問題は、カスケードが6つのレベルしか無いことです。

しかし、ゲームの中のあるオブジェクトは巨大なサイズであり、長い距離をトレースしようとするとコーンの広さがゲームの環境データを上回る大きさになります。

クリップマップを利用することもできますが、ゲームの中にあるテクスチャーアドレスアプローチにはあまりよくフィットしない。

もう一つの解決方法は、6つより多いカスケードレベルを足すこと、ただし、これもク

リップマップと同じアプローチの欠点があります。

ゲームデーターは3次元でフィルタされています。そしてミップマップを作成しようとした場合、もしくはカスケードレベルでボクセル化する場合に、オクルージョンデータがカスケードに上昇する速度が、コーントレースの速度より早いという問題が発生します。

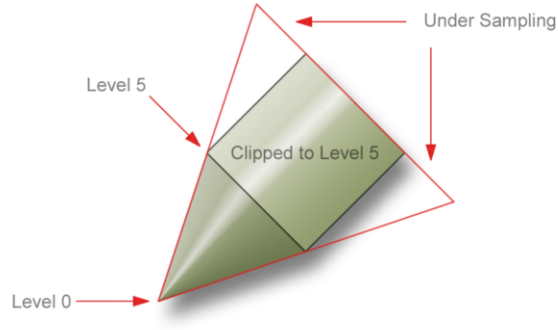
そうなるとボクセルチェーンの最後のミップ、が半透明ですので結果として、空の光がはっきりしない時点で消えます。

だからこそ、直接照明としてあまりお勧めできないアプローチである。

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



Our current, workaround for this, issue is to trace a big cone, but to just clamp our texture sampling to top out at the top level of our cascade data.

This somewhat solves the “always semiopaque” issue, but has the unfortunate effect of turning our cone into a cylinder in the distance, and leaves us with undersampling.

This becomes apparent on some of our shadows in a few places.

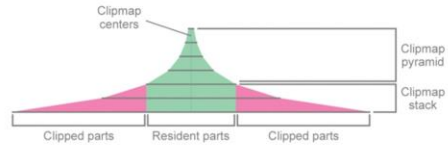
ですので、現在利用している解決方法は、大きなコーントレースをすることです。ただし、テクスチャーサンプルはカスケードデータの高いレベルで終了させます。

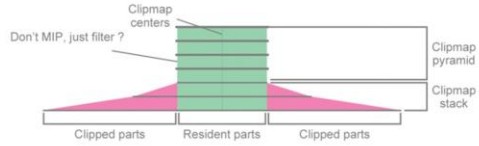
これは不透明の問題を解決しますが、残念ながらコーンの遠いところは広がらずにシリンダー形状になる。結果はアンダーサンプル現象です。

日陰を見るとところどころこれがわかります。

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





What we've been thinking about doing to solve this is to add the extra cascade levels

But keep the resolution of those new cascades the same as our current top level,

And filter each of our precombined cascades for each of our 16 directions in the plane perpendicular to their direction.

A number of extra cascade levels produced like this should hopefully allow us to keep our cone trace, a cone trace, but avoid the "always semi opaque" problem, we hope!

ですので、最終的な解決方法を考えまして、カスケードレベルを足すしかないと考えている。

ただし足すカスケードレベルの解像度は現在のトップレベルの解像度と同様にします。

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。

これでコーントレースのままで、半透明問題を解決するのではないか。

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

Just a few quick words before I wrap up about how expensive this all is.

It's generally taking us on the order of about 3ms a frame to update our cascades, slightly more if we have to voxelize.

Our screen space cone tracing takes somewhere on the order of 3ms.

3.5 ms for the specular ray march,

and 2.5ms to do our final upscale and combine pass, that takes all the various elements, including SSDO and occlusion, and spits out a shaded pixel.

And of course, as you could probably guess from my repeated mention of the word "texture" we use a rather large amount of memory for textures,

currently somewhere north of 600mb.

最後にコストについて一言、言いたいと思います。

各フレームが大体3msごとにカスケードをアップデートされ、ボクセル化するならより長い時間がかかる

スクリーンスペースコーントレースも3msかかります。

スペキュラーレイマーチが3.5ms

そして、最後のアップスケールと合体するパスが2.5ms

それがすべてSSDOもオクルージョンも込みで一つの計算済みピクセルをアプトプットします。

そして皆さんは多分“Texture”の言葉から想像しているとおもいますが、テクスチャ用のメモリとしてかなり大きな容量を使っています。今のところは600mb.

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！

One thing I would like to mention before I finish is about our use of Async Compute.

We've used it very heavily throughout the project, and most of our Screen Space (or Voxel Space) work is in compute shaders,

with large amounts of that running on 3 async compute queues that we have set up in addition to our graphics context.

On a heavy scene we get back around 5ms on a 33ms frame from using Async Compute.

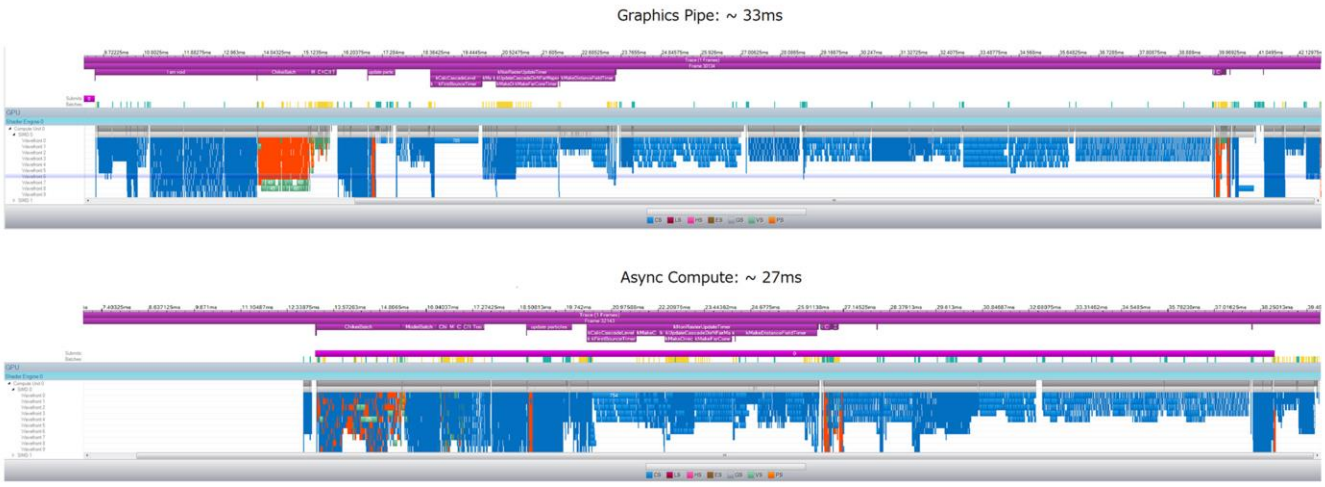
Asyncコンピュートについて話したいことがあります。

このプロジェクトで良く使っていました。

ほとんどのスクリーンスペース。あるいはボクセルスペースの働きはコンピュートシェーダを利用して通常のグラフィック処理よりAsyncコンピュートを3つの行列で特別に設置しました。

Asyncコンピュートのおかげで、絵的に重たいシーンだったら33msのフレームより、5msバックされます。

Async Compute



Here is a RTTV capture of the same, fairly heavy frame.

On the top we're using just the graphics pipe.

On the bottom we're using Async Compute.

As you can see on the bottom, everything is a lot more overlapped, and we take about 5 or 6ms less.

This is with exactly the same shaders, doing exactly the same work.

So, anyway, if you aren't looking at using Async Compute on PS4 yet, YOU SHOULD!

これは画面全体を描画し、(私は光沢のある反射率の高い壁を見ていた)、同じフレームです

上の画像にではグラフィックス·パイプを使用しています。

下の画像ではAsync Computeを使用しています。

下の画像で気がつくように、沢山のオーバーラップがあり、そこで5msを稼いでいます。

これはまったく同じ仕事をして、まったく同じシェーダである。

まだPS4でAsyncコンピュートを使用していないのであれば、とにかく、このことを確認してください！

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

I'll just quickly wrap up with the things we want to fix in the future

Firstly we'd like to be able to support higher frequency shadows.

We have been able to get away without them for this game,

But if you wanted to make a game with a more normal look, then you'd probably want them

We don't think that adding RSMs would be a particularly difficult thing to do in our engine in the future,

but we'd really like to experiment with generating a more accurate distance field and using a tight cone trace to create soft voxel shadows

Obviously we'd like to improve the resolution of our voxel grid, possibly by

Using bricks, which would increase our complexity slightly, but probably still be preferable to switching to an Octree.

We'd also like to get some bounce from characters and vehicles in there, possibly via some limited form of injection, or perhaps by doing it in screen space.

And finally, we'd really like to improve our material model, as whilst we are energy conserving in the cone tracing, our specular model is currently far from physically correct.

未来に解決したい物を手短に説明します。

最初に、高精度のシャドウをサポートしたいのですが・・・、今回のゲームはそれを使わずに逃げ切ることが出来ました。

しかし、みなさんがより通常のビジュアルでゲームを作りたいのであれば、シャドウは必要になるでしょう。

将来のエンジンであれば、RSMを追加するは難しくないだろう。

私達はソフトボクセルの影を生成するためにタイトなコーントレースをつかい、より正確なディスタンスフィールドを生成することを試してみたい。

ボクセルグリッドbの解像度もアップしたい。一つの方法はBlicksを使うことですが、少し複雑になりますが、まだオークツリーよりましかもしれない。

そして、キャラクター、もしくは車体へバウンスを追加したいと思っています。

マテリアルモデルを改善したいです。コーントレースはエネルギー節約という面はあるのですが、当社のスペキュラーモデルは残念ながら物理的に正確な表現ができていないところである。

Special thanks to Tao Yung, who implement our particles and refractive objects.
SCEJ for being willing to let us go in our various crazy directions and for supporting us,
Yoshida san for helping me to be able to give this presentation
And the rest of the awesome team back at Q.

パーティクルとリフラクティブの実装をしたTao、
このプロジェクトに粘り強くつきあい、そしてサポートしてくれたSCEJの方々
もちろんQ-gamesのスタッフの方々に感謝します。
日本語の説明はQ-gamesでスタジオディレクターをしている私、吉田が行いました。

ありがとうございます。

We're Hiring!

Q-Games

Any Questions?