

Acceleration Data Structures for Ray Tracing

Shinji Ogaki

スライドはネットで配ります。なので写真は撮らなくても大丈夫です。PPTだけはいろんな人が見られるように英語で書いるので、ご了承ください。

今日は現在主流となっているアルゴリズム、また知つておいた方が良いと思われるアルゴリズムを中心に紹介します。激辛としましたが、たいして難しくないので、気楽に聞いてください。

論文そのままの実装ではなく、どうすれば楽をして同じようなことが出来るか、ということを意識して話をていきたいと思います。それではよろしくお願ひします。

~~Acceleration Data Structures for Ray Tracing~~ Bounding Volume Hierarchy

Shinji Ogaki

タイトルはレイトレーシングのデータ構造を極めるとしましたが、いまはBVHが主流なのでBounding Volume Hierarchyとすべきだったかもしれません。

Acceleration Data Structures for Ray Tracing

- Uniform grid
 - The tea-pot-in-a-stadium problem
 - Not suitable for motion blur
- kd-tree
 - Construction cost
 - Not suitable for motion blur
- BVH
 - Suitable for motion blur!
 - Fast traversal (wide BVH)

さて、レイトレーシングのデータ構造は様々なものが存在します。過去にはユニフォームグリッドやkd-treeなどが良く使われていましたが、現在はBVHが主流となっています。ユニフォームグリッドはteapot in a stadium問題、モーションブラーを苦手とし、kd-treeは構築に時間がかかり、同じようにモーションブラーの扱いを苦手としています。一方BVHはモーションブラーを容易に扱うことができ、またWide BVHの登場によってトラバーサルも高速に行えるようになったことから、人気になりました。



知らない方もいると思いますので、Tea pot in a stadium問題について紹介します。さてこのスタジアムのどこにティーポットがあるでしょうか？



実はPの下に小さいティー pocot が置いてあります。

これは、球場でティー pocot を見つけるのは本当に大変だ、ということではなくて、ユニフォームグリッドを使っている場合に、レイが、一か所ポリゴンが密集したところに当たると、著しくパフォーマンスが落ちる問題のことと言います。kd-treeやBVHといったデータ構造は階層的にデータを保持するので、この問題に悩まされることはありません。

Acceleration Data Structures for Ray Tracing

- Other data structures
 - Bounding Interval Hierarchy
 - Matrix Tree
 - R-Tree
 - Dual-Split Tree
 - and a lot more...
- BVH
 - Intel® Embree and NVIDIA® OptiX use BVH
 - Production renderers use BVH
 - This talk focuses on BVH

他にも様々なデータ構造が提案されています。しかし、今日では全てのレンダラといっていいほど、多くのレンダラがBVHを使用しています。IntelのEmbreeやNVIDIAのOptiXはともにBVHを使用します。当然ながら、この2つを利用したレンダラはBVHを使用していることになります。また自前のデータ構造を使用するプロダクションレンダラも自分が知る限りではほぼ全てがBVHを使用しています。

ですので、この発表ではBVHの構築の仕方や最適化の方法を紹介していきます。

Preliminaries

Traversal

Wide BVH

Cost function

まずは予備知識です。みなさんレイトレーシングについては詳しいかと思いますが、基本からお話しします。

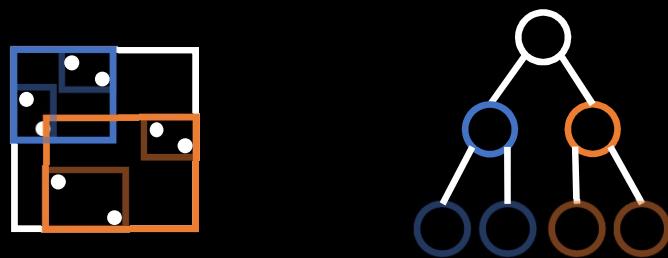
Ray Traversal

- First-hit
 - Radiance ray
- Any-hit
 - Occlusion ray
- Multi-hit
 - Useful for rendering non-opaque objects, etc.

レイトレーシングで用いられるテストは大きく分けて3つあります。最もよく使われるのはレイと交差する一番近いオブジェクトを見つけるFirst hit。これは輝度の計算などに使われます。次に何かがレイと当たるかどうかだけを調べるAny hit。これはAmbient Occlusionや影の計算に使われます。最後にMulti hitはレイの原点から近い順に1つより多くのレイと交差するオブジェクトを見つけるもので、透明なオブジェクトのレンダリングなどに使用されます。

Multi hitを用いて先に交点を全部リストアップしてから、あとで一気にまとめてクリップや透明度を計算するためのシェーダを実行することもあります。

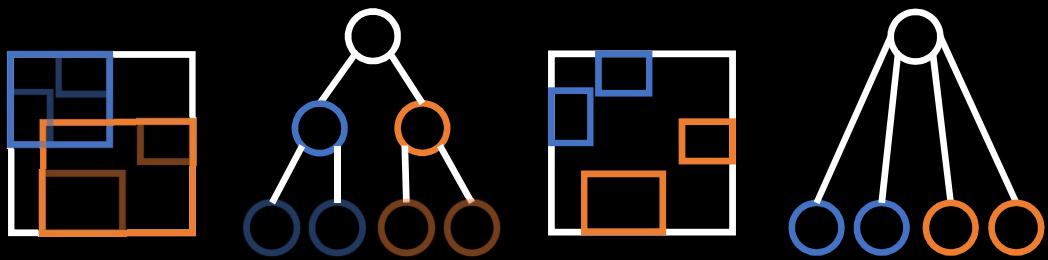
Bounding Volume Hierarchy



BVHでは左のような分割があった場合に、右のような木の構造で表します。
ここに来てくださっている皆さんにはもう当たり前のことと思います。

Wide BVH

- The standard (binary) BVH node has two children per node
- Wide BVH node has more children (e.g. 4, 8, 16, ...)
- Less redundant data
- Better SIMD utilization
 - SSE, AVX, AVX512



次にWide BVHに触れておきます。

先のスライドにあったように、標準的なBVHは各ノードが2つの子を持ちます。Wide BVHは各ノードが4,8,16といった多くの子を持ちます。こうすることによってSIMDの利用率を上げられるというメリットが強調されますが、実は標準的なBVHのノードは親のボックスと接している面が多くて、冗長にデータを保持していることが分かります。左の図だと、各ボックスは親と2面を共有しているのに対し、右の図では一面しか接していないことが分かります。このようにWide BVHにすることで重複したデータが減るので、メモリの消費が少なくなり、トラバースステップも減ります。また、SOAを利用し、子ノードのAABBを連続して保持することで、メモリのアクセスパターンを改善することができます。

計算量に関しては、本来2分木では、しなくてよかった部分についてもテストする必要が出てくるなど、若干不利な部分もありますが、いまのコンピュータは計算は速く、メモリが遅いため、可能なら常にWide BVHを使うべきです。GPUでもWide BVHの使用で高速化が達成できたというレポートもいくつあります。本当に速くなるの、と聞かれることがあります、私の答えはYESです。もちろんいくらでも幅を広げればいいわけではありません。

また、Many Lightsの計算でもWide BVHを使うと、副作用的な効果でノイズを減らすことが出来ます。これについては後ほど時間があれば、紹介します。

Cost Function

- Cost metric directly affects ray tracing performance
- Ray Tracing
 - Surface Area Heuristic (SAH)
 - Easy to compute
 - End Point Overlap (EPO)
 - More accurate
 - Not easy to compute on the fly
- Many Lights
 - Surface Area Oriented Heuristic (SAOH)

BVHの構築の仕方は後で紹介するように色々ありますが、いずれにせよ、適当に構築したのでは品質の良いBVHを作ることが出来ません。ここでいう品質というのはレイ・トレーシングや後ほど紹介する照明計算でのパフォーマンスに直結します。

コスト関数は影計算に特化したものなど、トラバーサルの種類によって最適なものが異なりますが、一般的にはSAHと呼ばれるものが最もよく利用されます。EPOというよりよいものがありますが、その計算自体にコストがかかるためあまり使用されません。SAHは簡単に言ってしまうと、ツリーの「ノードが内包するオブジェクトの交差判定コスト」と「表面積」を掛け合わせたものの総和で表されます。もちろん、厳密にはノードのBounding Boxとの交差判定コストや、メモリのアクセス時間なども加味する必要があります。

また最近はやりのメニーライトの計算ではSAOHと呼ばれるメトリックが用いられます。

Cost Function

- Sunburst chart could be useful for debugging



デバッグや、最適化アルゴリズムが意図したように動作しているか確認するため、どの部分がコストが高いのか可視化してみたい時があります。

色々なツリーの可視化の仕方がありますが、ボリュームレンダリングのようにして立体で表示した場合はノード同士の重なりによって見辛いこともあります。その場合は、Sunburstチャートをつかって、クリックしたとこの表面積や座標を表示するのも一つの方法です。もちろん他のグラフでもよいので、面白い可視化が出来たらぜひシェアしてください。

Construction

Top-down (divisive)

Bottom-up (agglomerative)

Hybrid (e.g. LBVH + Restructuring)

では本題に入りたいと思います。まずは構築方法です。構築方法は大まかにトップダウンとボトムアップに分けられますが、両者を混ぜたハイブリッドな手法も存在します。今日は時間が限られていますので、トップダウン、ボトムアップそれぞれひとつづつ、よく使われているものを詳細にみていくこう思います。

Construction: Top-down

- Recursively divide groups into subgroups
- Leads to high quality trees
- Two common methods
 - Binning
 - Full Sweep
- Object partitioning vs spatial partitioning

トップダウン型の構築は入力されたプリミティブなどのグループを2つ以上のサブグループに分割していく形で行われます。一般的には非常に高品質のツリーを作ることができ、BVHの場合はBinningとFull Sweepの2つが良く用いられます。

分割の仕方はオブジェクトの重複を許さないobject partitioningと重複を許すspatial partitioningの2種類がありますが、どちらにもメリット、デメリットがあります。Objectの重複を許さない場合は必要なBVHのノードの数が事前にわかるので、ハードウェアなどで実装するときには都合が良いです。重複を許す場合は高速なトラバースが可能になりますが、いたずらにメモリを消費してしまうのを防ぐために、どのくらい重複を許すか事前にバジェットを設定して、上限を固定しておく必要があります。

違った見方をすれば、Object partitioningはノードのBounding Box同士が重なる、すなわち空間的な重なりが発生します。そのため、近い順にレイのトラバースを行ってリーフノードで何かにヒットしてもトラバースを終えることが出来ません。しかし、Spatial Splittingの場合はそういった重なりは発生しないため、レイは何かにヒットしたら即座にトラバースを終ることが出来ます。

どちらがよいか、というのは現状Object partitioningに分があるようです。というのも、プロダクションレベルのアセットではある程度メッシュが均等に細かくテッセレーションされているため、ノードの重なりというのはそれほど大きくならないためです。またObject partitioningの場合モーションブラーの実装もシンプルに保つことが出来ます。

Construction: Top-down

- Parallel construction
 - Per primitive at top level nodes
 - Parallelized binning and partition
 - Divide and conquer algorithm
 - std::execution::par
 - Per subtree at deep level nodes
- Simple task manager example
 - <https://github.com/shinjigaki/bvh/blob/master/taskqueue.cpp>

構築の並列化は少し工夫が必要で、トップレベルに近い時にはプリミティブの数が多いので、プリミティブ単位で並列処理を行う必要があります。分割統治法、それからC++だとstd::for_eachやstd::partitionを使うことで簡単に処理の並列化を行うことが出来ます。

分割が進んで、サブツリーの数が増えてきたら、サブツリー単位で並列処理を行う方が効率が良いです。途中では、プリミティブ単位での並列処理と、サブツリー単位の並列処理が混ざります。サブツリーの構築は再帰処理で簡単に書くことが出来ますが、もちろん再帰で書かなければならぬということはありません。

タスク管理を行う必要が出てきますが、サンプルコードを置いてあるので、興味がある人は見てみてください。タスクの終了は全てのプリミティブが処理されたらという条件を使うとバグの生じにくいきれいなコードを書くことが出来ますが、このマシンでテストした限りではActiveなスレッドの数を管理して終了の判定を行った方が高速になりました。様々な方法が考えられるので工夫してみて下さい。

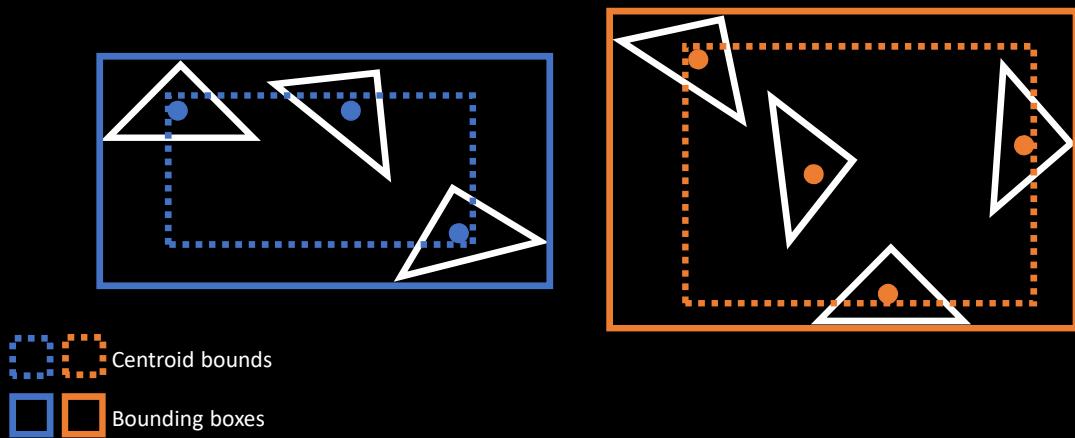
Construction: Top-down - Binning

- Approximation of full sweep but fast and accurate enough

まずは、トップダウンの構築で現在おそらく最も広く使われているであろう Binningと呼ばれる方法について紹介します。BinnigはFull Sweepと呼ばれる方法の近似ですが、高速に動作し、十分に質の良いBVHを作ることが出来ます。N個のプリミティブを2つのグループに分ける方法は、ざっくり 2^N ありますが、それらすべてを試してSAHが最小になるものを探す、というのは現実的ではありません。Binningでは限られた位置でのみSAHを評価し、分割位置を決定します。

Construction: Top-down - Binning

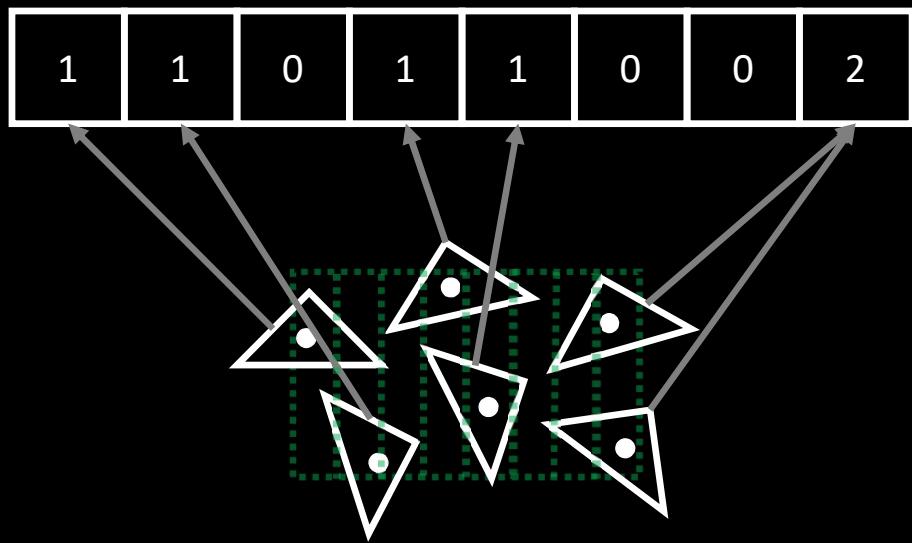
- Divide centroid bound into e.g. 8, 16



まず気を付けてほしいのが、構築にはプリミティブのバウンディングボックスだけではなくてCentroid Boundと呼ぶ、各プリミティブの重心を含むバウンディングボックス、の2種類を使用するというところです。

このCentroid Boundを8や16といった一定数に分割します。深い部分では16、深い部分では8といったように、深さによって分割数を変えて構いませんが、変えても経験的にはトラバーサルのパフォーマンスに劇的な変化はありません。

Construction: Top-down - Binning

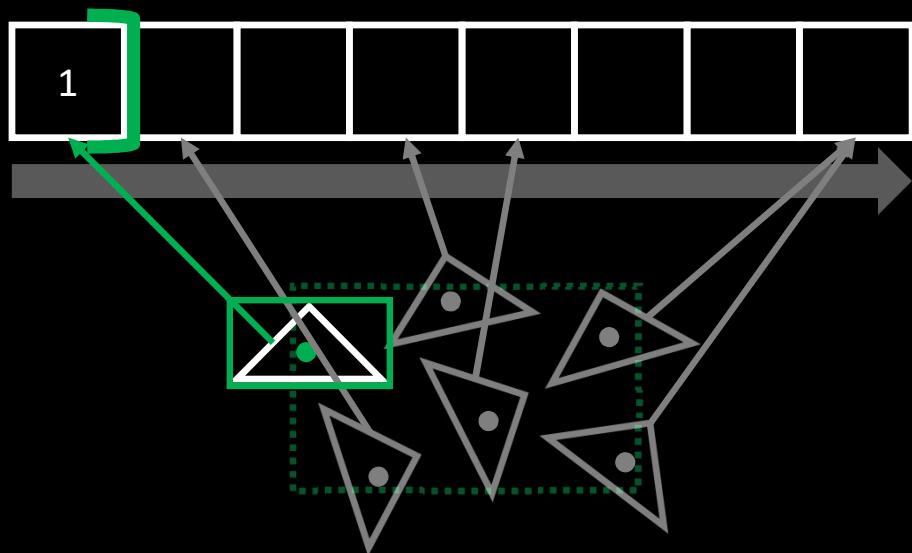


ここではBINの分割数を8にして具体的に見てみましょう。

繰り返しますが、BINはAABBを均等に分割したものではなくて、Centroid Boundを分割したものであることに注意してください。こうすることで安定してBVHを構築することができます。各BINは、そのBINに入ってきたプリミティブの数と、入ってきたプリミティブを全て覆うAABBを持ちます。重心のCentroid Boundにおけるの相対的な位置から、各プリミティブがどのBINに入るかは簡単にわかります。

この方法ではSAHが最小になる分割位置を見つけるため、BINを左から、右から、もう一度左(あるいは右)からと3回スキャンします。

Construction: Top-down - Binning

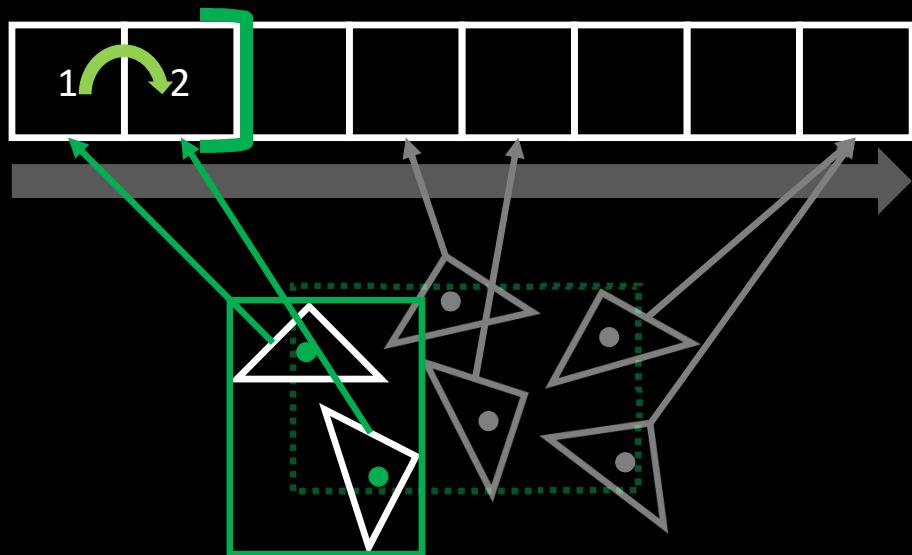


最初のパスではBinを左から右にスキャンします。

プリミティブはある境界（ここでは緑で表されたもの）で左のグループに属するものと右のグループに属するもの、とに分けます。明るい三角形は左側に属するものを表していて、暗い三角形は右側に属するものを表しています。

仮にこの位置で分割したとき、左のグループには1つの三角形、右には5つの三角形が所属します。

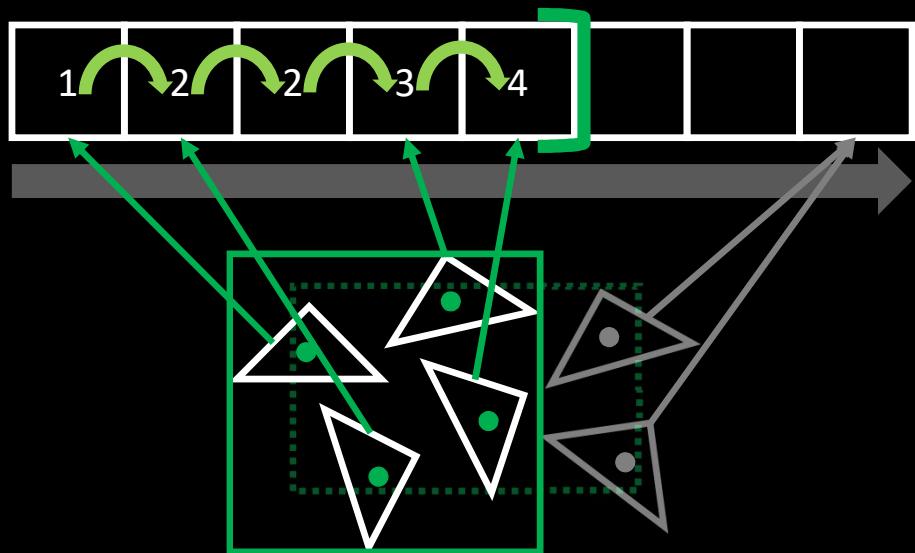
Construction: Top-down - Binning



各BINは自身の値を右隣りのBINの値に足していきます。今は左から走査しているので当然右に行くほど値が大きくなります。また、同じように右隣のBINのAABBを左側のAABBが含まれるように拡大します。Summed Area Tableをイメージしてもらうと分かり易いと思います。

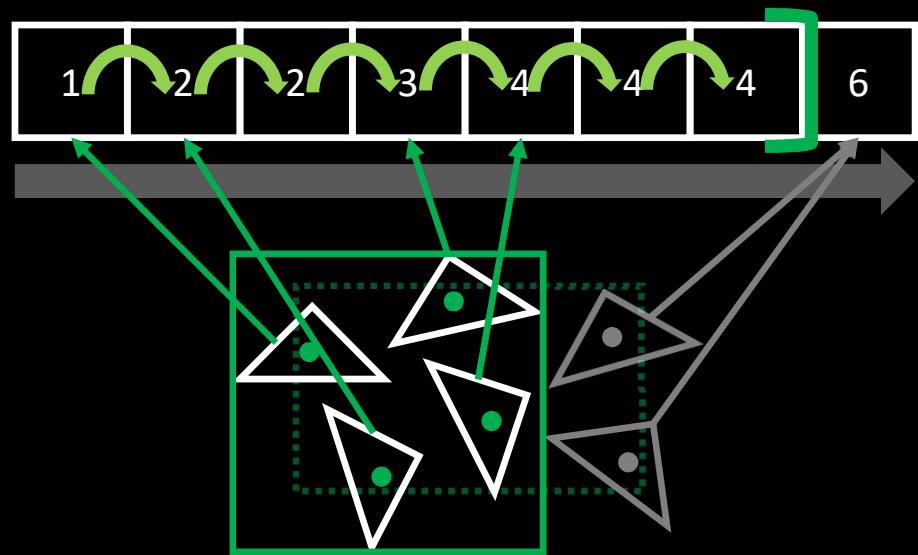
ですので、BINのAABBは右に行くほど大きくなっていきます。先ほどと同様、仮にここで分割したとき、左のグループには2つの三角形、右には4つの三角形が所属します。

Construction: Top-down - Binning



同じように、ここで分割したときには、左のグループには4つの三角形、右には2つの三角形が所属します。

Construction: Top-down - Binning



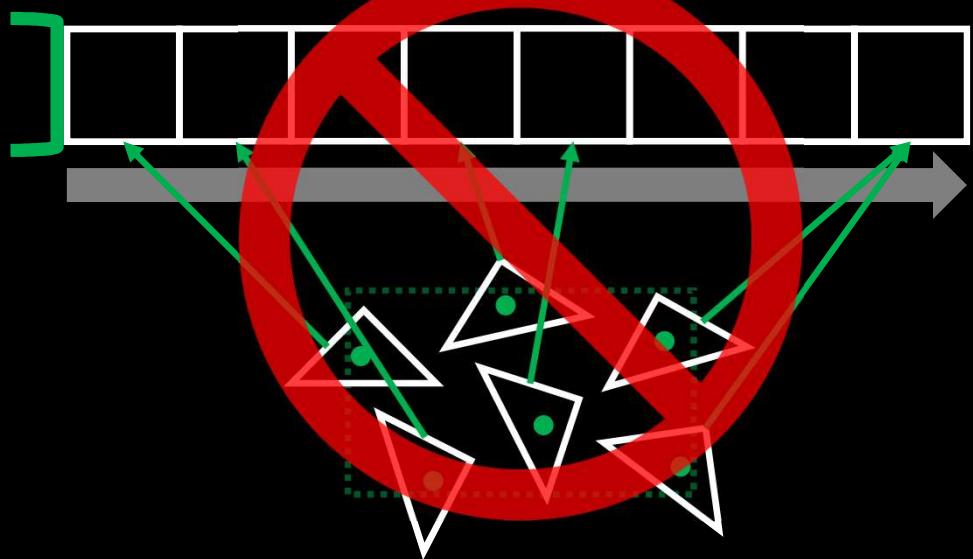
ここまでくると右への走査は終了です。

Construction: Top-down - Binning



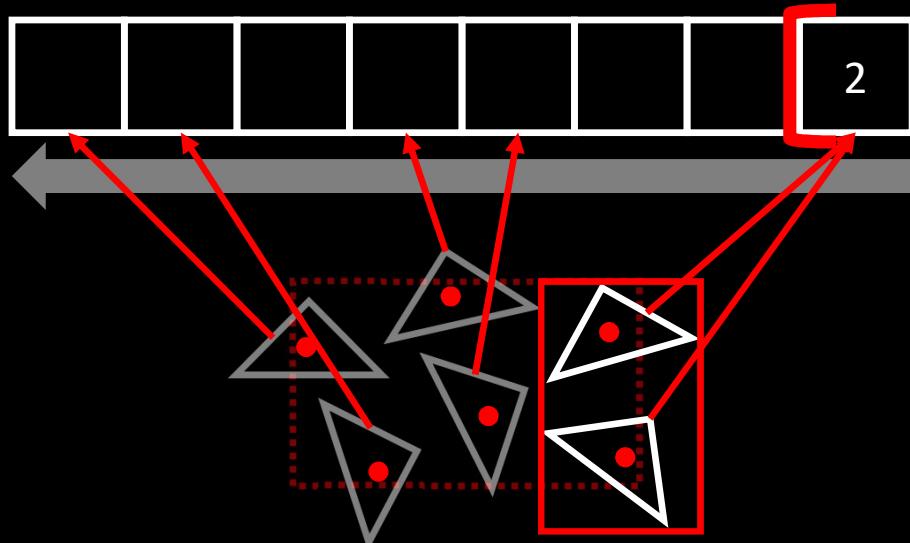
注意ですが、ここはSplitする際の候補には入りません。全部のプリミティブ
が左側に属してしまい、分割する意味がないからです。

Construction: Top-down - Binning



同様に、こっちの端も候補には入りません。

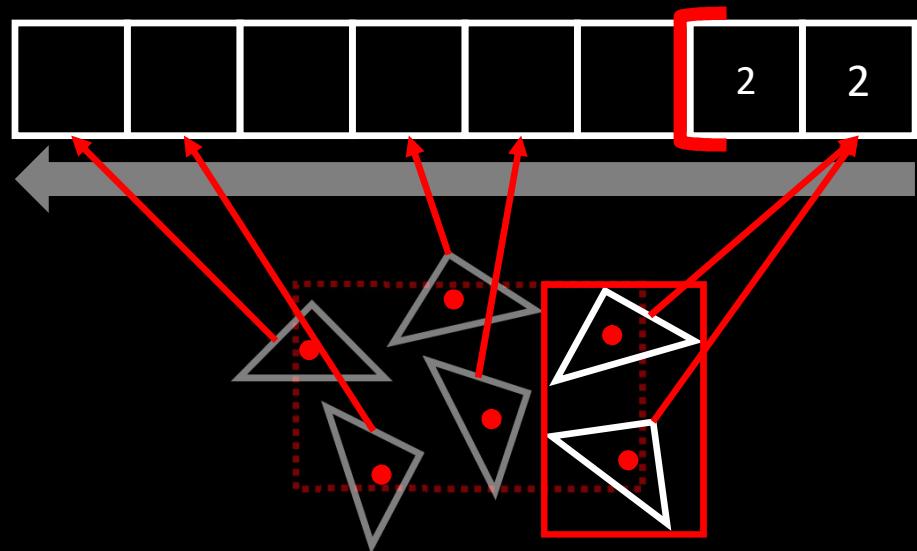
Construction: Top-down - Binning



ここまでで、どこで分割した場合、右と左に属する三角形の数がいくつになるのか、というのを既に求まっています。しかし、右のグループに所属する三角形全てを含むAABBはまだ分かりません。ですので、同様の処理を今度は右から行います。

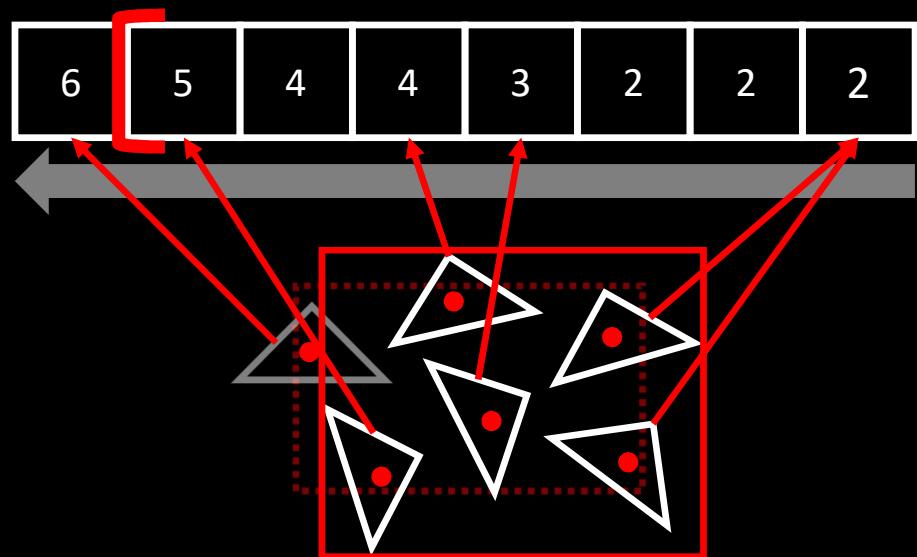
ただし、今回はカウンタを更新する必要はなく、左のBINのAABBを右のAABBが含まれるよう拡大していくだけです。

Construction: Top-down - Binning



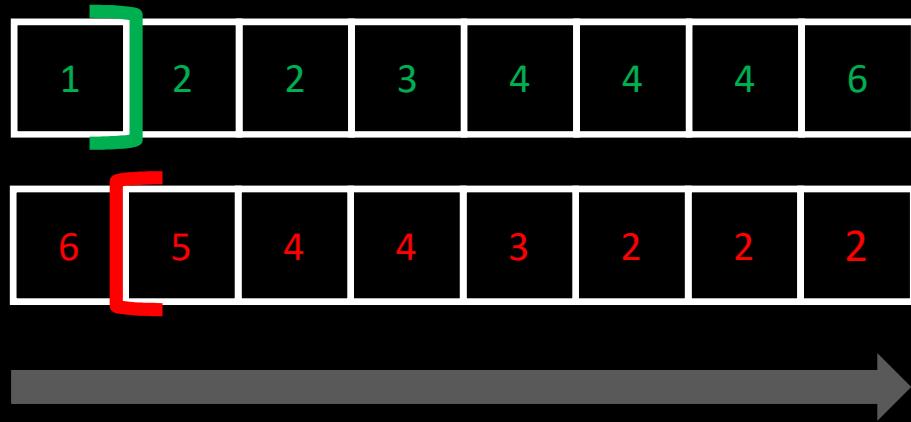
右端のボックスを含むよう、右から 2 番目の bin の AABB を拡大します。

Construction: Top-down - Binning



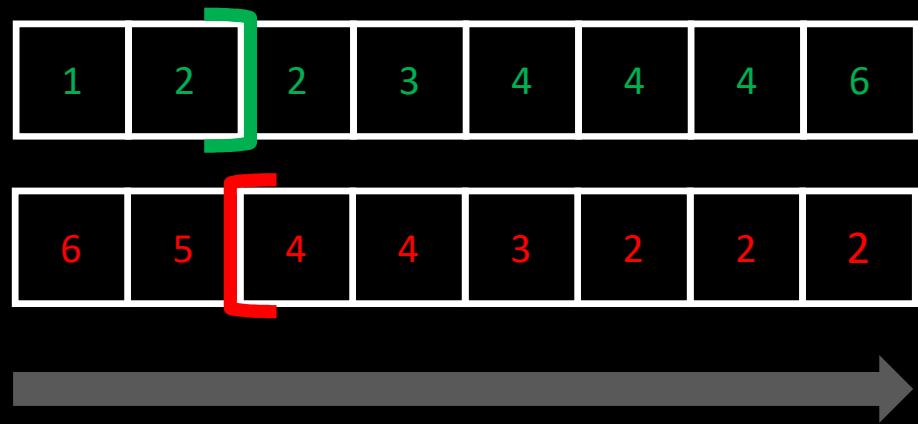
ずっと進めていって、ここまでくれば走査は終了です。

Construction: Top-down - Binning



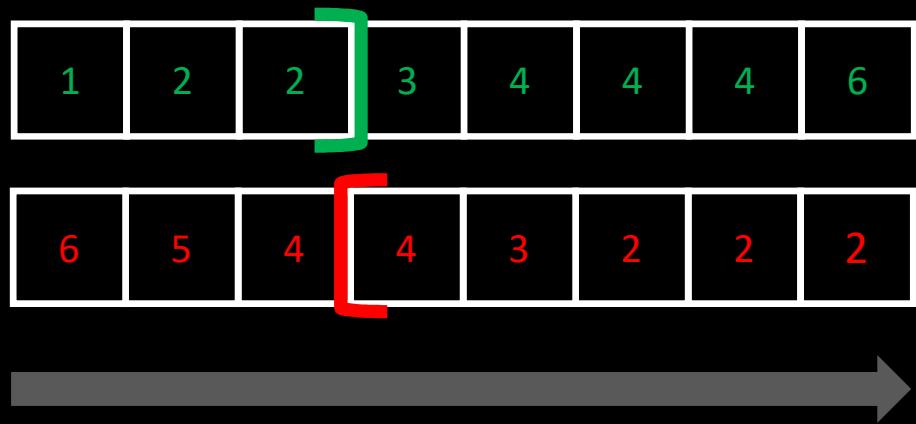
右方向、左方向へのスキャンが終わると、次はどこでグループを分けるのが一番良いかを探します。言い換えるとSAHが最小になる場所を探します。緑のカッコの左側の数字と、赤のカッコの右側の数字を足せば6、つまり与えられた三角形の数となっていることが分かります。各分割位置で先ほど紹介したSAHを計算します。ここでは $1 \times$ （緑で1と書かれているBINのAABB）の表面積 + $5 \times$ （赤で5と書かれているBINのAABB）の表面積がコストになります。

Construction: Top-down - Binning

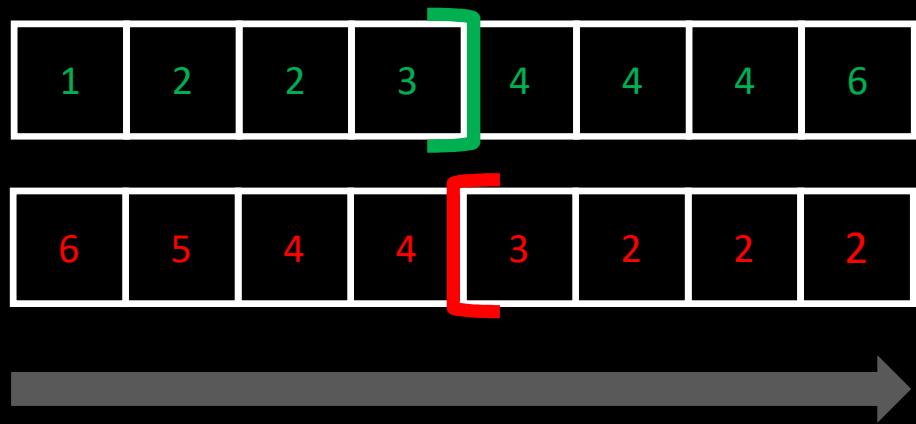


どこでグループ分けしようが数値の和は6になっています。

Construction: Top-down - Binning

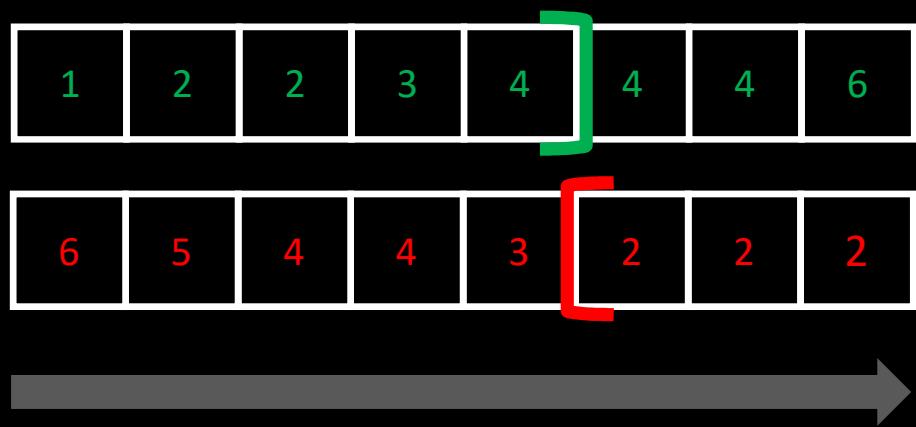


Construction: Top-down - Binning

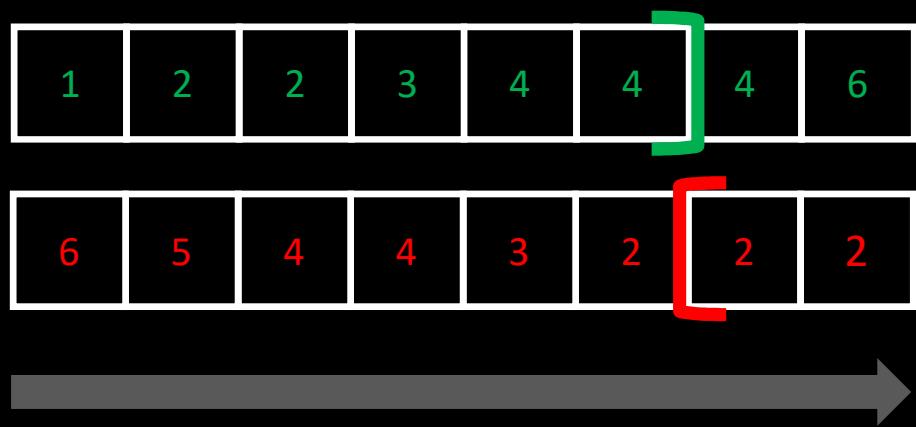


順々にSAHを計算していきます。

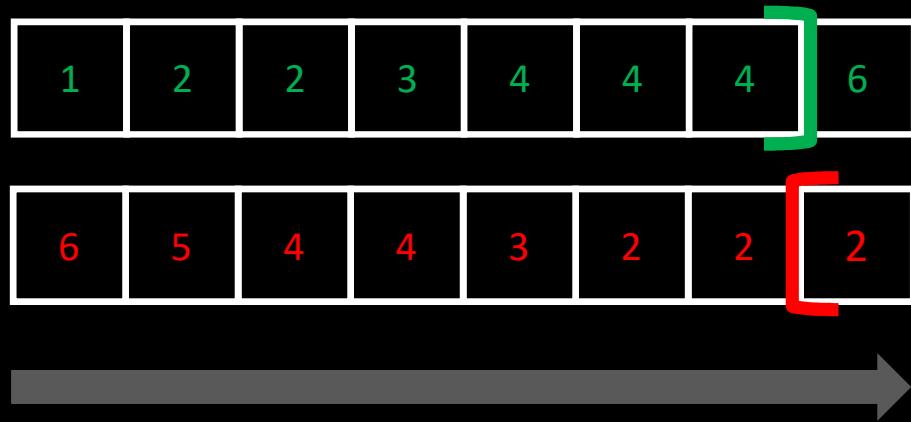
Construction: Top-down - Binning



Construction: Top-down - Binning



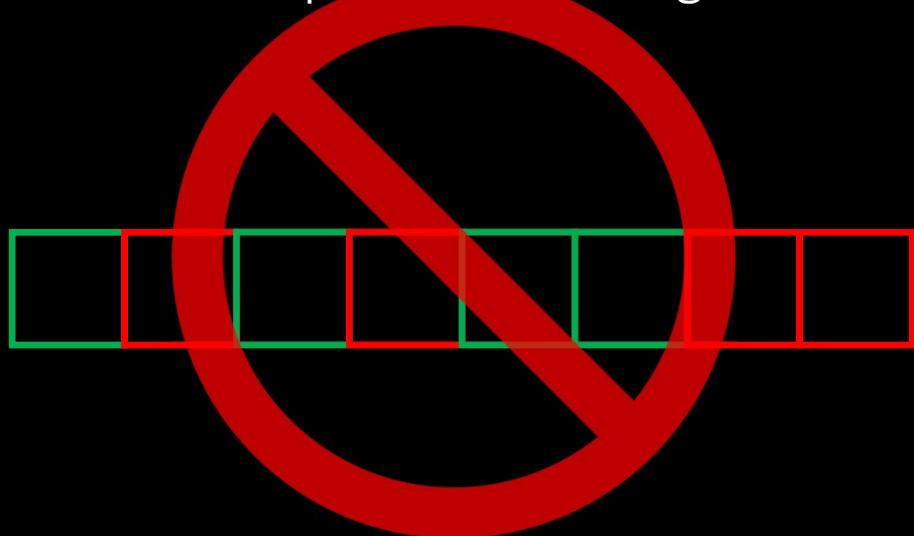
Construction: Top-down - Binning



右端まで行くと、SAHが最小となった分割位置が分かるので、そこで三角形を2つのグループに分けるパーティショニングといわれる処理をして終了となります。

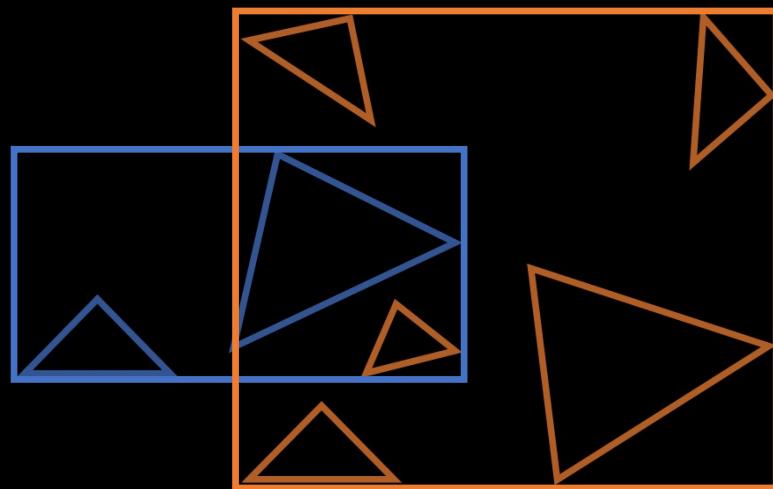
以上の処理をAABBの最も長い軸方向だけに行っても良いですし、XYZ各軸に対して行って一番SAHが小さくなる軸と分割位置を見つけても構いません。パーティションされたグループに対し繰り返しパーティションを行うことでBVHが構築できます。

Construction: Top-down - Binning



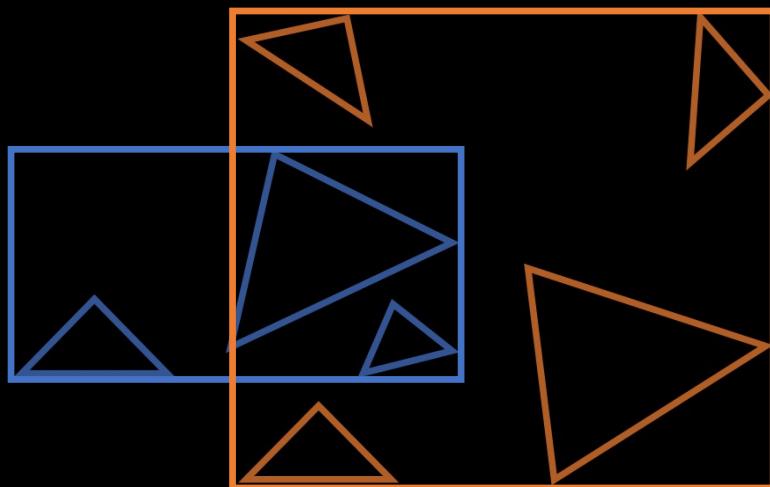
記憶にとどめておいてほしいのは、こういった分け方はしないということです。当然いろいろな組み合わせを調べたほうがSAHを小さくできる可能性がありますが、この例だと、およそ 2^8 通りの組み合わせがあり現実的ではありません。

Construction: Top-down - Binning



パーティションは重心が分割した面のどちら側のBINに入るかで行いますが、しばしばこのような状況に陥ります。ここでは青いAABBの中にオレンジの三角形が一つ入っていますが、これは重心を使ってBinningするためにおこる現象です。

Construction: Top-down - Binning

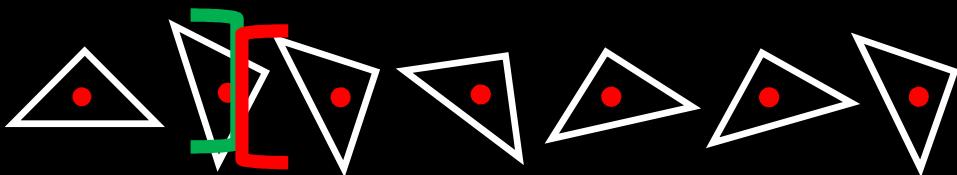


この場合は当然ながら、オレンジの三角形は青いAABBに入れたほうがコストが小さくなります。Stefan Popov氏の“Algorithms and Data Structures for Interactive Ray Tracing on Commodity Hardware”に詳細がありますので、ぜひ読んでみてください。

この考え方は、後で紹介する最適化のテクニックを構築時に使用しているともみなすことが出来るので、面白いと思いました。

Construction: Top-down – Full sweep

1. Prepare a ref array for each axis
2. Sort ref arrays along x, y & z
3. Test all splitting positions (center of each object) to find the best splitting plane that minimizes SAH
4. Partition ref arrays
5. Go back to 3 if there is more than 1 subgroup to divide



Bottom Upの構築に移る前に、Full Sweepといわれる方法をさっと紹介したいと思います。

まず、XYZ3軸それぞれにリファレンスの配列を用意します。ここでリファレンスはプリミティブを指すポインタあるいはIDのことを意味しています。

初めに一度だけ各配列をソートします。この時、ソートにはCentroidの座標を使います。

プリミティブそのものではなくリファレンスをソートするには理由が二つあります。

一つは、バウンディング・ボックスやトライアングルをスワップするのはコストがかかるのでそれを避けるため。もう一つは、データをホストアプリケーションとシェアする場合、プリミティブの実体を並べ替えてしまうとホスト側で順番が変わってしまい動作がおかしくなるのを避けるためです。

次に、各軸にそって最もSAHを小さくする分割場所を探し（Sweep）、もっともSAHを小さくする軸と分割位置を見つけます。見つかったら、ソート済

みのリファレンスに対し、パーティションを行います。分割されたグループがまだ大きければ3に戻ります。分割されたサブグループが十分小さくなるまで同様の処理を繰り返します。

以上がトップダウンです。

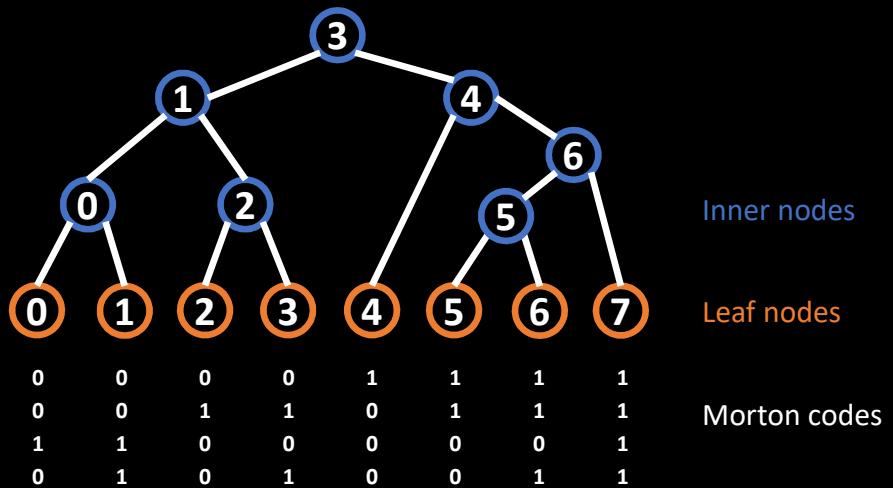
Construction: Bottom-up

- Agglomerative clustering
 - Fast Agglomerative Clustering for Rendering
 - Efficient BVH Construction via Approximate Agglomerative Clustering
- PLOC
 - Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction
- LBVH
 - Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees
 - [Fast and Simple Agglomerative LBVH Construction](#)

ボトムアップの構築方法にも様々な手法が提案されています。ここではLBVHも一緒に並べました。

Agglomerative Clusteringを使った最初の論文は近傍のペアを探すために、補助的な空間データ構造を使っているので、実用上問題があります。後に Distance Matrixを使い効率よく Agglomerative Clusteringを行う方法も提案されました。しかしその考え方は最適化の手法であるRestructuringに用いることが出来ます。PLOCは最近傍ペアをうまく見つけて品質の良いツリーを作ります。最近傍ペアの考え方は知つておいて損がないと思います。LBVHはここに挙げた以外にもたくさん論文が出ていますが、今回は、その中でも一番わかりやすいFast and Simple Agglomerative LBVH Constructionを紹介したいと思います。

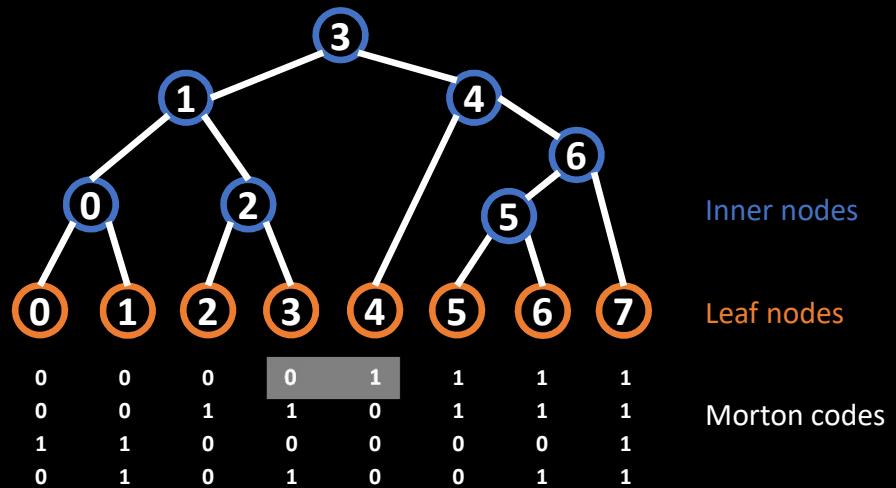
Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



下に並んでいるビット列はMorton Codeです。青はインナーノード、オレンジはリーフノードです。

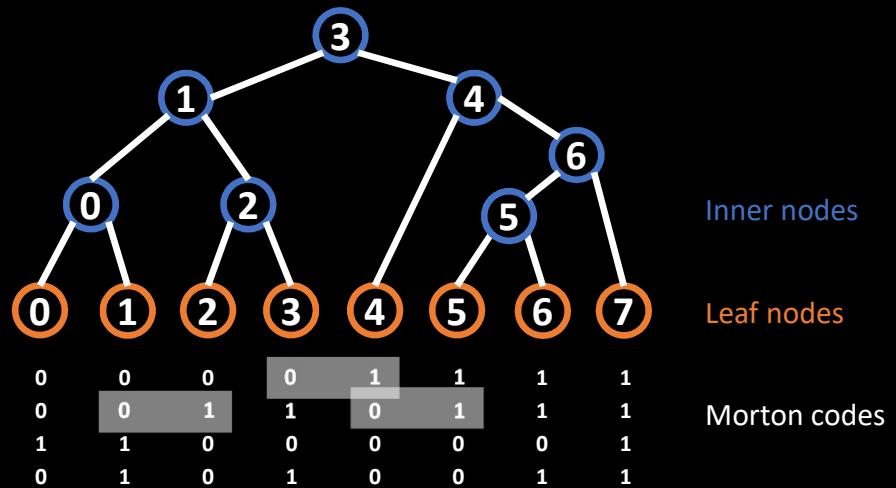
まず初めに、このツリーがどのようにできているか見てみましょう。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



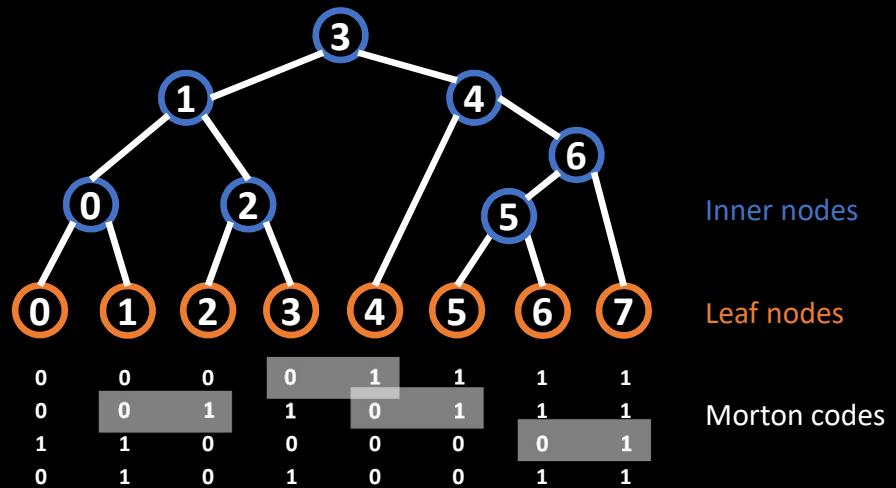
ルートは最上位ビットが異なるところで全体を2グループに分割しています。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



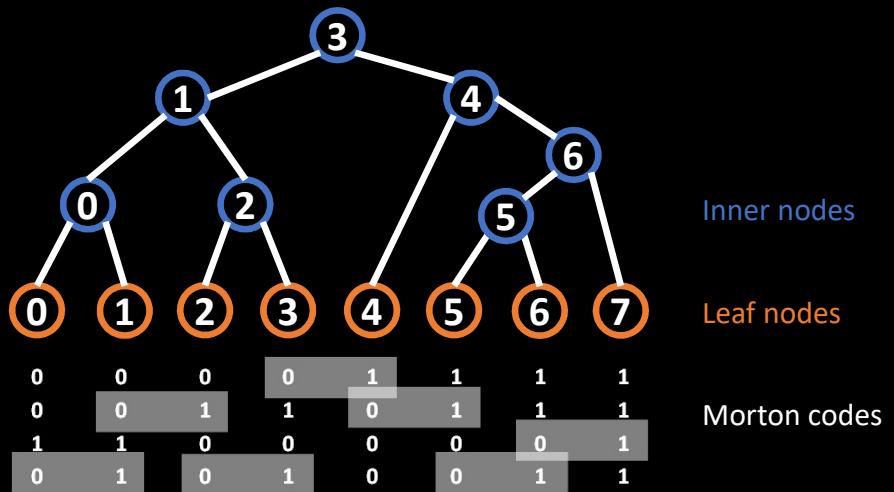
ルートで分けられた2つのグループはそれぞれ2番目のビットが異なる場所でさらに分割されているのが分かります。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



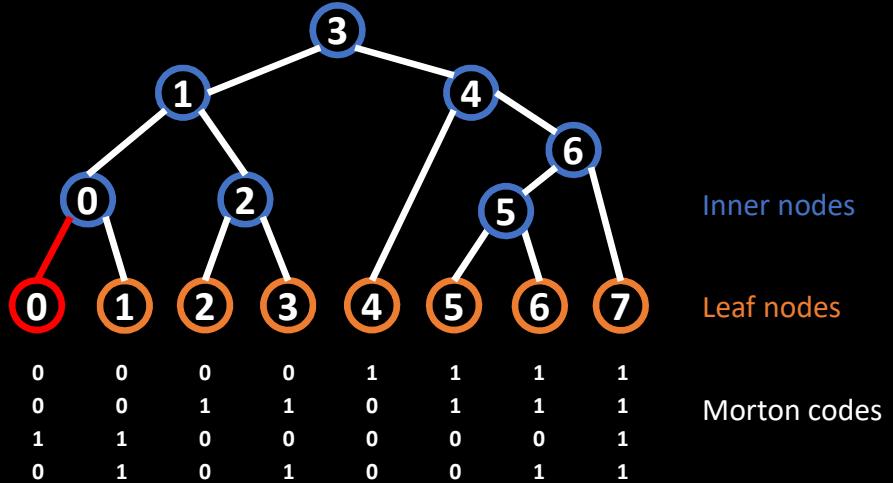
6番のノードは3番目のビットが異なるところにあり、

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



同じように0番,2番,5番のノードは最下位のビットが異なるところにあります。

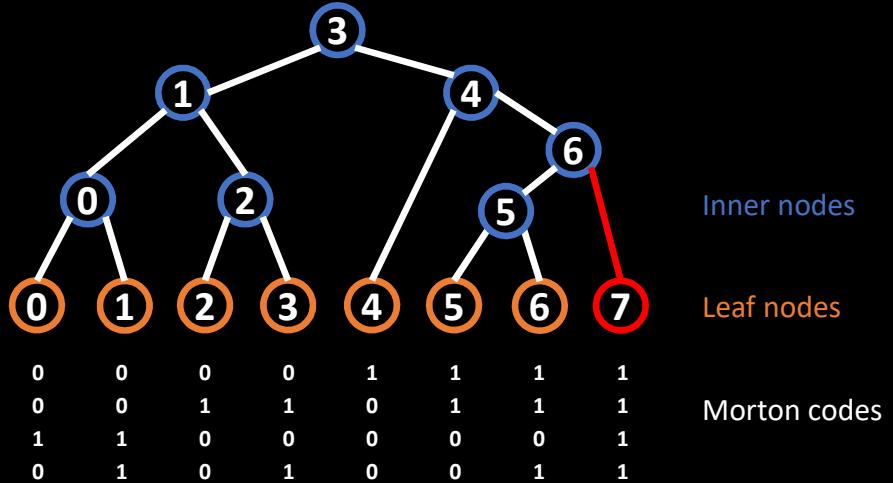
Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



以上を踏まえて、構築の仕方に話をうつします。構築はそれぞれのリーフノードから、ルートに向かって処理を進めます。

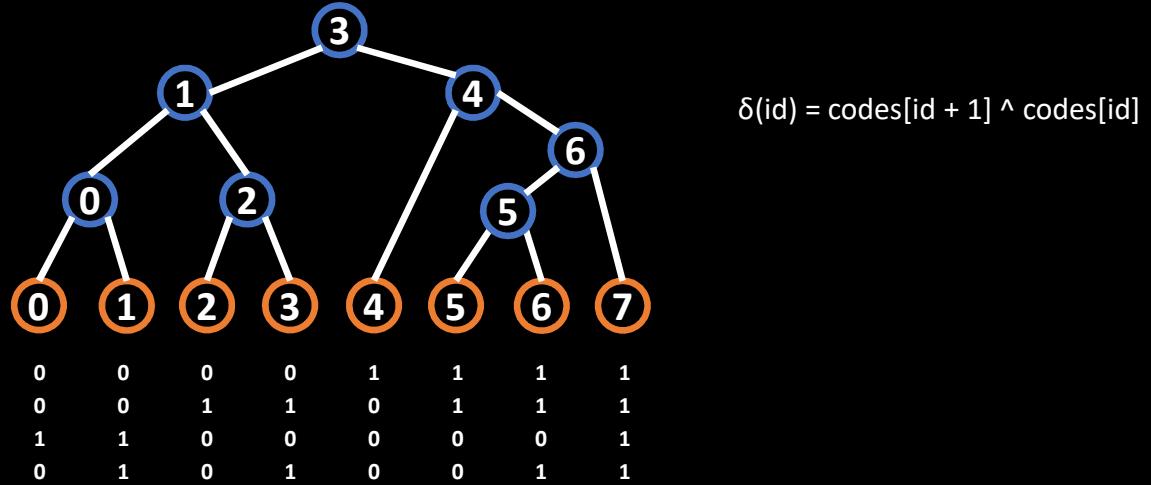
手始めにリーフ0に注目します。このときは必ず右にあるインナーノードが親になります。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



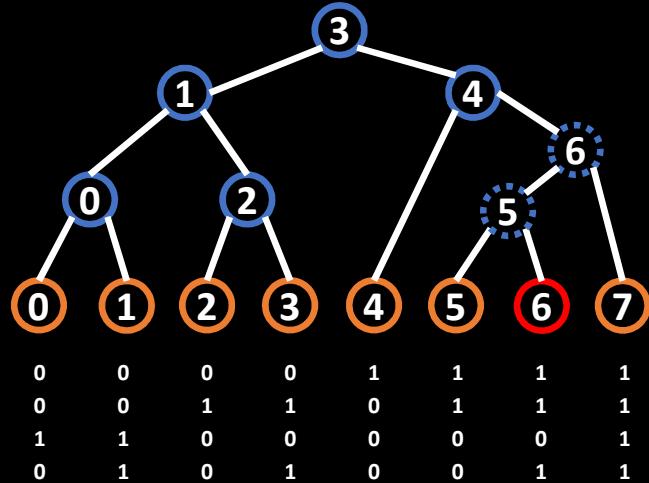
最後のリーフは必ず左にあるインナーノードが親になります。これも右端にあるので当然ですね。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



それ以外のリーフノードについてですが、右か左どちらが親なのかを決めるために、関数 δ を使います。これは今のノードとその右隣のノードのモートンコードのXORをとったものになっていて、より上位のビットが異なっているほど値は大きくなります。完全に一致するときXORなので当然0になります。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



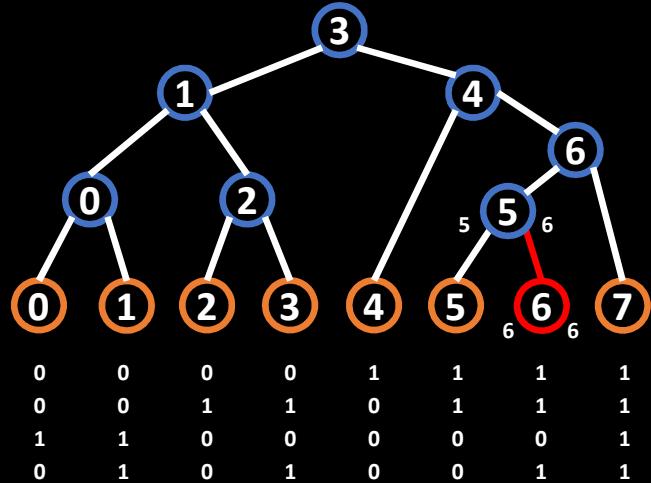
$$\delta(id) = \text{codes}[id + 1] \wedge \text{codes}[id]$$

$$\begin{aligned}\delta(6-1) &= \text{codes}[5 + 1] \wedge \text{codes}[5] \\ &= 1100 \wedge 1101 \\ &= 0001\end{aligned}$$

$$\begin{aligned}\delta(6) &= \text{codes}[6 + 1] \wedge \text{codes}[6] \\ &= 1101 \wedge 1111 \\ &= 0010\end{aligned}$$

これだけではピンとこないので、具体的にリーフノード6に注目してみましょう。この親は必ずインナーノード5かインナーノード6です。どちらが親かを決めるため $\Delta(6-1)$ と $\Delta(6)$ を計算してみます。こうすると $\Delta(6-1)$ が小さいので、5が親であることが分かります。 Δ が大きい場合、隣り合うノードとは、上位ビットが異なるということであり、よりルートに近い部分で分割されている状態を表します。したがって、親を決めるには Δ が小さい方を選択します。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



$$\delta(id) = \text{codes}[id + 1] \wedge \text{codes}[id]$$

$$\begin{aligned}\delta(6-1) &= \text{codes}[5 + 1] \wedge \text{codes}[5] \\ &= 1100 \wedge 1101 \\ &= 0001\end{aligned}$$

$$\begin{aligned}\delta(6) &= \text{codes}[6 + 1] \wedge \text{codes}[6] \\ &= 1101 \wedge 1111 \\ &= 0010\end{aligned}$$

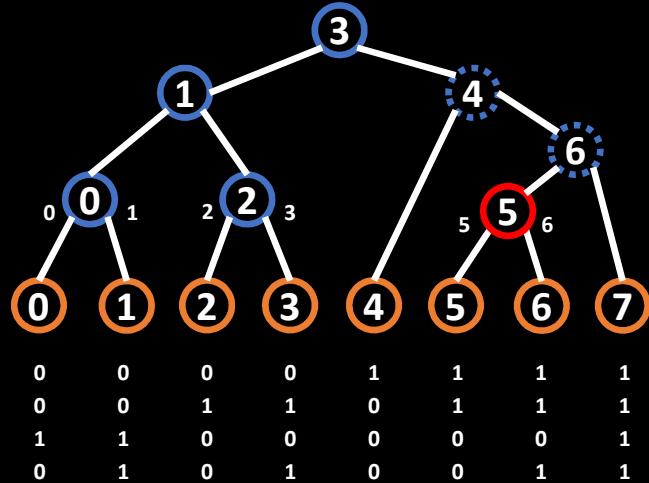
ノード6は親に自身の値を渡します。これは5のノードがカバーする範囲を決めるのに使われます。

同様にリーフノード5の親はインナーノード5なので、リーフノード5は親に自身の値を渡します。

インナーノード5がカバーする範囲は5~6となります。

また、値を渡すのと同時に親のAABBを自身のAABBを含むよう拡大します。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



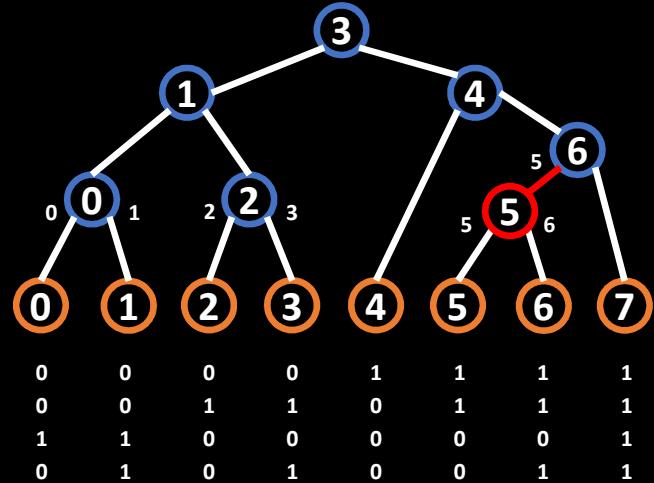
$$\delta(id) = \text{codes}[id + 1] \wedge \text{codes}[id]$$

$$\begin{aligned} \delta(5-1) &= \text{codes}[4 + 1] \wedge \text{codes}[4] \\ &= 1100 \wedge 1001 \\ &= 0100 \end{aligned}$$

$$\begin{aligned} \delta(6) &= \text{codes}[6 + 1] \wedge \text{codes}[6] \\ &= 1101 \wedge 1111 \\ &= 0010 \end{aligned}$$

リーフノードは簡単だったので、次にインナーノードを見てみます。5のノードに注目してみましょう。この親は必ずインナーノード4かインナーノード6です。どちらが親かを決めるため $\Delta(5-1)$ と $\Delta(6)$ を計算してみます。この5-1と6という数字はインナーノード5がカバーしている範囲によって決まります。5から6がカバーされている場合、5 - 1の4と6が親になる可能性があります。先ほどと同様に δ を計算してみます。そうすると $\delta(6)$ が小さいので6が親になります。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



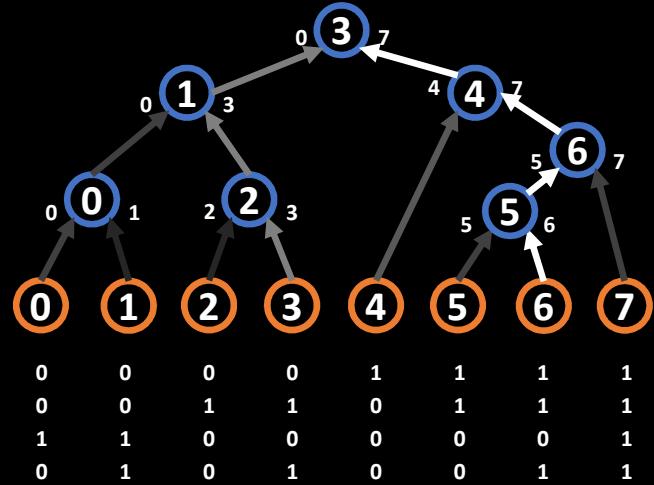
$$\delta(id) = \text{codes}[id + 1] \wedge \text{codes}[id]$$

$$\begin{aligned}\delta(4) &= \text{codes}[5 + 1] \wedge \text{codes}[5] \\ &= 1100 \wedge 1101 \\ &= 0100\end{aligned}$$

$$\begin{aligned}\delta(6) &= \text{codes}[6 + 1] \wedge \text{codes}[6] \\ &= 1101 \wedge 1111 \\ &= 0010\end{aligned}$$

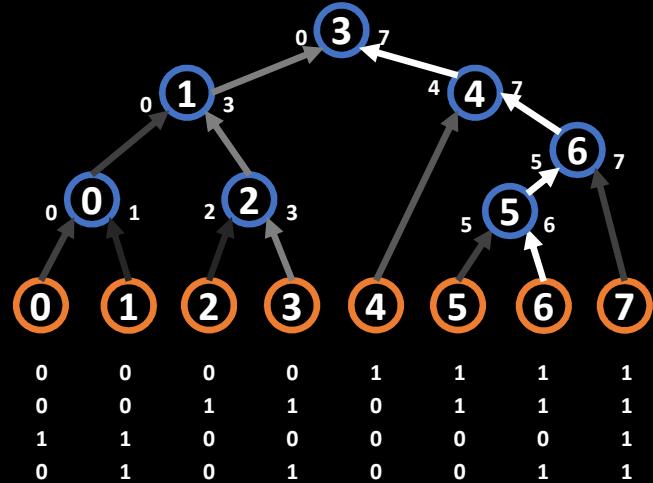
リーフノードの時と同じように親に自身がカバーする範囲の端点、この場合は下限を渡し、親のAABBを自身のAABBが含まれるよう拡大します。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



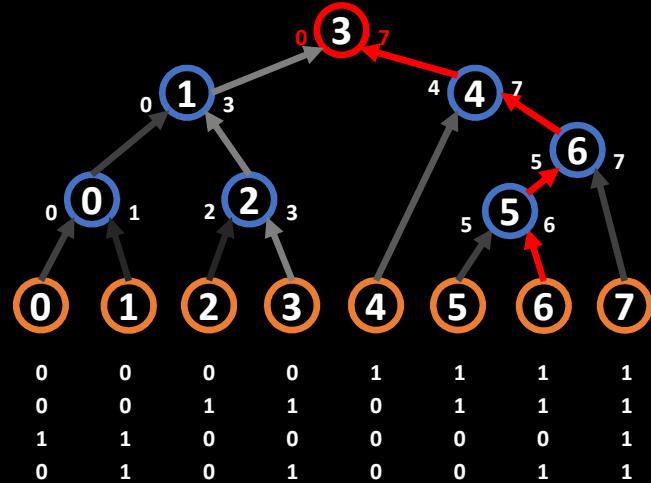
実際に並列で構築していくときは、各スレッドがそれぞれのリーフノードから、ルートに向かって処理を進めます。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



リーフ0から始まったスレッドとリーフ1から始まったスレッドはともにインナーノード0に向かいます。仮にリーフ1から始まったスレッドが先にインナーノード0に到着したとします。すると、インナーノード0に範囲を渡し、インナーノードのAABBを拡張し、終了します。このように、先着スレッドは親に自身がカバーする範囲の最小値あるいは最大値を渡し、親のAABBを拡大して終了し、別のリーフの処理にかかります。したがって各ノードは同時に2つのスレッドが訪れる可能性があります。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction



```
struct AABB
{
    union
    {
        std::atomic<uint32_t> AtomicMin[3];
        glm::vec3 Min;
    };
    union
    {
        std::atomic<uint32_t> AtomicMax[3];
        glm::vec3 Max;
    };
};
```

ルートに2つ目のスレッドが到達したら、ルートノードが0から7までの全範囲をカバーした状態になり、ツリーの構築が終了します。

AABBのデータ構造について少し触れておきます。C++は違う型のunionが許されないので(http://www.pbr-book.org/3ed-2018/Shapes/Managing_Rounding_Error.html)による)、VS2017ではこの記述で問題なく動作するので、私はこう実装しました。よりよい方法をご存じの方は後でこっそり教えてください。AABBを複数スレッドが同時に拡張することがありますが、その時はfloatの値をuint32_tにキャストしてcompare_exchange_weakを使います。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction

- Pros

- Very simple
- Freedom to choose $\delta()$

- Cons

- ID of root node is not 0
 - It's okay, though...
- Each node has two child pointers
 - Child nodes are not consecutive
- # of Nodes is “N – 1”
 - Predictable but AABBs consume a lot of memory
 - Build tree over chunks of triangles

```
// not ideal
struct Node
{
    AABB aabb;
    uint32_t left, right;
};

// preferable
struct Node
{
    AABB aabb;
    // left = index + 0
    // right = index + 1
    uint32_t index;
};
```

以上のように簡単に完全に並列でBVHが構築できる素晴らしいアルゴリズムです。論文中にも書かれていますが、関数デルタは実は任意のものを使用することができるため、LBVHでありながらSAHを小さくするようなものにすることも可能性としては考えられます。後続の研究が出てくるのが楽しみです。

さて、素晴らしい方法なんですが、完璧ではありません。論文の方法を「そのまま」実装した場合、少し不便な点がいくつかあります。まずはルートのノードIDが0ではないこと。次に各ノードが2つの子へのリファレンスを持つこと。子ノードが隣接するならば、リファレンスは一つで済むので、メモリが無駄になります。

また、N個のリーフがある場合、インナーノードの数はN-1となり、数の予測はしやすいのですが、メモリの消費が問題となります。一般的にリーフには数個のプリミティブをまとめたほうがトラバースの効率が良いので、構築のアルゴリズムを工夫してインデックス用のメモリなどを少し減らしても、AABBが一番メモリを消費するために、根本的な解決にはなりません。いくつかのプリミティブのまとまりをリーフとみなしつリーを構築することでこの問題は避けられますが、より良い「まとまり」をつくるには最近傍ペアを用いるなど工夫が必要です。

Construction: Bottom-up Fast and Simple Agglomerative LBVH Construction

- Example code:
 - https://github.com/shinjigaki/bvh/blob/master/bvh_binary.h
 - https://github.com/shinjigaki/bvh/blob/master/bvh_binary.cpp
- Open problem:
 - How to directly build wide BVH in an agglomerative manner?

サンプルコードを用意しました。実装に少しメモリの消費に無駄がありますが、どのようにC++で記述できるか参考になれば幸いです。

また、Mortonコードをつかって、Wide BVHを直接構築する方法が知っている限り提案されていないので、チャレンジしてみるのも面白いかもしれません。

Optimization

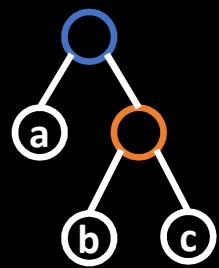
Rotation/Restructuring/Reinsertion/Reordering/Re-braiding

Contraction

Leaf Node Merging

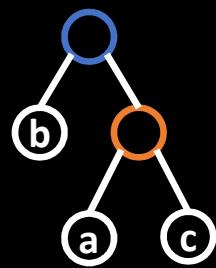
次は最適化のテクニックをいくつか紹介します。

Rotation



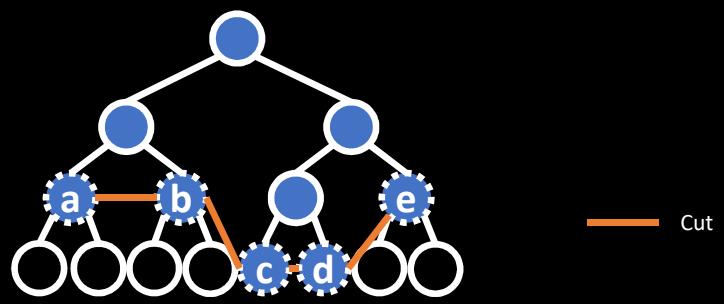
まず、回転です。回転はトポロジーを変更する最小の操作です。SAHが小さくなるよう回転させ、ツリーの品質を改善します。これが、

Rotation



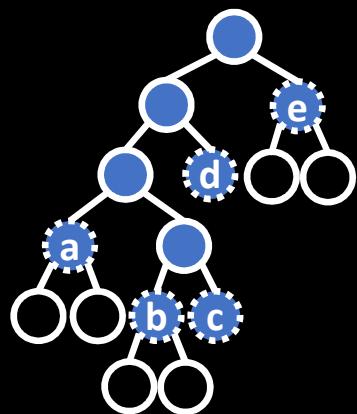
こうなります。ただ処理が局所的なので、回転をツリーのいたるところで行っても、BVH全体として品質が改善されるとは限らない、という点を覚えておいてください。入力のメッシュがある程度細かくテッセレーションされていた場合は、ノード同士の一バーラップを増やしてしまう可能性もあります。

Restructuring



回転より一般的な物に部分木の再構築を行う方法があります。N個のノードを含むツリーレットを選び、その末端部分をループとみなしてツリーレットを構築しなおします。その際、BVH構築の方法は何でも使用することができます。例えば、オレンジの線で示した部分でツリーをカットし、このツリーレットを組み替えて

Restructuring

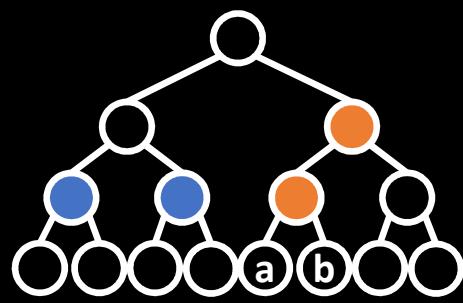


このようにした場合、ノードの数には変化がないことが分かります。ただし、ツリーが想定よりずっと深くなってしまうといったことが起こります。

このアルゴリズムを並列化する際には、ツリーレットのオーバーラップを考慮したり、一番効果がある組み換えを優先して採用したりと、実装が複雑になります。

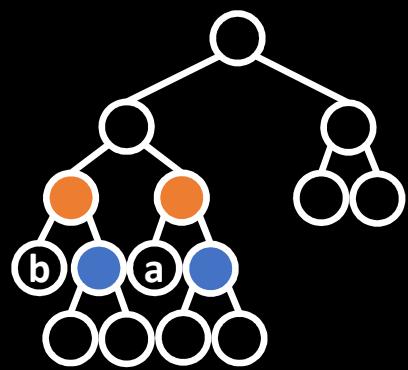
距離行列をつかって近似のAgglomerative Clusteringを行ってRestructuringを行う方法が高速で簡単に実装できでお勧めですが、既存手法はどれも特許がとられています。

Reinsertion



Reinsertionはノードを引き抜いて、SAHが小さくなる位置を探し、そこに挿入する手法です。例えば、このBVHからノードaとbを引き抜いてほかの位置に移すことを考えます。aとbを引き抜くとオレンジのノード2つは当然ながら存在価値を失います。この2つの空きノードを取っておいて、

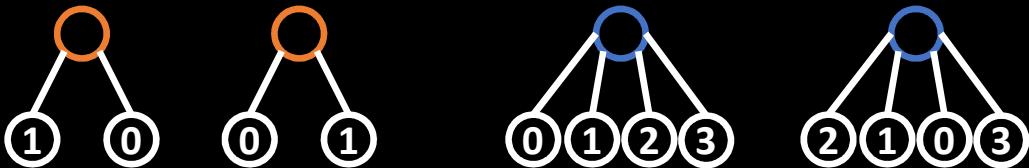
Reinsertion



挿入時にはこのように使用します。本来は逐次的に処理を行う方法ですが、並列化バージョンも存在し、実は並列化したほうがBVHの品質が上がります。これは並列化した場合の方が最適化問題を解くにあたって、より広い解の領域を探索できるようになるから、と考えることができます。

Reordering

- Change the order of child nodes
- Useful for occlusion tests
- The topology of BVH remains untouched
- Difference from sophisticated traversal order is that reordering has no additional overhead at runtime
- Code example: <http://jcgt.org/published/0005/02/02/code.zip>



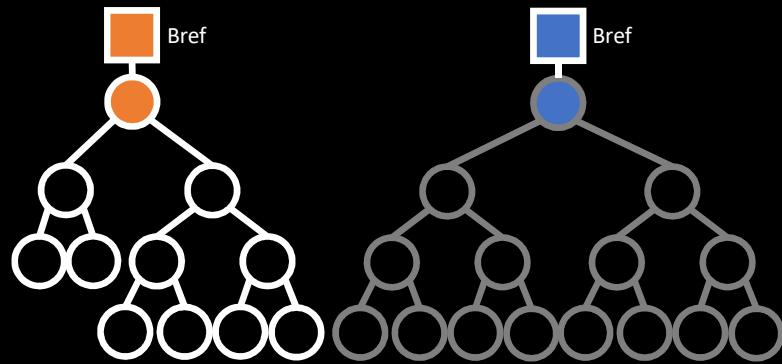
並べ替えを使うと子ノードの順序を変えることで、オクルージョンの計算を高速化することができます。並べ替えにはなんらかのコスト関数を用いますが、オクルージョンの場合は、基本的にレイを多くブロックするものを前に持ってくる、という操作を行います。このときツリーのトポロジーは一切変化しないので、First-Hitトラバーサルのパフォーマンスは変化しない、という面白い性質があります。複雑なトラバーサルを行うアルゴリズムとの違いは、ソートは基本的に一回だけ行われるので、実行時にオーバーヘッドが発生しないということです。

ソースコードはここにあります。Wide BVHのトラバースも含まれていますが、当時のコードはそれほど最適化されていませんので、ご了承ください。

<http://jcgt.org/published/0005/02/02/>

Re-Braiding

- Two-level BVH
 - Build BVH for each object

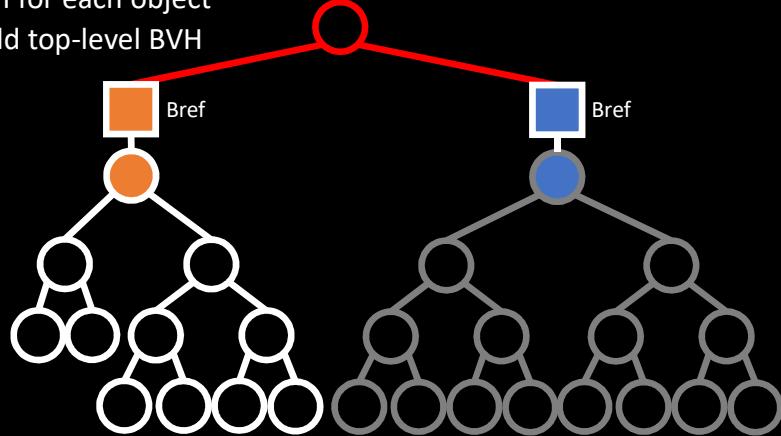


次はRe-braidingを紹介します。これは非常に実用的なアルゴリズムです。

さて、インタラクティブなシーンを扱うためには、2段階のBVHがよく用いられます。2段階のBVHを使う場合、まず各オブジェクトごとにBVHを構築します。

Re-Braiding

- Two-level BVH
 - Build BVH for each object
 - Then build top-level BVH

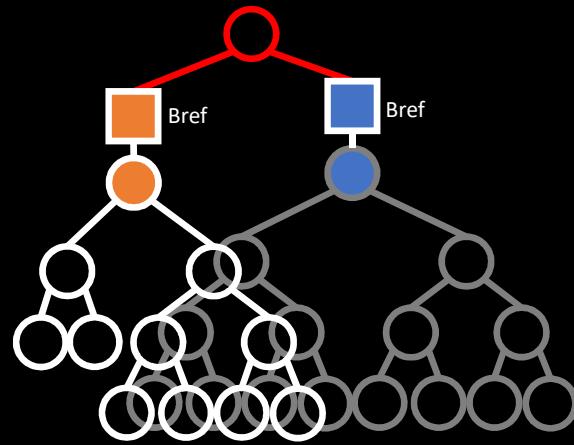


その後、構築された各BVHのルートをリーフとみなして、さらに上位の BVH、top level BVHを構築します。こうすることによって、変化があったオブジェクトのBVHと上位のBVHだけを再構築すればよくなり、すべてを再構築した場合に比べると応答が速くなります。変化したオブジェクトが複数ある場合は、オブジェクト単位で並列化できるので、実装をシンプルにしておくこともできます。

ここにあるBrefというのはBVH node build referenceの略で、文字通りBVH のノードへのリファレンスを表しています。

Re-Braiding

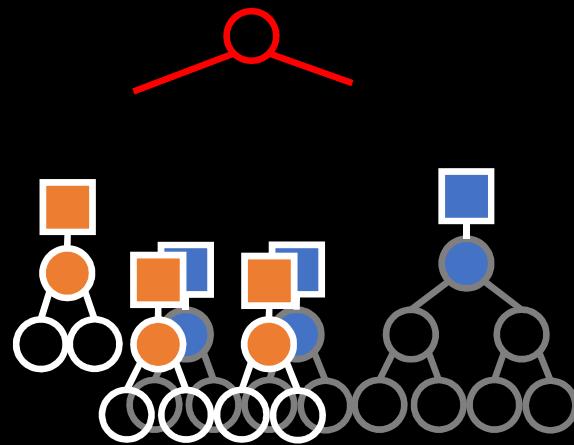
- Performance degrades when objects overlap



ただし、オブジェクト同士が重なってしまった場合、レイトレーシングのパフォーマンスが落ちてしまうという致命的な問題があります。この図のような状態は明らかによくありません。

Re-Braiding

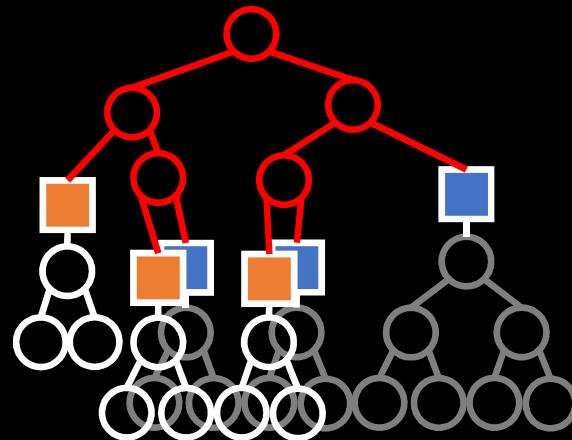
- Open Brefs



Re-braidingはそのような状況を改善するため、重なりが大きいノードを展開します。

Re-Braiding

- Improve two-level BVH



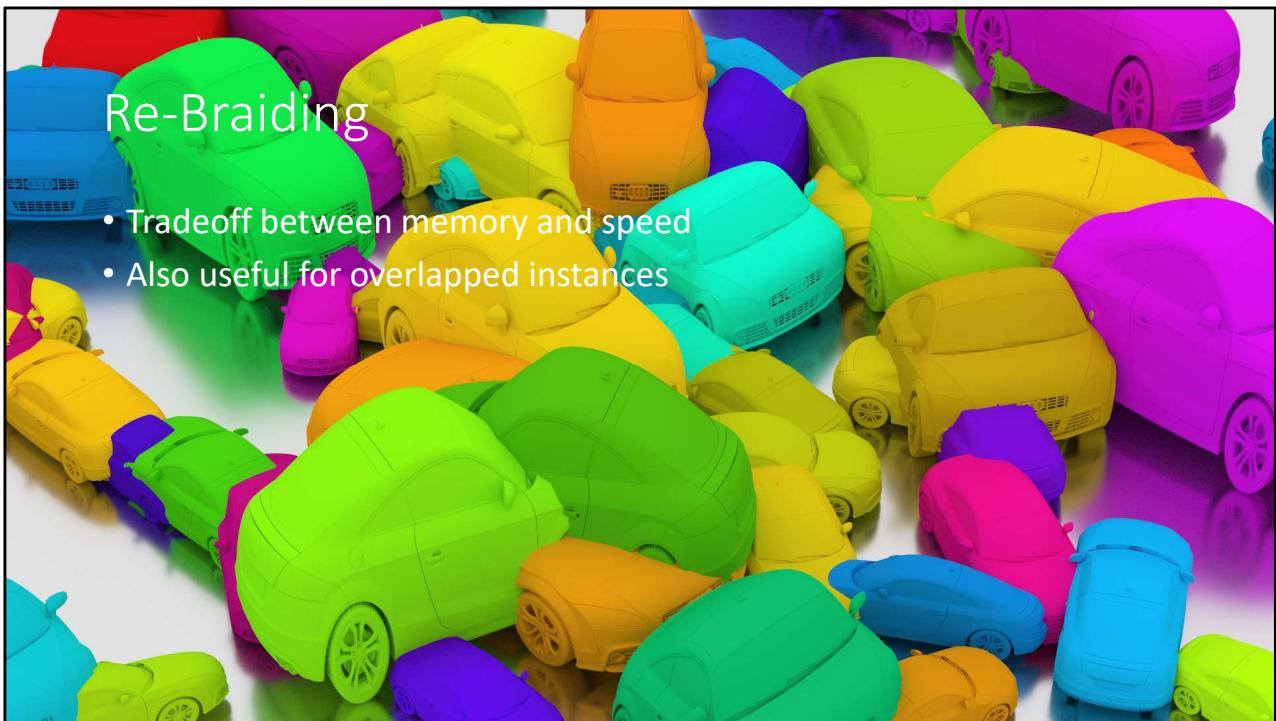
そしてtop level BVHの品質が改善するよう、Brefsを組み替えます。注意したいのは各オブジェクトごとに構築したBVHには手を入れないということです。Bonsaiという発想の良く似たアルゴリズムがあり、そちらでは剪定されたルート付近のいらない部分が捨てられてしまいますが、Re-braidingはあくまでも補助のロープでつなぎ変えを行うという点で異なります。

メモリと計算速度はトレードオフの関係にあります。Wide BVHだとBrefsが大量に生成される可能性があるので、あまり多くのノードを開くことはお勧めしませんが、開かれるノードは他のオブジェクトとオーバーラップするものだけであるので、そこまで心配する必要もありません。実際は、Wide BVHとくに幅が8や16等と広い場合、ルートノードの直下だけつなぎ変えてやるだけでも速度が改善しますのでそのように割り切っても良いかと思います。また、同じことはBrefを使わずトラバースのコードに手を入れることで実現できますが、その場合レイが複数のノードに同時に侵入することになり、コードが複雑化してしまいます。



当然ながら、Re-braidingは、インスタンスにも使用することが出来ます。余談ですが、2段階のBVHで十分だという記述をしている論文がありますが、私はこれには反対です。マルチレベルインスタンシングの方が、人工物また木など圧倒的に少ないメモリでレンダリングすることができます。

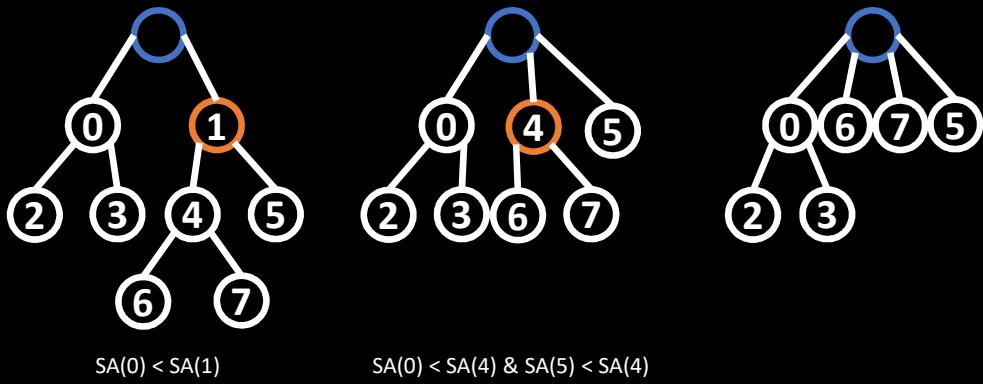
このシーンではWide BVHを使っていますがルートノードだけ開いた場合、15%程度高速になりました。



また、少し話はそれますが、階層の深さをうまく利用すれば、このように簡単に色のバリエーションを加えることが出来ます。

Contraction

- Surface-Area Guided Contraction
- Ray-Distribution Guided Contraction

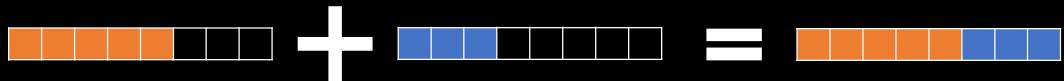


Re-なんとかというのが4つほど続きました。

さて、コントラクションというのは、2分木をWide BVHに変換することをいいます。Collapseと呼ばれることもあります。通常はContractionを行う際に、表面積の大きいノードから引っ張り上げてCollapseさせます。これをSurface Area Guided Contractionといいます。実際にはBinary BVHを構築してからContractionするのではなくて直接Wide BVHを構築します。これは余計な計算時間や、重複したメモリの使用を避けるためです。しかし、明示的にこのプロセスを独立させておくと、Ray-Distribution Guided Contractionのように面白いアルゴリズムを使うことが出来ます。この方法は、まずBinary BVHを構築し、ある程度のレイを使ってレンダリングを行います。その際、ノードに訪れたレイの数を記録しておきます。これらのレイはRepresentative Raysといいます。次にContractionを行いますが、そのときSurface Areaが大きいノードではなく、たくさんのレイが当たったノードから優先にCollapseしていきます。そうすることで大幅な高速化を達成できます。

Leaf Node Merging

- Wide BVH leaf nodes often have empty slots
- Merging multiple nodes reduces memory consumption
- Each wide BVH node needs masks
- From 5 to 10% speedup in my experience (LPDDR3@1867MHz)



Wide BVHのリーフノードには空きスロットが発生することが多々あります。リーフノードが4や8の倍数となりやすいよう修正したSAHを使用しても完全に防ぐことはできません。空きがあるノード同士はマージすることで、メモリの消費を大きく抑えることが出来ます。LPDDR3の1867MHzを使ってているこのラップトップで試した所、マージ有りでは5-10%ほどレンダリングが高速になりました。メモリが遅い環境ほど、効果が高いのではないかと思います。Wide BVHは空きができるから使えないよ、ということはなくなります。
(<https://github.com/shinjiogaki/about/blob/master/Fragmentation-Aware%20BVH%20Construction.pdf>)

Many Lights

Adaptive Tree Splitting

Stochastic Lightcuts

Stochastic Light Culling

最後に最近流行っているメニーライトについて少し触れたいと思います。実験するなどして気づいたことをざっと並べたものなので、参考程度に聞いてください。

Many-Lights

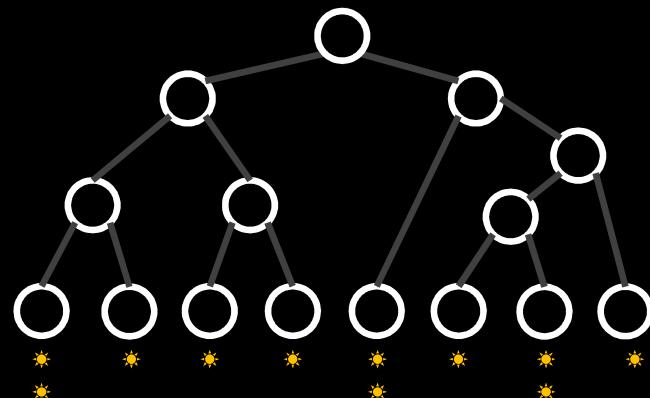
- Recent techniques use BVH
 - Adaptive Tree Splitting
 - Importance Sampling of Many Lights with Adaptive Tree Splitting
 - Importance Sampling of Many Lights on the GPU
 - Dynamic Many-Light Sampling for Real-Time Ray Tracing
 - Adaptive BRDF-Oriented Multiple Importance Sampling of Many Lights
 - Lightcuts
 - Stochastic Lightcuts
 - Light Culling
 - Stochastic Light Culling
 - Hierarchical Russian Roulette for Vertex Connections
 - Blue-Noise Dithered QMC Hierarchical Russian Roulette

どうしてBVHの話なのにメニーライトなのかと思うかもしれません、最近メニーライトを扱った論文ではBVHが使用されています。ですので、それらの論文で紹介された手法にはBVHの最適化をそのまま生かすことが出来ます。Dynamic Many-Light Sampling for Real-Time Ray Tracingがその典型例で、2 レベルのBVHを用いてAdaptive Tree Splittingをダイナミックなシーンで利用可能にしています。

それから、Stochastic Lightcutsはバイアスドな方法ですが、提案されているメトリックはAdaptive Tree Splittingでもそのまま使用可能なものなので一読の価値あります。

Stochastic Light CullingはCEDECでも過去に徳吉さんと原田さんによるセッションがあったと思いますので、ここに来てくださった皆さんは詳しいかもしません。

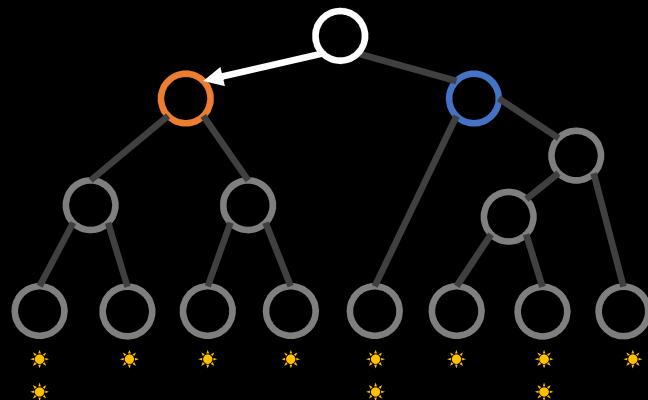
Adaptive Tree Splitting



まずAdaptive Tree Splittingから紹介します。

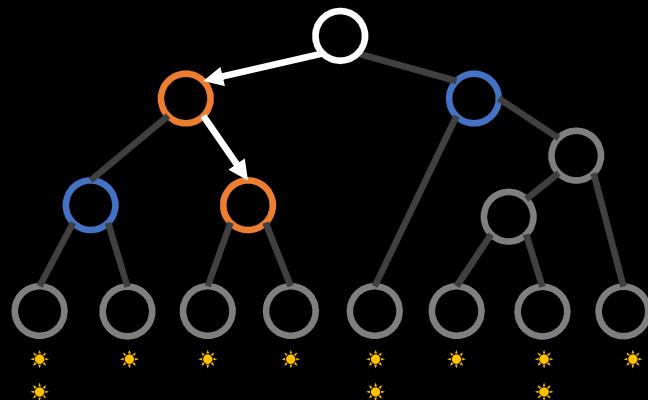
膨大な数のライトがあったとき、各シェーディングポイントでそれら全ての影響を計算することはできません。ですので、基本的には確率的にどれか一つ、あるいは、数個を選ぶようにして計算を端折ります。もちろん、完全にランダムに選んだのではノイズが多くなってしまうので、なるべく寄与の大きいライトを選ぶようにするのですが、その際BVHを利用します。各リーフノードにはご覧のようにライトが含まれます。

Adaptive Tree Splitting



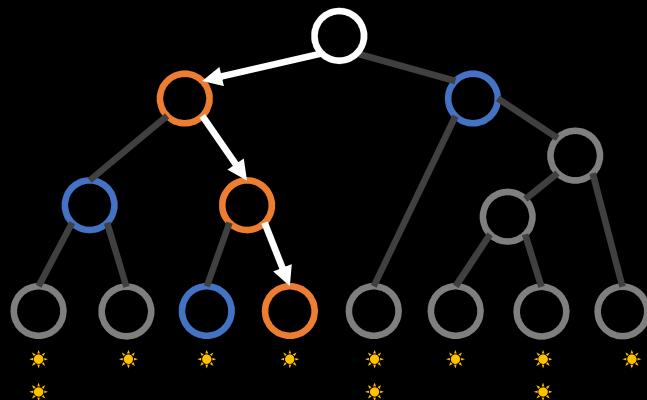
ライトのサンプリングはBVHをトラバースしながら行います。ふわっとした説明になってしまいますが、子ノードを大きなライトとみなし、際遮を無視した明るさと幾何項を用い、それぞれの寄与を計算し、どちらの子ノードをたどるかを確率的に決定します。

Adaptive Tree Splitting



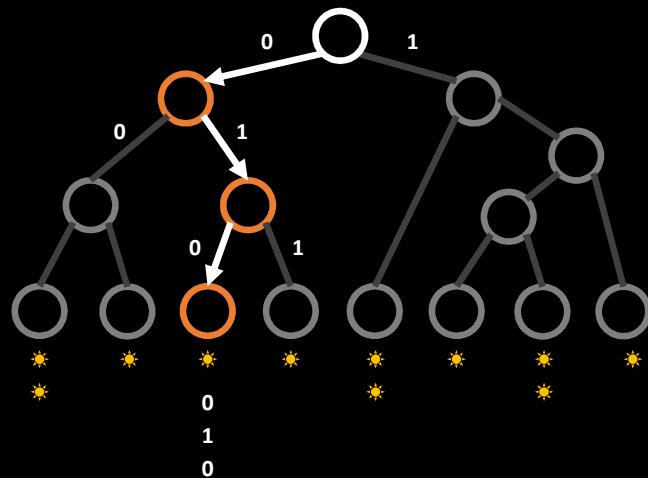
順々にたどっていって

Adaptive Tree Splitting



リーフノードまでたどり着くと、そこに含まれるライトの寄与を計算します。各ノードを訪れるたび子ノードが選ばれる確率をかけていくことで、選ばれたライトのPDFが求まります。

Adaptive Tree Splitting



Multiple Importance Samplingを使用する場合、他のサンプリング手法で生成されたレイが当たったライトのPDFを計算する必要があります。

BVHを作る段階で、各リーフノードに、左の子を0右の子を1とした、ビット列を入れておくと、サンプリングの場合と同じ要領でPDFを計算することが出来ます。

Adaptive Tree Splitting

- Cost metric: Surface Area Orientation Heuristic (SAOH)

$$C_{\text{split}}(\text{axis}, \text{pos}) = \frac{E_L M_A(L) M_\Omega(L) + E_R M_A(R) M_\Omega(R)}{M_A M_\Omega}$$

- Conservative geometric term



BVHはSAOHというメトリックを用いて構築します。SAHとそれほど変わりませんが、面の向きが近いものをグルーピングすという点が異なります。当然ながら、光源がシェーディングポイントを向いていないと意味がないので、そういういたジオメトリをまとめて棄却するために面の向きをコスト関数に取り入れます。

この式のEは光源の明るさ、MAはAABBの表面積であり、M ω はOrientation Bounds Area Measureとよばれ光源の向きと光の広がり具合を表したものになっています。M ω を計算する過程でコーンをまとめる処理などが入ってきます。ライトの向きは逆向きの場合もあり、当然単純に平均をとることが出来ません。このコーンをまとめる処理は実装は大変ではありませんがやや面倒です。

代わりに、入力された面を最初に向きによってグループ分けしても良いかと思います。AVXを使ってOBVHを構築する場合にはルートノードで面を8方向 (+-X, +-Y, +-Z)に分類することができます。結果は割と良好だったので、実装をシンプルにしたい人は試してみてください。

幾何項はトラバースの段階ではノードのAABBを使って計算する必要があるの

で、AABBと同じ程度の大きな球状のライトを想定するなどして、保守的な計算を行いますが、このとき使われるGeometric TermをConservative Geometric Termと呼びます。

Adaptive Tree Splitting

- Wide BVH improves
 - Performance
 - SIMD utilization
 - Less memory
 - Accuracy as well? -> Yes, but not always
 - SAOH is inaccurate for nodes close to the root
 - possible to choose a node from more than two

Wide BVHの使用は大幅に速度を改善します。これはメモリの節約、アクセスパターンによるもの、それからSIMDの使用による影響も大きいですが、それ以外にもWide BVHは理論的に誤差を減らす副作用的な効果をもたらすと考えられます。まず、ルートノードの近くは保守的な幾何項による近似の誤差が大きくなる傾向があります。したがって、Wide BVHを使うということはルートに近い大きな誤差を含んだノードをある程度除去する役割を持ちます。また、複数の子ノードから一番良いものを選ぶことが出来るため、影響の大きい子ノードをより効率よく選択出来るようになります。（この選択部分にSIMDを利用します。）ただ、遮蔽を考慮していないので、どんなシーンでも絶対によくなるというわけではありません。



少し比較を見てみましょう。これは完全にランダムに光源をサンプリングした場合。FullHD 32SPPでレンダリングは19秒。

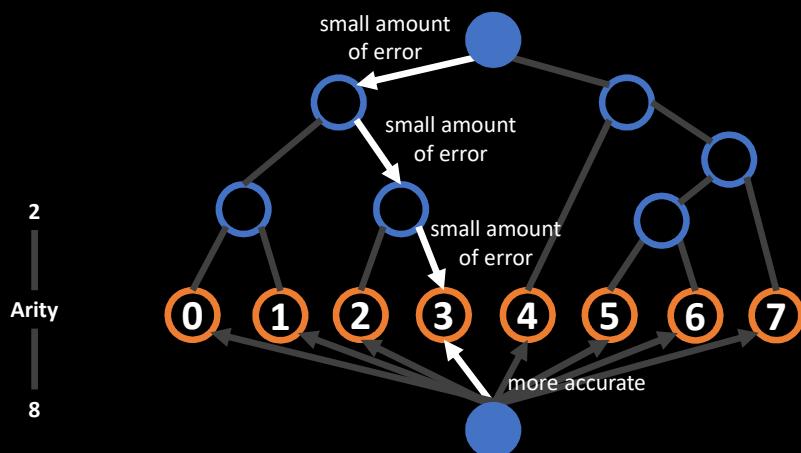


Binary BVHを使った場合。32SPPでレンダリングは31秒。



Wide BVHを使った場合。32SPPでレンダリングは22秒。Binary BVHより高速でノイズが大きく減った場所があることが分かります。逆に少し増えているように見える部分もありますが、全体としてはこちらの方がきれいになっています。

Adaptive Tree Splitting



上側は二分探索、下は8個のノードから直接1つを選びます。各中間ノードは誤差を含むため、Wide BVHの方がより正確な結果をもたらすと考えられます。

Stochastic Lightcuts

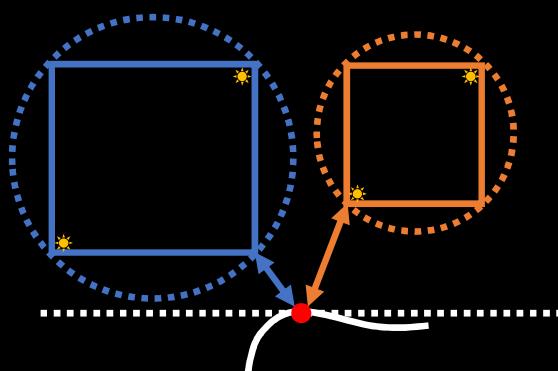
- Similar to Adaptive Tree Splitting
- Converges quicker than the state-of-the-art algorithms
- Modified geometric term (even more conservative)

Stochastic LightcutsはAdaptive Tree Splittingと非常に似ていますのでさらっと紹介します。

確率的にライトを含むBVHのどちらのノードを訪れるかを決めるため、クラシックなLightcutであったような低周波のノイズが出ません。また、近似手法であり、ある程度誤差が小さくなるとトラバースを終えるため、非常に高速です。SAOHで用いられていたより、さらに保守的なGeometric Termを使用します。式はスライドに乗せなかったので、興味のある方は論文を参照してみてください。

Stochastic Lightcuts

- The modified geometric term is still not perfect

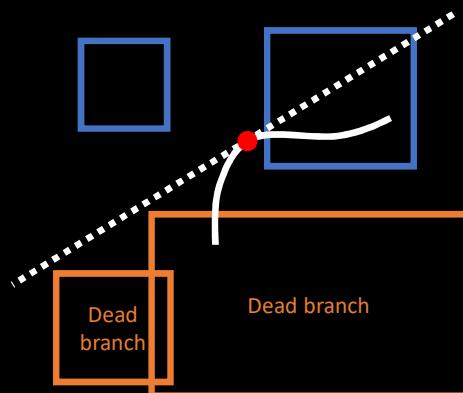


Corner case: a light source in the orange box is actually closer

先述のようにStochastic Lightcutsではより保守的な幾何項を導入していますが、それでも完全ではなく、コーナーケースが存在します。ここに示したようなライトの配置では右のノードを訪れる確率を上げたほうが良いにもかかわらず、左のノードを優先して訪れてしまいます。

Stochastic Lightcuts

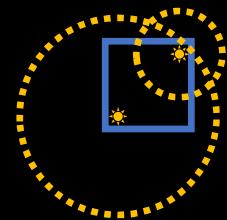
- Reject boxes if they are below the shading tangent plane



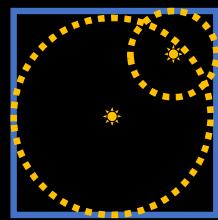
Stochastic Lightcutsの論文ではDead Branchという用語が出てきます。Dead Branchは、訪れるに何の価値もない部分木のことを指します。これは、シェーディングポイントとその法線から求まる接平面より、完全に下になっているノードで、単純な計算で見つけることが出来ます。このテクニックはいろいろな場所で使えると思います。

Hierarchical Russian Roulette

- Improvement on stochastic light culling
- Can be used for many lights



The range of each light changes stochastically



Refitting is not a good idea!

最後にHierarchical Russian Rouletteについて少し触れたいと思います。

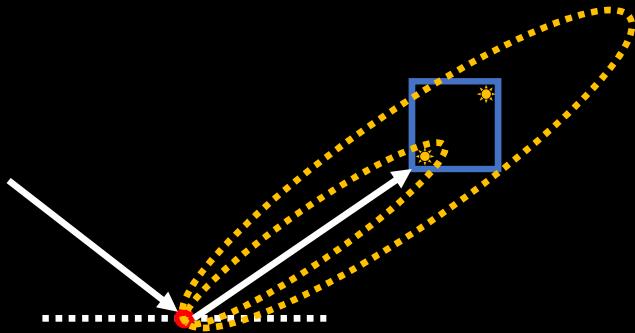
HRRはStochastic Light Cullingを改良するものと捉えられると思います。論文では、シャープなロープが使われているシーンでの品質の改善が扱われていると思いますが、使おうと思えば、一般的なMany Lightsの計算にも使用できます。

もともとのStochastic Light Cullingの論文ではBounding Sphere Treeの使用が提案されています。Stochastic Light Cullingでは確率的にライトの影響範囲を変える手法です。明るさは距離の2乗に反比例しますが、それに従うようにライトの影響範囲の球を大きくしたり小さくしたりして、シェーディングポイントがその外にあった場合はその光源からの寄与を計算しません。

高速化のため、ライトに対しBVHを構築した場合、影響範囲の変わる度にBVHをリフィットさせるわけにはいきません。また、光源の影響範囲を大きくしたり小さくしたりするだけではシェーディングポイントでのBSDFのロープの形状を反映させることができません。

Hierarchical Russian Roulette

- Change lobe size instead
- Efficient for extremely glossy reflections thanks to SEL (squared ellipsoidal lobe)



代わりに、BVHを固定し、シェーディングポイントを起点とするロープを確率的に大きくしたり、小さくしたりすることで、ライトの影響範囲を変えるのと同じ効果を得つつ、様々なロープの形状をサポートすることができます。

HRRの論文ではSquared Ellipsoidal Lobeという関数を導入していてそのお陰で、Roughnessの非常に小さい面を効率よく扱うことが出来ます。

ライト自身もIESなどを使えば複雑なロープの形状を持つので、難しいところだと思いますが、そのうち解決してくれるでしょう。

Hierarchical Russian Roulette

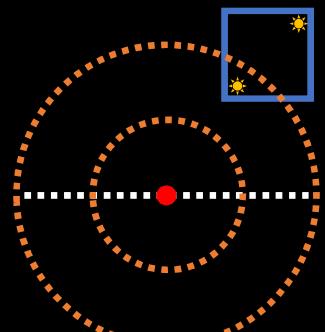
- Implicit nodes overlap



繰り返しますがHRRはGlossy Surface向けの方法で、ロープが太いといいますか、大きい場合には、ライトが密集しているとき、右の図のように陰的なノードのオーバーラップ大きくなり過ぎて、あまり実用的ではありません。この問題はAdaptive Tree Splittingの論文でも言及されています。特に解像度の高いメッシュが発光している場合などは負荷が大きくなります。

Hierarchical Russian Roulette

- For diffuse surface, don't do this:



$$AcceptanceRatio = \max\left\{1, \frac{1}{|ShadingPoint - Light|^2}\right\}$$

$$Range = \sqrt{\frac{1}{\xi}}$$

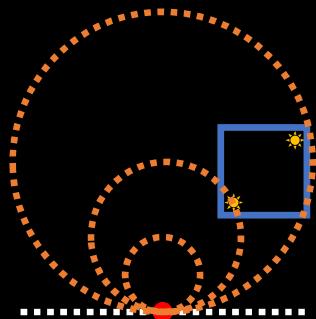
$$\xi = \mathcal{U}(0, 1)$$

さて、不向きというのは承知なのですが、ランバートの扱いを考えてみましょう。

少しでも効率を上げるためにには、単純にシェーディングポイントを中心に全方向に広がる球の半径を大きくしたり、小さくしたりするのはお勧めしません。シェーディング面の裏側のノードは極力省きたいですし、コサインの重みづけを行った方が良いからです。

Hierarchical Russian Roulette

- For diffuse surface
 - Use the lobe proposed in “Splatting Indirect Illumination” [Dachsbacher 06]
 - with dead branch culling
 - Or use sphere touching the shading tangent plane
 - Not optimal but sphere-box intersection test is easy
 - Compensate the doubly counted cosine term

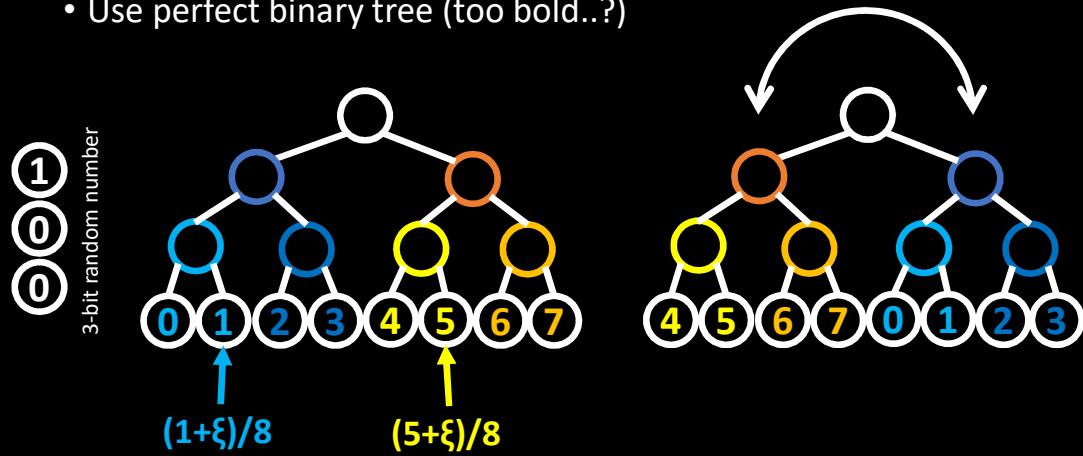


この場合のより良いローブは”Splatting Indirect Illumination”に記載されています。接平面の下に少しBounding Sphereがはみ出していますが、それは先程あったDead Branchの除去と同じ要領で避けることが出来ます。

また、単純にシェーディング接平面に接する球でも、最適なコサインの重みづけではなくなりますが、計算を行うことが出来ます。同じサンプル当たりのノイズは増えますが、接平面の下のノードを考える必要はなくなりますし、球とAABBの交差判定は非常に単純です。

Lazy Man's Hierarchical Russian Roulette

- Use perfect binary tree (too bold..?)



HRRは大変面白いアルゴリズムです。

他のアルゴリズム同様、ライトを全てシェーディングポイントとつないでしまうと計算時間がいくらあっても足りませんのでBVHを使用して、いらない部分の計算をまとめて省きます。このとき、肝になるのはサブツリーの中で一番小さい乱数の値を見つける、という処理です。詳細は論文を参照してください。

HRRは乱数が小さいとロープが大きくなるようデザインされていて、大きなロープがサブツリーと交差するときは、その瞬間、そのサブツリーが影響範囲の大きいライトを含んでいる、と言い換えることが出来ます。

私は面白くさがりで、論文で提案されたシンプルなアルゴリズムすら実装が面倒です。なので、怠け者向けのアルゴリズムを紹介したいと思います。まず、ツリーが完全二分木であるという制約を設けます。これは大胆なように思われるかもしれません、若干データを重複させたり、サンプル数を調整したりするだけでよいので、実用上そこまで問題にならないでしょう。

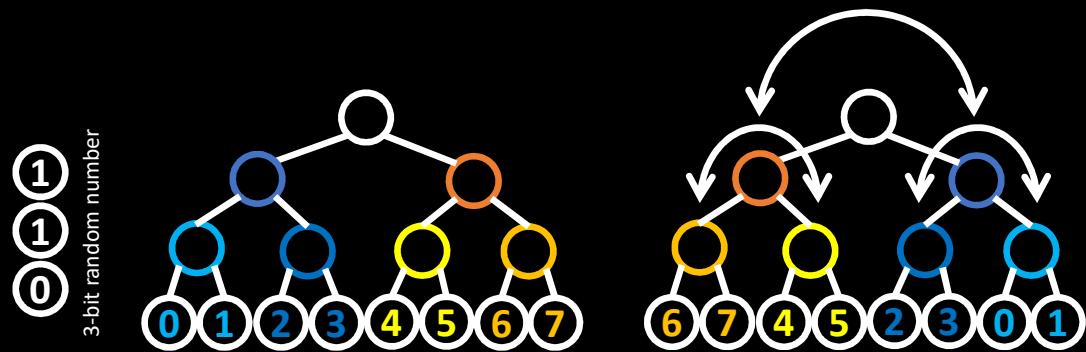
(Light Vertexの場合は数の調整がそれほど難しくありませんし、メッシュライトの場合はテクスチャが張られているとポリゴンを分割したり、表面に点

光源を生成したりすることで容易に調整が可能です。サンプル数が大きければ少し捨てたりしても結果にそれほど影響はありません。)

まず、何もしない状態ではこの左側のようにリーフに0から7までの数字がアサインされているとしましょう。実際は、アサインする数値は0から1でなければならないので、各数値に一様乱数を足して8で割ったものを割り当てます。

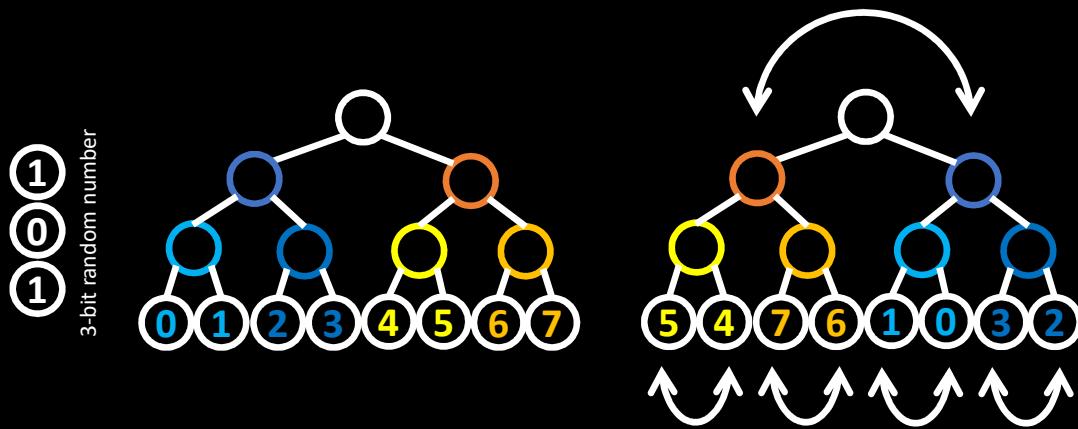
さて、割り当てた数字をランダマイズする方法ですが、ここに挙げた例では、ルートを省いたツリーの深さが3なので3ビットの乱数を使用します。例えば乱数の最上位ビットが1の時、深さ1でアサインする数値を入れ替えます。

Lazy Man's Hierarchical Russian Roulette



最上位ビットと次のビットが1の時には、深さ1と2で数値を入れ替えます。

Lazy Man's Hierarchical Russian Roulette



最上位ビットと最下位のビットが1の時には、深さ1と3で数値を入れ替えます。3ビットの乱数が均等に分布しているとき、すなわち3ビットの乱数が0~7までの値を均等にとるとき、リーフにアサインされる各数字は当然ながら同じ回数出現します。このようにすると、乱数は一つしか使用しないため、各深度で乱数を振りなおす必要がなくなります。

こんなことをして何がうれしいかというと、各サブツリーのノードでの最小値を非常に簡単に知ることが出来るようになるということです。それから、HRRの本質的な問題は「陰なノード同士の重なり」です。このような数値のアサインを行えば、例えばライトが一直線上にきれいに並んでいると、右のノードが大きくなる時、左のノードは小さくなり、「陰なノード同士の重なり」は小さくなります。実際、ライトの配置はアセット次第ですが、完全二分木の時に限らず、乱数の並び順をうまく工夫することで、「陰なノード同士の重なり」は小さくすることが出来るはずですので、最適な方法を探してみるのも面白いかと思います。

日本人発案の方法ですので、後続研究が出て、いろいろなケースで実用的になって欲しいと思います。

Conclusion

- Explained de-facto standard BVH construction algorithms and several optimization techniques
- BVH optimization techniques are directly applicable to accelerating ray tracing with many lights
- Find more info at: <https://github.com/shinjiogaki/bvh>
- Impossible to kiwameru (極める)

まだまだ盛り込みたかったのですが、今日用意したものはここまでです。

いくつか標準的なBVHの構築方法と、最適化手法を紹介しました。また最近のMany Lights向けのアルゴリズムに触れましたが、それらはBVHを用いているものばかりなので、紹介したBVHの最適化手法すぐに改善することができます。もっとほかの論文などについて知りたい方はここに紹介したリンク先に行ってみてください。

さて、紹介した技術はまだまだほんの一握りで、ハードウェアについて書かれた論文などについては一切触れることができませんでした。というわけで、「極める」という公約は守れていませんが、今日、皆さんのが何か一つでも得るものがあったなら嬉しく思います。簡単に見えるテーマでも大変な広がりをもつこと、データ構造は本当に面白いと思います。

レイトレーシングのハードウェアもレンダラーも海外製品ばかりなので日本から面白いものが出てくるとよいなと思います。



Thank you for listening!

以上です。

会場をうろうろしていますので、気になった方は話しかけてください。

ご清聴ありがとうございました。

Other Interesting Papers

- Dual-Split Trees
- RTX Beyond Ray Tracing - Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location
- Massively Parallel Construction of Radix Tree Forests for the Efficient Sampling of Discrete Probability Distributions

大変喜ばしいことに、最近になっても新しいデータ構造が提案されています。Dual Split Treesは先ほど述べたように、親と子のバウンディングボックスが重複した情報を持っていることに着目し、消費メモリとレイと平面の交差判定回数を減らす方法ですが、残念ながらWide BVHには使用できません。そもそもWide BVHは重複した部分が少ないというのもあり、どの程度発展するかは未知数です。また、Dynamicなシーンにも適用しづらいです。

2つ目はレイトレーシング以外のタスクにRTXを使用しよう、というもので、ある点がどの四面体に入っているか特定する、という問題を扱っています。GPGPUがはやった時のように言葉が適切かは分かりませんが、汎用目的RT Coresのような論文が今後増えてくるのではないかでしょうか。

最後は私は知らないで、徳吉さんに教えていただいた論文ですが、エイリアス法の改善、フォレストの効率的な構築方法などが書かれた論文です。