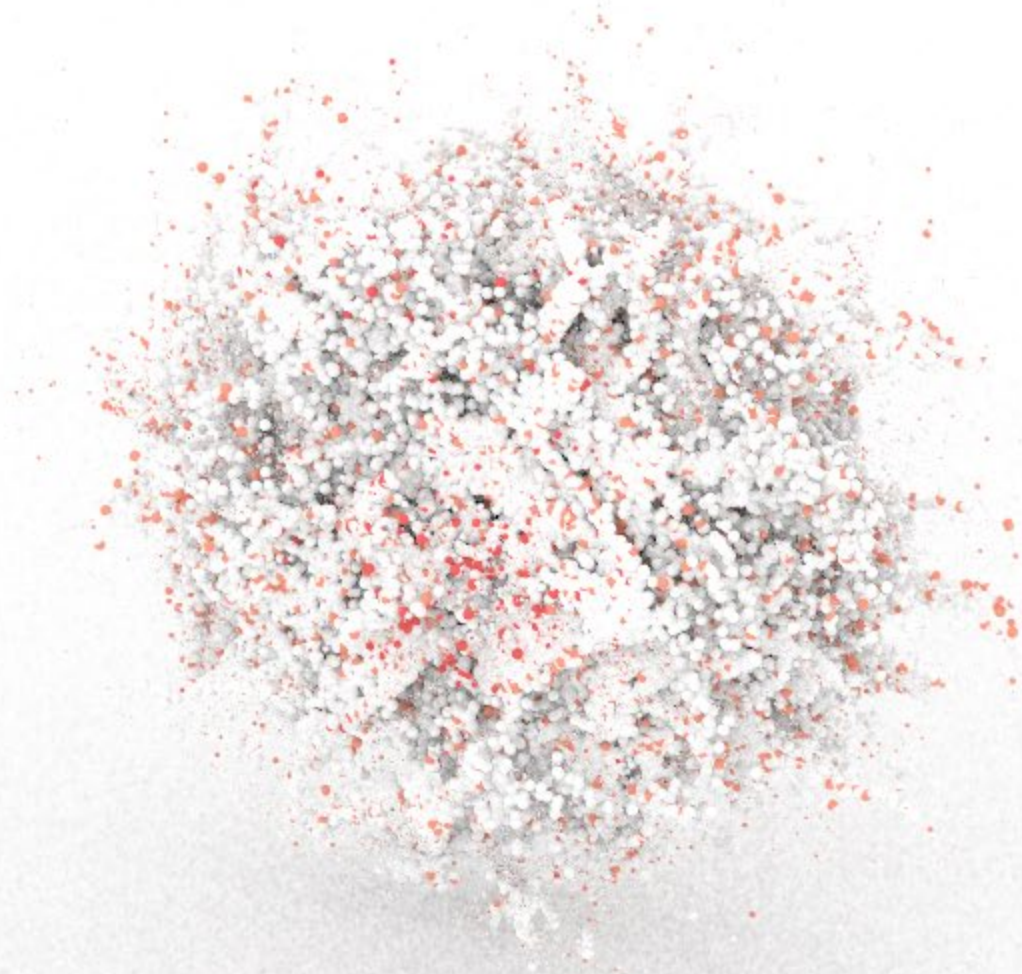
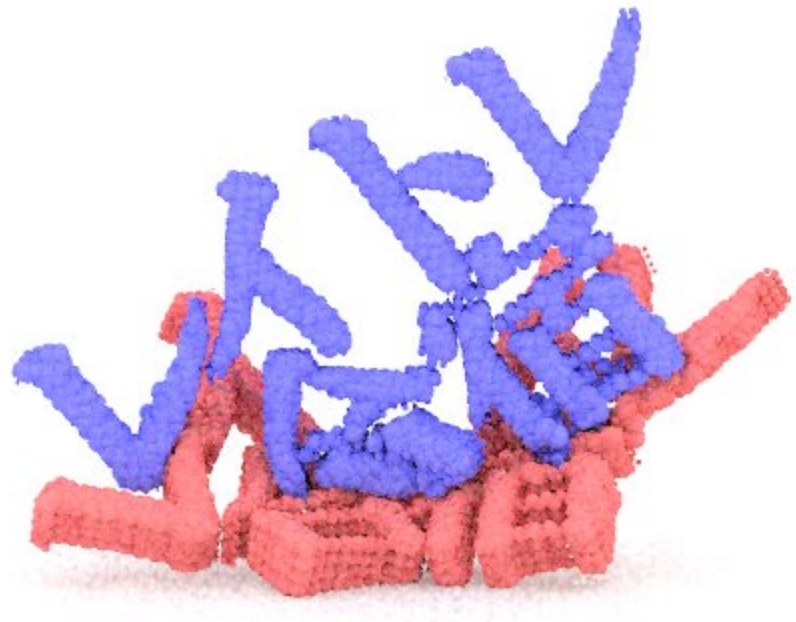
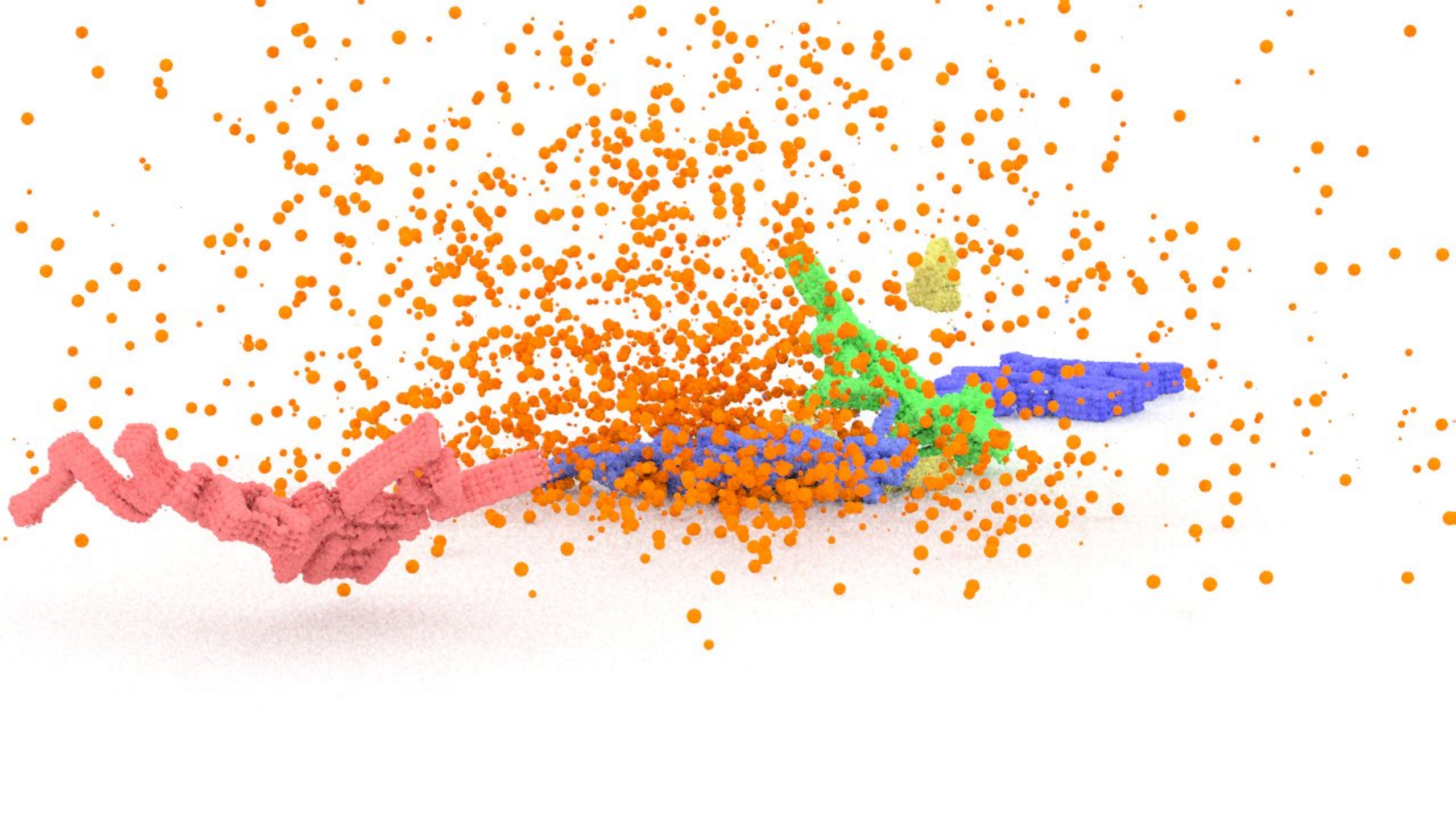
akari10

RayTracingCamp10

hole(@h013)







いままでいろんなものをレイトレしてきたが

パーティクル（的なの）をレイトレしたことはなかった！

akari10の主な機能

- パーティクル（というか球体）のレイトレ（パストレ）だけできる

akari10の主な機能

- パーティクル（というか球体）のレイトレ（パストレ）だけできる
- パーティクルといえばPosition Based Dynamicsなので、それも実装（ついでにCurl Noiseも）

akari10の主な機能

- パーティクル（というか球体）のレイトレ（パストレ）だけできる
- パーティクルといえばPosition Based Dynamicsなので、それも実装（ついでにCurl Noiseも）
- いよいよ計算力が厳しくなってきたので、泥臭いことをした

akari10の主な機能

- パーティクル（というか球体）のレイトレ（パストレ）だけできる
- パーティクルといえばPosition Based Dynamicsなので、それも実装（ついでにCurl Noiseも）
- いよいよ計算力が厳しくなってきたので、泥臭いことをした
- 10秒,30fpsの1280x720動画を出力（=300枚レンダリング）
  - 毎フレーム1秒未満で物理シミュレーションとレンダリングを同時に行っており、大変！

akari10の主な機能

- 使用ライブラリ
  - stbで画像読み書き
  - 後は全部C++標準ライブラリ

- 使用アセット
  - フォトショップで描いたテクスチャのみ

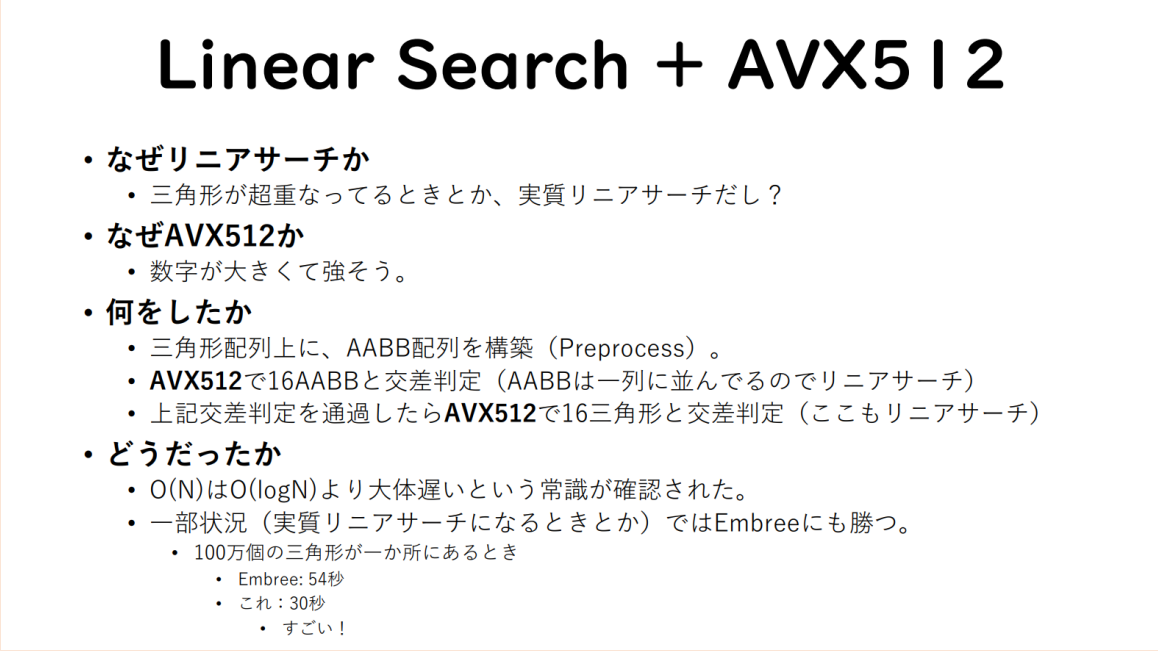
パーティクルのレイトレ

- AVX512を用いて、16個の球体と同時に交差判定を実現
  - 512 = 32bit x 16
  - 素朴なレイ・球体交差判定を愚直にSIMD化した

パーティクルのレイトレ

- AVX512を用いて、16個の球体と同時に交差判定を実現
  - 512 = 32bit x 16

  - 素朴なレイ・球体交差判定を愚直にSIMD化した



レイトレ合宿７でやった交差判定大会を
思い出す（なつかしい）

この時はAVX512を用いてメッシュとの交差判定を線形探索で実装した（厳密にはAABBへの線形探索と、AABB内の三角形への線形探索の二段階）

これで2位をとった

パーティクルのレイトレ

- さすがにただのAVX512だけだとどうにもならないので何らかの空間分割構造を入れたくなる

パーティクルのレイトレ

- さすがにただのAVX512だけだとどうにもならないので何らかの空間分割構造を入れたくなる

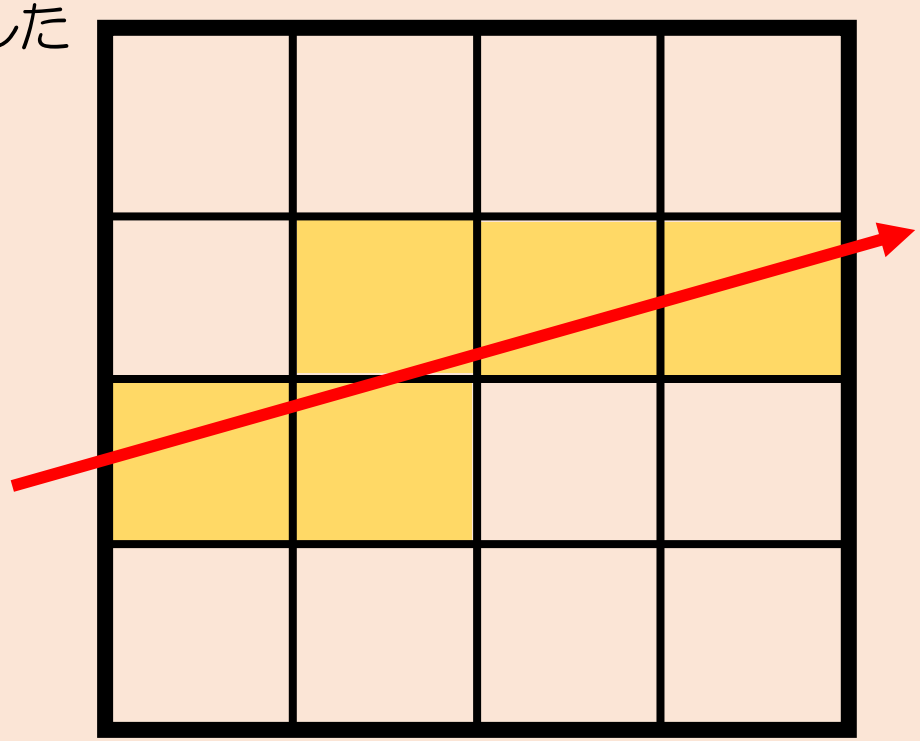
- 毎フレーム作り直しになる（パーティクルが激しくアニメーションするため）ので、爆速で構築できないといけない

パーティクルのレイトレ

- さすがにただのAVX512だけだとどうにもならないので何らかの空間分割構造を入れたくなる
- 毎フレーム作り直しになる（パーティクルが激しくアニメーションするため）ので、爆速で構築できないといけない
- そうだ、Uniform Gridをつかおう
  - （これが間違いの元だった）

Uniform Grid

- 有名だけど意外とレイトレで作ったことがなかったのでやってみようと思った
  - 各セル単位で球体を格納する
    - 各セル＝AVX512表現された16個ずつのリストとした
    - 時間がなかったのでこの辺手抜き
  - 交差判定するときはDDAして手前から探索
  - この辺で盛大にバグりちらかした



Uniform Grid感想

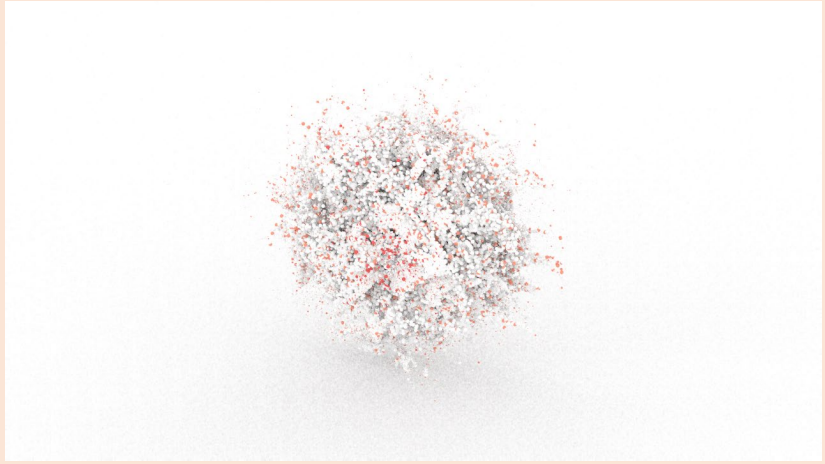
- 慣れないデータ構造を使うと実装にてまどり、バグる
- 結局、ちゃんと階層化したほうが高速に交差判定できそう
- 構築は確かに速い
  - O(球体の数)

Curl Noise

- 前半パートで使用
  - もはや伝統芸の域に達した、有名な疑似流体アルゴリズム
  - 2007年初出

- 三次元ノイズ場（PerlinNoiseなどから作る）に対してCurlをとると、ベクトル場の性質よりDivergence-freeなベクトル場が得られる
  - そして、Divergence-freeということは非圧縮性流体の速度場とみなせる

$$\vec{v}(x,y,z) = \left(\frac{\partial \psi_3}{\partial y} - \frac{\partial \psi_2}{\partial z}, \frac{\partial \psi_1}{\partial z} - \frac{\partial \psi_3}{\partial x}, \frac{\partial \psi_2}{\partial x} - \frac{\partial \psi_1}{\partial y} \right) \quad (1)$$



Position Based Dynamics

- 後半パートで使用
  - 弾性体シミュレーション部分

Position Based Dynamics

- パーティクルをアニメーションといえばPBD
  - というかPBDの実装がパーティクルを用いる

Position Based Dynamics

- パーティクルをアニメーションといえばPBD
  - というかPBDの実装がパーティクルを用いる

- 伝統的な物理シミュレーションは速度ベース、つまり速度の時間発展を計算する

Position Based Dynamics

- パーティクルをアニメーションといえばPBD
  - というかPBDの実装がパーティクルを用いる
- 伝統的な物理シミュレーションは速度ベース、つまり速度の時間発展を計算する
- PBDは前回の位置と今回の位置から速度を計算し、更新する
  - このとき、様々な「拘束条件」を満たすように位置を補正することで良い感じに物理がシミュレーションされる
  - 弾性体のシミュレーションを中心に発展したが、近年は流体や剛体も取り扱えるような拡張がされてる

Position Based Dynamicsの実装

- 今回は最も素朴なPBDを実装
  - 割と適当に実装してもそれっぽく動いて満足度が高い！（パストレのように）

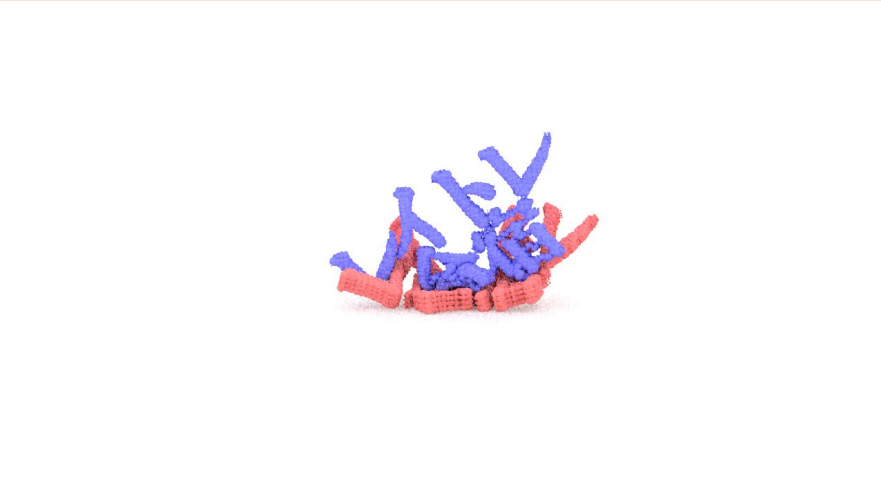
- アルゴリズム
  - 粒子に外力を加える
  - 拘束を解く
  - 位置から速度を更新する
  - 衝突関係を解決する

Position Based Dynamicsの実装の大変だったところ

- シミュレーションステップのdtが大きいとまともに動作しない
  - 後続手法では解決されてる風だが、素朴なPBDなのでこの辺の条件がきびしい
  - しょうがないので、dtを小さくしたうえでシミュレーションステップ自体を1フレームに何度も回すようにした
- 拘束を解くときの反復回数が小さいと挙動が微妙になる
  - 制限時間以内に良い感じのシミュレーションが得られるようになるまで地道に反復回数を調節…
- その他、さまざまな物理パラメータをいい感じに調節…

Position Based Dynamicsの実装の大変だったところ

- 粒子間の衝突判定もUniform Grid作って近傍探索することで実現
  - これは簡単に実装でき、かつ、効率も良かった
  - 衝突判定後、衝突の解決コードはChatGPTにおまかせ



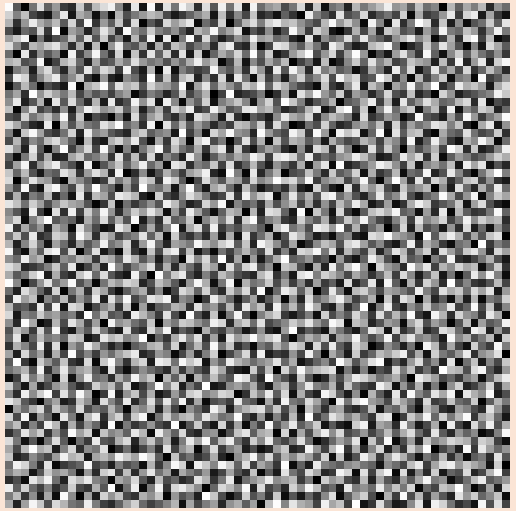
さまざまな泥臭い手当

- シミュレーションやらなにやらやっているとレンダリング予算がどんどん減る
  - 最終的に5sppまで減ってしまった！

- このままだとノイズだらけなので、空間方向・時間方向の両方向にガウスブラーを適用してデノイズした
  - ただし、同一球体由来のピクセルのみ考慮する
  - 「高度」なデノイズは実装も計算も余裕がなかったので単純な方法を採用
  - 時間方向のエイリアスも減り、動画にしたときのちらつきも軽減！

さまざまな泥臭い手当

- また、各ピクセルの乱数シード値はブルーノイズテクスチャのピクセルを直接使用した
  - この辺ももっと賢い手法がたくさんあるが、何も考えずに乱数シードを上記テクスチャ由来にするだけでも何もしないよりは良くなる



おわり