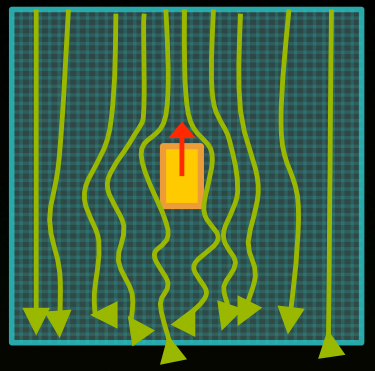

Turbulence:
オイラー流体力学モデルの
導入
Bryan Dudash
NVIDIA

概要

- Turbulence 序論
- CUDA シミュレーション
  - オイラー・グリッド・流体シミュレーション
  - 移流パーティクル
- パーティクル・レンダリングの考察
  - High Speed Particle Rendering with VTF
  - パーティクルシステムのボリュームライティング
- エンジンへの統合問題
  - 一般的考察と例
  - LODとfallbacks
- デモ

Turbulence 序論

Turbulence High Level



- “Turbulence”(乱流)は激しい動きによって引き起こされるシミュレーショングリッド上のパーティクルで表現されます
- ここで述べるテクニックは完全にGPU上で走らすことが出来るスケーラブルな物理シミュレーションモジュールです
- この手法は流体シミュレーショングリッド上での動作によって発生する任意複数のパーティクルをサポートします

NVIDIA Car Demo

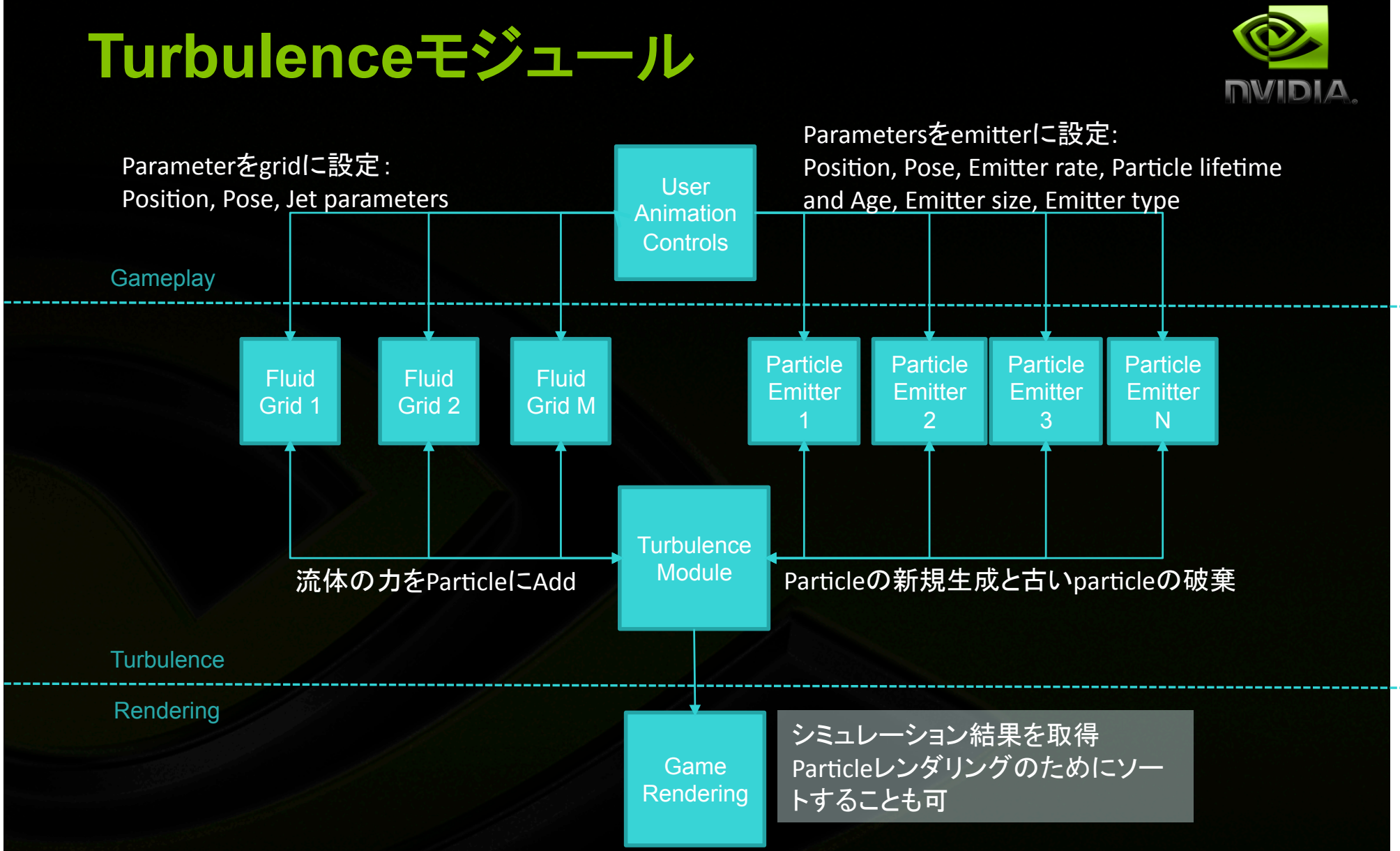
Dark Void – Airtight Studios/Capcom

CUDA でのTurbulence シミュレーション

Simulation

- 動的オイラー流体シミュレーションのグリッドは注目しているエリアを追跡します
- 流体シミュレーションは、そのグリッド内のParticleに対して行います
- Particleがシミュレーションの範囲外に出たときは他のシンプルな手法にシームレスに移行します

Turbulenceモジュール



流体シミュレーション

- GPUでのオイラーシミュレーション
  - (CUDAで実装)
- リファレンスフレームの動きを扱えます
- マルチグリッド手法で(流体の)圧力を解決します
- 高次方程式を使って移流速度を計算します
- この解法に要する時間は反復毎に一定です

オイラーシミュレーション

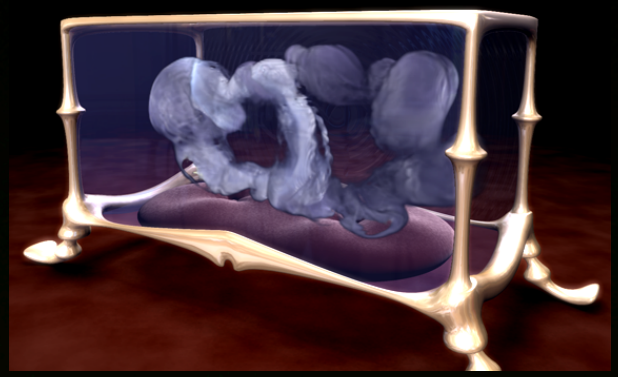
- Particleでの方法は大きな領域での均一流体のシミュレーションには向いていません
  - 非常にたくさんのparticlesが必要になります
- オイラーの手法は相互に作用する流体に対して使われています
  - しかし基本的に有限直線定義域に限定されます
  - 広大な環境下での非制約動作に使うのは難しいです

SmokeをBoxの外へ

- このエフェクトをどこにおいても適用するために、次の二つを使います

  1. 移動性流体グリッド

  2. オイラーグリッドの境界を自由に出入りできる流体の移動

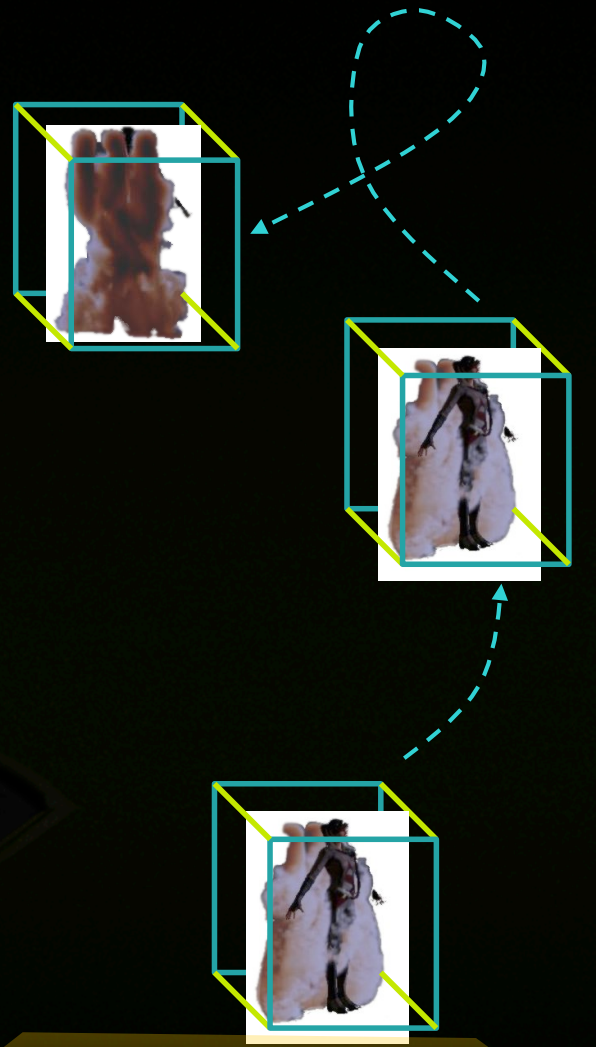


Fluid in a Box demo
固定されたboxの中で流体シミュレーションが行われている

APEX Turbulence
移動性グリッドとParticleによって流体エフェクトがどこにでも動けるようになります

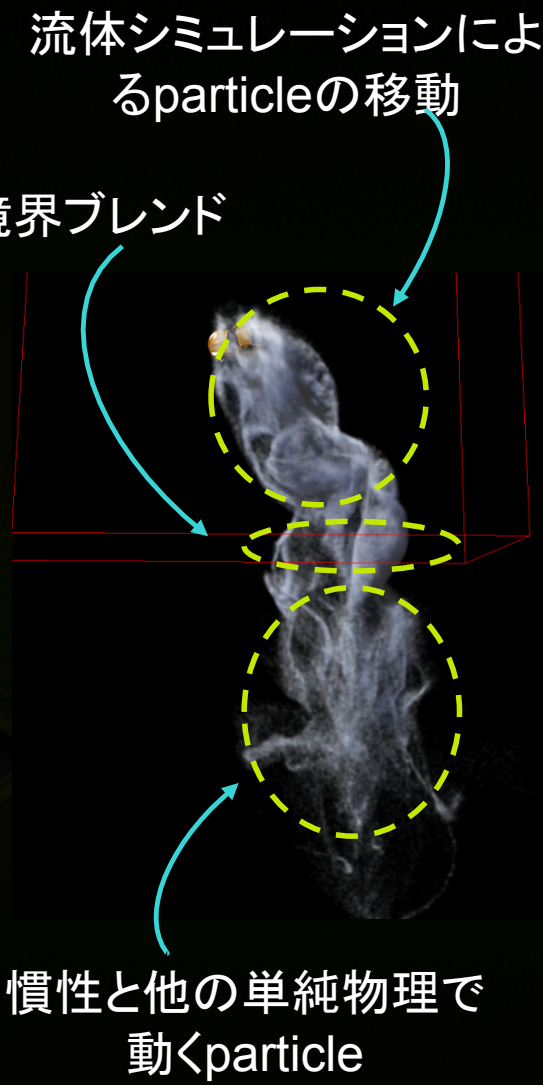
SmokeをBoxの外へ

- 流体グリッドは注目エリアを追跡するために移動することができます
  - 例：動いているキャラクターの周りの乱気流
- シミュレーショングリッドは注目しているオブジェクトと同じ速さで動きます
  - 注目オブジェクトはシミュレーションの座標系内では同じ位置に留まります



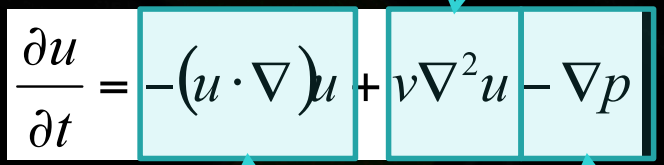
SmokeをBoxの外へ

- 流体力学が有界領域でのみ起こることを仮定します
  - したがって、有界領域内に関してのみ力学的正確さが保障されます
  - その上で、簡単な手法を範囲外に対して使うことができます
  - シミュレーショングリッドの境界において、これら二つの方法をブレンドします



流体シミュレーション

- シングルフェーズ非圧縮流体の速度 u の運動方程式で表わされる:



明示的な粘度の項
これは、グリッドサイズと移流の式から伝播の数値が得られているので無視できる

圧力勾配

慣性力による加速度

- 非圧縮という制約条件

$$\nabla \cdot u = 0$$

流体シミュレーションの仮想コード

u = SelfAdvection(u)　(MacCormack法を使った 運搬係数 u)

For i=1 to iop do

　uに固体境界条件を適用
　uに力積を適用

　圧力を消去
　$\nabla^2 p = \nabla \cdot u$　を解く　(マルチグリッド手法を使って)
　$u = u - \nabla p$

end for

MacCormack運搬

- 半ラグランジュ運搬係数は無条件で安定しているが、必要以上に数値的平滑さを持つ
- MacCormack法を使う
  - ディテールを失わない
  - どのtime stepに対しても安定
  - 明らかに並列演算可能なのでGPU上で計算するのに効果的

Basic 半ラグランジュ運搬

MacCormack運搬

MacCormack運搬

- Basic 半ラグランジュ運搬はシングル・フォワードな移流ステップ (下記で *A*で示される)
- MacCormack運搬*:

$$forward\quad step: \quad \hat{\phi}^{n+1} = A(\phi^n)$$

$$backward\quad step: \quad \hat{\phi}^{n} = A(\hat{\phi}^{n+1})$$

$$estimate\quad error: \quad e = (\hat{\phi}^{n} - \phi^{n})/2$$

$$final\quad estimate: \quad \phi^{n+1} = \hat{\phi}^{n+1} + e$$

$$clamp\quad \phi^{n+1}:$$

$$\phi_{\max} = \max(\phi^n, \phi^{n+1})$$

$$\phi_{\min} = \min(\phi^n, \phi^{n+1})$$

$$if\ (\phi^{n+1} > \phi_{\max})or(\phi^{n+1} < \phi_{\min}) \quad \phi^{n+1} = \hat{\phi}^{n+1} + 0.5 * e$$

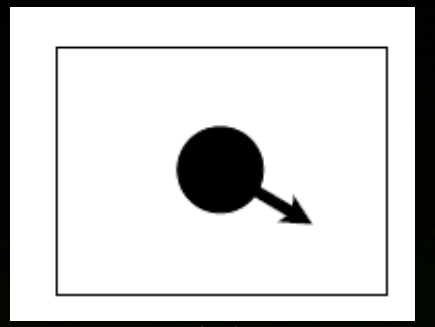
$$clamp\quad \phi^{n+1}\quad between\quad \min(\phi^{n+1}, \phi_{\max})\quad and\quad \max(\phi^{n+1}, \phi_{\min})$$

圧力の解法と内部境界

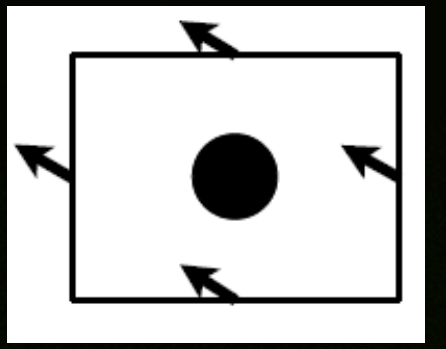
- マルチグリッド圧力解法を使います
  - Red-Black SOR法のsmoothingを使います
- 内部境界条件を反復直交射影の骨組みを使って扱います
  - 内部の固体オブジェクトを処理すればいいようになります
  - また任意の衝撃を射影段階前に速度に適用できるようになります
    - これらの衝撃はジェットのために使います

リファレンスフレームの動き

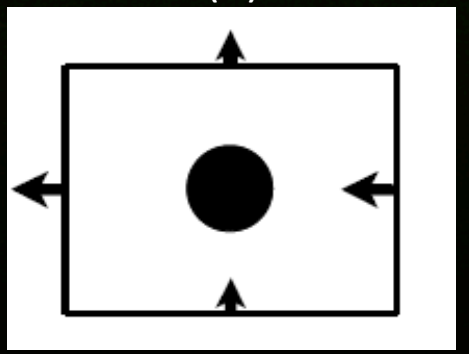
- リファレンスフレームの動きでの非圧縮Navier-Stokes解法を計算するためにガリレイ不変性を利用します
- これによりリファレンスフレームの動きを流出入境界条件に使うことが出来るようになります



(a)



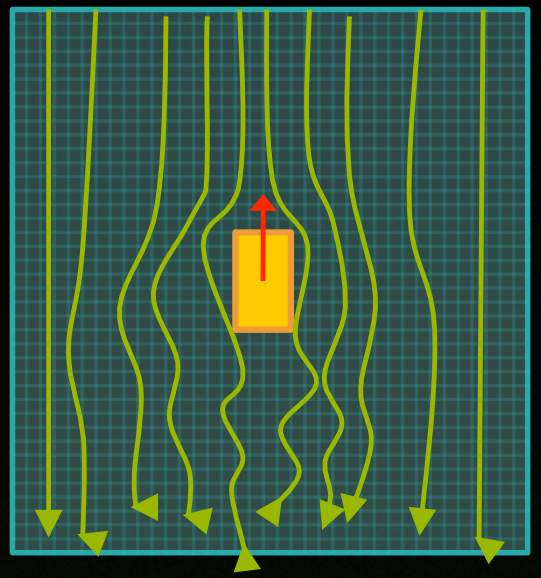
(b)



(c)

Turbulenceの作成

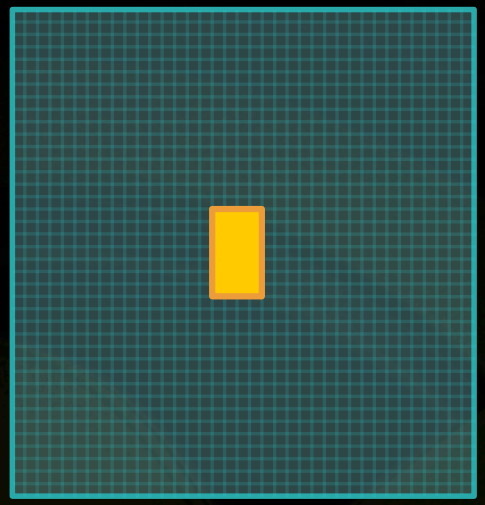
- オブジェクトは動くと空気を移動させ、背後に乱流(Turbulent)を生じさせます
- シミュレーション上では、オブジェクトはコリジョン障害物として表わされます
- 障害物の表面はシミュレーション上で乱流を増やすために荒いものとして扱うこともできます



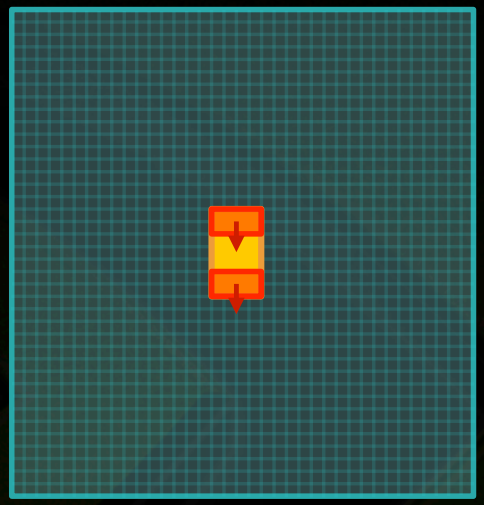
Turbulenceの作成

- また噴出(Jet)をシミュレーションに力を注入するために導入することもできます
- さまざまな力の方向や大きさを徐々に与えることによって、システム内の乱流を増やすことができます
- 噴出(Jet)はユーザー定義のものとして表わされます。陰関数に落とし込んだ全てのグリッドセルは噴出(Jet)力を得ます

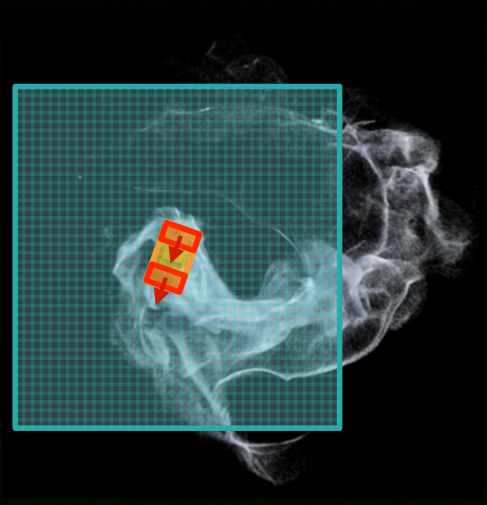
Case Study



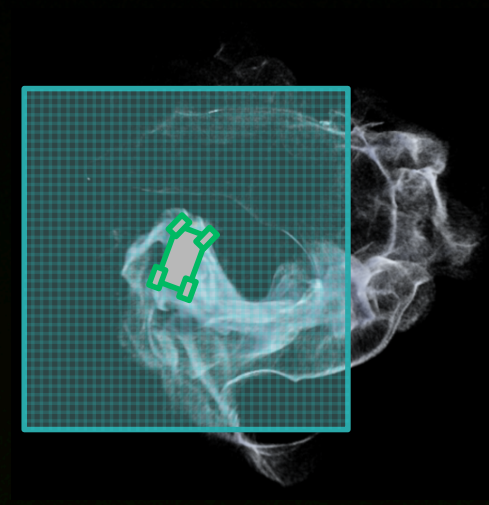
(a)
オレンジで示された暗黙のコリジョンを持つ流体グリッド



(b)
二つのジェットが車輪の前と後ろとして付け加えられた状態。力の方向は矢印で示されています



(c)
車の動きに合わせてシミュレーショングリッド(暗黙のコリジョンと噴出も)が合わせて動きます。
車が回転すればコリジョンとジェットも回転します



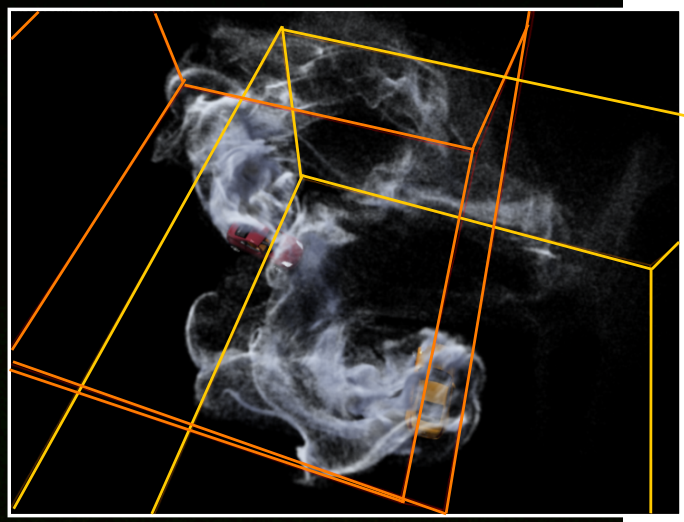
(d)
ジェットの放出は独立に移動と回転が出来ます。ここではホイールに合わせてジェット放出が回転しています

Particle Simulation

- シーン内のエミッターは一つの大きなparticleバッファーに加えられます
  - 各々のエミッターはこのバッファ内で分離された位置にアロケートされます
- 全てのparticleをこのバッファに入れるために
  - 流体グリッドに基づいて力を与えます
  - Particleに対しての外力はグリッドに落とし込みません
  - Particleをそれぞれの速度に応じて動かしたり、減少させたり、寿命を延ばしたりします

複数のSimulations

- 流体グリッドは独立してシミュレートされます
- しかしながら、重複したシミュレーショングリッドに落とし込まれた複数のparticleは、それら全ての力をグリッドに集めてしまいます



二つの重複した流体シミュレーショングリッド

複数のSimulations

- 各々のシミュレーションはウェイト付けされた衝撃を各々のパーティクルに与えます
  - グリッド外のウェイトは0です
  - グリッドの中心で最もウェイトが高くなります

$$fluidInfluencedVelocity = \sum_{allgrids} grid.velocity.trilinearSample(particlePosition) * weight(particlePosition)$$

- 最終的なparticleの速度は下のように計算できます

$$particleVelocity = MaxWeight * fluidInfluencedVelocity * \left(1.0f / WeightsSum\right) + (1.0 - MaxWeight) * particleVelocity$$

ソート

- Particleを正しくレンダリングするために、それらをソートするメカニズムを提供しています
- ソートはゲームによって与えられる向きに沿ってCUDAを使ったRadixソートで行われます
- レンダリング等を高速化するために、この段階でDead particleはparticle bufferから削除されます

高速パーティクルレンダリング

伝統的Particleレンダリング

- Lock and Fill Dynamic Vertex Buffer
  - CPUボトルネックをしばしば生じさせます
  - 非常に大きな帯域幅ボトルネック
  - そこそこのSetup (DA)ボトルネック
- 伝統的 Stream Instancing
  - Instance Data Vertex Bufferの分離
  - Vertexサイズは膨張します
    - 基本vertexデータ + instanceデータ
  - 非常に大きなDAのボトルネック

Vertex Texture Fetch Particles

- フレーム毎のdynamic dataをtextureに落とし込みます
  - GPUで直接計算 – CUDA & Graphics shared mem
    - 非常に早いパス
  - それか CPUで – Lock and Update
    - CPU側での変更が可能 (注意深く!)
    - Readbackが必要
- Vertex ShaderでのSample
  - Vertex Texture Fetch (VTF)
    - Tex2dlod
    - 大きなtexture cacheを利用
  - Quad単位でワールドスペースへ変換

VTF Particles (2) – 描画

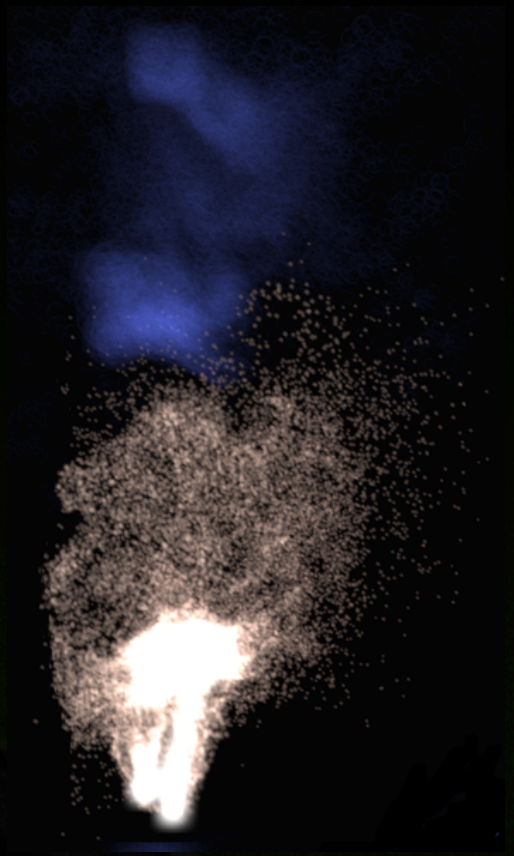
- [DX10] Hybrid Instancing
  - NULL Vertex Buffer
  - SV_InstanceID が texture coordsを生成
  - Particle triangleはGeomery Shaderが生成
- [DX9] Manual Instancing
  - staticなVBを最大particle数分アロケート
  - 原点上のFace upしたビルボードを含みます
  - VTF textureを参照するための事前生成texture座標
  - Static bufferはGPU上に置きます
    - ビデオメモリを消費してしまいますが。

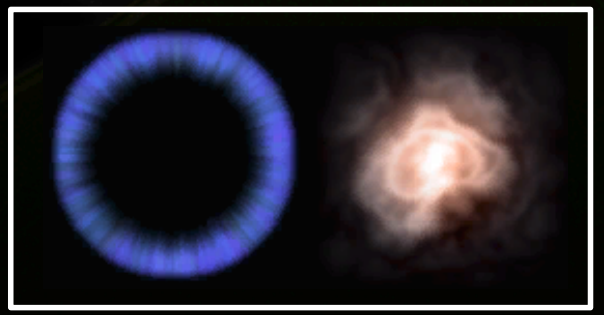
DX9 Example Vertex Shader

```
// Normal interpolated texture coords for pixel shader
Result.TexCoord = Input.TexCoord.xy;

// Using VTF to load particle position
float4 PositionAndTextureIndex = tex2Dlod(VTFPosSampler, float4(Input.TexCoord.zw,0,0));

// VTF to load velocity of particle plus life remaining
Result.InstanceVelocity_Life = tex2Dlod(VTFVelSampler, float4(Input.TexCoord.zw,0,0));

// A little shrinking of particles over their life
float uniformScale = SpriteScale * (0.5+0.5*saturate(Result.InstanceVelocity_Life.w));

// zero area tris for dead particles
if(Result.InstanceVelocity_Life.w < 0.01) uniformScale = 0;

float3x3 scale = float3x3(
                      float3(uniformScale,0,0),
                      float3(0,uniformScale,0),
                      float3(0,0,uniformScale));

Result.Position = Input.Position;

// scale sprite
Result.Position = float4(MulMatrix(scale,Result.Position),1);

// rotate to camera
Result.Position = float4(MulMatrix((float3x3)InvView,Result.Position),1);

// move to proper world location
Result.Position += PositionAndTextureIndex.xyz;
```

Multiple Textures?

- Problem: 全てのparticleに対して1 draw call
  - ビューでソート
    - 正しくblendできることを保障
  - Materialパラメータを変えることが難しく
- Solution: Texture Atlasを使う
  - パーティクル毎のtexture indexをエンコードしておきます
  - パーティクル毎に”sub UV”を生成します





Lighting High Density Particle Systems

Different looks

Smoke

Leaves

Sand

Lighting Options

- Simple Alpha Test
  - Leaves, paper, 破片
- 付加的なBlending
  - 魔法のエフェクト
  - “放出型” particles
- Phong lightingとAlpha Blending
  - “伝統的” alpha’d particles
  - 内部depthキューを覆い隠す
- Alpha Blending volumetric lighting
  - より正確な内部depth cues
  - 乱流の動きを強調します

Volumetric Lighting

- “Volumetric Particle Shadows”
  - Simon Green 2008, NVIDIA
  - http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/smokeParticles/doc/smokeParticles.pdf
  - lightとviewの半角
  - Volumeをスライスに分割し加算shadowing
- Min-max shadow map
  - zminとzmaxをshadow bufferへレンダリング
    - MIN blendingを使ってone pass
  - Beer法を光の減衰に使います
  - Exponential Shadow Map (ESM)を使ってフィルタできます
    - http://www.mpi-inf.mpg.de/~tannen/papers/gi_08_esm.pdf

Volumetric Lighting (2)

- Opacity Shadow Maps
  - Tae-yong Kim, Ulrich Neumann, Rendering Techniques (2001)
  - http://graphics.usc.edu/cgit/pdf/papers/EGRW_KIM2001H.pdf
  - 2D textureを使った3D shadow volume近似
  - Volumeのスライスに対して不透明度を加算
    - 全スライスに対して反復
    - 二つのrender targetを交互に使用
  - Shadowは各ポイントにおいて不透明度に比例

Game Engineへの統合と
パフォーマンスの問題

APEX Turbulence

- APEXはPhysXの上にあるミドルウェアライブラリ
  - PhysXに対して問題無く追加
  - GPUでのphysics effectsに対してLOD スケーラビリティを提供
- APEX Turbulence Module
  - このセッションでの全ての要素をインプリメント済みです
    - CUDAを使ってGPU上で行います
- Unreal Engine 3のmain branchに統合される予定です
  - Epicと共同開発しています

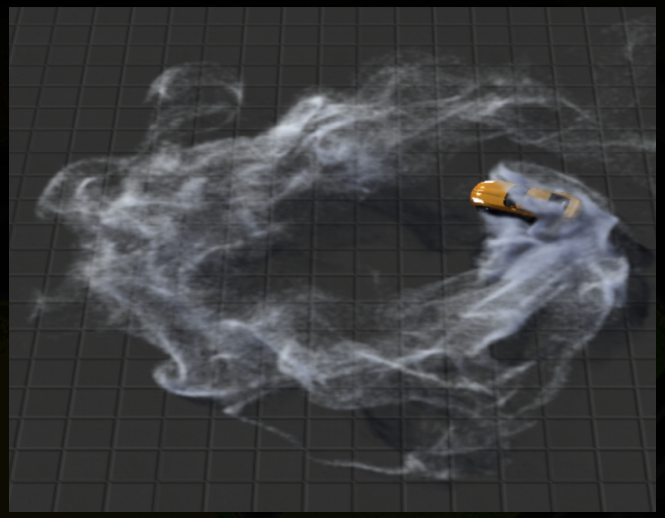
Engineへの統合

- Simulation module
  - 自分自身で書く
  - APEX Turbulenceを使う
- Gameplay code
  - Dynamicシーンで動くactorに対してグリッドとエミッターをアタッチ
  - Gameplayのイベントに応じてdynamic parameterをトリガー
    - emitterのon/off
    - 噴出(jet)力のon/off
    - アタッチされたactorに応じてグリッド速度をアップデート
- Rendering code
  - VTF particleレンダリングを追加
    - 他のαオブジェクトとのブレンドでノイズが起こるかも知れません
    - エミッター毎に描画することでこの問題を解決することができます
  - Volumeライティングの追加 (望むなら)
    - 通常のαブレンディングレンダリングの後にインプリメントされています

Performanceボトルネック

- Simulation
  - Grid cells
    - 低レゾリューションのグリッドは早いです
    - シミュレーションにおいて最もパフォーマンスコストが高い箇所です
  - グリッドの分割数
    - 64x64x64 != 8x 32x32x32 (大きなグリッドはより効果的です)
    - グリッドがOverlapしている場合、追加のparticle移動コストがかかります
  - ワールド内でのアクティブなparticleの数
    - 単一グリッドに対して移動はO(N)
    - GPUメモリ上でのバッファサイズに影響します
- General Rendering
  - 描画particleの数
    - VTF textureのアップロードのための帯域に影響します
    - プリミティブカウントの増加はシェーダロードを増加させます
  - 密集したαブレンディング箇所は重い多重描画を引き起こします
    - 低レゾリューションのオフスクリーンRTを使ってフィルのボトルネックを減らすことが可能です

シミュレーションコストのScaling



128x32x128 grid
500K particles

64x16x64 grid
200K particles

32x16x32 grid
100K particles

コストレベルの違う3つのシーンです
低コストな設定を使ってもシミュレーション結果の雰囲気は似ていることに注目してください

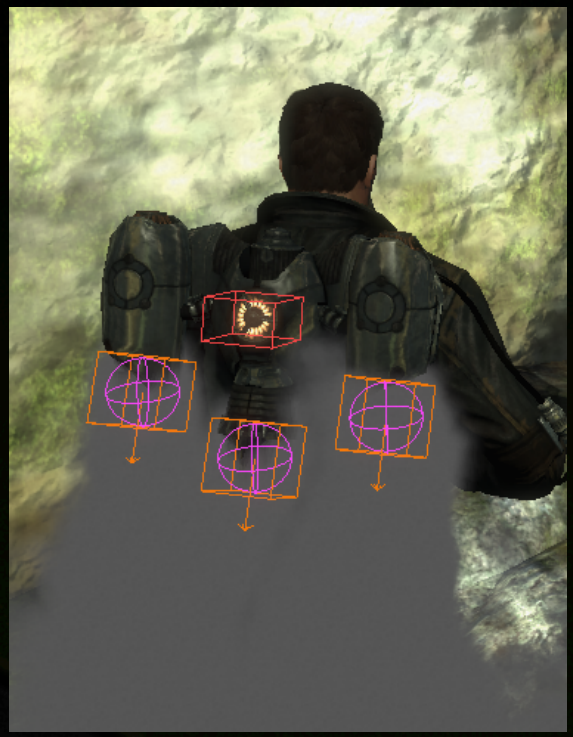
LOD options

- Simulation Cost
  - アップデートのスキップによってグリッドをLOD化
    - 1フレームおきの速度アップデート、など
    - Particleの動きはスムーズなまま
  - エミッターは最大particle数でLOD化できます
    - GPUパワーに応じて1フレーム当たりの放出particleの量を減らします
- Rendering
  - 低レゾリューション　オフスクリーン 透過性
    - 透過オブジェクトの描画を ¼ (かそれ以下)のサイズのRTに
    - そのRTとback bufferをblend
    - 多重描画の大半を避けることができます

Example – Dark Void

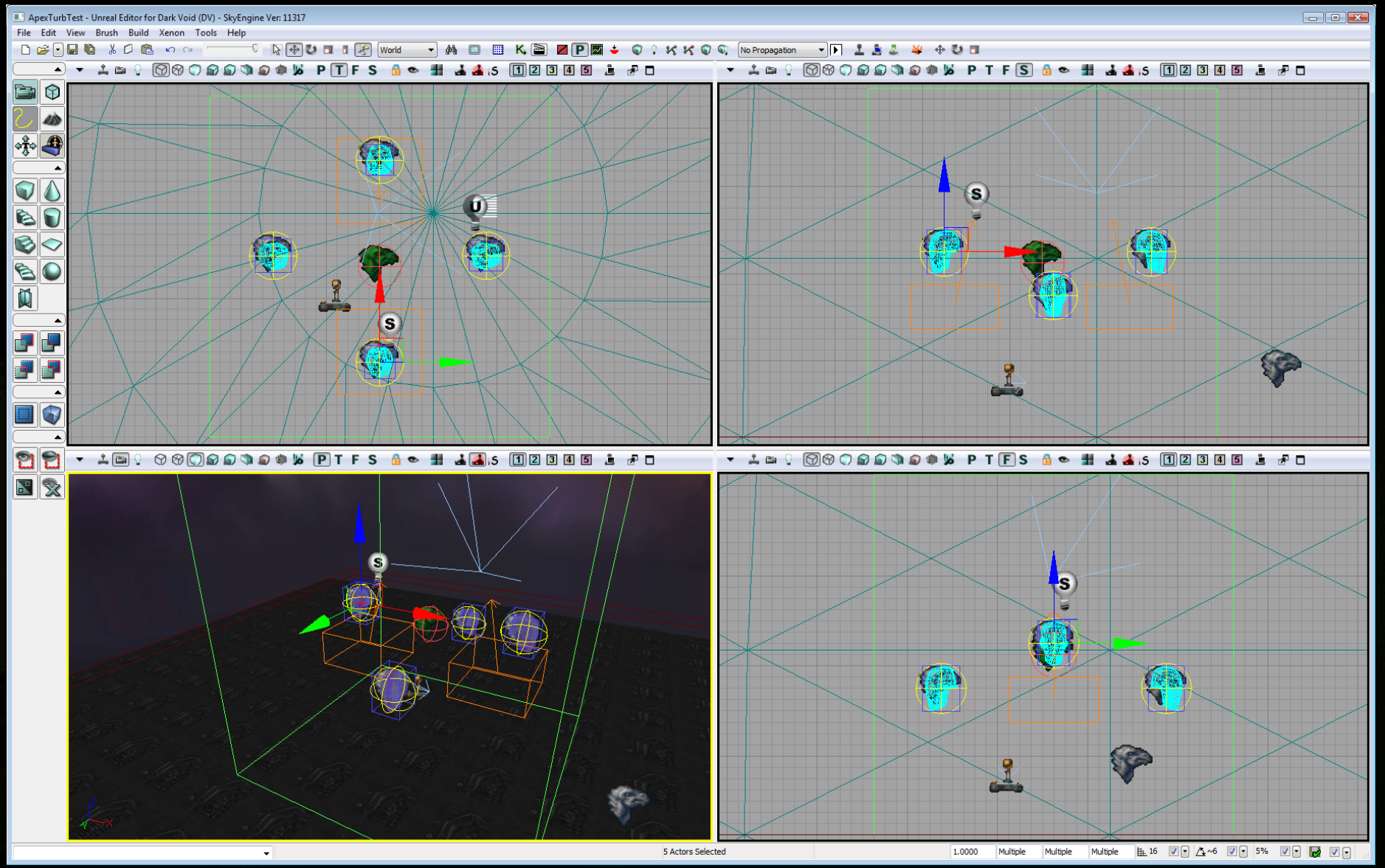
- Airtight Studios開発
- Epic's Unreal Engine 3を使用

- playerのjetpackにシミュレーショングリッドをアタッチ
- jetpackのnozzlesにエミッタ
  - デザイナーが決めたソケットに対してアタッチ
- エミッタ周辺にjetを
  - ソケットにもアタッチ



Dark Void Shots

Unreal Editor Support



Demo

Questions?

- bdudash@nvidia.com
- devsupport@nvidia.com
- NVIDIA Japanは、エンジニア募集中
  jp-recruitment@nvidia.com
- NVIDIA APEX
- NVIDIA SDK 11 release early next year