UNREAL
ENGINE

Profiling and Optimizing UE4（日本語版）

目次

- プロファイリングのベストプラクティス
- ひと目で分かるプロファイリングの流れ
- CPUプロファイリング
- Blueprintの最適化
- GPUプロファイリング
- 一般的なGPU最適化方法

プロファイリングのベストプラクティス

When and how to approach the process

いつ / どのようにプロファイリングするべきか

- 早く そして 頻繁に！
  - 終盤まで待つのではなく、すぐトライしましょう！
  - ただし、過度にするのはおすすめしません
    - プロファイリングして問題があった所にだけ最適化
- 可能な限り早く
  対象のハードウェア上でテストしましょう
- 全員が基本的なプロファイリング方法を知っているように
  - アセットのコストを知るアーティストは多くの時間を節約します！

プロファイリング環境の構築

- プロファイリングの邪魔をするノイズを最小限に
  - 使用しない機能は全てOFFにしましょう
- Framerate Smoothingを切りましょう
  - Project Settings > General Settings > Framerate group
- Testビルドを作りましょう
  - Developmentビルドでテストすると、
    Drawスレッドがノイズで膨れ上がります！

エディタを使ったプロファイリング

- エディタ上では詳細なプロファイリングはしないように
  - エディタは多くのものにノイズを与えるため、多くの数値の信頼性を下げます
  - 一般的なパフォーマンステストには良いですが、
    100%理想的というわけではありません
- エディタでプロファイリングをする場合は
  - Standaloneで実行
  - エディタのリアルタイム更新をOFFに
  - エディタを最小化
  - Frame Rate Smoothing (Project Settings) をOFFに
  - Vsyncを切る (r.vsync 0)

プロファイリング手順

An at-a-glance look at the approach to profiling

一般的な方法

- ボトルネックを特定しよう
  - Game Thread
  - Render Thread
    - CPU
    - GPU
  - 最適化すると、頻繁にボトルネックが変化します
- ボトルネック範囲の問題を解決しましょう
  - Game Thread – C++コード or Blueprint
  - CPU Render – オブジェクト数, ドローコール, カリング
  - GPU Render – シェーダ, オーバードロー, ライト

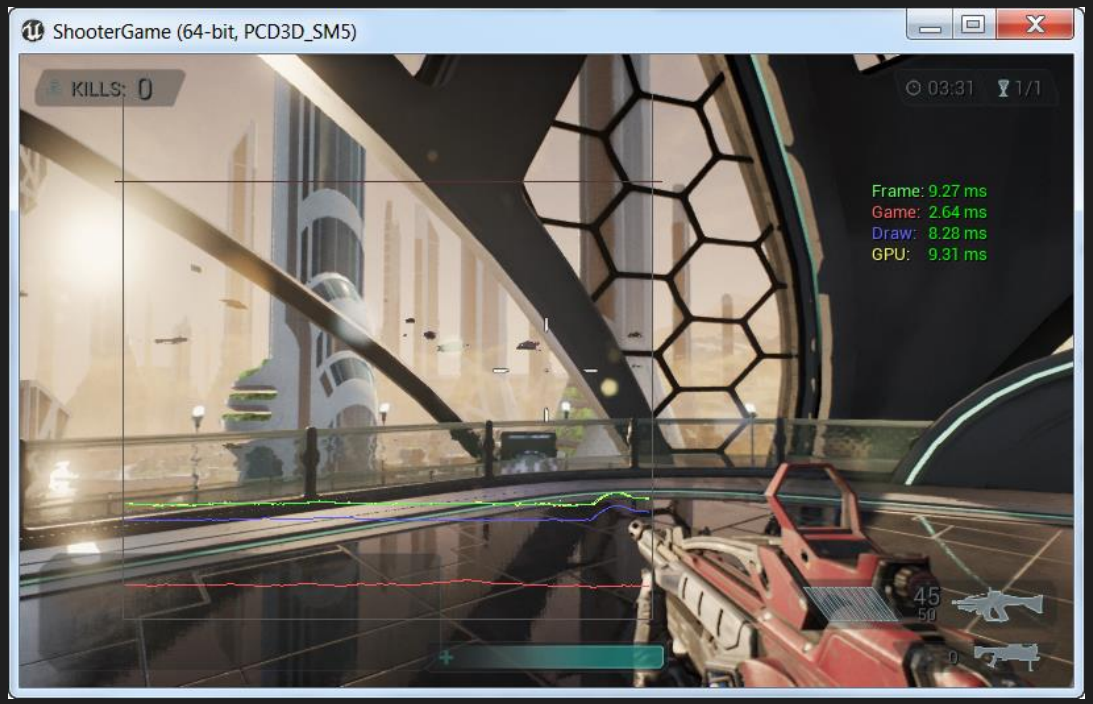
自動化?

- 一部のチームはシーンを飛び回るカメラを設定し、
  テストのために特定のポイントを使うことを好みます
- これは物事の一貫性を保つのに役立ちます。
  しかし、ゲームプレイのテストには難しいです
- カメラが高密度なエリアを通過することを確認してください。
  シンプルなエリアではよくありません
- キャラクタやエフェクトをテストすることもお忘れなく
  シーケンサが便利です！
- Paragonの場合:
  - テクニカルアーティストはどのエリアに問題があるのかを知っていました
  - そして、プロファイリング時間の大半をそれらのエリアに充てました

ミリ秒単位で測定しよう

- *stat fps* ではなく *stat unit* 使おう
  - 最も大きい値がボトルネックの可能性を示します

- フレームごとの処理時間（ms)
  - Frame: 各フレームの合計処理時間
  - Game: C++ or BPによるゲームプレイ処理
  - Draw: CPU render time
  - GPU: GPU render time
- *stat unitGraph* を使うことで
  線グラフで測定することができます
  - 主に繰り返すヒッチに使用されます
    (心臓の鼓動のように)

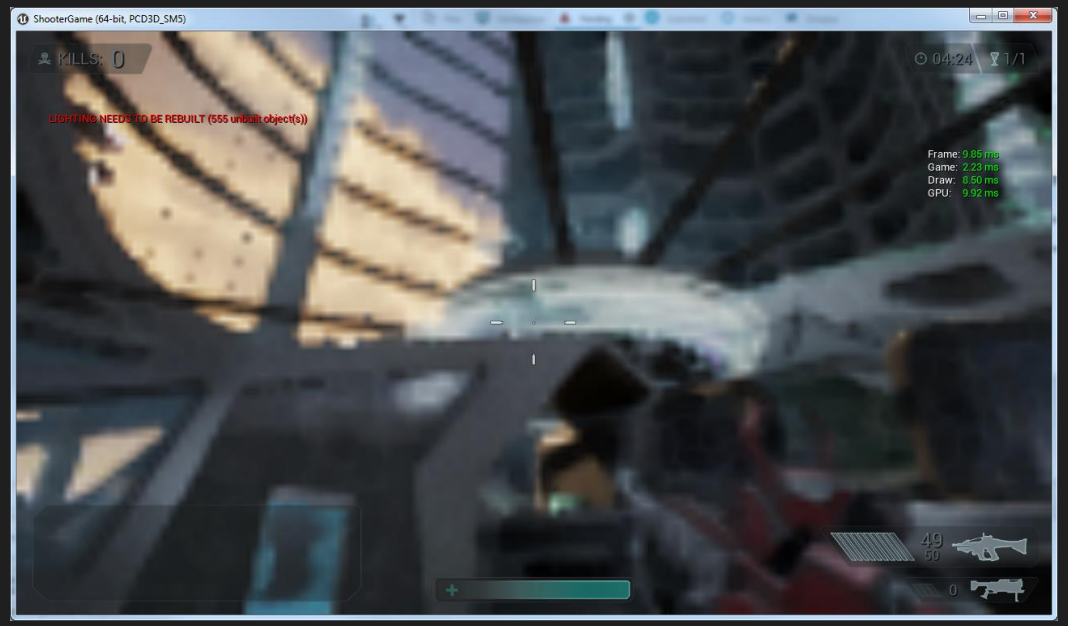


スイッチを捨てて、何が壊れているのかを見よう！

- ボトルネックの原因を探すのは面倒です
  まずは簡単な所からはじめよう！
- ときには最善の最初のステップは、
  機能をON/OFFして改善されるか確認することです！
- これは、シーンの一部分の表示 / 非表示をするのと
  同じように簡単にできます
- また、異なる解像度でレンダリングしたときに
  何が起こるのかを見ることも非常に便利です
  - GPUに負荷をかけすぎているなど

ScreenPercentage

- Game Threadとは無関係な問題を計測する際にとても役に立ちます
- *stat unit* で処理時間を表示
- *r.ScreenPercentage 10* を使う
  - または100よりも小さい数字
  - GPUに送られるピクセル数が減ります
  - もし速くなったら、GPUがボトルネック
  - もし速くならなかったら、CPUがボトルネック



Show Flags

- 問題を探す最も簡単な方法の一つはシーンの一部をOFFにすることです

- 削減する時間を知ることに役立ちます
  - より多くの LODs
  - より少ないtranslucency（半透明）
  - 調整されたライティング

show <assetType> or

showFlag.<assetType> 0-1

- *Staticmeshes*
- *Skeletalmeshes*
- *Particles*
- *Lighting*
- *Translucency*
- *Reflectionenvironment*
- まだまだ沢山あります！

診断ツール

Realtime stats and view modes

Stat コマンド

- statコマンドは
  エンジンの様々な部分における
  統計情報をリアルタイムに
  フィードバックします
- 沢山あります
- パフォーマンスのために
  最も一般的なものをご紹介します
  - 完全なリストに関しては
    公式ドキュメントをご確認ください

stat fps

stat unit

stat scenerendering

stat gpu

stat engine

stat streaming

stat emitters

stat lighting

Stat SceneRendering

- ドローコールのみを見るコマンド
  - ドローコールは何かを描画するために
    GPUに送る単一のリクエストです
  - ローエンドなマシンやモバイルではCPUにて速度が下がる可能性が
    あります(Metal とVulkanの場合は影響が少ないです)
- 他に見たほうが良い箇所：
  - Shadows
  - Decals
  - Post Processing
  - Lighting

Stat GPU

- 4.15で追加されました
- GPUからリアルタイムで計測
- ハイライトは見れますが、詳細は見れません
  - すばやく問題点を見つける際に有用です
- 詳細を見る場合はGPUプロファイラを使用してください
  - 例えば、影を落としている特定のライトを探したい場合など

Optimization View Modes

- プロファイリングブレデターになる！
- View Modes は問題の原因を明らかにします
- エディタ画面で全てを使えます
- プロファイリングツールと併用するべきです

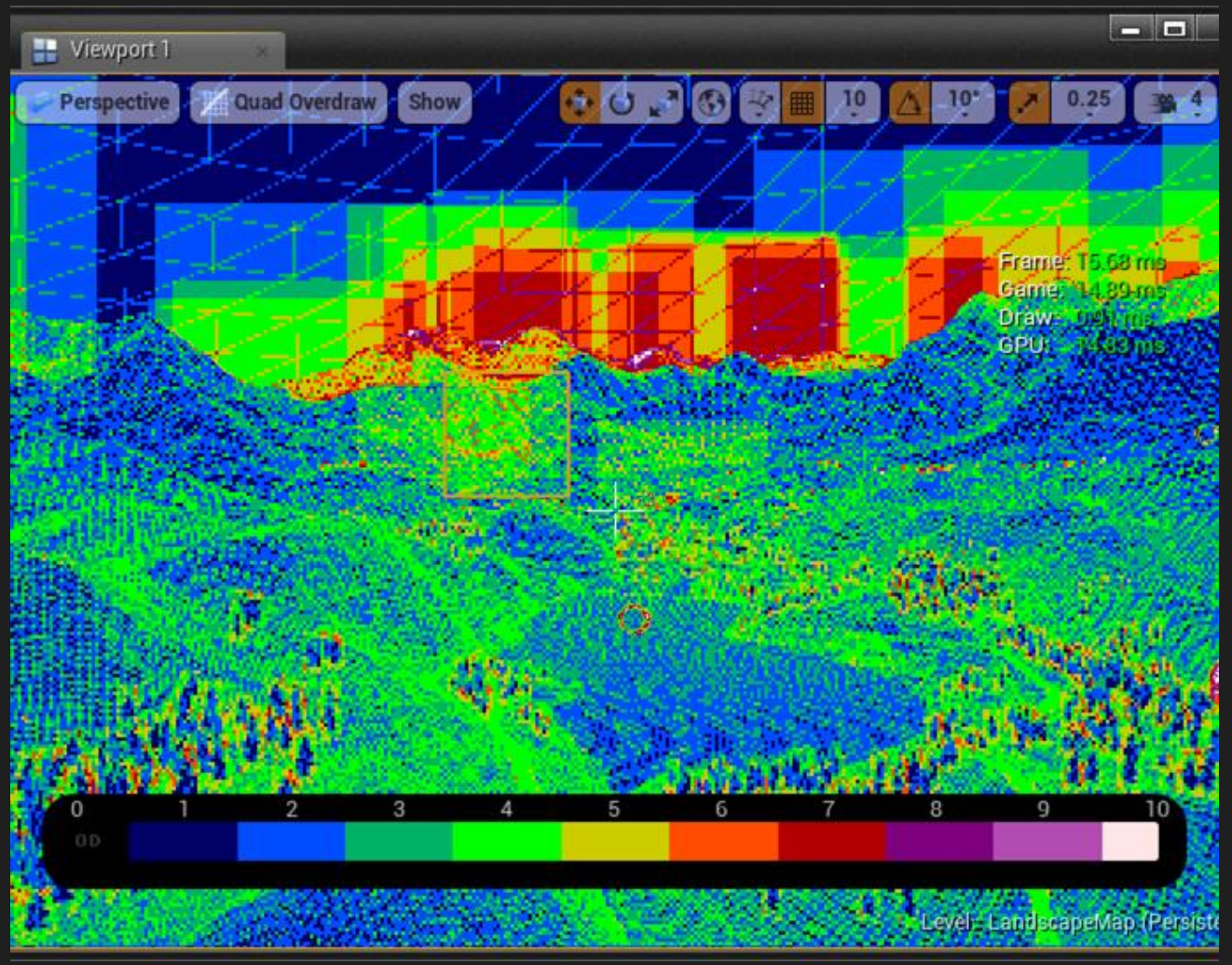
Shader Complexity

- シェーダがGPU上でどれほどのコストになっているのかを表示します
- オーバードローの問題を見る良い方法
  - オーバードローとは、ピクセルを何度も描画することです
  - 最適化のための最も一般的なコンテンツの問題の一つです
- 下部のグラフはピクセル・頂点シェーダのパフォーマンスを表しています
- 赤や白が多く見える場合は、アプローチを再考しましょう

Quad Overdraw

- 比較的新しいビューポート
- 多くのアーティストが見逃す問題を示します
- メッシュをLOD化する必要がある箇所を示します
  - 緑が多い箇所はシンプルにするべきです
  - 緑以上のものはコストがかかってきます一般的に半透明の重なりなことが多いです

- forward rendererで非常に便利です



Quad Overdraw in-depth

- GPUはビューをクアッドに分割します
  - 2*2のピクセルグループ
  - これは全てのピクセルで
    全ての処理を実行するよりも効率的です
- 非常に小さい、又は非常に長い、細いジオメトリがピクセルを無駄にする
  - 通常、大きなポリゴンはピクセルクアッドを
    最大限に活用し、GPUを最大限に活用します
- 正三角形によるモデルと積極的なLOD

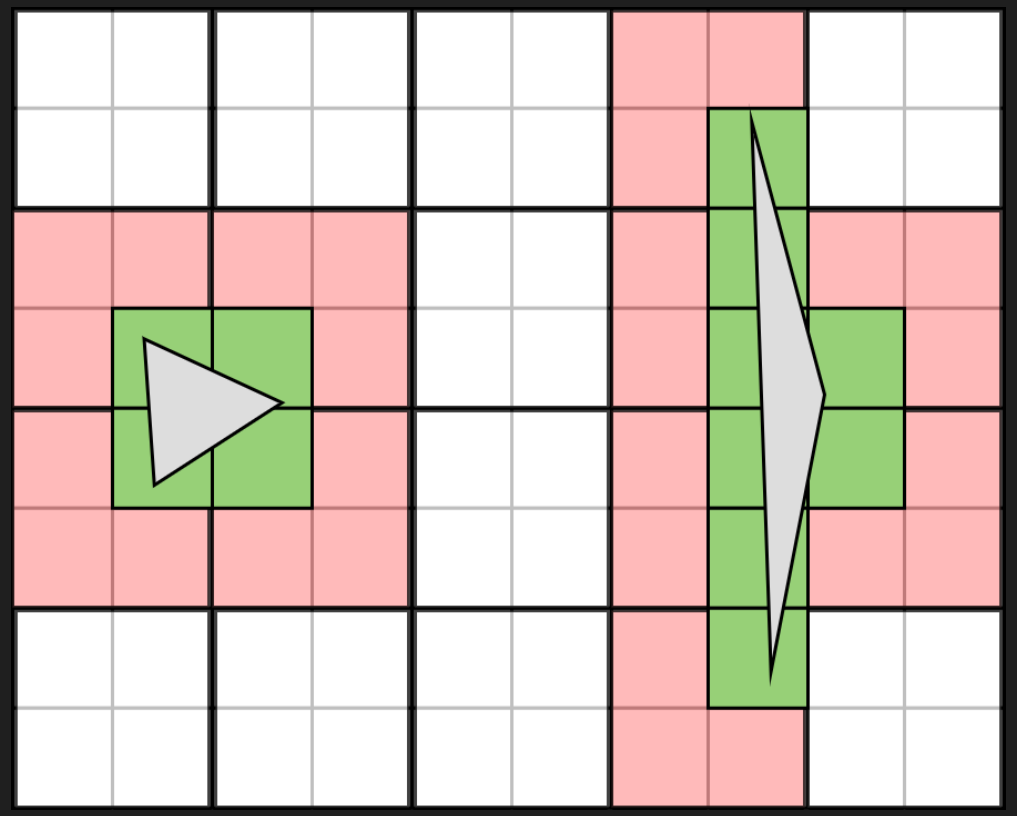
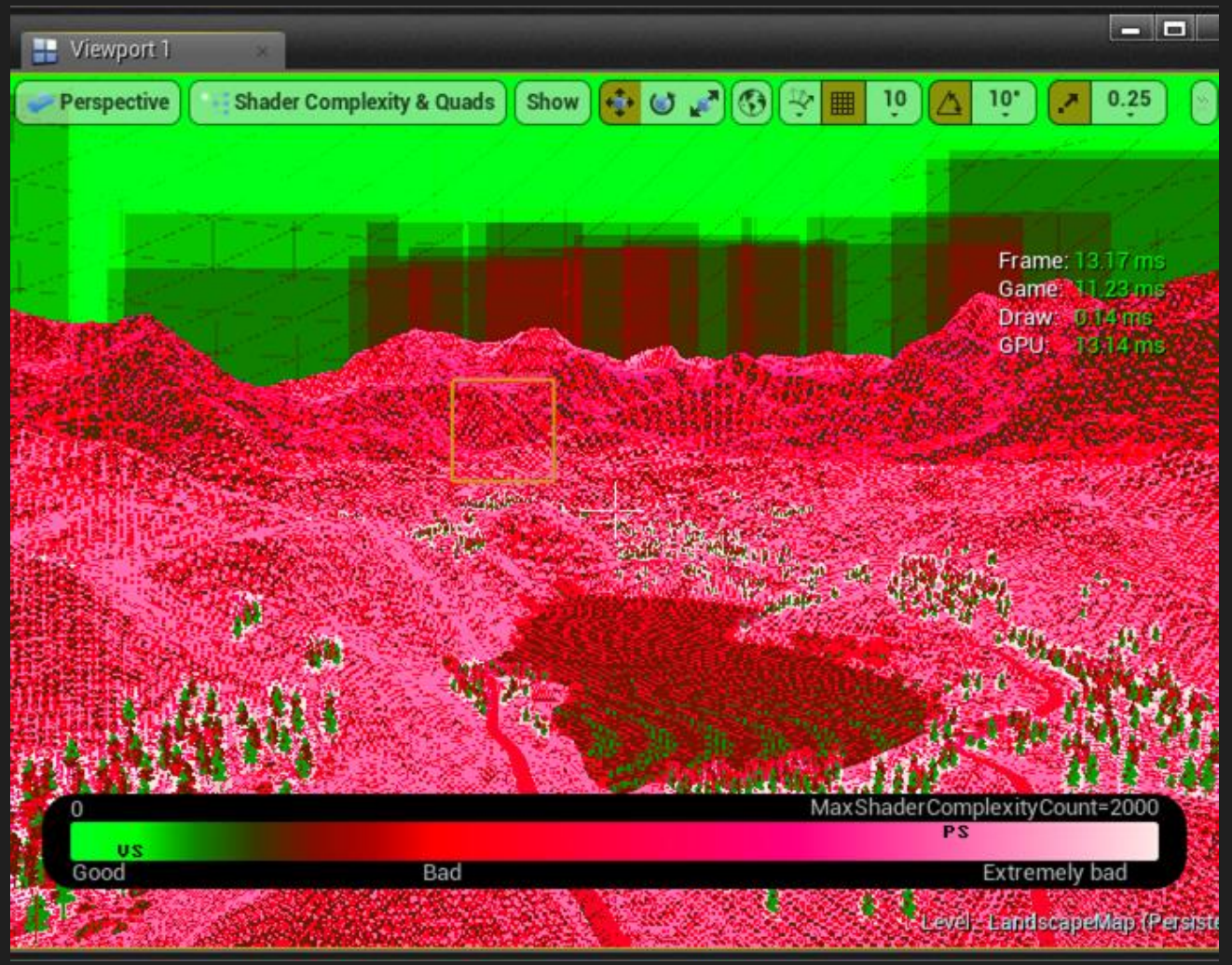


Image copyright fragmentbuffer.com

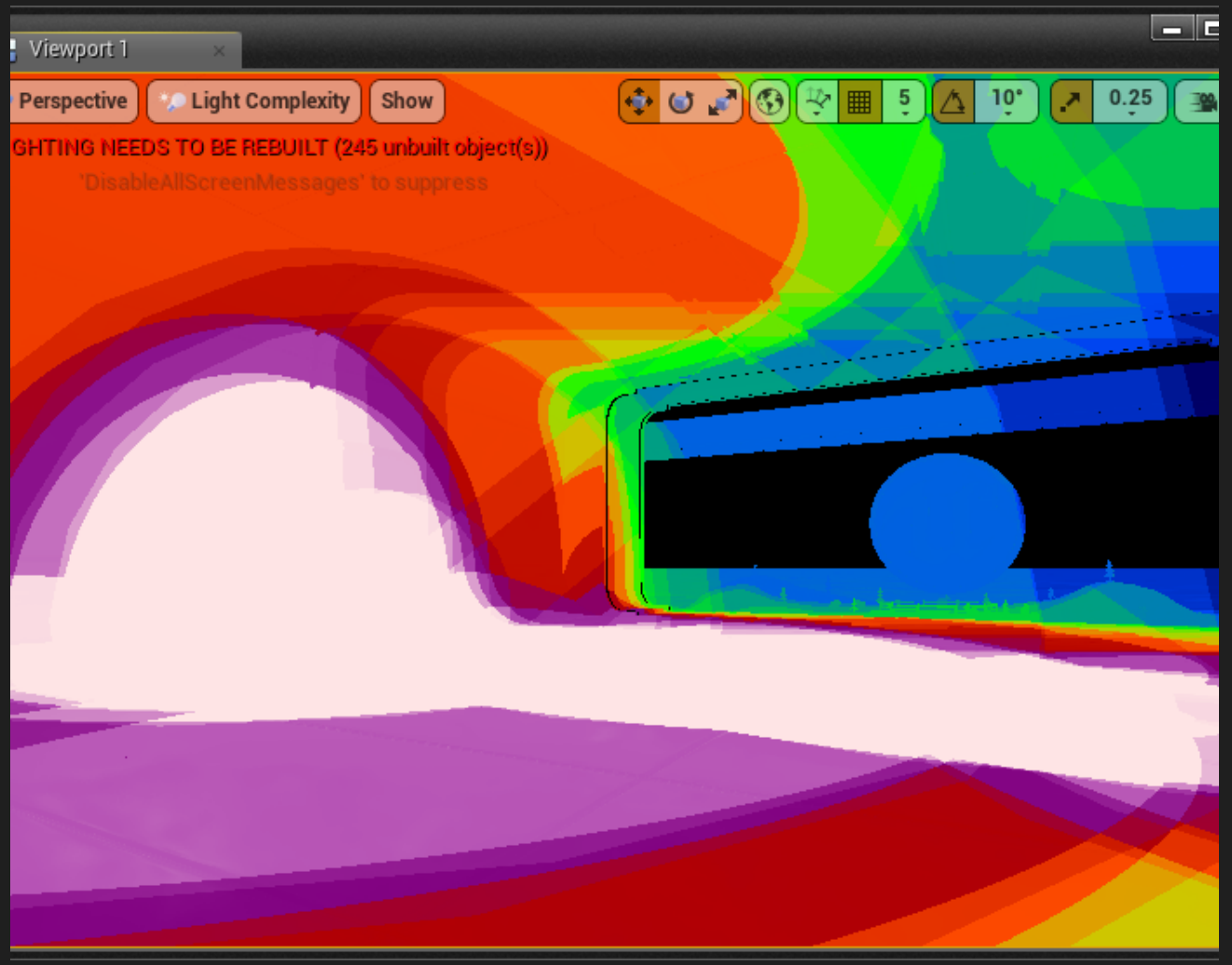
Shader Complexity + Quad Overdraw

- 2つの強力なビューモードを一つに結合
- 高価なシェーダとジオメトリをひと目で把握するのに便利です
- 詳細を計測するためには、まだ個々の設定が必要です



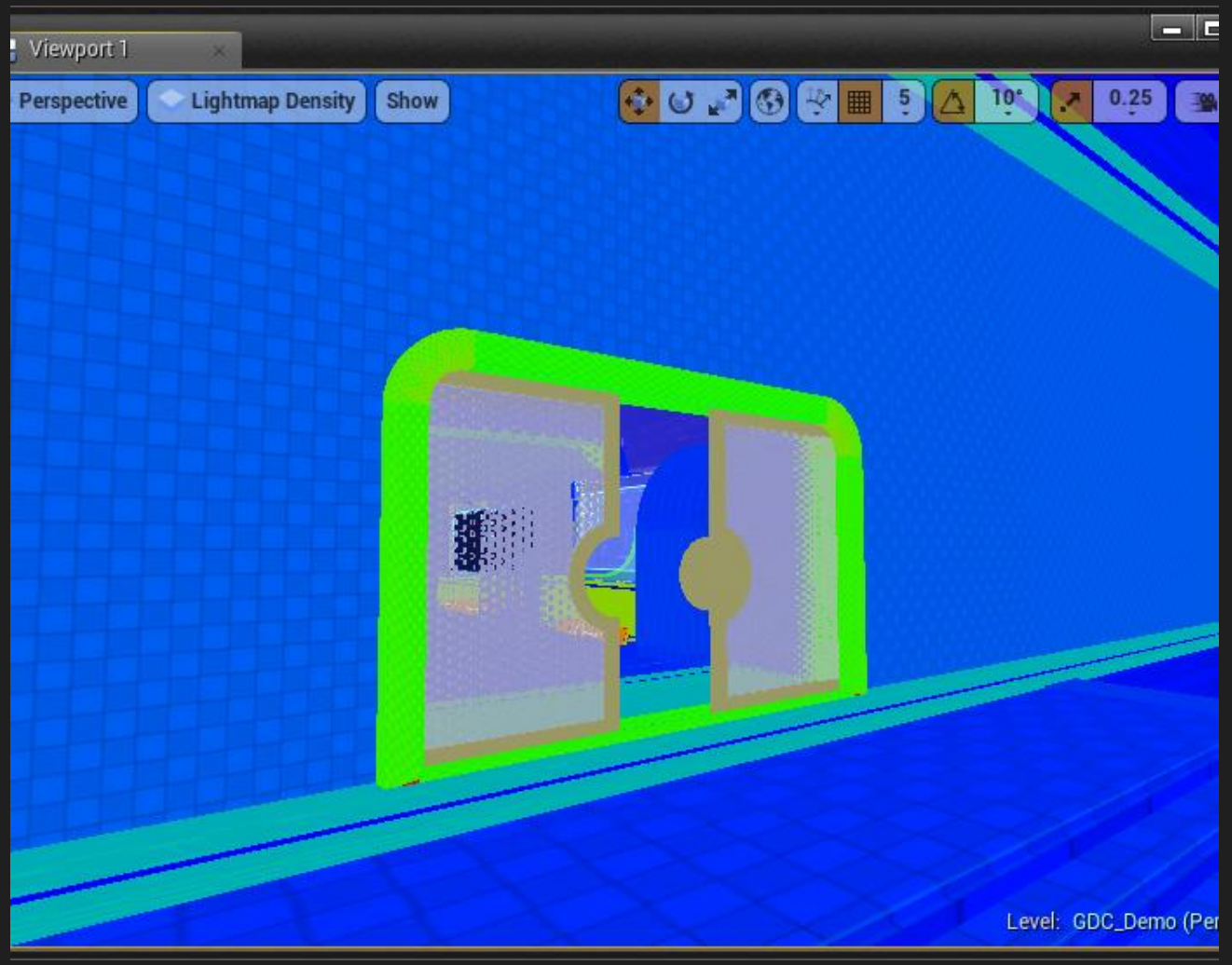
Light Complexity

- シーンのライティングのコストを可視化
- ライトが重なりあうと、寒色から暖色に変化します
- ライティングのコストは表示しますが、シャドウのコストは表示しません
- 明らかに白は悪いです
- ライトの半径を小さくするべきであることがよく分かります
- これを反転することで、特定にライトのコストが「価値がある」かどうかを素早く確認できます



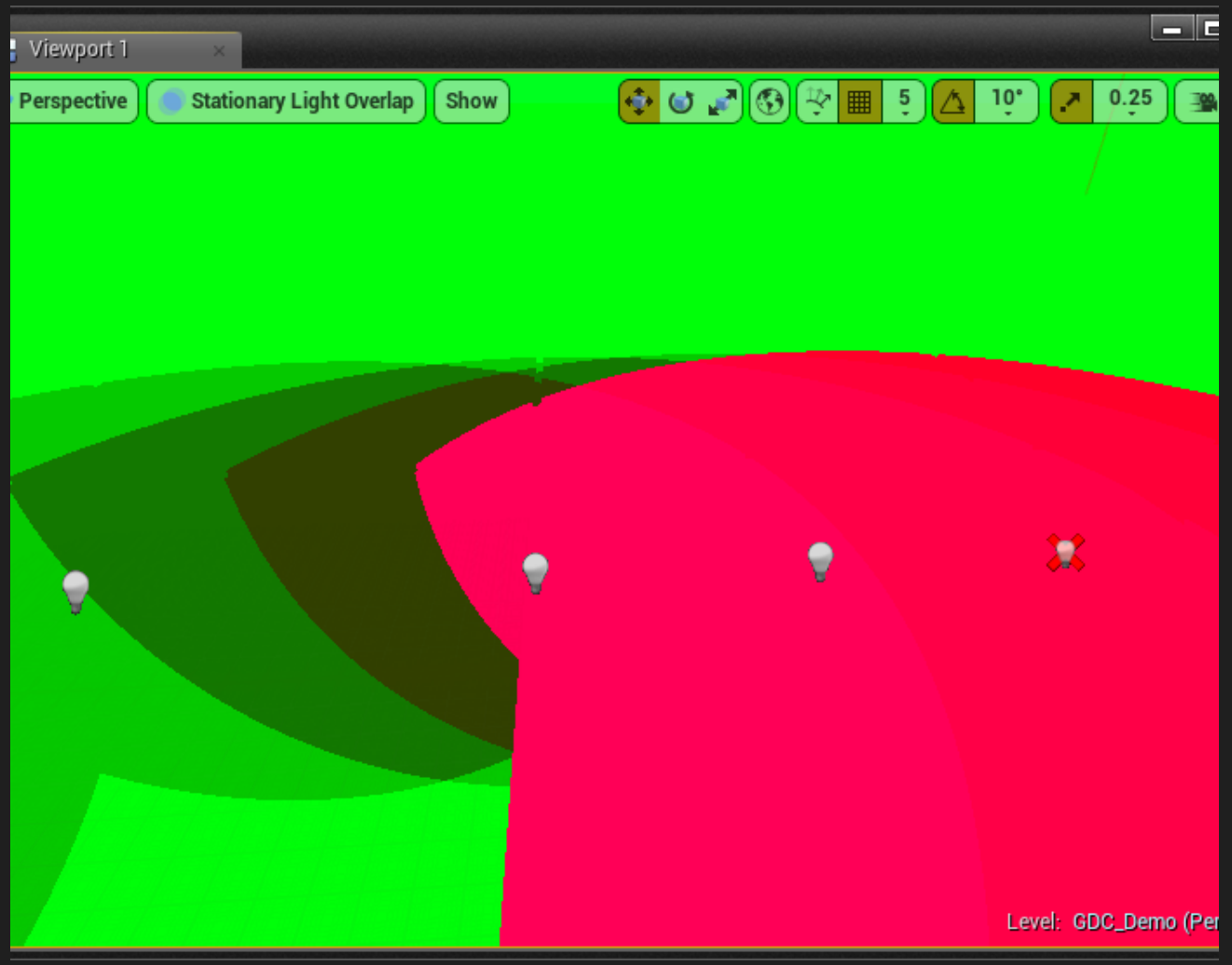
Lightmap Density

- ライトマップのためのテクセル密度を可視化
- 密度が増すにつれて、寒色から暖色に変化します
  - 殆どのものは青にできます
  - シャドーマップは非常に高い解像度にする必要はないことが多いです
- 可能な限り低く保ちましょう
  - メモリ消費量がすぐ増大します



Stationary Light Overlap

- 特定のオブジェクトには最大4つまでのStationaryライトが影響します
- それを超えると、他のライトは全てMovable(完全に動的)になります
- このビューモードはそれがどこで起きているのかを追跡するのに役立ちます
- ライトの半径を可能な限り小さく保つように注意しましょう
- 固定された太陽光を使いますか？
  - それは4つの利用可能なライトの一つを占めます！
  - 可能なときは太陽はOFFにしましょう

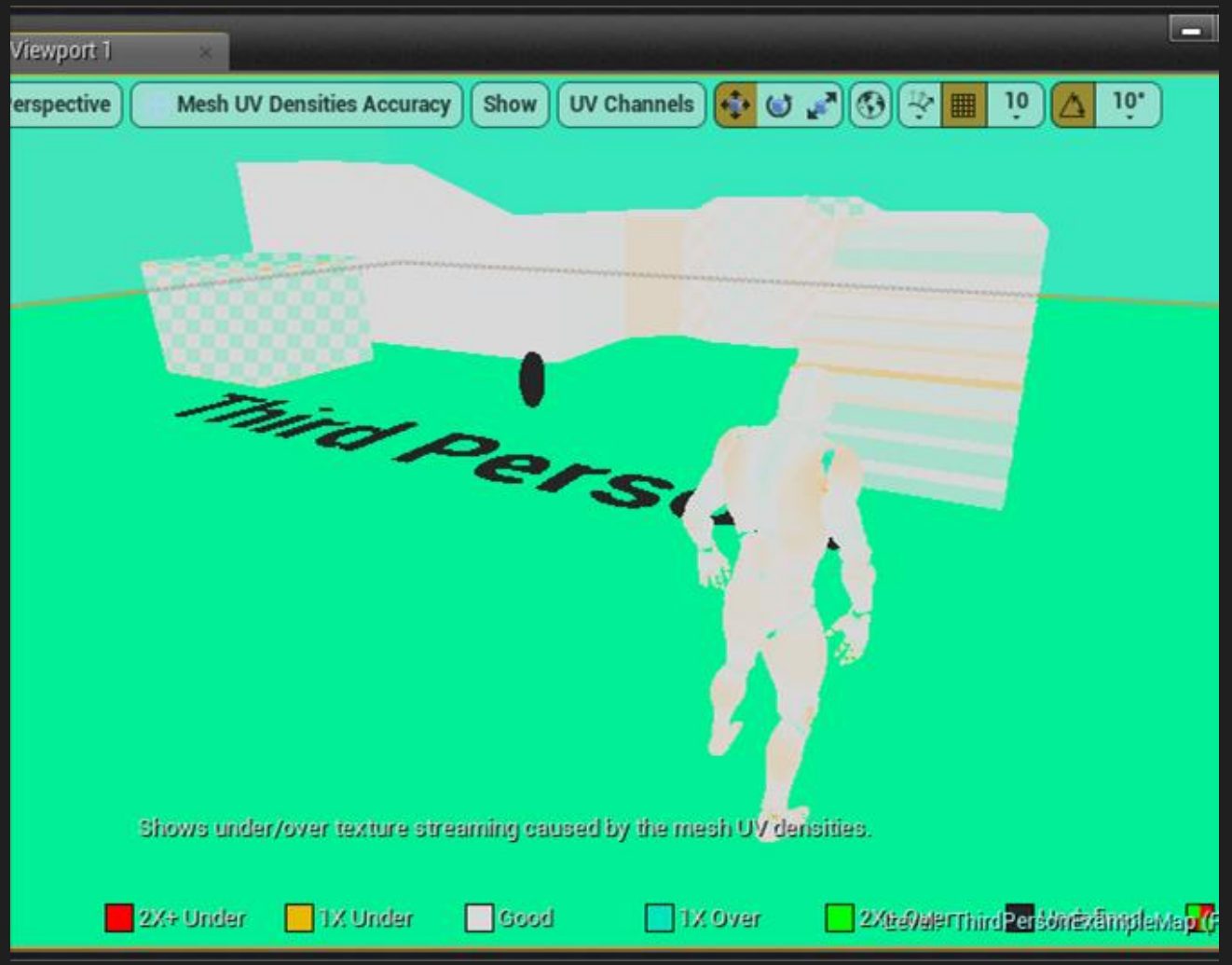


Primitive Distance Accuracy

- テクスチャストリーミングのための可視化システム
- 賢いmip制限を可能にするために、システムで使用する必要があるミップを確認可能にします
  - Red = 2 以上 mipが少ない
  - Orange = 1 mip 少ない
  - White = ちょうどいい
  - Cyan = 1 mip 多い
  - Green = 2 以上 mipが多い
- この設定は *StreamingDistanceMultiplier* プロパティで調整可能です

Mesh UV Densities Accuracy

- これはメッシュのUV密度を使用します
- UV密度がストリーミングデータにどのように影響しているのかを可視化します
- Primitive Distance Accuracyと同じ
  - Red = 2 以上 mipが少ない
  - Orange = 1 mip 少ない
  - White = ちょうどいい
  - Cyan = 1 mip 多い
  - Green = 2 以上 mipが多い
- これを修正するためには、各メッシュのUVを調整する必要があります

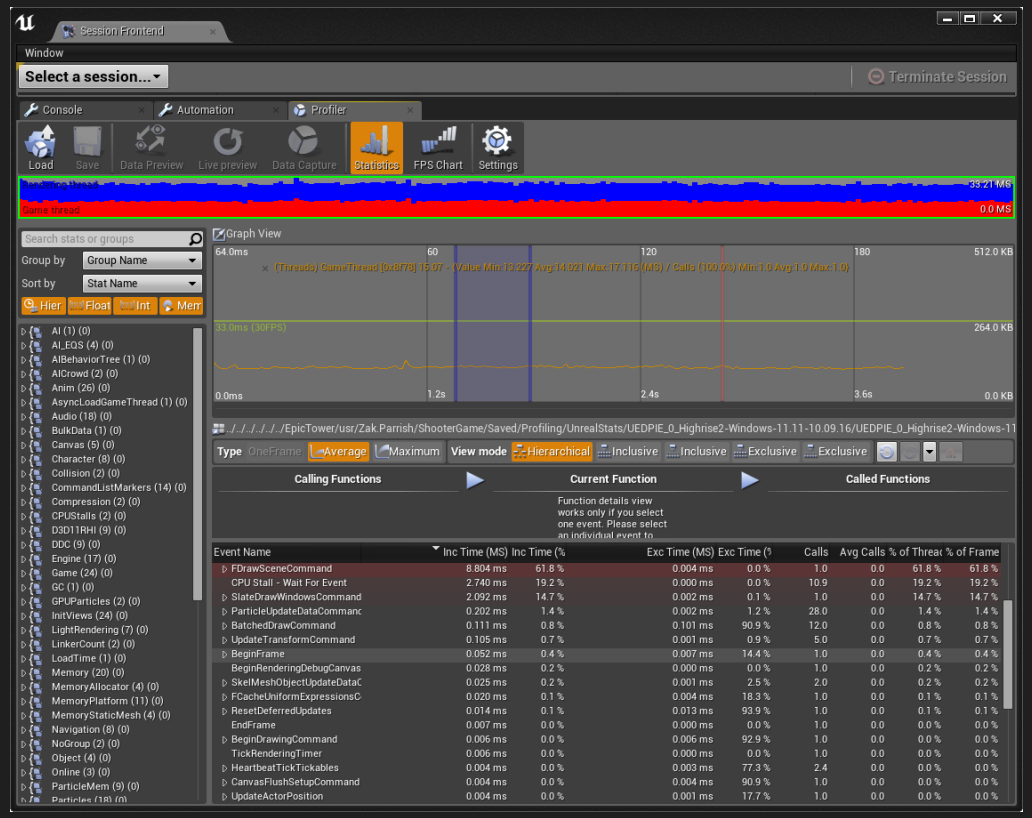


プロファイリングツール

Built-in CPU and GPU profiling

CPU プロファイリング

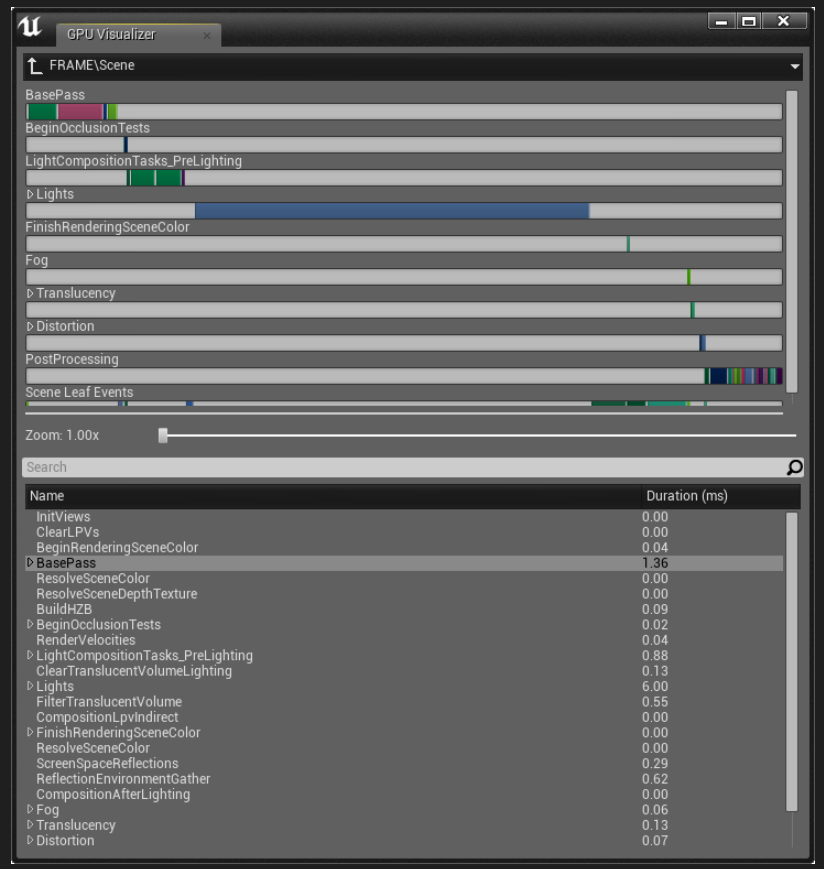
- 統合ツールを使用して、ゲームプレイを分割し、各Tickで何が起きているのかを確認する
  - Blueprintのパフォーマンスをプロファイルする非常に便利な方法です！
- 時間の1区間を測定する
  - その区間における各フレームや平均を見ることができます
- 2つの特殊なStatコマンドが必要です
  - *stat startfile* & stat stopfile
  - これらのコマンドの間におけるログファイルを生成する
  - プロファイラはログの詳細な分析を可能にする
- WorldのTickに潜り、各blueprintの機能を見る
- CPU (Game and Draw) と GPU に使用可能!



GPU プロファイリング

- GPU機能をプロファイリング3つの方法
  - ビューポートでの stat gpu コマンド
  - セッションフロントエンドで記録されたログ
  - GPU Profiler
    - ログか独自のUIにダンプできます

- コストを可視化するための良い方法:
  - Base pass
  - Lighting
  - Shadows
  - Post processing



遅いフレームの追跡

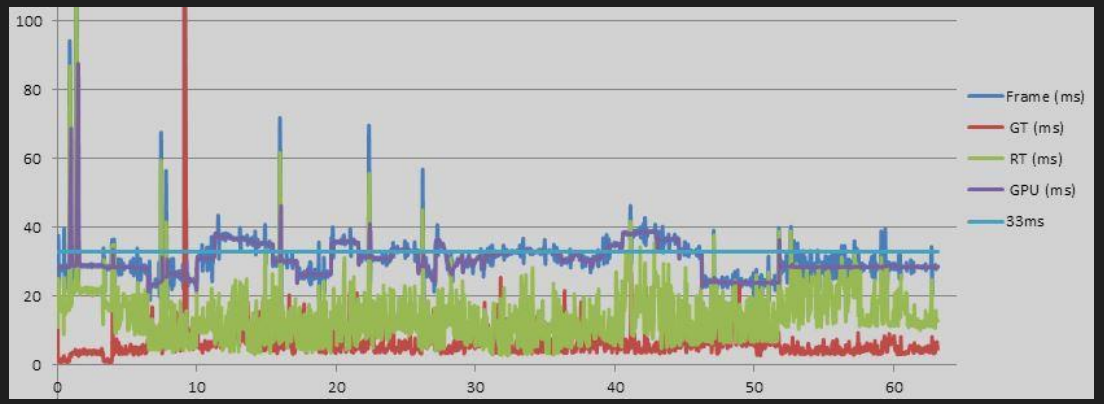
- *stat dumpHitches*
  - このコマンドは、指定された時間（ミリ秒単位）以上のヒッチをログにダンプするために使用されます
  - 値を設定するには、t.hitchThreshhold 0.XX コマンドを使います(デフォルトは0.075)
- *memReport –full*
  - 使用されたメモリ量の内訳
- 以下は、毎週Unreal Tournamentのために生成されたチャートです

	Hitches Per Minute	Hitch Time (% of Gameplay)	Time Played	Peak Memory
UT	0	0.00%	304.75 sec	1230.80 MB

	Average Frame Time	Average Frame Time Delta	% Over 30 FPS	% Over 60 FPS
UT	8.33 ms	**-0.18 ms**	100.00%	99.84%

startFPSChart and stopFPSChart

- *startFPSChart* と *stopFPSChart* を使用して、時間経過に伴うフレームレートの図を作成できます
- これらをレベルシーケンスの開始時と終了時に呼び出すことで指定されたコースに沿ったフレームレートを自動的に呼び出すことができます



Blueprint 最適化

Or: Keeping the Kids from Eating the Crayons

Blueprint のボトルネックと最適化

- Blueprintはゲームプレイロジックを組み立てることを非常に簡単にします
  - 誰もがそのような”力”のための準備ができているわけではない！ ☺
- エンジニアの指導を受けることで最高の結果が得られることが多い
  - 最終的に、全ての人が速くなります
- 共通の課題
  - Tick機能への依存
  - 高価な機能の過剰利用 (多くのオブジェクトを反復)
  - ハードリファレンスの乱用

Tick Blueprintへの依存

- Tick はフレームごとに計算することを意味します
- Blueprints はTickをほとんど必要としません
  - *Class Defaults の Enable Actor Tick* をOFFにしてください
    - これはTickイベントが機能するように、デフォルトでONになっています
- Tickの代替案
  - Timers!
  - Timelines
  - Tick を必要に応じて手動でON/OFFする
- Tickの周期(Tick Frequency)を調整しましょう
  - 0.0 = 毎フレーム

高価な機能

- 一部の機能は非常に高価です
  - Get All Actors of Class
  - Spawn
  - オブジェクトやプロパティの大きなグループを反復処理するもの
- これらを可能な限り使用しないようにしましょう
  - 参照を取得するためにそれを使用する場合は、参照されるクラスが自身を渡すことで、照会する必要がないようにすることを検討してください
  - 配列の代わりにTSetsを使いましょう
- それらを使用する必要がある場合、できるだけあまりしないでください
  - 好ましくは一度だけ、例えばBegin Playのように

Blueprintのハード参照

- Blueprintが互いに参照を生成することは非常に簡単です
- Blueprintをロードした時、それが参照する他のすべてのBlueprintをロードする必要があります
  - そして、それらがさらに参照しているBlueprintも…
  - そしてさらに…さらに…

- これにより、ゲーム内のパフォーマンスが低下することはありませんが、メモリやロード時間が犠牲になる可能性があります
- いくつかのスタジオではエディタの動作が重い
  - 起動時にゲームの大部分（または全て）をロードしていたことが分かります

Blueprintのハード参照 (つづき)

- ハード参照の回避:
  - 必要かつ問題を起こさない場合以外では、Cast操作は避けてください
    - 例えば、PickupクラスがPlayerのみとしか対話しない場合は、それはうまくいくかもしれません
    - しかし、Playerクラスが他のPickupタイプの参照も持つ場合、問題が発生する可能性があります
  - 代わりに、Blueprint インターフェイスを使いましょう
  - 参照を必要としないよう考えるようにしましょう
    - インターフェイスを介して、より一般的なクラスにメッセージを送信
    - それらがインターフェイスを実装する何かにたどり着けば素晴らしい！
    - そうでなければ、大きな問題はない

その他のBlueprint 最適化

- いずれのことをあまりしないように（スクリプト言語のように）
  - 単一のクラスに機能を入れすぎない
    - 分割しましょう
    - クラス階層を使用しましょう
      - しかし、深すぎるクラス階層は避けましょう
  - クラス内に大量のコンポーネントを置かないように
  - あまりにも高価な数学処理
    - Math Expression ノードを使いましょう – 高速化のために最適化されています
  - BPのパフォーマンスがこれでも改善しない場合は… GO NATIVE!
    - Epicでは、Blueprintの多くは汎用的なC++クラスから発生しています
    - あなたも是非そうするべきです
    - 全ての重い処理はコードに残して、Blueprintには軽量なものを残しましょう

どのActorがTickしてる？

- 何がTickしていたのか見逃しましたか？ *dumpticks* を使いましょう
- 全てのTick Actor のリストをログにダンプし、Tickが何回呼ばれたのかを示します
- また、シーン内にどれほどの有効 / 無効化された Tick Actorがいるのかも示します

```
[0083.62][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_cars_straight_line_6.ParticleSystemComponent0[TickComponent], Enabled, ActualTickGr
oup: TG_DuringPhysics, Prerequesities: 0
[0083.62][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_cars_straight_line_7.ParticleSystemComponent0[TickComponent], Enabled, ActualTickGr
oup: TG_DuringPhysics, Prerequesities: 0
[0083.62][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_cars_straight_line_8.ParticleSystemComponent0[TickComponent], Enabled, ActualTickGr
oup: TG_DuringPhysics, Prerequesities: 0
[0083.62][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_waterfall_mist_14.ParticleSystemComponent0[TickComponent], Enabled, ActualTickGroup
: TG_DuringPhysics, Prerequesities: 0
[0083.62][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_lightning_4.ParticleSystemComponent0[TickComponent], Enabled, ActualTickGroup: TG_D
uringPhysics, Prerequesities: 0
[0083.62][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_clouds_highrise_sky_4.ParticleSystemComponent0[TickComponent], Enabled, ActualTickG
roup: TG_DuringPhysics, Prerequesities: 0
[0083.63][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_clouds_highrise_sky_5.ParticleSystemComponent0[TickComponent], Enabled, ActualTickG
roup: TG_DuringPhysics, Prerequesities: 0
[0083.63][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_lightning_2.ParticleSystemComponent0[TickComponent], Enabled, ActualTickGroup: TG_D
uringPhysics, Prerequesities: 0
[0083.63][905]ParticleSystemComponent /Temp/Autosaves/Game/Maps/UEDPCHighrise_Vista.Highrise_Vista:PersistentLevel.P_lightning_3.ParticleSystemComponent0[TickComponent], Enabled, ActualTickGroup: TG_D
uringPhysics, Prerequesities: 0
[0083.63][905]
[0083.63][905]Total registered tick functions: 87, enabled: 77, disabled: 10.
[0083.63][905]
```

Draw Thread 最適化

Dealing with too much stuff

CPU Rendering に関する考慮事項

- Draw Threadのボトルネックは、
  あまりにも多くのことをすることで発生することが多いです:
  - 大量のドローコール
  - オクルージョンクエリ
  - 大量のパーティクルのシミュレート
  - 多くのライトの追加 – GPUにより影響する事が多い
- Draw Threadを高速化する最も良い方法は、減らすことです
- 画面上のものを少なくするための方法を見つけましょう
- 一般的に、これはコンテンツを整理するか、
  またはUE4内の統合ツールを使用して、オブジェクトの結合を
  開始することを意味します

Actor 結合(Merge)ツール

- *Window* > *Developer Tools* にあります
- 選択したメッシュを結合し、新しいアセットに置き換える
- Simplygonを介してマテリアルを結合することもできます
  - Simplygonがなくても標準機能で結合することも可能です
- 同じマテリアルを持つメッシュは良い働きをします

Instanced Static Meshes

- 与えられたメッシュの複数のインスタンスを生成する仕組み。
  同じメッシュオブジェクトの一部として考慮される
- 現時点では、コードまたはBlueprintを介してのみ作成することができます。たいていはConstruction Scriptを介します。

- これを生成するのに役に立つBlueprintを作成するのは非常に簡単です
  - メッシュがどこに配置されるのかをプレビューするBlueprint
  - プレビューBPからTransformを収集する 半径ベースのISM BP
  - Editor Utility class の BPには注意！ – エディタのみです!

- *Hierarchical Instanced Static Meshes* も考慮しましょう
  - 独自のオクルージョン / Visiblityを扱います

Hierarchical LOD

- Hierarchical LODを使用すると、複数のメッシュを結合し、一つのメッシュとして縮小することができます
- テクスチャをアトラスに組み合わせて、全体のマテリアル要求を減らします
- 非常に遠い距離で見る大きなメッシュのグループ、建物や都市に非常に便利です

GPU 最適化

What to do about all those pixels

GPU Pipeline Overview

- Preprocess
- 頂点シェーダ
- ピクセルシェーダ

頂点シェーダの最適化

- World Position Offsetをどの程度使っているかに注意して下さい
  - 頂点アニメーションの代替方法よりもしばしば安価です
- 頂点カラーは最終的には効果になる可能性がある
  - Paragonの場合, それを削除して、インスタンスごとに追加しました

ピクセルシェーダの最適化

Pixel Shader Don’ts

- 大量の数学処理
- 大量のテクスチャ
- 大量のプロシージャル関数
  - Noise
- 大量のMaterial layers
- If への 依存
  - 両方共計算する必要がある

Pixel Shader Dos

- 数学処理の代わりにテクスチャのルックアップを使用する
- グレースケールに圧縮して、一つにテクスチャにまとめる
- レイヤーの使用を最小限に抑える
- Switchパラメータを使用して、必要のないものをOFFに

Material Instruction Count

- Material instruction Countには常に注意しましょう
- 注意: 表示された数値は、Applyをクリックするまでは正確ではありません
  - 時には安全のためマテリアルを再コンパイルするのが良いです

- Paragonの場合、 Material Instructionのバジェットをマテリアル毎に維持（しようと）しました
  - キャラクタ: 350 instructions
    - これは画面上の3人称サイズによるものです
  - 環境: 150-200 instructions (255 limit)
  - 注: これはParagon、プラットフォーム、必要とされたパフォーマンスに非常に特化した値です。あなたの予算とは異なる場合があります

Overdrawの取扱い

- Overdraw はGPU負荷の主な原因の一つです
- Overdrawのために、ジオメトリ領域を最小化しましょう
  - Overdrawに頼るよりも、頂点を追加したほうがずっと安価です
  - 例えば、カイトと少年では、芝生のテクスチャアルファの輪郭にほぼ正確に一致するように芝生の面をカットしていました
- Particle Cutout プロパティを使用する！
  - Cascade Required Module以下にあります
  - テクスチャにそれを与えることで、自動的にスプライトを切り取ります
  - subUVでも動作します、フレームごとに異なるカットアウトがあります

テクスチャ解像度の管理

- 好きな解像度でテクスチャを作成しても、常にフルの解像度を使用するとは限りません
- Texture Streaming viewsを使用して、任意のテクスチャに使用しているmip levelを確認してください
- Texture Statisticsに設定された統計パネル (*Window* > *Statistics*)を使用して、現在のレベルで使用しているmip levelを確認できます
- そのあと、テクスチャエディタを使ってmip biasを矯正します
- または、より低い解像度で再インポート☺

ライティングに関する考慮事項

- 動的なライトは高価です (ディファードでは少し安価です)
  - 小さく、影を落とさないライトが最も安価です！
  - これらを多く持つことができます！
- 動的なライトの数を最小限に
- 動的なライトが影響を与える物の数を最小限に
- 動的なライトの半径を最小限に – タイトな方が良い
- 可能な限り、動的なライトからの影は最小限に
  - UEにおいて、ライトからの動的な影は最も高価です
- Stationary Lightの重なりに注意して下さい
  - 動的ライトへの後戻りは非常に高価です

ライティングに関する考慮事項 (つづき)

- できるだけベイクしましょう！
  - ベイクには時間がかかりますが…ごめんなさい…
- 動的ライトを必要とは思わないでください
  - コンテンツが静的な場合は、おそらく静的ライトを使用できます
- Mesh Distance Field shadowsを使用しましょう
  - Fortniteで使いました – 多くの時間を節約しました
- 密なカスケードシャドウに注意してください
- シャドウバイアスで多くのアーティファクトはクリーンアップされます
- 可能な限りライトマップ解像度は低く抑えてください
  - ビューモードを使用し、可能な限り青にします

ライティングに関する考慮事項 (つづき)

- ライトファンクションは本当に必要でない限り避けましょう
  - IESプロファイルもコストがあることを理解して検討しましょう
- 透過ライティング も高価です、注意して下さい
- 影を早めに消しましょう(できるだけ近い距離に)
- 可能な限り動的ライトは消しましょう
- スポットライト は ポイントライトよりも安価です
- フェイクシャドウを恐れないで！
  - 私達はこれを沢山使ってます、特にVRのために！

デバイスのための最適化

Making the most of your content

デバイスのための最適化

- Unreal Engine は “一度ビルドすると, どこにでも展開する ” というゲーム開発アプローチを推進しています
  - 全てのものが等しく、常にうまくいきます >:D
- UE4の多くのコンテンツツールには、Cook時に自動的に縮小する賢いコンテンツ製作のためのオプションがあります
  - これにより、コンテンツの見栄えがよくなったり、様々なサポートしているプラットフォームで正常に動作しますPrevents having to build content multiple times
- EpicではFortniteとParagonにこれらのツールを幅広く使用しています
  - Paragon はPS4で60Hz 1080pで動作する必要があります

デバイスプロファイル

- デバイスプロファイル エディタを使用すると、デバイス毎に様々なコンソール変数(cvars)を制御できます
- これにより、コンソールやモバイルのような性能が低いデバイスでよりシンプルな車道、低品質なMipmap、偏ったメッシュLODなどを使用することができます
- これらのcvarsを設定すると、Cook処理中にアセットの品質が制御されます
- *Window > Developer Tools > Device Profiles*

デバイス毎のマテリアル最適化

- マテリアル内で *QualitySwitch* ノードを使用します
- これにより、異なるレベルにグラフを分けることができます
- 複数の *QualitySwitch* ノードが使えます。各主要Material inputにも (Base Color, Roughness, etc.)
- r.*MaterialQualityLevel* cvar を使用して、プロジェクト全体の品質を設定できます
- Paragonの場合
  - PC は High quality
  - PS4 はMed quality

ネットワーク 最適化

How to properly use the Series of Tubes

Replication 最適化

- ネットワークに関する一般な問題:
  - 多すぎ
  - 頻繁すぎ
- Replicateは可能な限り小さく、低頻度に！
- *net.** コマンドを使用して何が起きているのか確認
  - サーバ上で実行する必要があります！
  - *net.DumpRelevantActors* を使って、現在Replicate中の物を確認
    - このコマンドは 4.19 で改善されます
  - net.* コマンドは沢山あります – 全てのリストはドキュメントで

ネットワーク診断機能

- 100人プレイのネットワーキングゲームを制作したおかげで、Replication問題を追跡するための新しい Toy が手に入ります
- *stat net*
  - これは、新しく信頼できるデータを表示するために4.19で改善されました
  - 最終的にはクライアントから確実に使用できます!

Network Relevancy ビューモード (4.19)

- 青= 休止
- オレンジ = 休止してませんが、チャンネルがない
- 緑 = Replicateするチャンネルがある
- 赤 = 常に関連する
- 白 = 距離で間引かれる (これらの多くはまだ見える).
- 基本的に、オレンジは休止の候補であり、プロパティの差分チェックにCPUを使っていますが、Replicateはしてません

コンテンツ ストリーミング

Control yourself; take only what you need from it

Level Streaming

- レベルストリーミングはゲームで使用されているコンテンツを制御する理想的な方法です
- 現在必要なものがストリーミング INし、不要なものはストリーム OUTする
- 一度にどれほどストリームするか注意してください！
  - プレイヤーを何らかの方法で制限するか、小さなチャンクでストリームする必要があります
- コード、Blueprint、又は Level Streaming Volumeを介して、レベルストリーミングを制御できます
- 注意: コードやBlueprintでコンテンツを過度に参照してしまった場合、いくつかのメリットが打ち消されます

おまけ: コラボレーションとしてのレベルストリーミング

- レベルストリーミングは、レベルデザイナーとアーティストが一緒に働く主要な方法です
- 異なる層が異なるレベルに別れる
  - 物理的な領域が異なるということではない
- Paragonにおける例 (簡略化):
  - Persistent
    - Lighting
    - Base Geometry
    - Collision
    - Foliage
    - Gameplay – これには全てのタワー、Spawner、MOBA要素が含まれる

World Composition

- 広いワールド向けの専用ストリーミングシステム
- 過去のLevel Streaming volumeは機能しません
- しかし、Blueprintによるストリーミングは動作します
  - Pro Tip: Level Streaming Volumeのような機能を持つBlueprintを簡単に作れるし、同じように動作します