広がる可能性: Arm Mobile StudioのUnityとの統合

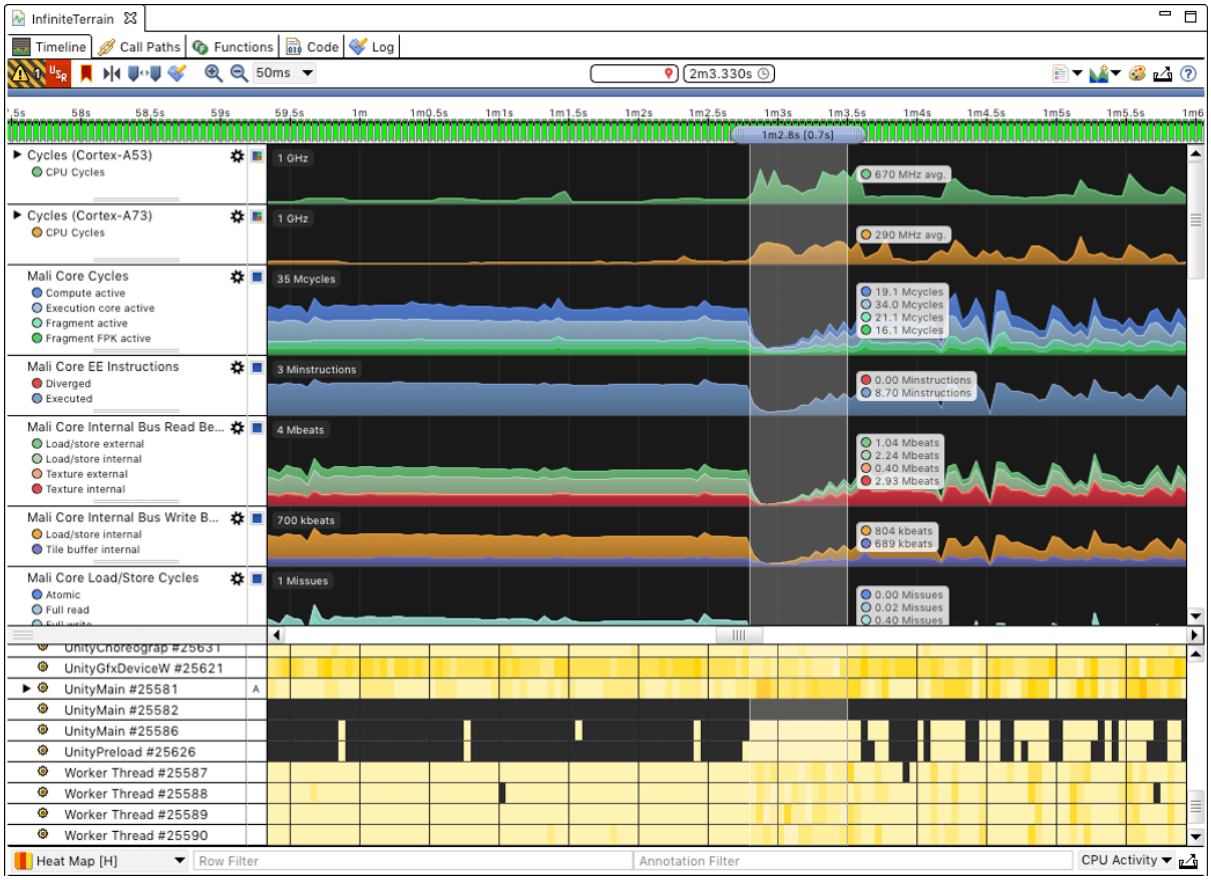
モバイルゲームの開発者は、最新のハイエンドプレミアムスマートフォンから量販品や旧式の製品までの幅広いデバイスで、コンテンツがうまく動作するように尽力しています。モバイルゲームのコンテンツが複雑化するにつれて、開発者は、フレームレートを安定させる一方で電力消費を抑えるために必要な有益な情報を提供してくれる、良質のツールに頼るようになっています。

このブログでは、新しいArm Mobile Studioのツールのコレクションが、Androidのパフォーマンス分析にどのように役立つか、そしてゲームエンジンとどのように連携して、より説得力のあるパフォーマンス分析能力を生み出すことができるかについて説明しています。Arm Mobile StudioのStarter Editionは無償で利用でき、このブログで使用しているサンプルプロジェクトのソースコードはgithubで入手できます。

このブログでは、Arm Mobile Studioの中で最も汎用的かつ最も詳細なパフォーマンス分析コンポーネントであるStreamlineに注目することにします。モバイルデバイス上のArm Cortex-A CPUやArm Mali GPUのリソースがゲームのさまざまな局面でどう利用されているかを実際に確認するために、StreamlineとUnityをどのように統合するかについて説明します。

Streamlineについて

Streamlineは、Androidデバイス上のいくつかの情報源からサンプルベースおよびイベントベースのパフォーマンスデータを収集して、その集計結果をさまざまなビューに表示します。それらのビューのうち、このブログではタイムラインビューに注目しています。画面の上半分には収集したシステムパフォーマンスカウンタが表示され、下半分にはさまざまな種類の情報が同じタイムラインで表示されます。次の図は、演算アクティビティーがプロファイリング対象アプリケーションのスレッド間でどのように分散されているかを表すヒートマップを示しています。



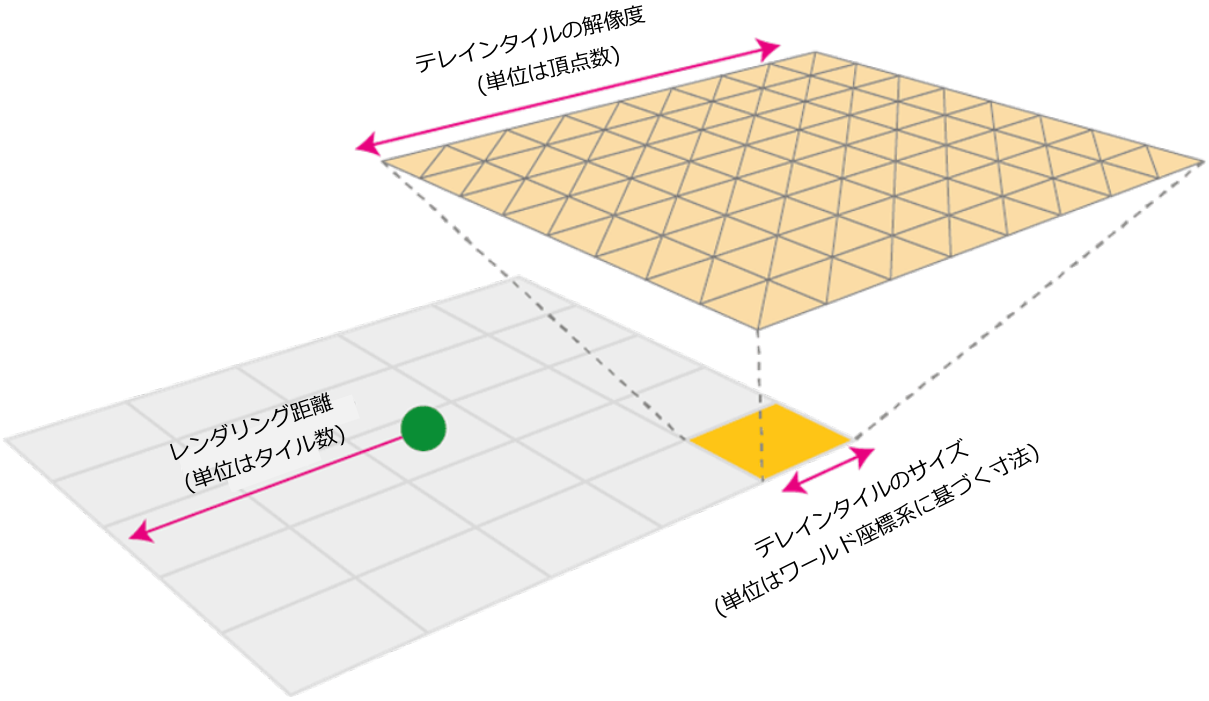
分析対象

ここでは、非常にシンプルなUnityコンテンツである、手続き型で生成されたテレインのフライスルーを分析します。カメラは曲がりくねりながら進み、必要に応じてオンザフライで新しいテレインタイルが生成されます。カメラから離れすぎたタイルは削除されるため、いったんシーンが満たされたあとは、時間が経過してもシーンの複雑さはおおむね同じ状態のままです。カメラがゆっくり移動するときは新しいテレイン生成の速度は非常にゆっくりであり、カメラの移動速度が上がったときはテレイン生成の速度も上がる必要があります。各タイルのエッジは暗めの色でレンダリングされているため、各タイルのサイズを確認できます。

テレインタイルの生成は多くの演算を必要とするため、Unityのメインスレッドの実行を妨げないバックグラウンドスレッドをディスパッチできるUnity Job Schedulerを使用します。そうすることで、新しいテレインが生成されるたびにぎくしゃくした動きになることはなく、フレームレートが安定します。

このデモは、4つの異なるシーンを通過するように設定されています。各シーンはまったく同じように見えますが、テレインタイルの生成方法が異なっています。次の図に示しているように、このテレインは多数のテレインブロックで構成されていて、各ブロックは固定サイズです。各ブロックはいくつかのメッシュ (緑色のテレイン用に1つ、黄色のテレイン用に1つ、水面用に1つ) で構成されてい

て、各メッシュは固定解像度です。プレイヤーの周囲に生成されるタイルの数は、レンダリング距離によって制御されています。



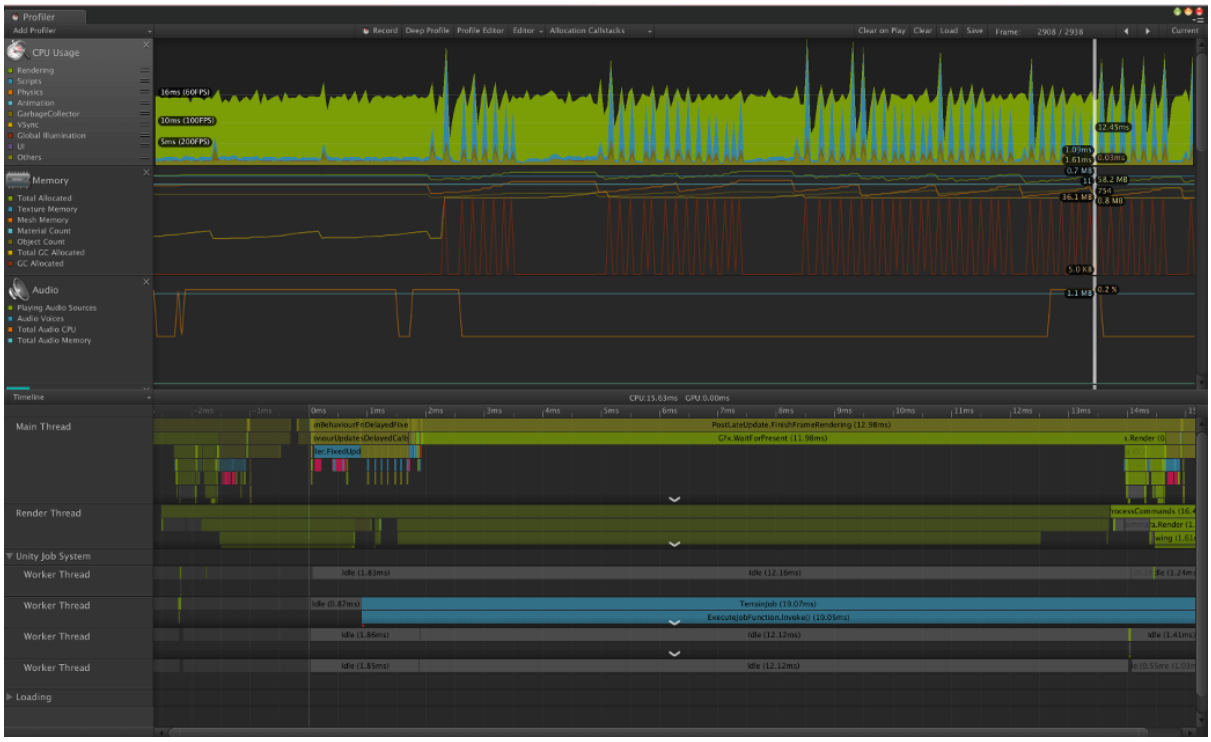
4つのシーンは次のように設定されています。

シーン	レンダリング距離	テレインタイルのサイズ	テレインの解像度	テレインの並列生成数
1	3	20x20	32x32	8
2	3	20x20	32x32	1
3	6	10x10	16x16	8
4	6	10x10	16x16	1

2017年にリリースされたHuawei P10スマートフォンでプロファイリングアクティビティーを実行します。このスマートフォンには、4基の高性能なArm Cortex-A73 CPUコア、4基の高効率のArm Cortex-A53 CPUコア、およびArm Mali-G71 MP8 GPUで構成されているHiSilicon Kirin 960チップが搭載されています。

Unityでのプロファイリング

Unity自体にはプロファイラーが含まれていて、Androidデバイスでは非常に役立ちます。



Unityのプロファイラーでは、ジョブがいつスケジュールされているかは明確に示されますが、プラットフォームの物理リソース (このブログの目的であるCPUやGPU) や、それらがどのように使用されているかは示されません。60FPSに到達することもあるようですが、すべてのCPUコアが限界まで使われているでしょうか、そしてそのためにバッテリが消耗されているのでしょうか?そこでStreamlineの出番です。このブログの残りの部分では、Streamlineがどのようなデータをキャプチャおよび提示するか、Streamlineのアノテーション機能をどのように使用して、Unityゲームからの高水準のコンテキストをStreamlineに渡してデータを解釈しやすくできるかについて説明します。

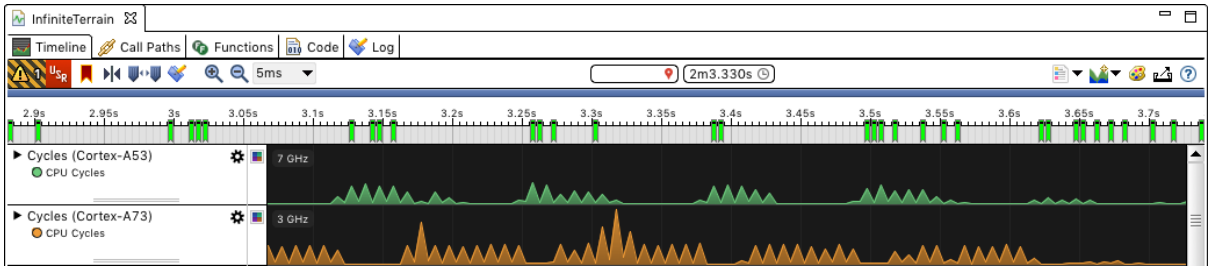
StreamlineによるUnityのプロファイリング - その機能

Unityゲームにアノテーションをどのように挿入できるかについて説明する前に、Streamlineの3つのアノテーション機能を利用するようにゲームを変更した場合、サンプルコンテンツの最終的な結果がどのように見えるかを確認しておきましょう。

- **マーカー**は最もシンプルな形式のアノテーションであり、Streamlineのタイムラインビューに表示される一時点に対するラベルです。
- **チャネル**は、より構造的で、各スレッドに沿った行ごとに情報を提供します。アノテーションをチャネル内に配置でき、マーカーとは違って、各アノテーションは一定時間に及びます。
- **カスタムアクティビティーマップ**は最も高度な形式のアノテーションであり、複雑な依存関係が存在することがあるグローバルな (スレッド間の) アクティビティーを示すためのメカニズムです。各カスタムアクティビティーマップは、StreamlineのUIの下半分に独自のビューとして表示されます。

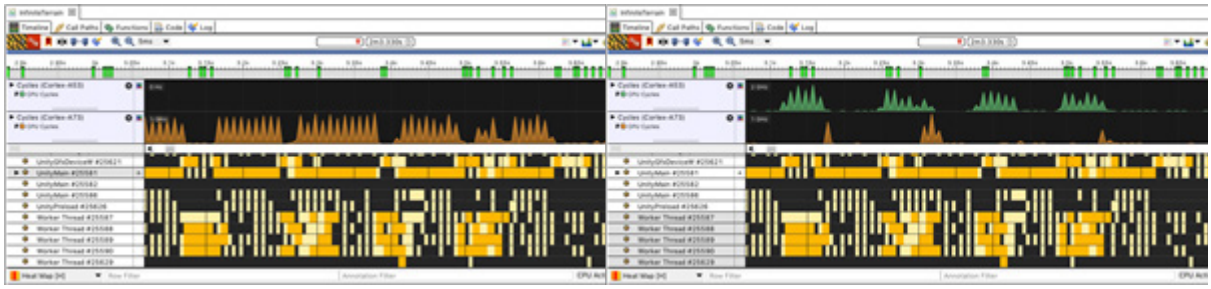
1つのプロファイルが収集されると (その後もプロファイルの収集は続行される)、Streamlineに表示されるので、何が起きているのかを把握し始めることができます。

コンテンツを調べるときに、まず目を引くのはマーカー (タイムラインの上部に緑色で表示) であり、この例では各フレームが始まっている位置を表しています。



フレームレートが規則的ではなく、すべてのCPUコアにわたって相当量の集中的なアクティビティーが見られます。フレームレートがゆっくり始まってから、上昇し、ときどき休止しています。それはテレイン生成から予期されることとかなり一致していますが、さらに深く調べることはできるでしょうか。

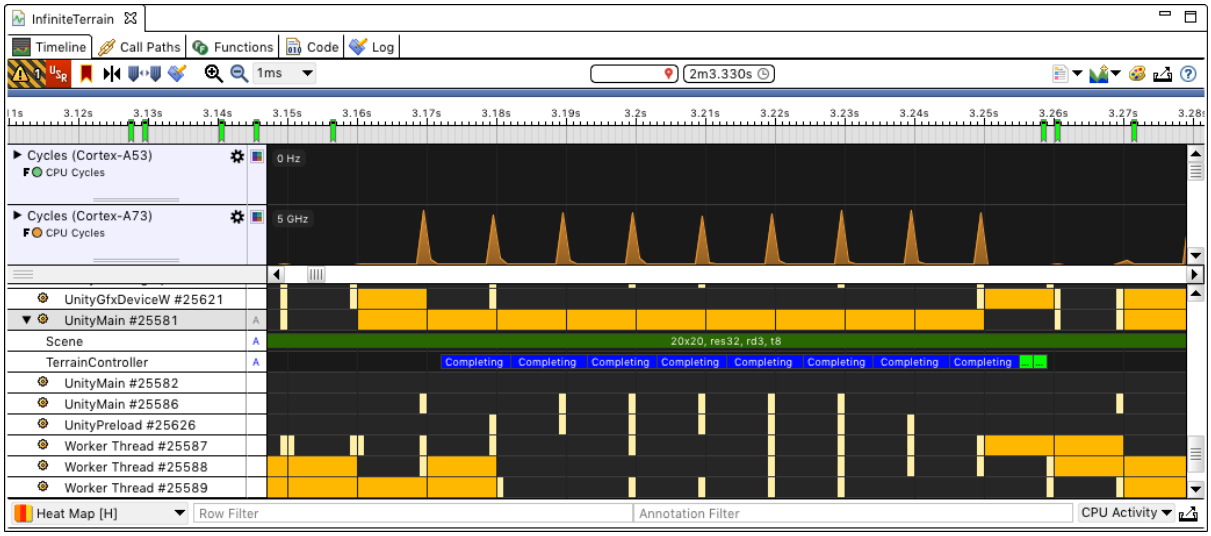
Streamlineのタイムラインビューは2つに分けられていて、上部にはメトリックグラフが表示され、下半分にはヒートマップなどさまざまなものが表示されます。ヒートマップには、システム全体で作業がどのように分散されているかが示され、特定のプロセスやスレッドに起因する作業のみが表示されるように上部のタイムラインをフィルタできます。ヒートマップを調べて、まず*UnityMain*スレッドを選択し、その後にすべてのワーカースレッドを選択することによって、Unityのメインスレッドとジョブスケジューラ内のスレッド間でCPUアクティビティーがどのように分割されているかを確認できます。



UnityMainスレッドのCPUプロファイル。*UnityMain*Cortex-A73 CPUの集中的なアクティビティーが大きいことを示している。

すべてのワーカースレッドのCPUプロファイル。Cortex-A73とCortex-A53 CPUの両方にまたがる集中的なアクティビティは小さいことを示している。

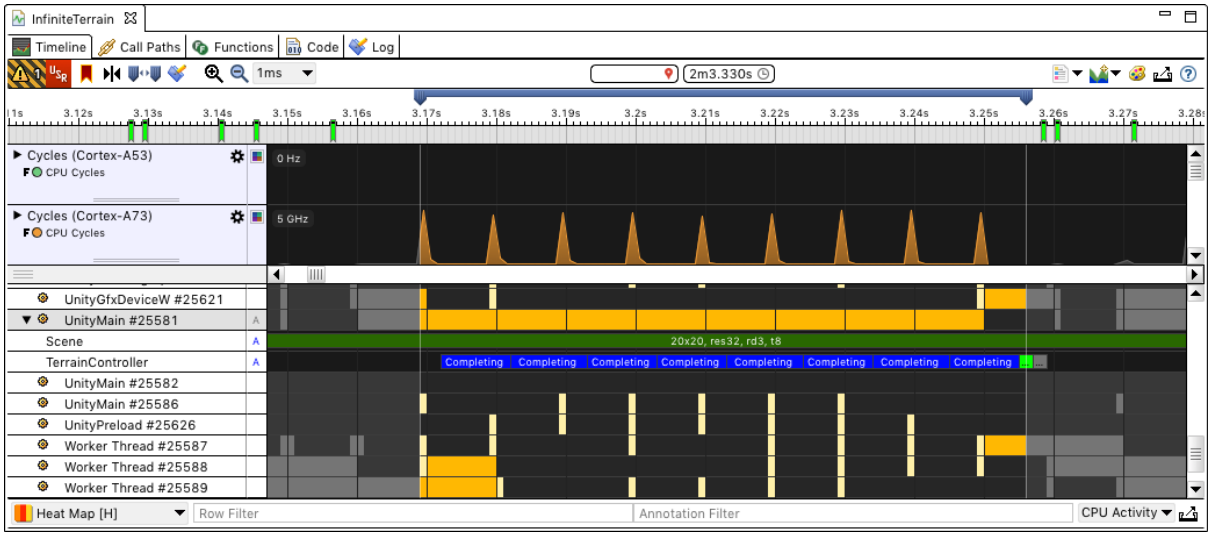
メインスレッドを見ていきましょう。左側のスクリーンショットをよく見ると、注目している*UnityMain*スレッドの横に「A」マーカーがあります。これは、Streamlineのアノテーションチャネルが存在することを意味しています。タイムラインを少し拡大し、*UnityMain*を展開して、何が起きているのかを確認します。



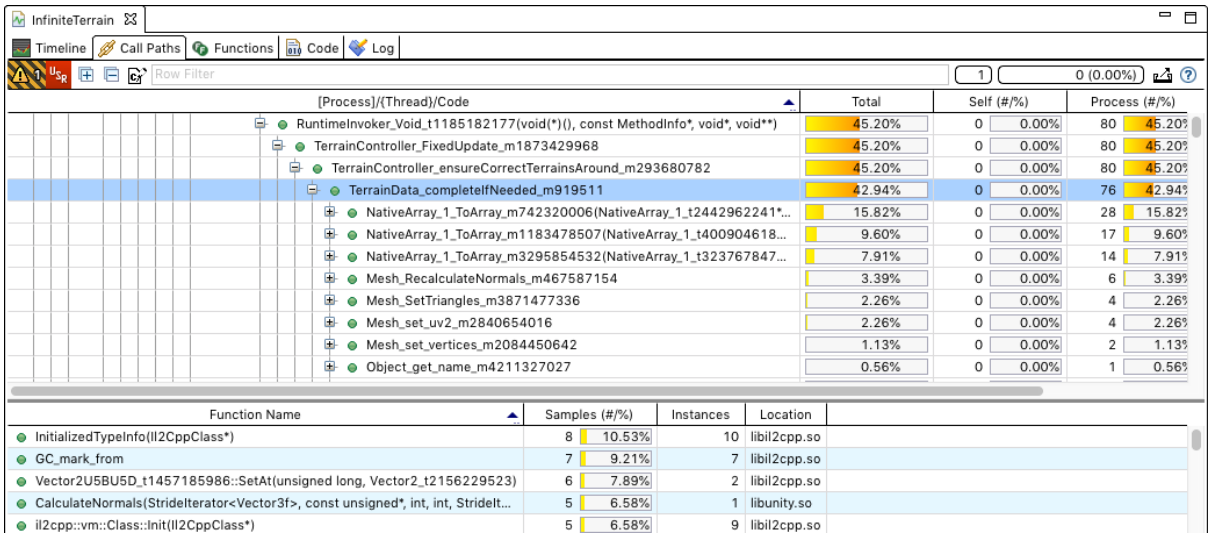
*Scene*行と*TerrainController*行は、ゲームに配置されているアノテーションによって生成された、Streamlineのチャネルです。*Scene*チャネルには現在どのシーンが実行されているかが示されていて、テレインタイルのサイズが20x20、テレインの解像度が32x32、レンダリング距離が3、8スレッドを並列に実行可能であることがわかります。

*TerrainController*チャネルは、コード内の注目すべき特定のピースがUnityのメインスレッドでいつ実行されているかを表すために使用されています。青色のブロックは、Terrainジョブが完了するときに実行されていたコードを示しています。緑色のブロックは、新しいTerrainジョブが生成のためにスケジュールされている位置を示しています。メインスレッドのすべてのアクティビティーは、基本的には、ジョブが完了するときに行われる必要がある作業と、最終的なメッシュが生成されてシーンに挿入されるのに必要な作業のためであることがわかります。

特定のスレッドに注目するだけでなく、特定の期間に限定して分析することもできます。Streamlineの*calipers* (キャリパ) を使用すると、特定の時間範囲を分析用にマークできます。この例では、Terrainジョブの完了に関連するアクティビティーの集中期間の開始と終了を選択しています (キャリパはタイムラインビューの上部で設定されています)。



ここでコールパスビューに切り替えると、キャリパで選択している時間範囲内のどこで時間がかかっているかを正しく把握できます。Unity用のIL2CPPスクリプティングバックエンドを使用しているため、デフォルトのMonoランタイムを使用している場合よりずっと多くの情報が取得されています。ここでは何が起きているのかについて徹底的に詳しく調べることはしませんが、もっと深く掘り下げるに値するさまざまなことが起こっているのは明らかです。



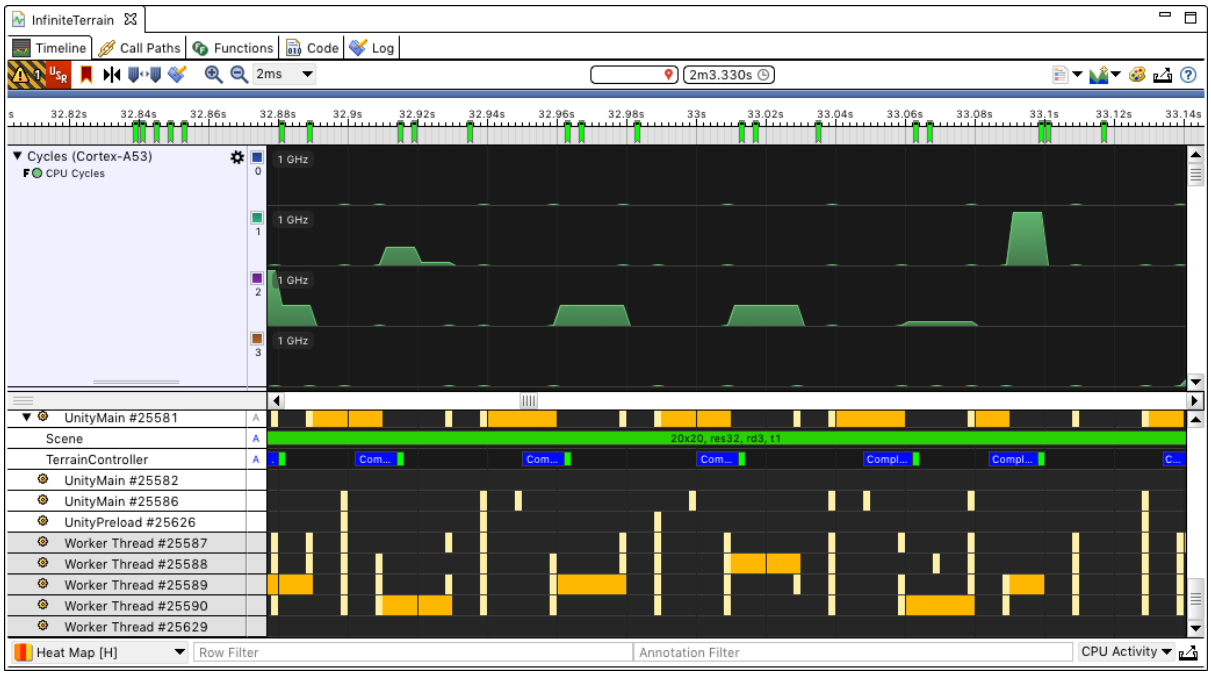
ワーカースレッドについて確認する

ワーカースレッドのみが表示されるようにフィルタした場合、最大の8つのジョブが並列に実行されるように要求したとすれば、意外なことは何もありません。次のスクリーンショットでは、Cortex-A53のクラスターを展開しているので、個々のコアの使用状況がわかります。

新しいTerrainジョブがスケジュールされていることを表すTerrainControllerチャネルのいくつかの緑色のブロックがあり、すべてのコアにわたってある程度集中的なアクティビティーがあり、Terrai

nジョブが生成されたときにそのジョブをメインスレッドで処理するためのTerrainControllerでの青色のアクティビティーがあります (UnityMainを選択していないため、このグラフにはメインスレッドのアクティビティーは表示されていません)。

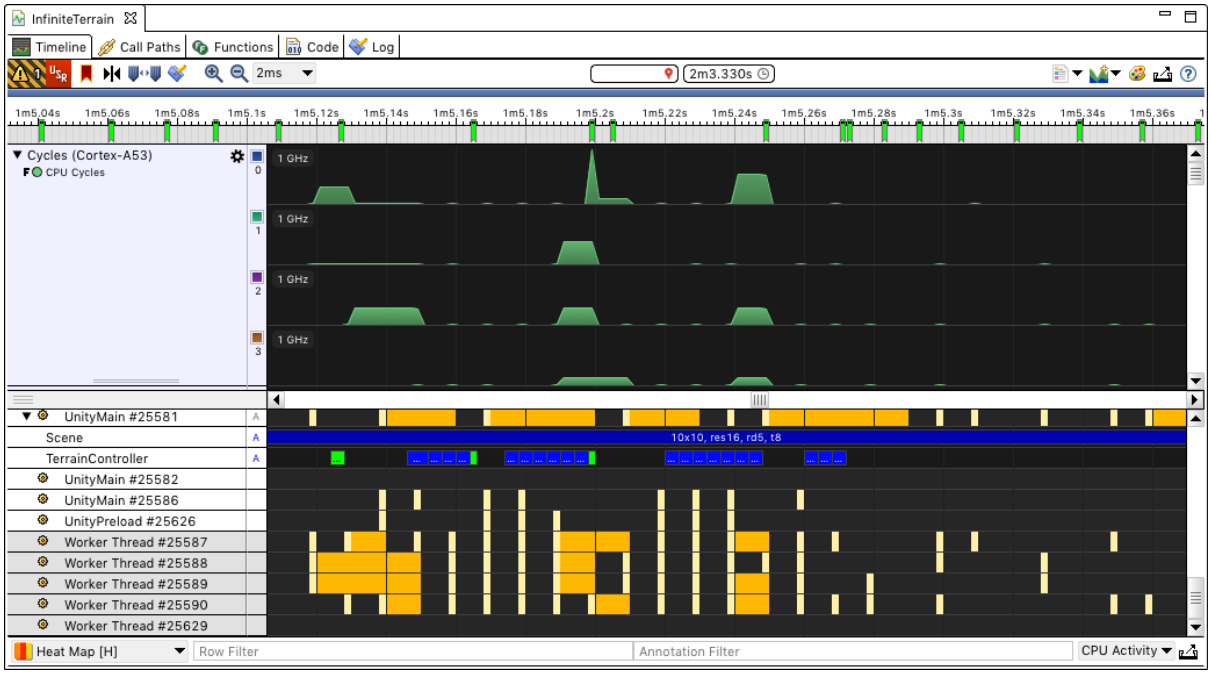
これを2番目のシーンのアクティビティーと比較するのは興味深いものがあります。2番目のシーンでは、テレインタイルの複雑さは同じですが、一度に1つのテレインタイルのみがスケジュールされるようにしています。



ここでは以下のことに注意してください。

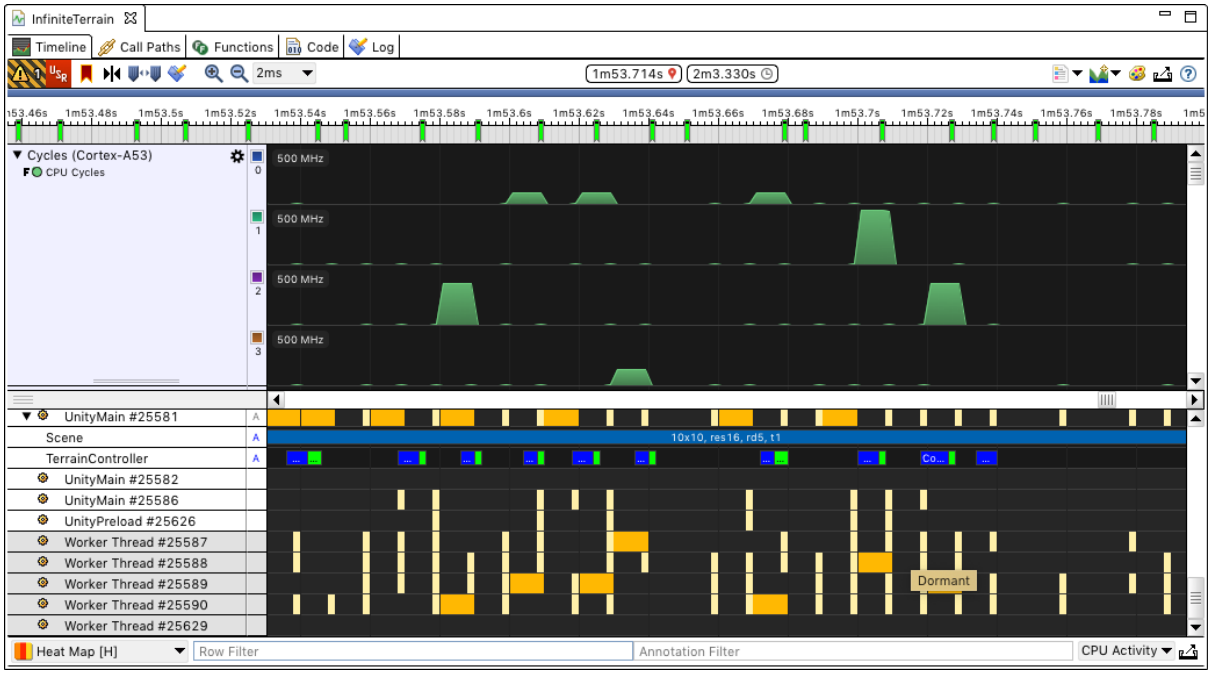
1. CPUアクティビティーの集中はかなり減っていて、大部分のコアはほとんどの時間でアイドル状態またはそれに近い状態です。
2. フレームレートは完璧ではありませんが、とても滑らかです。一度に完了するフレームは1つだけなので、メインスレッドでのアクティビティーの大きな集中は減っています。

このプロファイルを、より小さいタイルを使用している3番目のシーンと比較することもできます。

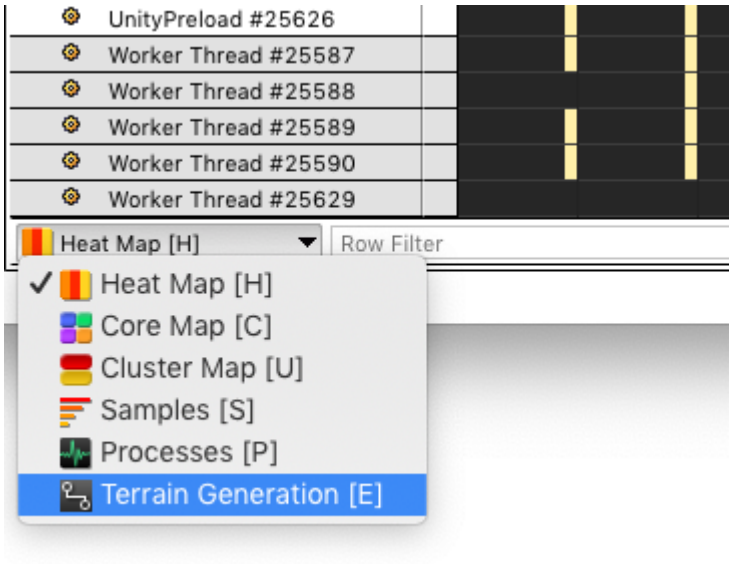


ご覧のように、CPUアクティビティーの集中は大幅に少なくなっていて、メインスレッドでの青色の作業完了のブロックは大幅に短縮されています。そのため、1番目のシーンに比べてフレームレートがより滑らかになっています (ただし、当然ながら全体的なジョブ数は増えているので、テレイン生成が、カメラがテレインを飛び越える速度に遅れないようにしておく必要があります)。

小さいタイルサイズで一度に1つのみのテレイン生成ジョブが実行されている4番目のシーンでは、全体的なフレームレートは最も滑らかですが、テレイン生成がカメラに遅れないような十分な速度で行われるように細心の注意を払う必要があります。元のビデオでわかるように、カメラが4番目のシーン上で高速に移動するときに常にそうなっているとは限りません。

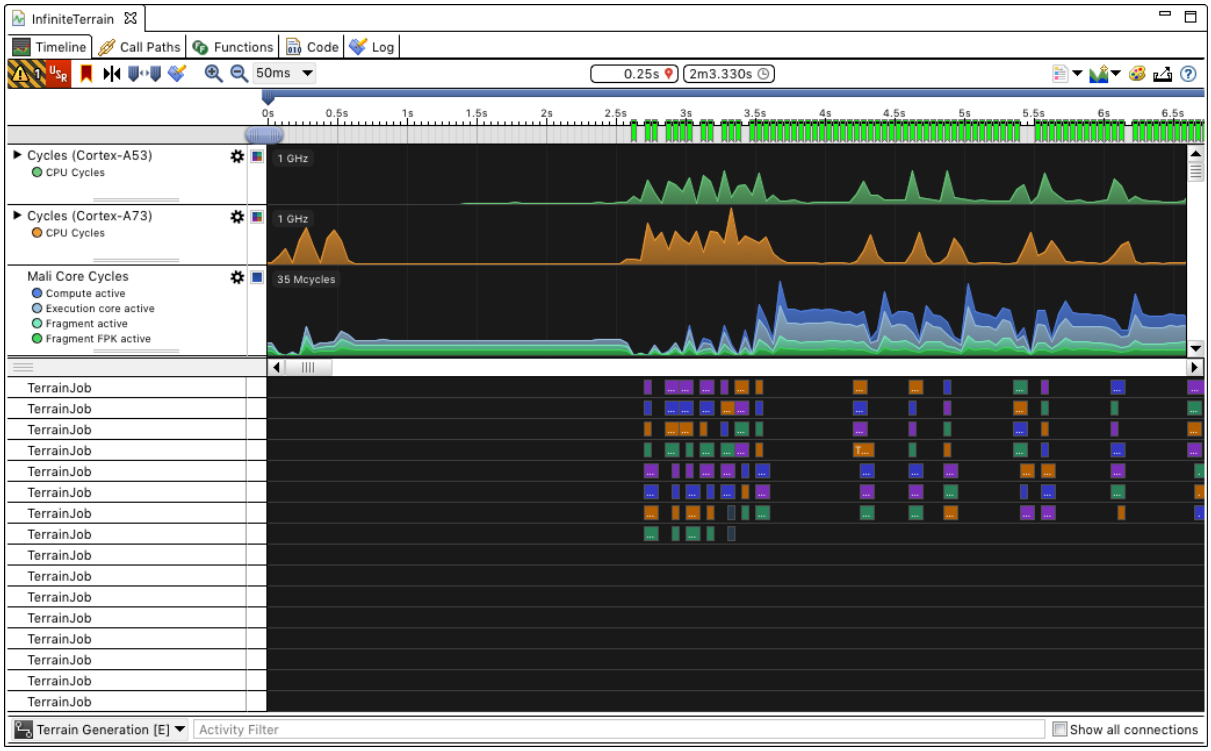


最後に、カスタムアクティビティーマップを使用して、ワーカースレッドがどのようにテレイン生成を行っているかについての有益な情報をさらに多く取得します。各カスタムアクティビティーマップは、これまでヒートマップを表示するために使用していた、左下にあるメニューで選択肢として表示されます。

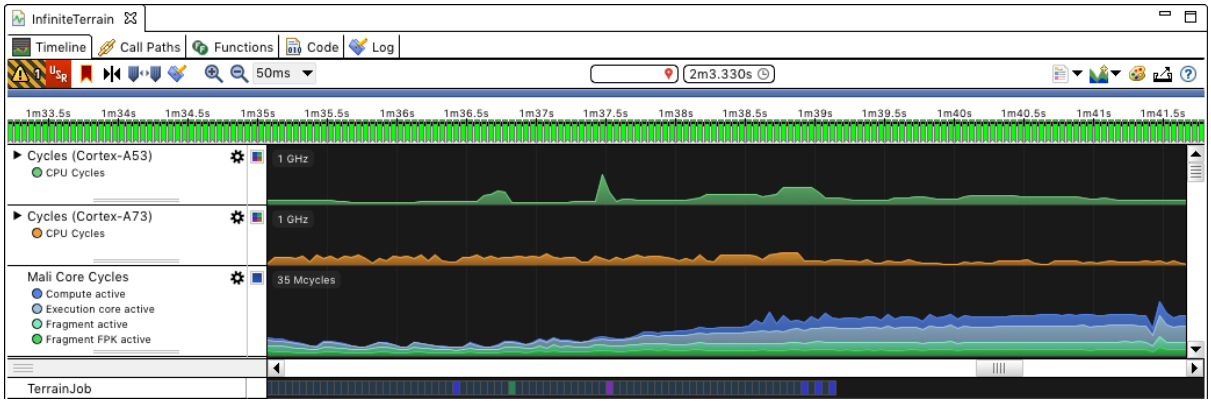


[*Terrain Generation*] ビューを選択すると、テレイン生成の各アクティビティーに対して色付きのボックスが表示されて、その開始時間と終了時間が表示されます。マウスポインタを置くと、そのテレインタイルのワールド座標、開始時間、および完了までの使用時間が表示されます。このスクリーンショットでは、Mali GPUで行われた演算動作もグラフ化しています。予想どおりですが、テレインが満たされるにつれてGPUアクティビティーは着実に増加しています。このスクリーンショットは、1番目のシーンの始めに注目しているときに、最大で8つの大きいタイルを同時に生成している

ところを撮ったものです。メインスレッドですべての新しいジオメトリが準備されている間の休止によって、GPUは長時間アイドル状態になっています。



より小さいサイズのタイルを順次生成している4番目のシーンに移動すると、GPUアクティビティーの傾斜がずっと滑らかになっていることがわかり、Terrainジョブが一度に1つだけ実行されていたことをはっきり確認できます (また、タイルサイズが小さいため、各ジョブの所要時間が短くなっています)。



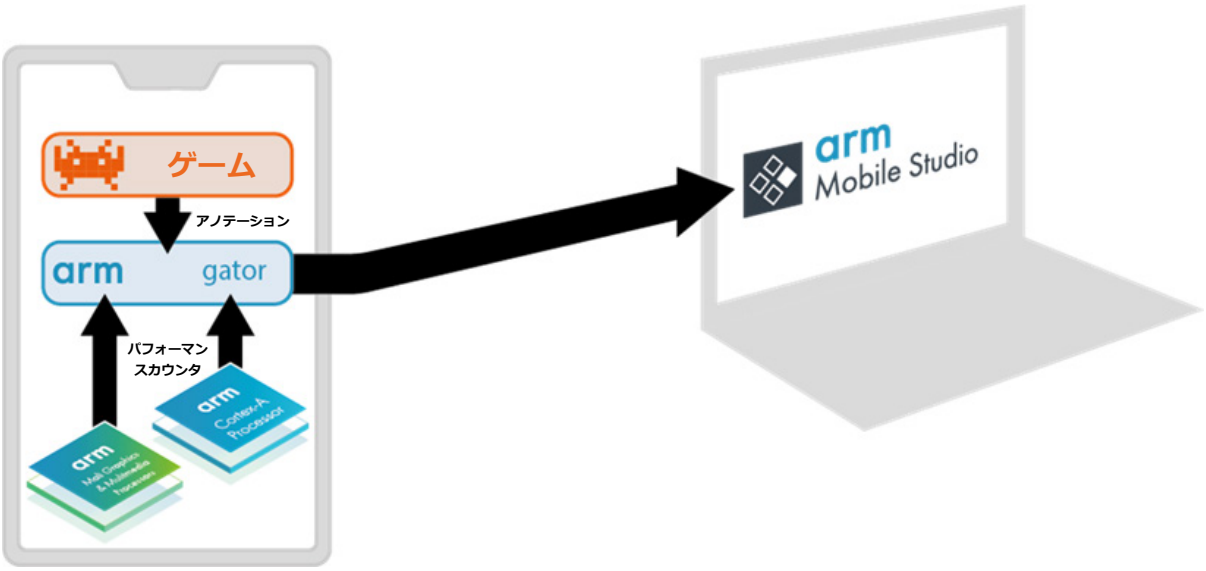
ここまで、より概要レベルのコンテキストが提供されるようにゲームそのものでアノテーションを使用した場合に、Streamlineで得られるいくつかの付加的な有益情報について段階的に説明してきました。以下のアノテーションを使用しました。

- 新しいフレームが開始されたことを示すためのマーカー。
- どのシーンが実行中であり、メインスレッドでどのアクティビティーが実行されていたかを示すためのチャネル。
- 非同期でスケジュールされたテレイン生成ジョブの挙動を示すためのカスタムアクティビティーマップ。

どれもすばらしいものですが、その仕組みはどうなっているのでしょうか?

Streamlineのアノテーションについて

Streamlineの仕組みについてさらに詳しく見ていきましょう。Androidアプリケーションを分析するときに、そのデバイス上で*gator*という別のプロセス (同じユーザーがアプリケーションとして実行している) が実行されていて、さまざまなハードウェアソース (Mali GPUやArm Cortex-A CPUなど) からプロファイリング情報を収集し、メトリックを集計したストリームをユーザーのコンピューターに送信します。Streamlineのアノテーションは、アプリケーション自体が独自のマーカーやメトリックをそのストリームに挿入できるようにするためのメカニズムです。

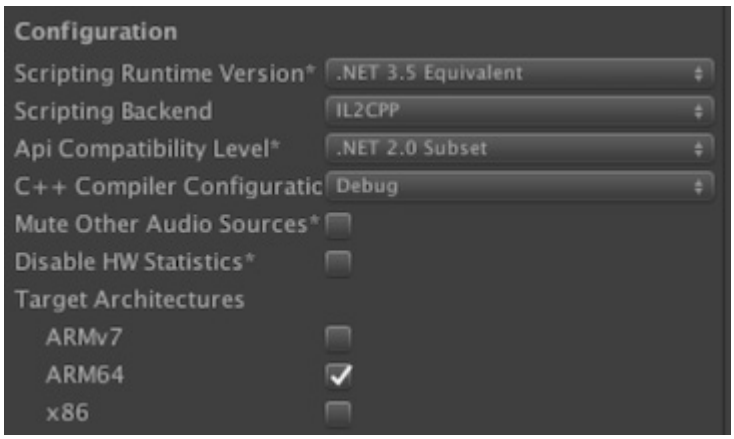


UnityでStreamlineのアノテーションを使用する方法

Streamlineのアノテーションでは特有のプロトコルが使用されていて、オープンソースのC言語での実装がArm Mobile Studioの一部として提供されています。StreamlineのアノテーションをUnityコンテンツ内から簡単に生成できるように、C言語の実装に対するC#のラッパーをいくつかユーザーが準備する必要があります。このウォークスルーで使用するラッパーおよびC言語での実装はUnityアセットパッケージとして入手できます。そのパッケージをダウンロードし、自分のプロジェクトにカスタムアセットパッケージとしてインポートします、そのパッケージではArm名前空間に新しいメソッドが追加されていて、そのメソッドを使用するとユーザーのプロジェクトでStreamlineのアノテーションを簡単に使用できます。APIのドキュメントは、そのパッケージに含まれているREADME.mdファイルに記載されています。

最適なエクスペリエンスのためのUnityプロジェクトのセットアップ

最も高速かつ最も簡単にAndroidビルドの分析をUnityから取得する場合は、Android Playerの特定の設定を以下のようにしておく必要があります。



スクリプティングバックエンドとして*IL2CPP*を使用していることと、[C++ Compiler Configuration] が [*Debug*] に設定されていることを確認します。そうすることによって、スクリプトがネイティブコードにコンパイルされてパフォーマンスが向上するだけでなく、Streamlineでデバッグ情報を表示して、コールパスビューでパフォーマンスデータをユーザーの関数に戻してマッピングできることも意味します。

ターゲットアーキテクチャを「ARM64」に設定します (デフォルトでは「ARMv7」)。最近のほとんどのモバイルデバイスは64ビットであり、結果的にコード生成の品質が向上します。

マーカーの追加

マーカーは最も簡単にできるアノテーションです。提供されているメソッドは文字列およびオプションとして色を受け取ります。たとえば、フレームごとのマーカーを緑色で出力するには、Game Objectsのいずれかで次のコードが使用されます (Unityアーキテクチャに慣れていないユーザーのために説明しておくと、Update() メソッドはフレームごとに1回自動的に呼び出されます)。

フル画面

チャネルの追加

チャネルを使用するのは大して難しくありません。まず、名前を指定してチャネルを作成します。そうすると、Channelオブジェクトのメソッドを使用してそのチャネルにアノテーションを送り込むことができます。次にサンプルを示します。

フル画面

チャネル内のアノテーションは一定時間に及ぶことを忘れないでください。次のアノテーションを開始する前にアノテーションを終了する場合は、end()メソッドを使用できます。たとえば、メインスレッドでTerrainジョブの完了を行うTerrainControllerの一部は、次のようにラップされています。

フル画面

カスタムアクティビティーマップの使用

CAM (カスタムアクティビティーマップ) は、チャネルの上位にある単なるもうひとつのレイヤーと考えることができます。まずCAMに名前を付けてから、その中にトラックを作成します。そうすると、チャネルにアノテーションを追加したのとほとんど同じようにして、そのトラックにアノテーションを追加できます。

このサンプルでは、テレイン生成CAMは次のように作成されています。

フル画面

ただし、この適用例には厄介な問題が1つあります。Unity Job Systemでジョブが実行中である場合、そのジョブはゲームの他のオブジェクトモデルとほとんどやり取りできません (これはスレッドセーフに保つのに役立ちます)。そのジョブ内でできることは、開始時刻と終了時刻を覚えておくことだけであり、ジョブのアクティビティーをCAMに登録できるのは、メインスレッドでそのジョブが除去されるときです。

C#のラッパーでは、Streamlineのアノテーションで必要とされる形式で現在時間を返す関数が提供されていて、その関数をジョブ内から安全に呼び出すことができます。

フル画面

メインスレッドに戻ったときに、使用するトラック (オーバーラップしないようにトラックをプールで管理します。その方が視覚的に便利です) を選択し、ジョブをそのトラックに登録します。ここで、job.timingsは、ジョブの開始時刻と終了時刻が入っている2要素の配列です。

フル画面

これで終了です。このUnityパッケージを徐々に改良して、もっと多くの機能を追加していこうと考えています。みなさまからのフィードバックをお待ちしています。

Streamlineでの基本的なプロファイルの収集

Streamlineで最初にプロファイルを収集する際には、いくつかの手順を行う必要がありますが、一度セットアップすれば、あとは非常に単純明快です。

まず、Windows、Mac、またはLinuxの無償版のArm Mobile StudioのStarter Editionをダウンロードしてインストールします。

前述のStreamlineのアーキテクチャで説明したように、下記の手順を、ユーザーが実施しておく必要があります

- モバイルデバイス上でgatorを実行し、gatorがアプリケーションにアクセスできるようにしておく必要があります。
- デバイスからデータを取得してgatorに渡す手段が必要です。

Streamlineでは、それを行うための手段が提供されていますが、広範なデバイスにわたる堅実で最もシンプルな方法は、最初に下記の主要な情報を確実に把握しておくことです。

- アプリケーションが32ビットと64ビットのどちらであるか。 前述の手順に従って、ARM64オプションを指定してUnityゲームをビルドしていれば、64ビットです。
- アプリケーションのパッケージ名 (UnityのAndroid Player設定で指定した名前)。ここでのサンプルアプリケーションの場合、この名前は `com.Arm.InfiniteTerrain`です。

- gatorのバイナリを入手する方法。Arm Mobile Studioのインストール先のstreamline/bin/arm/フォルダ (32ビット) または streamline/bin/arm64/フォルダ (64ビット) にあります。

これらの情報を把握したら、以下の手順に従って分析を実行します。

- アプリケーションがデバイスにインストールされていることを確認します。
- Androidのadbツールを使用して、モバイルデバイスから、Arm Mobile Studioを実行しているシステム上のローカルポートにネットワークポートを転送します。
- adbを使用してgator (ユーザーのアプリケーションに応じて32ビット版または64ビット版のいずれか) をデバイスにプッシュし、アプリケーションと同じパッケージ名を使用してgatorの実行を開始します。
- Streamlineを起動し、トラフィックの転送先としてadbで使用したローカルポートに接続します。この時点で、どのパフォーマンスカウンタを収集対象にするかを選択できます。
- モバイルデバイス上でアプリケーションを起動します。Streamlineは分析データを受信すると、その収集を開始します。

gatorが実行中であれば、新しいバージョンのアプリケーションをインストールし、Streamlineを起動および停止して、gatorを再起動することなくさらに分析を実行できます。

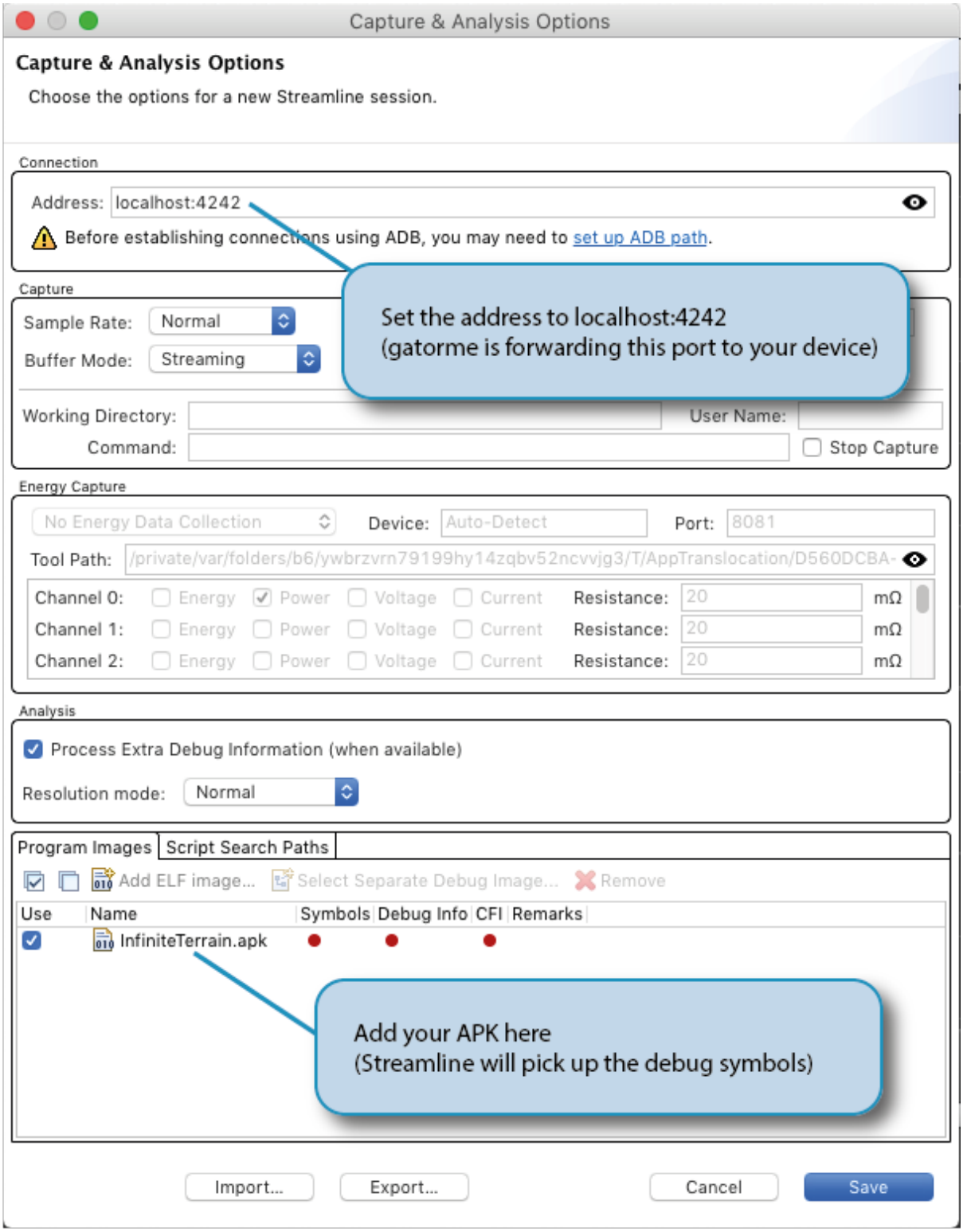
このプロセスを簡単にするために、gatorとadbを設定して実行するために使用できる gatormeスクリプトをダウンロードできます。このスクリプトで指定する必要があるのは、実行するgatorバイナリのパス、アプリケーションのパッケージ名、およびどのMali GPUがデバイスに搭載されているか (gatorがデバイスを探って、どのGPUが搭載されているかを割り出すことができない場合に役立ちます) だけです。このスクリプトでは、このスクリプトが幅広いモバイルデバイス上で確実にうまく動作するようにし、ユーザーがプロファイリングアクティビティーを終了したときにgatorが適切にシャットダウンされるようにするための、いくつかの手順も実行されます (いずれ近いうちにgatormeの機能をStreamlineに直接取り入れる予定です)。

詳細はgatormeのドキュメントに説明されていますが、ここでは作業サンプルとして、デバイスにこのAPKをインストールすると、InfiniteTerrainのコンテンツをどのようにプロファイリングできるかを示します。

まず、コマンドラインでgatormeを次のように実行します。

フル画面

これで、Streamlineを起動してキャプチャする準備が整いました。設定が下図のように正しく行われていることを確認します。



セットアップできたら、開発、分析、修正の手順を繰り返すのは簡単です。アプリケーションをシャットダウンしてgatormeスクリプトは実行したままにしておいて、Unityで [Build And Run] を指示してStreamlineの情報をさらにキャプチャすることができます。

このブログが興味深いもので役立つものと思っていただければ幸いです。InfiniteTerrainのサンプルのすべてのソースコードはGitHub (Apache 2.0ライセンス) で入手していただけます。ソースコー

ドとグラフィックスアセットだけでなくArmMobileStudio.unitypackageもあります。このUnityのカスタムアセットパッケージを使用すると、ユーザーは自分のプロジェクトにインポートしてプロジェクトにStreamlineのアノテーションを追加できます。ビルド済みのInfiniteTerrain.apkもあります。このAPKは64ビット版のAndroid開発用ビルドであり、一部のコンテンツの分析をすばやく始めるだけであればすぐにデバイスにデプロイできます。

最後になりましたが、Arm Mobile Studioやグラフィックスやゲーム全般でのArmに関して不明な点がありましたら、グラフィックスとマルチメディアのフォーラムに参加いただくか、または下記のArm Mobile Studioの開発者向けサイトでツールに関する詳細情報を参照してください。

Arm Mobile Studioの関連情報