

Unity®

PERFORMANCE TUNING BIBLE

Written By

**CyberAgent Smartphone Games &
Entertainment Division
SGE Core Technology Team**

 CyberAgent.®

Unityパフォーマンスチューニングバイブル

株式会社サイバーエージェント | ゲーム・エンターテイメント事業部SGEコア技術本部 著



Unity®

PERFORMANCE TUNING BIBLE

Written By

**CyberAgent Smartphone Games &
Entertainment Division
SGE Core Technology Team**

CyberAgent®

Unityパフォーマンスチューニングバイブル

株式会社サイバーエージェント | ゲーム・エンターテイメント事業部 SGEコア技術本部 著

はじめに

本書は Unity アプリケーションのパフォーマンスチューニングで困った時にリファレンスとして活用されることを目指して作成しました。

パフォーマンスチューニングは過去のノウハウが活用できるため、属人化が進みやすい分野であると感じています。未経験の方からするとなんとなく難しそうという印象を持つてしまうかもしれません。それは性能低下の原因が多岐に渡ることが理由の1つでしょう。

しかしパフォーマンスチューニングのワークフローは型にはめることができます。そのフローに従うことでの原因の特定は容易になり、その事象にあった解決策を探すだけになります。解決策を模索するときには知識や経験が手助けになるでしょう。そこで本書は「ワークフロー」や「経験からくる知識」を主に学習できるように設計しました。

社内向けドキュメントとして制作予定でしたが、せっかくならたくさんの方に見て頂き、よりブラッシュアップできればと考えています。手に取って頂いた方にとって、少しでも開発の手助けになれば幸いです。

本書について

本書はスマートフォン向けアプリを前提とした内容になります。一部の解説は他プラットフォームに当てはまらないものもあるかもしれませんのがご留意ください。また本書内で利用する Unity のバージョンは、とくに断りがない場合 Unity 2020.3.24f1 とします。

本書の構成

本書は大きく分けて3部構成となっています。第2章まではチューニングを行うまでの基礎知識、第3章では各種計測ツールの使い方、そして第4章以降はさまざまなチューニングプラクティスを取り上げます。それぞれの章は独立しているため、自身のレベルに合わせて必要な箇所のみ読み進めていくのもよいでしょう。次に各章の概要について説明します。

第1章「パフォーマンスチューニングを始めよう」では、パフォーマンスチューニ

ングのワークフローに関して説明します。まずは取りかかるまでの事前準備について解説し、次に原因を切り分けて調査を進めていく方法について解説します。この章を読むことでパフォーマンスチューニングに着手できる状態を目指します。

第2章「基礎知識」では、ハードウェア、描画の流れ、Unityの仕組みなど、パフォーマンスチューニングを行う上で知っておくとよい基礎知識について解説します。第2章以降を読み進めていく中で、知識が不足していると感じたときに読み返すのもよいでしょう。

第3章「プロファイリングツール」では、原因調査に使用するさまざまなツールの使い方を学ぶことができます。計測ツールをはじめて利用する際にリファレンスとして活用することを推奨します。

第4章以降の「Tuning Practice」ではアセットからスクリプトまで、さまざまなプラクティスを詰め込んだ内容になります。ここに記載されている内容は現場ですぐに使えるものが多いので、ぜひ一読してみてください。

GitHub

本書のリポジトリ^{*1}を公開します。随時、加筆修正を行う予定です。またPRやIssueを用いて修正点の指摘や追記の提案が可能です。よろしければご利用ください。

免責事項

本書に記載された内容は情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について当社はいかなる責任も負いません。

^{*1} <https://github.com/CyberAgentGameEntertainment/UnityPerformanceTuningBible/>

目次

はじめに	ii
本書について	ii
本書の構成	ii
GitHub	iii
免責事項	iii
第1章 パフォーマンスチューニングを始めよう	1
1.1 事前準備	2
1.1.1 指標を決める	3
1.1.2 メモリの最大使用量を把握する	5
1.1.3 動作保証端末を決める	7
1.1.4 品質設定の仕様を決める	8
1.2 未然に防ぐ	8
1.3 パフォーマンスチューニングに取り組む	9
1.3.1 心構え	9
1.3.2 性能低下の種類	10
1.4 メモリ超過の原因切り分け	11
1.4.1 メモリがリークしている	11
1.4.2 メモリ使用量が単純に多い	12
1.5 メモリリークを調査しよう	12
1.6 メモリを削減しよう	13
1.6.1 Assets	13
1.6.2 GC (Mono)	15
1.6.3 Other	15
1.6.4 プラグイン	16
1.6.5 仕様を検討する	17

1.7	処理落ちの原因切り分け	17
1.8	瞬間的な負荷を調査する	18
1.8.1	GCによるスパイク	18
1.8.2	重い処理によるスパイク	19
1.9	定常的な負荷を調査する	19
1.9.1	CPUバウンド	20
1.9.2	GPUバウンド	20
1.10	まとめ	22
第2章	基礎知識	23
2.1	ハードウェア	24
2.1.1	SoC	24
2.1.2	iPhone・AndroidとSoC	25
2.1.3	CPU	26
2.1.4	GPU	30
2.1.5	メモリ	33
2.1.6	ストレージ	37
2.2	レンダリング	40
2.2.1	レンダリングパイプライン	41
2.2.2	半透明描画とオーバードロー	43
2.2.3	ドローコール・セットパスコールとバッティング	44
2.3	データの表現方法	45
2.3.1	ビットとバイト	46
2.3.2	画像	47
2.3.3	画像の圧縮	48
2.3.4	メッシュ	49
2.3.5	キーフレームアニメーション	51
2.4	Unityの仕組み	53
2.4.1	バイナリとランタイム	53
2.4.2	アセットの実体	57
2.4.3	スレッド	58
2.4.4	ゲームループ	60
2.4.5	GameObject	62
2.4.6	AssetBundle	64
2.5	C#の基礎知識	67

目次

2.5.1	スタックとヒープ	68
2.5.2	ガベージコレクション	68
2.5.3	構造体 (struct)	70
2.6	アルゴリズムと計算量	73
2.6.1	計算量について	74
2.6.2	基本的なコレクションとデータ構造	77
2.6.3	計算量を下げる工夫	80
第3章	プロファイリングツール	81
3.0.1	計測時の注意点	82
3.1	Unity Profiler	82
3.1.1	計測方法	84
3.1.2	CPU Usage	88
3.1.3	Memory	93
3.2	Profile Analyzer	99
3.2.1	導入方法	99
3.2.2	操作方法	100
3.2.3	解析結果 (Single モード)	101
3.2.4	解析結果 (Compare モード)	106
3.3	Frame Debugger	107
3.3.1	分析画面	108
3.3.2	詳細画面	108
3.4	Memory Profiler	111
3.4.1	導入方法	112
3.4.2	操作方法	113
3.5	Heap Explorer	122
3.5.1	導入方法	122
3.5.2	操作方法	123
3.6	Xcode	127
3.6.1	プロファイル方法	127
3.6.2	Debug Navigator	128
3.6.3	GPU Frame Capture	132
3.6.4	Memory Graph	141
3.7	Instruments	143
3.7.1	Time Profiler	144

3.7.2	Allocations	146
3.8	Android Studio	149
3.8.1	プロファイル方法	150
3.8.2	CPU 計測	151
3.8.3	Memory 計測	153
3.9	RenderDoc	154
3.9.1	計測方法	155
3.9.2	キャプチャデータの見方	158
第 4 章	Tuning Practice - Asset	166
4.1	Texture	167
4.1.1	インポート設定	167
4.1.2	Read/Write	168
4.1.3	Generate Mip Maps	169
4.1.4	Aniso Level	169
4.1.5	圧縮設定	170
4.2	Mesh	171
4.2.1	Read/Write Enabled	171
4.2.2	Vertex Compression	172
4.2.3	Mesh Compression	173
4.2.4	Optimize Mesh Data	174
4.3	Material	175
4.4	Animation	176
4.4.1	スキンウェイト数の調整	176
4.4.2	キーの削減	177
4.4.3	更新頻度の削減	179
4.5	Particle System	180
4.5.1	パーティクルの個数を抑える	180
4.5.2	ノイズは重いので注意	183
4.6	Audio	184
4.6.1	Load Type	184
4.6.2	Compression Format	186
4.6.3	サンプルレートの指定	187
4.6.4	効果音は Force To Mono を設定	187
4.7	Resources / StreamingAssets	188

目次

4.7.1	起動時間を遅くする Resources フォルダー	189
4.8	ScriptableObject	189
第 5 章	Tuning Practice - AssetBundle	191
5.1	AssetBundle の粒度	192
5.2	AssetBundle のロード API	192
5.3	AssetBundle のアンロード戦略	193
5.4	同時にロードされている AssetBundle 数の最適化	193
第 6 章	Tuning Practice - Physics	195
6.1	物理演算のオン・オフ	196
6.2	Fixed Timestep と Fixed Update の頻度の最適化	196
6.2.1	Maximum Allowed Timestep	197
6.3	コリジョン形状の選定	198
6.4	衝突マトリックスとレイヤーの最適化	198
6.5	レイキャストの最適化	199
6.5.1	レイキャストの種類	199
6.5.2	レイキャストのパラメーターの最適化	200
6.5.3	RaycastAll と RaycastNonAlloc	200
6.6	コライダーと Rigidbody	201
6.6.1	Rigidbody とスリープ状態	202
6.7	衝突検出の最適化	205
6.8	その他プロジェクト設定の最適化	206
6.8.1	Physics.autoSyncTransforms	206
6.8.2	Physics.reuseCollisionCallbacks	206
第 7 章	Tuning Practice - Graphics	207
7.1	解像度の調整	208
7.1.1	DPI の設定	208
7.1.2	スクリプトによる解像度設定	209
7.2	半透明とオーバードロー	210
7.3	ドローコールの削減	211
7.3.1	動的バッティング	211
7.3.2	静的バッティング	213
7.3.3	GPU インスタンシング	214
7.3.4	SRP Batcher	217

7.4	SpriteAtlas	219
7.5	カーリング	223
7.5.1	視錐台カーリング	223
7.5.2	背面カーリング	223
7.5.3	オクルージョンカーリング	224
7.6	シェーダー	226
7.6.1	浮動小数点数型の精度を下げる	227
7.6.2	頂点シェーダーで計算を行う	227
7.6.3	テクスチャに情報を事前に仕込む	228
7.6.4	ShaderVariantCollection	228
7.7	ライティング	230
7.7.1	リアルタイムシャドウ	230
7.7.2	ライトマッピング	234
7.8	Level of Detail	238
7.9	テクスチャストリーミング	239
第8章	Tuning Practice - UI	241
8.1	Canvas の分割	242
8.2	UnityWhite	243
8.3	Layout コンポーネント	244
8.4	Raycast Target	245
8.5	マスク	245
8.6	TextMeshPro	246
8.7	UI の表示切り替え	247
第9章	Tuning Practice - Script (Unity)	249
9.1	空の Unity イベント関数	250
9.2	tag や name のアクセス	251
9.3	コンポーネントの取得	252
9.4	transform へのアクセス	252
9.5	明示的な破棄が必要なクラス	253
9.6	文字列指定	253
9.7	JsonUtility の落とし穴	254
9.8	Render や MeshFilter の落とし穴	255
9.9	ログ出力コードの除去	256

目次

9.10	Burst を用いたコードの高速化	257
9.10.1	Burst を用いたコードの高速化	258
第 10 章	Tuning Practice - Script (C#)	261
10.1	GC.Alloc するケースと対処法	262
10.1.1	参照型の new	262
10.1.2	ラムダ式	263
10.1.3	ジェネリックを使用してボックス化するケース	265
10.2	for/foreach について	267
10.3	オブジェクトプーリング	270
10.4	string	271
10.5	LINQ と遅延評価	273
10.5.1	LINQ の使用による GC.Alloc を軽減する	273
10.5.2	LINQ の遅延評価	274
10.5.3	「LINQ の使用を避ける」という選択	275
10.6	async/await のオーバーヘッドの避け方	277
10.6.1	不要な箇所での async を避ける	277
10.6.2	同期コンテキストのキャプチャを避ける	278
10.7	stackalloc による最適化	279
10.8	sealed による IL2CPP バックエンド下でのメソッド呼び出しの最適化	280
10.9	インライン化による最適化	283
第 11 章	Tuning Practice - Player Settings	285
11.1	Scripting Backend	286
11.1.1	Debug	287
11.1.2	Release	287
11.1.3	Master	287
11.2	Strip Engine Code / Managed Stripping Level	287
11.3	Accelerometer Frequency (iOS)	288
第 12 章	Tuning Practice - Third Party	289
12.1	DOTween	290
12.1.1	SetAutoKill	290
12.1.2	SetLink	291
12.1.3	DOTween Inspector	292
12.2	UniRx	293

12.2.1 購読の解除	293
12.3 UniTask	294
12.3.1 UniTask v2	294
12.3.2 UniTask Tracker	294
さいごに	297
著者紹介	298



PERFORMANCE TUNING BIBLE

CHAPTER

01

第1章

パフォーマンス
チューニングを始めよう

CyberAgent Smartphone Games & Entertainment

第1章

パフォーマンスチューニングを始めよう

本章ではパフォーマンスチューニングを行うにあたり必要な事前準備や、取り組む際のフローに関して解説します。

まずはパフォーマンスチューニングを始める前に決めておくべきことや、考慮すべきことについて取り上げます。プロジェクトがまだ初期フェーズの場合はぜひ目を通してみてください。ある程度プロジェクトが進んでいたとしても、記載されている内容が考慮されているか改めてチェックするのもよいでしょう。次に性能低下が発生しているアプリケーションに対して、どのように取り組んでいくべきかを解説します。原因の切り分け方とその解決手法を学ぶことで、パフォーマンスチューニングの一連のフローを実践できるようになるでしょう。

1.1 事前準備

パフォーマンスチューニングの前に達成したい指標を決めましょう。言葉にすると簡単ですが実は難易度が高い作業です。それは世の中にはさまざまなスペックの端末が溢れかえっており、低スペック端末を使っているユーザーを無視することはできないからです。そのような状況の中でゲーム仕様やターゲットユーザー層、海外展開の有無など、さまざまなことを考慮する必要があります。この作業はエンジニアだけでは完結しません。他職種の方と協議しながらクオリティラインを決める必要があり、技術検証も必要になるでしょう。

これらの指標は負荷を測るほどの機能実装やアセットが存在しない初期フェーズから決めることは難易度が高いです。そのためある程度プロジェクトが進んでから決めるのも1つの手です。ただしプロジェクトが量産フェーズに入る前までに必ず決めるように心掛けておきましょう。一度量産を開始すると変更コストが莫大なものになるためです。本節で紹介する指標を決めるには時間がかかりますが、焦らずしっかり進めていきましょう。

量産フェーズ後の仕様変更の恐ろしさ

いま量産フェーズ後でありながら、低スペック端末で描画にボトルネックのあるプロジェクトがあるとします。メモリはすでに限界近い使用量なので、距離によって低負荷モデルに切り替える手法も使えません。そのためモデルの頂点数を削減することにしました。

まずはデータをリダクションするために発注し直します。新たに発注書が必要になるでしょう。次にディレクターが品質をチェックし直す必要があります。そして最後にデバッグする必要もあります。簡易的に書きましたが、実際にはより細かいオペレーションやスケジュール調整があるでしょう。

上記のような対応を必要とするアセットが、量産後ともなると数十～数百個はあるでしょう。これは時間と労力が大変掛かるため、プロジェクトにとって致命傷となりかねません。

このような事態を防ぐためにもっとも負荷の掛かるシーンを作成し、指標を満たしているか事前に検証を行うことがとても大事なのです。

1.1.1 指標を決める

指標を決めると目指すべき目標が定まります。逆に指標がないといつまで経っても終わりません。表 1.1 に決めるとよい指標を取り上げます。

▼表 1.1 指標

項目	要素
フレームレート	常時どのくらいのフレームレートを目指すか
メモリ	どの画面でメモリが最大になるか試算し、限界値を決める
遷移時間	遷移時間待ちはどれくらいが適切か
熱	連続プレイ X 時間で、どのくらいの熱さまで許容できるか
バッテリー	連続プレイ X 時間で、どれくらいのバッテリー消費が許容できるか

表 1.1 の中でも、とくにフレームレートとメモリは重要な指標なので必ず決めましょう。この時点では低スペック端末のことは置いておきましょう。まずはボリュームゾーンにある端末に対して指標を決めることが大事です。

第1章 パフォーマンスチューニングを始めよう

ボリュームゾーンの定義はプロジェクト次第です。ベンチマークになる他タイトルや市場調査を行って決めるのもよいでしょう。もしくはモバイルデバイスのリプレイスが長期化した背景から、4年ぐらい前のミドルレンジをひとまず指標にしてもよいでしょう。根拠は少し曖昧でも目指すべき旗を立てましょう。そこから調整していけばよいのです。

ここで実例を考えてみましょう。いま次のような目標を掲げているプロジェクトがあるとします。

- ・競合アプリケーションのよくない点はすべて改善したい
- ・とくにインゲームは滑らかに動かしたい
- ・上記以外は競合と同等ぐらいでよい

この曖昧な目標をチームで言語化すると次のような指標が生まれました。

- ・フレームレート
 - インゲームは 60 フレーム、アウトゲームはバッテリー消費の観点から 30 フレームとする。
- ・メモリ
 - 遷移時間の高速化のために、インゲーム中にアウトゲームの一部リソースも保持しておく設計とする。最大使用量を 1GB とする。
- ・遷移時間
 - インゲームやアウトゲームへの遷移時間は競合と同じレベルを目指す。時間にすると 3 秒以内。
- ・熱
 - 競合と同じレベル。検証端末で連続 1 時間は熱くならない。(充電していない状態)
- ・バッテリー
 - 競合と同じレベル。検証端末で連続 1 時間でバッテリー消費は 20% ほど。

このように目指すべき指標が決まれば基準となる端末で触ってみましょう。まったく目標に届かないという状態でなければ指標としてはよいでしょう。

ゲームジャンルによる最適化

今回は滑らかに動くことがテーマだったのでフレームレートを 60 フレームとしました。他にもリズムアクションゲームや、FPS（ファーストパーソンシューティング）のような判定がシビアなゲームも高フレームレートが望ましいでしょう。しかし高フレームレートにはデメリットもあります。それはフレームレートが高いほどバッテリーを消費することです。他にもメモリは使用量が多いほどサスPEND時に OS からキルされやすくなります。このようなメリット・デメリットを考慮しゲームジャンルごとに適切な目標を決めましょう。

1.1.2 メモリの最大使用量を把握する

この節ではメモリの最大使用量について焦点をあてます。メモリの最大使用量を知るためにには、まずはサポート対象となるデバイスがメモリをどれぐらい確保できるかを把握しましょう。基本的には動作を保証する端末のうち最低スペックのデバイスで検証するのがよいでしょう。ただし OS バージョンによってメモリ確保の仕組みが変更されている可能性があるため、出来ればメジャーバージョンが違う端末を複数用意するのがよいでしょう。また計測するツールによっても計測ロジックが違うため、使用するツールは必ず 1 つに限定しましょう。

参考までに筆者が iOS にて検証した内容を記載しておきます。検証プロジェクトでは Texture2D をランタイムで生成し、どれぐらいでクラッシュするかを計測しました。コードは次のとおりです。

▼リスト 1.1 検証コード

```
1: private List<Texture2D> _textureList = new List<Texture2D>();  
2: ...  
3: public void CreateTexture(int size) {  
4:     Texture2D texture = new Texture2D(size, size, TextureFormat.RGBA32, false);  
5:     _textureList.Add(texture);  
6: }
```

第1章 パフォーマンスチューニングを始めよう

検証結果は図 1.1 のようになりました。

端末	搭載メモリ (GB)	OS	メモリ使用量 (GB)
iPhone6	1	12.4.1	0.65
iPhone6S	2	10.0.1	1.37
		11.3	2.61
		12.1.2	1.37
		13.6	1.42
iPhone7	2	10.3.1	1.31
		11	2.64
		12.4	1.37
		13.3.1	1.42
iPhone7 Plus	3	12.0.1	2.00
iPhoneX	3	12.1	1.76
iPhoneXR	3	13.5.1	1.81

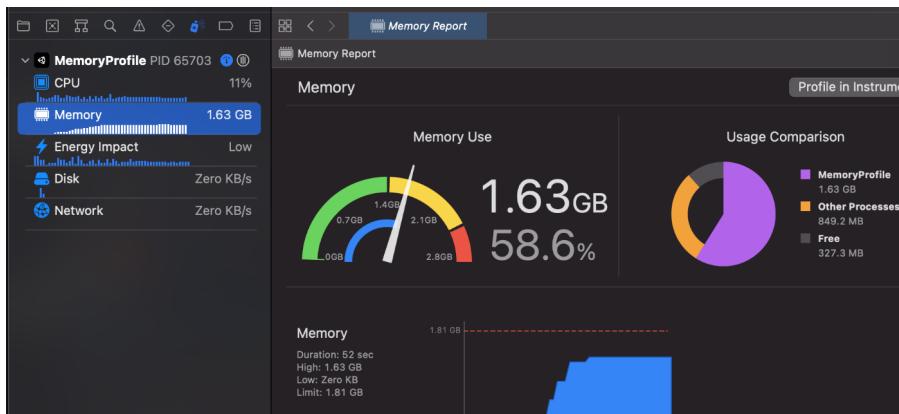
▲図 1.1 クラッシュのしきい値

検証環境は Unity 2019.4.3、Xcode 11.6 を用いて、Xcode の Debug Navigator の Memory セクションの数値を参考にしています。この検証結果から iPhone6S や 7 などの搭載メモリが 2GB の端末では、メモリを 1.3GB 以内におさめておくのがよいと言えるでしょう。また iPhone6 のような搭載メモリが 1GB の端末をサポートする場合、メモリ使用量の制約がかなり厳しくなるのも見て取れます。他にも特徴的なのは iOS11 の場合、メモリ管理の仕組みが違うためかメモリ使用量が突出していることです。検証する際にはこのような OS による差分がまれにあることを注意してください。

図 1.1 では検証環境が少し古いため、執筆時の最新環境を用いて一部計測し直しました。環境は Unity 2020.3.25、2021.2.0 の 2 バージョンと Xcode13.3.1 を用いて、OS バージョンが 14.6 と 15.4.1 の iPhoneXR にビルドを行いました。結果として計測値に差異はとくにみられなかったので、まだ信頼性のあるデータかと思います。

メモリ計測ツール

計測するツールとして推奨したいのは、Xcode や AndroidStudio などのネイティブ準拠のツールです。たとえば Unity Profiler の計測では、プラグインなどが独自で確保したネイティブメモリ領域は計測対象外になります。他にも IL2CPP ビルドの場合、IL2CPP のメタデータ（100MB ほど）も計測対象になってしまいます。一方でネイティブツールの Xcode の場合、アプリで確保されたメモリはすべて計測されます。そのためより正確に値が計測されるネイティブ準拠のツールを使うのがよいでしょう。



▲図 1.2 Xcode Debug Navigator

1.1.3 動作保証端末を決める

パフォーマンスチューニングをどこまで行うかを決める指標として、最低限の動作を保証する端末を決めることが重要です。この動作保証端末の選定は経験がないと即座に決めることが難しいですが、勢いで決めずにまずは低スペック端末の候補を洗い出すことから始めましょう。

筆者がオススメする方法としては「SoC のスペック」を計測したデータを参考にする方法です。具体的にはベンチマーク計測アプリで計測されたデータを Web 上で探しします。まずは基準とした端末のスペックを把握し、それよりいくらか計測値の低い端

末を何パターンか選定しましょう。

端末が洗い出せれば、実際にアプリケーションをインストールして動作確認をします。動作が重くても落胆しないでください。これでやっと何を削ぎ落とすか議論できるスタートラインに立てました。次項では機能を削ぎ落とすにあたって考えておきたい大事な仕様について紹介します。

ベンチマーク計測アプリはいくつかありますが、筆者は AnTuTu を基準としています。こちらは計測データのまとめサイトが存在していますし、有志の方も積極的に計測データを報告してくれているのが理由です。

1.1.4 品質設定の仕様を決める

市場にさまざまなスペックの端末が溢れ返っているいま、1つの仕様で多くの端末をカバーするのは難しいでしょう。そのため近年ではゲーム内にいくつかの品質設定を設けることで、さまざまな端末に対して安定した動作を保証することが主流となっています。

たとえば次のような項目を品質設定の高、中、低で切り分けるとよいでしょう。

- ・画面解像度
- ・オブジェクトの表示数
- ・影の有無
- ・ポストエフェクト機能
- ・フレームレート
- ・CPU 負荷の高いスクリプトのスキップ機能など

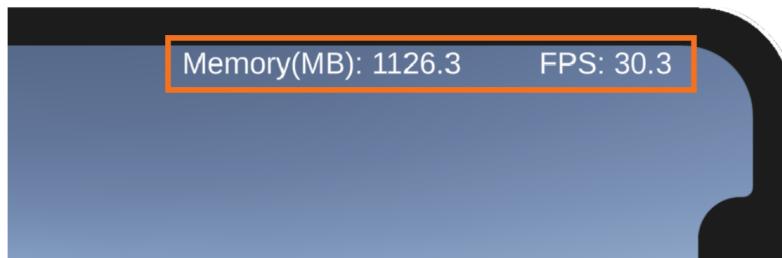
ただし見た目のクオリティを下げる事になるので、ディレクターと相談しながらどこまでのラインをプロジェクトとして許容できるかと一緒に探っていきましょう。

1.2 未然に防ぐ

性能低下は不具合と同じく時間が経過することでさまざまな原因が絡み合い、調査の難易度が上がってしまいます。なるべく早期に気づけるような仕組みをアプリケーションに実装しておくのがよいでしょう。実装が簡単で効果の高い方法としては、画面上に現在のアプリケーションの状態を表示しておくことです。少なくとも次の要素

は画面上に常に表示することを推奨します。

- 現在のフレームレート
- 現在のメモリ使用量



▲図 1.3 パフォーマンスの可視化

フレームレートは体感で性能が低下していることがわかりますが、メモリはクラッシュすることでしか検知できません。図 1.3 のように画面に常に表示しておくだけでメモリリークが早期に発見できる確率が上がるでしょう。

この表示方法はさらに工夫することで効果が高くなります。たとえばフレームレートの達成したい目標が 30 フレームなら、25~30 フレームを緑色、20~25 フレームを黄色、それ以下を赤色にしてみましょう。そうすることで直感的にアプリケーションが基準を満たしているのか一目でわかるようになります。

1.3 パフォーマンスチューニングに取り組む

いくら未然に性能低下を防ぐ努力をしても、すべてを防ぐことは厳しいでしょう。これは仕方ありません。開発を進める上で性能低下は切っても切り離せないものです。パフォーマンスチューニングと向き合う時は必ず来るでしょう。以降ではどのようにしてパフォーマンスチューニングに取り組んでいくべきかを解説します。

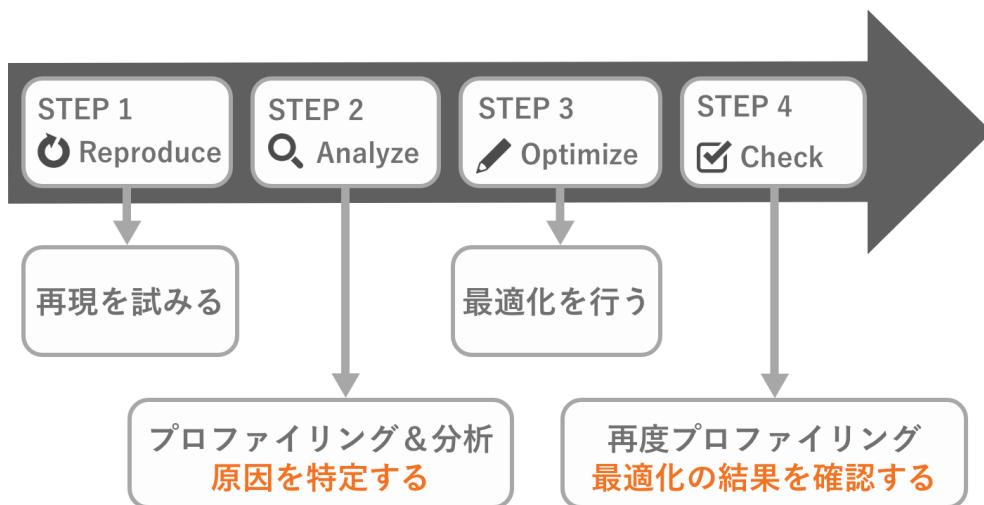
1.3.1 心構え

パフォーマンスチューニングを始める前にまずは大事な心構えを紹介します。たとえばフレームレートが低下しているアプリケーションがあるとしましょう。明らかにリッチなモデルが複数体表示されています。周りの人はこのモデルがきっと原因に違いないと言っています。果たして本当にそうでしょうか、その証拠はどこにあるのか

しっかり見定める必要があるでしょう。パフォーマンスチューニングの心構えとして、必ず意識して欲しいことが2つあります。

1つ目は、計測し、原因を特定することです。**推測ではいけません**。

2つ目は、修正したら必ず結果を比較しましょう。前後のプロファイルを比較するといいでしょう。ポイントとしては修正箇所だけでなく、全体で性能低下が発生していないかを確認することです。パフォーマンスチューニングの恐ろしい所は、修正箇所は高速化したが他の箇所で負荷が上がり、全体でみると性能低下しているという事がまれにあります。これでは本末転倒です。



▲図 1.4 パフォーマンスチューニングの心構え

確実に原因を突き止め確実に速くなったことを確認する。それがパフォーマンスチューニングの大事な心構えなのです。

1.3.2 性能低下の種類

性能低下といってもそれぞれが指し示すものは違うでしょう。本書では大別して以下の3つと定義します。(図1.5)



▲図 1.5 性能低下の原因

まずクラッシュする場合は「メモリ超過」か「プログラムの実行エラー」の2種類に大別されるでしょう。後者についてはパフォーマンスチューニングの領域ではないため、具体的な内容は本書では取り扱いません。

次に画面の処理落ちやロードが長いのは「CPU や GPU の処理時間」が原因の大半を占めるでしょう。以降では「メモリ」と「処理時間」に焦点を当てて性能低下を深掘ります。

1.4 メモリ超過の原因切り分け

クラッシュの原因にはメモリ超過が考えられると取り上げました。ここからはメモリ超過の原因をさらに分解していきましょう。

1.4.1 メモリがリークしている

メモリ超過の原因の1つとしてメモリのリークが考えられます。これを確認するためにシーンの遷移に伴ってメモリ使用量が徐々に増えるかどうか確認しましょう。ここでいうシーンの遷移とはただの画面遷移ではなく、大きく画面が転換する事を指します。たとえばタイトル画面からアウトゲーム、アウトゲームからインゲームなどです。計測する際は次の手順で進めましょう。

1. あるシーンでのメモリ使用量をメモする
2. 違うシーンに遷移する
3. 「1」～「2」の流れを3～5回ほど繰り返す

計測した結果、メモリ使用量が純増していれば間違いなく何かがリークしています。これは目に見えない不具合と言ってもいいでしょう。まずはリークを無くしましょう。

また「2」の遷移を行う前に、いくつか画面遷移を挟んでおくのもよいでしょう。なぜなら特定の画面でロードしたリソースだけが、例外的にリークしていたという事も

あり得るためです。

リークに確信が持てたらリークの原因を探りましょう。「1.5 メモリリークを調査しよう」にて具体的な調査方法について解説します。

繰り返す理由

これは筆者の経験談ですが、リソース解放後（UnloadUnusedAssets 後）にタイミングの問題でいくつかリソースが解放されていないケースがありました。この解放されていないリソースは次のシーンに遷移すると解放されます。これに対して、遷移を繰り返した際にメモリ使用量が徐々に増加する場合は、いずれクラッシュを引き起こします。前者の問題と切り分けるために本書では、メモリ計測時に遷移を何度も繰り返す手法を推奨しています。

ちなみに前者のような問題があった場合、リソース解放時には何かしらのオブジェクトがまだ参照を握っており、その後に解放されているのでしょうか。致命傷とはなりませんが、原因調査を行い解決しておくのが良いでしょう。

1.4.2 メモリ使用量が単純に多い

リークしていない状態でメモリ使用量が多い場合、削減できる箇所を探っていく必要があります。「1.6 メモリを削減しよう」にて、具体的な方法について解説します。

1.5 メモリリークを調査しよう

まずはメモリリークを再現させた上で、次に紹介するツールを用いて原因を探っていきましょう。ここでは簡単にツールの特徴を説明します。ツールの使い方の詳細は第3章「プロファイリングツール」にて取り扱うので、参考にしながら調査してみてください。

Profiler (Memory)

Unity エディターにデフォルトで搭載されているプロファイラーツールです。そのため気軽に計測ができます。基本的には「Detailed」かつ「Gather object references」を設定した状態でメモリをスナップショットし、調査することになるでしょう。このツールでの計測データは、他のツールと違いスナップショットの比較はできません。使い

方の詳細は「3.1.3 Memory」を参照してください。

Memory Profiler

こちらは Package Manager からインストールする必要があります。ツリーマップではグラフィカルにメモリの内容が表示されます。Unity 公式でサポートされており、現在も頻繁にアップデートされています。v0.5 以降、参照関係を追跡する方法が大きく改善されているため最新版の利用を推奨します。使い方の詳細は「3.4 Memory Profiler」を参照してください。

Heap Explorer

こちらは Package Manager からインストールする必要があります。個人が開発したツールですが、非常に使いやすく動作も軽量です。参照関係を追跡する際にリスト形式で見ることが可能で、v0.4 以前の Memory Profiler の痒い所に手が届いたツールです。v0.5 の Memory Profiler が利用できない際の代替ツールとして利用すると良いでしょう。使い方の詳細は「3.5 Heap Explorer」を参照してください。

1.6 メモリを削減しよう

メモリを削減するポイントは大きな箇所から削ることです。なぜなら 1KB を 1,000 個削っても 1MB の削減にしかなりません。しかし 10MB のテクスチャを圧縮して 2MB になると 8MB も削減されます。費用対効果を考えて、まずは大きいものから削減していくことを意識しましょう。

本節ではメモリ削減に使用するツールは Profiler(Memory) として話を進めます。使用したことがない方は「3.1.3 Memory」にて詳細を確認してください。

以降の節では、削減する際に見るべき項目を取り上げていきます。

1.6.1 Assets

Simple View で Assets 関連が多い場合、不要なアセットやメモリリークの可能性が考えられます。ここでいう Assets 関連とは図 1.6 の矩形で囲った部分です。



▲図 1.6 Assets 関連の項目

この場合、調査するべきことは次の3つです。

不要アセット調査

不要なアセットとは、現在のシーンにまったく必要のないリソースを指します。たとえばタイトル画面でしか使用しないBGMがアウトゲーム内でもメモリに常駐しているなどです。まずは今のシーンに必要なものだけにしましょう。

重複アセット調査

これはアセットバンドル対応を行なった際によく発生します。アセットバンドルの依存関係の切り分け方が良くないために、同じアセットが複数のアセットバンドルに含まれている状態です。しかし依存関係は切り分けすぎてもダウンロードファイル数の増加や、ファイル展開コストの増加に繋がります。この辺りは計測しながらバランス感覚を養う必要があるかもしれません。アセットバンドルに関する詳細は、「2.4.6 AssetBundle」を参照してください。

レギュレーションをチェックする

それぞれの項目のレギュレーションが守られているかを見直しましょう。レギュレーションがない場合は、メモリの見積もりが適切にできていない可能性があるので確認しましょう。

たとえばテクスチャなら次のような内容をチェックをするとよいでしょう。

- サイズは適切か
- 圧縮設定は適切か

- MipMap の設定が適切か
- Read/Write の設定が適切かなど

それぞれのアセットごとに気をつけるべき点は第4章「Tuning Practice - Asset」を参照してください。

1.6.2 GC (Mono)

Simple View で GC (Mono) が多い場合は、一度に大きな GC.Alloc が発生している可能性が高いです。もしくは毎フレーム GC.Alloc が発生してメモリが断片化しているかもしれません。これらが原因でマネージドヒープ領域が余計に拡張している可能性があります。この場合は GC.Alloc を地道に削減していきましょう。

マネージドヒープに関しては「2.1.5 メモリ」を参照してください。同様に GC.Alloc に関する詳細は「2.5.1 スタックとヒープ」にて取り扱っています。

バージョンによる表記違い

2020.2 以降では「GC」と表示されていますが、2020.1 以下のバージョンまでは「Mono」と記載されています。どちらもマネージドヒープの占有量を指しています。

1.6.3 Other

Detailed View で怪しい項目がないかを確認しましょう。たとえば **Other** の項目などは一度開いて調査するとよいでしょう。

Name	Memory
▶ Assets (33256)	490.0 MB
▼ Other (792)	338.6 MB
System.ExecutableAndDLLs	208.0 MB
▶ SerializedFile (592)	64.4 MB
▶ Profiling (8)	32.5 MB
▶ PersistentManager.Remapper (1)	16.0 MB
▶ Rendering (9)	8.2 MB
▶ Managers (28)	7.9 MB
Objects	1.3 MB
▶ MemoryPools (16)	105.4 KB
▶ Physics2D Module (1)	104.1 KB
▶ File System (2)	79.7 KB
▶ Job System (1)	2.9 KB
▶ ParticleSystem Module (2)	2.3 KB
▶ Animation Module (4)	1.7 KB
▶ MemoryProfiling (1)	96 B

▲図 1.7 Other の項目

筆者の経験では SerializedFile や PersistentManager.Remapper が、かなり肥大化していた事例がありました。複数プロジェクトで数値比較ができる場合、一度比較してみるのも良いでしょう。それぞれの数値を見比べると異常値がわかるかもしれません。詳細は「2. Detailed ビュー」を参照してください。

1.6.4 プラグイン

ここまで Unity の計測ツールを使って原因の切り分けをしてきました。しかし Unity で計測できるのは Unity が管理しているメモリのみです。つまりプラグインで独自に確保しているメモリ量などは計測されません。ThirdParty 製品で余分にメモリを確保していないかを調べましょう。

具体的にはネイティブ計測ツール（Xcode の Instruments）を利用します。詳細は「3.7 Instruments」を参照してください。

1.6.5 仕様を検討する

これは最後の手段です。ここまでに取り上げてきた内容で削れる部分がない場合、仕様を検討するしかありません。以下に例を挙げます。

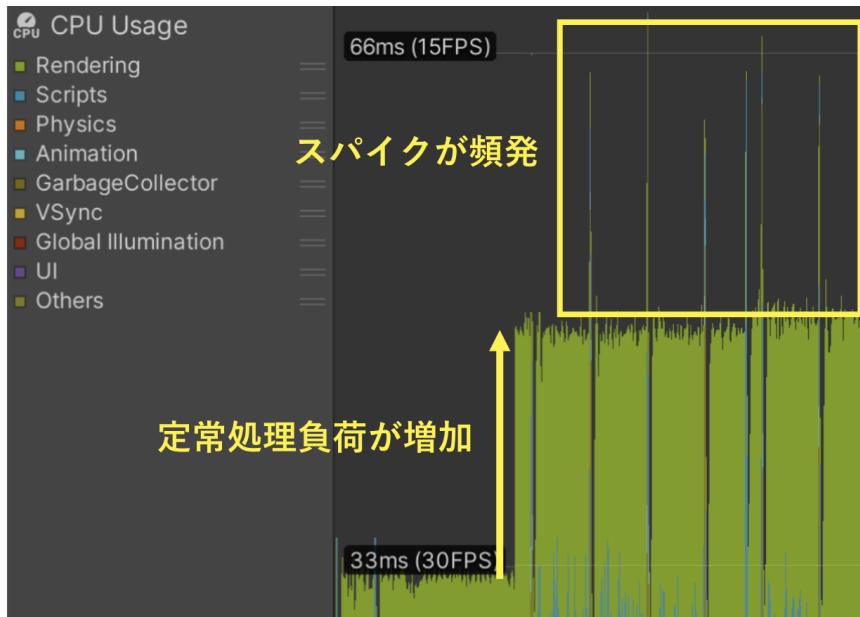
- テクスチャの圧縮率を変更する
 - テクスチャの一部分だけ圧縮率を一段回上げる
- ロード・アンロードするタイミングを変更する
 - 常駐メモリに載っているオブジェクトを解放し都度ロードにする
- ロード仕様を変更する
 - インゲームでロードするキャラクターのバリエーションを1種類減らす

どれも影響範囲は大きく、ゲームの面白さに根本的に関わる可能性もあります。そのため仕様の検討は最後の手段なのです。そうならないように早い段階でメモリの見積もり、計測はしっかりと行いましょう。

1.7 処理落ちの原因切り分け

ここからは処理時間を計測し最適化していく流れについて紹介します。画面の処理落ちは「瞬間的な処理落ち」か「定常的な処理落ち」かで対処法が変わります。

瞬間的な処理落ちは針のように尖った処理負荷が計測されます。その見た目からスパイクとも言われています。



▲図 1.8 スパイクと定常処理負荷

図 1.8 では、定常負荷が急に増加しており、さらに定期的なスパイクも発生している計測データです。どちらの事象もパフォーマンスチューニングが必要でしょう。まずは比較的シンプルな瞬間的な負荷調査を解説します。その後、定常的な負荷調査について解説します。

1.8 瞬間的な負荷を調査する

スパイクの調査方法としては Profiler(CPU) を用いて原因を調査します。

ツールの詳細な使い方は「3.1.2 CPU Usage」を参照してください。まずは原因が GC によるものかどうかを切り分けましょう。原因の切り分け自体には Deep Profile は必要ありませんが、解決するためには必要になるでしょう。

1.8.1 GC によるスパイク

GC (ガベージコレクション) が発生している場合、GC.Alloc を削減する必要があります。どの処理がどれくらいアロケートしているかは Deep Profile を利用するとよいでしょう。まず削減すべきところは費用対効果の高い箇所です。次にあげるような項

目を中心に修正するとよいでしょう。

- 毎フレームアロケートしている箇所
- 大量のアロケートが発生している箇所

アロケーションは少ないほど良いですが、必ずゼロにすべきということではありません。たとえば生成処理 (Instantiate) などで発生するアロケートは防ぎようがないでしょう。このような場合は都度オブジェクトを生成せずに、オブジェクトを使い回すプリーリングといった手法が有効です。GC の詳細は「[2.5.2 ガベージコレクション](#)」を参照してください。

1.8.2 重い処理によるスパイク

GC が原因でない場合は何かしらの重い処理が瞬間的に行われています。ここでも Deep Profile を利用して何の処理がどれくらい重いのかを調査し、もっとも処理に時間がかかっている箇所を見直していきましょう。

よくある一時的な重い処理といえば次のようなものが考えられるでしょう。

- Instantiate 処理
- 大量のオブジェクト、あるいは階層が深いオブジェクトのアクティブ切り替え
- 画面のキャプチャ処理など

このようにプロジェクトコードにかなり依存する部分なので、解決方法は一律にこうすればよいというものはありません。実際に計測して原因が判明したらプロジェクトメンバーに計測結果を共有し、どのように改善すべきか検討してみてください。

1.9 定常的な負荷を調査する

定常的な処理負荷を改善する際は、1 フレーム内での処理をいかに削減するかが重要になります。1 フレーム内で行っている処理は大別すると CPU 処理と GPU 処理に分けることができます。まずはこの 2 つの処理のうち、どちらがボトルネックになっているのか、もしくは同じくらいの処理負荷なのかを切り分けるとよいでしょう。

CPU にボトルネックがある状態を CPU バウンド、GPU にボトルネックがある状態を GPU バウンドと呼びます

切り分ける簡単な方法として、次に当てはまる内容があれば GPU バウンドの可能性が高いでしょう。

- 画面の解像度を下げた際に処理負荷が劇的に改善する
- Profiler で計測した際に **Gfx.WaitForPresent** が存在する

逆にこれらがない場合は CPU バウンドの可能性があります。以降では CPU バウンドと GPU バウンドの調査方法について説明します。

1.9.1 CPU バウンド

CPU バウンドは前節でも取り扱った CPU (Profiler) を利用します。Deep Profile を利用して調査し、特定のアルゴリズムに大きな処理負荷がかかっていないかを確認します。大きな処理負荷がない場合は均等に重いということになるので地道に改善していきましょう。地道な改善を重ねても到底目標の削減値に及ばない場合は「1.1.4 品質設定の仕様を決める」に立ち戻って再考し直すのもよいでしょう。

1.9.2 GPU バウンド

GPU バウンドの場合は Frame Debugger を利用して調査するのがよいでしょう。使い方の詳細は「3.3 Frame Debugger」を参照してください。

解像度が適切か

GPU バウンドの中でも解像度は GPU の処理負荷に大きく影響します。そのため解像度を適切に設定できていない状態であれば、まずは最優先に適切な解像度にする必要があります。

まずは想定の品質設定で適切な解像度になっているかを確認しましょう。確認方法としては Frame Debugger 内で処理しているレンダーターゲットの解像度に注目するとよいでしょう。次にあげるようなことを意図的に実装していない場合、最適化に取り組みましょう。

- UI 要素だけデバイスのフル解像度で描画されている
- ポストエフェクト用の一時テクスチャの解像度が高いなど

不要なオブジェクトがあるか

Frame Debugger で不要な描画がないかを確認しましょう。たとえば必要のないカメラがアクティブ状態になっており、裏で関係のない描画が行われているかもしれません。また他の遮蔽物によって直前の描画がムダになっているケースが多いのであれば、オクルージョンカリングを検討するのもよいでしょう。オクルージョンカリングの詳細は「7.5.3 オクルージョンカリング」を参照してください。

オクルージョンカリングはデータを事前に準備する必要があり、そのデータをメモリに展開するためメモリ使用量が増えることにも注意しましょう。このようにパフォーマンスを向上させるために、メモリに事前に準備した情報を構築するのによくある手法です。メモリとパフォーマンスは反比例することが多いので、何かを採用する際はメモリも意識すると良いでしょう。

バッチングが適切か

描画対象をまとめて描画する事をバッチングと言います。まとめて描画されることで描画効率が向上するため GPU バウンドに効果があります。たとえば Static Batching を利用すると複数の動かないオブジェクトのメッシュをまとめてくれます。

バッチングに関しては色々な方法があるため、代表的なものをいくつかあげておきます。気になったものがあれば「7.3 ドローコールの削減」を参照してみてください。

- Static Batching
- Dynamic Batching
- GPU Instancing
- SRP Batcher など

個別に負荷を見る

まだ処理負荷が高い場合は個別にみていくしかありません。オブジェクトの頂点数が多すぎたり、Shader の処理に原因があるかもしれません。これを切り分けるには個々のオブジェクトに対してアクティブを切り替え、処理負荷の変わり具合を見ていきます。具体的には背景を消してみてどうなるか、キャラクターを消してみてどうなるか、といったようにカテゴリごとに絞っていきます。処理負荷の高いカテゴリがわ

ければ、さらに次のような要素をみていくとよいでしょう。

- 描画するオブジェクトが多すぎないか
 - まとめて描画できないか検討する
- 1 オブジェクトの頂点数が多すぎないか
 - リダクション、LOD を考える
- シンプルな Shader に付け替えて処理負荷が改善するか
 - Shader の処理を見直す

それ以外

それぞれの GPU 処理が積み上がって重いと言えるでしょう。この場合は地道に1つ1つ改善していくしかありません。

またこちらも CPU バウンドと同様に、到底目標の削減値に及ばない場合「1.1.4 品質設定の仕様を決める」に立ち戻って再考するのもよいでしょう。

1.10 まとめ

この章では「パフォーマンスチューニング前」と「パフォーマンスチューニング中」に気をつけて欲しい事を取り上げました。

パフォーマンスチューニング前に気をつけることは以下のとおりです。

- 「指標」「動作保証端末」「品質設定の仕様」を決めること
 - 量産前までに検証を行い指標を確定させること
- 性能低下に気づきやすい仕組みを作っておくこと

パフォーマンスチューニング中に気をつけることは以下のとおりです。

- 性能低下の原因を切り分け、適切な対処を行うこと
- 「計測」、「改善」、「再度計測(結果のチェック)」の一連の流れを必ず行うこと

パフォーマンスチューニングはここまで解説してきたように、計測して原因を切り分けていくことが重要です。このドキュメントに記載されていない事例が発生しても、その基礎が守られていれば大きな問題になることはないでしょう。一度もパフォーマンスチューニングをしたことがない方は、ぜひこの章の内容を実践してもらえると幸いです。



PERFORMANCE TUNING BIBLE

CHAPTER

02

第2章

基礎知識

第 2 章

基礎知識

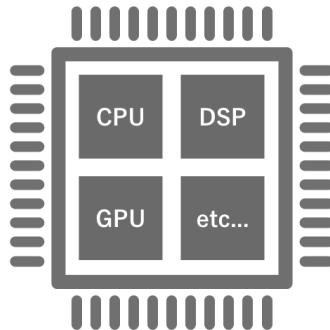
パフォーマンスチューニングを行う際には、そのアプリケーションの全体を調査し、修正する必要があります。したがって効果的にパフォーマンスチューニングを行うためには、ハードウェアから 3D のレンダリング、Unity の仕組みに至るまで幅広い知識が求められます。そこでこの章では、パフォーマンスチューニングを行うために必要な基礎知識についてまとめます。

2.1 ハードウェア

コンピューターのハードウェアは主に、入力装置、出力装置、記憶装置、演算装置、制御装置という 5 つの装置から構成されます。これらをコンピューターの五大装置と呼びます。この節ではこれらのハードウェアのうち、パフォーマンスチューニングを行う上で重要なものについて、その基礎知識をまとめます。

2.1.1 SoC

コンピューターはさまざま装置で構成されています。代表的な装置としては、制御や演算を行うための CPU、グラフィックスの計算を行うための GPU、音声や映像のデジタルデータを処理する DSP といったものが挙げられます。大半のデスクトップ PC などでは、これらは別々の集積回路として独立しており、それらを組み合わせることでコンピューターを構成します。これに対してスマートフォンでは、小型化や省電力化のためにこれらの装置が 1 つのチップ上に実装されています。これを System-on-a-chip、すなわち SoC と呼びます。



▲図 2.1 SoC

2.1.2 iPhone・Android と SoC

スマートフォンはその機種によって搭載されている SoC が異なります。

たとえば iPhone には、Apple 社により設計されている A シリーズと呼ばれる SoC が使用されています。このシリーズは A15 のように、「A」という文字と数字の組み合せによる命名がされており、バージョンアップに伴って数字が大きくなっています。

これに対して多くの Android では Snapdragon と呼ばれる SoC が使用されています。これは Qualcomm という会社が製造している SoC で、Snapdragon 8 Gen 1 や Snapdragon 888 のように命名されます。

また、iPhone が Apple 社により製造されているのに対し、Android はさまざまなメーカーが製造しています。このため、Android には以下の表 2.1 に示すように Snapdragon 以外にもさまざまな SoC が存在します。Android で機種依存の不具合が起こりやすいのはこのためです。

▼表 2.1 Android の主要な SoC

シリーズ名	メーカー	搭載されている端末の傾向
Snapdragon	Qualcomm 社	幅広い端末で使用されている
Helio	MediaTek 社	一部の廉価端末で使用されている
Kirin	HiSilicon 社	Huawei 社製の端末
Exynos	Samsung 社	Samsung 社製の端末

パフォーマンスチューニングを行う際には、その端末の SoC に何が使用されていて、それがどのようなスペックのものなのかを理解することが重要です。

Snapdragonの命名はこれまで、「Snapdragon」という文字列と3桁の数字の組み合わせが主流でした。

この数字には意味があります。800番台はフラッグシップモデルで、いわゆるハイエンド端末に搭載されます。ここから小さい数字になるほど性能と価格が低下し、400番台になるといわゆるローエンド端末になります。

たとえ400番台であっても発売時期が新しいほど性能が向上するため一概には言えませんが、基本的には数字が大きいほど性能が高いとみなすことができます。

さらに、この命名規則だと近いうちに番号が足りなくなってしまうため、今後はSnapdragon 8 Gen 1のような命名に変更することが2021年に発表されました。

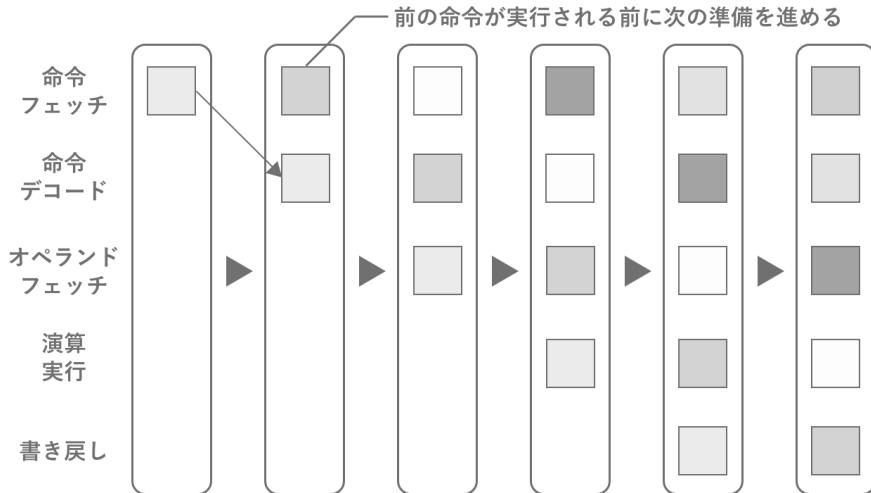
このような命名規則については端末の性能を判別するための指標となるため、パフォーマンスチューニングの際には覚えておくと便利です。

2.1.3 CPU

CPU (Central Processing Unit) はコンピューターの頭脳とも言うべき存在で、プログラムの実行はもちろん、コンピューターを構成するさまざまなハードウェアとの連携も行っています。実際にパフォーマンスチューニングする場合に、CPUの中でどういう処理が行われてどういう特性があるかを知ることで役立つので、ここではパフォーマンス観点で説明します。

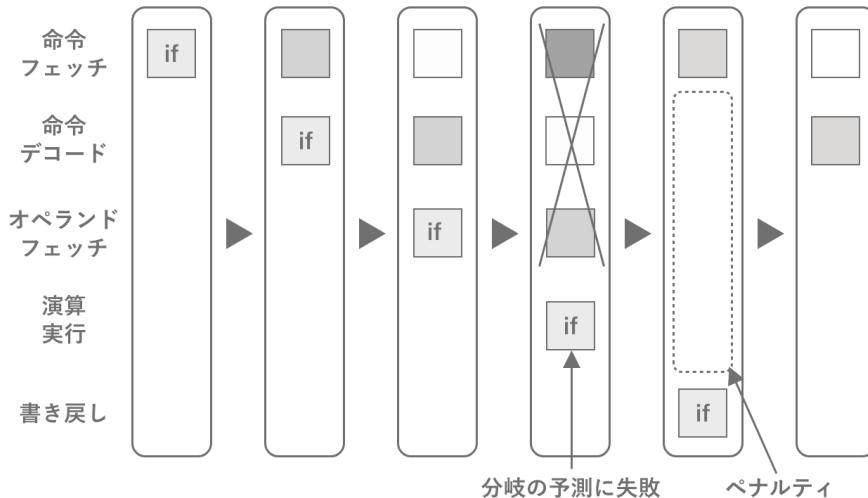
CPUの基本

プログラムの実行速度を決めるのは単純な演算能力だけではなく、複雑なプログラムのステップをいかに高速に実行できるかどうかです。たとえばプログラムの中には四則演算もありますが、分岐処理もあります。CPUにとっては次にどの命令が呼び出されるかは、プログラムを実行するまではわかりません。そこでCPUは、さまざまな命令を高速に連續で処理できるようハードウェアが設計されています。



▲図 2.2 CPU のパイプライン・アーキテクチャ

CPU 内部での命令が処理する流れをパイプラインと呼び、パイプラインの中で次の命令を予測しながら処理されています。次の命令がもし予測されていない場合は、パイプライン・ストールと呼ばれる一時停止が発生し一度リセットされます。ストールする原因の大部分は分岐処理です。分岐自体もある程度は結果を先読みしていますが、それでも間違えることはあります。内部構造を覚えなくてもパフォーマンスチューニングは可能ですが、こういうことを知っておくだけでもコードを書く際にループの中で分岐を避けるなどが意識できるようになります。



▲図 2.3 CPU のパイプライン・ストール

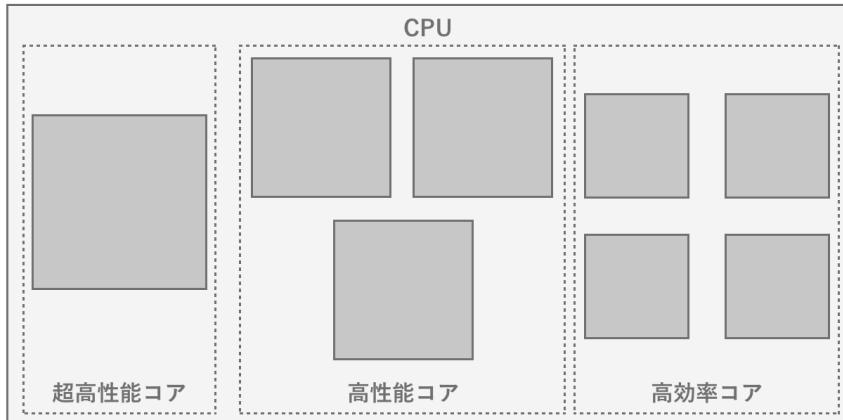
CPU の演算能力

CPU の演算能力は、クロック周波数（単位は Hz）とコア数で決定されます。クロック周波数は 1 秒間に何回 CPU が動作できるかを表します。そのためクロック周波数が高ければ高いほどプログラムの実行速度は速いです。

一方でコア数は、CPU の並列演算能力に寄与します。コアは CPU の動作する基本単位で、それが複数ある場合はマルチコアと呼ばれています。もともとはシングルコアのみしかありませんでしたが、シングルコアの場合は複数のプログラムを実行させるために、交互に動作させるプログラムを切り替えていました。これをコンテキストスイッチと呼び、そのコストはとても高いです。スマートフォンに慣れている場合、動作しているアプリ（プロセス）は常に 1 つと思うかもしれません、実際には OS などさまざまなプロセスが並行して動作しています。そこでこのような状況でも最適な処理能力を提供するために、コアを複数積んだマルチコアが主流となりました。スマートフォン向けの場合、2022 年現在 2-8 コア程度が主流です。

近年のマルチコア（とくにスマートフォン向け）は非対称コア（big.LITTLE）を搭載する CPU が主流となってきました。非対称コアとは、高性能コアと省電力コアと一緒に搭載しているような CPU を指します。非対称コアのメリットは普段は省電力コアのみを動かして電池消費を節約し、ゲームなどパフォーマンスを出さないといけない時にコアを切り替えて使えるというところです。ただし省電力コアの分、並列性能の最

大値は低下するので、コア数だけでは判断できないことに注意が必要です。

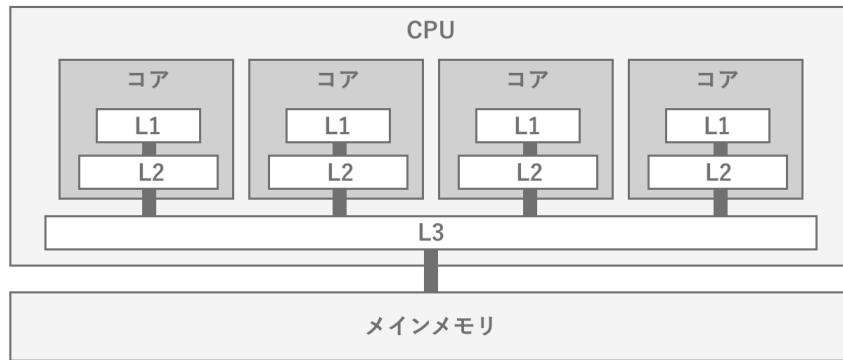


▲図 2.4 Snapdragon 8 gen 1 の異種コア構成

またプログラムが複数のコアを使い切れるかどうかは、プログラムの並列処理の記述に依存します。たとえばゲームエンジン側で物理エンジンを別スレッドで動作させるなどの効率化してあるケースや、Unity の JobSystem などを通じて並列処理を活用しているケースもありますが、ゲームのメインループ自体の動作は並列化できないため、マルチコアであってもコア自体の性能は高いほうが有利です。

CPU のキャッシュメモリ

CPU とメインメモリは物理的に離れた場所に存在し、アクセスするためにほんの僅かな時間（レイテンシ）が必要になります。そのためプログラムを実行する際にメインメモリに格納されたデータにアクセスしようとすると、この距離が性能の大きなボトルネックとなります。そこでこのレイテンシの問題を解決するために、CPU 内部にはキャッシュメモリが搭載されています。キャッシュメモリは、主にメインメモリに格納されているデータの一部を格納することで、プログラムが必要とするデータに素早くアクセスできるようにします。キャッシュメモリには L1、L2、L3 キャッシュがあり、数字が小さいほど高速ですが小容量です。どれくらい小容量かというと、L3 キャッシュでも 2-4MB レベルです。そのため CPU キャッシュにはすべてのデータを保存することはできず、あくまで直近扱っているデータのみがキャッシュされます。



▲図 2.5 CPU の L1、L2、L3 キャッシュとメインメモリとの関係

そこでプログラムのパフォーマンスを高めるためには、データをいかにキャッシュに効率よく載せるかが鍵となります。プログラム側で自由にキャッシュを制御できないので、データの局所性が重要となります。ゲームエンジンにおいてはデータの局所性を意識したメモリ管理は難しいですが、Unity の JobSystem など一部の仕組みではデータの局所性を高めたメモリ配置を実現できます。

2.1.4 GPU

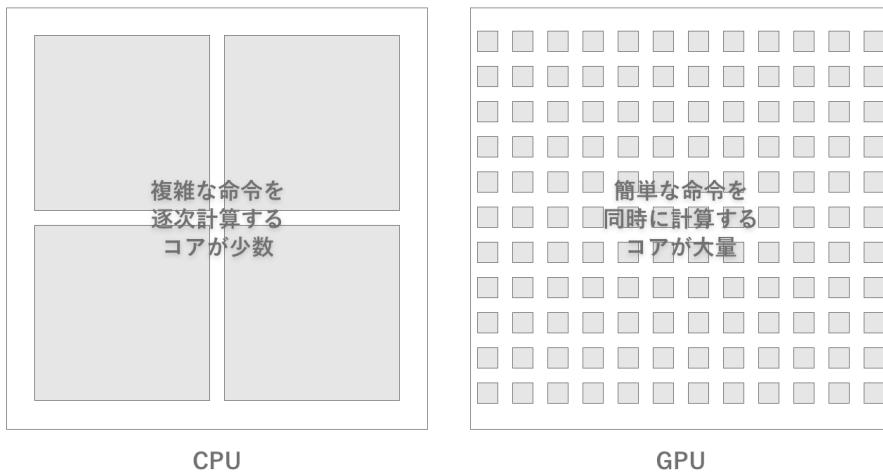
CPU がプログラムを実行することに特化している一方で、**GPU** (Graphics Processing Unit) は画像処理やグラフィックスの描画に特化したハードウェアです。

GPU の基本

GPU はグラフィックス処理に特化させるため、CPU とは大きく構造が異なり、単純な計算を大量かつ並行して処理できるような設計となっています。たとえば 1 枚の画像を白黒にしたい場合、CPU を使って計算する場合はある座標の RGB 値をメモリから読み取り、グレースケールに変換してメモリに戻す処理を画素毎に実行する必要があります。このような処理は分岐もなく、かつそれぞれの画素の計算は他の画素の結果に依存しないので、各画素における計算を並列で行うことが容易です。

そこで GPU では大量のデータに対して同じ演算を適用するような並列処理が高速に実行でき、その結果グラフィックス処理が高速に実行できます。とくにグラフィックス系では浮動小数点の演算が大量に必要となるので、GPU は浮動小数点演算が得意です。そのため 1 秒間に何回浮動小数点の演算が行えるかという FLOPS と呼ばれる性

能指標が一般的に用いられます。また演算能力だけではわかりづらいので、1秒間に何画素描画できるかというフィルレートと呼ばれる指標も用いられます。



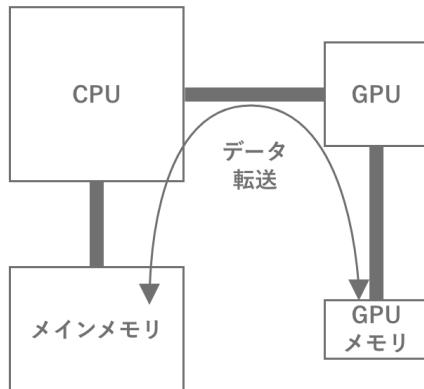
▲図 2.6 CPU と GPU の違い

GPU の演算能力

GPU のハードウェアの特徴として、整数および浮動小数点の演算ユニットを含んだコアが大量（数十～数千）に配置されているところにあります。コアを多く配置するために、CPU で必要だった複雑なプログラムを実行するのに必要なユニットは不要なので削ってあります。また CPU と同じように動作するクロック周波数が高ければ高いほど 1 秒間にたくさん演算できます。

GPU のメモリ

GPU ももちろん、データを処理するために一時的に保存するためのメモリ領域を必要とします。通常、この領域はメインメモリと異なり、GPU 専用のメモリとなります。そのため何かしらの処理を行うためには、メインメモリから GPU のメモリにデータを転送する必要があります。処理後には逆の手順でメインメモリに戻します。たとえば複数の解像度の大きいテクスチャの転送など転送量が大きい場合、転送に時間がかかり処理のボトルネックとなるため注意が必要です。



▲図 2.7 GPU のメモリ転送

ただしモバイルにおいては、GPU 専用のメモリを搭載するのではなく、メインメモリを CPU と GPU で共用するアーキテクチャが一般的です。これは GPU のメモリ容量を動的に変えることができるメリットがある一方で、転送帯域を CPU と GPU でシェアするというデメリットがあります。またこの場合においても CPU と GPU のメモリ領域との間で、データの転送は必要です。

GPGPU

CPU では得意だった大量データに対する並列演算が GPU では高速に実行できるため、近年は GPU をグラフィックス処理以外の目的にも適用する事例があり、**GPGPU** (General Purpose GPU) と呼ばれています。とくに AI などの機械学習や、ブロックチェーンなどの計算処理に適用される事例が多くあり、そのため GPU の需要が急増し、価格高騰などの影響も出ています。また Unityにおいてもコンピュートシェーダーという機能を利用することで、GPGPU を利用できます。

2.1.5 メモリ

CPU はその時の計算に必要なデータのみを持つため、基本的にすべてのデータはメインメモリに保持されます。物理容量以上のメモリを使うことはできないため、使いすぎるとメモリを確保できなくなり、プロセスが OS から強制終了させられます。一般的にこれを **OOM** (Out Of Memory) で Kill されたと呼びます。2022 年現在のスマートフォンでは 4-8GB のメモリ容量を備えた端末がメジャーですが、それでもメモリを使いすぎないように気をつける必要があります。

また前述のようにメモリが CPU と離れているため、メモリを意識した実装を行うかどうかでパフォーマンス自体も変わってきます。この節ではパフォーマンスを意識した実装が行えるように、プログラムとメモリの関係を解説します。

メモリ・ハードウェア

メインメモリが SoC の中にあったほうが物理的な距離上有利ではあるのですが、メモリは SoC には含まれていません。これは SoC の中に含まれているとメモリ搭載量を端末ごとに変えられないなどの理由があります。とは言えメインメモリが低速だとプログラムの実行速度に顕著に影響するので、比較的高速なバスで SoC とメモリを繋ぎます。このメモリとバスの規格でスマートフォンで一般的に使われているのが **LPDDR** という規格です。LPDDR もにいくつかの世代がありますが、理論上は数 Gbps 程度の転送速度です。もちろん常に理論性能を引き出せるわけではないですが、ゲーム開発ではここがボトルネックとなることはほぼないためそこまで意識する必要はありません。

メモリと OS

1 つの OS の中ではたくさんのプロセスが同時に実行されていて、主にシステムプロセスとユーザープロセスがあります。システム系は OS を動作させるための重要な役割のプロセスが多く、サービスとして常駐しそのほとんどがユーザーの意思とは関係なく動き続けます。一方でユーザー系はユーザーの意思で起動したプロセスで、OS を動作させるためには必須ではありません。

スマートフォンでのアプリの表示状態としてフォアグラウンド（最前面）とバックグラウンド（非表示）状態があり、一般的には特定のアプリをフォアグラウンドにした場合は他のアプリはバックグラウンドになります。アプリがバックグラウンドにある間も復帰処理をスムーズにするためにプロセスは一時停止状態で存在し、メモリもそのまま維持されます。ところが全体で使用しているメモリが不足してきた場合は、OS で決められた優先順位にしたがってプロセスを Kill します。この時に Kill されやすいの

が、メモリをたくさん使っているバックグラウンド状態のユーザー系アプリ（≒ゲーム）です。つまりメモリをたくさん使うゲームは、バックグラウンドに移った際にKillされる可能性が高くなり、その結果ゲームに戻ってきても起動からのやり直しとなるためユーザー体感が悪化します。

もしメモリを確保しようとした際、他に殺せるプロセスがなかった場合は自身がKillの対象となります。またiOSなどのように、物理容量の一定割合以上のメモリを1つのプロセスで使えないように制御されている場合もあります。そのためそもそもメモリを確保できる限界というものは存在します。2022年時点ではメジャーなRAMが3GBのiOS端末では、1.3~1.4GB程度が限界となりますので、ゲームを作る上ではこの辺りが上限となりやすいです。

メモリスワップ

現実にはさまざまなハードウェアの端末があり、搭載されているメモリの物理容量がとても小さいものもあります。OSはそのような端末でもなるべく多くのプロセスを動作させるために、さまざまな手法で仮想的なメモリ容量を確保しようとします。それがメモリスワップです。

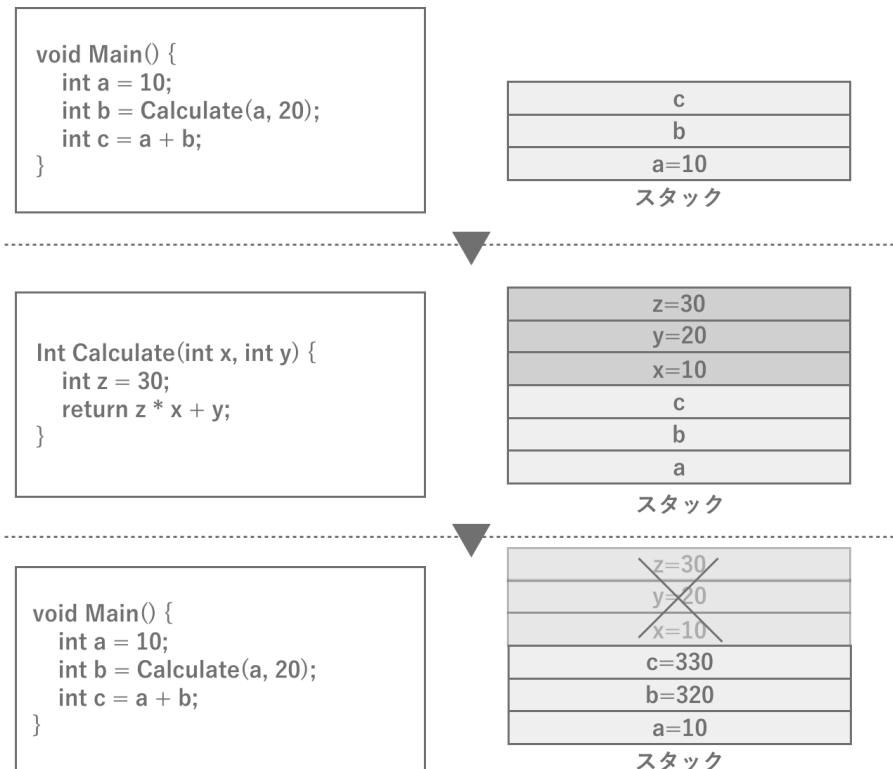
メモリスワップで使われる1つの手法がメモリの圧縮です。しばらくアクセスのないメモリを中心に、圧縮してメモリ上に保管することで物理容量を節約します。ただし圧縮と展開コストがあるため、利用が活発な領域に対して行われず、たとえばバックグラウンドに行ったアプリに対して行われます。

もう1つの手法が不使用メモリのストレージ退避です。PCのようなストレージが潤沢なハードウェアではプロセスを終了してメモリを確保するのではなく、あまり使われていないメモリをストレージに退避させることで物理メモリの空きを確保しようとする場合があります。これはメモリ圧縮より大容量のメモリを確保できるというメリットがありますが、ストレージはメモリより低速なのでパフォーマンス上の制約が強いのと、そもそもストレージのサイズが小さいスマートフォンではあまり現実的ではないため採用されていません。

スタックとヒープ

スタックとヒープという言葉を一度は聞いたことがあるかもしれません。スタックは実はプログラムの動作に深く関係する専用の固定領域になります。関数が呼び出されるタイミングで引数やローカル変数などの分が確保され、元の関数へ戻る際に確保した分を解放し、戻り値を積み上げます。つまり関数の中で次の関数を呼び出す場合、現時点の関数の情報をそのままに、次の関数をメモリに積んでいきます。このようにすることで関数呼び出しの仕組みを実現しています。スタックメモリはアーキテクチャに

依りますが1MBと容量自体がとても少ないので、限られたデータのみを格納します。



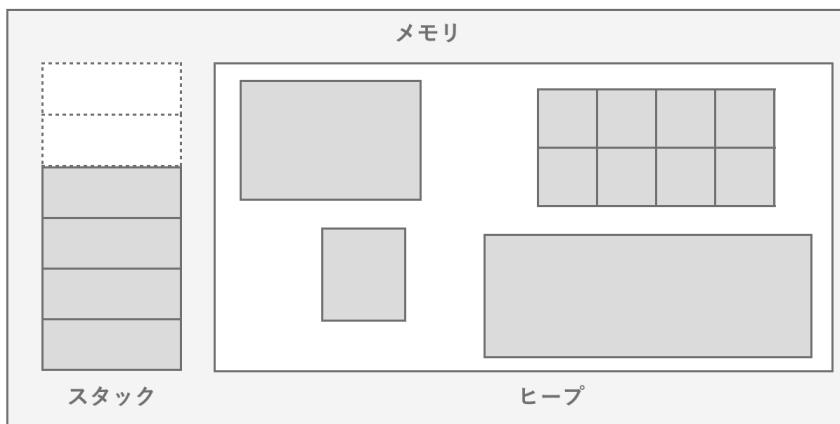
※厳密にはスタックは引数やローカル変数以外にも使われるがここでは省略

▲図 2.8 スタックの動作模式図

一方でヒープはプログラム内部で自由に使えるメモリ領域になります。プログラムが必要とすればいつでもメモリ確保命令（Cではmalloc）を出し、大容量のデータを確保して使うことができます。もちろん使い終わったら解放処理（free）が必要です。C#ではメモリ確保と解放処理が自動的にランタイム側で行われるため、実装者が明示的に行うことはありません。

OS側はいつどれくらいのメモリ容量が必要とされるかがわからないため、必要とされたタイミングでメモリの空き領域から確保して渡します。メモリ確保しようとした際に、連続してそのサイズを確保できない場合はメモリ不足となります。この連続というキーワードが重要です。一般的にメモリ確保と解放を繰り返すと、メモリの断片

化が発生します。メモリが断片化すると、全体合計での空き領域が足りていても、連続で空いている領域がない場合が考えられます。このような場合、OS がまずはヒープの拡張を実行します。つまりプロセスに割り当てるメモリを新規に割り当てることで、連続領域を確保します。ただしシステム全体での有限なメモリのため、新規に割り当てるメモリがなくなった場合は OS からプロセスを Kill されてしまいます。



▲図 2.9 スタックとヒープ

スタックとヒープを比較した際にはメモリ確保のパフォーマンスに顕著な差が生じます。それは関数に必要なスタックのメモリ量はコンパイル時点で確定するため、メモリ領域は確保済みなのに對し、ヒープは実行するまで必要なメモリ量がわからないため、都度空き領域から探して確保するからです。これがヒープが遅く、スタックは速いという所以です。

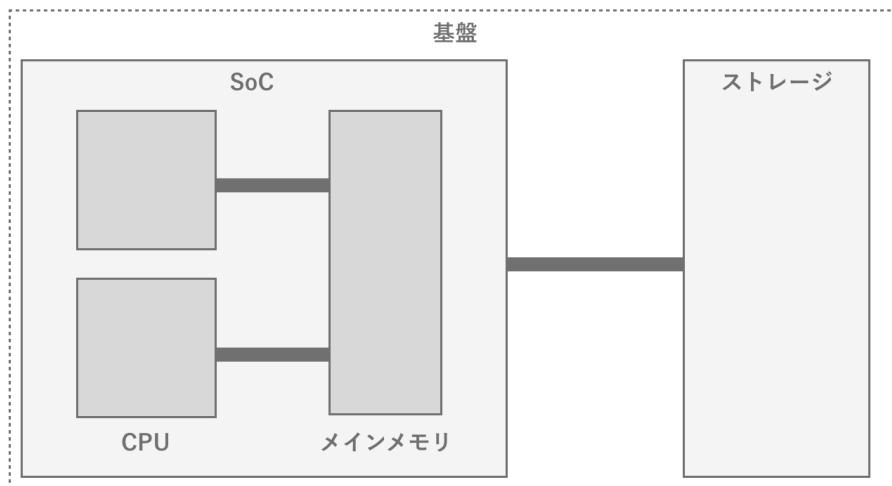
Stack Overflow エラー

Stack Overflow エラーは関数の再帰呼び出しなどでスタックメモリを使い切ったときに出てしまうエラーです。iOS/Android のデフォルトのスタックサイズは 1MB のため、再帰呼び出しによる探索規模が大きくなると発生しやすくなります。一般的には再帰呼び出しにならない、ないしは再帰呼び出しが深くならないアルゴリズムへの変更などで対策が可能です。

2.1.6 ストレージ

実際にチューニングを進めると、ファイルを読み込む場面で時間がかかることが多いことに気付くかもしれません。ファイルを読み込むということは、ファイルが保存されているストレージからデータを読み出して、プログラムから扱えるようにメモリに書き込んでいます。そこで実際に何が起きているかを知っておくとチューニングする時に便利です。

まず一般的なハードウェアアーキテクチャの場合は、データを永続化するために専用のストレージを持ちます。ストレージは大容量かつ電源なしでデータを永続化できる（不揮発）という特徴があります。この特徴を活かし、膨大なアセットはもちろんのこと、アプリ本体のプログラムなどもストレージに格納され、起動時などにストレージから読み込まれて実行されます。



▲図 2.10 SoC とストレージの関係性

RAM と ROM

とくに日本ではスマホのメモリのことを RAM、ストレージのことを ROM と書くことが主流となっていますが、実は ROM は Read Only Memory のことを指します。その名の通り読み取り専用で書き込みできないはずなのに、この用語が使われる原因是日本の慣習が強いようです。

ところがこのストレージに対する読み書きの処理は、いくつかの観点からプログラムの実行周期と比較しても遅いものとなっています。

- CPU との物理的な距離がメモリと比べて離れているため、レイテンシが大きく読み書きの速度が遅い
- 命令されたデータとその周辺を含めてブロック単位で読み込むのでムダが多い
- シーケンシャルな読み書きは速い一方で、ランダムな読み書きは遅い

とくにこのランダムな読み書きが遅いというのは重要なポイントです。そもそもどういう場面でシーケンシャルになってどういう場面でランダムになるかと言えば、1つのファイルを先頭から順番に読み書きする場合はシーケンシャルになりますが、1つのファイルの複数箇所を飛び飛びに読み書きしたり、複数の小さなファイルを読み書きする場合はランダムになります。注意したいのは、同じディレクトリにある複数のファイルを読み書きする場合でも、物理的に連続して配置されているとは限らないため、物理的に離れている場合はランダムになります。

ストレージからの読み出し処理

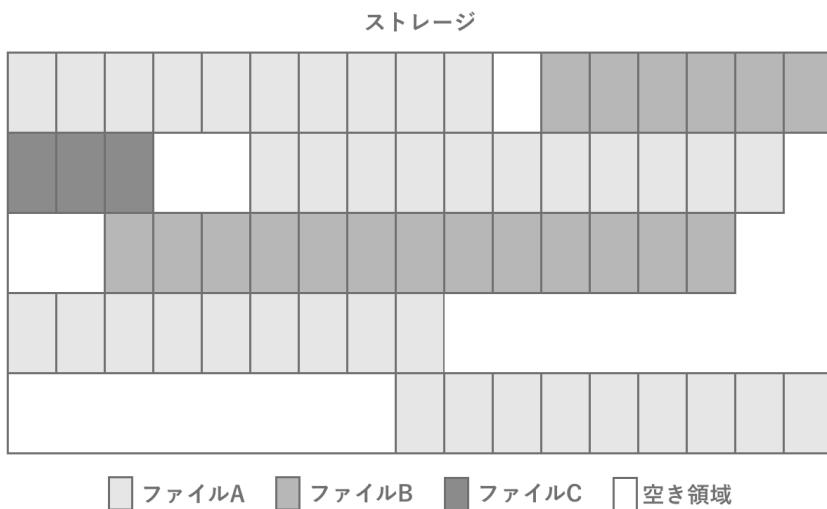
ストレージからファイルを読み出す時に、細かい部分は省略しますが、ざっくり下記の流れで処理されます。

1. プログラムがストレージから読み込みたいファイルの領域をストレージコントローラーに命令する
2. ストレージコントローラーが命令を受け取り、データのある物理上の読み込む領域を計算する
3. データを読み込む
4. データをメモリ上に書き込む

5. プログラムがメモリを通してデータにアクセスする

またハードウェアやアーキテクチャによってはコントローラなどのレイヤーが増えたりもします。正確に覚えておく必要はないですが、メモリからの読み出しと比較してハードウェアの処理工程が多いというのは意識しましょう。

また一般的なストレージは1つのファイルを4KBなどのブロック単位に書き込むことで、パフォーマンスと空間効率を達成しています。このブロックは1つのファイルだとしても物理的に連続して配置されるとは限りません。ファイルが物理的に分散している状態を断片化（フラグメンテーション）と呼び、断片化を解消する操作をデフラグと呼びます。PCで主流だったHDDでは断片化が問題となることが多かったのですが、フラッシュストレージになり影響はほぼなくなりました。スマホにおいてはファイルの断片化を意識する必要はありませんが、PCを考慮する場合は気をつける必要があります。



▲図 2.11 ストレージの断片化

PCとスマホにおけるストレージの種類

PCの世界ではHDDとSSDが主流です。HDDを見たことないという人もいるかもしれません、CDのように円盤状に記録されるメディアで、ディスクの上をヘッドが動いて磁気を読み取ります。そのため構造的にも大きく、また物理的な動きが発生するためレイテンシが大きい装置でした。近年はSSDが普及し、これはHDDと異なり物理的な動きが発生しないため高速な性能を発揮しますが、その一方で読み書き回数の限界（寿命）があるため頻繁に読み書きが発生すると使えなくなるという特徴があります。スマホはSSDとは違いますが、NANDと呼ばれるフラッシュメモリの一一種が使われています。

最後に、実際にスマホにおいてストレージがどれくらいの読み書きの速度があるかですが、2022年現在の1つの目安としては読み込みで100MB/s程度となります。仮に10MBのファイルを読み取りたい場合では、理想的な状況であってもファイル全体を読み取るために100ms必要となります。さらに複数の細かいファイルを読み込む場合はランダムアクセスが発生するので、ますます読み取りに時間がかかるようになります。このように実は意外とファイルの読み込みに時間がかかるというのは常に意識しておいた方がよいです。個別の端末の具体的な性能に関してはベンチマーク結果を集めたサイト^{*1}があるので参考にしましょう。

最後にまとめると、ファイルの読み書きが発生する場合は以下の観点を意識するといいです。

- ストレージの読み書き速度は意外と遅く、メモリと同等の速度を期待しない
- 同時に読み書きするファイルの数はできる限り減らす（タイミングを分散させる、1つのファイルに纏めるなど）

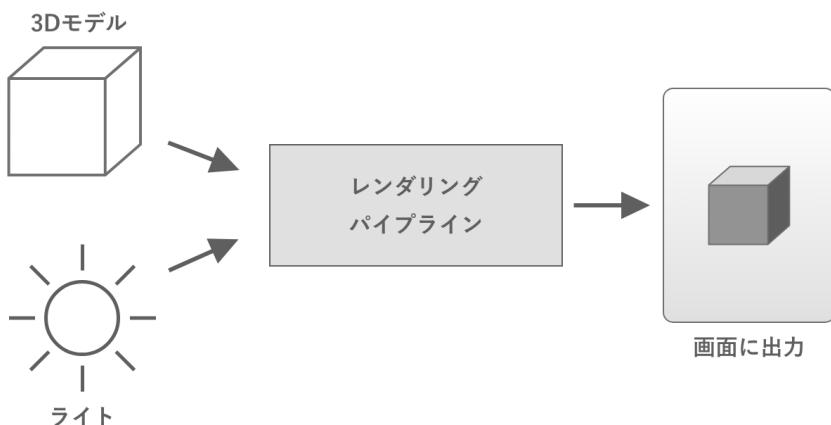
2.2 レンダリング

ゲームにおいてレンダリングの処理負荷はしばしばパフォーマンスに悪影響を及ぼします。したがって、レンダリングに関する知識はパフォーマンスチューニングを行う上で必須であるといえます。そのためこの節では、レンダリングの基礎知識についてまとめます。

^{*1} https://maxim-saplin.github.io/cpdt_results/

2.2.1 レンダリングパイプライン

コンピュータグラフィックスでは、3Dモデルの頂点座標やライトの座標と色などのデータに対して一連の処理を行なうことで、最終的に画面上の各画素に出力する色を出力します。この処理の仕組みをレンダリングパイプラインと呼びます。

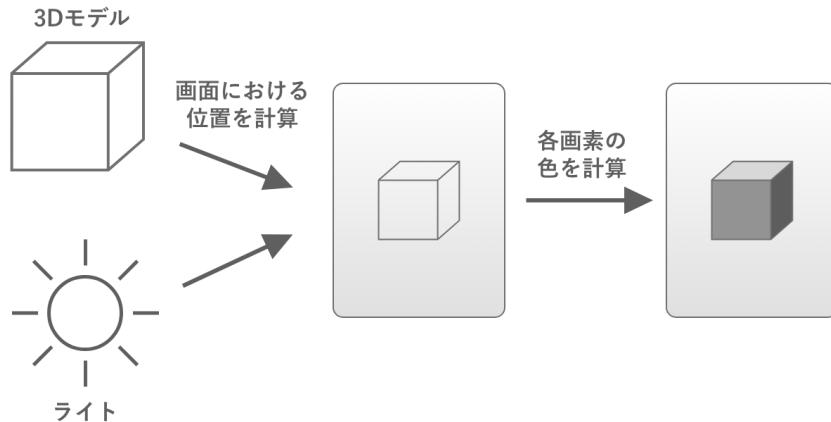


▲図 2.12 レンダリングパイプライン

レンダリングパイプラインは、CPU から GPU に必要なデータを送るところから始まります。描画すべき 3D モデルの頂点座標やライトの座標をはじめとして、オブジェクトの材質の情報やカメラの情報などさまざまなデータが送られます。

このとき送られてくるのは、3D モデルの頂点座標やカメラの座標、向き、画角などそれぞれ個別のデータです。GPU はこれらの情報をまとめて「そのカメラでそのオブジェクトを映した場合に、画面上のどの位置にオブジェクトが表示されるか」を計算して求めます。この処理を座標変換と呼びます。

オブジェクトが画面上のどの位置に表示されるかが決まつたら、次にオブジェクトの色を求める必要があります。そこで今度は GPU は「そのライトでその材質のモデルを照らしたときに、画面上の各画素に対応する部分はどのような色になるか」を計算して求めます。



▲図 2.13 位置と色を計算

上述の処理のうち、「画面上のどの位置にオブジェクトが表示されるか」は頂点シェーダーと呼ばれるプログラムにより計算され、「画面上の各画素に対応する部分はどのような色になるか」はフラグメントシェーダーと呼ばれるプログラムにより計算されます。

そしてこれらのシェーダーは自由に記述できます。したがって、頂点シェーダーやフラグメントシェーダーに重い処理を書いてしまうと処理負荷が増大します。

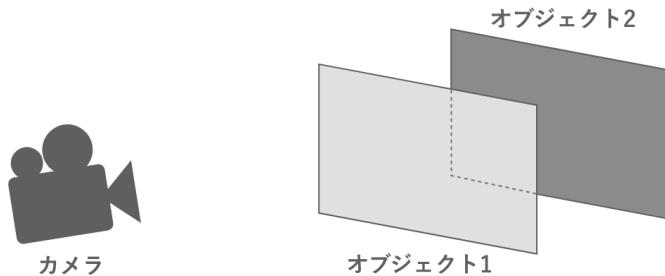
また、頂点シェーダーの処理は3Dモデルの頂点の数だけ処理されるため、頂点の数が多いほど処理負荷が大きくなります。フラグメントシェーダーはレンダリング対象の画素数が多いほど処理負荷が大きくなります。

実際のレンダリングパイプライン

実際のレンダリングパイプラインでは頂点シェーダーやフラグメントシェーダー以外にも多くのプロセスが存在しますが、本書ではパフォーマンスチューニングに必要な概念の理解を目的としているため、簡易的な説明に留めます。

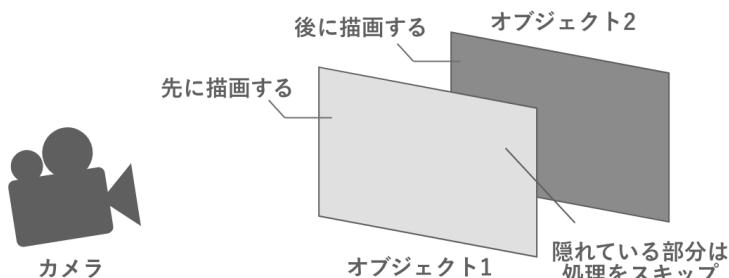
2.2.2 半透明描画とオーバードロー

レンダリングを行うにあたり、対象のオブジェクトの透明度は重要な問題です。たとえばいま、カメラから見たときに一部分が重なっている2つのオブジェクトについて考えます。



▲図 2.14 重なっている2つのオブジェクト

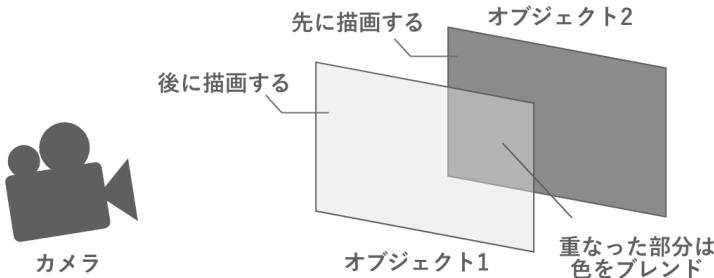
まずこれらのオブジェクトが両方とも不透明であるケースを考えます。この場合、カメラから見て手前にあるオブジェクトから順番に描画処理が行われます。こうすると、奥側のオブジェクトを描画する際に、手前のオブジェクトに重なって見えていない部分は処理する必要がありません。つまりこの部分はフラグメントシェーダーの演算をスキップできるということになり、結果として処理負荷を最適化できます。



▲図 2.15 不透明描画

一方、両方のオブジェクトが半透明だった場合には、手前のオブジェクトに重なっている部分であっても奥側のオブジェクトが透けて見えていなければ不自然です。こ

の場合には、カメラから見て奥側にあるオブジェクトから順番に描画処理を行い、重なった部分の色はすでに描画されている色とブレンドします。



▲図 2.16 半透明描画

このように、半透明描画は不透明描画と異なり、オブジェクト同士が重なっている部分についても描画処理を行う必要があります。もし画面いっぱいに描画される半透明なオブジェクトが2つ存在していたら、画面いっぱい分の処理が2回行われるということになります。このように、半透明なオブジェクトを重ねて描画することをオーバードローと呼びます。オーバードローが多くなるとGPUに大きな処理負荷がかかり、パフォーマンスの低下に繋がるため、半透明描画を行う際には適切にレギュレーションを設ける必要があります。

フォワードレンダリングを想定

レンダリングパイプラインにはいくつかの実装方法があります。そのうち、本項の記述はフォワードレンダリングを想定しています。デファードレンダリングなど他のレンダリング手法には部分的に当てはまらない点もあります。

2.2.3 ドローコール・セットパスコールとバッチング

レンダリングの際にはGPUだけではなくCPUにも処理負荷がかかります。

上述の通り、オブジェクトをレンダリングする際にはCPUからGPUに描画するための命令を出します。これはドローコールと呼ばれ、レンダリングするオブジェクトの数だけ実行されます。またこのときに、テクスチャなどの情報が前回のドローコー

ルで描画したオブジェクトのものと異なっている場合には、それらを GPU に設定する処理を行います。これはセットパスコールと呼ばれ、比較的重い処理になります。この処理は CPU のレンダースレッドで行われるため、CPU に処理負荷がかかり、多すぎるとパフォーマンスに影響を及ぼします。

Unity には、ドローコールを削減するためにドローコールバッティングと呼ばれる仕組みが実装されています。これは同じテクスチャなどの情報、つまり同じマテリアルを持つオブジェクトのメッシュをあらかじめ CPU 側の処理で結合してしまい、1回のドローコールで描画する仕組みです。ランタイムでバッティングするダイナミックバッティングと、あらかじめ結合したメッシュを作成しておくスタティックバッティングがあります。

また、**Scriptable Render Pipeline** には **SRP Batcher** という仕組みが実装されています。これを使うと、シェーダーバリエントが同一であれば、メッシュやマテリアルが違っていたとしてもセットパスコールを 1 回にまとめることができます。ドローコールは減りませんが、大きな処理負荷がかかるのはセットパスコールであるため、こちらを減らすための仕組みです。

これらのバッティングについてのより詳細な情報は「7.3 ドローコールの削減」を参照してください。

GPU インスタンシング

バッティングに似た効果を得られる機能として、**GPU インスタンシング**があります。これは GPU の機能を使うことで、同じメッシュを持つオブジェクトを一度のドローコール・セットパスコールで描画できる機能です。

2.3 データの表現方法

ゲームには画像や 3D モデル、音声、アニメーションなどさまざまなデータが使われます。これらがデジタルデータとしてどのように表現されているかを知ることは、メモリやストレージの容量を計算したり、圧縮などの設定を適切に行ったりする上で重要です。この節では基本的なデータの表現方法についてまとめます。

2.3.1 ビットとバイト

コンピューターが表現できる最小の単位はビットです。1ビットでは2進数の1桁で表せる範囲、つまり0か1の2通りの組み合わせを表現できます。これではたとえばスイッチのON・OFFなどといった簡単な情報しか表せません。



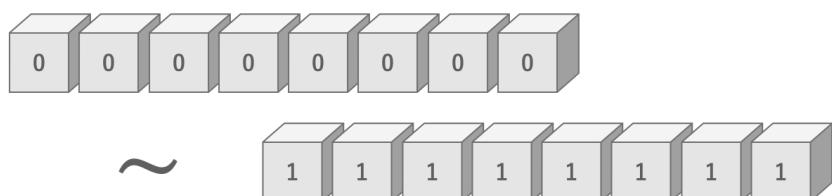
▲図 2.17 1ビットの情報量

ここでビットを2つ使うと、2進数の2桁で表せる範囲、つまり4通りの組み合わせを表現できることができます。4通りなのでたとえば上・下・左・右などのキーが押されたかといった情報を表せそうです。



▲図 2.18 2ビットの情報量

同様に8ビットになると2進数の8桁で表せる範囲、つまり $2\text{通り}^8\text{桁} = 256\text{通り}$ です。ここまでくると色々な情報が表現できそうです。そしてこの8ビットは1バイトという単位で表されます。つまり1バイトとは256通りの情報量を表せる単位であることができます。



▲図 2.19 8ビットの情報量

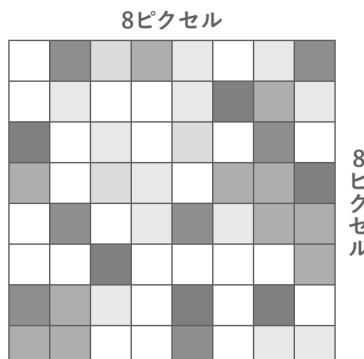
また、さらに大きな数を表す単位として、1000 バイトを表す 1 キロバイト (KB) や、1000 キロバイトを表す 1 メガバイト (MB) が存在します。

キロバイトとキビバイト

上記では 1KB を 1,000 バイトと書きましたが、文脈によっては 1KB を 1,024 バイトとする場合もあります。明示的に呼び分ける場合には、1000 バイトを 1 キロバイト (KB) と呼び、1,024 バイトを 1 キビバイト (KiB) と呼びます。メガバイトについても同様です。

2.3.2 画像

画像データはピクセルの集合として表されています。たとえば 8×8 ピクセルの画像であれば、合計 $8 \times 8 = 64$ 個のピクセルで構成されています。

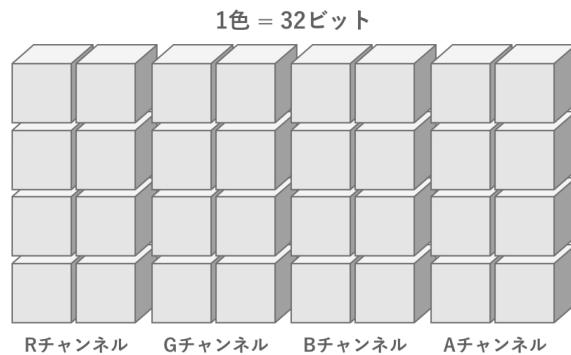


▲図 2.20 画像データ

このとき、各ピクセルはそれぞれ色のデータを持っています。では色はデジタルデータでどのように表現されるのでしょうか。

まず色は赤 (Red)、緑 (Green)、青 (Blue)、透明度 (Alpha) の 4 つの要素を組み合わせて作られます。これらをチャンネルと呼び、それぞれのチャンネルの頭文字をとつて RGBA と表現します。

よく使われる True Color という色の表現方法では、RGBA の各値をそれぞれ 256 段階で表します。前節で説明した通り、256 段階とはつまり 8 ビットです。すなわち True Color は 4 チャンネル × 8 ビット = 32 ビットの情報量で表すことができます。



▲図 2.21 1 色の情報量

したがって、たとえば 8×8 ピクセルの True Color の画像であればその情報量は $8 \text{ ピクセル} \times 8 \text{ ピクセル} \times 4 \text{ チャンネル} \times 8 \text{ ビット} = 2,048 \text{ ビット} = 256 \text{ バイト}$ となります。 $1,024 \times 1,024$ ピクセルの True Color の画像であれば、その情報量は $1,024 \text{ ピクセル} \times 1,024 \text{ ピクセル} \times 4 \text{ チャンネル} \times 8 \text{ ビット} = 33,554,432 \text{ ビット} = 4,194,304 \text{ バイト} = 4,096 \text{ キロバイト} = 4 \text{ メガバイト}$ となります。

2.3.3 画像の圧縮

実際には、画像は圧縮されたデータとして使用されることがほとんどです。圧縮とは、データの格納方法を工夫することでデータ量を減らすことです。たとえばいま、同じ色をしたピクセルが 5 つ隣り合っていたとします。この場合、各ピクセルの色情報を 5 つ持つよりも、色の情報ひとつと、それが 5 個並んでいるという情報を持った方が情報量は減ります。



▲図 2.22 圧縮

実際にはもっと複雑な圧縮方法がたくさん存在します。

具体例として、モバイルで代表的な圧縮フォーマットである ASTC を紹介します。ASTC6x6 というフォーマットを適用すると、1024x1024 のテクスチャが 4 メガバイトから約 0.46 メガバイトに圧縮されます。つまり、容量は 8 分の 1 以下に圧縮されたという結果となり、圧縮を行うことの重要性を認識できます。

参考までに、モバイルで主に利用される ASTC フォーマットの圧縮率について以下に記載します。

▼表 2.2 圧縮形式と圧縮率

圧縮形式	圧縮率
ASTC RGB(A) 4x4	0.25
ASTC RGB(A) 6x6	0.1113
ASTC RGB(A) 8x8	0.0625
ASTC RGB(A) 10x10	0.04
ASTC RGB(A) 12x12	0.0278

なお Unity では、テクスチャのインポート設定によりさまざまな圧縮方法を、プラットフォームごとに指定できます。そのため非圧縮の画像をインポートし、このインポート設定により圧縮をかけることで最終的に使用されるテクスチャを生成するというやり方が一般的となっています。

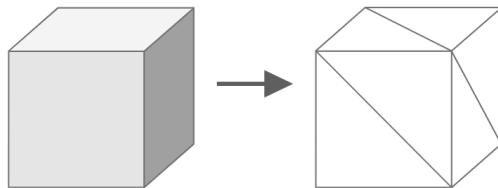
GPU と圧縮形式

あるルールに基づいて圧縮した画像は、当然ですがそのルールに基づいて展開する必要があります。この展開処理はランタイムで行われます。この処理負荷を最小限に抑えるため、GPU が対応した圧縮形式を使うことが重要です。モバイルデバイスの GPU が対応している代表的な圧縮形式として ASTC が挙げられます。

2.3.4 メッシュ

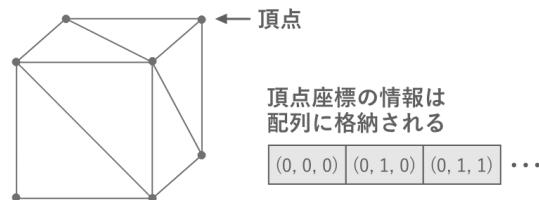
3DCG では、3D 空間上に三角形を多数繋ぎ合わせることで立体形状を表現しています。この三角形の集まりをメッシュと呼びます。

立体は三角形で構成される



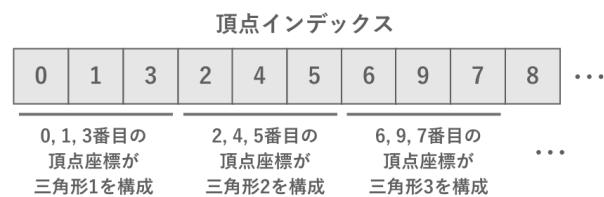
▲図 2.23 三角形の組み合わせによる立体

この三角形は、データとしては 3D 空間上の 3 点の座標情報として表すことができます。この各点を頂点と呼び、その座標を頂点座標と呼びます。またメッシュ 1 つあたりの頂点情報はすべて 1 つの配列に格納されます。



▲図 2.24 頂点情報

頂点情報は 1 つの配列に格納されるため、そのうちのどれを組み合わせて三角形を構成するかを表す情報が別途必要です。これを頂点インデックスと呼び、頂点情報の配列のインデックスを表す int 型の配列として表現されます。



▲図 2.25 頂点インデックス

オブジェクトにテクスチャを貼り付けたり、ライティングを行なったりする上ではさらに追加の情報が必要です。たとえばテクスチャをマッピングするには UV 座標が必要です。またライティングをする上では、頂点カラーや法線、接線などの情報も使われます。

次の表は、主な頂点情報と 1 頂点あたりの情報量をまとめたものです。

▼表 2.3 頂点情報

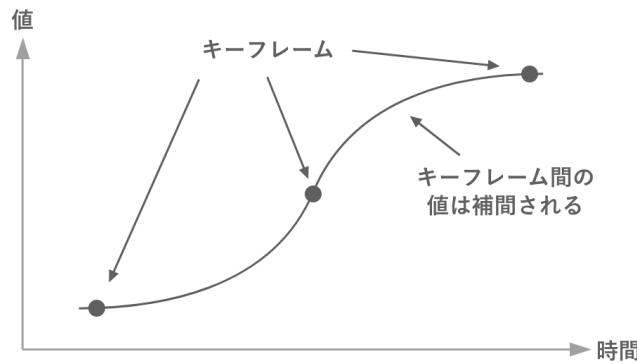
名前	1 頂点あたりの情報量
頂点座標	3 次元の float = 12 バイト
UV 座標	2 次元の float = 8 バイト
頂点カラー	4 次元の float = 16 バイト
法線	3 次元の float = 12 バイト
接線	3 次元の float = 12 バイト

メッシュのデータは頂点の数や 1 つの頂点で扱う情報の量が増えるほど大きくなるため、頂点数や頂点情報の種類を事前に決めておくことは重要です。

2.3.5 キーフレームアニメーション

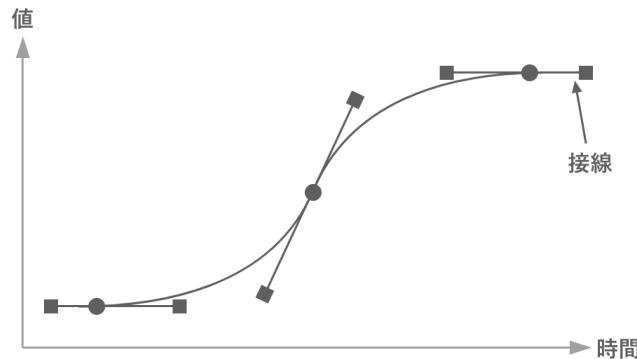
ゲームでは UI のアニメーションや 3D モデルのモーションなど、多くの箇所にアニメーションを使用します。アニメーションの代表的な実現手法として、キーフレームアニメーションがあります。

キーフレームアニメーションは、ある時間（キーフレーム）における値を表すデータの配列で構成されます。キーフレーム間の値は補間により求められるので、あたかも滑らかに連続したデータであるかのように取り扱うことができます。



▲図 2.26 キーフレーム

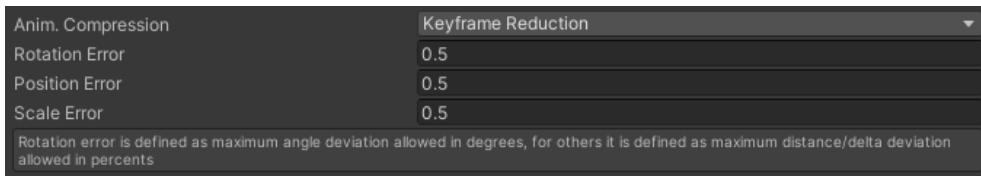
なおキーフレームが持つ情報は時間と値の他に、接線やその重みといったものがあります。これらを補間の計算に利用することで、少ないデータ量でより複雑なアニメーションを実現することができます。



▲図 2.27 接線と重み

キーフレームアニメーションにおいてはキーフレームが多ければ多いほど複雑なアニメーションを表現できます。しかしながら、データ量もキーフレームの数に応じて増大します。このような理由から、キーフレームの数は適切に設定する必要があります。

できるだけ同じようなカーブを保ちつつキーフレームを削減してデータ量を圧縮する手法もあります。Unity の場合、モデルのインポート設定で次図のようにキーフレームを削減できます。



▲図 2.28 インポート設定

設定方法の詳細は「4.4 Animation」を参照してください。

2.4 Unity の仕組み

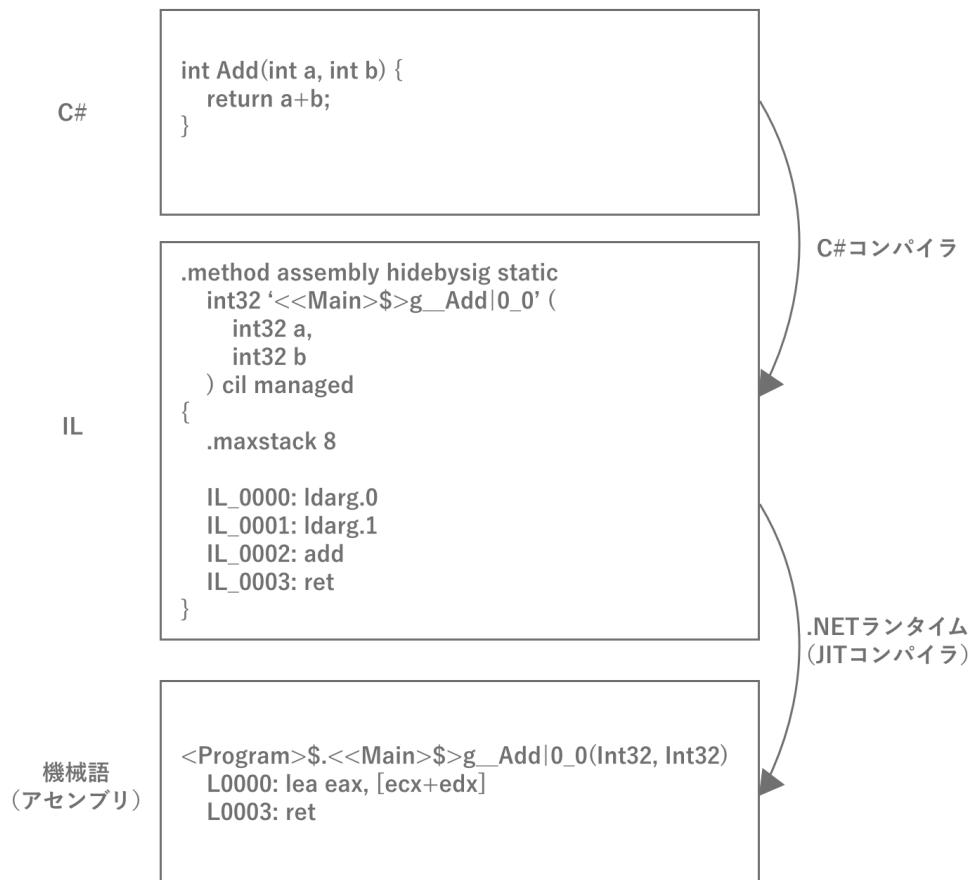
Unity エンジンが実際にどういう仕組みで動いているかを理解することは、ゲームをチューニングする上で重要であることはいうまでもありません。この節で知っておくべき Unity の動作原理を説明します。

2.4.1 バイナリとランタイム

まずここでは Unity が実際にどういう仕組みでランタイムを動かしているかを解説します。

C#とランタイム

Unity でゲームを作る際、開発者は C# で挙動をプログラミングします。Unity でゲームを開発する際、度々コンパイル（ビルド）が実行されるように、C# はコンパイラ型言語です。ところが C# が伝統的な C 言語などと異なるのは、コンパイルするとマシンで単体実行可能な機械語ではなく、.NET における中間言語（Intermediate Language；以降 IL）にコンパイルされることです。IL に変換された実行コードは単体では実行できないので、.NET Framework のランタイムを用いて逐次機械語に変換しながら実行されます。



▲図 2.29 C#のコンパイル過程

一度 IL を挿むのは、機械語に変換してしまうと単一のプラットフォームでしか実行できないバイナリとなってしまうためです。IL であれば、どのようなプラットフォームでもそのプラットフォームに対応したランタイムを用意するだけで動作するようになります。プラットフォーム毎にバイナリを用意する必要がなくなります。そのため Unity の基本原理としては、ソースコードをコンパイルして得られた IL をそのままそれぞれの環境向けのランタイムで実行することで、マルチプラットフォームを実現しています。

IL コードを確認してみよう

普段は目にすることが少ない IL コードは、メモリ確保や実行速度などのパフォーマンスを意識する上で非常に重要です。たとえば配列と List では、一見同じ foreach ループでも異なる IL コードが output され、配列の方がパフォーマンスに優れているコードとなります。また意図しない隠れたヒープアロケーションも発見できるかもしれません。こういった C# と IL コードの対応感覚を身につけるために、普段から自分の書いた C# コードの IL 変換結果を確認しておくことはオススメです。Visual Studio や Rider といった IDE で IL コードを閲覧できますが、IL コード自体はアセンブリと呼ばれる低級言語のため理解するのが難しい言語です。そのような場合には SharpLab^{*2} という Web サービスを利用すると C# -> IL -> C# と IL から逆変換したコードを確認することで理解しやすくなります。本書の後半の第 10 章「Tuning Practice - Script (C#)」にて、実際の変換例を紹介します。

IL2CPP

前述のように Unity では基本的には C# を IL コードにコンパイルしてランタイムで実行しますが、2015 年頃から一部の環境で問題が生じるようになりました。それは iOS や Android で動作するアプリの 64bit 対応です。C# は IL コードを実行するためにそれぞれの環境で動作するためのランタイムが必要になるのは前述の通りですが、実はそれまでの Unity は長年 .NET Framework の OSS 実装である Mono をフォークして Unity 自ら改変して利用していました。つまり Unity が 64bit 対応するためには、フォークした Mono を 64bit 対応させる必要がありました。もちろんそれはとてつもない労力が必要となるため、Unity はここで代わりに **IL2CPP** と呼ばれる技術を開発することでこの難題を乗り切りました。

IL2CPP とは名前の通り IL to CPP のことであり、IL コードを C++ コードに変換する技術です。C++ はどのような開発環境でもネイティブサポートされるような汎用性の高い言語であるため、C++ コードに出力してしまえばそれぞれの開発ツールチェインにて機械語にコンパイルすることができます。したがって 64bit 対応はツールチェインの仕事となるため、Unity 側はその対応をする必要がなくなります。また C# と違つてビルド時点での機械語にコンパイルされるため、ランタイムにて機械語に変換する必

^{*2} <https://sharplab.io/>

要がなくなり、パフォーマンスが向上するという恩恵もあります。

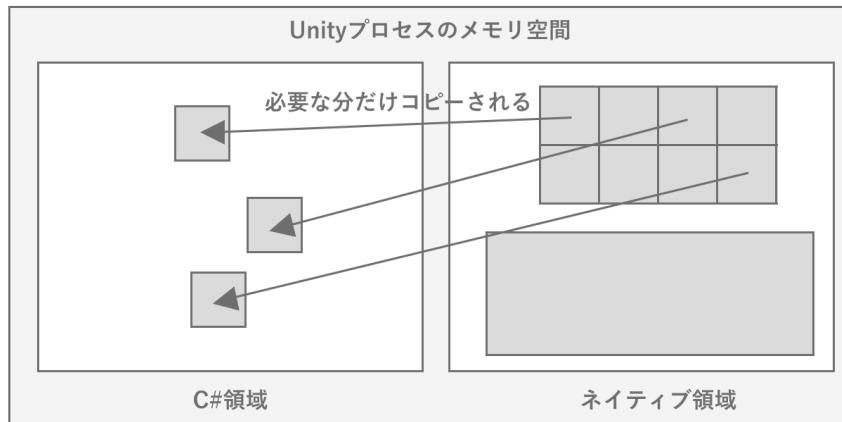
C++ コードは一般的にビルドに時間を要するという欠点はありますが、64bit 対応とパフォーマンスを一挙に解決する IL2CPP という技術は Unity の要となりました。

Unity ランタイム

ところで Unity では開発者は C#でゲームをプログラミングしますが、エンジンと呼ばれる Unity 自体のランタイムは実は C#で動いているわけではありません。ソース自体は C++ で記述され、プレイヤーと呼ばれる部分は各環境で実行するために事前にビルドされた状態で配布されます。Unity がエンジンを C++ で記述するのは、いくつかの理由が考えられます。

- 高速かつ省メモリのパフォーマンスを得るため
- なるべく多くのプラットフォームに対応するため
- エンジンの知的財産権の保護のため（ブラックボックス化）

開発者が記述した C#コードはあくまで C#で動作するため、Unity ではネイティブで動作するエンジン部分と、C#ランタイムで動作するユーザーコード部分に 2つの領域が必要となります。エンジンとユーザーコードは、実行中に適宜データをやり取りすることで動作しています。たとえば `GameObject.transform` を C#から呼び出した場合、シーンの状態などゲームの実行状態はすべてエンジン内部で管理されているため、まずネイティブ呼び出しを行ってネイティブ領域のメモリデータにアクセスし、C#に値を返すという手順を踏んでいます。ここで注意したいのは、C#とネイティブではメモリは共有されないため、C#で必要となったデータは都度 C#側でメモリが確保されることです。また API呼び出しもネイティブ呼び出しが発生するなど高価なものになるため、頻繁に呼び出さずに値をキャッシュするという最適化手法が必要となります。



▲図 2.30 Unity におけるメモリの状態イメージ

このように、Unity を開発する上では見えないエンジン部分もある程度意識する必要があります。そのため適宜 Unity エンジンのネイティブ領域と C#を繋ぐインターフェイスのソースコードを見るとよいでしょう。幸いにも Unity 社が C#の部分であれば GitHub で公開³しているため、ほとんどネイティブ呼び出しになっていることがわかるなど非常に役立ちます。必要に応じて活用することをオススメします。

2.4.2 アセットの実体

前節で説明したように、Unity エンジンはネイティブで実行されているため、基本的には C#側ではデータを持ちません。アセットの取り扱いに関しても同様で、ネイティブ領域でアセットをロードし、C#に参照を返したり、データをコピーして返していくだけです。そのためアセットをロードする際は、大別すると、Unity エンジン側でロードさせるためにパスを指定する方法と、バイト配列など生データを直接渡す方法の 2 種類があります。パスを指定した場合はネイティブ領域でロードするため C#側でメモリを消費することはありませんが、バイト配列などデータを C#側からロード・加工して渡した場合は C#側とネイティブ側で二重にメモリを消費してしまいます。

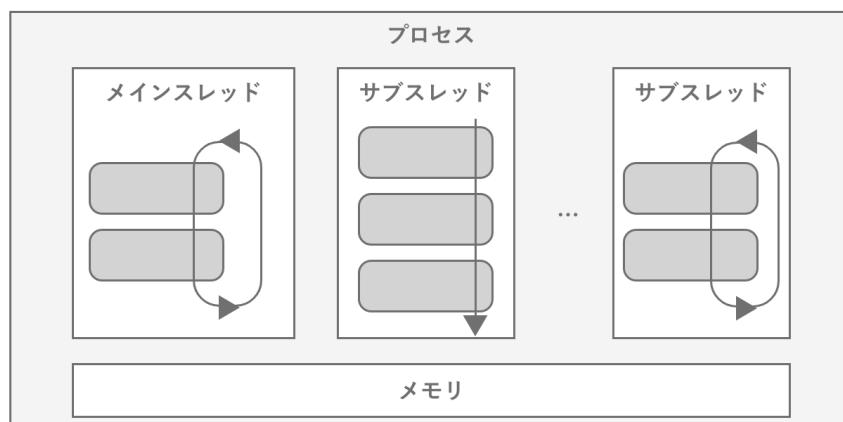
またアセットの実体がネイティブ側にあるため、アセットの多重ロードやリーアクに関する調査の難易度も上がります。これは開発者は主に C#側のプロファイリングやデバッグを中心に行うためです。C#側の実行状態だけ見ても理解することは難しく、工

³ <https://github.com/Unity-Technologies/UnityCsReference>

エンジン側の実行状態と突き合わせながら解析する必要がありますが、ネイティブ領域のプロファイリングはUnityが提供するAPIに依存するためツールが限られるという問題があります。本書でさまざまなツールを駆使して分析する手法を紹介しますが、その際にC#とネイティブの空間を意識すると理解しやすくなります。

2.4.3 スレッド

スレッドはプログラムの実行単位で、一般的には1つのプロセスの中に複数のスレッドを生成しながら処理が進みます。CPUの1つのコアは同時に1つのスレッドしか処理することができないため、複数のスレッドを処理するために高速にスレッドを切り替えながらプログラムを実行します。これをコンテキストスイッチと呼びます。コンテキストスイッチする際はオーバーヘッドが生じるため、頻繁に発生すると処理効率が低下してしまいます。



▲図2.31 スレッドの模式図

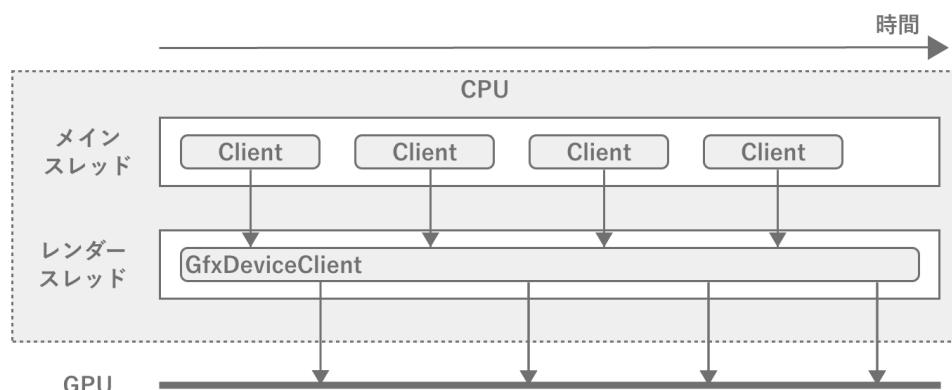
プログラムの実行時には基底となるメインスレッドが生成され、そこからプログラムが必要に応じて別のスレッドを生成・管理します。Unityのゲームループはシングルスレッドで動作する設計となっているため、ユーザーが記述したスクリプトは基本的にはメインスレッド上で動作することになります。逆にメインスレッド以外からUnity APIを呼び出そうとすると、ほとんどのAPIはエラーが発生してしまいます。

メインスレッドから別のスレッドを作成して処理を実行する場合、そのスレッドがいつ実行されて、いつ完了するかはわかりません。そのためスレッド間で処理を同期

させる手段としてシグナルと呼ばれる機構があります。別スレッドの処理を待機する場合、そのスレッドからシグナルを通知してもらうことで待機を解除できます。このシグナル待機は Unity 内部でも使われているためプロファイリング時などに観測できますが、`WaitFor~`という名前の通りただ別の処理を待機しているだけということは注意しましょう。

Unity 内部のスレッド

とはいあらゆる処理をメインスレッドで実行していると、プログラム全体の処理に時間がかかるようになってしまいます。複数の重い処理があり、それが相互に依存がなかった場合、ある程度処理を同期することで並列処理を行うことができれば、プログラムの実行を短縮することが可能となります。こうした高速化のために、ゲームエンジン内部では並列処理が多数用いられます。その1つがレンダースレッド（Render Thread）です。名前の通りレンダリング専用のスレッドで、メインスレッドで計算したフレームの描画情報を、グラフィックスコマンドとして GPU に送る役割を担います。



▲図 2.32 メインスレッドとレンダースレッド

メインスレッドとレンダースレッドはパイプラインのように実行されるため、レンダースレッドが処理中に次のフレームの計算が始まります。ところがもしレンダースレッド内で1フレームを処理する時間が長くなってくると、次のフレームの描画の計算が終わったとしても描画を開始することができなくなり、メインスレッドは待たされることになります。ゲーム開発ではメインスレッド、レンダースレッドどちらが重くとっても FPS が低下してしまうため注意しましょう。

並列処理可能なユーザー処理のスレッド化

またゲーム特有の部分として、物理エンジンや揺れものなど並列処理を実行できる計算タスクが多数存在します。そのような計算をメインスレッド以外で実行させるために、Unity ではワーカースレッド（Worker Thread）が存在します。ワーカースレッドは JobSystem を通して生成された計算タスクを実行します。JobSystem を利用することでメインスレッドの処理負荷を軽減できる場合は積極的に利用しましょう。もちろん JobSystem を利用せずに、自前でスレッドを生成する方法もあります。

スレッドはパフォーマンスチューニングで便利な半面、使いすぎると逆にパフォーマンスが低下したり、処理の複雑性が向上する危険性もあるため、閻雲に使わないことをオススメします。

2.4.4 ゲームループ

Unity を含む一般的なゲームエンジンは、ゲームループ（プレイヤーループ）と呼ばれる、エンジンのルーチン処理があります。簡潔にループを表現するのならば、概ね以下のようになります。

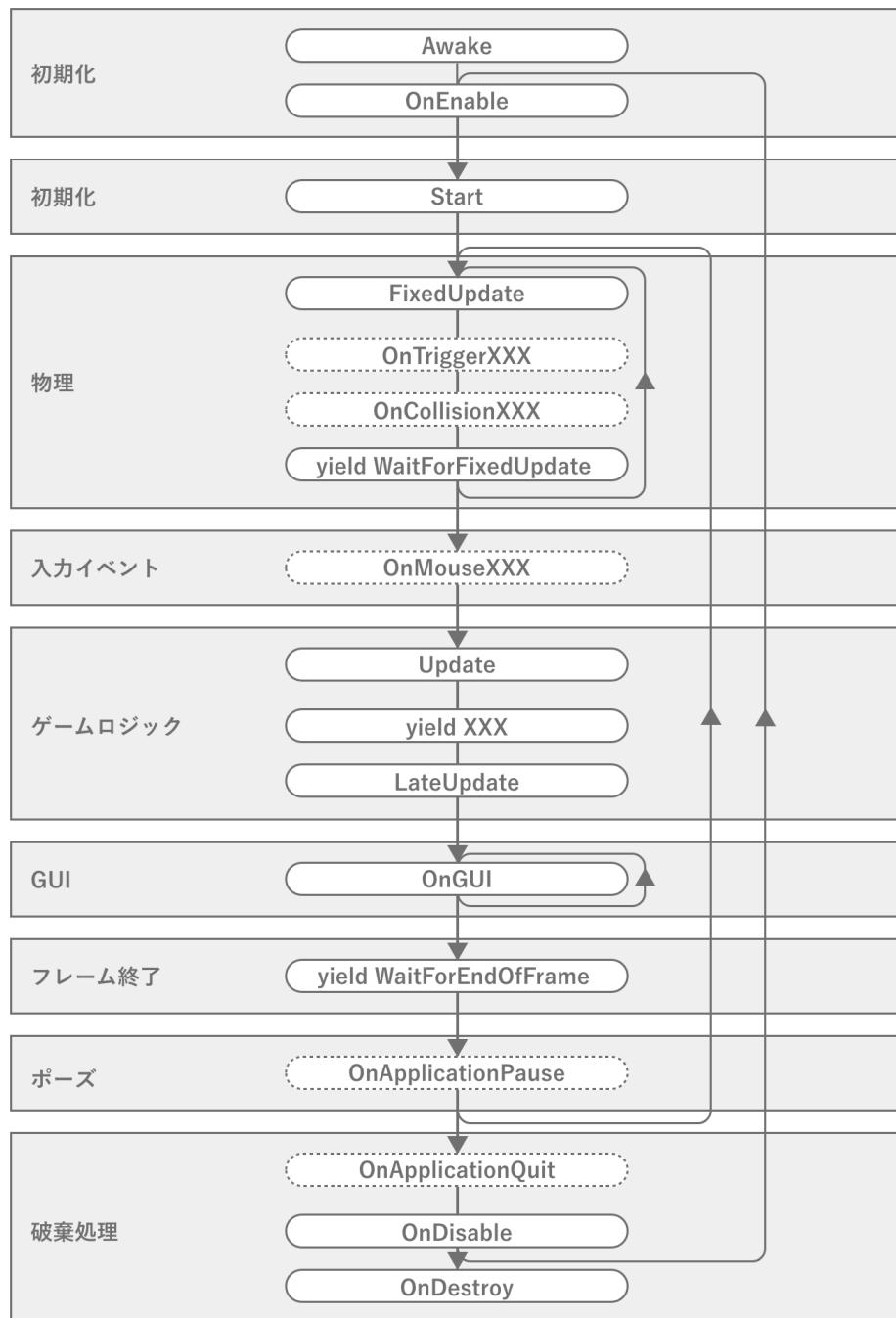
1. キーボード、マウス、タッチディスプレイなどのコントローラーの入力処理
2. 1 フレームの時間で進行すべきゲームの状態を計算
3. 新しいゲームの状態をレンダリング
4. ターゲット FPS に応じて、次のフレームまで待機

このループを繰り返すことでゲームを映像として GPU に出力します。もし 1 フレーム内の処理に時間がかかるようになると、もちろん FPS が低下することになります。

Unity におけるゲームループ

Unity におけるゲームループは、皆さん一度は見たことがある Unity の公式リファレンスにゲームループの模式図^{*4}が存在します。

^{*4} <https://docs.unity3d.com/ja/current/Manual/ExecutionOrder.html>



▲図 2.33 Unity のイベントの実行順序

この図は厳密には MonoBehaviour のイベントの実行順を表したもので、ゲームエンジンとしてのゲームループ⁵とは異なりますが、開発者が知っておくべきゲームループとしてはこれで十分です。とくに重要なイベントとして、Awake, OnEnable, Start, FixedUpdate, Update, LateUpdate, OnDisable, OnDestroy と各種コルーチンの処理タイミングです。イベントの実行順やタイミングを勘違いしてしまうと、思わぬメモリリークや余計な計算につながってしまう可能性があります。そのため重要なイベントの呼び出しタイミングや、同イベント内での実行順序などの性質は把握しておくべきでしょう。

物理演算に関しては、通常のゲームループと同じ間隔で実行していると衝突判定されずにオブジェクトがすり抜けてしまうなど特有の問題があります。そのため通常は物理演算ルーチンのループを高頻度に回すように、ゲームループとは異なる間隔でループを回します。ただ閻雲に回すとメインのゲームループの更新処理と競合する可能性があるため、ある程度は処理を同期させる必要があります。そのため物理演算が必要以上に重くなるとフレームの描画処理に影響したり、またフレームの描画処理が重くなると物理演算が遅れてすり抜けが発生したりと、互いに影響する可能性があるため注意しましょう。

2.4.5 GameObject

前述のように、Unity のエンジン自体はネイティブで動作しているため、C#の Unity API もその大部分は内部のネイティブ API を呼び出すためのインターフェイスです。それは GameObject やそれにアタッチするコンポーネントを定義する MonoBehaviour も同様で、常に C#側からネイティブの参照を持ち続けることになります。ところがネイティブ側でデータを管理しつつ、C#側でもそれらの参照を持っている場合、破棄のタイミングで不都合が発生します。それはネイティブ側で破棄されたデータに対して、C#からの参照を勝手に消すことができないからです。

実際にリスト 2.1 で破棄した GameObject が null かどうかチェックしていますが、ログには true が出力されます。これは標準の C#の挙動としては不自然で、_gameObject には null を代入していないため GameObject 型のインスタンスの参照が残っているはずです。

⁵ <https://tsubakit1.hateblo.jp/entry/2018/04/17/233000>

▼リスト 2.1 破棄後の参照テスト

```
public class DestroyTest : UnityEngine.MonoBehaviour
{
    private UnityEngine.GameObject _gameObject;

    private void Start()
    {
        _gameObject = new UnityEngine.GameObject("test");
        StartCoroutine(DelayedDestroy());
    }

    System.Collections.IEnumerator DelayedDestroy()
    {
        // cache WaitForSeconds to reuse
        var waitOneSecond = new UnityEngine.WaitForSeconds(1f);
        yield return waitOneSecond;

        Destroy(_gameObject);
        yield return waitOneSecond;

        // _gameObject is not null, but result is true
        UnityEngine.Debug.Log(_gameObject == null);
    }
}
```

これは Unity の C# 側の仕組みで、破棄済みデータへのアクセスの制御を行っているからです。実際に Unity の C# 実装部の `UnityEngine.Object` のソースコード^{*6}を参照すると、以下のようになっています。

▼リスト 2.2 `UnityEngine.Object` の`==`オペレーターの実装

```
// 抜粋
public static bool operator==(Object x, Object y) {
    return CompareBaseObjects(x, y);
}

static bool CompareBaseObjects(UnityEngine.Object lhs,
    UnityEngine.Object rhs)
{
    bool lhsNull = ((object)lhs) == null;
    bool rhsNull = ((object)rhs) == null;

    if (rhsNull && lhsNull) return true;

    if (rhsNull) return !IsNativeObjectAlive(lhs);
    if (lhsNull) return !IsNativeObjectAlive(rhs);
```

^{*6} <https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Runtime/Export/Scripting/UnityEngineObject.bindings.cs>

```
        return lhs.m_InstanceID == rhs.m_InstanceID;
    }

    static bool IsNativeObjectAlive(UnityEngine.Object o)
    {
        if (o.GetCachedPtr() != IntPtr.Zero)
            return true;

        if (o is MonoBehaviour || o is ScriptableObject)
            return false;

        return DoesObjectWithInstanceIdExist(o.GetInstanceID());
    }
```

要約すると、null 比較をしたときはネイティブ側のデータが存在するかどうかをチェックしているために、破棄されたインスタンスへの null 比較が true になります。そのために null でない GameObject のインスタンスが一部 null のように振る舞います。この特性は一見すると便利なのですが、非常に厄介な側面もあります。それは_gameObject は実際には null ではないので、メモリリークを引き起こすからです。_gameObject1 個分のメモリリークは当然ですが、たとえばそのコンポーネントの中からマスターなどの巨大なデータへの参照を持っている場合、C#としては参照が残るため、ガベージコレクションの対象とはならないので巨大なメモリリークに繋がってしまいます。これを回避するためには、_gameObject に null を代入するなどの対策が必要となります。

2.4.6 AssetBundle

スマホ向けゲームはアプリのサイズに制限があり、すべてのアセットをアプリに含めることができません。そのため必要に応じてアセットをダウンロードするために、Unity には AssetBundle という複数のアセットをパッキングして、動的にロードする仕組みがあります。一見簡単に扱えるように感じるかもしれません、大規模プロジェクトの場合は適切に設計しないと思わぬところでメモリをムダに使ってしまうなど、メモリや AssetBundle に対する十分な理解と丁寧な設計が求められます。そのためこの節では AssetBundle についてチューニングの観点で知っておくべきことを説明します。

AssetBundle の圧縮設定

AssetBundle はビルト時にデフォルトで LZMA 圧縮されます。これを BuildAssetBundleOptions の UncompressedAssetBundle に変えることで無圧縮に、ChunkBased Compression に変えることで LZ4 圧縮に変更することができます。これらの設定の差は以下の表 2.4 のような傾向があります。

▼表 2.4 AssetBundle の圧縮設定による違い

項目	無圧縮	LZMA	LZ4
ファイルサイズ	特大	特小	小
ロード時間	速い	遅い	かなり速い

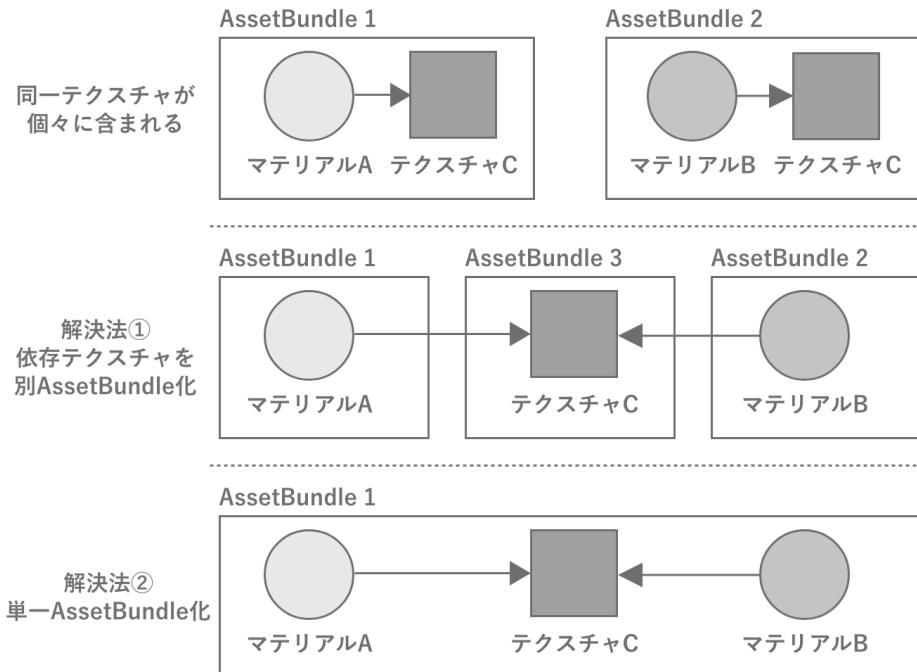
つまりロード時間を最速にするなら無圧縮がよいですが、ファイルサイズが致命的に大きくなるためスマートフォンにおける記憶領域の浪費を避けるためには基本的に使用できません。一方で LZMA はファイルサイズが一番小さくなりますが、アルゴリズムの問題で展開に時間がかかる、部分的な展開処理ができないという欠点があります。LZ4 は速度とファイルサイズのバランスのよい圧縮設定で、ChunkBasedCompression の名の通り部分展開が可能なため LZMA のように全体を展開しなくとも部分読み込みが可能です。

また AssetBundle には端末キャッシュ時に圧縮設定を変える `Caching.compressionEnabled` があります。つまり配信は LZMA で、端末で LZ4 に変換することで、ダウンロードサイズを最小にしつつ、実際に使う際には LZ4 の恩恵を受けられるようになります。ただし端末側で再圧縮するということは、それだけ端末での CPU の処理コストがかかる、メモリや記憶領域を一時的に浪費してしまうといった問題があります。

AssetBundle の依存関係と重複

あるアセットが複数のアセットから依存されている場合、AssetBundle 化する際に注意が必要です。たとえばマテリアル A とマテリアル B がテクスチャ C に依存している場合、テクスチャを AssetBundle 化せずに、マテリアル A と B だけ AssetBundle 化すると、生成される 2 つの AssetBundle のそれぞれにテクスチャ C が含まれるため、重複してムダになってしまいます。もちろん容量を使うという点でもムダなのですが、2 つのマテリアルをメモリにロードする際にテクスチャが別々にインスタンス化されるため、メモリもムダになってしまいます。

同一アセットが複数の AssetBundle に含まれるのを避けるためには、テクスチャ C も単体で AssetBundle 化してマテリアルの AssetBundle から依存される形にするか、マテリアル A、B とテクスチャ C を 1 つにした AssetBundle にする必要があります。

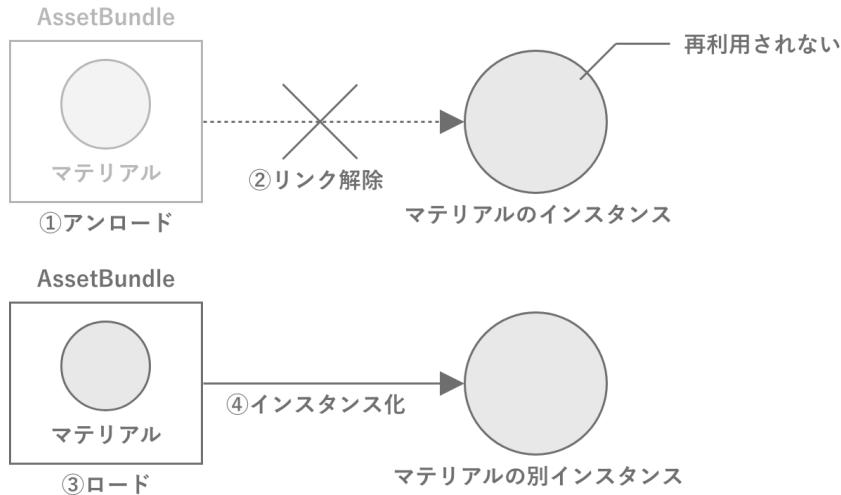


▲図 2.34 AssetBundle の依存関係がある例

AssetBundle からロードされたアセットの同一性

AssetBundle からのアセットをロードする時の重要な性質として、AssetBundle がロードされている間は同じアセットを何度ロードしても同じインスタンスが返ってきます。これは Unity 内部でロード済みのアセットを管理していることを示し、Unity 内部では AssetBundle とアセットは紐付けられた状態になります。この性質を利用することで、ゲーム側でアセットのキャッシング機構を作らずに Unity 側に委ねることも可能です。

ただし `AssetBundle.Unload(false)` でアンロードした場合のアセットは、図 2.35 のように再度同じ AssetBundle から同じアセットをロードしても別インスタンスとなるため、注意が必要です。これはアンロードするタイミングで AssetBundle とアセットの紐付けが解除されるため、アセットの管理が宙に浮いた状態に状態になるためです。



▲図 2.35 AssetBundle とアセットの管理が不適切でメモリリークする例

AssetBundle からロードしたアセットの破棄

`AssetBundle.Unload(true)` で AssetBundle をアンロードする場合は、ロードしたアセットも完全に破棄されるためにメモリに関してとくに困ることはありませんが、`AssetBundle.Unload(false)` を使用する場合は適切なタイミングでアセットのアンロード命令を呼び出さないとアセットが破棄されません。そのため後者を使用する場合は、シーン切替時などでアセットが破棄されるように適切に `Resources.UnloadUnusedAssets` を呼び出す必要があります。また `Resources.UnloadUnusedAssets` の名前の通り、参照が残っている場合は解放されないことにも注意が必要です。なお、Addressable を使用する場合は内部で `AssetBundle.Unload(true)` を呼び出します。

2.5 C#の基礎知識

この節では、パフォーマンスチューニングをする上で欠かせない、C#の言語仕様やプログラム実行時の挙動について説明します。

2.5.1 スタックとヒープ

「スタックとヒープ」ではプログラム実行時のメモリ管理方式としてのスタックとヒープが存在することを紹介しました。スタックはOSが管理するのに対して、ヒープはプログラム側が管理します。つまりヒープメモリはどうやって管理されているかを知ることで、メモリを意識した実装を行うことができます。ヒープメモリの管理の仕組みは、プログラムの元となったソースコードの言語仕様に依るところが大きいので、C#におけるヒープメモリの管理について解説します。

本来のヒープメモリは必要なタイミングでメモリ確保し、使い終わったらメモリを解放する必要があります。もしメモリを解放しない場合はメモリリークとなり、アプリケーションが使うメモリ領域が膨らみ、最終的にはクラッシュに繋がってしまいます。ところがC#には明示的なメモリ解放処理はありません。これはC#のプログラムが実行される.NETランタイム環境では、ヒープメモリがランタイムによって自動で管理され、使い終わったメモリは適切なタイミングで解放されるためです。このためヒープメモリのことをマネージドヒープとも呼びます。

スタックに確保されたメモリは関数のライフタイムと一致するので、関数の最後にメモリを解放してあげるだけでよいのですが、ヒープで確保されたメモリは関数のライフタイムを超えて生存することがほとんどです。つまりヒープメモリを必要としたり使い終わったりするタイミングがさまざまであるため、自動かつ効率よくヒープメモリを使うための仕組みが必要になります。詳細については次の項で紹介しますが、その仕組みをガベージコレクション(Garbage Collection)と呼びます。

実はUnityにおける**GC Alloc**は独自の用語で、ガベージコレクションで管理されているヒープメモリに確保(Assignment)されたメモリのことを表しています。そのため**GC Alloc**を減らすことは、動的に確保されるヒープメモリの量を減らすことになります。

2.5.2 ガベージコレクション

C#のメモリ管理において、未使用のメモリの検索や解放はガベージコレクション、略して「GC」と呼ばれます。ガベージコレクターは周期的に実行されます。ただし、正確な実行タイミングはアルゴリズムによって異なります。これにより、ヒープ上のすべてのオブジェクトが一斉調査され、すでに参照されなくなっているすべてのオブジェクトが削除されます。つまり、参照の外されたオブジェクトが削除され、メモリ領域が解放されます。

ガベージコレクターにはさまざまなアルゴリズムがありますが、Unity ではデフォルトで Boehm GC アルゴリズムが使用されています。Boehm GC アルゴリズムの特徴は、「非世代別」で「非圧縮型」であることです。「非世代別」とは、ガベージコレクションを 1 回実行するごとにヒープ全体を一斉調査しなければならないことを意味しています。このため、ヒープが拡張するのに応じて検索範囲も拡がるためパフォーマンスが低下します。「非圧縮型」とは、オブジェクト同士の隙間を詰めるためにメモリ内のオブジェクト移動が行われないことを意味します。つまり、メモリ上に細かい隙間を生む断片化が起こりやすく、マネージヒープの拡張がされやすい傾向にあります。

それぞれ計算コストが高い処理でありかつ他の処理をすべて止めてしまう同期的な処理であるため、ゲーム中に走るといわゆる「Stop the World」と呼ばれる処理落ちの原因に繋がります。

Unity 2018.3 からは GCMode を指定できるようになり、一時的に無効化することが可能になりました。

```
1: GarbageCollector.GCMode = GarbageCollector.Mode.Disabled;
```

しかし、当然のことながら無効化している期間に GC.Alloc をしてしまうと、ヒープ領域は拡張かつ消費され、最終的には新たに確保できなくなりアプリのクラッシュへと繋がります。メモリ使用量は簡単に増大していくため、無効化している期間では GC.Alloc が一切行われないように実装する必要があり、実装コストも高くなることから実際に利用できる場面は限られています。(例: シューティングゲームのシューティングパートのみ無効化するなど)

また、Unity 2019 から Incremental GC が選択できるようになりました。Incremental GC では、ガベージコレクションの処理がフレームを跨いで行われるようになり、大きなスパイクは以前より軽減可能になりました。しかしながら、1 フレームあたりの処理時間を削減しつつ最大限のパワーを発揮しなければならないようなゲームの場合、突き詰めると GC.Alloc の発生を避けた実装が必要になります。具体的な例については、「10.1 GC.Alloc するケースと対処法」で述べます。

いつから取り組むべきか

ゲームはコード量が多くなるため、全機能実装完了してからパフォーマンスチューニングを実施すると往々にして GC.Alloc を回避できないような設計／

実装に遭遇してしまうことがあります。設計初期段階から、どこで発生するのか常に意識した上でコーディングしていくと、作り直しによるコストも軽減できるようになり、トータルでの開発効率は改善される傾向にあります。

理想的な実装の流れとしては、まずはスピード重視でプロトタイプを制作し手触りや遊びのコアとなる部分を検証し、その次の本制作フェーズに進む際に一度設計を見直し再構築します。この再構築するフェーズで GC.Alloc の撲滅に取り組むと健全でしょう。場合によってはコードの可読性を下げてでも高速化を図る必要も出てくるため、プロトタイプから取り組んでいては開発速度も低下してしまいます。

2.5.3 構造体（struct）

C#では複合型の定義はクラスと構造体が存在します。大前提、クラスは参照型、構造体は値型となります。MSDNの「Choosing Between Class and Struct」⁷を引用しつつ、それぞれの特性と選択すべき基準、使い方の注意事項について確認します。

メモリの割り当て先の違い

参照型と値型の1つ目の違いは、メモリの割り当て先が異なる点です。少々正確性には欠けますが、次のように認識しておいて問題はありません。参照型はメモリ上のヒープ領域に割り当てられ、ガベージコレクションの対象となります。値型はメモリ上のスタック領域に割り当てられ、ガベージコレクションの対象にはなりません。値型の割り当てと割り当て解除は、参照型よりも一般的に低コストです。

ただし、参照型のフィールドに宣言されている値型や static 変数はヒープ領域に割り当てられます。このため、構造体として定義した変数が必ずしもスタック領域に割り当てられるわけではない点に注意しましょう。

配列の扱い

値型の配列はインラインで割り当てられ、配列要素は値型の実体（インスタンス）がそのまま並びます。一方、参照型の配列では、配列要素は参照型の実体への参照（アドレス）が並びます。したがって、値型の配列の割り当てと割り当て解除は、参照型より

⁷ <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-between-class-and-struct>

もはるかに低成本です。また、ほとんどの場合、値型の配列は参照の局所性（空間的局所性）が大幅に向上するため、CPU キャッシュメモリのヒット確率が高くなり、処理が高速化しやすくなるメリットがあります。

値のコピー

参照型の代入（割り当て）では、参照（アドレス）がコピーされます。一方、値型の代入（割り当て）では、値全体がコピーされます。アドレスのサイズは 32bit 環境の場合で 4 バイト、64bit 環境の場合で 8 バイトとなります。したがって、大きな参照型の割り当てでは、アドレスサイズより大きな値型の割り当てよりも低成本です。

また、メソッドを用いたデータのやり取り（引数・戻り値）に関しても、参照型は参照（アドレス）が値渡しされるのに対し、値型はインスタンスそのものが値渡しされます。

```
1: private void HogeMethod(MyStruct myStruct, MyClass myClass){...}
```

たとえばこちらのメソッドでは、`MyStruct` の値全体がコピーされます。つまり、`MyStruct` のサイズが大きくなるとその分コピーコストも増大します。一方 `MyClass` の方では、`myClass` の参照が値としてコピーされるだけになるため、`MyClass` のサイズが増大してもコピーコストはアドレスサイズ分のみであるため一定になります。コピーコストの増加は処理負荷に直結するため、扱うデータサイズに応じて適切に選択する必要があります。

不变性

参照型のインスタンスに加えた変更は、同じインスタンスを参照している別の場所にも影響します。一方、値型のインスタンスは、値渡しされるときにコピーが生成されます。値型のインスタンスが変更された場合、当然、そのインスタンスのコピーには影響しません。コピーはプログラマによって明示的に作成されるのではなく、引数が渡されるとき、または戻り値が返されるときに暗黙的に作成されます。プログラマとしては値を変更したつもりが、実はコピーに対して値をセットしていただけで、目的の処理とは異なっていたという不具合を一度は経験していることでしょう。変更可能な値型は多くのプログラマに混乱を招くおそれがあるため、値型は不变であることが推奨されています。

参照渡し

よくある誤用で「参照型は常に参照渡しになる」が挙げられますが、先述した通り参照（アドレス）のコピーが基本であり、参照渡しは ref/in/out パラメーター修飾子を用いたときに行われます。

```
1: private void HogeMethod(ref MyClass myClass){...}
```

参照型の値渡しでは参照（アドレス）がコピーされていたため、インスタンスの置き換えをしてもコピー元のインスタンスには影響しませんでしたが、参照渡しにすると元のインスタンスの置き換えも可能になります。

```
1: private void HogeMethod(ref MyClass myClass)
2: {
3:     // 引数で渡された元のインスタンスを書き換えてしまう
4:     myClass = new MyClass();
5: }
```

ボックス化

ボックス化とは、値型から object 型、または値型からインターフェイス型へ変換するプロセスのことです。ボックスはヒープに割り当てられ、ガベージコレクションの対象になるオブジェクトです。そのため、ボックス化とボックス化解除が過剰になると、GC.Alloc が発生します。これに対し、参照型がキャストされるとき、このようなボックス化は行われません。

▼リスト 2.7 値型から object 型にキャストするとボックス化

```
1: int num = 0;
2: object obj = num; // ボックス化
3: num = (int) obj; // ボックス化解除
```

このようにわかりやすく無意味なボックス化を使うことはありませんが、メソッドで使われている場合はどうでしょうか。

▼リスト 2.8 暗黙キャストでボックス化が行われる例

```
1: private void HogeMethod(object data){ ... }
2:
3: // 中略
4:
5: int num = 0;
6: HogeMethod(num); // 引数でボックス化
```

このようなケースで、無意識のうちにボックス化してしまっているケースは存在します。

簡単な代入と比べて、ボックス化およびボックス化解除は負荷の大きいプロセスです。値型をボックス化するときは、新しいインスタンスを割り当てて構築する必要があります。また、ボックス化ほどではありませんが、ボックス化解除に必要なキャストも大きな負荷がかかります。

クラスと構造体を選ぶ基準について

- 構造体を検討すべき条件:
 - 型のインスタンスが小さく、有効期間が短いことが多い場合
 - 他のオブジェクトに埋め込まれることが多い場合
- 構造体を避ける条件: ただし、型が次のすべての特性を持つ場合を除く
 - プリミティブ型 (int, double など) と同様に、論理的に单一の値を表すとき
 - インスタンスのサイズが 16 バイト未満である
 - 不変 (イミュータブル) である
 - 頻繁にボックス化する必要がない

上記の選択条件に当てはまらないものの、構造体と定義されている型も多数存在しています。Unity で頻繁に使用されている Vector4 や Quaternion など、16 バイト未満ではありませんが構造体で定義されています。これらを効率よく扱う方法を確認した上で、コピーコストが増大しているようでしたら回避する方法を含めて選択し、場合によっては自前で同等の機能を持った最適化版を作ることも検討してください。

2.6 アルゴリズムと計算量

ゲームプログラミングにはさまざまなアルゴリズムが利用されます。アルゴリズムは作り方次第で計算結果は同じでも、途中の計算過程が異なることでパフォーマンス

は大きく変わることがあります。たとえば、C#に標準で用意されているアルゴリズムはどれくらい効率のよいものなのか、あなたが実装したアルゴリズムはどれくらい効率のよいものなのか、それぞれ評価する尺度が欲しくなります。これらを測る目安として、計算量という指標が用いられています。

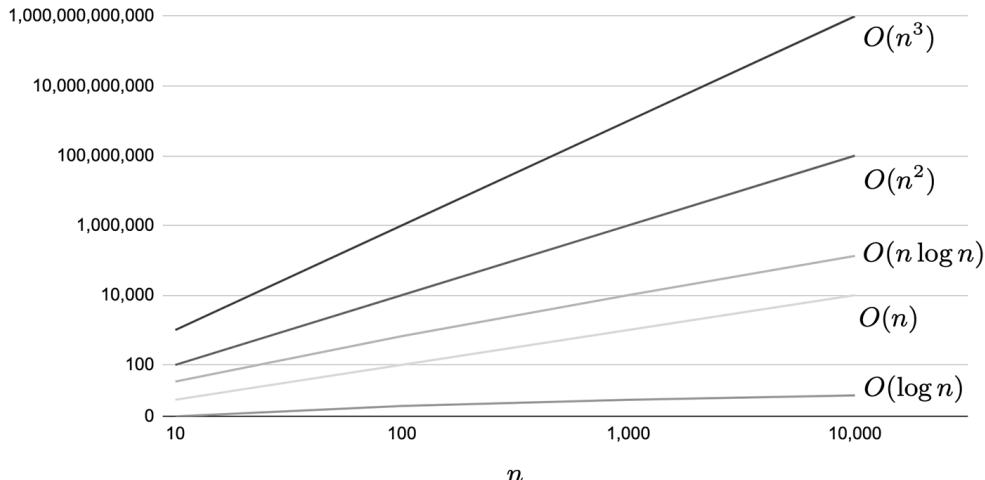
2.6.1 計算量について

計算量とはアルゴリズムの計算効率を測る尺度のことです、細かく分けると時間効率を測る時間計算量やメモリ効率を測る領域計算量などがあります。計算量オーダーは O 記法（ランダウの記号）で表されます。計算機科学や数学的な定義などはここでは本質ではないため、気になる方は他の書籍を参照してください。また、本稿では計算量と記載しているものは時間計算量として取り扱います。

一般的に使われるおもな計算量は $O(1)$ 、 $O(n)$ 、 $O(n^2)$ 、 $O(n \log n)$ のように表記されます。括弧内の n はデータ数を示しています。ある処理がどれくらいデータ数に依存して処理回数が増えていくかをイメージするとわかりやすいでしょう。計算量の観点から性能を比較すると、 $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$ となります。表 2.5 にデータ数と計算ステップ数の比較と、図 2.36 に対数表示した比較グラフを示しました。 $O(1)$ はデータ数によらないため比べるまでもなく明らかに性能が高いため除いてあります。たとえば、 $O(\log n)$ はデータ数が 1 万サンプルあったとしても計算ステップ数は 13、1,000 万サンプルあったとしても計算ステップ数が 23 回と極めて優秀であることがわかります。

▼表 2.5 おもな計算量におけるデータ数と計算ステップ数

n	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
10	3	10	33	100	1,000
100	7	100	664	10,000	1,000,000
1,000	10	1,000	9,966	1,000,000	1,000,000,000
10,000	13	10,000	132,877	100,000,000	1,000,000,000,000



▲図 2.36 各計算量の対数表示による性能差比較

それぞれの計算量を示すため、いくつかコードサンプルを挙げて行きます。まず、 $O(1)$ はデータ数に依存せず一定の計算量であることを示します。

▼リスト 2.9 $O(1)$ のコード例

```

1: private int GetValue(int[] array)
2: {
3:     // arrayには何らかの整数値が入っている配列とする
4:     var value = array[0];
5:     return value;
6: }
```

このメソッドの存在意義はさておき、明らかに array のデータ数に依存することなく処理は一定回数（ここでは 1 回）で終わります。

次に $O(n)$ のコード例を見てみましょう。

▼リスト 2.10 $O(n)$ のコード例

```

1: private bool HasOne(int[] array, int n)
2: {
3:     // arrayはlength=nで、何らかの整数値が入っているとする
4:     for (var i = 0; i < n; ++i)
5:     {
6:         var value = array[i];
7:         if (value == 1)
8:     }
```

```
9:             return true;
10:        }
11:    }
12: }
```

こちらは、整数値の入った配列に 1 が存在していたら `true` を返すだけの処理です。偶然 `array` の最初に 1 が入っていたら最速で処理が終わる可能性もありますが、`array` のなかにどこにも 1 がない場合や、`array` の最後にはじめて 1 があった場合にはループは最後まで回るため n 回処理をすることになります。この最悪のケースのときを $O(n)$ として表し、データ数に応じて計算量が増えしていくイメージが浮かぶことでしょう。

次に $O(n^2)$ のときの例を見てみましょう。

▼リスト 2.11 $O(n^2)$ のコード例

```
1: private bool HasSameValue(int[] array1, int[] array2, int n)
2: {
3:     // array1, array2はlength=nで、何らかの整数値が入っているとする
4:     for (var i = 0; i < n; ++i)
5:     {
6:         var value1 = array1[i];
7:         for (var j = 0; j < n; ++j)
8:         {
9:             var value2 = array2[j];
10:            if (value1 == value2)
11:            {
12:                return true;
13:            }
14:        }
15:    }
16:
17:    return false;
18: }
```

こちらは二重ループで 2 つの配列のどこかに同じ値が含まれていたら `true` を返すだけのメソッドです。最悪のケースを考えるとすべて不一致のケースとなるため、その場合は n^2 回処理が走ることになります。

余談ですが、計算量の考え方では最大次数の項のみで表現します。上記例の 3 つのメソッドを 1 回ずつ実行するメソッドを作ると、最大次数の $O(n^2)$ になります。
($O(n^2 + n + 1)$ にはなりません)

また、計算量はあくまでデータ数が十分多いときの目安であり、実計測時間と必ずしも連動するものではないことに注意しておきましょう。 $O(n^5)$ のような巨大な計算量に見えてもデータ数が少ない場合、問題にならないケースもあるため、計算量は参考にしつつも都度データ数を考慮して問題ない処理時間に収まるか測定することを推奨します。

2.6.2 基本的なコレクションとデータ構造

C#にはさまざまなデータ構造を持つコレクションクラスが用意されています。よく使うものを例に挙げつつ、おもなメソッドの計算量を踏まえてそれぞれどういうシチュエーションで採用るべきかを紹介します。

ここで紹介しているコレクションクラスにおけるメソッドの計算量については MSDN にすべて掲載されているため、最適なコレクションクラスを選定するときに確認できることより安全でしょう。

List<T>

もっともよく使われているであろう List<T>です。データ構造は配列です。データの並び順が重要な場合や、インデックスによるデータの取得や更新が多い場合に用いると効果的です。逆に要素の挿入や削除が多くなる場合には操作したインデックス以降のコピーが必要になり計算量が大きくなるため、List<T>の使用を避けたほうが無難でしょう。

また、Add でキャパシティを超えようとしたときには、配列の確保メモリの拡張が行われます。メモリの拡張時は現在の Capacity の 2 倍を確保することになるため、Add を $O(1)$ で使うためにも拡張を発生させずに使用できるように適切な初期値を設定して使用しましょう。

▼表 2.6 List<T>

メソッド	計算量
Add	$O(1)$ ただしキャパシティを超えたときは $O(n)$
Insert	$O(n)$
IndexOf/Contains	$O(n)$
RemoveAt	$O(n)$
Sort	$O(n \log n)$

LinkedList<T>

LinkedList<T>のデータ構造は連結リストです。連結リストは基本的なデータ構造で、各ノードが次のノードの参照を持っているようなイメージです。C#の LinkedList<T>は双方向の連結リストであるため、前後のノードへの参照をそれぞれ持っています。LinkedList<T>は、要素の追加や削除に強い特徴がありますが、配列内の特定の要素にアクセスするのは苦手です。頻繁に追加や削除を行う必要があるような一時的にデータを保持する処理を作りたいときなどに適しています。

▼表 2.7 LinkedList<T>

メソッド	計算量
AddFirst/AddLast	$O(1)$
AddAfter/AddBefore	$O(1)$
Remove/RemoveFirst/RemoveLast	$O(1)$
Contains	$O(n)$

Queue<T>

Queue<T>は先入れ先出し法: FIFO (First in first out) を実現したコレクションクラスです。入力操作などを管理するときなど、いわゆる待ち行列を実装するときに用いられます。Queue<T>では循環配列が用いられています。Enqueue で要素を末尾に追加して、Dequeue で先頭の要素を取り出しつつ削除します。キャパシティを超えて追加する際には拡張が行われます。Peek は削除をせずに先頭の要素を取り出す操作です。計算量を見ても明らかなように Enqueue と Dequeue に留めて使うと高いパフォーマンスを得られます。探索などの操作には向かないでしょう。TrimExcess はキャパシティを削減するメソッドですが、パフォーマンスチューニング観点から見ると、そもそもキャパシティが増減しないように使用できるとさらに Queue<T>の強みを活かせます。

▼表 2.8 Queue<T>

メソッド	計算量
Enqueue	$O(1)$ ただしキャパシティを超えたときは $O(n)$
Dequeue	$O(1)$
Peek	$O(1)$
Contains	$O(n)$
TrimExcess	$O(n)$

Stack<T>

Stack<T>は後入れ先出し法: LIFO (Last in first out) を実現したコレクションクラスです。Stack<T>は配列で実装されています。Push で先頭に要素を追加し、Pop で先頭の要素を取り出しつつ削除します。Peek は削除をせずに先頭の要素を取り出す操作です。よく使われる場面としては画面遷移を実装するときに遷移時に進んだ先のシン情報を Push しておき、戻るボタンを押したときに Pop するときなどが挙げられます。Stack も Queue と同様に Push と Pop のみを用いると高いパフォーマンスが得られます。要素の探索などは行わずに、キャパシティの増減にも注意しましょう。

▼表 2.9 Stack<T>

メソッド	計算量
Push	$O(1)$ ただしキャパシティを超えたときは $O(n)$
Pop	$O(1)$
Peek	$O(1)$
Contains	$O(n)$
TrimExcess	$O(n)$

Dictionary< TKey, TValue >

これまで紹介したコレクションは順序に意味を持つものでしたが、Dictionary< TKey, TValue >は索引性に特化したコレクションクラスです。データ構造はハッシュテーブル（連想配列の一種）で実装されています。キーに対応する値がある辞書（辞書の場合単語がキー、説明が値）のような構造です。Dictionary< TKey, TValue >はメモリを多く消費するデメリットはありますが、その分参照速度が $O(1)$ と高速です。列挙や探索を必要とせず、値を参照することに重きを置くようなケースでとても重宝します。また、キャパシティの事前設定を必ず行いましょう。

▼表 2.10 Dictionary< TKey, TValue >

メソッド	計算量
Add	$O(1)$ ただしキャパシティを超えたときは $O(n)$
TryGetValue	$O(1)$
Remove	$O(1)$
ContainsKey	$O(1)$
ContainsValue	$O(n)$

2.6.3 計算量を下げる工夫

これまで紹介したコレクション以外にもさまざまなもののが用意されています。もちろん、List<T>（配列）だけでも同様の処理を実装することは可能ですが、より適したコレクションクラスを選択することで計算量の最適化が可能になります。計算量を意識してメソッドを実装をしていくだけでも重い処理を避けることができるようになるでしょう。コード最適化における1つの切り口として、自分が作ったメソッドの計算量を確認して、より少ない計算量にできないか検討してみてはいかがでしょうか。

工夫の手段: メモ化

とある複雑な計算をしなければならないような、とても高い計算量のメソッド（ComplexMethod）があるとします。しかしどうにも計算量を減らすことができないときもあるでしょう。こうしたときに用いられる手段としてメモ化と呼ばれる手法があります。

ここでの ComplexMethod は引数を与えると対応した結果が一意に返るものとします。まず、渡された引数が初回のときには複雑な処理を通して計算後、引数と計算結果を Dictionary< TKey, TValue >に入れてキャッシュしておきます。2回目以降はまずはキャッシュされてないか調べ、すでにキャッシュされていたらその結果だけを返して終了します。こうすることで、初回がどれだけ高い計算量でも2回目以降は $O(1)$ に抑えることが可能です。もし事前に渡されうる引数がある程度決まっているようでしたら、ゲームの前に計算を済ませてキャッシュしておくことで、事実上 $O(1)$ の計算量で処理することが可能になります。



PERFORMANCE TUNING BIBLE

CHAPTER

03

第3章

プロファイリング
ツール

CyberAgent Smartphone Games & Entertainment

第3章

プロファイリングツール

プロファイリングツールはデータを収集・解析し、ボトルネックの特定やパフォーマンスの指標を決める際に使用します。これらのツールは Unity エンジンが提供しているものだけでもいくつかあります。他にも Xcode や Android Studio と言ったネイティブ準拠のツールや、GPU に特化した RenderDoc など、さまざまなツールが存在します。そのためそれぞれのツールの特徴を理解し、適切に取捨選択することが重要になるでしょう。この章ではそれぞれのツールの紹介とプロファイル方法について取り上げ、適切に使用できる状態を目指します。

3.0.1 計測時の注意点

Unity はエディター上でアプリケーションを実行できるため「実機」と「エディター」のどちらでも計測が可能です。計測を行うにあたって、それぞれの環境での特徴を押さえておく必要があります。

エディターで計測する場合、トライ & エラーを素早くできることが最大の強みでしょう。しかしエディター自身の処理負荷や、エディターが使用しているメモリ領域も計測されるため、計測結果にノイズが多くなります。また実機とはスペックがまったく違うので、ボトルネックがわかりにくく結果が違うこともあるでしょう。

そのためプロファイリングは基本的には実機での計測を推奨します。ただし「どちらの環境でも発生する」場合に限り、作業コストが低いエディターだけで作業を完結させるのが効率的です。大体はどちらの環境でも再現しますが、まれにどちらかの環境でしか再現しないことがあります。そのため、まずは実機で現象を確認します。次にエディターでも再現を確認した上で、エディター上で修正するというフローがよいでしょう。もちろん最後に実機での修正確認は必ず行いましょう。

3.1 Unity Profiler

Unity Profiler は Unity エディターに組み込まれているプロファイリングツールです。このツールは 1 フレームごとに情報を収集できます。計測できる項目は多岐に渡り、

3.1 Unity Profiler

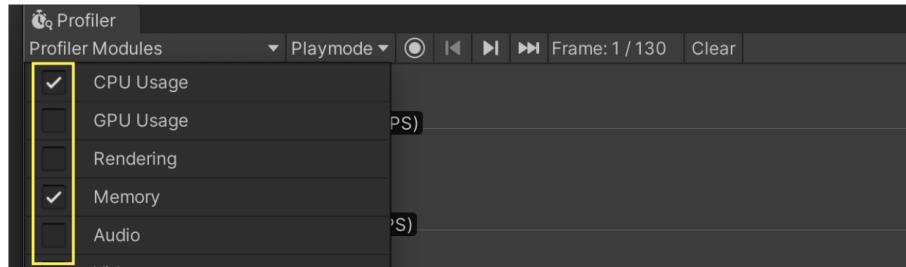
それぞれの項目をプロファイラー モジュールと呼び、Unity 2020 のバージョンでは 14 項目も存在します。このモジュールはいまなお更新されており、Unity 2021.2 では新たに Asset に関するモジュールや、File I/O に関するモジュールも追加されました。このように Unity Profiler は、さまざまなモジュールがあるため、ざっくりとパフォーマンスの外観を掴むのに最適なツールと言えるでしょう。モジュール一覧は図 3.1 のようになります。

モジュール名	説明
CPU Usage	スクリプトやアニメーションなど、CPU が使用した時間の内訳
GPU Usage	オブジェクトのレンダリングなど、GPU が使用した時間の内訳
Rendering	SetPass や Batchingなど描画に関わる情報
Memory	アプリケーション全体でのメモリ割り当てに関する情報
Audio	Audioに関するメモリ割り当て、CPU使用率などの情報
Video	Videoに関するメモリ割り当て、バッファリングフレーム数などの情報
Physics	物理エンジンに関するオブジェクト
Physics2D	2Dの物理エンジンに関するオブジェクトの情報 (RigidBody2D)
Network Messages (非推奨)	マルチプレイヤー高レベルAPI (非推奨)に関する送受信の情報
Network Operations (非推奨)	マルチプレイヤー低レベルAPI (非推奨)に関する送受信の情報
UI	UIに関する処理時間の情報
Global Illumination	UI表示時のバッチ数や頂点数の情報
Virtual Texturing	ライトプローブなど、GIライティングに使用した時間の情報
Asset Loading (2021.2 以降)	Texture や Mesh など、ロードタイミングやサイズなどの情報
File Access (2021.2 以降)	I/O に費やした時間など、ファイルアクセスに関わる情報

▲図 3.1 プロファイラー モジュール一覧

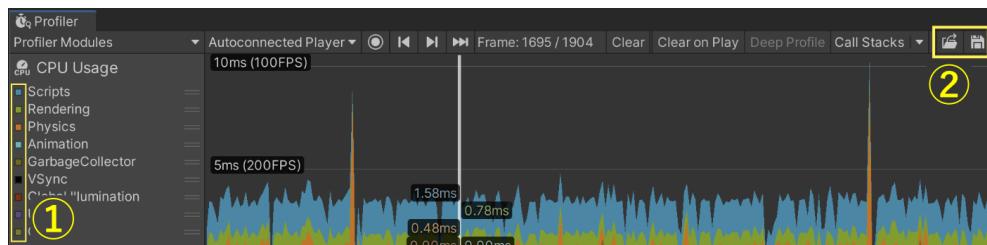
これらのモジュールはプロファイラー上に表示するかどうかの設定が可能です。ただし表示していないモジュールは計測もされていません。逆に言うとすべてを表示している場合、それだけエディターに負荷がかかります。

第3章 プロファイリングツール



▲図 3.2 Profiler Modules の表示/非表示機能

また、プロファイラーツール全体で共通する便利な機能を紹介します。



▲図 3.3 Profiler の機能説明

図 3.3において「①」は各モジュールが計測している項目が列挙されています。この項目をクリックすることで右側のタイムラインへの表示・非表示の切り替えができます。必要な項目だけを表示することでビューが見やすくなるでしょう。またこの項目はドラッグで並び替えることもでき、右側のグラフはその並び順で表示されます。「②」は計測したデータの保存、読み込みの機能です。必要であれば計測結果を保存しておくとよいでしょう。保存されるデータはプロファイラー上に表示しているデータのみです。

本書籍では図 3.1 のうち、使用頻度が高い CPU Usage と Memory モジュールについて解説を行います。

3.1.1 計測方法

ここでは Unity Profiler で実機を用いた計測方法について取り上げます。ビルト前に行う作業とアプリケーション起動後に行う作業の 2 つに分けて解説します。なお、エディターでの計測方法は実行中に計測ボタンを押下するだけなので詳細は割愛します。

ビルド前に行う作業

ビルド前に行う作業は **Development Build** を有効にすることです。この設定が有効化されることでプロファイラーと接続ができるようになります。

また、より詳細に計測を行う **Deep Profile** というオプションもあります。このオプションを有効にすると、すべての関数コールの処理時間が記録されるため、ボトルネックとなる関数の特定が容易にできます。欠点としては計測自体に非常に大きなオーバーヘッドをするため、動作が重くなり、メモリも大量に消費します。そのため処理にとても時間がかかるているように見えても、通常のプロファイルではそれほどでもないこともあるので注意してください。基本的には通常のプロファイルでは情報が足りない場合にのみ使用します。

Deep Profile は大規模プロジェクトなどでメモリを多く使用している場合、メモリ不足で計測ができないこともあるでしょう。その場合は「3.1.2 CPU Usage」節の「補足：Samplerについて」を参考に、独自に計測処理を追加するしかありません。

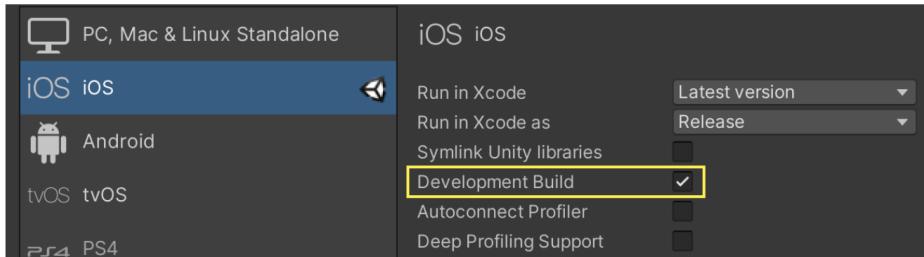
これらの設定方法はスクリプトから明示的に指定する方法と、GUI から行う方法があります。まずスクリプトから設定する方法を紹介します。

▼リスト 3.1 スクリプトからの Development Build を設定する方法

```
1: BuildPlayerOptions buildPlayerOptions = new BuildPlayerOptions();
2: /* シーンやビルドターゲットの設定は省略 */
3:
4: buildPlayerOptions.options |= BuildOptions.Development;
5: // Deep Profileモードを有効にしたい場合のみ追加
6: buildPlayerOptions.options |= BuildOptions.EnableDeepProfilingSupport;
7:
8: BuildReport report = BuildPipeline.BuildPlayer(buildPlayerOptions);
```

リスト 3.1 で重要な点は `BuildOptions.Development` を指定することです。

次に GUI から設定する場合は、Build Settings から図 3.4 のように Development Build にチェックしてビルドをします。



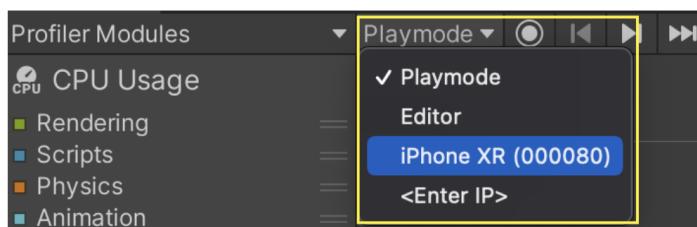
▲図 3.4 Build Settings

アプリケーション起動後に行う作業

アプリケーション起動後に Unity Profiler と接続する方法は「リモート接続」と「有線(USB)接続」の2種類があります。リモート接続は有線接続に比べて環境の制約が増え、プロファイルが思ったようにできないこともあります。たとえば同じ Wifi ネットワークへの接続が必須だったり、Android のみモバイル通信を無効にする必要があったり、他にもポート開放が必要だったりします。そのため、本節では手順がシンプルで確実にプロファイルができる有線接続について取り上げます。もしリモート接続をしたい場合は、公式ドキュメントを見ながら試してください。

はじめに iOS の場合、プロファイルへの接続は次のような手順で行います。

1. Build Settings から Target Platform を iOS に変更しておく
2. デバイスを PC に接続し、Development Build のアプリケーションを起動する
3. Unity Profiler から接続先の端末を選択する(図 3.5)
4. Record を開始する



▲図 3.5 接続先端末の選択

計測を行う Unity エディターは、ビルドしたプロジェクトでなくても構いません。計測用に新規プロジェクトを作成すると動作も軽量なのでオススメです。

次に Android の場合は、iOS に比べ少し手順が増えます。

1. Build Settings から Target Platform を Android に変更しておく
2. デバイスを PC に接続し、Development Build のアプリケーションを起動する
3. adb forward コマンドを入力する。(コマンドの詳細は後述)
4. Unity Profiler から接続先の端末を選択する
5. Record を開始する

adb forward コマンドには、アプリケーションの Package Name が必要になります。たとえば Package Name が「jp.co.sample.app」だった場合、以下のように入力します。

▼リスト 3.2 adb forward コマンド

```
1: adb forward tcp:34999 localabstract:Unity-jp.co.sample.app
```

adb が認識されていない場合は、adb のパス設定を行ってください。設定方法に関しては Web 上に解説してある情報がいくつもあるため割愛します。

簡易トラブルシューティングとして、接続できない場合は以下を確認してください。

- 両デバイス共通
 - 実行したアプリケーションの右下に Development Build という表記があるか
- Android の場合
 - 端末の USB デバッグを有効化しているか
 - adb forward コマンドの入力したパッケージ名が正しいか
 - adb devices コマンドを入力した際に、デバイスが正常に認識されているか

補足となりますが Build And Run で直接アプリケーションを実行した場合、上述の adb forward コマンドが内部的に行われます。そのため計測時にコマンド入力は必要ありません。

Autoconnect Profiler

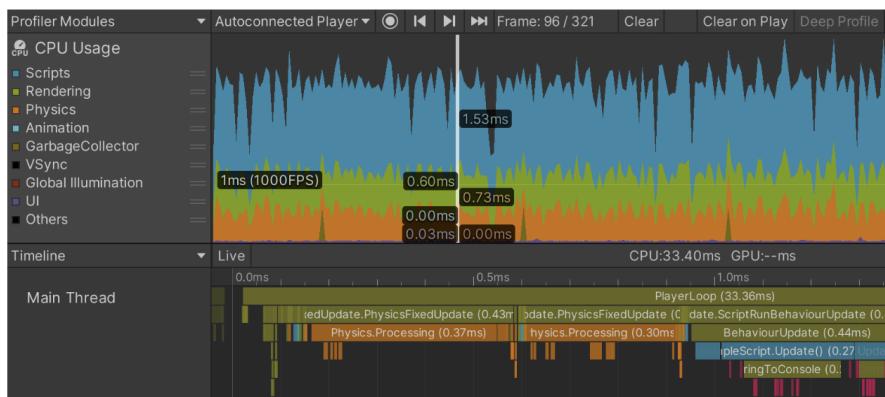
ビルド設定には Autoconnect Profiler というオプションがあります。このオプションはアプリ起動時にエディターのプロファイラーに自動接続するための機能です。そのためプロファイルに必須の設定ではありません。リモートプロファイル時も同様です。WebGL のみ、このオプションがないとプロファイルできませんが、モバイルに関してはそれほど重宝するオプションではないでしょう。

もう少し踏み込んだ話をすると、このオプションが有効な場合、ビルド時にエディターの IP アドレスがバイナリへ書き込まれ、起動時にそのアドレスへ接続を試みるようになります。専用のビルトマシンなどでビルドしている場合、そのマシンでプロファイルをしない限りは不要です。むしろアプリケーション起動時に自動接続がタイムアウトする（8 秒ほどの）待ち時間が増えるだけになります。

スクリプトからは `BuildOptions.ConnectWithProfiler` というオプション名になっており、必須のように勘違いしやすいので気をつけましょう。

3.1.2 CPU Usage

CPU Usage は図 3.6 のように表示されます。



▲図 3.6 CPU Usage モジュール (Timeline 表示)

このモジュールの確認方法は大きく分けて次の2つになります。

- Hierarchy (Raw Hierarchy)
- Timeline

まずは Hierarchy ビューに関して、表示内容と使い方について解説します。

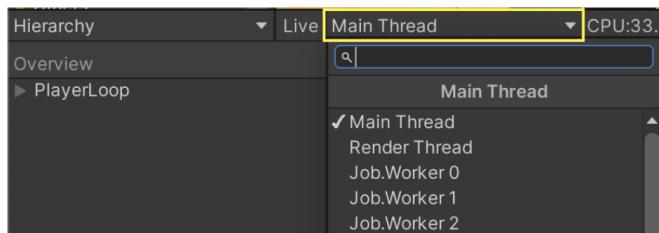
1. Hierarchy ビュー

Hierarchy ビューは図 3.7 のような表示になります。

Hierarchy	Live	Main Thread	CPU:33.46ms	GPU:--ms		
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	99.8%	0.1%	1	2.4 KB	33.42	0.06
► WaitForTargetFPS	88.5%	88.5%	1	0 B	29.63	29.62
► PostLateUpdate.FinishFrameRendering	7.7%	0.2%	1	0 B	2.59	0.07
▼ Update.ScriptRunBehaviourUpdate	1.5%	0.0%	1	2.4 KB	0.51	0.00
▼ BehaviourUpdate	1.5%	0.0%	1	2.4 KB	0.51	0.01
► SampleScript.Update()	1.2%	0.2%	2	2.4 KB	0.41	0.08
EventSystem.Update()	0.1%	0.1%	1	0 B	0.04	0.04
DebugUpdater.Update()	0.0%	0.0%	1	0 B	0.02	0.02

▲図 3.7 Hierarchy ビュー

このビューの特徴としては、リスト形式で計測結果が並んでおり、ヘッダーの項目でソートが可能なことです。調査を行う際はリストの中から気になる項目を開いていくことで、ボトルネックが判別できます。ただし、表示されている情報は「選択しているスレッド」で費やされた時間の表示です。たとえば、Job System やマルチスレッドレンダリングを使用している場合、別スレッドでの処理時間は含まれません。確認したい場合は図 3.8 のようにスレッドを選択することで可能です。



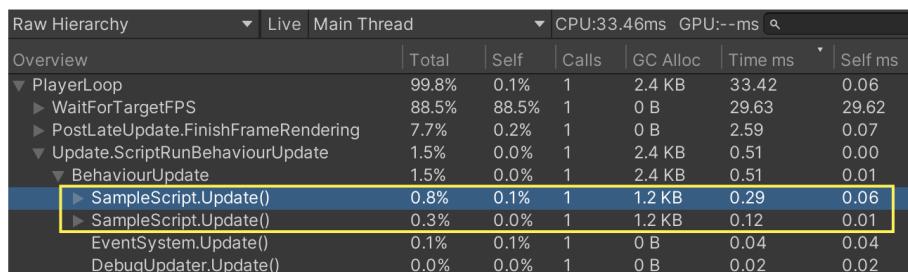
▲図 3.8 スレッド選択

次にヘッダーの項目について解説します。

▼表 3.1 Hierarchy ヘッダー情報

ヘッダー名	説明
Overview	サンプル名。
Total	この関数の処理にかかった合計時間。(%表示)
Self	この関数自体の処理時間。サブ関数の処理時間は含まれません。(%表示)
Calls	1 フレーム内で呼ばれた回数。
GC Alloc	この関数で割り当てたスクリプトのヒープメモリ。
Time ms	Total を ms で表示したもの。
Self ms	Self を ms で表示したもの。

Calls は複数回の関数呼び出しを 1 つの項目としてまとめるのでビューとして見やすくなります。しかし、すべてが等しい処理時間なのか、うち 1 つだけ処理時間が長いのかはわかりません。このような場合に **Raw Hierarchy** ビューを利用します。Raw Hierarchy ビューは、必ず Calls が 1 に固定されるという点で Hierarchy ビューと違います。図 3.9 では Raw Hierarchy ビューにしているため、同じ関数呼び出しが複数表示されています。



▲図 3.9 Raw Hierarchy ビュー

今までの内容をまとめると Hierarchy ビューは次のような用途で使用します。

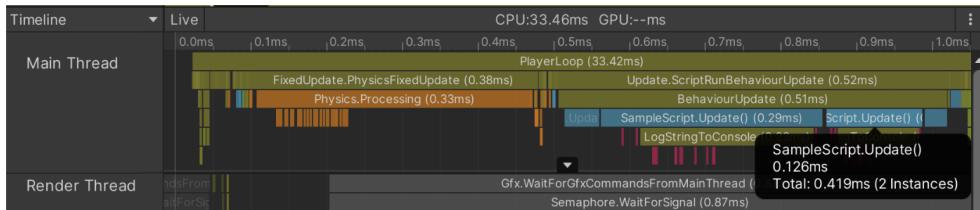
- 処理落ちしているボトルネックの把握、最適化 (Time ms、Self ms)
- GC アロケーションの把握、最適化 (GC Allocation)

これらの作業を行う際は、目的の項目ごとに降順ソートを行ってから確認することを推奨します。

項目を開いていく際、深い階層になっていることがあります。この時、Mac の場合は Option キー（Windows の場合は Alt キー）を押下しながら開くことで、すべての階層が開けます。逆にキーを押しながら項目を閉じると、その階層以下が閉じた状態になります。

2. Timeline ビュー

もう 1 つの確認方法となる Timeline ビューは次のような表示になります。



▲図 3.10 Timeline ビュー

Timeline ビューでは、Hierarchy ビューの項目がボックスとして可視化されているため、一目で全体を見たときに負荷のかかっている箇所が直感的にわかります。そしてマウスによる操作が可能なため、階層が深い場合もドラッグするだけで全容が把握できます。さらに Timeline ではわざわざスレッドを切り替える必要がなく、すべて表示されています。そのため各スレッドでいつどのような処理が行われているかも簡単に把握できます。このような特徴から、主に次のような用途で使用します。

- 処理負荷を全体俯瞰してみたい
- 各スレッドの処理負荷を把握、チューニングしたい

Timeline が向いていないのは、重い処理順を把握するためのソート操作や、アロケーション総量を確認する場合などです。そのためアロケーションのチューニングに関しては Hierarchy ビューの方が向いているでしょう。

補足 : Sampler について

関数単位で処理時間を計測する場合、2通りの方法があります。1つは先に紹介した Deep Profile モードです。もう 1 つはスクリプトに直接埋め込む方法です。

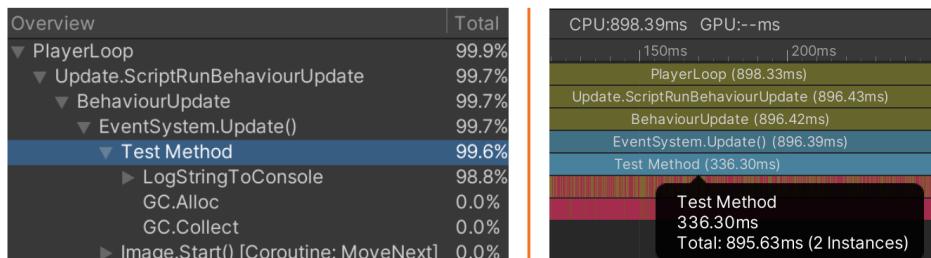
直接埋め込む場合は次のように記載します。

▼リスト 3.3 Begin/EndSample を用いた方法

```

1: using UnityEngine.Profiling;
2: /* ...省略... */
3: private void TestMethod()
4: {
5:     for (int i = 0; i < 10000; i++)
6:     {
7:         Debug.Log("Test");
8:     }
9: }
10:
11: private void OnClickedButton()
12: {
13:     Profiler.BeginSample("Test Method")
14:     TestMethod();
15:     Profiler.EndSample()
16: }
```

埋め込んだサンプルは Hierarchy ビュー、Timeline ビューのどちらにも表示されます。



▲図 3.11 Sampler の表示

もう 1 つ特筆すべき特徴があります。それはプロファイリングのコードは Development Build でない場合、呼び出し側が無効化されるためオーバーヘッドはゼロになることです。今後処理負荷が増えそうな箇所には事前に仕込んでおくのも手かもしれません。

BeginSample メソッドは static 関数なので気軽に使えますが、似たような機能を持った CustomSampler というのも存在します。こちらは Unity 2017 以降に追加され、BeginSample より計測のオーバーヘッドが少ないとため、より正確な時間が計測できるという特徴があります。

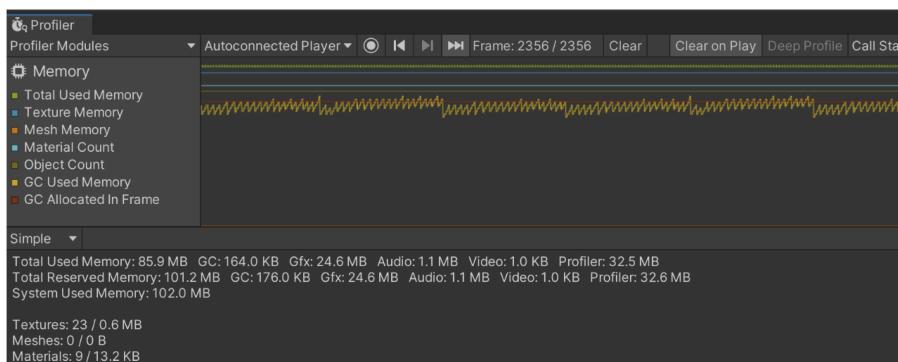
▼リスト 3.4 CustomSampler を用いた方法

```
1: using UnityEngine.Profiling;
2: /* ...省略... */
3: private CustomSampler _samplerTest = CustomSampler.Create("Test");
4:
5: private void TestMethod()
6: {
7:     for (int i = 0; i < 10000; i++)
8:     {
9:         Debug.Log("Test");
10:    }
11: }
12:
13: private void OnClickedButton()
14: {
15:     _samplerTest.Begin();
16:     TestMethod();
17:     _samplerTest.End();
18: }
```

違いとしては事前にインスタンスを生成しておく必要があるところです。CustomSampler は計測後、スクリプト内で計測時間を取り得るのも特徴です。より正確性が必要な場合や、処理時間に応じて警告などを出したい場合は、CustomSampler を使用するのがよいでしょう。

3.1.3 Memory

メモリモジュールは図 3.12 のように表示されます。



▲図 3.12 Memory モジュール

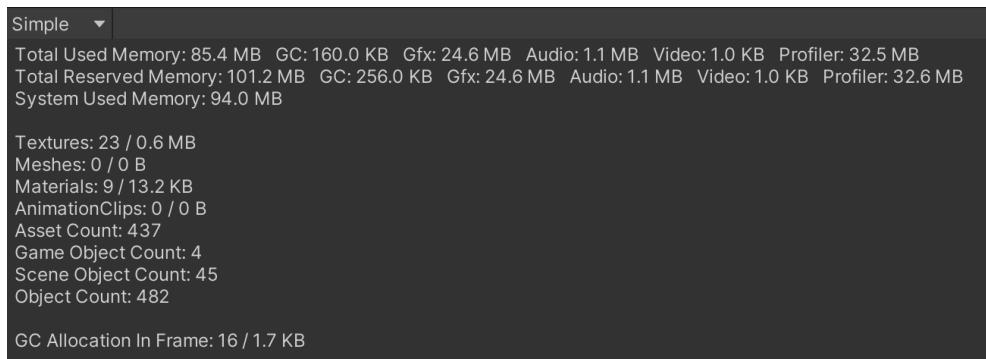
このモジュールの確認方法は次の 2 つになります。

- Simple ビュー
- Detailed ビュー

まずは Simple ビューに関して、表示内容と使い方について解説します。

1. Simple ビュー

Simple ビューは図 3.13 のような表示になります。



▲図 3.13 Simple ビュー

ビューに記載されている項目について説明します。

Total Used Memory

Unity が割当済み（使用中）のメモリ合計量。

Total Reserved Memory

Unity が現在確保しているメモリの合計量。OS 側にあらかじめ一定量の連続メモリ領域をプールとして確保しておき、必要になったタイミングで割り当てていきます。プール領域が足りなくなると再度 OS 側に要求し拡張します。

System Used Memory

アプリケーションで使用しているメモリの合計量。この項目では、Total Reserved で計測されていない項目（プラグインなど）も計測されます。ただし、これでもすべてのメモリ割り当ては追跡できません。正確に把握する場合は、Xcode などネイティブ準拠のプロファイリングツールを使う必要があります。

図 3.13 の Total Used Memory の右側に記載されている項目の意味は次の通りです。

▼表 3.2 Simple View 用語説明

用語名	説明
GC	ヒープ領域で使用しているメモリ量。GC Alloc などが要因で増加します。
Gfx	Texture、Shader、Mesh などで確保しているメモリ量。
Audio	オーディオ再生のために使用しているメモリ量。
Video	Video 再生のために使用しているメモリ量。
Profiler	プロファイリングを行うために使用しているメモリ量。

用語名に関する補足ですが、Unity 2019.2 以降から「Mono」は「GC」に、「FMOD」は「Audio」に表記が変更されました。

図 3.13 には他にも、次に示すアセットの使用個数とメモリ確保量も記載されています。

- Texture
- Mesh
- Material
- Animation Clip
- Audio Clip

また、次に示すようなオブジェクト数や GC Allocation に関する情報もあります。

Asset Count

ロード済みのアセット総数。

Game Object Count

シーンに存在するゲームオブジェクト数。

Scene Object Count

シーンに存在するコンポーネントやゲームオブジェクトなどの総数。

Object Count

アプリケーションで生成した、ロードしたすべてのオブジェクト総数。この値が増えている場合、何かのオブジェクトがリークしている可能性が高いです。

GC Allocation in Frame

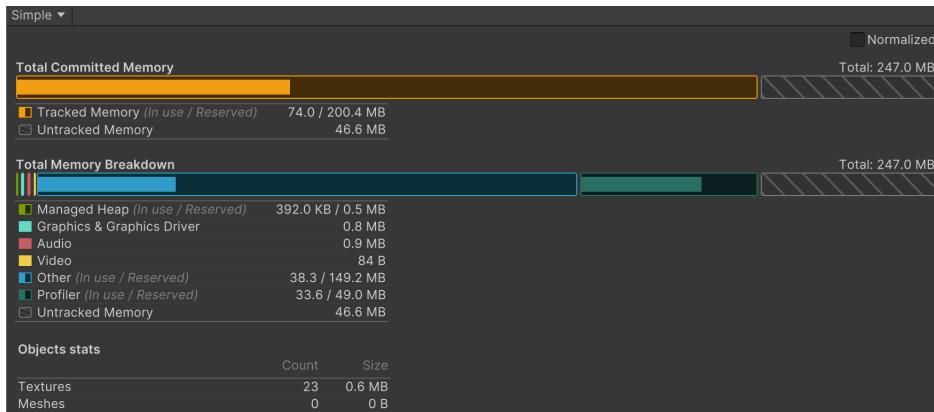
1 フレーム内で Allocation の発生した回数と総量。

最後に、これらの情報から Simple ビューの利用ケースについてまとめます。

- ヒープ領域や Reserved の拡張タイミングの把握・監視
- 各種アセットやオブジェクトがリークしていないかの確認

- GC Allocation の監視

Unity 2021 以降の Simple ビューは UI が大きく改善され、表示項目が見やすくなりました。内容そのものには大きな変更はないため、ここで紹介した知識はそのまま使えます。ただし名称が一部変更されているので注意してください。たとえば GC が Managed Heap という名称に変更されました。



▲図 3.14 2021 以降のシンプルビュー

2. Detailed ビュー

Detailed ビューは図 3.15 のような表示になります。

The screenshot shows the Unity Profiler's Detailed View. It lists objects and their memory usage:

Name	Memory	Ref count	Referenced By:
Others (81)	44.9 MB		
Assets (420)	1.5 MB		
AudioManager (1)	1.1 MB		
Shader (7)	233.3 KB		
Hidden/Internal-GUIRoundedRectWithColorPerBorder	45.2 KB	2	
Skybox/Procedural	44.8 KB	1	
Hidden/Internal-GUIRoundedRect	40.5 KB	2	
Sprites/Default	32.2 KB	3	<ul style="list-style-type: none"> SplashScreen-Foreground(Material) GraphicsSettings(GraphicsSettings) Sprites-Default(Material)
Hidden/Internal-GUITextureClip	31.9 KB	3	
Hidden/Internal-GUITexture	24.0 KB	3	
Hidden/BlitCopy	14.6 KB	2	
MonoScript (396)	135.9 KB		

▲図 3.15 Detailed ビュー

この表示結果は「Take Sample」ボタンを押下することで、その時点でのメモリスナップショットを取得できます。Simple ビューと違いリアルタイム更新ではないため、表示を更新したい場合は再度 Take Sample を行う必要があります。

図 3.15 の右側に「Referenced By」という項目があります。これは現在選択中のオブジェクトを参照しているオブジェクトが表示されます。リークしているアセットが存在する場合、オブジェクトの参照元の情報が解決の糸口になるでしょう。この表示は「Gather object references」を有効にしている場合のみ表示されます。この機能を有効にすると、Take Sample 時の処理時間が伸びますが、基本的には有効にしておくとよいでしょう。

Referenced By には `ManagedStaticReferences()` という表記を見かけることがあります。これは何かしらの static なオブジェクトから参照されているということです。プロジェクトに精通している方なら、この情報だけである程度目星がつくかもしれません。そうでない場合は「3.5 Heap Explorer」を使用することを推奨します。

Detailed ビューのヘッダー項目は、見たとおりの意味なのでここでは解説を行いません。操作方法は「3.1.2 CPU Usage」の「1. Hierarchy ビュー」と同じで、ヘッダーごとのソート機能があり、項目が階層表示になっています。ここでは Name 項目に表示されるトップノードについて解説を行います。

▼表 3.3 Detailed のトップノード

名前	説明
Assets	シーンに含まれていないロードしたアセット。
Not Saved	コードによって実行時に生成されたアセット。 たとえば、 <code>new Material()</code> など、コードから生成したオブジェクトのこと。
Scene Memory	ロードしたシーンに含まれているアセット。
Others	上記以外のオブジェクトで、Unity がさまざまなシステムで使用するものへの割り当て。

トップノードの中でも Others の中に記載されている項目は普段馴染みがないものばかりでしょう。その中でも知っておくとよいものを次に取り上げます。

System.ExecutableAndDLLs

バイナリや DLL などに使用された割り当て量を示します。プラットフォームや端末によって取得できない場合があり、その場合は 0B として扱われます。共通フレームワークを使用している他のアプリケーションと共有している場合もあり、プロジェクトに対するメモリ負荷は記載されている数値ほど大きくはありません。この項目の削減に躍起になるより、Asset の改善をするほうがよいでしょう。削減するには、DLL や不要な Script を削減するのが効果的です。もっとも簡単な方法としては、Stripping Level の変更です。ただし、実行時に型やメソッドが欠落するリスクがあるのでデバッグは入念に行いましょう。

SerializedFile

AssetBundle 内のオブジェクトのテーブルや、型情報となる Type Tree などのメタ情報を示します。これは AssetBundle.Unload(true or false) で解放することができます。もっとも効率が良いのはアセットロード後、このメタ情報のみ解放する Unload(false) を行うことですが、解放タイミングやリソースの参照管理をシビアに行わないと、リソースを二重ロードしたり、容易にメモリリークを引き起こすので注意してください。

PersistentManager.Remapper

メモリ上とディスク上のオブジェクトの関連性を管理しています。Remapper はメモリプールを利用し、足りなくなると倍々に拡張していき減少することはありません。拡張されすぎないように気をつけましょう。具体的には AssetBundle を大量にロードすると、マッピング領域が足りず拡張されます。そのため同時にロードされるファイル数を削減するために、不要な AssetBundle をアンロードするのがよいでしょう。また、1 つの AssetBundle にその場で不要なアセットが大量に含まれている場合は分割するのがよいでしょう。

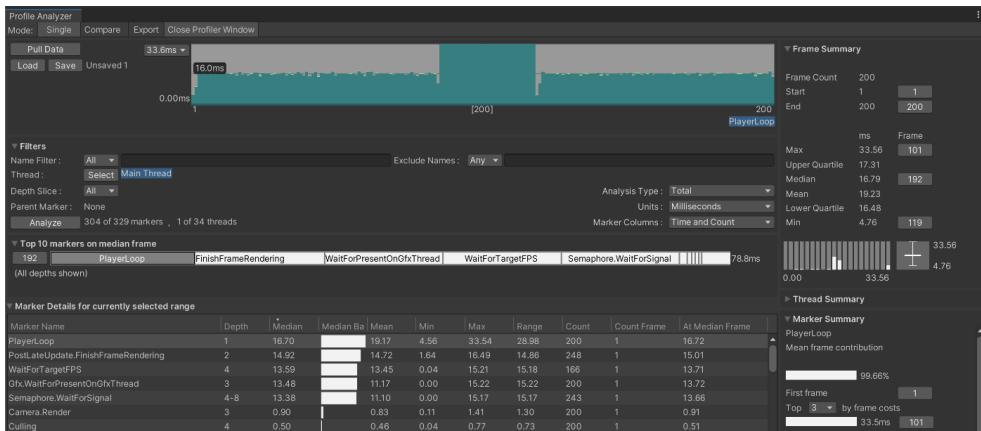
最後に、ここまで紹介してきた内容から Detailed ビューの利用するケースをまとめます。

- 特定タイミングでのメモリの詳細な把握、チューニング
 - 不要なアセットがないか、想定外なものがないか
- メモリリークの調査

3.2 Profile Analyzer

3.2 Profile Analyzer

Profile Analyzer は Profiler の CPU Usage で取得したデータをより細かく分析するためのツールです。Unity Profiler では 1 フレーム単位のデータしか見られなかったのに対して、Profile Analyzer は指定したフレームの区間をもとに平均値や中央値、最小、最大値を取得できます。フレームごとにばらつきのあるデータを適切に扱えるため、最適化を行った際に改善効果をより明確に示せるようになるでしょう。また CPU Usage では出来なかった計測データ同士の比較機能もあるため、最適化の結果を比較・可視化するのに非常に便利なツールです。

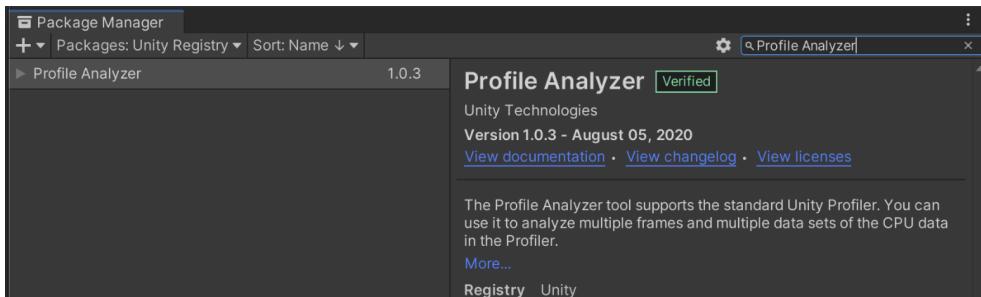


▲図 3.16 Profile Analyzer

3.2.1 導入方法

このツールは Package Manager からインストールが可能です。Unity が公式でサポートしているので、Packages を Unity Registry に変更し、検索ボックスに「Profile」と入力すると表示されます。インストール後「Window -> Analysis -> Profile Analyzer」でツールが起動できます。

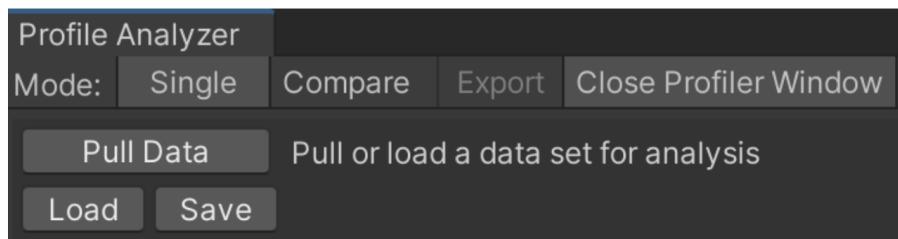
第3章 プロファイリングツール



▲図 3.17 PackageManager からのインストール

3.2.2 操作方法

Profile Analyzer は起動直後は図 3.18 のようになっています。



▲図 3.18 起動直後

機能として「Single」と「Compare」の2つのモードがあります。Single モードは1つの計測データを分析する場合に使用し、Compare モードは2つの計測データを比較する場合に利用します。

「Pull Data」は Unity Profiler で計測したデータを解析させ、結果を表示できます。事前に Unity Profiler で計測をしておきましょう。

「Save」「Load」は Profile Analyzer で分析したデータの保存・読み込みができます。もちろん Unity Profiler のデータだけを保持しておいても問題はありません。その場合は Unity Profiler でデータをロードし、Profile Analyzer で毎回 Pull Data を行う必要があります。その手順が面倒な場合に専用のデータとして保存しておくのがよいでしょう。

3.2.3 解析結果 (Single モード)

解析結果の画面は次のような構成になっています。ここではマーカーという単語が出てきますが、処理の名称（メソッド名）を指しています。

- ・分析区間の設定画面
- ・表示項目のフィルター入力画面
- ・マーカーの中央値 Top10
- ・マーカーの分析結果
- ・フレームのサマリ
- ・スレッドのサマリ
- ・選択中マーカーのサマリ

それぞれの表示画面について見ていきましょう。

1. 分析区間の設定画面

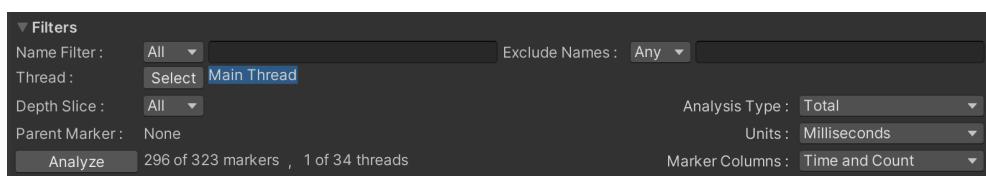
各フレームの処理時間が表示されており、初期状態では全フレームが選択された状態になっています。フレーム区間を図 3.19 のようにドラッグで変更できるので、必要であれば調整しましょう。



▲図 3.19 フレーム区間の指定

2. フィルター入力画面

フィルター入力画面は分析結果のフィルタリングができます。



▲図 3.20 フィルター入力画面

それぞれの項目は次のようにになります。

▼表 3.4 Filters の項目

項目名	説明
Name Filter	検索したい処理名でフィルターします。
Exclude Filter	検索から除外したい処理名でフィルターします。
Thread	選択対象のスレッドが分析結果に表示されます。 他スレッドの情報が必要であれば追加しましょう。
Depth Slice	CPU Usage で紹介した Hierarchy 表示の階層数です。 たとえば Depth が 3 の場合は、階層が 3 つ目のものが表示されます。
Analysis Type	CPU Usage ヘッダー項目にあった Total と Self の切り替えができます。
Units	時間表示をミリ秒かマイクロ秒に変更できます。
Marker Columns	分析結果のヘッダー表示を変更できます。

Depth Slice が All の場合、PlayerLoop というトップノードが表示されたり、同じ処理の階層違いが表示されるため、見づらいときがあります。その場合、Depth を 2~3 ぐらいに固定し、レンダリングやアニメーション、物理演算などのサブシステムが表示されるくらいに設定するとよいでしょう。

3. マーカーの中央値 Top10

この画面は各マーカーの処理時間を中央値で並び替え、上位 10 個のマーカーのみを示したものです。上位 10 個のマーカーがそれぞれどのくらい処理時間を占めているのかが一目でわかります。



▲図 3.21 マーカーの中央値 Top10

4. マーカーの分析結果

それぞれのマーカーの分析結果が表示されています。Marker Name に記載されている処理名と、Median (中央値)、Mean (平均値) の値から改善すべき処理を分析するのがよいでしょう。ヘッダー項目にマウスポインターを合わせると、その項目の説明が表示されるので、内容が分からぬ場合は参照してみてください。

3.2 Profile Analyzer

Marker Name	Depth	Media	Media	Mean	Min	Max	Range	Count	Count Fr	At Median F
PlayerLoop	1	16.71		19.24	4.56	33.54	28.98	199	1	16.71
PostLateUpdate.FinishFrameRendering	2	14.93		14.80	1.73	16.49	14.76	247	1	14.83
WaitForTargetFPS	4	13.59		13.53	0.04	15.21	15.18	165	1	13.35
Gfx.WaitForPresentOnGfxThread	3	13.49		11.22	0.00	15.22	15.22	199	1	13.35
Semaphore.WaitForSignal	4-8	13.38		11.16	0.00	15.17	15.17	242	1	13.28
Camera.Render	3	0.90		0.82	0.11	1.41	1.30	199	1	1.16
Culling	4	0.50		0.46	0.04	0.77	0.73	199	1	0.69
SceneCulling	5	0.35		0.33	0.02	0.62	0.60	199	1	0.52
PostLateUpdate.ProfilerEndFrame	2-3	0.30		0.30	0.13	1.10	0.97	247	1	0.33
Profiler.FlushCounters	3	0.28		0.26	0.03	1.10	1.07	199	1	0.33
PrepareSceneNodes	6	0.27		0.25	0.01	0.53	0.52	199	1	0.43
Profiler.FlushMemoryCounters	4	0.22		0.20	0.02	1.08	1.06	199	1	0.23

▲図 3.22 各処理の分析結果

平均値と中央値

平均値はすべての値を足し合わせ、データ数で割った値です。それに対して中央値は、ソートしたデータの中央に位置する値のことを指します。データが偶数個の場合、中央にある前後のデータから平均値を取ります。

平均値は極端に値が離れているデータがあると影響を受けやすい性質があります。頻繁にスパイクが発生していたり、サンプリング数が十分でない場合、中央値を参考にする方がよい場合があるでしょう。

図 3.23 は中央値と平均値で差が大きくなる例です。

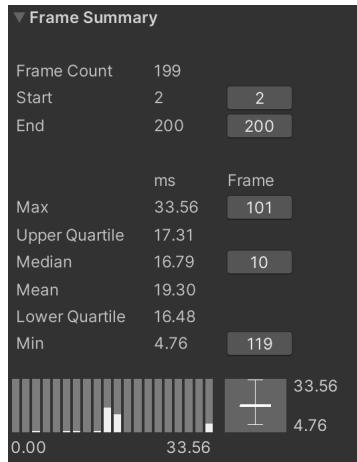


▲図 3.23 中央値と平均値

この 2 つの値の特徴を知った上でデータを分析しましょう。

5. フレームのサマリ

この画面には計測したデータのフレーム統計情報が表示されています。



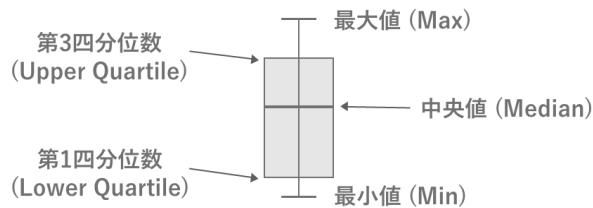
▲図 3.24 フレームサマリ画面

分析しているフレームの区間情報や、値のばらつき具合を箱ひげ図（Boxplot）やヒストグラムを用いて表示しています。箱ひげ図は四分位数を理解する必要があります。四分位数とはデータをソートした状態で、表 3.5 のように値を定めたものです。

▼表 3.5 四分位数

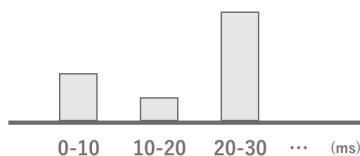
名前	説明
最小値 (Min)	最小の値
第 1 四分位数 (Lower Quartile)	最小値から 25% の位置にある値
中央値 (Median)	最小値から 50% の位置にある値
第 3 四分位数 (Upper Quartile)	最小値から 75% の位置にある値
最大値 (Max)	最大の値

25% から 75% の区間を箱で囲ったものを箱ひげグラフと言います。



▲図 3.25 箱ひげグラフ

ヒストグラムは横軸に処理時間、縦軸にデータ数を示しており、こちらもデータの分布を見るのに便利です。フレームサマリではカーソルを合わせることで、区間とフレーム数を確認できます。

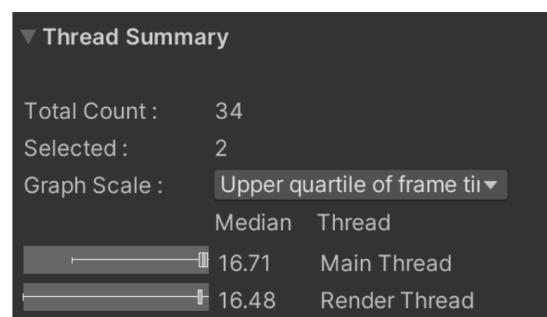


▲図 3.26 ヒストグラム

これらの図の見方を理解した上で、特徴を分析するとよいでしょう。

6. スレッドのサマリ

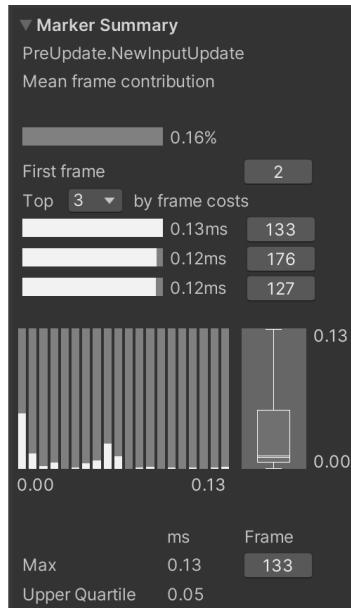
この画面は選択しているスレッドの統計情報が表示されています。各スレッドに関する箱ひげ図を見ることができます。



▲図 3.27 フレームサマリ画面

7. 選択中マーカーのサマリ

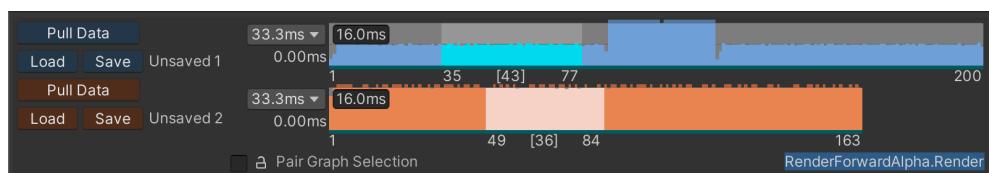
「4. マーカーの分析結果」画面で選択しているマーカーに関するサマリです。選択中のマーカーに関する処理時間が箱ひげ図やヒストグラムで表されます。



▲図 3.28 選択中マーカーのサマリ

3.2.4 解析結果 (Compare モード)

このモードでは2つのデータを比較できます。上下それぞれのデータごとに解析する区間の設定ができます。



▲図 3.29 比較データの設定

3.3 Frame Debugger

画面の使い方は Single モードとほとんど変わりませんが、図 3.30 のように「Left」と「Right」という単語が色々な画面に出てきます。

Marker Name	Left Median	<	>	Right Median	Diff	Abs Diff	Co
Gfx.WaitForGfxCommandsFromMainThread	0.66			31.85	31.19	31.19	76
WaitForTargetFPS	13.90			32.44	18.55	18.55	43
Semaphore.WaitForSignal	14.64			31.85	17.21	17.21	121
PlayerLoop	16.70			33.29	16.59	16.59	43
PostLateUpdate.FinishFrameRendering	15.18		■	0.40	-14.78	14.78	43
Gfx.PresentFrame	14.29		■	0.10	-14.20	14.20	43

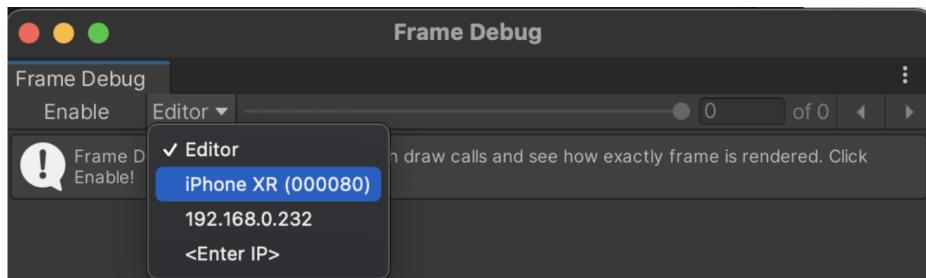
▲図 3.30 マーカーの比較

これはどちらのデータであるかを示しており、図 3.29 で表示されている色と一致しています。Left は上部のデータ、Right は下部のデータです。このモードを利用することで、チューニング結果の良し悪しを簡単に分析することができるでしょう。

3.3 Frame Debugger

Frame Debugger は現在表示されている画面がどのような流れでレンダリングされているかを分析できるツールです。このツールはデフォルトでエディターにインストールされており「Window -> Analysis -> Frame Debugger」で開きます。

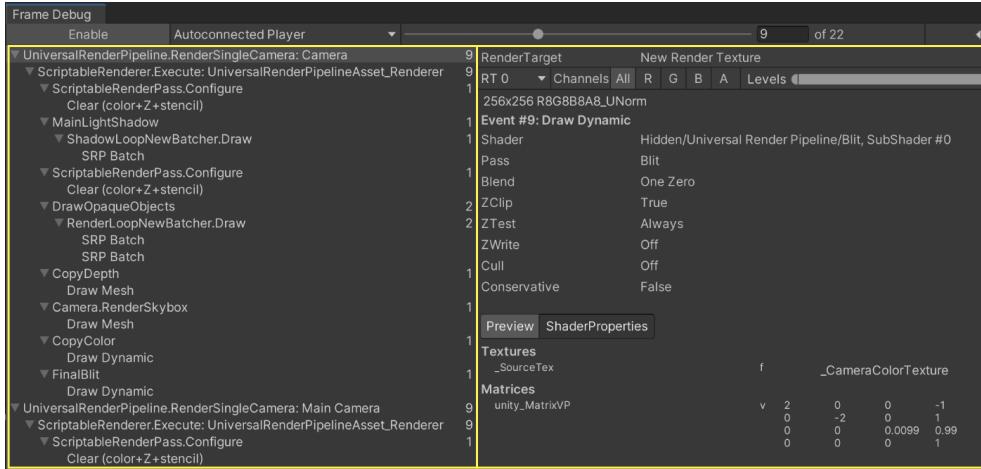
エディターや実機での使用が可能です。実機で使用する際は Unity Profiler と同様「Development Build」でビルドされたバイナリが必要です。アプリケーションを起動し、デバイス接続先を選択した上で「Enable」を押下すると描画命令が表示されます。



▲図 3.31 FrameDebugger 接続画面

3.3.1 分析画面

「Enable」を押下すると次のような画面になります。



▲図 3.32 FrameDebugger キャプチャ

左枠は1項目が1つの描画命令になっており、上から順番に発行された命令が並んでいます。右枠は描画命令に関する詳細情報です。どのShaderがどのようなプロパティと共に処理されたかがわかります。この画面を見ながら次のことを意識して分析を行います。

- 不要な命令がないか
- 描画バッティングが適切に効いているか、まとめられないか
- 描画先の解像度が高すぎないか
- 意図しないShaderを使用していないか

3.3.2 詳細画面

前節で紹介した図3.32の右枠の内容を詳細に解説します。

操作パネル

まずは上段部分にある操作パネルについてです。

3.3 Frame Debugger



▲図 3.33 上段の操作パネル

RT0 と記載されている部分は複数のレンダリングターゲットが存在する場合に変更できます。とくにマルチレンダーターゲットを利用している際に、それぞれのレンダリング状態を確認する際に重宝するでしょう。Channels は、RGBA すべてか、いずれかの 1 チャンネルのみで表示するかの変更ができます。Levels はスライダーになっており、描画結果の明るさを調節できます。たとえばアンビエントや間接照明など描画結果が暗い際に、明るさを調整して見やすくするのに便利です。

描画概要

この領域では描画先の解像度やフォーマットの情報がわかります。明らかに解像度の高い描画先があればすぐに気付くことができるでしょう。他にも、使用している Shader 名、Cull などの Pass 設定、使用しているキーワードなどがわかります。下部に記載されている「Why this~」という文章は、描画のバッチングができなかった理由が書かれています。図 3.34 の場合、最初の描画コールを選択しているのでバッチングできないと記載されています。このように細かく原因が記載されているので、バッチングを工夫したい場合、この情報を頼りに調整するとよいでしょう。

```
824x1210 B10G11R11_UFloatPack32
Event #13: SRP Batch
Draw Calls           3
Shader              Universal Render Pipeline/Lit, SubShader #0
Pass                ForwardLit (UniversalForward)
Keywords            _MAIN_LIGHT_SHADOWS
Blend               One Zero
ZClip               True
ZTest               LessEqual
ZWrite              On
Cull                Back
Conservative        False

Why this draw call can't be batched with the previous one
SRP: First call from ScriptableRenderLoopJob
```

▲図 3.34 中段の描画概要

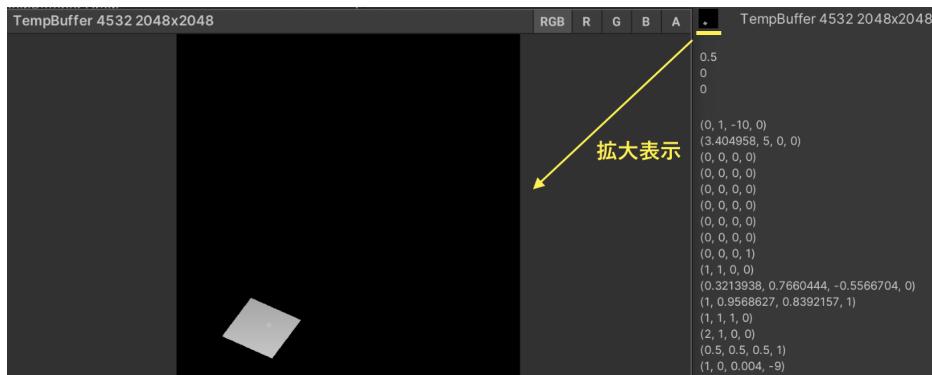
Shader プロパティの詳細情報

この領域は描画している Shader のプロパティ情報が記載されています。デバッグ時に重宝します。

Preview				ShaderProperties	
Textures					
unity_SpecCube0		f			
_BaseMap		f		UnityWhite	
_MainLightShadowmapTexture		f		TempBuffer 1	2048x2048
FLOATS					
_Smoothness		f	0.5		
_Metallic		f	0		
_Surface		f	0		
Vectors					
_WorldSpaceCameraPos		vf	(0, 1, -10, 0)		
unity_OrthoParams		v	(2.310268, 5, 0, 0)		

▲図 3.35 下段の Shader プロパティの詳細情報

プロパティ情報に表示されている Texture2D がどのような状態になっているか詳細に確認したいときがあります。その際、Mac の場合は Command キー (Windows の場合 Control キー) を押下しながら画像をクリックすると拡大できます。



▲図 3.36 Texture2D のプレビューを拡大する

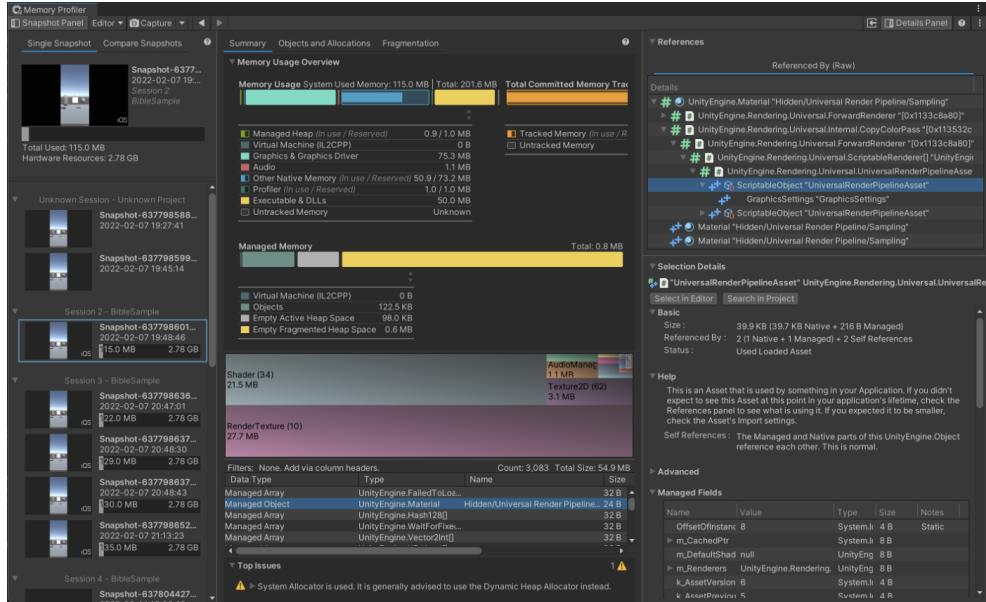
3.4 Memory Profiler

3.4 Memory Profiler

Memory Profiler は Preview Package として Unity が公式で提供しているツールです。Unity Profiler の Memory モジュールと比較して、主に以下のような点で優れています。

- ・キャプチャしたデータがスクリーンショット付きでローカルに保存される
- ・各カテゴリーのメモリ占有量がビジュアライズされ、わかりやすい
- ・データの比較が可能

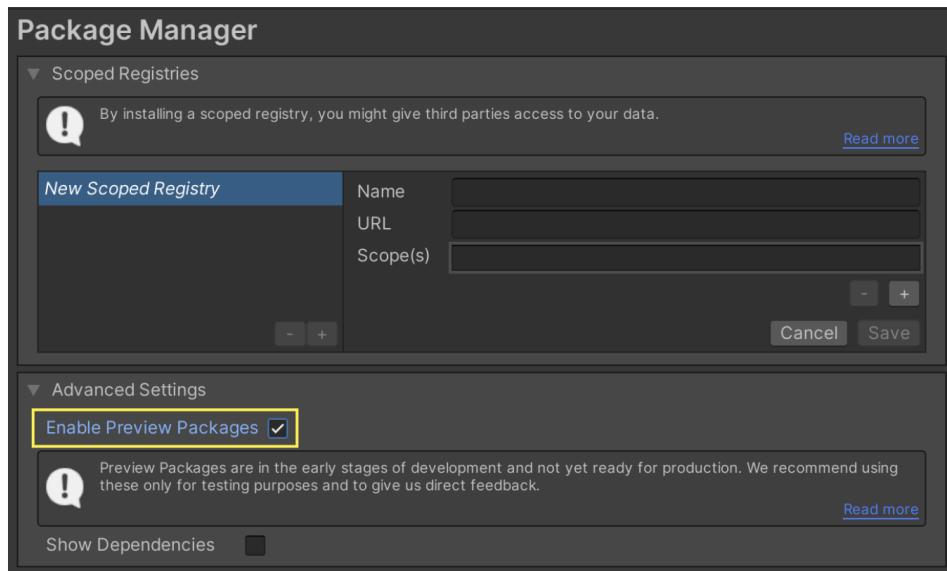
Memory Profiler は v0.4 以降とそれ未満で UI が大きく変わりました。本書籍では執筆時点で最新版となる v0.5 を使用します。また v0.4 以降のバージョンで、すべての機能を利用する場合 Unity 2020.3.12f1 以降のバージョンが必要になります。さらに v0.4 と v0.5 は一見同じように見えますが、v0.5 で大幅に機能がアップデートされました。とくにオブジェクトの参照関係がとても追いやすくなったので、基本的には v0.5 以降の利用を推奨します。



▲図 3.37 Memory Profiler

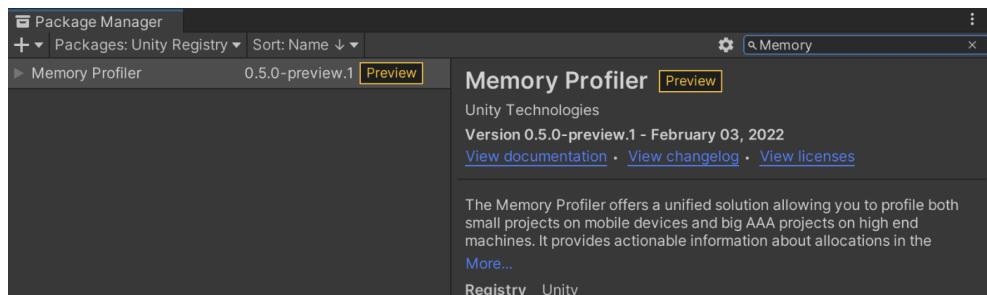
3.4.1 導入方法

Unity 2020 では、Preview 版のパッケージは「Project Settings -> Package Manager」から「Enable Preview Packages」を有効にする必要があります。



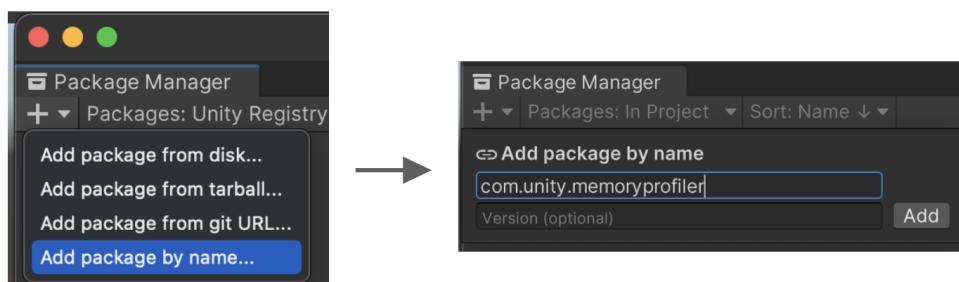
▲図 3.38 Preview Packages の有効化

その後、Unity Registry の Package から Memory Profiler をインストールします。インストール後「Window -> Analysis -> Memory Profiler」でツールが起動できます。



▲図 3.39 PackageManager からインストール

また Unity 2021 以降では、パッケージの追加方法が変更されています。追加するには「Add Package by Name」を押下し「com.unity.memoryprofiler」と入力する必要があります。



▲図 3.40 2021 以降の追加方法

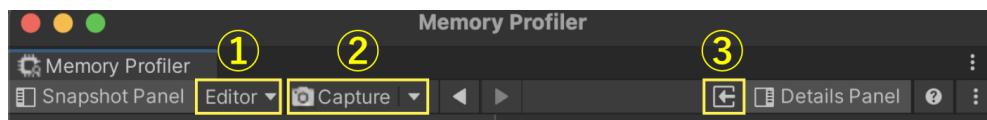
3.4.2 操作方法

Memory Profiler は大きく分けて 4 つの要素から構成されています。

- ツールバー
- Snapshot Panel
- 計測結果
- Detail Panel

それぞれの領域について解説を行います。

1. ツールバー

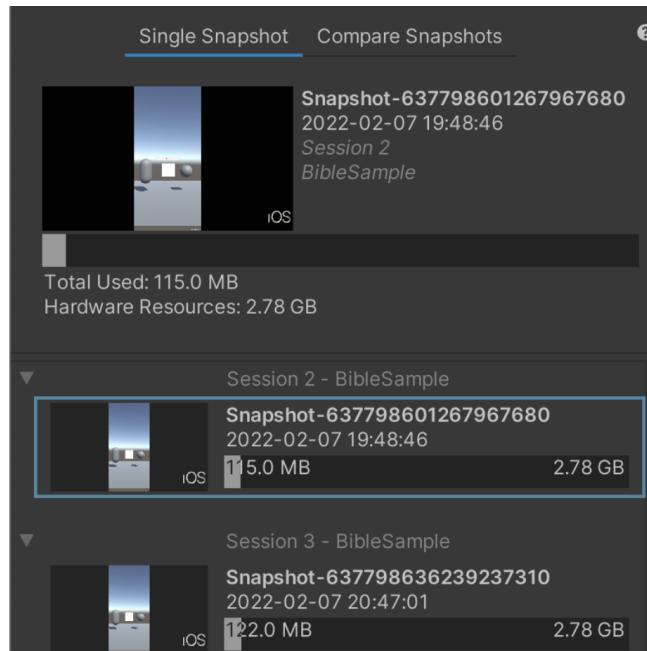


▲図 3.41 ツールバー領域

図3.41はHeaderのキャプチャを示しています。①のボタンは計測対象を選択できます。②のボタンは押下するとその時点のメモリを計測します。オプションで計測対象をNative Objectだけにしたり、スクリーンショットを無効にできます。基本デフォルト設定で問題ないでしょう。③のボタンは押下すると計測済みのデータを読み込みができます。その他「Snapshot Panel」や「Detail Panel」を押下することで、左右にある情報パネルの表示／非表示が可能です。Tree Mapの表示だけを見たい場合に非表示にすることがよいでしょう。また「?」を押下することで公式ドキュメントを開くことができます。

計測に関する注意点が1つあります。それは計測時に必要なメモリが新たに確保され、以降解放されないことです。ただし無限には増え続けず、何度か計測するといずれ落ち着きます。計測時のメモリ確保量はプロジェクトの複雑度によるでしょう。この前提を知らないと、膨れ上がったメモリ使用量を見てリークがあるかもと勘違いしてしまうので気をつけましょう。

2. Snapshot Panel

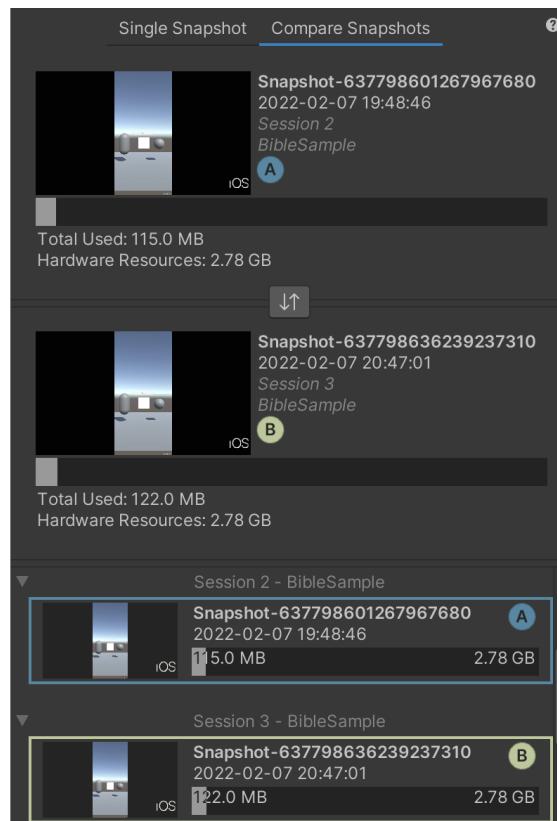


▲図3.42 Snapshot Panel (Single)

3.4 Memory Profiler

Snapshot Panel には計測したデータが表示され、どのデータを見るか選択できます。このデータはアプリを起動してから終了するまでを 1 セッションとし、セッションごとにまとめられます。また計測データを右クリックすることで削除や名称のリネームが可能です。

上部に「Single Snapshot」「Compare Snapshots」があります。Compare Snapshots を押下すると、計測データを比較する UI に表示が変わります。

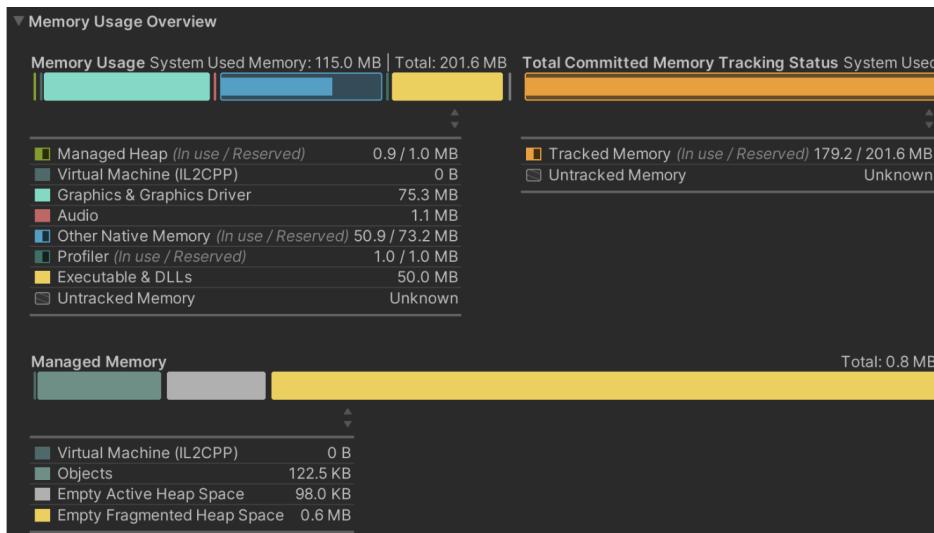


▲図 3.43 Snapshot Panel (Comapre)

「A」が Single Snapshot で選択したデータ、「B」が Comapre Snapshots で選択したデータです。入れ替えボタンを押下することで「A」と「B」を入れ替えることができる、わざわざ Single Snapshot 画面に戻らずに切り替えも可能です。

3. 計測結果

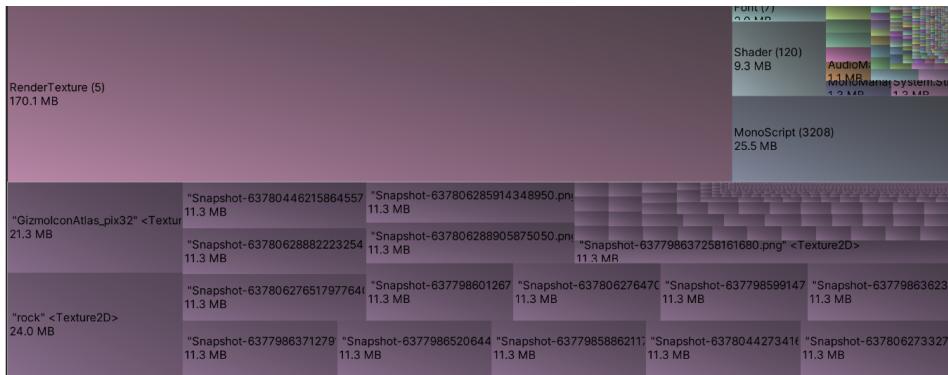
計測結果は「Summary」「Objects and Allocations」「Fragmentation」の3つのタブがあります。本節ではよく使用する Summary を解説し、その他は補足として簡単に取り上げます。Summary の画面上部は Memory Usage Overview と呼ばれる領域で、現在のメモリの概要が表示されています。項目を押下すると Detail Panel に解説が表示されるので、分からぬ項目は確認するとよいでしょう。



▲図 3.44 Memory Usage Overview

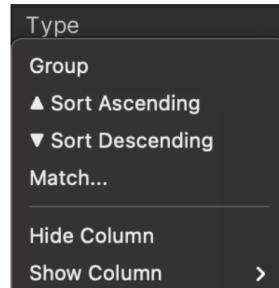
次に画面中部は Tree Map と呼ばれています。オブジェクトのカテゴリごとにメモリ使用量をグラフィカルに表示します。各カテゴリを選択すると、カテゴリ内のオブジェクトを確認できます。図 3.45 では Texture2D のカテゴリを選択している状態です。

3.4 Memory Profiler



▲図 3.45 Tree Map

そして画面下部は Tree Map Table と呼ばれています。ここではオブジェクトの一覧がテーブル形式で並んでいます。表示項目は Tree Map Table のヘッダーを押下することで、グループ化やソート、フィルターを行うことができます。



▲図 3.46 ヘッダー操作

とくに Type をグループ化することで分析しやすくなるため、積極的に利用しましょう。

Data Type	[Type]	Name	Size	Referenced By
Native Obj...	► AudioListener (2)		432 B	2
Native Obj...	AudioManager (1)	AudioManager	1.1 MB	0
Native Obj...	BoxCollider (1)	Cube	256 B	1
Native Obj...	BuildSettings (1)	BuildSettings	0.6 KB	0
	► Camera (2)		8.5 KB	4

▲図 3.47 Type によるグループ化

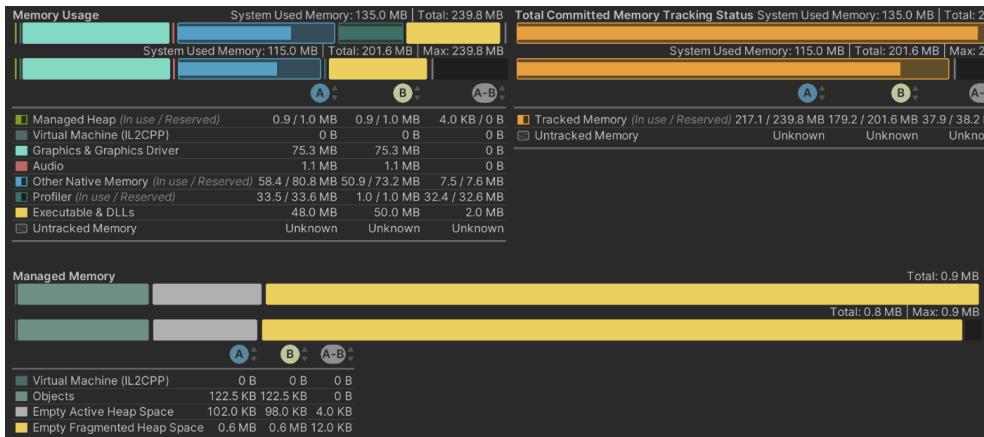
第3章 プロファイリングツール

また Tree Map でカテゴリを選択した場合は、そのカテゴリ内のオブジェクトだけを表示するフィルターが自動で設定されます。

Filters:		Count: 34 Total Size: 21.5 MB		
X 'Type' Is Shader				
X 'Data Type' Is Native Object				
X Sort▼ 'Size'				
Data Type	Type	Name	Size	Referenced By
Native Obj...	Shader	Hidden/Universal Render Pipeline...	19.1 MB	4
Native Obj...	Shader	Hidden/Universal Render Pipeline...	417.1 KB	3
Native Obj...	Shader	Hidden/Universal Render Pipeline...	217.8 KB	4
Native Obj...	Shader	Hidden/Universal Render Pipeline...	217.2 KB	4
Native Obj...	Shader	Hidden/Universal Render Pipeline...	211.3 KB	4
Native Obj...	Shader	Hidden/Universal Render Pipeline...	195.4 KB	4
		Universal Render Pipeline/Lit	181.4 KB	1

▲図 3.48 フィルターの自動設定

最後に Compare Snapshots を使用した場合の UI の変化について説明します。Memory Usage Overview にはそれぞれのオブジェクトの差分が表示されます。



▲図 3.49 Compare Snapshots の Memory Usage Overview

また Tree Map Table には Header に Diff という項目が追加されます。Diff には次のようなタイプがあります。

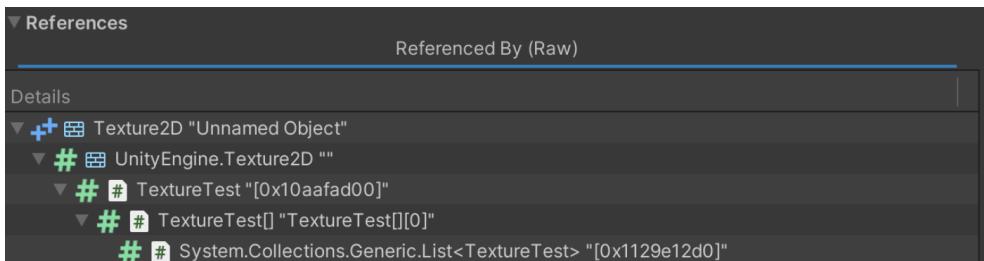
▼表 3.6 Tree Map Table (Compare)

Diff	説明
Same	A、B 同一オブジェクト
Not in A (Deleted)	A にあるが、B にはないオブジェクト
Not in B (New)	A はないが、B にあるオブジェクト

これらの情報を見ながらメモリが増減しているかを確認できます。

4. Detail Panel

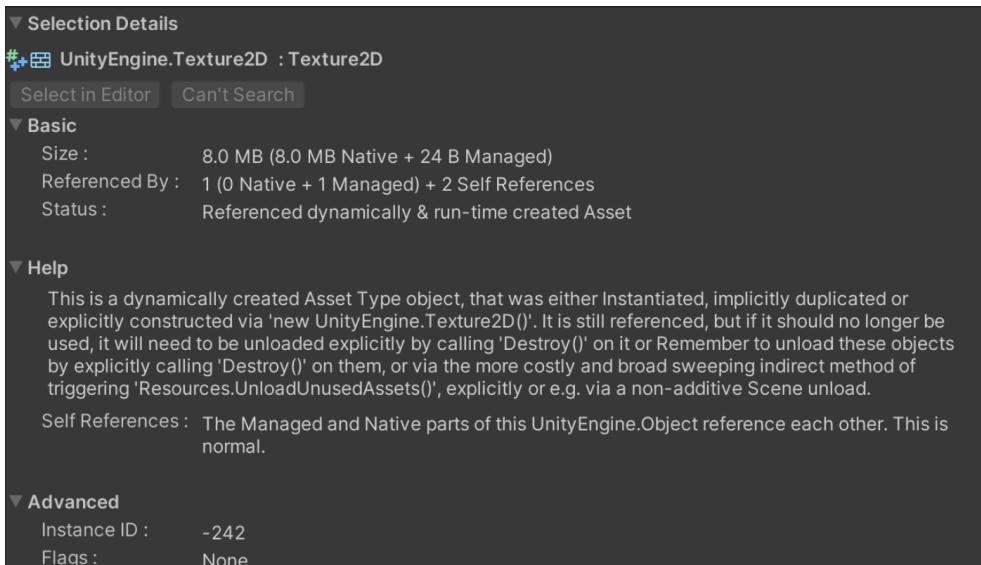
このパネルは選択しているオブジェクトの参照関係を追跡したい際に利用します。この Referenced By を確認することで、参照を握り続けている原因を把握できるでしょう。



▲図 3.50 Referenced By

下部の Selection Details という箇所には、オブジェクトの詳細な情報が載っています。その中でも「Help」という項目には、解放するためのアドバイスなどが記載されています。何をするべきか分からぬときは一読するとよいかもしれません。

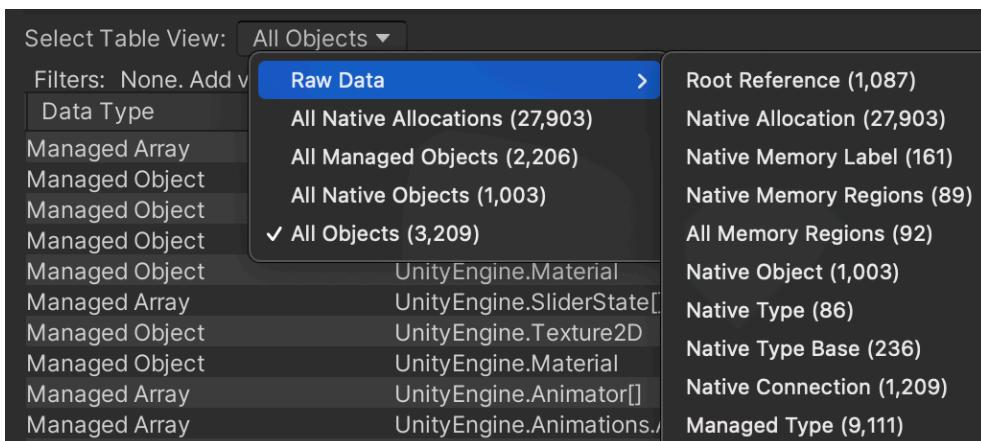
第3章 プロファイリングツール



▲図 3.51 Selection Details

補足：Summary 以外の計測結果について

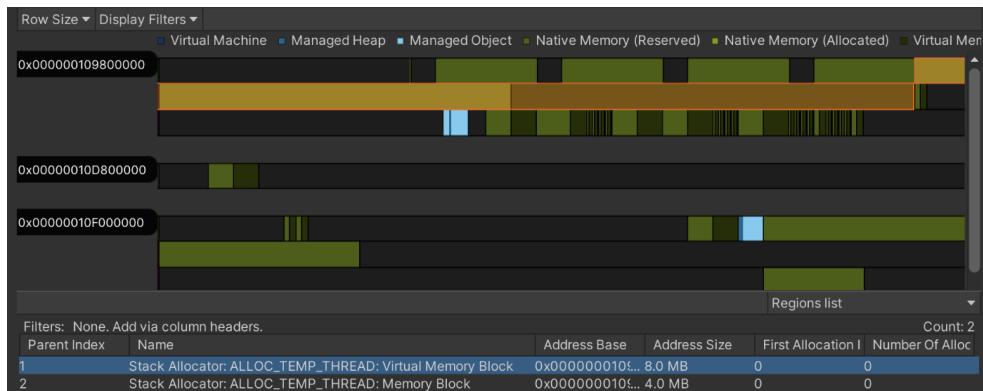
「Objects and Allocations」は Summary と違い、アロケーションなどのより細かい内容がテーブル形式で確認できます。



▲図 3.52 Table View の指定

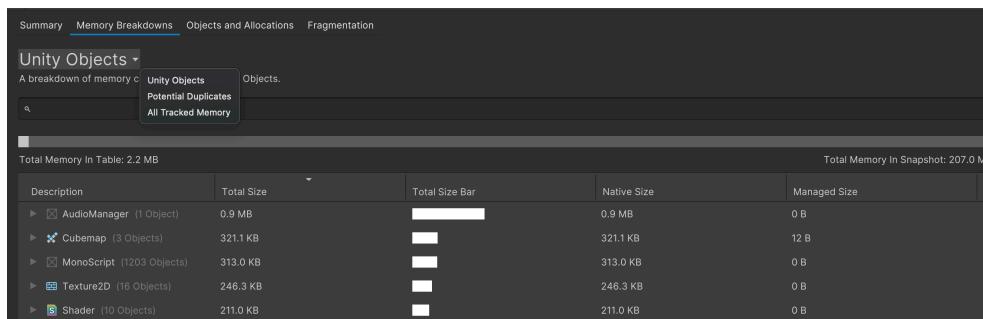
3.4 Memory Profiler

「Fragmentation」は仮想メモリの状況を可視化してくれるため、断片化の調査に利用できます。メモリアドレスなど直感的でない情報が多いため利用する難易度は高いでしょう。



▲図 3.53 Fragmentation

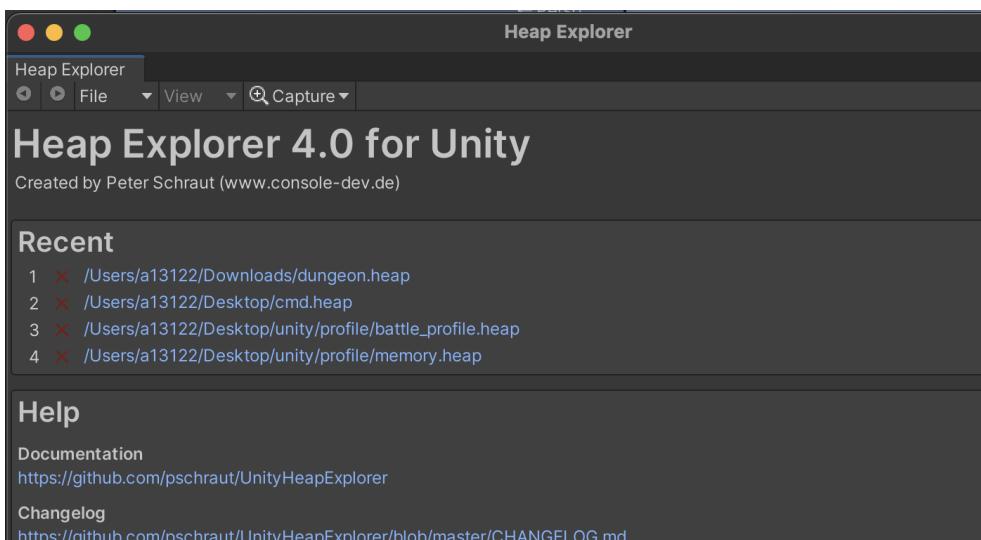
また Memory Profiler の v0.6 から「Memory Breakdowns」という新機能が追加されました。Unity 2022.1 以降が必須となります。TreeMap をリスト表示で閲覧できたり、Unity Subsystems などのオブジェクト情報も見ることが可能になりました。他にも、重複の可能性があるオブジェクトを確認できるようになりました。



▲図 3.54 Memory Breakdowns

3.5 Heap Explorer

Heap Explorer は個人開発者である Peter77^{*1}氏のオープンソースのツールです。こちらは Memory Profiler と同様にメモリの調査を行う際によく利用するツールです。Memory Profiler は 0.4 以前のバージョンだと、参照がリスト形式で表示されないため追跡するのに非常に労力がかかりました。0.5 以降で改善されたものの、対応していない Unity バージョンを利用している方もいるでしょう。その際の代替ツールとしてはまだまだ利用価値が高いので今回取り上げたいと思います。



▲図 3.55 Heap Explorer

3.5.1 導入方法

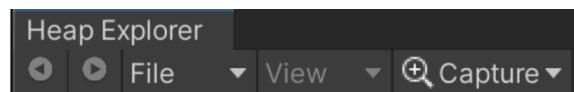
本ツールは GitHub のリポジトリ^{*2}に記載されている Package URL's をコピーし、Package Manager の Add Package from Git url から追加します。インストール後「Window -> Analysis -> Memory Profiler」でツールが起動できます。

^{*1} <https://github.com/pschraut>

^{*2} <https://github.com/pschraut/UnityHeapExplorer>

3.5.2 操作方法

Heap Explorer のツールバーは次のようにになっています。



▲図 3.56 Heap Explorer のツールバー

左右の矢印

操作の戻る、進むができます。とくに参照を追跡する場合に便利です。

File

計測ファイルの保存、読み込みが可能です。.heap という拡張子で保存されます。

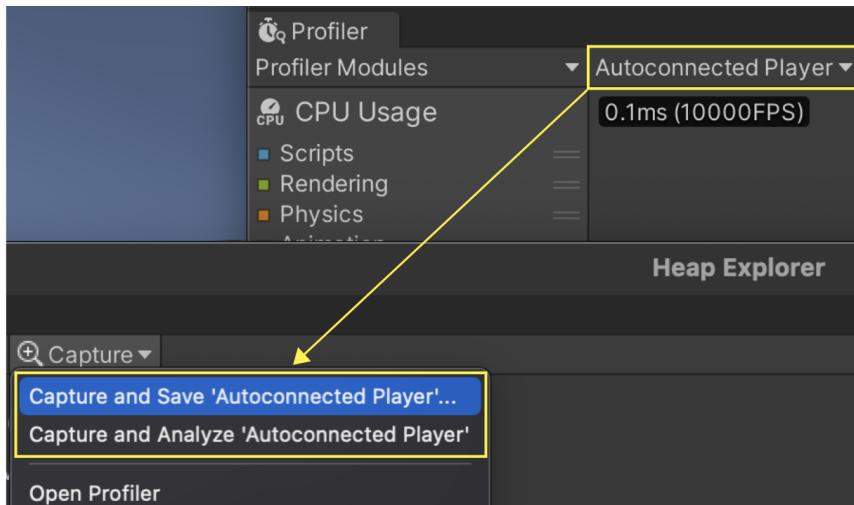
View

表示画面の切り替えができます。色々な種類があるので、興味があればドキュメントを見てください。

Capture

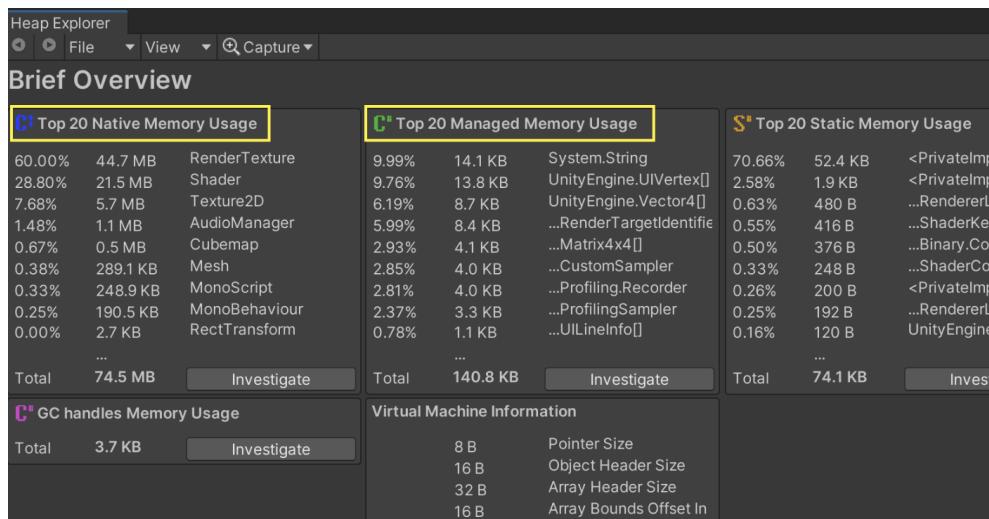
計測を行います。ただし計測対象は **Heap Explorer** 上では変更ができません。対象の切り替えは Unity Profiler などの Unity が提供するツール上で変更する必要があります。また Save はファイルに保存した上で結果が表示され、Analyze は保存せずに表示します。なお、Memory Profiler と同様に、計測する際に確保されるメモリは解放されないので注意してください。

第3章 プロファイリングツール



▲図 3.57 計測対象の切り替え

計測結果の画面は次のようにになります。この画面を Overview と呼びます。



▲図 3.58 Heap Explorer の計測結果 (Overview)

Overview の中でとくに気にするべきカテゴリは緑の線が引いてある Native Memory Usage と Managed Memory Usage です。Investigate のボタンを押下するとそれぞれ

のカテゴリの詳細を確認できます。

以降ではカテゴリ詳細の表示について、重要な部分に絞って解説します。

1. Object

Native Memory を Investigate した場合、C++ Objects がこの領域に表示されます。Managed Memory の場合は C# Objects がこの領域に表示されます。

C++ Objects							
Type	Name	Size	Count	DDoL	Persistent	Address	InstanceId
▶ RenderTexture		44.7 MB	5				
▶ Shader		21.5 MB	35				
▼ Texture2D		5.7 MB	63				
Texture2D	rock	2.7 MB		False	True	0x151D107E0	4114
(Texture2D)	AreaTex	350.5 KB		False	True	0x14FF75F50	4066
(Texture2D)	Large01	256.5 KB		False	True	0x15199E440	4092
(Texture2D)	Medium01	256.5 KB		False	True	0x14FF73010	4010

▲図 3.59 Object 表示領域

ヘッダーにいくつか見慣れない項目があるので補足しておきます。

DDoL

Don't Destroy On Load の略です。シーンを遷移しても破棄しないオブジェクトとして指定されているかがわかります。

Persistent

永続オブジェクトかどうかです。起動時に Unity が自動生成するものがこれに当たります。

これ以降に紹介する表示領域は、図 3.59 のオブジェクトを選択することで更新されます。

2. Referenced by

対象オブジェクトの参照元となるオブジェクトが表示されます。

第3章 プロファイリングツール

Referenced by 2 object(s)		
Type	C++ Name	Address
PostProcessData	PostProcessData	0x11413E660
...Texture2D	Medium06	0x1112B97C0

▲図 3.60 Referenced by

3. References to

対象オブジェクトの参照先となるオブジェクトが表示されます。

References to 1 object(s)		
Type	C++ Name	Address
GCHandle	UnityEngine.Texture2D	0x1112B97C0

▲図 3.61 References to

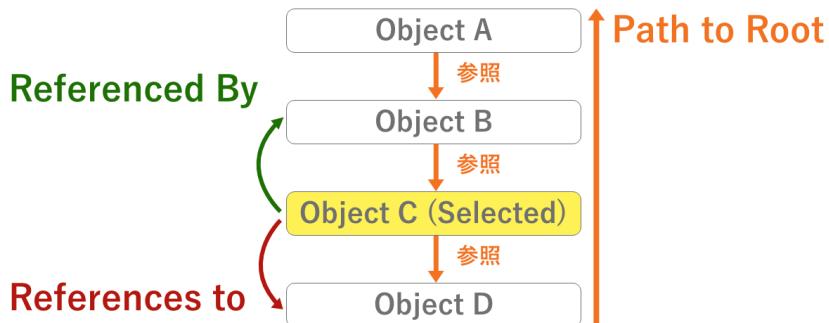
4. Path to Root

対象オブジェクトを参照しているルートオブジェクトが表示されます。メモリリークを調査する際、何が参照を握っているのか把握できるので重宝します。

2 Path(s) to Root				
Type	C++ Name	Depth	Address	
UnityEngine.Rendering.RenderPipelineManager	static s_CurrentPipelineAsset	6		
...Universal.UniversalRenderPipelineAsset	UniversalRenderPipelineAsset		0x106F1B	
UniversalRenderPipelineAsset	UniversalRenderPipelineAsset		0x1519CF	
ForwardRendererData	UniversalRenderPipelineAsset_Renderer		0x1519D2	
PostProcessData	PostProcessData		0x1519D2	
Texture2D	Large01		0x15199E	
GraphicsSettings	GraphicsSettings	5	0x14FF50	
UniversalRenderPipelineAsset	UniversalRenderPipelineAsset		0x1519CF	
ForwardRendererData	UniversalRenderPipelineAsset_Renderer		0x1519D2	
PostProcessData	PostProcessData		0x1519D2	
Texture2D	Large01		0x15199E	

▲図 3.62 Path to Root

今までの項目をまとめると次のようなイメージになります。



▲図 3.63 参照のイメージ

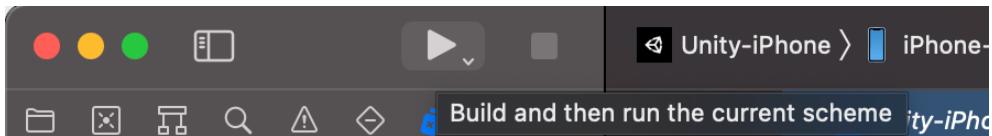
ここまで紹介したように Heap Explorer にはメモリリークやメモリ調査に必要な機能が一式揃っています。動作も軽快なのでぜひこのツールの利用も検討してみてください。気に入ったら感謝の意を込めて Star もつけると尚よいでしょう。

3.6 Xcode

Xcode は Apple が提供する統合開発環境ツールです。Unity でターゲットプラットフォームを iOS に設定した際、そのビルド成果物が Xcode プロジェクトになります。Unity で計測するよりも正確な数値が取れるので、厳密な検証をする際は Xcode を利用することを推奨します。本節では Debug Navigator、GPU Frame Capture、Memory Graph の 3 つのプロファイリングツールに触れていきます。

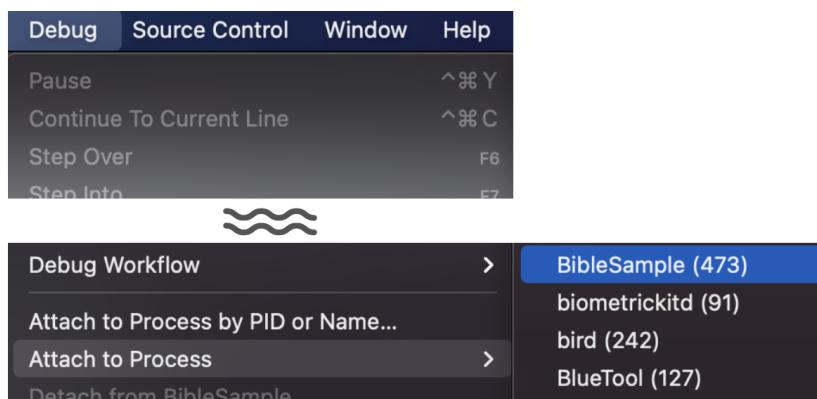
3.6.1 プロファイル方法

Xcode からプロファイルする方法は 2 通りあります。1 つ目は Xcode から直接端末にビルドし、アプリケーションを実行する方法です。図 3.64 に示すように実行ボタンを押下するだけでプロファイルが開始されます。ビルドを行う際の証明書などの設定は本書では割愛します。



▲図 3.64 Xcode の実行ボタン

2つ目は実行中のアプリケーションを Xcode のデバッガーにアタッチする方法です。これはアプリを実行した後に、Xcode メニューの「Debug -> Attach to Process」から実行中のプロセスを選択することでプロファイルができます。ただし、ビルト時の証明書が開発者用 (Apple Development) でないといけません。Ad Hoc や Enterprise の証明書ではアタッチできないので注意してください。

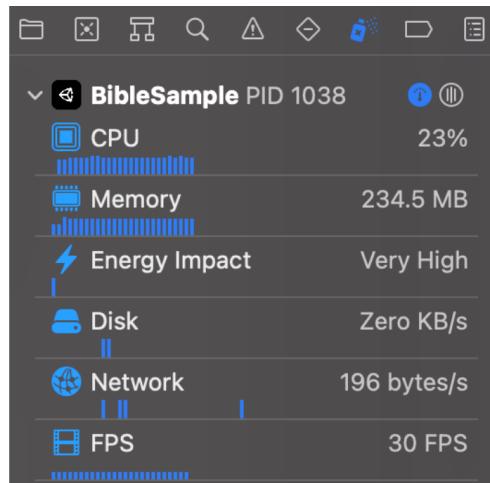


▲図 3.65 Xcode のデバッガーアタッチ

3.6.2 Debug Navigator

Debug Navigator は Xcode からアプリケーションを実行するだけで、CPU や Memory などのデバッグゲージが確認できます。アプリケーション実行後に図 3.66 のスプレーマークを押下することで 6 つの項目が表示されます。もしくは Xcode メニューの「View -> Navigators -> Debug」から開くことも可能です。以降ではそれぞれの項目について解説します。

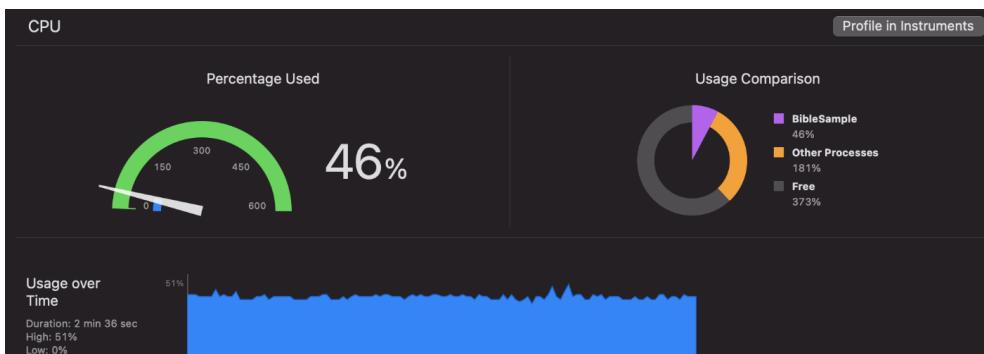
3.6 Xcode



▲図 3.66 Debug Navigator の選択

1. CPU ゲージ

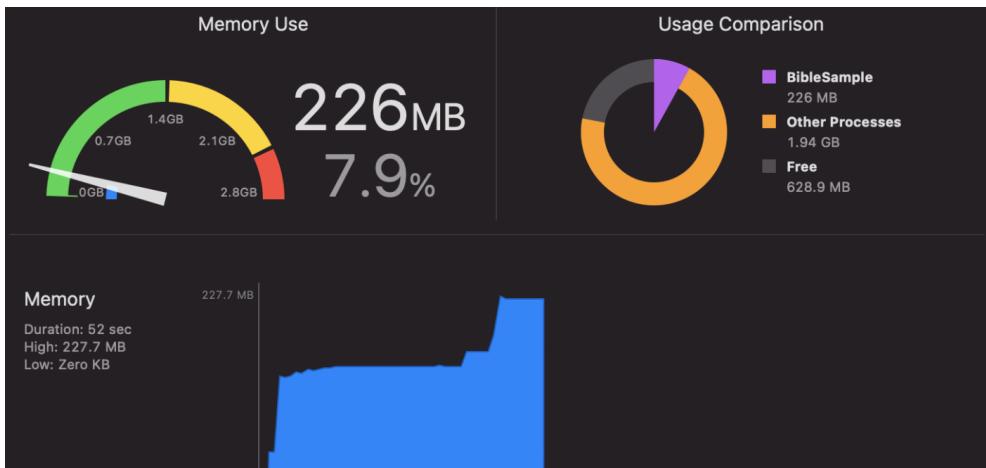
どれぐらい CPU を使用しているかを見ることができます。また各スレッドごとの使用率も見ることができます。



▲図 3.67 CPU ゲージ

2. Memory ゲージ

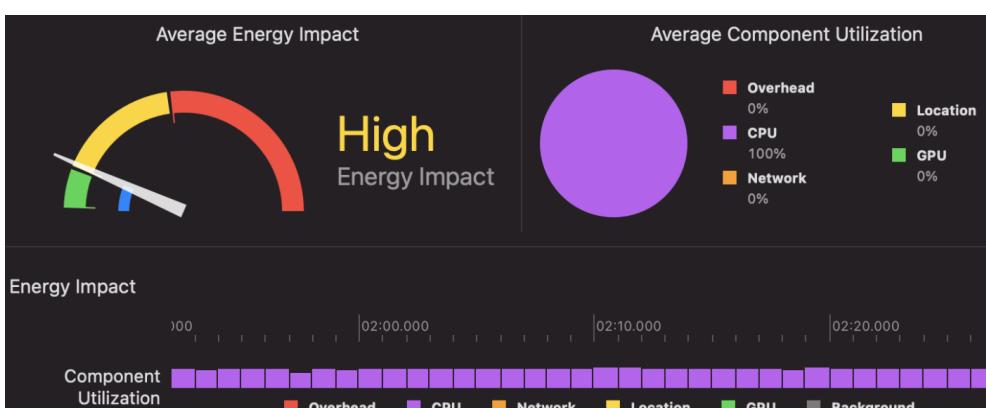
メモリ消費量の概要を見ることができます。内訳などの細かい分析はできません。



▲図 3.68 Memory ゲージ

3. Energy ゲージ

電力消費に関する概要を見ることができます。CPU、GPU、Network などの使用率の内訳が把握できます。

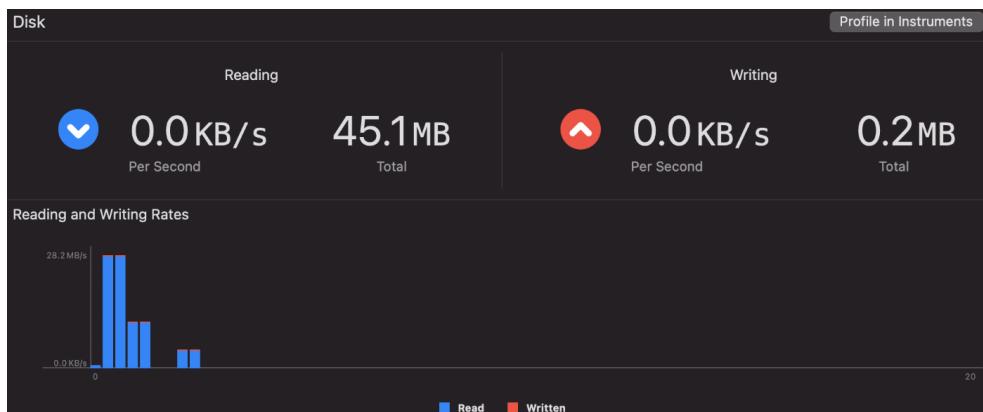


▲図 3.69 Energy ゲージ

3.6 Xcode

4. Disk ゲージ

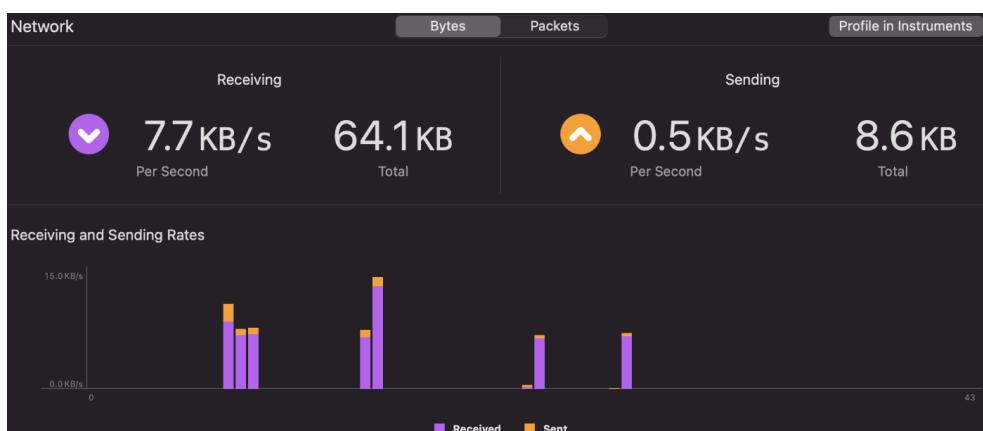
File I/O の概要を見ることができます。予期せぬタイミングでファイルの読み書きが行われていないかのチェックに役立つでしょう。



▲図 3.70 Disk ゲージ

5. Network ゲージ

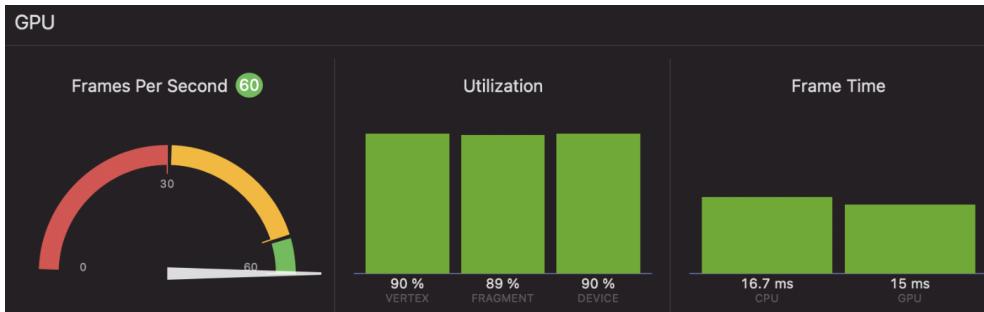
ネットワーク通信の概要を見ることができます。Disk と同じく予期しない通信をしていないかなどのチェックに役立つでしょう。



▲図 3.71 Network ゲージ

6. FPS ゲージ

このゲージはデフォルトでは表示されていません。「3.6.3 GPU Frame Capture」で解説する GPU Frame Capture を有効化すると表示されます。FPS はもちろんのこと、シェーダーステージの使用率、各 CPU や GPU の処理時間を確認できます。



▲図 3.72 FPS ゲージ

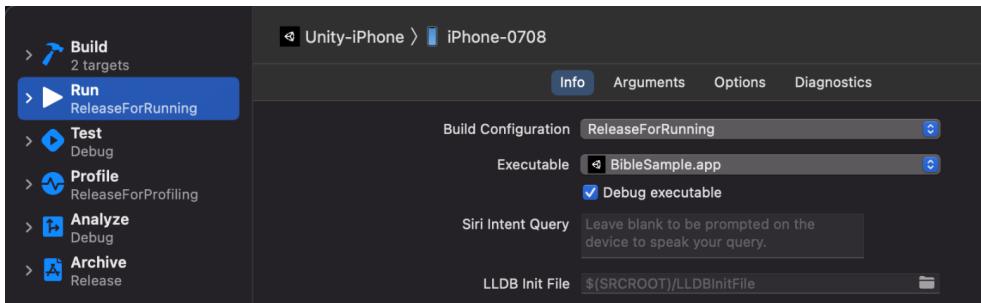
3.6.3 GPU Frame Capture

GPU Frame Capture とは Xcode 上でフレームデバッグができるツールのことです。Unity の Frame Debugger と同様に、レンダリングが完了するまでの過程が確認できます。Unity と比べシェーダーの各ステージでの情報量が多いため、ボトルネックの分析や改善に役立つかかもしれません。以下では使用方法について解説します。

1. 準備

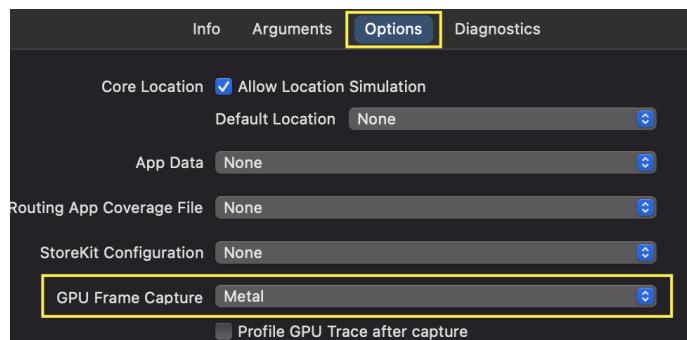
Xcode で GPU Frame Capture を有効にするにはスキームの編集が必要になります。初めに「Product -> Scheme -> Edit Scheme」でスキーム編集画面を開きましょう。

3.6 Xcode



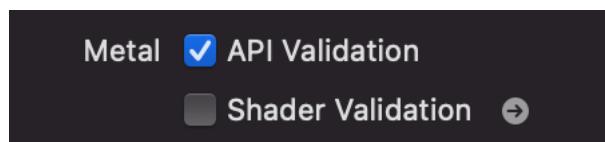
▲図 3.73 スキーム編集画面

次に「Options」タブから GPU Frame Capture を「Metal」に変更しましょう。



▲図 3.74 GPU Frame Capture の有効化

最後に「Diagnostics」タブから Metal の「Api Validation」を有効にしましょう。



▲図 3.75 Api Validation の有効化

2. キャプチャ

実行中にデバッグバーからカメラマークを押下することでキャプチャが行われます。シーンの複雑度によりますが、初回のキャプチャには時間がかかるので気長に待ちましょう。なお、Xcode13以降の場合はMetalのアイコンに変更されています。

Xcode12以前



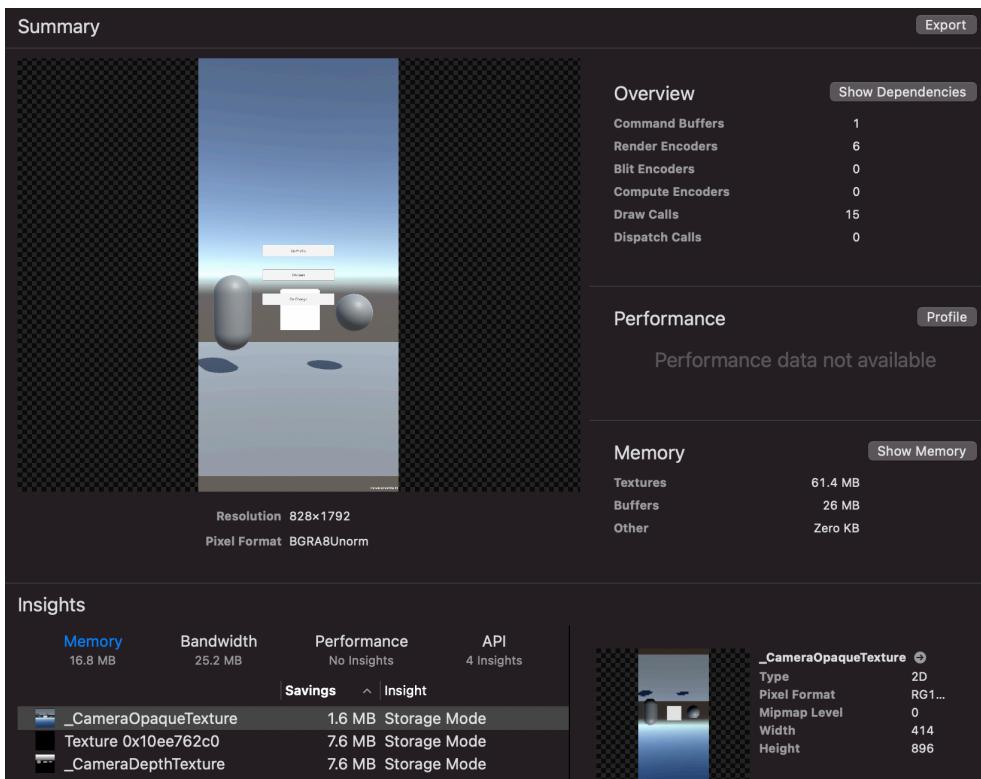
Xcode13以降



▲図 3.76 GPU Frame Capture ボタン

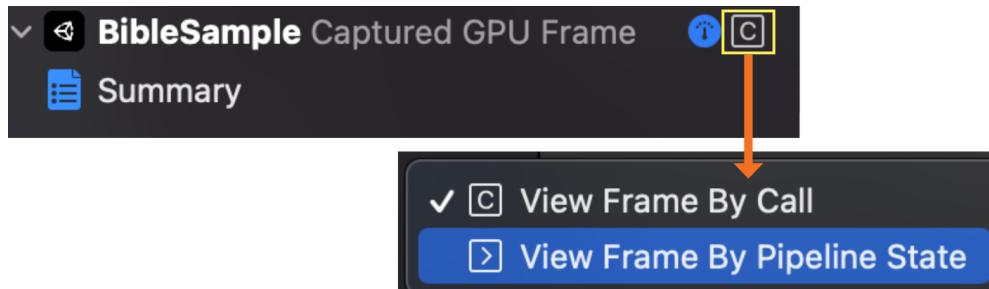
キャプチャが完了すると次のようなサマリー画面が表示されます。

3.6 Xcode



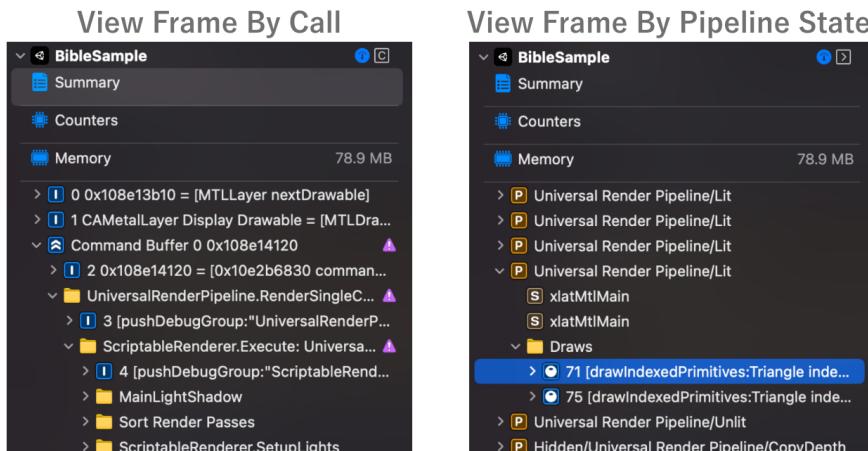
▲図 3.77 サマリー画面

このサマリー画面からは描画の依存関係やメモリなどの詳細を確認できる画面へ遷移できます。また Navigator エリアには描画に関するコマンドが表示されています。この表示方法は「View Frame By Call」と「View Frame By Pipeline State」があります。



▲図 3.78 表示方法の変更

By Call 表示では描画コマンドがすべて呼び出された順に並びます。これは描画の前準備になるバッファーの設定なども含まれるため非常に多くのコマンドが並ぶことになります。一方、By Pipeline State は各シェーダーで描画されたジオメトリに関する描画コマンドだけが並びます。どういったことを調査するかで表示を切り替えるとよいでしょう。



▲図 3.79 表示の違い

Navigator エリアに並ぶ描画コマンドは押下することで、そのコマンドに使用されているプロパティ情報が確認できます。プロパティ情報にはテクスチャ、バッファー、サンプラー、シェーダーの関数、ジオメトリなどがあります。それぞれのプロパティはダブルクリックすることで詳細を確認できます。たとえばシェーダーコードそのものや、サンプラーが Repeat なのか Clamp なのかといったことまで把握できます。

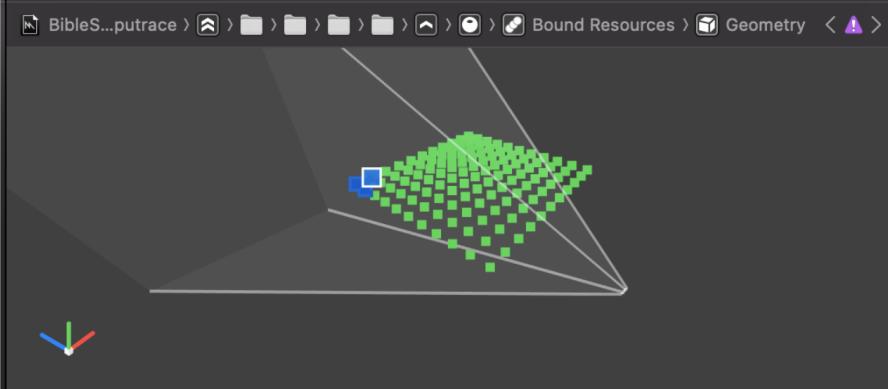
3.6 Xcode

Vertex					
Plane	Index	1 KB	Offset: 0x0	VGlobals	Read
ScratchBuffer0_1	Buffer 0	4 MB	Offset: 0x240	UnityPerDraw	Read
ScratchBuffer0_1	Buffer 1	4 MB	Offset: 0x403c0	UnityPerMaterial	Read
Compute_0	Buffer 2	80 bytes	Offset: 0x0	MainLightShadows	Read
ScratchBuffer0_1	Buffer 3	4 MB	Offset: 0x4c0	vertexBuffer.0	Read
Plane	Buffer 4	6 KB	Offset: 0x0		
Geometry	Post Vertex Tran...				
Vertex Attributes	Vertex Attributes				
xlatMtlMain (Universal Re...	Vertex Function	Library 0x11aaa6a20 [Jus...]			
Fragment					
Texture 0x114305d50	Texture 0	(Cube) 128 × 128	RGBA8Unorm	unity_SpecCube0	Read
UnityWhite	Texture 1	4 × 4	RGBA8Unorm	_BaseMap	Read
TempBuffer 1 2048x2048	Texture 2	2048 × 2048	Depth32Float	_MainLightShado...	Read

▲図 3.80 描画コマンド詳細

またジオメトリのプロパティは頂点情報がテーブル形式で表示されるだけでなく、カメラを動かして形状の確認も可能です。

BibleS...putrace > [] > [] > [] > [] > [] > [] > [] > [] Bound Resources > [] Geometry < [] >								
Index	in - ushort Vertex	in - float3 POSITION0			in - float3 NORMAL0			in TEX
		X	Y	Z	X	Y	Z	
0	9	-4.000	-1e-16	5.000	0.000	1.000	0.000	0.906
1	21	-5.000	-9e-17	4.000	0.000	1.000	0.000	1.000
2	10	-5.000	-1e-16	5.000	0.000	1.000	0.000	1.000
3	9	-4.000	-1e-16	5.000	0.000	1.000	0.000	0.906



▲図 3.81 ジオメトリビューウィンドウ

第3章 プロファイリングツール

次にサマリー画面の Performance 欄にある「Profile」について説明します。このボタンを押下するとより細かい解析を開始します。解析が終わると描画にかかった時間が Navigator エリアに表示されるようになります。

> Hidden/Universal Render Pipeline/Blit	541.92 µs
> Universal Render Pipeline/Lit	437.30 µs
> Skybox/Procedural	398.49 µs
> Hidden/Universal Render Pipeline/CopyDepth	296.52 µs

▲図 3.82 プロファイル後の表示

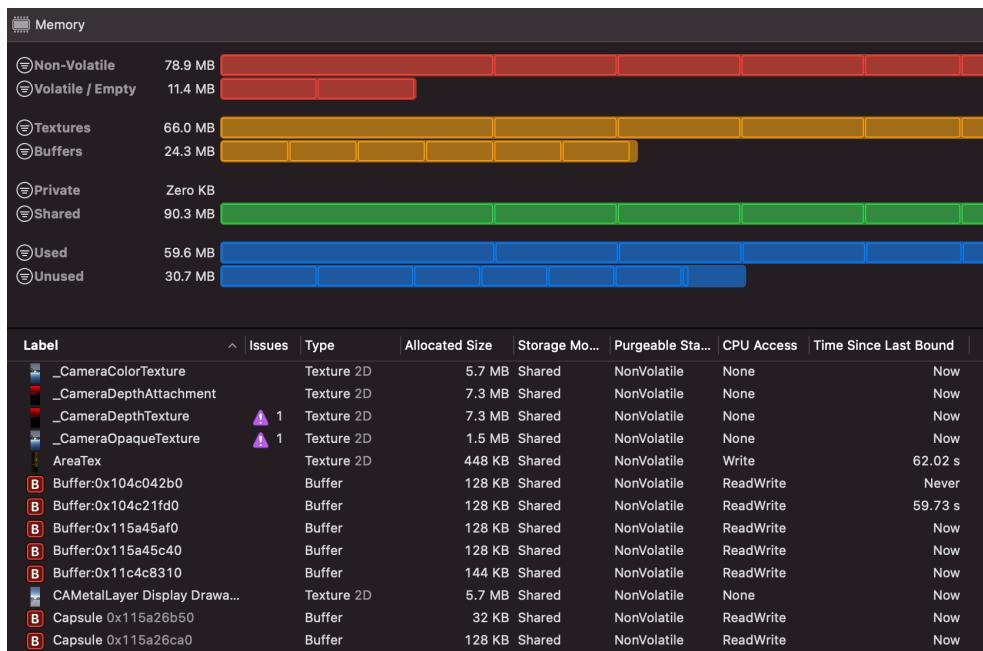
それだけでなく解析結果を Counters という画面でより詳細に確認ができます。この画面では各描画の Vertex や Rasterized、Fragment などの処理時間をグラフィカルに見ることができます。



▲図 3.83 Counters 画面

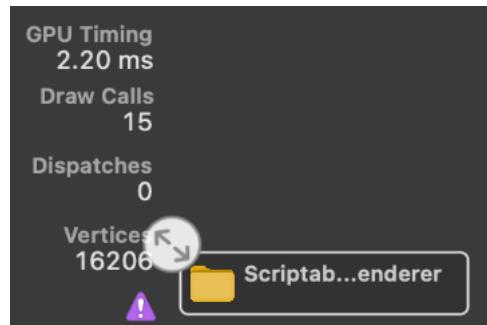
続いてサマリー画面の Memory 欄にある「Show Memory」について説明します。このボタンを押下すると GPU で使用しているリソースが確認できる画面へ遷移します。表示される情報は主にテクスチャとバッファーです。不要なものがないか確認するといいでしょう。

3.6 Xcode



▲図 3.84 GPU リソース確認画面

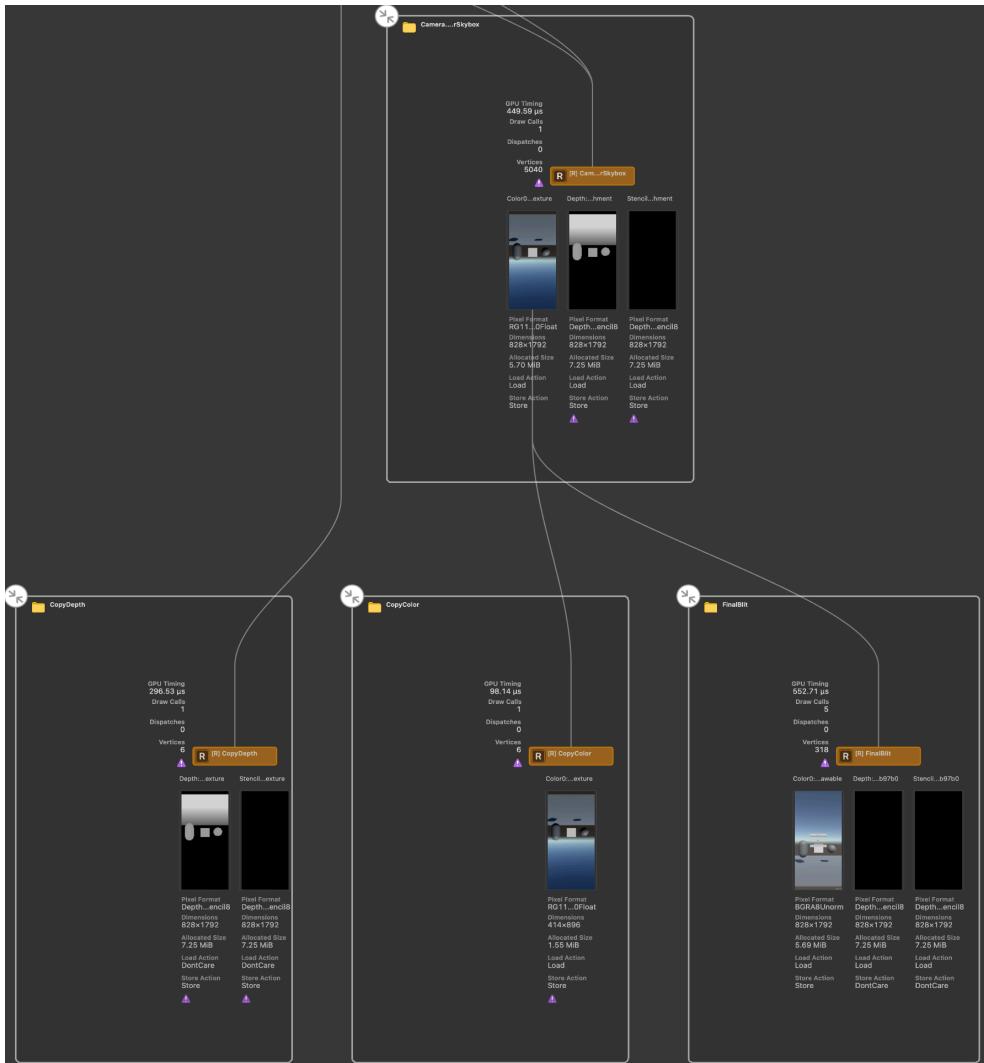
最後にサマリー画面の Overview にある「Show Dependencies」について説明します。このボタンを押下すると各レンダーパスの依存関係が表示されます。Dependency を見ていく際には「矢印が外向き」になっているボタン押下することで、その階層以下の依存関係をさらに開くことができます。



▲図 3.85 Dependency を開くボタン

第3章 プロファイリングツール

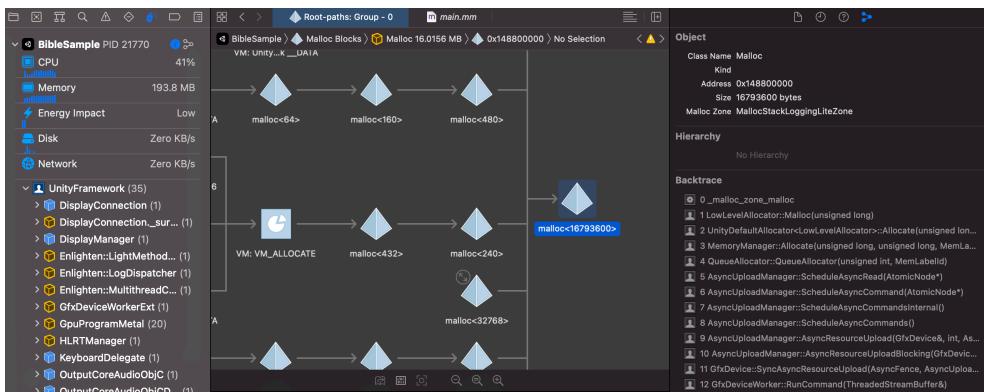
この画面はどの描画が何に依存しているかを見たいときに使用しましょう。



▲図 3.86 階層を開いた状態

3.6.4 Memory Graph

このツールではキャプチャタイミングでのメモリ状況を分析できます。左側の Navigator エリアにはインスタンスが表示され、そのインスタンスを選択することで参照関係がグラフで表示されます。右側の Inspector エリアにはそのインスタンスの詳細情報が表示されます。



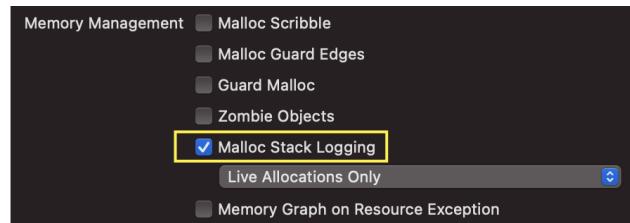
▲図 3.87 MemoryGraph キャプチャ画面

このツールではプラグインなど Unity 上では計測できないオブジェクトのメモリ使用量を調査をする場合に利用するとよいでしょう。以下では使用方法について解説します。

1. 事前準備

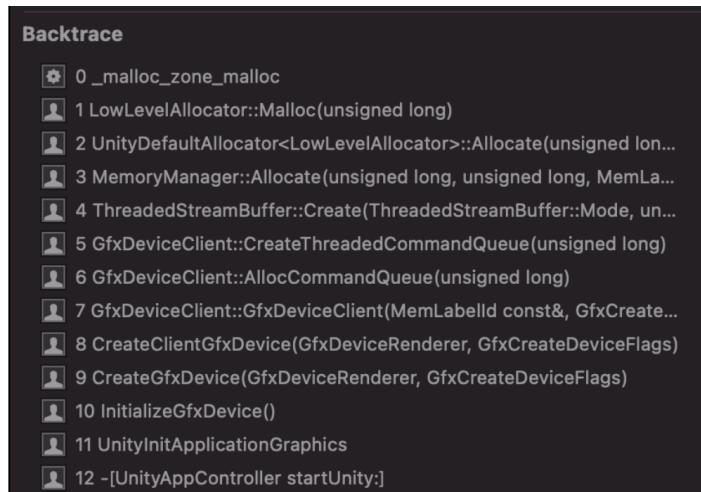
Memory Graph で有益な情報を得るためにスキームの編集を行う必要があります。「Product -> Scheme -> Edit Scheme」でスキーム編集画面を開きましょう。そして「Diagnostics」タブから「Malloc Stack Logging」を有効にしましょう。

第3章 プロファイリングツール



▲図 3.88 Malloc Stack Logging の有効化

こちらを有効にすることで Inspector に Backtrace が表示されるようになります。どのような流れでアロケーションされたかがわかります。



▲図 3.89 Backtrace の表示

2. キャプチャ

アプリケーション実行中にデバッグバーから枝のようなアイコンを押下することでキャプチャが行われます。



▲図 3.90 Memory Graph Capture ボタン

また Memory Graph は「File -> Export MemoryGraph」からファイルとして保存が可能です。このファイルに対して、vmmmap コマンドや heap コマンド、malloc_history コマンドを駆使してさらに深い調査をすることが可能です。興味があればぜひ調べてみてください。例として vmmmap コマンドの Summary 表示を紹介します。MemoryGraph だと掴みにくかった全体像を把握できます。

▼リスト 3.5 vmmmap summary コマンド

```
1: vmmmap --summary hoge.memgraph
```

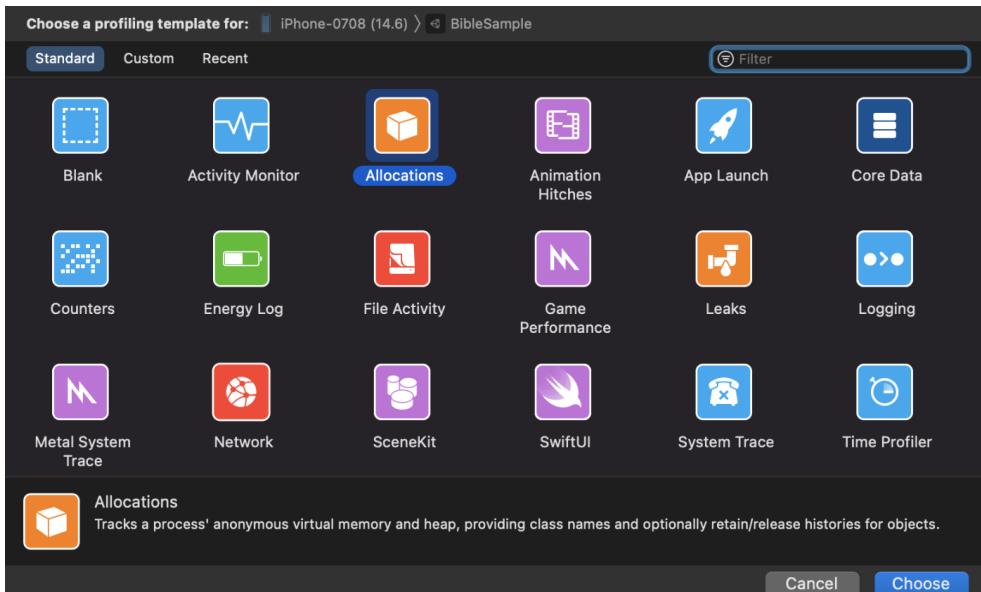
REGION TYPE	VIRTUAL SIZE	RESIDENT SIZE	DIRTY SIZE	SWAPPED SIZE	VOLATILE SIZE	NONVOL SIZE	EMPTY SIZE	REGION COUNT
<hr/>								
Activity Tracing	256K	32K	32K	0K	0K	32K	0K	1
CoreAnimation	32K	32K	32K	0K	0K	16K	0K	2
Foundation	16K	16K	16K	0K	0K	0K	0K	1
IOAccelerator	99.2M	78.4M	78.4M	0K	0K	78.4M	13.2M	98
IOKit	304K	304K	304K	0K	0K	0K	0K	19
IOSurface	17.1M	17.1M	17.1M	0K	0K	17.1M	0K	3
Image IO	16K	0K	0K	16K	0K	0K	0K	1
Kernel Alloc Once	32K	16K	16K	0K	0K	0K	0K	1
MALLOC guard page	192K	0K	0K	0K	0K	0K	0K	12
MALLOC metadata	336K	288K	288K	0K	0K	0K	0K	15
MALLOC_LARGE	69.8M	23.7M	23.7M	27.0M	0K	0K	0K	1215
MALLOC_LARGE metadata	256K	144K	144K	80K	0K	0K	0K	1
MALLOC_NANO	512.0M	304K	304K	32K	0K	0K	0K	1
MALLOC_SMALL	64.0M	5360K	3888K	9.9M	0K	0K	0K	8
MALLOC_SMALL (empty)	8192K	0K	0K	48K	0K	0K	0K	1
MALLOC_TINY	19.0M	8320K	8320K	2416K	0K	0K	0K	19
MALLOC_TINY (empty)	4096K	128K	128K	0K	0K	0K	0K	4
Performance tool data	4784K	4592K	4592K	192K	0K	0K	0K	43
								
<hr/>								
TOTAL	1.6G	355.9M	143.5M	41.6M	0K	95.6M	13.2M	4362

▲図 3.91 MemoryGraph Summary 表示

3.7 Instruments

Xcode には Instruments (インストゥルメンツ) という詳細な計測・分析を得意とするツールがあります。Instruments は「Product -> Analyze」を選択することでビルドが始まります。完了すると次のような計測項目のテンプレートを選択する画面が開きます。

第3章 プロファイリングツール



▲図 3.92 Instruments テンプレート選択画面

テンプレートの多さからわかるように、Instruments はさまざまな内容を分析できます。本節ではこの中からとくに利用する頻度の高い「Time Profiler」と「Allocations」を取り上げます。

3.7.1 Time Profiler

Time Profiler はコードの実行時間を計測するツールです。Unity Profiler にあった CPU モジュールと同じく、処理時間を改善する際に使用します。

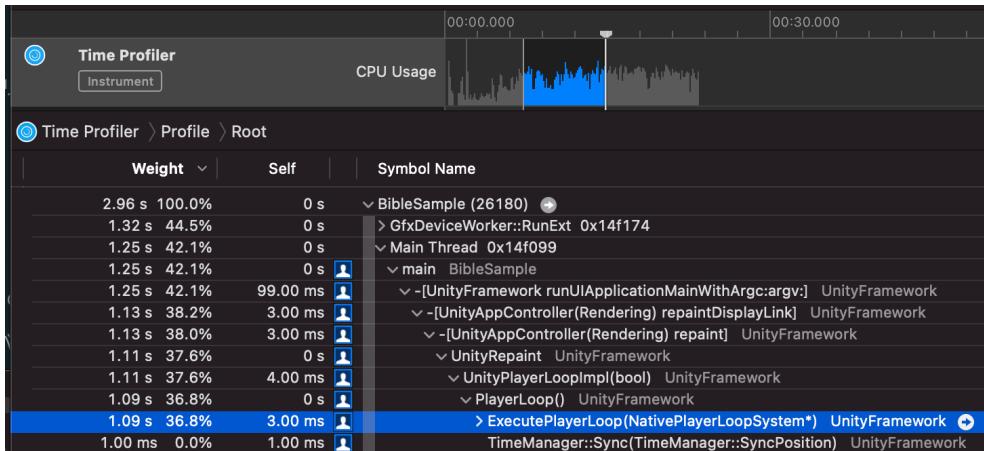
計測を開始するためには Time Profiler のツールバーにある赤い丸印のレコードボタンを押下する必要があります。



▲図 3.93 レコード開始ボタン

3.7 Instruments

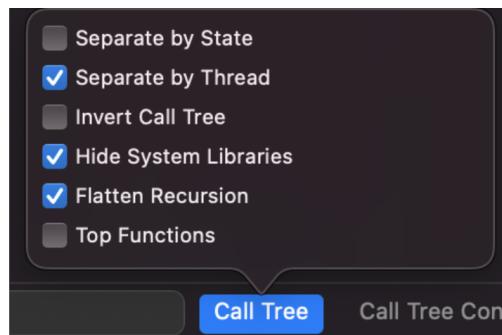
計測を行うと図 3.94 のような表示になります。



▲図 3.94 計測結果

Unity Profiler と違いフレーム単位ではなく、区間での分析を行っていきます。下部の Tree View には区間内の処理時間が表示されます。ゲームロジックの処理時間を最適化する場合、Tree View の PlayerLoop 以下の処理を分析していくとよいでしょう。

Tree View の表示を見やすくするために、Xcode の下部にある Call Trees の設定を図 3.95 のようにしておくとよいでしょう。とくに Hide System Libraries にチェックすることで、手の出せないシステムコードが非表示になり調査がしやすくなります。



▲図 3.95 Call Trees 設定

このようにして処理時間を分析し、最適化していくことができます。

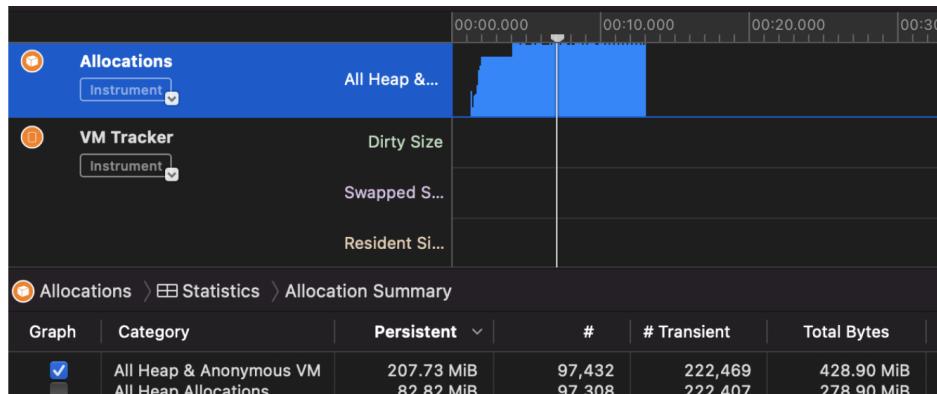
Time Profiler ではシンボル名が Unity Profiler と違います。大きく変わりはないですが「クラス名_関数名_ランダムな文字列」という表記になります。

```
▽ RuntimeInvoker__TrueVoid_t700C6383A2A510C2CF4D86DABD5CA9FF70ADAC5(void *(), MethodInfo const*,  
    ▽ SampleScript_Update_mB3FBFF57BD6F501BF7906ECFF223E17CCB509C38 UnityFramework  
        > DebugLogHandler_CUSTOM_Internal_Log(LogType, LogOption, ScriptingBackendNativeStringPtrOpaque*, Sc  
            ▶ SampleScript_TestMethod_m390D7F0A04D12D51B91D3A4D94835D6D77DA194F UnityFramework ⚡  
                > DebugLogHandler_LogFormat_mB876FBE8959FC3D9E9950527A82936F779F7A00C UnityFramework  
                    > EventSystem_Update_mF0C1580BB2C9A125C27282F471DB4DE6B772DE6D UnityFramework
```

▲図 3.96 Time Profiler でのシンボル名

3.7.2 Allocations

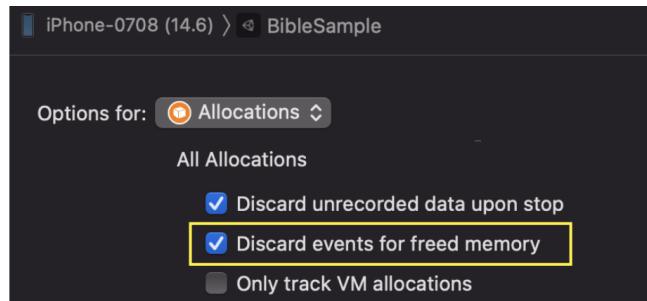
Allocations はメモリ使用量を計測するためのツールです。メモリリークや使用量の改善を行う際に使用します。



▲図 3.97 Allocations 計測画面

計測を行う前に「File -> Recording Options」を開き「Discard events for freed memory」にチェックを入れましょう。

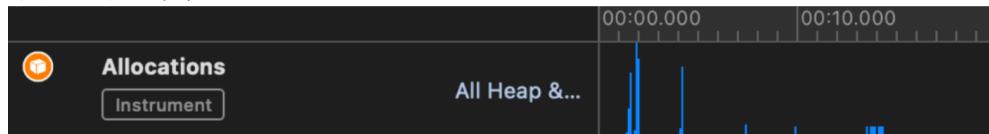
3.7 Instruments



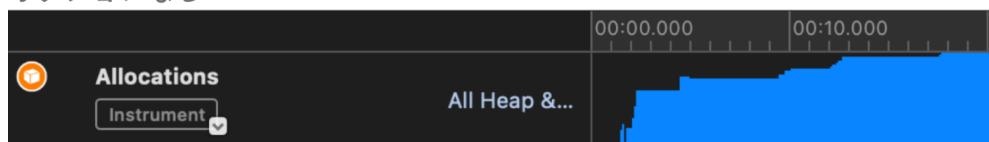
▲図 3.98 オプション設定画面

このオプションを有効にするとメモリが解放された際に記録を破棄します。

オプションあり



オプションなし

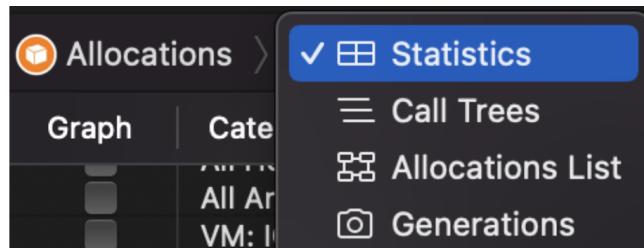


▲図 3.99 オプション設定による違い

図 3.99 を見てわかるようにオプションの有無で見た目が大きく変わります。オプションありの場合、メモリがアロケーションしたタイミングでのみ線が記録されます。また記録された線は、その確保された領域が解放されると破棄されます。つまりこのオプションを設定することで、線が残り続けていればメモリから解放されていないということになります。たとえばシーン遷移によってメモリを解放する設計の場合、遷移前のシーン区間に線が多く残っていた場合メモリリークの疑いがあります。その際は Tree View で詳細を追いましょう。

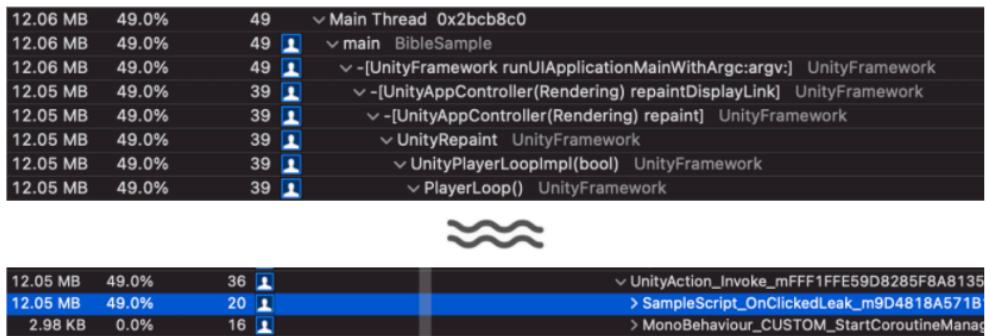
画面下部の Tree View は Time Profiler と同様に指定した範囲の詳細が表示されます。この Tree View の表示方法は 4 種類あります。

第3章 プロファイリングツール



▲図 3.100 表示方法の選択

もっともオススメの表示方法は Call Trees です。これによってどのコードによってアロケーションが発生したのかを追うことができます。画面下部に Call Trees の表示オプションがあるので、Time Profiler で紹介した図 3.95 と同じように Hide System Libraries などのオプションを設定するとよいでしょう。図 3.101 では Call Trees の表示をキャプチャしました。12.05MB のアロケーションが SampleScript の OnClicked で発生していることがわかります。



▲図 3.101 Call Tree 表示

最後に Generations という機能を紹介します。Xcode の下部に「Mark Generations」というボタンがあります。



▲図 3.102 Mark Generation ボタン

これを押下するとそのタイミングでのメモリが記憶されます。その後、再度 Mark

3.8 Android Studio

Generations を押下すると前回のデータと比較して新たに確保されたメモリ量が記録されます。

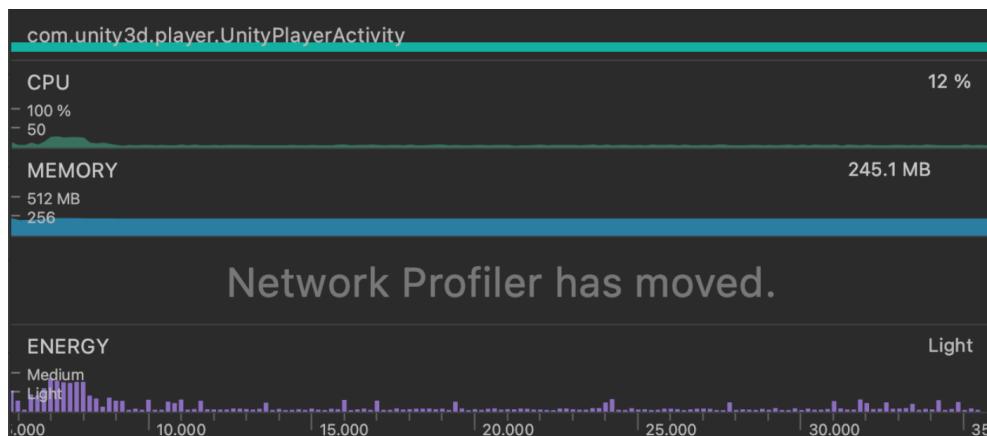
Snapshot	Timestamp	Growth	# Persistent
> Generation A	00:15.602.827	168.47 MiB	96,330
> Generation B	00:20.858.176	75.71 MiB	6,219
> Generation C	00:24.390.265	256 Bytes	1

▲図 3.103 Generations

図 3.103 の各 Generations は、詳細を見ると Call Tree 形式で表示されるため、メモリ増加が何によって発生したかを追うことが可能です。

3.8 Android Studio

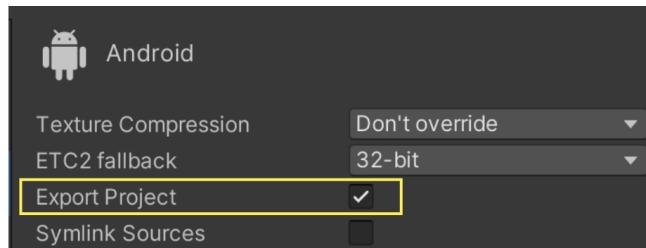
Android Studio は Android の統合開発環境ツールです。このツールを用いてアプリケーションの状態を計測できます。プロファイル可能な項目は CPU、Memory、Network、Energy の 4 つです。本節ではまずプロファイル方法について紹介し、その後 CPU と Memory の計測項目について解説します。



▲図 3.104 プロファイル画面

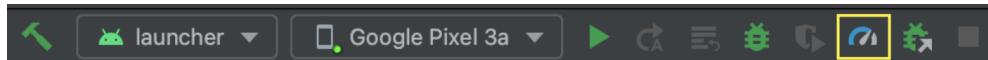
3.8.1 プロファイル方法

プロファイル方法は2通りあります。1つ目はAndroid Studio経由でビルドし、プロファイルする方法です。この方法ではまずAndroid StudioのプロジェクトをUnityから書き出します。Build Settingsで「Export Project」にチェックをいれてビルドします。



▲図 3.105 プロジェクトの Export

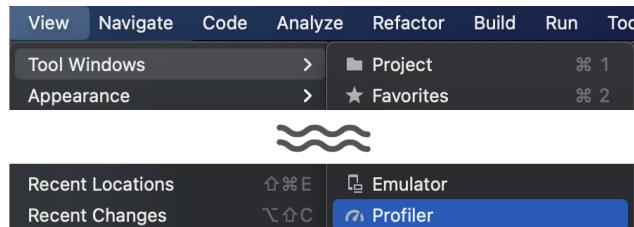
次に書き出されたプロジェクトをAndroid Studioで開きます。そしてAndroid端末を接続した状態で、右上にあるゲージのようなアイコンを押下するとビルドが開始します。ビルド完了後、アプリが立ち上がりプロファイルが開始します。



▲図 3.106 Profile 開始アイコン

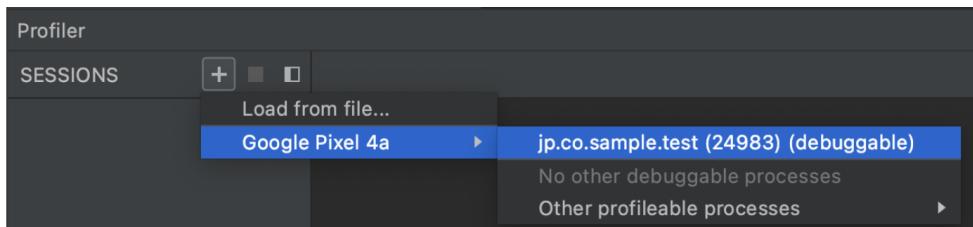
2つ目は実行中のプロセスをデバッガーにアタッチして計測する方法です。まずAndroid Studioメニューの「View -> Tool Windows -> Profiler」からAndroid Profilerを開きます。

3.8 Android Studio



▲図 3.107 Android Profiler を開く

次に開いた Profiler の **SESSIONS** から計測対象のセッションを選択します。セッションを接続するためには、計測するアプリケーションを起動しておく必要があります。また **Development Build** を適応したバイナリでないといけません。セッションの接続が完了するとプロファイルが開始します。



▲図 3.108 プロファイルする SESSION を選択する

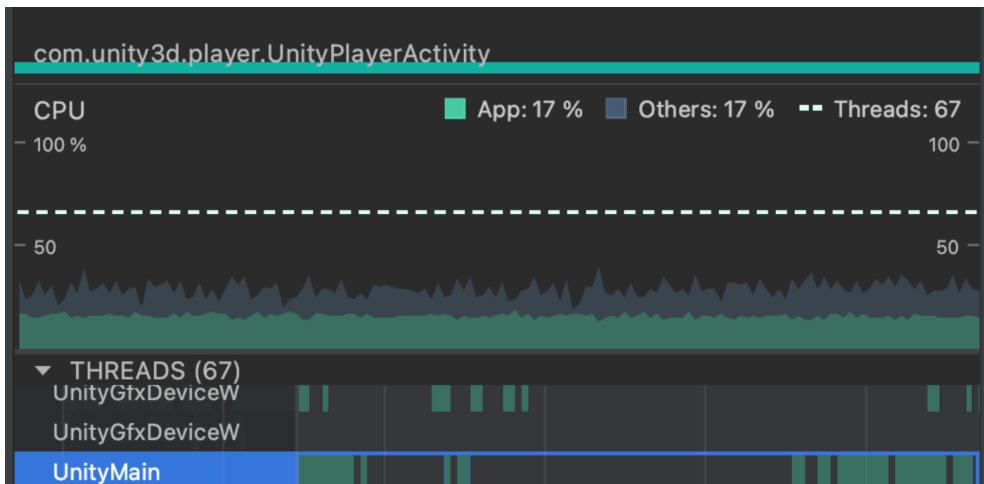
2つ目のデバッガーにアタッチする方法はプロジェクトを書き出す必要がなく、気軽に利用できるので覚えておくと良いでしょう。

厳密には Unity の Development Build ではなく、AndroidManifest.xml にて `debuggable` や `profileable` の設定を行う必要があります。Unity では Development Build を行った場合、自動的に `debuggable` が `true` になります。

3.8.2 CPU 計測

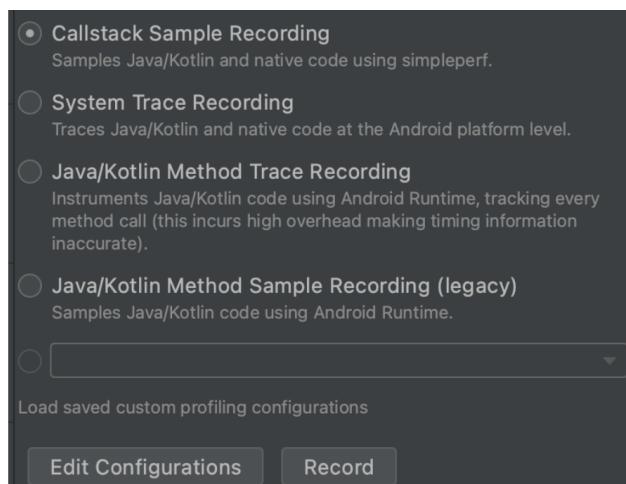
CPU 計測の画面は図 3.109 のようになります。この画面だけでは何がどれくらいの処理時間を消費しているかわかりません。より詳細に見るためには、詳細に見たいス

レッドを選択する必要があります。



▲図 3.109 CPU 計測トップ画面、スレッド選択

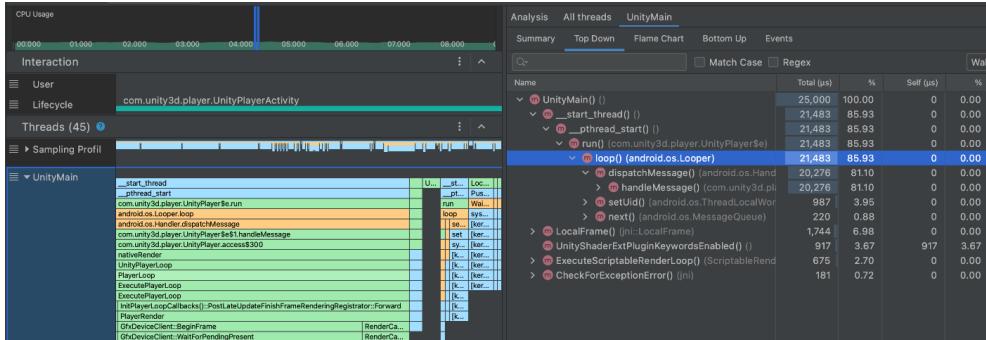
スレッド選択後、Record ボタンを押下することでスレッドのコールスタックが計測されます。図 3.110 のように計測タイプがいくつかありますが「Callstack Sample Recording」で問題ないでしょう。



▲図 3.110 Record 開始ボタン

3.8 Android Studio

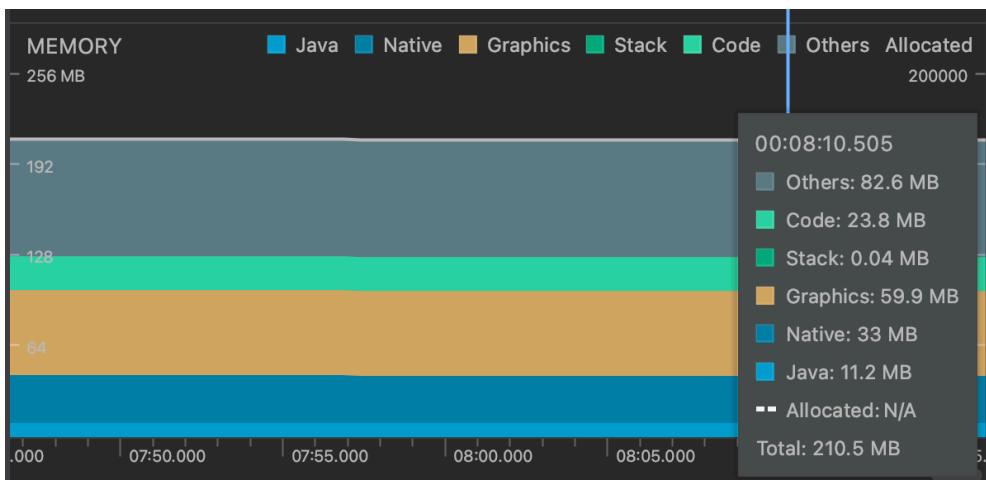
Stop ボタンを押下すると計測が終了し、結果が表示されます。結果画面としては Unity Profiler の CPU モジュールのような表示になります。



▲図 3.111 コールスタック計測結果画面

3.8.3 Memory 計測

Memory 計測の画面は図 3.112 のようになります。この画面ではメモリの内訳を見ることはできません。

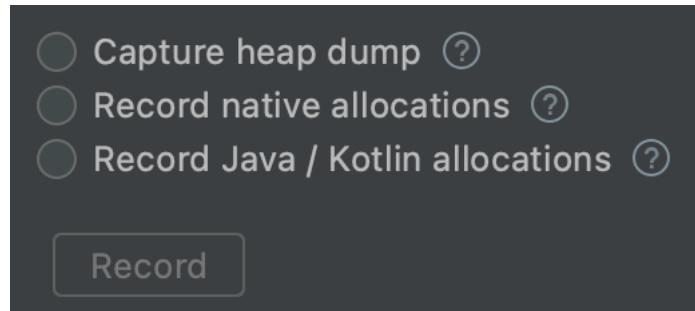


▲図 3.112 Memory 計測画面

メモリの内訳を見る場合は追加で計測を行う必要があります。計測方法は 3 種類あ

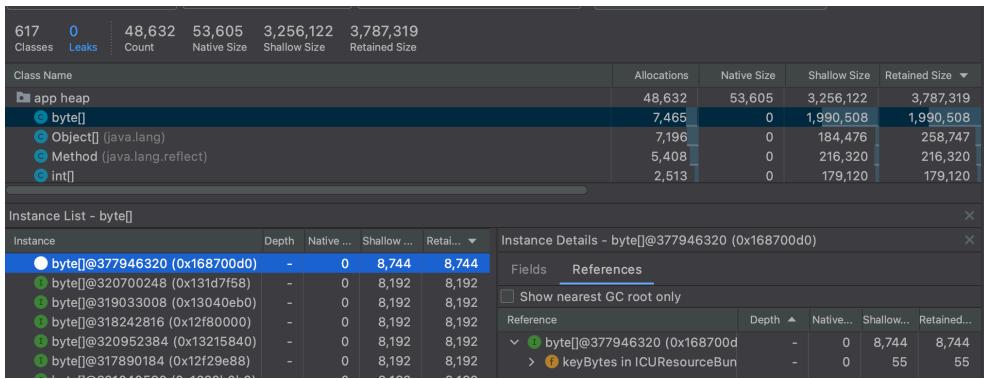
第3章 プロファイリングツール

ります。「Capture heap dump」は押下したタイミングでのメモリ情報が取得できます。それ以外のボタンは計測区間中のアロケーションを分析するためのものです。



▲図 3.113 Memory 計測オプション

例として図 3.114 に Heap Dump の計測結果をキャプチャしました。詳細分析するには少し粒度が粗いので難易度が高いでしょう。



▲図 3.114 Heap Dump 結果

3.9 RenderDoc

RenderDoc はオープンソースで開発されており、無料で利用できる高品質なグラフィックスデバッガーツールです。このツールは現時点では Windows と Linux に対応していますが、Mac には対応していません。またサポートしている Graphics API は

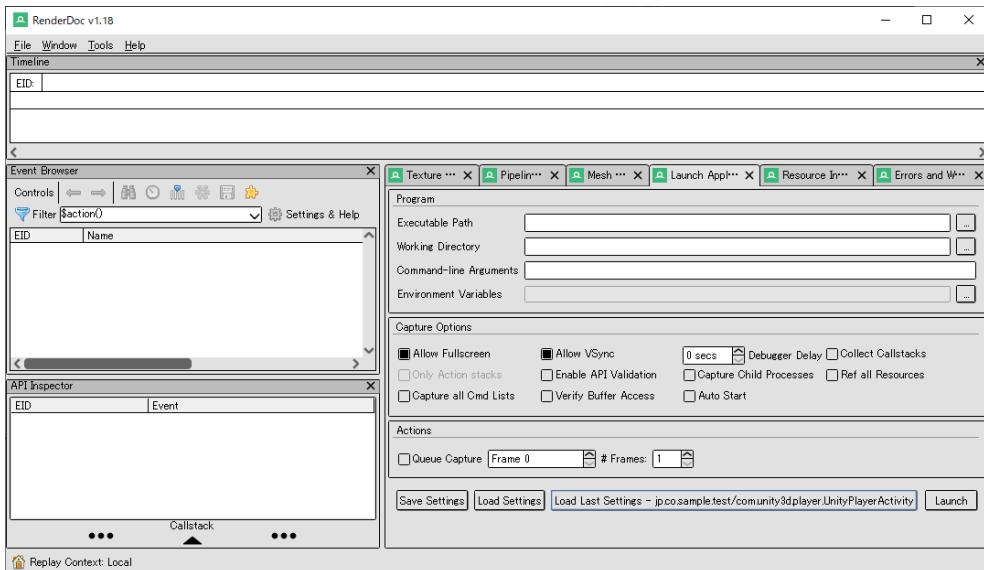
3.9 RenderDoc

Vulkan や OpenGL(ES)、D3D11、D3D12 などです。そのため Android での利用はできますが iOS では利用できません。

本節では Android アプリケーションを実際にプロファイルしてみます。ただし Android のプロファイリングにはいくつか制約があるので注意してください。まず Android OS バージョンは 6.0 以降が必須です。そして計測対象のアプリケーションは Debuggable を有効にする必要があります。これはビルド時に Development Build を選択しておけば問題ありません。なお、プロファイルに使用した RenderDoc のバージョンは v1.18 となります。

3.9.1 計測方法

まずは RenderDoc の準備を行いましょう。公式サイト^{*3}からインストーラーをダウンロードし、ツールのインストールを行いましょう。インストール後、RenderDoc のツールを開きます。

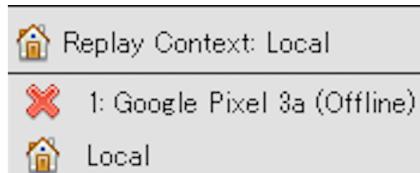


▲図 3.115 RenderDoc 起動後の画面

次に Android デバイスを RenderDoc と接続します。左下隅にある家マークを押下すると PC と接続されているデバイスリストが表示されます。リストの中より計測した

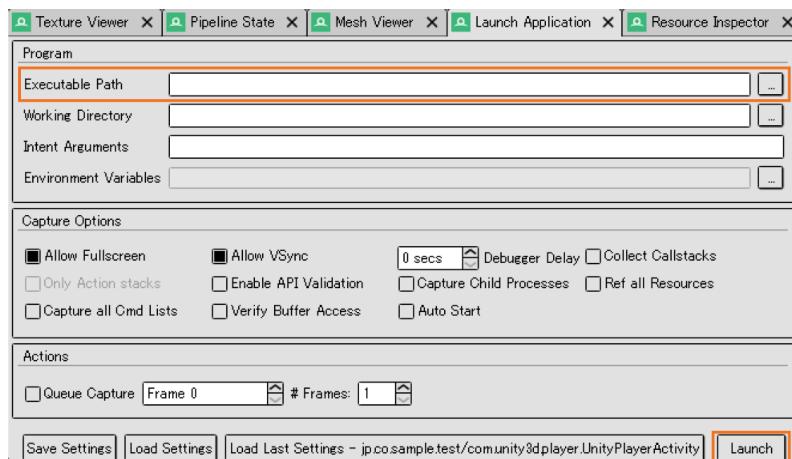
^{*3} <https://renderdoc.org/>

い端末を選択してください。



▲図 3.116 デバイスとの接続

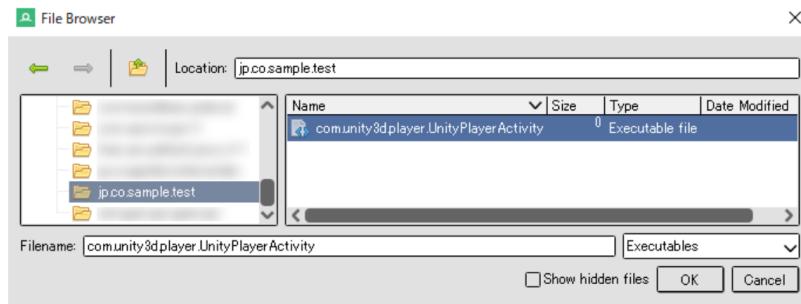
次に接続したデバイスから起動するアプリケーションを選択しましょう。右側にあるタブから Launch Application を選択し、Executable Path から実行するアプリを選択します。



▲図 3.117 Launch Application タブ 実行アプリ選択

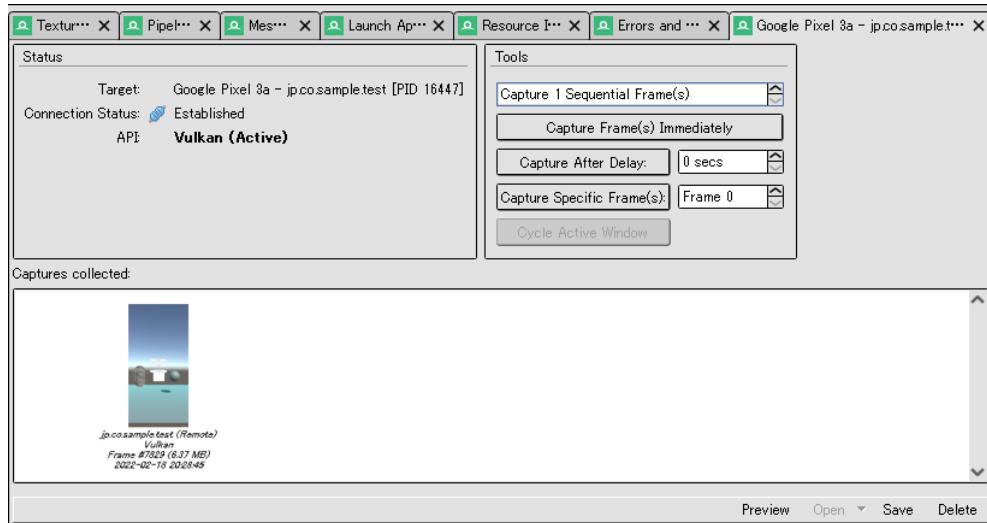
File Browser のウィンドウが開くので、今回計測する Pacakge Name を探し Activity を選択します。

3.9 RenderDoc



▲図 3.118 計測するアプリケーション選択

最後に Launch Application から Launch ボタンを押下するとデバイスでアプリケーションが起動します。また RenderDoc 上では計測するためのタブが新たに追加されます。

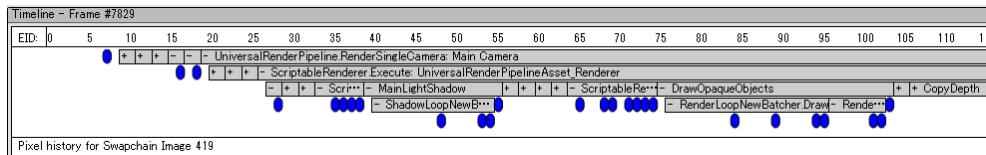


▲図 3.119 計測用のタブ

試しに Capture Frame(s) Immediately を押下すると、フレームデータがキャプチャされ、Capture collected に並びます。このデータをダブルクリックすることでキャプチャデータを開くことができます。

3.9.2 キャプチャデータの見方

RenderDoc にはさまざまな機能がありますが、本節では重要な部分に絞って機能を説明します。まず画面上部にはキャプチャしたフレームのタイムラインが表示されます。それぞれの描画コマンドがどのような順番で行われたかを視覚的に捉えることができます。



▲図 3.120 タイムライン

次に Event Browser です。ここには各コマンドが上から順番に記載されています。

EID	Name	Duration (μs)
64-73	➤ ScriptableRenderPass.Configure	0.00
75-102	▼ DrawOpaqueObjects	4.53125
76-94	▼ RenderLoopNewBatcher.Draw	3.75
84	vkCmdDrawIndexed(600, 1)	2.13542
89	vkCmdDrawIndexed(2304, 1)	0.83333
94	vkCmdDrawIndexed(36, 1)	0.78125
96-101	▼ RenderLoop.Draw	0.78125
101	└ vkCmdDrawIndexed(2496, 1)	0.78125
104	ScriptableRenderPass.Configure	0.00

▲図 3.121 Event Browser

Event Browser 内の上部にある「時計マーク」を押下すると、各コマンドの処理時間が「Duration」に表示されます。計測タイミングによって処理時間が変動するのであくまで目安と考えるとよいでしょう。また、DrawOpaqueObjects のコマンドの内訳を見ると 3 つが Batch 処理されており、1 つだけ Batch から外れた描画になっているのがわかります。

次に右側にタブで並んでいる各ウインドウについて説明します。このタブの中には Event Browser で選択したコマンドの詳細情報を確認できるウインドウがあります。

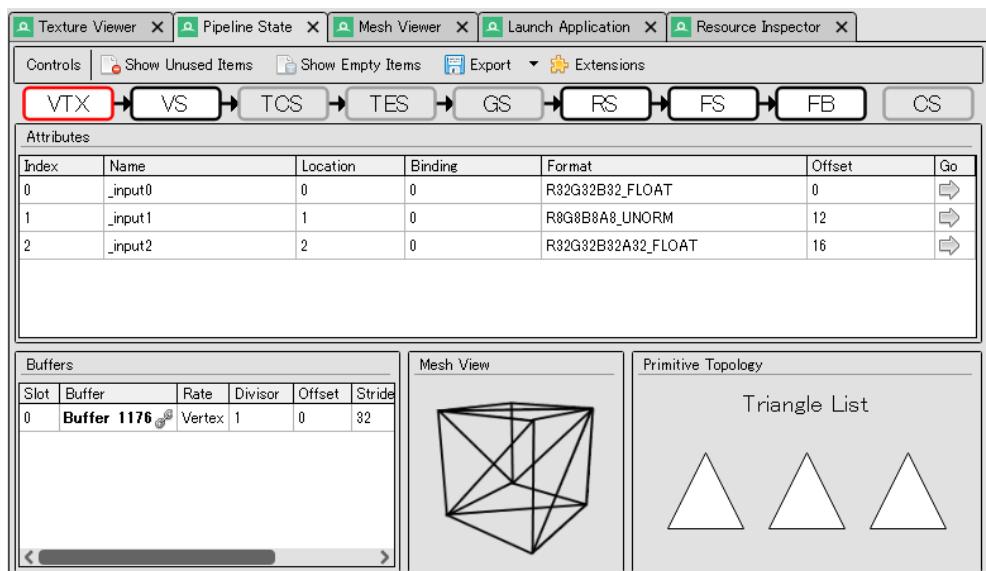
3.9 RenderDoc

とくに重要なのは Mesh Viewer、Texture Viewer、Pipeline State の 3 つです。



▲図 3.122 各ウインドウ

初めに Pipeline State についてです。Pipeline State はそのオブジェクトが画面に描画されるまでの各シェーダーステージでどのようなパラメーターだったかを確認できます。また使用されたシェーダーやその中身も閲覧可能です。



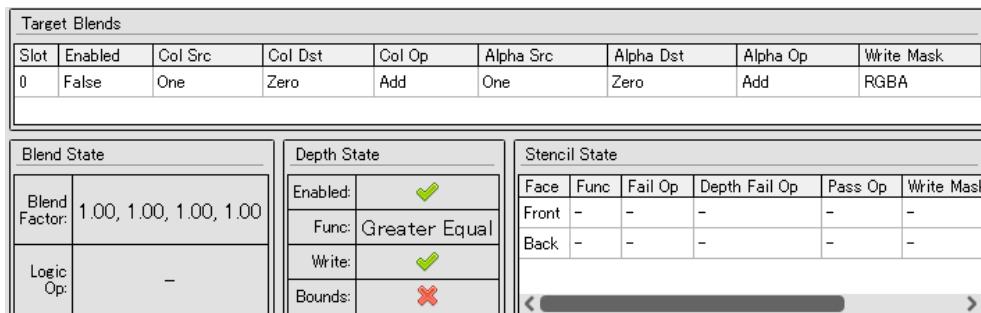
▲図 3.123 Pipeline State

パイプラインステートに表示されるステージ名は省略されているため正式名称を表 3.7 にまとめます。

▼表 3.7 PipelineState の正式名称

ステージ名	正式名
VTX	Vertex Input
VS	Vertex Shader
TCS	Tessellation Control Shader
TES	Tessellation Evaluation Shader
GS	Geometry Shader
RS	Rasterizer
FS	Fragment Shader
FB	Frame Buffer
CS	Compute Shader

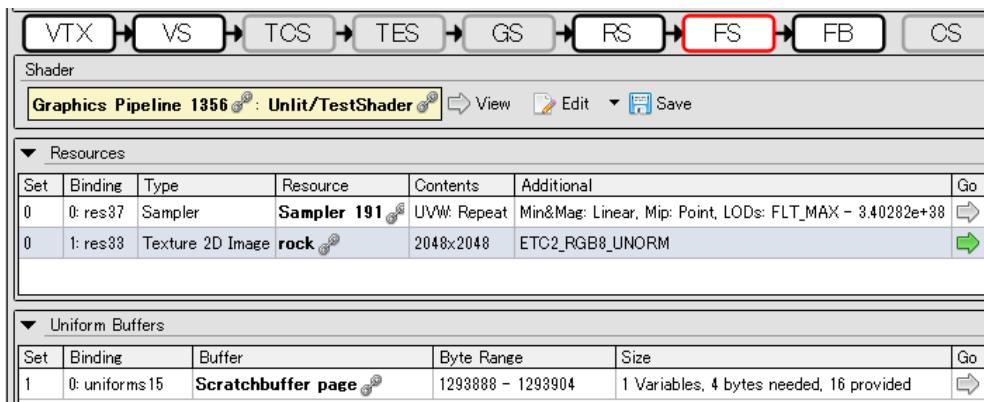
図 3.123 では VTX ステージが選択されており、トポロジーや頂点入力データが確認できます。他にも図 3.124 の FB ステージでは、アウトプット先のテクスチャの状態や Blend State などの詳細を見られます。



▲図 3.124 FB(Frame Buffer) の State

また図 3.125 の FS ステージでは、フラグメントシェーダー内で利用されたテクスチャやパラメーターの確認ができます。

3.9 RenderDoc



▲図 3.125 FS(Fragment Shader) の State

FS ステージの中央にある Resources には、使用されたテクスチャやサンプラーが表記されています。また FS ステージの下段にある Uniform Buffers には、CBuffer が表示されています。この CBuffer には float や color など、数値情報のプロパティが格納されています。各項目の右側には「Go」という矢印アイコンがあり、これを押下するとデータの詳細を確認できます。

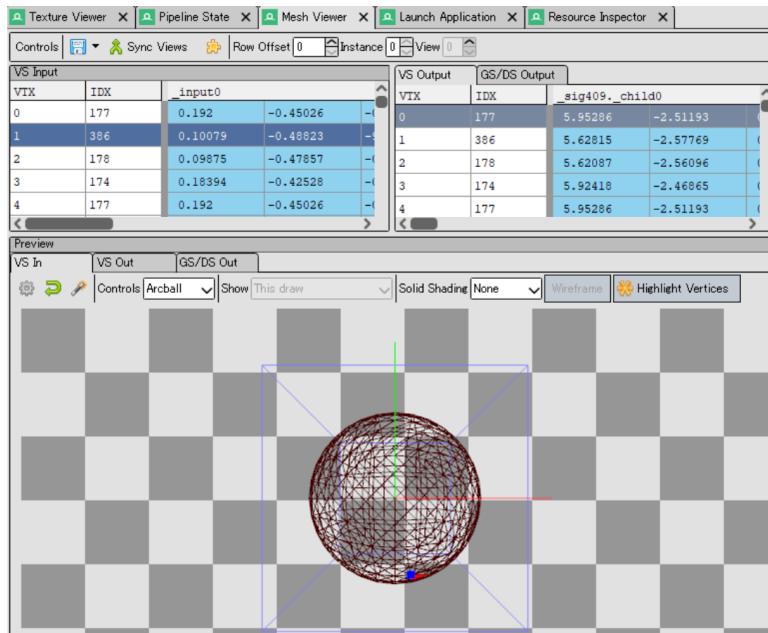
FS ステージの上段には使用したシェーダーが表示されています。View を押下するとそのシェーダーコードを確認できます。表示をわかりやすくするために Disassembly type は GLSL がオススメです。

```
#version 450
layout(set = 1, binding = 0, std140) uniform _13_15
{
    float _m0;
} _15;
layout(set = 0, binding = 1) uniform mediump texture2D _33;
layout(set = 0, binding = 0) uniform mediump sampler _37;
```

▲図 3.126 Shader コードの確認

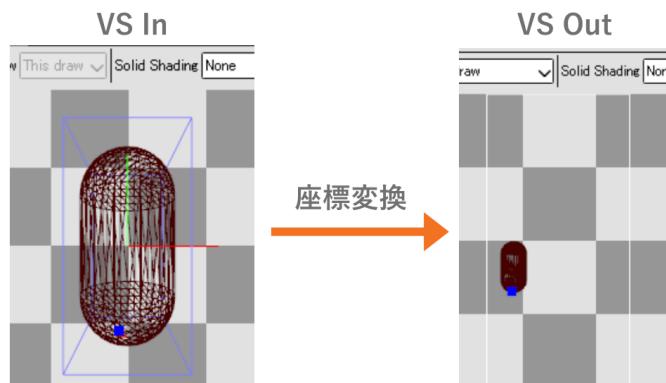
次に Mesh Viewer です。これは Mesh の情報を視覚的に見ることができ、最適化やデバッグに便利な機能です。

第3章 プロファイリングツール



▲図 3.127 Mesh Viewer

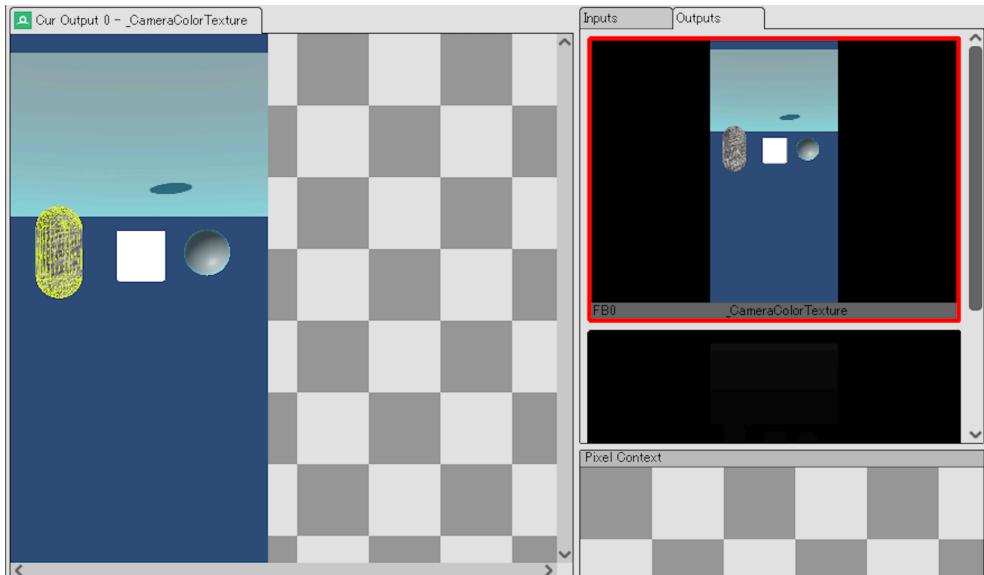
Mesh Viewer の上段にはメッシュの頂点情報がテーブル形式で表記されています。下段にはカメラを動かして形状が確認できるプレビュー画面があります。どちらも In と Out でタブが分かれており、変換の前後で値や見た目がどのように変わったかを把握できます。



▲図 3.128 Mesh Viewer の In、Out の Preview 表示

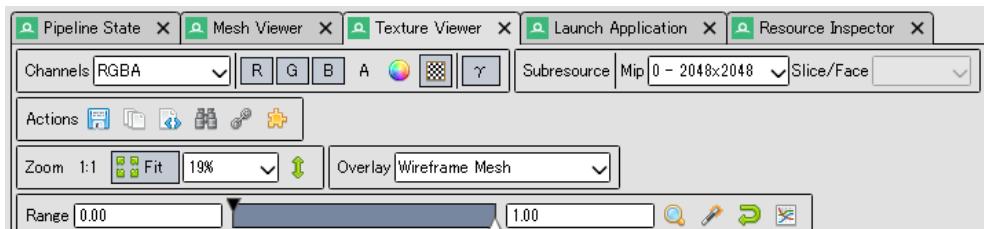
3.9 RenderDoc

最後に Texture Viewer です。この画面では Event Browser で選択しているコマンドの「入力に使用したテクスチャ」と「出力結果」がわかります。



▲図 3.129 Texture Viewer テクスチャ確認画面

画面右側の領域では入出力のテクスチャを確認できます。表示されているテクスチャを押下すると画面左側の領域に反映されます。左側の画面はただ表示するだけでなく、カラーチャンネルをフィルターしたり、ツールバーの設定を適用させることができます。



▲図 3.130 Texture Viewer のツールバー

図 3.129 では、Overlay に「Wireframe Mesh」を選択していたので、このコマンドで描画されたオブジェクトに黄色のワイヤーフレーム表示がされており視覚的にわかりやすくなっています。

また Texture Viewer には Pixel Context という機能があります。これは選択したピクセルの描画履歴を見るための機能です。履歴がわかるということはあるピクセルに対して塗りつぶしがどれくらいの頻度で行われているかを把握できます。これはオーバードローの調査や最適化を行うのに便利な機能です。ただしあくまでもピクセル単位なので全体的にオーバードローを調査するのには向いていません。調査方法としては図 3.129 の左側の画面で、調査したい箇所を右クリックすると Pixel Context にその位置が反映されます。



▲図 3.131 Pixel Context への反映

次に Pixel Context の History ボタンを押下すると、そのピクセルの描画履歴が確認できます。

3.9 RenderDoc

Pixel History on _CameraColorTexture for (195, 734)		X
Preview colours displayed in visible range 0.00 - 1.00 with RGB channels visible.		
Double click to jump to an event. Right click to debug an event, or hide failed events.		
Event		
> UniversalRenderPipeline.RenderSingleCamera: Main Camera	Tex Before	Tex After
> ScriptableRenderer.Execute: UniversalRenderPipelineAsset_Renderer	R: 0.00 G: 0.00 B: 0.00	R: 0.19141 G: 0.30078 B: 0.46875
> ScriptableRenderPass.Configure	D: 0.00 S: 0x00	D: 0.00 S: 0x00
EID 73 vkCmdBeginRenderPass(C=Clear, DS=Clear) 1 Fragments touching pixel		
> UniversalRenderPipeline.RenderSingleCamera: Main Camera ...	Tex Before	Tex After
> DrawOpaqueObjects	R: 0.19141	R: 0.54688
> RenderLoopNewBatcher.Draw	G: 0.30078	G: 0.82031
> EID 84 vkCmdDrawIndexed(600, 1) Depth test failed 1 Fragments touching pixel	B: 0.46875 D: 0.00 S: 0x00	B: 0.84375 D: 0.02157 S: 0x00
> UniversalRenderPipeline.RenderSingleCamera: Main Camera ...	Tex Before	Tex After
> DrawOpaqueObjects	R: 0.54688	R: 0.51563
> RenderLoop.Draw	G: 0.82031	G: 0.51563
> EID 101 vkCmdDrawIndexed(2496, 1) 1 Fragments touching pixel	B: 0.84375 D: 0.02157 S: 0x00	B: 0.51563 D: 0.03177 S: 0x00
> UniversalRenderPipeline.RenderSingleCamera: Main Camera	Tex Before	Tex After
> ScriptableRenderer.Execute: UniversalRenderPipelineAsset_Renderer	R: 0.51563	R: 0.51563
> Camera.RenderSkybox	G: 0.51563	G: 0.51563
> EID 137 vkCmdDraw(5040, 1) Depth test failed 1 Fragments touching pixel	B: 0.51563 D: 0.03177 S: 0x00	B: 0.51563 D: 0.03177 S: 0x00

▲図 3.132 ピクセルの描画履歴

図 3.132 では 4 つの履歴があります。緑色の行は深度テストなどのパイプラインのテストにすべて合格し描画されたことを示します。一部のテストに失敗し描画されなかった場合は赤色になります。キャプチャした画像では、画面のクリア処理とカプセルの描画が成功し、Plane と Skybox が深度テストに失敗しています。



PERFORMANCE TUNING BIBLE

CHAPTER

04

第4章

Tuning Practice

— Asset —

CyberAgent Smartphone Games & Entertainment

第4章

Tuning Practice - Asset

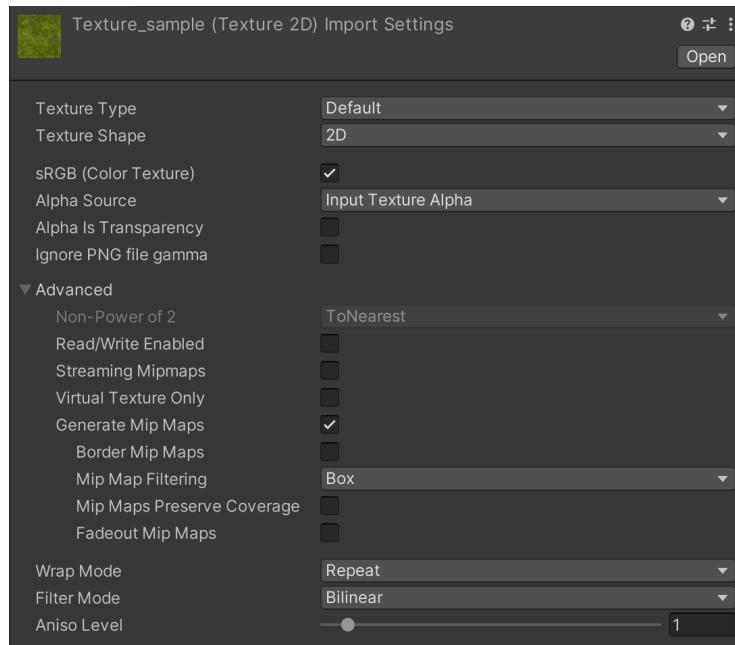
ゲーム製作ではテクスチャやメッシュ、アニメーション、サウンドなどさまざまな種類のアセットを大量に扱います。そこで本章ではそれらのアセットに関して、パフォーマンスチューニングを行う上で気をつけるべき設定項目など、実践的な知識についてまとめます。

4.1 Texture

テクスチャの元となる画像データはゲーム製作において欠かせない存在です。その一方でメモリ消費量は比較的多くなるため設定は適切に行う必要があります。

4.1.1 インポート設定

図 4.1 は Unity におけるテクスチャのインポート設定です。



▲図 4.1 テクスチャ設定

この中でもとくに注意すべきものについて紹介します。

4.1.2 Read/Write

このオプションはデフォルトでは無効になっています。無効な状態であれば GPU メモリにしかテクスチャは展開されません。有効にした場合は GPU メモリだけでなくメインメモリにもコピーされるため、2 倍の消費量になってしまいます。そのため `Texture.GetPixel` や `Texture.SetPixel` といった API を使用せず Shader でしかアクセスしない場合、必ず無効にしましょう。

またランタイムで生成したテクスチャは、リスト 4.1 で示すように `makeNoLongerReadable` を `true` に設定することで、メインメモリへのコピーを回避できます。

▼リスト 4.1 makeNoLongerReadable の設定

```
1: texture2D.Apply(updateMipmaps, makeNoLongerReadable: true)
```

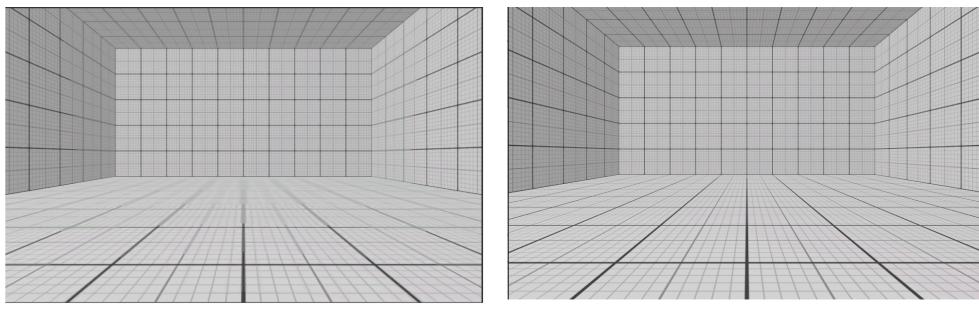
GPU メモリからメインメモリへのテクスチャ転送は時間がかかるため、読み取り可能な場合は、どちらにもテクスチャを展開することでパフォーマンスの向上が図られています。

4.1.3 Generate Mip Maps

Mip Map の設定を有効にするとテクスチャのメモリ使用量が約 1.3 倍になります。この設定は 3D 上のオブジェクトに対して行うことが一般的で、遠くのオブジェクトに対して、ジャギ軽減やテクスチャ転送量削減を目的に設定します。2D スプライトや UI 画像に対しては基本的に不要なので、無効にしておきましょう。

4.1.4 Aniso Level

Aniso Level はオブジェクトを浅い角度で描画した際に、テクスチャの見栄えをボケずに描画するための機能です。この機能は主に地面や床など遠くまで続いているオブジェクトに使用されます。Aniso Level 値が高いほどその恩恵を受けられますが、処理コストはかかります。

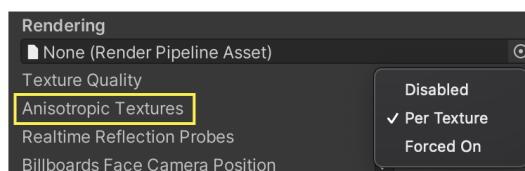


▲図 4.2 Aniso Level 適応イメージ

設定値は0~16までありますが少し特殊な仕様となっています。

- 0: プロジェクト設定によらず必ず無効
- 1: 基本的には無効。ただし、プロジェクト設定がForced Onの場合は、9~16の値にクランプされる
- それ以外: その値での設定

テクスチャをインポートするとデフォルトでは値が1になっています。そのためハイスペック端末が対象でない限りForced Onの設定は推奨できません。Forced Onの設定は「Project Settings -> Quality」のAnisotropic Texturesから設定可能です。



▲図 4.3 Forced On の設定

Aniso Level の設定が効果のないオブジェクトで有効になっていないか、もしくは効果のあるオブジェクトに対して無闇に高い設定値になっていないかを確認しましょう。

Aniso Level による効果は、線形ではなく段階で切り替わっている挙動になっています。筆者が検証した結果では、0~1、2~3、4~7、8 以降という4段階で変化していました。

4.1.5 圧縮設定

テクスチャは特別な理由がない限り圧縮するべきでしょう。もしプロジェクト内に無圧縮なテクスチャが存在した場合、ヒューマンエラーかレギュレーションがない可能性があります。早急に確認しましょう。圧縮設定に関する詳細は「2.3.3 画像の圧縮」にてご確認ください。

このような圧縮設定に関してはヒューマンエラーが起きないように TextureImporter を利用し自動化することを推奨します。

▼リスト 4.2 TextureImporter の自動化例

```
1: using UnityEditor;
2:
3: public class ImporterExample : AssetPostprocessor
4: {
5:     private void OnPreprocessTexture()
6:     {
7:         var importer = assetImporter as TextureImporter;
8:         importer.isReadable = false; // Read/Writeの設定なども可能
9:
10:        var settings = new TextureImporterPlatformSettings();
11:        // Android = "Android", PC = "Standalone"を指定
12:        settings.name = "iPhone";
13:        settings.overridden = true;
14:        settings.textureCompression = TextureImporterCompression.Compressed;
15:        settings.format = TextureImporterFormat.ASTC_6x6; // 圧縮形式を指定
16:        importer.SetPlatformTextureSettings(settings);
17:    }
18: }
```

またすべてのテクスチャが同じ圧縮形式である必要はありません。たとえば UI 画像の中でも全体的にグラデーションがかかっている画像は、圧縮による品質低下が目立ちやすいです。そのような場合は対象となる一部の画像だけ、圧縮率を低めに設定すると良いでしょう。逆に 3D モデルなどのテクスチャは品質低下がわかりにくいため、圧縮率を高くするなど適切な設定を探るのがよいでしょう。

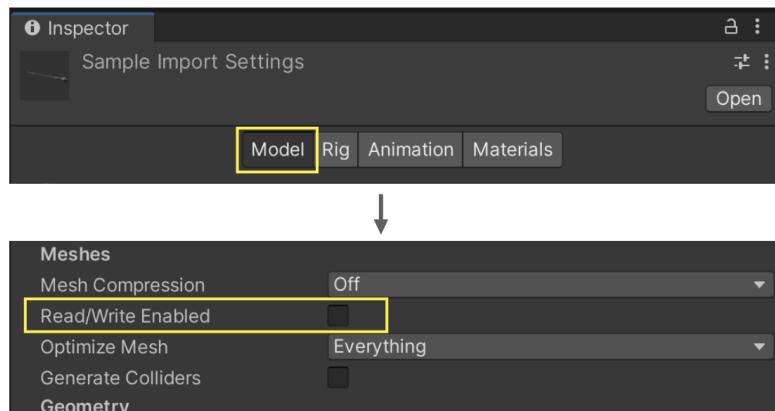
4.2 Mesh

Unity にインポートしたメッシュ（モデル）を扱う場合の注意点を紹介します。インポートしたモデルデータは設定次第でパフォーマンスは上がります。注意すべきポイントは次の 4 つです。

- Read/Write Enabled
- Vertex Compression
- Mesh Compression
- Optimize Mesh Data

4.2.1 Read/Write Enabled

Mesh の注意点 1 つ目は Read/Write Enabled です。モデルのインスペクターにあるこのオプションはデフォルトでは無効になっています。



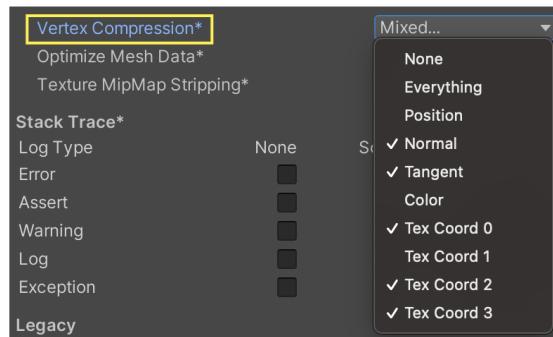
▲図 4.4 Read/Write 設定

ランタイム中、メッシュにアクセスする必要がなければ無効にしておきましょう。具体的にはモデルを Unity 上に配置して、AnimationClip を再生するくらいの使い方であれば、Read/Write Enabled は無効で問題ありません。

有効にすると、CPU でアクセス可能な情報をメモリに保持するためメモリを 2 倍消費します。無効にするだけでメモリの節約になるためぜひ確認してみてください。

4.2.2 Vertex Compression

Vertex Compression はメッシュの頂点情報の精度を float から half に変更するオプションです。これによってランタイム時のメモリ使用量とファイルサイズを小さくすることができます。「Project Settings -> Player」の Other で設定ができ、デフォルト設定では次のようになっています。



▲図 4.5 Vertex Compression のデフォルト設定

ただし、この Vertex Compression は次のような条件に当てはまると無効化されるので気をつけましょう。

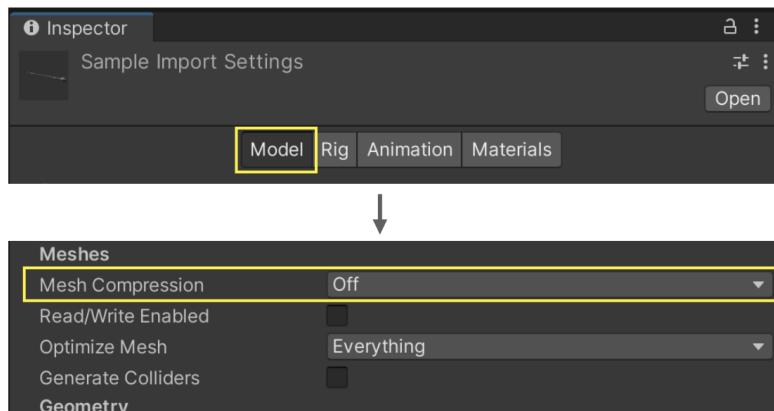
- Read/Write が有効
- Mesh Compression が有効
- Dynamic Batching が有効かつ適応可能なメッシュ（頂点数が 300 以下、頂点属性が 900 以下）

4.2.3 Mesh Compression

Mesh Compression はメッシュの圧縮率を変更できます。圧縮率が高いほどファイルサイズが小さくなりストレージの占める割合が削減されます。圧縮されたデータは実行時に展開されます。そのためランタイム時のメモリ使用量には影響がありません。

Mesh Compression の圧縮設定には 4 つの選択肢があります。

- Off: 非圧縮
- Low: 低圧縮
- Medium: 中程度の圧縮
- High: 高圧縮



▲図 4.6 Mesh Compression

「4.2.2 Vertex Compression」で触れましたが、このオプションを有効にすると **Vertex Compression** が無効化されます。とくにメモリ使用量の制限が厳しいプロジェクトでは、このデメリットを把握した上で設定をしてください。

4.2.4 Optimize Mesh Data

Optimize Mesh Data はメッシュに不要な頂点データを自動で削除する機能です。不要な頂点データの判定には使用している Shader から自動判定されます。これはランタイム時のメモリ、ストレージともに削減効果があります。「Project Settings -> Player」の Other で設定が可能です。



▲図 4.7 Optimize Mesh Data の設定

ただしこのオプションは頂点データが自動削除されるので便利ですが、予期せぬ不具合を引き起こす可能性があるので注意しましょう。たとえばランタイムで Material や Shader を切り替えた際、アクセスしたプロパティが削除されており描画結果がおかしくなることがあります。他にも Mesh のみをアセットバンドル化する際、Material の設定が正しくなかった場合に不要な頂点データと判定されることもあります。これは Particle System のような Mesh の参照だけを持たせる場合にありがちです。

4.3 Material

Material（マテリアル）はオブジェクトをどのように描画するか決める重要な機能です。身近な機能ですが、使い方を誤ると簡単にメモリリークしてしまいます。本節では安全なマテリアルを使用する方法を紹介します。

パラメーターにアクセスするだけで複製される

マテリアルの最大の注意点は、パラメーターにアクセスするだけで複製されてしまうことです。そして複製されていることに気づきづらいことです。

次のコードを見てみましょう。

▼リスト 4.3 Material の複製される例

```
1: Material material;
2:
3: void Awake()
4: {
5:     material = renderer.material;
6:     material.color = Color.green;
7: }
```

マテリアルの color プロパティに Color.green をセットしているだけの簡単な処理です。この renderer のマテリアルは複製されています。そして複製されたオブジェクトは明示的に Destroy する必要があります。

▼リスト 4.4 複製された Material の削除例

```
1: Material material;
2:
3: void Awake()
4: {
5:     material = renderer.material;
6:     material.color = Color.green;
7: }
8:
9: void OnDestroy()
10: {
11:     if (material != null)
12:     {
13:         Destroy(material)
14:     }
15: }
```

このように複製されたマテリアルを Destroy することでメモリリークを回避できます。

生成したマテリアルの掃除を徹底しよう

動的に生成したマテリアルもメモリリークの原因になりやすいです。生成したマテリアルも使い終わったら確実に Destroy しましょう。

次のサンプルコードを見てみましょう。

▼リスト 4.5 動的に生成した Material の削除例

```
1: Material material;
2:
3: void Awake()
4: {
5:     material = new Material(); // マテリアルを動的生成
6: }
7:
8: void OnDestroy()
9: {
10:    if (material != null)
11:    {
12:        Destroy(material); // 使い終わったマテリアルをDestroy
13:    }
14: }
```

生成したマテリアルは使い終わったタイミング(OnDestroy)で Destroy しましょう。プロジェクトのルールや仕様に合わせて、適切なタイミングでマテリアルは Destroy しましょう。

4.4 Animation

アニメーションは 2D、3D 問わず多く使用されるアセットです。本節では Animation Clip や Animator に関するプラクティスを紹介します。

4.4.1 スキンウェイト数の調整

モーションは内部的にはそれぞれの頂点がどの骨からどれくらい影響を受けているかを計算して位置を更新しています。この位置計算に加味する骨の数をスキンウェイト数、またはインフルエンス数と呼びます。そのためスキンウェイト数を調整することで負荷削減ができます。ただしスキンウェイト数を減らすと見た目がおかしくなる可能性があるので調整する際には検証しましょう。

スキンウェイト数は「Project Settings -> Quality」の Other から設定が可能です。



▲図 4.8 スキンウェイトの調整

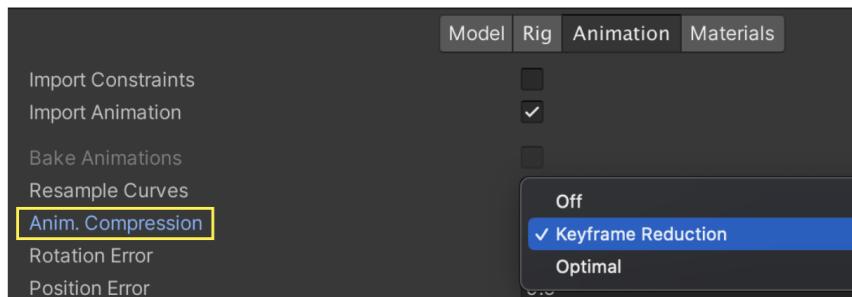
この設定はスクリプトから動的に調整することも可能です。そのため低スペック端末は Skin Weights を 2 に設定し、高スペック端末は 4 に設定するなどの微調整が可能です。

▼リスト 4.6 SkinWeight の設定変更

```
1: // QualitySettingsを丸ごと切り替える方法
2: // 引数の番号は設定画面のLevelsの並び順で上から0、1..となっています
3: QualitySettings.SetQualityLevel(0);
4:
5: // SkinWeightsだけ変更する方法
6: QualitySettings.skinWeights = SkinWeights.TwoBones;
```

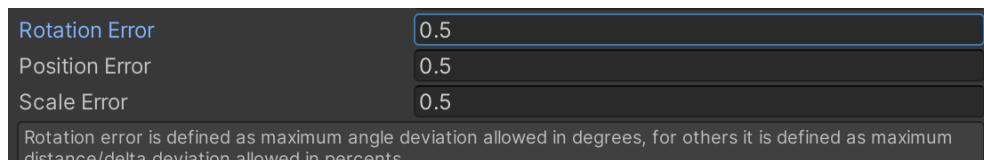
4.4.2 キーの削減

アニメーションファイルはキーの数に依存してストレージとランタイム時のメモリを圧迫します。キー数を削減する方法の 1 つとして Anim. Compression という機能があります。このオプションはモデルのインポート設定からアニメーションタブを選択することで表示されます。Anim. Compression を有効にするとアセットインポート時に不要なキーが自動で削除されます。



▲図 4.9 Anim. Compression 設定画面

Keyframe Reduction は値の変化が少ない場合にキーが削減されます。具体的には直前の曲線と比較して誤差（Error）範囲内に収まっていた場合に削除されます。この誤差範囲は調整することができます。



▲図 4.10 Error の設定

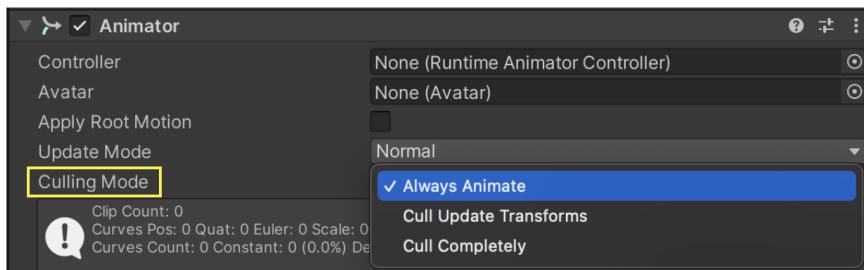
少しややこしいですが Error の設定は項目によって値の単位が違います。Rotation は角度、Position と Scale がパーセントです。キャプチャした画像の許容誤差は Rotation が 0.5 度、Position と Scale は 0.5% となります。詳しいアルゴリズムは Unity のドキュメント^{*1}にあるので気になる方は覗いてみてください。

Optimal はさらにわかりにくいのですが、Dense Curve というフォーマットと、Keyframe Reduction の 2 つの削減方法を比較し、データが小さくなる方を採用します。押さえておくべきポイントとしては、Dense は Keyframe Reduction と比べるとサイズは小さくなります。ただしノイジーになりやすいためアニメーションクオリティが低下する可能性があります。この特性を把握した上で、あとは実際のアニメーションを目視して許容できるか確認していきましょう。

^{*1} <https://docs.unity3d.com/Manual/class-AnimationClip.html#tolerance>

4.4.3 更新頻度の削減

Animator はデフォルト設定では画面に映っていないくとも毎フレーム更新を行います。この更新方法を変更できるカリングモードというオプションがあります。



▲図 4.11 カリングモード

それぞれのオプションの意味は次のようにになります。

▼表 4.1 カリングモードの説明

種類	意味
Always Animate	画面外にいても常に更新を行います。(デフォルト設定)
Cull Update Transform	画面外にいるときに IK や Transform の書き込みを行いません。 ステートマシンの更新は行います。
Cull Completely	画面外にいるとステートマシンの更新を行いません。 アニメーションが完全に止まります。

それぞれのオプションについて注意点があります。まず Cull Completely を設定する場合、Root モーションを利用している際は注意が必要です。たとえば画面外からフレームインするようなアニメーションの場合、画面外にいるためアニメーションは即座に停止されます。その結果いつまでたってもフレームインしなくなります。

次に Cull Update Transform です。これは Transform の更新がスキップされるだけなので、とても使い勝手のよいオプションのように感じます。しかし揺れものといった Transform に依存した処理がある場合は注意が必要です。たとえばキャラクターがフレームアウトすると、そのタイミングのポーズから更新がされなくなります。そして再びフレームインした際に新たなポーズに更新されるため、その弾みで揺れものが大きく動く可能性があります。各オプションの一長一短を把握した上で設定を変更するとよいでしょう。

また、これらの設定を用いても動的にアニメーションの更新頻度を細かく変更することはできません。たとえばカメラから距離が離れているオブジェクトのアニメーションの更新頻度を半分にするなどの最適化です。その場合は AnimationClipPlayable を利用するか、Animator を非アクティブにしたうえで自身で Animator.Update を呼ぶ必要があります。どちらも自前でスクリプトを書く必要がありますが、前者に比べ後者の方が簡単に導入できるでしょう。

4.5 Particle System

ゲームエフェクトはゲーム演出に欠かせません。Unity では Particle System をよく使います。本章ではパフォーマンスチューニング観点で、Particle System の失敗しない使い方、注意点について紹介します。

大事なことは次の 2 点です。

- パーティクルの個数を抑える
- ノイズは重いので注意する

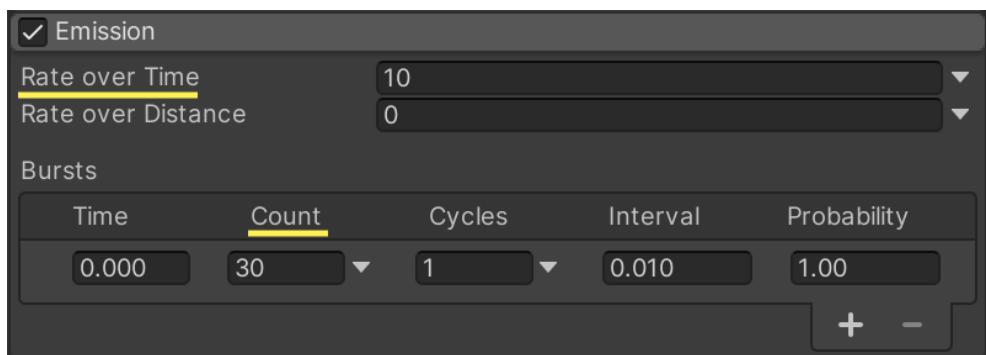
4.5.1 パーティクルの個数を抑える

パーティクルの個数は負荷につながります。Particle System は CPU で動作するパーティクル (CPU パーティクル) のため、パーティクルの個数が多ければ多いほど CPU 負荷は上がります。基本方針として必要最低限のパーティクル数に設定しましょう。必要に応じて個数を調整してみてください。

パーティクルの個数を制限する方法は 2 つです。

- Emission モジュールの放出個数の制限
- メインモジュールの Max Particles で最大放出個数の制限

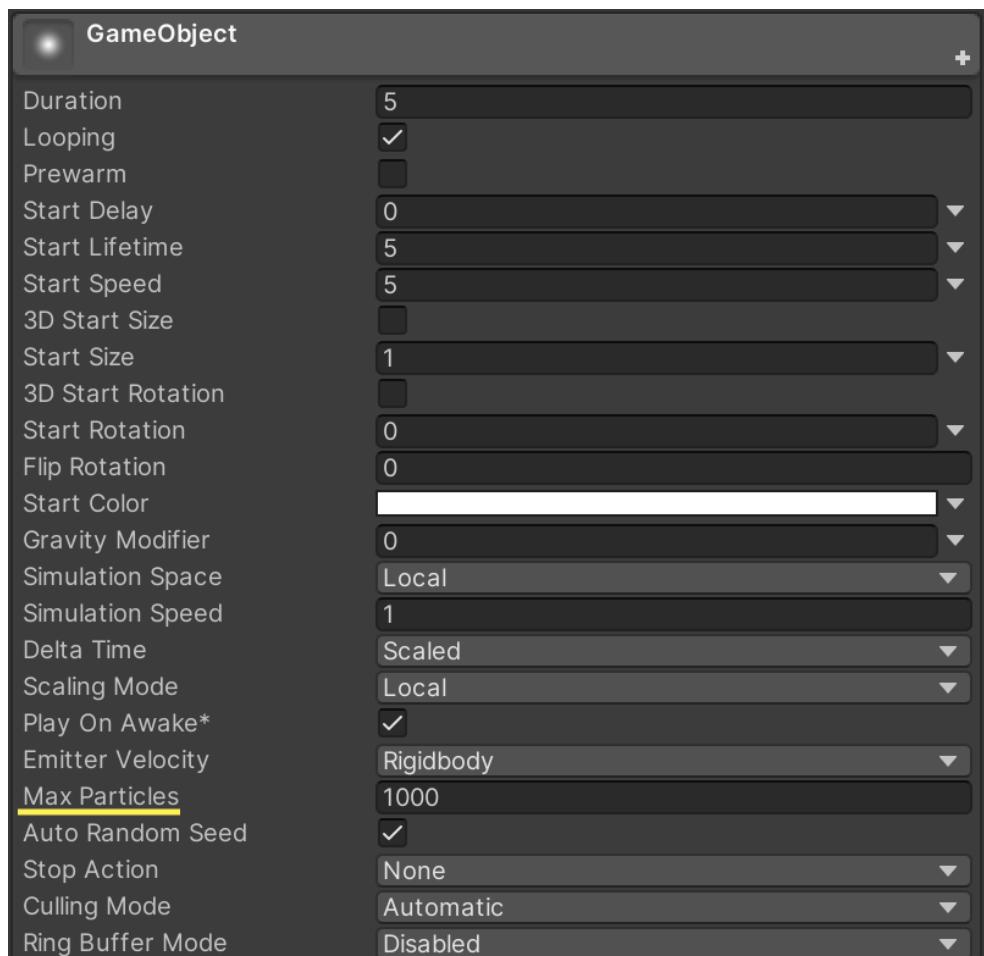
4.5 Particle System



▲図 4.12 Emission モジュールで放出個数の制限

- Rate over Time: 毎秒放出する個数
- Bursts > Count: バーストタイミングで放出する個数

これらの設定を調整して、必要最低限のパーティクル数になるよう設定してください。

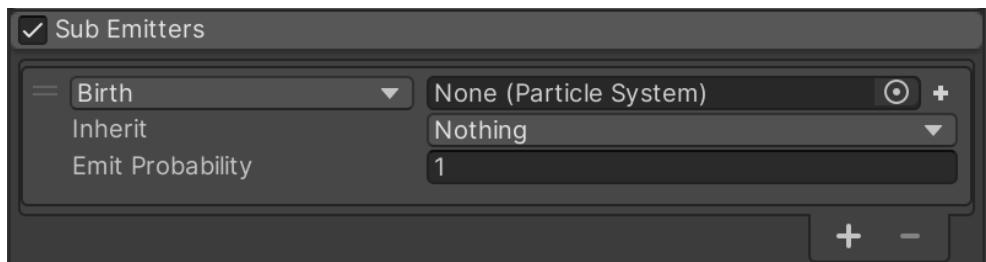


▲図 4.13 Max Particles で放出個数の制限

もう1つの方法はメインモジュールの Max Particles です。上の例では 1000 個以上のパーティクルは放出されなくなります。

Sub Emitters にも注意

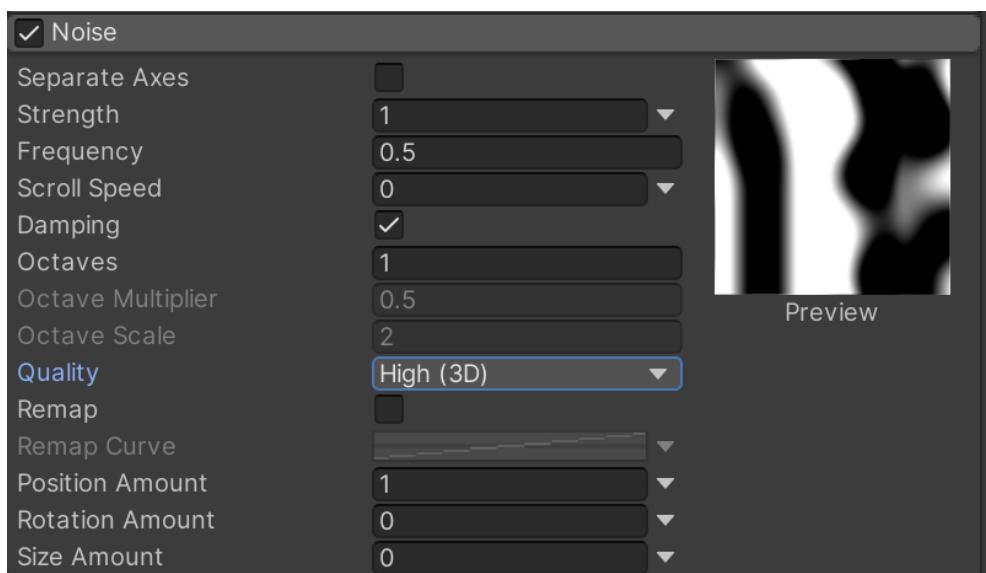
パーティクルの個数を抑える上で、Sub Emitters モジュールも注意が必要です。



▲図 4.14 Sub Emitters モジュール

Sub Emitters モジュールは特定のタイミング（生成時、寿命が尽きた時など）で任意の Particle System を生成します。Sub Emitters の設定によっては、一気にピーク数まで到達してしまうため使用の際には注意しましょう。

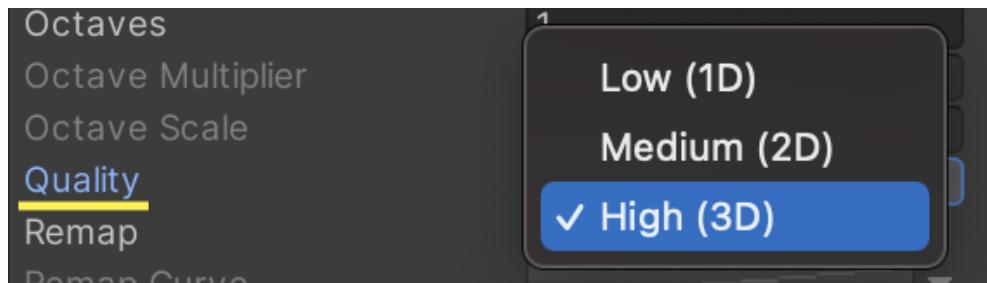
4.5.2 ノイズは重いので注意



▲図 4.15 Noise モジュール

Noise（ノイズ）モジュールの Quality は負荷が上がりやすいです。ノイズは有機的なパーティクルを表現可能で、お手軽にエフェクトのクオリティをあげられるためよ

<使われます。よく使う機能だからこそパフォーマンスには気をつかいたい所です。



▲図 4.16 Noise モジュールの Quality

- Low (1D)
- Midium (2D)
- High (3D)

Quality は次元が上がるほど負荷は上がります。もし Noise が不要であれば Noise モジュールをオフにしましょう。また、ノイズを使う必要があれば、Quality の設定は Low を優先し、要求に応じて Quality をあげていきましょう。

4.6 Audio

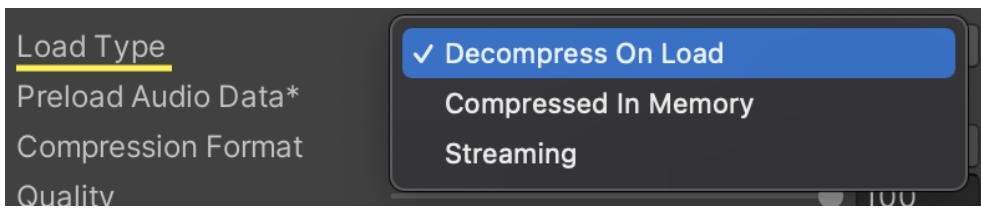
サウンドファイルをインポートしたデフォルト状態はパフォーマンス的には改善ポイントがあります。設定項目は次の 3 点です。

- Load Type
- Compression Format
- Force To Mono

これらをゲーム開発でよく使う BGM、効果音、ボイスで適切な設定にしましょう。

4.6.1 Load Type

サウンドファイル (AudioClip) をロードする方法は 3 種類あります。



▲図 4.17 AudioClip LoadType

- Decompress On Load
- Compressed In Memory
- Streaming

Decompress On Load

Decompress On Load は、非圧縮でメモリにロードします。CPU 負荷が低いため待機時間が小さく再生されます。その反面、メモリを多く使用します。

尺が短くすぐに再生してほしい効果音にオススメです。BGM や尺の長いボイスファイルでの使用は、メモリを多く使用してしまうため注意が必要です。

Compressed In Memory

Compressed In Memory は、AudioClip を圧縮した状態でメモリにロードします。再生するタイミングで展開するということです。つまり CPU 負荷が高く、再生遅延が起きやすいです。

ファイルサイズが大きく、メモリにそのまま展開したくないサウンドや、多少の再生遅延に問題ないサウンドが適しています。ボイスで使うことが多いです。

Streaming

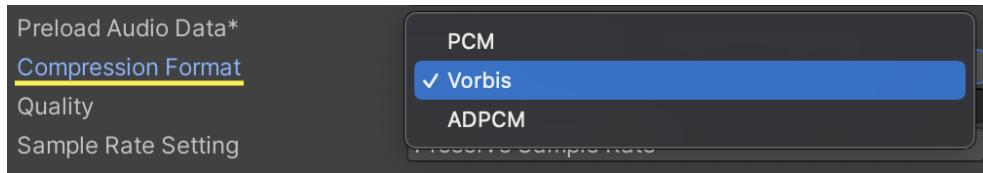
Streaming は、その名の通りロードしながら再生する方式です。メモリ使用量は少ない代わりに CPU 負荷が高くなります。尺の長い BGM での使用がオススメです。

▼表 4.2 ロード方法と主な使用用途まとめ

種類	用途
Decompress On Load	効果音
Compressed In Memory	ボイス
Streaming	BGM

4.6.2 Compression Format

Compression format とは、 AudioClip 自体の圧縮フォーマットです。



▲図 4.18 AudioClip Compression Format

PCM

非圧縮で、メモリを大量に消費します。音質によほどクオリティを求める限り設定することはありません。

ADPCM

PCM に比べてメモリ使用量は 70% 減りますが、クオリティは低くなります。CPU 負荷が Vorbis と比較して格段に小さいのが特徴です。つまり展開スピードが速いため、即時再生や大量に再生するサウンドに適しています。具体的には、足音や衝突音、武器などのノイズを多く含む且つ、大量に再生する必要のあるサウンドです。

Vorbis

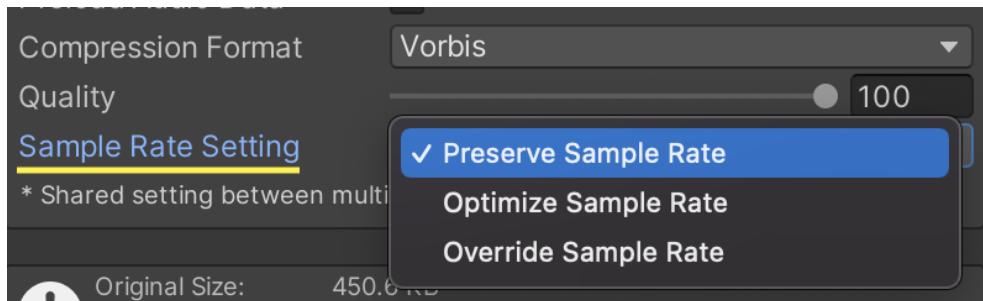
非可逆圧縮フォーマットのため、PCM よりクオリティは下がりますが、ファイルサイズは小さくなります。唯一 Quality を設定できるため、微調整も可能です。全サウンド（BGM、効果音、ボイス）でもっとも使われる圧縮形式です。

▼表 4.3 圧縮方法と主な使用用途まとめ

種類	用途
PCM	使用しない
ADPCM	効果音
Vorbis	BGM、効果音、ボイス

4.6.3 サンプルレートの指定

サンプルレートを指定してクオリティの調節できます。全圧縮フォーマットに対応しています。Sample Rate Setting から 3 種類の方法を選べます。



▲図 4.19 Sample Rate Settings

Preserve Sample Rate

デフォルト設定です。元音源のサンプルレートが採用されます。

Optimize Sample Rate

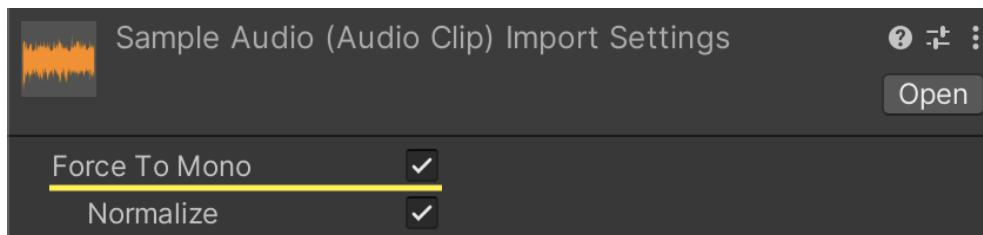
Unity 側で解析され、最高周波数の成分に基づいて自動で最適化されます。

Override Sample Rate

元音源のサンプルレートを上書きします。8,000~192,000Hz まで指定可能です。元の音源より高くしても品質は上がりません。元音源よりサンプルレートを下げたい場合に使用します。

4.6.4 効果音は Force To Mono を設定

Unity はデフォルト状態でステレオ再生しますが、Force To Mono を有効にすることでモノラル再生になります。モノラル再生を強制することで左右それぞれのデータを持たなくて良くなるため、ファイルサイズとメモリサイズは半分になります。



▲図 4.20 AudioClip Force To Mono

効果音はモノラル再生でも問題ないことが多いです。また 3D サウンドもモノラル再生の方が良い場合もあります。検討した上で Force To Mono を有効にするとよいでしょう。パフォーマンスチューニング効果も塵も積もれば山となります。モノラル再生で問題ない場合は積極的に Force To Mono を活用しましょう。

パフォーマンスチューニングとは話がずれますが音声ファイルは無圧縮のものを Unity に取り込みましょう。圧縮済みのものをインポートした場合、Unity 側でコード & 再圧縮を行うので品質の低下が発生します。

4.7 Resources / StreamingAssets

プロジェクトには特別なフォルダーが存在します。とくに次の 2 つはパフォーマンス観点で注意が必要です。

- Resources フォルダー
- StreamingAssets フォルダー

通常 Unity は、シーンやマテリアル、スクリプトなどから参照されたオブジェクトのみがビルドに含まれます。

▼リスト 4.7 スクリプトで参照されたオブジェクトの例

```
1: [SerializeField] GameObject sample; // 参照されたオブジェクトはビルドに含まれる
```

先の特別なフォルダーはルールが違います。格納したファイルはビルドに含まれます。つまり、実際には不要なファイルも格納されていればビルドに含まれ、ビルドサ

イズの膨張につながります。

問題はプログラムから確認することができないということです。不要なファイルを目視で確認しなければならないので時間がかかります。これらのフォルダーには注意してファイル追加しましょう。

しかし、プロジェクトが進行する中で格納ファイルはどうしても増えていきます。中には使用しなくなった不要なファイルが混入することもあるでしょう。結論として、定期的な格納ファイルの見直しをオススメします。

4.7.1 起動時間を遅くする Resources フォルダー

Resources フォルダーに大量のオブジェクトを格納すると、アプリの起動時間が伸びてしまいます。Resources フォルダーは文字列参照でオブジェクトをロードできる昔ながらの便利機能です。

▼リスト 4.8 スクリプトで参照されたオブジェクトの例

```
1: var object = Resources.Load("aa/bb/cc/obj");
```

このようなコードでオブジェクトをロードできます。Resources フォルダーに格納しておけば、スクリプトからオブジェクトにアクセスできるため、多用してしまいがちです。しかし、Resources フォルダーに詰め込みすぎると前述の通りアプリの起動時間が伸びます。原因是 Unity 起動時に、全 Resources フォルダー内の構造を解析し、ルックアップテーブルを作成するからです。できる限り Resources フォルダーの使用は最小限にした方がよいでしょう。

4.8 ScriptableObject

ScriptableObject は YAML のアセットで、テキスト形式としてファイル管理しているプロジェクトが多いと思われます。明示的に [PreferBinarySerialization] Attribute を指定することで保存形式をバイナリ形式に変更できます。おもにデータが大量になるようなアセットの場合、バイナリ形式にすることで書き込み・読み込みのパフォーマンスが向上します。

ただし、当然ながらバイナリ形式の場合マージツールなどでは扱いにくくなります。アセット更新が上書きだけで済む、変更差分をテキストで確認する必要がないようなアセットや、ゲーム開発が完了しデータの変更が行われなくなったアセットについては積極的に [PreferBinarySerialization] を指定するとよいでしょう。

ScriptableObject を使用するにあたって陥りやすいミスは、クラス名とソースコードのファイル名の不一致です。クラスとファイルは同名にする必要があります。クラス作成時には命名に注意しつつ、.asset ファイルが正しくシリアル化され、バイナリ形式で保存されていることを確認しましょう。

▼リスト 4.9 ScriptableObject の実装例

```
1: /*
2: * ソースコードのファイル名がScriptableObjectSample.csのとき
3: */
4:
5: // シリアライズ成功
6: [SerializeField]
7: public sealed class ScriptableObjectSample : ScriptableObject
8: {
9:     ...
10: }
11:
12: // シリアライズ失敗
13: [SerializeField]
14: public sealed class MyScriptableObject : ScriptableObject
15: {
16:     ...
17: }
```



PERFORMANCE TUNING BIBLE

CHAPTER

05

第5章

Tuning Practice
— AssetBundle —

第 5 章

Tuning Practice - AssetBundle

AssetBundle の設定に問題があると、ユーザーの貴重な通信容量や記憶領域を浪費してしまうだけでなく、快適なゲームプレイを妨げてしまうなど多くの問題が生じてしまします。この章では AssetBundle に関する設定や実装の方針について説明します。

5.1 AssetBundle の粒度

依存関係の問題で、AssetBundle をどの程度細かくするかという粒度に関しては、慎重に検討する必要があります。極端に言えば、すべてのアセットを 1 つの AssetBundle に入れる方法と、それぞれのアセットを 1 つずつ AssetBundle にする方法があります。どちらもシンプルな方法ですが、前者の方法は致命的な問題があります。それはアセットを追加したり、1 つのアセットを更新するだけだとしても、ファイル全体を作り直して配布する必要があるためです。アセットの総量が GB 単位になる場合は、更新の負荷が非常に高くなります。

そのため AssetBundle をなるべく分割することになりますが、細かすぎてもさまざまな部分でオーバーヘッドが生じてしまいます。そこで基本的には以下の方針で AssetBundle 化することをオススメします。

- ・同時に使用される前提のアセットは 1 つの AssetBundle にまとめる
- ・複数のアセットから参照されるアセットは別の AssetBundle にする

完璧にコントロールするのは難しいですが、プロジェクト内で粒度に関する何かしらのルールを決めるといいでしょう。

5.2 AssetBundle のロード API

AssetBundle からアセットをロードする際の API として 3 種類あります。

`AssetBundle.LoadFromFile`

ストレージに存在するファイルパスを指定してロードします。最速かつもっと

も省メモリなため、通常はこちらを使います。

`AssetBundle.LoadFromMemory`

メモリにロード済みの AssetBundle データを指定してロードします。AssetBundle を使っている間は非常に大きいデータをメモリに維持する必要があり、メモリ負荷が非常に大きいです。そのため通常は使用しません。

`AssetBundle.LoadFromStream`

AssetBundle データを返す Stream を指定してロードします。暗号化された AssetBundle の復号処理をしながらロードする場合はメモリ負荷を考慮してこちらの API を使います。ただし Stream はシークできる必要があるため、シークに対応できない暗号アルゴリズムを使わないように注意する必要があります。

5.3 AssetBundle のアンロード戦略

AssetBundle が不要になった時点でアンロードしないとメモリを圧迫してしまいますが、その際に使用する API である `AssetBundle.Unload(bool unloadAllLoadedObjects)` の引数 `unloadAllLoadedObjects` は、非常に重要なので開発の初期にどう設定するか決定する必要があります。この引数が `true` の場合、AssetBundle をアンロードする際に、その AssetBundle からロードされたすべてのアセットもアンロードします。`false` の場合はアセットはアンロードされません。

つまり、アセットを使っているあいだ AssetBundle もロードし続けないといけない `true` の方はメモリ負荷が高い一方で、アセットの破棄も確実に行えるので安全性は高いです。一方で `false` の場合はアセットをロードし終わったタイミングで AssetBundle をアンロードできるのでその場のメモリ負荷は低いですが、使い終わったアセットのアンロードを忘れるときメモリリークに繋がったり、同じアセットがメモリ上で複数ロードされたりするため、適切なメモリ管理が求められます。一般に厳密なメモリ管理はシビアになるため、メモリ負荷に余裕がある場合は `AssetBundle.Unload(true)` を推奨します。

5.4 同時にロードされている AssetBundle 数の最適化

`AssetBundle.Unload(true)` の場合はアセットを使用している間は AssetBundle をアンロードできません。そのため実際の場面では、AssetBundle の粒度次第では 100 以上の AssetBundle を同時にロードしている状態といった場面も出てくるかもしれません。この場合気をつけたいのは、ファイルディスクライプターの制限と、Persistent

Manager.Remapper のメモリ使用量です。

ファイルディスクリプターとは、ファイルを読み書きする際に OS 側から割り当てられる操作用の ID です。1つのファイルを読み書きするために1つのファイルディスクリプターが必要で、ファイル操作が完了したタイミングでファイルディスクリプターを解放します。このファイルディスクリプターを1つのプロセスが持てる数の上限が決まっているので、その上限以上のファイルを同時に開くことはできません。*"Too many open files"* というエラーを見かけた場合は、この上限に引っかかったことを示しています。そのため AssetBundle の同時にロードできる数がこの制限に影響を受け、また Unity 側もある程度はファイルを開いているので制限に対して余裕を保つ必要があります。この制限値は OS やバージョンなどによって変化するので、ターゲットとするプラットフォームの値を事前に調査する必要がありますが、一例として iOS や macOS では制限値が 256 のバージョンもあります。また仮に上限に当たったとしても、OS によっては一時的に上限を引き上げることも可能¹なので、必要な場合は実装を検討しましょう。

同時にロードする AssetBundle が多いことの2つ目の問題として、Unity の PersistentManager.Remapper の存在があります。PersistentManager は簡単に言えば、Unity 内部でオブジェクトとデータのマッピング関係を管理している機能です。つまり同時にロードする AssetBundle の数に応じてメモリを使うことが想像できると思いますが、問題は AssetBundle を解放しても使用したメモリ領域は解放されずにプールされるということになります。この性質のために同時ロード数に比例してメモリを圧迫するようになるため、同時ロード数を削減することが重要となってきます。

以上のことから、AssetBundle.Unload(true) の方針で運用する場合は、同時にロードする AssetBundle を最大でも 150~200 程度を目安に、AssetBundle.Unload(false) の方針で運用する場合は最大でも 150 以下を目安とするとよいでしょう。

¹ Linux/Unix 環境では、setrlimit 関数を用いて実行時に制限値を変更することが可能



PERFORMANCE TUNING BIBLE

CHAPTER

06

第6章

Tuning Practice
— Physics —

CyberAgent Smartphone Games & Entertainment

第 6 章

Tuning Practice - Physics

本章では、Physics（物理演算）の最適化について紹介します。

ここで Physics とは PhysX による物理演算を指します。ECS の Unity Physics は扱っていません。

また本章では、3D Physics をメインで取り上げていますが、2D Physics でも参考になる箇所は多いでしょう。

6.1 物理演算のオン・オフ

Unity 標準では、たとえシーン上に 1 つも物理演算に関するコンポーネントが配置されていなかったとしても、物理エンジンによる物理演算の処理は毎フレーム必ず実行されます。そのため、ゲーム内で物理演算を必要としない場合は、物理エンジンをオフにしておくとよいでしょう。

物理エンジンの処理は、`Physics.autoSimulation` に値を設定することでオン・オフを切り替えることができます。たとえば、インゲーム中のみ物理演算を用いて、それ以外では利用しない場合は、インゲーム中のみこの値を `true` に設定しておくとよいでしょう。

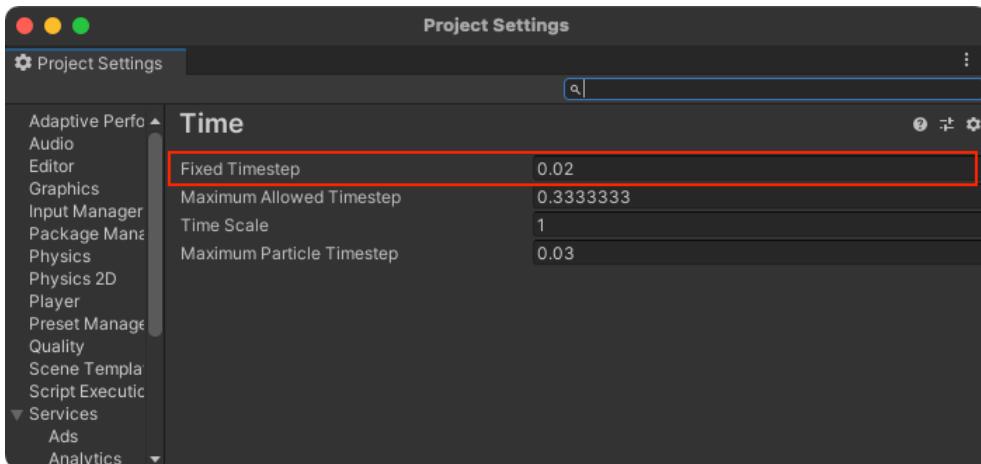
6.2 Fixed Timestep と Fixed Update の頻度の最適化

MonoBehaviour の `FixedUpdate` は `Update` とは違い、固定時間で実行されます。

物理エンジンは前フレームの経過時間に対して、1 フレーム内で `Fixed Update` を複数回呼び出すことで、ゲームの世界の経過時間と物理エンジンの世界の時間をあわせます。そのため `Fixed Timestep` の値が小さいと、より多くの回数 `Fixed Update` が呼び出され、負荷の原因となります。

この時間は、図 6.1 の示すように、Project Settings の **Fixed Timestep** で設定できます。この値の単位は秒となります。デフォルトでは 0.02、つまり 20 ミリ秒が指定されています。

6.2 Fixed Timestep と Fixed Update の頻度の最適化



▲図 6.1 Project Settings の Fixed Timestep 項目

また、スクリプト中からは `Time.fixedDeltaTime` を操作することで変更できます。**Fixed Timestep** は一般に、小さいほど物理演算の精度が上がり、コリジョン抜けなどの問題が発生しにくくなります。そのため、精度と負荷のトレードオフにはなりますが、ゲームの挙動に不備が発生しない範囲でこの値をターゲット FPS に近い値にすることが望ましいです。

6.2.1 Maximum Allowed Timestep

前節のとおり、Fixed Update は前フレームからの経過時間をもとに複数回呼び出されます。

「あるフレームでレンダリング処理が重たい」などの理由で前フレームの経過時間が大きくなつた場合、そのフレームでは通常より多くの回数 Fixed Update が呼び出されることになります。

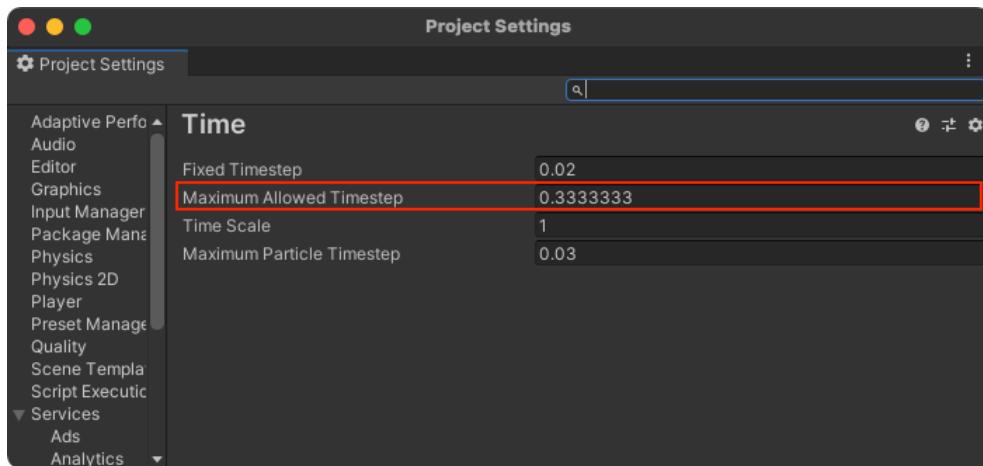
たとえば、**Fixed Timestep** が 20 ミリ秒で前フレームに 200 ミリ秒かかったとき、Fixed Update は 10 回呼び出されることになります。

つまり、あるフレームで処理落ちした場合は次のフレームでの物理演算のコストが高くなります。それが原因でそのフレームも処理落ちするリスクが高くなることで、次フレームでの物理演算も重くなる、といった負のスパイラルに陥る現象が物理エンジンの世界では知られています。

この問題を解決するために Unity では図 6.2 に示すように、Project Settings から **Maximum Allowed Timestep** という、1 フレーム内で物理演算が利用する時間の最

第6章 Tuning Practice - Physics

大値が設定できます。この値はデフォルトで 0.33 秒が設定されていますが、ターゲット FPS に近い値にして Fixed Update の呼び出し回数を制限し、フレームレートを安定させたほうがよいでしょう。



▲図 6.2 Project Settings の Maximum Allowed Timestep 項目

6.3 コリジョン形状の選定

一般的に複雑なコリジョン形状は、当たり判定にかかるコストが高くなります。そのため物理演算のパフォーマンスを最適化するときに、コライダーの形状は可能な限りシンプルな形状を採用することが重要です。

たとえば、人型のキャラクター形状の近似にカプセルコライダーをよく利用しますが、ゲームとして身長が仕様に影響しない場合、スフィアコライダーに置き換えたほうが当たり判定のコストは小さくなります。

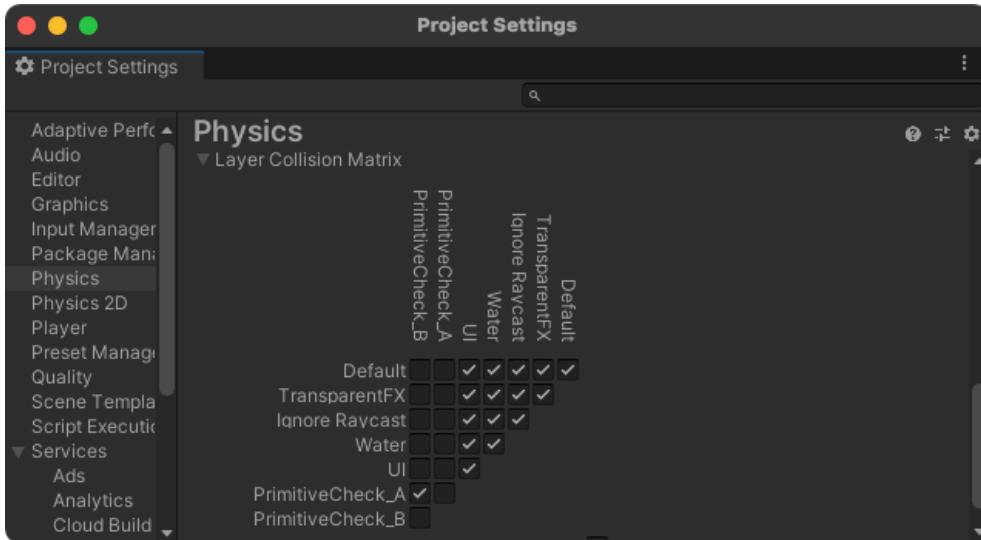
形状の選定は、当たり判定の対象や状況によってコストが異なるので一概に言えないうですが、判定コストを低い順に並べるとスフィアコライダー、カプセルコライダー、ボックスコライダー、メッシュコライダーと覚えておくとよいでしょう。

6.4 衝突マトリックスとレイヤーの最適化

Physics には、どのゲームオブジェクトのレイヤー同士が衝突可能かを定義する「衝突マトリックス」という設定があります。この設定は図 6.3 に示すように、Project

6.5 レイキャストの最適化

Settings の Physics > Layer Collision Matrix から変更できます。



▲図 6.3 Project Settings の Layer Collision Matrix 項目

衝突マトリックスは、2つのレイヤーが交わる位置のチェックボックスにチェックが入っていれば、それらのレイヤーは衝突することを示します。

衝突しないレイヤー同士はブロードフェーズと呼ばれるオブジェクトの大雑把な当たり判定を取る前計算からも除外されるため、この設定を適切に行うのが、衝突する必要がないオブジェクト同士の計算を省くのに最も効率的です。

パフォーマンスを考慮すると、物理演算には専用のレイヤーを用意し、衝突する必要のないレイヤー間のチェックボックスはすべてオフにすることが望ましいです。

6.5 レイキャストの最適化

レイキャストは、飛ばしたレイと衝突したコライダーの衝突情報を取得できる便利な機能ですが、その反面、負荷の原因にもなるので注意が必要です。

6.5.1 レイキャストの種類

レイキャストは、線分との衝突判定を取る `Physics.Raycast` 以外にも、`Physics.SphereCast` などの、その他の形状で判定を取るメソッドが用意されています。

ただし判定を取る形状が複雑になるほど、その負荷が高くなります。パフォーマンスを考慮すると、可能な限り `Physics.Raycast` の利用のみに留めるのが望ましいです。

6.5.2 レイキャストのパラメーターの最適化

`Physics.Raycast` は、レイキャストの始点と向きの 2 つのパラメーター以外に、パフォーマンスの最適化に関わるパラメーターとして、`maxDistance` と `layerMask` があります。

`maxDistance` はレイキャストの判定を行う最大の長さ、つまりレイの長さを指定します。

このパラメーターを省略すると既定値として `Mathf.Infinity` が渡され、非常に長いレイで判定を取ろうとします。このようなレイは、ブロードフェーズに対して悪影響を与えたり、そもそも当たりを取る必要のないオブジェクトと当たり判定を取る可能性があるので、必要以上の距離を指定しないようにします。

また `layerMask` も、当たりを取る必要のないレイヤーのビットは立てないようにします。

衝突マトリックスと同様に、ビットが立っていないレイヤーとはブロードフェーズからも除外されるため、計算コストを抑えることができます。このパラメーターを省略すると、既定値として `Physics.DefaultRaycastLayers` という、`Ignore Raycast` 以外のすべてのレイヤーと衝突する値が指定されるため、こちらも必ず指定します。

6.5.3 RaycastAll と RaycastNonAlloc

`Physics.Raycast` では、衝突したコライダーのうち 1 つの衝突情報が返却されますが、`Physics.RaycastAll` メソッドを利用すると、複数の衝突情報を取得できます。

`Physics.RaycastAll` は衝突情報を、`RaycastHit` 構造体の配列を動的に確保して返却します。そのため、このメソッドを呼び出すたびに GC Alloc が発生し、GC によるスパイクの原因になります。

この問題を回避するために、確保済みの配列を引数に渡すと、結果をその配列に書き込んで返却する `Physics.RaycastNonAlloc` というメソッドが存在します。

パフォーマンスを考慮すると `FixedUpdate` 内では、可能な限り GC Alloc を発生させないようにすべきです。

リスト 6.1 に示すように、結果を書き込む配列をクラスのフィールドやポーリングなどの機構で保持し、その配列を `Physics.RaycastNonAlloc` に渡すことで、配列の初期化時以外の GC Alloc を回避できます。

▼リスト 6.1 Physics.RaycastAllNonAlloc の利用方法

```

1: // レイを飛ばす始点
2: var origin = transform.origin;
3: // レイを飛ばす方向
4: var direction = Vector3.forward;
5: // レイの長さ
6: var maxDistance = 3.0f;
7: // レイが衝突する対象のレイヤー
8: var layerMask = 1 << LayerMask.NameToLayer("Player");
9:
10: // レイキャストの衝突結果を格納する配列
11: // この配列を初期化時に事前に確保したり、
12: // プールに確保されているものを利用する
13: // 事前にレイキャストの結果の最大数を決める必要がある
14: // private const int kMaxResultCount = 100;
15: // private readonly RaycastHit[] _results = new RaycastHit[kMaxResultCount];
16:
17: // すべての衝突情報が配列で返ってくる
18: // 戻り値に衝突個数が返されるので合わせて利用する
19: var hitCount = Physics.RaycastNonAlloc(
20:     origin,
21:     direction,
22:     _results,
23:     layerMask,
24:     query
25: );
26: if (hitCount > 0)
27: {
28:     Debug.Log($"{hitCount}人のプレイヤーとの衝突しました");
29:
30:     // _resultsには配列の0番目から順に衝突情報がはいる
31:     var firstHit = _results[0];
32:
33:     // 個数以上のインデックスを指定するとそれは無効な衝突情報なので注意
34: }
```

6.6 コライダーと Rigidbody

Unity の Physics には、スフィアコライダーやメッシュコライダーなどの衝突について扱う Collider コンポーネントと、物理シミュレーションを剛体ベースで行うための Rigidbody コンポーネントがあります。これらのコンポーネントの組み合わせとその設定によって、3 つのコライダーに分類されます。

Collider コンポーネントがアタッチされ、Rigidbody コンポーネントがアタッチされていないオブジェクトは、静的コライダー（Static Collider）と呼ばれます。

このコライダーは常に同じ場所に留まる、動くことのないジオメトリにのみ使用することを前提に最適化が行われます。

そのため、ゲーム中に静的コライダーの有効・無効を切り替えたり、移動やスケーリ

ングを行なうべきではありません。これらの処理を行うと、内部のデータ構造の変更に伴う再計算が行われ、パフォーマンスを著しく低下させる原因となります。

`Collider` コンポーネントと `Rigidbody` コンポーネントの両方がアタッチされているオブジェクトは、**動的コライダー**（Dynamic Collider）と呼ばれます。

このコライダーは、物理エンジンによって他のオブジェクトと衝突できます。また、スクリプトから `Rigidbody` コンポーネントを操作することによって適用される衝突や力に反応できます。

そのため、物理演算が必要なゲームでは、もっともよく利用されるコライダーになります。

`Collider` コンポーネントと `Rigidbody` コンポーネントの両方がアタッチして、かつ `Rigidbody` の `isKinematic` プロパティを有効にしたコンポーネントは、**キネマティックな動的コライダー**（Kinematic Dynamic Collider）として分類されます。

キネマティックな動的コライダーは、`Transform` コンポーネントを直接操作することで動かせますが、通常の動的コライダーのように `Rigidbody` コンポーネントを操作することで衝突や力を加えて動かせません。

物理演算の実行を切り替えたい場合や、ドアなどのたまに動かしたいが大半は動かない障害物などにこのコライダーを利用することで、物理演算を最適化できます。

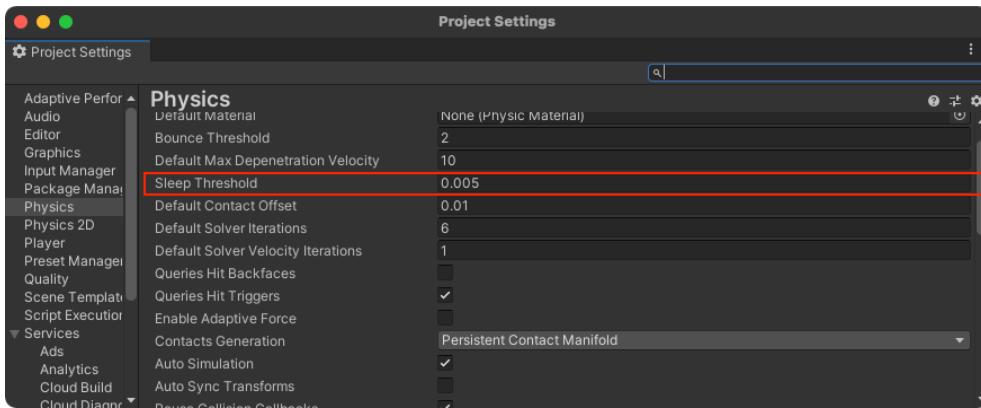
6.6.1 Rigidbody とスリープ状態

物理エンジンでは最適化の一環として、`Rigidbody` コンポーネントをアタッチしたオブジェクトが一定時間動かない場合、そのオブジェクトは休止中と判断して、そのオブジェクトの内部状態をスリープ状態に変更します。スリープ状態に移行すると、外力や衝突などのイベントによって動かない限りは、そのオブジェクトにかかる計算コストが最小限に抑えられます。

そのため、`Rigidbody` コンポーネントがアタッチされたオブジェクトのうち動く必要のないものは、可能な限りスリープ状態に遷移させておくことで物理演算の計算コストを抑えることができます。

`Rigidbody` コンポーネントがスリープ状態へ移行すべきかを判定する際に利用されるしきい値は図 6.4 に示すように、Project Settings の Physics 内部の **Sleep Threshold** で設定できます。また、個別のオブジェクトに対してしきい値を指定したい場合は、`Rigidbody.sleepThreshold` プロパティから設定できます。

6.6 コライダーと Rigidbody



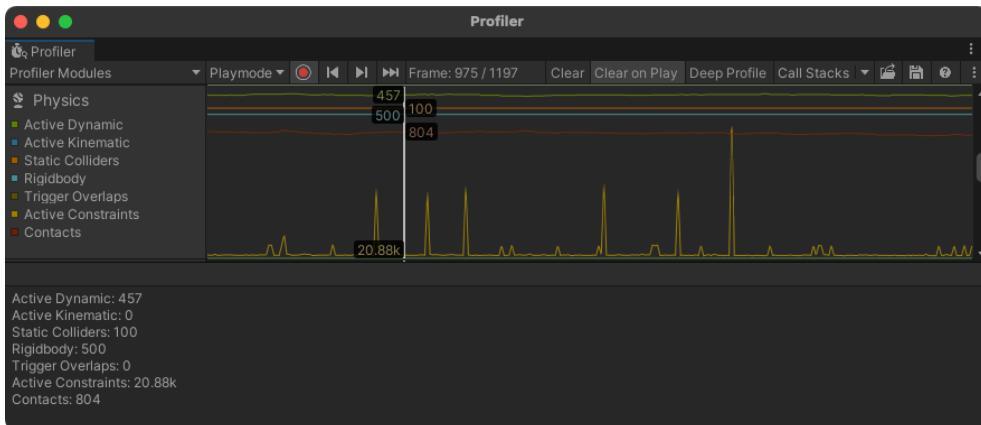
▲図 6.4 Project Settings の Sleep Threshold 項目

Sleep Threshold はスリープ状態に移行する際の質量で正規化された運動エネルギーを表します。

この値を大きくすると、オブジェクトはより早くスリープ状態へ移行するため、計算コストを低く抑えられます。しかし、ゆっくり動いている場合にもスリープ状態へ移行する傾向にあるため、オブジェクトが急停止したように見える場合があります。この値を小さくすると、上記の現象は発生しにくくなりますが、一方でスリープ状態へは移行しづらくなるため、計算コストを抑えられにくい傾向になります。

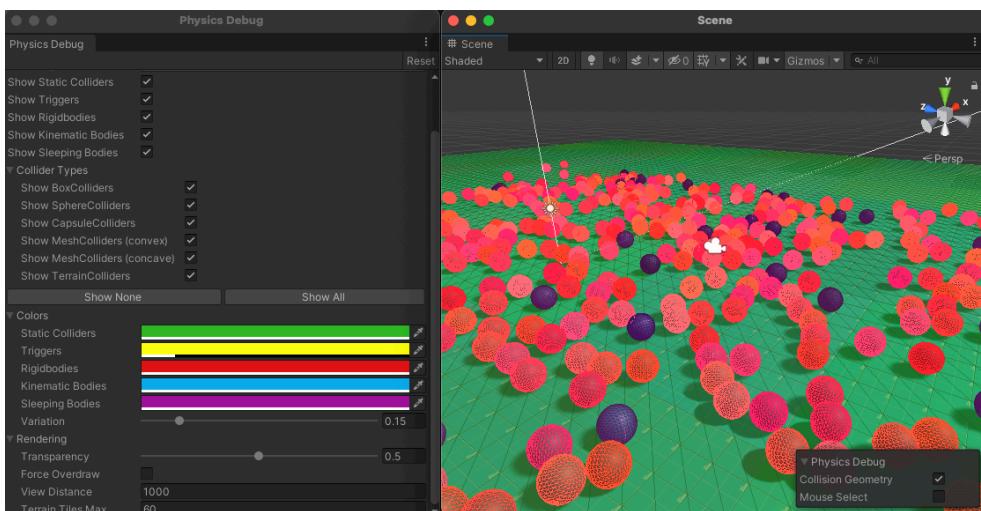
Rigidbody がスリープ状態かどうかは、`Rigidbody.IsSleeping` プロパティで確認できます。シーン上でアクティブな Rigidbody コンポーネントの総数は図 6.5 に示すように、プロファイラーの Physics 項目から確認できます。

第6章 Tuning Practice - Physics



▲図 6.5 Profiler の Physics 項目。アクティブな Rigidbody の個数だけでなく、物理エンジン上のそれぞれの要素数が確認できる。

また、**Physics Debugger** を用いると、シーン上のどのオブジェクトがアクティブ状態なのかを確認できます。



▲図 6.6 Physics Debugger。シーン上のオブジェクトが物理エンジン上どのような状態か、色分けされて表示される。

6.7 衝突検出の最適化

Rigidbody コンポーネントは Collision Detection 項目にて、衝突検出で利用するアルゴリズムが選択できます。

Unity 2020.3 時点で、衝突判定には下記の 4 つがあります。

- Discrete
- Continuous
- Continuous Dynamic
- Continuous Speculative

これらのアルゴリズムは大きく分けて離散的衝突判定と連続的衝突判定の 2 つに分類されます。**Discrete** は離散的衝突判定で、それ以外は連続的衝突判定に属します。

離散的衝突判定は名前のとおり、1 シミュレーションごとにオブジェクトが離散的にテレポート移動し、すべてのオブジェクトが移動後に衝突判定を行います。そのため、とくにオブジェクトが高速に移動している場合に、衝突を見逃してすり抜けを起こす可能性があります。

一方で連続的衝突判定は、移動前後のオブジェクトの衝突を考慮するために、高速に動くオブジェクトのすり抜けを防ぎます。その分、計算コストは離散的衝突判定と比べて高くなります。パフォーマンスを最適化するには、可能な限り **Discrete** を選択できるようにゲームの挙動を作ります。

もし不都合がある場合は、連続的衝突判定を検討します。**Continuous** は動的コライダーと静的コライダーの組み合わせにのみ連続的衝突判定が有効になり、**Continuous Dynamic** は動的コライダー同士でも連続的衝突判定が有効になります。計算コストは Continuous Dynamic のほうが高くなります。

そのためキャラクターがフィールドを走り回る、つまり動的コライダーと静的コライダーの衝突判定のみを考慮する場合は Continuous を選択し、動くコライダー同士のすり抜けも考慮したい場合は Continuous Dynamic を選択します。

Continuous Speculative は動的コライダー同士で連続的衝突判定が有効にもかかわらず Continuous Dynamic より計算コストが低いですが、複数のコライダーが密接している箇所で誤衝突してしまうゴースト衝突 (**Ghost Collision**) と呼ばれる現象が発生するため、導入には注意が必要です。

6.8 その他プロジェクト設定の最適化

これまでに紹介した設定以外で、とくにパフォーマンスの最適化に影響するプロジェクト設定の項目を紹介します。

6.8.1 Physics.autoSyncTransforms

Unity 2018.3 より前のバージョンでは、Physics.Raycast などの物理演算に関する API を呼び出すたびに、Transform と物理エンジンの位置が自動的に同期されていました。

この処理は比較的重たいので、物理演算の API を呼び出したときにスパイクの原因になります。

この問題を回避するために、Unity 2018.3 以降、Physics.autoSyncTransforms という設定が追加されています。この値に `false` を設定すると、上記で説明した、物理演算の API を呼び出したときの Transform の同期処理が行われなくなります。

Transform の同期は物理演算のシミュレーション時の、 `FixedUpdate` が呼び出された後になります。つまり、コライダーを移動してから、そのコライダーの新しい位置に対してレイキャストを実行しても、レイキャストがコライダーに当たらないことを意味します。

6.8.2 Physics.reuseCollisionCallbacks

Unity 2018.3 より前のバージョンでは、`OnCollisionEnter` などの Collider コンポーネントの衝突判定を受け取るイベントが呼び出されるたびに、引数の Collision インスタンスを新たに生成して渡されるため、GC Alloc が発生していました。

この挙動は、イベントの呼び出し頻度によってはゲームのパフォーマンスに悪影響を及ぼすため、2018.3 以降では新たに `Physics.reuseCollisionCallbacks` というプロパティが公開されました。

この値に `true` を設定すると、イベント呼び出し時に渡される Collision インスタンスを内部で使い回すため、GC Alloc が抑えられます。

この設定は 2018.3 以降ではデフォルト値として `true` が設定されているため、比較的新しい Unity でプロジェクトを作成した場合には問題ないですが、2018.3 より前のバージョンでプロジェクトを作成した場合、この値が `false` になっている場合があります。もしこの設定が無効になっている場合は、この設定を有効にしたうえでゲームが正常に動作するようコードを修正すべきです。



PERFORMANCE TUNING BIBLE

CHAPTER

07

第7章

Tuning Practice
— Graphics —

CyberAgent Smartphone Games & Entertainment

第7章

Tuning Practice - Graphics

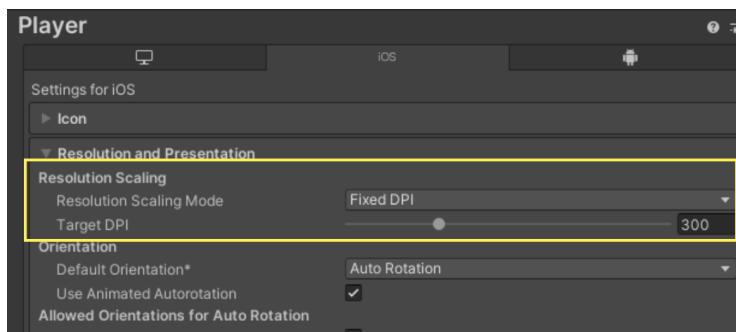
本章では、Unity のグラフィックス機能周辺のチューニング手法について紹介します。

7.1 解像度の調整

レンダリングパイプラインの中でもフラグメントシェーダーのコストはレンダリングする解像度に比例して増加します。とくに昨今のモバイルデバイスはディスプレイの解像度が高く、描画解像度を適切な値に調整する必要があります。

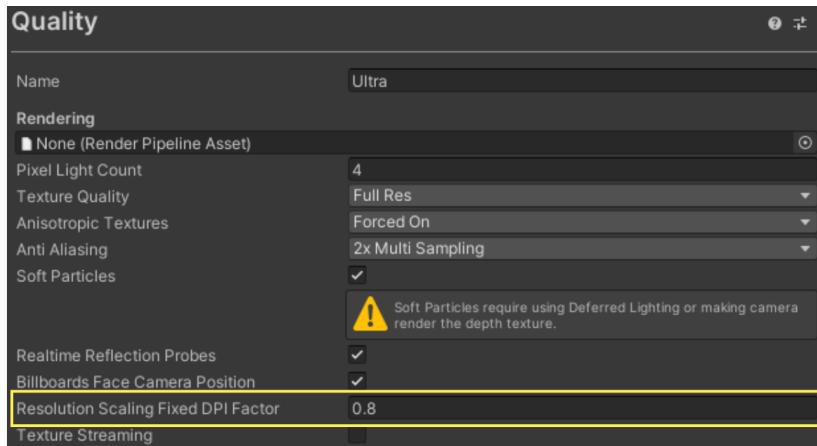
7.1.1 DPI の設定

モバイルプラットフォームの Player Settings の解像度関連の項目に含まれている Resolution Scaling Mode を **Fixed DPI** に設定すると、特定の **DPI (dots per inch)** をターゲットに解像度を落とすことができます。



▲図 7.1 Resolution Scaling Mode

最終的な描画解像度は、Target DPI の値と Quality Settings に含まれる Resolution Scaling DPI Scale Factor の値が乗算されて決定します。



▲図 7.2 Resolution Scaling DPI Scale Factor

7.1.2 スクリプトによる解像度設定

スクリプトから描画解像度を動的に変更するには、`Screen.SetResolution` を呼び出します。

現在の解像度は `Screen.width` や `Screen.height` で取得することができ、DPI は `Screen.dpi` で取得できます。

▼リスト 7.1 Screen.SetResolution

```

1: public void SetupResolution()
2: {
3:     var factor = 0.8f;
4:
5:     // Screen.width, Screen.height で現在の解像度を取得
6:     var width = (int)(Screen.width * factor);
7:     var height = (int)(Screen.height * factor);
8:
9:     // 解像度を設定
10:    Screen.SetResolution(width, height, true);
11: }
```

`Screen.SetResolution` での解像度設定は実機でのみ反映されます。
Editor では変更が反映されないため注意しましょう。

7.2 半透明とオーバードロー

半透明マテリアルの使用はオーバードローの増加の大きな原因となります。オーバードローとは画面の1ピクセルに対して複数回フラグメントの描画が行われることで、フラグメントシェーダーの負荷に比例してパフォーマンスに影響を与えます。

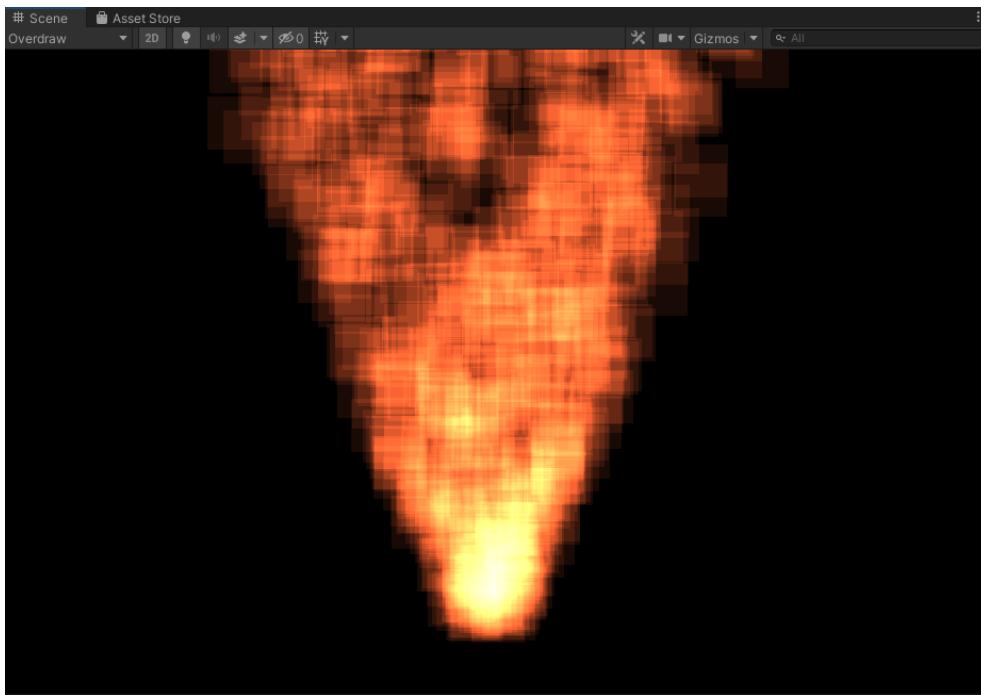
とくに Particle System などで大量に半透明のパーティクルを発生させる場合などにオーバードローが大量に発生することが多いです。

オーバードローによる描画負荷の増加を軽減するには、以下のような方法が考えられます。

- 不要な描画領域を削減する
 - テクスチャが完全に透明な領域も描画対象になるためなるべく減らす
- オーバードローが発生する可能性のあるオブジェクトにはなるべく軽量なシェーダーを使用する
- 半透明マテリアルはなるべく使用しない
 - 不透明マテリアルで擬似的に半透明のような見た目を再現できるディザリングという手法も検討する

Built-in Render Pipeline の Editor では Scene ビューのモードを **Overdraw** に変更することでオーバードローを可視化することができ、オーバードローの調整の基準として有用です。

7.3 ドローコールの削減



▲図 7.3 Overdraw モード

Universal Render Pipeline では、Unity 2021.2 から実装されている **Scene Debug View Modes** によってオーバードローを可視化できます。

7.3 ドローコールの削減

ドローコールの増加は CPU 負荷にしばしば影響を与えますが、Unity にはドローコールの数を削減するための機能がいくつか存在します。

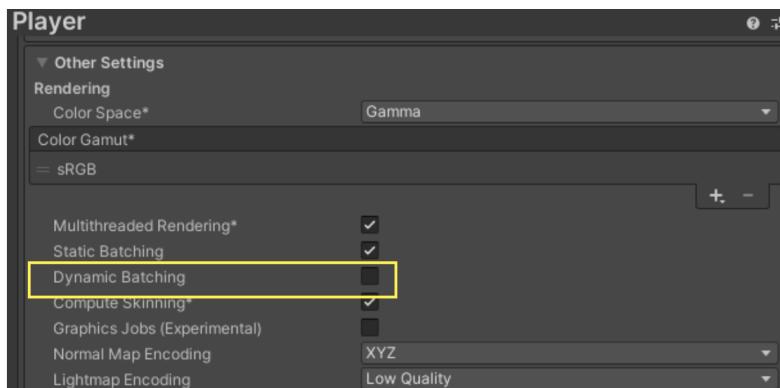
7.3.1 動的バッチング

動的バッチングは、動的なオブジェクトをランタイムでバッチングするための機能です。この機能を使用することで、同じマテリアルを使用している動的なオブジェク

トのドローコールを統合して削減できます。

使用するには、Player Settings から **Dynamic Batching** の項目を有効にします。

また、Universal Render Pipeline では Universal Render Pipeline Asset 内の **Dynamic Batching** の項目を有効にする必要があります。ただ Universal Render Pipeline では Dynamic Batching の使用は非推奨となっています。



▲図 7.4 Dynamic Batching の設定

動的バッティングは CPU コストを使用する処理であるため、オブジェクトに適用させるには多くの条件をクリアする必要があります。以下に主な条件を記載します。

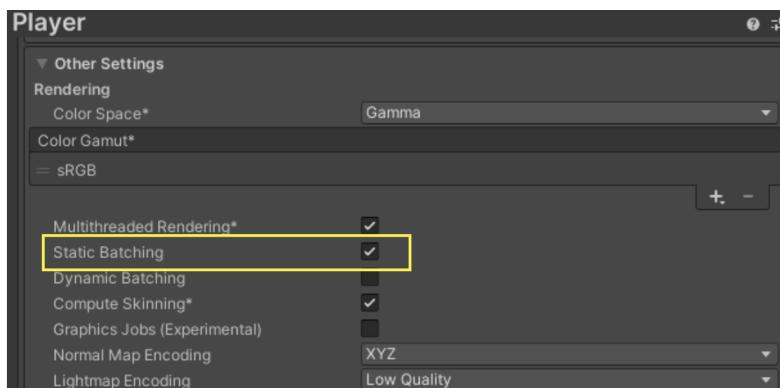
- 同一のマテリアルを参照している
- MeshRenderer か Particle System で描画を行っている
 - SkinnedMeshRenderer などの他のコンポーネントでは動的バッティングの対象とならない
- メッシュの頂点数が 300 以下である
- マルチパスを使用していない
- リアルタイムシャドウの影響を受けていない

動的バッティングは CPU の定常負荷に影響を与えるため、推奨されない場合があります。後述する **SRP Batcher** を使用すると動的バッティングに近い効果を得ることができます。

7.3.2 静的バッ칭

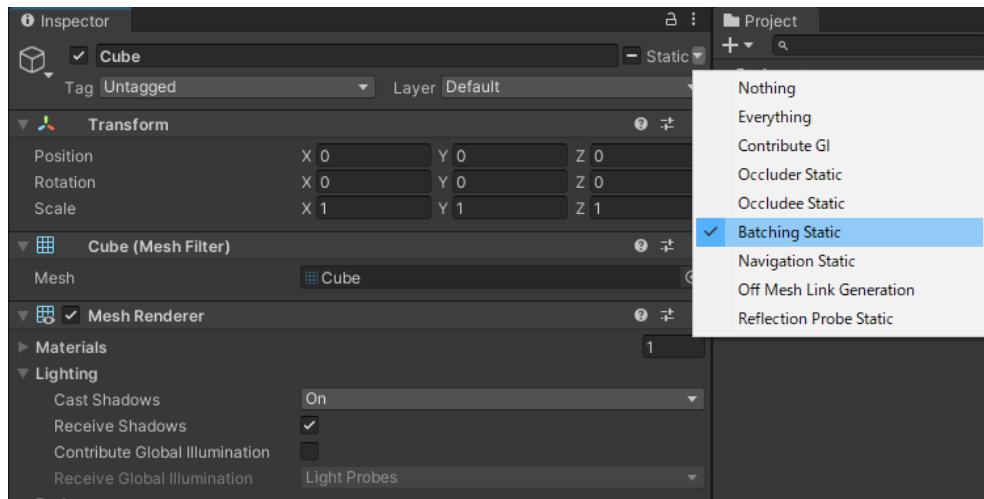
静的バッ칭は、シーン内で動かないオブジェクトをバッ칭するための機能です。この機能を使用することで、同一マテリアルを使用している静的なオブジェクトのドローコールを削減できます。

動的バッ칭と同様に Player Settings から **Static Batching** を有効にすることで使用できます。



▲図 7.5 Static Batching の設定

オブジェクトを静的バッ칭の対象とするには、オブジェクトの **static** フラグを有効にする必要があります。具体的には static フラグの中の **Batching Static** というサブフラグを有効にする必要があります。



▲図 7.6 Batching Static

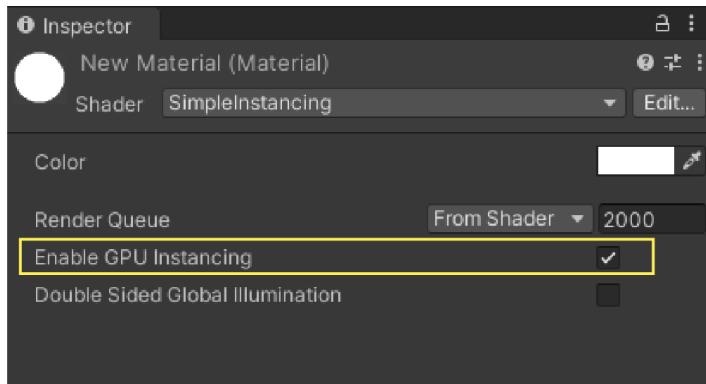
静的バッティングは動的バッティングとは違いランタイムでの頂点の変換処理を行わないので低負荷で実行できます。しかし、バッチ処理によって結合されたメッシュ情報を保存するためメモリを多く消費することに注意が必要です。

7.3.3 GPU インスタンシング

GPU インスタンシングとは同一メッシュ・同一マテリアルのオブジェクトを効率的に描画するための機能です。草や木のように同じメッシュを複数回描画する場合のドローコールの削減が期待できます。

GPU インスタンシングを使用するには、マテリアルの Inspector から **Enable Instancing** を有効にします。

7.3 ドローコールの削減



▲図 7.7 Enable Instancing

GPU インスタンシングを使用できるシェーダーを作成するためにはいくつか専用の対応が必要となります。以下に Built-in Render Pipeline で GPU インスタンシングを使用するための最低限の実装を行ったシェーダーコードの例を記載します。

▼リスト 7.2 GPU インスタンシングに対応したシェーダー

```
1: Shader "SimpleInstancing"
2: {
3:     Properties
4:     {
5:         _Color ("Color", Color) = (1, 1, 1, 1)
6:     }
7:
8:     CGINCLUDE
9:
10:    #include "UnityCG.cginc"
11:
12:    struct appdata
13:    {
14:        float4 vertex : POSITION;
15:        UNITY_VERTEX_INPUT_INSTANCE_ID
16:    };
17:
18:    struct v2f
19:    {
20:        float4 vertex : SV_POSITION;
21:        // フラグメントシェーダーでINSTANCED_PROPにアクセスしたい場合にのみ必要
22:        UNITY_VERTEX_INPUT_INSTANCE_ID
23:    };
24:
25:    UNITY_INSTANCING_BUFFER_START(Props)
26:        UNITY_DEFINE_INSTANCED_PROP(float4, _Color)
27:    UNITY_INSTANCING_BUFFER_END(Props)
28:
29:    v2f vert(appdata v)
```

```
30:      {
31:          v2f o;
32:
33:          UNITY_SETUP_INSTANCE_ID(v);
34:
35:          // フラグメントシェーダーでINSTANCED_PROPにアクセスしたい場合にのみ必要
36:          UNITY_TRANSFER_INSTANCE_ID(v, o);
37:
38:          o.vertex = UnityObjectToClipPos(v.vertex);
39:          return o;
40:      }
41:
42:      fixed4 frag(v2f i) : SV_Target
43:      {
44:          // フラグメントシェーダーでINSTANCED_PROPにアクセスしたい場合にのみ必要
45:          UNITY_SETUP_INSTANCE_ID(i);
46:
47:          float4 color = UNITY_ACCESS_INSTANCED_PROP(Props, _Color);
48:          return color;
49:      }
50:
51:      ENDCG
52:
53:      SubShader
54:      {
55:          Tags { "RenderType"="Opaque" }
56:          LOD 100
57:
58:          Pass
59:          {
60:              CGPROGRAM
61:              #pragma vertex vert
62:              #pragma fragment frag
63:              #pragma multi_compile_instancing
64:              ENDCG
65:          }
66:      }
67: }
```

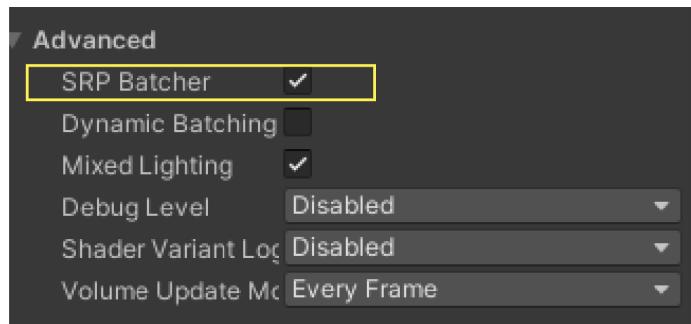
GPU インスタンシングは同一のマテリアルを参照しているオブジェクトにのみ作用しますが、各インスタンスごとのプロパティを設定できます。上記のシェーダーコードのように対象のプロパティを `UNITY_INSTANCING_BUFFER_START(Props)` と `UNITY_INSTANCING_BUFFER_END(Props)` で囲むことで個別に変更するプロパティとして設定できます。

このプロパティを C# で **MaterialPropertyBlock** の API を使用して変更することで個別のカラーなどのプロパティを設定できます。ただ、あまりに大量のインスタンスに対して **MaterialPropertyBlock** を使用すると、**MaterialPropertyBlock** でのアクセスが CPU のパフォーマンスに影響を与えることがあるので気をつけましょう。

7.3.4 SRP Batcher

SRP Batcher とは、**Scriptable Render Pipeline** でのみ使用できる描画の CPU コストを削減するための機能です。この機能を使用することで、同一のシェーダーバリエントを使用する複数のシェーダーのセットパスコールをまとめて処理することができるようになります。

SRP Batcher を使用するには、**Scriptable Render Pipeline Asset** の Inspector から **SRP Batcher** の項目を有効にします。



▲図 7.8 SRP Batcher の有効化

また、ランタイムでは以下の C#コードで SRP Batcher の有効・無効を変更できます。

▼リスト 7.3 SRP Batcher の有効化

```
1: GraphicsSettings.useScriptableRenderPipelineBatching = true;
```

シェーダーを SRP Batcher に対応させるには以下の 2 点の条件をクリアする必要があります。

1. オブジェクトごとに定義されるビルトインのプロパティを **UnityPerDraw** という 1 つの CBUFFER に定義する
2. マテリアルごとのプロパティを **UnityPerMaterial** という 1 つの CBUFFER に定義する

UnityPerDraw に関しては、Universal Render Pipeline などのシェーダーでは基本的にデフォルトで対応されていますが、**UnityPerMaterial** の CBUFFER の設定は自分で行う必要があります。

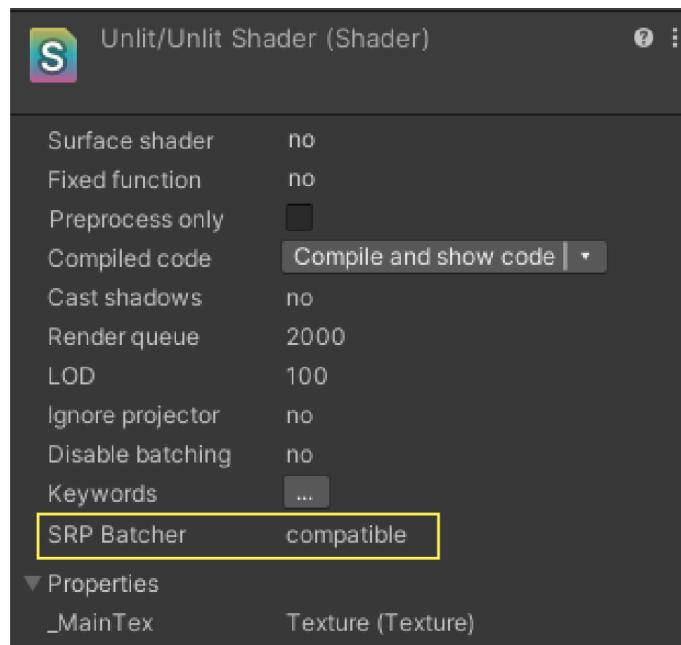
以下のように、マテリアルごとのプロパティを CBUFFER_START(UnityPerMaterial) と CBUFFER_END で囲みます。

▼リスト 7.4 UnityPerMaterial

```
1: Properties
2: {
3:     _Color1 ("Color 1", Color) = (1,1,1,1)
4:     _Color2 ("Color 2", Color) = (1,1,1,1)
5: }
6:
7: CBUFFER_START(UnityPerMaterial)
8:
9: float4 _Color1;
10: float4 _Color2;
11:
12: CBUFFER_END
```

以上の対応で SRP Batcher に対応したシェーダーを作成できますが、Inspector から該当のシェーダーが SRP Batcher に対応しているかどうか確認することもできます。

シェーダーの Inspector の **SRP Batcher** の項目が **compatible** となっていたら SRP Batcher に対応しており、**not compatible** となっていたら対応していないことがわかります。



▲図 7.9 SRP Batcher に対応しているシェーダー

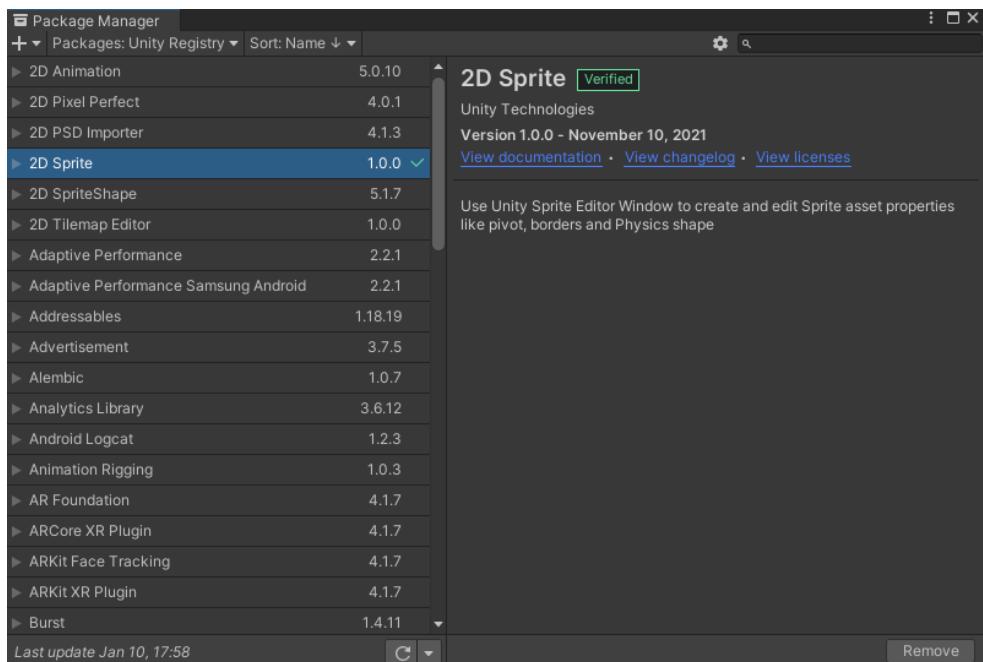
7.4 SpriteAtlas

2D ゲームや UI では多くのスプライトを使用して画面を構築することがしばしばあります。そういった場合に大量のドローコールを発生させないための機能が **SpriteAtlas** です。

SpriteAtlas を使用すると、複数のスプライトを 1 つのテクスチャとしてまとめてすることでドローコールが削減できます。

SpriteAtlas を作成するには、まず Package Manager から **2D Sprite** をプロジェクトにインストールする必要があります。

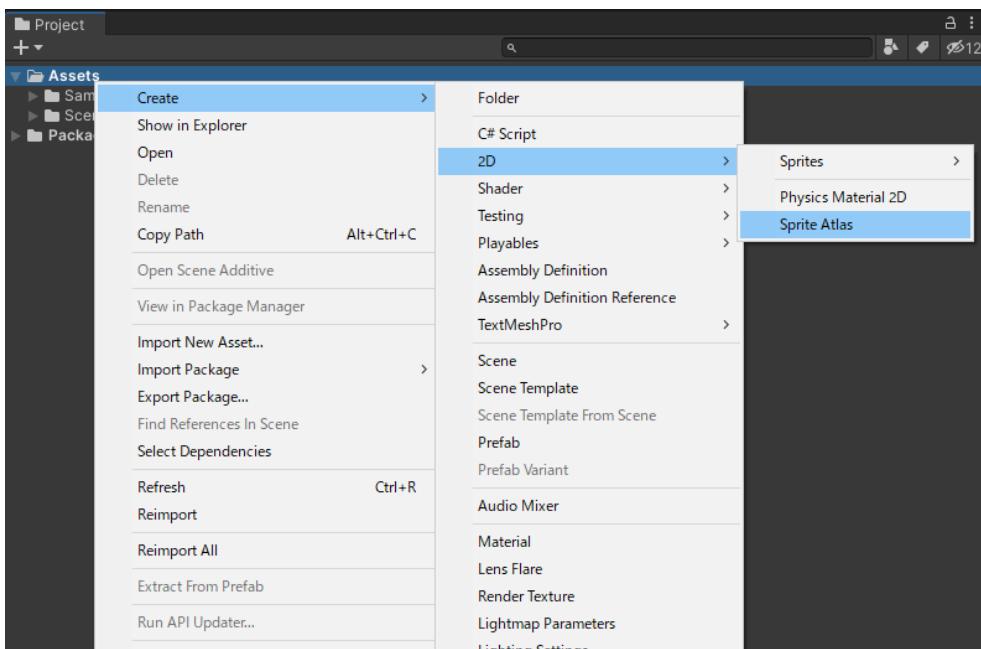
第7章 Tuning Practice - Graphics



▲図 7.10 2D Sprite

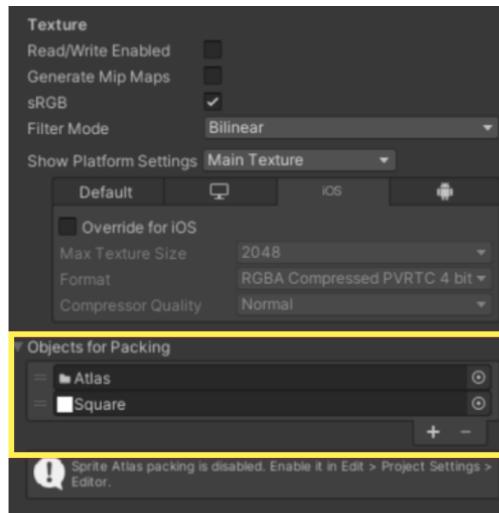
インストール後、Project ビューで右クリックして「Create -> 2D -> Sprite Atlas」を選択し、SpriteAtlas のアセットを作成します。

7.4 SpriteAtlas



▲図 7.11 SpriteAtlas の作成

アトラス化する.sprite ファイルを指定するには、SpriteAtlas の Inspector から **Objects for Packing** の項目に.sprite ファイルを含むフォルダーを設定します。



▲図 7.12 Objects for Packing の設定

以上の設定を行うことでビルド時や Unity Editor の再生時にアトラス化の処理が行われ、対象のスプライトの描画の際は SpriteAtlas で統合されたテクスチャが参照されるようになります。

以下のようなコードで SpriteAtlas から直接スプライトを取得することも可能です。

▼リスト 7.5 SpriteAtlas から Sprite をロードする

```
1: [SerializeField]
2: private SpriteAtlas atlas;
3:
4: public Sprite LoadSprite(string spriteName)
5: {
6:     // Spriteの名前を引数にしてSpriteAtlasからSpriteを取得する
7:     var sprite = atlas.GetSprite(spriteName);
8:     return sprite;
9: }
```

SpriteAtlas に含まれる Sprite を 1 つロードするとアトラス全体のテクスチャが読み込まれるため、1 つだけロードするよりも多くのメモリを消費します。そのため、SpriteAtlas を適切に分割するなど注意して使用する必要があります。

この節は SpriteAtlas V1 をターゲットにして記載しています。SpriteAtlas V2 ではアトラス化する対象のスプライトのフォルダー指定ができなかったりなど動作に大きく変更が加わる可能性があります。

7.5 カリング

Unity では、最終的に画面に表示されない部分の処理を事前に省くためのカリングという処理が行われます。

7.5.1 視錐台カリング

視錐台カリングとは、カメラの描画範囲となる視錐台の外側にあるオブジェクトを描画対象から省くための処理です。これによりカメラの範囲外のオブジェクトは描画の計算が行われないようになります。

視錐台カリングは何も設定せずともデフォルトで行われています。頂点シェーダーの負荷が高いオブジェクトなどの場合は、メッシュを適切に分割することでカリングの対象とし描画コストを下げる手法も有効です。

7.5.2 背面カリング

背面カリングとは、カメラから見えない（はずの）ポリゴンの裏側を描画から省く処理です。ほとんどのメッシュは閉じている（表側のポリゴンのみがカメラに映る）ため裏側は描画する必要がありません。

Unity ではシェーダーに明記しない場合ポリゴンの背面がカリングの対象となっていますが、明記することでカリングの設定を切り替えることが可能です。SubShader 内に以下のように記述します。

▼リスト 7.6 カリングの設定

```

1: SubShader
2: {
3:     Tags { "RenderType"="Opaque" }
4:     LOD 100
5:
6:     Cull Back // Front, Off
7:
8:     Pass

```

```
9:      {
10:         CGPROGRAM
11:         #pragma vertex vert
12:         #pragma fragment frag
13:         ENDCG
14:     }
15: }
```

Back、Front、Off の 3 つの設定がありますが、それぞれの効果は以下のようになっています。

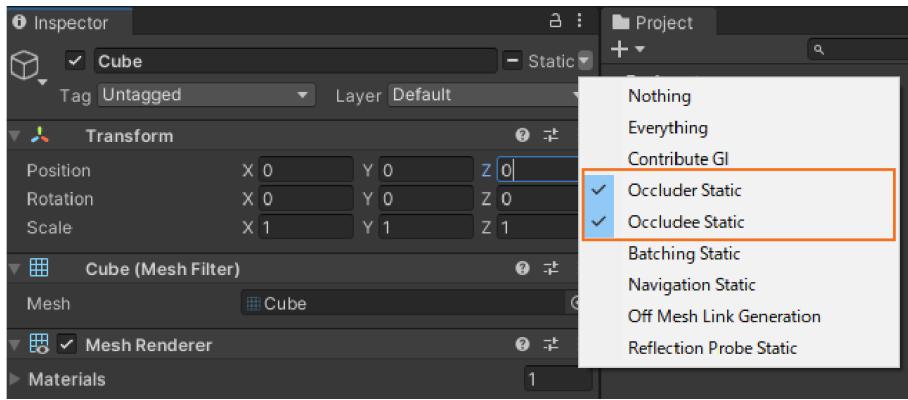
- **Back** - 視点と反対側のポリゴンを描画しない
- **Front** - 視点と同じ方向のポリゴンを描画しない
- **Off** - 背面カリングを無効化し、すべての面を描画する

7.5.3 オクルージョンカリング

オクルージョンカリングとは、オブジェクトに遮蔽されてカメラに映らないオブジェクトを描画対象から省く処理です。この機能は事前にベイクした遮蔽判定用のデータをもとにランタイムでオブジェクトが遮蔽されているか判定し、遮蔽されているオブジェクトを描画対象から外します。

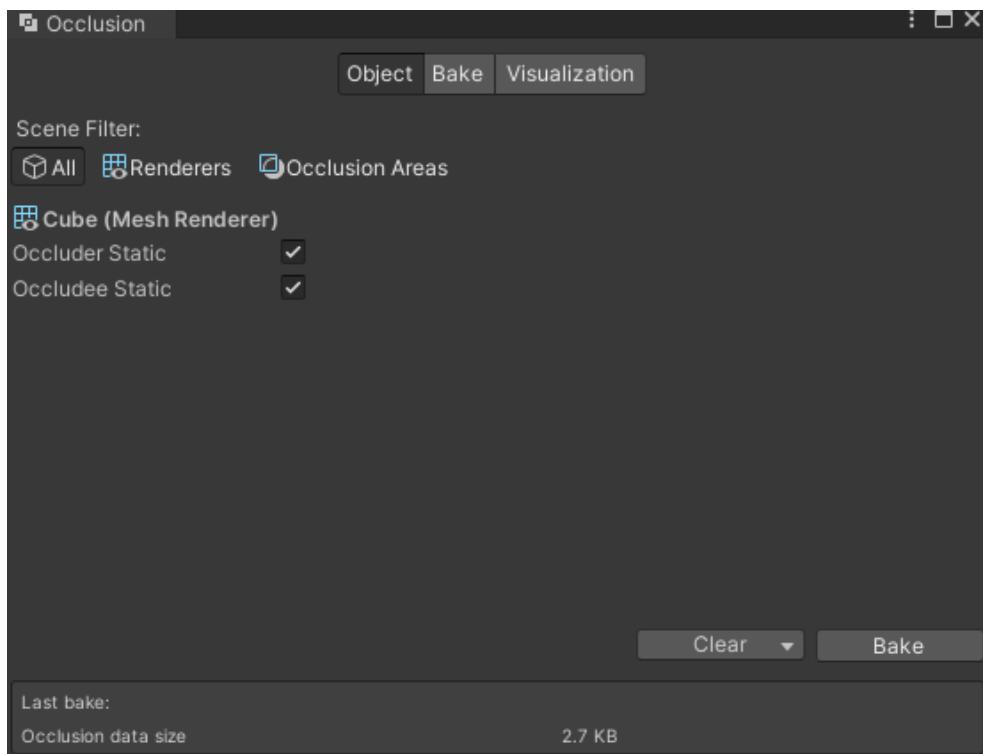
オブジェクトをオクルージョンカリングの対象とするには、Inspector の static フラグから **Occluder Static** または **Occludee Static** を有効にします。Occluder Static を無効にし Occludee Static を有効にした場合はオブジェクトを遮蔽する側としては判定されなくなり、遮蔽される側のみとして処理が行われるようになります。逆の場合は遮蔽される側として判定されなくなり、遮蔽する側のみとして処理されます。

7.5 カリング



▲図 7.13 オクルージョンカリングの static フラグ

オクルージョンカリング用の事前ベイクを行うため、**Occlusion Culling** ウィンドウを表示します。このウィンドウでは各オブジェクトの static フラグの変更やベイクの設定変更などを行うことができます。**Bake** ボタンを押すことでベイクを行うことができます。



▲図 7.14 Occlusion Culling ウィンドウ

オクルージョンカリングは描画コストを削減する代わりにカリング処理のためにCPUへ負荷をかけるため、各負荷のバランスを見て適切に設定を行う必要があります。

オクルージョンカリングで削減されるのはオブジェクトの描画処理のみで、リアルタイムシャドウの描画などの処理はそのまま行われます。

7.6 シェーダー

シェーダーはグラフィックス表現に非常に有効ですが、しばしばパフォーマンスの問題を引き起こします。

7.6.1 浮動小数点数型の精度を下げる

GPU（とくにモバイルプラットフォーム）の計算速度は大きいデータ型より小さいデータ型のほうが速くなります。そのため、置き換え可能な場合は浮動小数点数型を **float 型（32bit）** から **half 型（16bit）** に置き換えることが有効です。

深度計算など精度が必要な場合は float 型を使うべきですが、Color の計算などでは精度を落としてしまっても結果的な見た目に大きな差異は起こりづらいです。

7.6.2 頂点シェーダーで計算を行う

頂点シェーダーの処理はメッシュの頂点数分だけ実行され、フラグメントシェーダーの処理は最終的に書き込まれるピクセル数分実行されます。一般的に頂点シェーダーの実行回数はフラグメントシェーダーよりも少ないことが多く、そのため複雑な計算は可能な限り頂点シェーダーで行うのが良いでしょう。

頂点シェーダーの計算結果はシェーダーセマンティクスを介してフラグメントシェーダーに渡されますが、ここで渡される値は補間されたものであり、フラグメントシェーダーで計算した場合と見た目が違う可能性に注意する必要があります。

▼リスト 7.7 頂点シェーダーによる事前計算

```

1: CGPROGRAM
2: #pragma vertex vert
3: #pragma fragment frag
4:
5: #include "UnityCG.cginc"
6:
7: struct appdata
8: {
9:     float4 vertex : POSITION;
10:    float2 uv : TEXCOORD0;
11: };
12:
13: struct v2f
14: {
15:     float2 uv : TEXCOORD0;
16:     float3 factor : TEXCOORD1;
17:     float4 vertex : SV_POSITION;
18: };
19:
20: sampler2D _MainTex;
21: float4 _MainTex_ST;
22:
23: v2f vert (appdata v)
24: {
25:     v2f o;
26:     o.vertex = UnityObjectToClipPos(v.vertex);

```

```
27:     o.uv = TRANSFORM_TEX(v.uv, _MainTex);
28:
29:     // 複雑な事前計算を行う
30:     o.factor = CalculateFactor();
31:
32:     return o;
33: }
34:
35: fixed4 frag (v2f i) : SV_Target
36: {
37:     fixed4 col = tex2D(_MainTex, i.uv);
38:
39:     // 頂点シェーダーで計算した値をフラグメントシェーダーで使用する
40:     col *= i.factor;
41:
42:     return col;
43: }
44: ENDCG
```

7.6.3 テクスチャに情報を事前に仕込む

シェーダー内の複雑な計算の結果がもし外部の値によって変化しないのであれば、テクスチャの要素として事前計算した結果を格納しておくことも有効な手段です。

方法としては、Unity で専用のテクスチャを生成するツールを実装したり各種 DCC ツールの拡張機能として実装するなどが考えられます。すでに使用しているテクスチャのアルファチャンネルがもし使用されていなければそこに書き込んだり、専用のテクスチャを用意するのが良いでしょう。

たとえば、カラーグレーディングに使用される **LUT**（色対応表）は、各ピクセルの座標が各カラーに対応するテクスチャに対して事前に色調補正をかけます。そのテクスチャをシェーダー内で元のカラーをもとにサンプリングすることで、事前にかけた色調補正をもとのカラーに対してかけたのとほぼ同一の結果を得ることができます。



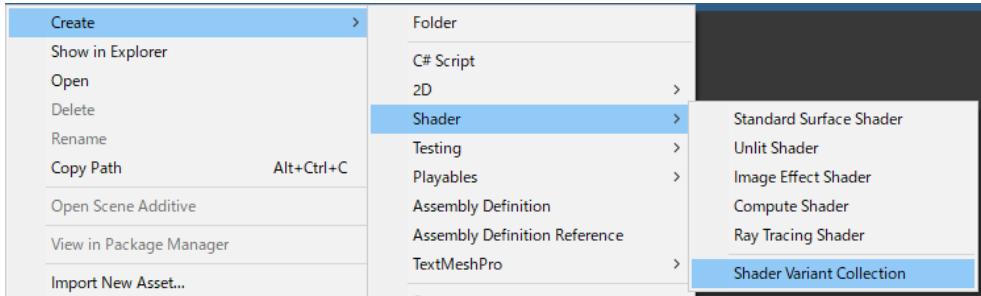
▲図 7.15 色調補正前の LUT テクスチャ (1024x32)

7.6.4 ShaderVariantCollection

ShaderVariantCollection を使用すると、シェーダーを使用する前にコンパイルしスパイクを防ぐことができます。

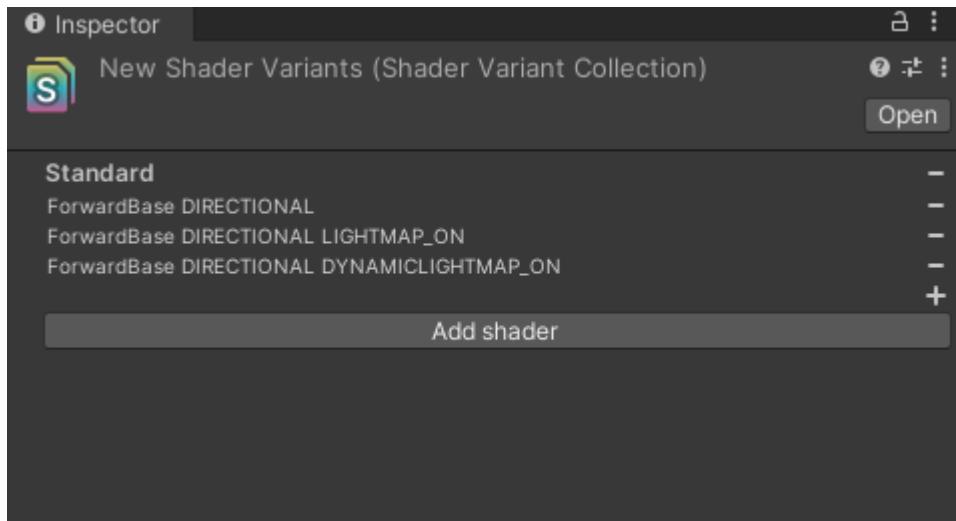
ShaderVariantCollection ではゲーム内で使用するシェーダーバリアントのリストをア

セットとして保持できます。Project ビューから「Create -> Shader -> Shader Variant Collection」を選択して作成します。



▲図 7.16 ShaderVariantCollection の作成

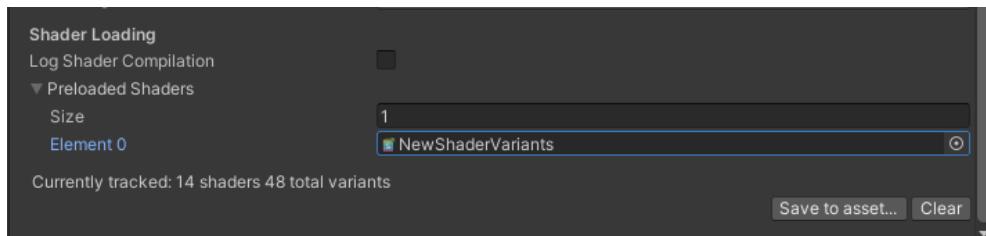
作成した ShaderVariantCollection の Inspector ビューから Add Shader を押下して対象のシェーダーを追加し、さらにシェーダーに対してどのバリアントを追加するかを選択します。



▲図 7.17 ShaderVariantCollection の Inspector

ShaderVariantCollection を Graphics Settings の **Shader preloading** の項目内の **Preloaded Shaders** に追加することで、アプリケーションの起動時にコンパイルする

シェーダーバリエントを設定できます。



▲図 7.18 Preloaded Shaders

また、スクリプトから **ShaderVariantCollection.WarmUp()** を呼び出すことで該当の **ShaderVariantCollection** に含まれるシェーダーバリエントを明示的に事前コンパイルすることも可能です。

▼リスト 7.8 ShaderVariantCollection.WarmUp

```
1: public void PreloadShaderVariants(ShaderVariantCollection collection)
2: {
3:     // 明示的にシェーダーバリエントを事前コンパイルする
4:     if (!collection.isWarmedUp)
5:     {
6:         collection.WarmUp();
7:     }
8: }
```

7.7 ライティング

ライティングはゲームのアート表現において非常に重要な要素の1つですが、パフォーマンスに大きく影響を与えることが多いです。

7.7.1 リアルタイムシャドウ

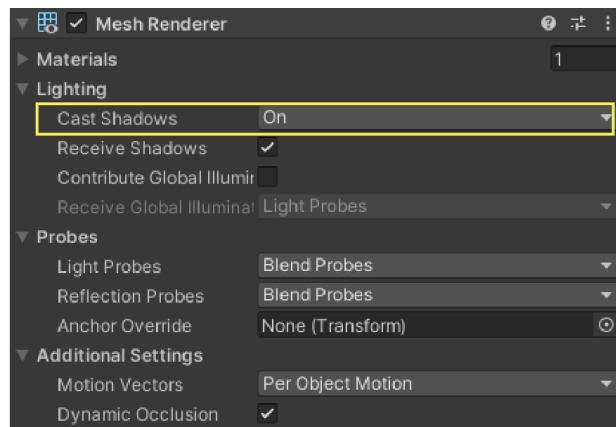
リアルタイムシャドウの生成はドローコールやフィルレートを大きく消費します。そのため、リアルタイムシャドウを使用する際は慎重に設定を検討する必要があります。

ドローコールの削減

シャドウ生成のドローコールを削減するためには、以下のような方針が考えられます。

- ・シャドウを落とすオブジェクトの数を減らす
- ・バッチングによりドローコールをまとめる

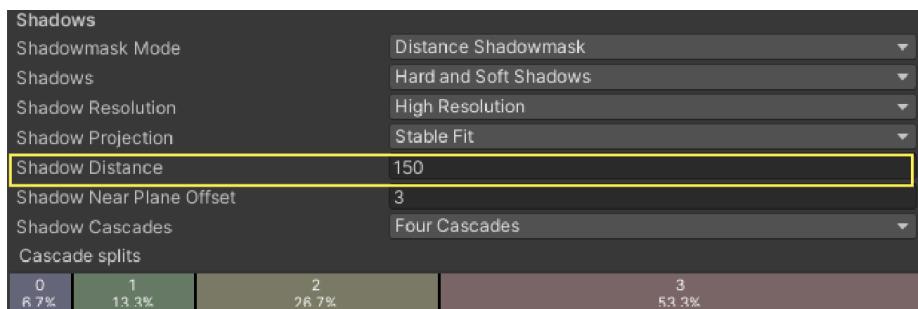
シャドウを落とすオブジェクトを減らす方法はいくつかありますが、シンプルな方法は MeshRenderer の **Cast Shadows** の設定をオフにすることです。これによりオブジェクトをシャドウの描画対象から外すことができます。この設定は通常 Unity ではオンになっているため、シャドウを使用しているプロジェクトでは注意するべきでしょう。



▲図 7.19 Cast Shadows

また、オブジェクトがシャドウマップに描画される最大距離を短くすることも有効です。Quality Settings の **Shadow Distance** の項目でシャドウマップの最大描画距離を変更することができ、この設定によりシャドウを落とすオブジェクトの数を必要最低限に減らすことができます。この設定を調整することでシャドウマップの解像度に対して最低限の範囲でシャドウを描画することができるため、シャドウの解像感の低下を抑えることにも繋がります。

第7章 Tuning Practice - Graphics



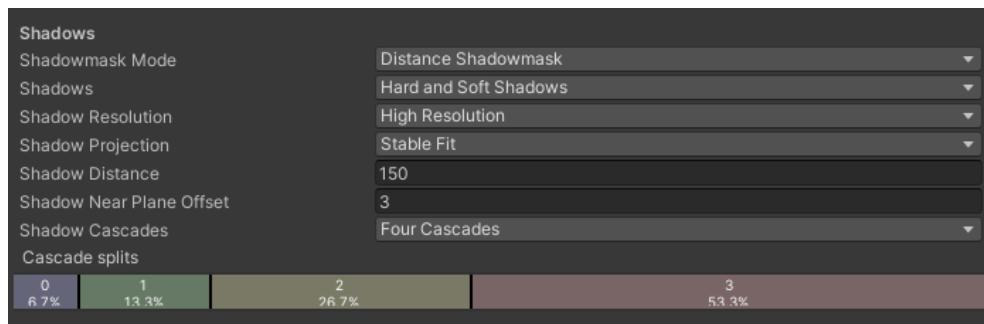
▲図 7.20 Shadow Distance

通常の描画と同様、シャドウ描画でもバッティング処理の対象にすることでドローコールを削減できます。バッティングの手法については「7.3 ドローコールの削減」を参照してください。

フィルレートの節約

シャドウによるフィルレートはシャドウマップの描画とシャドウの影響を受けるオブジェクトの描画の両方に左右されます。

Quality Settings の Shadows の項目にあるいくつかの設定を調整することでそれぞれのフィルレートを節約できます。



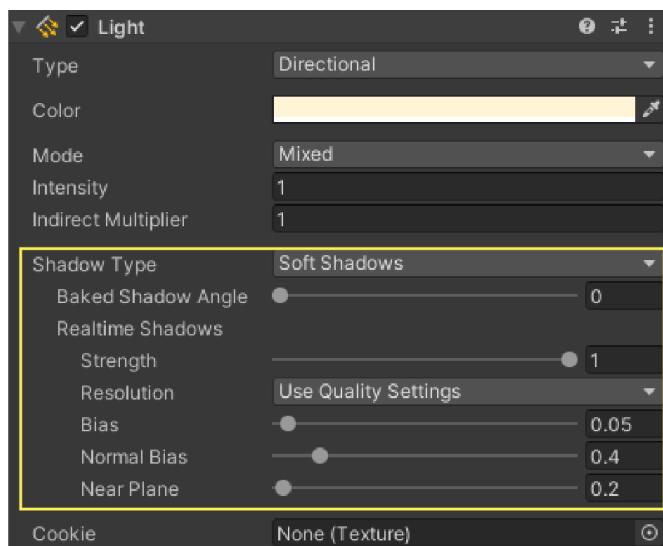
▲図 7.21 Quality Settings -> Shadows

Shadows の項目ではシャドウの形式を変更することができ、**Hard Shadows** は影の境界線がはっきりと出ますが比較的負荷が低く、**Soft Shadows** は影の境界線をぼかしたような表現ができますが負荷が高いです。

Shadow Resolution と **Shadow Cascades** の項目はシャドウマップの解像度に影

響する項目で、大きく設定するとシャドウマップの解像度が大きくなりフィルレートの消費が大きくなります。ただ、この設定はシャドウの品質に大きく関係するところでもあるため、パフォーマンスと品質のバランスを見ながら慎重に調整する必要があります。

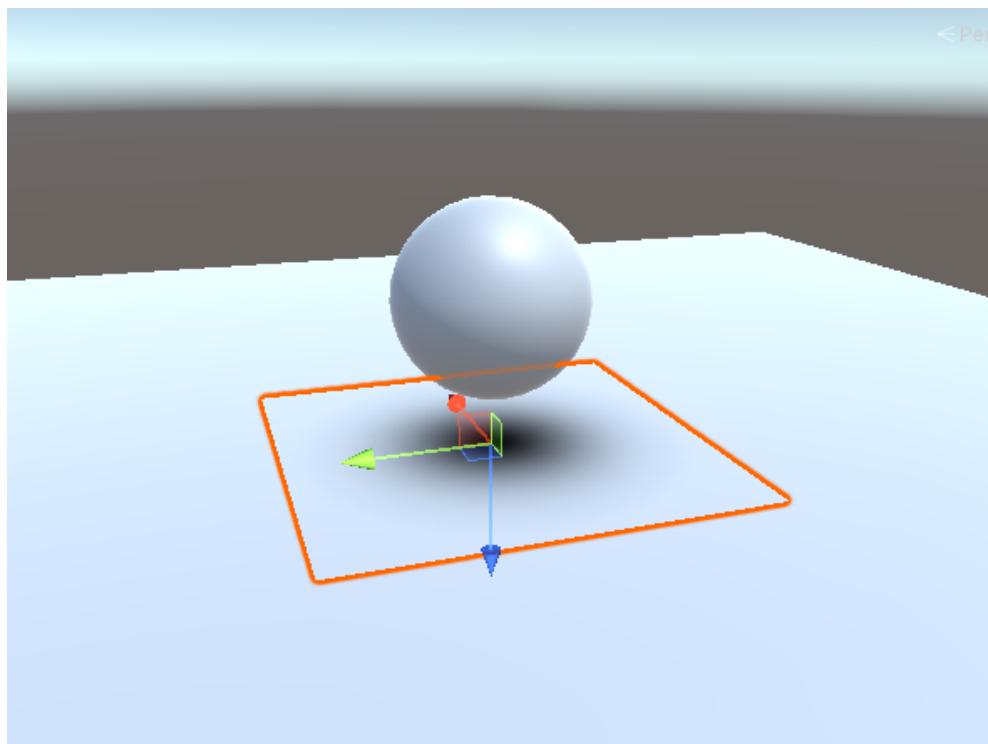
一部の設定は **Light** コンポーネントの Inspector から変更できるため、個別のライトごとに設定を変えることも可能です。



▲図 7.22 Light コンポーネントのシャドウ設定

疑似シャドウ

ゲームジャンルやアートスタイルによっては、板ポリゴンなどでオブジェクトの影を擬似的に表現する手法も有効です。この手法は使用上の制約が強く自由度が高いものではありませんが、通常のリアルタイムシャドウの描画手法に比べて圧倒的に軽量に描画できます。

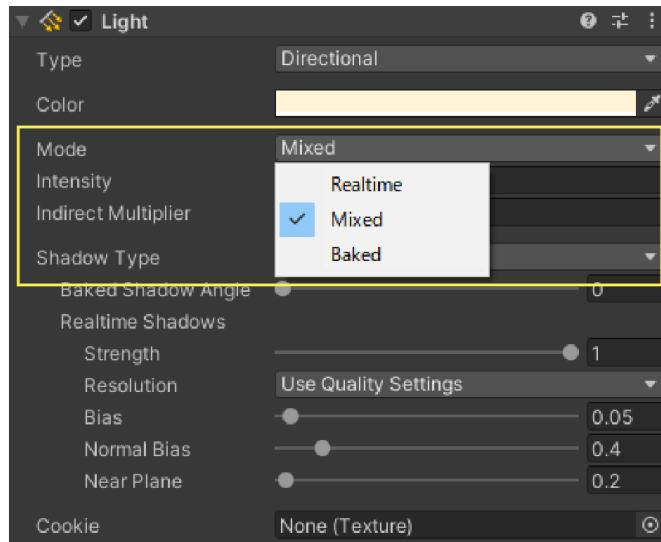


▲図 7.23 板ポリゴンによる擬似シャドウ

7.7.2 ライトマッピング

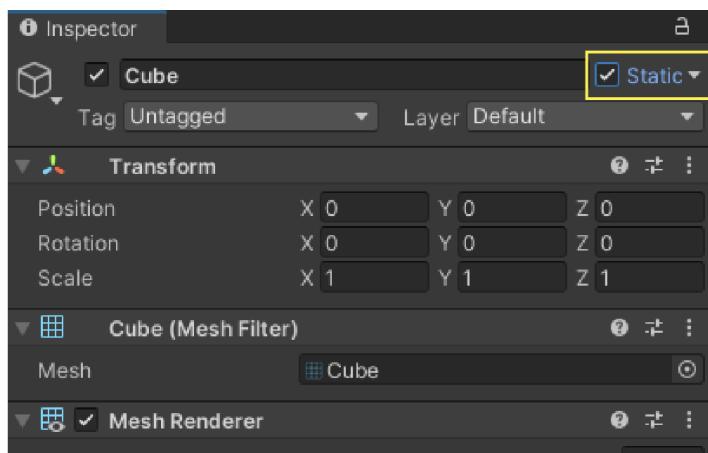
事前にライティングの効果とシャドウをテクスチャにベイクしておくことで、リアルタイム生成よりもかなり低負荷に品質の高いライティング表現を実現できます。

ライトマップをベイクするには、まずシーンに配置した Light コンポーネントの **Mode** の項目を **Mixed** か **Baked** に変更します。



▲図 7.24 Light の Mode 設定

また、バイク対象となるオブジェクトの static フラグを有効化します。

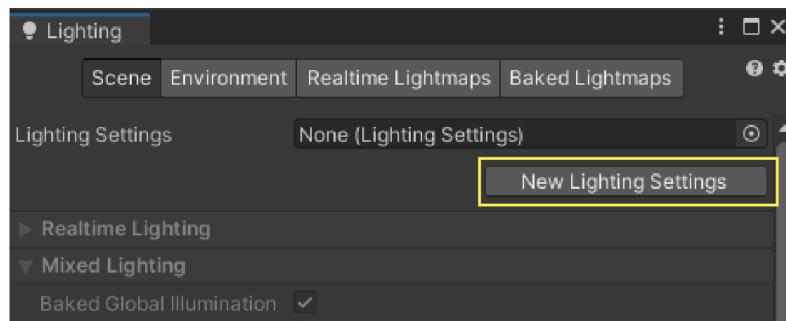


▲図 7.25 static の有効化

この状態で、メニューから「Window -> Rendering -> Lighting」を選択し Lighting ビューを表示します。

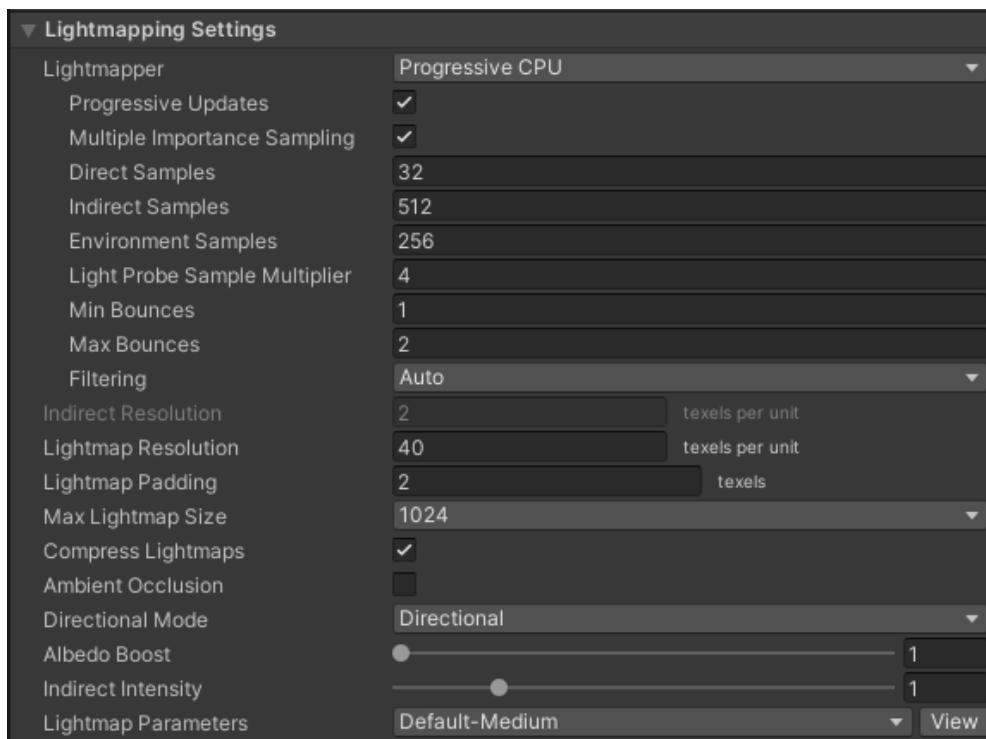
デフォルトの状態だと **Lighting Settings** アセットが指定されていないため、New

Lighting Settings ボタンを押下して新規作成を行います。



▲図 7.26 New Lighting Settings

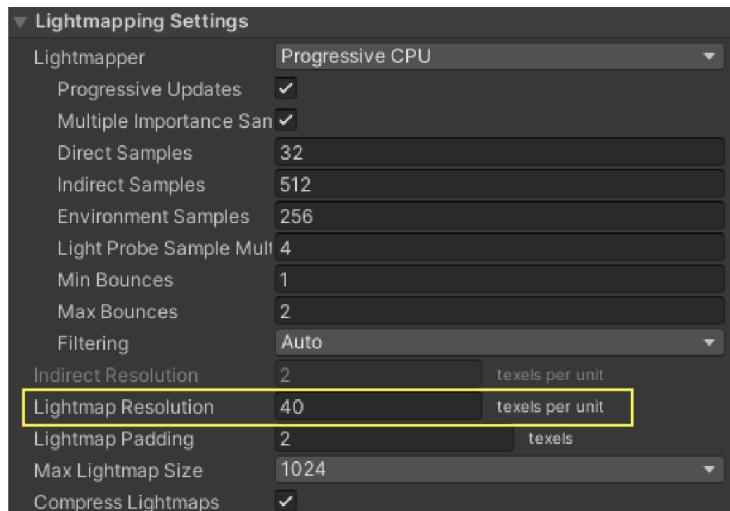
ライトマップについての設定は主に **Lightmapping Settings** タブで行います。



▲図 7.27 Lightmapping Settings

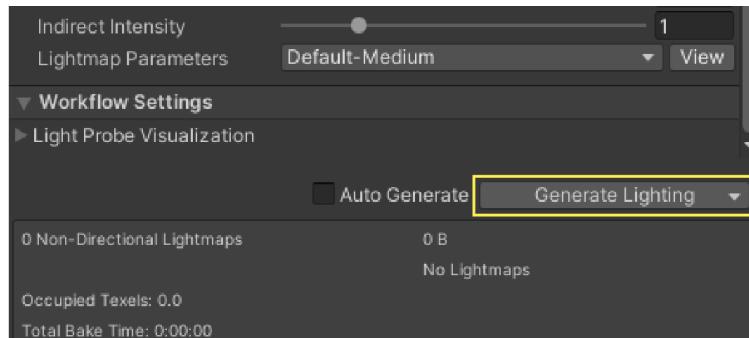
多くの設定項目がありますが、これらの値を調整することでライトマップのベイクの速度や品質が変わります。そのため、求める速度や品質に合わせて適切に設定する必要があります。

これらの設定の中でもっともパフォーマンスへの影響が大きいのは **Lightmap Resolution** です。この設定は Unity における 1unitあたりにライトマップのテクセルをどれだけ割り当てるかという設定で、この値により最終的なライトマップのサイズが変動するため、ストレージやメモリの容量、テクスチャへのアクセス速度などに大きな影響を与えます。

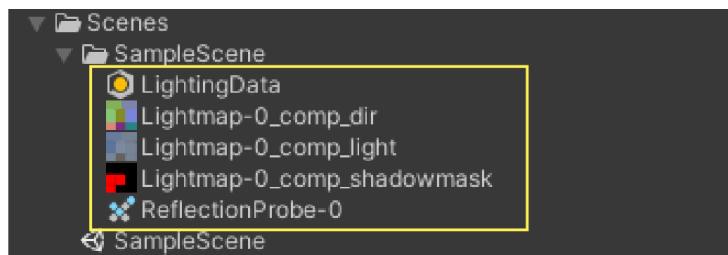


▲図 7.28 Lightmap Resolution

最後に、Inspector ビューの下部にある **Generate Lighting** ボタンを押下することでライトマップのベイクを行うことができます。ベイクが完了すると、シーンと同名のフォルダーにベイクされたライトマップが格納されていることを確認できます。



▲図 7.29 Generate Lighting



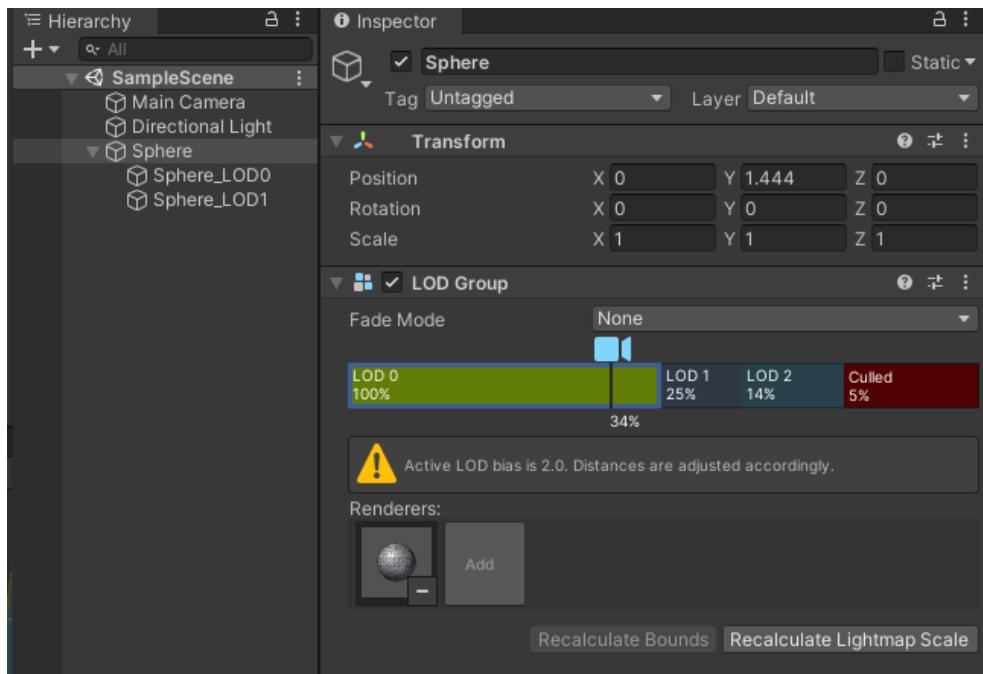
▲図 7.30 ベイクされたライトマップ

7.8 Level of Detail

カメラから遠い距離にあるオブジェクトをハイポリゴン・高精細に描画するのは非効率です。 **Level of Detail (LOD)** という手法を利用することでカメラからの距離に応じてオブジェクトの詳細度を削減できます。

Unity では、オブジェクトに **LOD Group** コンポーネントを追加することで LOD の制御を行うことができます。

7.9 テクスチャストリーミング



▲図 7.31 LOD Group

LOD Group がアタッチされた GameObject の子に各 LOD レベルのメッシュを持った Renderer を配置し、LOD Group の各 LOD レベルに設定することでカメラに応じて LOD レベルが切り替えられるようになります。カメラの距離に対してどの LOD レベルを割り当てるかを LOD Group ごとに設定することも可能です。

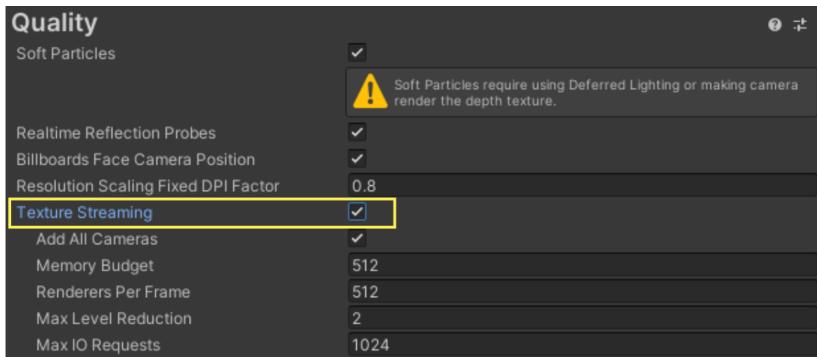
LOD を使用することで一般に描画の負荷を削減できますが、各 LOD レベルのメッシュがすべてロードされるためメモリやストレージの圧迫には注意が必要です。

7.9 テクスチャストリーミング

Unity のテクスチャストリーミングを利用することで、テクスチャのために必要なメモリ容量やロード時間を削減できます。テクスチャストリーミングとは、シーンのカメラ位置に応じてミップマップをロードすることで GPU メモリを節約するための機能です。

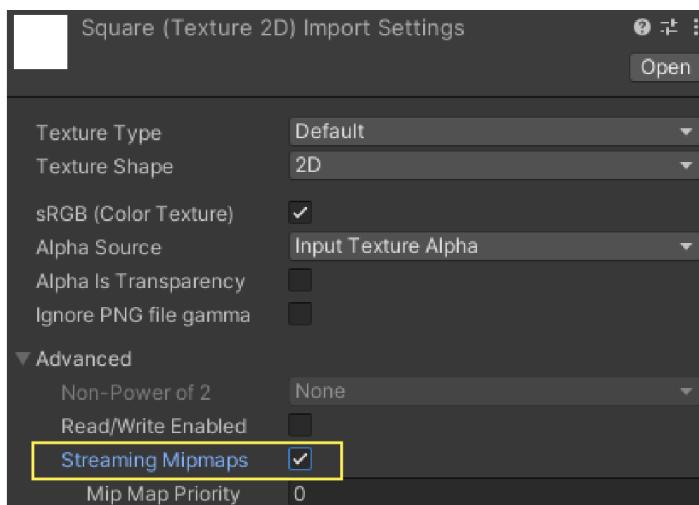
この機能を有効にするには、Quality Settings の **Texture Streaming** を有効化します。

第7章 Tuning Practice - Graphics



▲図 7.32 Texture Streaming

さらに、テクスチャのミップマップをストリーミングできるようにするためテクスチャのインポート設定を変更する必要があります。テクスチャの Inspector を開き、Advanced 設定内の **Streaming Mipmaps** を有効化します。



▲図 7.33 Streaming Mipmaps

これらの設定により指定されたテクスチャのミップマップがストリーミングされるようになります。また Quality Settings の **Memory Budget** の項目を調整することでロードするテクスチャの合計メモリ使用量を制限できます。テクスチャストリーミングシステムはここで設定したメモリ量を超えないようにミップマップをロードします。



PERFORMANCE TUNING BIBLE

CHAPTER

08

第8章

Tuning Practice

— UI —

CyberAgent Smartphone Games & Entertainment

第 8 章

Tuning Practice - UI

Unity 標準の UI システムである uGUI と、画面にテキストを描画する仕組みである TextMeshPro について、チューニングプラクティスを紹介します。

8.1 Canvas の分割

uGUI では Canvas 内の要素に変化があったとき、Canvas 全体の UI のメッシュを再構築する処理（リビルド）が走ります。変化とは、アクティブ切り替えや移動やサイズの変更など、見た目が大きく変わるものから一見ではわからないような細かいものまで、あらゆる変更を指します。リビルドの処理のコストは高いため、実行される回数が多かったり Canvas 内の UI の数が多かったりするとパフォーマンスに悪影響を及ぼします。

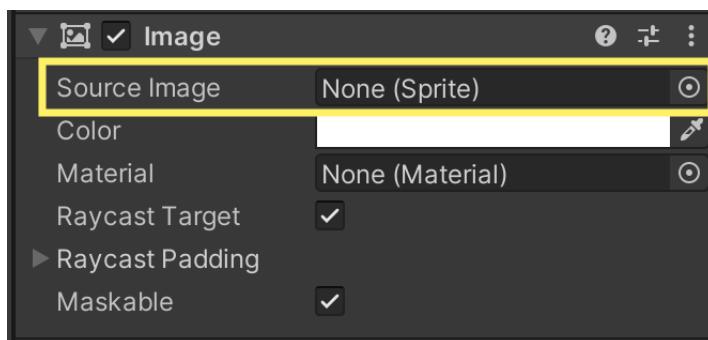
これに対して、ある程度の UI のまとまりごとに Canvas を分割することで、リビルドのコストを抑えることができます。たとえば、アニメーションで動く UI と何も動かない UI があったとき、それらを別の Canvas の下に配置することで、アニメーションによるリビルドの対象となるものを最小限にできます。

ただし、Canvas を分割すると描画のバッチが効かなくなるため、どのように分割すればよいかに関しては注意深く考える必要があります。

Canvas の分割は、Canvas の配下に Canvas を入れ子で配置する場合でも有効です。子の Canvas に含まれる要素が変化しても、子の Canvas のリビルドが走るだけで親の Canvas のリビルドは走りません。ただし詳しく確認したところ、SetActive によって子の Canvas 内の UI をアクティブ状態に切り替えたときは事情が違うようです。このとき、親の Canvas 内に UI が大量に配置されている場合は高負荷になる現象があるようです。なぜそのような挙動になるのかの詳細は分かりませんが、入れ子の Canvas 内の UI のアクティブ状態を切り替えるときは注意が必要そうです。

8.2 UnityWhite

UI の開発をしていると、単純な長方形型のオブジェクトを表示したいということがあります。そこで注意するべきなのが、UnityWhite の存在です。UnityWhite は、Image コンポーネントや RawImage コンポーネントで利用する画像を指定しなかったとき（図 8.1）に使われる Unity 組み込みのテクスチャです。UnityWhite が使われている様子は Frame Debugger で確認できます（図 8.2）。この仕組みを使うと白色の長方形を描画できるため、これに乗算する色を組み合わせることによって単純な長方形型の表示を実現できます。



▲図 8.1 UnityWhite の利用



▲図 8.2 UnityWhite が使われている様子

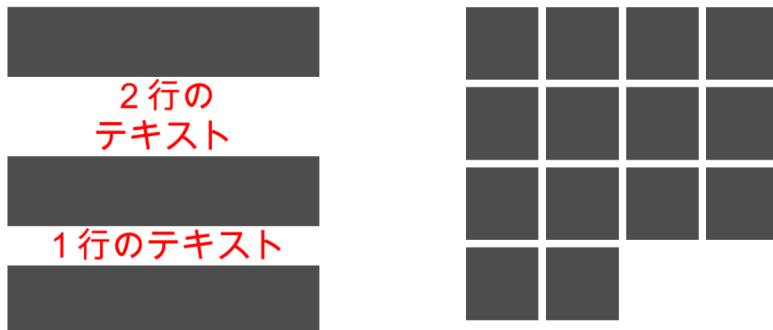
しかし、UnityWhite はプロジェクトで用意した SpriteAtlas とは別のテクスチャであるため、描画のバッチが途切れてしまうという問題が起こります。これによって、ドローコールが増加し描画効率が悪化してしまいます。

そのため、SpriteAtlas に小さい（たとえば 4×4 ピクセルの）白色正方形の画像を追加し、その Sprite を利用して単純な長方形を描画するようにするべきです。これに

よって、同じ SpriteAtlas を使っていれば同一マテリアルになるため、バッチを効かせることができます。

8.3 Layout コンポーネント

uGUI にはオブジェクトをきれいに整列させるための機能を持つ Layout コンポーネントが用意されています。たとえば縦方向に整列するなら `VerticalLayoutGroup`、グリッド上に整列するなら `GridLayoutGroup` が使われます（図 8.3）。



▲図 8.3 左側が `VerticalLayoutGroup`、右側が `GridLayoutGroup` を使った例

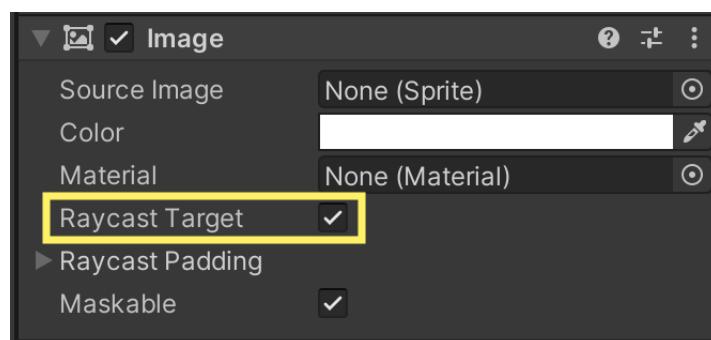
Layout コンポーネントを利用すると、対象のオブジェクトを生成したときや、特定のプロパティを編集したときに Layout のリビルドが発生します。Layout のリビルドも、メッシュのリビルドと同様にコストの高い処理です。

Layout のリビルドによるパフォーマンス低下を避けるには、Layout コンポーネントを極力使わないというのが有効です。

たとえば、テキストの内容に応じて配置が変わるとといった動的な配置が必要ないのであれば、Layout コンポーネントを使う必要がありません。本当に動的な配置が必要な場合も、画面上で多く使用される場合などは、独自のスクリプトで制御したほうがよい場合もあります。また、親のサイズが変わっても親から見て特定の位置に配置したいという要件であれば、`RectTransform` のアンカーを調整することで実現できます。プレハブを作成するときに配置に便利だからという理由で Layout コンポーネントを使った場合は、必ず削除して保存するようにしましょう。

8.4 Raycast Target

Image や RawImage のベースクラスである Graphic には、Raycast Target というプロパティがあります（図 8.4）。このプロパティを有効にすると、その Graphic がクリックやタッチの対象になります。画面をクリックしたりタッチしたりしたとき、このプロパティが有効なオブジェクトが処理の対象となるため、できる限りこのプロパティを無効にすることでパフォーマンスを向上できます。



▲図 8.4 Raycast Target プロパティ

このプロパティはデフォルトで有効ですが、実際のところ多くの Graphic ではこのプロパティを有効にする必要がありません。一方、Unity ではプリセット^{*1}と呼ばれる機能があり、デフォルトの値をプロジェクトで変更することが可能です。具体的には、Image コンポーネントと RawImage コンポーネントに対してそれぞれプリセットを作成し、それを Project Settings のプリセットマネージャーからデフォルトのプリセットとして登録します。この機能を使って Raycast Target プロパティをデフォルトで無効にしてもよいかもしれません。

8.5 マスク

uGUI でマスクを表現するには、Mask コンポーネントか RectMask2d コンポーネントを利用します。

^{*1} <https://docs.unity3d.com/ja/current/Manual/Presets.html>

Mask ではステンシルを利用してマスクを実現しているため、コンポーネントが増えたびに描画コストが大きくなります。それに対して RectMask2d はシェーダーのパラメーターでマスクを実現しているため描画コストの増加が抑えられています。ただし、Mask は好きな形でくり抜ける一方、RectMask2d は長方形でしかくり抜けないという制約があります。

利用できるなら RectMask2d を選択するべきだというのが通説ですが、最近の Unity では RectMask2d の利用にも注意が必要です。

具体的には、RectMask2d が有効のとき、そのマスク対象が増えるに連れ、それに比例して毎フレームカリングの CPU 負荷が発生します。UI を何も動かさなくとも毎フレーム負荷が発生するこの現象は、uGUI の内部実装のコメントを見る限り Unity 2019.3 で入ったとある issue² の修正の副作用によるものようです。

そのため、RectMask2d も極力使わないようにする、使ったとしても必要ない状態のときは enabled を false にする、マスク対象は必要最低限にするなどの対策を取ることが有効です。

8.6 TextMeshPro

TextMeshPro でテキストを設定する一般的な方法は text プロパティにテキストを代入する方法ですが、それとは別に SetText というメソッドを使う方法があります。

SetText には多くのオーバーロードが存在しますが、たとえば文字列と float 型の値を引数に取るものがあります。このメソッドをリスト 8.1 のように利用すると、第 2 引数の値を表示できます。ただし、label は TMP_Text (もしくはそれを継承した) 型、number は float 型の変数であるとします。

▼リスト 8.1 SetText の利用例

```
1: label.SetText("{0}", number);
```

この方法の利点は、文字列の生成コストを抑えられるという点です。

▼リスト 8.2 SetText を使わない例

```
1: label.text = number.ToString();
```

² <https://issuetracker.unity3d.com/issues/rectmask2d-differently-masks-image-in-the-play-mode-when-animating-rect-transform-pivot-property>

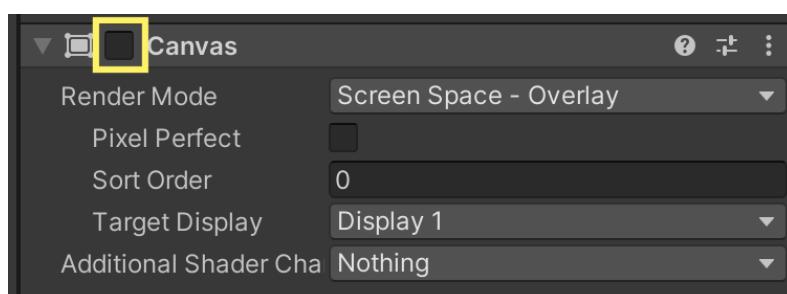
リスト 8.2 のように `text` プロパティを使う方法では、`float` 型の `ToString()` が実行されるのでこの処理が実行されるたびに文字列の生成コストが発生します。それに対して `SetText` を使った方法は、文字列を極力生成しないような工夫が行われているため、とくに頻繁に表示するテキストが変わるような場合、パフォーマンス的に有利です。

またこの `TextMeshPro` の機能は、`ZString`³と組み合わせると非常に強力なものになります。`ZString` は文字列生成におけるメモリアロケーションを削減できるライブラリです。`ZString` は `TMP_Text` 型に対する多くの拡張メソッドを提供しており、それらのメソッドを使うことで文字列の生成コストを抑えつつ柔軟なテキスト表示を実現できます。

8.7 UI の表示切り替え

uGUI のコンポーネントは、`SetActive` によるオブジェクトのアクティブ切り替えのコストが大きいという特徴があります。これは、`OnEnable` で各種リビルトの Dirty フラグを立てたり、マスクに関する初期化を行ったりしていることが原因です。そのため、UI の表示非表示の切り替えの方法として、`SetActive` による方法以外の選択肢も検討することが重要です。

まず 1 つ目の方法は、`Canvas` の `enabled` を `false` にするという方法です（図 8.5）。これによって、`Canvas` 配下のオブジェクトがすべて描画されなくなります。そのためこの方法は、`Canvas` 配下のオブジェクトを丸ごと非表示にしたい場合のみにしか使えないという欠点があります。



▲図 8.5 Canvas を無効にする

³ <https://github.com/Cysharp/ZString>

もう1つの方法は、`CanvasGroup`を使った方法です。`CanvasGroup`には、その配下のオブジェクトの透明度を一括で調整できる機能があります。この機能を利用して、透明度を0にしてしまえば、その`CanvasGroup`配下のオブジェクトをすべて非表示にできます（図8.6）。



▲図8.6 `CanvasGroup` の透明度を0にする

これらの方は`SetActive`による負荷を避けることが期待できますが、`GameObject`はアクティブ状態のままとなるため注意が必要な場合もあります。たとえば`Update`メソッドが定義されている場合には、その処理は非表示の状態でも実行され続けるため、思わぬ負荷の向上に繋がってしまうかもしれないことに気をつけましょう。

参考までに、`Image`コンポーネントを付けた1280個の`GameObject`に対して、それぞれの手法で表示非表示の切り替えをしたときの処理時間を計測しました（表8.1）。処理時間はUnityエディターで計測し、Deep Profileは用いていません。実際に切り替えを行ったまさにその処理の実行時間⁴と、そのフレームでの`UIEvents.WillRenderCanvases`の実行時間を足し合わせたものをその手法の処理時間としています。`UIEvents.WillRenderCanvases`の実行時間を足し合はせているのは、この中でUIのリビルドが実行されるためです。

▼表8.1 表示切り替えの処理時間

手法	処理時間（表示）	処理時間（非表示）
<code>SetActive</code>	323.79ms	209.93ms
<code>Canvas</code> の <code>enabled</code>	61.25ms	61.23ms
<code>CanvasGroup</code> の <code>alpha</code>	3.64ms	3.40ms

表8.1の結果から、今回試したシチュエーションでは`CanvasGroup`を使った手法が圧倒的に処理時間が短いことがわかりました。

⁴ たとえば`SetActive`なら、`SetActive`メソッドを呼び出す部分を`Profiler.BeginSample`と`Profiler.EndSample`で囲って計測しています。



PERFORMANCE TUNING BIBLE

CHAPTER

09

第9章

Tuning Practice
— Script (Unity) —

CyberAgent Smartphone Games & Entertainment

第 9 章

Tuning Practice - Script (Unity)

Unity で提供されている機能を何気なく使っていると思わぬ落とし穴にはまることがあります。本章では、Unity の内部実装に関連したパフォーマンスチューニング手法について実例を交えながら紹介します。

9.1 空の Unity イベント関数

`Awake`、`Start`、`Update` など Unity が提供しているイベント関数が定義されている場合、実行時に Unity 内部のリストにキャッシュされて、リストのイテレーションによって実行されます。

関数内で何も処理を行っていないとも、定義されているだけでキャッシュ対象となるため、不要なイベント関数を残したままにするリストが肥大化し、イテレーションのコストが増大します。

たとえば下記サンプルコードのように Unity 上で新規生成したスクリプトには `Start`、`Update` が最初から定義されていますが、これらの関数が不要であれば必ず削除しておきましょう。

▼リスト 9.1 Unity 上で新規生成したスクリプト

```
1: public class NewBehaviourScript : MonoBehaviour
2: {
3:     // Start is called before the first frame update
4:     void Start()
5:     {
6:
7:     }
8:
9:     // Update is called once per frame
10:    void Update()
11:    {
12:
13:    }
14: }
```

9.2 tag や name のアクセス

UnityEngine.Object を継承したクラスには tag プロパティと name プロパティが提供されています。オブジェクトの識別に便利なこれらプロパティですが、実は GC.Alloc が発生しています。

それぞれの実装を UnityCsReference から引用しました。どちらもネイティブコードで実装された処理を呼び出していることが分かります。

Unity ではスクリプトを C# で実装しますが、Unity 自体は C++ で実装されています。C# メモリ空間と C++ メモリ空間は共有できないため、C++ 側から C# 側に文字列情報を受け渡すためにメモリの確保が行われます。これは呼び出すたびに行われる所以、複数回アクセスする場合はキャッシュしておきましょう。

Unity の仕組みと C# と C++ 間のメモリに関する詳細は「Unity ランタイム」を参照してください。

▼リスト 9.2 UnityCsReference GameObject.bindings.cs¹ から引用

```

1: public extern string tag
2: {
3:     [FreeFunction("GameObjectBindings::GetTag", HasExplicitThis = true)]
4:     get;
5:     [FreeFunction("GameObjectBindings::SetTag", HasExplicitThis = true)]
6:     set;
7: }
```

▼リスト 9.3 UnityCsReference UnityEngineObject.bindings.cs² から引用

```

1: public string name
2: {
3:     get { return GetName(this); }
4:     set { SetName(this, value); }
5: }
6:
7: [FreeFunction("UnityEngineObjectBindings::GetName")]
8: extern static string GetName([NotNull("NullExceptionObject")]) Object obj;
```

¹ <https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Runtime/Export/Scripting/GameObject.bindings.cs>

² <https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Runtime/Export/Scripting/UnityEngineObject.bindings.cs>

9.3 コンポーネントの取得

同じ `GameObject` にアタッチされている他のコンポーネントを取得する `GetComponent()` も注意が必要な 1 つです。

前節の `tag` プロパティや `name` プロパティ同様にネイティブコードで実装された処理を呼び出していることもそうですが、指定した型のコンポーネントを「検索する」コストがかかることにも気をつけなければなりません。

下記サンプルコードでは毎フレーム `Rigidbody` コンポーネントを検索するコストがかかることになります。頻繁にアクセスする場合は、あらかじめキャッシュしたものを使い回すようにしましょう。

▼リスト 9.4 毎フレーム GetComponent() するコード

```
1: void Update()
2: {
3:     Rigidbody rb = GetComponent<Rigidbody>();
4:     rb.AddForce(Vector3.up * 10f);
5: }
```

9.4 transform へのアクセス

`Transform` コンポーネントは位置や回転、スケール（拡大・縮小）、親子関係の変更など頻繁にアクセスするコンポーネントです。下記サンプルコードのように複数の値を更新することも多いでしょう。

▼リスト 9.5 transform にアクセスする例

```
1: void SetTransform(Vector3 position, Quaternion rotation, Vector3 scale)
2: {
3:     transform.position = position;
4:     transform.rotation = rotation;
5:     transform.localScale = scale;
6: }
```

`transform` を取得すると Unity 内部では `GetTransform()` という処理が呼び出されます。前節の `GetComponent()` に比べて最適化されていて高速です。しかしキャッシュした場合よりは遅いので、これも下記サンプルコードのようにキャッシュしてアクセスしましょう。位置と回転の 2 つは `SetPositionAndRotation()` を使うことで関数呼び出し回数を減らすこともできます。

▼リスト 9.6 transform をキャッシュする例

```
1: void SetTransform(Vector3 position, Quaternion rotation, Vector3 scale)
2: {
3:     var transformCache = transform;
4:     transformCache.SetPositionAndRotation(position, rotation);
5:     transformCache.localScale = scale;
6: }
```

9.5 明示的な破棄が必要なクラス

Unity は C#で開発を行うため、GC によって参照されなくなったオブジェクトは解放されます。しかし Unity のいくつかのクラスは明示的に破棄する必要があります。代表的な例としては Texture2D、Sprite、Material、PlayableGraph などです。new や専用の Create 関数で生成した場合、必ず明示的に破棄を行いましょう。

▼リスト 9.7 生成と明示的な破棄

```
1: void Start()
2: {
3:     _texture = new Texture2D(8, 8);
4:     _sprite = Sprite.Create(_texture, new Rect(0, 0, 8, 8), Vector2.zero);
5:     _material = new Material(shader);
6:     _graph = PlayableGraph.Create();
7: }
8:
9: void OnDestroy()
10: {
11:     Destroy(_texture);
12:     Destroy(_sprite);
13:     Destroy(_material);
14:
15:     if (_graph.IsValid())
16:     {
17:         _graph.Destroy();
18:     }
19: }
```

9.6 文字列指定

Animator の再生するステートの指定、Material の操作するプロパティの指定に文字列を使うのは避けましょう。

▼リスト 9.8 文字列指定の例

```
1: _animator.Play("Wait");
2: _material.SetFloat("_Prop", 100f);
```

これらの関数の内部では `Animator.StringToHash()` や `Shader.PropertyToID()` を実行して、文字列から一意な識別値に変換をしています。何回もアクセスする場合に都度変換が行われるのはムダなので、識別値をキャッシュしておいて使い回すようにしましょう。下記サンプルのようにキャッシュした識別値の一覧となるクラスを定義しておくと、取り回しがよいでしょう。

▼リスト 9.9 識別値のキャッシュの例

```
1: public static class ShaderProperty
2: {
3:     public static readonly int Color = Shader.PropertyToID("_Color");
4:     public static readonly int Alpha = Shader.PropertyToID("_Alpha");
5:     public static readonly int ZWrite = Shader.PropertyToID("_ZWrite");
6: }
7: public static class AnimationState
8: {
9:     public static readonly int Idle = Animator.StringToHash("idle");
10:    public static readonly int Walk = Animator.StringToHash("walk");
11:    public static readonly int Run = Animator.StringToHash("run");
12: }
```

9.7 JsonUtility の落とし穴

Unity では JSON のシリアル化/デシリアル化のために `JsonUtility` というクラスが提供されています。公式ドキュメント^{*3}にも C# 標準のものよりも高速であることが記載されていて、パフォーマンスを意識した実装をするなら利用することも多いでしょう。

`JsonUtility` は(機能は .NET JSON より少ないですが)、よく使用されている .NET JSON よりも著しく早いことが、ベンチマークテストで示されています。

しかしパフォーマンスに関わることでひとつ気をつけるべきことがあります。それは「`null` の扱い」です。

下記サンプルコードでシリアル化処理とその結果を示しています。クラス A のメンバー `b1` を明示的に `null` にしているにもかかわらず、クラス B およびクラス C を

^{*3} <https://docs.unity3d.com/ja/current/Manual/JSONSerialization.html>

デフォルトコンストラクターで生成した状態でシリアル化されているのが分かります。このようにシリアル化対象となるフィールドに null があった場合、JSON 化の際にダミーオブジェクトが newされるので、そのオーバーヘッドは考慮しておいたほうがよいでしょう。

▼リスト 9.10 シリアル化の挙動

```
1: [Serializable] public class A { public B b1; }
2: [Serializable] public class B { public C c1; public C c2; }
3: [Serializable] public class C { public int n; }
4:
5: void Start()
6: {
7:     Debug.Log(JsonUtility.ToJson(new A() { b1 = null, }));
8:     // {"b1": {"c1": {"n": 0}, "c2": {"n": 0}}}
9: }
```

9.8 Render や MeshFilter の落とし穴

Renderer.material で取得したマテリアル、MeshFilter.mesh で取得したメッシュは複製されたインスタンスなので使い終わったら明示的な破棄が必要です。公式ドキュメント^{*4*5}にもそれぞれ下記のように明記されています。

If the material is used by any other renderers, this will clone the shared material and start using it from now on.

It is your responsibility to destroy the automatically instantiated mesh when the game object is being destroyed.

取得したマテリアルやメッシュはメンバー変数に保持しておき、然るべきタイミングで破棄するようにしましょう。

▼リスト 9.11 複製されたマテリアルの明示的な破棄

```
1: void Start()
2: {
3:     _material = GetComponent<Renderer>().material;
4: }
5:
6: void OnDestroy()
7: {
```

^{*4} <https://docs.unity3d.com/ja/current/ScriptReference/Renderer-material.html>

^{*5} <https://docs.unity3d.com/ja/current/ScriptReference/MeshFilter-mesh.html>

```
8:     if (_material != null) {
9:         Destroy(_material);
10:    }
11: }
```

9.9 ログ出力コードの除去

Unity では `Debug.Log()`、`Debug.LogWarning()`、`Debug.LogError()` といったログ出力用の関数が提供されています。便利な機能ではありますがいくつかの問題点もあります。

- ログ出力自体がそこそこ重たい処理
- リリースビルドでも実行される
- 文字列の生成や連結によって `GC.Alloc` が発生する

Unity の Logging 設定をオフにした場合、スタックトレースは停止しますが、ログは出力されます。`UnityEngine.Debug.unityLogger.logEnabled` に Unity では `false` を設定すると、ログは出力されませんが、関数内部で分岐しているだけなので、関数の呼び出しコストや不要なはずの文字列の生成や連結は行われてしまいます。`#if` ディレクティブを使うという手段もありますが、すべてのログ出力処理に手を入れるのは現実的ではありません。

▼リスト 9.12 #if ディレクティブ

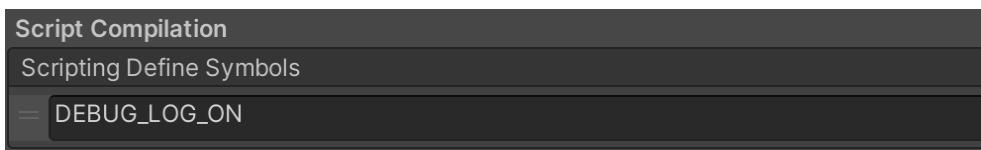
```
1: #if UNITY_EDITOR
2:     Debug.LogError($"Error {e}");
3: #endif
```

このような場合に活用できるのが `Conditional` 属性です。`Conditional` 属性が付いた関数は、指定したシンボルが定義されていない場合、コンパイラによって呼び出し部分が除去されます。リスト 9.13 のサンプルのように、自作のログ出力クラスを通して Unity 側のログ機能を呼び出すのをルールとして、自作クラス側の各関数に `Conditional` 属性を付加することで、必要に応じて関数の呼び出しごと除去できるようになるとよいでしょう。

▼リスト 9.13 Conditional 属性の例

```
1: public static class Debug
2: {
3:     private const string MConditionalDefine = "DEBUG_LOG_ON";
4:
5:     [System.Diagnostics.Conditional(MConditionalDefine)]
6:     public static void Log(object message)
7:         => UnityEngine.Debug.Log(message);
8: }
```

注意点として、指定したシンボルが関数の呼び出し側から参照できる必要があるということです。#define で定義されたシンボルのスコープは、記述したファイル内に限定されてしまいます。Conditional 属性が付いた関数を呼び出しているすべてのファイルにシンボルを定義していくのは現実的ではありません。Unity には Scripting Define Symbols というプロジェクト全体に対してシンボルを定義する機能があるので活用しましょう。「Project Settings -> Player -> Other Settings」で設定ができます。



▲図 9.1 Scripting Define Symbols

9.10 Burst を用いたコードの高速化

Burst⁶は Unity 公式が開発する、ハイパフォーマンスな C#スクリプティングを行うためのコンパイラです。

Burst では C# のサブセット言語を用いてコードを記述します。Burst が C# コードを LLVM というコンパイラ基盤⁷の中間構文である IR (Intermediate Representation) に変換し、IR を最適化をした上で機械語に変換されます。

このときにコードを可能な限りベクトル化し、SIMD という命令を積極的に使った処理に置き換えます。これによって、より高速なプログラムの出力が期待できます。

SIMD は Single Instruction/Multiple Data の略で、单一の命令を同時に複数のデータ

⁶ <https://docs.unity3d.com/Packages/com.unity.burst@1.6/manual/docs/QuickStart.html>

⁷ <https://llvm.org/>

に適用するような命令を指します。つまり SIMD 命令を積極的に利用することで、1 命令でデータがまとめて処理されるため、通常の命令と比べて高速に動作します。

9.10.1 Burst を用いたコードの高速化

Burst では High Performance C#(HPC#)⁸と呼ばれる C# のサブセット言語を用いてコードを記述します。

HPC# の特徴の 1 つとして C# の参照型、つまりクラスや配列などが利用できません。そのため原則として構造体を用いてデータ構造を記述します。

配列のようなコレクションは代わりに NativeArray<T>などの NativeContainer⁹を利用します。HPC# の詳細については脚注記載のドキュメントを参考にしてください。

Burst は C# Job System と組み合わせて利用します。そのため自身の処理を IJob を実装したジョブの Execute メソッド内に記述します。定義したジョブに BurstCompile 属性を付与することで、そのジョブが Burst によって最適化されます。

リスト 9.14 に、与えられた配列の各要素を二乗して Output 配列に格納する例を示します。

▼リスト 9.14 簡単な検証用の Job 実装

```
1: [BurstCompile]
2: private struct MyJob : IJob
3: {
4:     [ReadOnly]
5:     public NativeArray<float> Input;
6:
7:     [WriteOnly]
8:     public NativeArray<float> Output;
9:
10:    public void Execute()
11:    {
12:        for (int i = 0; i < Input.Length; i++)
13:        {
14:            Output[i] = Input[i] * Input[i];
15:        }
16:    }
17: }
```

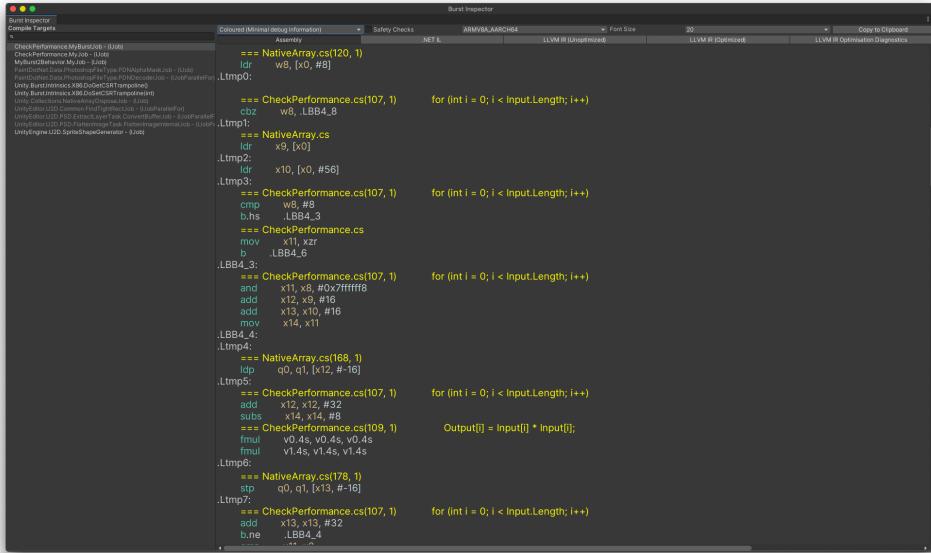
リスト 9.14 の 14 行目の各要素はそれぞれ独立して計算でき（計算に順序依存がない）、かつ出力配列のメモリアライメントは連続しているため SIMD 命令を用いてまとめて計算が可能です。

⁸ https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/CSharpLanguageSupport_Types.html

⁹ <https://docs.unity3d.com/Manual/JobSystemNativeContainer.html>

9.10 Burst を用いたコードの高速化

コードがどのようなアセンブリに変換されるかは、図 9.2 のように Burst Inspector を用いて確認できます。



▲図 9.2 Burst Inspector を用いることで、コードがどのようなアセンブリに変換されるか確認できる

リスト 9.14 の 14 行目の処理は、ARMV8A_AARCH64 向けのアセンブリでリスト 9.15 に変換されます。

▼リスト 9.15 リスト 9.14 の 14 行目の ARMV8A_AARCH64 向けのアセンブリ

```
1:      fmul      v0.4s, v0.4s, v0.4s
2:      fmul      v1.4s, v1.4s, v1.4s
```

アセンブリのオペランドに、`.4s` というサフィックスがついていることから、SIMD 命令が利用されていることが確認できます。

ピュアな C#により実装されたコードと Burst により最適化されたコードのパフォーマンスを実機で比較します。

実機には Android Pixel 4a、IL2CPP をスクリプトバックエンドとしてビルドを行い比較しています。また配列のサイズは $2^{20} = 1,048,576$ としています。計測は同じ処理を 10 回繰り返し、処理時間の平均をとりました。

第9章 Tuning Practice - Script (Unity)

表9.1にパフォーマンス比較の計測結果を示します。

▼表9.1 ピュアなC#実装とBurstによる最適化されたコードの処理時間の比較

手法	処理時間（非表示）
ピュアなC#実装	5.73ms
Burstによる実装	0.98ms

ピュアなC#実装と比べて約5.8倍ほど高速化を確認できました。



PERFORMANCE TUNING BIBLE

CHAPTER

10

第10章

Tuning Practice
— Script (C#) —

CyberAgent Smartphone Games & Entertainment

第 10 章

Tuning Practice - Script (C#)

本章では、主に C# コードのパフォーマンスチューニング手法について実例を交えながら紹介します。C# の基礎的な記法についてはここでは扱わず、パフォーマンスを要求されるようなゲームの開発において意識すべき設計や実装について解説します。

10.1 GC.Alloc するケースと対処法

「2.5.2 ガベージコレクション」で紹介しましたが、本節では具体的にどのような処理を行ったときに GC.Alloc をするのか、まずは理解していきましょう。

10.1.1 参照型の new

まずはとてもわかりやすく GC.Alloc が発生するケースです。

▼リスト 10.1 每フレーム GC.Alloc するコード

```
1: private void Update()
2: {
3:     const int listCapacity = 100;
4:     // List<int>のnewでGC.Alloc
5:     var list = new List<int>(listCapacity);
6:     for (var index = 0; index < listCapacity; index++)
7:     {
8:         // 特に意味はないけどindexをListに詰めていく
9:         list.Add(index);
10:    }
11:    // listから値をランダムに取り出す
12:    var random = UnityEngine.Random.Range(0, listCapacity);
13:    var randomValue = list[random];
14:    // ... ランダムな値から何かする ...
15: }
```

このコードの大きな問題点は、毎フレーム実行される Update メソッドで、List<int>を new している点です。

こちらを修正するには、List<int>を事前に生成して使い回すことで毎フレームの GC.Alloc を回避することができます。

▼リスト 10.2 毎フレームの GC.Alloc を無くしたコード

```
1: private static readonly int listCapacity = 100;
2: // 事前にListを生成しておく
3: private readonly List<int> _list = new List<int>(listCapacity);
4:
5: private void Update()
6: {
7:     _list.Clear();
8:     for (var index = 0; index < listCapacity; index++)
9:     {
10:         // 特に意味はないけどindexをListに詰めていく
11:         _list.Add(index);
12:     }
13:     // listから値をランダムに取り出す
14:     var random = UnityEngine.Random.Range(0, listCapacity);
15:     var randomValue = _list[random];
16:     // ... ランダムな値から何かする ...
17: }
```

こちらのサンプルコードのような無意味なコードを書くことはないと思いますが、類似した例は想像よりも見つかるケースが多いです。

GC.Alloc を無くしたら

気付いた方も多いかと思いますが、上記リスト 10.2 のサンプルコードはこれだけで事足ります。

```
1: var randomValue = UnityEngine.Random.Range(0, listCapacity);
2: // ... ランダムな値から何かする ...
```

パフォーマンスチューニングにおいて GC.Alloc を無くすことを考えるのは重要ですが、無意味な計算を省くことを常に考えることが高速化の一歩に繋がります。

10.1.2 ラムダ式

ラムダ式も便利な機能ですが、こちらも使い方によっては GC.Alloc が発生してしまうため、ゲームでは使用できる場面は限られます。ここでは、次のようなコードが定義されている前提とします。

第 10 章 Tuning Practice - Script (C#)

▼リスト 10.4 ラムダ式サンプルの前提コード

```
1: // メンバー変数
2: private int _memberCount = 0;
3:
4: // static変数
5: private static int _staticCount = 0;
6:
7: // メンバーメソッド
8: private void IncrementMemberCount()
9: {
10:     _memberCount++;
11: }
12:
13: // staticメソッド
14: private static void IncrementStaticCount()
15: {
16:     _staticCount++;
17: }
18:
19: // 受け取ったActionをInvokeするだけのメンバーメソッド
20: private void InvokeActionMethod(System.Action action)
21: {
22:     action.Invoke();
23: }
```

このとき、次のようにラムダ式内で変数を参照した場合、GC.Alloc が発生します。

▼リスト 10.5 ラムダ式内で変数を参照して GC.Alloc するケース

```
1: // メンバー変数を参照した場合、Delegate Allocationが発生
2: InvokeActionMethod(() => { _memberCount++; });
3:
4: // ローカル変数を参照した場合、Closure Allocationが発生
5: int count = 0;
6: // 上記と同じDelegate Allocationも発生
7: InvokeActionMethod(() => { count++; });
```

ただし、以下のように static 変数を参照すると、これらの GC.Alloc を回避できます。

▼リスト 10.6 ラムダ式内で static 変数を参照して GC.Alloc しないケース

```
1: // static変数を参照した場合、GC Allocは発生せず
2: InvokeActionMethod(() => { _staticCount++; });
```

10.1 GC.Alloc するケースと対処法

ラムダ式内のメソッド参照も記述方法によって GC.Alloc のされ方が異なります。

▼リスト 10.7 ラムダ式内でメソッドを参照して GC.Alloc するケース

```
1: // メンバーメソッドを参照した場合、Delegate Allocationが発生
2: InvokeActionMethod(() => { IncrementMemberCount(); });
3:
4: // メンバーメソッドを直接指定した場合、Delegate Allocationが発生
5: InvokeActionMethod(IncrementMemberCount);
6:
7: // staticメソッドを直接指定した場合、Delegate Allocationが発生
8: InvokeActionMethod(IncrementStaticCount);
```

これらを回避するためには、以下のようにステートメント形式で static メソッドを参照する必要があります。

▼リスト 10.8 ラムダ式内でメソッドを参照して GC.Alloc しないケース

```
1: // ラムダ式内でstaticメソッドを参照した場合、GC.Allocは発生せず
2: InvokeActionMethod(() => { IncrementStaticCount(); });
```

こうすることで初回のみ Action が new されますが、内部的にキャッシュされることによって 2 回目以降 GC.Alloc が回避されます。

しかし、すべての変数やメソッドを static にすることはコードの安全性や可読性の面からとても採用できるものではありません。高速化が必要なコードでは、static を多用して GC.Alloc を無くすよりも毎フレームもしくは不定なタイミングで発火するイベントなどはラムダ式を使わずに設計する方が安全と言えるでしょう。

10.1.3 ジェネリックを使用してボックス化するケース

ジェネリックを使用した以下の場合、何が原因でボックス化する可能性があるでしょうか。

▼リスト 10.9 ジェネリックを使用してボックス化する可能性がある例

```
1: public readonly struct GenericStruct<T> : IEquatable<T>
2: {
3:     private readonly T _value;
4:
5:     public GenericStruct(T value)
6:     {
7:         _value = value;
8:     }
9:
10:    public bool Equals(T other)
```

```
11:     {
12:         var result = _value.Equals(other);
13:         return result;
14:     }
15: }
```

このケースでは、プログラマーは GenericStruct に IEquatable<T>インターフェイスの実装をしましたが、T に制限を設けるのを忘れてしました。その結果、IEquatable<T>インターフェイスの実装がされていない型を T に指定できてしまい、Object 型へ暗黙的にキャストされて以下の Equals が使われるケースが存在してしまいます。

▼リスト 10.10 Object.cs

```
1: public virtual bool Equals(object obj);
```

たとえば IEquatable<T>インターフェイスの実装がされていない struct を T に指定すると、Equals の引数で object にキャストされることになるため、ボックス化が発生します。これが起こらないように事前に対策するには、次のように変更します。

▼リスト 10.11 ボックス化しないように制限をかけた例

```
1: public readonly struct GenericOnlyStruct<T> : IEquatable<T>
2:     where T : IEquatable<T>
3:     {
4:         private readonly T _value;
5:
6:         public GenericOnlyStruct(T value)
7:         {
8:             _value = value;
9:         }
10:
11:        public bool Equals(T other)
12:        {
13:            var result = _value.Equals(other);
14:            return result;
15:        }
16:    }
```

where 句（ジェネリック型制約）を用いて、T が受け入れられる型を IEquatable<T>を実装している型に制限してあげることで、こうした予期せぬボックス化を未然に防ぐことができます。

本来の目的を見失わない

「2.5.2 ガベージコレクション」で紹介したようにゲームではランタイム中の GC.Alloc を避けたい意図があるため、構造体が選択されるケースも多く存在します。ただし、GC.Alloc を削減したいあまりにすべてを構造体にしたところで高速化できるわけではありません。

よくある失敗としては、GC.Alloc を避ける目的で構造体を取り入れたところ、期待通り GC に関するコストが減ったものの、データサイズが大きいために値型のコピーコストがかかってしまい結果的に非効率な処理になってしまうようなケースが挙げられます。

また、これをさらに回避するためにメソッドの引数を参照渡しを利用することでコピーコストを削減する手法も存在します。結果的に高速化できる可能性はありますが、この場合は最初からクラスを選択し、インスタンスを事前に生成して使い回すような実装を検討すべきでしょう。GC.Alloc を撲滅することが目的ではなく、あくまでも 1 フレームあたりの処理時間を短くすることが最終目的であることを忘れないようにしましょう。

10.2 for/foreachについて

「2.6 アルゴリズムと計算量」で紹介したようにループはデータ数に依存して時間がかかるようになります。また、一見同じような処理に見えるループもコードの書き方次第で効率が変わります。

ここでは、SharpLab^{*1}を利用して、foreach/for を使用した List や配列の中身を 1 つずつ取得するだけのコードを IL から C# にデコンパイルした結果を見てみましょう。

まずは foreach でループを回した場合を見てみましょう。List への値の追加などは省略してます。

▼リスト 10.12 List を foreach で回す例

```
1: var list = new List<int>(128);
2: foreach (var val in list)
3: {
4: }
```

^{*1} <https://sharplab.io/>

第 10 章 Tuning Practice - Script (C#)

▼リスト 10.13 List を foreach で回す例のデコンパイル結果

```
1: List<int>.Enumerator enumerator = new List<int>(128).GetEnumerator();
2: try
3: {
4:     while (enumerator.MoveNext())
5:     {
6:         int current = enumerator.Current;
7:     }
8: }
9: finally
10: {
11:     ((IDisposable)enumerator).Dispose();
12: }
```

foreach で回した場合は、列挙子を取得して MoveNext() で次へ進めて Current で値を参照する実装になっていることがわかります。さらに、list.cs^{*2} の MoveNext() の実装を見ると、サイズのチェックなど各種プロパティアクセス回数が多くなっており、インデクサーによる直アクセスより処理が多くなるように見えます。

次に、for で回したときを見てみましょう。

▼リスト 10.14 List を for で回す例

```
1: var list = new List<int>(128);
2: for (var i = 0; i < list.Count; i++)
3: {
4:     var val = list[i];
5: }
```

▼リスト 10.15 List を for で回した際のデコンパイル結果

```
1: List<int> list = new List<int>(128);
2: int num = 0;
3: while (num < list.Count)
4: {
5:     int num2 = list[num];
6:     num++;
7: }
```

C# では for 文は while 文の糖衣構文であり、インデクサー (public T this[int index]) による参照で取得されていることがわかります。また、この while 文をよく見ると、条件式に list.Count が入っています。つまり、Count プロパティへのアクセ

^{*2} <https://referencesource.microsoft.com/#mscorlib/system/collections/generic/list.cs>

スがループを繰り返すたびに行われることになります。Count の数が多くなればなるほど、Count プロパティへのアクセス回数が比例して増加し、数によっては無視できない負荷になっていきます。もし、ループ内で Count が変わらないのであれば、ループの前でキャッシュしておくことでプロパティアクセスの負荷を削減できます。

▼リスト 10.16 List を for で回す例: 改良版

```
1: var count = list.Count;
2: for (var i = 0; i < count; i++)
3: {
4:     var val = list[i];
5: }
```

▼リスト 10.17 List を for で回す例: 改良版のデコンパイル結果

```
1: List<int> list = new List<int>(128);
2: int count = list.Count;
3: int num = 0;
4: while (num < count)
5: {
6:     int num2 = list[num];
7:     num++;
8: }
```

Count をキャッシュすることでプロパティアクセス回数が削減され、高速化されました。今回のループ中の比較はどちらも GC.Alloc による負荷ではなく、実装内容の違いによる差分となります。

また、配列の場合は foreach も最適化されており、for で記述したものとほぼ変化はないものとなります。

▼リスト 10.18 配列を foreach で回す例

```
1: var array = new int[128];
2: foreach (var val in array)
3: {
4: }
```

▼リスト 10.19 配列を foreach で回す例のデコンパイル結果

```
1: int[] array = new int[128];
2: int num = 0;
3: while (num < array.Length)
4: {
5:     int num2 = array[num];
6:     num++;
7: }
```

```
7: }
```

検証のため、データ数 10,000,000 として事前にランダムな数値をアサインし、List<int> データの和を計算するものとしました。検証環境は Pixel 3a、Unity 2021.3.1f1 で実施しました。

▼表 10.1 List<int>における記述方法ごとの計測結果

種類	Time ms
List: foreach	66.43
List: for	62.49
List: for (Count キャッシュ)	55.11
配列: for	30.53
配列: foreach	23.75

List<int> の場合は、条件を細かく揃えて比較してみると foreach よりも for、Count の最適化を施した for のほうがさらに速くなったことがわかります。List の foreach は、Count の最適化を施した for に書き換えることで、foreach の処理における MoveNext() や Current プロパティのオーバーヘッドを削減し、高速化が可能です。

また、List と配列のそれぞれ最速同士で比較すると、List より配列のほうが約 2.3 倍以上高速になりました。foreach と for で IL が同じ結果になるように記述しても、foreach が速い結果となり、配列の foreach が十分に最適化されていると言えるでしょう。

以上の結果から、データ数が多くかつ処理速度を高速にしなければならない場面については List<T>ではなく配列を検討すべきでしょう。しかし、フィールドに定義した List をローカルキャッシュせずそのまま参照してしまうなど、書き換えが不十分な場合は高速化できないこともありますので、foreach から for に変更する際には必ず計測をしつつ適切に書き換えましょう。

10.3 オブジェクトプーリング

随所で触れてきましたが、ゲーム開発では動的にオブジェクトを生成せずに、事前生成して使い回すことが重要です。これをオブジェクトプーリングと呼びます。たとえば、ゲームフェーズで使用予定のオブジェクトをロードフェーズでまとめて生成してプーリングしておき、使用するときはプールしているオブジェクトへの代入と参照のみ行いながら扱うことで、ゲームフェーズ中の GC.Alloc を避けることが可能です。

また、オブジェクトプリングはアロケーションの削減の他にも、画面を構成するオブジェクトを都度作り直すことなく画面遷移を可能にしておくことでロード時間の短縮を実現したり、計算コストが非常に高い処理の結果を保持しておいて重い計算を複数回実行することを避けたり、さまざまな場面で用いられます。

ここでは広義にオブジェクトと表現しましたが、これは最小単位のデータに留まらず、Coroutine や Action などにも該当します。たとえば、事前に Coroutine を想定される実行数分以上生成しておき、必要なタイミングで使用して使い潰していくようなことも検討しましょう。2 分間で終わるゲームで最大で 20 回実行されるようなときは Ienumerator をそれぞれ先に生成しておき、使用する時は StartCoroutine するだけにすることで生成コストは抑えられます。

10.4 string

string オブジェクトは文字列を表す System.Char オブジェクトのシーケンシャルコレクションです。string は使い方 1 つで GC.Alloc が簡単に起こります。たとえば、文字連結演算子 + を利用して 2 つの文字列の連結をすると、新しい string オブジェクトを生成することになります。string の値は生成後に変更できない（イミュータブル）ため、値の変更が行われているように見える操作は新しい string オブジェクトを生成して返しています。

▼リスト 10.20 文字列結合で string を作る場合

```

1: private string CreatePath()
2: {
3:     var path = "root";
4:     path += "/";
5:     path += "Hoge";
6:     path += "/";
7:     path += "Fuga";
8:     return path;
9: }
```

上記例の場合は、各文字列結合で string が生成されていき、合計で 164Byte アロケーションが発生します。

文字列が頻繁に変更されるときは、値が変更可能な StringBuilder を利用することで string オブジェクトの大量生成を防ぐことができます。文字連結や削除などの操作を StringBuilder オブジェクトで行い、最終的に値を取り出して string オブジェクトに ToString() することで、取得時のみのメモリアロケーションに抑えることができます。また、StringBuilder を使用する際には、Capacity を必ず設定するようにしま

しよう。未指定のときは初期値が 16 になり、Append などで文字数が増えバッファーが拡張されるときにメモリの確保と値のコピーが走るため、不用意な拡張が発生しない適切な Capacity を設定するようにしましょう。

▼リスト 10.21 StringBuilder で string を作る場合

```
1: private readonly StringBuilder _stringBuilder = new StringBuilder(16);
2: private string CreatePathFromStringBuilder()
3: {
4:     _stringBuilder.Clear();
5:     _stringBuilder.Append("root");
6:     _stringBuilder.Append("/");
7:     _stringBuilder.Append("Hoge");
8:     _stringBuilder.Append("/");
9:     _stringBuilder.Append("Fuga");
10:    return _stringBuilder.ToString();
11: }
```

StringBuilder を用いた例では、事前に StringBuilder を生成（上記例の場合、生成時に 112Byte アロケーション）しておけば、以降は生成した文字列を取り出す ToString() 時に掛かる 50Byte のアロケーションで済みます。

ただし、StringBuilder も値の操作中にアロケーションが起こりにくいだけで、前述のように ToString() 実行時には string オブジェクトを生成することになるため、GC.Alloc を避けたいときに使用するのは推奨されません。また、\$""構文は string.Format に変換され、string.Format の内部実装は StringBuilder が使われているため、結局は ToString() のコストは避けられません。前項のオブジェクトの使い回しをここでも応用し、あらかじめ使用される可能性のある文字列は string オブジェクトを事前に生成し、それを使うようにしましょう。

しかしながらゲーム中に文字列操作と string オブジェクトの生成をどうしても行わなければならない場合もあります。そういう場合には、文字列用のバッファーを事前に持つておいて、そのバッファーをそのまま使えるようにするような拡張を行う必要があります。unsafe なコードを自前で実装するか、ZString^{*3}のような Unity 向けの拡張機能（たとえば TextMeshPro への NonAlloc な適用機能）を備えたライブラリの導入を検討しましょう。

^{*3} <https://github.com/Cysharp/ZString>

10.5 LINQ と遅延評価

本節では LINQ の使用による GC.Alloc を軽減する方法と遅延評価のポイントについて解説します。

10.5.1 LINQ の使用による GC.Alloc を軽減する

LINQ の使用では、リスト 10.22 のような場合に GC.Alloc が発生します。

▼リスト 10.22 GC.Alloc が発生する例

```
1: var oneToTen = Enumerable.Range(1, 11).ToArray();
2: var query = oneToTen.Where(i => i % 2 == 0).Select(i => i * i);
```

リスト 10.22 で GC.Alloc が発生する理由は LINQ の内部実装に起因します。加えて、LINQ の一部メソッドは呼び出し側の型に合わせた最適化を行うため、呼び出し元の型によって GC.Alloc のサイズが変化します。

▼リスト 10.23 型ごとの実行速度検証

```
1: private int[] array;
2: private List<int> list;
3: private IEnumerable<int> ienumerable;
4:
5: public void GlobalSetup()
6: {
7:     array = Enumerable.Range(0, 1000).ToArray();
8:     list = Enumerable.Range(0, 1000).ToList();
9:     ienumerable = Enumerable.Range(0, 1000);
10: }
11:
12: public void RunAsArray()
13: {
14:     var query = array.Where(i => i % 2 == 0);
15:     foreach (var i in query){}
16: }
17:
18: public void RunAsList()
19: {
20:     var query = list.Where(i => i % 2 == 0);
21:     foreach (var i in query){}
22: }
23:
24: public void RunAsIEnumerable()
25: {
26:     var query = ienumerable.Where(i => i % 2 == 0);
27:     foreach (var i in query){}
28: }
```

リスト 10.23 に定義した各メソッドのベンチマークを測定すると図 10.1 のような結果が得られました。この結果から $T[] \rightarrow List<T> \rightarrow IEnumerable<T>$ の順番にヒープアロケーションのサイズが大きくなっていることがわかります。

このように、LINQ を使用する場合は、実行時の型を意識することで GC.Alloc のサイズを削減することができます。

Method	Mean	Error	StdDev	Ratio	RatioSD	Allocated
RunAsArray	4.210 us	0.2735 us	0.0150 us	1.00	0.00	48 B
RunAsList	4.942 us	0.2517 us	0.0138 us	1.17	0.00	72 B
RunAsIEnumerable	7.326 us	3.4885 us	0.1912 us	1.74	0.04	96 B

▲図 10.1 型ごとの実行速度比較

LINQ の GC.Alloc の原因

LINQ の使用による GC.Alloc の原因の一部は、LINQ の内部実装です。LINQ のメソッドは $IEnumerable<T>$ を受け取り、 $IEnumerable<T>$ を返すものが多く、この API 設計によりメソッドチェーンを用いた直感的な記述ができるようになっています。このときメソッドが返す $IEnumerable<T>$ の実体は、各機能に合わせたクラスのインスタンスとなっています。LINQ は内部的に $IEnumerable<T>$ を実装したクラスをインスタンス化してしまい、さらにループ処理を実現するために $GetEnumerator()$ の呼び出しなどが行われるため内部的に GC.Alloc が発生します。

10.5.2 LINQ の遅延評価

LINQ の `Where` や `Select` といったメソッドは、実際に結果が必要になるまで評価を遅らせる遅延評価となっています。一方で、`ToArray` のような即時評価となるメソッドも定義されています。

ここで、下記のリスト 10.24 のコードの場合を考えます。

▼リスト 10.24 即時評価を挟んだメソッド

```

1: private static void LazyExpression()
2: {
3:     var array = Enumerable.Range(0, 5).ToArray();
4:     var sw = Stopwatch.StartNew();
5:     var query = array.Where(i => i % 2 == 0).Select(HeavyProcess).ToArray();
6:     Console.WriteLine($"Query: {sw.ElapsedMilliseconds}");
7:
8:     foreach (var i in query)
9:     {
10:         Console.WriteLine($"diff: {sw.ElapsedMilliseconds}");
11:     }
12: }
13:
14: private static int HeavyProcess(int x)
15: {
16:     Thread.Sleep(1000);
17:     return x;
18: }
```

リスト 10.24 の実行結果がリスト 10.25 になります。即時評価となる `ToArray` を末尾に追加したことによって、`query` への代入時に `Where` や `Select` のメソッドを実行し値を評価した結果が返されます。そのため `HeavyProcess` も呼び出されるので、`query` を生成するタイミングで処理時間がかかっていることがわかります。

▼リスト 10.25 即時評価のメソッドを追加した結果

```

1: Query: 3013
2: diff: 3032
3: diff: 3032
4: diff: 3032
```

このように LINQ の即時評価のメソッドを意図せず呼び出してしまってその箇所がボトルネックになってしまふ可能性があります。`ToArray` や `OrderBy`, `Count` などシーケンスすべてを一度見る必要があるメソッドは即時評価となるため、呼び出し時のコストを意識して使用しましょう。

10.5.3 「LINQ の使用を避ける」という選択

LINQ を使用した際の `GC.Alloc` の原因や軽減方法、遅延評価のポイントについて解説しました。本節では LINQ を使用する基準について解説します。前提として LINQ は便利な言語機能ではありますが、使用するとヒープアロケーションや実行速度は使用しない場合に比べて悪化します。実際に Microsoft の Unity のパフォーマンスに関する

第 10 章 Tuning Practice - Script (C#)

推奨事項⁴では「Avoid use of LINQ」と明記されています。LINQ を使用した場合と、使用しない場合で同じロジックを実装した場合のベンチマークをリスト 10.26 で比較してみます。

▼リスト 10.26 LINQ の使用有無によるパフォーマンス比較

```
1: private int[] array;
2:
3: public void GlobalSetup()
4: {
5:     array = Enumerable.Range(0, 100_000_000).ToArray();
6: }
7:
8: public void Pure()
9: {
10:    foreach (var i in array)
11:    {
12:        if (i % 2 == 0)
13:        {
14:            var _ = i * i;
15:        }
16:    }
17: }
18:
19: public void UseLinq()
20: {
21:    var query = array.Where(i => i % 2 == 0).Select(i => i * i);
22:    foreach (var i in query)
23:    {
24:    }
25: }
```

結果は図 10.2 になります。実行時間を比較すると LINQ を使用しない場合に対して LINQ を使った処理は 19 倍ほど時間がかかってしまっていることがわかります。

Method	Mean	Error	StdDev	Ratio	RatioSD	Allocated
Pure	26.06 ms	3.230 ms	0.177 ms	1.00	0.00	26 B
UseLinq	514.55 ms	354.586 ms	19.436 ms	19.75	0.79	920 B

▲図 10.2 LINQ の使用有無によるパフォーマンス比較結果

上記の結果から LINQ を使用することによるパフォーマンスの悪化は明確ですが、LINQ を使用することでコーディングの意図が伝わりやすい場合などもあります。これ

⁴ <https://docs.microsoft.com/en-us/windows/mixed-reality/develop/unity/performance-recommendations-for-unity#avoid-expensive-operations>

らの挙動を把握した上で、LINQ を使用するか、使用する場合のルールなどはプロジェクト内で議論の余地があるかと思います。

10.6 `async/await` のオーバーヘッドの避け方

`async/await` は C#5.0 で追加された言語機能であり、非同期処理をコールバックを使わず一筋の同期的処理のように記述できるものです。

10.6.1 不要な箇所での `async` を避ける

`async` を定義されたメソッドは、コンパイラによって非同期処理を実現するためのコードが生成されます。そして `async` キーワードがあれば、コンパイラによるコード生成は必ず行われます。そのため、リスト 10.27 のように同期的に完了する可能性のあるメソッドも実際にはコンパイラによるコード生成が行われています。

▼リスト 10.27 同期的に完了する可能性のある非同期処理

```
1: using System;
2: using System.Threading.Tasks;
3:
4: namespace A {
5:     public class B {
6:         public async Task HogeAsync(int i) {
7:             if (i == 0) {
8:                 Console.WriteLine("i is 0");
9:                 return;
10:            }
11:            await Task.Delay(TimeSpan.FromSeconds(1));
12:        }
13:
14:        public void Main() {
15:            int i = int.Parse(Console.ReadLine());
16:            Task.Run(() => HogeAsync(i));
17:        }
18:    }
19: }
```

このリスト 10.27 のような場合は、同期的に終了する可能性のある `HogeAsync` を分割し、リスト 10.28 のように実装することで同期的に完了する場合に不要な `IAsyncStateMachine` 実装のステートマシン構造体を生成するコストを省略できます。

▼リスト 10.28 同期処理と非同期処理を分割した実装

```
1: using System;
2: using System.Threading.Tasks;
3:
4: namespace A {
5:     public class B {
6:         public async Task HogeAsync(int i) {
7:             await Task.Delay(TimeSpan.FromSeconds(1));
8:         }
9:
10:        public void Main() {
11:            int i = int.Parse(Console.ReadLine());
12:            if (i == 0) {
13:                Console.WriteLine("i is 0");
14:            } else {
15:                Task.Run(() => HogeAsync(i));
16:            }
17:        }
18:    }
19: }
```

async/await の仕組み

async/await 構文はコンパイル時にコンパイラによるコード生成を用いて実現されています。async キーワードのついたメソッドはコンパイル時点では `IAsyncStateMachine` を実装した構造体を生成する処理が追加され、await 対象の処理が完了するとステートを進めるステートマシンを管理することで `async/await` の機能を実現しています。また、この `IAsyncStateMachine` は `System.Runtime.CompilerServices` 名前空間に定義されたインターフェースであり、コンパイラのみが使用可能なものとなっています。

10.6.2 同期コンテキストのキャプチャを避ける

別スレッドに退避させた非同期処理から、呼び出し元のスレッドに復帰する仕組みが同期コンテキストであり、`await` を使用することで直前のコンテキストをキャプチャできます。この同期コンテキストのキャプチャは `await` のたびに行われるため、`await`ごとのオーバーヘッドが発生します。そのため、Unity での開発に広く利用されてい

る UniTask⁵では、同期コンテキストのキャプチャによるオーバーヘッドを避けるために ExecutionContext と SynchronizationContext を使用しない実装となっています。Unity に関してはこのようなライブラリを導入することで、パフォーマンスの改善が見られる場合があります。

10.7 stackalloc による最適化

配列の確保は通常ヒープ領域に確保されるため、ローカル変数として配列を確保すると、都度 GC.Alloc が発生してスパイクの原因となります。また、ヒープ領域への読み書きはスタック領域と比べると、少しですが効率が悪くなります。

そのため C#では、**unsafe** コード限定で、スタック上に配列を確保するための構文が用意されています。

リスト 10.29 のように、new キーワードを用いる代わりに、stackalloc キーワードを用いて配列を確保すると、スタック上に配列が確保されます。

▼リスト 10.29 stackalloc を用いたスタック上への配列確保

```

1: // stackallocはunsafe限定
2: unsafe
3: {
4:     // スタック上にintの配列を確保
5:     byte* buffer = stackalloc byte[BufferSize];
6: }
```

C# 7.2 から Span<T>構造体を用いることで、リスト 10.30 に示すように **unsafe** なしで stackalloc を利用できるようになりました。

▼リスト 10.30 Span<T>構造体を併用したスタック上への配列確保

```
1: Span<byte> buffer = stackalloc byte[BufferSize];
```

Unity の場合は 2021.2 から標準で利用できます。それ以前バージョンの場合は Span<T>が存在しないため、System.Memory.dll を導入する必要があります。

stackalloc で確保した配列はスタック専用なため、クラスや構造体のフィールドに持てません。必ずローカル変数として使う必要があります。

スタック上への確保とはいえ、要素数が大きい配列の確保はそれなりに処理時間がかかります。もし Update ループ内などのヒープアロケーションを避けたい箇所で要

⁵ <https://tech.cygames.co.jp/archives/3417/>

素数の大きい配列を利用したい場合は、初期化時の事前確保か、オブジェクトプールのようなデータ構造を用意して、利用時に貸し出すような実装のほうがよいでしょう。

また、`stackalloc` で確保したスタック領域は、関数を抜けるまで解放されない点に注意が必要です。たとえばリスト 10.31 に示すコードは、ループ内で確保した配列はすべて保持され、`Hoge` メソッドを抜けるときに解放されるので、ループを回しているうちに `Stack Overflow` を起こす可能性があります。

▼リスト 10.31 `stackalloc` を用いたスタック上への配列確保

```
1: unsafe void Hoge()
2: {
3:     for (int i = 0; i < 10000; i++)
4:     {
5:         // ループ数分配列が蓄積される
6:         byte* buffer = stackalloc byte[10000];
7:     }
8: }
```

10.8 `sealed` による IL2CPP バックエンド下でのメソッド呼び出しの最適化

Unity で IL2CPP をバックエンドとしてビルドをすると、クラスの `virtual` なメソッド呼び出しを実現するために、C++ の `vtable` のような仕組みを用いてメソッド呼び出しを行います⁶。

具体的には、クラスのメソッド呼び出しの定義ごとに、リスト 10.32 に示すようなコードが自動生成されます。

▼リスト 10.32 IL2CPP が生成するメソッド呼び出しに関する C++ コード

```
1: struct VirtActionInvoker0
2: {
3:     typedef void (*Action)(void*, const RuntimeMethod* );
4:
5:     static inline void Invoke (
6:         Il2CppMethodSlot slot, RuntimeObject* obj)
7:     {
8:         const VirtualInvokeData& invokeData =
9:             il2cpp_codegen_get_virtual_invoke_data(slot, obj);
10:        ((Action)invokeData.methodPtr)(obj, invokeData.method);
11:    }
12:};
```

⁶ <https://blog.unity.com/technology/il2cpp-internals-method-calls>

10.8 sealed による IL2CPP バックエンド下でのメソッド呼び出しの最適化

これは virtual なメソッドだけでなく、コンパイル時に継承をしていない、virtual でないメソッドであっても同様な C++ コードを生成します。このような自動生成の挙動によって、コードサイズが肥大化したり、メソッド呼び出しの処理時間が増大します。

この問題は、クラスの定義に sealed 修飾子をつけることで回避できます⁷。

リスト 10.33 のようなクラスを定義してメソッドを呼び出した場合、IL2CPP で生成された C++ コードではリスト 10.34 のようなメソッド呼び出しが行われます。

▼リスト 10.33 sealed を用いないクラス定義とメソッド呼び出し

```
1: public abstract class Animal
2: {
3:     public abstract string Speak();
4: }
5:
6: public class Cow : Animal
7: {
8:     public override string Speak() {
9:         return "Moo";
10:    }
11: }
12:
13: var cow = new Cow();
14: // Speakメソッドを呼び出す
15: Debug.LogFormat("The cow says '{0}'", cow.Speak());
```

▼リスト 10.34 リスト 10.33 のメソッド呼び出しに対応する C++ コード

```
1: // var cow = new Cow();
2: Cow_t1312235562 * L_14 =
3:     (Cow_t1312235562 *)il2cpp_codegen_object_new(
4:         Cow_t1312235562_il2cpp_TypeInfo_var);
5: Cow__ctor_m2285919473(L_14, /*hidden argument*/NULL);
6: V_4 = L_14;
7: Cow_t1312235562 * L_16 = V_4;
8:
9: // cow.Speak()
10: String_t* L_17 = VirtFuncInvoker0< String_t* >::Invoke(
11:     4 /* System.String AssemblyCSharp.Cow::Speak() */, L_16);
```

リスト 10.34 に示すように、virtual なメソッド呼び出しではないにもかかわらず VirtFuncInvoker0< String_t* >::Invoke を呼び出しており、virtual メソッドのようなメソッド呼び出しが行われていることが確認できます。

⁷ <https://blog.unity.com/technology/il2cpp-optimizations-devirtualization>

第 10 章 Tuning Practice - Script (C#)

一方で、リスト 10.33 の Cow クラスをリスト 10.35 に示すように sealed 修飾子を用いて定義すると、リスト 10.36 のような C++ コードが生成されます。

▼リスト 10.35 sealed を用いたクラス定義とメソッド呼び出し

```
1: public sealed class Cow : Animal
2: {
3:     public override string Speak() {
4:         return "Moo";
5:     }
6: }
7:
8: var cow = new Cow();
9: // Speakメソッドを呼び出す
10: Debug.LogFormat("The cow says '{0}'", cow.Speak());
```

▼リスト 10.36 リスト 10.35 のメソッド呼び出しに対応する C++ コード

```
1: // var cow = new Cow();
2: Cow_t1312235562 * L_14 =
3:     (Cow_t1312235562 *)il2cpp_codegen_object_new(
4:         Cow_t1312235562_il2cpp_TypeInfo_var);
5: Cow__ctor_m2285919473(L_14, /*hidden argument*/NULL);
6: V_4 = L_14;
7: Cow_t1312235562 * L_16 = V_4;
8:
9: // cow.Speak()
10: String_t* L_17 = Cow_Speak_m1607867742(L_16, /*hidden argument*/NULL);
```

このように、メソッド呼び出しが Cow_Speak_m1607867742 を呼び出しており、直接メソッドを呼び出していることが確認できます。

ただし、比較的最近の Unity では、このような最適化では一部自動で行われていることを Unity 公式で明言しています⁸。

つまり、sealed を明示的に指定しない場合でも、このような最適化が自動で行われている可能性があります。

しかし、「[il2cpp] Is ‘sealed’ Not Worked As Said Anymore In Unity 2018.3?」⁸というフォーラムで言及している通り、2019 年 4 月の段階で、この実装は完全というわけではありません。

このような現状から、IL2CPP の生成するコードを確認しながら、プロジェクトごとに sealed 修飾子の設定を決めるといいでしょう。

⁸ <https://forum.unity.com/threads/il2cpp-is-sealed-not-worked-as-said-anymore-in-unity-2018-3.659017/#post-4412785>

より確実に直接的なメソッド呼び出しを行うために、また、今後の IL2CPP の最適化を期待して、最適化可能なマークとして `sealed` 修飾子を設定するのもよいかもしれません。

10.9 インライン化による最適化

メソッド呼び出しには多少のコストがかかります。そのため、C#に限らず一般的な最適化として、比較的小さいメソッドの呼び出しは、コンパイラなどによってインライン化という最適化が行われます。

具体的には、リスト 10.37 のようなコードに対して、インライン化によってリスト 10.38 のようなコードが生成されます。

▼リスト 10.37 インライン化前のコード

```

1: int F(int a, int b, int c)
2: {
3:     var d = Add(a, b);
4:     var e = Add(b, c);
5:     var f = Add(d, e);
6:
7:     return f;
8: }
9:
10: int Add(int a, int b) => a + b;

```

▼リスト 10.38 リスト 10.37 に対してインライン化を行ったコード

```

1: int F(int a, int b, int c)
2: {
3:     var d = a + b;
4:     var e = b + c;
5:     var f = d + e;
6:
7:     return f;
8: }

```

インライン化はリスト 10.38 のように、リスト 10.37 の Func メソッド内の `Add` メソッドの呼び出しを、メソッド内の内容をコピーして展開することで行われます。

IL2CPP では、コード生成時にはとくにインライン化による最適化は行われません。

しかし、Unity 2020.2 からメソッドに `MethodImpl` 属性を指定し、そのパラメーターに `MethodOptions.AggressiveInlining` を指定することで、生成される C++ コードの対応する関数に `inline` 指定子が付与されるようになりました。つまり、C++ のコードレベルでのインライン化が行えるようになりました。

インライン化のメリットは、メソッド呼び出しのコストが削減されるだけでなく、メソッド呼び出し時に指定した引数のコピーも省けることです。

たとえば算術系のメソッドは、`Vector3` や `Matrix` のような、比較的大きい構造体を複数個引数に取ります。構造体はそのまま引数として渡すと、すべて値渡しとしてコピーされてメソッドに渡されるため、引数の個数や渡す構造体のサイズが大きいと、メソッド呼び出しと引数のコピーでかなりの処理コストがかかる可能性があります。また、物理演算やアニメーションの実装など、定期処理などで利用されることが多いため、メソッド呼び出しが処理負荷として見逃せないケースになることがあります。

このようなケースでは、インライン化による最適化は有効です。実際に、Unity の新しい算術系ライブラリである **Unity.Mathematics** では、いたるメソッド呼び出しに `MethodOptions.AggressiveInlining` が指定されています⁹。

一方で、インライン化はメソッド内の処理を展開する処理のため、展開した分コードサイズが増大するというデメリットがあります。

そのため、とくに 1 フレームで頻繁に呼び出されホットパスとなるようなメソッドに対して、インライン化を検討するとよいでしょう。また、属性を指定すると必ずインライン化が行われるわけではない点にも注意が必要です。

インライン化されるメソッドは、その中身が小さいものに限定されるため、インライン化を行いたいメソッドは処理を小さく保つ必要があります。

また、Unity 2020.2 以前では属性指定に対して `inline` 指定子がつかないと、C++ の `inline` 指定子を指定してもインライン化が確実に行われる保証はありません。

そのため確実にインライン化を行いたい場合、可読性は落ちますがホットパスとなるメソッドは手動でのインライン化も検討するとよいでしょう。

⁹ <https://github.com/Unity-Technologies/Unity.Mathematics/blob/f476dc88954697f71e5615b5f57462495bc973a7/src/Unity.Mathematics/math.cs#L1894>



PERFORMANCE TUNING BIBLE

CHAPTER

11

第11章

Tuning Practice
— Player Settings —

CyberAgent Smartphone Games & Entertainment

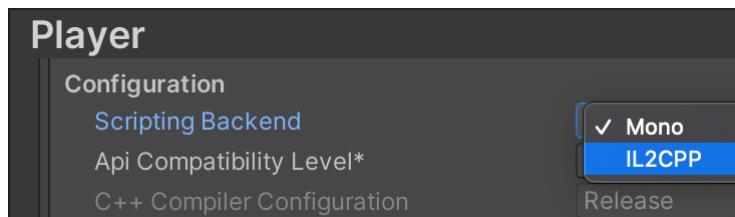
第 11 章

Tuning Practice - Player Settings

この章では、パフォーマンスに影響を与える Project Settings にある Player の項目について紹介します。

11.1 Scripting Backend

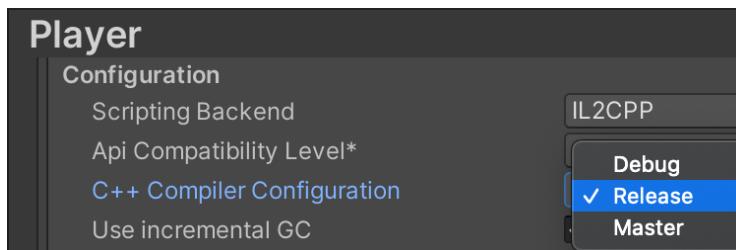
Unity では、Android や Standalone (Windows、macOS、Linux) といったプラットフォームで、Scripting Backend を Mono か IL2CPP のどちらかから選択できます。第 2 章「基礎知識」の「IL2CPP」で述べたとおりパフォーマンスが向上するため、IL2CPP を選択することを推奨します。



▲図 11.1 Scripting Backend の設定

また、Scripting Backend を IL2CPP に変更すると、一部プラットフォームを除いて **C++ Compiler Configuration** が選択できるようになります。

11.2 Strip Engine Code / Managed Stripping Level



▲図 11.2 C++ Compiler Configuration の設定

ここでは Debug、Release、Master から選べますが、それぞれビルド時間と最適化度合いのトレードオフがありますので、ビルドの目的に応じて使い分けるとよいでしょう。

11.1.1 Debug

最適化が行われないためランタイムでのパフォーマンスは良くありませんが、ビルド時間は他の設定と比較してもっとも短くなります。

11.1.2 Release

最適化によりランタイムのパフォーマンスが向上し、ビルドしたバイナリのサイズもより小さくなりますが、ビルドに要する時間は伸びます。

11.1.3 Master

そのプラットフォームで利用可能なすべての最適化が有効化されます。たとえば Windows 向けのビルドでは、リンク時コード生成 (LTCG) が使用されるなど、よりアグレッシブな最適化が行われます。その代わりとしてビルド時間は Release 設定よりもさらに伸びますが、それが許容できる場合、製品版のビルドには Master 設定を使用することを Unity は推奨しています。

11.2 Strip Engine Code / Managed Stripping Level

Strip Engine Code は Unity の機能から、**Managed Stripping Level** は C#をコンパイルして生成される CIL バイトコードから、それぞれ使用されないコードを取り除くことにより、ビルドしたバイナリのサイズを削減する効果が期待できます。

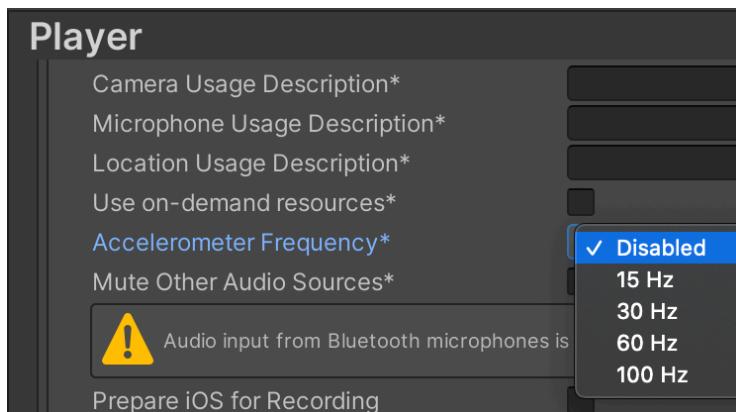
しかしながら、あるコードが使用されているかどうかの判定は静的解析に強く依存

しているため、コード中で直接参照されていない型や、リフレクションで動的に呼び出しているコードが誤って除去されてしまう場合があります。

その場合には **link.xml** ファイルや、**Preserve** 属性を指定することにより除去されることを回避できます。^{*1}

11.3 Accelerometer Frequency (iOS)

iOS 固有の設定で、加速度センサーのサンプリング周波数を変更できます。初期設定では 60Hz になっているため、適切な周波数に設定しましょう。とくに加速度センサーを利用していない場合は、必ず設定を無効化しましょう。



▲図 11.3 サンプリング周波数の設定

^{*1} <https://docs.unity3d.com/2020.3/Documentation/Manual/ManagedCodeStripping.html>



PERFORMANCE TUNING BIBLE

CHAPTER

12

第12章

Tuning Practice
— Third Party —

CyberAgent Smartphone Games & Entertainment

第 12 章

Tuning Practice - Third Party

この章では、Unity でゲームを開発する際によく使用されるサードパーティライブラリを導入する上で、パフォーマンスの観点から気をつけるべきことを紹介します。

12.1 DOTween

DOTween^{*1}は、スクリプトで滑らかなアニメーションを実現できるライブラリです。たとえば、拡大して縮小するアニメーションは以下のコードのように簡単に記述できます。

▼リスト 12.1 DOTween 使用例

```
1: public class Example : MonoBehaviour {
2:     public void Play() {
3:         DOTween.Sequence()
4:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
5:             .Append(transform.DOScale(Vector3.one, 0.125f));
6:     }
7: }
```

12.1.1 SetAutoKill

DOTween.Sequence() や transform.DOScale(...) など、Tween を生成する処理は基本的にメモリアロケーションを伴うため、頻繁に再生されるアニメーションはインスタンスの再利用を検討しましょう。

デフォルトではアニメーション完了時に自動的に Tween が破棄されてしまうので、SetAutoKill(false) でこれを抑制します。最初の使用例は、次のコードに置き換えることができます。

^{*1} <http://dotween.demigiant.com/index.php>

▼リスト 12.2 Tween インスタンスを再利用する

```

1:     private Tween _tween;
2:
3:     private void Awake() {
4:         _tween = DOTween.Sequence()
5:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
6:             .Append(transform.DOScale(Vector3.one, 0.125f))
7:             .SetAutoKill(false)
8:             .Pause();
9:     }
10:
11:    public void Play() {
12:        _tween.Restart();
13:    }

```

`SetAutoKill(false)` を呼び出した Tween は、明示的に破棄しなければリークしてしまうため注意が必要です。不要になったタイミングで `Kill()` を呼び出すか、後述する **SetLink** を使用するとよいでしょう。

▼リスト 12.3 Tween を明示的に破棄する

```

1:     private void OnDestroy() {
2:         _tween.Kill();
3:     }

```

12.1.2 SetLink

`SetAutoKill(false)` を呼び出した Tween や、`SetLoops(-1)` で無限に繰り返し再生されるようにした Tween は自動的には破棄されなくなるため、そのライフタイムを自前で管理する必要があります。そのような Tween には `SetLink(gameObject)` で関連する `GameObject` と紐付け、`GameObject` が `Destroy` されると同時に Tween も破棄されるようにするとよいでしょう。

▼リスト 12.4 Tween を GameObject のライフタイムに紐付ける

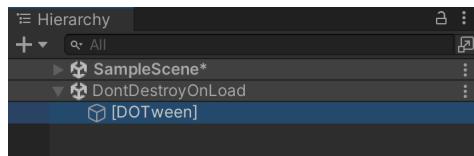
```

1:     private void Awake() {
2:         _tween = DOTween.Sequence()
3:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
4:             .Append(transform.DOScale(Vector3.one, 0.125f))
5:             .SetAutoKill(false)
6:             .SetLink(gameObject)
7:             .Pause();
8:     }

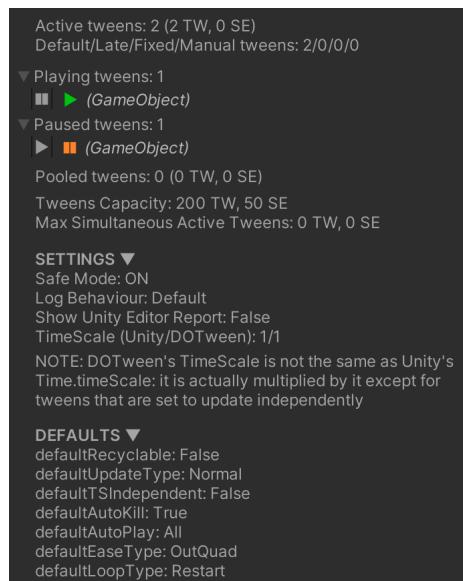
```

12.1.3 DOTween Inspector

Unity Editor で再生中に、[DOTween] という名前の GameObject を選択することで、Inspector 上から DOTween の状態や設定を確認できます。



▲図 12.1 [DOTween] GameObject



▲図 12.2 DOTween Inspector

関連する GameObject が破棄されているにもかかわらず動き続けている Tween がないか、また、Pause 状態で破棄されずにリープしている Tween がないかなど調査するときに役立ちます。

12.2 UniRx

UniRx^{*2}は Unity に最適化された Reactive Extensions を実装したライブラリです。豊富なオペレーター群や Unity 向けのヘルパーにより、複雑な条件のイベントハンドリングを簡潔に記述できます。

12.2.1 購読の解除

UniRx では、ストリーム発行元の `I0bservable` に対して購読 (`Subscribe`) することで、そのメッセージの通知を受け取ることができます。

この購読時に、通知を受け取るためのオブジェクトや、メッセージを処理するコードバックなどのインスタンスが生成されます。これらのインスタンスが `Subscribe` した側の寿命を超えてメモリに残り続けるのを避けるため、通知を受け取る必要がなくなった場合は、基本的に `Subscribe` した側の責任で購読を解除しましょう。

購読を解除する方法はいくつかありますが、パフォーマンスを考慮する場合 `Subscribe` の戻り値の `IDisposable` を保持して明示的に `Dispose` するのがよいでしょう。

```

1: public class Example : MonoBehaviour {
2:     private IDisposable _disposable;
3:
4:     private void Awake() {
5:         _disposable = Observable.EveryUpdate()
6:             .Subscribe(_ => {
7:                 // 毎フレーム実行する処理
8:             });
9:     }
10:
11:    private void OnDestroy() {
12:        _disposable.Dispose();
13:    }
14: }
```

また `MonoBehaviour` を継承したクラスであれば `AddTo(this)` を呼ぶことで、自身が `Destroy` されるタイミングで自動的に解除することもできます。`Destroy` を監視するため内部的に `AddComponent` が呼ばれるオーバーヘッドがありますが、記述が簡潔になるこちらを利用するのもよいでしょう。

^{*2} <https://github.com/neuecc/UniRx>

```
1:     private void Awake() {
2:         Observable.EveryUpdate()
3:             .Subscribe(_ => {
4:                 // 每フレーム実行する処理
5:             })
6:             .AddTo(this);
7:     }
```

12.3 UniTask

UniTask は Unity でハイパフォーマンスな非同期処理を実現するための強力なライブラリで、値型ベースの UniTask 型によりゼロアロケーションで非同期処理を行えることが特徴です。また Unity の PlayerLoop に沿った実行タイミングの制御も可能なため、従来のコルーチンを完全に置き換えることができます。

12.3.1 UniTask v2

UniTask は 2020 年 6 月、メジャーバージョンアップになる UniTask v2 がリリースされました。UniTask v2 は async メソッド全体のゼロアロケーション化など大幅な性能改善と、非同期 LINQ 対応や外部アセットの await サポートなどの機能追加が行われています。³

一方で、UniTask.Delay(...) など Factory で返すタスクが呼び出し時に起動されたり、通常の UniTask インスタンスへの複数回 await が禁止⁴されるなど、破壊的な変更も含まれるため UniTask v1 からアップデートする際は注意が必要です。しかしながらアグレッシブな最適化によりさらにパフォーマンスが向上しているため、基本的には UniTask v2 を使用するとよいでしょう。

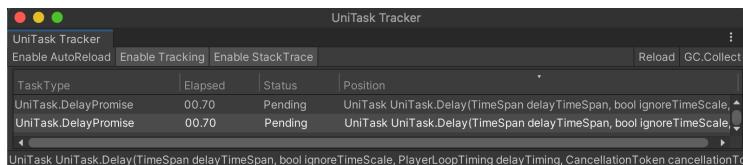
12.3.2 UniTask Tracker

UniTask Tracker を使用することで待機中の UniTask と、その生成時のスタックトレースを可視化できます。

³ <https://tech.cygames.co.jp/archives/3417/>

⁴ UniTask.Preserve を使用することで複数回 await できる UniTask に変換することができます。

12.3 UniTask



▲図 12.3 UniTask Tracker

たとえば、何かに衝突したら`_hp`が1ずつ減っていくMonoBehaviourがあるとします。

```
1: public class Example : MonoBehaviour {
2:     private int _hp = 10;
3:
4:     public UniTask WaitForDeadAsync() {
5:         return UniTask.WaitUntil(() => _hp <= 0);
6:     }
7:
8:     private void OnCollisionEnter(Collision collision) {
9:         _hp -= 1;
10:    }
11: }
```

このMonoBehaviourの`_hp`が減り切る前にDestroyされた場合、それ以上`_hp`が減ることはないと想定してWaitForDeadAsyncの戻り値のUniTaskは完了する機会を失い、そのままずっと待機し続けてしまいます。

このように終了条件の設定ミスなどでリークしているUniTaskがないか、このツールを用いて確認するとよいでしょう。

タスクのリークを防ぐ

例示したコードでタスクがリークするのは、終了条件を満たす前に自身がDestroyされるケースを考慮できていないのが原因でした。

これには、シンプルに自身がDestroyされていないかチェックする。または自身への`this.GetCancellationTokenOnDestroy()`で得られる`CancellationToken`を`WaitForDeadAsync`へ渡し、Destroy時にタスクがキャンセルされるようにする、といった対応が考えられます。

```
1: // 自身がDestroyされているかチェックするパターン
2: public UniTask WaitForDeadAsync() {
3:     return UniTask.WaitUntil(() => this == null || _hp <= 0);
4: }
5:
6: // CancellationTokenを渡すパターン
7: public UniTask WaitForDeadAsync(CancellationToken token) {
8:     return UniTask.WaitUntil(
9:         () => _hp <= 0,
10:        cancellationToken: token);
11: }
```

▼リスト 12.9 WaitForDeadAsync(CancellationToken)呼び出し例

```
1: Example example = ...
2: var token = example.GetCancellationTokenOnDestroy();
3: await example.WaitForDeadAsync(token);
```

Destroy時に前者の UniTask は何事もなく完了しますが、後者は Operation CanceledException が投げられます。どちらの挙動が望ましいかは状況によって異なりますので、適切な実装を選択するとよいでしょう。

さいごに

本書はここで終わりです。本書を通して「パフォーマンスチューニングには自信がない」という方が「なんとなく分かった、やってみたい」と思えるようになったなら幸いです。プロジェクト内で実践する人が増えると、問題への対処が格段に早くなり、プロジェクトの安定感は増していくことでしょう。

また本書で紹介した内容では解決できない複雑な事象に出会うことがあるかもしれません。しかし、そんなときでもやることは変わらないでしょう。それはプロファイルを行い、原因を分析し、何かしらの手を打っていくことです。

ここから先は実践を通してあなたの知識や経験、そして発想力を存分に活かしてください。そうしてパフォーマンスチューニングを楽しんでもらえると幸いです。最後までお読み頂きありがとうございました。

著者紹介

本書に携わった著者を紹介します。なお、各著者のプロフィールや担当箇所は執筆時点のものになります。

飯田 卓也（Takuya Iida）

株式会社グレンジ所属、SGE コア技術本部兼務 / エンジニアリングマネージャー

第1章「パフォーマンスチューニングを始めよう」、第3章「プロファイリングツール」などの執筆を担当。現在は子会社を跨いで最適化に関わっています。業務上さまざまなことをしていますが、開発速度や品質の向上を目指して日々励んでいます。

矢野 春樹（Haruki Yano） / Twitter : @harumak_11 / GitHub : Haruma-K

株式会社サイバーエージェント SGE コア技術本部 / クライアントサイドエンジニア

第2章「基礎知識」の「2.2 レンダリング」、「2.3 データの表現方法」などの執筆を担当。開発効率向上のための共通基盤開発を業務の主軸としています。業務でも個人でも Unity 用のいろんな OSS を開発して公開しています。Unity ブログ LIGHT11 も運用中。

石黒 祐輔（Yusuke Ishiguro）

株式会社サイバーエージェント SGE コア技術本部

第2章「基礎知識」の一部と、第5章「Tuning Practice - AssetBundle」の執筆を担当。Ameba ゲーム（現クオリアーツ）の基盤開発チームに Unity エンジニアとして配属され、リアルタイム基盤、チャット基盤、AssetBundle 管理基盤「Octo」、認証・課金基盤などさまざまな基盤の開発に従事。現在は SGE コア技術本部に異動し、基盤全般の開発をリードしつつ、ゲーム事業部全体の開発効率と品質の最適化に注力。

袴田 大貴（Daiki Hakamata）

株式会社サイバーエージェント SGE コア技術本部

第9章「Tuning Practice - Script (Unity)」の執筆を担当。株式会社グレンジ、株式会社ジークレストにてゲーム開発・運用に従事。現在は SGE コア技術本部に所属し基盤を開発中。

中村 光寿 (NAKAMURO.) / Twitter: @megalo_23

株式会社アリボット所属 / ゲームクリエイター

「2.5 C#の基礎知識」、第 10 章「Tuning Practice - Script (C#)」の前半の執筆を担当。本書を執筆することで開発終盤にヘルプで呼ばれる機会を減らし、新しいゲームを開発する時間を確保することを企んでいる。ゲーム開発では最適化やディレクション、譜面制作から声の出演までと活動は幅広い。個人では Famulite Lab. でアプリ運営中。

大庭 俊介 (Shunsuke Ohba) / Twitter : @ohbashunsuke

株式会社サムザップ所属 / エンジニアリングマネージャー

第 4 章「Tuning Practice - Asset」の執筆を担当。元デザイナーのエンジニア。Flash を使ったインタラクティブな Web サイトの制作後、株式会社サイバーエージェントに中途入社。アーバンゲーミングの開発後、Unity エンジニアに転向。麻雀、ピンボール、リアルタイムバトルなど数多くのゲームの立ち上げにエンジニアリーダーとして参画。個人では Twitter やブログ「渋谷ほととぎす通信 (<https://shibuya24.info>)」で情報発信しています。

石井 岳 (Gaku Ishii)

株式会社サムザップ所属 / サーバー兼クライアントサイドエンジニア

第 11 章「Tuning Practice - Player Settings」、第 12 章「Tuning Practice - Third Party」の執筆を担当。株式会社サムザップへ配属後、Unity エンジニアとして新規ゲームアプリの開発に従事する。複数のアプリのリリースに携わった後、サーバーサイドエンジニアへ転向。現在サムザップではサーバーサイドエンジニア、SGE コア技術本部では Unity エンジニアとしてサーバー/クライアント両面で活動中。

斎藤 俊介 (Shunsuke Saito) / Twitter: @shun_shun_mummy

株式会社カラフルパレット所属 / クライアントサイドエンジニア

第 10 章「Tuning Practice - Script (C#)」の一部の執筆を担当。株式会社カラフルパレットに配属後、担当プロジェクトでクライアントサイドのリアルタイム通信の設計・実装や、UI システム周りの開発に従事。また、既存機能のチューニングやテンプレートコードの自動生成ツール開発なども行う。

付録 著者紹介

田村 和範 (Kazunori Tamura)

株式会社クオリアーツ所属

第8章「Tuning Practice - UI」の執筆を担当。株式会社クオリアーツにて、Unityエンジニアとしてゲーム開発や社内基盤開発に従事。社内基盤は主にUIに関わるものを開発。AIによるゲーム開発の効率化にも興味を持ち、ゲーム事業部内でのAIの活用を目指して奮闘中。

山口 智也 (Tomoya Yamaguchi) / Twitter: @togucchi

株式会社カラフルパレット所属 / クライアントサイドエンジニア

第7章「Tuning Practice - Graphics」の執筆を担当。株式会社カラフルパレットにて3D描画やライブ関連のシステムの開発に従事。現在は3D関連の新規技術の検証などを行っている。

向井 祐一郎 (Yuichiro Mukai) / Twitter: @yucchily_

株式会社アトリボット所属 / クライアントサイドエンジニア

第6章「Tuning Practice - Physics」と第10章「Tuning Practice - Script (C#)」の一部の執筆を担当。

Unity パフォーマンスチューニング バイブル

2022 年 7 月 8 日 初版第 1 刷 発行

著 者 株式会社 サイバーエージェント SGE コア技術本部
デザイン 株式会社 サイバーエージェント SGE 統括本部 PR デザイン室
発行所 株式会社 サイバーエージェント



CyberAgent Smartphone Games & Entertainment

CyberAgent.[®]