インテルの x86 プラットフォーム向け Unity* 最適化ガイド: パート 2

この記事は、インテル® デベロッパー・ゾーンに公開されている「Unity* Optimization Guide for Intel x86 Platforms: Part 2」の日本語参考訳です。

目次

チュートリアルのパート 1 に戻る:
インテルの x86 プラットフォーム向け Unity* 最適化ガイド: パート 1

最適化

このガイドでは、最適化の 2 つの主な分野 (スクリプトおよびエディターベース) について説明します。機能に基づいた各最適化に固有の項目があります。

スクリプトの最適化

スクリプトの錐台カリングとコルーチン

アプリケーションをプロファイルして、スクリプトの Update() 関数をすべてのフレームで呼び出す必要がないことが分かった場合、いくつかの手法により更新の量を減らすことができます。

- **スクリプト錐台カリング**
  - 次の Monobehavior コールバックを使用して、フォーカスされていない場合に更新する必要がない、カメラ錐台外のスクリプトをカリングします。

```
0 references
void OnBecameVisible()
{
    enabled = true;
}

0 references
void OnBecameInvisible()
{
    enabled = false;
}
```

図 12. オブジェクトがカメラ錐台から出た (またはカメラ錐台に入った) 場合に行われる Monobehavior コールバック

- **コルーチン**
  - コルーチンは、実行を一時停止および再開する基本的な機能です。コルーチンを活用するには、スクリプトのオリジナルの Update() 関数をコルーチンに置換します。次に、**yield** コマンドを使用して、コルーチンを呼び出す頻度を設定します。次のコードは、デフォルト設定 (フレームごとに 1 回コルーチンを呼び出す) ではなく、2 秒ごとにコルーチンを呼び出すカスタム更新を行います。

```
0 references
void Start()
{
    StartCoroutine("MyCoroutine");
}

0 references
IEnumerator MyCoroutine()
{
    while (true)
    {
        MoreEfficientUpdate();
        yield return new WaitForSeconds(2.0f);
    }
}

1 reference
void MoreEfficientUpdate()
{
    Debug.Log("I'm so efficient!");
}
```

図 13. コルーチンを使用したより効率的な更新

スマートなメモリー管理

メモリー使用量を最適化する場合、最初に Unity* Profiler で確認すると良いでしょう。メモリー管理の概要を調べるには、[Overview (概要)] ウィンドウの [GC Alloc (ガベージ・コレクションの割り当て)] セクション (図 14) を確認して、目的とする割り当てが見つかるまでフレームを進めます。

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
MemoryAllocation.Update()	0.5%	0.5%	1	0.5 MB	0.07	0.07
GameView.GetMainGameViewRenderRect()	0.1%	0.1%	1	40 B	0.02	0.02
AudioManager.FixedUpdate	0.0%	0.0%	1	0 B	0.00	0.00
AudioManager.Update	0.0%	0.0%	1	0 B	0.00	0.00
Camera.Render	0.4%	0.0%	1	0 B	0.06	0.01

図 14. フレームを調査し、ガベージ・コレクションがいつ発生して調整されているか判断する

ガベージ・コレクションが呼び出されている頻度を確認することも役立ちます。頻度を確認するには、[CPU Usage (CPU 使用状況)] サブプロファイラーの [GarbageCollector (ガベージコレクター)] フィールドを選択します (図 15)。

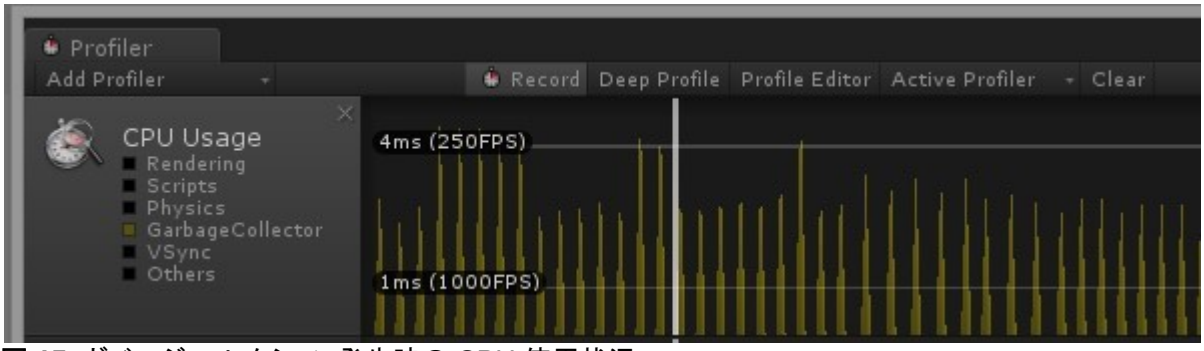

図 15. ガベージ・コレクション発生時の CPU 使用状況

収集が表示されたら、グラフの山をクリックして GC.Collect の呼び出しを見つけ (図 16)、各 GC.Collect にかかる時間を確認します。

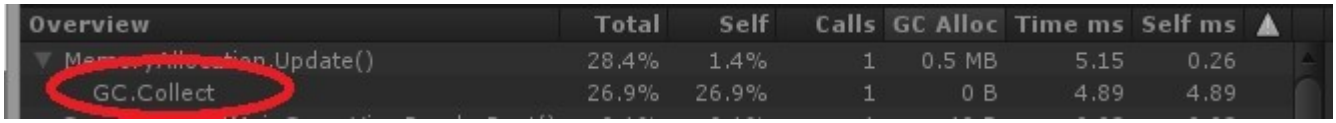

図 16. ガベージ・コレクションの統計

頻繁な割り当てを防ぐには、クラスの代わりに構造体を使用し、ヒープの代わりにスタックで割り当てを行います。ヒープで複数の割り当てを行うと、大量のメモリー断片化と高い頻度のガベージ・コレクションが発生します。

繰り返し使用されるオブジェクトとコンポーネントをキャッシュする

通常、アプリケーションを解析して最も頻繁に使用されるゲーム・オブジェクトとコンポーネントを特定し、これらの値がキャッシュされていることを確認すべきです。オブジェクトがフェッチされるすべてのシーンで、オブジェクトをキャッシュして不要な計算を排除できる可能性があります。

これは、ゲーム・オブジェクトの**インスタンス化**にも当てはまります。一般に、インスタンス化は比較的時間のかかる呼び出しであるため回避すべきです。すべてのシーンで同じオブジェクト型を繰り返し作成・破棄している場合、オブジェクト・マネージャー・スクリプトで再利用されるオブジェクトのリストを管理することで利点が得られます。

Unity* は、ゲーム・マネージャーを使用して、キャッシュされたすべてのゲーム・オブジェクトのリストを管理することを推奨しています。この手法を実装した後、[State (ステート)] ボタンを切り替えて 2 つの手法間のパフォーマンス・デルタを比較する次のコード、またはリアルタイムで **CPU 使用状況プロファイラー**を表示中にほかの制御メカニズムを追加できます。使用状況の差を示すコードを次に示します (図 17)。

```
// Update is called once per frame
0 references
void Update () {
    if(mMyState == STATE.GOOD)
    {
        Profiler.BeginSample("Object fetching the good way");
        for(int i = 0; i < PlainCubeManager.Singleton.mPlainCubes.Count; ++i)
        {
            PlainCubeManager.Singleton.mPlainCubes[i].DoSomething();
        }
        Profiler.EndSample();
    }
    else
    {
        Profiler.BeginSample("Object fetching the bad way");
        PlainCube[] objects = FindObjectsOfType<PlainCube>();
        for(int i = 0; i < objects.Length; ++i)
        {
            objects[i].DoSomething();
        }
        Profiler.EndSample();
    }
}
```

図 17. STATE を使用してオブジェクトを切り替える

Unity* 物理システムでの操作のベスト・プラクティス

Unity* でダイナミック・オブジェクトを操作する場合、良く知られている最適化と、避けるべきことがあります。オブジェクトを手動で移動する場合、または Unity* にオブジェクトの物理現象を制御させる場合は、オブジェクトに **Rigidbody** コンポーネントを追加して、オブジェクトが移動可能なことを Unity* 物理システムに知らせます。オブジェクトを手動で移動する場合は、**isKinematic** フラグをオンにします (図 18)。[Inspector (インスペクター)] タブの右上隅の **[Static (スタティック)]** チェックボックスがオフになっていることを確認します (図 19)。

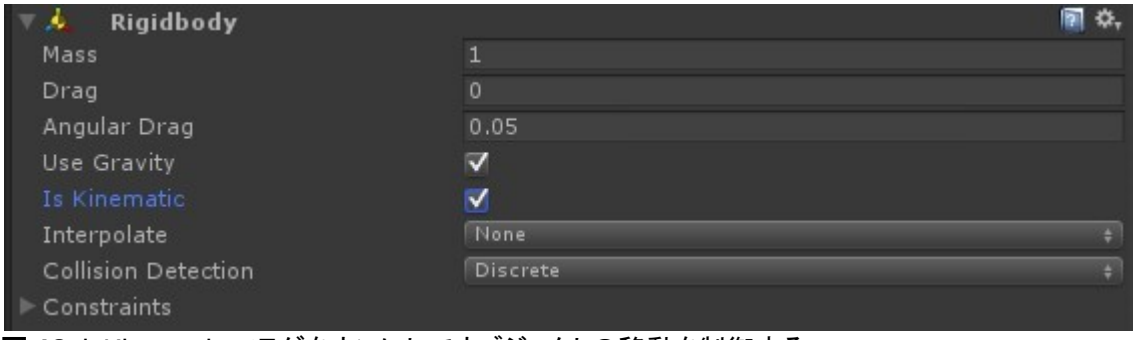

図 18. isKinematic フラグをオンにしてオブジェクトの移動を制御する

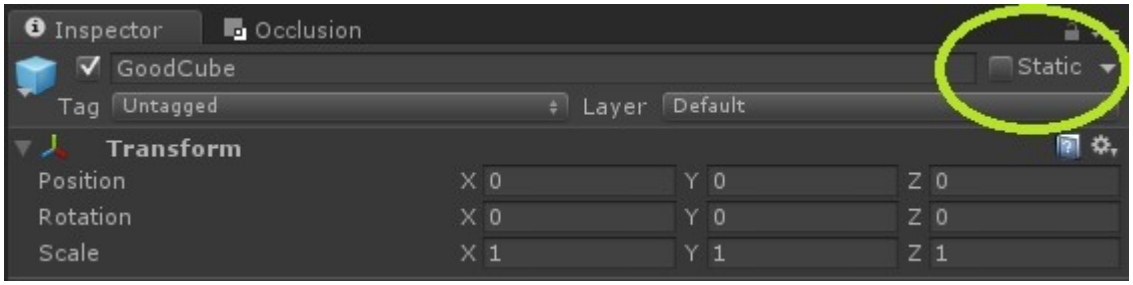

図 19. Static プロパティーをオフにしてダイナミック・オブジェクトをスタティック・セットから除外する

アプリケーションでダイナミック・オブジェクトを適切に制御するには、プロファイラーを開いて、**[CPU Profiler (CPU プロファイラー)]** の **[Physics (物理)]** サブセクションを選択し、物理時間ステップ (デフォルトでは 1 秒あたり 24 回更新) のフレームをハイライトして、オブジェクトの FixedUpdate() 呼び出しの概要ウィンドウに **Static Collider.Move (Expensive delayed cost)** エントリー (図 20) が表示されていないことを確認します。Static Collider.Move メッセージが表示されていない場合、このセクションの物理現象は適切に動作しています。

図 20. ダイナミック・オブジェクトを適切に管理していない場合、Static Collider.Move (Expensive delayed cost) が表示される

完全に透明なオブジェクトの無効化

「フェードアウト」レンダリング・モードでマテリアルを使用するオブジェクトや、完全に透明になるオブジェクトは、そのオブジェクトに **MeshRenderer** コンポーネントを設定し、オブジェクトが完全に透明になったら無効にすることが重要です。これらのオブジェクトは、アルファ値に関係なく、描画呼び出しでディスパッチされます。例えば、開発者は、4 分割画面を使用して、イベントがトリガーとなるパルスをダメージ・インジケーターまたはビネット効果に描画することがあります。エンジンはオブジェクトがいつ完全に透明になるか追跡しないため、注意しないとリソースが浪費されます。以下の 2 つのスクリーンショットは、前景の半透明オブジェクトのアルファ値を変更して同じシーンを撮影したものです。

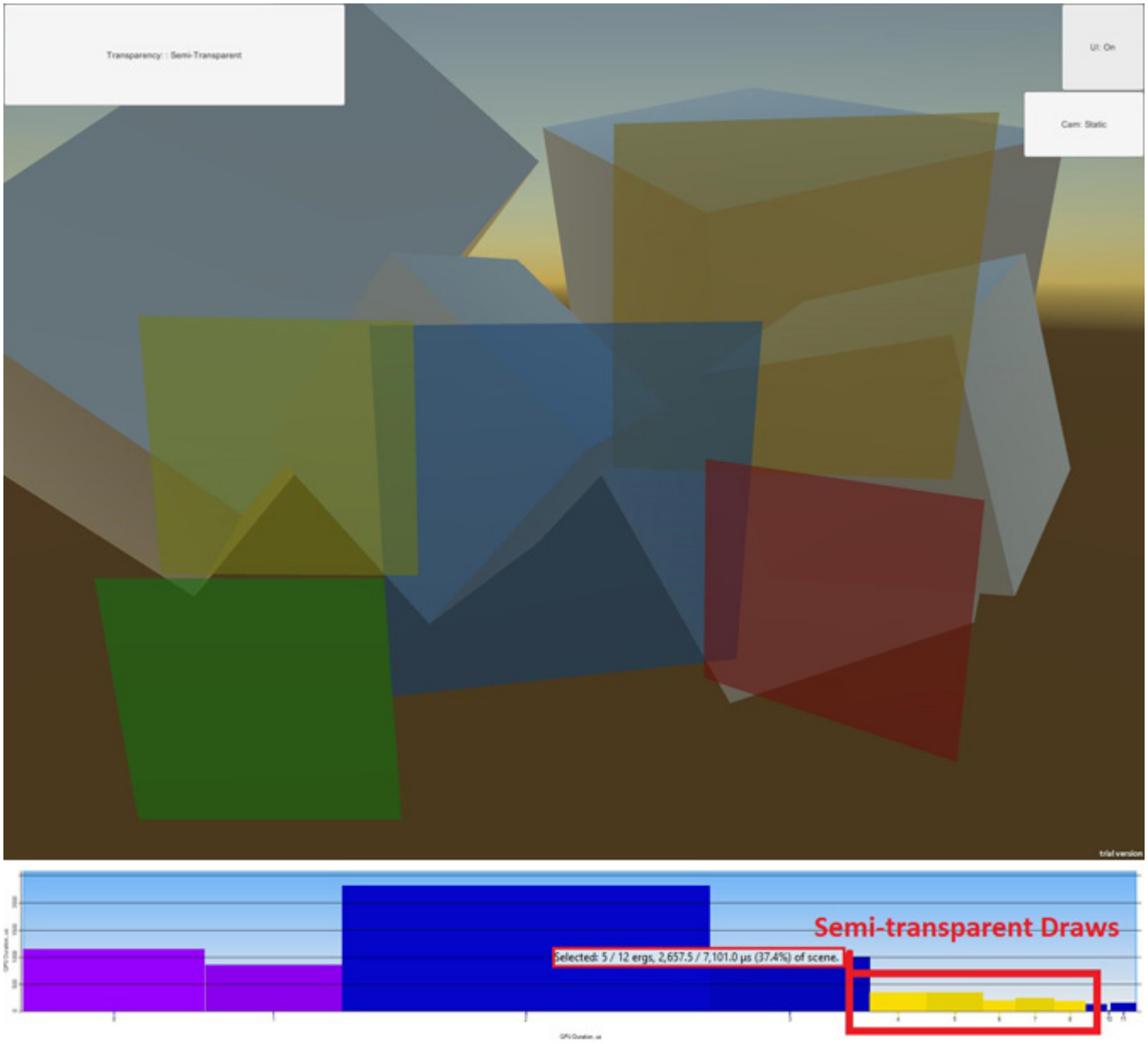

図 21. 5 つの半透明平面と対応する GPA フレーム・キャプチャー。半透明オブジェクトは可視で、2,657.5 µs のシーンの 37.4% を占めています。

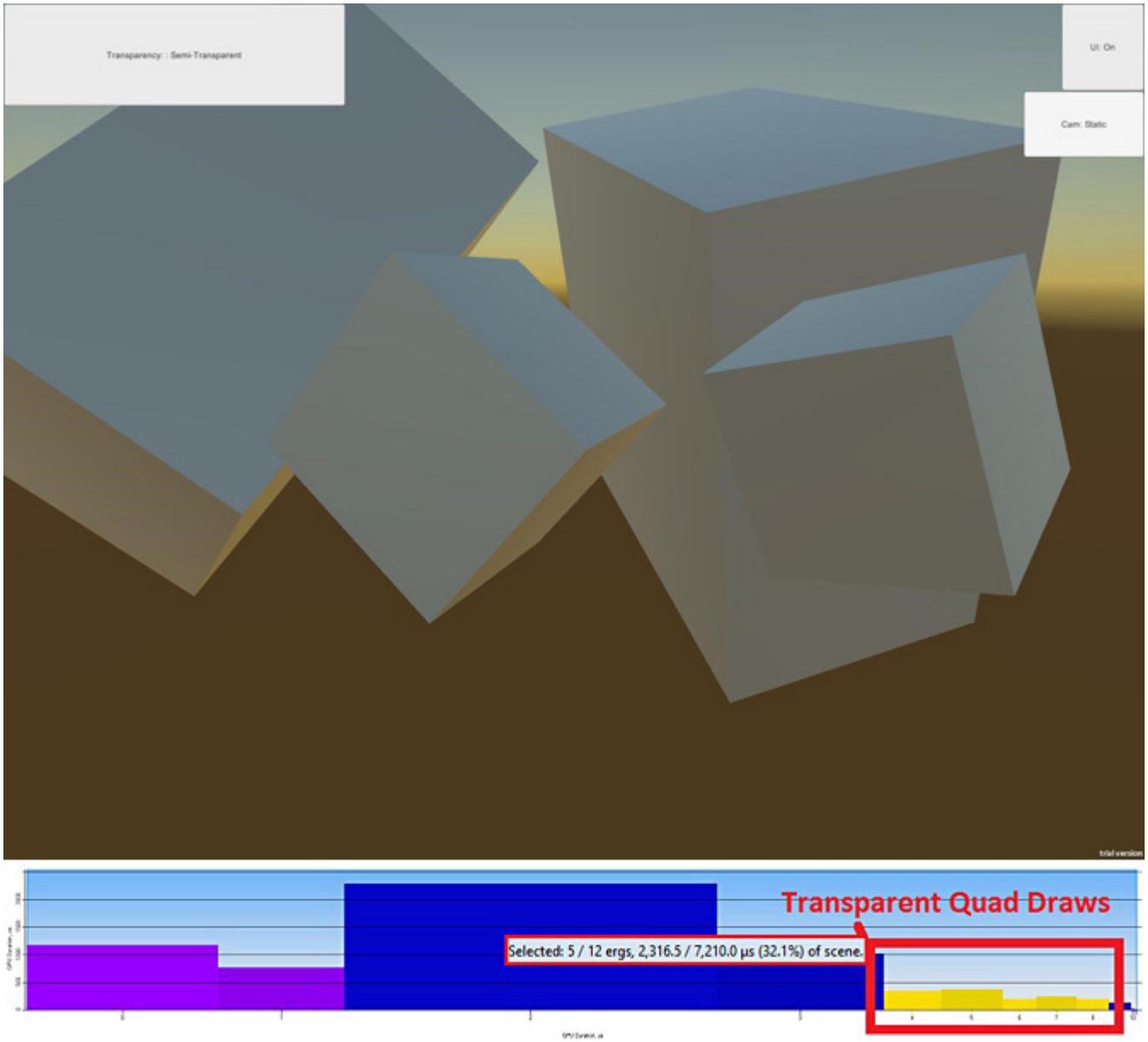

図 22. 図 21 の 5 つの平面についてマテリアルのアルファ値を 0 に設定する。対応するインテル® GPA のフレーム・キャプチャーは、まだ描画コマンドが GPU 上で行われていることを示しています。これらの描画は、2,316.5µs のシーンの 32.1% を占めています。

不要な描画呼び出しをディスパッチしないように、可能な場合は透明になる可能性があるオブジェクトに関連するアルファ値を常にチェックしてください。色に基づいてレンダリングする単純なマテリアルでは、次のようなコードを使用します。

```
void UpdateMaterialColor(Color newColor)
{
    // If fully transparent (alpha == 0), disable renderer
    if(newColor.a == 0.0f)
    {
        myMeshRenderer.enabled = false;
    }
    else
    {
        myMeshRenderer.enabled = true;
    }
    myMeshRenderer.material.color = newColor;
}
```

図 23. オブジェクトの透過処理を編集する場合、オブジェクトが可視かどうか確認し、必要に応じて無効にすることでリソースを節約する

チュートリアルのパート 3 に続く:
インテルの x86 プラットフォーム向け Unity* 最適化ガイド: パート 3 (英語)

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。