高品質かつ低負荷な3Dライブを実現するシェーダー開発

～『ラブライブ！スクールアイドルフェスティバル　ALL STARS』(スクスタ)の開発事例～

KLab株式会社 細田 翔

KLab株式会社

KLabGames事業本部

グラフィックスエンジニア / テクニカルアーティスト

細田 翔

2015年にKLab株式会社へ入社。

『ラブライブ！スクールアイドルフェスティバル ALL STARS』の3Dグラフィックス開発を担当。

3Dライブの演出機能の実装、描画のパフォーマンス最適化を行う。

シェーダーやレイマーチングによる映像制作やデモシーン制作が趣味。

『ラブライブ！スクールアイドルフェスティバル ALL STARS』（スクスタ）について


梨子
握手！？
ラブライブ！シリーズ最新作の
オールスターストーリー、スタート

開発情報

- Unity 2018.4　LTS
- Unity Timeline
  - プロジェクト専用のカスタムトラックが多数
- Post Processing Stack v1（軽量化して利用）
- Graphics API
  - Android: OpenGL ES 3.0
  - iOS: Metal
- Forward Rendering（built-in render pipeline）

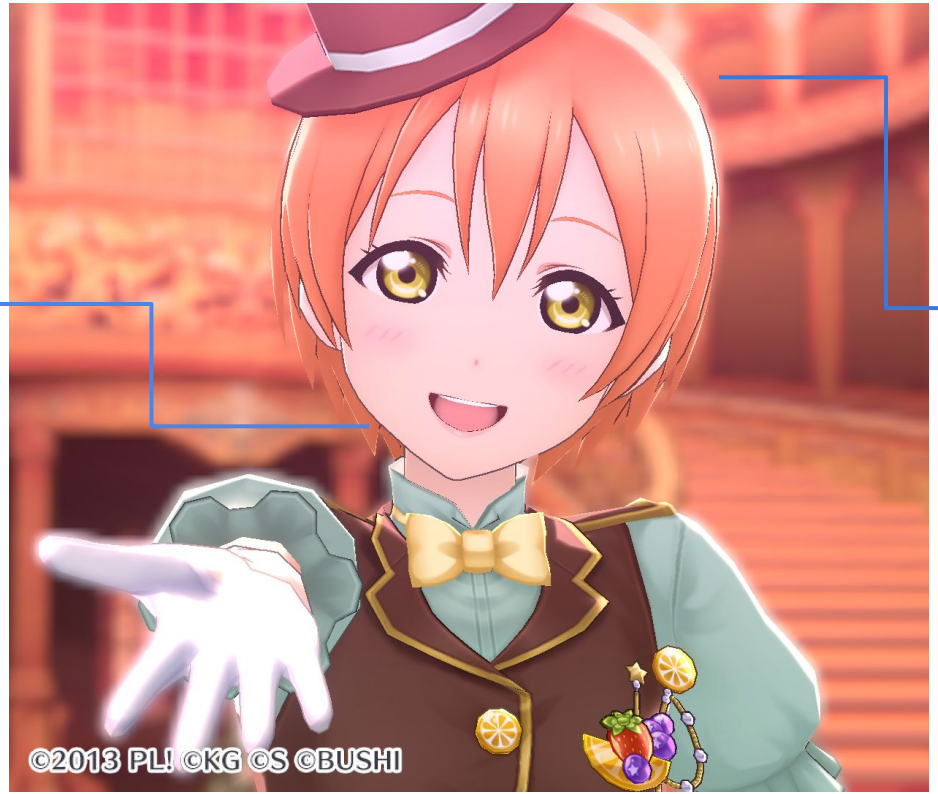
本日のアジェンダ

- **高品質**を実現するシェーダー
  - メンバー（キャラクター）用シェーダー
  - ステージ用シェーダー
- **低負荷**を実現するシェーダー
  - 負荷の計測
  - エフェクト汎用シェーダー
  - ドローコールとオーバードロー
  - エフェクト専用シェーダー

本日のアジェンダ

- **高品質**を実現するシェーダー
  - メンバー（キャラクター）用シェーダー
  - ステージ用シェーダー
- **低負荷**を実現するシェーダー
  - 負荷の計測
  - エフェクト汎用シェーダー
  - ドローコールとオーバードロー
  - エフェクト専用シェーダー

メンバー（キャラクター）シェーダー



アウトライン

背面ポリゴンを膨らませる「よくある」アウトラインの方式

こだわりポイント

- **カメラ距離とFoV**で
  アウトライン幅を自動補正
- インライン線への対応
- 頂点カラーによる微調整が可能

「カメラ距離とFoV」でアウトライン幅を自動補正

通常のアウトライン（補正なし）

アウトライン幅は、近づけば太くなるし、遠ざかれば細くなる

- 背面ポリゴンは普通の3Dモデルと同様に**遠近法**でスケールが変わる
- カメラからの距離やカメラのFoVの影響を受ける

「カメラ距離とFoV」でアウトライン幅を自動補正

通常のアウトライン（補正なし）

アウトライン幅は、近づけば太くなるし、遠ざかれば細くなる

- 背面ポリゴンは普通の3Dモデルと同様に**遠近法**でスケールが変わる
- カメラからの距離やカメラのFoVの影響を受ける

理想のアウトライン（補正あり）

どんなカメラの距離やFoVでも、
常にバランス良いアウトライン幅に補正される

「カメラ距離とFoV」でアウトライン幅を自動補正

	補正なし😔	補正あり😍
近景 望遠レンズ		
遠景 広角レンズ		

「カメラ距離とFoV」でアウトライン幅を自動補正

	補正なし😔	補正あり😍
近景 望遠レンズ		
遠景 広角レンズ		

「カメラ距離とFoV」でアウトライン幅を自動補正

アウトライン幅を**常に一定**に保つ方式

カメラからの距離とFoVの影響を打ち消す処理を実装

「カメラ距離とFoV」でアウトライン幅を自動補正

アウトライン幅を**常に一定**に保つ方式

カメラからの距離とFoVの影響を打ち消す処理を実装

遠景時のアウトライン幅が太すぎて
バランスが悪かった😭

「カメラ距離とFoV」でアウトライン幅を自動補正

アウトライン幅を**常に一定**に保つ方式

カメラからの距離とFoVの影響を打ち消す処理を実装

遠景時のアウトライン幅が太すぎて
バランスが悪かった😭

不採用🙅

「カメラ距離とFoV」でアウトライン幅を自動補正

近景と遠景で2段階の補正

近景の場合

- 常に一定の幅に補正

遠景の場合

- **補正率**(0～1)で補正の強度を調整
  - **補正率0**:補正なし(通常の遠近法と同じ)
  - **補正率1**:常に一定の幅に補正

パラメータは制作と相談して決定

- 近景と遠景の **境界値**
- 遠景時の **補正率**

「カメラ距離とFoV」でアウトライン幅を自動補正

近景と遠景で**2段階の補正**

理想のアウトラインが完成！

近景の場合
- 常に一定の幅に補正

遠景の場合
- **補正率**(0～1)で補正の強度を調整
  - **補正率0**:補正なし(通常の遠近法と同じ)
  - **補正率1**:常に一定の幅に補正

パラメータは制作と相談して決定
- 近景と遠景の **境界値**
- 遠景時の **補正率**

「カメラ距離とFoV」でアウトライン幅を自動補正

近景と遠景で**2段階の補正** ▶ **理想のアウトライン**が完成！ ▶ **採用**🤦

近景の場合
- 常に一定の幅に補正

遠景の場合
- **補正率**(0〜1)で補正の強度を調整
  - **補正率0**:補正なし(通常の遠近法と同じ)
  - **補正率1**:常に一定の幅に補正

パラメータは制作と相談して決定
- 近景と遠景の **境界値**
- 遠景時の **補正率**

インライン線

背面ポリゴンがない**インライン**にも線を出す機能

フラグメントシェーダーで実装（頂点カラーの **B** 成分を2値化）

テクスチャへの線の描き込みとの違い

- アウトライン線と同じロジックで色を計算できる👍
- テクスチャ解像度による**ジャギーが発生しない**👍

アウトラインの品質向上

頂点カラーで微調整が可能

R: **アウトライン幅**の重みづけ
0に近づくほど細くなる

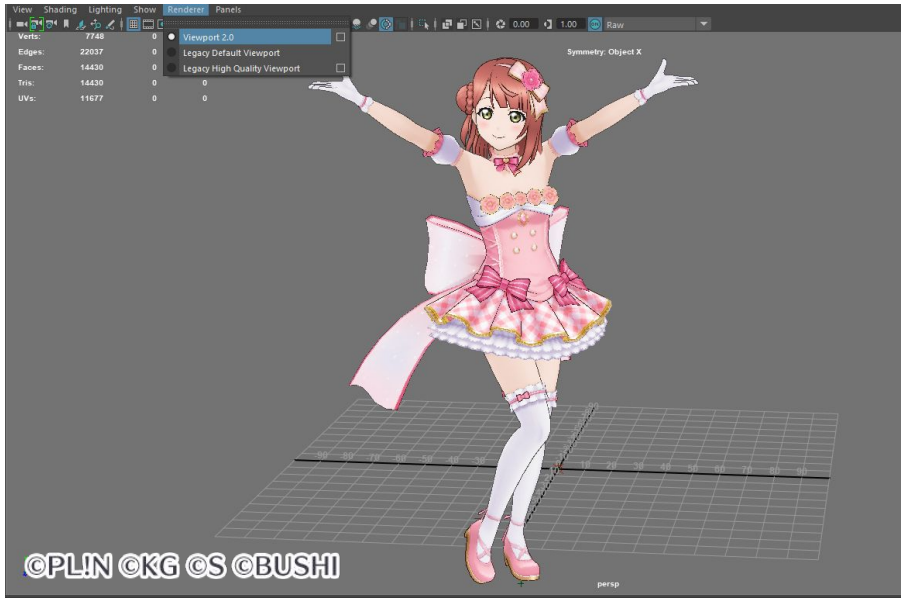
G: アウトラインの**押し込み距離**の重みづけ
不要な線をカメラ奥に押し込んで見えなくする
0に近づくほど奥に行く

B: **インライン線**の幅の重み付け
0に近づくほど細くなる

線を出したくないときは
RGBをそれぞれ0に

アウトラインをMaya上で確認

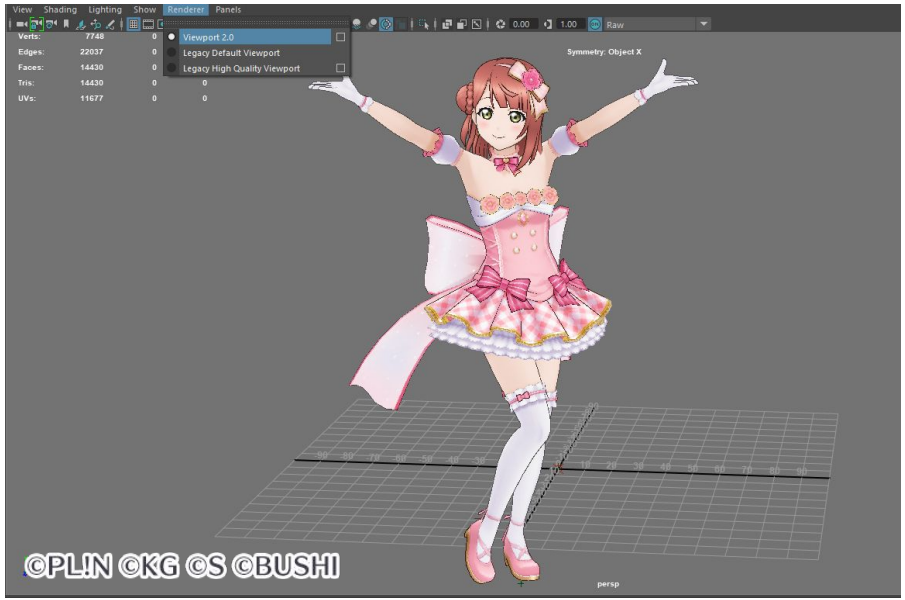
Maya用のCgFxシェーダーにも移植



Mayaのビューポート上で確認ができる！

アウトラインをMaya上で確認

Maya用のCgFxシェーダーにも移植

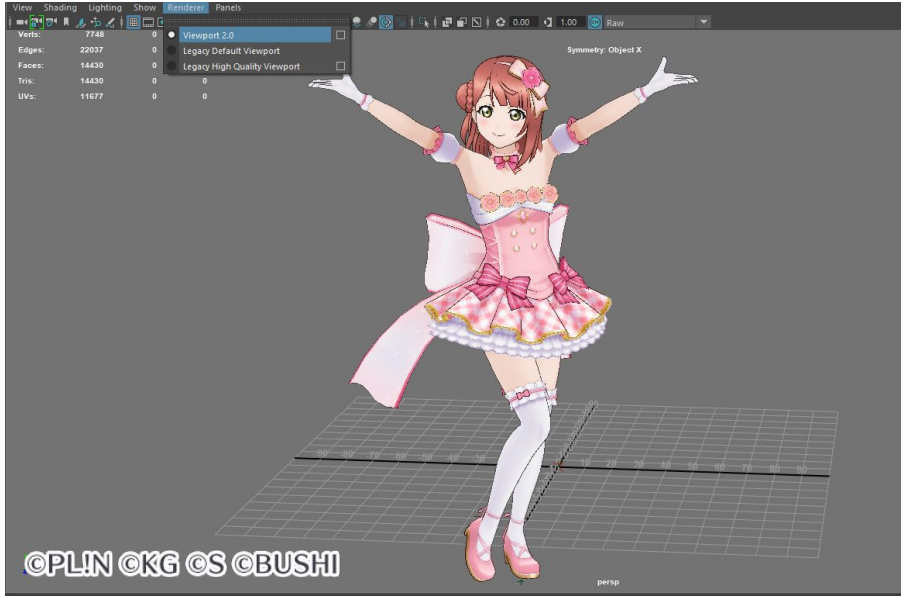


Mayaのビューポート上で確認ができる！

Unityへエクスポート不要に
作業イテレーションを高速化

アウトラインをMaya上で確認

Maya用のCgFxシェーダーにも移植



Mayaのビューポート上で確認ができる！

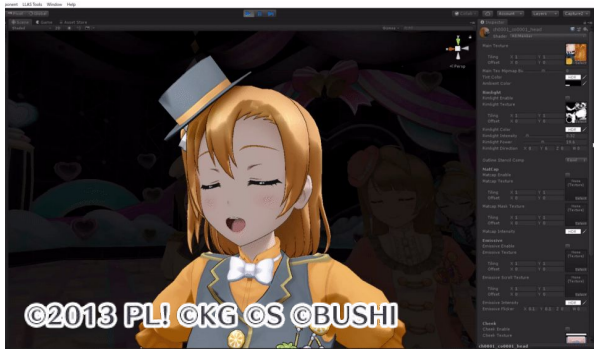
Unityへエクスポート不要に
作業イテレーションを高速化

品質向上😍

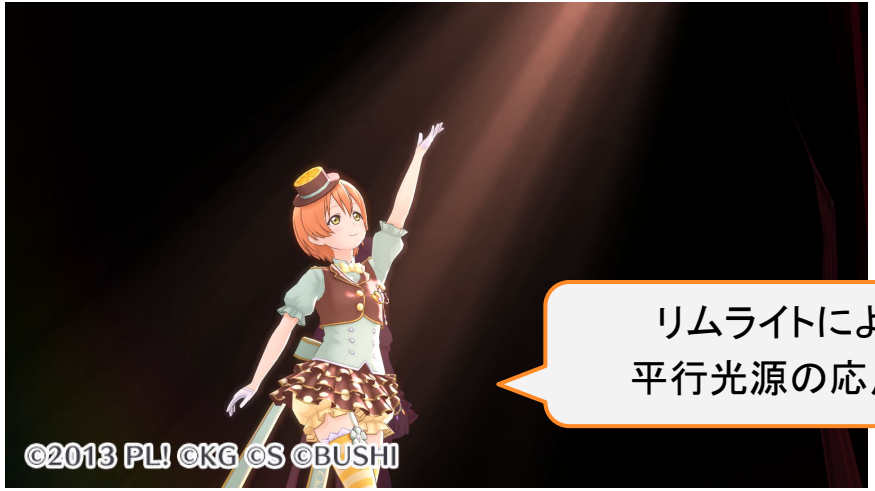
リムライト

カメラ空間での平行光源の方向をパラメータに

- 方向をZ軸（デフォルト）にすれば、**逆光表現**のリムライト
- 方向を設定すれば、**平行光源のライティング**



光の方向を調整できる



リムライトによる
平行光源の応用例

リムライトとアウトラインの組み合わせ

通常のアウトライン
リムライト部分にもアウトラインが表示されてしまう

アウトラインの部分キャンセル ON
リムライト部分のアウトラインを消す
ステンシルで実装

本日のアジェンダ

- **高品質**を実現するシェーダー
  - メンバー（キャラクター）用シェーダー
  - ステージ用シェーダー
- **低負荷**を実現するシェーダー
  - 負荷の計測
  - エフェクト汎用シェーダー
  - ドローコールとオーバードロー
  - エフェクト専用シェーダー

ステージシェーダー

ライトマップ
による**静的**GI

リアリティのある質感

Parallax-corrected Cubemap
による**静的**な鏡面反射

ツルツルとした質感

ステージシェーダー

ライトマップ
による**静的**GI

Parallax-corrected Cubemap
による**静的**な鏡面反射

静的 = **事前計算**の結果を**テクスチャにベイク**して、ランタイム時の負荷を削減

Parallax-corrected Cubemapとは

通常のCubemap

カメラ位置が考慮されないため、
反射の位置が不自然😔

カメラが移動すると反射の映り込みに違和感がある

Parallax-corrected Cubemap

カメラ位置を考慮するため、
自然な反射😍

カメラが移動しても反射の映り込みが破綻しない

Parallax-corrected Cubemapとは

通常のCubemap

カメラ位置が考慮されないため、
反射の位置が不自然😔

カメラが移動すると反射の映り込みに違和感がある

Parallax-corrected Cubemap

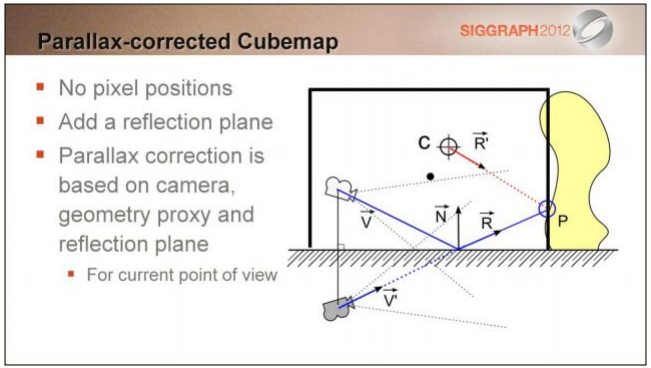
カメラ位置を考慮するため、
自然な反射😍

カメラが移動しても反射の映り込みが破綻しない

Parallax-corrected Cubemapの実装

Cubemapサンプル用の**反射ベクトルを補正**する手法

視差補正なし: 反射ベクトル R の計算
- 視線ベクトル V と 法線 N だけで計算



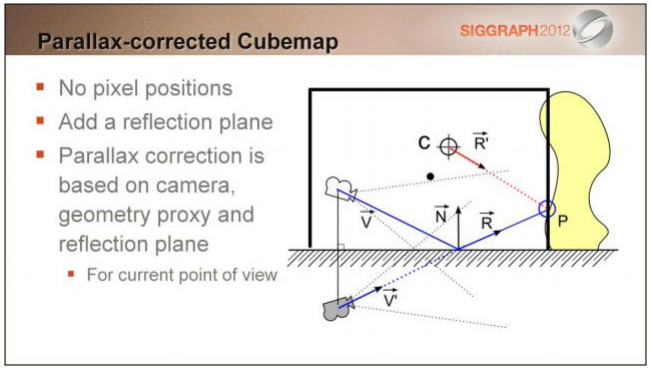
Sébastien Lagarde (2012).
“Local Image-based Lighting With Parallax-corrected Cubemap”,
SIGGRAPH 2012 Talk
https://seblagarde.wordpress.com/2012/11/28/siggraph-2012-talk/

Parallax-corrected Cubemapの実装

Cubemapサンプル用の**反射ベクトルを補正**する手法

視差補正なし: 反射ベクトル **R** の計算
- 視線ベクトル **V** と 法線 **N** だけで計算

視差補正あり: Parallax-corrected Cubemapの **R'**
- 追加のパラメーター
  - 部屋の壁の **バウンディングボックス**
  - Cubemap（Reflection Probe）座標 **C**



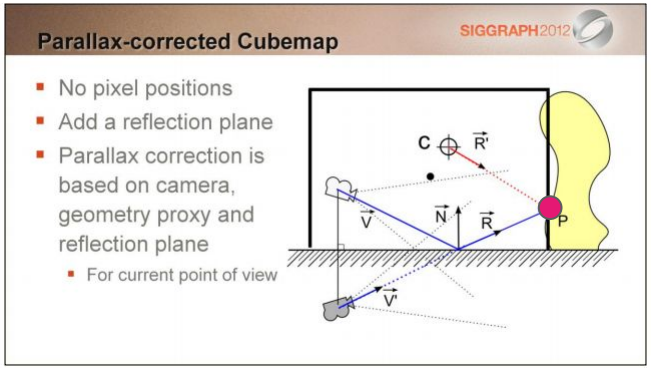
Sébastien Lagarde (2012).
"Local Image-based Lighting With Parallax-corrected Cubemap",
SIGGRAPH 2012 Talk
https://seblagarde.wordpress.com/2012/11/28/siggraph-2012-talk/

Parallax-corrected Cubemapの実装

Cubemapサンプル用の**反射ベクトルを補正**する手法

視差補正なし: 反射ベクトル **R** の計算
- 視線ベクトル **V** と 法線 **N** だけで計算

視差補正あり: Parallax-corrected Cubemapの **R'**
- 追加のパラメーター
  - 部屋の壁の **バウンディングボックス**
  - Cubemap（Reflection Probe）座標 **C**
- **バウンディングボックス** と **R** の交差点 **P** を求める



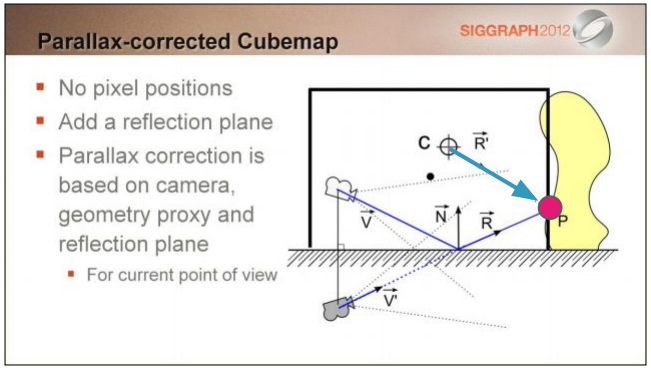
Sébastien Lagarde (2012).
"Local Image-based Lighting With Parallax-corrected Cubemap",
SIGGRAPH 2012 Talk
https://seblagarde.wordpress.com/2012/11/28/siggraph-2012-talk/

Parallax-corrected Cubemapの実装

Cubemapサンプル用の**反射ベクトルを補正**する手法

視差補正なし: 反射ベクトル **R** の計算
- 視線ベクトル **V** と 法線 **N** だけで計算

視差補正あり: Parallax-corrected Cubemapの **R'**
- 追加のパラメーター
  - 部屋の壁の **バウンディングボックス**
  - Cubemap（Reflection Probe）座標 **C**
- **バウンディングボックス** と **R** の交差点 **P** を求める
- **C** から **P** に向かうベクトルが **R'** になる



Sébastien Lagarde (2012).
"Local Image-based Lighting With Parallax-corrected Cubemap",
SIGGRAPH 2012 Talk
https://seblagarde.wordpress.com/2012/11/28/siggraph-2012-talk/

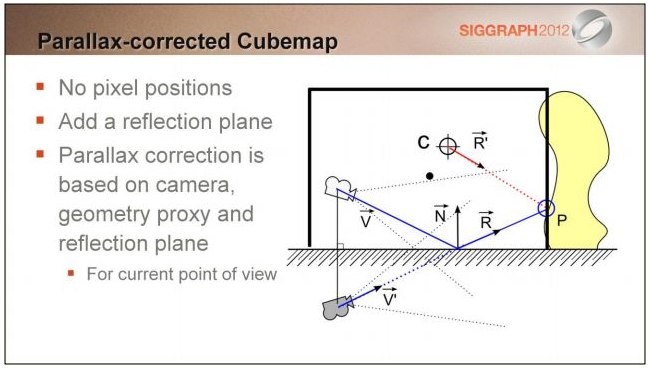
Parallax-corrected Cubemapの実装例

Unity用のHLSLによる実装例

反射ベクトル **R**

```
half3 calcParallaxCorrectedCubemapReflect(float3 worldPos, half3 worldRefl)
{
    float3 firstPlaneIntersect  = (_BoxMax - worldPos) / worldRefl;
    float3 secondPlaneIntersect = (_BoxMin - worldPos) / worldRefl;
    float3 furthestPlane = max(firstPlaneIntersect, secondPlaneIntersect);
    float dist = min(min(furthestPlane.x, furthestPlane.y), furthestPlane.z);
    float3 worldIntersectPos = worldPos + worldRefl * dist;

    return normalize(worldIntersectPos - _CubemapPos);
}
```



Sébastien Lagarde (2012)

Parallax-corrected Cubemapの実装例

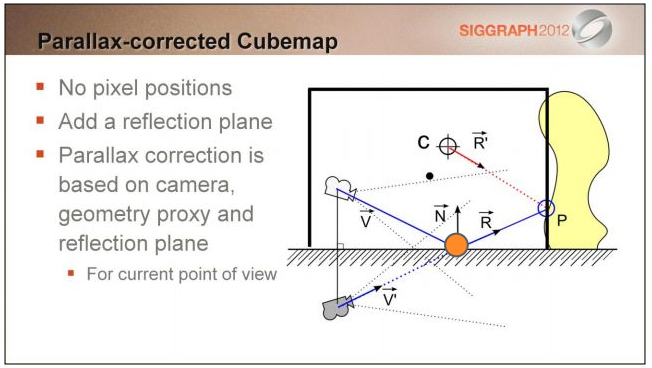
Unity用のHLSLによる実装例

表面の座標

反射ベクトル **R**

```
half3 calcParallaxCorrectedCubemapReflect(float3 worldPos, half3 worldRefl)
{
    float3 firstPlaneIntersect  = (_BoxMax - worldPos) / worldRefl;
    float3 secondPlaneIntersect = (_BoxMin - worldPos) / worldRefl;
    float3 furthestPlane = max(firstPlaneIntersect, secondPlaneIntersect);
    float dist = min(min(furthestPlane.x, furthestPlane.y), furthestPlane.z);
    float3 worldIntersectPos = worldPos + worldRefl * dist;

    return normalize(worldIntersectPos - _CubemapPos);
}
```



Sébastien Lagarde (2012)

Parallax-corrected Cubemapの実装例

Unity用のHLSLによる実装例

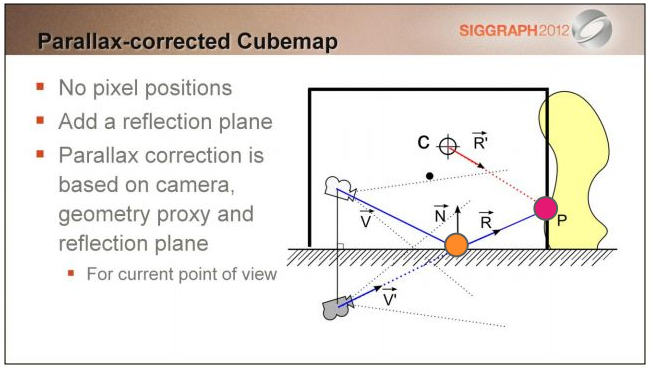
表面の座標

反射ベクトル **R**

```
half3 calcParallaxCorrectedCubemapReflect(float3 worldPos, half3 worldRefl)
{
    float3 firstPlaneIntersect  = (_BoxMax - worldPos) / worldRefl;
    float3 secondPlaneIntersect = (_BoxMin - worldPos) / worldRefl;
    float3 furthestPlane = max(firstPlaneIntersect, secondPlaneIntersect);
    float dist = min(min(furthestPlane.x, furthestPlane.y), furthestPlane.z);
    float3 worldIntersectPos = worldPos + worldRefl * dist;

    return normalize(worldIntersectPos - _CubemapPos);
}
```

Rとバウンディングボックス の
交差点 **P** を計算する処理
AABBに限定すれば簡単に計算可能



Sébastien Lagarde (2012)

Parallax-corrected Cubemapの実装例

Unity用のHLSLによる実装例

表面の座標

反射ベクトル **R**

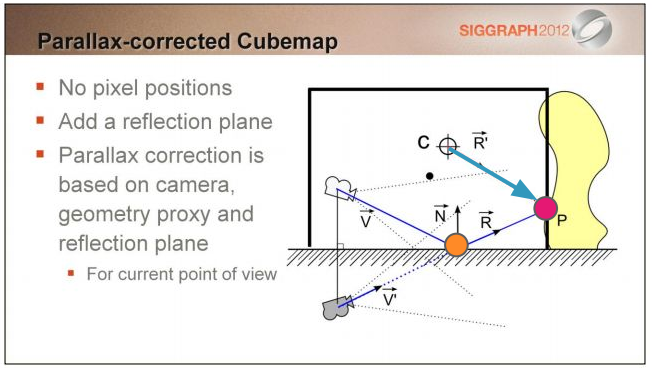
```
half3 calcParallaxCorrectedCubemapReflect(float3 worldPos, half3 worldRefl)
{
  float3 firstPlaneIntersect  = (_BoxMax - worldPos) / worldRefl;
  float3 secondPlaneIntersect = (_BoxMin - worldPos) / worldRefl;
  float3 furthestPlane = max(firstPlaneIntersect, secondPlaneIntersect);
  float dist = min(min(furthestPlane.x, furthestPlane.y), furthestPlane.z);
  float3 worldIntersectPos = worldPos + worldRefl * dist;

  return normalize(worldIntersectPos - _CubemapPos);
}
```

Rとバウンディングボックス の
交差点 **P** を計算する処理
AABBに限定すれば簡単に計算可能

最後に Cubemap **C** から交差点 **P**
に向かうベクトルを計算

これが補正された反射ベクトル **R'**



Sébastien Lagarde (2012)

本日のアジェンダ

- **高品質**を実現するシェーダー
  - メンバー（キャラクター）用シェーダー
  - ステージ用シェーダー
- **低負荷**を実現するシェーダー
  - 負荷の計測
  - エフェクト汎用シェーダー
  - ドローコールとオーバードロー
  - エフェクト専用シェーダー

最適化には負荷計測が最重要

- 憶測による最適化は**厳禁**🙅（逆効果もありえる）
- **実機上の負荷計測**と**ボトルネックの特定**が**重要**🙆

最適化には負荷計測が最重要

- 憶測による最適化は**厳禁**🙅（逆効果もありえる）
- **実機上の負荷計測**と**ボトルネックの特定**が**重要**🤦

それは知っているけど…

最適化には負荷計測が最重要

- 憶測による最適化は**厳禁**🙅（逆効果もありえる）
- **実機上の負荷計測**と**ボトルネックの特定**が**重要**🤦

それは知っているけど…

忙しくて計測する時間がない😭

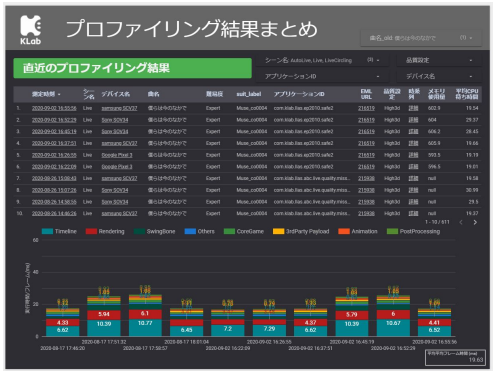
CI/CDと連携した負荷の自動計測

自動プロファイリングシステムで解決！

- アプリのビルドと連携し、**完全放置**で実機プロファイリング
  - 自動でアプリを実機インストール
  - 自動でアプリが起動
  - 自動でライブ周回
  - 自動でプロファイリング開始と終了
  - 自動でプロファイリング結果を集計

CI/CDと連携した負荷の自動計測

プロファイリング結果はダッシュボードに**自動集計**



計測サマリーの一覧

アプリバージョン×
端末×品質設定×楽曲で
検索結果を絞り込み

負荷の詳細

重たいメソッドTOP
負荷のカテゴリー分類

時系列の分析

フレーム単位の負荷
スパイクの検出に役立つ

CI/CDと連携した負荷の自動計測

- 利用技術
  - Unity Profiler Reader
  - Google BigQuery
  - Google データポータル

Unity Profilerで見れる**全ての情報**を取得

- CEDEC 2019 / GDC 2019で詳細を発表！資料公開あり

“Android向けUnity製ゲーム最適化のための
CI/CDと連携した自動プロファイリングシステム”, CEDEC 2019
https://www.slideshare.net/klab-tech/androidunitycicd

“Continuous Profiling for Android Game Performance Optimization”, GDC 2019
https://www.slideshare.net/klab-tech/continuous-profiling-for-android-game-performance-optimization-216466184

リリース9ヶ月前

リリース9ヶ月前

Xperia XZ,『未来の僕らは知ってるよ』上級,「3D高」設定

リリース9ヶ月前

Xperia XZ,『未来の僕らは知ってるよ』上級,「3D高」設定

1フレームあたりの**CPU**のメインスレッドの処理時間の平均:

リリース9ヶ月前

Xperia XZ,『未来の僕らは知ってるよ』上級,「3D高」設定

1フレームあたりの**CPU**のメインスレッドの処理時間の平均: **103.11 ms**

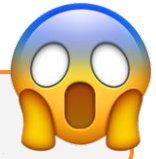
9.7 FPS！

ギャー

負荷の遷移

- Xperia XZ,『未来の僕らは知ってるよ』上級,「3D高」設定
- この端末では、30FPSを目標

自動計測の導入時期

	リリース9ヶ月前	リリース5ヶ月前	リリース時
1フレーム全体の CPU処理の平均値	103.11 ms 9.7 FPS 	39.64 ms 25 FPS 	19.34 ms 57 FPS！ 
レンダリングの CPU処理の平均値	–	12.24 ms	4.84 ms

負荷対策の背景

- 初期リリース向けのデータは既に完成していた

品質はそのままに
負荷を下げる必要があった

負荷対策の背景

- 初期リリース向けのデータは既に完成していた

品質はそのままに
負荷を下げる必要があった

演出用の**エフェクトシェーダー**の
創意工夫で解決！

本日のアジェンダ

- **高品質**を実現するシェーダー
  - メンバー（キャラクター）用シェーダー
  - ステージ用シェーダー
- **低負荷**を実現するシェーダー
  - 負荷の計測
  - エフェクト汎用シェーダー
  - ドローコールとオーバードロー
  - エフェクト専用シェーダー

エフェクト汎用シェーダー

色んなエフェクトに利用できる
シンプルな汎用シェーダー

床投影ライト

スポットライト

ディスプレイ

エフェクト汎用シェーダー

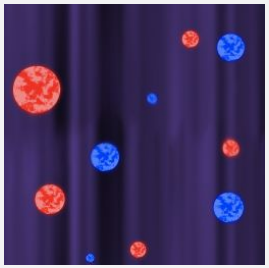
プロパティとして設定可能な項目

- **ブレンドモード**（アルファブレンド or 加算 or 乗算）
- **カリングモード**（両面描画 or 片面描画）
- **深度書き込み**のON/OFF
- **深度テスト**のモード（通常の描画 or 常に最前面で描画）
- **ステンシル**の操作
- **RenderQueue**（描画順）

レンダリングに関する様々な設定

エフェクト汎用シェーダー

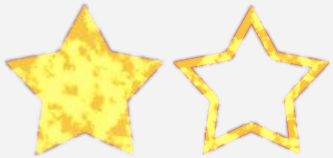
ディスプレイは3層のレイヤー構造（3層とも汎用シェーダーを利用）



①背景

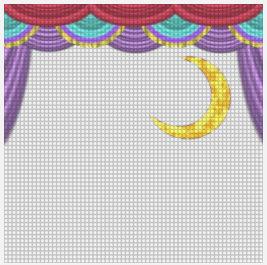
不透明で描画
ステンシル値を書き込む

UVスクロール



②演出

ステンシルテストで
①の内部の領域内だけ
描画する



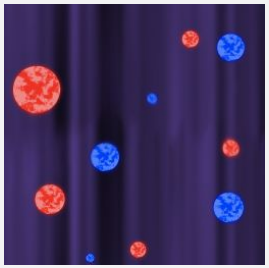
③ドット模様

アルファブレンドで描画

ディスプレイ完成！

エフェクト汎用シェーダー

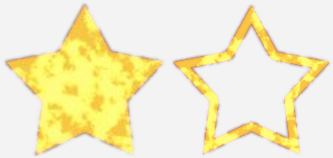
ディスプレイは3層のレイヤー構造（3層とも汎用シェーダーを利用）



①背景

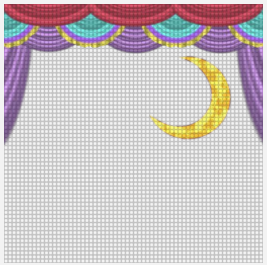
不透明で描画
ステンシル値を書き込む

UVスクロール



②演出

ステンシルテストで
①の内部の領域内だけ
描画する



③ドット模様

アルファブレンドで描画

ディスプレイ完成！

エフェクト汎用シェーダー

メリット😊

- 単一のシェーダーで**幅広い用途に対応可能**
- 開発側のシェーダー修正なしに
  制作側で幅広い表現ができる
- 複数のシェーダーを1つに統合して**管理コストの削減**

エフェクト汎用シェーダー

課題

プロパティが多すぎて設定に手間がかかる😩

エフェクト汎用シェーダー

課題

プロパティが多すぎて設定に手間がかかる😩

解決策

マテリアルの命名規則からプロパティを**自動設定**😎

AssetGraphを利用

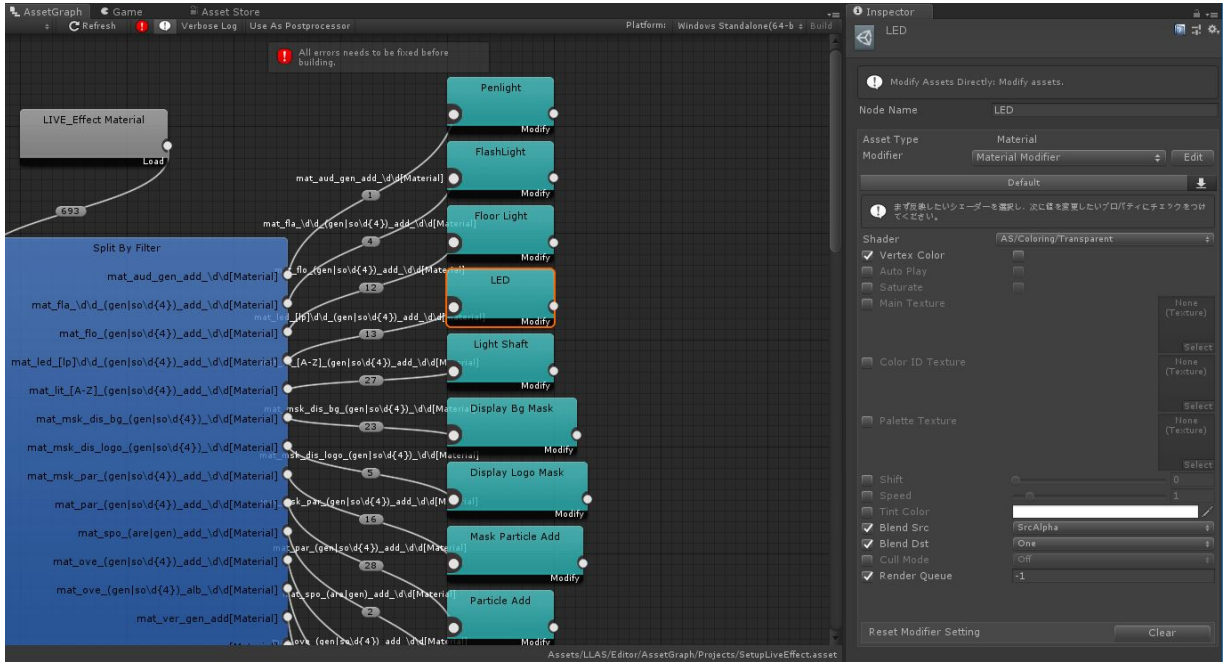
AssetGraphとは

- **Unity公式ツール**
  - https://github.com/Unity-Technologies/AssetGraph

- **ノードベース**でアセットの操作を定義
  - 複数の用途に利用できる（ノードの組み方次第）
  - スクスタでは**AssetPostprocessor**として利用

AssetGraphのマテリアル用ノード

標準では無かったので、**マテリアル設定用のカスタムノード**を実装

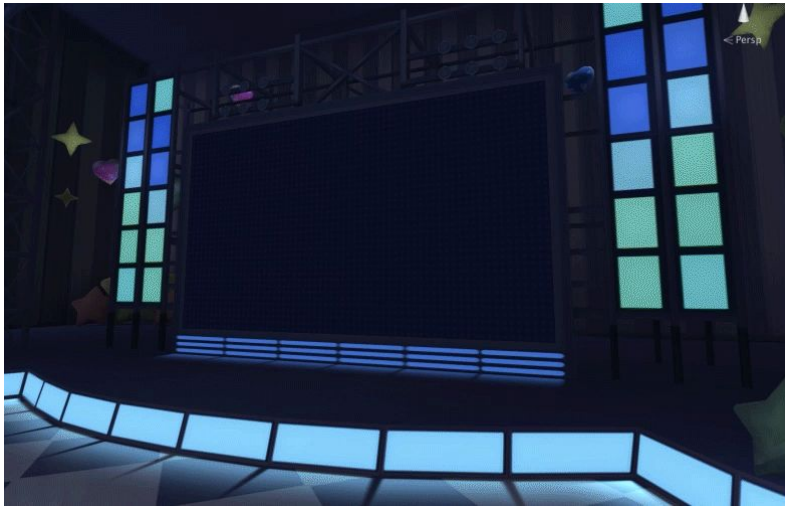


ノードベースで
マテリアル設定の
ルールを定義

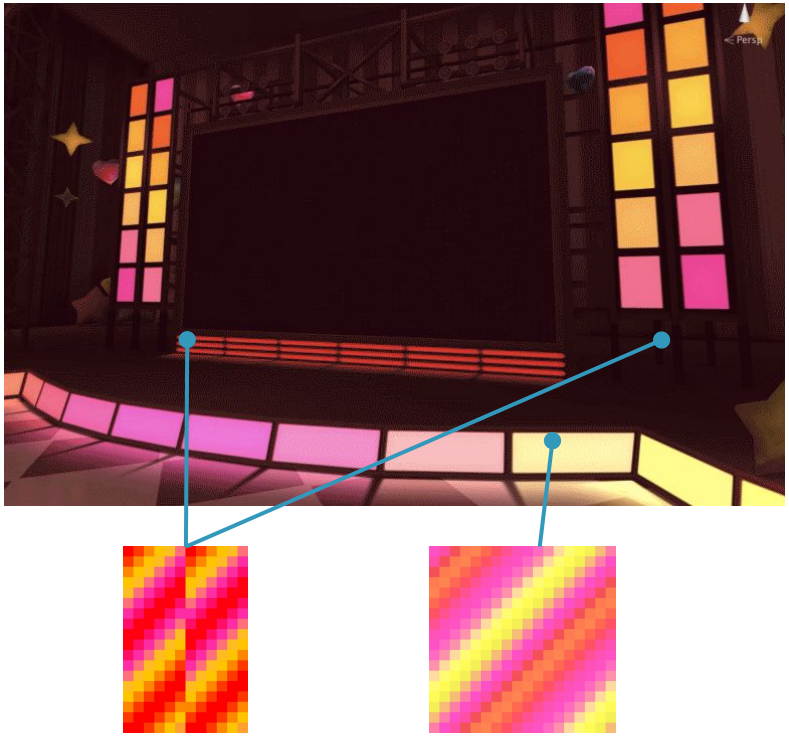
非エンジニアでも
管理できた

カラーリングシェーダー

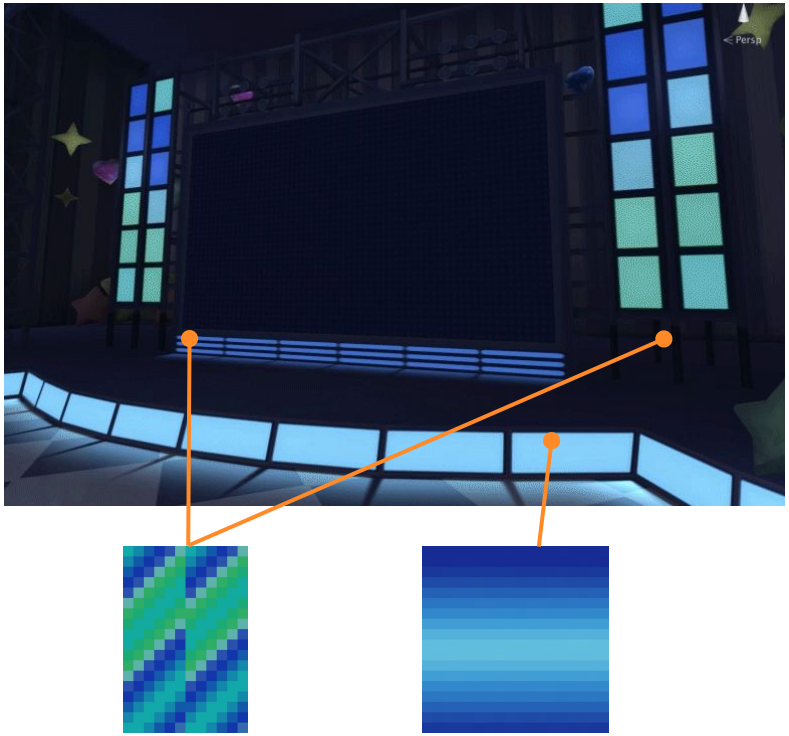
カラーリングシェーダー



カラーリングシェーダー



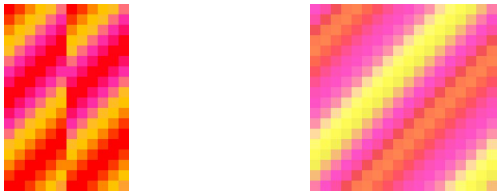
カラーパレット暖色系

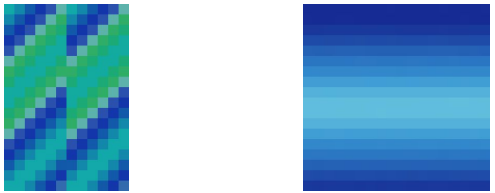


カラーパレット寒色系

カラーリングシェーダー

カラーパレット（小さなテクスチャ）
で点滅パターンを制御



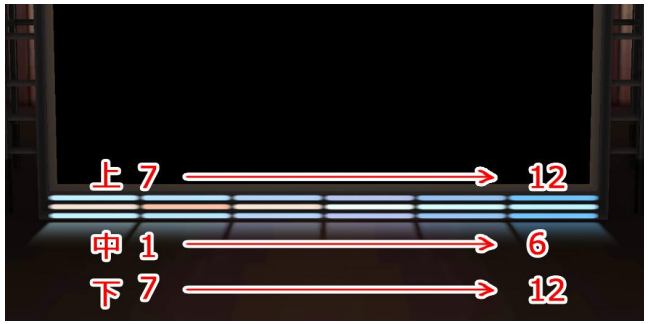


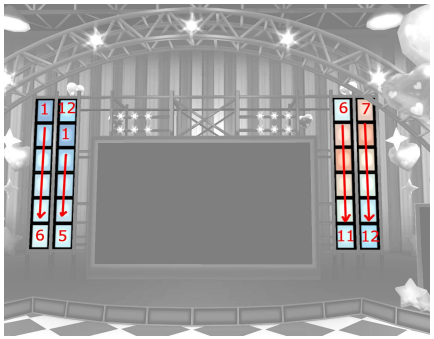
カラーパレット**暖色系**

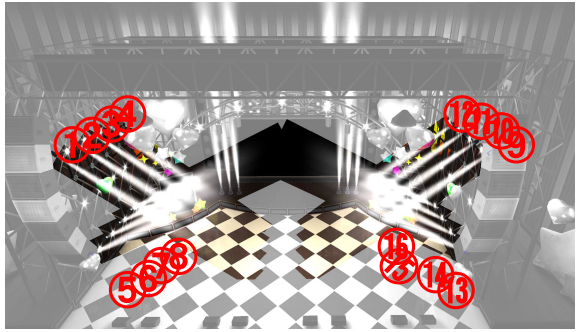
カラーパレット**寒色系**

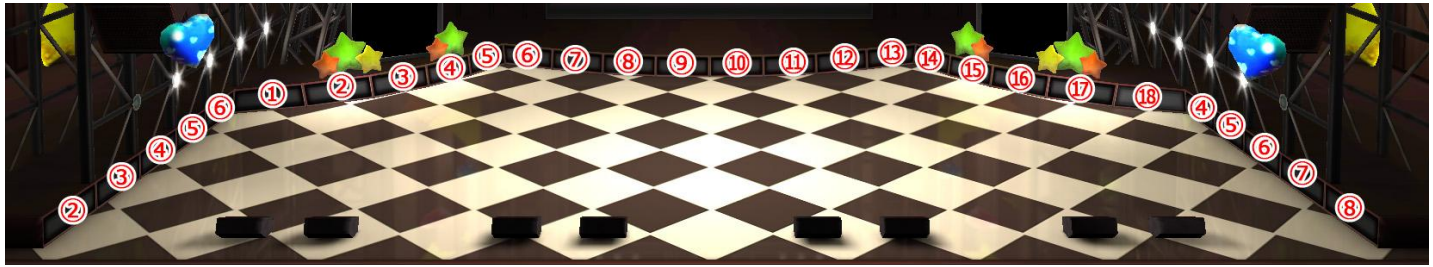
カラーリングシェーダー

LEDパネルのステージのカラーIDの例



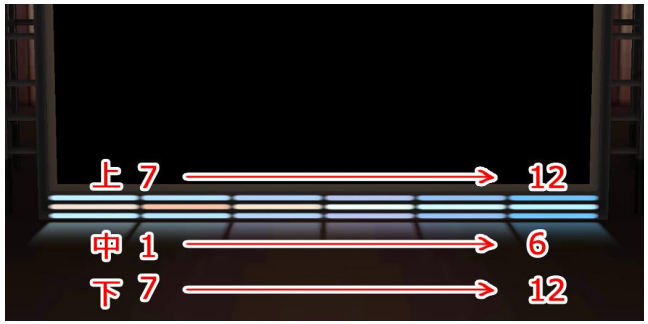


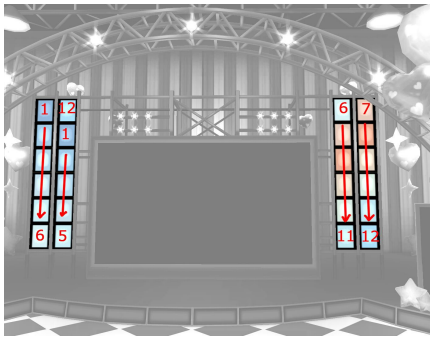


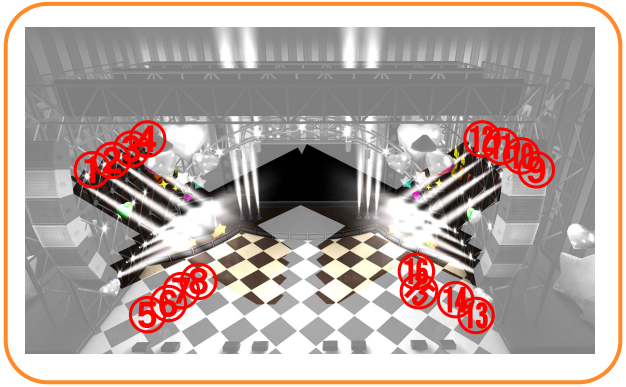


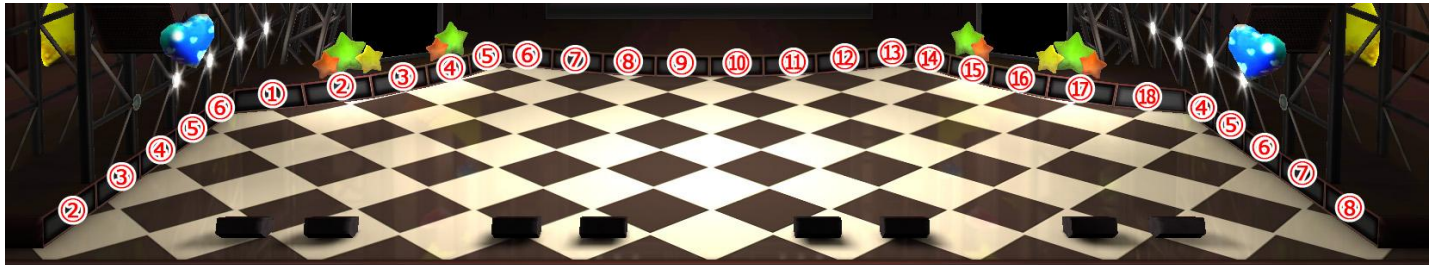
カラーリングシェーダー

LEDパネルのステージのカラーIDの例





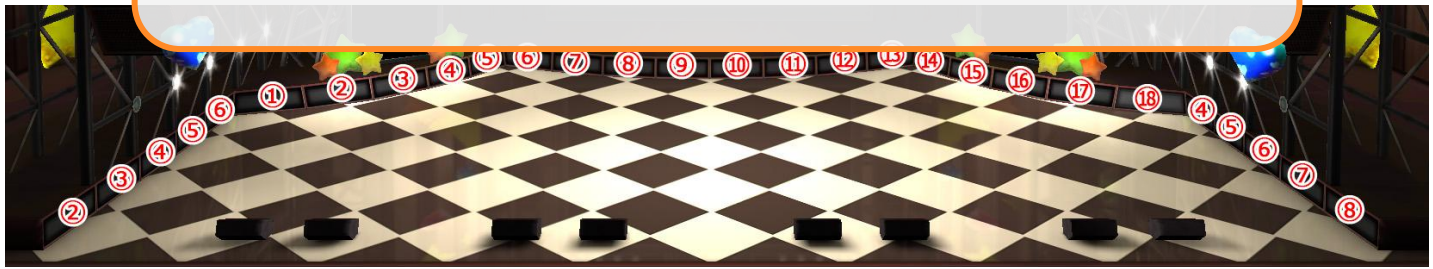




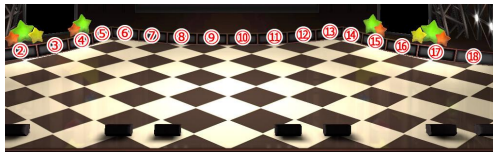
カラーリングシェーダー

LEDパネルのステージのカラーIDの例

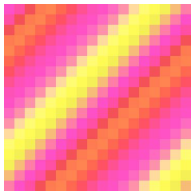
UV2のテクスチャの
濃淡で**256階調のID**を設定可能



カラーリングシェーダー



カラーID



カラーパレット

シェーダーで合成

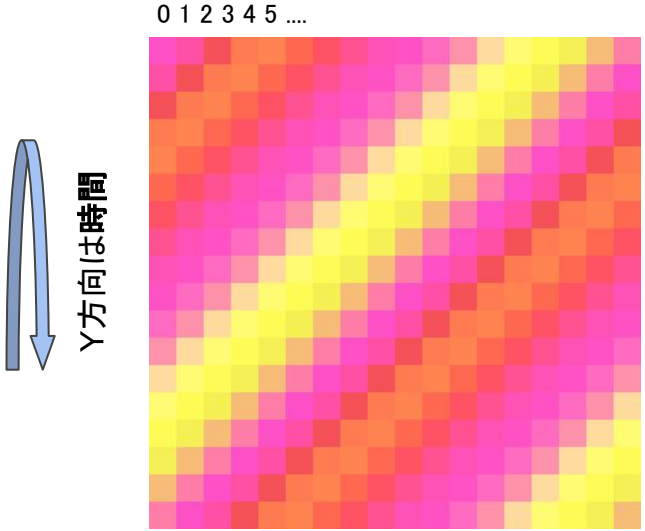
合成結果のアニメーション

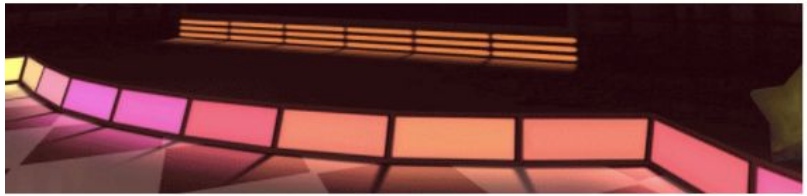
カラーリングシェーダー

カラーパレットの詳細

X方向は**カラーID**

0 1 2 3 4 5 ....





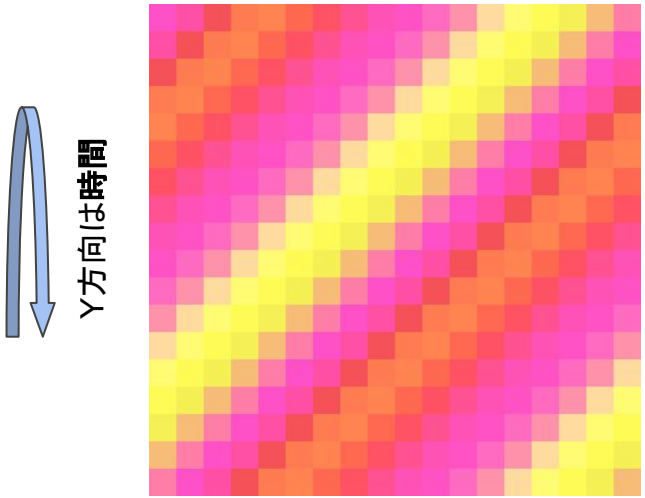
合成結果のアニメーション

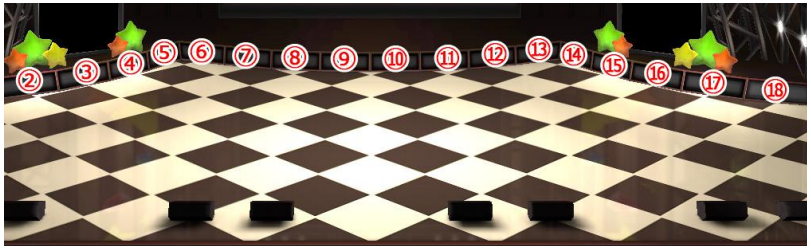
カラーリングシェーダー

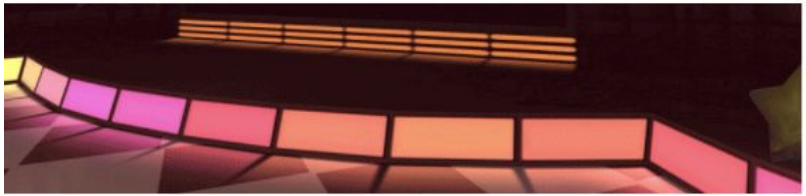
カラーパレットの詳細

X方向は**カラーID**

0 1 2 3 4 5 ....





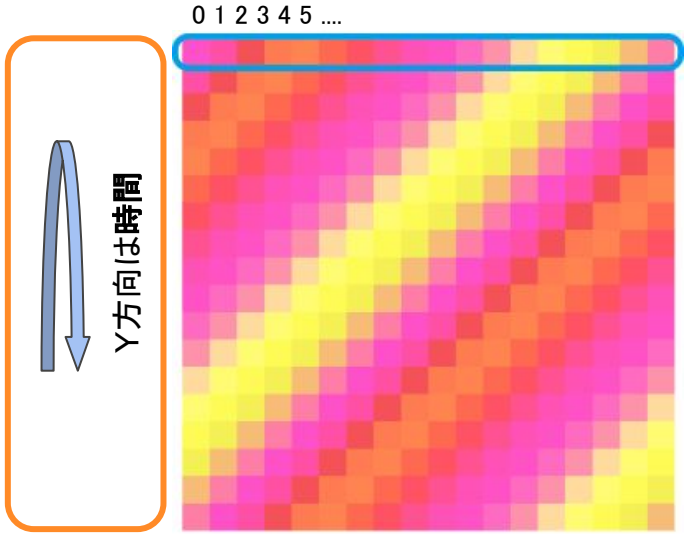


合成結果のアニメーション

カラーリングシェーダー

カラーパレットの詳細

X方向は**カラーID**



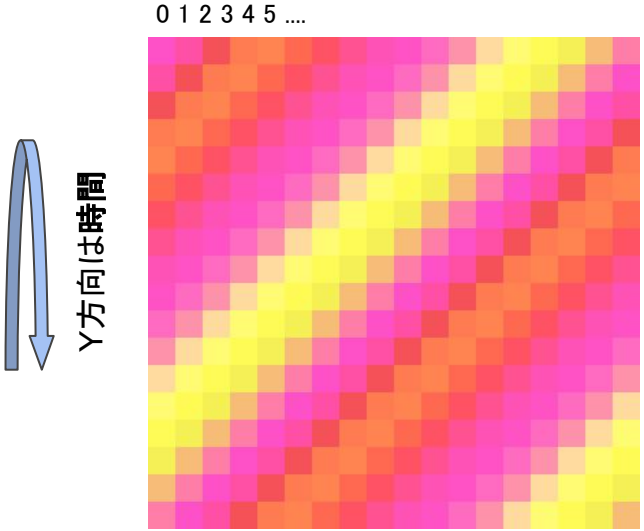
合成結果のアニメーション

カラーリングシェーダー

カラーパレットの詳細

X方向は**カラーID**

0 1 2 3 4 5 ....



- **シェーダーの実装**
  - カラーIDと時間に応じて
    カラーパレットのサンプリング用UVを計算

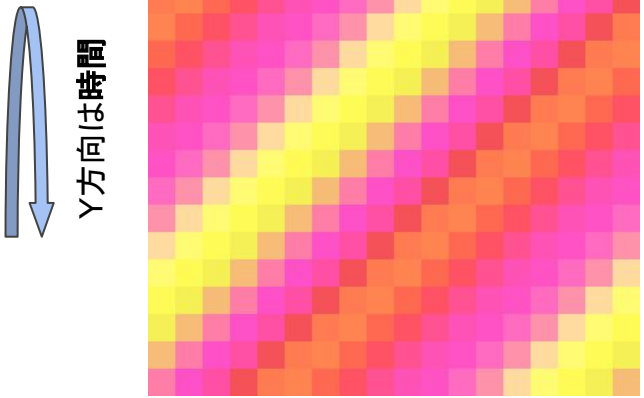
合成結果のアニメーション

カラーリングシェーダー

カラーパレットの詳細

X方向は**カラーID**

0 1 2 3 4 5 ....

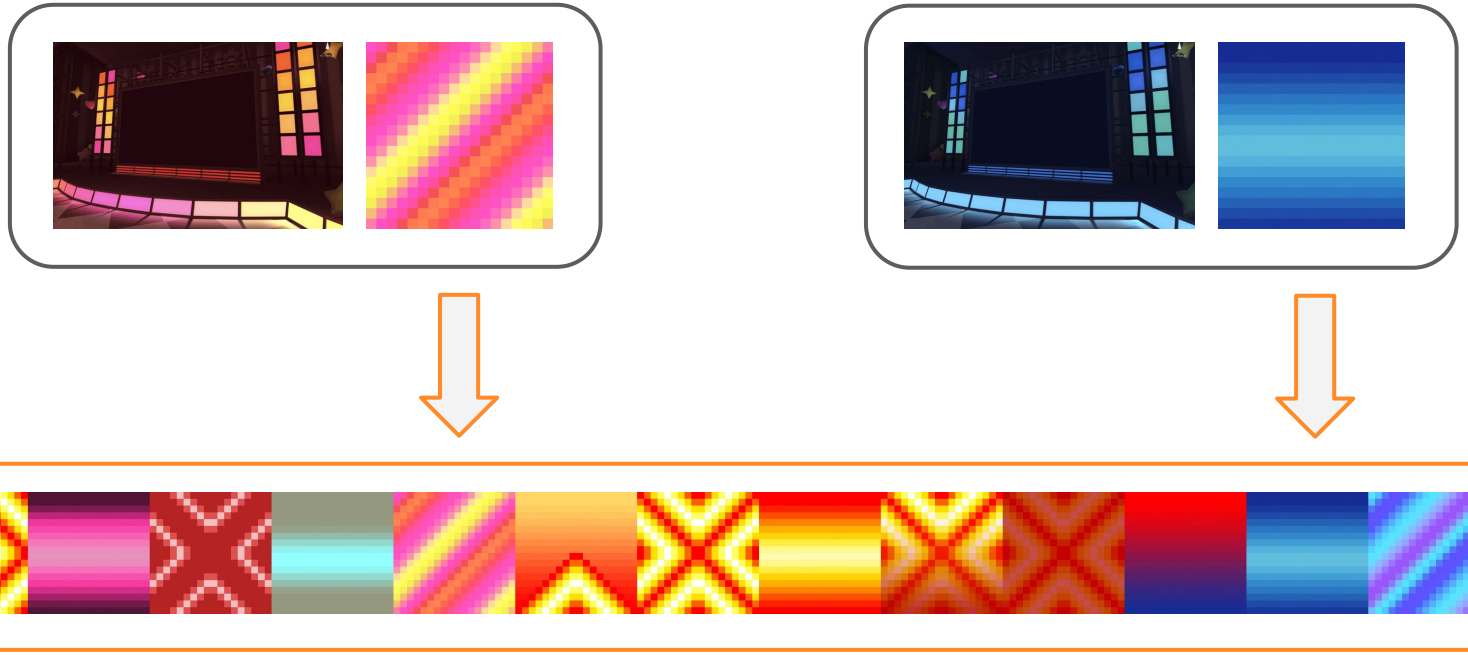


- 時間方向のループ
  - テクスチャ設定の**Repeat**
- 色のグラデーションによる滑らかな変化
  - テクスチャ設定の**BilinearFilter**

シェーダーでは特殊な処理を実装せずに
テクスチャの設定だけで実現した

合成結果のアニメーション

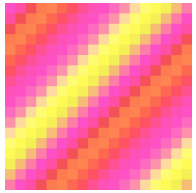
カラーリングシェーダー



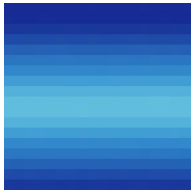
複数のカラーパレットを1枚のテクスチャにパッキング

カラーリングシェーダー

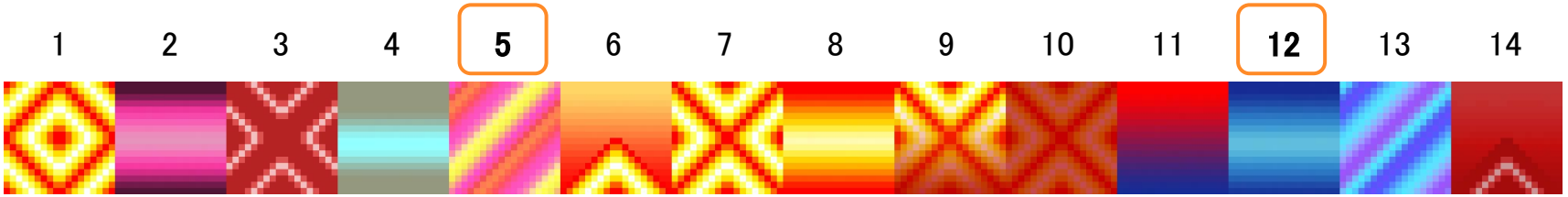
演出パターンの切り替え = **何番目のカラーパレット**を使うかの指定



左から**5番目**の
カラーパレット



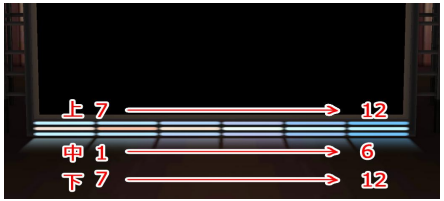
左から**12番目**の
カラーパレット



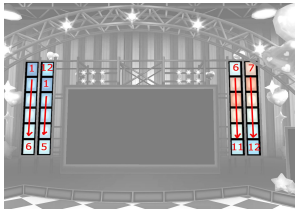
カラーリングシェーダー

- マテリアルの数の**合計は4**
- 色分け可能なIDの数は**合計58**（12 + 12 + 16 + 18）

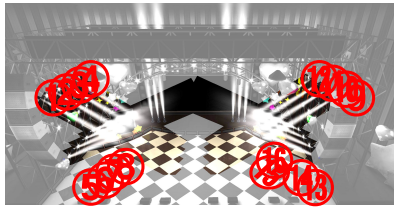
12

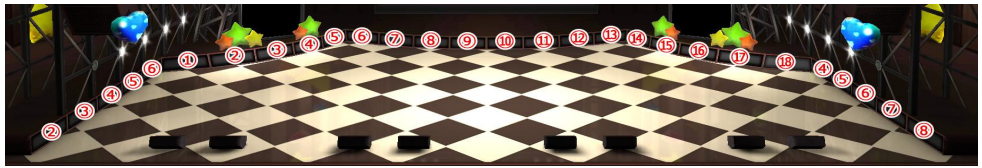


12



16





18

カラーリングシェーダー

演出パターン制御用のタイムラインの**トラックの数**に注目すると

カラーリングシェーダーが無かったら

トラックの数は**58個**（色分け可能なIDの数）　→ **入力工数が膨大**😵

カラーリングシェーダー

演出パターン制御用のタイムラインの**トラックの数**に注目すると

カラーリングシェーダーが無かったら

トラックの数は**58個**（色分け可能なIDの数）　→ **入力工数が膨大**😩

カラーリングシェーダーを使うと

トラックの数は**4個**（マテリアルの数）　→ **入力工数を削減**😉

カラーリングシェーダーまとめ

カラーパレットという小さなテクスチャで
演出パターンを制御する**特殊な汎用シェーダー**

古き良き**パレットアニメーション**
をソフトウェア的に再現

メリット

- **カラーパレット**だけ変更すれば、**演出パターンを量産可能**
  - 同じステージ素材を複数楽曲で再利用できる
- タイムラインのトラック数を減らして、演出の**入力工数を削減**
- **描画負荷の削減**（ドローコール数の削減）
  - このあと、詳しく説明！

本日のアジェンダ

- **高品質**を実現するシェーダー
  - メンバー（キャラクター）用シェーダー
  - ステージ用シェーダー
- **低負荷**を実現するシェーダー
  - 負荷の計測
  - エフェクト汎用シェーダー
  - ドローコールとオーバードロー
  - エフェクト専用シェーダー

ドローコールとオーバードロー

- **ドローコール数**
  - Meshを描画した回数
  - 描画のCPU負荷の指標になる

自動計測で確認可能

StatsのBatchesがドローコール数

- **オーバードロー**
  - 重ね塗りをした回数
    - 1ピクセルあたりの描画回数
  - 描画のGPU負荷の指標になる
  - 半透明描画で増加しやすい

SceneViewのOverdraw表示

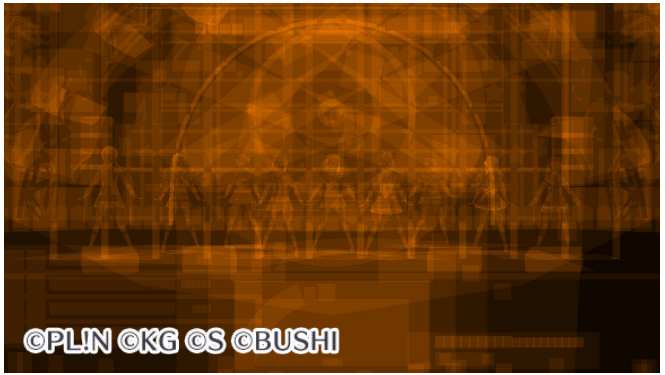
オーバードロー確認用シェーダー

GameViewで使えるオーバードロー表示

- **Replaced Shader**で実装
- 描画モードを考慮した正確な数値になる
  - Unity標準のSceneViewのオーバードロー表示では、
    不透明マテリアルでも深度テストがされず、オーバードローが実際より過剰になる

実際の**カメラワーク**や
ゲームUIを反映できる

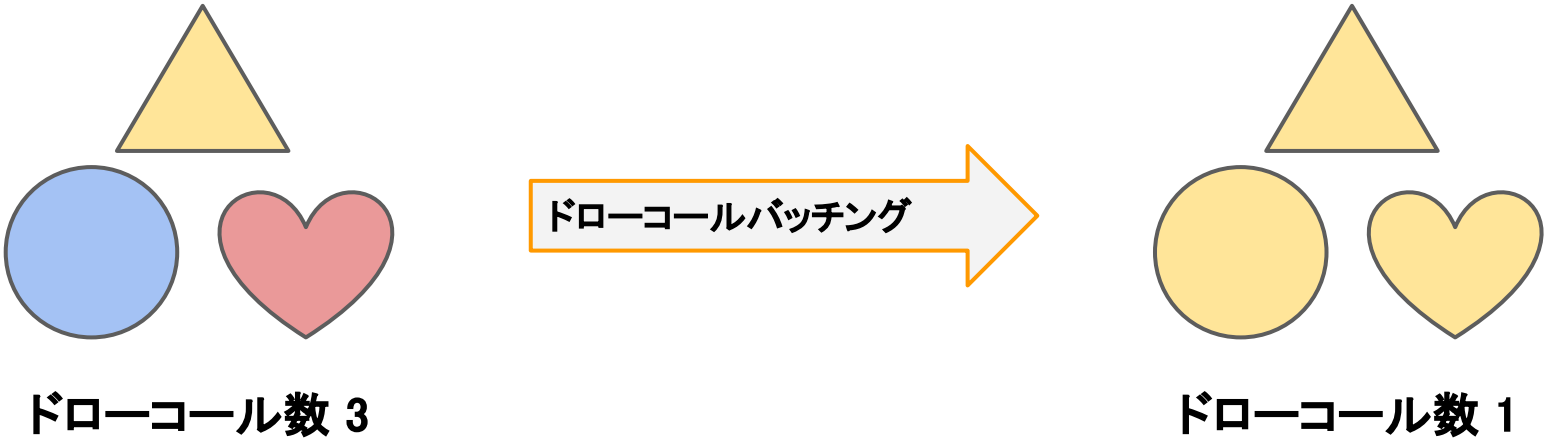
通常表示



オーバードロー表示

ドローコールバッチングとは？

- 複数のMeshを**まとめて描画**するUnityによる最適化
- ドローコールバッチングの条件
  - 同じマテリアルを共有
  - トータル頂点数が900以下（厳密には頂点情報の数やUnityバージョンによって変わる）



ドローコールバッチングしたい

- 同じマテリアルを共有した状態で**色分け**する方法
  - 頂点カラー
  - テクスチャのUV配置
  - **カラーリングシェーダー**

カラー指定だけでなく、
個別に**色のアニメーション**も可能！

描画順の整理によるドローコール削減

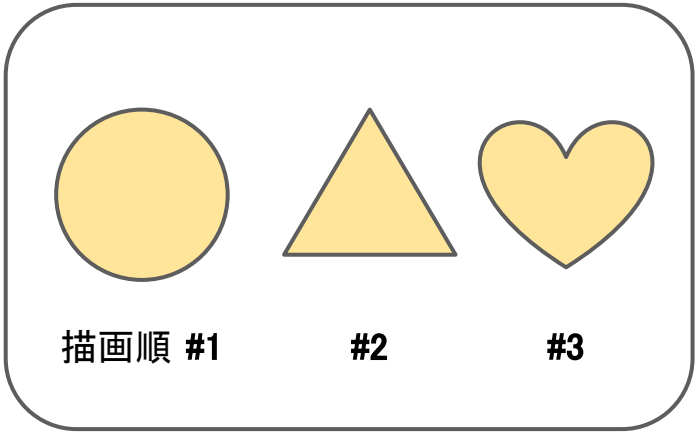
描画順も大事！

同じマテリアルが連続で描画されたときだけ
ドローコールバッチングができる

描画順の整理によるドローコール削減

同じマテリアルが連続して描画される場合

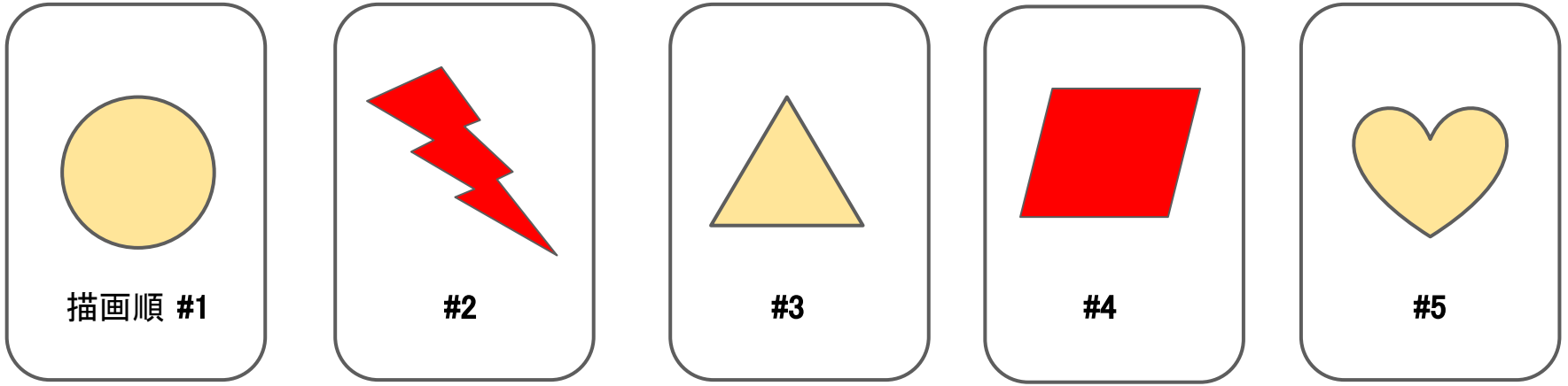
合計ドローコール数 **2**



描画順の整理によるドローコール削減

間に別のマテリアルが挟まって描画された場合

合計ドローコール数 **5**



描画順の整理によるドローコール削減

同じマテリアルを連続して描画したい！

描画順の整理によるドローコール削減

同じマテリアルを連続して描画したい！

RenderQueueのルールを決めて、
同じマテリアルを連続して描画すれば解決

描画順の整理によるドローコール削減

マテリアルの種類ごとにRenderQueueを細分化

不透明

- 1900 メンバー
- 2000 ステージ・ステージ小物・不透明パーティクル（不透明のデフォルト）

半透明

- 2450 草や木などの TransparentCutout のオブジェクト
- 2800 メンバーの影
- 2900 ディスプレイ用パーティクル
- 3000 LEDパネル・フラッシュライト・床投影ライト（半透明のデフォルト）
- 3010 半透明パーティクル
- 3020 ライトシャフト

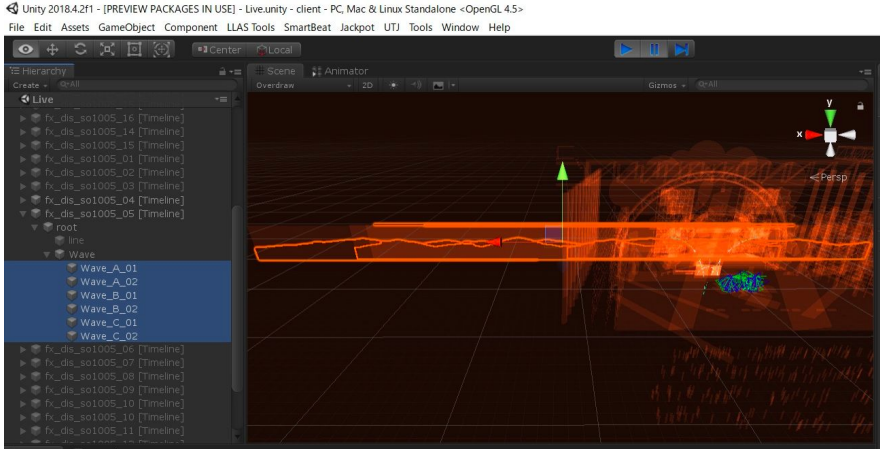
大量のマテリアルをAssetGraphで一括で**自動設定**😎

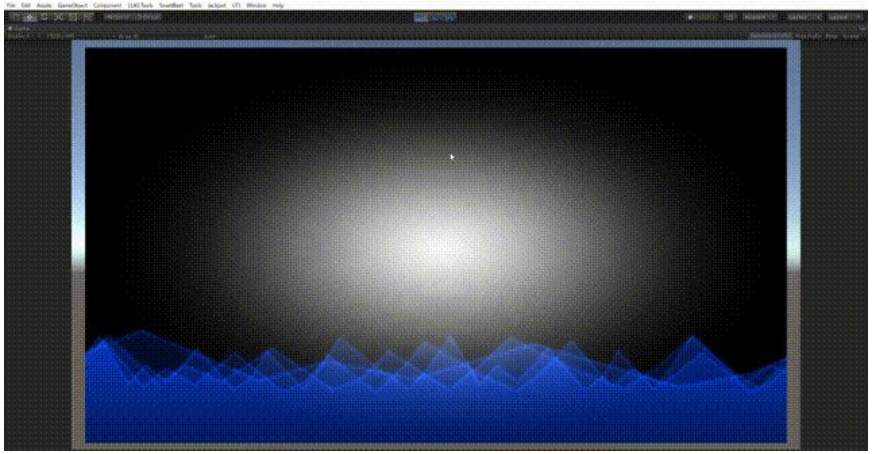
本日のアジェンダ

- **高品質**を実現するシェーダー
  - メンバー（キャラクター）用シェーダー
  - ステージ用シェーダー
- **低負荷**を実現するシェーダー
  - 負荷の計測
  - エフェクト汎用シェーダー
  - ドローコールとオーバードロー
  - エフェクト専用シェーダー

ディスプレイ専用シェーダー

ディスプレイ演出を専用シェーダーで最適化





最適化前: 汎用シェーダー

6枚の半透明Mesh の重ね合わせで
三角波のスクロールを表現

最適化

最適化後: 専用シェーダー

1枚の不透明Mesh だけで
シェーダーにより三角波のスクロールを表現

ディスプレイ演出を専用シェーダーで最適化

	最適化前: 汎用シェーダー	最適化後: 専用シェーダー
Mesh構成	**6枚**の半透明Mesh	**1枚**の不透明Mesh
ドローコール数 ※バッチングを考慮しない	**6**	**1**
オーバードロー ※不透明が最前面なら0とカウント	**6**	**0**
ライブ全体のCPU負荷 ※楽曲専用シェーダー以外の最適化も含まれた自動計測による結果 XperiaXZ『トリコリコPLEASE!!』3D高	30.14 ms	20.57 ms（**−9.57 ms**）

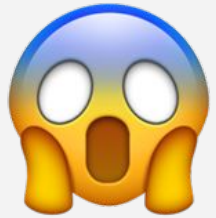
専用シェーダーによって**負荷削減が成功**😉

ペンライト
専用シェーダー

開発当初のペンライトが抱えていた課題

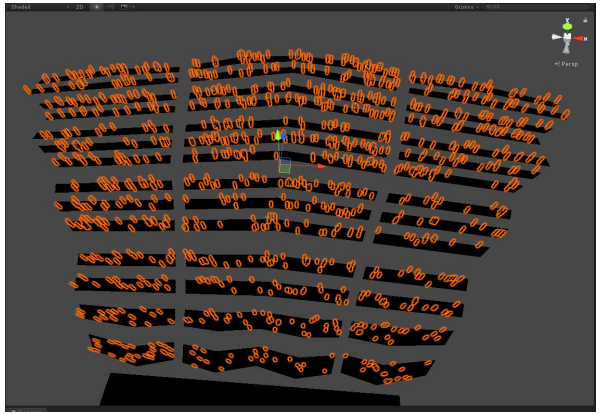
ParticleSystemによるペンライトでは**2つの課題**が発生

- **課題①** ParticleSystemの**CPU負荷が高い**
  - **パーティクル数**に比例してCPU計算が増える
    - 粒の動きのシミュレーション計算
    - ドローコールバッチングのMesh結合計算
  - すべて毎フレーム必要な計算となってしまう…



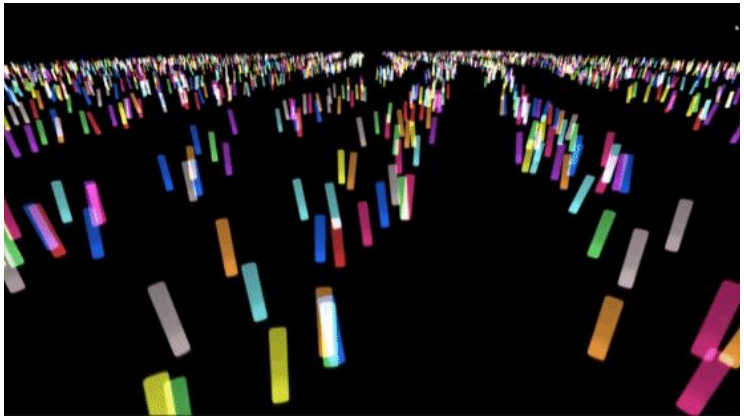
- **課題②** ライブ編成と連動した動的な**カラー指定が困難**

専用シェーダーで課題を解決！



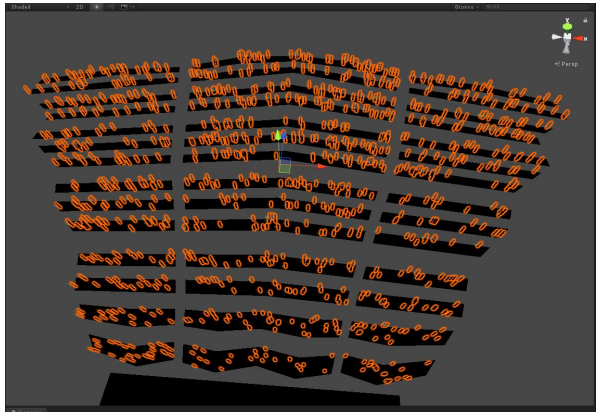
静的な1Mesh（ツールで生成）

専用シェーダー



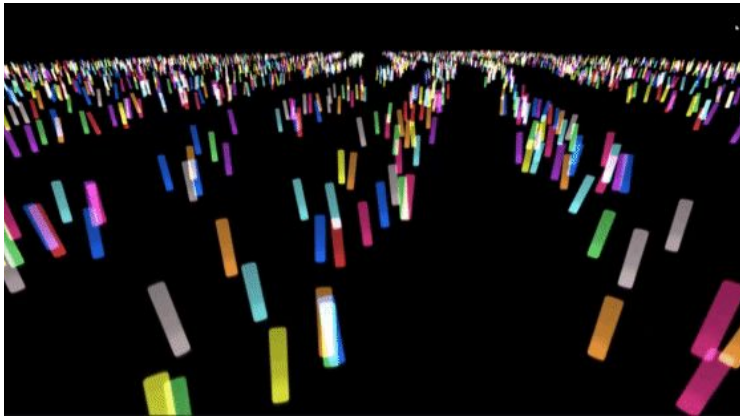
アニメーションと色を適用！

専用シェーダーで課題を解決！



静的な1Mesh（ツールで生成）

専用シェーダー

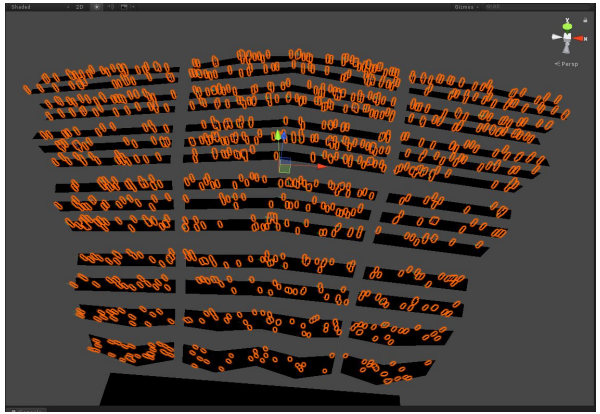


アニメーションと色を適用！

課題① CPU負荷😱

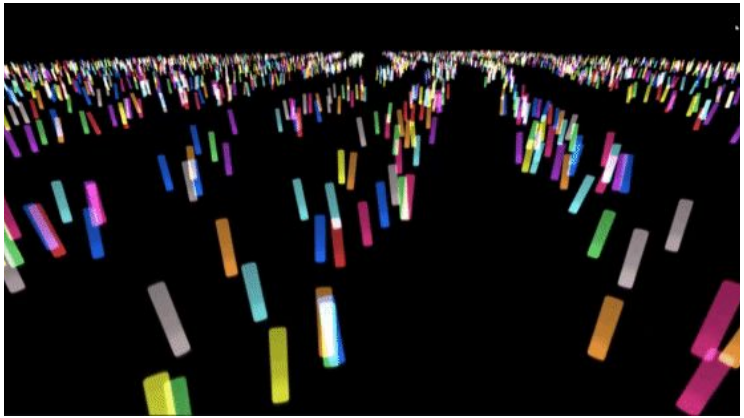
課題② カラー指定😱

専用シェーダーで課題を解決！



静的な1Mesh（ツールで生成）

専用シェーダー



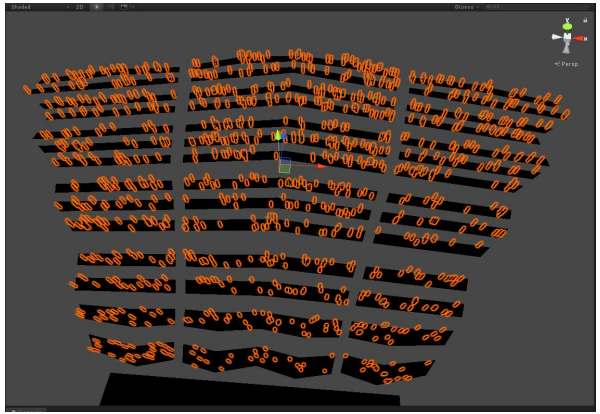
アニメーションと色を適用！

課題① CPU負荷😱

頂点シェーダーでアニメーションをGPU計算に🥰

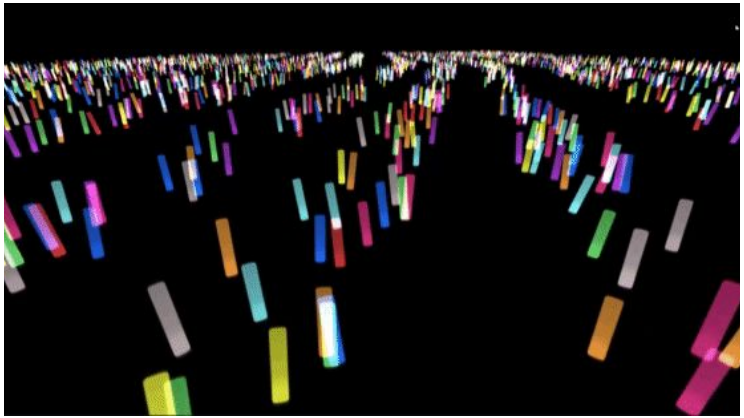
課題② カラー指定😱

専用シェーダーで課題を解決！



静的な1Mesh（ツールで生成）

専用シェーダー



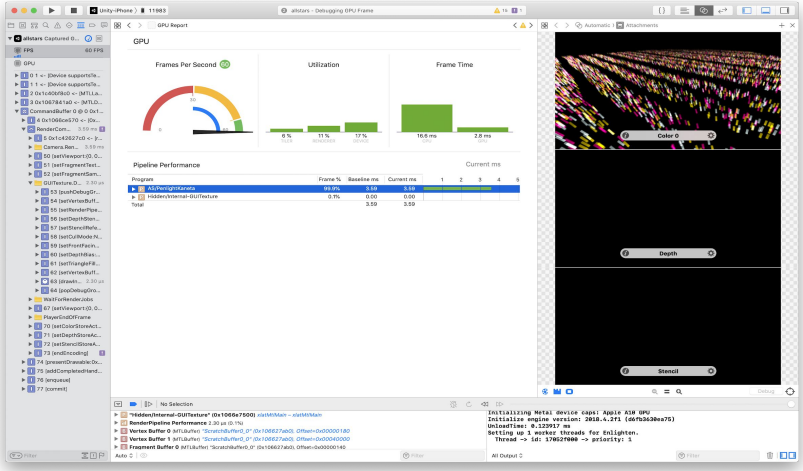
アニメーションと色を適用！

課題① CPU負荷😱	**頂点シェーダー**で**アニメーション**を**GPU計算に**🥰
課題② カラー指定😱	**カスタムシェーダー**で動的な**カラー指定**を実装🥰

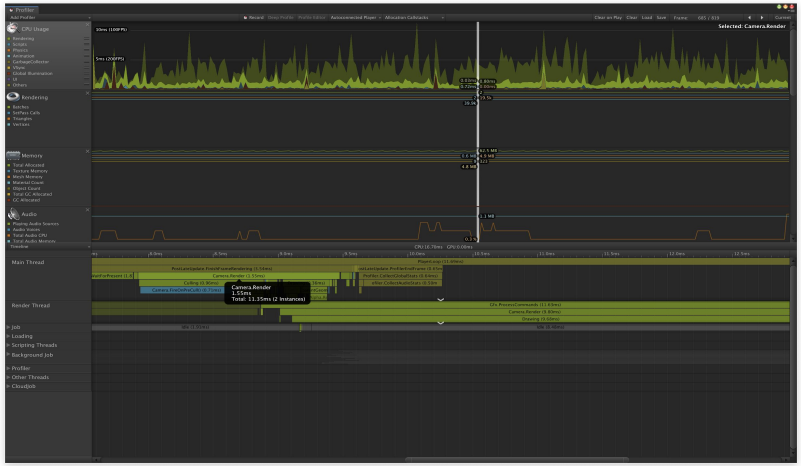
ペンライトの負荷

1ドローコール

- iPhone 7 Plus, ペンライト10000本
  - GPU負荷 実質 **0.476 ms**（2.8ms, Device Utilization 17%, フレーム全体）
  - CPU負荷 **1.55ms**（Main Thread の Camera.Render の処理時間）



Xcode



Unity Profiler

性能改善の理由

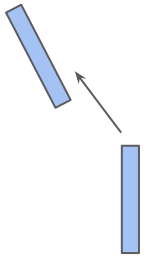
GPUは**並列計算**が得意

大量のペンライトの
動きは**並列計算**が適切

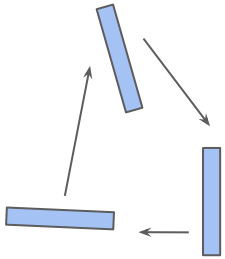
大量のペンライトの動きは**GPU計算が最適**

ペンライトの動き（品質向上もできた）

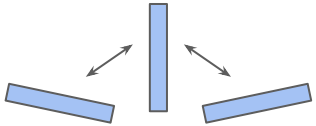
「打ち」「伸び」「横振り」など
現実のライブで使われる動きを
完全再現！



打ち

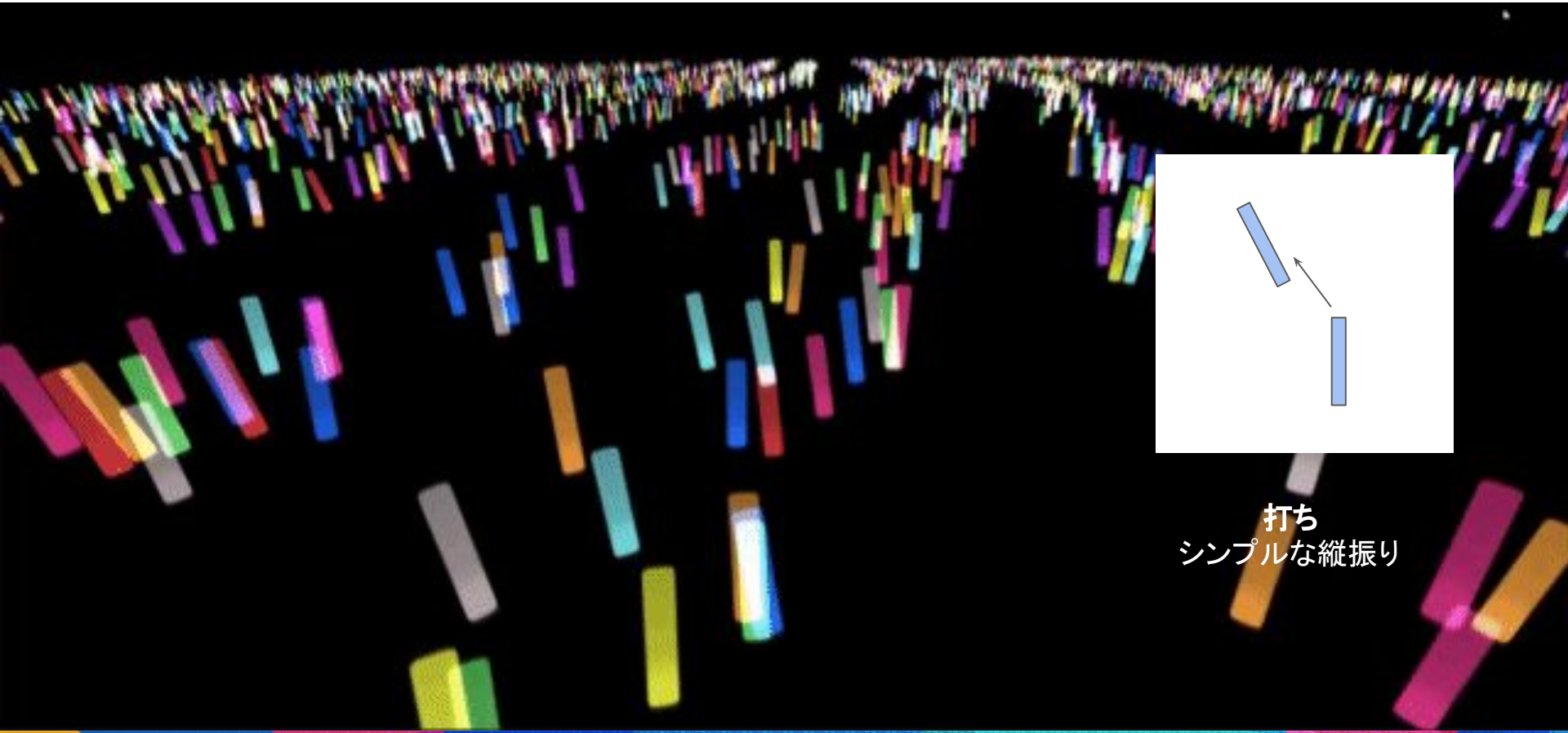


伸び



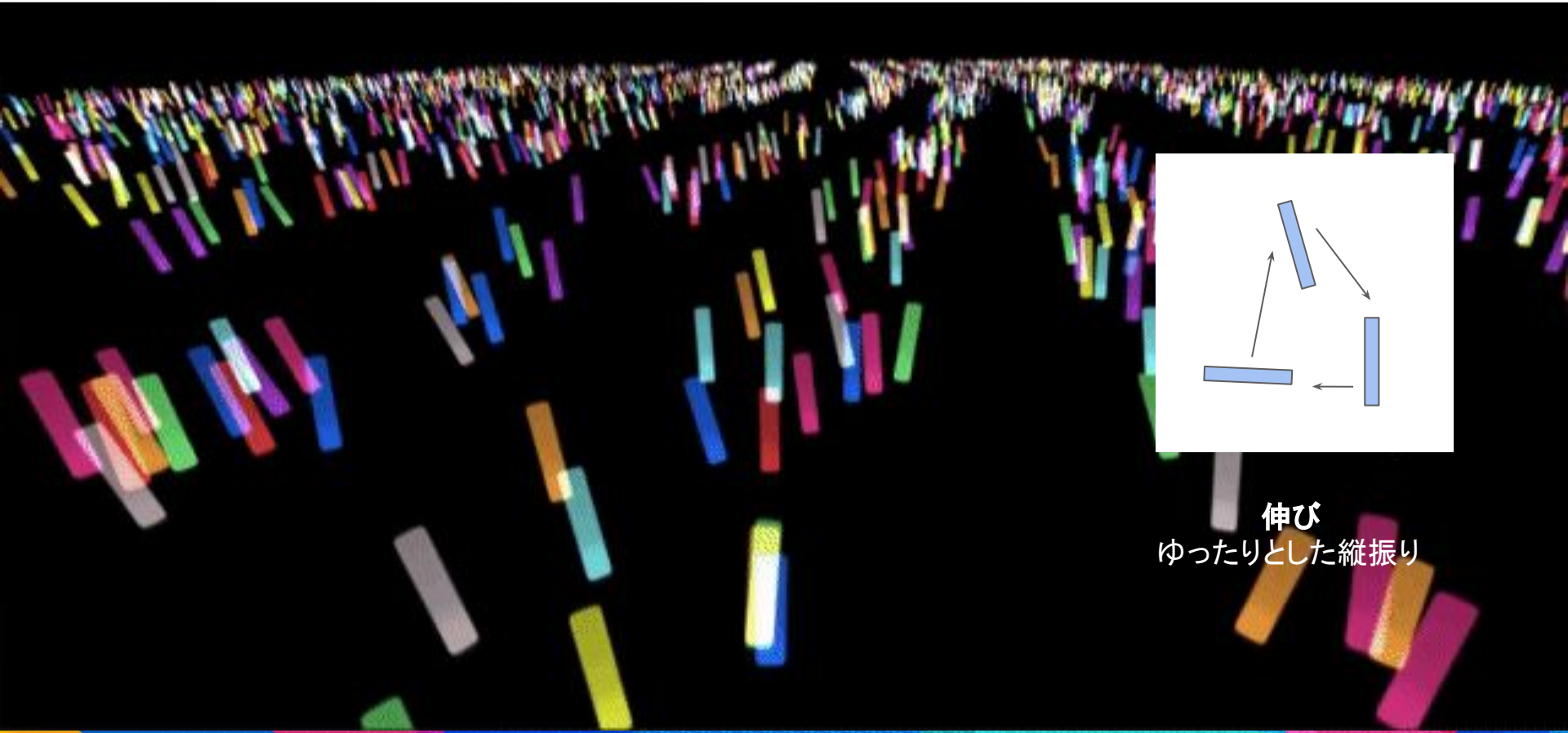
横振り

ペンライトの動き



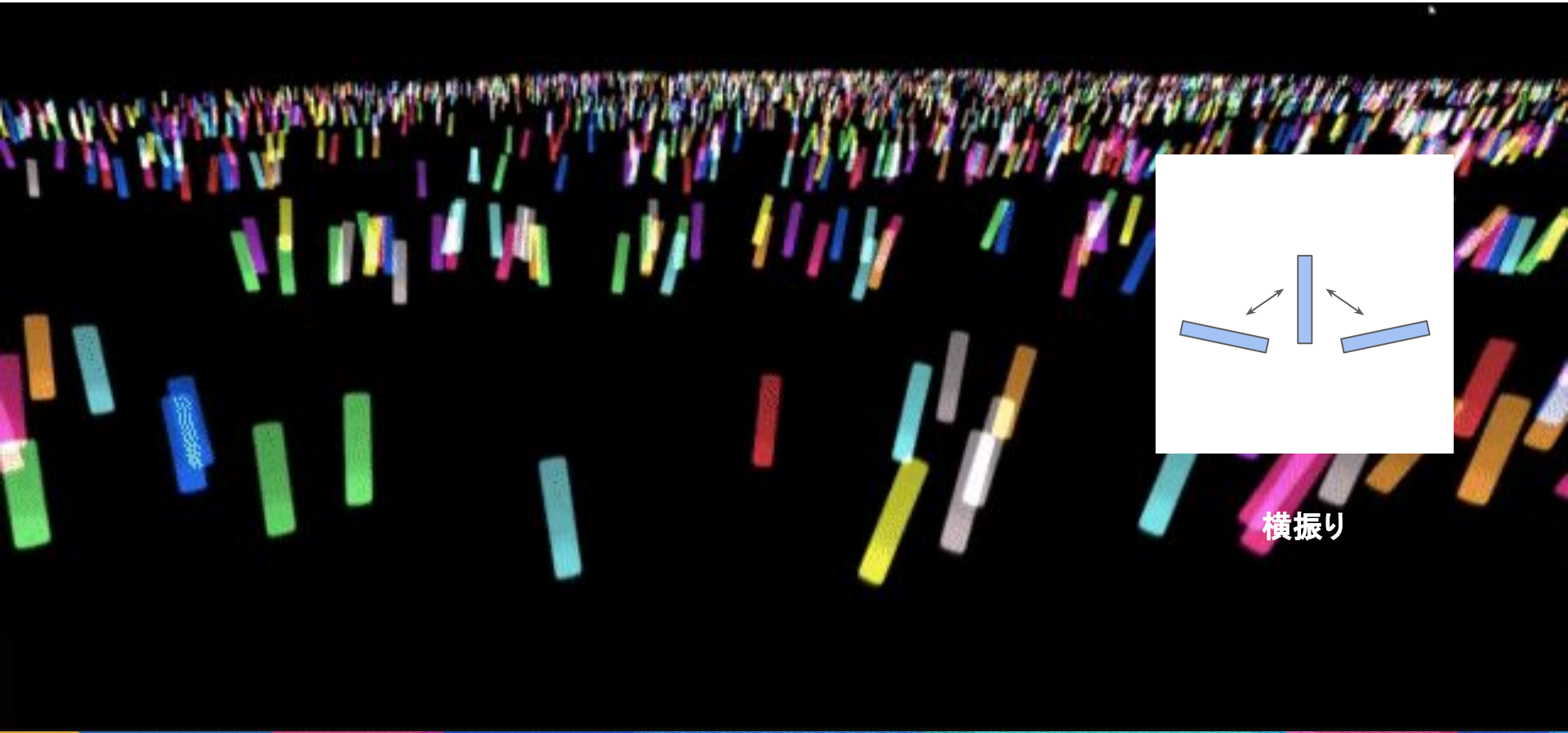
打ち
シンプルな縦振り

ペンライトの動き

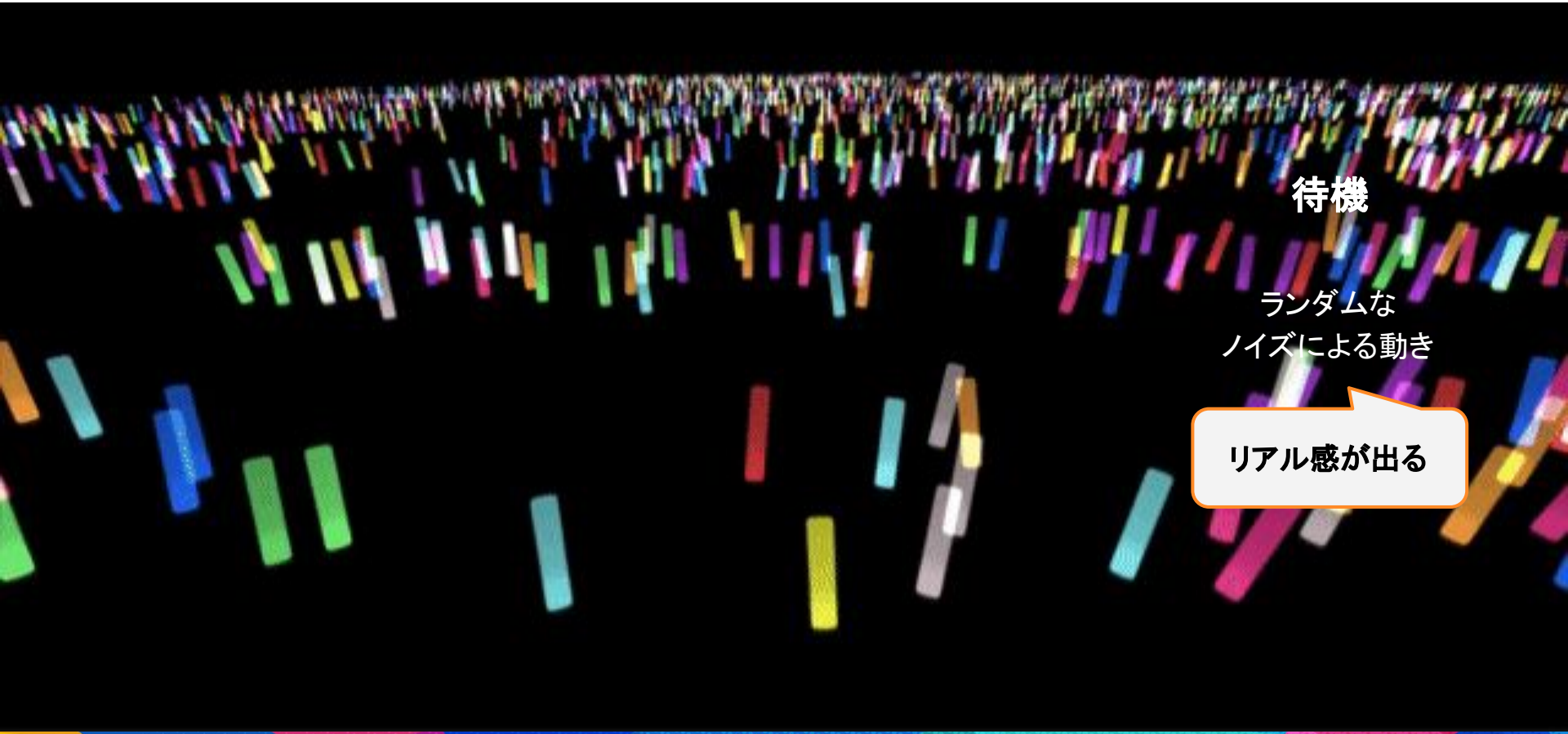


伸び
ゆったりとした縦振り

ペンライトの動き

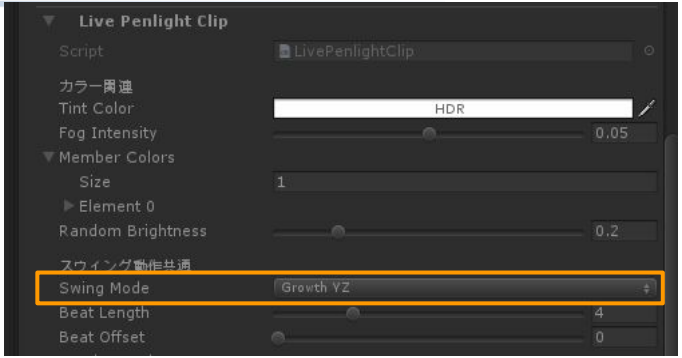


ペンライトの動き

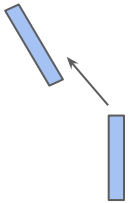


動作モードの指定

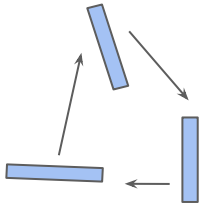
- Basic YZ: 打ち（普通の縦振り）
- Growth YZ: 伸び（ゆっくりした縦振り）
- Basic X: 横振り
- Stop: 待機（少しだけ動く）



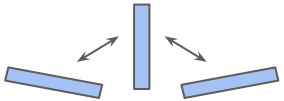
ペンライト用の Timelineのカスタムクリップ



Basic YZ
打ち



Growth YZ
伸び



Basic X
横振り

楽曲のビートとタイミングを同期

CRIのビートシンクに連動した**速度同期**を実装

秒数ではなく**ビート**数をシェーダーのプロパティに

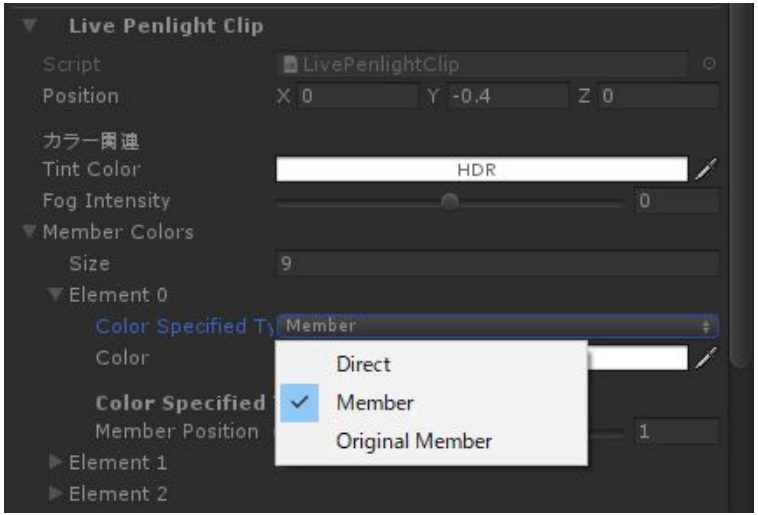
- 制作側のペンライトの速度調整の**工数削減**
- 楽曲の途中でBPM（テンポ）や拍子が変わる曲に対応
  - Music S.T.A.R.T!!（サビ前のみ2/4拍子）
  - Wonder zone（テンポチェンジ）
  - One More Sunshine Story（テンポチェンジ、拍変更あり）
  - おやすみなさん！（introにテンポチェンジあり）
  - さかなかなんだか？（拍変更あり）

ライブ編成に応じたカラー指定

3種類の**カラー指定モード**

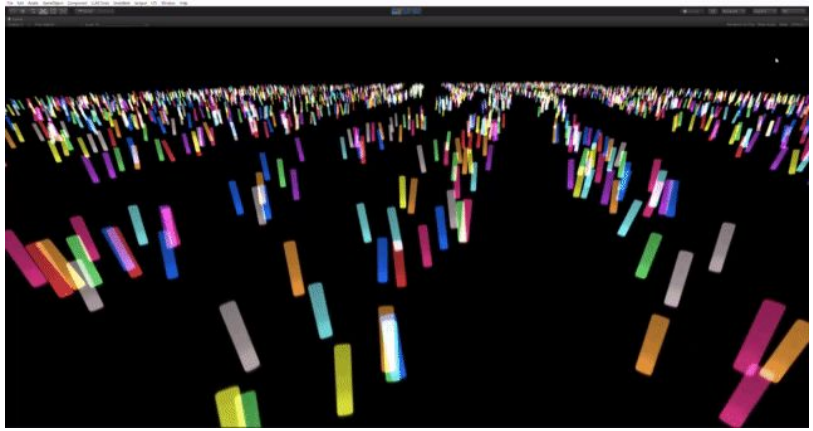
- **Direct**
  - 色を直接指定
- **Member**
  - **ライブ編成**（ダンスしている）の
    メンバーカラーを指定
- **Original Member**
  - オリジナル（**楽曲を歌っている**）の
    メンバーカラーを指定

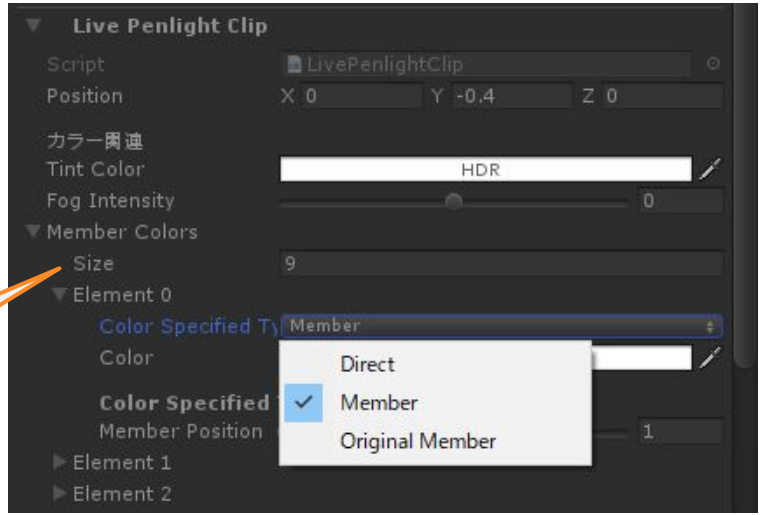
ライブ編成（推し）に
応じた**カラー指定**ができる



ペンライト用の Timelineのカスタムクリップ

ライブ編成に応じたカラー指定



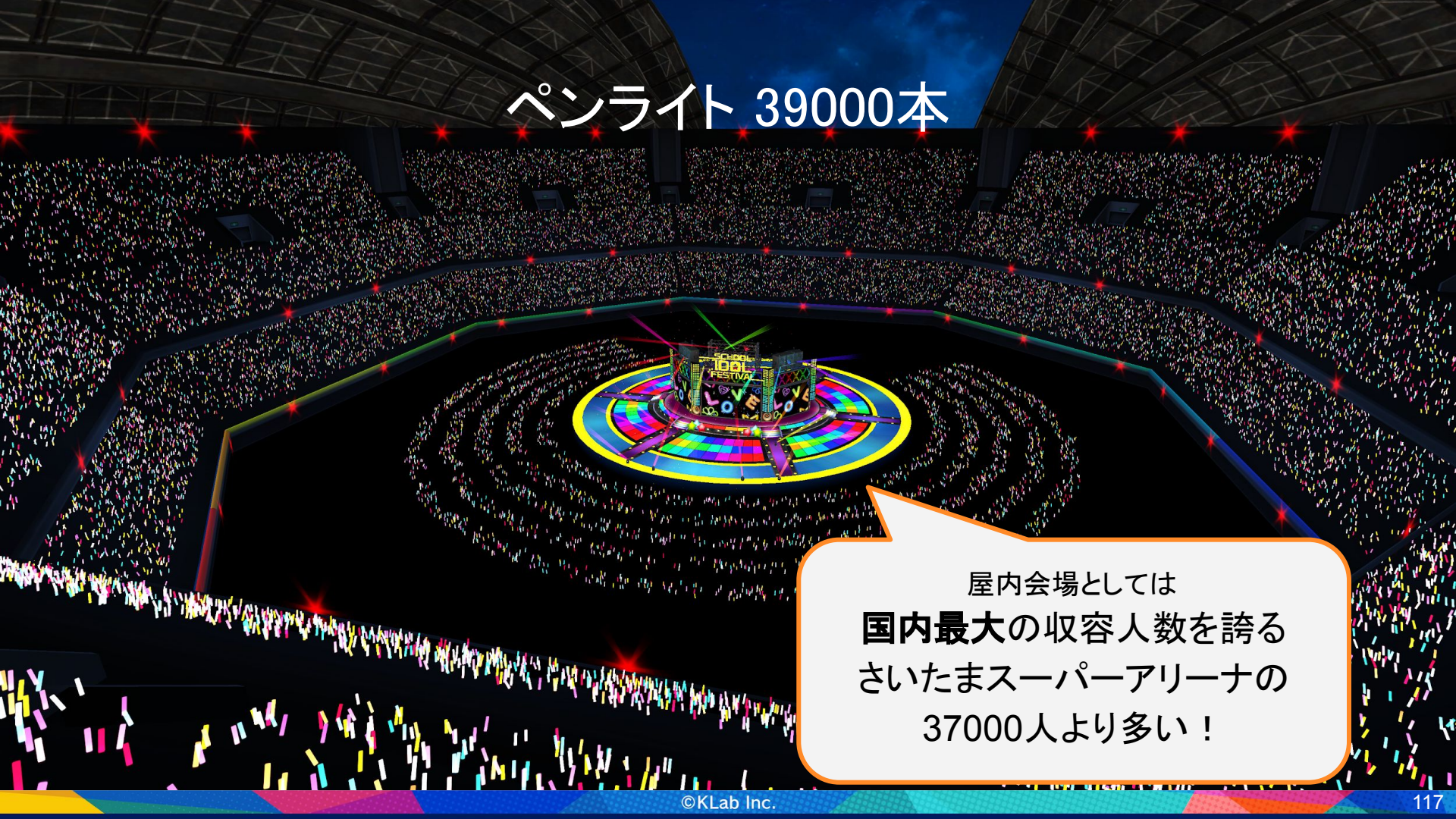


複数指定が可能

カラフルに！

ペンライト用の Timelineのカスタムクリップ

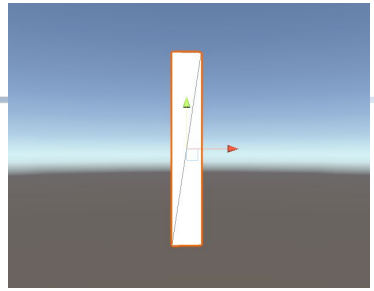
ペンライト 39000本



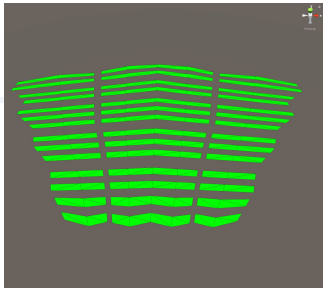
ズームすると…

Mesh生成ツール

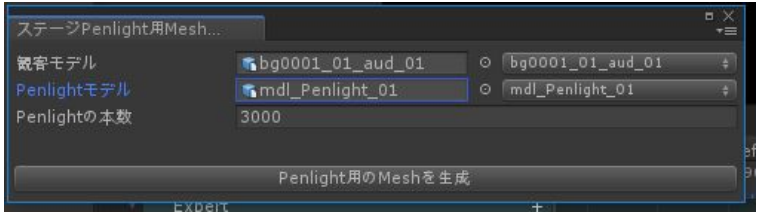
- 入力
  - 単一のペンライトのMesh
  - ペンライトの本数
  - 観客席の形状を指定するMesh
- 出力
  - 複数のペンライトを結合したMesh



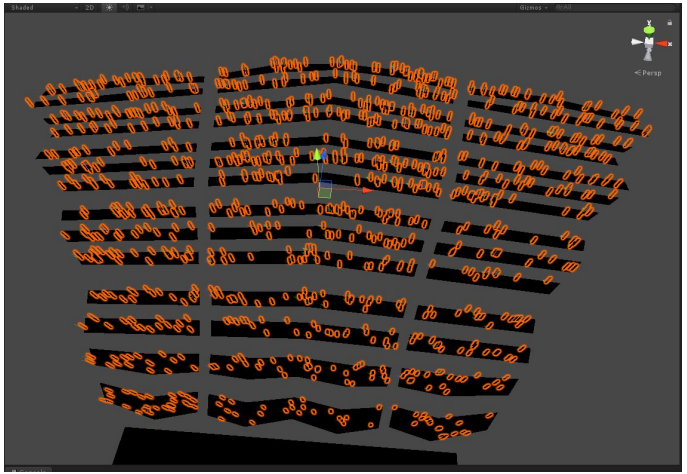
単一のペンライトMesh



観客席の形状を
指定するMesh



Unity Editor拡張として実装

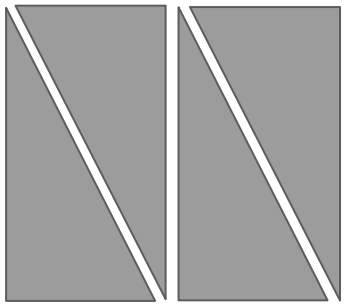


ペンライトのランダムな配置

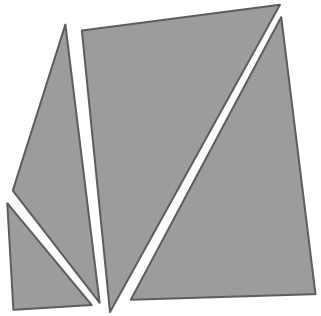
観客席の形状を指定するMeshの表面上からランダムに1点をサンプリング

手順

1.Meshを構成するポリゴンを1つ選ぶ
2.ポリゴン上からランダムに1点を選ぶ



面積が同じなら一様乱数でOK🙆



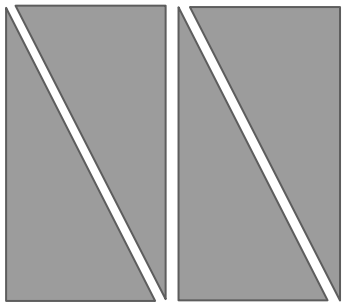
面積がバラバラだと密度が偏る🤦

ペンライトのランダムな配置

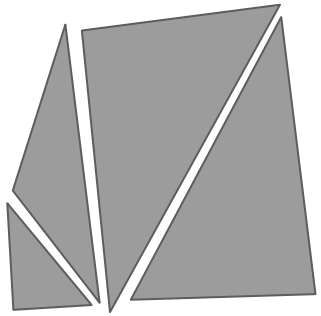
観客席の形状を指定するMeshの表面上からランダムに1点をサンプリング

手順

1.Meshを構成するポリゴンを1つ選ぶ
2.ポリゴン上からランダムに1点を選ぶ



面積が同じなら一様乱数でOK🙆



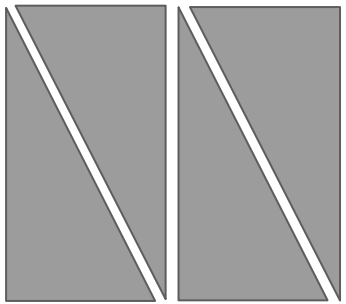
面積がバラバラだと密度が偏る🤦

ペンライトのランダムな配置

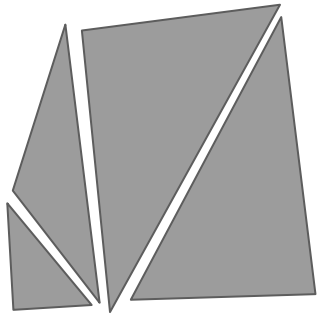
観客席の形状を指定するMeshの表面上からランダムに1点をサンプリング

手順

1.Meshを構成するポリゴンを1つ選ぶ
2.ポリゴン上からランダムに1点を選ぶ



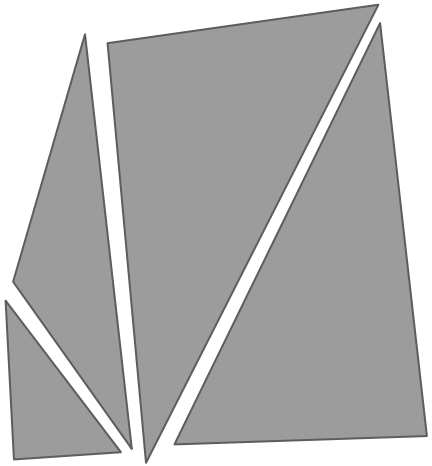
面積が同じなら一様乱数でOK🙆

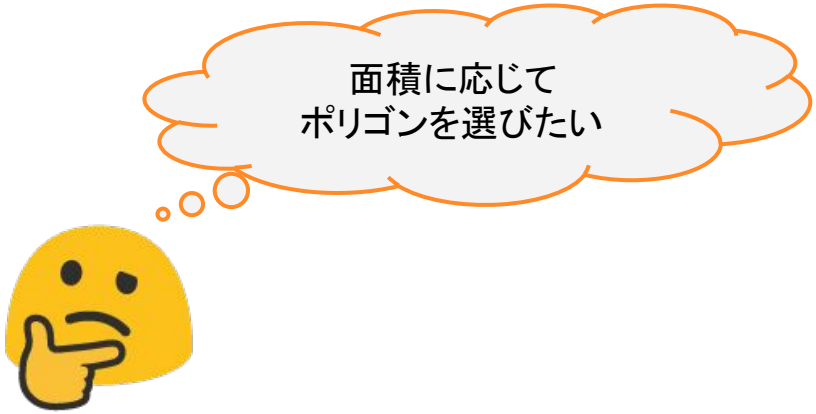


面積がバラバラだと密度が偏る🙅

ペンライトのランダムな配置

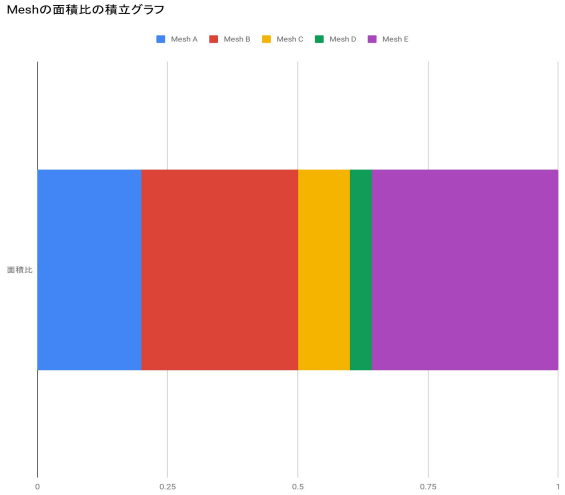
面積比率に比例した確率でポリゴンをランダムに選ぶ
重み付きランダム抽出が必要





累積分布関数（累積和）による重み付きランダム抽出

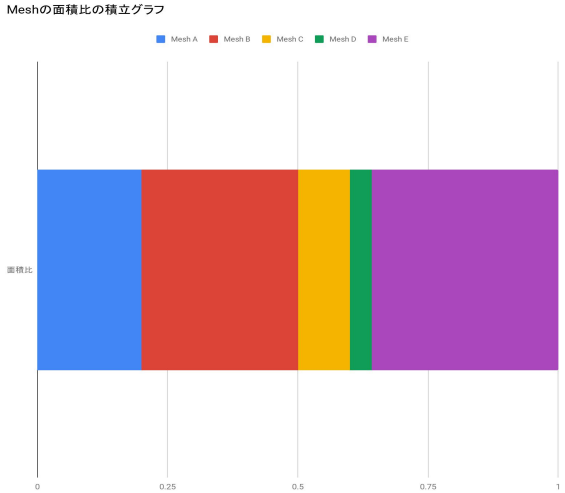
- **事前準備**
  - 面積の合計を1.0に正規化（確率になる）
  - 確率の箱を並べる



累積分布関数（累積和）による重み付きランダム抽出

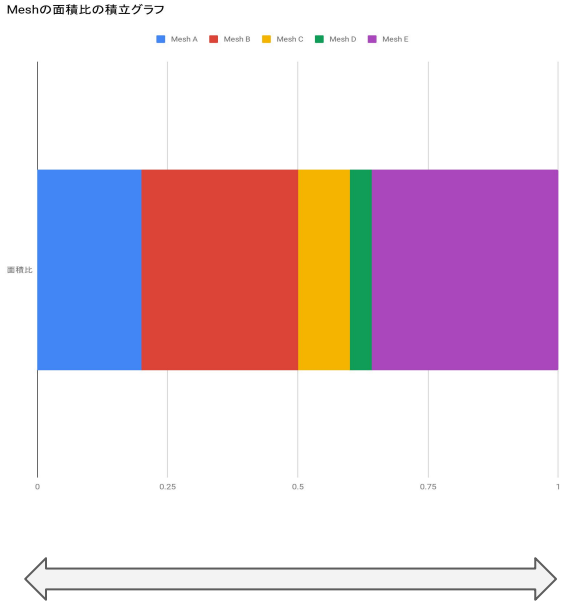
- **事前準備**
  - 面積の合計を1.0に正規化（確率になる）
  - 確率の箱を並べる

- **箱を選ぶ手順**



累積分布関数（累積和）による重み付きランダム抽出

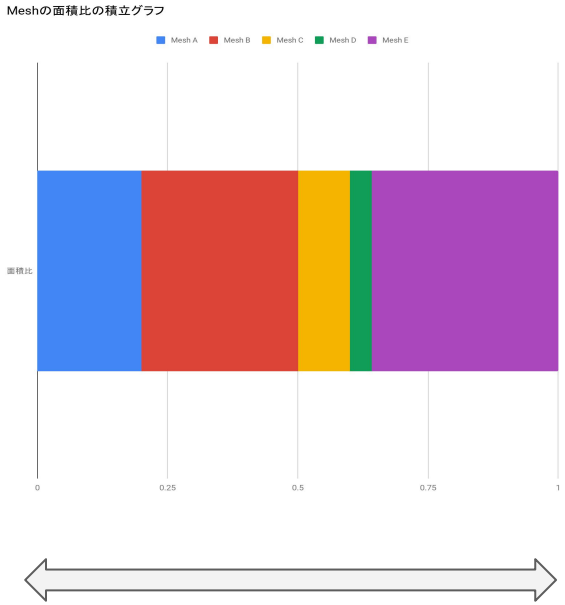
- **事前準備**
  - 面積の合計を1.0に正規化（確率になる）
  - 確率の箱を並べる
- **箱を選ぶ手順**
  - 0.0～1.0の一様乱数を生成



0.0～1.0の乱数を生成

累積分布関数（累積和）による重み付きランダム抽出

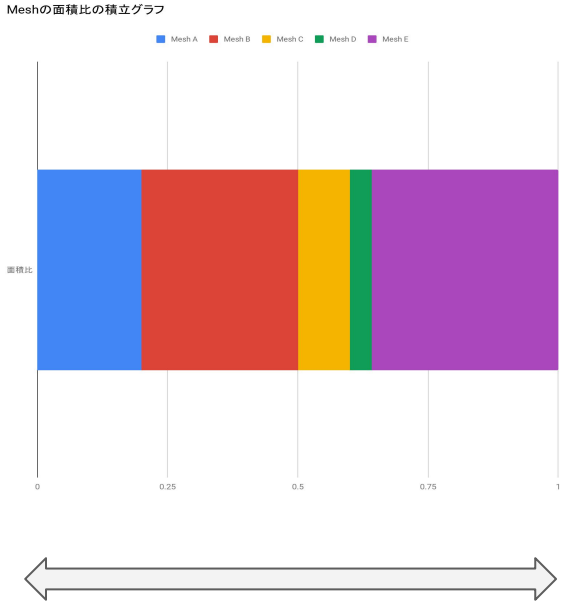
- **事前準備**
  - 面積の合計を1.0に正規化（確率になる）
  - 確率の箱を並べる
- **箱を選ぶ手順**
  - 0.0～1.0の一様乱数を生成
  - 乱数がどの箱の範囲に属するかを調べる



0.0～1.0の乱数を生成
どの箱に属するかを探索

累積分布関数（累積和）による重み付きランダム抽出

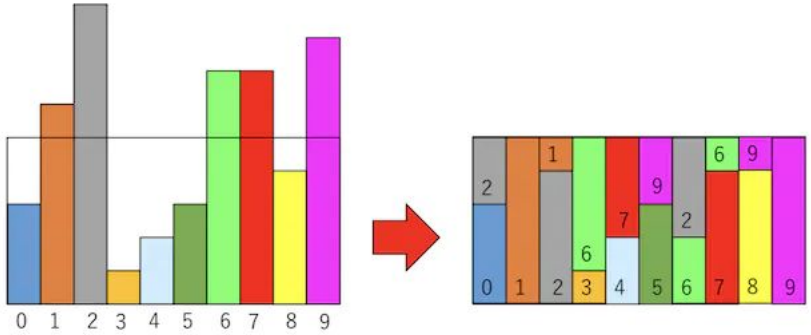
- **事前準備**
  - 面積の合計を1.0に正規化（確率になる）
  - 確率の箱を並べる
- **箱を選ぶ手順**
  - 0.0～1.0の一様乱数を生成
  - 乱数がどの箱の範囲に属するかを調べる
- **計算量**
  - 線形探索すれば O(N)
  - 二分探索すれば O(log N)



0.0～1.0の乱数を生成
どの箱に属するかを探索

Walker's Alias Method の簡単な解説

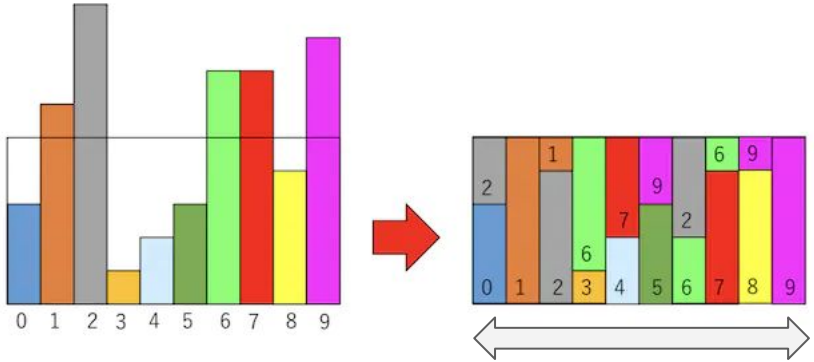
- 重み（面積）を保ったまま、Indexのペアを並べたデータ構造に置き換え



画像出典: @kaityo256. "Walker's Alias Methodの箱の作り方のわかりやすい説明"

Walker's Alias Method の簡単な解説

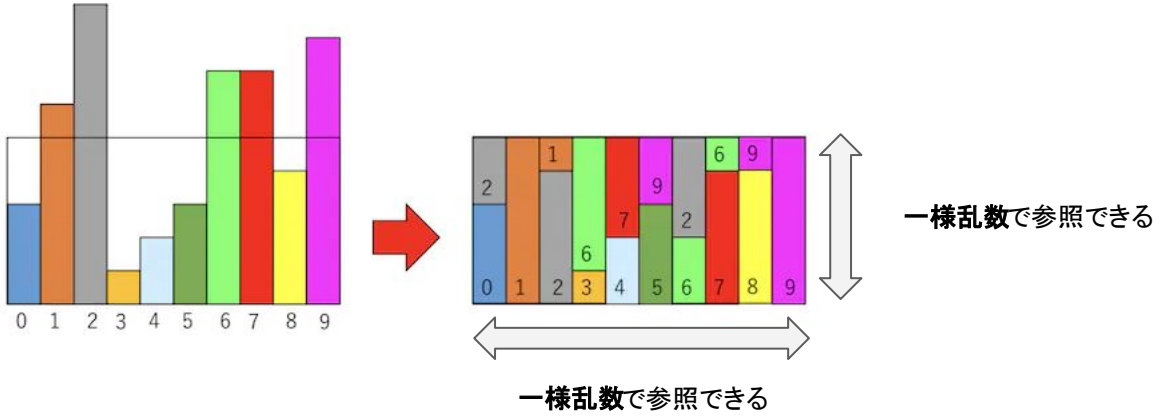
- 重み（面積）を保ったまま、Indexのペアを並べたデータ構造に置き換え



画像出典: @kaityo256. "Walker's Alias Methodの箱の作り方のわかりやすい説明"

Walker's Alias Method の簡単な解説

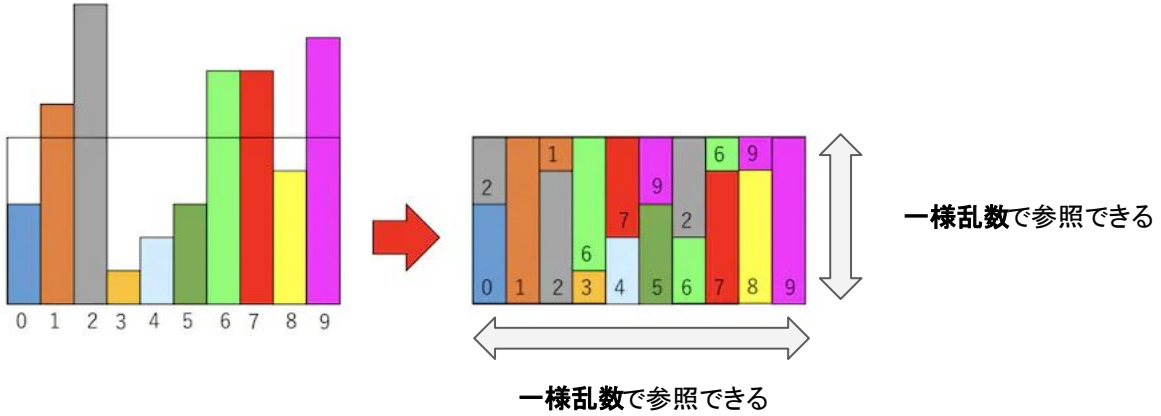
- 重み（面積）を保ったまま、Indexのペアを並べたデータ構造に置き換え



画像出典: @kaityo256. "Walker's Alias Methodの箱の作り方のわかりやすい説明"

Walker's Alias Method の簡単な解説

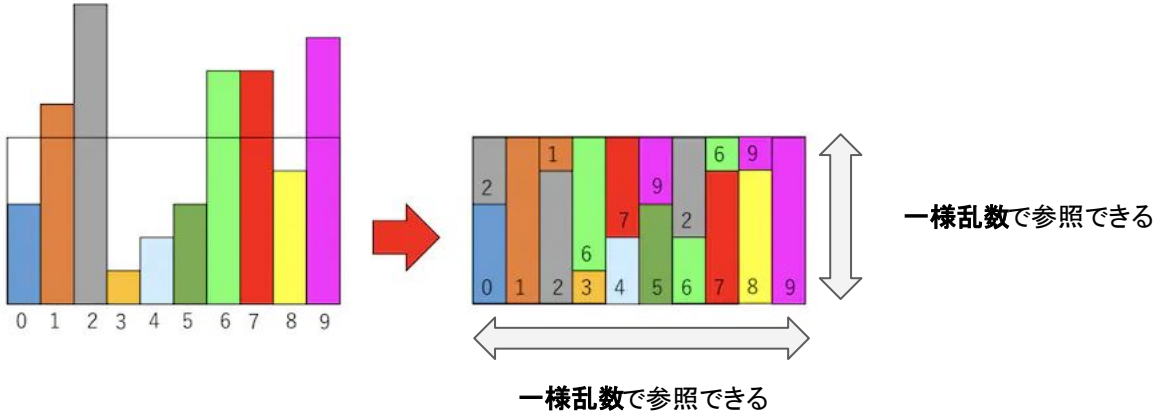
- 重み（面積）を保ったまま、Indexのペアを並べたデータ構造に置き換え
  - **一様乱数から2回**で参照できるので、O(1)



画像出典: @kaityo256. "Walker's Alias Methodの箱の作り方のわかりやすい説明"

Walker's Alias Method の簡単な解説

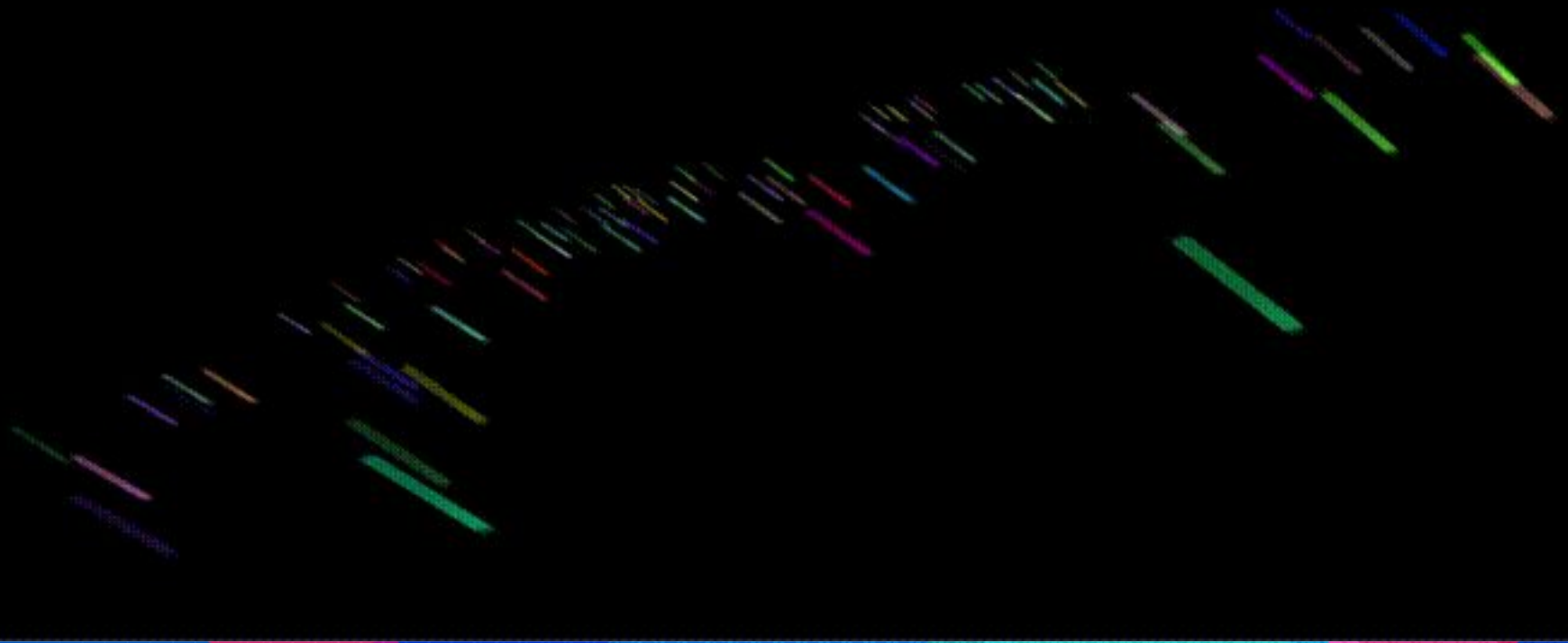
- 重み（面積）を保ったまま、Indexのペアを並べたデータ構造に置き換え
  - **一様乱数から2回**で参照できるので、O(1)
  - 事前データの生成は **O(N)** だが、
    今回はペンライトを何万本も生成するので、初期コストは相対的に無視できる



画像出典: @kaityo256. "Walker's Alias Methodの箱の作り方のわかりやすい説明"

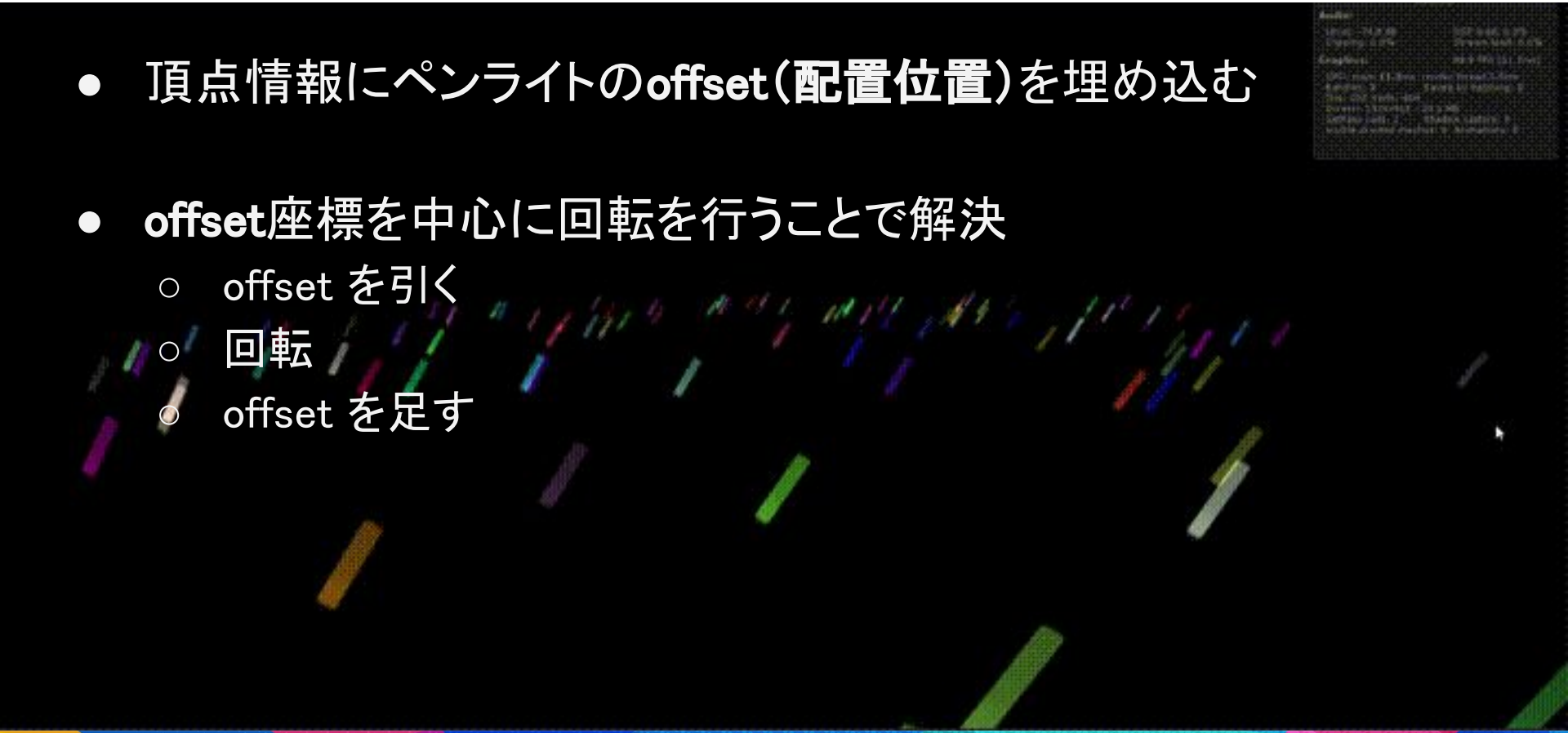
1Meshでの回転

普通に回転させるとMesh全体が回転してしまう…



1Meshでの回転

- 頂点情報にペンライトのoffset(配置位置)を埋め込む
- offset座標を中心に回転を行うことで解決
  - offset を引く
  - 回転
  - offset を足す



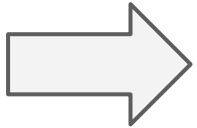
回転行列を 3x3 から 2x2 にして演算回数を削減

X軸を中心に回転する処理を最適化する例

```
float3x3 RotateX3x3(float angle)
{
  // X軸回転行列
  return float3x3(
     1,0,0,
     0,cos(angle),-sin(angle),
     0,sin(angle), cos(angle));
}
```

```
up = mul(RotateX3x3(rotateX), up);
```

最適化前: 3x3の回転行列でX軸回転



```
float2x2 Rotate2x2(float x)
{
   float c = cos(x);
   float s = sin(x);
   return float2x2(c, s, -s, c);
}
```

cos/sin の結果を変数にキャッシュ

yz成分だけ操作して2D回転に落とし込む

```
up.yz = mul(Rotate2x2(rotateX), up.yz);
```

最適化後: 2x2の回転行列でX軸回転

アニメーションカーブをシェーダで実装したい

アニメーションカーブは関数

- **入力**：時間
- **出力**：回転角 / 移動量

関数なので、**シェーダー**で**実装可能**

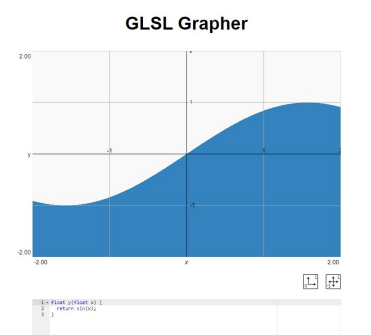
アニメーションカーブをシェーダで実装したい

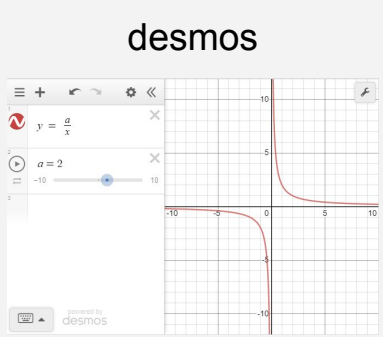
脳内だけでアニメーションカーブの設計は難しい🤯

シェーダーでアニメーションカーブを設計する**便利ツール**を利用🥰

GLSL Grapher
- GLSLの関数をグラフに表示するツール

Desmos
- 数式をグラフに表示するツール
- パラメータをスライダー制御できる





ペンライトのビルボード処理

ペンライトは**板ポリのMesh**
横から見ると、板なのがバレてしまう😨

ペンライトのビルボード処理

ペンライトは**板ポリのMesh**
横から見ると、板なのがバレてしまう😰

ビルボード処理で解決🥰
ペンライトMeshを常にカメラ側を向くように**回転**
いい感じに回転して板ポリなのをバレないように🤫

ペンライトのビルボード処理

ペンライトは**板ポリのMesh**
横から見ると、板なのがバレてしまう😰

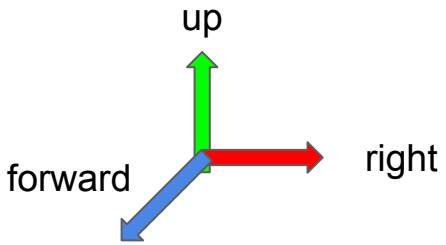
ビルボード処理で解決🥰
ペンライトMeshを常にカメラ側を向くように**回転**
いい感じに回転して板ポリなのをバレないように🤫

ビルボード処理 = いい感じの**回転行列の生成**

回転行列とは何だったのか

回転行列 = 回転後の空間の**基底ベクトル**を並べたもの

$$M_R = \begin{bmatrix} \text{right.x} & \text{right.y} & \text{right.z} \\ \text{up.x} & \text{up.y} & \text{up.z} \\ \text{forward.x} & \text{forward.y} & \text{forward.z} \end{bmatrix}$$



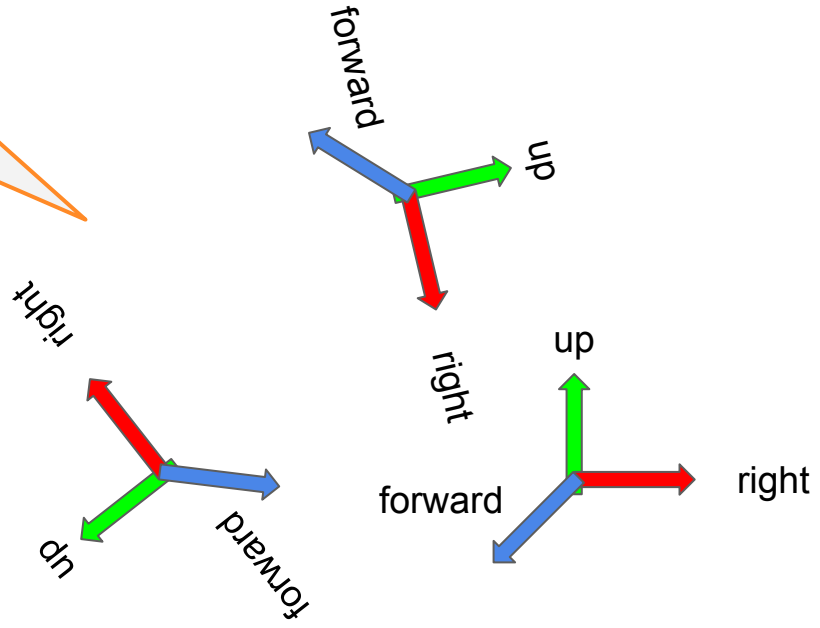
Unityシェーダーの行列のメモリ配置は行優先

回転行列とは何だったのか

回転行列 = 回転後の空間の**基底ベクトル**を並べたもの

どれだけ回転して
基底ベクトルの値が変化しても
回転行列の定義は変わらない

$$M_R = \begin{bmatrix} \text{right.x} & \text{right.y} & \text{right.z} \\ \text{up.x} & \text{up.y} & \text{up.z} \\ \text{forward.x} & \text{forward.y} & \text{forward.z} \end{bmatrix}$$

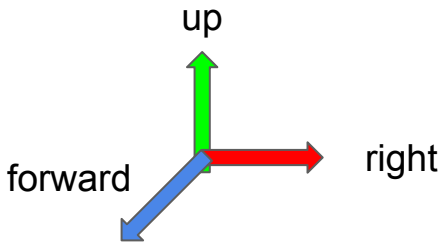


Unityシェーダーの行列のメモリ配置は行優先

回転行列とは何だったのか

回転行列 = 回転後の空間の**基底ベクトル**を並べたもの

$$M_R = \begin{bmatrix} \text{right.x} & \text{up.x} & \text{forward.x} \\ \text{right.y} & \text{up.y} & \text{forward.y} \\ \text{right.z} & \text{up.z} & \text{forward.z} \end{bmatrix}$$



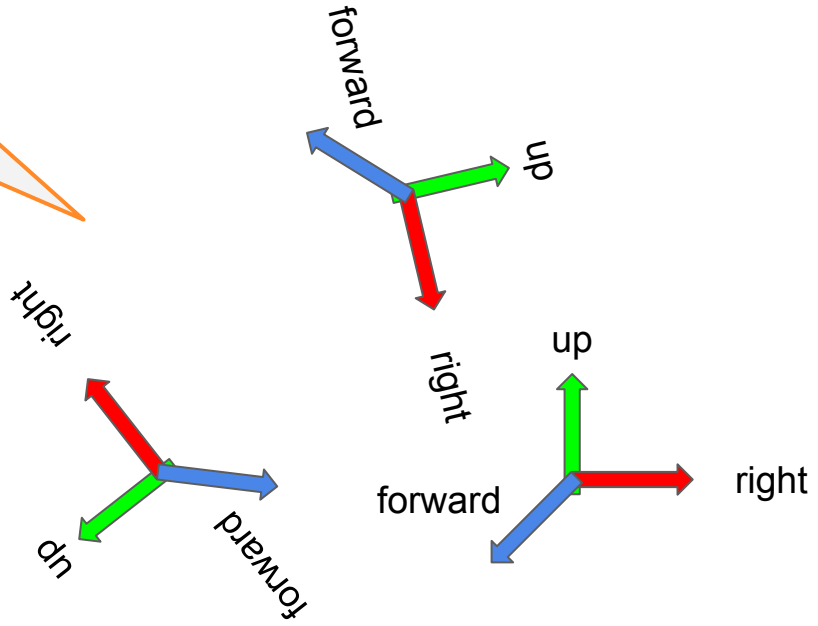
列優先の行列

回転行列とは何だったのか

回転行列 = 回転後の空間の**基底ベクトル**を並べたもの

どれだけ回転して
基底ベクトルの値が変化しても
回転行列の定義は変わらない

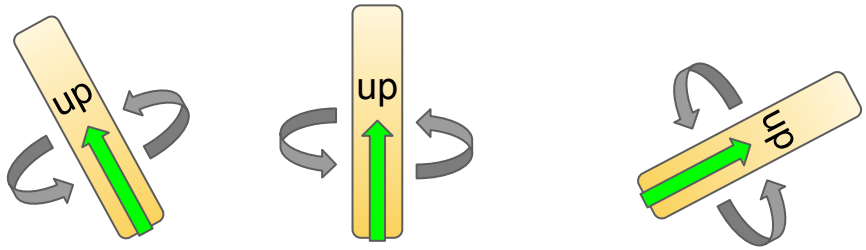
$$M_R = \begin{bmatrix} \text{right.x} & \text{up.x} & \text{forward.x} \\ \text{right.y} & \text{up.y} & \text{forward.y} \\ \text{right.z} & \text{up.z} & \text{forward.z} \end{bmatrix}$$



列優先の行列

upベクトル = ペンライトの芯線

- upベクトル
  - ペンライトの芯線の向き
  - デフォルトはY軸(0, 1, 0)だが、ペンライトの動きにより変化
  - ビルボードの回転の中心軸



ビルボード用の回転行列の実装例

```
up = mul((float3x3)unity_ObjectToWorld, up);
float3 worldPos = mul(unity_ObjectToWorld,
                      float4(v.offset + move, 1.0));
float3 toCamera = _WorldSpaceCameraPos - worldPos;
float3 right = normalize(cross(toCamera, up));
float3 forward = normalize(cross(up, right));
```

```
float4x4 mat = unity_ObjectToWorld;
mat._m00_m10_m20 = right;
mat._m01_m11_m21 = up;
mat._m02_m12_m22 = forward;
mat._m03_m13_m23 = worldPos;
```

ビルボード用の回転行列の実装例

up(Y軸)はペンライトの動きで変化

```
up = mul((float3x3)unity_ObjectToWorld, up);
float3 worldPos = mul(unity_ObjectToWorld,
                      float4(v.offset + move, 1.0));
float3 toCamera = _WorldSpaceCameraPos - worldPos;
float3 right = normalize(cross(toCamera, up));
float3 forward = normalize(cross(up, right));
```

```
float4x4 mat = unity_ObjectToWorld;
mat._m00_m10_m20 = right;
mat._m01_m11_m21 = up;
mat._m02_m12_m22 = forward;
mat._m03_m13_m23 = worldPos;
```

ビルボード用の回転行列の実装例

up(Y軸)はペンライトの動きで変化

toCamera はペンライトから
カメラの方向ベクトル

```
up = mul((float3x3)unity_ObjectToWorld, up);
float3 worldPos = mul(unity_ObjectToWorld,
                      float4(v.offset + move, 1.0));
float3 toCamera = _WorldSpaceCameraPos - worldPos;
float3 right = normalize(cross(toCamera, up));
float3 forward = normalize(cross(up, right));
```

```
float4x4 mat = unity_ObjectToWorld;
mat._m00_m10_m20 = right;
mat._m01_m11_m21 = up;
mat._m02_m12_m22 = forward;
mat._m03_m13_m23 = worldPos;
```

ビルボード用の回転行列の実装例

up(Y軸)はペンライトの動きで変化

toCamera はペンライトから
カメラの方向ベクトル

right(X軸) は toCamera と up の両方に
直交するベクトルなので外積で計算

```
up = mul((float3x3)unity_ObjectToWorld, up);
float3 worldPos = mul(unity_ObjectToWorld,
                      float4(v.offset + move, 1.0));
float3 toCamera = _WorldSpaceCameraPos - worldPos;
float3 right = normalize(cross(toCamera, up));
float3 forward = normalize(cross(up, right));
```

```
float4x4 mat = unity_ObjectToWorld;
mat._m00_m10_m20 = right;
mat._m01_m11_m21 = up;
mat._m02_m12_m22 = forward;
mat._m03_m13_m23 = worldPos;
```

ビルボード用の回転行列の実装例

up(Y軸)はペンライトの動きで変化

toCamera はペンライトから
カメラの方向ベクトル

right(X軸) は toCamera と up の両方に
直交するベクトルなので外積で計算

forward(Z軸) は up と right の両方に直
交するベクトルなので外積で計算

```
up = mul((float3x3)unity_ObjectToWorld, up);
float3 worldPos = mul(unity_ObjectToWorld,
                      float4(v.offset + move, 1.0));
float3 toCamera = _WorldSpaceCameraPos - worldPos;
float3 right = normalize(cross(toCamera, up));
float3 forward = normalize(cross(up, right));
```

```
float4x4 mat = unity_ObjectToWorld;
mat._m00_m10_m20 = right;
mat._m01_m11_m21 = up;
mat._m02_m12_m22 = forward;
mat._m03_m13_m23 = worldPos;
```

ビルボード用の回転行列の実装例

up(Y軸)はペンライトの動きで変化

toCamera はペンライトから
カメラの方向ベクトル

right(X軸) は toCamera と up の両方に
直交するベクトルなので外積で計算

forward(Z軸) は up と right の両方に直
交するベクトルなので外積で計算

up, right, forward を並べた行列を作る

```
up = mul((float3x3)unity_ObjectToWorld, up);
float3 worldPos = mul(unity_ObjectToWorld,
                      float4(v.offset + move, 1.0));
float3 toCamera = _WorldSpaceCameraPos - worldPos;
float3 right = normalize(cross(toCamera, up));
float3 forward = normalize(cross(up, right));
```

```
float4x4 mat = unity_ObjectToWorld;
mat._m00_m10_m20 = right;
mat._m01_m11_m21 = up;
mat._m02_m12_m22 = forward;
mat._m03_m13_m23 = worldPos;
```

ビルボード用の回転行列の実装例

up(Y軸)はペンライトの動きで変化

toCamera はペンライトからカメラの方向ベクトル

right(X軸) は toCamera と up の両方に直交するベクトルなので外積で計算

forward(Z軸) は up と right の両方に直交するベクトルなので外積で計算

up, right, forward を並べた行列を作る

```
up = mul((float3x3)unity_ObjectToWorld, up);
float3 worldPos = mul(unity_ObjectToWorld,
                      float4(v.offset + move, 1.0));
float3 toCamera = _WorldSpaceCameraPos - worldPos;
float3 right = normalize(cross(toCamera, up));
up = normalize(cross(right, toCamera));
```

```
float4x4 mat = unity_ObjectToWorld;
mat._m00_m10_m20 = right;
mat._m01_m11_m21 = up;
mat._m02_m12_m22 = forward;
mat._m03_m13_m23 = worldPos;
```

up ベクトルとカメラへの向きが決まれば、
right と forward は自動で決定するので、
ビルボード用の回転行列は計算可能

ペンライトまとめ

- 頂点シェーダーでアニメーションを実装
  - **シェーダー芸を実用的に応用**
  - 全楽曲で利用できるので、費用対効果も高い

品質向上と負荷削減を両立！

負荷対策まとめ

- **カラーリングシェーダー**
  - マテリアルを統合してドローコールを削減
- **描画順の整理**
  - ドローコールバッチングを効率化
- **ディスプレイ専用シェーダー**
  - ドローコールとオーバードローを削減
- **ペンライト専用シェーダー**
  - 1Mesh化によりドローコール削減
  - アニメーションをGPU計算

負荷対策まとめ

- **カラーリングシェーダー**
  - マテリアルを統合してドローコールを削減
- **描画順の整理**
  - ドローコールバッチングを効率化
- **ディスプレイ専用シェーダー**
  - ドローコールとオーバードローを削減
- **ペンライト専用シェーダー**
  - 1Mesh化によりドローコール削減
  - アニメーションをGPU計算

負荷対策が**大成功**！

関連セッションの紹介

- **Post Processing Stack v1の軽量化**（GPUの負荷削減）
- **Unity Timeline**
  - プロジェクト専用の**カスタムトラック**（ペンライトの他にも多数あります）
  - 制作視点でのメリットや利用した感想

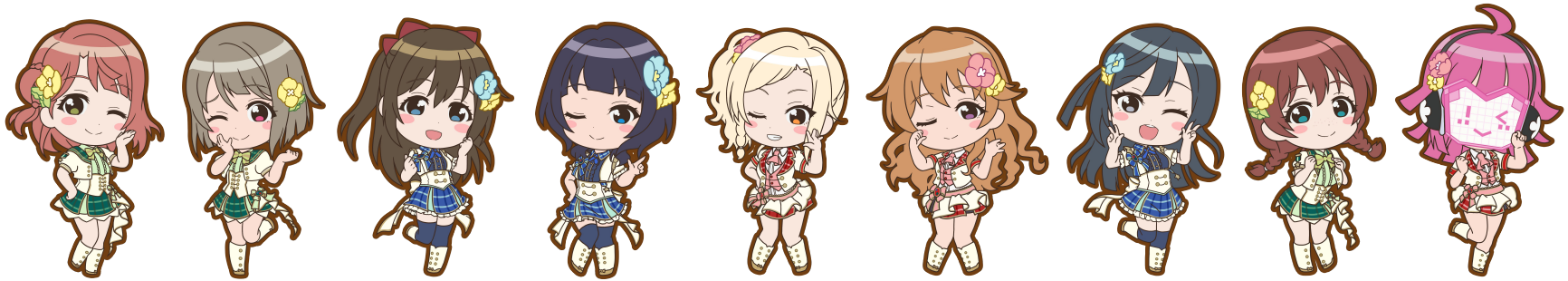
09月04日(金) 15:30 ～ 16:30

『ラブライブ！スクールアイドルフェスティバル ALL STARS』（スクスタ）のライブ演出制作秘話

～超高品質な3Dライブ演出を表現するための制作概要とオリジナルTimelineツールについて

今後も **最強で最高のアイドルゲーム**

の提供を目指して頑張ります！



質疑応答