Cubism SDK for Cocos Creator alpha マニュアル

変更履歴

Please check CHANGELOG and NOTICE

Cubism SDK for Cocos Creatorの値の操作タイミングについて

Cocos Creatorでモデルのパラメータの値を操作する場合、 Component. **lateUpdate** () のタイミングで行うことを推奨しております。

```
//非推奨
update(deltaTime: number)
{
  model.parameters[0].value = value;

  CubismParameterExtensionMethods.blendToValue(model.parameters[1].value,
CubismParameterBlendMode.Additive, value);
}

//推奨
lateUpdate(deltaTime: number)
{
  model.parameters[0].value = value;

  CubismParameterExtensionMethods.blendToValue(model.parameters[1].value,
CubismParameterBlendMode.Additive, value);
}
```

Live2D Cubism SDK for Cocos Creatorでは、アニメーション再生はCocos Creatorのビルトイン機能であるAnimationを利用しており、これらはパラメータの値を `Component.update()` から `Component.lateUpdate()` の間で適用します。 そのため、`Component.update()` でパラメータの値を設定した場合、実行順序の関係により、アニメーションでその値を上書きしてしまうこともあります。

Cocos Creatorのイベント関数に関してはCocos Creator公式のドキュメントをご覧ください。

Cubism SDK for Cocos Creatorのパフォーマンス・チューニングについて

概要

Cubism SDKにおいては、モデルの構造によってプログラム上でのパフォーマンスに影響が出る場合があります。 また、Live2D SDK for Cocos Creatorは構造のわかりやすさを優先しているため、一部意図的にパフォーマンスを犠牲にしている部分があります。 以下では、SDKのパフォーマンスに影響がある点を説明します。

モデルの構造

詳細については こちら をご覧ください。

プロジェクトの構造

Cocos Creatorのイベント関数 前述の通りLive2D SDK for Cocos Creatorは構造のわかりやすさを優先しております。 そのため、各コンポーネントの更新処理は、lateUpdate()などのCocos Creatorのイベント関数から行っています。 このCocos Creatorのイベント関数は、呼び出すコストが少なくありません。 SDKをそのまま使用してモデルを複数表示した場合、実行環境によっては非常にパフォーマンスが悪くなってしまうことがございます。 パフォーマンスを重視するなら、各コンポーネントの更新処理をひとつのControllerから呼び出すようにすることが推奨されています。 Cubism SDK for Cocos Creatorに同梱されている **CubismUpdateController** は、Cubism SDKのコンポーネントの実行順を制御するために同様の処理を行っているため、上記の対応を行うのであればCubismUpdateControllerの実装が参考になります。

CubismUpdateControllerの詳細は チュートリアル - UpdateControllerの設定方法 をご覧ください。

Cubism SDK for Cocos Creatorでのパラメータ操作について

Cubism SDK for Cocos Creatorでモデルのパラメータの値を操作する場合、Cocos Creatorのイベント関数の `Component.lateUpdate()` のタイミングで行う必要があります。

Cocos Creatorのイベント関数にの実行順ついては こちら をご覧ください。

AnimationClipを再生させる処理は、Cocos Creatorのイベント関数の `Component.update()` と `Component.lateUpdate()` の間で行われます。 仮にパラメータの値を `Component.update()` で操作した場合、直後に再生されるAnimationClipの値によって上書きされてしまいます。 そのため、パラメータの値を操作する場合はAnimationClipが値を設定した後に行います。

Cubism SDK for Cocos Creator同梱のコンポーネントはすべて `Component.lateUpdate()` からパラメータの値を操作しております。

また、各フレームでパラメータの値からモデルの頂点更新処理をCocos Creator の　描画後イベント (Director.EVENT_AFTER_DRAW) で行っているため、これよりも後で値の操作を行うと計算されません。

【Cocos Creator】乗算色・スクリーン色

モデルに乗算色・スクリーン色を適用することで、色合いをリアルタイムに変化させることが出来ます。Cubism Editor上で設定した乗算色・スクリーン色はCubism 4.2以降の SDK for Cocos Creatorを利用することで、特に追加のコーディングをすることなく適用されます。 Cubism Editor上での乗算色・スクリーン色の設定はEditorマニュアルの 「乗算色・スクリーン色」 を参照してください。

また、必要に応じたコーディングを行うことでSDKから乗算色・スクリーン色を操作し、以下のような動作も可能になります。

- インタラクティブに乗算色・スクリーン色を適用する
- Cubism Editor上で設定していない乗算色・スクリーン色を適用する
- Cubism Editor上で設定した乗算色・スクリーン色を無効にする

以降はその手順の説明になります。

処理手順

以下の流れで処理を行います。

- モデルの配置
- 乗算色・スクリーン色の上書きフラグ設定
- 乗算色・スクリーン色の設定

モデルの配置

任意のシーンに乗算色・スクリーン色を設定したいモデルを配置してください。

乗算色・スクリーン色の上書きフラグ設定

乗算色・スクリーン色の上書きフラグを true にします。 デフォルトでは false となっており、モデルからの色情報を利用するようになっています。

上書きフラグは [CubismRenderController] が持つモデル全体にかかるフラグと [CubismRenderer] の持つ各Drawableオブジェクトにかかるフラグの2種類があります。

コードの例は以下の通りです。 スクリプトをモデルのルートオブジェクトにコンポーネントとしてアタッチしている状態を想定しています。

```
CubismRenderController renderController
=Component.getComponent(CubismRenderController);

renderController?.overwriteFlagForModelMultiplyColors = true;
renderController?.overwriteFlagForModelScreenColors = true;

renderController.renderers[0].overwriteFlagForDrawableMultiplyColors = true;
renderController.renderers[0].overwriteFlagForDrawableScreenColor = true;
```

モデル全体にかかる上書きフラグが有効となった場合は、個別のDrawableにかかる上書きフラグが無効であった場合でもSDKから乗算色・スクリーン色を操作することが可能となっています。

[CubismRenderController] はモデルのルートオブジェクトにコンポーネントとして追加されており、getComponentを利用することで取得する事が可能です。 [CubismRenderController] は合わせてモデルの各Drawableオブジェクトの [CubismRenderer] を配列に確保しており、[CubismRenderController] から各 [CubismRenderer] を参照することが可能です。

乗算色・スクリーン色の設定

乗算色・スクリーン色を定義してモデルに設定します。 下記のコードでは全てのDrawableに対して、乗算色に赤、スクリーン色に緑を設定する場合の設定値です。 設定色は各 [CubismRenderer] にそれぞれmath.Color型で格納されています。 以下の例ではRGBAで設定していますが、Aは乗算色・スクリーン色の計算には利用されません。

```
const multiplyColor = new math.Color(1.0, 0.5, 0.5, 1.0);
const screenColor = new math.Color(0.0, 0.5, 0.0, 1.0);

for (let i = 0; i < renderController.renderers.length; i++)
{
    // MultiplyColor
    renderController.renderers[i].multiplyColor = multiplyColor;
    // ScreenColor
    renderController.renderers[i].screenColor = screenColor;
}
```

乗算色・スクリーン色適用前

乗算色に赤、スクリーン色に緑を適用後

モデルの乗算色・スクリーン色の処理は色情報に変更があった場合に全てのDrawableで更新が入ります。

Tips

今回は全てのDrawableに同じ乗算色・スクリーン色を設定していますが、各Drawableごとに異なる乗算色・スクリーン色を設定することも可能です。 乗算色・スクリーン色を無効にする場合は

乗算色に(1.0, 1.0, 1.0, 1.0) スクリーン色に(0.0, 0.0, 0.0, 1.0)

を設定することで可能となります。

その他の関連関数・処理

モデル側からの乗算色・スクリーン色の更新の通知を受け取る

モデルのパラメータに乗算色・スクリーン色の変更が結びつけられている場合、SDK側からの操作ではなく、モデルがアニメーションした際などにモデル側から乗算色・スクリーン色が変更される事があります。

この時に乗算色・スクリーン色が変更されたことを受け取る事が出来るプロパティ、 isBlendColorDirty が [CubismDynamicDrawableData] に実装されています。

このプロパティは乗算色、もしくは、スクリーン色のいずれかがモデル側で変更された際に立つフラグとなっており、乗算色とスクリーン色のどちらが変更されたかは判別しません。

[CubismDynamicDrawableData] のデータを扱うには[CubismModel] のイベント onDynamicDrawableData を利用する必要があります。

関数の登録

```
let model: CubismModel;

// Register listener.
model.onDynamicDrawableData += onDynamicDrawableData;
```

登録する関数の例

```
private onDynamicDrawableData(sender: CubismModel, data:
CubismDynamicDrawableData[]): void {
    for (let i = 0; i < data.length; i++)
    {
        if (data[i].isBlendColorDirty)
        {
            console.log(i + ": Did Changed.");
        }
    }
}
```

Eyeblink

概要

EyeBlinkは、まばたき用パラメータの現在の値に対して開閉状態の値を適用する機能です。 モデルにまばたきのパラメータを設定する方法については こちら をご覧ください。

まばたき用のパラメータの指定は、モデル自体に設定しておく他、Cocos Creator上でユーザが任意に指定することができます。

Cubism SDK for Cocos Creator におけるEyeBlinkは3種類の要素によって構成されています。

1. パラメータ指定用のコンポーネント
2. 各パラメータに値を適用するコンポーネント
3. 2で適用する値の操作

1. パラメータ指定用のコンポーネント

EyeBlinkに使用するパラメータを指定するには、CubismEyeBlinkParameterを使用します。

CubismEyeBlinkParameterはComponentを継承したコンポーネントで、 [Prefabのルート]/Parameters/ 以下に配置されたGameObjectにアタッチして使用します。 これがアタッチされたGameObjectと同じIDのパラメータをまばたき用のパラメータとして扱います。

モデル自体にまばたき用のパラメータが設定されている場合、インポートの際にそのパラメータのGameObjectにCubismEyeBlinkParameterがアタッチされます。

CubismEyeBlinkParameterは参照先を取得するためのマーカーとして使用しているので、内部では何も処理を行っておらず、データも持っていません。

2. 各パラメータに値を適用するコンポーネント

各パラメータに開閉の値を適用するには、CubismEyeBlinkControllerを使用します。 これはComponentを継承したコンポーネントで、使用する際はCubismのPrefabのルートにアタッチします。

初期化時に、Prefabにアタッチされたすべてのの参照を取得します。 実行中にまばたき用のパラメータを追加／削除した場合、CubismEyeBlinkController.refresh()を呼んで参照を取得し直します。

```
  /** Refreshes controller. Call this method after adding and/or removing <see
cref="CubismEyeBlinkParameter"/>s. */
  public refresh(): void {
    const model = CoreComponentExtensionMethods.findCubismModel(this);

    // Fail silently...
    if (model == null) {
      return;
    }
```

```
    // Cache destinations.
    const tags =
      model.parameters != null
        ? ComponentExtensionMethods.getComponentsMany(model.parameters,
CubismEyeBlinkParameter)
        : null;

    this.destinations = new Array(tags?.length ?? 0);

    for (var i = 0; i < this.destinations.length; i++) {
      this.destinations[i] = tags![i].getComponent(CubismParameter);
    }

    // Get cubism update controller.
    this.hasUpdateController = this.getComponent(CubismUpdateController) != null;
  }

...

  /** Called by Cocos Creator. Makes sure cache is initialized. */
  protected start(): void {
    // Initialize cache.
    this.refresh();
  }
```

CubismEyeBlinkControllerは、毎フレームのonLateUpdate()のタイミングで、CubismEyeBlinkParameterでマーキングされたパラメータに対してCubismEyeBlinkController.eyeOpeningの値を適用します。

```
// Apply value.
CubismParameterExtensionMethods.blendToValueArray(
  this.destinations,
  this.blendMode,
  this.eyeOpening
);
```

EyeOpeningに設定する値は0.0f～1.0fの範囲です。 CubismEyeBlinkControllerはこの値を対象のパラメータに対して、CubismEyeBlinkController.blendModeで設定された計算方式で適用します。

このEyeOpeningの値を外から操作することで、モデルの目を開閉させることができます。

```
/** Opening of the eyes. */
@property({ type: CCFloat, visible: true, serializable: true, range: [0.0, 1.0,
0.01] })
public eyeOpening: number = 1.0;
```

3. 2で適用する値の操作

「2. 各パラメータに値を適用するコンポーネント」で説明した通り、CubismEyeBlinkController.eyeOpeningの値を操作することで、まばたき用のパラメータに値を適用できます。 Cubism SDK for Cocos Creator には、以下の2種類の方法でこの値を操作することができます。

モーションによって値を操作

コンポーネントによって値を操作 また、ユーザ側でこの値を操作する処理を実装していただくことで、まばたきの速度やタイミング等を独自にカスタマイズすることができます。

Tips

CubismEyeBlinkController.eyeOpeningを操作するコンポーネントの実行順がCubismEyeBlinkControllerよりも後である場合、意図した動作にならない可能性があります。 もし動作に問題が生じた、ユーザ側で明示的にコンポーネントの実行順を制御することで回避が可能です。 Cubism SDK for Cocos Creator では各コンポーネントの実行順をのCubismUpdateControllerで制御しているので、これを利用することもできます。

また、上記2種類の設定方法はそれぞれ異なるタイミングで同じ値を操作しているため、工夫無しで一つのモデルに両方を共存させることは難しくなっております。

モーションによって値を操作

まばたき用のパラメータを設定したモデルを使用してCubismのAnimatorでモーションを作成する場合、まばたき用のカーブを設定することが可能です。

まばたき用のカーブが設定された.motion3.jsonをプロジェクトにインポートした場合、AnimationClipにはCubismEyeBlinkController.eyeOpeningの値を対象としてそのカーブが生成されます。 そのため、そのAnimationClipをAnimatorコンポーネントなどで再生することでCubismEyeBlinkController.eyeOpeningの値が操作されます。

コンポーネントによって値を操作

Cubism SDK for Cocos Creator では、CubismAutoEyeBlinkInput コンポーネントによってもまばたき用の値を操作することができます。

CubismAutoEyeBlinkInputは、Inspectorから設定した速度や間隔、間隔に加えるランダムな揺らぎの幅からまばたき用の値を算出して設定します。

```
protected lateUpdate(dt: number): void {
  // Fail silently.
  if (this.controller == null) {
    return;
  }

  // Wait for time until blink.
  if (this.currentPhase == Phase.Idling) {
    this.t -= dt;

    if (this.t < 0) {
      this.t = Math.PI * -0.5;
      this.lastValue = 1;
```

```
        this.currentPhase = Phase.ClosingEyes;
      } else {
        return;
      }
    }

    // Evaluate eye blinking.
    this.t += dt * this.timescale;
    let value = Math.abs(Math.sin(this.t));

    if (this.currentPhase == Phase.ClosingEyes && value > this.lastValue) {
      this.currentPhase = Phase.OpeningEyes;
    } else if (this.currentPhase == Phase.OpeningEyes && value < this.lastValue) {
      value = 1;
      this.currentPhase = Phase.Idling;
      const range = this.maximumDeviation * 2;
      this.t = this.mean + random() * range - this.maximumDeviation;
    }

    this.controller.eyeOpening = value;
    this.lastValue = value;
  }
```

CubismAutoEyeBlinkInputには3つの設定項目があります。

- mean
- maximumDeviation
- timescale

```
/** Mean time between eye blinks in seconds. */
@property({ type: CCFloat, serializable: true, range: [1.0, 10.0, 0.001] })
public mean: number = 2.5;

/** Maximum deviation from {@link mean} in seconds. */
@property({ type: CCFloat, serializable: true, range: [0.5, 5.0, 0.001] })
public maximumDeviation: number = 2.0;

/** Timescale. */
@property({ type: CCFloat, serializable: true, range: [1.0, 20.0, 0.001] })
public timescale: number = 10.0;
```

- mean まばたきを行うまでの時間を設定します。
  単位は秒になります。
  実際は、この値に Maximum Deviationによる誤差を足した時間が使われます。

- maximumDeviation Meanに設定した時間に加えるランダムな揺らぎの幅を設定します。
  これらは以下のように値を計算されます。

```
const range = this.maximumDeviation * 2;
this.t = this.mean + random() * range - this.maximumDeviation;
```

- timescale まばたきを行う速度になります。
  前フレームからの経過時間に対して乗算されます。

HarmonicMotion

概要

HarmonicMotionは、指定したパラメータの値を周期的に反復させる機能です。 主に呼吸のように常に動作し続けるものに対して使用します。 HarmonicMotionの設定方法については チュートリアル - パラメータを周期的に動作させる方法 をご覧ください。

Cubism SDK for Cocos CreatorにおけるHarmonicMotionは2つの要素によって構成されています。

1. 動作させるパラメータ指定用のコンポーネント
2. 各パラメータの値を操作するコンポーネント

1. 動作させるパラメータ指定用のコンポーネント

HarmonicMotionに使用するパラメータを指定するには、 **CubismHarmonicMotionParameter** を使用します。

CubismHarmonicMotionParameterはComponentを継承したコンポーネントで、 [Prefabのルート]/Parameters/ 以下に配置されたNodeにアタッチして使用します。 これがアタッチされたNodeと同じIDのパラメータの値を周期的に動作させます。

CubismHarmonicMotionParameterには、5つの設定項目があります。

```
  /** Timescale channel. */
  @property({ type: CCInteger, serializable: true, visible: true })
  public channel: number = 0;

  /** Motion direction. */
  @property({ type: Enum(CubismHarmonicMotionDirection), serializable: true,
visible: true })
  public direction: CubismHarmonicMotionDirection =
CubismHarmonicMotionDirection.Left;

  /**
   * Normalized origin of motion.
   * The actual origin used for evaluating the motion depends limits of the {@link
CubismParameter}.
   */
  @property({ type: CCFloat, slide: true, range: [0.0, 1.0, 0.01], serializable:
true, visible: true, })
  public normalizedOrigin: number = 0.5;

  /**
   * Normalized range of motion.
   * The actual origin used for evaluating the motion depends limits of the {@link
CubismParameter}.
   */
  @property({ type: CCFloat, slide: true, range: [0.0, 1.0, 0.01], serializable:
true, visible: true, })
  public normalizedRange: number = 0.5;
```

```
  /** Duration of one motion cycle in seconds. */
  @property({ type: CCFloat, slide: true, range: [0.01, 10.0, 0.01], serializable:
true, visible: true, })
  public duration: number = 3.0;
```

- channel
  CubismHarmonicMotionControllerで設定された、正弦波の周期の倍率を指定します。
  HarmonicMotionでは、一つのモデルに対して複数の周期を設定することができ、
  CubismHarmonicMotionControllerで設定することができます。
  ここには、CubismHarmonicMotionController.channelTimescalesのインデックスを設定します。

- direction
  パラメータの中心を基準に、どの範囲で周期的に動作させるかを設定します。
  設定項目は以下の3つです。

  - Left : パラメータの中心から左半分の範囲だけで動作します。
  - Right : パラメータの中心から右半分の範囲だけで動作します。
  - Centric : パラメータの全体で動作します。

- normalizedOrigin
  directionで基準にするパラメータの中心を設定します。
  そのパラメータの最小値を0、最大値を1としたときの値を中心に設定します。

- normalizedRange
  normalizedOriginで設定された値を中心から値を動作させる振幅を設定します。
  そのパラメータの最小値を0、最大値を1としたときの、中心からの移動距離を設定します。
  この値は、normalizedOriginで設定された中心の位置からパラメータの最小値または最大値までの範囲だけ設定できます。

- duration
  パラメータの周期を調整します。

```
  /** Evaluates the parameter. */
  public evaluate(): number {
    // Lazily initialize.
    if (!this.isInitialized) {
      this.initialize();
    }

    // Restore origin and range.
    let origin = this.minimumValue + this.normalizedOrigin * this.valueRange;
    let range = this.normalizedRange * this.valueRange;

    // Clamp the range so that it stays within the limits.
    const outputArray = this.clamp(origin, range);

    const originIndex = 0;
    const rangeIndex = 1;
    origin = outputArray[originIndex];
```

```
      range = outputArray[rangeIndex];

      // Return result.
      return origin + range * Math.sin((this.t * (2 * Math.PI)) / this.duration);
    }

    /**
     * Clamp origin and range based on {@link direction}.
     * @param origin Origin to clamp.
     * @param range Range to clamp.
     * @returns
     */
    private clamp(origin: number, range: number): [number, number] {
      switch (this.direction) {
        case CubismHarmonicMotionDirection.Left: {
          if (origin - range >= this.minimumValue) {
            range /= 2;
            origin -= range;
          } else {
            range = (origin - this.minimumValue) / 2.0;
            origin = this.minimumValue + range;
            this.normalizedRange = (range * 2.0) / this.valueRange;
          }
          break;
        }
        case CubismHarmonicMotionDirection.Right: {
          if (origin + range <= this.maximumValue) {
            range /= 2.0;
            origin += range;
          } else {
            range = (this.maximumValue - origin) / 2.0;
            origin = this.maximumValue - range;
            this.normalizedRange = (range * 2.0) / this.valueRange;
          }
          break;
        }
        case CubismHarmonicMotionDirection.Centric:
          break;
        default: {
          const neverCheck: never = this.direction;
          break;
        }
      }

      // Clamp both range and NormalizedRange.
      if (origin - range < this.minimumValue) {
        range = origin - this.minimumValue;
        this.normalizedRange = range / this.valueRange;
      } else if (origin + range > this.maximumValue) {
        range = this.maximumValue - origin;
        this.normalizedRange = range / this.valueRange;
      }
```

```
    return [origin, range];
  }
```

また、CubismHarmonicMotionParameterは、CubismHarmonicMotionControllerが参照先を取得するためのマーカーとしても使用しています。

2. 各パラメータの値を操作するコンポーネント

各パラメータに開閉の値を適用するには、 **CubismHarmonicMotionController** を使用します。 これはComponentを継承したコンポーネントで、使用する際はCubismのPrefabのルートにアタッチします。

初期化時に、Prefabにアタッチされたすべてのパラメータの参照を取得します。 実行中に周期的に値を動作させるパラメータを追加／削除した場合、CubismHarmonicMotionController.refresh()を呼んで参照を取得し直します。

```
  /** Refreshes the controller. Call this method after adding and/or removing
{@link CubismHarmonicMotionParameter}. */
  public refresh() {
    const model = CoreComponentExtensionMethods.findCubismModel(this);

    if (model == null || model.parameters == null) {
      return;
    }

    // Catch sources and destinations.
    this.sources = FrameworkComponentExtensionMethods.getComponentsMany(
      model.parameters,
      CubismHarmonicMotionParameter
    );
    this.destinations = new Array<CubismParameter>(this.sources.length);

    for (let i = 0; i < this.sources.length; ++i) {
      this.destinations[i] = this.sources[i].getComponent(CubismParameter);
    }

    // Get cubism update controller.
    this.hasUpdateController = this.getComponent(CubismUpdateController) != null;
  }

  ...

  /** Called by Cocos Creator. Makes sure cache is initialized. */
  protected start() {
    // Initialize cache.
    this.refresh();
  }
```

CubismHarmonicMotionControllerは、毎フレームのlateUpdate()のタイミングで、CubismHarmonicMotionParameterでマーキングされたパラメータに対して算出された値を適用します。

```
  /** Called by cubism update controller. Updates controller. */
  protected onLateUpdate(deltaTime: number) {
    // Return if it is not valid or there's nothing to update.
    if (!this.enabled || this.sources == null) {
      return;
    }

    // Update sources and destinations.
    for (let i = 0; i < this.sources.length; ++i) {
      this.sources[i].play(this.channelTimescales);

      CubismParameterExtensionMethods.blendToValue(
        this.destinations[i],
        this.blendMode,
        this.sources[i].evaluate()
      );
    }
  }

  ...

  /** Called by Cocos Creator. Updates controller. */
  protected lateUpdate(deltaTime: number) {
    if (!this.hasUpdateController) {
      this.onLateUpdate(deltaTime);
    }
  }
```

CubismHarmonicMotionControllerには、2つの設定項目があります。

```
  /** Blend mode. */
  @property({ type: Enum(CubismParameterBlendMode), serializable: true, visible:
true })
  public blendMode: CubismParameterBlendMode = CubismParameterBlendMode.Additive;

  /** The timescales for each channel. */
  @property({ type: [CCFloat], serializable: true, visible: true })
  public channelTimescales: number[] = [];
```

- blendMode
  パラメータに値を適用する際のブレンドモードです。
  設定できる値は以下の3つです。
  - Override : 現在の値を上書きします。
  - Additive : 現在の値に加算します。
  - Multiply : 現在の値に乗算します。
- channelTimescales
  正弦波の周期を設定します。 周期は複数設定できます。

Json

概要

Live2D Cubismでは、ランタイム用データのいくつかをjson形式で扱っています。 Cubism SDK for Cocos Creatorには、それらのjson形式のファイルをパース、及びインスタンス化を行うクラスが同梱されています。

SDK同梱のアセットインポーターによって、それらがインポートされたときにパース、インスタンス化していますが、ユーザがランタイムで読み込ませることも可能です。

- CubismModel3Json
- CubismMotion3Json
- CubismUserData3Json
- CubismPhysics3Json
- CubismExp3Json
- CubismPose3Json
- CubismDisplayInfo3Json

CubismModel3Json

.model3.jsonのパーサです。

.model3.json内に記述されたモーションや表情など、各種他のjsonのパスを取得することができます。

CocosCreator 既定の resources にある JsonAsset をパースする場合は CubismModel3Json.loadAtPath()を使用します。

```
const model3Json = await CubismModel3Json.loadAtPath("path/to/file");
```

CocosCreator 既定の resources 以外にある.model3.json をパースする場合は任意の方法で文字列を取得しJSON.parse()で処理したオブジェクトを CubismModel3Json.loadFromJson() に渡すことで行えます。

```
const json = JSON.parse(jsonSourceText);
const model3Json = CubismModel3Json.loadFromJson(json);
```

パースしたデータからは、.model3.json に記述された各種ファイルの相対パスを取得することが可能です。

```
// .moc3
const mocPath = modelj.fileReferences.moc;

// textures
for(let i = 0; i < modelj.fileReferences.textures.length; i++) {
  const texturePath = modelj.fileReferences.textures[i];
}
```

```
// .physics3.json
const physicsPath = model3Json.fileReferences.physics;

// .userdata3.json
const userdataPath = model3Json.fileReferences.userData;

...
```

基本的に.model3.json の階層と同じ構造でパスを取得することができますが、.model3.json に記述された表情用データの参照につきましては独自の構造となっております。

Cocos Creator では Editor Platform 上では アセット収集ができないため CubismModel3Json.toModel() を 使用してPrefabを生成することはできません。

独自のワークフローでPrefabを作成する必要がある場合は、ユーザー側で実装していただく必要があります。

またCubism SDK for Cocos Creator に実装されている処理は、Cocos Creator の エディタ上からプロジェクト内のアセットを読み込むことを前提としております。 ランタイムで AssetBundle などから読み込む場合、読み込み処理をユーザ側で実装して頂く必要がございます。

CubismMotion3Json

.motion3.jsonのパーサです。

.motion3.json内に記述されたカーブ情報からAnimationClipを生成することができます。

CubismMotion3Json で.motion3.json をパースするには、CubismMotion3Json.loadFrom() を使用します。

```
const json = resources.load<TextAsset>("path/to/file").text;

const motion3Json = CubismMotion3Json.loadFrom(json);
```

パースした.motion3.json から AnimationClip を生成するには、CubismMotion3Json.toAnimationClip() を使用します。

```
// Initialize
const animationClip = motion3Json.toAnimationClipA();

// Original Workflow
const animationClipForOW = motion3Json.toAnimationClipA(
  true,
  true,
  true,
  pose3Json
);
```

CubismUserData3Json

.userdata3.jsonのパーサです。

.userdata3.json内に記述された情報からモデルのアートメッシュにユーザデータを適用することができます。

CubismUserData3Json で.userdata3.json をパースするには、CubismUserData3Json.loadFrom()を使用します。

```
const json = resources.load<TextAsset>("path/to/file").text;

const userData3Json = CubismUserData3Json.loadFrom(json);
```

パースした.userdata3.json からユーザデータを取得するには、CubismUserData3Json.toBodyArray()を使用します。

```
const drawableBodies = userData3Json.toBodyArray(
  CubismUserDataTargetType.ArtMesh
);
```

CubismPhysics3Json

.physics3.jsonのパーサです。

.physics3.json内に記述された物理演算設定をCocos Creatorで使用できるよう変換することができます。

CubismPhysics3Json で.physics3.json をパースするには、CubismPhysics3Json.loadFrom()を使用します。

```
const json = resources.load<TextAsset>("path/to/file").text;

const physics3Json = CubismPhysics3Json.loadFrom(json);
```

パースした.physics3.json から Cocos Creator で扱う形式に物理演算設定を変換するには、CubismPhysics3Json.ToRig() を使用します。

```
const cubismPhysicsController = modelNode.getComponent<
  CubismPhysicsController
>();
cubismPhysicsController.initialize(physics3Json.toRig());
```

CubismExp3Json

.exp3.jsonのパーサです。

.exp3.json 内に記述された表情差分の情報を Cocos Creator 上で扱う形式に変換することができます。

CubismExp3Json で.exp3.json をパースするには、CubismExp3Json.loadFrom()を使用します。

```
const json = resources.load<TextAsset>("path/to/file").text;

const exp3Json = CubismExp3Json.loadFrom(json);
```

CubismPose3Json

.pose3.jsonのパーサです。

.pose3.json内に記述された情報から、Cocos Creator上でパーツの表示状態を制御する設定を取得できます。

.pose3.json 内に記述された情報から、Cocos Creator 上でパーツの表示状態を制御する設定を取得できます。

Pose機能の詳細につきましては こちら をご覧ください。 Cubism SDK for Cocos Creator においては、Pose は生成される AnimationClip のカーブを加工するのに使用されます。

CubismPose3Json で.pose3.json をパースするには、CubismPose3Json.loadFrom()を使用します。

```
const json = resources.load<TextAsset>("path/to/file").text;

const pose3Json = CubismPose3Json.loadFrom(json);
```

CubismDisplayInfo3Json

.cdi3.jsonのパーサです。 .cdi3.jsonには、Cubismエディタで設定されたパラメータやパーツ、パラメータグループの名前と、それらと対になる各IDが記述されています。 .cdi3.jsonが存在しないモデルの場合はIDが表示されます。

Cubism SDK for Cocos Creator においては、Inspector ウィンドウに表示されるパラメータやパーツの名前表示に使用しています。 CubismDisplayInfo3Json で.cdi3.json をパースするには、CubismDisplayInfo3Json.loadFrom()を使用します。

```
const json = resources.load<TextAsset>("path/to/file").text;

const cdi3Json = CubismDisplayInfo3Json.loadFrom(json);
```

LookAt

概要

LookAtは、任意のパラメータを特定の座標に追従するよう値を操作する機能です。 追従させる座標をユーザ側でカスタマイズすることで、Cubismのモデルを特定のGameObjectなどに追従させることが可能です。
LookAtの使用方法については チュートリアル - 視線追従の設定 をご覧ください。

Cubism SDK for Cocos Creator におけるLookAtは3種類の要素によって構成されています。

1. 追従させるパラメータ指定用のコンポーネント
2. 各パラメータに値を適用するコンポーネント
3. 2で適用する値の操作

1. 追従させるパラメータ指定用のコンポーネント

LookAtで追従させるパラメータを指定するには、CubismLookParameterを使用します。

CubismLookParameterは[Prefabのルート]/Parameters/ 以下に配置されたGameObjectにアタッチして使用します。 これがアタッチされたGameObjectと同じIDのパラメータを視線追従用のパラメータとして扱います。

CubismLookParameterには、Axis , Factor の2つの設定項目があります。

```
/** Look axis. */
@property({ type: Enum(CubismLookAxis), serializable: true, visible: true })
public axis: CubismLookAxis = CubismLookAxis.X;

/** Factor. */
@property({ type: CCFloat, serializable: true, visible: true })
public factor: number = 0;
```

- Axis

入力された座標のどの軸の値を使用するかを設定します。 設定できる項目は、X, Y, Zの3つです。

```
/** Look axis. */
enum CubismLookAxis {
  /** X axis. */
  X,
  /** Y axis. */
  Y,
  /** Z axis. */
  Z,
}
export default CubismLookAxis;
```

- Factor

適用される値への補正値を設定します。 入力される座標は、適用される際には-1.0fから1.0fの範囲に加工された値になります。 Factorに設定する値は、入力された値を各パラメータの最小値と最大値に合うように乗算する倍率になります。

```
public tickAndEvaluate(targetOffset: math.Vec3): number {
  const result =
    this.axis == CubismLookAxis.X
      ? targetOffset.x
      : this.axis == CubismLookAxis.Z
      ? targetOffset.z
      : targetOffset.y;
  return result * this.factor;
}
```

また、CubismLookParameterは、CubismLookControllerが参照先を取得するためのマーカーとしても使用しています。

2. 各パラメータに値を適用するコンポーネント

各パラメータに視線追従を適用するには、CubismLookControllerを使用します。 CubismLookControllerを使用する際はCubismのPrefabのルートにアタッチします。

CubismLookControllerの初期化時に、Prefabにアタッチされたすべてのパラメータの参照を取得します。 実行中に視線追従用のパラメータを追加／削除した際にはCubismLookController.Refresh()を呼んで参照を取得し直します。

```
/** Refreshes the controller. Call this method after adding and/or removing {@link
CubismLookParameter}s. */
public refresh(): void {
  const model = CoreComponentExtensionMethods.findCubismModel(this);
  if (model == null) {
    return;
  }
  if (model.parameters == null) {
    return;
  }

  // Catch sources and destinations.

  this.sources = ComponentExtensionMethods.getComponentsMany(
    model.parameters,
    CubismLookParameter
  );
  this.destinations = new Array<CubismParameter>(this.sources.length);

  for (let i = 0; i < this.sources.length; i++) {
    this.destinations[i] = this.sources[i].getComponent(CubismParameter);
```

```
    }

    // Get cubism update controller.
    this.hasUpdateController = this.getComponent(CubismUpdateController) != null;
  }

  ...

  /** Called by Cocos Creator. Makes sure cache is initialized. */
  protected start(): void {
    // Default center if necessary.
    if (this.center == null) {
      this.center = this.node;
    }

    // Initialize cache.
    this.refresh();
  }
```

CubismLookControllerは、毎フレームのLateUpdate()のタイミングで、CubismLookParameterでマーキングされたパラメータに対してCubismLookController.Targetに設定したオブジェクトが返す座標を適用します。

```
// Update position.
let position = this.lastPosition;

const inverseTransformPoint = this.node.inverseTransformPoint(
  new math.Vec3(),
  target.getPosition()
);
this.goalPosition = math.Vec3.subtract(
  new math.Vec3(),
  inverseTransformPoint,
  this.center.position
);
if (position != this.goalPosition) {
  const temp = MathExtensions.Vec3.smoothDamp(
    position,
    this.goalPosition,
    this.velocityBuffer,
    this.damping
  );
  position = temp[0];
  this.velocityBuffer = temp[1];
}

// Update sources and destinations.
for (let i = 0; i < this.destinations.length; i++) {
  CubismParameterExtensionMethods.blendToValue(
    this.destinations[i],
    this.blendMode,
    this.sources[i].tickAndEvaluate(position)
```

```
    );
  }

  // Store position.
  this.lastPosition = position;
```

3. 2で適用する値の操作

「2. 各パラメータに値を適用するコンポーネント」で説明した通り、CubismLookControllerが視線追従用のパラメータに適用する座標はCubismLookController.Targetに設定したオブジェクトが返ります。

これはICubismLookTargetインターフェースを実装したもので、ユーザ側でこちらを実装していただくことでモデルを任意の座標に追従させることができます。

```
/** Target to look at. */
interface ICubismLookTarget {
  readonly [ICubismLookTarget.SYMBOL]: typeof ICubismLookTarget.SYMBOL;

  /**
   * Gets the position of the target.
   *
   * @returns The position of the target in world space.
   */
  getPosition(): math.Vec3;

  /**
   * Gets whether the target is active.
   *
   * @returns true if the target is active; false otherwise.
   */
  isActive(): boolean;
}
export default ICubismLookTarget;
```

- GetPosition()

追従する座標を返します。 ここで返す座標はワールド座標として扱われます。

- IsActive()

追従が有効かどうかを返します。 trueが返されているときだけ追従します。

SDKにはICubismLookTargetを実装した例としてCubismLookTargetBehaviourが同梱されています。
CubismLookTargetBehaviourは、それがアタッチされたGameObjectの座標を返すサンプルになります。

```
/** Straight-forward {@link ICubismLookTarget} {@link Component}. */
@ccclass('CubismLookTargetBehaviour')
export default class CubismLookTargetBehaviour extends Component implements
ICubismLookTarget {
```

```
  readonly [ICubismLookTarget.SYMBOL]: typeof ICubismLookTarget.SYMBOL =
ICubismLookTarget.SYMBOL;

  //#region Implementation of ICubismLookTarget

  /**
   * Gets the position of the target.
   * @returns The position of the target in world space.
   */
  public getPosition(): Readonly<math.Vec3> {
    return this.node.worldPosition;
  }

  /**
   * Gets whether the target is active.
   * @returns true if the target is active; false otherwise.
   */
  public isActive(): boolean {
    return this.enabledInHierarchy;
  }

  //#endregion
}
```

MouthMovement

概要

MouthMovementは、リップシンク用パラメータの現在の値に対して開閉状態の値を適用する機能です。 モーションに設定されたリップシンク用のカーブや再生している音声ファイルからリアルタイムにサンプリングした値などをモデルに適用することが可能です。 モデルにリップシンクのパラメータを設定する方法については チュートリアル - リップシンクの設定 をご覧ください。

MouthMovementで設定するものは口の開閉状態のみです。 口の形状を母音に合わせるというような操作をすることはできません。

Cocos Creator上でリップシンク用のパラメータを指定するには、Cubismエディタでモデルに設定しておく他、Cocos Creator上でユーザが任意に指定することができます。

Cubism SDK for Cocos Creator におけるMouthMovementは3種類の要素によって構成されています。

1. パラメータ指定用のコンポーネント
2. 各パラメータに値を適用するコンポーネント
3. 2で適用する値の操作

1. パラメータ指定用のコンポーネント

MouthMovementに使用するパラメータを指定するには、CubismMouthParameterを使用します。

CubismMouthParameterはComponentを継承したコンポーネントで、 [Prefabのルート]/Parameters/ 以下に配置されたNodeにアタッチして使用します。 これがアタッチされたNodeと同じIDのパラメータをリップシンク用のパラメータとして扱います。

モデル自体にリップシンク用のパラメータが設定されている場合、インポートの際にそのパラメータのNodeにCubismMouthParameterがアタッチされます。

CubismMouthParameterは参照先を取得するためのマーカーとして使用しているので、内部では何も処理を行っておらず、データも持っていません。

2. 各パラメータに値を適用するコンポーネント

各パラメータに開閉の値を適用するには、CubismMouthControllerを使用します。 これはComponentを継承したコンポーネントで、使用する際はCubismのPrefabのルートにアタッチします。

初期化時に、Prefabにアタッチされたすべてのパラメータの参照を取得します。 実行中にまばたき用のパラメータを追加／削除した場合、CubismMouthController.refresh()を呼んで参照を取得し直します。

```
    public refresh() {
      const model = CoreComponentExtensionMethods.findCubismModel(this);

      // Fail silently...
      if (model == null || model.parameters == null) {
        return;
```

```
    }

    // Cache destinations.
    const tags = ComponentExtensionMethods.getComponentsMany(
      model.parameters,
      CubismMouthParameter
    );

    this.destinations = new Array(tags.length);

    for (let i = 0; i < tags.length; ++i) {
      this.destinations[i] = tags[i].getComponent(CubismParameter);
    }

    // Get cubism update controller.
    this.hasUpdateController = this.getComponent(CubismUpdateController) != null;
  }

...

  protected start() {
    // Initialize cache.
    this.refresh();
  }
```

CubismMouthControllerは、毎フレームのlateUpdate()のタイミングで、CubismMouthParameterでマーキングされたパラメータに対してCubismMouthController.MouthOpeningの値を適用します。

```
  protected onLateUpdate(deltaTime: number) {
    // Fail silently.
    if (!this.enabled || this.destinations == null) {
      return;
    }

    // Apply value.
    CubismParameterExtensionMethods.blendToValueArray(
      this.destinations,
      this.blendMode,
      this.mouthOpening
    );
  }
```

MouthOpeningに設定する値は0.0f～1.0fの範囲です。 CubismMouthControllerはこの値を対象のパラメータに対して、CubismMouthController.blendModeで設定された計算方式で適用します。

このMouthOpeningの値を外から操作することで、モデルの口を開閉させることができます。

```
  @property({ type: CCFloat, slide: true, range: [0.0, 1.0, 0.01] })
  public mouthOpening: number = 1.0;
```

3. 2で適用する値の操作

「2. 各パラメータに値を適用するコンポーネント」で説明した通り、CubismMouthController.mouthOpeningの値を操作することで、リップシンク用のパラメータに値を適用できます。

Cubism SDK for Cocos Creator には、以下の3種類の方法でこの値を操作することができます。

- モーションによって値を操作
- 周期的に値を操作
- AudioClipからサンプリングして値を操作

また、ユーザ側でこの値を操作する処理を実装していただくことで、リップシンクの速度やタイミング等を独自にカスタマイズすることができます。

Tips

CubismMouthController.mouthOpeningを操作するコンポーネントの実行順がCubismMouthControllerよりも後である場合、意図した動作にならない可能性があります。 もし動作に問題が生じた、ユーザ側で明示的にコンポーネントの実行順を制御することで回避が可能です。 Cubism SDK for Cocos Creator では各コンポーネントの実行順をのCubismUpdateControllerで制御しているので、これを利用することもできます。

また、上記3種類の設定方法はそれぞれが同じ値を操作しているため、工夫無しで一つのモデルに共存させることは難しくなっております。

モーションによって値を操作

まばたき用のパラメータを設定したモデルを使用してCubismのAnimatorでモーションを作成する場合、まばたき用のカーブを設定することが可能です。

まばたき用のカーブが設定された.motion3.jsonをCocos Creatorプロジェクトにインポートした場合、AnimationにはCubismMouthController.mouthOpeningの値を対象としてそのカーブが生成されます。 そのため、そのAnimationClipをAnimatorコンポーネントなどで再生することでCubismMouthController.mouthOpeningの値が操作されます。

周期的に値を操作

周期的にリップシンク用の値を操作させるには、CubismAutoMouthInputを使用します。
CubismAutoMouthInputは、正弦波でリップシンクの値を算出して設定するコンポーネントです。

CubismAutoMouthInputを使用するには、CubismのPrefabのルートにアタッチします。

CubismAutoMouthInputには1つの設定項目があります。

- Timescale
  正弦波の周期が変化します。

```
@property(CCFloat)
public Timescale: number = 10.0;
```

```
  lateUpdate(deltaTime: number) {
    // Fail silently.
    if (this.Controller == null) {
      return;
    }

    // Progress time.
    this.T += deltaTime * this.Timescale;

    // Evaluate.
    this.Controller.mouthOpening = Math.abs(Math.sin(this.T));
  }
```

オーディオからサンプリングして値を操作

Cocos Creator上で再生される音声からリップシンクの値を設定する場合はCubismAudioMouthInputを使用します。

CubismAudioMouthInputは、AudioSourceから取得した再生中の音声情報からサンプリングしてリアルタイムにリップシンクの値を生成、設定します。

CubismAudioMouthInputを使用するには、CubismのPrefabのルートにアタッチします。

```
  protected update(deltaTime: number) {
    const { audioInput, samples, target, sampleRate, gain, smoothing } = this;

    // 'Fail' silently.
    if (audioInput == null || target == null || samples == null || sampleRate ==
0) {
      return;
    }
    const { trunc, sqrt } = Math;

    const { currentTime } = audioInput;
    const pos = trunc(currentTime * this.sampleRate);
    let length = 256;
    switch (this.samplingQuality) {
      case CubismAudioSamplingQuality.veryHigh:
        length = 256;
        break;
      case CubismAudioSamplingQuality.maximum:
        length = 512;
        break;
      default:
        length = 256;
        break;
    }
```

```
    // Sample audio.
    let total = 0.0;

    for (let i = 0; i < length; i++) {
      const sample = samples.getData((pos + i) % samples.length);
      total += sample * sample;
    }

    // Compute root mean square over samples.
    let rms = sqrt(total / length) * gain;

    // Clamp root mean square.
    rms = math.clamp01(rms);

    // Smooth rms.
    const output = MathExtensions.Float.smoothDamp(
      this.lastRms,
      rms,
      this.velocityBuffer,
      smoothing * 0.1,
      undefined,
      deltaTime
    );

    rms = output[0];
    this.velocityBuffer = output[1];

    // Set rms as mouth opening and store it for next evaluation.
    target.mouthOpening = rms;

    this.lastRms = rms;
  }
```

CubismAudioMouthInputには4つの設定項目があります。

```
  @property(AudioSource)
  public audioInput: AudioSource | null = null;

  @property({ type: Enum(CubismAudioSamplingQuality) })
  public samplingQuality: CubismAudioSamplingQuality =
CubismAudioSamplingQuality.high;

  @property({ type: CCFloat, slide: true, range: [1.0, 10.0, 0.01] })
  public gain: number = 1.0;

  @property({ type: CCFloat, slide: true, range: [0.0, 1.0, 0.01] })
  public smoothing: number = 0.0;
```

- audioInput
  サンプリングするAudioSourceの参照です。

- samplingQuality
  サンプリングする音声の精度です。

- gain
  サンプリングした値の倍率です。
  1で等倍、値を大きくするほどリップシンクの値も大きくなります。

- smoothing
  サンプリングした値のスムージング量です。
  値が大きいほどリップシンクの値が滑らかに変化します。

Raycasting

概要

Raycastingは、ユーザが指定したCubismのメッシュと任意の座標とが交差するかどうかを判定する機能です。 クリック／タップされた座標や、シーン中の任意のオブジェクトと指定されたメッシュとの当たり判定を取得することが可能です。 Raycastingの使用方法については チュートリアル - 当たり判定の設定 をご覧ください。

Cubism SDK for Cocos Creator におけるRaycastingは大きく2種類の要素によって構成されています。

1. 判定させるアートメッシュ指定用のコンポーネント
2. 入力された座標と指定されたアートメッシュの当たり判定を行うコンポーネント

1. 判定させるアートメッシュ指定用のコンポーネント

当たり判定に使用するメッシュを指定するには、CubismRaycastableを使用します。

CubismRaycastableは[Prefabのルート]/Drawables/ 以下に配置されたNodeにアタッチして使用します。 これがアタッチされたNodeと同じIDのアートメッシュを当たり判定に使用します。

CubismRaycastableは、そのメッシュの当たり判定の精度を設定するパラメータを持っています。

```
@ccclass('CubismRaycastable')
export default class CubismRaycastable extends Component {
  /** The precision. */
  @property({
    type: Enum(CubismRaycastablePrecision),
    serializable: true,
    visible: true,
    readonly: false,
  })
  public precision: CubismRaycastablePrecision =
CubismRaycastablePrecision.boundingBox;
}
```

CubismRaycastable.Precisionには設定可能な値が2つあり、それぞれ以下のようになっております。

- BoundingBox
  そのメッシュを囲う、四辺が水平または垂直の矩形を当たり判定として使用します。
  メッシュの形状によってはメッシュ外の座標でも当たり判定を取りますが、Trianglesと比較してパフォーマンスに優れます。

- Triangles
  メッシュの形状を当たり判定の範囲として使用します。
  BoundingBoxと比較するとパフォーマンスに劣りますが、正確な範囲で判定を行うことができます。

```
enum CubismRaycastablePrecision {
  /** Cast against bounding box. */
  boundingBox,

  /** Cast against triangles. */
  triangles,
}
```

2. 入力された座標と指定されたアートメッシュの当たり判定を行うコンポーネント

実際の当たり判定の取得はCubismRaycasterを使用します。 CubismRaycasterを使用する際はCubismのPrefabのルートにアタッチします。

CubismRaycasterの初期化時に、Prefabにアタッチされたすべてのの参照を取得します。実行中に当たり判定用のメッシュを追加／削除した際には、CubismRaycaster.refresh()を呼んで参照を取得し直します。

```
  /** Refreshes the controller. Call this method after adding and/or removing
{@link CubismRaycastable}. */
  private refresh(): void {
    const candidates = ComponentExtensionMethods.findCubismModel(this)?.drawables
?? null;
    if (candidates == null) {
      console.error('CubismRaycaster.refresh(): candidates is null.');
      return;
    }

    // Find raycastable drawables.
    const raycastables = new Array<CubismRenderer>();
    const raycastablePrecisions = new Array<CubismRaycastablePrecision>();

    for (var i = 0; i < candidates.length; i++) {
      // Skip non-raycastables.
      if (candidates[i].getComponent(CubismRaycastable) == null) {
        continue;
      }
      const renderer = candidates[i].getComponent(CubismRenderer);
      console.assert(renderer);
      raycastables.push(renderer!);

      const raycastable = candidates[i].getComponent(CubismRaycastable);
      console.assert(raycastable);
      console.assert(raycastable!.precision);
      raycastablePrecisions.push(raycastable!.precision!);
    }

    // Cache raycastables.
    this.raycastables = raycastables;
    this.raycastablePrecisions = raycastablePrecisions;
```

```
  }

  /** Called by Cocos Creator. Makes sure cache is initialized. */
  protected start(): void {
    // Initialize cache.
    this.refresh();
  }
```

座標から当たり判定を取得するには、CubismRaycaster.raycast1()やCubismRaycaster.raycast2()を利用します。

```
  public raycast1(
    origin: Vector3,
    direction: Vector3,
    result: CubismRaycastHit[],
    maximumDistance: number = Number.POSITIVE_INFINITY
  ): number {
    return this.raycast2(
      geometry.Ray.create(origin.x, origin.y, origin.z, direction.x, direction.y,
direction.z),
      result,
      maximumDistance
    );
  }

  /**
   * Casts a ray.
   * @param ray
   * @param result  The result of the cast.
   * @param maximumDistance [Optional] The maximum distance of the ray.
   * @returns
   * true in case of a hit; false otherwise.
   *
   * The numbers of drawables had hit
   */
  public raycast2(
    ray: geometry.Ray,
    result: CubismRaycastHit[],
    maximumDistance: number = Number.POSITIVE_INFINITY
  ): number {
    // Cast ray against model plane.
    const origin = Vector3.from(ray.o);
    const direction = Vector3.from(ray.d);
    const intersectionInWorldSpace = origin.add(
      direction.multiplySingle(direction.z / origin.z)
    );
    let intersectionInLocalSpace = Vector3.from(
      this.node.inverseTransformPoint(new math.Vec3(),
intersectionInWorldSpace.toBuiltinType())
    );
    intersectionInLocalSpace = intersectionInLocalSpace.copyWith({ z: 0 });
```

```
      const distance = intersectionInWorldSpace.magnitude();
      // Return non-hits.
      if (distance > maximumDistance) {
        return 0;
      }
      // Cast against each raycastable.
      let hitCount = 0;
      console.assert(this.raycastables);
      const raycastables = this.raycastables!;
      console.assert(this.raycastablePrecisions);
      const raycastablePrecisions = this.raycastablePrecisions!;
      for (let i = 0; i < raycastables.length; i++) {
        const raycastable = raycastables[i];
        const raycastablePrecision = raycastablePrecisions[i];
        // Skip inactive raycastables.
        console.assert(raycastable.meshRenderer);
        if (!raycastable.meshRenderer!.enabled) {
          continue;
        }
        const bounds = raycastable.mesh.calculateBounds();

        // Skip non hits (bounding box)
        if (!bounds.contains(intersectionInLocalSpace)) {
          continue;
        }

        // Do detailed hit-detection against mesh if requested.
        if (raycastablePrecision == CubismRaycastablePrecision.triangles) {
          if (!this.containsInTriangles(raycastable.mesh, intersectionInLocalSpace))
{
            continue;
          }
        }

        result[hitCount] = new CubismRaycastHit({
          drawable: raycastable.getComponent(CubismDrawable),
          distance: distance,
          localPosition: intersectionInLocalSpace,
          worldPosition: intersectionInWorldSpace,
        });

        ++hitCount;

        // Exit if result buffer is full.
        if (hitCount == result.length) {
          break;
        }
      }

      return hitCount;
    }
```

CubismRaycaster.raycast()は返り値に当たり判定を取得したメッシュの数を返します。 また、引数に渡したCubismRaycastHit[]型のインスタンスに当たり判定を取得したメッシュの情報が設定されます。

```
        result[hitCount] = new CubismRaycastHit({
          drawable: raycastable.getComponent(CubismDrawable),
          distance: distance,
          localPosition: intersectionInLocalSpace,
          worldPosition: intersectionInWorldSpace,
        });

        ++hitCount;
```

CubismRaycaster.raycast()は、同座標上に複数のメッシュが重なっていた場合、最大でCubismRaycastHit[]型のインスタンスの要素の数まで取得します。 要素数以上のメッシュが重なっていた場合、超えた分のメッシュは結果が取得されません。

CubismRaycastHitは当たり判定を取得したメッシュの情報を持つクラスです。

```
    /** The hit {@link CubismDrawable} */
    public readonly drawable: CubismDrawable | null;

    /** The distance the ray traveled until it hit the {@link CubismDrawable}. */
    public readonly distance: number;

    /** The hit position local to the {@link CubismDrawable}. */
    public readonly localPosition: Vector3;

    /** The hit position in world coordinates. */
    public readonly worldPosition: Vector3;
```

- drawable
  当たり判定を取得したアートメッシュの参照です。
- distance
  指定した座標からの距離です。
  CubismRaycaster.Raycast()の引数に渡したoriginまたはray.originとDrawableのTransform.positionの直線距離です。
- localPosition
  当たり判定を取得したアートメッシュのローカル座標です。
- worldPosition
  当たり判定を取得したアートメッシュのワールド座標です。

UserData

概要

UserDataは、モデルのアートメッシュに設定されたユーザデータをCocos Creator上で可視化する機能です。ユーザデータはアートメッシュに対して任意のタグを付与できる機能で、SDK上で特殊な処理を行うアートメッシュを指定する場合などに活用できます。

ユーザデータの詳細については こちら をご覧ください。

Cocos Creator上でユーザデータを使用するには以下の手順を行います。

1. .userdata3.jsonをパース
2. アートメッシュにUserDataを設定

1. .userdata3.jsonをパース

.userdata3.jsonを、CubismUserData3Jsonを用いてパースします。 CubismUserData3Jsonについては こちら をご覧ください。

```
const userData3Json = CubismUserData3Json.loadFrom(jsonString);
```

.userdata3.jsonのパスは、.model3.jsonからもCubismModel3Jsonを用いて取得することができます。 取得できるパスは.model3.jsonからの相対パスです。

```
const userDataPath = modelJson.fileReferences.userData;

const userData3Json = await this.loadJson(this.fileReferences.userData);
```

CubismModel3Jsonの詳細については こちら をご覧ください。

パースした.userdata3.jsonから、アートメッシュのデータを取得します。

```
const drawableBodies =
userData3Json.toBodyArray(CubismUserDataTargetType.ArtMesh);
```

2. アートメッシュにUserDataを設定

.userdata3.jsonから取得したユーザデータを、Prefabのアートメッシュに適用します。

アートメッシュにユーザデータの情報を付与するには、CubismUserDataTagを使用します。

```
    const drawableBodies =
userData3Json.toBodyArray(CubismUserDataTargetType.ArtMesh);
```

```
    for (let i = 0; i < drawables.length; i++) {
      const index = model3.getBodyIndexById(drawableBodies, drawables[i].id);
      if (index >= 0) {
        const tag =
          drawables[i].getComponent(CubismUserDataTag) ??
          drawables[i].addComponent(CubismUserDataTag);
        tag!.initialize(drawableBodies[index]);
      }
    }
```

Cocos Creator For OW

Original Workflow（以下「OW」）につきましては こちら をご覧ください。

概要

OW用のコンポーネントを追加することで、全体的なSDKの使用感は変わらずにOWの表現ができるようになります。

OWの機能を確認するにはEditor付属のViewer for OWで使用します。

OWモードのインポートでは、非OWモードのインポートと比べて以下の4点が異なります。

- エディタ拡張により、Cocos Creator上にOWモード用のエディターメニューが追加されます。
    - 「Live2D/Cubism/OriginalWorkflow/ **Toggle should Import As Original Workflow** 」 こちらはOWモードを切り替えるための項目です。
      この項目を有効(Enable)にした状態でモデルデータをインポートすると、OW対応のモデルが作成されます。
    - 「Live2D/Cubism/OriginalWorkflow/Should Clear Animation Curves」br>こちらはAnimationClipのカーブをクリアするための項目です。
      この項目を無効(Disable)にした状態でモーションを再インポートすると、既存のAnimationClipに上書きして生成されます。
      有効(Enable)にしてモーションを再インポートした場合、既存のAnimationClipに設定されたAnimationCurveをすべてクリアしてからモーションを上書きします。
      この機能は「Live2D/Cubism/OriginalWorkflow/Should Import As Original Workflow」がチェックした状態のみ使用できます。
- 以下のOW用のコンポーネントがアタッチされます。
    - CubismUpdateController
    - CubismParameterStore
    - CubismPoseController
    - CubismExpressionController
    - CubismMotionFadeController

AnimationClipに設定されるOpacityのカーブがOW用に加工されます。 プロジェクトに.fade, fadeMotionList, .exp, .expressionListアセットが追加されます。

CubismParameterStore

概要

CubismParameterStoreを使用することで、CubismModelのパラメータとパーツの値を保存／復元することが出来ます。

CubismParameterStoreを使用しない場合、Expressionなどで値を加工した結果が正しいものにならないことがあります。

Expressionなどの値の操作を復元～保存の間で行った場合、値を操作した結果が保存されるため、復元された操作後の値に加算／乗算してしまい、期待通りの結果になりません。 値の操作を復元～保存の外で行う場合、値を操作する前の状態が復元されるため、その後の加算／乗算による値の操作が適正な結果となります。

該当のチュートリアル記事は チュートリアル - 操作する値の保存／復元をおこなう をご覧ください。

CubismModelのパラメータとパーツの値を保存／復元するには以下の処理を行っています。

1. 保存／復元するパラメータとパーツの参照を取得
2. パラメータとパーツの値を保存
3. パラメータとパーツの値を復元

保存／復元するパラメータとパーツの参照を取得

CubismParameterStore.onEnableでCubismModelのパラメータとパーツへの参照をキャッシュします。

```
    if (this.destinationParameters == null) {
      this.destinationParameters =
        ComponentExtensionMethods.findCubismModel(this)?.parameters ?? null;
    }

    if (this.destinationParts == null) {
      this.destinationParts =
ComponentExtensionMethods.findCubismModel(this)?.parts ?? null;
    }
```

この処理はCubismParameterStore.onEnable()で行っています。

パラメータとパーツの値を保存

モデルの現在のパラメータとパーツの値を保存します。 値を保存するタイミングはアニメーションを適用した後、Cubismの各種コンポーネントで値を操作する前に行います。

```
    // save parameters value
    if (this.destinationParameters != null && this._parameterValues == null) {
      this._parameterValues = new Array<number>
(this.destinationParameters.length);
```

```
    }

    if (this._parameterValues != null && this.destinationParameters != null) {
      for (let i = 0; i < this._parameterValues.length; ++i) {
        if (this.destinationParameters[i] != null) {
          this._parameterValues[i] = this.destinationParameters[i].value;
        }
      }
    }

    // save parts opacity
    if (this.destinationParts != null && this._partOpacities == null) {
      this._partOpacities = new Array(this.destinationParts.length);
    }

    if (this._partOpacities != null && this.destinationParts != null) {
      for (let i = 0; i < this._partOpacities.length; ++i) {
        this._partOpacities[i] = this.destinationParts[i].opacity;
      }
    }
```

この処理は`CubismParameterStore.saveParameters()`で行います。

パラメータとパーツの値を復元

保存したパラメータとパーツの値を復元します。 値を復元するタイミングは、アニメーションを適用する前に行います。

```
    // restore parameters value
    if (this._parameterValues != null && this.destinationParameters != null) {
      for (let i = 0; i < this._parameterValues.length; ++i) {
        this.destinationParameters[i].value = this._parameterValues[i];
      }
    }

    // restore parts opacity
    if (this._partOpacities != null && this.destinationParts != null) {
      for (let i = 0; i < this._partOpacities.length; ++i) {
        this.destinationParts[i].opacity = this._partOpacities[i];
      }
    }
```

この処理は`CubismParameterStore.restoreParameters()`で行っています。

CubismUpdateController

概要

CubismUpdateControllerをPrefabにアタッチすることで、Cubism SDK for Cocos Creatorの各コンポーネントの実行順を制御することができます。

OW方式では各コンポーネントの実行順序が重要となるため、CubismUpdateControllerを使用して仕様に沿うことが必須となります。 実行順の制御が不要などでCubismUpdateControllerを使用しない場合、コンポーネントの実行順はCocos Creator側で決められるため、実行順によってはExpressionなどの機能が正しく処理されないことがあります。

ユーザが独自に作成したコンポーネントの実行順を制御するには、そのコンポーネントにICubismUpdateableを実装します。

ICubismUpdateableの実装

実行順の制御対象となるコンポーネントにてICubismUpdateableインターフェイスを継承します。

CubismUpdateControllerがアタッチされていない場合、onLateUpdateの処理は実行されないため、以下のスニペットではそのコンポーネント単独でも動作するようにしています。

```
@ccclass('CubismEyeBlinkController')
export default class CubismEyeBlinkController extends Component implements
ICubismUpdatable {
  ...

  /** Model has update controller component. */
  @property({ type: CCFloat, visible: false, serializable: false })
  private _hasUpdateController: boolean = false;
  public get hasUpdateController(): boolean {
    return this._hasUpdateController;
  }
  public set hasUpdateController(value: boolean) {
    this._hasUpdateController = value;
  }

  /** Refreshes controller. Call this method after adding and/or removing <see
cref="CubismEyeBlinkParameter"/>s. */
  public refresh(): void {
    ...

    // Get cubism update controller.
    this.hasUpdateController = this.getComponent(CubismUpdateController) != null;
  }

  /** Called by cubism update controller. Order to invoke OnLateUpdate. */
  public get executionOrder(): number {
    return CubismUpdateExecutionOrder.CUBISM_EYE_BLINK_CONTROLLER;
  }
```

```
  /** Called by cubism update controller. Needs to invoke OnLateUpdate on Editing.
*/
  public get needsUpdateOnEditing(): boolean {
    return false;
  }

  /** Called by cubism update controller. Updates controller. */
  public onLateUpdate(): void {
    // LateUpdateの処理を実装
  }

  /** Called by Cocos Creator. Makes sure cache is initialized. */
  protected start(): void {
    // Initialize cache.
    this.refresh();
  }

  /** Called by Cocos Creator. */
  protected lateUpdate(): void {
    if (!this.hasUpdateController) {
      // CubismUpdateControllerがアタッチされていない場合、自身のLateUpdateから処理を行う。
      this.onLateUpdate();
    }
  }

  ...

}
```

- hasUpdateController
  - PrefabにCubismUpdateControllerがアタッチされているかを設定します。
- needsUpdateOnEditing
  - Cocos Creatorでシーンが非実行中にも更新を行う必要があるかを返します。
    CubismUpdateControllerはエディターモードでも実行されるので、そのコンポーネントが非実行中にも更新する必要がない場合はfalseを返します。
- executionOrder
  - そのコンポーネントの実行順を返します。
    この値が小さいコンポーネントほど先に呼び出されます。
    Cubism SDK for Cocos Creatorの各コンポーネントの実行順序は CubismUpdateExecutionOrder に定義されています。
- ユーザが実行順序を制御するコンポーネントを追加した場合、CubismUpdateExecutoinOrderの定義を参考にして、ここで返す値でどのタイミングで呼び出されるかを注意してください。
  - 例（独自のコンポーネントをCubismLookControllerとCubismPhysicsControllerの間のタイミングで呼び出したい場合）：

        - CubismLookControllerは700、CubismPhysicsControllerは800が設定されるため、独自のコンポーネントは750を返すよう設定します。

- onLateUpdate()
    - CubismUpdateControllerによって実行順序を制御され呼び出される関数です。
      そのコンポーネントが行う更新処理を記述します。

実行中にコンポーネントを追加、削除する場合の注意

CubismUpdateControllerが実行順を制御するコンポーネントは、`onEnable()` のタイミングで行います。 すべてのコンポーネントがあらかじめPrefabにアタッチされている場合は問題ありませんが、シーン実行中、動的にコンポーネントを追加または削除した場合、明示的に `CubismUpdateController.refresh()` を呼び出して読み込み直す必要があります。

```
this.addComponent(MyComponent1);

let component = this.getComponent(MyComponent2);
component.destroy();

let updateController = this.getComponent(CubismUpdateController);
updateController.refresh();
```

Expression

概要

Expressionは、現在のパラメータの値に対して、表情用のパラメータの値を加算または乗算して設定することができる、Cubismの表情モーションを扱う機能です。 Cocos Creator標準のモーション再生機能であるAnimatorでは、レイヤーに設定できるブレンドモードに乗算（Multiply）が無いため、表現することが出来ません。

表情機能の設定ファイルは.exp3.json形式で書き出されます。 表情の仕組みにつきましては こちら をそれぞれご覧ください。

Expressionを利用するには、あらかじめPrefabに UpdateControllerの設定 と ParameterStoreの設定 を行う必要があります。

該当のチュートリアル記事は チュートリアル - 表情機能を使用する をご覧ください。

表情モーションを再生するには以下の手順を行います。

1. [.exp3.asset]を作成
2. [.expressionList.asset]を作成し、[.exp3.asset]を追加
3. 表情モーションを再生
4. 表情モーションの計算と適用

[.exp3.asset]を作成

[.exp3.asset]は、[exp3.json]からコンバートしたScriptableObjectのアセットです。 [.exp3.asset]を変更した場合、正常な動作を保証できません。

[exp.json]を[.exp3.asset]にコンバートするには以下の処理を行います。

1. [exp.json]をパース
2. CubismExpressionData作成
3. [.exp3.asset]作成

[exp.json]をパース

[exp3.json]のパースは CubismExp3Json.loadFrom(string exp3Json) または CubismExp3Json.loadFrom(TextAsset exp3JsonAsset) を使います。

CubismExpressionData作成

CubismExpressionDataは、パースされた.exp3.jsonの情報を記録するクラスで、以下のデータを保持しています。

フィールド	型	説明
Type	string	jsonファイルの種類。.exp3.jsonの場合は "Live2D Expression" が設定される。

フィールド	型	説明
FadeInTime	number	表情がフェードインするまでの時間。値の単位は秒。
FadeOutTime	number	表情がフェードアウトするまでの時間。値の単位は秒。
Parameters	SerializableExpressionParameter[]	表情を適用するパラメータのID、適用する値、計算方式。

[.exp3.asset]の新規作成には CubismExpressionData.CreateInstance(CubismExp3Json json) を使います。

[.exp3.asset]を上書きするには CubismExpressionData.CreateInstance(CubismExpressionData expressionData, CubismExp3Json json) を使います。

- CubismExp3Json json：CubismExp3Jsonは[exp3.json]をパースしたデータ。
- CubismExpressionData expressionData：上書き対象となるCubismExpressionData。

[.exp3.asset]作成

[.exp3.asset]は以下のように作成します。

```
const jsonSrc = readFileSync(asset.source, UTF8);
const json = CubismExp3Json.loadFrom(jsonSrc);
if (json == null) {
  console.error('CubismExp3Json.loadFrom() is failed.');
  return false;
}
const data = CubismExpressionData.createInstance(json);
const serialized = EditorExtends.serialize(data);
writeFileSync(outputFilePath, jsonSrc);
asset.saveToLibrary('.json', serialized);
refresh(outputFilePath);
```

この処理は CubismExpression3JsonImporter.import() で行っています。

※ ランタイムの場合、CubismExpressionDataを.asset化する必要はありません。

[.expressionList.asset]を作成し、[.exp3.asset]を追加

[.expressionList.asset]は、モデル毎の[.exp3.asset]の参照をリスト化したアセットです。 これはCubismExpressionControllerでCurrentExpressionIndexから再生する表情モーションを取得するために使用します。 [.exp3.asset]リストの順番は[exp3.json]のインポート順で追加されます。

[.expressionList.asset]を作成

[.expressionList.asset]はモデルのプレハブと同じ階層に作成しています。

```
    const source = JSON.parse(readFileSync(asset.source, UTF8));
    const expDataUuidsSource = Array.isArray(source.cubismExpressionObjects)
      ? (source.cubismExpressionObjects as any[])
      : undefined;
    if (expDataUuidsSource == null) {
      console.error('CubismExpressionListImporter.import(): parse error.');
      return false;
    }
    const expDataUuids = purseCubismExpressionObjects(expDataUuidsSource);
    if (expDataUuids == null) {
      console.error('CubismExpressionListImporter.import():
CubismExpressionObjects parse error.');
      return false;
    }
    const cubismExpressionObjects = new Array<CubismExpressionData>
(expDataUuids.length);
    for (let i = 0; i < cubismExpressionObjects.length; i++) {
      const uuid = expDataUuids[i];
      // @ts-ignore
      cubismExpressionObjects[i] = EditorExtends.serialize.asAsset(
        uuid,
        CubismExpressionData
      ) as CubismExpressionData;
    }
    const list = new CubismExpressionList();
    list.cubismExpressionObjects = cubismExpressionObjects;
    await asset.saveToLibrary('.json', EditorExtends.serialize(list));
```

※この項目の処理は CubismExpressionListImporter.import() で行っています。

[.exp3.asset]を追加

[.expressionList.asset]に[.exp3.asset]を追加するには、以下のように行っています。

```
    const jsonSrc = readFileSync(asset.source, 'utf8');
    const json = CubismExp3Json.loadFrom(jsonSrc);
    if (json == null) {
      console.error('CubismExp3Json.loadFrom() is failed.');
      return false;
    }
    const data = CubismExpressionData.createInstance(json);
    const serialized = EditorExtends.serialize(data);
    asset.saveToLibrary('.json', serialized);
    refresh(outputFilePath);

    const dirPath = Path.join(Path.dirname(asset.source), '../');
    const dirName = Path.basename(dirPath);
    if (dirPath.length == 0) {
      console.warn('CubismExpressionDataImporter.import(): Not subdirectory.');
      return true;
```

```
    }
    const expressionListFilePath = Path.join(
      dirPath,
      dirName + `.${CubismExpressionListImporter.extension}`
    );
    updateExpressionListFile(expressionListFilePath, asset.uuid);
    refresh(expressionListFilePath);
```

※この項目の処理は CubismExpressionDataImporter.import() で行っています。

表情モーションを再生

CubismExpressionController.CurrentExpressionIndexに再生する表情モーションのインデックスを設定することにより、表情モーションを再生・変更することが出来ます。

表情モーションの再生は以下の処理を行っています。

1. 再生中の表情モーションの終了タイムを設定
2. 新しい表情モーションを初期化し、表情モーションの再生リストに追加

再生中の表情モーションの終了タイムを設定

再生中の表情モーション存在する場合、再生中の表情モーションが終了するように終了時間を設定します。

```
    const playingExpressions = this._playingExpressions;
    if (playingExpressions.length > 0) {
      const playingExpression = playingExpressions[playingExpressions.length - 1];
      playingExpression.expressionEndTime =
        playingExpression.expressionUserTime + playingExpression.fadeOutTime;
      playingExpressions[playingExpressions.length - 1] = playingExpression;
    }
```

この処理は CubismExpressionController.startExpression() で行っています。

新しい表情モーションを初期化し、表情モーションの再生リストに追加する

表情モーションの再生情報を持つCubismPlayingExpressionを作成します。

CubismPlayingExpressionは、以下の情報を保持しています。

フィールド	型	説明
Type	string	jsonファイルの種類。.exp3.jsonの場合は "Live2D Expression" が設定される。
FadeInTime	number	表情がフェードインするまでの時間。値の単位は秒。
FadeOutTime	number	表情がフェードアウトするまでの時間。値の単位は秒。

フィールド	型	説明
Weight	number	表情のウェイト。値の範囲は0から1。
ExpressionUserTime	number	表情の再生が開始してからの経過時間。値の単位は秒。
ExpressionEndTime	number	表情の再生が終了する時間。値の単位は秒。
Destinations	CubismParameter[]	表情が適用するパラメータの配列。
Value	number[]	表情がパラメータに適用する値。要素数と並び順はDestinationsと同一。
Blend	CubismParameterBlendMode[]	表情がパラメータに適用する計算方式。要素数と並び順はDestinationsと同一。設定される計算方式はOverride, Additive, Multiply。

CubismPlayingExpressionは CubismPlayingExpression.create(CubismModel model, CubismExpressionData expressionData) を使用して作成することができます。

- CubismModel model：表情モーションを再生するモデル。
- CubismExpressionData expressionData：再生する表情モーションの[exp3.asset]データ。

作成した表情モーションは、以下のように再生リストに追加します。

```
const playingExpression = CubismPlayingExpression.create(
  this._model,
  expressionsList.cubismExpressionObjects[this.currentExpressionIndex]
);

if (playingExpression == null) {
  return;
}

// Add to PlayingExList.
playingExpressions.push(palyingExpression);
```

この処理は CubismExpressionController.startExpression() で行っています。

表情モーションの計算と適用

表情の計算と適用はCubismExpressionController.onLateUpdate()から呼び出されます。

※ 表情の適用はParameterStore.saveParameters()の後に行う必要があります。 Expressionは、現在の値に対して毎フレーム新たに表情用の値を適用する機能です。 ParameterStore.saveParameters()の前で適用した場合、表情で適用した結果が保存されてしまうため、表情用のパラメータの値が無限に加算または乗算されます。

Pose

概要

Poseは、Poseの設定形式の.pose3.jsonに記述されたパーツ群の表示状態を管理する機能です。 Poseを使用することで、「下ろした状態の腕」と「組んだときの腕」のように、画面には同時に表示させたくないパーツを管理することができます。 また、Poseを利用することで、PartのIDが異なるモデル（衣装替え等）にも対応することが可能です。

.pose3.jsonは、Cubism Viewer for OWで設定することができます。 パーツの設定方法などの詳細につきましては こちら をご覧ください。

OW方式でインポートしたモデルと、従来方式のモデルとを比較した場合、Poseによって以下の3点が変化します。

- [Prefabのルート]/Parts/ 以下のGameObjectに、CubismPosePartがアタッチされる。
- AnimationClip内のPartOpacityのカーブがSteppedで合った場合、Linearカーブに加工される。
- AnimationClipから、pose3.jsonに記述されていないOpacityカーブが削除される。

Poseでは、以下の処理を行います。

1. Poseモーション内にSteppedで設定されたPartsのOpacity（不透明度）のカーブを加工
2. Pose用パーツを特定
3. 表示するPoseパーツを判別するためパーツのOpacityを保存
4. Poseの計算と適用
5. 不透明度の連動

Poseモーション内にSteppedで設定されたPartsのOpacity（不透明度）のカーブを加工

Editor マニュアルの ポーズの設定 に記述されている通り、Poseで扱うパーツの不透明度（PartOpacity）用のカーブは、カーブの種類はステップ（Stepped）を前提としております。 Cubism SDK for Cocos Creatorでは、このカーブをリニア（Linear）に加工してAnimationClipに書き出します。

Stepped以外の種類のカーブを設定した場合、そのカーブは加工されずそのままAnimationClipに書き出します。

```
if (
  shouldImportAsOriginalWorkflow &&
  poseJson != null &&
  poseJson.adeInTime != 0.0
) {
  let track = CubismMotion3Json.convertSteppedCurveToLinerCurver(
    curve,
    poseJson.fadeInTime
  );
}
```

このOpacityのカーブを加工する処理は、 CubismMotion3Json.toAnimationClipB() で行っています。

Pose用パーツを特定

CubismPosePartは、OW方式でモデルをインポートした時にアタッチされます。 アタッチする Node は、[Prefabのルート]/Parts/ 以下に配置された、.pose3.json に記述された ID と同名のものになります。

```
// Pose用パーツを取得
const part = ArrayExtensionMethods.findByIdFromParts(
  parts,
  group[partIndex].id
);

// CubismPosePartをアタッチ
posePart = part.node.addComponent(CubismPosePart)!;

// Poseの情報をCubismPosePartに設定
posePart.groupIndex = groupIndex;
posePart.partIndex = partIndex;
posePart.link = group[partIndex].link;
```

表示するPoseパーツを判別するためパーツのOpacityを保存

現在の CubismPart.opacity の値を、前フレームにキャッシュした値と比較することで、.pose3.json に記述されたグループ内の表示パーツを判別します。 値のキャッシュは CubismPoseController.savePartOpacities() で行っています。

Poseの計算と適用

表示パーツの適用はAnimatinoClipで適用します。

非表示パーツの適用は CubismPoseController.doFade() で行ないます。

非表示パーツのOpacityは表示するパーツの不透明度との関係で背景が透けないレベルで不透明度を低下させます。 CubismPoseController.doFade()では表示パーツを探してから、非表示パーツ Opacity を設定します。

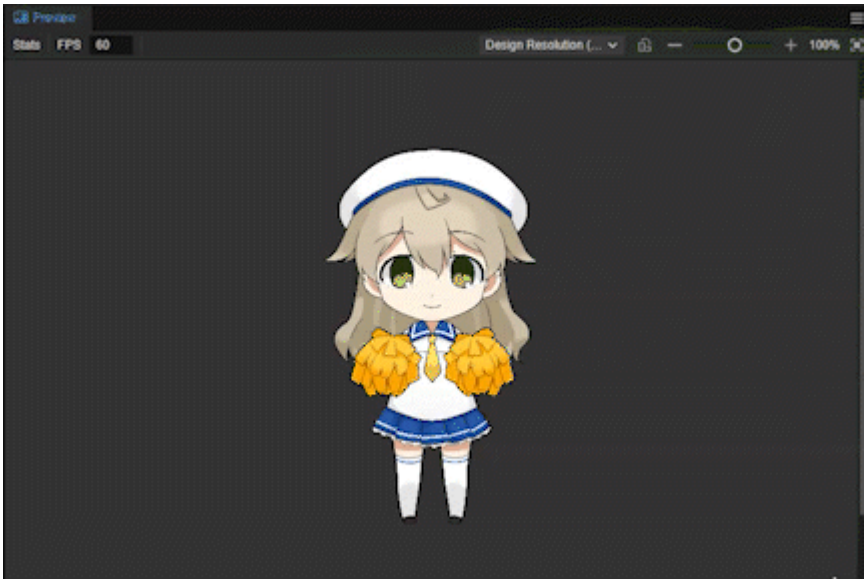
不透明度の連動

不透明度の連動は Pose の適用後、Opacity を CubismPosePart.link に書かれた連動パーツは CubismPoseController.copyPartOpacities で Opacity を設定します。

Animation

概要

Cubism Editorから書き出した組み込み用アニメーションファイル(.motion3.json)を、Cocos Creatorの標準機能であるAnimationを使用して再生するサンプルシーンです。



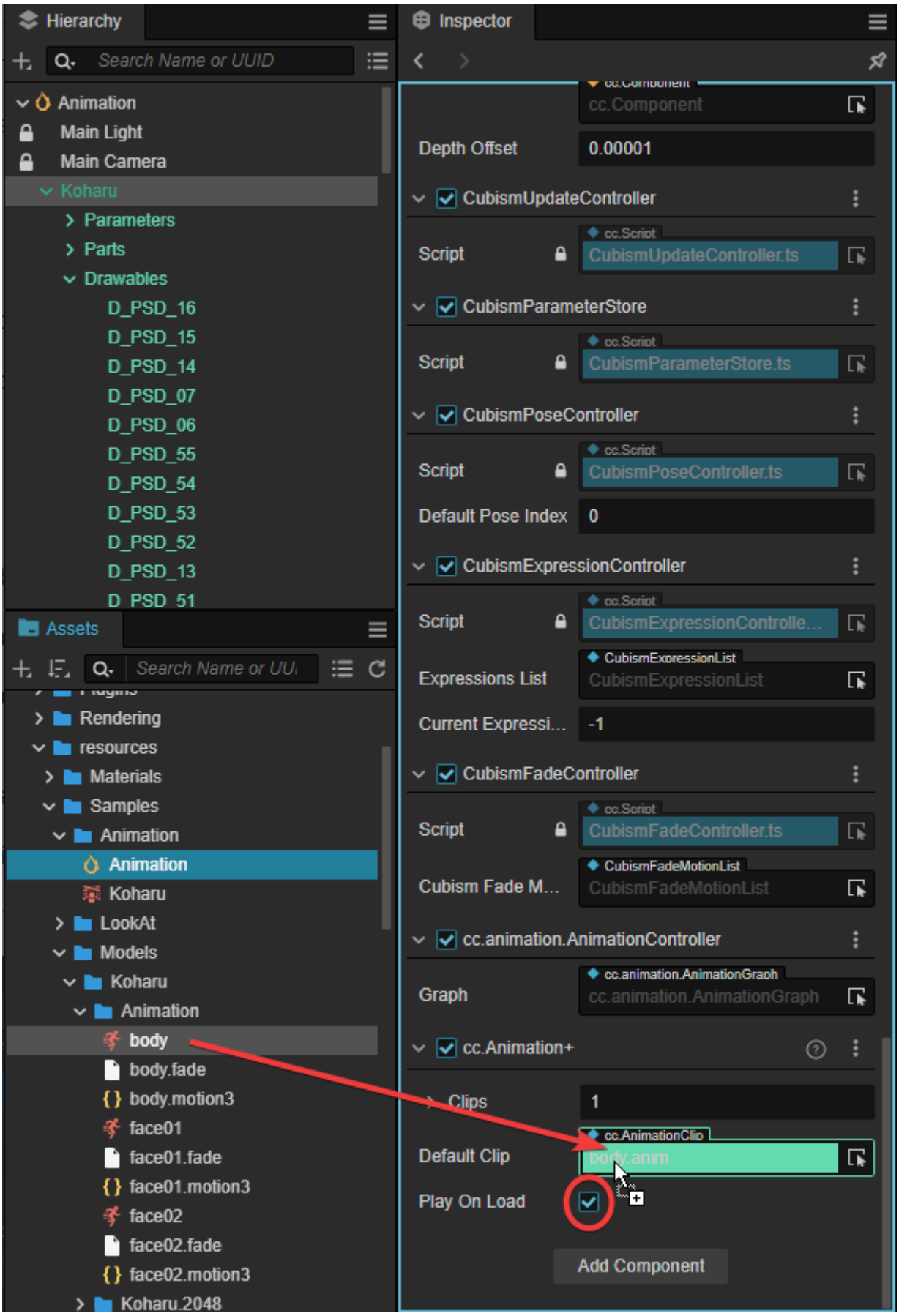
何の機能が確認できるか

Cocos Creatorの標準機能であるAnimationを利用したアニメーションの再生のさせ方を確認することができます。

Animatorについてはこちらをご確認ください。

シーンの設定方法

モデルにAnimationをアタッチし、モデルに付属されたAnimationClipをセットします。 その後、Play on loadにチェックをつけます。



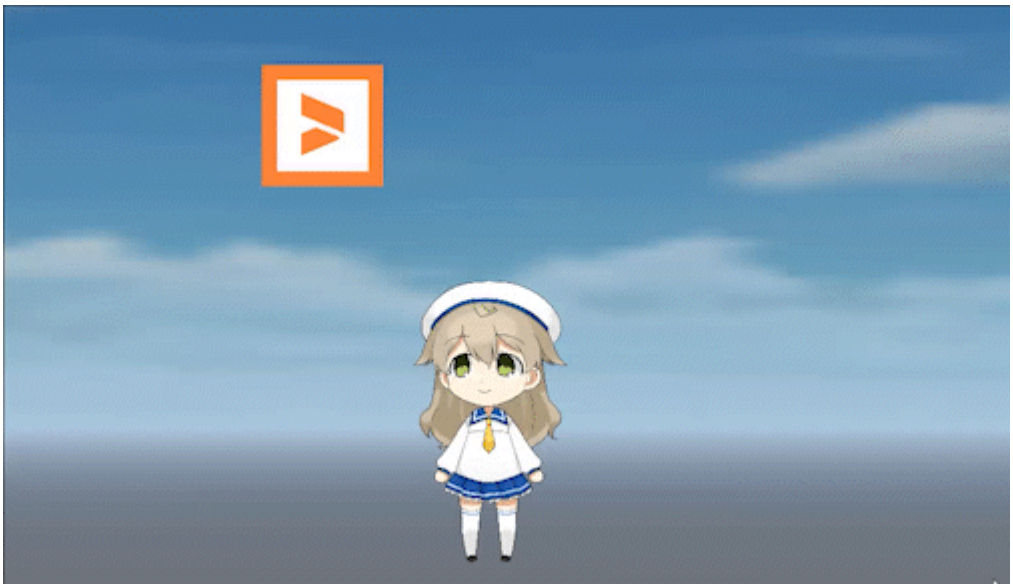
作成されたモデルをサンプルシーンで使用するには

チュートリアル - アニメーションの再生 のチュートリアルを進めていただくとこのサンプルシーンと同等のものが完成します。

LookAt (サンプル)

概要

ゲームシーン上で特定の座標を視線追従する機能を確認できるサンプルシーンです。



何の機能が確認できるか

- 視線追従のための２つの機能を確認できます。
  - CubismLookController（視線追従するCubismのPrefabにアタッチ）
  - ICubismLookTargetを実装したクラス（視線追従されるものにアタッチ）
    - サンプルシーンではICubismLookTargetを使用

簡単な使い方

RunするとCubismのモデルが左右に動くTargetオブジェクトを視線追従し続けます。

シーンの設定方法

Hierarchy上のTargetPivot -> Target にCubismLookTargetBehaviourがアタッチされているのが確認できます。
CubismLookControllerがアタッチされたモデルはこのオブジェクトを視線追従します。

モデルの設定方法は チュートリアル - 視線追従の設定 をご確認ください。

Cocos Creator版特有の注意点

Cubism SDK for Cocos Creator を Cocos Creator に読み込んだ際の注意点

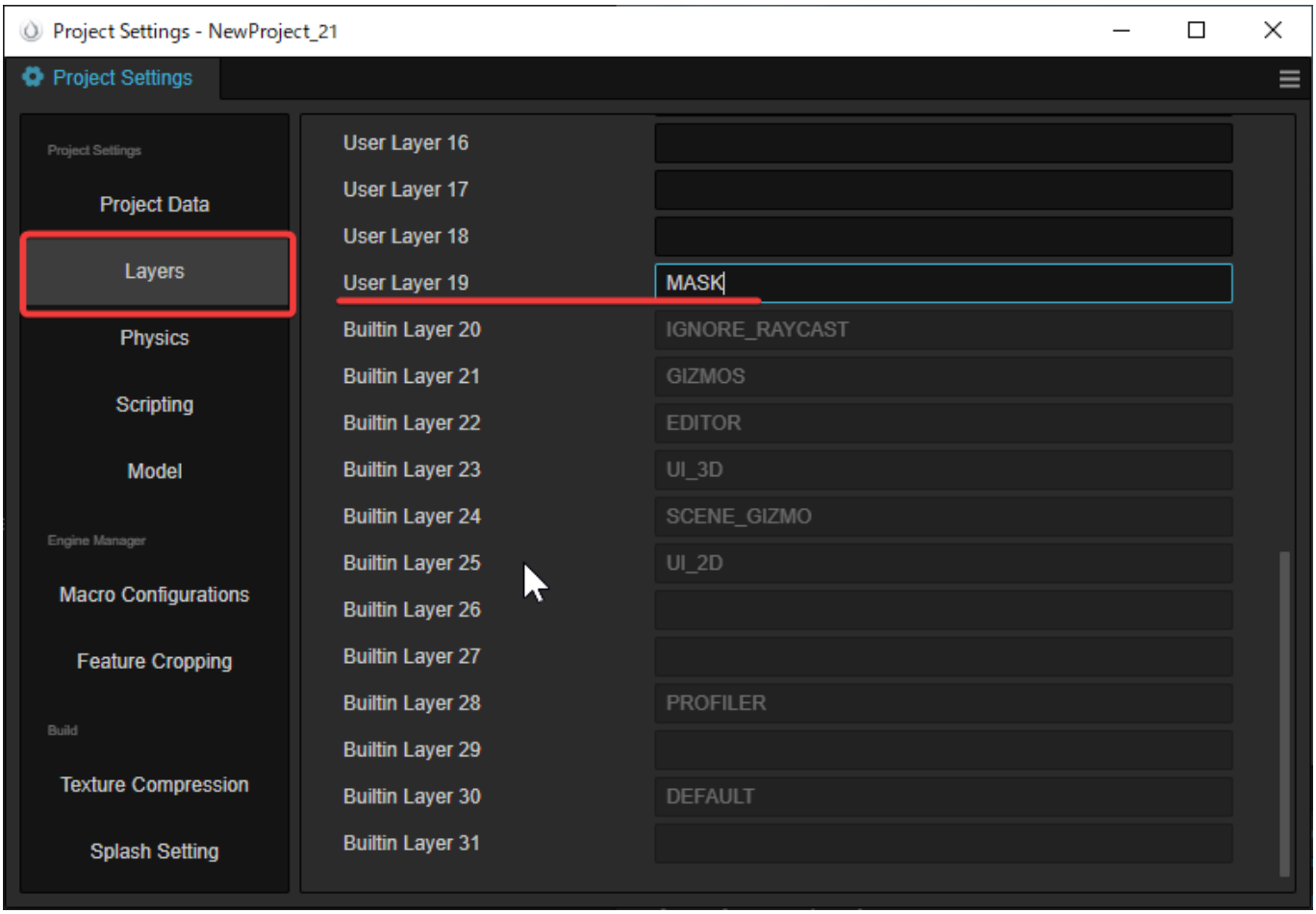
Cubism SDK for Cocos Creatorに同封されているサンプルモデルを使用する際、Cubism SDK for Cocos Creatorを読み込んだ直後ではインポート順が不定となるため、そのままでは使用できない場合がございます。 エラー等が表示された場合は必ずmodel3.jsonまたはそれらが入ったフォルダを[Reimport Asset]で再インポートしてください。 また、Cubism SDK for Cocos Creator を Cocos Creator に読み込んだ直後では[GlobalMaskTexture.asset]が正しく使用できない場合がございます。 その場合は必ず[GlobalMaskTexture.asset]を[Reimport Asset]で再インポートし、Cocos Creatorを再起動してください。

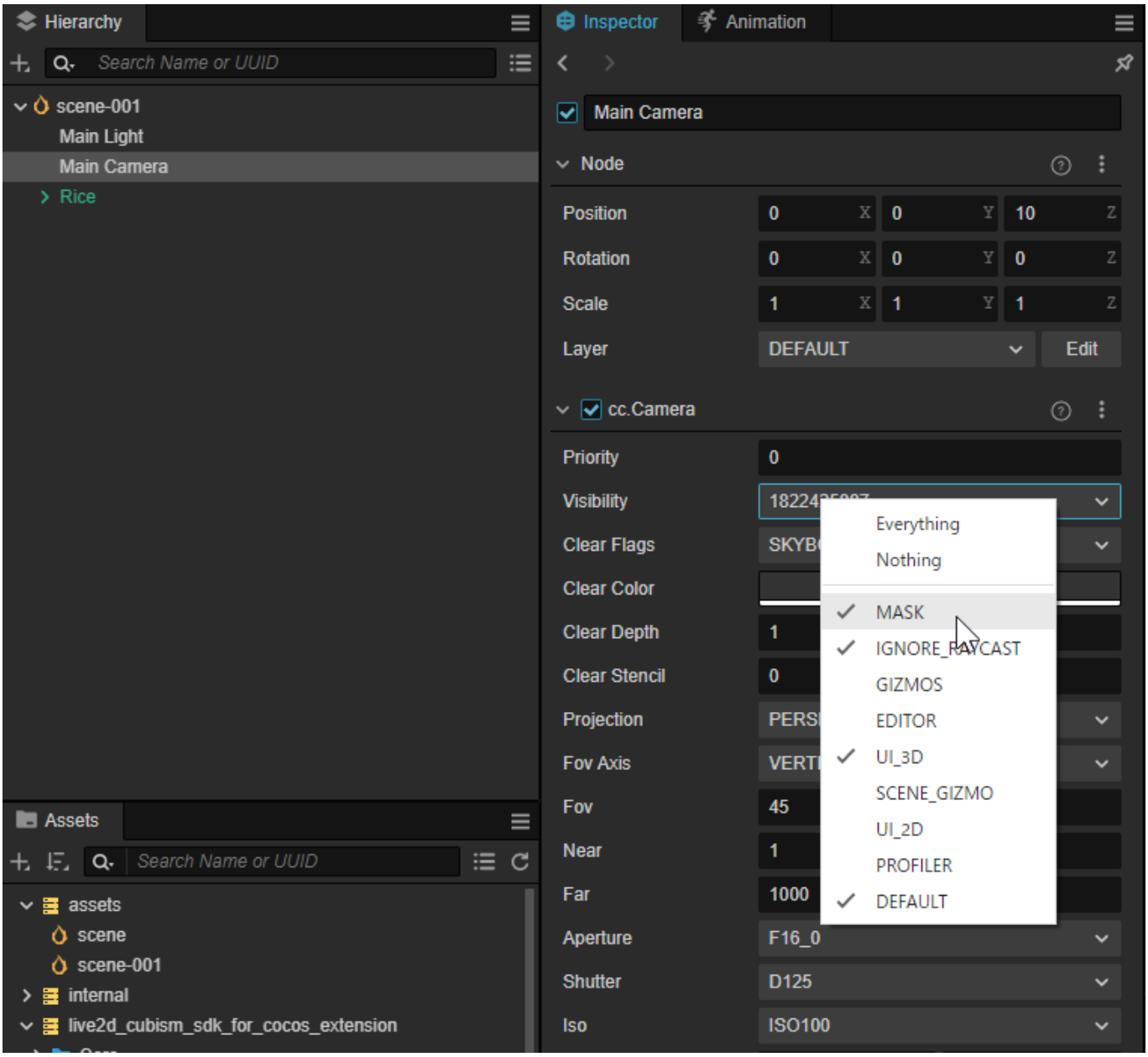
マスクの描画について

GlobalMaskTexture.asset

Cubism SDK for Cocos Creator を Cocos Creator に読み込んだ直後では[GlobalMaskTexture.asset]が正しくマスクが描画されない場合がございます。 その場合は必ず[GlobalMaskTexture.asset]を[Reimport Asset]で再インポートし、Cocos Creatorを再起動してください。

また、シーン全体にマスクで使用しているテクスチャが描画される場合がございます。 その場合はCocos Creatorの [Project]-[Project Settings]-[Layers]にございます`User Layer 19`に任意の名前(MASKなど)を付け、シーンのMain Camera`Visibility`からチェックを外してください。





TypeScript(JavaScript)言語仕様上の注意

TypeScript(JavaScript)言語仕様上の注意 Cocos Creator が提供する math.Vec2, math.Vec3 のようなクラス、本SDKが提供する IStructLike インターフェイス実装クラスは言語仕様上、インスタンスです。

C#のような言語のような構造体を扱うようにインスタンスのメンバープロパティの値を変更すると予期しない動作を引き起こす可能性がありますので注意して下さい。

※ 本SDKが提供する IStructLike インターフェイス実装クラスはメンバーをreadonlyに設定しておりますので基本的な操作で誤って上記のような操作をしてしまうことは避けられるようにしています。

Typescript 4.7.4 時点での言語仕様上の都合で以下のような場合にはエラーとならないので注意して下さい。

```
export interface IReadonlySample {
  readonly x: number;
}
export interface ISample {
  x: number;
```

```
}
export class Sample {
  public readonly x: number = 0;
}

const a: ISample = new Sample(); // エラーにならない。
a.x = 1;

const b: IReadonlySample = new Sample();
b.x = 2; // TS: 2540 エラー
```

©Live2D Inc.