Cubism SDK for Cocos Creator alpha チュートリアル

SDK をインポート

Cubism Editor から書き出した組み込み用モデルファイルを、Cocos Creator のプロジェクトにインポートして画面に表示するまでのチュートリアルです。

用意するもの

Cocos Creator と Cubism SDK for Cocos Creator

Cubism SDK for Cocos Creator が対応している Cocos Creator バージョンは、v3.6.2 以降になります。 v3.6.2 未満の場合、SDK 内のアセットにシリアライズされている情報が破棄される場合がございます。 Cocos Creator のインストールについては こちら をご覧ください。

また、 Cubism SDK for Cocos Creator をあらかじめダウンロードしておいてください。 こちらは zip形式として配布しています。 本SDK は Github でも公開しておりますがこちらにはCubism Coreライブラリが含まれないため、モデルを組み込む際にはパッケージを使用するか、パッケージにあるCubism Coreライブラリを同ディレクトリに配置してください。

Node.js

Cubism SDK for Cocos Creator を使用する際に npm コマンドを使用してモジュールをインストールする必要があります。 必ずCubism SDK for Cocos Creatorに対応しているバージョンのNode.jsを下記よりインストールしてください。

Node.js

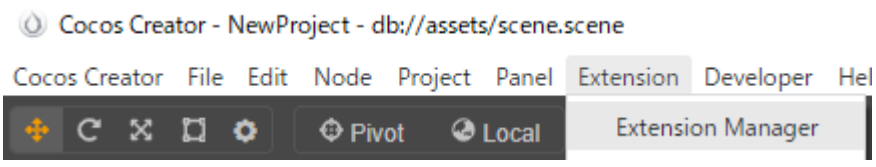
nvm等、Node.js 管理ツールからインストールしたものでも結構です。 インストールと同時にnpmが実行できる環境変数を設定してください。

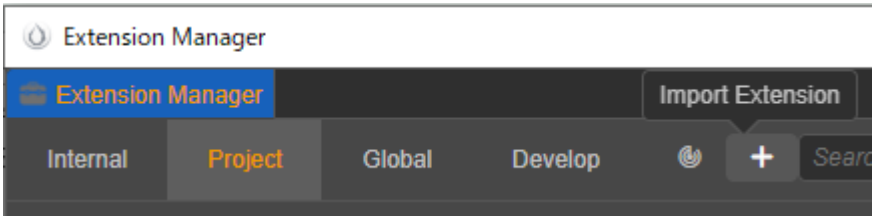
Cubism モデルデータ

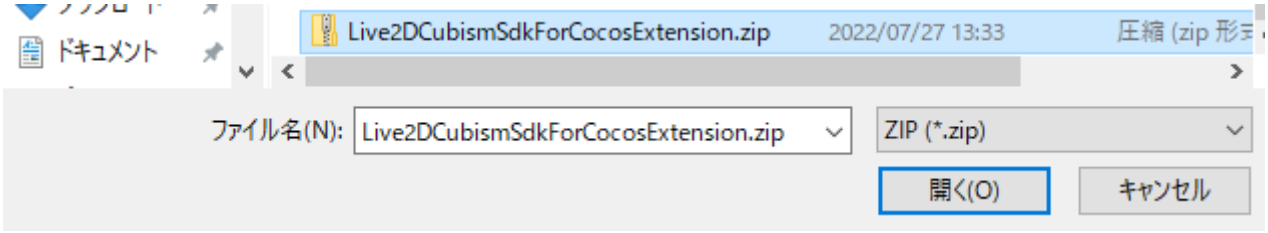
SDK で Live2D のモデルを扱う場合は、編集用の .cmo3 や.can3 ではなく、組み込み用モデルとして書き出す必要があります。 組み込み用のモデルの書き出し方については こちら をご覧ください。 書き出されるデータは、.moc3 ファイル、.model3.json ファイル、テクスチャの入ったフォルダです。 これらを一つのフォルダにまとめてください。

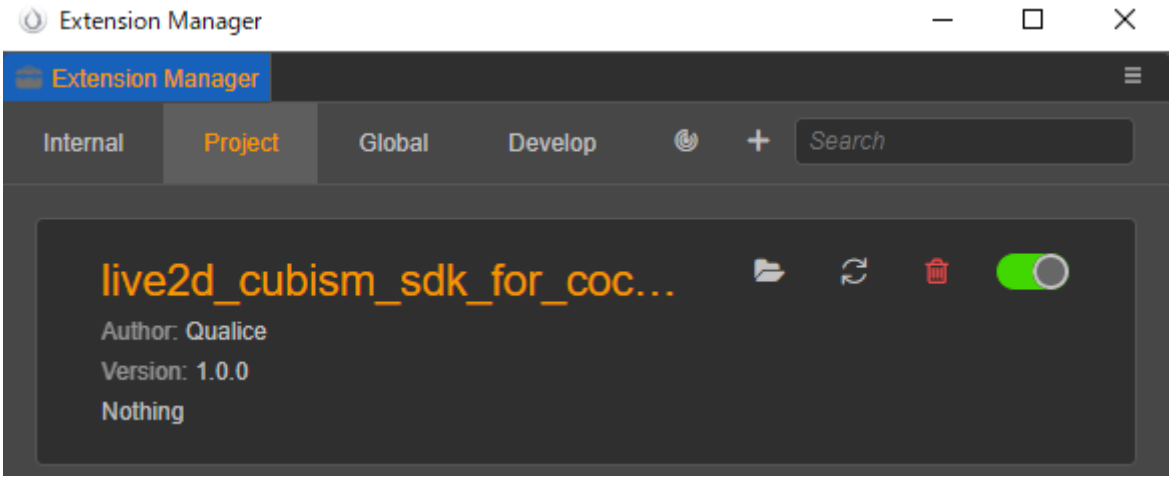
SDK をプロジェクトにインポート

- Cocos Creator の新規プロジェクトを作成します。
  - Cocos Creator の新規プロジェクト作成については、 こちら をご覧ください。





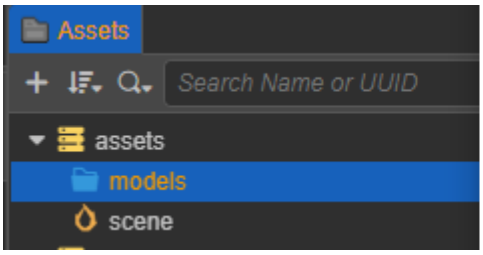


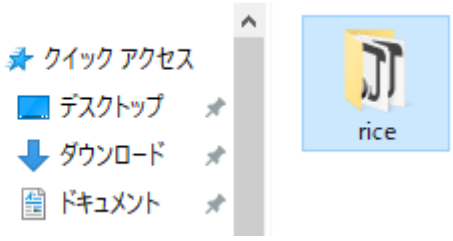


- **Cubism SDK for Cocos Creator** を **Extension** に追加します。
    - **Extension Manager** を開きます。(Menu->Extension->Extension Manager)
    - Extension Manager の **Project** を選択し、 **Import Extension** ([+]ボタン) をクリックします。
    - **Cubism SDK for Cocos Creator（.zip ファイル）** を選択し開きます。
    - インポートが完了したら、Cocos Creator を閉じます。
    - ※Extension Manager を使わず、エクスプローラーでプロジェクトルートに **extensions** フォルダを作成し、そこで zip ファイルを展開しても問題ありません。
- Extension プロジェクトをビルドします。
    - 上記で追加した Extension（{プロジェクトルート}/extensions/{SDK フォルダ}）に移動し、コマンドプロンプトなどで以下のコマンドを実行します。
        - npm install
        - npm run build
    - ビルドが完了したら、プロジェクトを開き直してください。

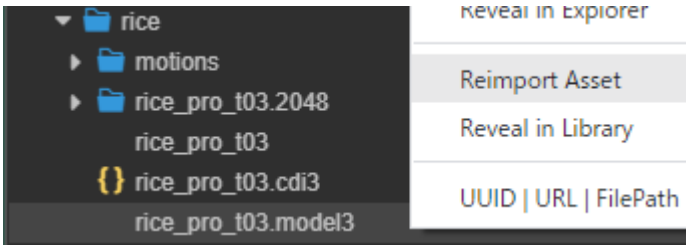
モデルをインポート

Cubism Editor から書き出した組み込み用モデル一式を、フォルダごと Asset パネルにドラッグ・アンド・ドロップします。





正常にインポートが完了すると、SDK に含まれている Cubism の Importer により自動的に Prefab が生成されます。 ※Prefab が生成されない場合、 **model3.json** ファイルを右クリックし、 **Reimport Asset** をクリックしてください



この生成された Prefab は、Live2D のモデルを Cocos Creator で取り扱える状態にコンバートされたものです。 パラメータやパーツの表示は、この段階で Hierarchy パネルから操作が可能な状態となっています。

生成された Prefab を Hierarchy パネル、または Scene パネルに追加してシーンを実行すれば、モデルを配置することができます。

Original Workflow 方式について

Original Workflow 方式でモデルをインポートする場合、Project ウィンドウにドラッグ・アンド・ドロップする前に、 メニューバーの Live2D > Cubism > Original Workflow > Should Import As Original Workflow にチェックを入れてください。

Cubism SDK for Cocos Creator は、デフォルトで Original Workflow 方式でのインポートが有効化されています。 Original Workflow 方式を使用しない場合、メニューバーの Live2D > Cubism > Original Workflow > Should Import As Original Workflow のチェックを外してください。

ここにチェックが入った状態でインポートした場合、生成される Prefab に追加で OW 方式のコンポーネントがアタッチされます。

アニメーションの再生

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を、Cocos Creator プロジェクトのモデルで再生するまでのチュートリアルです。

［SDK のインポート～モデルを配置］をおこなったプロジェクトに追加することを前提とした説明となっています。

概要

Cocos Creator プロジェクト上で Cubism のアニメーションを再生させるには、.motion3.json 形式のモーションファイルが必要です。 モーションファイルの書き出しについては こちら を参照してください。

SDK では、モデルと同様に motion3.json 用の Importer も用意しており、motion3.json はインポート時に Cocos Creator のアニメーション形式である AnimationClip に自動で変換されます。

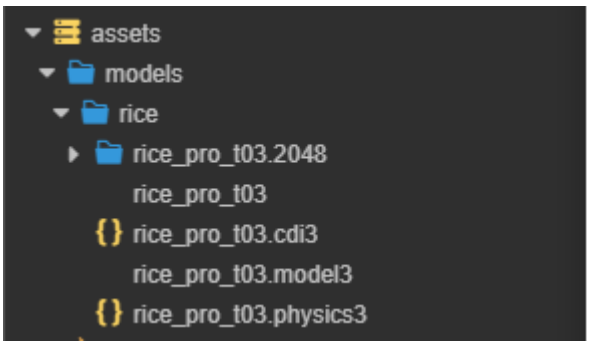
変換された AnimationClip を用いることで、Cocos Creator 上では Live2D の機能は使わず、Cocos Creator のビルトイン機能のみでアニメーションを扱うことが可能となっています。

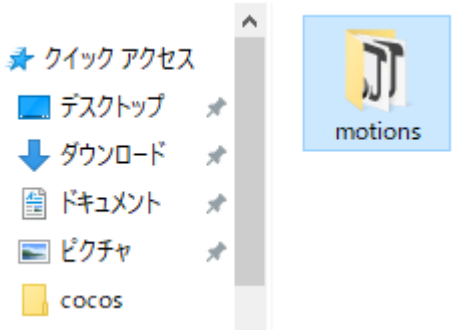
Cocos Creator プロジェクトでモーションを再生させる手順は以下のとおりです。

- モーションファイルをインポート
- AnimationClip を再生

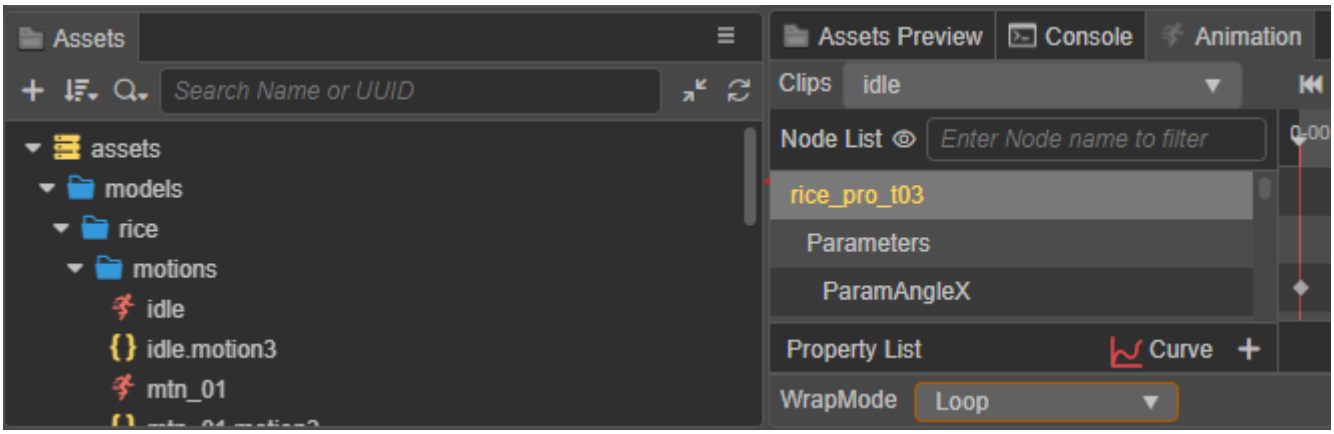
モーションファイルをインポート

Cubism Editor から書き出した組み込み用モーションファイルを、それが入ったフォルダごと Project ビューにドラッグ・アンド・ドロップします。





すると、 以下の画像のように、motion3.json から AnimationClip が生成されます。 生成された AnimationClip に Loop を設定することも可能です。

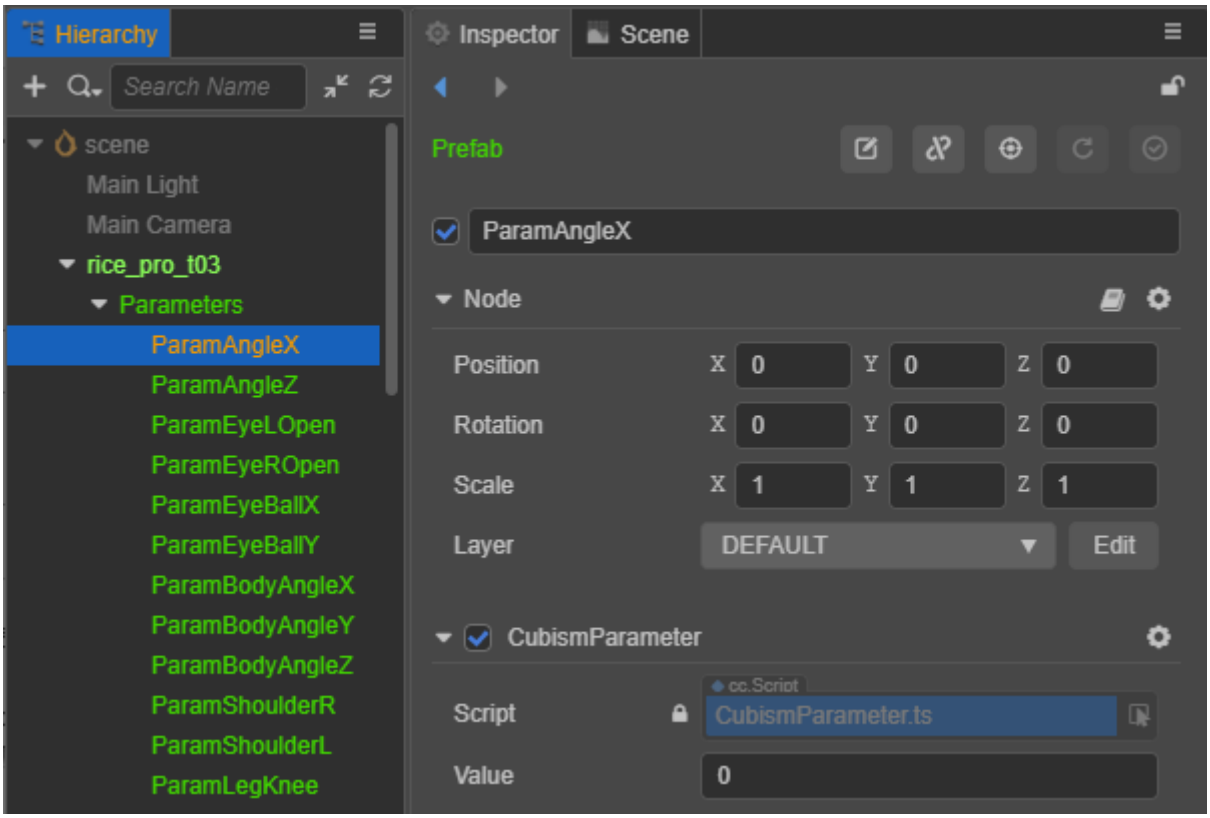


TIPS

この AnimationClip は、モデルの各パラメータに設定する値をカーブとして持っていますが、 このカーブで設定するパラメータ用のプロパティは、モデルの Prefab の以下の階層にあります。（Value は Inspector 上では非表示になっています）

> [モデルのルート]/Parameters/[パラメータ ID]/Cubism Parameter/Value

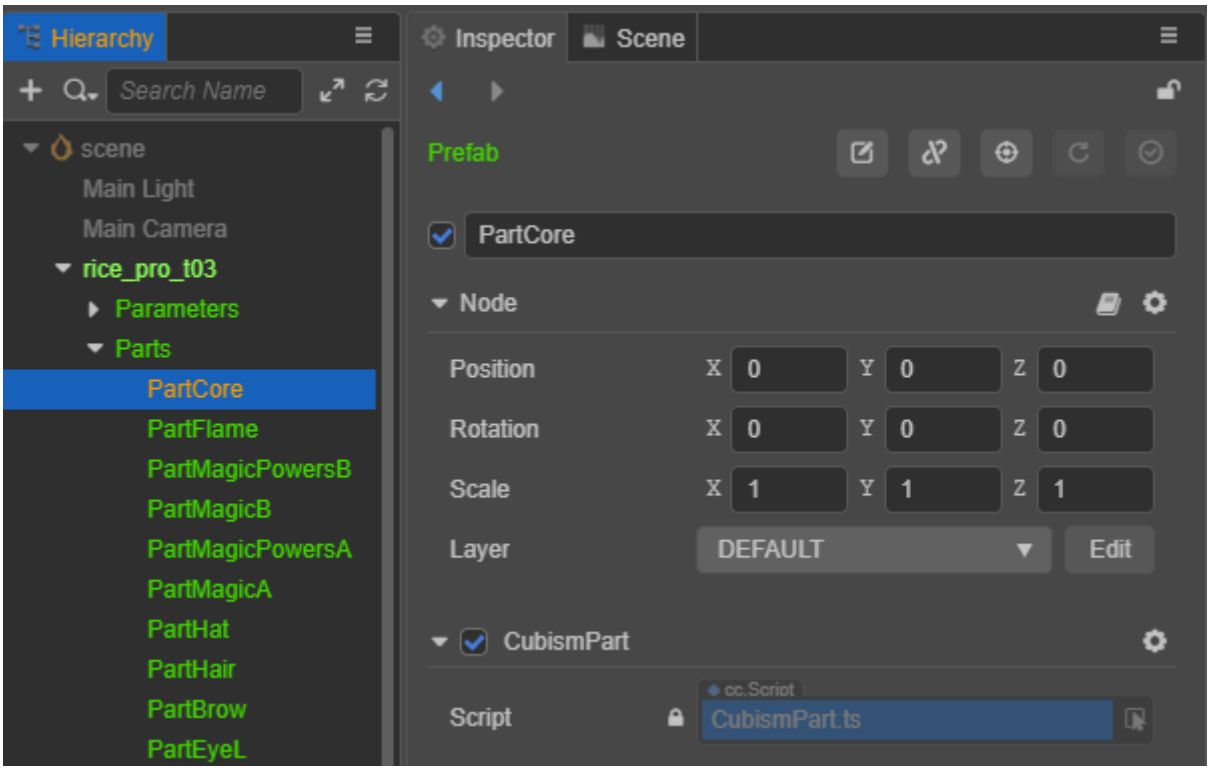
Value に設定する値の最大値と最小値は、パラメータ ID ごとに異なりますが、その範囲外の値は最大値または最小値として扱われます。



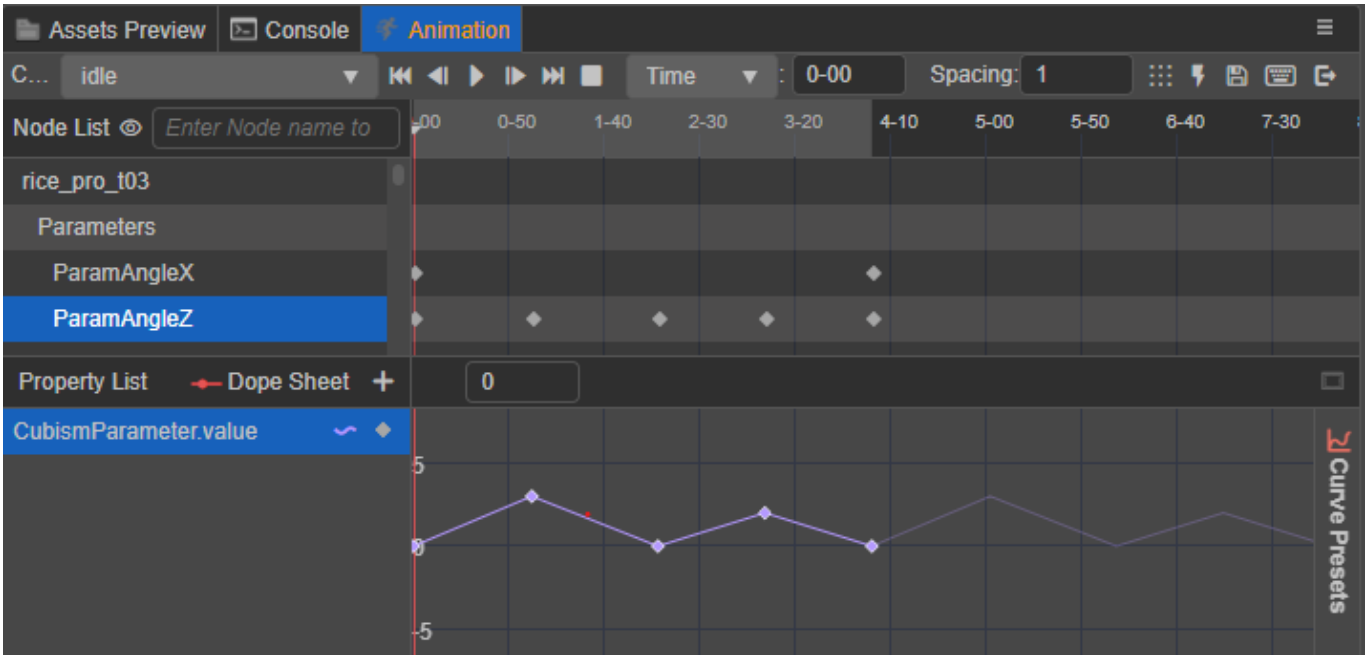
また、パーツの不透明度は、Prefab の以下の階層にあります。（Opacity は Inspector 上では非表示になっています）

> [モデルのルート]/Parameters/[パーツ ID]/Cubism Part/Opacity

Opacity に設定する値は 0 ～ 1 の範囲です。この範囲外の値は 0 または 1 として扱われます。



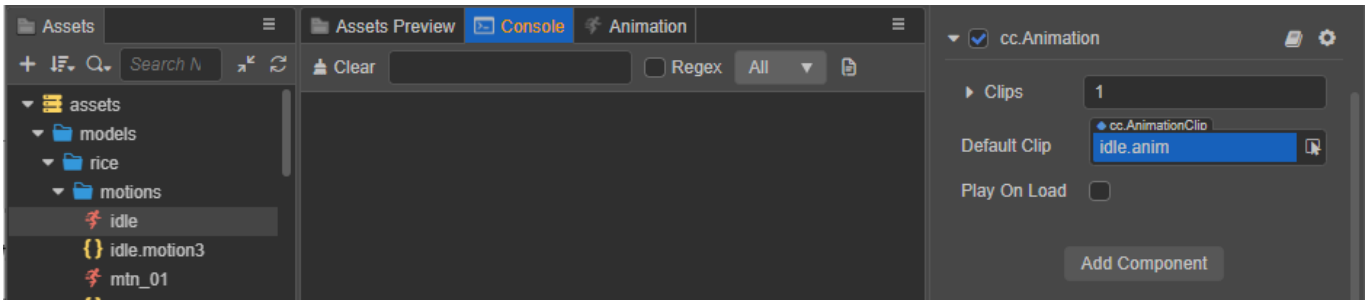
これらのプロパティを指定すれば、Cocos Creator 上で作成した AnimationClip やプログラムから モデルのパラメータやパーツの不透明度を操作することが出来ます。



AnimationClip を再生

Cocos Creator で AnimationClip を再生させるためには、 Animator を利用します。

- モデルの Prefab を Hierarchy にドラッグ・アンド・ドロップします。
- モデルの Prefab の Inspector で、 **Animation** コンポーネントを追加します。
- 「モーションファイルをインポート」で生成された AnimationClip を、 **Animation** コンポーネントの **Default Clip** にドラッグ・アンド・ドロップします。
- **Animation** コンポーネントの **Play on load** にチェックを入れます。



この状態で Scene を実行すると、アニメーションが再生されるようになっています。

Tips

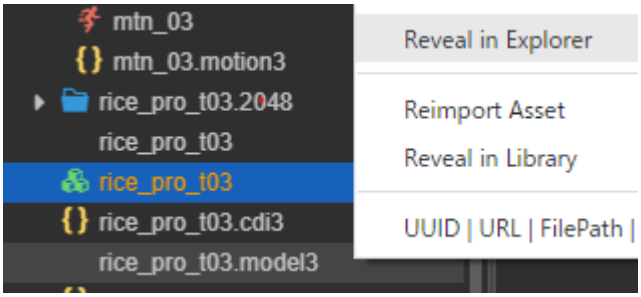
Live2D Cubism SDK for Cocos Creator では、motion3.json に設定されているフェードの時間は、デフォルトでは無効になっています。

インポートしたモデルを更新

モデルの修正などで、一度プロジェクトにインポートしたモデルを更新する場合、 Cocos Creator に再度ドラッグ・アンド・ドロップしてしまうと上書きされずに新しいモデルとしてインポートされてしまいます。

モデルを更新する場合は、以下の手順を行ってください。

moc3 ファイルや model3.json ファイルなどを右クリックし、 Windows の場合は [**Reveal in Explorer**] 、Mac の場合は [**Reveal in Finder**] をクリックします。



すると、右クリックしたファイルの階層で Windows のエクスプローラー、または Mac の Finder が開かれます。 このエクスプローラーや Finder 上で、更新するファイルを上書きします。

最後に、Cocos Creator のアセットマネージャーで model3.json またはそれらが入っているフォルダを右クリックし、[**Reimport Asset**] をクリックするとモデルの Prefab が更新されます。

Original Workflow 方式について

更新される Prefab が従来方式と Original Workflow 方式のどちらで設定されるかは、再インポートを行った時の設定が優先されます。

たとえば、チェックを入れた状態で生成した Prefab を、チェックを外して更新した場合、更新した Prefab には OW 方式のコンポーネントはアタッチされていません。 いません。

モデルのパラメータ更新について

ここでは、モデルに付けられているパラメータの更新についての方法と注意点の説明になっています。

概要

Cubism SDK for Cocos Creator では **toModel()** で生成されたノードにアタッチされたスクリプトから自動的に更新が行われています。

加えてパラメータの更新は別スクリプト上から **parameter.value = 値;** のような形で更新することができます。 注意点として、パラメータの更新は Cocos Creator のイベント関数実行順から **lateUpdate()** を用いて行う必要があります。 また、**ScriptExecutionOrder** でスクリプトの呼び出し順を設定している場合でも同様に注意が必要です。

詳細

Cubism SDK for Cocos Creator での更新方法

Cubism SDK for Cocos Creator では生成された Prefab にアタッチされている `CubismRenderController.ts` を通して自動的に更新が行われています。 これを利用して、アニメーションによって変更される値をパラメータに適用することで描画の更新を行うことができます。 値をパラメータに適用するには以下のような記述になります。

```
lateUpdate(deltaTime: number)
{
  parameter = model.parameters[index];
  parameter.value = value;
}
```

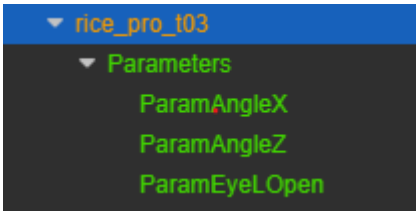
具体的な記述例としては

```
import { _decorator, Component } from 'cc';
import ComponentExtensionMethods from
'../extensions/live2d_cubismsdk_cocoscreator/static/assets/Core/ComponentExtension
Methods';
import CubismModel from
'../extensions/live2d_cubismsdk_cocoscreator/static/assets/Core/CubismModel';
const { ccclass, property } = _decorator;

@ccclass('ParameterUpdateSample')
export class ParameterUpdateSample extends Component {
    _model: CubismModel | null = null;
    _t: number = 0.0;

    start() {
        this._model = ComponentExtensionMethods.findCubismModel(this, true);
    }
```

```
    lateUpdate(deltaTime: number) {
        this._t += deltaTime * 4;
        let value = Math.sin(this._t) * 30;

        let parameter = this._model.parameters[1];
        parameter.value = value;
    }
}
```

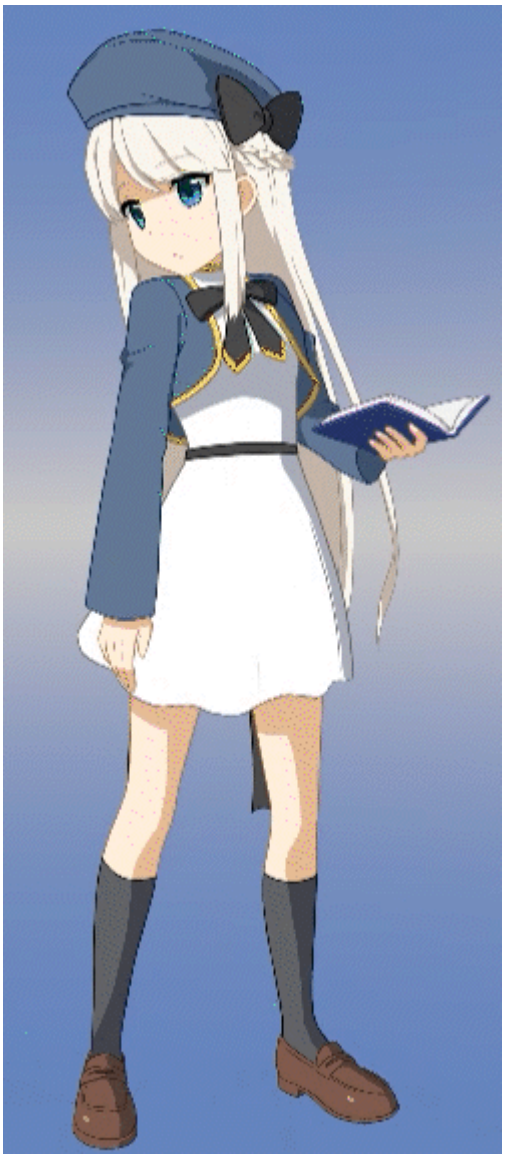
という形になります。 これは、後述の GIF に使われているコードになっています。

model.parameters[index]の数値が"1"になっていますが、これは rice のパラメータの ParamAngleZ が 1 番目(0 開始)なので"1"と記述しています。



注意点として、Cubism SDK for Cocos Creator を用いスクリプト上からパラメータを更新する場合は **lateUpdate()** を用いる必要があります。 Cocos Creator のアニメーションは `update()` の後に処理されます。そのため、`update()` 内でパラメータの値を変更するとアニメーションに上書きされてしまいます。 これを回避するには、アニメーションの処理が終わった後に呼び出される `lateUpdate()` を用いて、パラメータの更新をする必要があります。

以下の GIF は、アニメーションを再生しているモデルに、ParamAngleZ の値をスクリプトから操作している Scene です。 左のモデルは `update()` から、右は `lateUpdate()` からパラメータの値を操作しており、左は値の変更が上書きされてしまっているのが見て取れます。



TIPS1

例に上げている GIF ですが、これはアニメーションのパラメータを上書きして設定を行っています。 そのため、スクリプトから更新をしているパラメータはアニメーションの動きが適用されていない状態になっています。

もし、アニメーションと掛け合わせてパラメータを使いたい場合には Live2D.Cubism.Framework に含まれている **CubismParameterBlendMode** でブレンドモードを変更する必要があります。ブレンドモードは 3 種類あります。

- Override...パラメータを上書きして更新します
- Additive...パラメータを加算して更新します
- Multiply...パラメータを乗算して更新します

ぜひ、用途に合わせたブレンドモードをご利用ください。 以下はコード例になっています。

```
{
  parameter = model.paramters[index];

  // 上書き
```

```
  CubismParameterExtensionMethods.blendToValue(parameter,
CubismParameterBlendMode.Override, 値);

  // 加算
  CubismParameterExtensionMethods.blendToValue(parameter,
CubismParameterBlendMode.Additive, 値);

  // 乗算
  CubismParameterExtensionMethods.blendToValue(parameter,
CubismParameterBlendMode.Multiply, 値);
}
```

TIPS2

Cubism SDK for Cocos Creator での更新方法で記述した方法でもモデルのパラメータは取得できますが、以下の方法でも特定のパラメータを取得することができます。

これは Live2D.Cubism.Core に含まれている **findByIdFromParameters()** を用いることで CubismParameter を取得するコードです。

```
import { _decorator, Component } from 'cc';
import ArrayExtensionMethods from
'../extensions/live2d_cubismsdk_cocoscreator/static/assets/Core/ArrayExtensionMeth
ods';
import ComponentExtensionMethods from
'../extensions/live2d_cubismsdk_cocoscreator/static/assets/Core/ComponentExtension
Methods';
import CubismModel from
'../extensions/live2d_cubismsdk_cocoscreator/static/assets/Core/CubismModel';
import CubismParameter from
'../extensions/live2d_cubismsdk_cocoscreator/static/assets/Core/CubismParameter';
const { ccclass, property } = _decorator;

@ccclass('ParameterUpdateSample')
export class ParameterUpdateSample extends Component {
  private _model: CubismModel | null = null;
  private _paramAngleZ: CubismParameter | null = null;

  @property({serializable: true})
  public parameterID: string = "ParamAngleZ";

  protected start() {
    this._model = ComponentExtensionMethods.findCubismModel(this, true);

    this._paramAngleZ ??= ArrayExtensionMethods.findByIdFromParameters(
        this._model?.parameters,
        this.parameterID ?? ''
      );
  }
}
```

このコードが書かれたスクリプトを、モデルのルートにアタッチすることで指定したパラメータ(ParameterID)を取得することができます。

TIPS3

その他注意点として、 **CubismUpdateExecutionOrder** でスクリプトの実行順を変えている場合も注意が必要になります。 スクリプトの実行順によっては、パラメータの更新タイミングが変わり、パラメータの更新がうまくいかない場合があります。

Cubism Components の更新

ここでは、［SDK のインポート～モデルを配置］をおこなったプロジェクトを最新パッケージに差し替える方法を説明します。

概要

Cubism SDK の取得は、パッケージと GitHub からの二種類あります。

パッケージには、そのパッケージ公開時の最新の Core ライブラリと CubismComponents、及びそれらを利用したサンプルが含まれています。

Cubism SDK の更新を GitHub から Cocos Creator プロジェクトへ更新を適用するには以下で説明している手順を行います。

まだ Cocos Creator に慣れていない場合は、更新の前に予めプロジェクトのバックアップを取っておいてください。

Tips

SDK の更新は最新の zip を開き、エクスプローラ等でSDKがインストールされた場所へ上書きして更新されます。 プロジェクトのSDKインストール場所は、`[Cocos Creator プロジェクトパス]/extensions/live2d_cubismsdk_cocoscreator/`となります。 SDKを上書きしたら、必ずCocos Creatorの`Extension`メニューから`Extension Manager`を開き、`Reload`をおこなってください。

最新パッケージを取得

Cocos Creator プロジェクトに最新パッケージを適用

既存の Cocos Creator プロジェクトに上書きする際、Cocos Creator 側で Cubism Core を利用し続け更新ができない場合があります。

その場合、以下の手順で更新をおこないます。

1. 現在使用しているシーンをすべて保存します。
2. 保存が完了したら、「Ctrl+N」などで新規シーンを作成します。
3. 新規シーンが作成された状態で、Cocos Creator を再起動します。
4. Cocos Creator が起動したら、最新のパッケージを上書きします。

上書き後、コンパイルエラーが発生していなければ Cocos Creator プロジェクトの Cubism SDK 更新は完了です。

この時点でコンパイルエラーが発生していた場合、以下の原因が考えられます。

- パッケージに含まれるクラス自体をカスタマイズしている
- パッケージに追加されたクラスと同名のクラスが既にプロジェクトに存在している

当たり判定の設定

ここでは、入力された座標からモデルの当たり判定を取得する方法を説明します。［SDK のインポート～モデルを配置］をおこなったプロジェクトに追加することを前提とした説明となっています。

概要

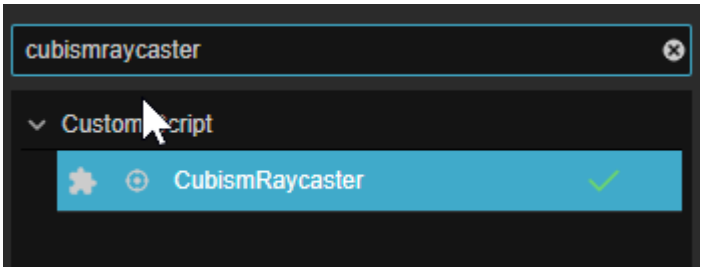
当たり判定を取得するには、Cubism SDKでは、 Raycast というコンポーネントを利用します。

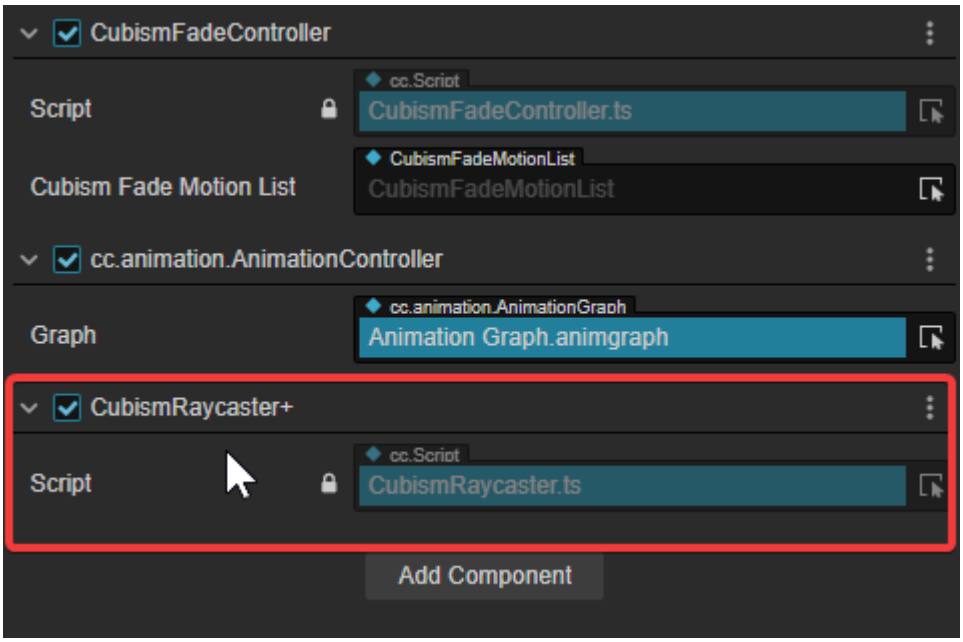
Raycastを設定するには、以下の3つを行ないます。

1. Raycastを実行するコンポーネントのアタッチ
2. 当たり判定に使用するアートメッシュを指定
3. CubismRaycaster.raycastから判定の結果を取得

Raycastを実行するコンポーネントのアタッチ

モデルのルートとなるNodeに、当たり判定の処理を行う［CubismRaycaster］というコンポーネントをアタッチします。

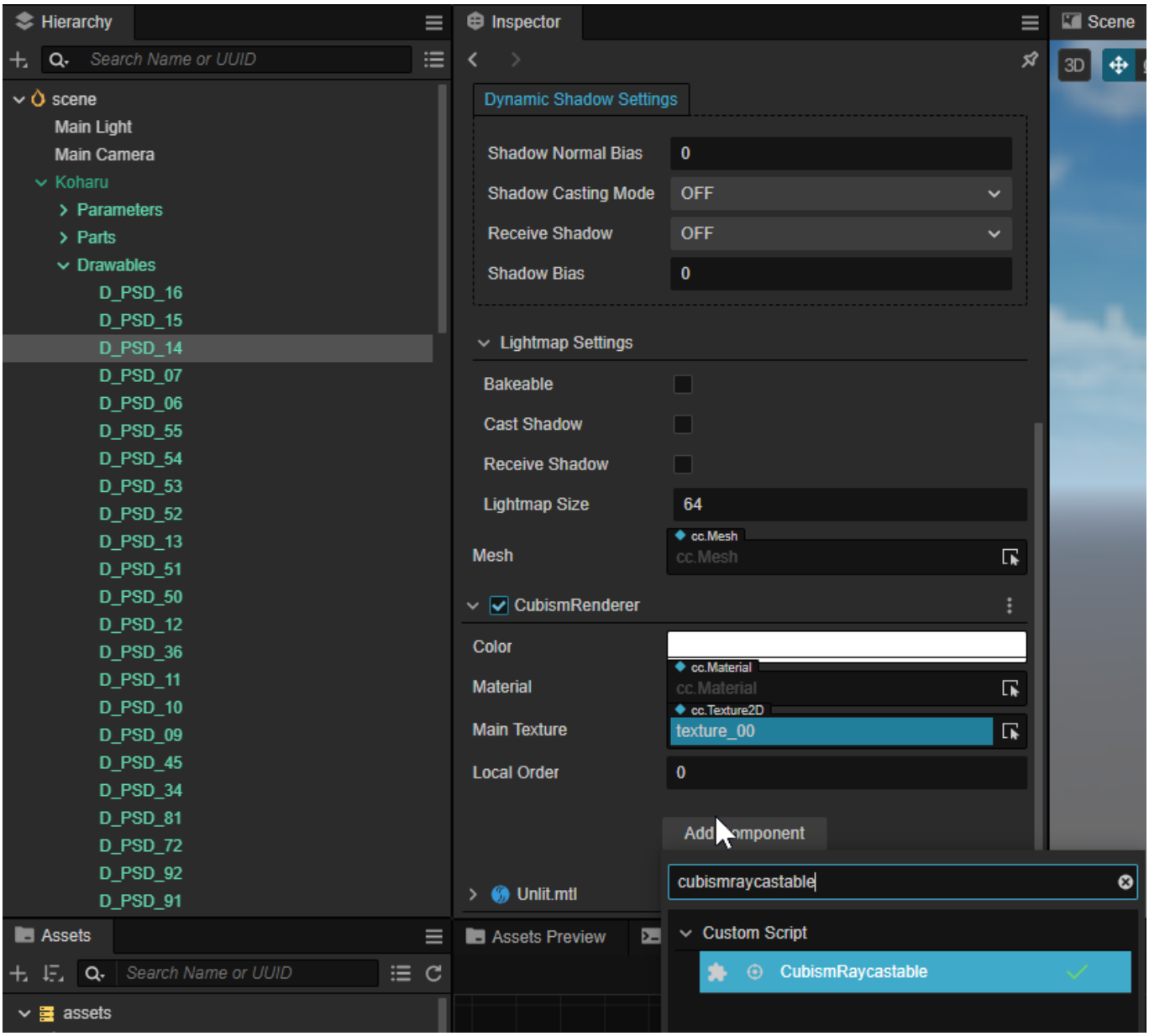




当たり判定に使用するアートメッシュを指定

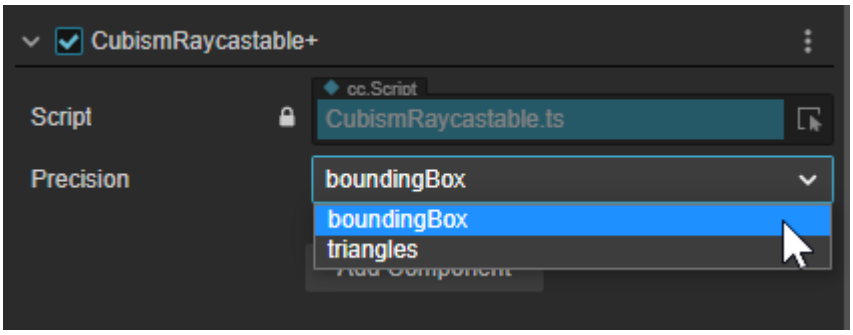
[モデル]/Drawables/ 以下には、描画されるアートメッシュひとつひとつを管理するNodeが配置されています。 Nodeの名前は、そのパラメータのIDとなっています。

このNodeの中から当たり判定用の範囲として扱うものに、［CubismRaycastable］をアタッチします。



CubismRaycastableからは、アタッチしたメッシュの当たり判定の範囲を選ぶことができます。

- Bounding Box : そのメッシュを囲う矩形を当たり判定とします。Trianglesよりも負荷が軽いです。
- Triangles : そのメッシュの形状を当たり判定とします。範囲を正確に判定したい場合はこちらを設定してください。



CubismRaycaster.raycastから判定の結果を取得

最後に、モデルのルートにアタッチした CubismRaycaster の Raycast() を使用して当たり判定の結果を取得します。

「CubismHitTest」というTypeScriptスクリプトを作成し、コードを以下のように書き加えて、モデルのルートにアタッチします。

```
import { _decorator, Component, Node, input, Input, EventTouch, Camera } from
'cc';
import CubismRaycaster from '../../Framework/Raycasting/CubismRaycaster';
import CubismRaycastHit from '../../Framework/Raycasting/CubismRaycastHit';
const { ccclass, property } = _decorator;

@ccclass('CubismHitTest')
export class CubismHitTest extends Component {
  @property({ type: Camera, visible: true })
  public _camera: Camera = new Camera;

  protected start() {
    input.on(Input.EventType.TOUCH_START,this.onTouchStart,this);
  }

  public onTouchStart(event: EventTouch) {
    const raycaster = this.getComponent(CubismRaycaster);

    if (raycaster == null) {
      return;
    }

    // Get up to 4 results of collision detection.
    const results = new Array<CubismRaycastHit>(4);

    // Cast ray from pointer position.
    const ray = this._camera.screenPointToRay(event.getLocationX(),
event.getLocationY());
    const hitCount = raycaster.raycast2(ray, results);

    // Show results.
    let resultsText = hitCount.toString();
    for (var i = 0; i < hitCount; i++)
    {
      resultsText += "\n" + results[i].drawable?.name;
    }

    console.log(resultsText);
  }
}
```

以上で設定は完了です。

この状態で、実行中にGameビューでCubismRaycastableがアタッチされたメッシュをクリックすると、当たり判定の結果をConsoleビューに出力します。

実行例

```
D_PSD_16<CubismDrawable>
D_PSD_15<CubismDrawable>
D_PSD_14<CubismDrawable>
D_PSD_07<CubismDrawable>
```

視線追従の設定

ここでは、モデルの視線などをマウスカーソルに追従させる方法を説明します。

概要

視線の追従を設定するには、Cubism SDKでは、 Lookat というコンポーネントを利用します。

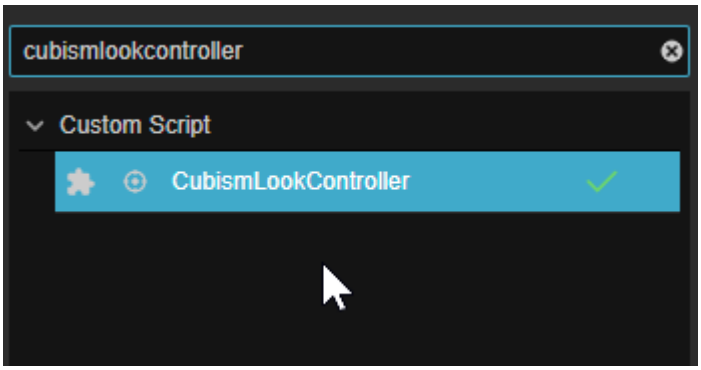
SDKにはこのコンポーネントを使った［LookAt］というサンプルシーンが同梱されておりますので、そちらもあわせてご覧ください。 こちらはモデルの頭上を回るNodeの位置を視線で追従するサンプルとなっております。 /assets/resources/Samples/Lookat

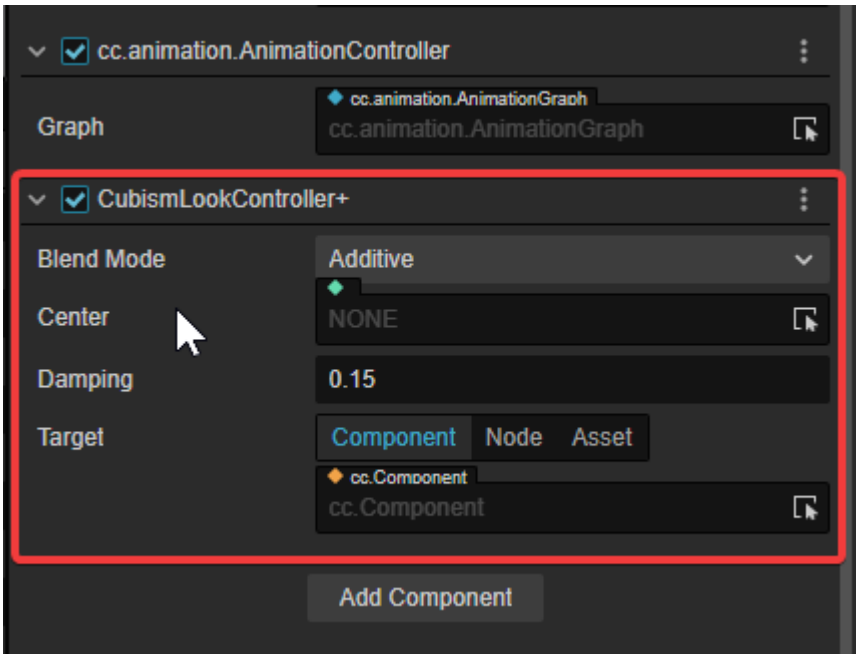
Lookatを設定するには、以下の3つを行ないます。

1. Lookatを管理するコンポーネントのアタッチ
2. 追従させるパラメータの指定
3. 視線を追従させる対象の設定

Lookatを管理するコンポーネントのアタッチ

モデルのルートとなるNodeに、視線追従を管理するCubismLookControllerというコンポーネントをアタッチします。





CubismLookControllerには、設定項目が4つあります。

- Blend Mode：視線追従させる際に変動する値をパラメータにどう反映するかの設定です。設定できるものは以下の3つです。
    - Multiply：現在設定されている値に乗算します。
    - Additive：現在設定されている値に加算します。
    - Override：現在設定されている値を上書きします。
- Center：ここには、追従させる座標の中心として扱うためのNodeを設定します。 中心は、設定されたNodeのBoundsの中心になります。 Centerに設定されるものは、［モデル］/Drawables/以下のNodeになります。 この項目に何も設定しなかった場合、CubismLookControllerがアタッチされたNodeの中心を使用します。
- Damping：対象に追従するまでにかかる時間です。値が小さいほど追従の速度が上がります。
- Target：視線を追従させるターゲットを設定します。詳細は後述しております。

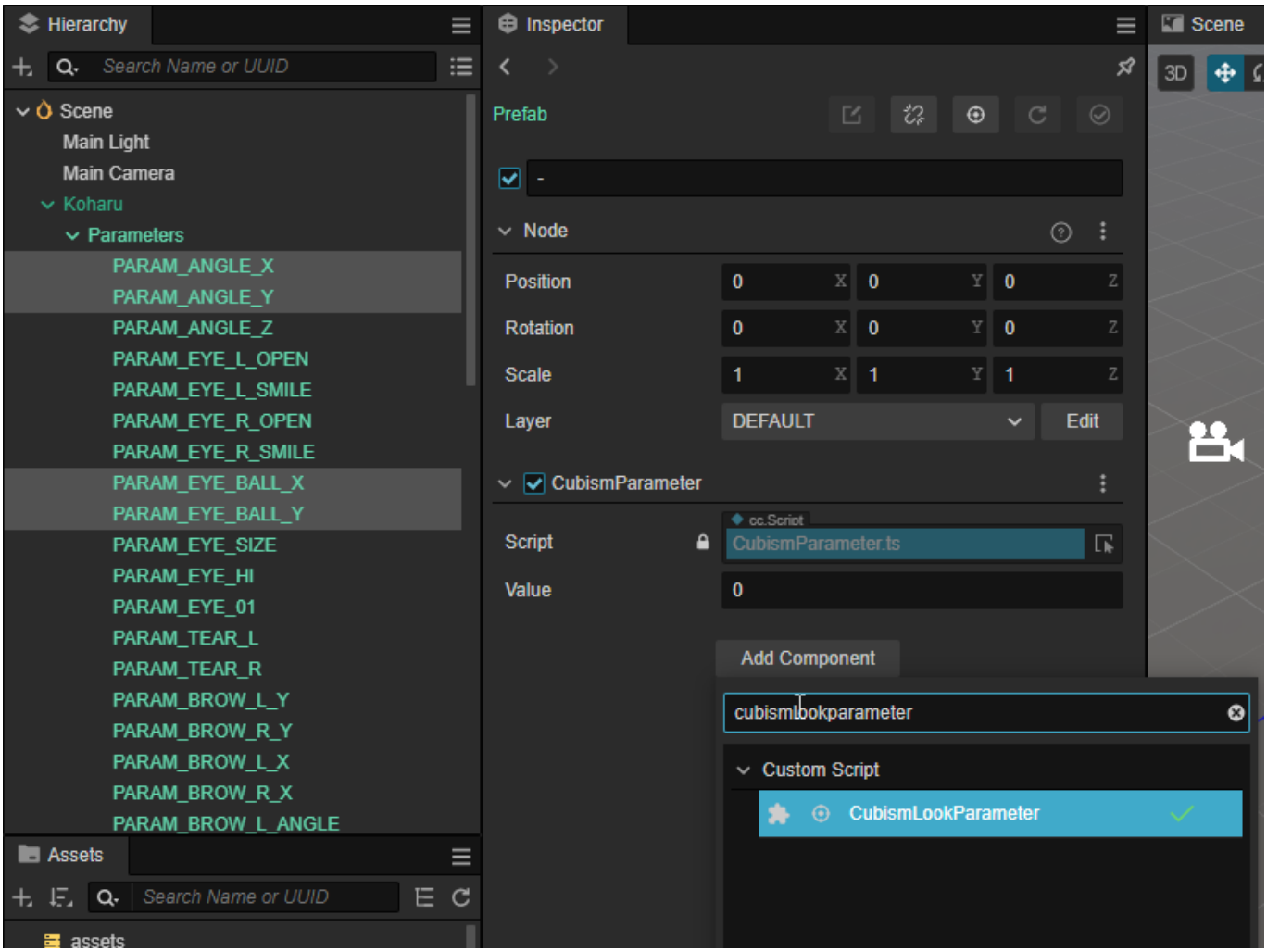
今回は、Blend Mode を［Override］に設定してください。

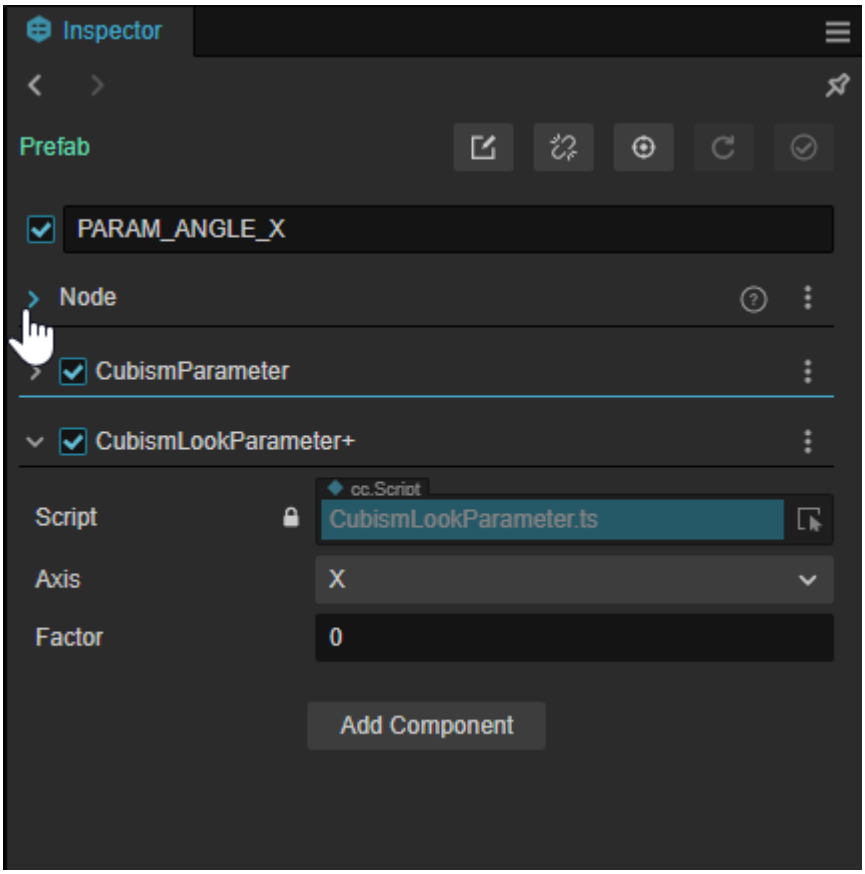
追従させるパラメータの指定

[モデル]/Parameters/ 以下には、そのモデルのパラメータを管理するNodeが配置されています。 また、このNodeに設定されている名前はパラメータのIDとなっております。 これらはCubismModel.parameters で取得できるものと同一です。

このパラメータ用のNodeの中から、追従をさせたいIDのものに、CubismLookParameterというコンポーネントをアタッチします。

パラメータ用のNodeに CubismLookParameter がアタッチされていると、前述のCubismLookControllerがそれを参照して追従させるようになります。





CubismLookParameterには、2つの設定項目があります。

- Axis ： 設定されたパラメータを、何軸の変形として扱うかを指定します。例えば、Xを指定した場合、TargetのX軸の値から計算して設定します。
- Factor ： 計算された値に掛ける数値を設定します。計算結果の値は-1～+1の範囲になるため、 パラメータによってはその範囲を大きくしたり小さくしたり、または+と-を反転したりする方が自然な動きになることがあります。

視線を追従させる対象の設定

最後に、追従させる対象を用意します。

CubismLookControllerコンポーネントの［Target］には、［ICubismLookTarget］インターフェースを実装したコンポーネントを設定します。

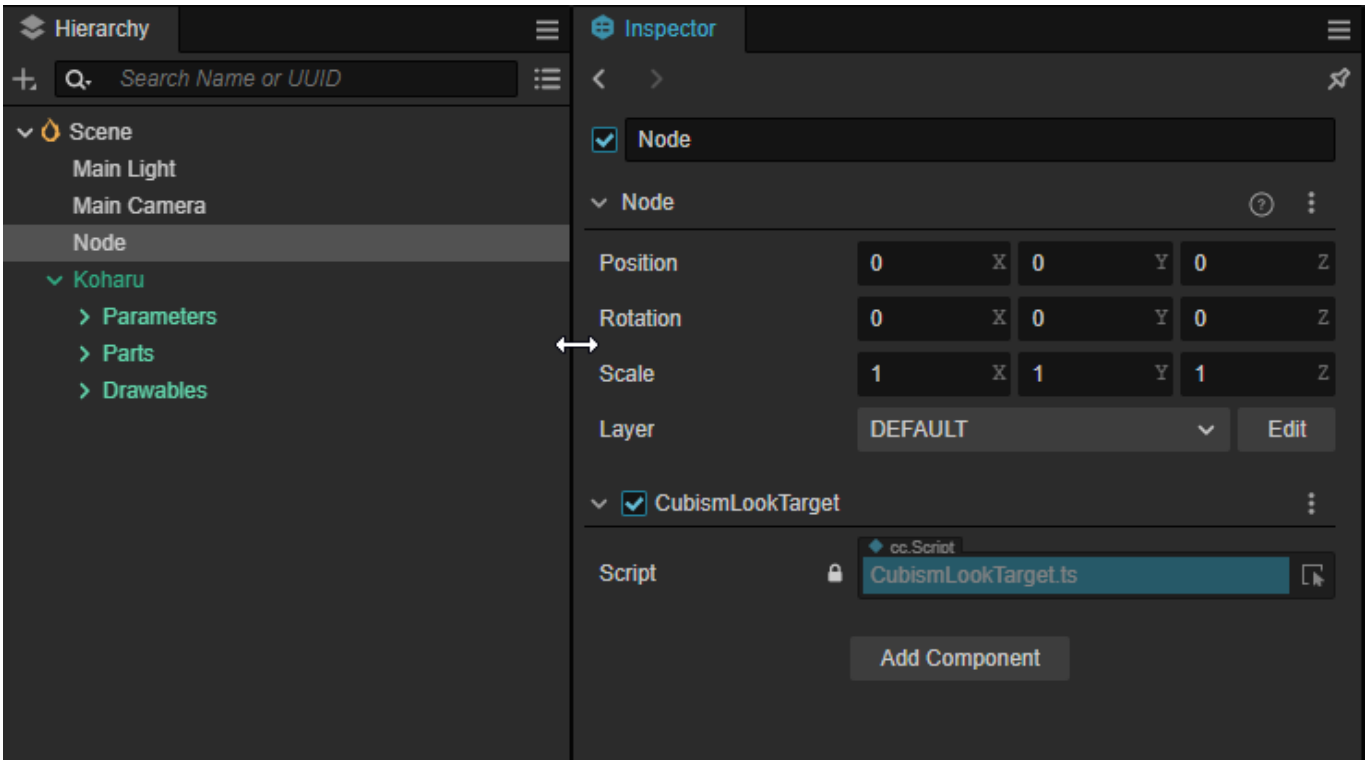
ターゲットの設定次第では、視線を追従させる対象をマウスカーソルやNodeの位置にしたり、ドラッグ中のみ追従させる等特定の条件をつけることも可能です。

「CubismLookTarget」というTypeScriptスクリプトを作成し、コードを以下のように書き換えます。 ここでは、ドラッグ中のマウスの座標を追従するようにしております。

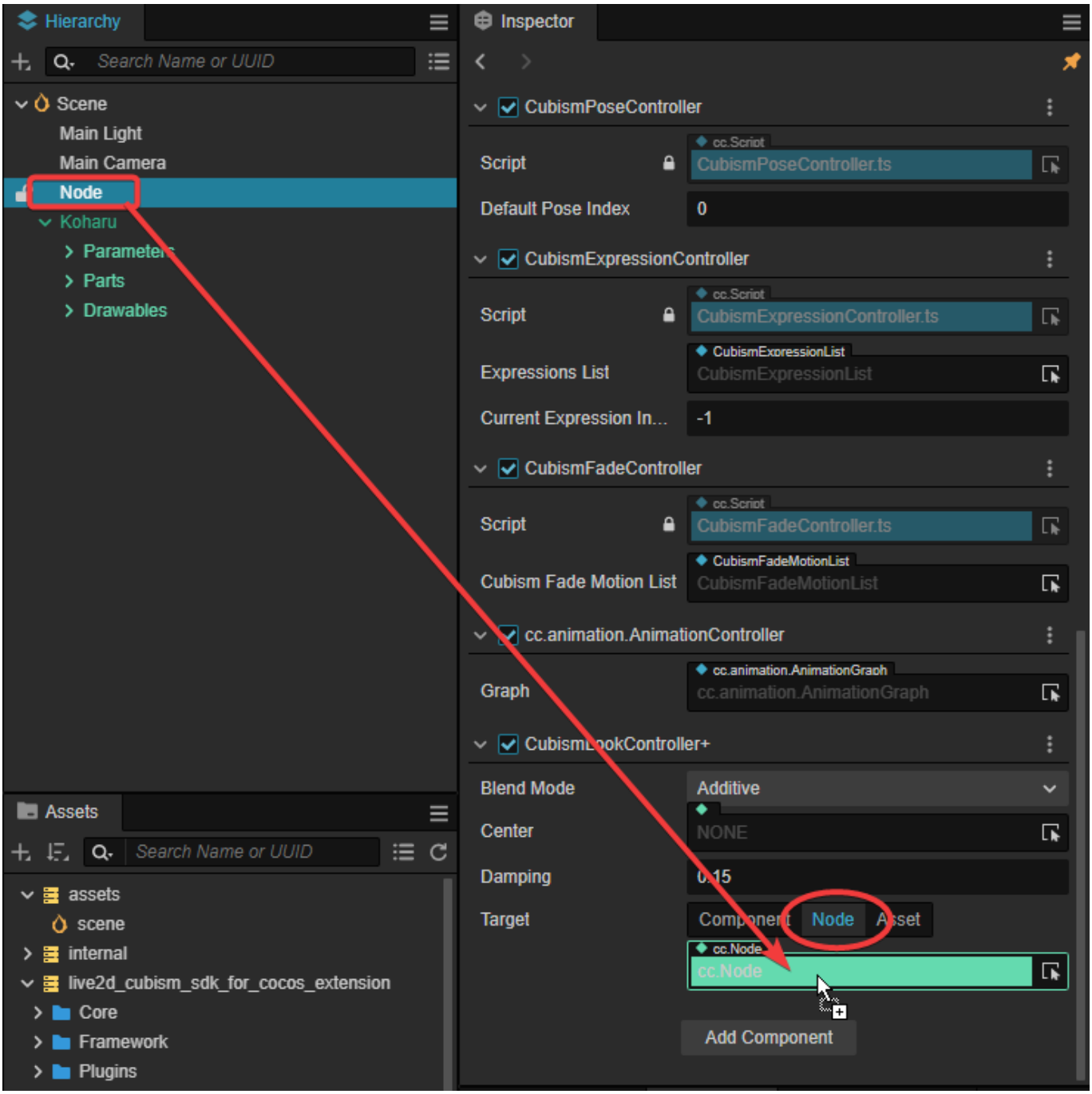
```
import { _decorator, Component, Node, Input, EventTouch, input, math, Vec3,
__private, Camera, Vec2, screen } from 'cc';
import ICubismLookTarget from '../../Framework/LookAt/ICubismLookTarget';
const { ccclass, property } = _decorator;

@ccclass('CubismLookTarget')
export class CubismLookTarget extends Component implements ICubismLookTarget {
  readonly [ICubismLookTarget.SYMBOL]: typeof ICubismLookTarget.SYMBOL =
ICubismLookTarget.SYMBOL;

  private _position: math.Vec3 = Vec3.ZERO;

  public getPosition(): math.Vec3 {
    return this._position;
  }

  public isActive(): boolean {
    return true;
  }

  protected start() {
    input.on(Input.EventType.TOUCH_MOVE,this.onTouchMove,this);
    input.on(Input.EventType.TOUCH_END,this.onTouchEnd,this);
  }

  public onTouchMove(event: EventTouch) {
    let targetPosition = event.getLocationInView();
    targetPosition.x = targetPosition.x / screen.resolution.width;
    targetPosition.y = targetPosition.y / screen.resolution.height;

    targetPosition = targetPosition.multiplyScalar(2).subtract(Vec2.ONE);

    this._position = new Vec3(targetPosition.x, targetPosition.y,
```

```
this._position.z);
  }

  public onTouchEnd(event: EventTouch) {
    this._position = Vec3.ZERO;
  }
}
```

空のNodeを作成し、それに上記のCubismLookTargetをアタッチします。



モデルを選択し、InspectorビューからCubismLookControllerの［Target］に、上で作成したNodeをドラッグ・アンド・ドロップします。



これで設定は完了です。

シーンを実行して、Gameビューをマウスの左ボタンでドラッグすると、モデルの視線が追従します。

自動まばたきの設定

ここでは、モデルに自動でまばたきをさせる方法を説明します。 以下は［SDK のインポート～モデルを配置］をおこなったプロジェクトに追加することを前提とした説明となっています。

概要

モデルに標準パラメータの［左眼　開閉（ParamEyeLOpen）］、［右目　開閉（ParamEyeROpen）］が設定されている場合、インポートして生成されるモデルのPrefabには自動でまばたきの設定が行われます。

上記の設定をしていないモデルに自動まばたきをさせる場合は、本記事で説明する手順で設定することが出来ます。

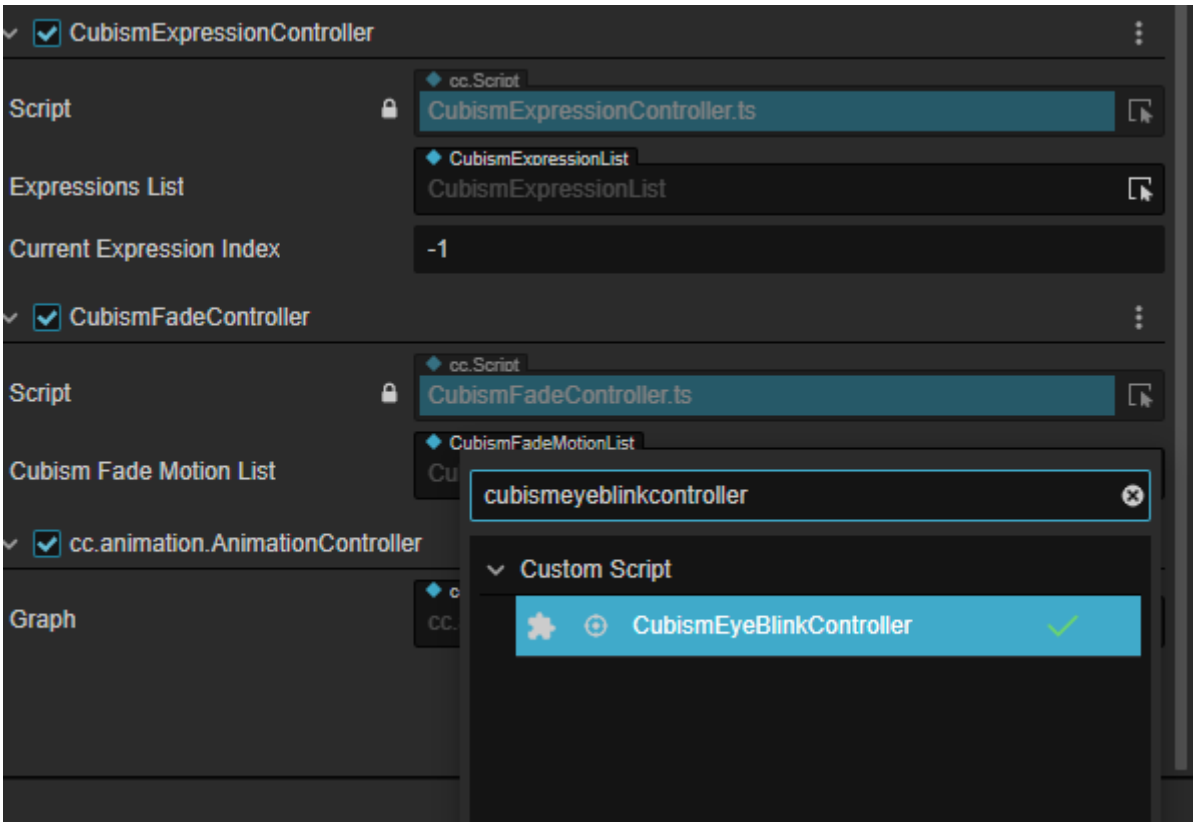
まばたきの設定は、Cubism SDKでは EyeBlink というコンポーネントを利用します。

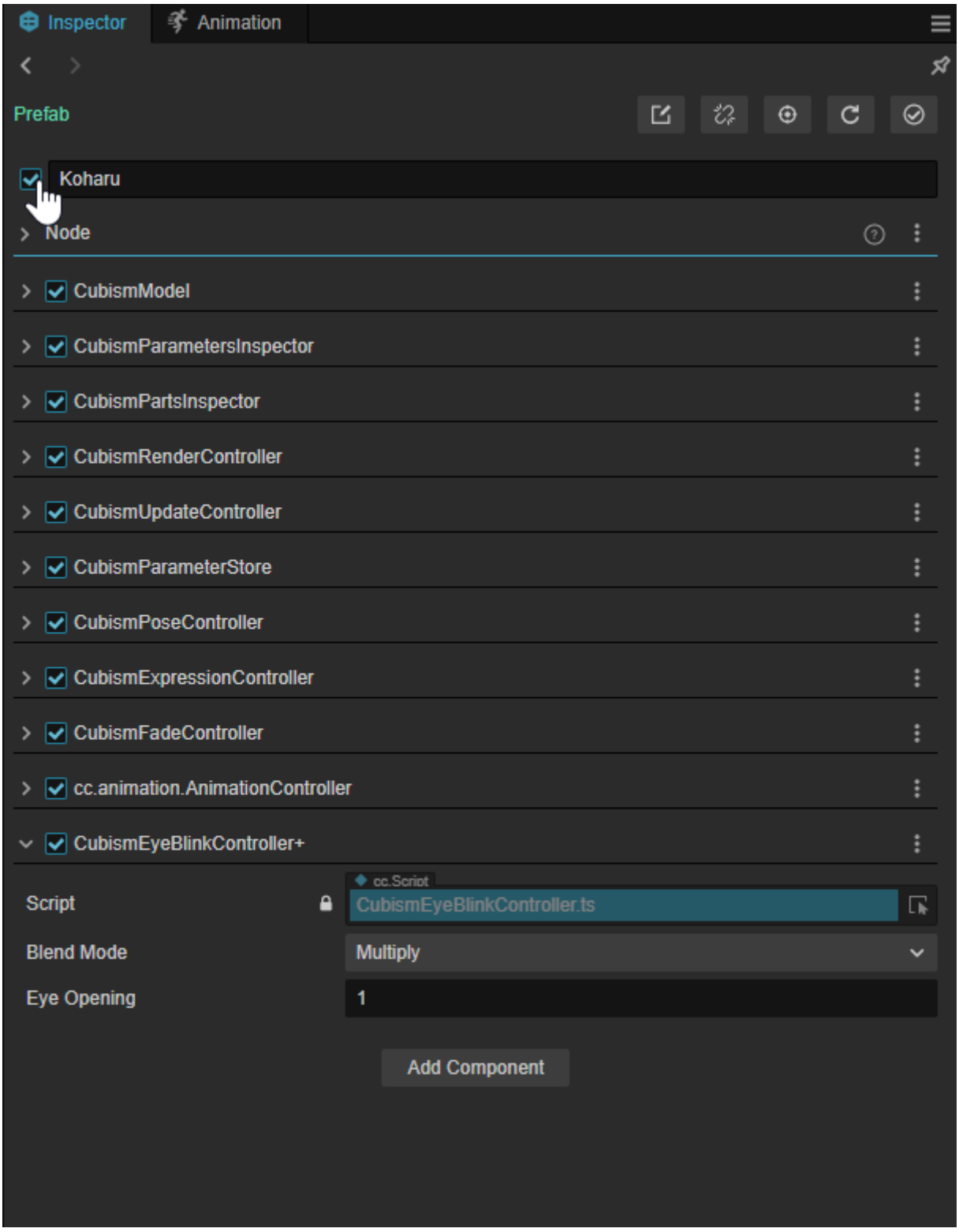
CubismのモデルにEyeBlinkを設定するには、以下の3つを行ないます。

1. まばたきを管理するコンポーネントをアタッチ
2. まばたきをさせるパラメータを指定
3. まばたき用パラメータの値を自動で操作するコンポーネントを設定

まばたきを管理するコンポーネントをアタッチ

まばたきを管理するCubismEyeBlinkControllerというコンポーネントをアタッチします。





CubismEyeBlinkControllerには、設定項目が2つあります。

- Blend Mode：指定のパラメータに現在設定されている値に対して、Eye Openingの値をどう計算するのかを指定します。
    - Multiply：現在設定されている値にEye Openingの値を乗算します。
    - Additive：現在設定されている値にEye Openingの値を加算します。
    - Override：現在設定されている値をEye Openingの値で上書きします。
- Eye Opening：目の開閉の値です。1で開いた状態、0で閉じた状態として扱います。この値が外から操作されると、指定されたパラメータの値も連動します。

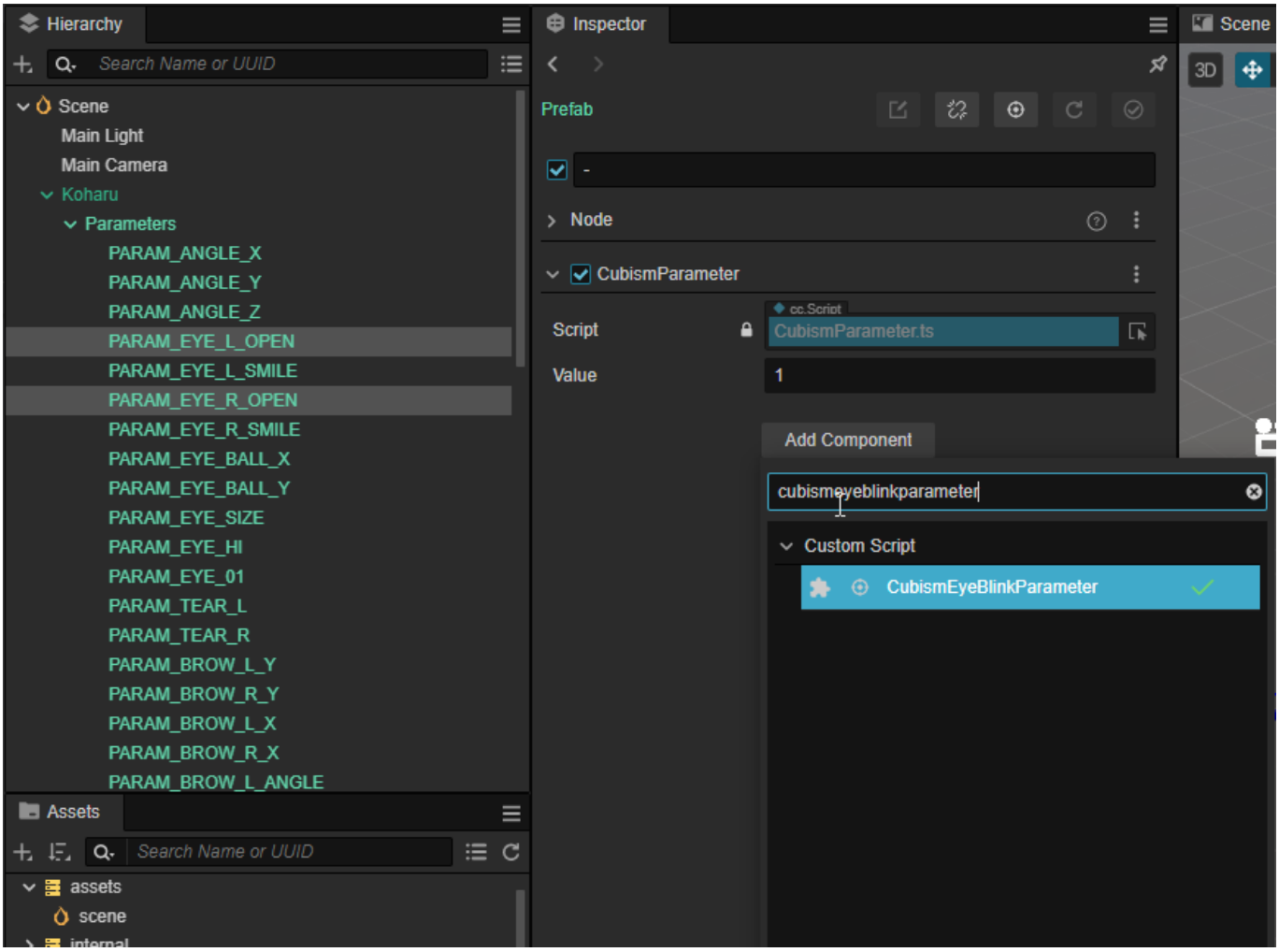
今回は、Blend Modeは［Override］に設定してください。

まばたきをさせるパラメータを指定

[モデル]/Parameters/ 以下には、そのモデルのパラメータを管理するNodeが配置されています。 また、このNodeに設定されている名前はパラメータのIDとなっております。 これらはCubismModel.parameters() で取得できるものと同一です。

このパラメータ用のNodeの中から、まばたきとして扱うIDのものに、CubismEyeBlinkParameterというコンポーネントをアタッチします。

パラメータ用のNodeに CubismEyeBlinkParameter がアタッチされていると、シーン実行時にCubismEyeBlinkControllerがそれを参照して目の開閉を設定するようになります。

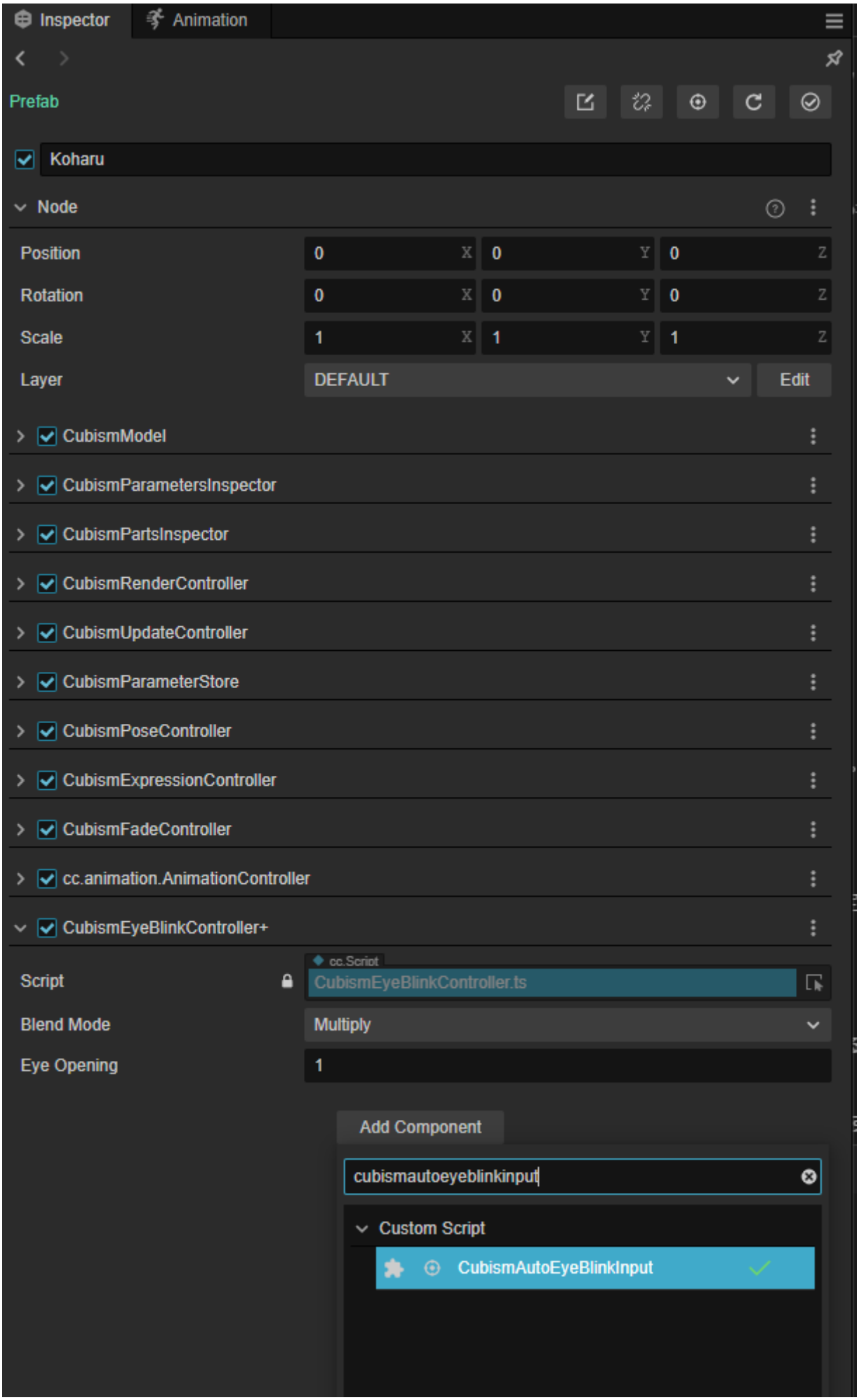


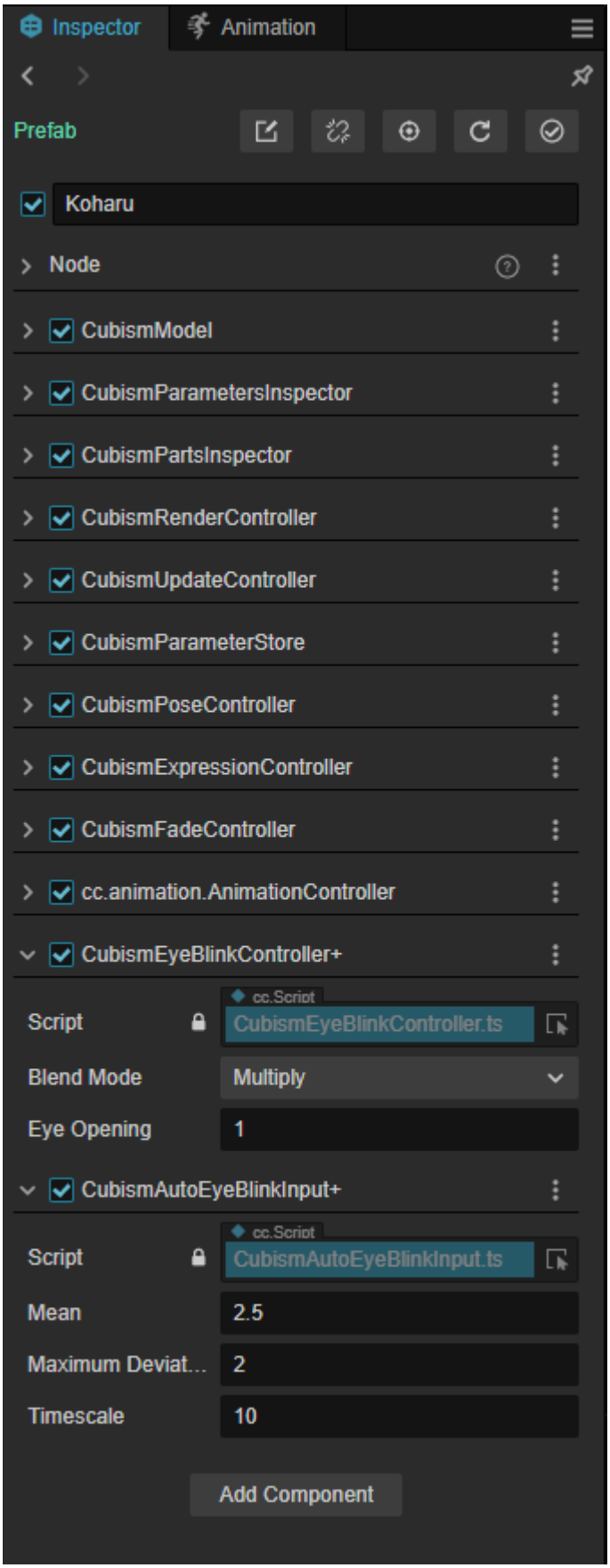
以上までの設定で、スクリプトなどからまばたきの操作を行うことができるようになりますが、これだけではまだ自動でまばたきは行いません。

自動でまばたきをさせるためには、定期的に値を操作するコンポーネントも設定する必要があります。

まばたき用パラメータの値を自動で操作するコンポーネントを設定

CubismEyeBlinkControllerと同じく、モデルのルートにCubismAutoEyeBlinkInputというコンポーネントをアタッチします。


Inspector
Animation
Prefab
Koharu
Node
Position
0 X 0 Y 0 Z
Rotation
0 X 0 Y 0 Z
Scale
1 X 1 Y 1 Z
Layer
DEFAULT
Edit
CubismModel
CubismParametersInspector
CubismPartsInspector
CubismRenderController
CubismUpdateController
CubismParameterStore
CubismPoseController
CubismExpressionController
CubismFadeController
cc.animation.AnimationController
CubismEyeBlinkController+
cc.Script
Script
CubismEyeBlinkController.ts
Blend Mode
Multiply
Eye Opening
1
Add Component
cubismautoeyeblinkinput
Custom Script
CubismAutoEyeBlinkInput



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

- Mean：まばたきを行うまでの時間を設定します。単位は秒です。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。
- Maximum Deviation：Meanで設定した時間に加える揺らぎの最大値を設定します。設定する値は0以上の数値です。

上記のMeanに設定した時間でまばたきをさせると、周期が均一になるため、キャラクターの動作としては不自然になってしまいます。
そのため、設定した時間の周期にランダムで揺らぎを加え、動作を自然なものにしています。
実際の計算は以下のようになっています。

```
次のまばたきまでの時間 = Mean + (-Maximum Deviation から +Maximum Deviation間のランダムな値)
```

- Timescale：目の開閉する速度を設定します。設定する数値が小さくなるほどゆっくりになります。

今回はMeanに2.5を、 Maximum Deviationに2、Timescaleに10を設定します。

以上で自動まばたきの設定は完了です。

この状態でSceneを実行すれば、自動でまばたきをさせることができます。

リップシンクの設定

ここでは、モデルにAudioSourceの音量からリップシンク（口パク）をさせる方法を説明します。 以下は［SDK のインポート～モデルを配置］をおこなったプロジェクトに追加することを前提とした説明となっています。

AudioSourceから音量を取得して設定するため、別途Cocos Creatorで扱える形式の音声ファイルをご用意ください。

概要

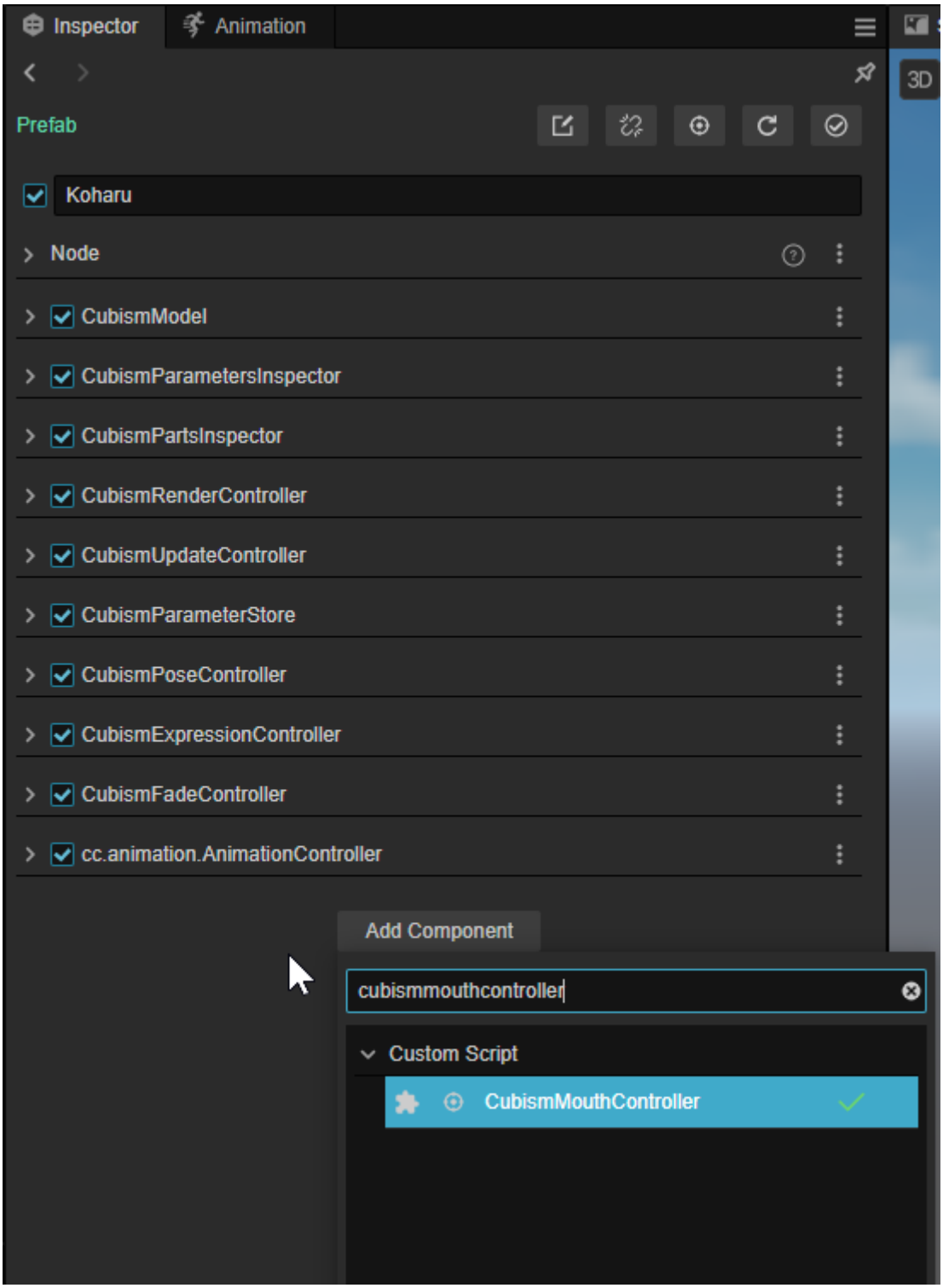
リップシンクの設定は、Cubism SDKでは MouthMovement というコンポーネントを利用します。

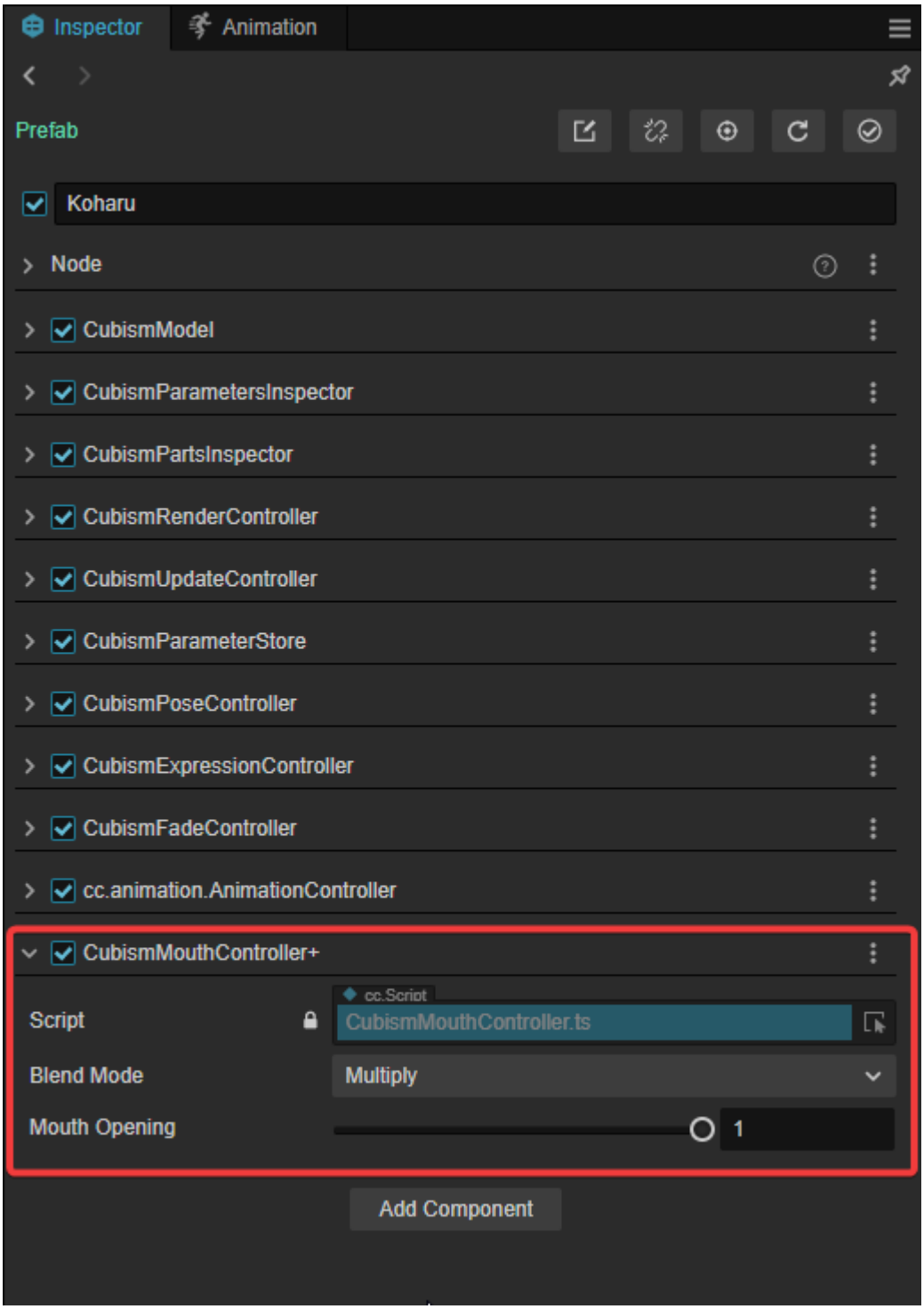
CubismのモデルにMouthMovementを設定するには、以下の3つを行ないます。

1.リップシンクを管理するコンポーネントをアタッチ 2.リップシンクをさせるパラメータを指定 3.指定されたパラメータの値を操作するコンポーネントを設定

リップシンクを管理するコンポーネントをアタッチ

モデルのルートとなるNodeに、リップシンクを管理するCubismMouthControllerというコンポーネントをアタッチします。

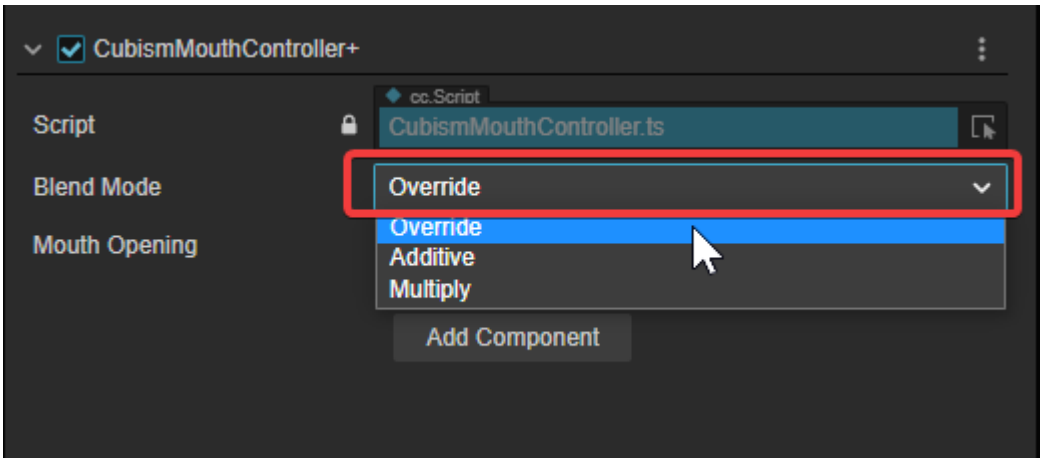

Inspector
Animation
3D
Prefab
Koharu
Node
CubismModel
CubismParametersInspector
CubismPartsInspector
CubismRenderController
CubismUpdateController
CubismParameterStore
CubismPoseController
CubismExpressionController
CubismFadeController
cc.animation.AnimationController
Add Component
cubismmouthcontroller
Custom Script
CubismMouthController



CubismMouthControllerには、設定項目が2つあります。

- Blend Mode：指定のパラメータに現在設定されている値に対して、Mouth Openingの値をどう計算するのかを指定します。
    - Multiply：現在設定されている値にMouth Openingの値を乗算します。
    - Additive：現在設定されている値にMouth Openingの値を加算します。
    - Override：現在設定されている値をMouth Openingの値で上書きします。
- Mouth Opening：口の開閉の値です。1で開いた状態、0で閉じた状態として扱います。この値が外から操作されると、指定されたパラメータの値も連動します。

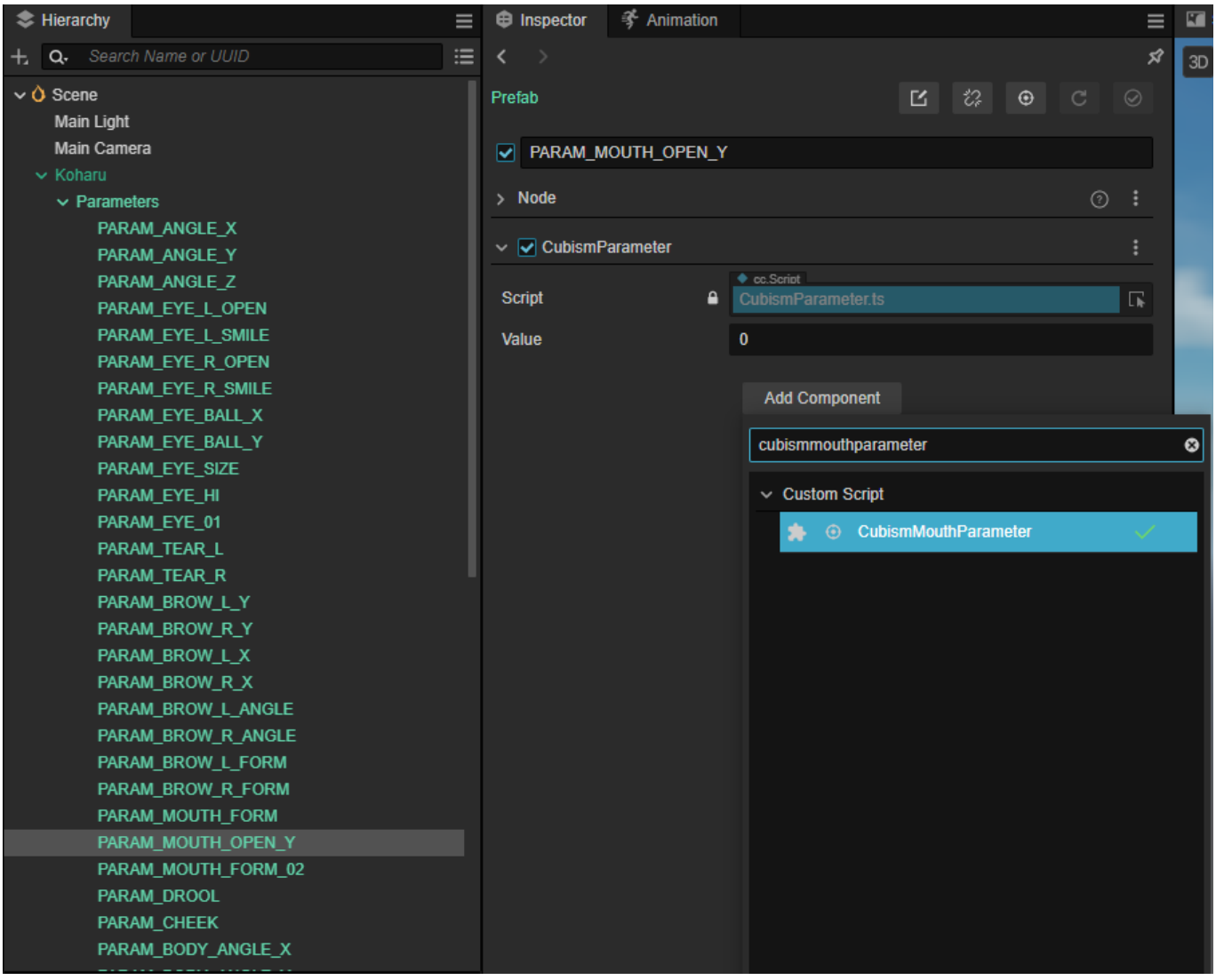
今回は、Blend Modeは［Override］に設定してください。

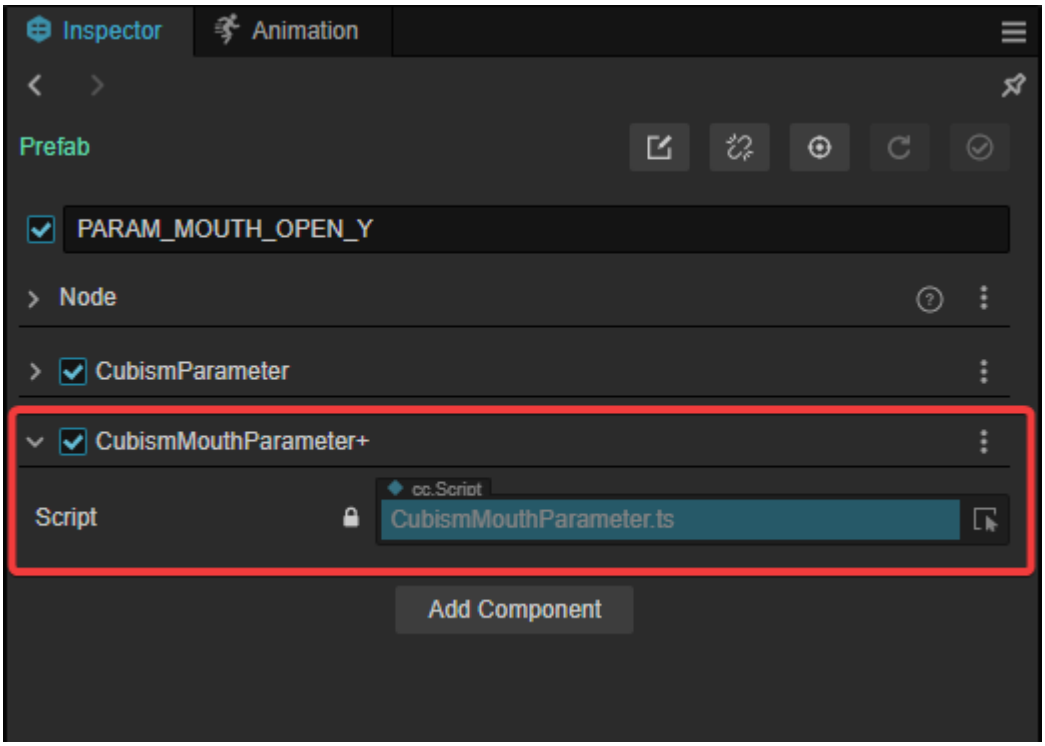


リップシンクをさせるパラメータを指定

[モデル]/Parameters/ 以下には、そのモデルのパラメータを管理するNodeが配置されています。 また、このNodeに設定されている名前はパラメータのIDとなっております。 これらのNodeにアタッチされているCubismParameterは、CubismModel.parameters で取得できるものと同一です。

このNodeの中から、リップシンクとして扱うIDのものに、CubismMouthParameterというコンポーネントをアタッチします。





以上までの設定で、口の開閉の操作を行うことができるようになります。 ですが、これだけではまだ実行してもリップシンクはしません。

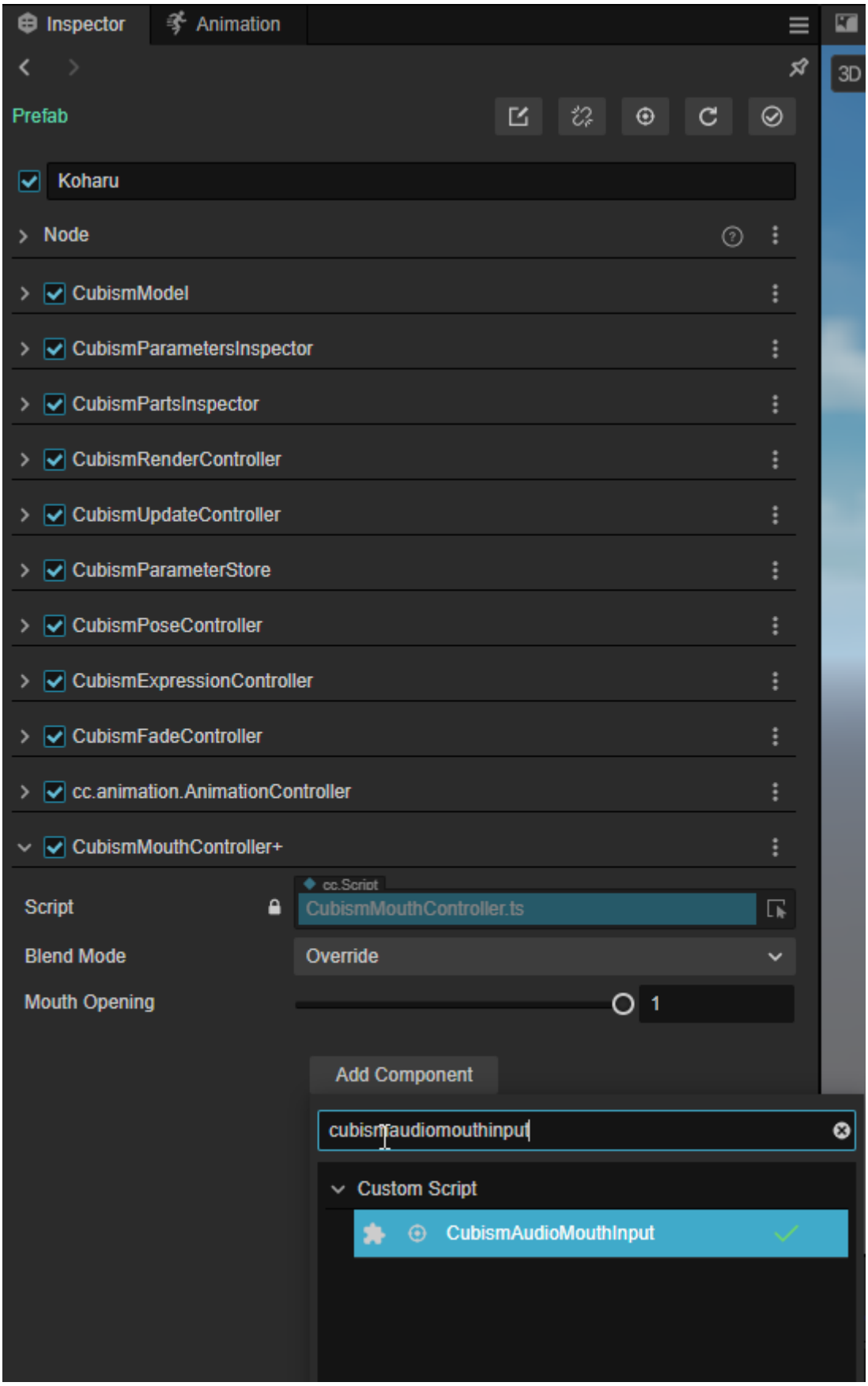
自動でリップシンクさせるためには、CubismMouthControllerのMouth Openingの値を操作するコンポーネントも設定する必要があります。

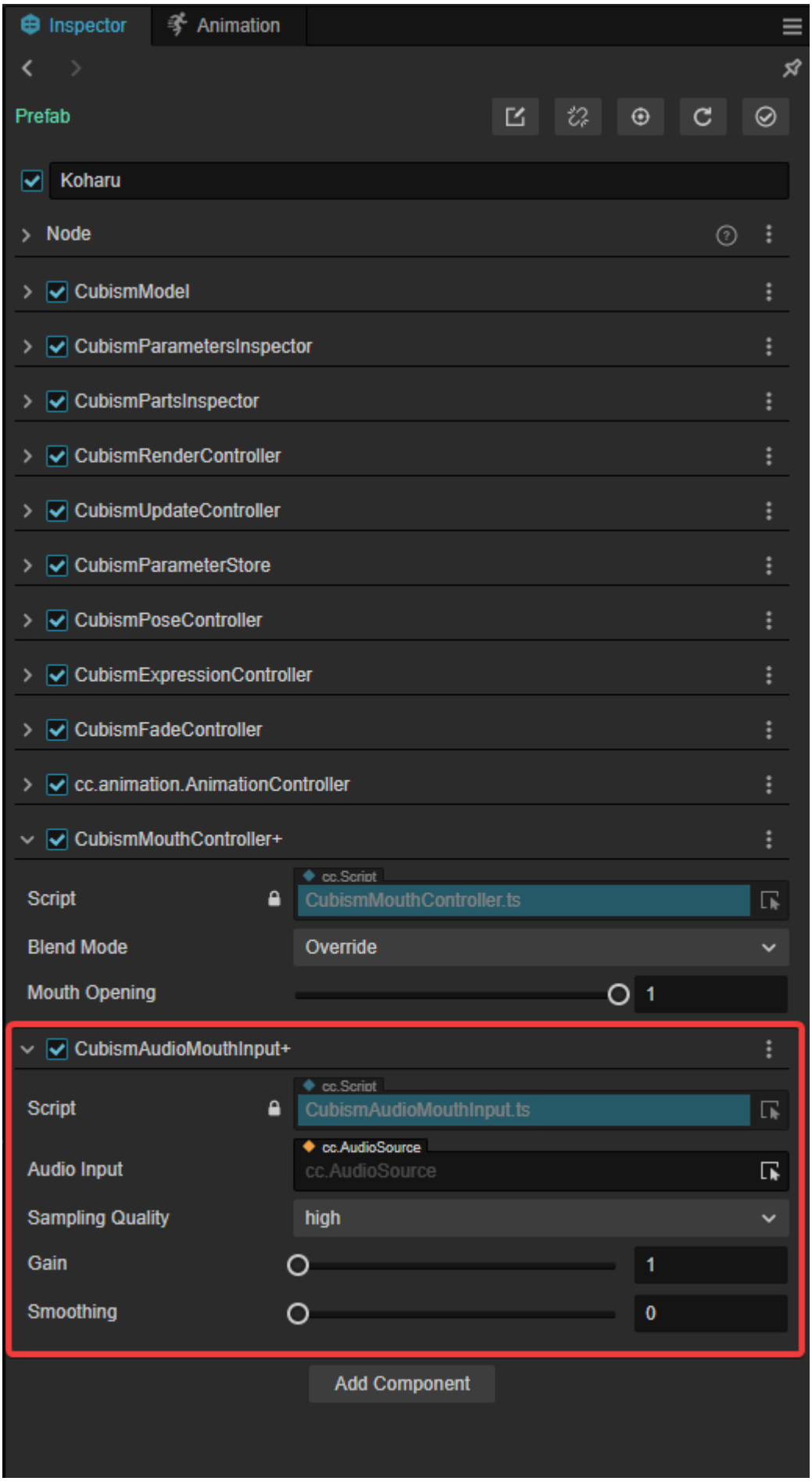
指定されたパラメータの値を操作するコンポーネントを設定

モデルのルートとなるNodeに、リップシンクの入力用コンポーネントをアタッチします。

MouthMovementには、入力のサンプルとして、AudioSourceのボリュームやSin波から口の開閉の値を操作するコンポーネントが含まれています。

今回はAudioSourceから値を設定できるよう、CubismAudioMouthInputというコンポーネントをアタッチします。


Inspector
Animation
3D
Prefab
Koharu
Node
CubismModel
CubismParametersInspector
CubismPartsInspector
CubismRenderController
CubismUpdateController
CubismParameterStore
CubismPoseController
CubismExpressionController
CubismFadeController
cc.animation.AnimationController
CubismMouthController+
cc.Script
Script
CubismMouthController.ts
Blend Mode
Override
Mouth Opening
1
Add Component
cubismaudiomouthinput
Custom Script
CubismAudioMouthInput



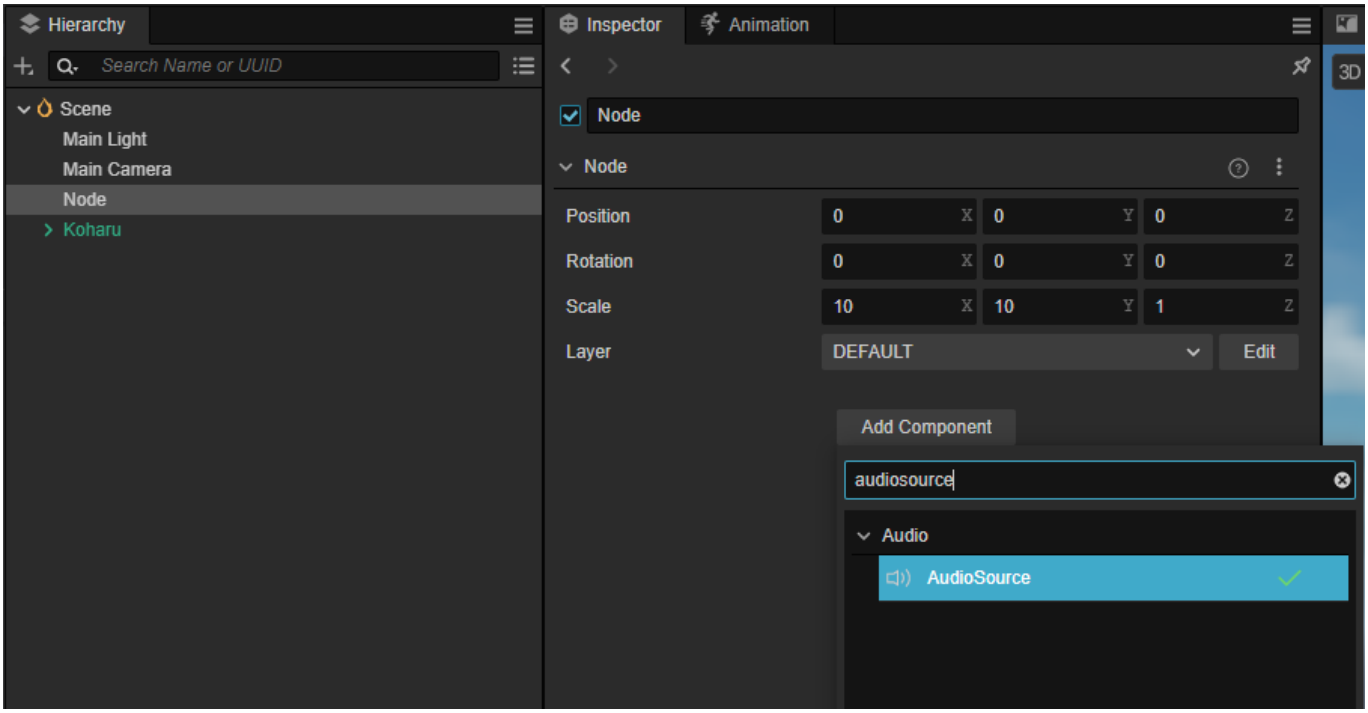
CubismAudioMouthInputには、以下の4つの設定項目があります。

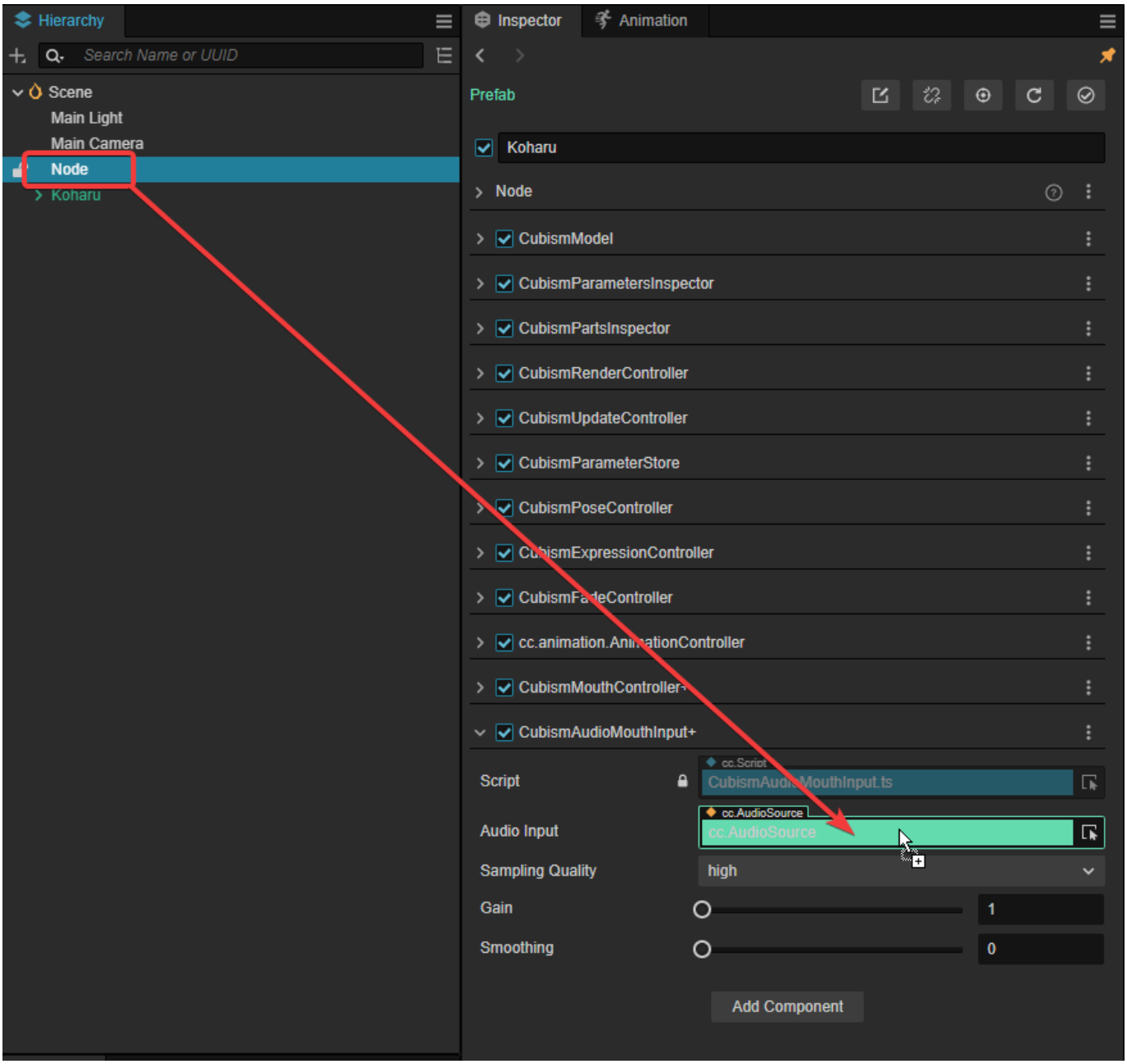
- Audio Input：入力に使用するAudioSourceを設定します。ここで設定したAudioSourceにセットされたAudioClipの音量を使用します。
- Sampling Quality：サンプリングする音量の精度を設定します。以下の設定の並びが下になるほど精度が上がりますが、計算の負荷も上がります。
    - High
    - Very High
    - Maximum
- Gain：サンプリングした音量を、何倍した値で扱うかを設定します。1で等倍です。
- Smoothing：音量から算出する開閉の値をどのくらいスムージングするかを設定します。値を大きくするほど滑らかになりますが、計算の負荷も上がります。

今回は、設定を以下のようにします。Audio Inputには、AudioSourceをアタッチしたNodeを設定しておいてください。

- Sampling Quality：High
- Gain：1
- Smoothing：5

最後に、音量を取得するために、任意のNodeにAudioSourceをアタッチし、上記のCubismAudioMouthInputのAudio Inputに設定します。





以上でリップシンクの設定は完了です。

この状態でシーンを実行すると、AudioSourceに設定された音声ファイルが再生され、その音量に合わせてモデルがリップシンクするようになります。

パラメータを周期的に動作させる方法

ここでは、呼吸や振り子のように、パラメータの値を周期的に動かす設定方法を説明します。

概要

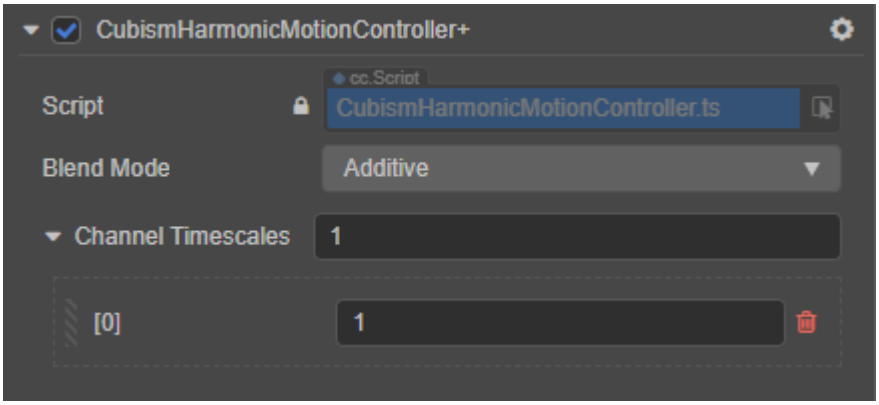
任意のパラメータを周期的に動かしたい場合は、 **CubismHarmonicMotionController** .ts と **CubismHarmonicMotionParameter** .ts を使うことで実現できます。

実装には以下の 2 つを行います

1. パラメータを制御するための設定を行う
2. 動かすパラメータを指定する

パラメータを制御するための設定を行う

最初にモーションを制御するためのスクリプトをモデルの一番親にアタッチします。 名前は **CubismHarmonicMotionController** .ts です。



CubismHarmonicMotionController.ts には設定項目が 2 つあります。

- Blend Mode：指定のパラメータに現在設定されている値に対して、どう計算するのかを指定します。
  - Override：現在設定されている値を数値を上書きします。
  - Additive：現在設定されている値に数値を加算します。

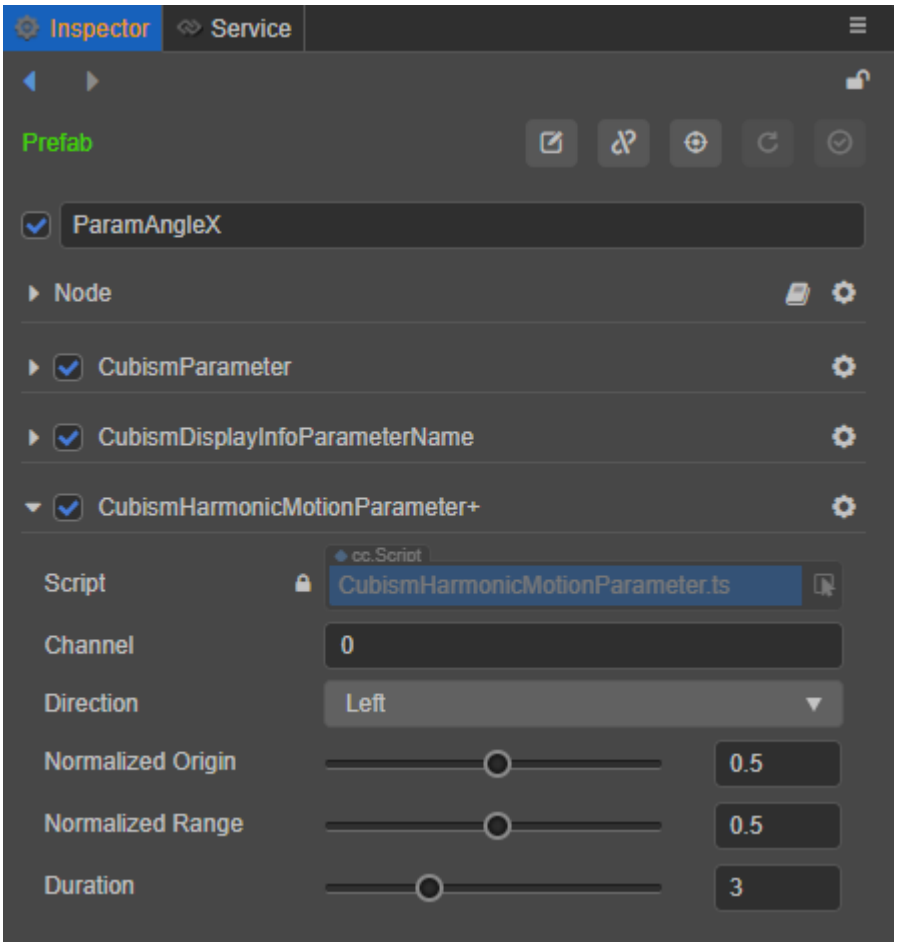
    - Mutiply：現在設定されている値に数値を乗算します。
- Channel Timescales：任意で複数タイムスケールを作り、タイムスケールのサイズを変更することができます。

今回は、Blend Mode の設定を以下のようにします。

- Blend Mode：Override

動かすパラメータを指定する

その後、動かしたいパラメータを選択し **CubismHarmonicMotionParameter** .ts をアタッチします。 ここでは角度 X にアタッチします。 角度 X は[root]/Parameters/ParamAngleX になります。



CubismHarmonicMotionParameter.ts には設定項目が 5 つあります。

- Channel：CubismHarmonicMotionController.ts で設定した Channel Timescale を指定します。
- Direction：パラメータの中心を基準に動く幅を指定します。
    - Left：パラメータの中心から左半分だけ動きます。
    - Right：パラメータの中心から右半分だけ動きます。
    - Centric：パラメータの中心から全体が動きます。
- Normalized Origin：中心にするパラメータの位置を設定します。
- Normalized Range：Normalized Origin で決めた中心を基準に、中心点から動かす最大距離を設定します。
- Duration：パラメータ周期を調整します。

今回は、設定を以下のようにします。

- Channel：0
- Direction：Centric
- Normalized Origin：0.5
- Normalized Range：0.5
- Duration：3

上記設定を行うと、以下の動画のようにパラメータを周期的に動かすことができます。

アートメッシュに設定されたユーザデータを取得

ここでは、モデルからアートメッシュに設定されている「UserData」の情報を取得するまでの説明を行います。

概要

［UserData］は、Cubism 3.1 から追加された機能で、ユーザがアートメッシュに任意のメタデータを付与することができます。 ユーザデータの扱い方によっては、そのアートメッシュを当たり判定に指定したり、特別なシェーディングを施すなど、様々なことに活用できます。

アートメッシュへのユーザデータの設定方法については こちら をご覧下さい。

アートメッシュに設定されたユーザデータの取得には、以下で説明する手順を行ないます。

具体的な記述例としては

```
import { _decorator, Component, Node } from 'cc';
import CubismUserDataTag from
'../extensions/Live2DCubismSdkForCocosExtension/static/assets/Framework/UserData/C
ubismUserDataTag';
const { ccclass, property } = _decorator;

@ccclass('UserDataTest')
export class UserDataTest extends Component {
  protected start() {
    let userDatas = this.getComponentsInChildren(CubismUserDataTag);

    for (let i = 0; i < userDatas.length; ++i) {
      let data = userDatas[i];
      console.log("id:"+data.node.name + "\n"
      +"value:"+data.value);
    }
  }
}
```

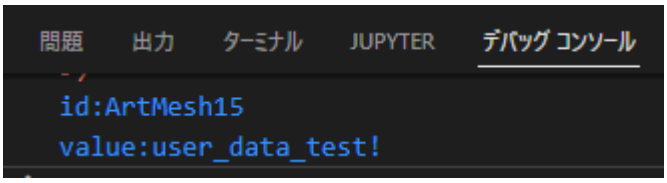
という形になります。

その後、ユーザデータが設定されたモデルの Prefab を Hierarchy に配置します。 配置した Prefab のルートに、上で作成した UserDataTest コンポーネントをアタッチします。

モデルのインポートについては こちら のチュートリアルを参照してください。

以上で設定は終了です。

この状態で Scene を実行すると、Console ウィンドウに、ユーザデータが設定されたアートメッシュの ID と、ユーザデータに設定された文字列が出力されます。


問題
出力
ターミナル
JUPYTER
デバッグ コンソール
id:ArtMesh15
value:user_data_test!

motion3.jsonに設定されたイベントを取得

ここでは、motion3.jsonに設定されたイベントの情報を、自動生成されるAnimationClipから発行されるイベントから取得する方法を説明します。

概要

［UserData］は、ユーザがモーションの任意のタイミングでイベントを発行することができます。 Cocos Creatorでは、motion3.jsonから変換されるAnimationClipからイベントが発行されます。 イベントの扱い方によっては、モーションの途中から音声を再生させたり、パーツの表示状態を変更するなど、様々なことに活用できます。

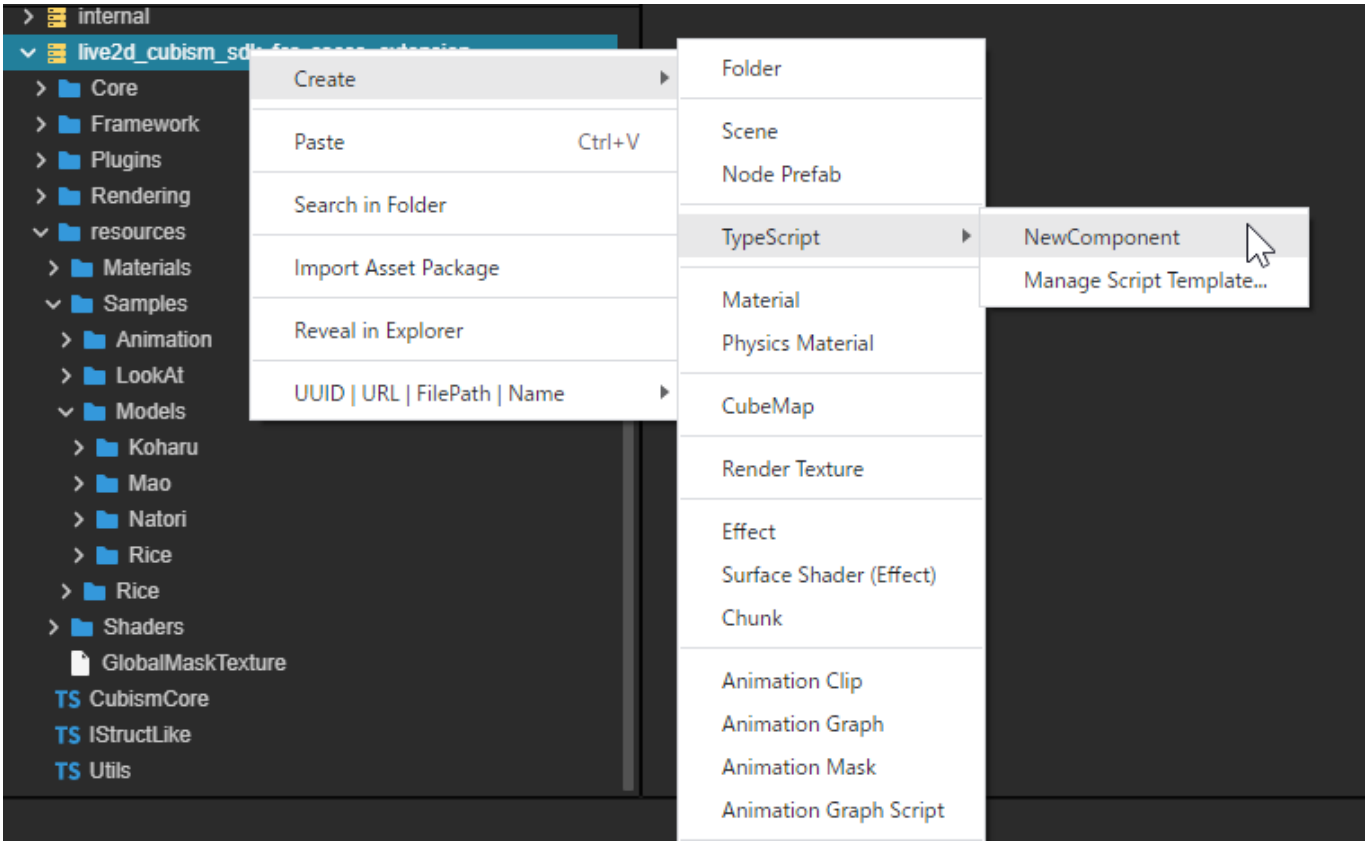
motion3.jsonへのイベントの設定方法については こちら をご覧下さい。

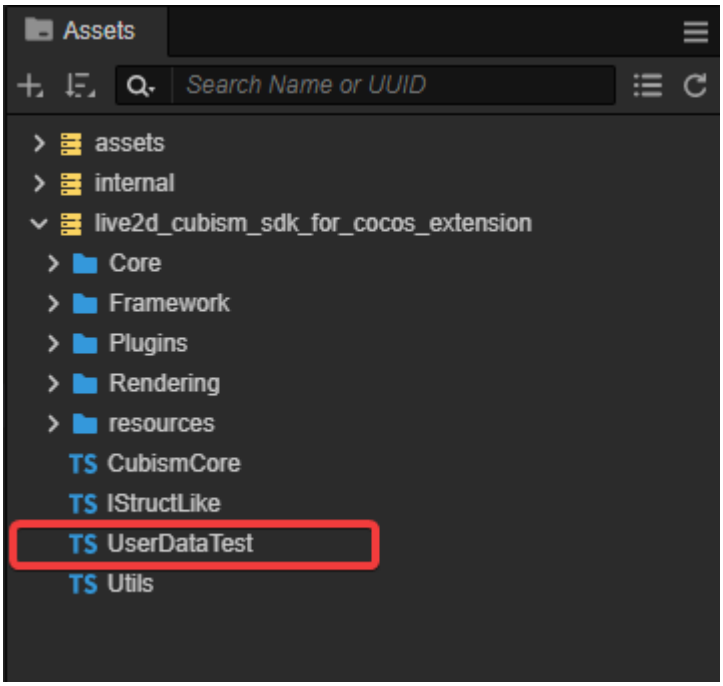
モーションから発行されるイベントの取得には、以下の手順を行ないます。

1. イベントを受け取るコンポーネントの作成
2. CubismMotion3Jsonに記述を追加
3. モデルとモーションをSceneに配置

イベントを受け取るコンポーネントの作成

Assetsウィンドウを右クリックし、［Create］-［TypeScript］-[New Component]をクリックしてTypeScriptスクリプトを作成します。 ここでは名前はUserDataTestとします。





作成したUserDataTestの中身を以下のように書き換えます。

```
import { _decorator, Component, Animation } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('UserDataTest')
export class UserDataTest extends Component {

  UserDataEventListener(value: String) {

    const anim = this.getComponents(Animation)[0];
    const currentState = anim.getState(anim.defaultClip!!.name)

    console.log("Time: " + currentState.current + "\n" + "Value: " + value);
  }

}
```

CubismMotion3Jsonに記述を追加

CubismMotion3Jsonクラスを開き、 toAnimationClipB() のAnimationEventを作成している箇所を以下のように記述します。

```
const frame = this.userData[i].time;
const functionName = `UserDataEventListener`
const params = new Array<string>();
params.push(this.userData[i].value);
animationClip.events.push({ frame: frame, func: functionName, params: params });
```

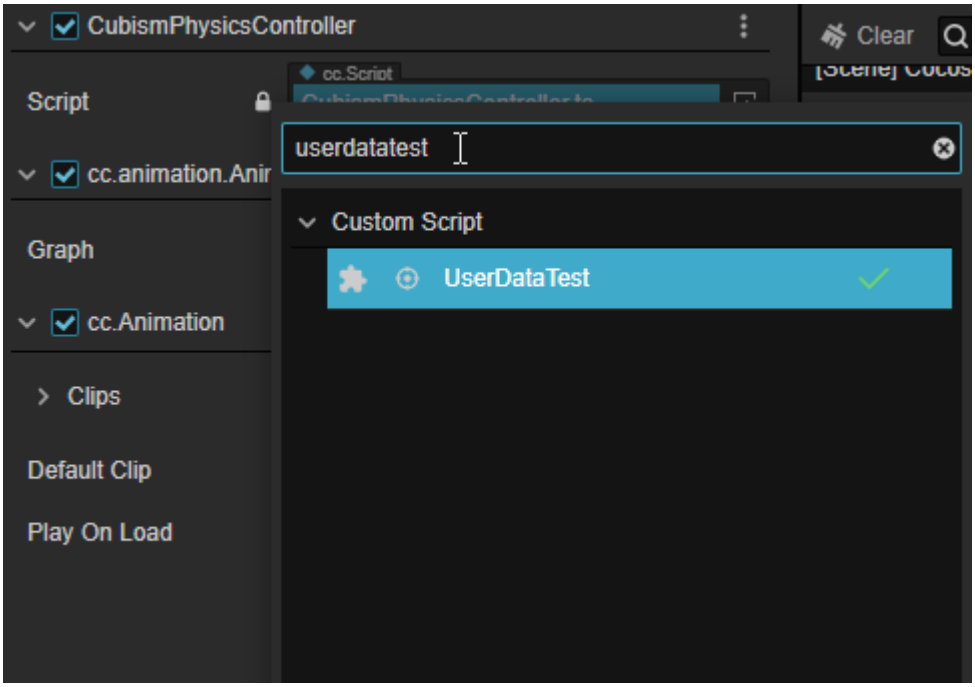
func には、前項で作成したUserDataTestクラスのイベントを受信するメソッドの名前を設定します。

モデルとモーションをSceneに配置

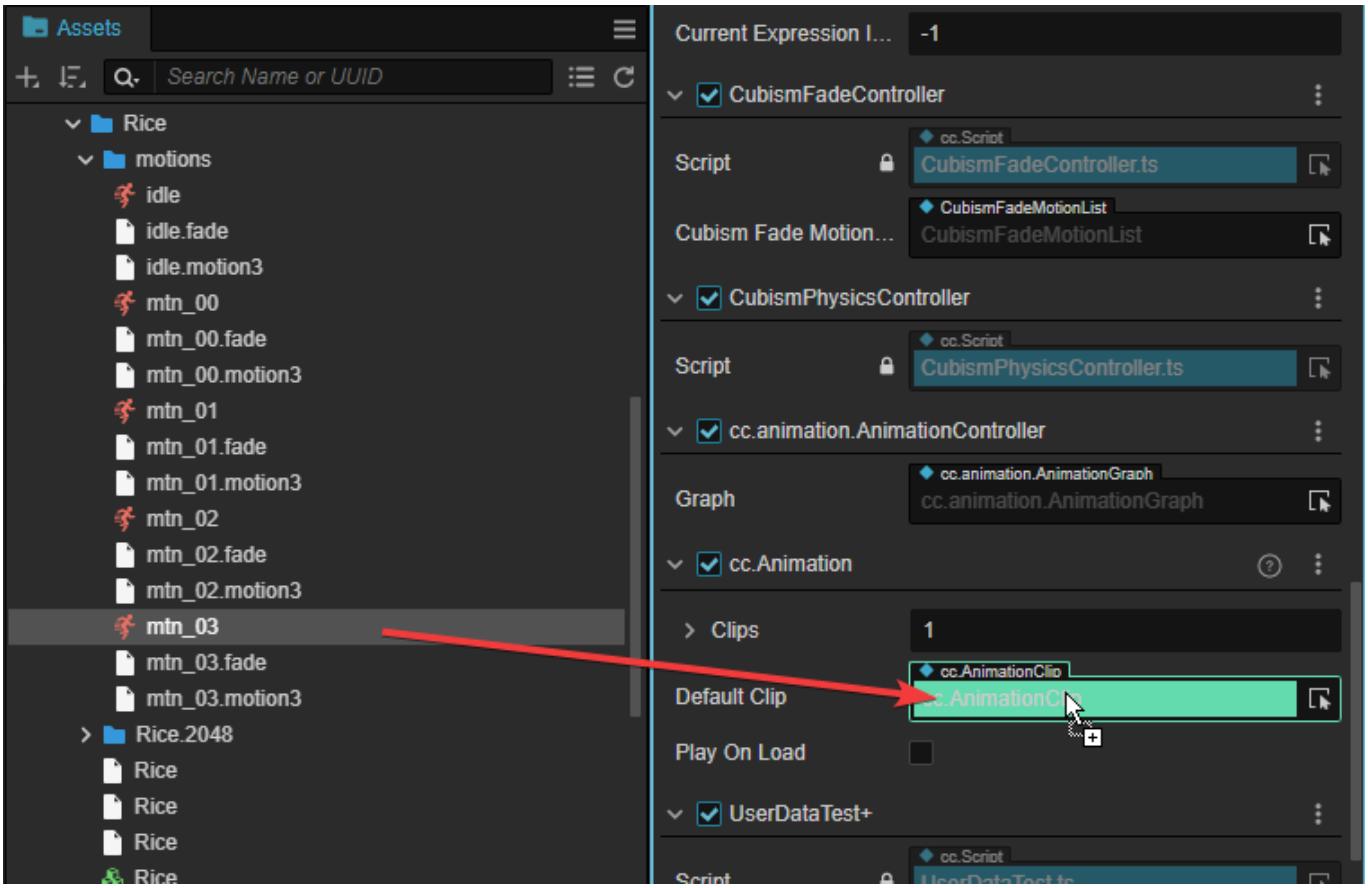
イベントが設定されたmotion3.jsonをインポート、または再インポートします。

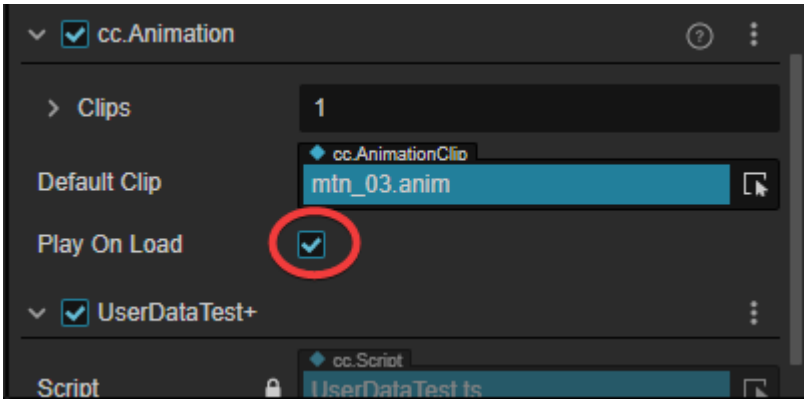
インポートや再インポートについては、該当のチュートリアルをご覧ください。

Hierarchyウィンドウにモデルの Prefabを配置し、PrefabのルートにAnimationコンポーネントとUserDataTestコンポーネントをアタッチします。



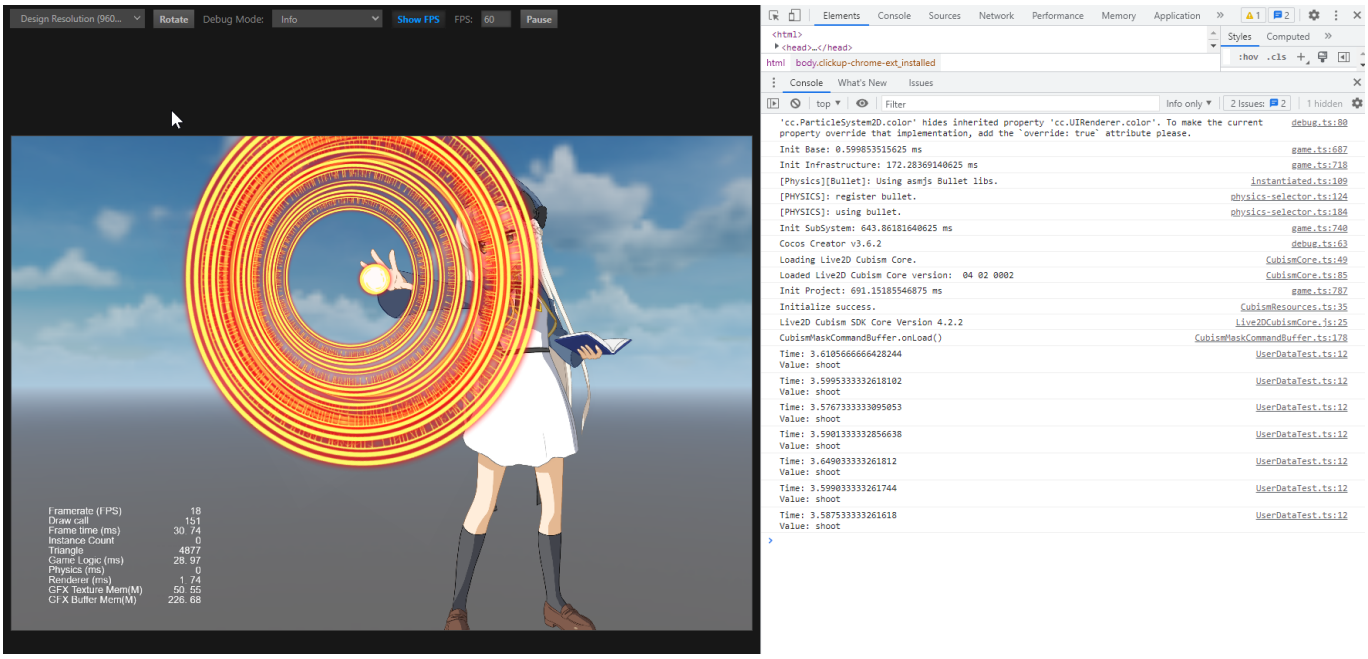
motion3.jsonから生成されたAnimationClipを、Hierarchyに配置したPrefabへのAnimationコンポーネント内のDefault Clip 欄へドラッグ・アンド・ドロップします。





以上で設定は終了です。

この状態でSceneを実行すると、再生中のAnimationClipから特定のタイミングでイベントが発行され、Consoleウィンドウやブラウザのconsoleに、イベントに設定された文字列が出力されます。


Design Resolution (960...
Rotate
Debug Mode:
Info
Show FPS
FPS:
60
Pause
Framerate (FPS) 18
Draw call 151
Frame time (ms) 30.74
Instance Count 0
Triangle 4877
Game Logic (ms) 28.97
Physics (ms) 0
Renderer (ms) 1.74
GFX Texture Mem(M) 50.55
GFX Buffer Mem(M) 226.68
Elements
Console
Sources
Network
Performance
Memory
Application
Styles
Computed
Console
What's New
Issues
Filter
Info only
2 Issues: 2
1 hidden
'cc.ParticleSystem2D.color' hides inherited property 'cc.UIRenderer.color'. To make the current property override that implementation, add the `override: true` attribute please.
Init Base: 0.599853515625 ms
Init Infrastructure: 172.28369140625 ms
[Physics][Bullet]: Using asmjs Bullet libs.
[PHYSICS]: register bullet.
[PHYSICS]: using bullet.
Init SubSystem: 643.86181640625 ms
Cocos Creator v3.6.2
Loading Live2D Cubism Core.
Loaded Live2D Cubism Core version: 04 02 0002
Init Project: 691.15185546875 ms
Initialize success.
Live2D Cubism SDK Core Version 4.2.2
CubismMaskCommandBuffer.onLoad()
Time: 3.6105666666428244
Value: shoot
Time: 3.5995333332618102
Value: shoot
Time: 3.5767333333095053
Value: shoot
Time: 3.5901333332856638
Value: shoot
Time: 3.649033333261812
Value: shoot
Time: 3.599033333261744
Value: shoot
Time: 3.587533333261618
Value: shoot

.cdi3.jsonの利用

概要

ここでは.cdi3.jsonを利用してパラメータやパラメータグループ、パーツの表示名をカスタマイズする方法を説明します。

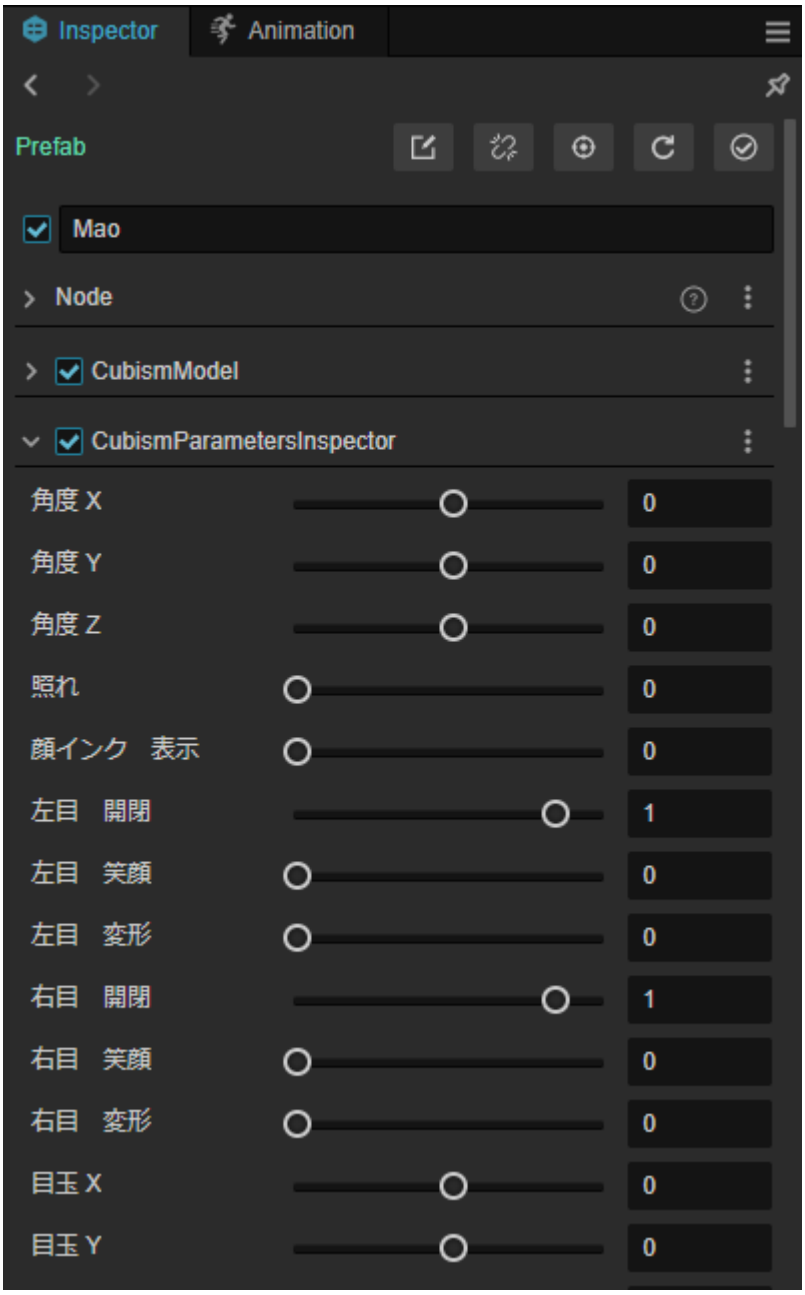
使用するモデルデータは、.cdi3.jsonが書き出されていることが前提となっています。

.cdi3.jsonについてやSDK for Native及びSDK for Webでの実装についてはこちらをご覧ください。

事前準備として、[SDKのインポート～モデルを配置] を参考に、.cdi3.jsonが書き出されたモデルデータのインポートとプレハブの配置を行った後、プレハブのルートオブジェクトを選択してください。

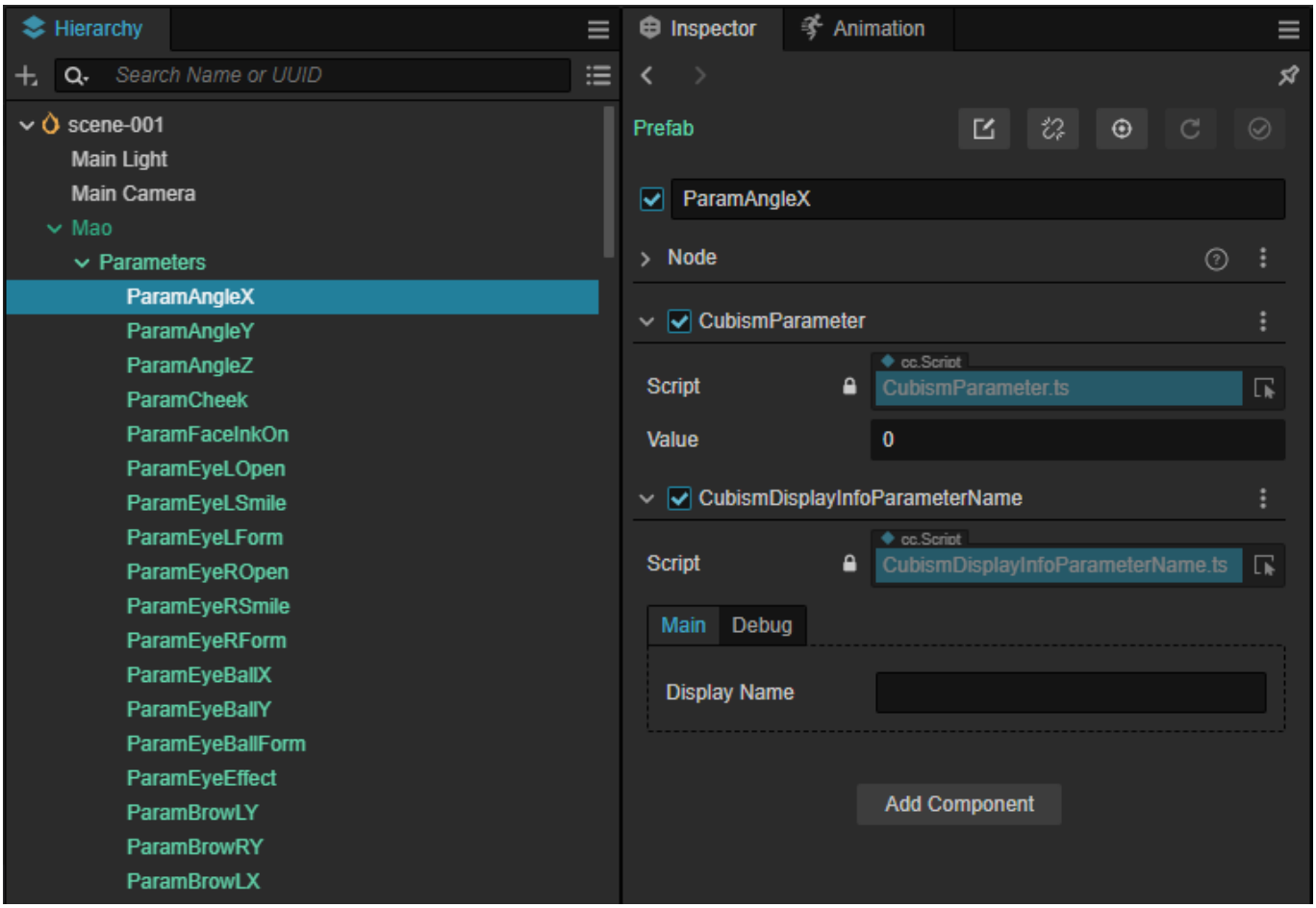
インスペクタ上での利用

.cdi3.jsonを含むモデルデータをインポートした場合、.cdi3.jsonに記載されているパラメータやパーツの名称を、モデルのプレハブのインスペクタ上に表示することが出来ます。

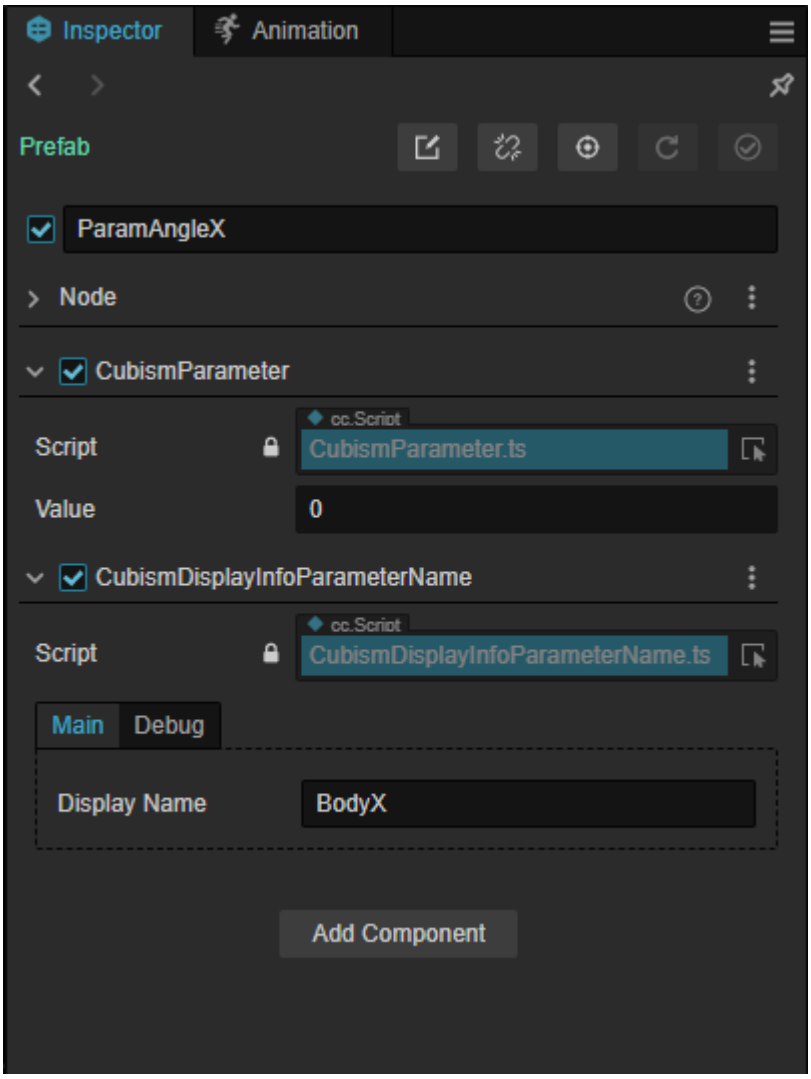


また、ユーザが任意の名称を割り当てることも出来ます。 名称を設定するには、［Cubism Display Info Parameter Name］もしくは［Cubism Display Info Part Name］のDisplay Nameに入力します。 Display Nameが空の場合、.cdi3.jsonに記載の名称が利用されます。

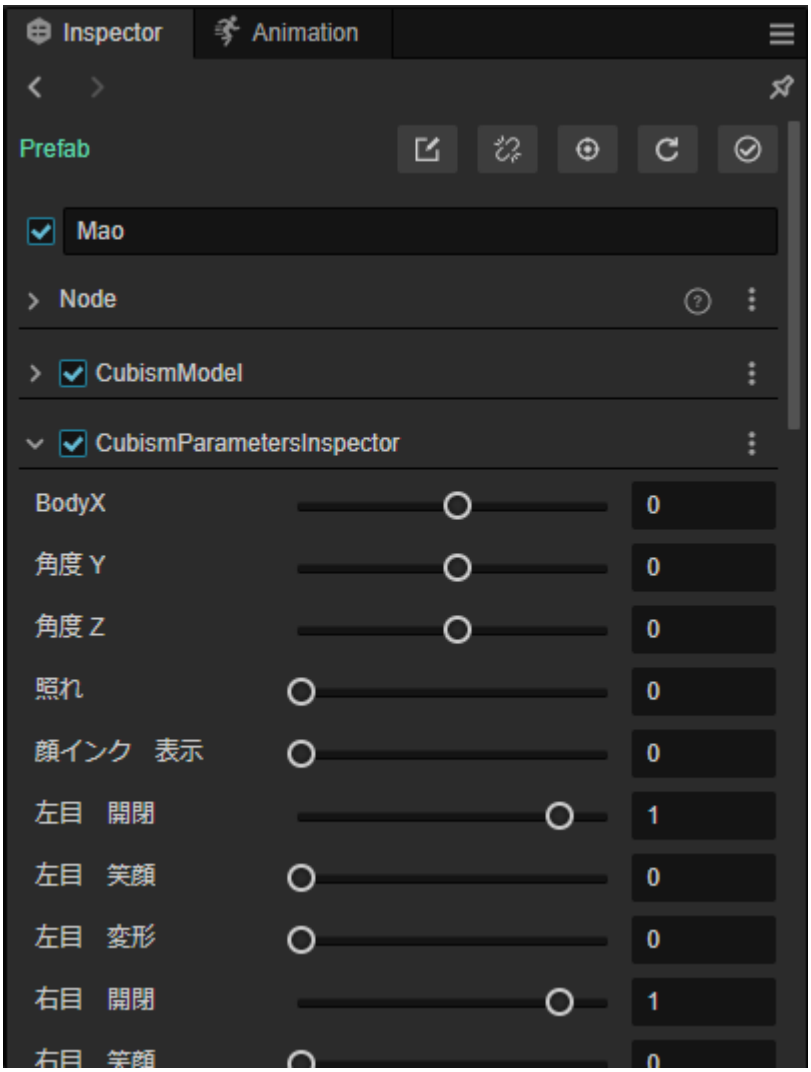
Display Nameが未設定の状態：



Display Nameを設定した状態：



Display Name設定時のパラメータの表記：


Inspector
Animation
Prefab
Mao
Node
CubismModel
CubismParametersInspector
BodyX
0
角度 Y
0
角度 Z
0
照れ
0
顔インク　表示
0
左目　開閉
1
左目　笑顔
0
左目　変形
0
右目　開閉
1

乗算色・スクリーン色

乗算色・スクリーン色を利用して、モデルに色をブレンドして描画する方法を説明します。

Cubism Editor 4.2以前に作成されたモデルなど、モデルに乗算色・スクリーン色が設定されていない場合も特に追加のコーディングを行うことなく利用可能です。 SDK for Cocos Creatorでの詳しい仕様や利用方法についてはSDKマニュアルの[乗算色・スクリーン色] をご覧ください。

事前準備として、[SDKのインポート] を参考に、モデルデータのインポートとプレハブの配置を行ってください。

初期状態では、モデルにあらかじめ設定された乗算色・スクリーン色を常に参照するように設定されており、モデルに乗算色・スクリーン色が設定されていない場合は以下の値が使用されます。

- 乗算色では(1.0, 1.0, 1.0, 1.0)
- スクリーン色では(0.0, 0.0, 0.0, 1.0)

インスペクタ上での利用

SDK側から乗算色・スクリーン色を操作出来るようにするには以下のフラグを有効にします。

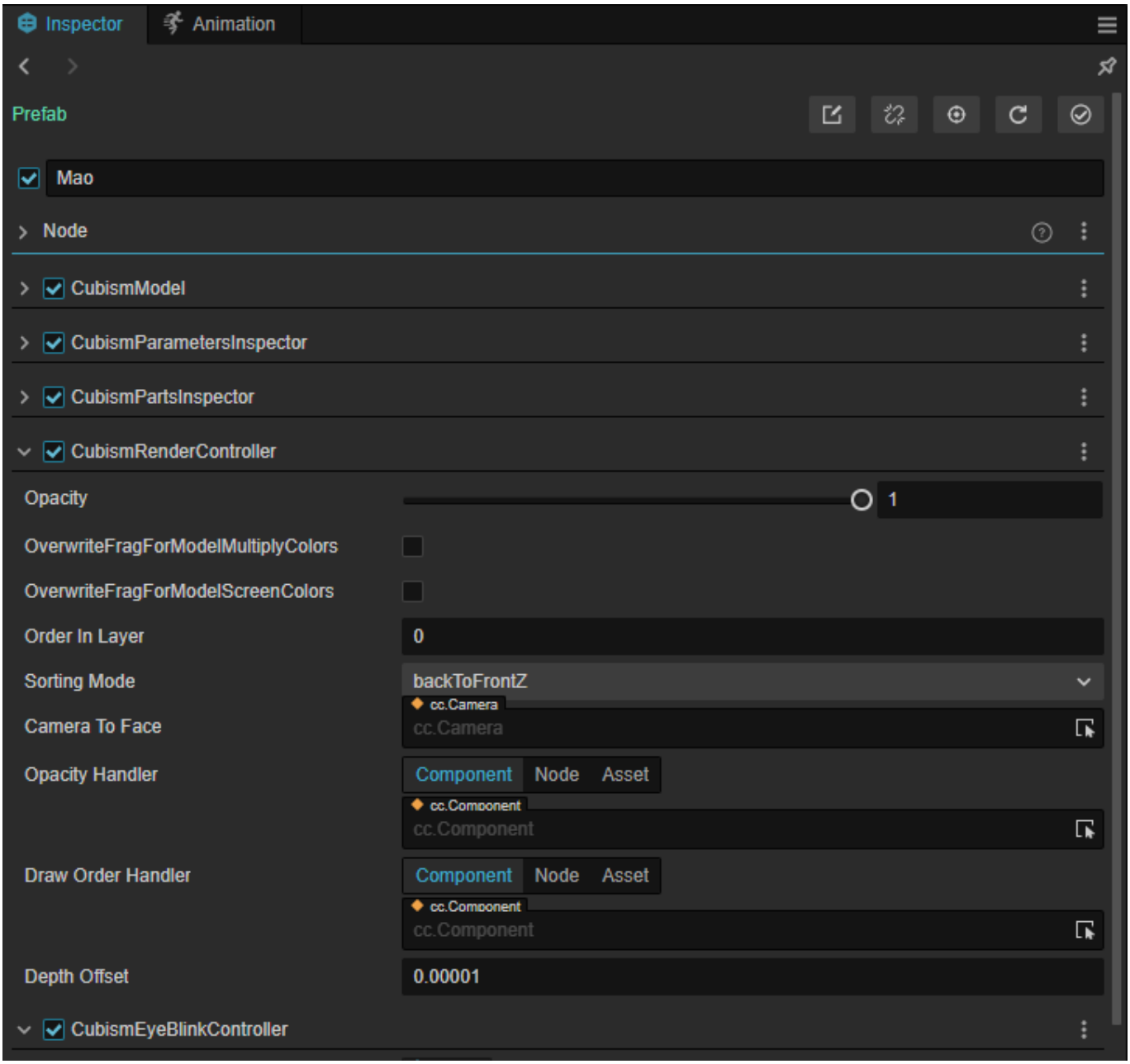
乗算色：

overwriteFlagForModelMultiplyColors、もしくは、overwriteFlagForMultiplyColors

スクリーン色：

overwriteFlagForModelScreenColors、もしくは、overwriteFlagForScreenColors

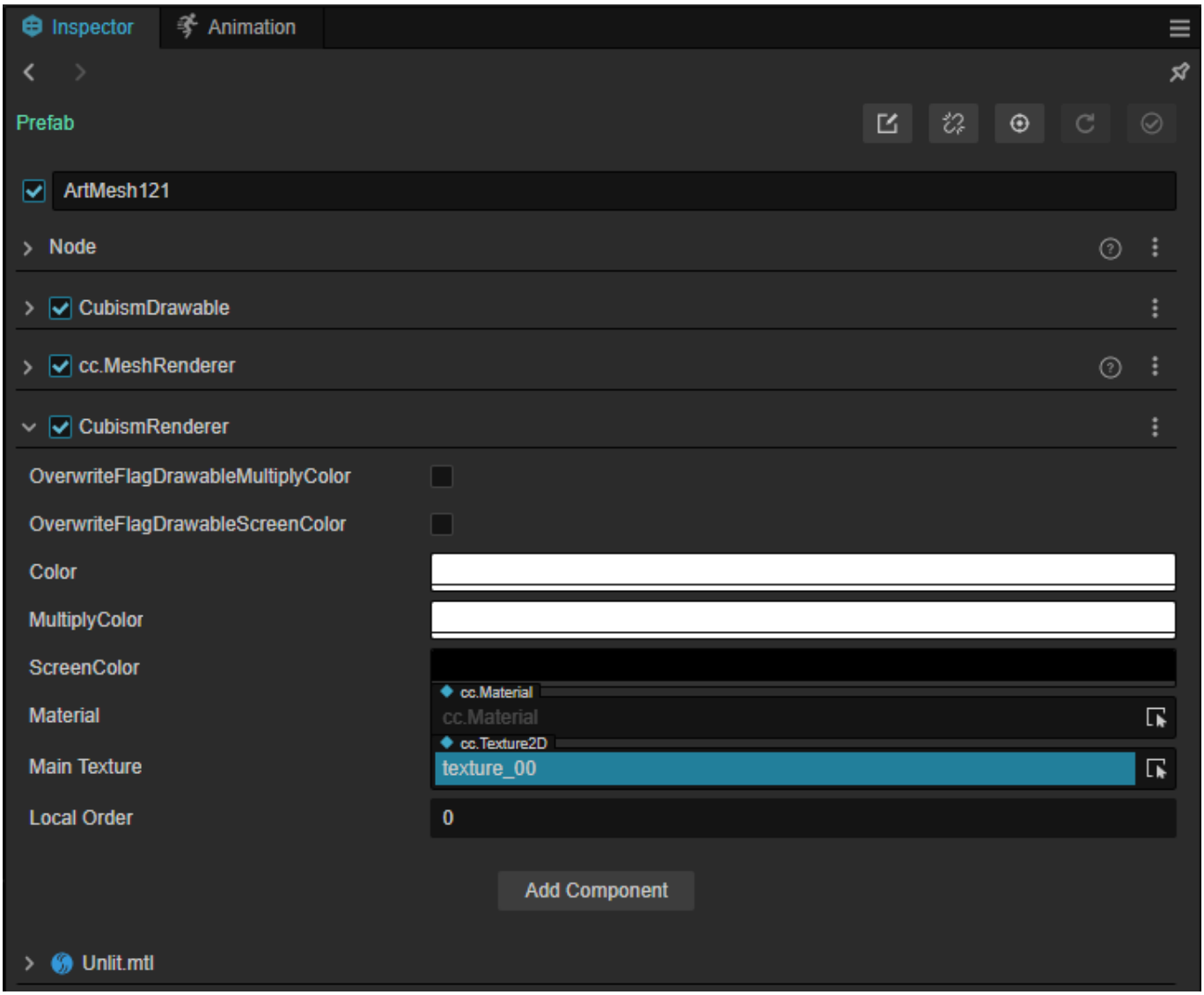
overwriteFlagForModelMultiplyColors 及び overwriteFlagForModelScreenColors は全てのDrawableに対してSDK側から乗算色・スクリーン色の操作を可能とするかを決めるフラグです。 これらのフラグはCocos Creator上ではモデルのプレハブのルートオブジェクトにアタッチされている [CubismRenderController] のインスペクタ上からも操作することが可能です。



overwriteFlagForMultiplyColors 及び overwriteFlagForScreenColors は個々のDrawableがそれぞれSDK側から乗算色・スクリーン色の操作を可能とするかを決めるフラグです。 モデルの各Drawableオブジェクトにアタッチされている [CubismRenderer] のインスペクタ上からも操作することが可能です。

前述した [CubismRenderController] の持つ乗算色・スクリーン色のフラグが有効化されている場合は、それらが優先されます。

乗算色・スクリーン色の設定はスクリプト上からの制御はもちろんのこと、 [CubismRenderer] のインスペクタ上においても操作することが可能です。



スクリプト上での利用

アプリケーション上で利用する場合をはじめ、スクリプト上で制御する必要がある場合には次のようなコードが有効です。

下記のコードでは、全てのDrawableオブジェクトのスクリーン色を一定時間で一斉に変化させるような処理を施しています。

中身を以下のように書き換えたTypeScriptスクリプトを作成し、モデルのプレハブのルートオブジェクトへアタッチする形で利用することが出来ます。

```
import { _decorator, Component, math } from 'cc';
import CubismRenderController from './Rendering/CubismRenderController';
const { ccclass, property } = _decorator;

@ccclass('BlendColorChange')
export class BlendColorChange extends Component {

  private renderController: CubismRenderController | null = null;
  private _colorValues: number[] = new Array<number>();
  private _time: number = 0;
```

```
  protected start() {
    this._colorValues = new Array<number>(3);
    this._time = 0;
    this.renderController = this.getComponent(CubismRenderController);
    this.renderController!.overwriteFlagForModelScreenColors = true;
  }

  protected update(deltaTime: number) {
    if (this._time < 1.0) {
      this._time += deltaTime;
      return;
    }

    for (let i = 0; i < this._colorValues.length; i++)
    {
      this._colorValues[i] = Math.random();
    }

    const color = new math.Color(
      this._colorValues[0] * 255,
      this._colorValues[1] * 255,
      this._colorValues[2] * 255,
      1.0);

    for (let i = 0; i < this.renderController!.renderers!.length; i++)
    {
      this.renderController!.renderers![i].screenColor = color;
    }

    this._time = 0.0;
  }
}
```

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trueとなり、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。

詳細はSDKマニュアルの [乗算色・スクリーン色]をご覧ください。

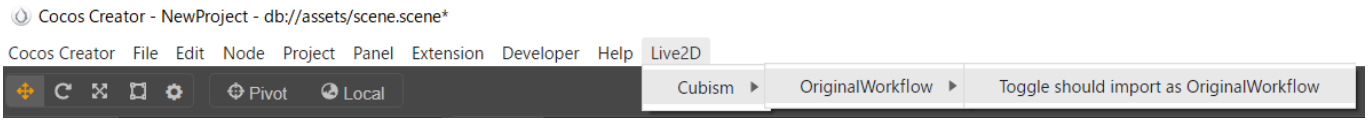
UpdateController の設定方法

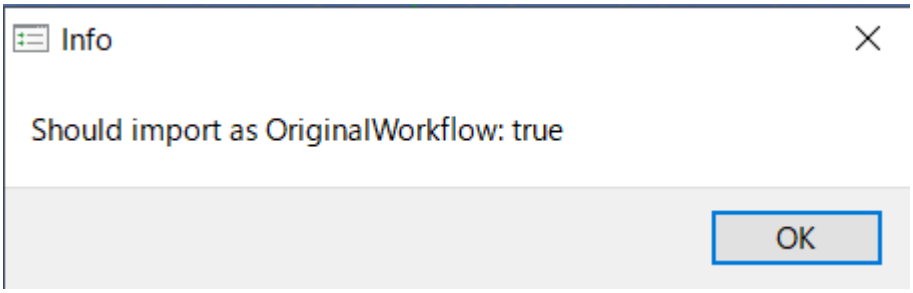
このページでは、Cocos Creator で Cubism の各コンポーネントの実行順を制御する手順を説明します。 以下は「SDK をインポート」を行なったプロジェクトに追加することを前提としています。

概要

各コンポーネントの実行順を制御するには、 **CubismUpdateController** を利用します。

Cocos Creator メニューの「Live2D/Cubism/OrignalWorkflow/ **Toggle Should Import As Original Workflow** 」をクリックして、以下の画像のように **true** の状態でモデルをインポートした場合、生成される Prefab にはこのコンポーネントが設定されます。





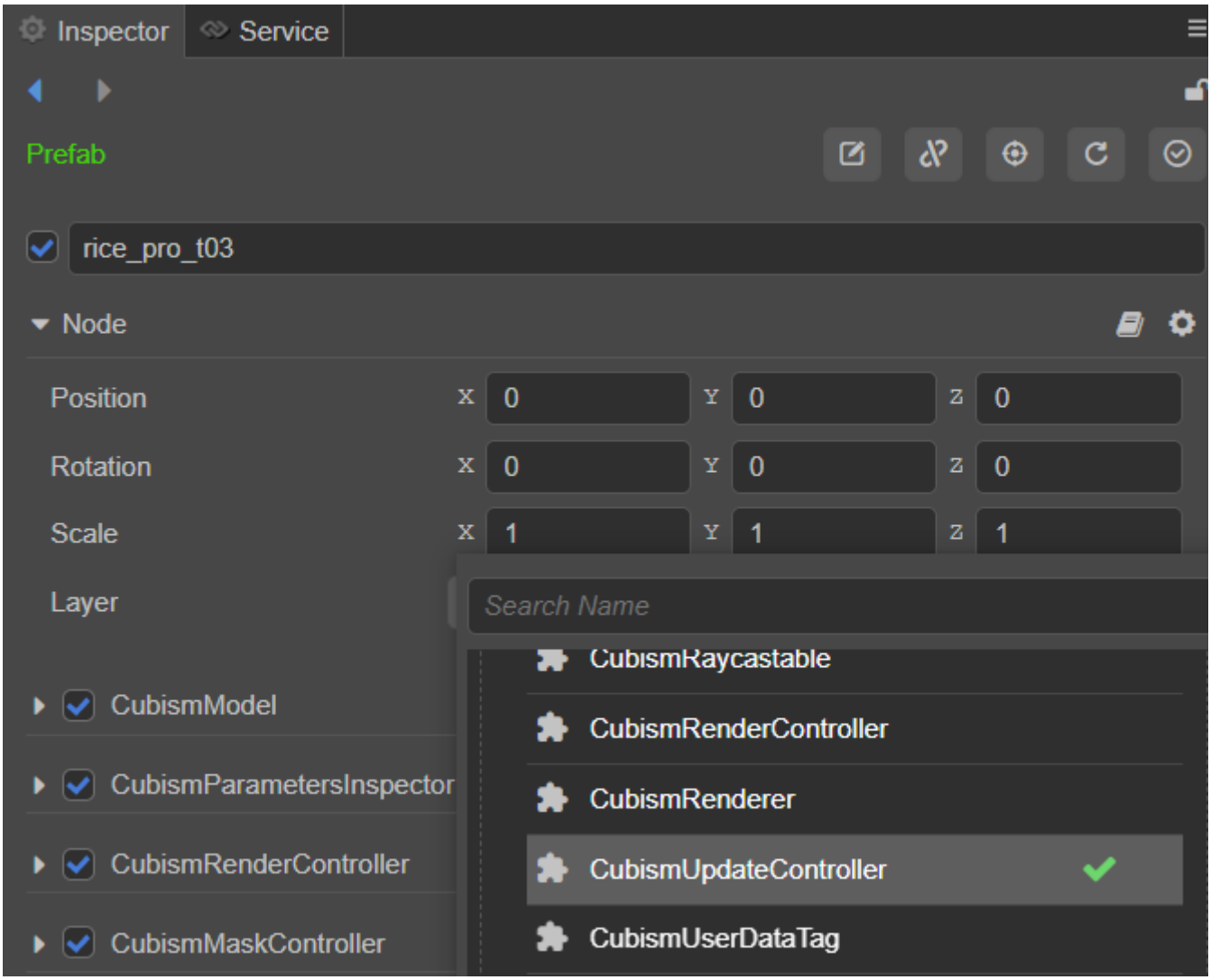
上記の設定をしていないモデルの各 Cubism コンポーネントの更新順を制御させる場合、本記事で説明する手順で設定することができます。

Cubism のモデルに **UpdateController** を設定するには、以下の手順を行います。

1. CubismUpdateController をアタッチ

1. CubismUpdateController をアタッチ

モデルのルートとなる Node に、コンポーネントの更新順を制御する「 **CubismUpdateController** 」をアタッチします。


Inspector
Service
Prefab
rice_pro_t03
Node
Position
Rotation
Scale
Layer
Search Name
CubismRaycastable
CubismRenderController
CubismRenderer
CubismUpdateController
CubismUserDataTag
CubismModel
CubismParametersInspector
CubismRenderController
CubismMaskController

表情機能を使用する

このページでは、CubismのモデルにExpressionを利用して表情を再生させる手順を説明します。 以下は[SDKをインポート [UpdateControllerの設定][ParameterStoreの設定]を行なったプロジェクトに追加することを前提としています。

概要

Live2D Cubism SDK for Cocos Creatorで.exp3.jsonによって表情を再生させるには、「Expression」というコンポーネントを利用します。

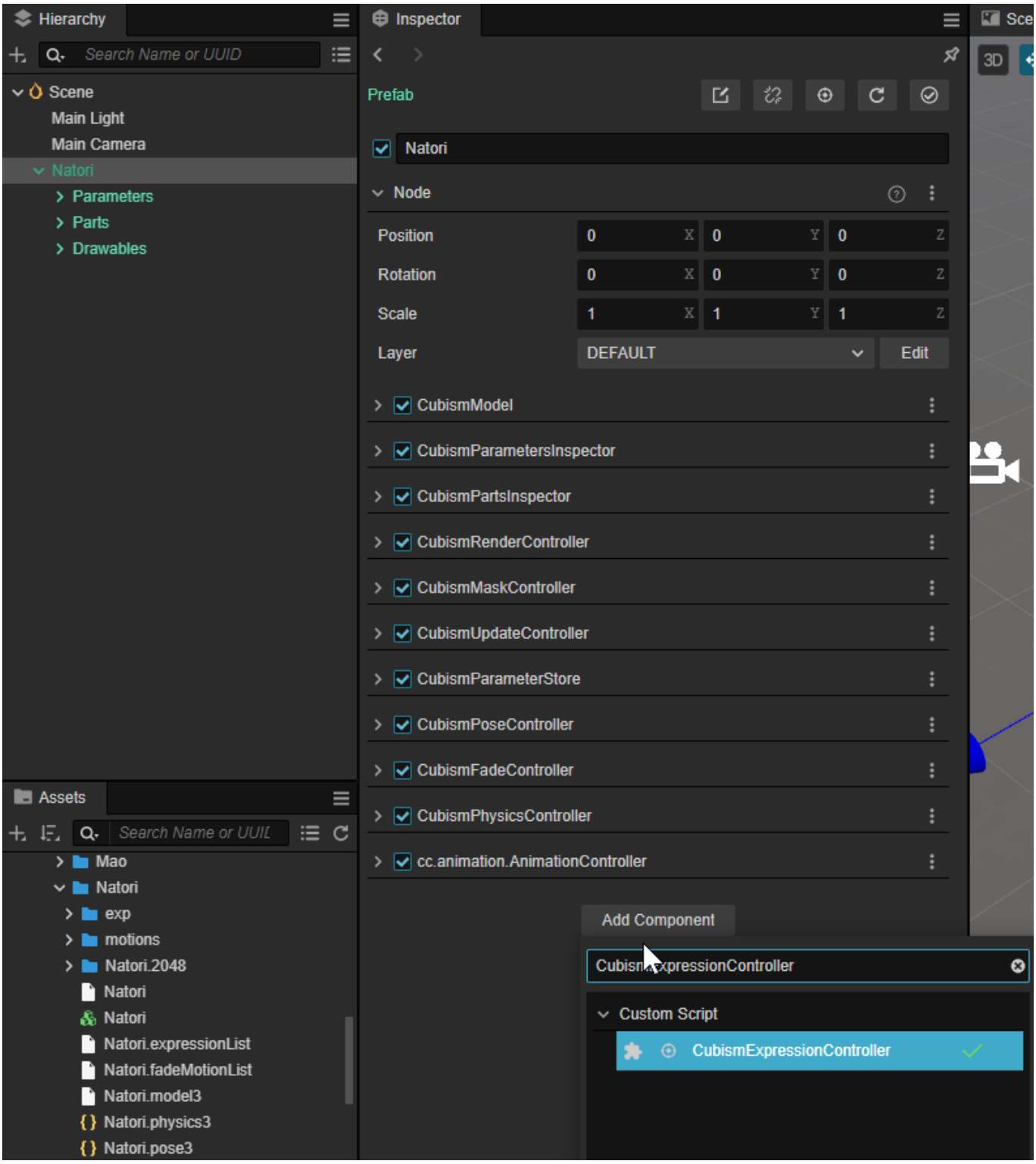
Cocos Creatorエディターメニューの「Live2D/Cubism/OrignalWorkflow/Should Import As Original Workflow」にチェックを入れた状態でインポートしたモデルである場合、生成されるPrefabにはExpressionが設定されます。

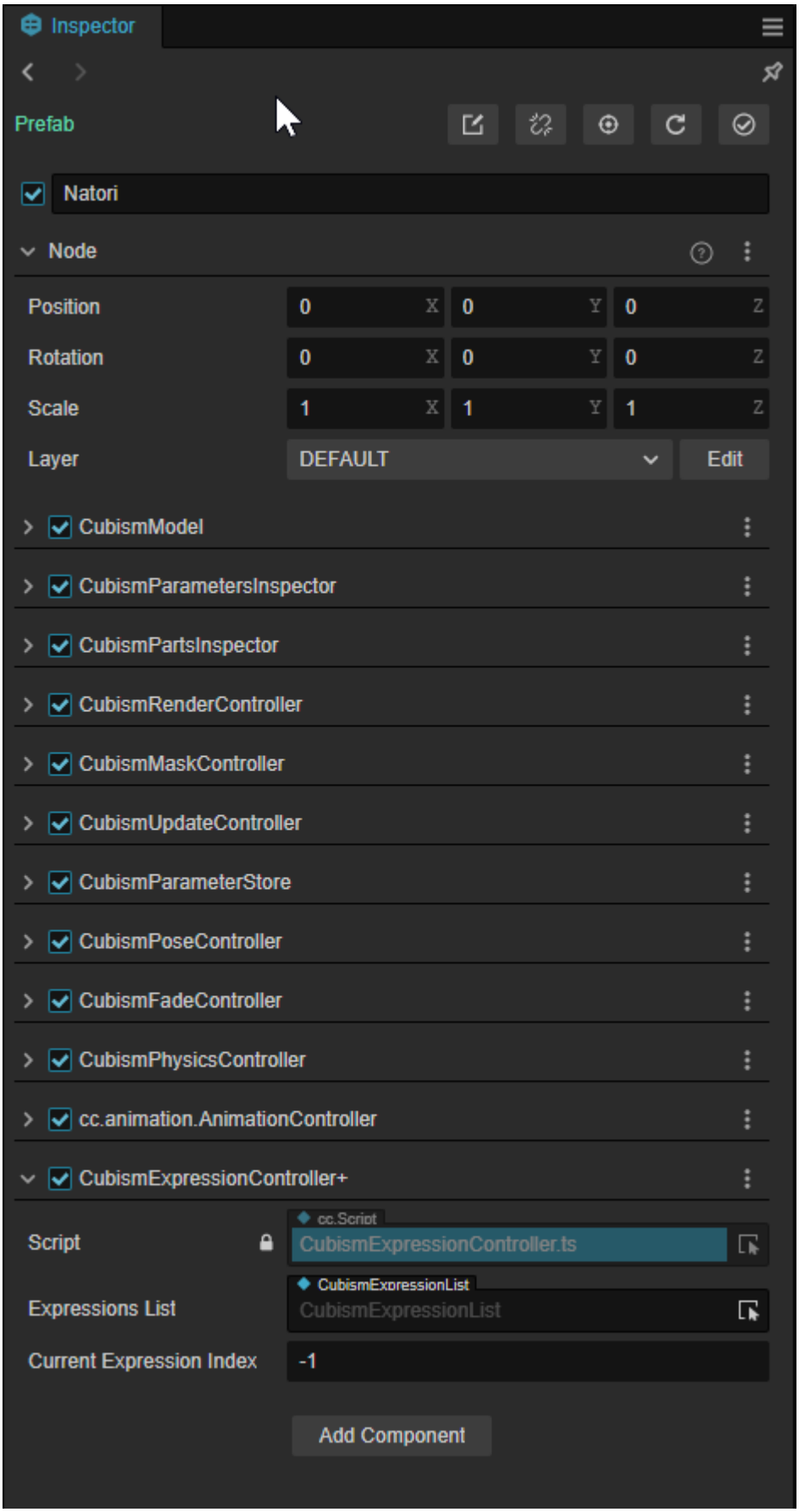
OW方式で生成されていないPrefabにExpressionを設定する場合、以下の3つの手順を行います。

1. CubismExpressionControllerをアタッチ
2. 「[モデル名].expressionList」を設定
3. 再生する表情を設定

1. CubismExpressionControllerをアタッチ

モデルのルートとなるNodeに「CubismExpressionController」をアタッチします。


Hierarchy
Search Name or UUID
Scene
Main Light
Main Camera
Natori
Parameters
Parts
Drawables
Inspector
Prefab
Natori
Node
Position
Rotation
Scale
Layer
DEFAULT
Edit
CubismModel
CubismParametersInspector
CubismPartsInspector
CubismRenderController
CubismMaskController
CubismUpdateController
CubismParameterStore
CubismPoseController
CubismFadeController
CubismPhysicsController
cc.animation.AnimationController
Add Component
Custom Script
CubismExpressionController
Assets
Mao
Natori
exp
motions
Natori.2048
Natori
Natori
Natori.expressionList
Natori.fadeMotionList
Natori.model3
Natori.physics3
Natori.pose3

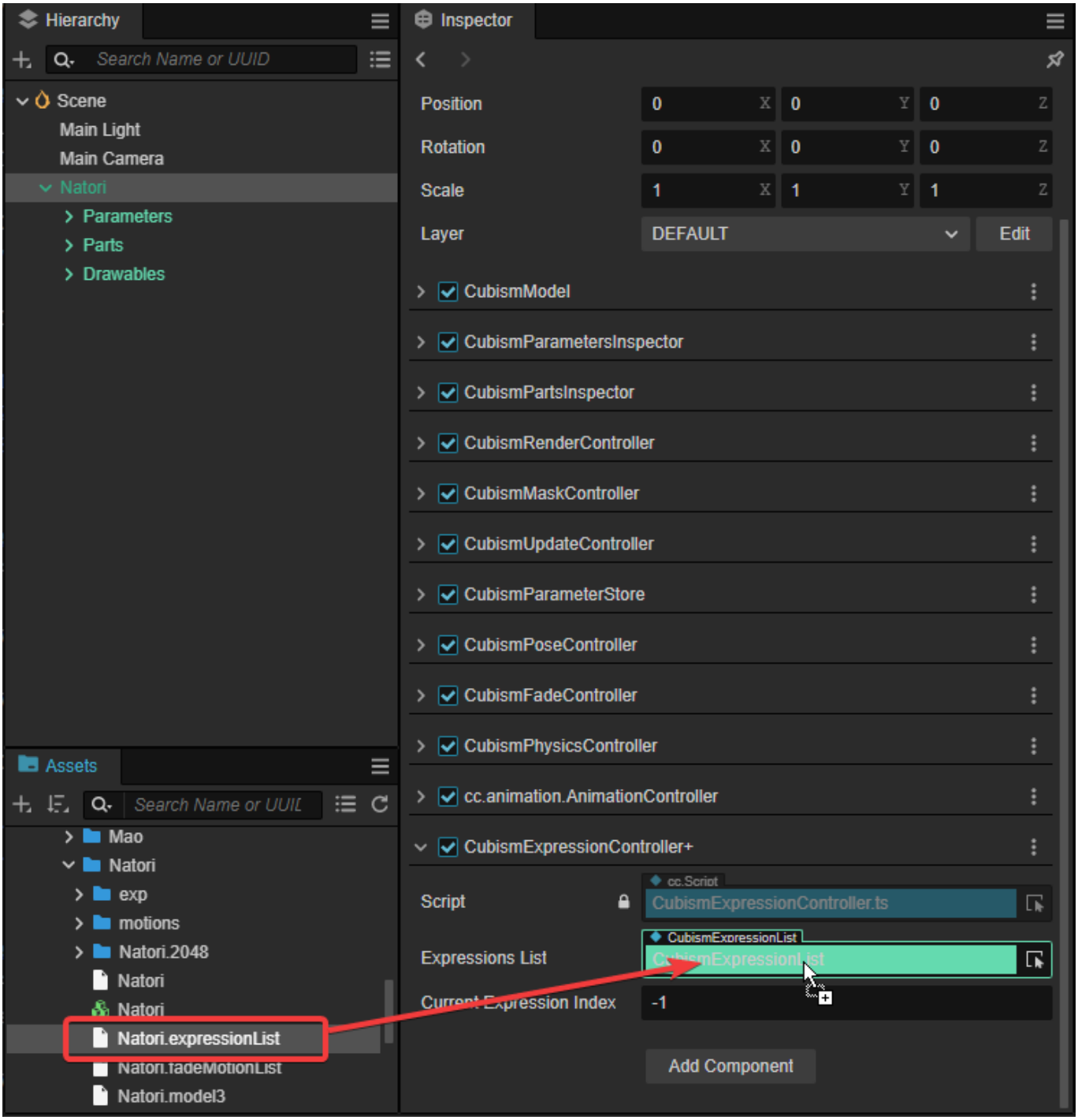


CubismExpressionControllerには、設定項目が2つあります。

- Expression List：「モデル名.expressionList」を設定 ※詳細は手順の2で説明
- Current Expression Index：再生する表情のインデックスを設定 ※詳細は手順の3で説明

2. 「モデル名.expressionList」を設定

モデルを選択し、Inspectorビューから CubismExpressionControllerの「Expression List」に、「[モデル名].expressionList」をドラッグ・アンド・ドロップします。



※「[モデル名].expressionList」は「.exp3」アセットをリストにまとめたアセットです。 ※「.exp」アセットはExpressionのデータが保存されているアセットです。「.exp3.json」がインポートされたときに自動生成されます。

3. 再生する表情を設定

再生する表情のインデックスをCubismExpressionControllerの「Current Expression Index」に設定します。今回は0に設定します。

以上でexp3.jsonから生成したexp3アセットを利用して表情を再生する設定は完了です。

この状態でSceneを実行すると、「[モデル名].expressionList」の0番の表情が再生されます。 「Current Expression Index」を再設定することで再生する表情を切り替えることができます。

※ 「[モデル名].expressionList」以外のインデックスが設定された場合、デフォルトの表情が再生されます。

Pose機能を利用する

このページでは、Cubismのモデルにパーツの表示状態を制御させる手順を説明します。 以下は［SDK のインポート～モデルを配置］を行ったプロジェクトに追加することを前提としています。

ポーズの設定方法につきましては こちら を、SDKのPoseの仕様につきましては マニュアル - Pose をご覧ください。

概要

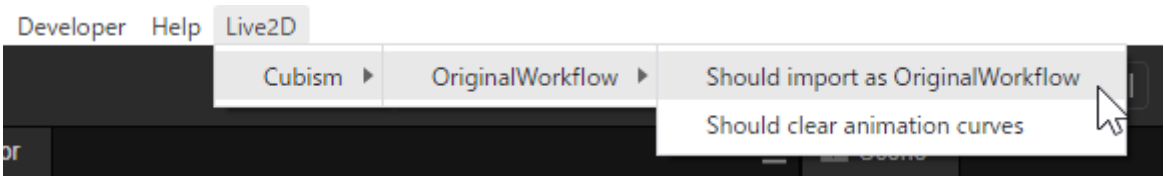
Cocos Creatorエディターメニューの「Live2D/Cubism/OrignalWorkflow/Should Import As Original Workflow」をチェックした状態でインポートしたモデルである場合、モデルにPoseが設定されます。

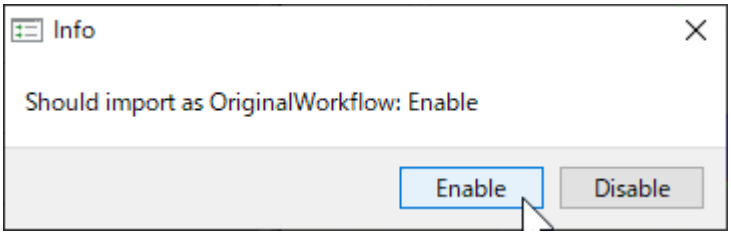
上記の設定をしていないモデルにPose機能を適用する場合、本記事で説明する手順で設定することができます。

1. AnimationClipのカーブをクリア
2. OWモードでモデルを再インポート ※ OWモードでモデルを再インポートした場合、Pose設定以外のOW用のコンポーネント「CubismUpdateController」「CubismParameterStore」「CubismExpressionController」も同時に追加されます。

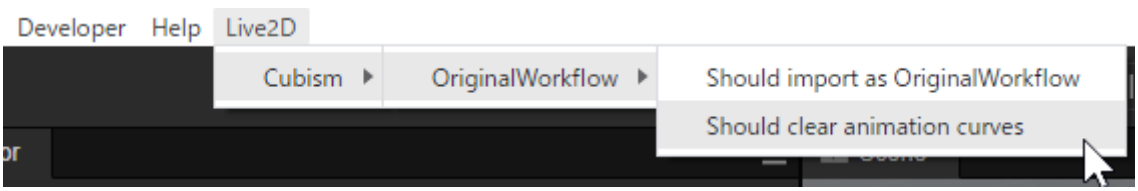
AnimationClipのカーブをクリア

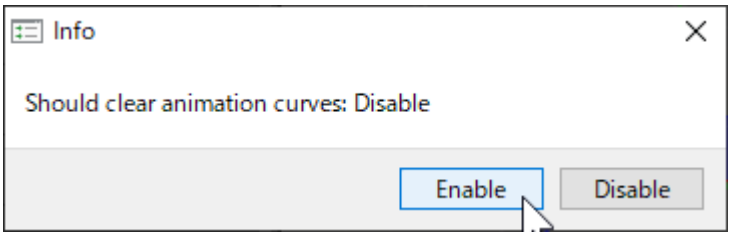
1. Cocos Creatorエディターメニューの「Live2D/Cubism/OrignalWorkflow/Should Import As Original Workflow」を選択し、表示されたダイアログでEnableにします。



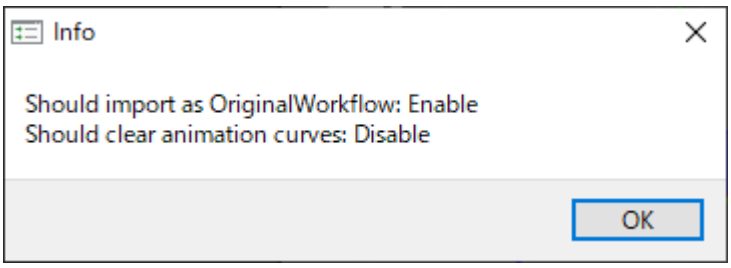


2. Cocos Creatorエディターメニューの「Live2D/Cubism/OrignalWorkflow/Should Clear Animation Curves」を選択し、表示されたダイアログでEnableにします。

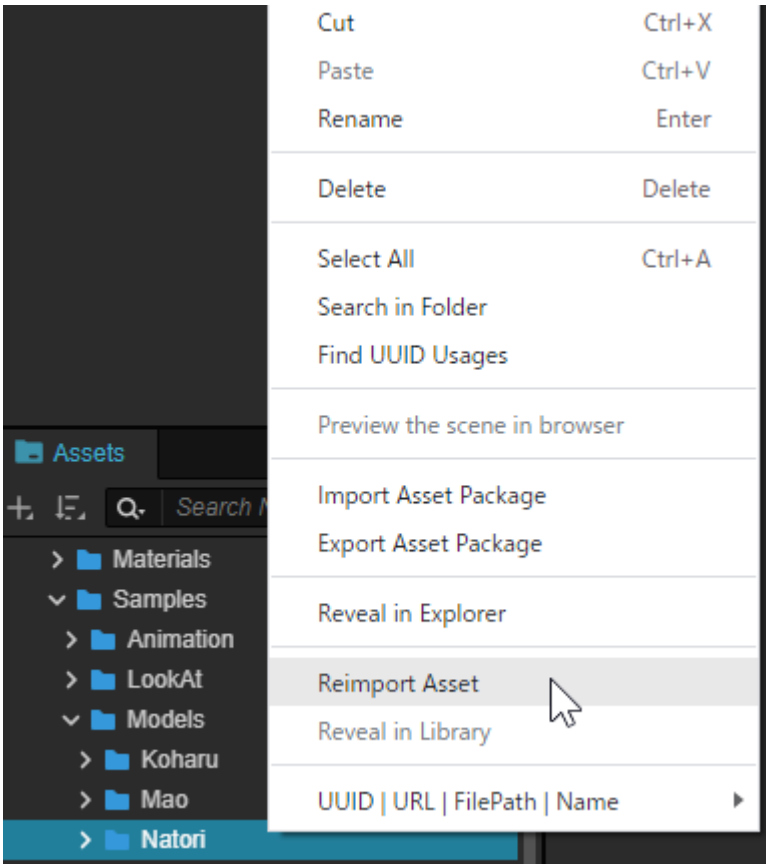




以下のような表示になればOKです。

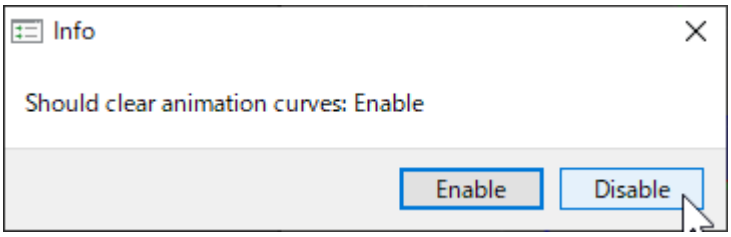


3. Projectウィンドウで、設定するモデルデータのフォルダを右クリックして「Reimport」をクリックします。

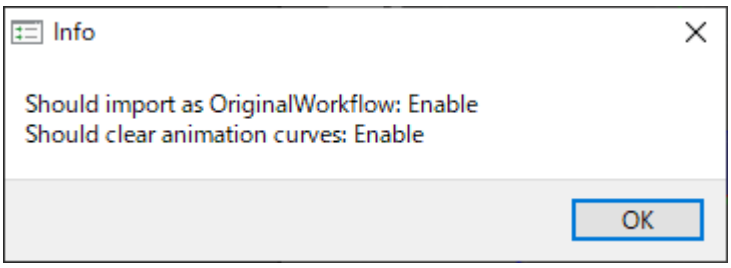


OWモードでモデルを再インポート

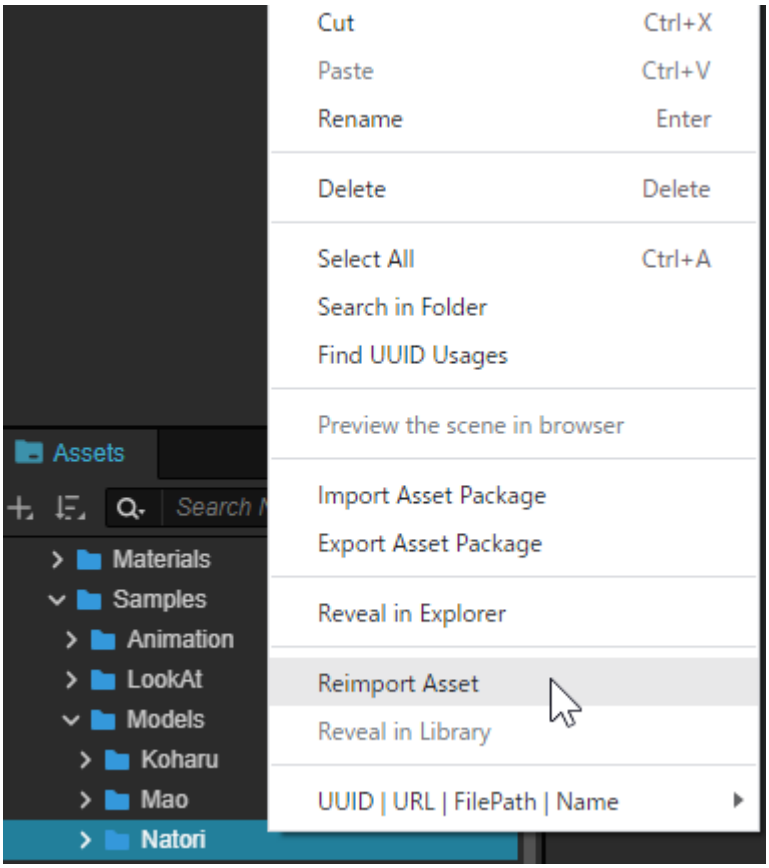
1. Cocos Creatorエディターメニューの「Live2D/Cubism/OrignalWorkflow/Should Clear Animation Curves」を選択し、表示されたダイアログでDisableにします。



以下のような表示になればOKです。



2. Projectウィンドウで、再インポートするモデルデータのフォルダを右クリックして「Reimport」をクリックします。



操作する値の保存／復元を行う

概要

このページでは、Cubism のモデルに **ParameterStore** を利用してパラメータの値とパーツの不透明度を保存／復元する手順を説明します。 以下は「SDK をインポート」「UpdateController の設定」を行なったプロジェクトに追加することを前提としています。

CubismParameterStore について

Cocos Creator メニューの「Live2D/Cubism/OrignalWorkflow/ **Toggle Should Import As Original Workflow** 」をクリックして、 **true** の状態でモデルをインポートした場合、生成する Prefab には CubismParameterStore コンポーネントが設定されます。

CubismParameterStore は、AnimationClip が再生される前後で Cubism モデルのパラメータの値やパーツの不透明度を復元、保存するコンポーネントです。 これを使用することで、他の Cubism のコンポーネントが Additive や Multiply といった相対的に値を操作するときに発生する不具合を回避することができます。
Cubism のコンポーネントが行う Additive や Multiply のブレンドでの値操作は、直前のフレームで行った値操作がリセットされていることを前提としたしようになっています。 AnimationClip によって値が上書きされるのであれば、直前の値操作は上書きされるため Cubism のコンポーネントによる値操作は正常に動作します。 しかし、仮に再生されたアニメーションがその値を操作しない場合、直前に操作した値がそのまま次の更新時にも残ってしまうため計算する値が重複するため期待通りの動作にならないことがあります。
CubismParameterStore は、アニメーションの処理直後の **lateUpdate()** のタイミングに、自身がアタッチされたモデルのすべてのパラメータ値を **保存** し、次のフレームの **update()** で保存した値を **復元** します。 これによってアニメーションによって値が上書きされないパラメータに対してもコンポーネントから正常な操作ができます。

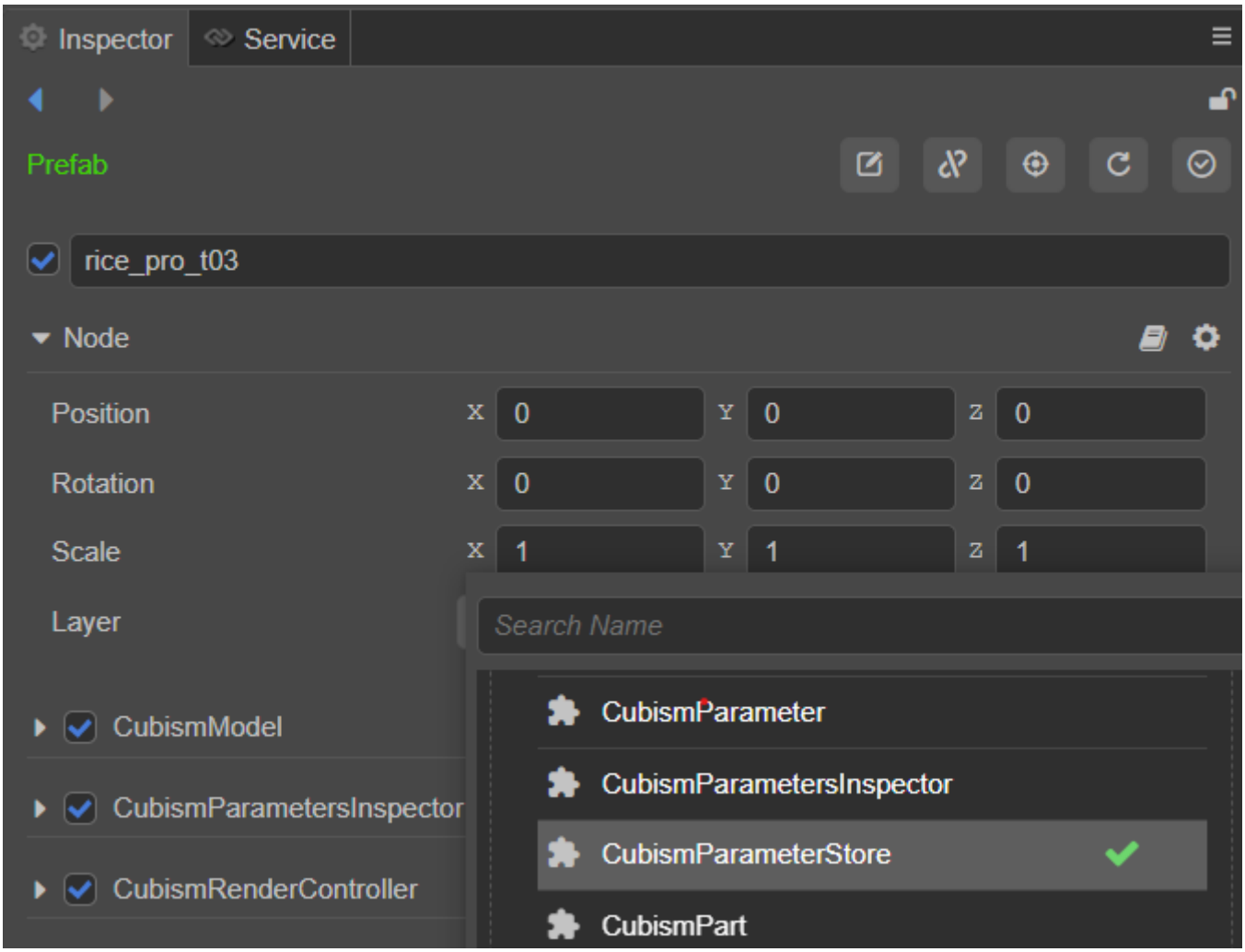
以下では従来方式で生成したモデルでパラメータの値とパーツの不透明度を保存／復元させる場合、以下の手順を行います。

1. CubismParameterStore をアタッチ

※ ParameterStore を利用するには「UpdateController の設定」が必要です。

1. CubismParameterStore をアタッチ

モデルのルートとなる Node に、値を保存／復元する「CubismParameterStore」をアタッチします。


Inspector
Service
Prefab
rice_pro_t03
Node
Position
Rotation
Scale
Layer
CubismModel
CubismParametersInspector
CubismRenderController
Search Name
CubismParameter
CubismParametersInspector
CubismParameterStore
CubismPart

独自のコンポーネントの実行順を制御させる

ここでは、ユーザ独自のコンポーネントに対して、他の Cubism のコンポーネント同士の実行順を制御する手順を説明します。 以下は［SDK のインポート～モデルを配置］をおこなったプロジェクトに追加することを前提とした説明となっています。

概要

Cubism SDK for Cocos Creator の **Original Workflow** のコンポーネントの中には、実行される順序に制限があるものがあります。

Cubism SDK for Cocos Creator では、 **CubismUpdateController** を用いることで制御することができ、上記のコンポーネントはこれによって実行順が制御されます。 CubismUpdateController が制御するコンポーネントは、Cubism モデルの Prefab のルートにアタッチされたコンポーネントになります。

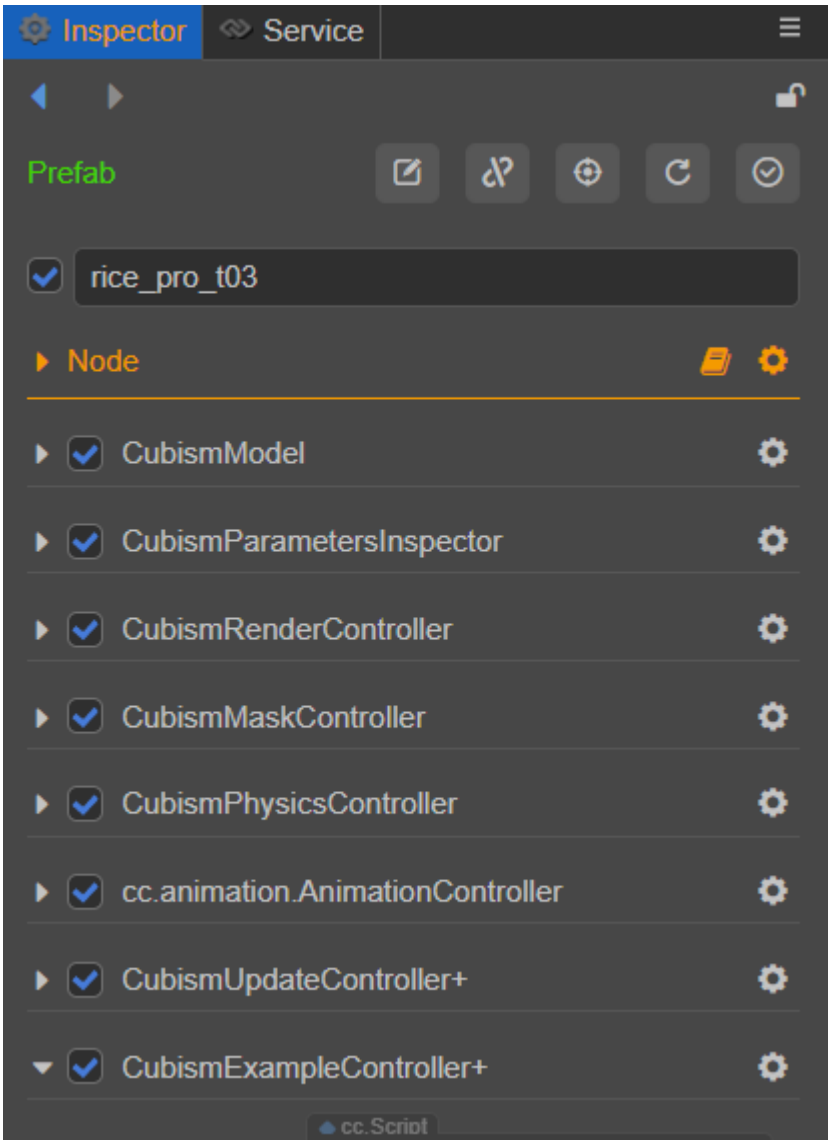
CubismUpdateController を用いれば、ユーザ独自のコンポーネントも同様に実行順を制御することが可能です。

ここでは例として以下のコンポーネントに対して実行順の制御を設定する手順を説明します。

```
@ccclass('CubismExampleController')
export class CubismExampleController extends Component {
    start() {
        // CubismExampleControllerの初期化処理
    }

    lateUpdate(deltaTime: number) {
        // CubismExampleControllerの更新処理
    }
}
```

1. Prefab にコンポーネントをアタッチ

Hierarchy に配置された Prefab のルートの Node に、 **CubismExampleController** をアタッチします。 また、Prefab が OW 形式でインポートされていない場合、 **CubismUpdateController** もアタッチします。



2. コンポーネントに ICubismUpdatable を実装

実行順を制御するコンポーネントに **ICubismUpdatable** インターフェースを実装します。

CubismUpdateController は実行時に ICubismUpdatable を実装したコンポーネントを取得し、それらの実行順を制御します。

```
@ccclass('CubismExampleController')
export class CubismExampleController extends Component implements ICubismUpdatable
{

  bindedOnLateUpdate: ICubismUpdatable.CallbackFunction;
  get executionOrder(): number {
    throw new Error('Method not implemented.');
  }
  get needsUpdateOnEditing(): boolean {
    throw new Error('Method not implemented.');
  }
  get hasUpdateController(): boolean {
    throw new Error('Method not implemented.');
  }
```

```
  set hasUpdateController(value: boolean) {
    throw new Error('Method not implemented.');
  }

  readonly [ICubismUpdatable.SYMBOL]: typeof ICubismUpdatable.SYMBOL;

  protected start() {
    // CubismExampleControllerの初期化処理
  }

  protected lateUpdate(deltaTime: number) {
    // CubismExampleControllerの更新処理
  }
}
```

ここで実装した ICubismUpdatable インターフェースは以下のようになっております。

```
interface ICubismUpdatable {
  readonly [ICubismUpdatable.SYMBOL]: typeof ICubismUpdatable.SYMBOL;
  readonly bindedOnLateUpdate: ICubismUpdatable.CallbackFunction;

  get executionOrder(): number;
  get needsUpdateOnEditing(): boolean;
  get hasUpdateController(): boolean;
  set hasUpdateController(value: boolean);
}
```

executionOrder は、このコンポーネントの実行順を決める値になります。 この値が小さいほど他のコンポーネントよりも先に呼ばれます。 SDK 同梱のコンポーネントに設定される値は CubismUpdateExecutionOrder に記述されています。

hasUpdateController は、ICubismUpdatable を実装したコンポーネントが CubismUpdateController のアタッチされていない場合、Cocos Creator のイベント関数から呼び出すようにするためのフラグです。

```
namespace CubismUpdateExecutionOrder {
  export const CUBISM_FADE_CONTROLLER = 100;
  export const CUBISM_PARAMETER_STORE_SAVE_PARAMETERS = 150;
  export const CUBISM_POSE_CONTROLLER = 200;
  export const CUBISM_EXPRESSION_CONTROLLER = 300;
  export const CUBISM_EYE_BLINK_CONTROLLER = 400;
  export const CUBISM_MOUTH_CONTROLLER = 500;
  export const CUBISM_HARMONIC_MOTION_CONTROLLER = 600;
  export const CUBISM_LOOK_CONTROLLER = 700;
  export const CUBISM_PHYSICS_CONTROLLER = 800;
  export const CUBISM_RENDER_CONTROLLER = 10000;
  export const CUBISM_MASK_CONTROLLER = 10100;
```

3. コンポーネントを CubismUpdateController に対応させる

CubismExampleControllerを以下のように修正します。

```
import { _decorator, Component, Node } from 'cc';
import CubismUpdateController from
'../../extensions/Live2DCubismSdkForCocosExtension/static/assets/Framework/CubismU
pdateController';
import ICubismUpdatable from
'../../extensions/Live2DCubismSdkForCocosExtension/static/assets/Framework/ICubism
Updatable';
const { ccclass, property } = _decorator;

@ccclass('CubismExampleController')
export class CubismExampleController extends Component implements ICubismUpdatable
{
  bindedOnLateUpdate: ICubismUpdatable.CallbackFunction;

  // このコンポーネントの実行順
  get executionOrder(): number {
      return 150;
  }

  // Sceneの非実行中に実行順の制御を行うか
  get needsUpdateOnEditing(): boolean {
      return false;
  }

  // 実行順が制御されているか
  @property({ serializable: false, visible: false })
  private _hasUpdateController: boolean = false;
  get hasUpdateController(): boolean {
      return this._hasUpdateController;
  }
  set hasUpdateController(value: boolean) {
      this._hasUpdateController = value;
  }

  readonly [ICubismUpdatable.SYMBOL]: typeof ICubismUpdatable.SYMBOL;

  protected start() {
    // CubismExampleControllerの初期化処理

    // モデルのPrefabにCubismUpdateControllerがアタッチされているかをチェック
    this.hasUpdateController = this.getComponent(CubismUpdateController) != null;
  }

  protected lateUpdate(deltaTime: number) {
    // CubismUpdateControllerがアタッチされていない場合、CubismExampleController自身のイベ
ント関数から更新処理を行う
    if (!this.hasUpdateController) {
        this.onLateUpdate(deltaTime);
    }
  }
```

```
    // 実行順序が制御されるアップデート関数
    public onLateUpdate(deltaTime: number) {
      // CubismExampleControllerの更新処理
    }
  }
```

lateUpdate()で行っていた更新処理を、CubismUpdateController が呼び出す **onLateUpdate** ()に移します。

以上で実行順を制御させる設定は終了です。 このスクリプトを Cubism モデルの Prefab にアタッチしてシーンを実行すると、このスクリプトの更新処理が CubismUpdateController によって呼び出されます。