Cubism SDK for Cocos Creator alpha Manual

Change history

Please check CHANGELOG and NOTICE

Timing of value manipulation in Cubism SDK for Cocos Creator

When manipulating the values of model parameters in Cocos Creator, it is recommended to do so at the time of Component.**lateUpdate** ().

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

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

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

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

In Live2D Cubism SDK for Cocos Creator, animation playback uses Cocos Creator's built-in functions Animation, which apply parameter values from `Component.update()` to `Component.lateUpdate()`. Therefore, if you set the value of a parameter in `Component.update()`, the animation may overwrite that value due to the execution order.

For more information on Cocos Creator's event functions, please refer to the official Cocos Creator documentation. Cocos Creator official documentation]() for more information about Cocos Creator event functions.

Performance Tuning of Cubism SDK for Cocos Creator

Overview

In Cubism SDK, the structure of a model may affect its performance on the program. In addition, Live2D SDK for Cocos Creator prioritizes clarity of structure, which intentionally sacrifices performance in some areas. Below is an explanation of how the performance of the SDK is affected.

Model Structure

For more information, see here.

Project structure

Cocos Creator event functions. As mentioned above, Live2D SDK for Cocos Creator prioritizes clarity of structure. Therefore, update processing of each component is performed from Cocos Creator event functions such as lateUpdate(). These Cocos Creator event functions are not costly to call. If multiple models are displayed using the SDK as is, performance may be very poor depending on the execution environment. If performance is important, it is recommended to call the update process of each component from a single Controller. The **CubismUpdateController** included with the Cubism SDK for Cocos Creator performs similar processing to control the execution order of Cubism SDK components, so if you want to address the above CubismUpdateController implementation will be helpful.

For details on CubismUpdateController, see Tutorial - How to setup UpdateController.

Parameter manipulation in Cubism SDK for Cocos Creator

When manipulating the values of model parameters in Cubism SDK for Cocos Creator, it is necessary to do so at the timing of `Component.lateUpdate()` in Cocos Creator's event function.

See here for the execution order of Cocos Creator's event functions.

The process of playing back an AnimationClip is done between `Component.update()` and `Component.lateUpdate()` in Cocos Creator's event functions. If the value of a parameter is manipulated in `Component.update()`, it will be overwritten by the value of the AnimationClip that is played immediately after. Therefore, manipulation of parameter values should be done after the AnimationClip has set the value.

All components bundled with Cubism SDK for Cocos Creator manipulate parameter values from `Component.lateUpdate()`.

Also, since the model's vertex update process from the parameter values at each frame is performed in Cocos Creator's post-draw event (Director.EVENT_AFTER_DRAW), any value manipulation later than this will not be computed.

[Cocos Creator] Multiply Color Screen Color

By applying multiply and screen colors to a model, you can change the hue in real time. Multiply and Screen colors set in Cubism Editor can be applied without any additional coding by using SDK for Cocos Creator (Cubism 4.2 or later). Please refer to "Multiply & Screen Colors" in the Editor manual for details on setting multiply and screen colors in Cubism Editor.

Also, by coding as needed, it is possible to manipulate multiply and screen colors from the SDK, and the following operations are also possible.

- Interactively apply multiply and screen colors
- Apply multiply/screen colors not set in Cubism Editor.
- Disable multiply and screen colors set on Cubism Editor.

The following is an explanation of the procedure.

Processing Procedure

The following is the flow of processing.

- Model placement
- Set overwrite flag for multiply and screen colors
- Set multiply and screen colors

Placing Models

Place the model for which you want to set the multiply and screen colors in any scene.

Set the overwrite flag for multiply and screen colors

Set the overwrite flag for multiply and screen colors to true. The default is false, which means that the color information from the model is used.

There are two types of override flags: one for the entire model, which is held by [CubismRenderController], and one for each Drawable object, which is held by [CubismRenderer].

An example code is shown below. It assumes that the script is attached to the root object of the model as a component.

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

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

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

If the Overwrite flag for the entire model is enabled, it is possible to manipulate the multiply and screen colors from the SDK even if the Overwrite flag for an individual Drawable is disabled.

[CubismRenderController] is added to the root object of the model as a component, and can be obtained by using getComponent. [CubismRenderController] also allocates [CubismRenderer] of each Drawable object in the model as an array, and each [CubismRenderer] can be referenced from [CubismRenderController]. CubismRenderController] can reference each [CubismRenderer].

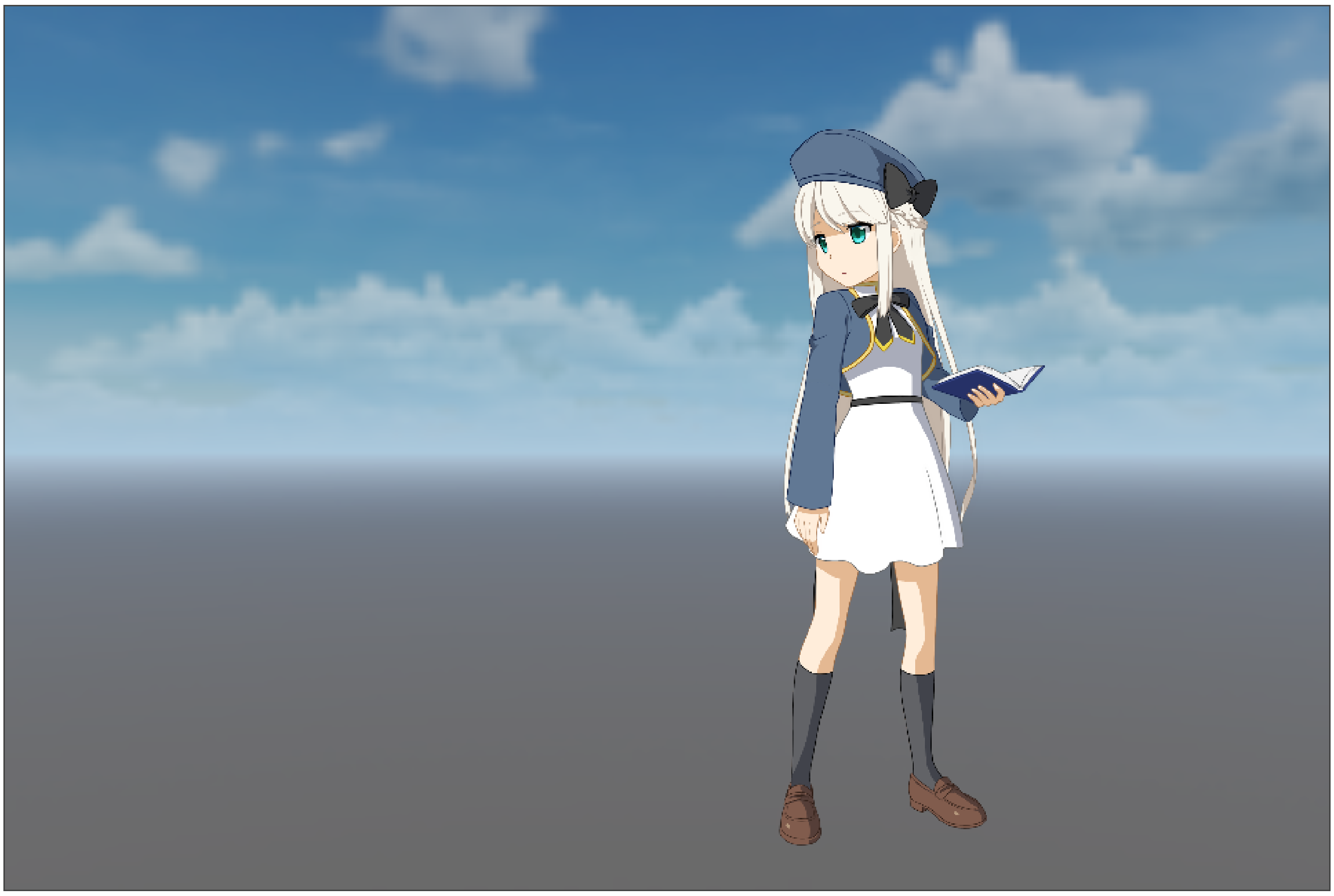
Setting Multiply and Screen Colors

Multiply and screen colors are defined and set for the model. In the code below, red is set as the multiply color and green is set as the screen color for all Drawables. The set colors are stored in each [CubismRenderer] with the math.Color type. In the example below, the colors are set in RGBA, but A is not used in the calculation of the multiply and screen colors.

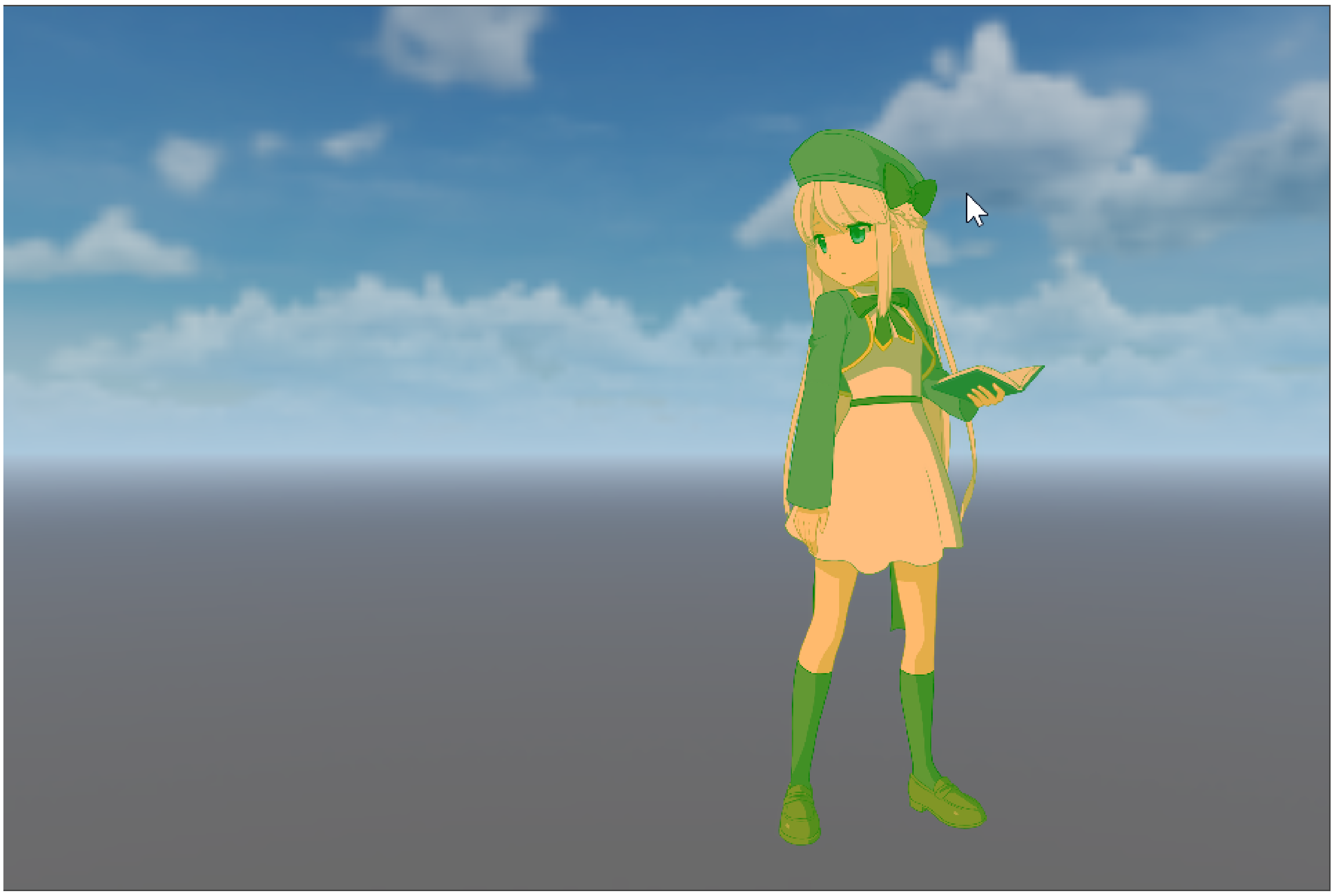
```
const multiplyColor = new math.Color(1.0, 0.5, 0.5, 1.0);
const screenColor = new math.Color(0.0, 0.5, 0.0, 1.0);

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

Before applying multiply and screen colors



After applying red for multiply color and green for screen color



The model's multiply and screen color process is updated in all Drawables when there is a change in color information.

Tips

In this example, the same multiply and screen colors are set for all drawables, but it is possible to set different multiply and screen colors for each drawable. To disable the multiply and screen colors, use the

To multiply color (1.0, 1.0, 1.0, 1.0) To screen color (0.0, 0.0, 0.0, 1.0)

This can be done by setting the

Other related functions/processes

Receive notification of multiply/screen color updates from the model side

If a model parameter is associated with a change in multiply/screen color, the model may change the multiply/screen color when the model is animated, etc., rather than from the SDK.

The property isBlendColorDirty is implemented in [CubismDynamicDrawableData] to indicate that the multiply and screen colors have been changed.

This property is a flag that is set when either the multiply or screen color is changed on the model side, and does not determine whether the multiply or screen color is changed.

To handle [CubismDynamicDrawableData] data, the event onDynamicDrawableData of [CubismModel] must be used.

Register Functions

```
let model: CubismModel;

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

Example of function to register

```
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

Overview

EyeBlink is a function that applies an open/closed state value to the current value of a blink parameter. See here for information on how to set blink parameters for your model.

The parameters for blinking can be set in the model itself or specified arbitrarily by the user on Cocos Creator.

EyeBlink in Cubism SDK for Cocos Creator is composed of three types of elements.

1. component for specifying parameters
2. component for applying values to each parameter
3. manipulation of the values applied by 2.

1. component for specifying parameters

Use CubismEyeBlinkParameter to specify parameters to be used for EyeBlink.

CubismEyeBlinkParameter is a component that inherits from Component It is used by attaching it to a GameObject placed under [Prefab root]/Parameters/. The parameter with the same ID as the GameObject to which it is attached is treated as a parameter for blinking.

If the model itself has a parameter for blinking, the CubismEyeBlinkParameter will be attached to the GameObject for that parameter during import.

The CubismEyeBlinkParameter is used as a marker to get a reference, so nothing is processed internally and no data is held.

2. component to apply a value to each parameter

To apply an open/close value to each parameter, use CubismEyeBlinkController. This is a component that inherits from Component and is attached to the root of Cubism's Prefab when used.

At initialization, retrieve references to all CubismEyeBlinkParameters attached to the Prefab. If you add/remove a blink parameter during execution, call CubismEyeBlinkController.refresh() to get the reference again.

```
  /** 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 applies the value of CubismEyeBlinkController. eyeOpening to the parameters marked by CubismEyeBlinkParameter at the timing of onLateUpdate() in every frame. eyeOpening values are applied to the parameters marked by CubismEyeBlinkParameter.

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

The value set for EyeOpening ranges from 0.0f to 1.0f. CubismEyeBlinkController applies this value to the target parameter using the calculation method set in CubismEyeBlinkController.blendMode.

By manipulating this EyeOpening value from the outside, the model's eyes can be opened and closed.

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

3. Manipulating values to be applied in 2.

As explained in "2. Components to Apply Values to Each Parameter," you can apply values to the parameter for blinking by manipulating the value of CubismEyeBlinkController.eyeOpening. Cubism SDK for Cocos Creator provides two ways to manipulate this value

Manipulate values by motion

Manipulate values by component The user can also customize the blink speed, timing, etc. by implementing a process to manipulate this value.

Tips

If the execution order of components that manipulate CubismEyeBlinkController.eyeOpening is after CubismEyeBlinkController, the intended behavior may not be achieved. If a problem occurs, it is possible to work around it by explicitly controlling the execution order of components on the user side. Cubism SDK for Cocos Creator controls the execution order of each component with CubismUpdateController, which can also be used.

In addition, since the above two setting methods manipulate the same value at different times, it is difficult for both to coexist in a single model without any innovations.

Manipulating values by motion

When creating motion in Cubism's Animator using a model with parameters for blinking, it is possible to set a curve for blinking.

When a .motion3.json file with a blink curve is imported into a project, the AnimationClip will have that curve generated for the value of CubismEyeBlinkController.eyeOpening. Therefore, when the AnimationClip is played in the Animator component, the value of CubismEyeBlinkController.eyeOpening will be manipulated.

Manipulating values with components

In Cubism SDK for Cocos Creator, blink values can also be manipulated by the CubismAutoEyeBlinkInput component.

CubismAutoEyeBlinkInput calculates and sets the value for blinking from the speed, interval, and random fluctuation width added to the interval set from Inspector.

```
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 has three settings.

- 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 Sets the amount of time before blinking.
  The unit is seconds.
  In practice, this value plus the error due to Maximum Deviation is used.

- maximumDeviation Sets the width of the random fluctuation to be added to the time set for Mean.
  These are calculated as follows

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

- timescale It will be the speed at which you blink.
  Multiplied by the time elapsed since the previous frame.

HarmonicMotion

Overview

HarmonicMotion is a function that periodically repeats the value of a specified parameter. It is mainly used for things that are constantly moving, such as breathing. For information on how to set up HarmonicMotion, please refer to Tutorial - How to make a parameter move cyclically.

HarmonicMotion in Cubism SDK for Cocos Creator consists of two elements.

1. component for specifying parameters to be operated
2. component for manipulating the value of each parameter

1. component for specifying parameters to operate

Use **CubismHarmonicMotionParameter** to specify the parameters to be used for HarmonicMotion.

The CustomHarmonicMotionParameter is a component that inherits from Component \모듈을 보려면 [Prefab のルート]/Parameters/ and is used by attaching it to a Node placed under the Node. It will cycle through the values of the parameters with the same ID as the Node to which it is attached.

The CustomHarmonicMotionParameter has five setting items.

```
  /** 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
  . Specifies the multiplier for the sine wave period set by the CustomHarmonicMotionController.
  HarmonicMotion allows multiple periods to be set for a single model, which can be set in theCubismHarmonicMotionController.
  Here, set the index of the CubismHarmonicMotionController.channelTimescales.

- direction
  . Sets the range of periodic motion with respect to the center of the parameter.
  <br There are three setting items
  - Left : Operates only in the left half of the range from the center of the parameter.
  - Right : Operates only in the right half of the range from the center of the parameter.
  - Centric : Works on the entire parameter.

- normalizedOrigin<br Sets the center of the parameter to be referenced by direction.
  Sets the center of the parameter to its minimum value of 0 and maximum value of 1.

- normalizedRange
  . Sets the amplitude at which to operate the value from the center of the value set by normalizedOrigin.
  Sets the distance to move the value from the center when the minimum value for that parameter is 0 and the maximum value is 1.
  This value can only range from the position of the center set in normalizedOrigin to the minimum or maximum value of the parameter.

- duration
  . Adjusts the period of the parameter.

```
  /** 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];
    }
```

The CustomHarmonicMotionParameter is also used as a marker for the CustomHarmonicMotionController to obtain references.

2. components that manipulate the value of each parameter

To apply opening and closing values to each parameter, use **CubismHarmonicMotionController**. This is a component that inherits from Component and attaches to the root of Cubism's Prefab when used.

At initialization, retrieves references to all CubismHarmonicMotionParameters attached to the Prefab. If you add/remove a parameter whose value is to operate cyclically during execution, call CubismHarmonicMotionController.refresh() to retrieve the reference again.

```
  /** 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;
  }

  ...

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

The CustomHarmonicMotionController applies the calculated values for the parameters marked by CustomHarmonicMotionParameter at the time of lateUpdate() each frame.

```
  /** 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);
    }
  }
```

The CustomHarmonicMotionController has two setting items.

```
  /** 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
  . The blend mode used when applying values to parameters.
  The following three values can be set
  - Override : Override the current value.
  - Additive : Add to the current value.
  - Multiply : Multiply the current value.
- channelTimescales
  . Sets the period of the sine wave. Multiple periods can be set.

Json

Overview

Live2D Cubism handles some of the runtime data in json format. Cubism SDK for Cocos Creator includes classes to parse and instantiate those json format files.

They are parsed and instantiated when imported by the asset importer included with the SDK, but can also be loaded at run-time by the user.

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

CubismModel3Json

Parser for .model3.json.

You can get the path to various other json, such as motion and facial expressions described in .model3.json.

Use CubismModel3Json.loadAtPath() to parse JsonAsset in CocosCreator default resources.

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

If you want to parse .model3.json file other than CocosCreator default resources, you can get the string by any method and pass the object processed by JSON.parse() to CubismModel3Json.loadFromJson().

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

From the parsed data, it is possible to obtain the relative paths to the various files described in .model3.json.

```
// .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;

...
```

Basically, the path can be obtained with the same structure as the .model3.json hierarchy, but the reference to the data for expressions described in .model3.json has its own structure.

Cocos Creator cannot generate a prefab using CubismModel3Json.toModel() because it is not possible to collect assets on the Editor Platform.

If you need to create a Prefab with your own workflow, you will need to implement it on your end.

The Cubism SDK for Cocos Creator is designed to load assets in a project from the Cocos Creator editor. If you want to load assets from AssetBundle or other sources at runtime, you need to implement the loading process on your own.

CubismMotion3Json

Parser for .motion3.json.

AnimationClip can be generated from the curve information described in .motion3.json.

To parse .motion3.json in CubismMotion3Json, use CubismMotion3Json.loadFrom().

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

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

To generate an AnimationClip from the parsed .motion3.json, use CubismMotion3Json.toAnimationClip().

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

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

CubismUserData3Json

Parser for .userdata3.json.

User data can be applied to the model's art mesh from the information described in .userdata3.json.

To parse .userdata3.json in CubismUserData3Json, use CubismUserData3Json.loadFrom().

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

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

To retrieve user data from the parsed .userdata3.json, use CubismUserData3Json.toBodyArray().

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

CubismPhysics3Json

Parser for .physics3.json.

The physics settings described in .physics3.json can be converted for use in Cocos Creator.

To parse .physics3.json in CubismPhysics3Json, use CubismPhysics3Json.loadFrom().

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

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

To convert physics settings from the parsed .physics3.json to the format handled by Cocos Creator, use CubismPhysics3Json.ToRig().

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

CubismExp3Json

Parser for .exp3.json.

You can convert the information on facial expression differences described in .exp3.json into a format that can be handled on Cocos Creator.

To parse .exp3.json in CubismExp3Json, use CubismExp3Json.loadFrom().

```
const json = resources.load<TextAsset>("path/to/file").text;
const exp3Json = CubismExp3Json.loadFrom(json);
```

CubismPose3Json

Parser for .pose3.json.

The information described in .pose3.json can be used to obtain settings that control the display state of the part on Cocos Creator.

The information described in .pose3.json can be used to obtain settings that control the display state of the part on Cocos Creator.

See here for more details on the Pose feature. In Cubism SDK for Cocos Creator, Pose is used to modify the curve of the generated AnimationClip.

To parse .pose3.json in CubismPose3Json, use CubismPose3Json.loadFrom().

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

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

CubismDisplayInfo3Json

Parser for .cdi3.json. .cdi3.json describes the names of parameters, parts, and parameter groups set in the Cubism editor, and each ID paired with them. If .cdi3.json does not exist for a model, the IDs are displayed.

In Cubism SDK for Cocos Creator, it is used to display parameters and part names in the Inspector window. To parse .cdi3.json in CubismDisplayInfo3Json, use CubismDisplayInfo3Json.loadFrom().

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

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

LookAt

Overview

LookAt is a function that manipulates the value of an arbitrary parameter to follow a specific coordinate. By customizing the coordinates to be followed by the user, Cubism models can be made to follow specific GameObjects and other objects. For more information on how to use LookAt, please refer to Tutorial - Setting Up LookAt.

LookAt in Cubism SDK for Cocos Creator is composed of three types of elements.

1. component for specifying parameters to be tracked
2. component for applying values to each parameter
3. manipulation of values applied by 2.

1. component for specifying parameters to be followed

Use CubismLookParameter to specify parameters to be followed by LookAt.

CubismLookParameter is used by attaching it to a GameObject placed under [Prefab root]/Parameters/. The parameter with the same ID as the GameObject to which it is attached is treated as a look tracking parameter.

CubismLookParameter has two setting items, Axis and Factor.

```
/** 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

Sets which axis value of the entered coordinates will be used. There are three items that can be set: X, Y, and Z.

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

- Factor

Sets the correction value to the value to be applied. The input coordinates will be processed to a value in the range of -1.0f to 1.0f when applied. The value set for Factor is a multiplying factor that multiplies the input values to match the minimum and maximum values of each parameter.

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

CubismLookParameter is also used as a marker for CubismLookController to obtain references.

2. component to apply a value to each parameter

To apply eye tracking to each parameter, use CubismLookController. When using CubismLookController, attach it to the root of Prefab of Cubism.

At the initialization of CubismLookController, get references to all CubismLookParameters attached to Prefab. When parameters for eye tracking are added/removed during execution, CubismLookController.Refresh() is called to reacquire the references.

```
/** 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;
  }

  ...

  /** 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 applies the coordinates returned by the object set in CubismLookController.Target to the parameters marked by CubismLookParameter at the timing of LateUpdate() each frame. Target is applied to the parameters marked by CubismLookParameter.

```
  // 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. Manipulation of values to be applied in 2.

As explained in "2. Components to Apply Values to Each Parameter," the coordinates that CubismLookController applies to the parameters for eye tracking will return the object set in CubismLookController.Target.

This is an implementation of the ICubismLookTarget interface, which can be implemented by the user to track the model to arbitrary coordinates.

```
/** 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;

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

- GetPosition()

Returns the coordinates to follow. The coordinates returned here are treated as world coordinates.

- IsActive()

Returns whether or not tracking is enabled. Only if true is returned will tracking be performed.

The SDK ships with CubismLookTargetBehaviour as an example implementation of ICubismLookTarget. CubismLookTargetBehaviour is an example that returns the coordinates of the GameObject to which it is attached.

```
/** Straight-forward {@link ICubismLookTarget} {@link Component} */
@ccclass('CubismLookTargetBehaviour')
export default class CubismLookTargetBehaviour extends Component implements
ICubismLookTarget {
  readonly [ICubismLookTarget.SYMBOL]: typeof ICubismLookTarget.SYMBOL =
ICubismLookTarget;

  //#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

Overview

MouthMovement is a function that applies open/closed state values to the current values of parameters for lip-sync. It is possible to apply lip-sync curves set in motion or sampled in real-time from a playing audio file to a model. See Tutorial - Lip Sync Settings for information on how to set lip-sync parameters to a model.

The only thing you can set with MouthMovement is the state of mouth opening and closing. It is not possible to set the shape of the mouth to match the vowel.

To specify parameters for lip-sync on Cocos Creator, the model can be set in the Cubism editor, or the user can specify them arbitrarily on Cocos Creator.

MouthMovement in Cubism SDK for Cocos Creator consists of three types of elements.

1. component for specifying parameters
2. component for applying values to each parameter
3. manipulation of values applied by 2.

1. component for specifying parameters

Use CubismMouthParameter to specify parameters to be used for MouthMovement.

CubismMouthParameter is a component that inherits from Component CubismMouthParameter is a component inheriting from Component and is used by attaching it to a Node placed under [Prefab root]/Parameters/. The parameter with the same ID as the Node to which it is attached is treated as a parameter for lip-sync.

If the model itself has parameters for lip-sync, a CubismMouthParameter will be attached to the Node of that parameter during import.

CubismMouthParameter is used as a marker to get a reference, so it does not process anything internally and has no data.

2. component to apply a value to each parameter

To apply an open/close value to each parameter, use CubismMouthController. This is a component that inherits from Component and attaches to the root of Cubism's Prefab when used.

At initialization, retrieve references to all CubismMouthParameters attached to Prefab. If a blinking parameter is added/removed during execution, call CubismMouthController.refresh() to get the reference again.

```
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 applies the value of CubismMouthController.MouthOpening to the parameters marked by CubismMouthParameter at the timing of lateUpdate() in every frame.

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

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

The value set for MouthOpening ranges from 0.0f to 1.0f. CubismMouthController applies this value to the target parameter using the calculation method set in CubismMouthController.blendMode.

By manipulating this MouthOpening value from the outside, the model's mouth can be opened and closed.

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

3. Manipulating values to be applied in 2.

As described in "2. Components to apply values to each parameter," values can be applied to parameters for lip-sync by manipulating the values in CubismMouthController.mouthOpening.

Cubism SDK for Cocos Creator provides three different ways to manipulate this value

- Manipulate values by motion
- Manipulate values periodically
- Manipulate values by sampling from AudioClip

Users can also customize their own lip-sync speed and timing by implementing a process to manipulate this value.

Tips

If the execution order of components operating CubismMouthController.mouthOpening is after CubismMouthController, the intended behavior may not be achieved. If a problem occurs, it is possible to work around it by explicitly controlling the execution order of components on the user side. Cubism SDK for Cocos Creator controls the execution order of each component with CubismUpdateController, which can also be used.

In addition, since each of the above three setting methods manipulates the same value, it is difficult for them to coexist in a single model without any innovations.

Manipulating values by motion

When creating motion in Cubism's Animator using a model with parameters for blinking, it is possible to set a curve for blinking.

When a .motion3.json file with a blink curve is imported into a Cocos Creator project, the Animation will be generated with the value of CubismMouthController.mouthOpening. Therefore, the value of CubismMouthController.mouthOpening will be manipulated when the AnimationClip is played by Animator component or other components.

Manipulate values cyclically

To have the values for lip-sync manipulated periodically, use CubismAutoMouthInput. CubismAutoMouthInput is a component that calculates and sets the value for lip-sync with a sine wave.

To use CubismAutoMouthInput, attach it to the root of Cubism's Prefab.

CubismAutoMouthInput has one setting item.

- Timescale
  . Changes the period of the sine wave.

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

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

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

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

Manipulating values by sampling from audio

Use CubismAudioMouthInput to set lip-sync values from audio played on Cocos Creator.

CubismAudioMouthInput generates and sets lip-sync values in real-time by sampling from the audio information during playback obtained from AudioSource.

To use CubismAudioMouthInput, attach it to the root of Cubism's Prefab.

```
  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 has four settings.

```
  @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
  . Reference to the AudioSource to be sampled.
- samplingQuality
  . The accuracy of the audio to be sampled.

- gain
  . The multiplier of the sampled value.
  Equal to 1. The larger the value, the larger the lip-sync value.

- smoothing
  . The amount of smoothing of the sampled value.
  The larger the value, the smoother the lip-sync value will change.

Raycasting

Overview

Raycasting is a function that determines whether a user-specified Cubism mesh intersects with arbitrary coordinates. It is possible to obtain the coordinates of a click/tap or a hit judgment between any object in the scene and the specified mesh. For more information on how to use Raycasting, please refer to the Tutorial - Setting up a hit detection.

Raycasting in Cubism SDK for Cocos Creator consists of two major components.

1. component for specifying the art mesh to be judged
2. component for judging whether the input coordinates correspond to the specified art mesh

1. component for specifying the art mesh to be judged

Use CubismRaycastable to specify the mesh to be used for judging.

CubismRaycastable is used by attaching it to a Node placed under [Prefab root]/Drawables/. The art mesh with the same ID as the Node to which it is attached will be used for hit detection.

CubismRaycastable has a parameter that sets the accuracy of its mesh hit detection.

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

CubismRaycastable.Precision has two values that can be set, each as follows.

- BoundingBox
  . A rectangle with four horizontal or vertical sides that surrounds the mesh and is used as a bounding box.
  <br Depending on the shape of the mesh, it may also take hits at coordinates outside the mesh, but it offers better performance than Triangles.

- Triangles
  . The shape of the mesh is used as the range of the triangles.
  <br It has less performance than the BoundingBox, but it provides a more accurate range.

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

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

2. component to determine the hit between the input coordinates and the specified art mesh

CubismRaycaster is used to obtain the actual hit detection. When using CubismRaycaster, attach it to the root of Prefab of Cubism.

At initialization of the CubismRaycaster, obtain references to all CubismRaycastables attached to the Prefab. When a hit mesh is added/removed during execution, CubismRaycaster.refresh() is called to reacquire the reference.

```
  /** 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();
  }
```

To get a hit from the coordinates, use CubismRaycaster.raycast1() or CubismRaycaster.raycast2().

```
  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;
    }
```

raycast() returns the number of meshes for which the hit detection was obtained in the return value. The instance of CubismRaycastHit[] type passed as an argument is set to the information of the meshes that were hit.

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

      ++hitCount;
```

CubismRaycaster.raycast() retrieves up to the number of elements of the instance of type CubismRaycastHit[] if multiple meshes overlap on the same coordinate. If more meshes than the number of elements are overlapped, the result will not be retrieved for the excess meshes.

CubismRaycastHit is a class that has information about the mesh from which the hit was obtained.

```
  /** 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
  . A reference to the art mesh from which the hit decision was obtained.
- distance
  . Distance from the specified coordinates.
  Linear distance between origin or ray.origin passed as argument to CubismRaycaster.Raycast() and Transform.position of Drawable.
- localPosition
  . The local coordinates of the art mesh from which the hit decision was taken.
- worldPosition
  . The world position of the art mesh from which the hit is obtained.

UserData

Overview

UserData is a function that visualizes user data set in the model's art mesh in Cocos Creator. UserData is a function that allows arbitrary tags to be assigned to art meshes, and can be used to specify art meshes for special processing in the SDK.

For more information on user data, see here.

To use user data on Cocos Creator, follow these steps

1. parse .userdata3.json
2. set UserData to art mesh

1. parse .userdata3.json

Parsing .userdata3.json using CubismUserData3Json. See here for more information about CubismUserData3Json.

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

The path to .userdata3.json can also be obtained from .model3.json using CubismModel3Json. The path that can be obtained is relative to .model3.json.

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

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

For more information on CubismModel3Json, click here.

Get the art mesh data from the parsed .userdata3.json.

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

2. Set UserData to the art mesh

Apply the user data obtained from .userdata3.json to the Prefab art mesh.

To add user data information to an art mesh, use the CubismUserDataTag.

```
    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

For Original Workflow ("OW"), please see here.

Overview

Adding a component for OW will allow OW to be expressed without changing the overall feel of the SDK.

To check the functionality of OW, use the Viewer for OW included with Editor.

OW mode import differs from non-OW mode import in the following four ways

- The editor extension adds an editor menu for OW mode on Cocos Creator.
  - Live2D/Cubism/OriginalWorkflow/ **Toggle should Import As Original Workflow**" This item is for toggling OW mode.
    <br If you import model data with this item enabled, a model will be created that supports OW.
  - Live2D/Cubism/OriginalWorkflow/Should Clear Animation Curves"
    This item is for clearing AnimationClip curves.
    If you re-import a motion with this item set to Disable, it will overwrite the existing AnimationClip.
    If you re-import a motion with this item enabled (Enable), all AnimationCurves set in the existing AnimationClip will be cleared before the motion is overwritten.
    This feature is only available when "Live2D/Cubism/OriginalWorkflow/Should Import As Original Workflow" is checked.
- The following OW components are attached
  - CubismUpdateController
  - CubismParameterStore
  - CubismPoseController
  - CubismExpressionController
  - CubismMotionFadeController

Opacity curves set in AnimationClip will be processed for OW. The .fade, fadeMotionList, .exp, and .expressionList assets will be added to the project.

CubismParameterStore

Overview

CubismParameterStore allows you to save/restore CubismModel parameters and part values.

If CubismParameterStore is not used, the results of processing values with Expression or other methods may not be correct.

If a value manipulation such as Expression is performed between Restore and Save, the result of the value manipulation is saved, resulting in addition/multiplication to the restored, post-manipulation value, which does not produce the expected result. If the value manipulation is performed outside of Restore-Save, the state before the value manipulation is restored, so subsequent manipulation of the value by addition/multiplication will result in the proper result.

See the corresponding tutorial article in Tutorial - Save/Restore the values to be manipulated.

The following process is used to save/restore CubismModel parameter and part values.

1. get a reference to the parameter and part to be saved/restored
2. save parameter and part values 3. restore parameter and part values Restore parameter and part values

Get references to parameters and parts to save/restore

CubismParameterStore.onEnable caches references to CubismModel parameters and parts.

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

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

This is done with `CubismParameterStore.onEnable()`.

Save parameter and part values

Saves the current parameter and part values for the model. The timing for saving the values is after applying the animation and before manipulating the values in the various Cubism components.

```
    // 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;
      }
    }
```

This is done with `CubismParameterStore.saveParameters()`.

Restore parameters and part values

Restores saved parameter and part values. The timing for restoring values is before applying the animation.

```
    // 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];
      }
    }
```

This is done with `CubismParameterStore.restoreParameters()`.

CubismUpdateController

Overview

By attaching CubismUpdateController to Prefab, you can control the execution order of each component of Cubism SDK for Cocos Creator.

Since the execution order of each component is important in the OW method, it is essential to use CubismUpdateController to comply with the specification. If CubismUpdateController is not used, for example, because the execution order of components is determined by Cocos Creator, functions such as Expression may not be processed correctly depending on the execution order.

To control the order of execution of user-created components, implement ICubismUpdateable on the component.

Implementing ICubismUpdateable

The ICubismUpdateable interface is inherited by the component that controls the execution order.

If CubismUpdateController is not attached, the onLateUpdate process will not be executed, so the following snippet allows that component to work by itself.

```
@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 {
    // Implement LateUpdate processing.
  }

  /** 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) {
      // If CubismUpdateController is not attached, process from its own
LateUpdate.
      this.onLateUpdate();
    }
  }

  ...

}
```

- hasUpdateController
  - Sets whether CubismUpdateController is attached to Prefab.
- needsUpdateOnEditing
  - Returns whether Cocos Creator needs to update the scene while it is not running.
    Returns false if the component does not need to be updated while it is not running, since CubismUpdateController is also run in editor mode.
- executionOrder
  - Returns the execution order of the component.
    The component with the lowest value is called first.
    <br The execution order of each component in Cubism SDK for Cocos Creator is defined in CubismUpdateExecutionOrder.
- If a user adds a component that controls the execution order, please refer to the definition of CubismUpdateExecutoinOrder and note when it is called by the value returned here.

    - Example (if you want your own component to be called at a timing between CubismLookController and CubismPhysicsController)
        - CubismLookController is set to 700 and CubismPhysicsController is set to 800, so set your own component to return 750.

- onLateUpdate()
    - Function called by CubismUpdateController to control the order of execution.

      Describes the update process to be performed by the component. Describes the update process to be performed by the component.

Caution when adding or removing components during execution

Components whose execution order is controlled by CubismUpdateController are done at the timing of `onEnable()`. If all components are attached to the prefab in advance, there is no problem, but if you add or remove components dynamically during scene execution, you must explicitly call `CubismUpdateController.refresh()` to reload them. CubismUpdateController.refresh()` explicitly to reload the scene.

```
this.addComponent(MyComponent1);

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

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

Expression

Overview

Expression is a function that handles Cubism's facial motion, which can be set by adding or multiplying the value of the parameter for the facial expression to the value of the current parameter. In Animator, the standard motion playback function of Cocos Creator, this function is not available because Multiply is not among the blend modes that can be set for layers.

The configuration file for the expression function is exported in .exp3.json format. Please refer to here for more information on the mechanism of expressions.

To use Expression, it is necessary to set up UpdateController and ParameterStore in Prefab in advance.

See Tutorial - Using Facial Expressions for the corresponding tutorial article.

To play back the facial expression motion, perform the following steps.

1. create [.exp3.asset
2. create [.expressionList.asset] and add [.exp3.asset
3. play the facial expression motion 4. Calculate and apply expression motion

Create [.exp3.asset

[.exp3.asset] is a ScriptableObject asset converted from [exp3.json]. If [.exp3.asset] is modified, normal operation cannot be guaranteed.

To convert [exp.json] to [.exp3.asset], do the following

1. parse [exp.json
2. create CubismExpressionData
3. create [.exp3.asset

Parsing [exp.json

To parse [exp3.json], use CubismExp3Json.loadFrom(string exp3Json) or CubismExp3Json.loadFrom(TextAsset exp3JsonAsset).

CubismExpressionData Creation

CubismExpressionData is a class that records information in the parsed .exp3.json and holds the following data

field	type	description
Type	string	json file type. For .exp3.json, "Live2D Expression" is set.
FadeInTime	number	The time it takes for the expression to fade in. The unit of the value is seconds.

field	type	description
FadeOutTime	number	The time it takes for the expression to fade out. The unit of the value is seconds.
Parameters	SerializableExpressionParameter[]	The ID of the parameter to which the expression is applied, the value to be applied, and the calculation method.

Use CubismExpressionData.CreateInstance(CubismExp3Json json) to create a new [.exp3.asset].

Use CubismExpressionData.CreateInstance(CubismExpressionData expressionData, CubismExp3Json json) to overwrite [.exp3.asset].

- CubismExp3Json json: CubismExp3Json is the data parsed from [exp3.json].
- CubismExpressionData expressionData: CubismExpressionData to be overwritten.

Creating [.exp3.asset

Create [.exp3.asset] as follows

```
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);
```

This is done by CubismExpression3JsonImporter.import().

- For runtime, it is not necessary to .asset CubismExpressionData.

Create [.expressionList.asset] and add [.exp3.asset

[.expressionList.asset] is an asset that lists references to [.exp3.asset] for each model. It is used in CubismExpressionController to retrieve the expression motion to play from CurrentExpressionIndex. The order of the [.exp3.asset] list is added in the import order of [exp3.json].

Creating [.expressionList.asset

[.expressionList.asset] is created in the same hierarchy as the model prefab.

```
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));
```

__* This item is processed by CubismExpressionListImporter.import().__

Adding [.exp3.asset

To add [.exp3.asset] to [.expressionList.asset], we do the following

```
      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);
```

__* This item is processed by CubismExpressionDataImporter.import().__

Playing Expression Motion

Expression motions can be played or changed by setting the index of the expression motion to be played to CubismExpressionController.CurrentExpressionIndex.

The following process is used to play back the facial expression motion.

1. set the end time of the facial expression motion currently playing
2. initialize a new facial expression motion and add it to the playback list of the facial expression motion

Set the end time for the facial expression motion being played back

Sets the end time so that the expression motion during playback, if it exists, will end.

```
      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;
      }
```

This is done in CubismExpressionController.startExpression().

Initialize a new facial expression motion and add it to the facial expression motion playback list

Create a CubismPlayingExpression with the expression motion playback information.

CubismPlayingExpression holds the following information

field	type	description
Type	string	json file type. For .exp3.json, "Live2D Expression" is set.
FadeInTime	number	The time it takes for the expression to fade in. The unit of the value is seconds.
FadeOutTime	number	The time it takes for the expression to fade out. The unit of the value is seconds.
Weight	number	The weight of the expression. Values range from 0 to 1.

field	type	description
ExpressionUserTime	number	The time elapsed since the expression started playing. The unit of the value is seconds.
ExpressionEndTime	number	The time elapsed since the end of expression playback. The unit of the value is seconds.
Destinations	CubismParameter[]	An array of parameters applied by the expression.
Value	number[]	The value that the expression applies to the parameter. The number of elements and the order are the same as those of Destinations.
Blend	CubismParameterBlendMode[]	The calculation method applied to the parameter by the expression. The number of elements and the order are the same as Destinations. The calculation methods set are Override, Additive, and Multiply.

CubismPlayingExpression can be created using CubismPlayingExpression.create(CubismModel model, CubismExpressionData expressionData).

- CubismModel model: Model to play expression motion.
- CubismExpressionData expressionData: [exp3.asset] data of expression motion to be played.

The created facial expression motion is added to the playback list as follows

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

if (playingExpression == null) {
  return;
}

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

This is done in CubismExpressionController.startExpression().

Calculating and Applying Expression Motion

Expression calculation and application is called from CubismExpressionController.onLateUpdate().

Expression must be applied after ParameterStore.saveParameters(). Expression is a function that applies a new value for the expression every frame to the current value. If it is applied before ParameterStore.saveParameters(), the result applied by Expression will be saved, and the values of the parameters for Expression will be added or multiplied infinitely.

Pose

Overview

Pose is a function that manages the display state of a group of parts described in the Pose configuration format .pose3.json. By using Pose, you can manage parts that you do not want to be displayed on the screen at the same time, such as "arms in a lowered state" and "arms in a crossed state". Pose can also be used to support models with different Part IDs (e.g., costume changes).

.pose3.json can be configured in Cubism Viewer for OW. Please see here for more information on how to configure the parts.

When comparing a model imported by the OW method with a model imported by the conventional method, the following three points change depending on Pose.

- CubismPosePart is attached to the following GameObjects in [Prefab root]/Parts/.
- If the curve of PartOpacity in AnimationClip is Stepped, it will be processed into a Linear curve.
- Opacity curves that are not described in pose3.json are removed from AnimationClip.

Pose performs the following processes

1. process the Opacity curve of the Parts set in Stepped in the Pose motion
2. identify the parts for Pose
3. save the Opacity of the parts to determine which Pose parts to display
4. calculate and apply Pose
5. link opacity

Processing curves for Opacity (PartOpacity) of Parts set in Stepped in Pose motion

As described in the Editor manual Pose Settings, the PartOpacity curve handled in Pose is a Stepped curve. The curve type is assumed to be Stepped. In Cubism SDK for Cocos Creator, this curve is processed to Linear and exported to AnimationClip.

If you set a curve of a type other than Stepped, the curve will not be processed and will be exported directly to AnimationClip.

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

The processing of this Opacity curve is done in CubismMotion3Json.toAnimationClipB().

Identifying Parts for Pose

CubismPosePart is attached when a model is imported using the OW method. The Node to be attached will be the same name as the ID described in .pose3.json, placed under [Prefab root]/Parts/.

```
// Get parts for Pose
const part = ArrayExtensionMethods.findByIdFromParts(
  part,
  group[partIndex].id
);

// Attach CubismPosePart
posePart = part.node.addComponent(CubismPosePart)! ;

// Set Pose information to CubismPosePart
posePart.groupIndex = groupIndex;
posePart.partIndex = partIndex;
posePart.link = group[partIndex].link;
```

Save Opacity of the part to determine which Pose parts to display.

The current CubismPart.opacity value is compared with the value cached in the previous frame to determine which parts to display in the group described in .pose3.json. The value is cached in CubismPoseController.savePartOpacities().

Pose Calculation and Application

Display parts are applied by AnimatinoClip.

The hidden parts are applied with CubismPoseController.doFade().

The Opacity of the hidden part is related to the opacity of the part to be displayed, so that the opacity is reduced to a level where the background is not transparent. CubismPoseController.doFade() looks for a visible part and then sets the Opacity of the hidden part.

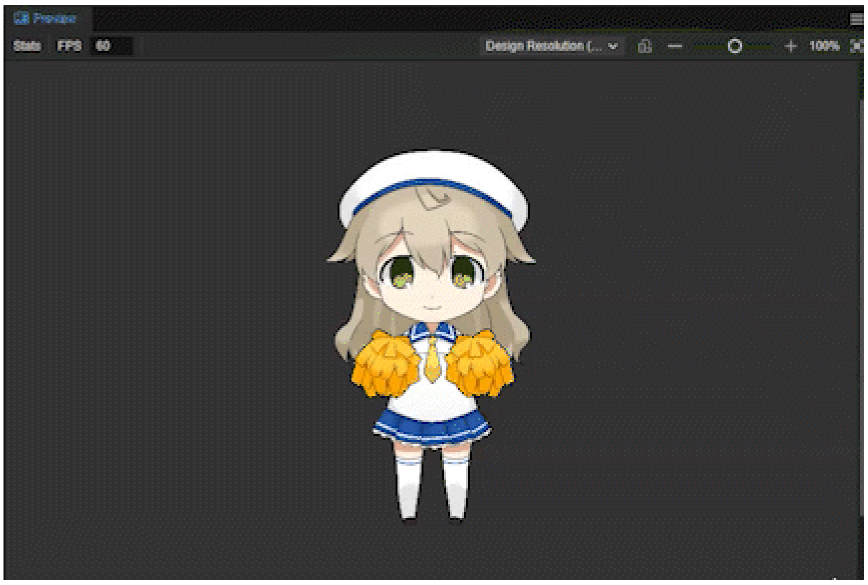
Interlocking opacity

Opacity is set in CubismPoseController.copyPartOpacities for interlocking parts written in CubismPosePart.link after Pose is applied.

Animation

Overview

This is a sample scene that plays an embedded animation file (.motion3.json) exported from Cubism Editor using Animation, a standard function of Cocos Creator.



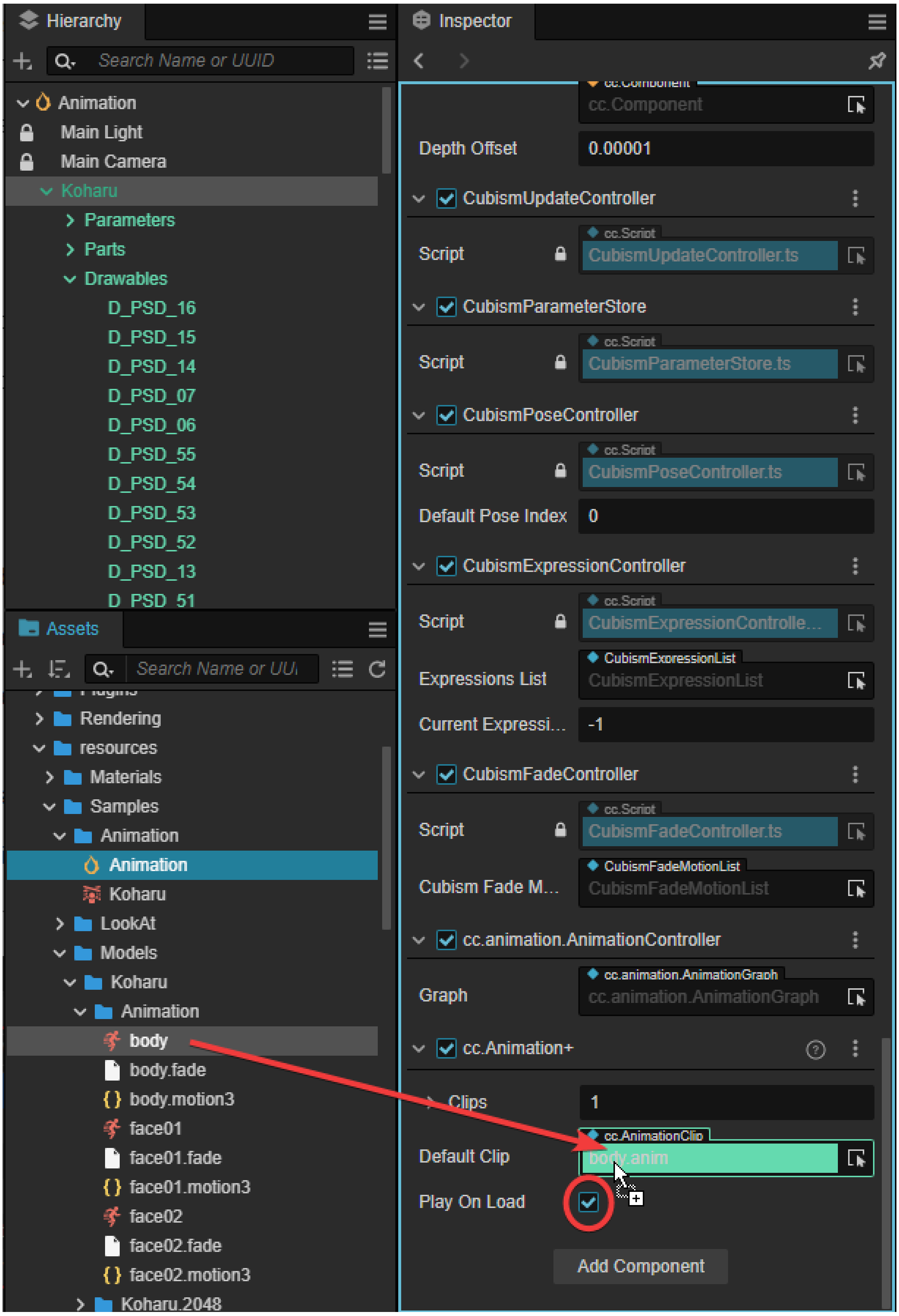
What functions can be checked

You can check how to make animations play using Animation, a standard feature of Cocos Creator.

For more information about Animator, please check here.

How to set up a scene

Attach Animation to the model and set the AnimationClip attached to the model. Then check the Play on load checkbox.



Depth Offset 0.00001
CubismUpdateController
Script CubismUpdateController.ts
CubismParameterStore
Script CubismParameterStore.ts
CubismPoseController
Script CubismPoseController.ts
Default Pose Index 0
CubismExpressionController
Script CubismExpressionControlle...
Expressions List CubismExpressionList
Current Expressi... -1
CubismFadeController
Script CubismFadeController.ts
Cubism Fade M... CubismFadeMotionList
cc.animation.AnimationController
Graph cc.animation.AnimationGraph
cc.Animation+
Clips 1
Default Clip body.anim
Play On Load ✓

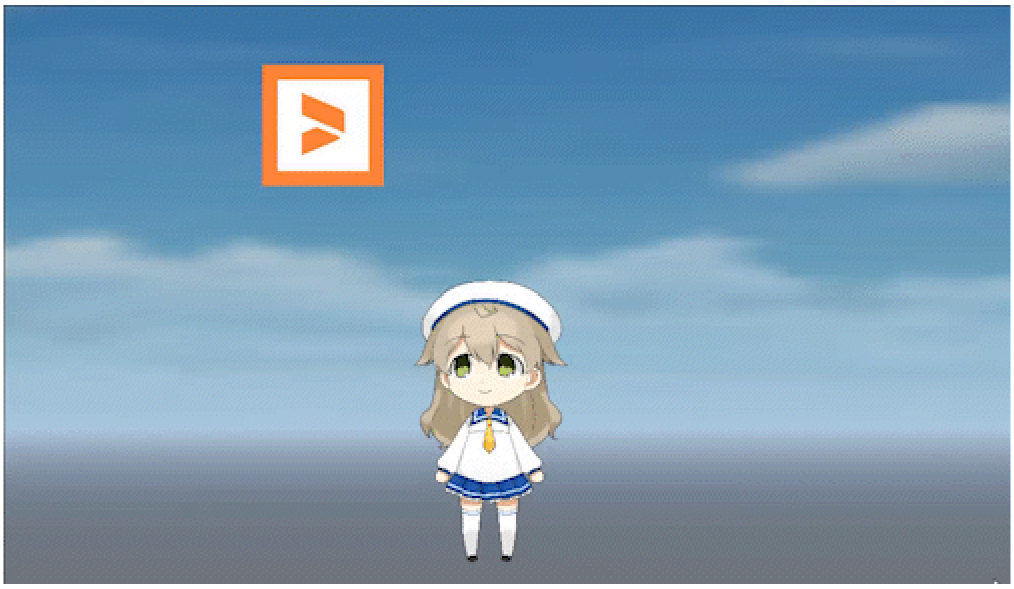
To use the created model in a sample scene

Follow the tutorial in Tutorial - Playback of Animation to complete a scene equivalent to this sample scene.

LookAt (sample)

Overview

This is a sample scene that allows you to check the functionality of eye tracking at specific coordinates on a game scene.



What functions can be checked

- Two functions for line-of-sight tracking can be checked.
  - CubismLookController (attached to Prefab of Cubism to be followed by eye tracking)
  - Class that implements ICubismLookTarget (attaches to the object to be followed by eye tracking)
    - ICubismLookTarget is used in the sample scene

Easy to use

When Run is performed, the Cubism model will continue to follow the line of sight of the target object moving left and right.

How to set up a scene

You can see that CubismLookTargetBehaviour is attached to TargetPivot -> Target on the Hierarchy, and the model with CubismLookController attached will follow the gaze of this object.

See Tutorial - Setting Up Eye Tracking for instructions on how to set up the model.

Notes specific to the Cocos Creator version

Notes on loading Cubism SDK for Cocos Creator into Cocos Creator

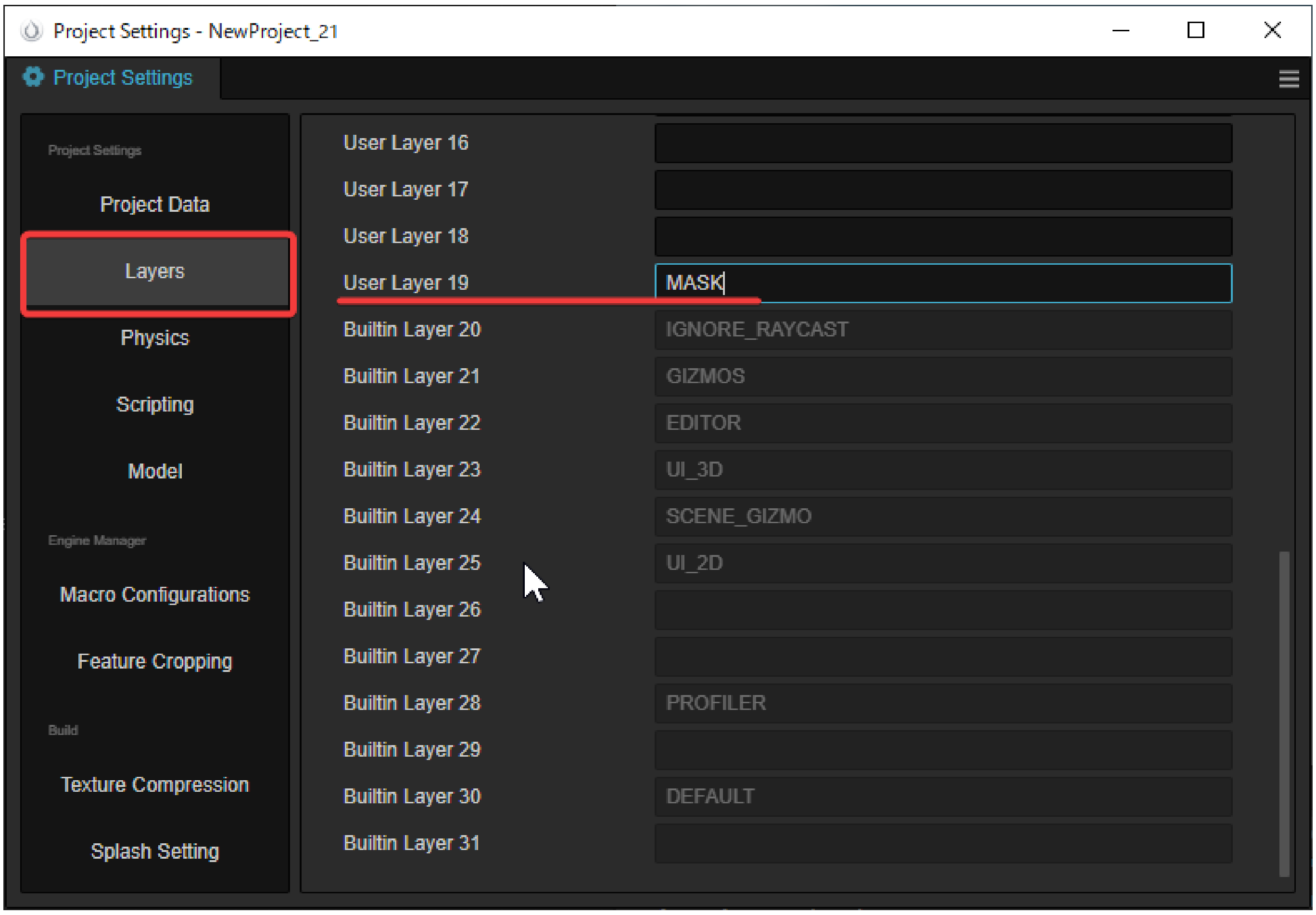
When using the sample models enclosed in Cubism SDK for Cocos Creator, you may not be able to use them as they are immediately after loading Cubism SDK for Cocos Creator because the import order is not fixed. If an error message is displayed, be sure to re-import model3.json or the folder that contains it by clicking [Reimport Asset]. In addition, you may not be able to use [GlobalMaskTexture.asset] correctly immediately after loading Cubism SDK for Cocos Creator into Cocos Creator. In this case, please make sure to re-import [GlobalMaskTexture.asset] by [Reimport Asset] and restart Cocos Creator.

About drawing masks

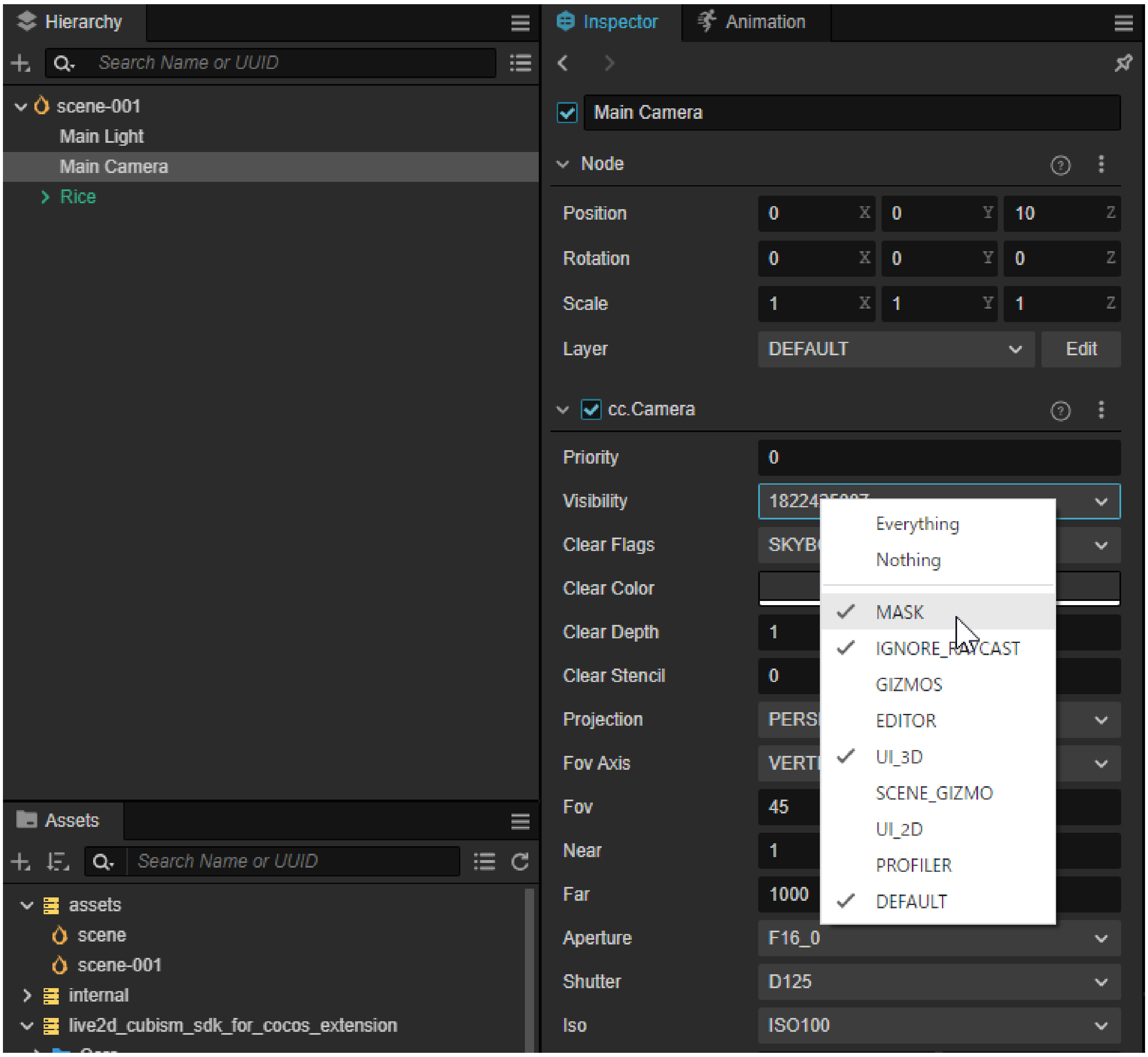
GlobalMaskTexture.asset

GlobalMaskTexture.asset] may not render masks correctly immediately after loading Cubism SDK for Cocos Creator into Cocos Creator. In this case, please make sure to re-import [GlobalMaskTexture.asset] by [Reimport Asset] and restart Cocos Creator.

In some cases, the texture used in the mask may be drawn on the entire scene. In this case, please add any name (such as MASK) to `User Layer 19` in [Project]-[Project Settings]-[Layers] of Cocos Creator, and uncheck `Visibility` in the Main Camera of the scene. Visibility` in the scene's Main Camera.



User Layer 16	
User Layer 17	
User Layer 18	
User Layer 19	MASK
Builtin Layer 20	IGNORE_RAYCAST
Builtin Layer 21	GIZMOS
Builtin Layer 22	EDITOR
Builtin Layer 23	UI_3D
Builtin Layer 24	SCENE_GIZMO
Builtin Layer 25	UI_2D
Builtin Layer 26	
Builtin Layer 27	
Builtin Layer 28	PROFILER
Builtin Layer 29	
Builtin Layer 30	DEFAULT
Builtin Layer 31	



Main Camera

Node

Position	0 X	0 Y	10 Z
Rotation	0 X	0 Y	0 Z
Scale	1 X	1 Y	1 Z
Layer	DEFAULT		Edit

cc.Camera

Priority	0
Visibility	18224
Clear Flags	SKYB
Clear Color	
Clear Depth	1
Clear Stencil	0
Projection	PERS
Fov Axis	VERT
Fov	45
Near	1
Far	1000
Aperture	F16_0
Shutter	D125
Iso	ISO100

Everything
Nothing
✓ MASK
✓ IGNORE_RAYCAST
GIZMOS
EDITOR
✓ UI_3D
SCENE_GIZMO
UI_2D
PROFILER
✓ DEFAULT

TypeScript (JavaScript) language specification notes

TypeScript (JavaScript) language specification notes Classes such as math.Vec2 and math.Vec3 provided by Cocos Creator and the Vec2 and math.Vec3 provided by Cocos Creator, and IStructLike interface implementation classes provided by this SDK are instances according to the language specification.

Note that changing the value of an instance member property so as to treat the structure like a language such as C# may cause unexpected behavior.

The IStructLike interface implementation class provided by this SDK has its members set to "readonly" so that the above operation can be avoided by mistake in basic operations.

Note that the following cases are not errors due to language specification reasons as of Typescript 4.7.4.

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

const a: ISample = new Sample(); // no error.
a.x = 1;

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