# Chapter 1: Meet Cocos Creator

*Mobile game market has grown rapidly in the past few years, as has the cross-platform game development engine Cocos2D-X (founded in 2014, now Cocos). With a market share of 45% in China, and 18% market share globally as #2 at the mobile engine area.*

*There are some famous games that maybe you know are based on Cocos2D-X such as **"Dota Legend"** (Dota truyền kỳ) or **"Onmyoji"** (Âm Dương Sư). In 2013, Cocos Team released the first official editor named "Cocos Studio", however it has not been widely accepted and used by developers due to its own bugs and the lack of an efficient process. Many studios are self-developing related editors and development tools, leading to fragmentation in community and difficulty in sharing knowledge and experience.*

*Until 2016, the engine team released the Cocos Creator. Cocos Creator has the characteristics of scripting, data-driven and efficient workflow. It has strong ease of use and can provide an efficient development process. This course will lead you into the world of Cocos Creator.*

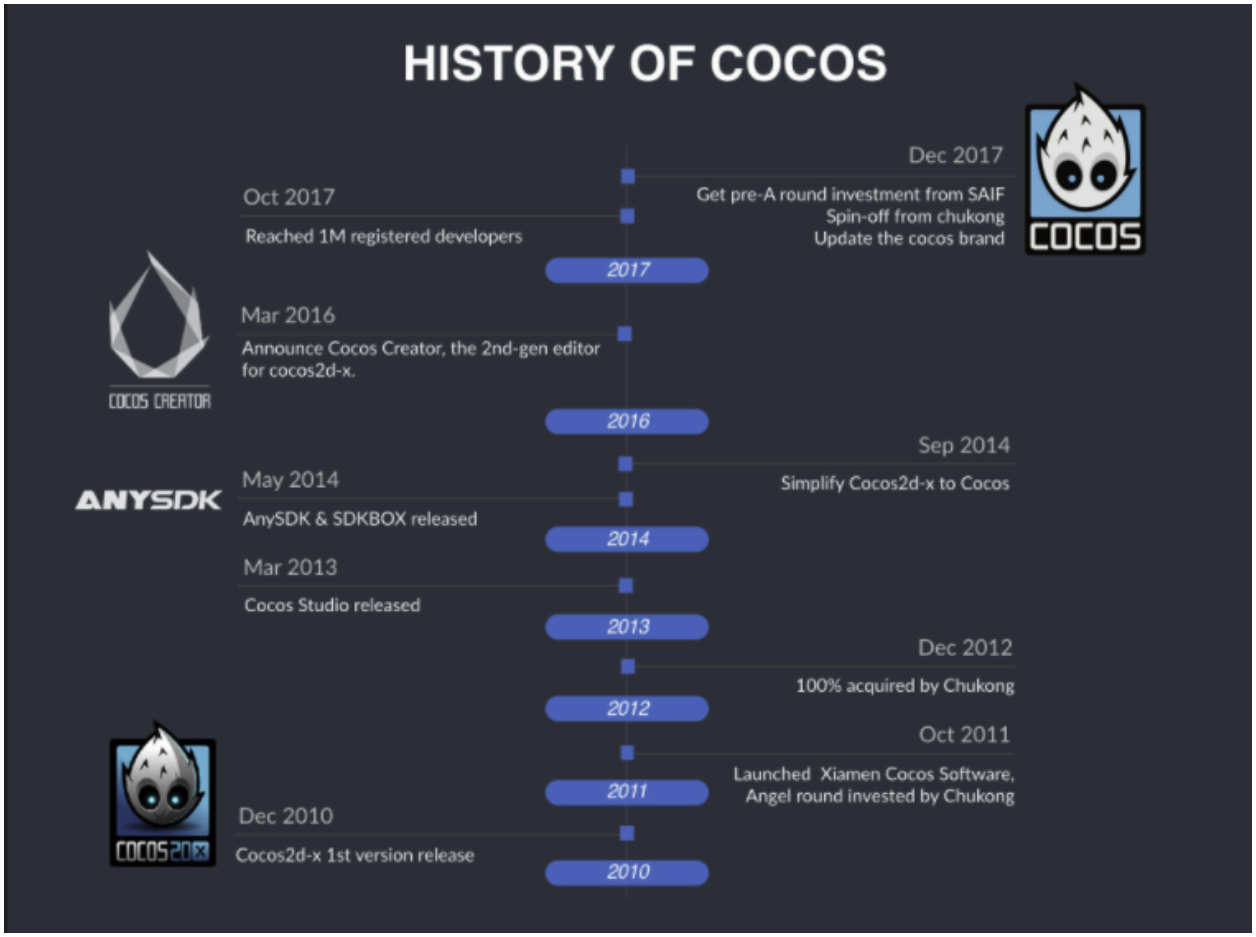



**Figure 1-1 The development history of Cocos engine**

Game development is a comprehensive technology stack. The development of games requires learning programming languages, platform knowledge, editor usage, algorithms, design patterns and other knowledge. In order to use Cocos Creator, we need to learn the following:

1) **Programming language JavaScript**: This is a scripting language supported by Cocos Creator. You can use JavaScript to write game logic and also extensions for the editor. The JavaScript in Cocos Creator is based on the basic syntax of the scripting language, and some unique rules as well, you need to learn basic grammar and local rules at the same time.

2) **The basic use of the editor**: To develop a game, you need not only write code, but also effectively organize resources such as pictures, music and sound effects, fonts, particles, maps, etc. Cocos Creator provides a complete solution for resource import and asset management . You need to learn resource management methods while being familiar with these resources.

3) **The basic system of the game**: The system of the game contains three major modules, UI system, animation system and physics system. These are the basis of game development.

# I. What is Cocos Creator

Cocos Creator is a content creation-focused, scripted, component-based and data-driven game development tool. It features an easy-to-use content production workflow and a powerful developer tool suite for implementing game logic and high-performance game effects. In keeping with Cocos's usual product features like open-source, easy-to-use, high-performance and cross-platform, this new game engine is designed to be the new choice for developers to create 2D and 3D games.

# II. The composition of Cocos Creator

By browsing the contents of the Cocos Creator program folder, you can find that it mainly includes three parts: editor, Cocos2D-X and Cocos2D-JS.

1) **Editor**: Similar to the previous Cocos Studio and CocosBuilder, the editor contains functions such as scene editing, UI editing and animation editing.  If you have tried Unity Editor, you would say it's more like a copy of Unity 3D editor.

2) **Cocos2D-X**: Core heart of Cocos ecosystem, Cocos2D-X game engine, written in C++. Cocos Creator uses a "lite" version of Cocos2D-X Engine, which contains fewer features, and fewer platforms are supported.

3) **Cocos2D-JS**: Cocos Creator was mainly based on JavaScript at the beginning, and Cocos Creator was written entirely in JavaScript, which shows the importance of JavaScript in Cocos Creator. Cocos2D-JS has also been inherited by Cocos Creator. Like the integrated Cocos2D-X, Cocos2D-js is also a reduced version.

Cocos Creator adopts the ECS (Entity Component System) design mode. **Entity–component–system** (**ECS**) is an architectural pattern that is mostly used in game development. ECS follows the composition over inheritance principle that allows greater flexibility in defining entities where every object in a game's scene is an entity (e.g. enemies, bullets, vehicles, etc.). Every entity consists of one or more components which contain data or state. Therefore, the behavior of an entity can be changed at runtime by systems that add, remove or mutate components. This eliminates the ambiguity problems of deep and wide inheritance hierarchies that are difficult to understand, maintain and extend. Common ECS approaches are highly compatible and often combined with data-oriented design techniques.

# Chapter 2: Get to know the Cocos Creator editor

**Cocos2D-X** engine is a 2D game engine based on **OpenGL ES**. Its main concept is to abstract all parts of the game into objects: director, scene, layer, etc.. and then use these objects to encapsulate the OpenGL rendering, which is easier for developers to call. Cocos Creator has carried out another encapsulation on the basis of Cocos2D-X, using the "entity-component" method.

This chapter mainly introduces the Cocos Creator Editor, nodes and components in Cocos Creator, the coordinate system in Cocos Creator, and finally guides you to make your first Cocos Creator Scene.

## I. Basic interface of Cocos Creator Editor

Cocos Creator is very similar to the Unity engine, in which the core is the editor. The editor contains some main modules:

- Resource Management (Assets)
- Node Tree
- Scene Preview
- Animation editor
- Console
- Node Library

In this section, we will first understand the editor from the Cocos Creator interface, as shown in Figure 2-1.

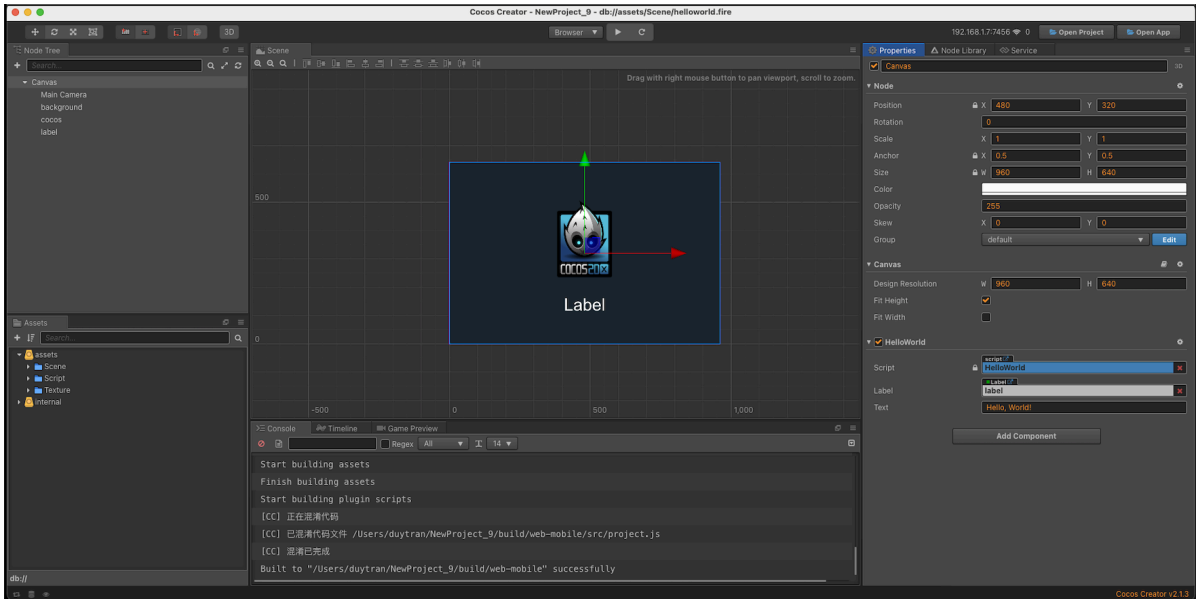



**Figure 2-1 Cocos Creator running interface**

Generally, the entire interface can be divided into five parts, the middle part is the scene editing and preview part, the upper left part is the NodeTree, the lower left part is the resource manager , the right part is the property inspector and Node Library, and the lower part is the running console and the animation editor. These panels in Cocos Creator can be moved and combined freely to meet the needs of different projects and developers.

The resource manager displays all the resources in the project resource folder, and displays the files in a tree structure. It can automatically synchronize the modification of the project resource folder in the operating system. The user can directly drag the files in from the outside of the project.
The scene editor is used to display and edit the displayed content in the scene, and the content of the edited scene is WYSIWYG. The node tree uses the attribute structure to display the nodes in the scene and displays their hierarchical relationship. The corresponding nodes can be found in the scene editor.

The property inspector is a working area for viewing and editing the properties of the currently selected node and component. This panel will display and edit the property data defined by the script in the most suitable form.

# II. Nodes and Components

In Cocos2D-X, the smallest rendering unit is the node class (Node). Cocos Creator also retains this design: All objects drawn on the screen are node classes or subclasses of node classes.
The new basic concept is the component, which is a change of Cocos Creator compared to the previous version of Cocos2D-X. This section will introduce these two concepts separately. The important basic nodes in Cocos Creator are shown in Figure 2-2.

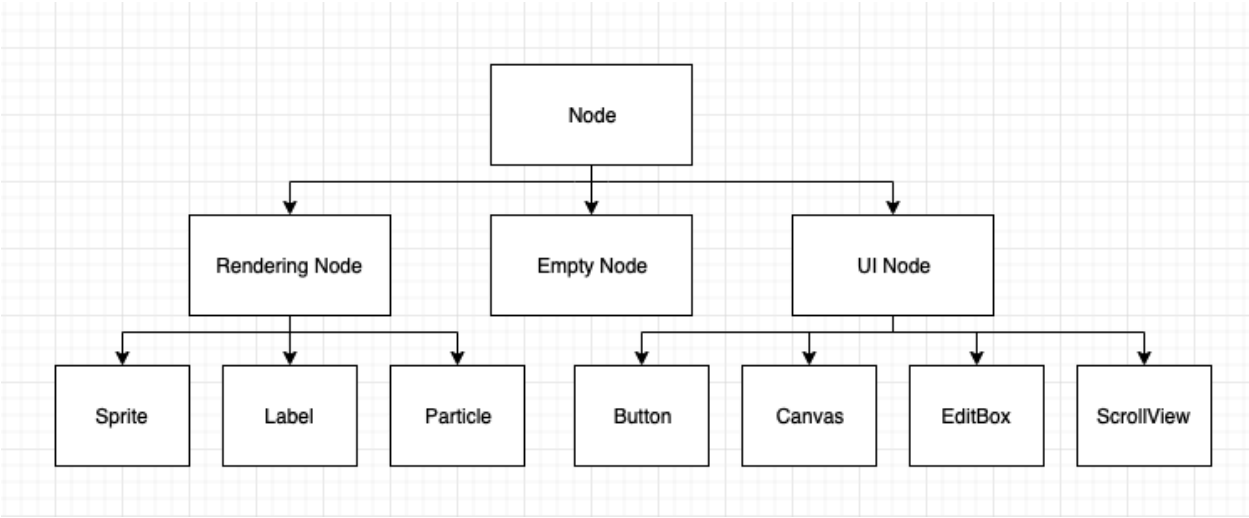



**Figure 2-2 Nodes in Cocos Creator**

- **Nodes in Cocos Creator**

In Cocos Creator, nodes are divided into empty nodes, rendering nodes and UI nodes.
The empty node is the same as the node in Cocos2D-X. It only contains information such as position and size, and does not contain any rendered elements. It will not be displayed on the screen, but it is useful. When we want to make some UI interface objects of composite picture splicing, we can treat the scattered elements as the child nodes of the empty node, so that the child nodes can move together with the parent node.

Render nodes include sprites, text, rich text, tiled maps, particle systems, etc., all of which can be rendered on the interface. (https://docs.cocos.com/creator/2.1/manual/en/render/)

UI nodes include the contents of UI components such as buttons, scrollViews, edit box, layouts, and canvases. (https://docs.cocos.com/creator/2.1/manual/en/ui/)

- **Components in Cocos Creator**

Compared to Cocos2D-X, an important change in Cocos Creator is the use of the "entity-component" model. From the perspective of design patterns, this is the use of composition instead of inheritance. What is the "entity-component" model? Generally speaking, games are programmed using an object-oriented approach. Each entity in the game corresponds to an object, and a class-based instantiation system is required to allow expansion through polymorphism. However, this approach cannot control the number of classes in the game. In order to solve this problem, we use the "entity-component" model to replace inheritance with composition. This method is used in Cocos Creator. The specific properties of the component are shown in Figure 2-3.

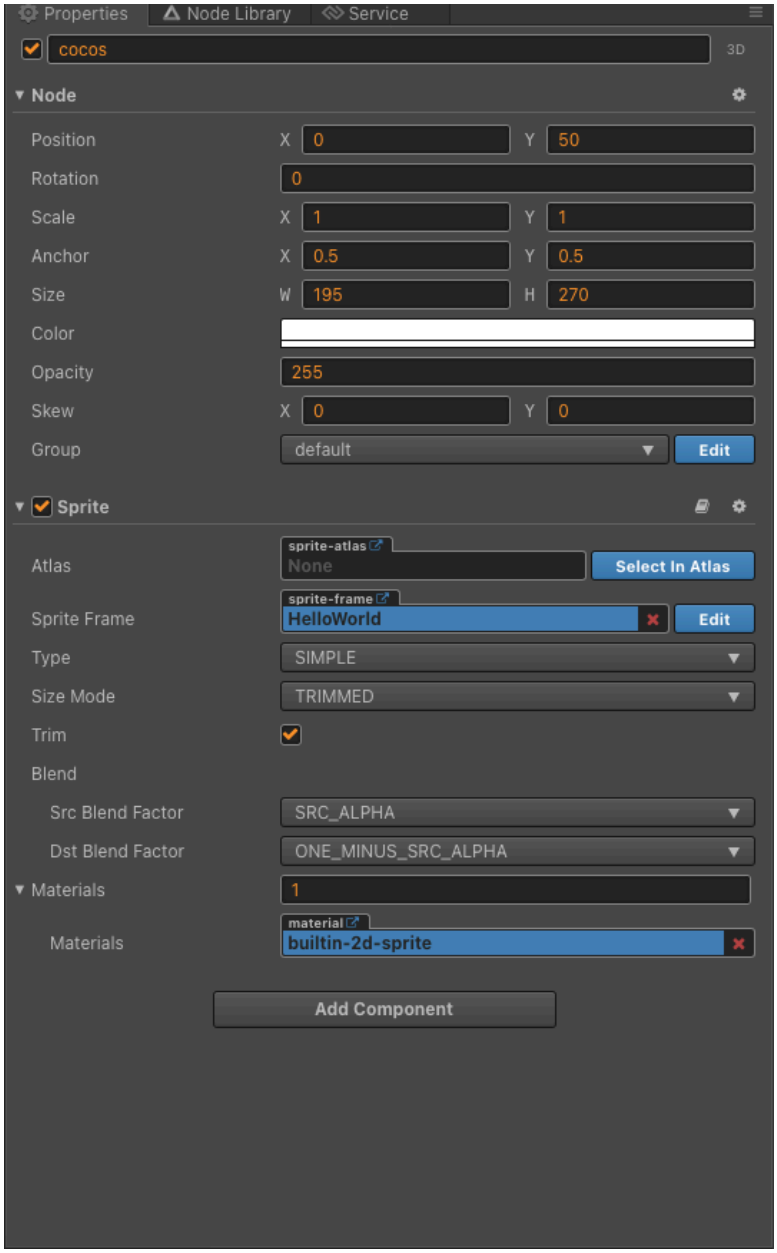



**Figure 2-3: Properties of Component**

The component is in the node's property inspector, as shown in the figure. In the sprite's property inspector, there are node attributes and sprite components. The node attributes include information such as node position, rotation, scaling and size, as well as anchor points , colors and opacity. After the node component and the sprite component are combined, the display mode of the picture can be controlled by modifying the node properties.

You can add multiple components to a node to add more functions to the node. Click the Add Component button in the property inspector to add a component. This component can be a built-in component or a script component you write. As shown in Figure 3-4.

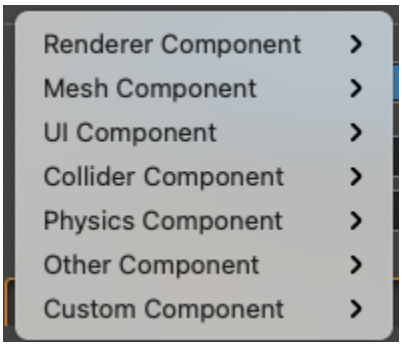



**Figure 2-4: Add Component**

# III. Coordinate system in Cocos Creator

Nodes need to be rendered to be displayed on the screen. The render process needs to locate the position, and the Coordinate System helps us locate the position on the screen where we want to draw it. In order to put  nodes exactly where we want, we need to understand coordinate systems. Different coordinate systems mean different positioning rules, and this section will introduce four systems that are commonly used in Cocos, and anchorpoint of nodes.

- **Cartesian coordinate system:**

Cocos2D-X is based on OpenGL ES, so naturally, its coordinate system is the same as the OpenGL coordinate system. The origin of the coordinate system is in the lower left corner of the screen. It adopts the right-hand rule. The positive x-axis direction goes to the right and the positive y-axis direction goes up. , The z axis is outward, as shown in Figure 2-5.


**Figure 2-5: Cartesian coordinate system**

### Standard screen coordinate system

The screen coordinate system is opposite to the OpenGL coordinate system. The origin of the coordinate system is at the upper left corner of the screen. Android, iOS and Windows Phone all use the standard screen coordinate system, which needs to be distinguished from the coordinate system of Cocos Creator, as shown in Figure 2- 6.

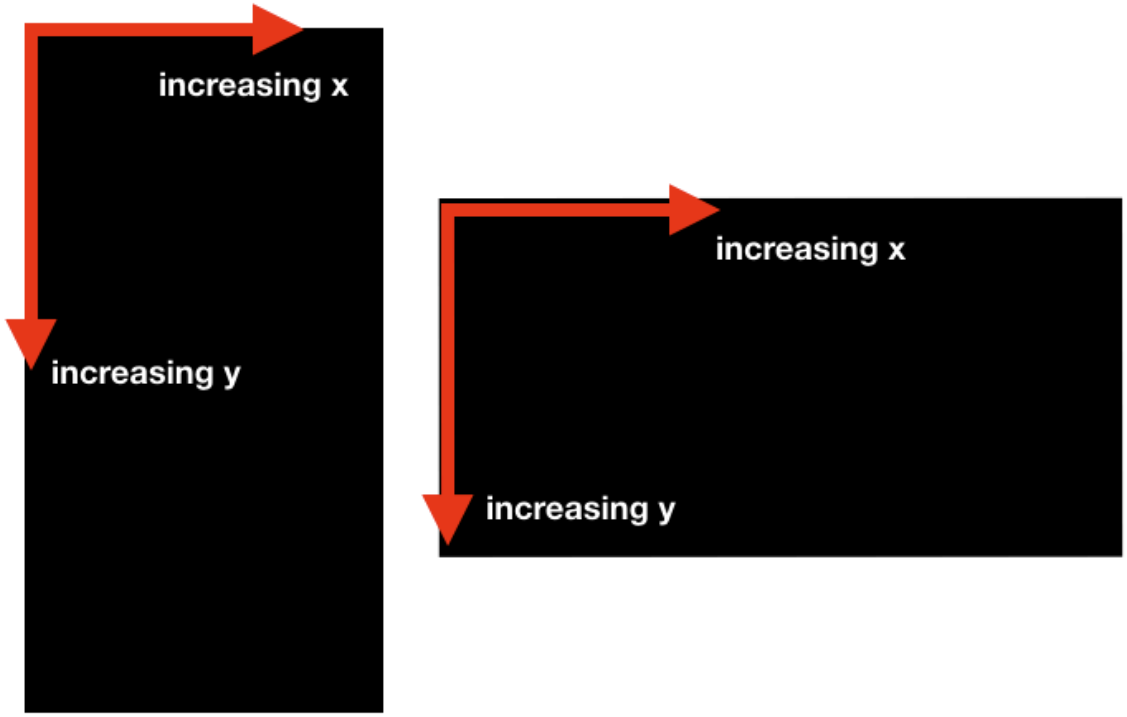



**Figure 2-6: Standard screen coordinate system**

- **World coordinate system**

The world coordinate system is also called the absolute coordinate system, which is a concept established in game development. Therefore, the "world" is the game world. It establishes the reference standards needed to describe other coordinate systems. We can use the world coordinate system to describe the position of other coordinate systems, which is a concept of global coordinates in Cocos Creator.

The elements in Cocos have a hierarchical structure with a parent-child relationship. The position set by the node uses the local coordinate system relative to its parent node instead of the world coordinate system. Finally, when drawing the screen, the rendering part of Cocos will map the local node coordinates of these elements to the world coordinate system coordinates. The world coordinate system is in the same direction as the OpenGL coordinate system, the

origin is in the lower left corner of the screen, the positive x-axis direction is to the right, and the positive y-axis is upward.

- **Local coordinate system**

The local coordinate system is also called the relative coordinate system, which is a coordinate system associated with a specific node. Each node has its own coordinate system. When the node moves or changes direction, the coordinate system (its child nodes) associated with the node will move or change direction accordingly. The local coordinate system is meaningful only in the Node coordinate system.

The method of setting the position of the node uses the node coordinate system of the parent node, which is in the same direction as the Cartesian coordinate system. The positive direction of the x-axis is to the right, the positive direction of the y-axis is upward, and the origin is at the lower left corner of the parent node. If the parent node is the top node in the scene tree, then the node coordinate system it uses coincides with the world coordinate system.

- **Anchor point of node**

Let's move to another important concept of positioning coordinates: anchor point. To make it simple, Anchor point is where the origin point of the local coordinate system is. It also affects some transformations such as scaling and rotating of the element.
Default value of Anchor point is (0.5, 0.5), which means it is not really a pixel point, but a multiplier factor, as shown in Figure 2-7.

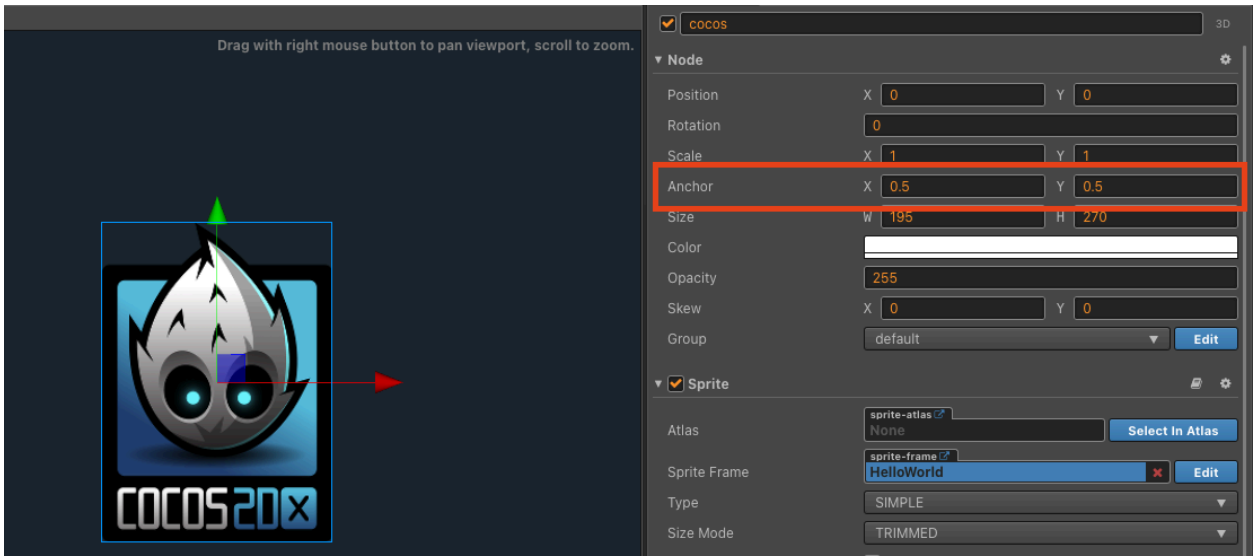



**Figure 2-7: Anchor point of a Node with Sprite**

# IV. Create your first Cocos Creator Project

After learning here, I believe you are already eager to try. After all, the purpose of learning is to make your own projects. In this section, let's edit and make your first Cocos Creator scene!

The process of creating a project is very simple. Click New Project, select the appropriate project template, enter the project directory below, and click New Project to create the project, as shown in Figure 2-8.

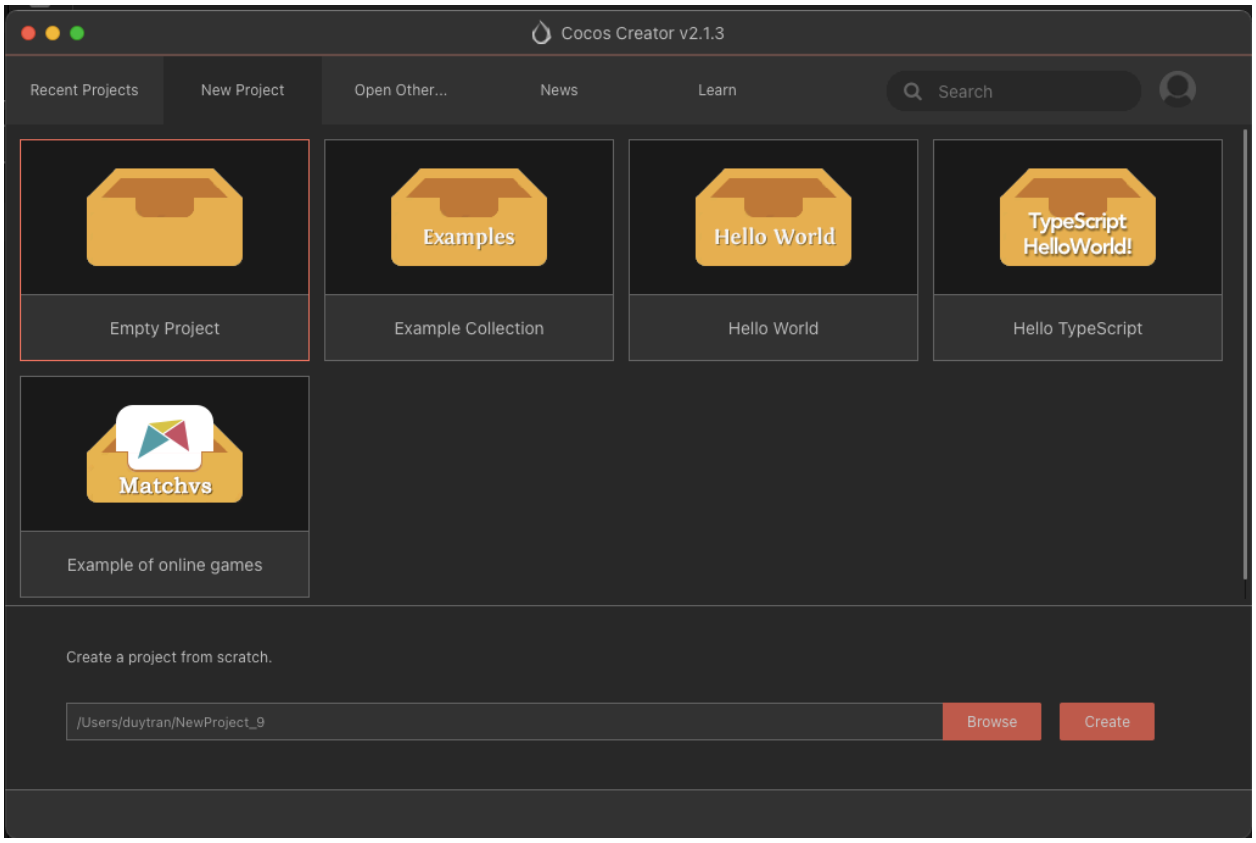



**Figure 2-8 Create project**

Here we choose the Hello World template, which contains the familiar Hello World scene. After successfully creating the project, go directly to the main interface of Cocos Creator, right-click in the resource management panel and select "New-Scene" to create a new scene. The default scene is empty, and there is only one default Canvas node.

- **Project structure**

The structure of the default HelloWorld project is shown in Figure 2-9.

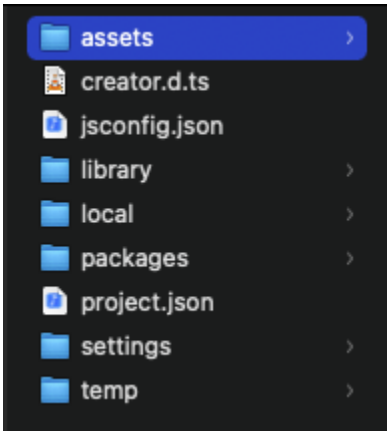



**Figure 2-9 Default Project structure**

**Assets folder** (assets): used to place all local resources, scripts and third-party library files in the game. Only the files under the resource folder can be displayed in the resource manager, open the folder, you can find that each resource file has a .meta file, which is used to store the information after the file is imported as a resource and the association with other information.

**Resource library** (library): It is generated after the resource is imported. The file structure and resource format here are processed into the form required for publishing. If you use version control to manage the project, this folder does not need to enter version control. If it is deleted or damaged, just delete it completely and open the project to regenerate it.

**Local settings** (local): The local settings of the editor are kept here, including the editor layout, etc. This folder does not need to enter version control.

**Project settings** (settings): Save project-related settings, such as the package name, scene, and platform selection in the build release menu.

**project.json**: The project.json itself is currently used to specify the currently used engine type and the storage location of the plug-in. Together with the asset, it serves as a sign for verifying the legitimacy of the project.

**Build target** (build): This folder will only be generated after the first build and release. After the project is released using the default release path, the project project of the target platform will be generated under build.

After the project is created, select the corresponding scene to run the preview scene, click the run button in the editor to run, and you can choose to run in the browser or simulator mode. The running effect of the Hello World scene is shown in Figure 2-10.

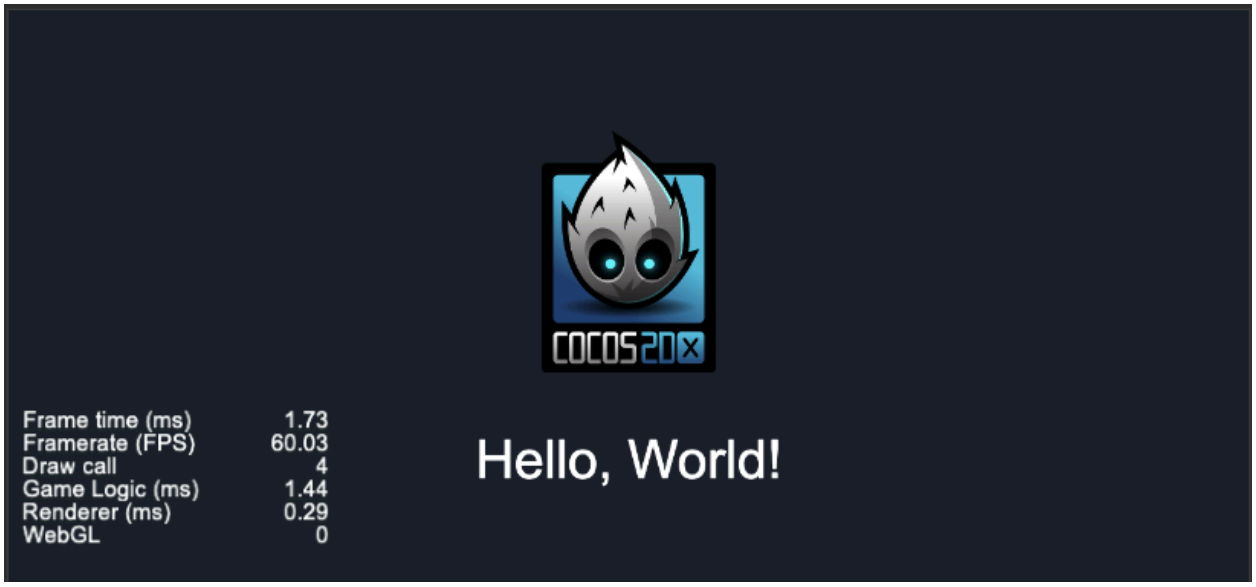



**Figure 2-10 HelloWorld**

## Summary of this chapter

This chapter introduces the basic usage of Cocos Creator, introduces the interface of Cocos Creator, and introduces the functions responsible for different windows of Cocos Creator. You can find that Cocos Creator is an editor that integrates editing, previewing, debugging, and exporting functions. It also introduces the two basic concepts of node and component in Cocos Creator. Node is the smallest unit of rendering, and component is a brand new concept, which makes the method of extending nodes in Cocos Creator change from inheritance to extension. This is also the most important concept in CocosCreator, then introduced the different coordinate systems in Cocos Creator, and finally completed the creation and editing of the first scene.

Through the study of this chapter, we are already familiar with the operation and use of the editor, and lay a good foundation for further project production in the future. To make a game is to combine resources and code.

# Chapter 3: Asset workflow in Cocos Creator

## I. Adding Assets

There are three ways to add assets into a project:

- Using the create button to add assets as in Figure 3-1:

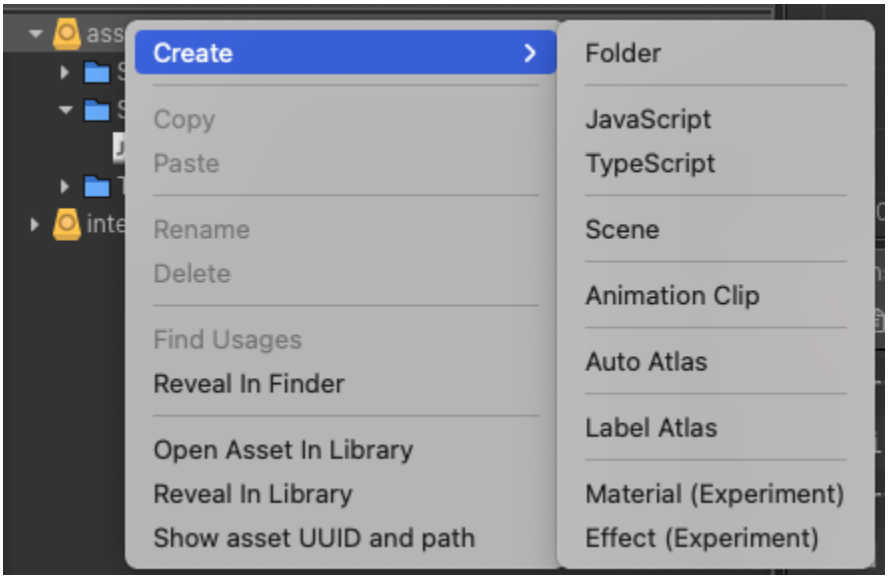



**Figure 3-1: Add Asset**

- In the operating system file manager, copy the asset file to the project assets folder and then reopen or activate the Cocos Creator window to finish importing the asset.
- Drag asset files from the operating system file manager (such as Explorer in Windows or Finder in Mac OS) to the assets panel to import the asset.

## II. Manage image resources

### 1. Import image resource

Image resources, also known as textures, are the rendering source for most objects in games, especially 2D games. Image resources are generally produced and provided by Art-Design, and output into a format by the game engine. Cocos Creator not only supports separated images, but also atlas resources and can generate atlas resources.

- Cocos Creator supports images in 2 formats: **PNG** and **JPG.**
- You can import images by any of 3 methods in section <**I**>. After being imported, the texture will appear like this in the **Assets Panel**. Once you select a texture in the **Assets Panel**, the thumbnail will also be shown at the bottom of the **Properties Panel**. Please **DO NOT** modify the properties of texture in the **Properties Panel**, unless you know what you are doing.
- As you can see in Figure 3-2, after expanding the image by clicking on the triangle, you can see the **SpriteFrame** with the same name.

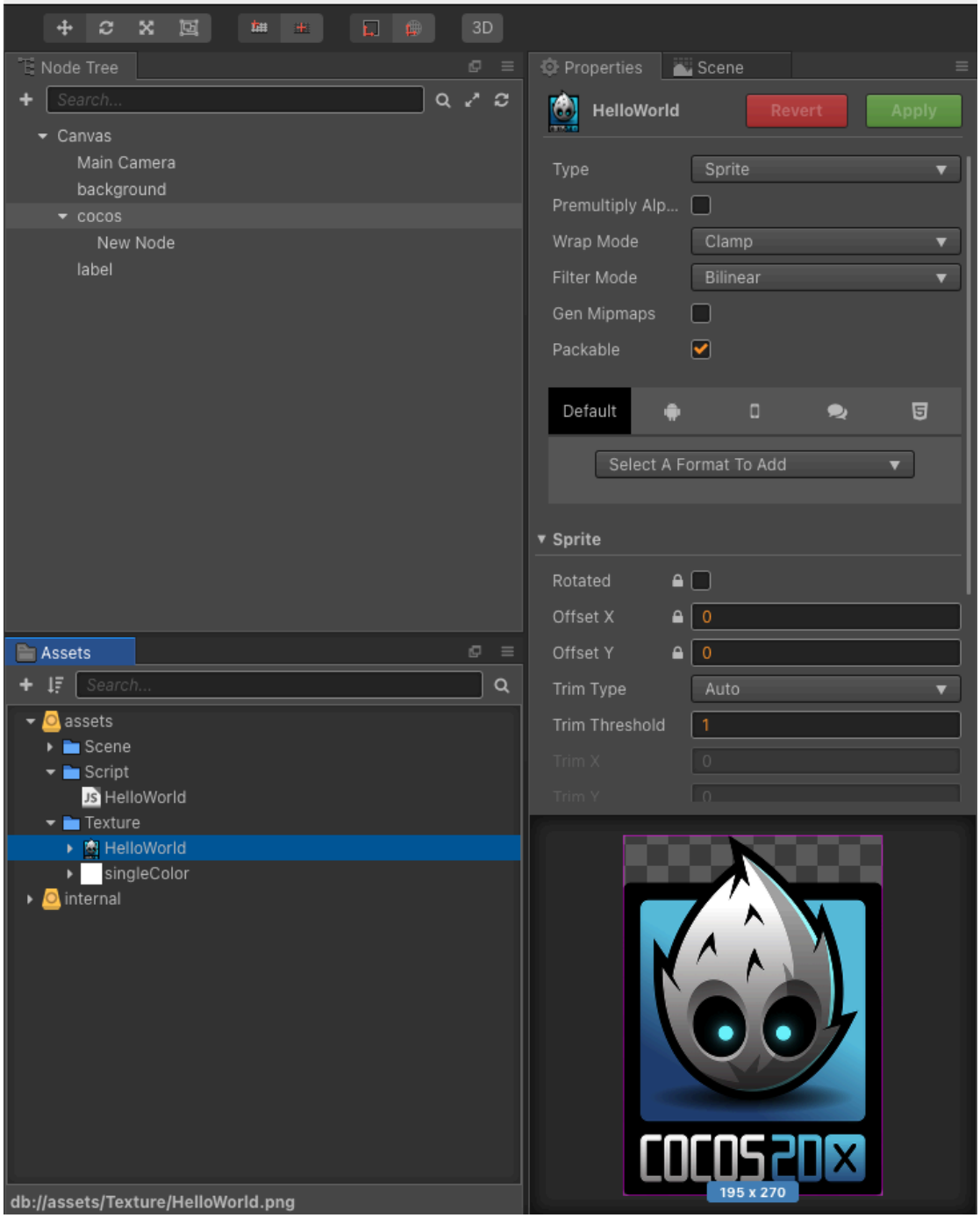



**Figure 3-2: Sprite Properties**

- **HOMEWORK:** make a small research about **Wrap mode, Filter Mode** and **Premultiply Alpha.**

## 2. Create and import atlas resource

- Atlas is also named Sprite Sheet, it's a common asset in game development. Atlas is generated by tools. The tools will combine many images into a large image and create an index file (such as **plist**). Atlas asset contains **plist** & **png** supported by Cocos Creator.

**Figure 3-3: Demo Sprite Atlas**

- Creating Atlas using Shoebox:
  - Link download Adobe Air: https://airsdk.harman.com/runtime
  - Link download Shoebox: https://renderhjs.net/shoebox/
- In Cocos Creator Editor, exported Atlas will appear like in Figure 3-4. A big png file for holding all small fragments, and a plist that can be expanded for getting sprite frames.

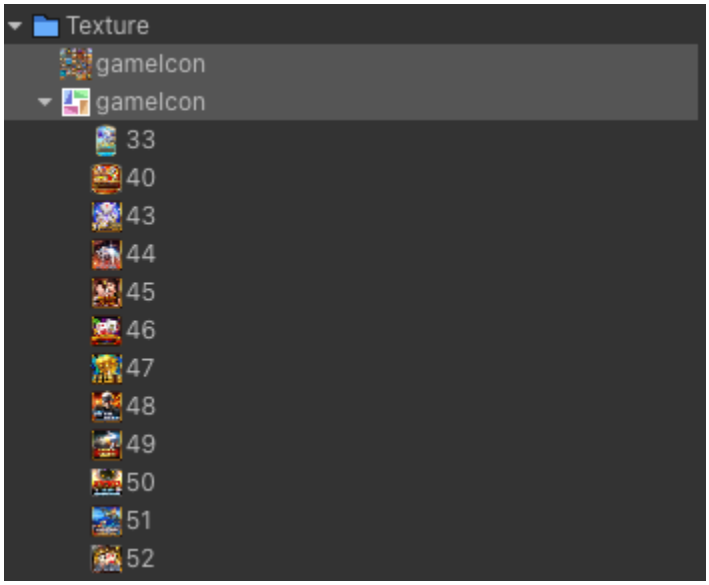



**Figure 3-4: Content of a Sprite Atlas**

- **HOMEWORK:** make a small research about Auto Atlas.

# III. Manage audio resource

In the **Assets menu**, select an audio, the **Properties** will have an option of load mode. This option is only valid for the web platform.

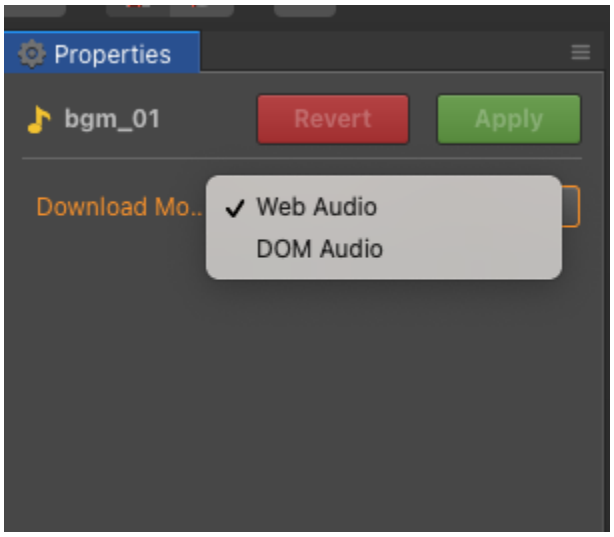



**Figure 3-5 Audio Asset Properties - Download Mode**

### 1. Loading audio in WebAudio mode

Loading audio resources with Web Audio, the audio resources will be cached in a buffer of the engine.
The advantage of this approach is good compatibility and robustness. However the disadvantage is that too much memory will be occupied.

### 2. Loading audio in DOMAudio mode

By generating a standard element to play the sound resources, the cache is the audio element.
When using standard audio elements to play sound resources, you may encounter some restrictions on some browsers. For example, each play must be played within the user action event (Web Audio only requires the first time), allowing only one sound resource to be played.
For larger Audio such as background music, DOM Audio is recommended.

# IV. Creating and using Prefabs

1. **What is a prefab**
   - Cocos Creator's **Prefab** allows you to create, configure, and store a **Node** complete with all its components, properties, and child nodes as a reusable Asset. The Prefab acts as a template from which you can instantiate a new clone in Scene.
   - When you want to reuse a Node configured in a particular way - like monsters, bullets, etc.. in multiple places in your Scene, or across Scenes in your Project, you should convert it to a Prefab. This is better than just copy and paste Node, because Prefab allows you to automatically keep all copies in sync.
   - Many people also use prefab as a solution for multi-people collaboration
   - You can choose Auto Sync or Manual Sync for every prefab instance in the scene.
2. **Create and use of prefab**
   - Creating a Prefab is super easy by dragging a Node from the **Node Tree** Panel to the **Assets Panel**.

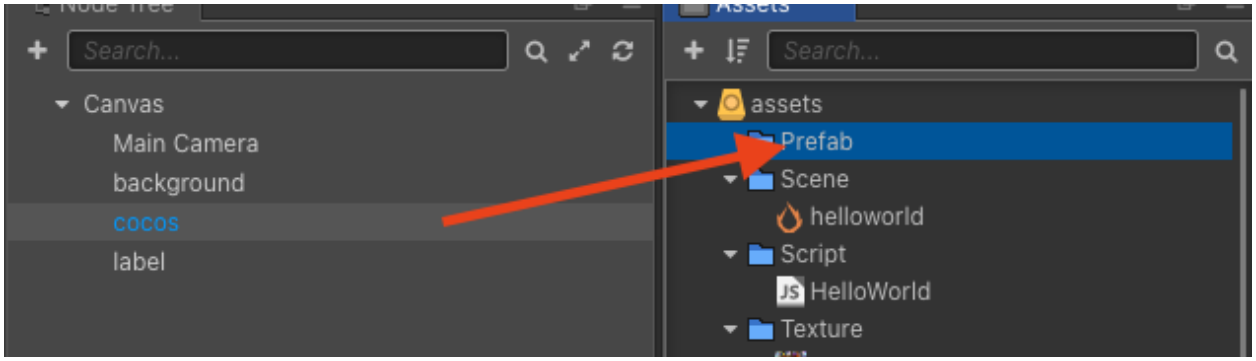



**Figure 3-6 Create Prefab**

   - To modify a prefab, simply double click it from the **Asset** panel. After selecting, you can see the main settings of the **Prefab** in the **Properties** Inspector as shown in Figure 3-7. Prefab has 3 main settings that you should keep in mind: "**Sync Mode**", "**Go Back**" and "**Save**". "**Sync Mode**" includes **Manual Sync** and **Auto Sync** (which will be discussed in IV-3). Clicking on "Save" or "GoBack" to save the instance attribute to the prefab attribute or restore the prefab attribute to the instance.


**Figure 3-7: Prefab Properties**

- In Editor, you can clone a prefab by simply dragging it to Scene or Node Tree
- In some cases, you will need to create and use prefabs by Code. We will talk about this in Chapter 4 (*ref: https://docs.cocos.com/creator/2.1/api/en/classes/Prefab.html*)

### 3. Auto sync and Manual Sync

- You can choose Auto Sync or Manual Sync for every prefab instance in the scene.
- When set to **Manual Sync**, when its originating asset changes, the prefab instances will NOT dynamically refresh to stay synchronized with the originating asset. Refreshing is only triggered when users manually revert the prefab.
- If set to **Auto Sync**, the prefab instances will dynamically refresh to stay synchronized with the originating asset.
- Some properties of **Prefab** will not be automatically synchronized: “Name”, “Active”, “Position” and “Rotation”, to facilitate individual customization of each scene instance. Other child nodes and all components will be kept in sync with the original resource. If a modification occurs, the Editor will ask whether to undo the modification or update the original instance. The components in the **Auto Sync** prefab cannot refer to other objects outside the Prefab, otherwise the Editor will pop up a prompt. Components outside the **Auto Sync** Prefab can only refer to the root node of the Prefab, and cannot refer to components and child nodes, otherwise the Editor will pop up a prompt.

# V. Font resources

Besides Texture Resources, Text also plays an important role in conducting a Game: Chat system, Tutorial, NPC guide etc…

Cocos Creator provides a flexible mechanism for Text rendering, Producer can either use System Font or self-rendering fonts. Cocos Creator supports 3 types of Font assets: **System Font**, **Dynamic Font** and **Bitmap Font**. **System fonts** call the system fonts of the platform to render text. You don't need to use any resources, just set the "UseSystemFont" property.

The only file type that is supported in **Dynamic Font** is **TTF.**
In this section, we will talk mainly about Bitmap fonts.

## 1. Bitmap fonts - introduce and create bitmap font

Like **Atlas**, Bitmap font Asset has two parts: **.fnt** font file and a **.png** file. You can think of `.png` as an image map and `.fnt` is the index to each character in the map.This file combo can be generated with specific tools, it is commonly made by **Shoebox** or **Glyph Designer.**

Advantages of bitmap fonts include:

1. Extremely fast and simple to render
2. Easier to create than other kinds.
3. Unscaled bitmap fonts always give exactly the same output when displayed on the same specification display
4. Best for very low-quality or small-size displays where the font needs to be fine-tuned to display clearly

## 2. Use bitmap font

Fonts generally need **Label** components to render. To create a **Node with Label**, In **Node Tree** click the top left **+** (Create Node) button and select **Create Renderer Nodes -> Node With Label**, this will create a node with the **Label** component attached in the scene.
You can also create from the main menu **Node -> Create Renderer Nodes -> Node With Label**.

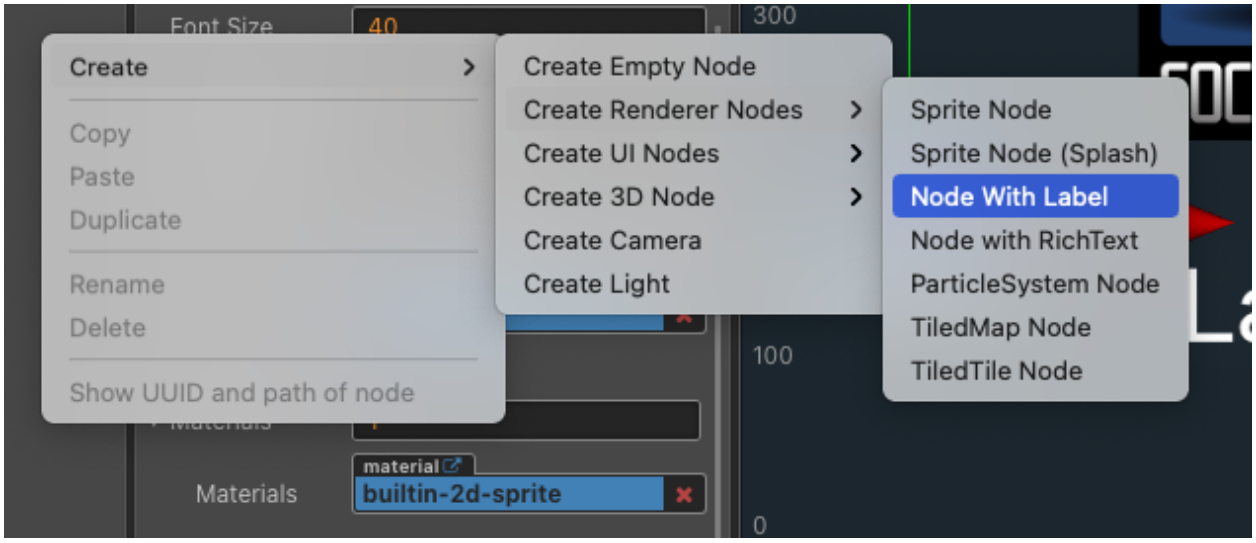



**Figure 3-8: Create Node With Label**

If you create a Label from the menu, the label will use system font by default, you can drag your font asset from **Assets** panel to the Font property field of **Label** component.

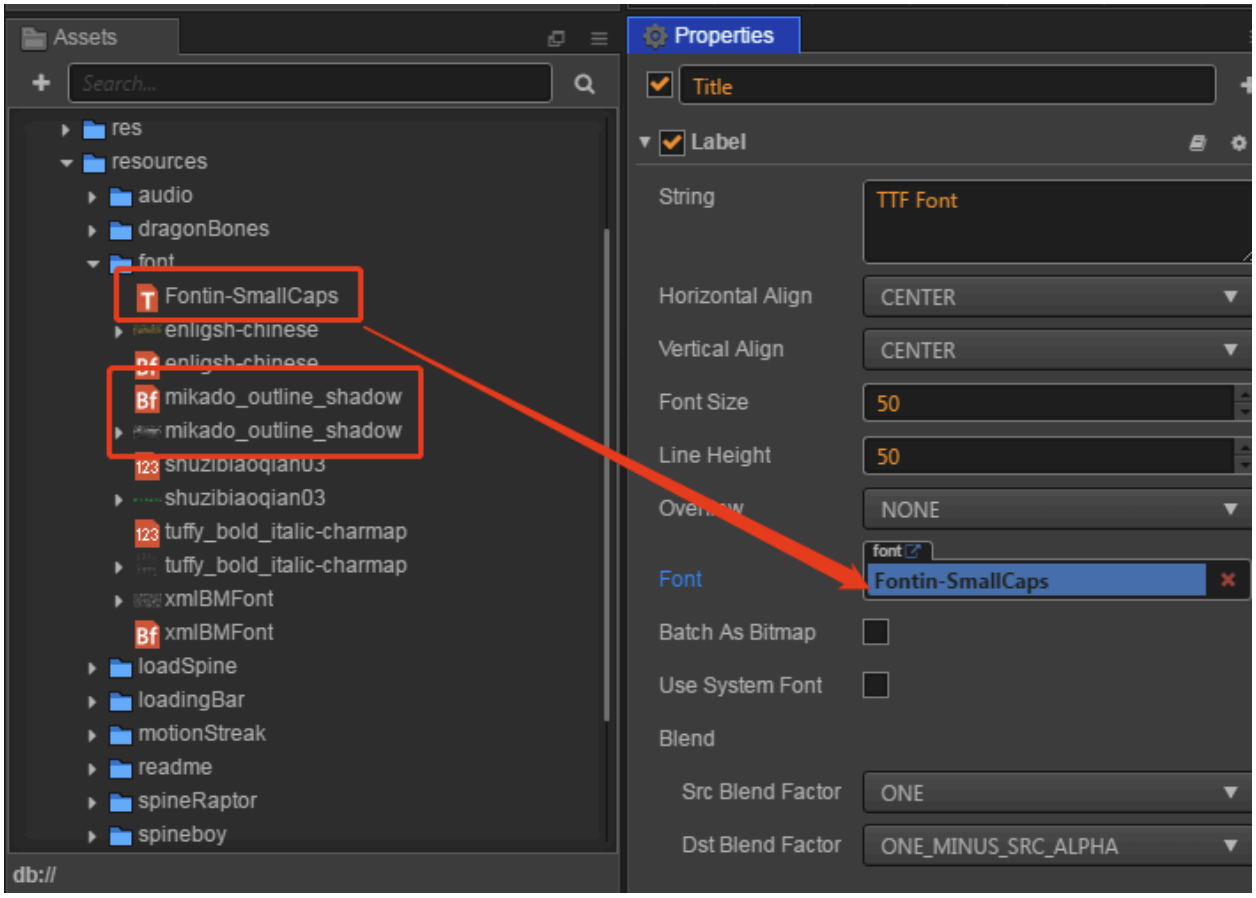



**Figure 3-9: Import Font asset to Label Nodes**

*Another shortcut method to create a font asset is to drag a TTF or Bitmap font asset from the **Assets panel** to the **Node Tree** or **Scene** panel. The selected font asset will be automatically assigned to the Font property of the Label component.*
**HOMEWORK:** Provide a PNG file, create bitmap font with Shoebox.

## Summary of this chapter

We have finished Chapter 2 and 3, got acquainted with Cocos Creator Editor and how to work with Assets, you are all set to start learning about Scripting.

As a fully scripted Cocos Creator, its preferred development language is JavaScript. JavaScript is a client-side scripting language based on object and event-driven and relatively secure. It is widely used in client-side Web development. Next Chapter we will learn about Javascript in Cocos Creator.

# Chapter 4: Workflow of script development

Cocos Creator's script is mainly developed by the extension component. Currently, Cocos Creator supports two script languages, **Javascript** and **TypeScript**. By writing the script component and putting it into the scene node, the object in the scene will be driven.

While writing the **component script**, you can map the variables needing adjustment onto the **Properties** panel by declaring the properties for the designer and graphic designer to adjust, and see the result of adjustments at any time through the preview interface. Meanwhile, you can handle specific callback events by registering specific callback functions, and add corresponding logic to the event.

## I. Creating component script

In Cocos Creator, script is also a part of the assets. You can add and select **JavaScript** or **TypeScript** to create a component script by clicking the **create** button in the **Assets** editor. By this time, you will get a new script in your asset editor.

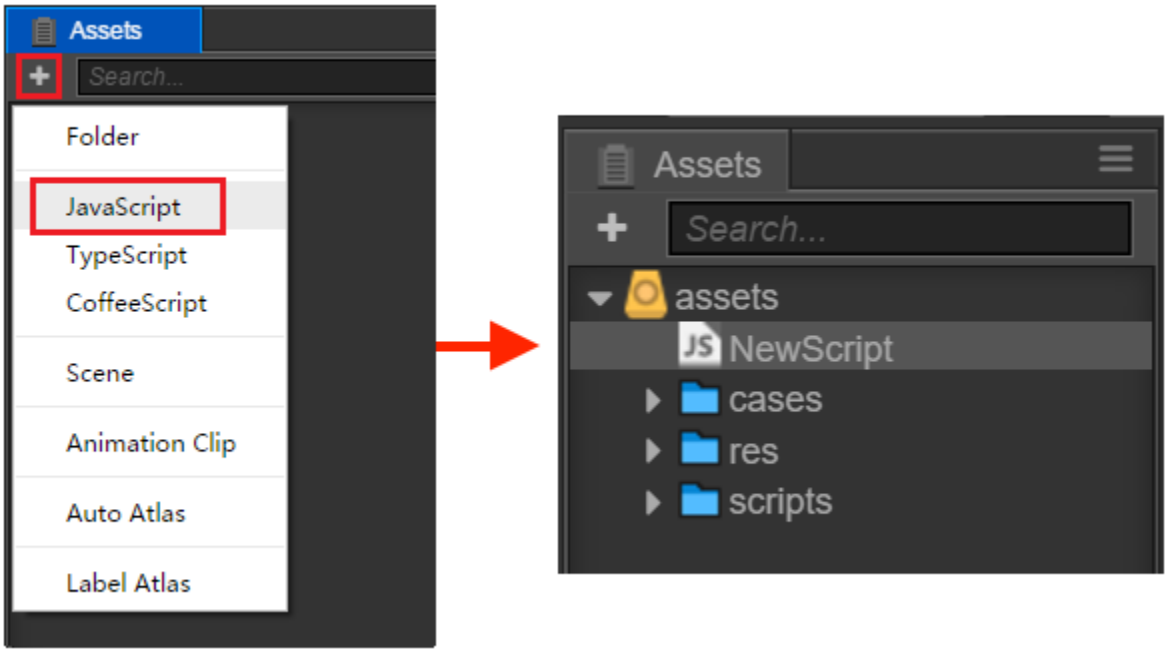



**Figure 4-1 Create new Script**

A default component script is as follows:

```
cc.Class({
    extends: cc.Component,
    properties: {
    },
// use this for initialization
    onLoad: function () {
    },
// called every frame, uncomment this function to activate update callback
    update: function (dt) {
    },
});
```

### Editing script

After creating the script, you can start editing the script by double-clicking it. The editor for editing the script can be set in "**Preferences → Data Editing**", as shown in Figure 4-2. You can set whether to automatically monitor by setting "Auto Compile script".

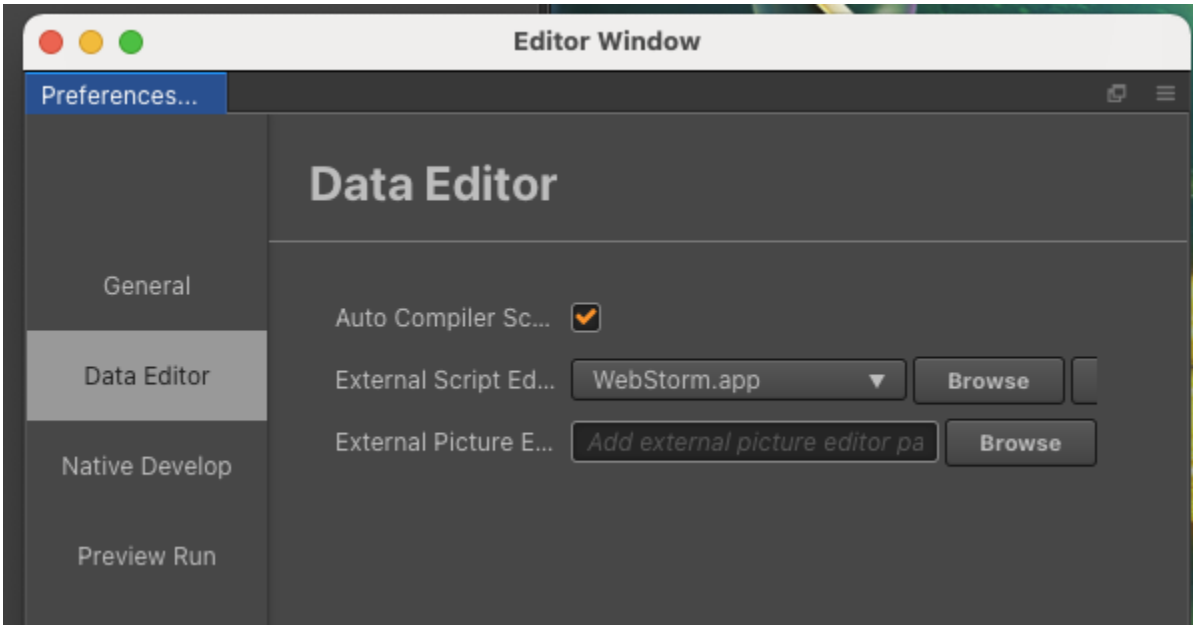



**Figure 4-2 Choose Script editing Application**

I recommend using **VS Code** or **Webstorm** for editing scripts.

Edit the scripts and save them. Cocos Creator will automatically detect the alteration of the script and compile it in no time.

### Adding script into the scene node

Adding the script into the scene node is actually adding a component to this node. Let's rename the new **NewScript.js** to **say-hello.js**. Then select the scene node you would like to add, by this time the property of this node will be shown in **Properties**.

There's an **Add Component** button at the very bottom of **Properties**. Click the button and choose **Custom Component -> say-hello** to add a new script component.

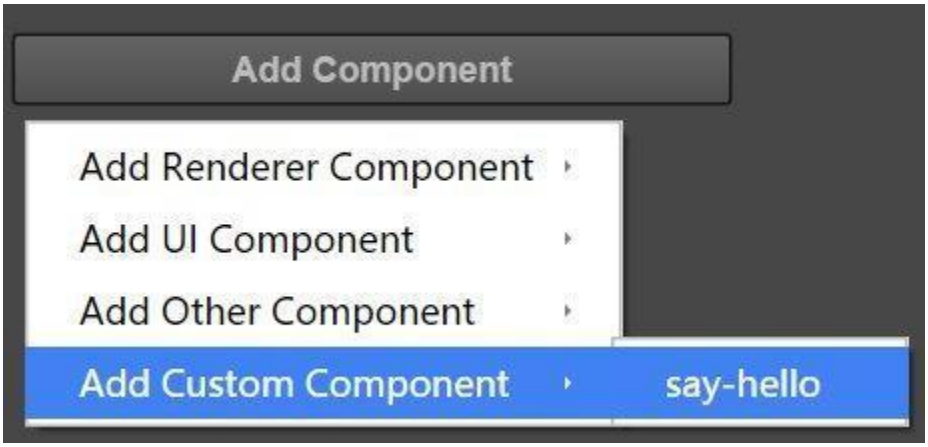



**Figure 4-3 Add Script to Node**

If everything goes well, you will see your script shown in **Properties** like in **Figure 4-4** :

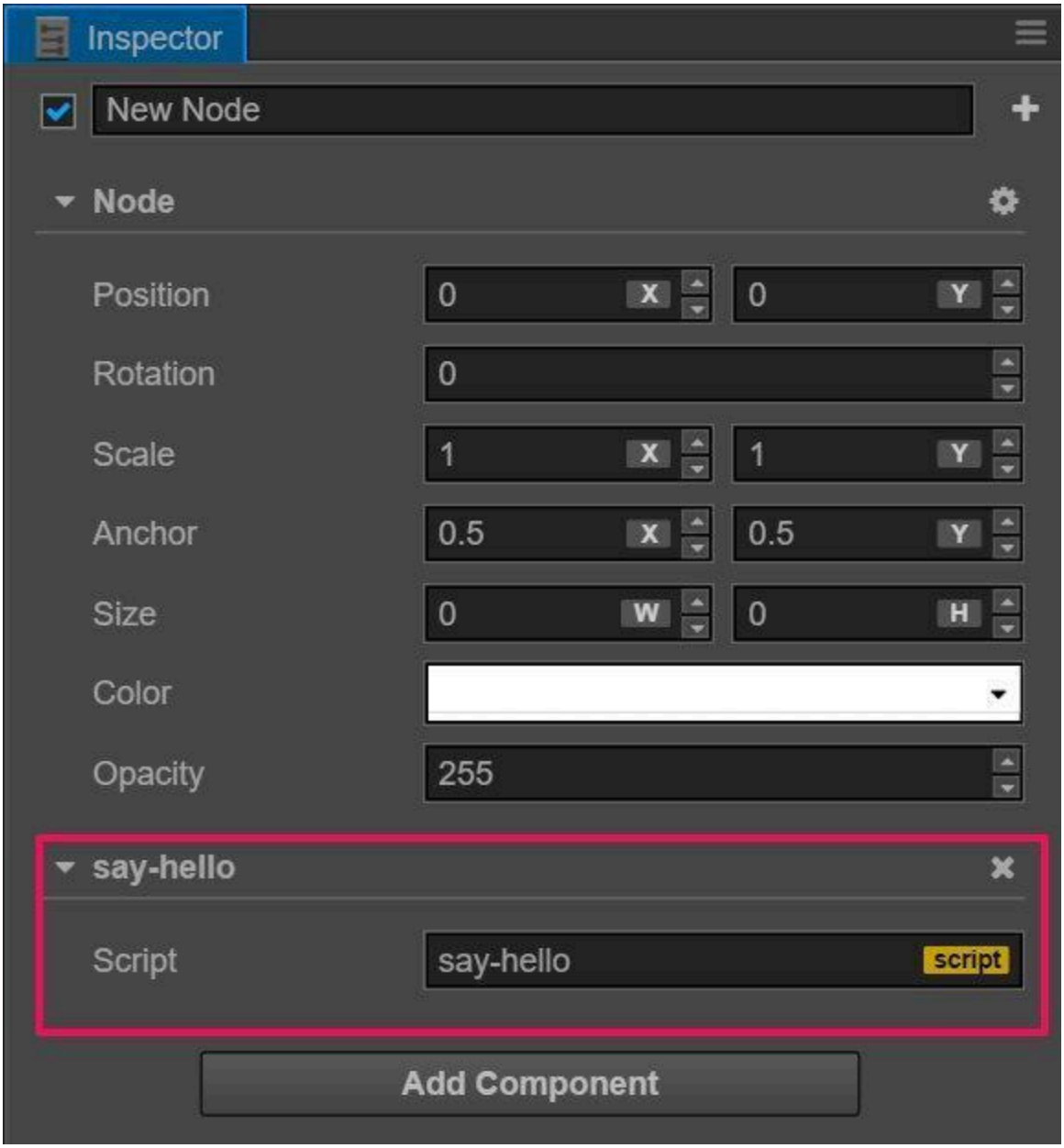



**Figure 4-4: Script in Properties Inspector**

***Note:*** *You can also add scripts by dragging script assets into* ***Properties****.*

# II. Using cc.Class declarations

cc.Class is a useful API, which is used to declare a Class in Cocos Creator. For the convenience of categorization, we call the Class that uses the cc.Class declaration **CCClass**.

#### Use CCClass Declaration

First invoke the **cc.Class** method, passing in a prototype object and set up the needed type parameters by way of a key-value pair in the prototype object, then the needed class can be created.

```
var mObject = cc.Class({
   name:"mObject"
});

var obj = new mObject();

cc.log(obj instanceof mObject);
```

The above code creates a type with CCClass, and assigns the created class to the `mObject` variable. Besides, it sets the class name to `mObject`. Class names are used for serialization, which can normally be omitted.

The above `mObject` variable is assigned to a JavaScript constructor, which can be used to create new object/

```
var obj = new mObject();
```

When you need to judge the class of an object, you can use the JavaScript built-in instanceof:

```
cc.log(obj instanceof mObject); // true
```

### Constructor

Use ctor to declare constructor:

```
var mObject = cc.Class({
   name: "mObject",
   ctor () {
       cc.log('Construct');
   },
});
```

## Instance method

```
var mObject = cc.Class({
   name: "mObject",
   ctor () {
       cc.log('Construct');
   },
   test(){
       cc.log('test');
   }
});
```

## Inheritance

Use extends to extends an already declared class:

```
  var mBigObject = cc.Class({
     extends: mObject
  });
```

## Superclass constructor

The constructor of the super class will be automatically invoked at first before the instantiation of the child class, you do **not** need to call it explicitly by yourself.

```
  var Shape = cc.Class({
     ctor: function () {
         cc.log("Shape"); // The superclass constructor will be automatically invoked during
  instantiation,
     }
  });
  var Rect = cc.Class({
     extends: Shape
  });
  var Square = cc.Class({
     extends: Rect,
     ctor: function () {
         cc.log("Square"); // then call the child constructor.
     }
  });
  var square = new Square();
```

The output of the above code is "Shape" and "Square".

## Property declaration

By employing attribute declaration in the component script, we can visually show the field of the script component in the **Properties** panel so that we can adjust the attribute value in the scene easily.

To declare the attribute, all you need to do is put the attribute name and parameters in the cc.Class defining properties field, for instance:

```
cc.Class({
    extends: cc.Component,
    properties: {
        userID: 20,
        userName: "Foobar"
    }
});
```

At this time, you can see in the **Properties** panel the two properties you just defined.

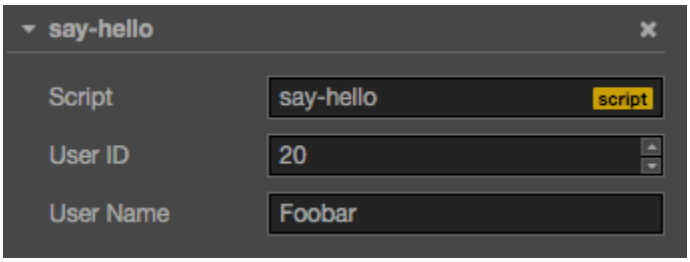



**Figure 4-5: Script Properties in Editor**

In Cocos Creator, we provide two kinds of attribute declaration methods:

## Simple declaration

In most cases, we can use simple declaration.

- When the property declared is the JavaScript primitive type, it can be written directly into:

```
properties: {
    height: 20,       // number
    type: "actor",    // string
    loaded: false,    // boolean
    target: null,     // object
}
```

- When the property declared has type (such as cc.Node, cc.Vec2 etc.), you can finish the declaration by writing its constructor in the declaration, such as:

```
properties: {
      target: cc.Node,
      pos: cc.Vec2,
  }
```

- When the delared property type is inherited from cc.ValueType (such as cc.Vec2, cc.Color, cc.Size etc.), in addition to using constructor above, it can also be assigned with an instance, such as:

```
  properties: {
      pos: new cc.Vec2(10, 20),
      color: new cc.Color(255, 255, 255, 128),
  }
```

- When the property declared is an array, you can finish the declaration by writing its type or constructor in the declaration.

```
properties: {
      any: [],      // array of arbitrary type
      bools: [cc.Boolean],
      strings: [cc.String],
      floats: [cc.Float],
      ints: [cc.Integer],

      values: [cc.Vec2],
      nodes: [cc.Node],
      frames: [cc.SpriteFrame],
  }
```

**Note:** Other than the situations above, we all need to use **complete declaration** to write for other types.

#### Complete declaration

Under some circumstances, we need to add attributes for the property declaration. These attributes control the property display mode in the **Properties** panel, and the property behavior in the scene serialization process. For example:

```
properties: {
    score: {
    default: 0,
            displayName: "Score (player)",
            tooltip: "The score of player",
    }
}
```

The code above specified three attributes: **default**, **displayName** and **tooltip**. They specify the default value of score is 0, and its property name in the **Properties** panel will be shown as "Score (player)", and when the cursor moves on to the property, it will show the corresponding Tooltip. Below are the general attributes.

- **default**: set default value for property, the default value will be used only when the component attaches to a node for the first time.
- **type**: restrict data type of property, see CCClass Advanced Reference: type attribute for details
- **visible**: the property is invisible in the **Properties** panel if set to false
- **serializable**: do not serialize this property if set to false
- **displayName**: display the assigned name on the **Properties** panel
- **tooltip**: add Tooltip of property in the **Properties** panel

For detailed usage please refer to the Property attribute.

## Array declaration

The default of array must be set to [], if you are about to edit in **Properties** panel, then you also need to set the type into **constructor**, **enumeration**, or **cc.Integer**, **cc.Float**, **cc.Boolean** and **cc.String**.

```
properties: {
    names: {
    default: [],
            type: [cc.String] // use type to specify that each element in array must
be type string
    },
    enemies: {
    default: [],
            type: [cc.Node] // type can also be defined as an array to improve
readability
    },
}
```

### get/set declaration

After setting get or set in the property, when you access the property, the pre-defined get or set method will be triggered. Defining the method is as follows:

```
properties: {
   width: {
       get: function () {
           return this._width;
       },
       set: function (value) {
           this._width = value;
       }
   }
}
```

You can define only the get method, so this is like one property of read only.

## III. Accessing Node and Component

You can edit nodes and components in the **Properties** panel, or dynamically modify them in script as well. The advantage of dynamic modification is let you modify or transit property continuously for a period of time, to achieve some easing effects. Script can also be used to respond to player inputs, or modify, create, or destroy nodes and components, so as to implement various game logics. To achieve these, you should acquire the node or component which you want to modify.

In this tutorial, we will introduce how to

- get the node which a component belongs to

Getting the node which the component belongs to is easy, just use *this.node* variable in the component:

```
testFunc () {
   var node = this.node;
   node.x = 100;
}
```

- get other component

```
testFunc () {
   var label = this.node.getComponent(cc.Label);
   var text = this.name + 'started';
```

```
// Change the text in Label Component
    label.string = text;
// You can also get a custom component by its name
    var helloComp = this.node.getComponent('say-hello');
}
```

- If the component does not exist in the node, getComponent will return null. If you are trying to access a null value at runtime, the "TypeError" exception will be thrown. So always remember to check if you are not sure of it.

```
testFunc () {
    var label = this.getComponent(cc.Label);
    if (label) {
        label.string = "Hello";
    }
    else {
        cc.error("Something wrong?");
    }
}
```

- setup node and component in **Properties** panel

The most obvious and straightforward is to set an object in the **Properties** panel. Take node as an example, it just needs to declare a property which type is cc.Label:

```
// say-hello.js
cc.Class({
    extends: cc.Component,
    properties: {
// declare player property
        lblHello: {
            default: null,
            type: cc.Label
        }
    }
});
```

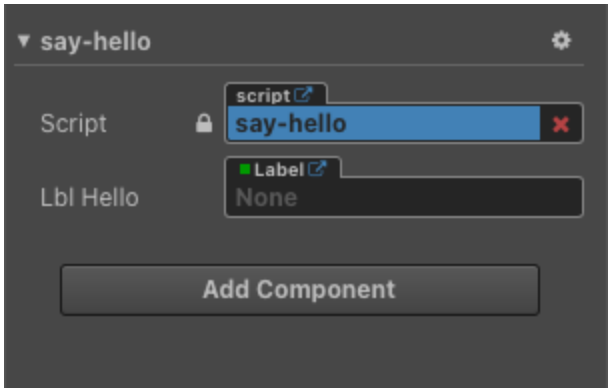



**Figure 4-6: Properties in Editor**

You can also retrieve other nodes or components from other nodes, but I'm not recommend to handle logic of a node in another node, so you can read it more here: https://docs.cocos.com/creator/2.1/manual/en/scripting/access-node-component.html

# IV. Basic Node and Component API

**Node active state and hierarchy**

Let's assume we are in a component script and use this.node to access the current node.

#### Activate/Deactivate node

The node is active by default, we can set its activation state in the code by setting the node's active property.

```
this.node.active = true/false;
```

The effect of setting the active property is the same as switching the activation and deactivation status of the node in the editor. When a node is deactivated, all of its components are disabled. At the same time, all of its child nodes and the components on the child nodes are also disabled. Note that this does not change the value of the active property on the child nodes, so they will return to their original state once the parent is reactivated.

In other words, active is actually the activation state of the node **itself**, and whether the node is **currently** active depends on its parent node. And if it is not in the current scene, it cannot be activated. We can determine whether it is currently active through the read-only property activeInHierarchy on the node.

```
this.node.active = true;
```

If the node was previously in the state that **can be activated**, modifying active to true immediately triggers the activation action:

- Reactivate the node in the scene, and all its child nodes that have the active property set to true.
- Enable all components on the current node and all child nodes, meaning the update method in these components will be called in every frame.
- If there's an onEnable method in these components, it will be called.

```
this.node.active = false;
```

If the node was previously activated, modifying active to false immediately triggers the deactivation action:

- Hide current node and all child nodes in scene.
- Disable all components on the current node and all child nodes, meaning the update method in these components will not be called.
- If there's an onDisable method in these components, it will be called.

#### Change node's parent

Assume the parent node is parentNode, child node is this.node
You can do:

```
this.node.parent = parentNode;
```

or

```
this.node.removeFromParent(false);
parentNode.addChild(this.node);
```

These two methods have equal effect.
Notice:

- removeFromParent usually needs to pass a false, otherwise it will clear all the events and actions on the node.
- A node needs to be given a parent to initialize it properly.

#### Access child node

**this.node.children** will return all child nodes of the current node.
**this.node.childrenCount** will return the number of child nodes.

## Update node transform (position, rotation, scale, size)

All basic properties of node can be set directly through code

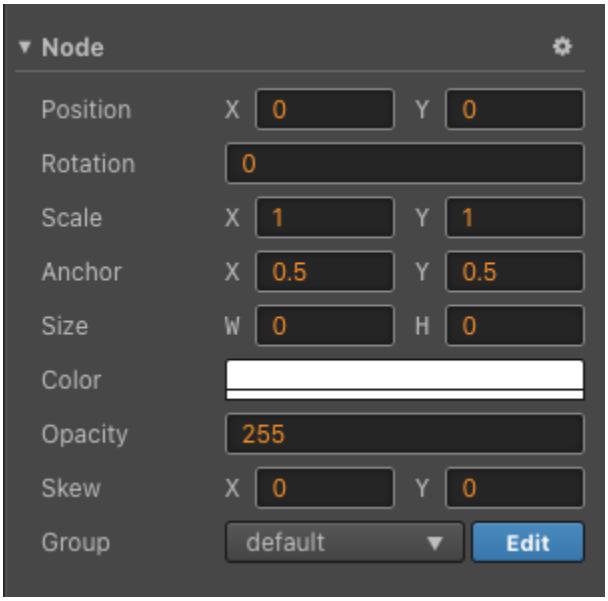



**Figure 4-7: Transformable properties**

### Position

```
//Assign position x and y
this.node.x = 100;
this.node.y = 100;
//Assign position directly
this.node.position = cc.v2(100,100);
this.node.setPosition(cc.v2(100,100));
//All above will give you the same result
```

### Rotation

```
//Set angle of a node
this.node.angle = 90;
```

### Scale

```
//Set scale of Node
this.node.scaleX = 0.5;
this.node.scaleY = 0.5;
this.node.scale = 0.5;
this.node.setScale(0.5, 0.5);
this.node.setScale(cc.v2(0.5,0.5));
```

If you pass only one parameter to setScale, both scaleX and scaleY will be changed.

#### Useful common component API

cc.Component is the base class for all components, so we can use all the following API (in the component script this is the instance of component):

- **this.node**: The node instance current component is attached to.
- **this.enabled**: When set to true, the update method will be called in each frame; for renderer components this can be used as a display switch.
- **update(dt)**: As a member method of a component, it will be called each frame when the **enabled** property is set to true.
- **onLoad()**: Will be called when the component is first loaded and initialized (when the node is inserted into the node tree)
- **start()**: Will be called before the first update run, usually used to initialize some logic which needs to be called after all components' onload methods are called.

## V. Life cycle callback

Cocos Creator provides the life cycle callback function for component scripts. As long as the user defines a specific callback function, Creator will execute the script in a specific period, users do not need to manually call them.
Currently, the major life-cycle callback functions provided for users are:

- onLoad
- start
- update
- lateUpdate
- onDestroy
- onEnable
- onDisable


**Figure 4-8: Lifecycle of a component**

**onLoad**

In the initialization phase of the component script, we provide the onLoad callback function. onLoad callback will be triggered when the node is first activated, such as when the scene is loaded, or when the node is activated. The onLoad phase guarantees that you can get other nodes from the scene and the resource data associated with the node. onLoad is always called before any start functions; this allows you to order initialization of scripts. Normally, we will do some operations associated with initialization in the onLoad phase. For example:

```
cc.Class({
    extends: cc.Component,

    properties: {
        bulletSprite: cc.SpriteFrame,
        gun: cc.Node,
    },

    onLoad: function () {
        this._bulletRect = this.bulletSprite.getRect();
        this.gun = cc.find('hand/weapon', this.node);
    },
});
```

**start**

The start callback function will be triggered before the first activation of the component, which is before executing the update for the first time. start is usually used to initialize some intermediate state data which may have changed during update and frequently enables and disables.

```
cc.Class({
    extends: cc.Component,

    start: function () {
        this._timer = 0.0;
    },

    update: function (dt) {
        this._timer += dt;
        if ( this._timer >= 10.0 ) {
            console.log('I am done!');
            this.enabled = false;
        }
    },
});
```

**update**

One of the key points for game development is to update an object's behavior, status and orientation before rendering every frame. These update operations are normally put in the update callback.

```
cc.Class({
    extends: cc.Component,

    update: function (dt) {
        this.node.setPosition( 0.0, 40.0 * dt );
    }
});
```

**lateUpdate**

update will execute before all the animations' update, but if we're going to do some extra work after the effects (such as animation, particle, physics, etc.) are updated or want to perform other operations after update of all the components are done, then we'll need the lateUpdate callback.

```
cc.Class({
    extends: cc.Component,

    lateUpdate: function (dt) {
        this.node.rotation = 20;
    }
});
```

**onEnable**

When the enabled property of the component turns from false to true, or the active property of the node turns from false to true, it will trigger onEnable callback. If the node is created for the first time, and enabled is true, then it will be called after onLoad but before start.

**onDisable**

When the enabled property of the component turns from true to false, or the active property of the node turns from true to false, it will trigger onEnable callback, and it will activate the onDisable callback.

**onDestroy**

When the component or node calls destroy(), it will call the onDestroy callback. Then they will be collected when this frame is done.

# VI. Script execution Order

Let's say I have a Node with 2 script components: ScriptA.js and ScriptB.js, and I need to control the order in which components are executed first, what should I do?
In the **Properties Inspector,** the **Component** that stays above will execute before the Component stays below.
If the above methods still can not provide the required fine grained control, you can also set the executionOrder of the component directly. The executionOrder affects the execution priority of the life cycle callbacks for components. Set as follows:

```
// Player.js
cc.Class({
    extends: cc.Component,
    editor: {
        executionOrder: -1
    },
    onLoad: function () {
        cc.log('Player onLoad!');
    }
});
```

```
// Menu.js
cc.Class({
    extends: cc.Component,
    editor: {
        executionOrder: 1
    },
    onLoad: function () {
        cc.log('Menu onLoad!');
    }
});
```

The smaller the executionOrder is, the earlier the component executes relative to the other components. The executionOrder defaults to 0, so if set to negative, it will execute before the other default components. The executionOrder will only affect onLoad, onEnable, start, update and lateUpdate while onDisable and onDestroy will not be affected.

# VII. Modular Script

Cocos Creator allows you to split the code into multiple script files and they can be called by each other. To implement this, you need to know how to define and use the module in Cocos Creator. This step is called **modularize** for short.

Modularization enables you to reference other script files in Cocos Creator:

- Access parameters exported from other files
- Call method other files that have been exported
- Use type other files that have been exported
- Use or inherit other Components

JavaScript in Cocos Creator uses the same Common JS standard as Node.js to implement modularization, in short:

- Each individual script file forms a module
- Each module is an individual action scope
- Reference other modules in the synchronized require method
- Set module.exports as an exported variable

If you still don't quite understand, don't worry, we will explain it here.
*In this article, the two terms "module" and "script" are equivalent. All the "comment" parts belong to advanced contents that don't need to be understood at the very start. No matter how we define the module, all user designation codes will eventually be compiled into native JavaScript by Cocos Creator and can be run directly in the browser.*

## Reference module

### require

Other than the interface provided by Creator, all the user-defined modules will need to be called "**require"** to be accessed. For instance, we have a component defined at **Rotate.js**:

```
// Rotate.js

cc.Class({
    extends: cc.Component,
    // ...
});
```

Now if you want to access it in another script, you can:

```
var Rotate = require("Rotate");
```

What "**require"** returned is the object exported by the module. Normally, we would save the result to a variable(`var Rotate`)immediately. The incoming "**require"** character string is the module's file name, the name contains neither path nor suffix and it is case-sensitive.

#### require complete example

Next, we can use Rotate to derive a subclass and create a new script SinRotate.js:

```
// SinRotate.js

var Rotate = require("Rotate");

var SinRotate = cc.Class({
    extends: Rotate,
    update: function (dt) {
        this.rotation += this.speed * Math.sin(dt);
    }
});
```

Here, we define a new component named **SinRotate**, which is inherited from **Rotate**, and rewrite the update method.
This component can also be accessed by other scripts as long as you use 'require("SinRotate")'.
Comments:

- require could be called at any place in the script at any time.
- All of the scripts will be automatically required when the game is started. At this time, the defined code in each module will be executed once, no matter how many times it is required, the same instance will be returned.
- When debugging, any module in the project can be required in the Console of Developer Tools.

### Define module

#### Define component

Each individual script file is a module, such as the new script Rotate.js mentioned above:

```
// Rotate.js

var Rotate = cc.Class({
    extends: cc.Component,
    properties: {
        speed: 1
    },
    update: function () {
        this.transform.rotation += this.speed;
    }
});
```

When you declare a component in the script, Creator will acquiesce to export it so other scripts can use it by requiring this module.

#### Define regular JavaScript module

You not only can define a component in the module, but you can also export any JavaScript object. Let's imagine that there is a script **config.js**

```
// config.js

var cfg = {
    moveSpeed: 10,
    version: "0.15",
    showTutorial: true,

    load: function () {
        // ...
    }
};
cfg.load();
```

Now, if we want to access the cfg object in another script:

```
// player.js

var config = require("config");
cc.log("speed is", config.moveSpeed);
```

The result will report an error: `"TypeError: Cannot read property 'moveSpeed' of null"`, this is because cfg has not been set as the export object. We also need to set `module.exports` as config at the end of config.js:

```
// config.js - v2
var cfg = {
    moveSpeed: 10,
    version: "0.15",
    showTutorial: true,

    load: function () {
        // ...
    }
};
cfg.load();

module.exports = cfg;
```

The reason for doing this is because as long as there is another script that requires it, what they actually get will be the module.exports object in here.

In this way, it can output correctly: "speed is 10".

*The default value of module.exports:*

*If a script does not declare module.exports, Creator will set exports as the Component declared in script automatically. And if a script does not declare any Component but declares other types of CCClass, it will set exports as declared CCClass automatically.*

Comments：

- The other variables added to module can not be exported, that is to say exports can not be replaced with other variable names, the system will only read the exports variable

# Chapter 5: Render and UI Components

*A complete game is generally composed of different systems. From a technical point of view, it generally includes a **UI system**, an **animation system**, a **physics system**, and a **sound system**. As a developer, we are not going to handle much about sound-related contents, and the rest of the content will be introduced separately from this chapter. For mobile games currently on the market, especially 2D games, in addition to the main gameplay and combat, the main workload is the development of the UI interface. This chapter will learn about the UI system in Cocos Creator.*

## I. Basic rendering components

- **Sprite Component**

For rendering components, the most basic is the rendering of images. In Cocos Creator, the image rendering components are sprites. The properties of the sprites are shown in Figure 5-1.

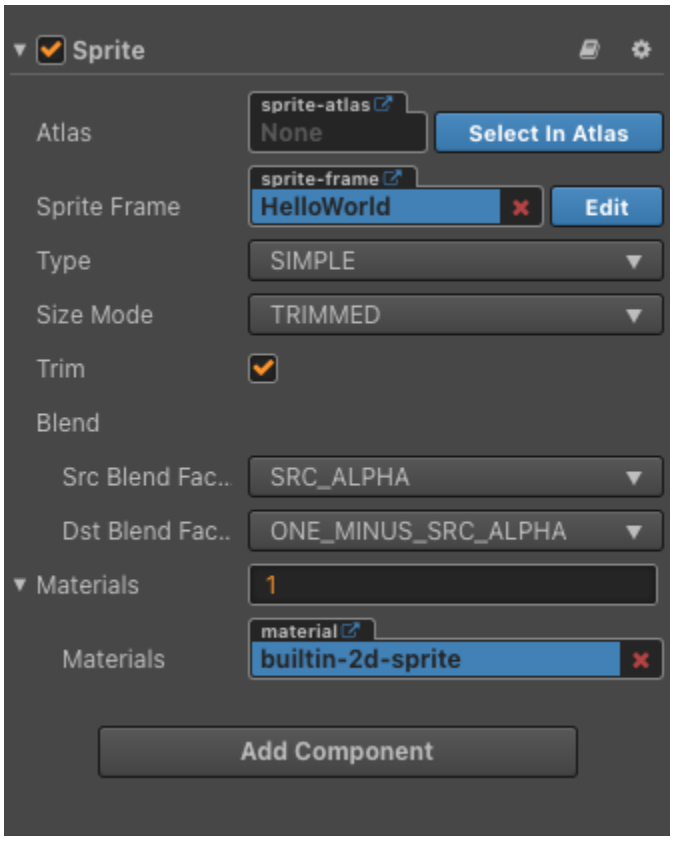



**Figure 5-1: Properties of Sprite**

The basic properties in the **Properties Inspector** are quite simple and easy to understand:

- **Atlas** represents the name of Atlas that SpriteFrame belongs to.
- **Sprite Frame** represents the name of the specific image. For single images, it has the same name as the file name.

- **Type** is the rendering method, currently Cocos Creator 2.1.3 supports 5 rendering methods, which will be introduced in detail later.
- **Size Mode** is the method to set the size of the node that the Sprite component belongs to. **TRIMMED** mode will use Image size after trimmed transparent pixel, **RAW** mode will use original size of picture. If you manually set the size of the node, **Size Mode** will automatically change to **CUSTOM**.
- **Blend** and **Materials** are blending settings, which will not be in this course's scope.

The sprite component supports 5 rendering modes as listed below:

1. `Simple mode`: rendering the Sprite according to the original image resource. It is normally used along with `Use Original Size` to guarantee the image shown in the scene is in full accordance with the image designed by the graphic designer.
2. `Sliced mode`: the image is cut up into a 9-slicing and according to certain rules is scaled to fit freely set dimensions (`size`). It is usually used in UI elements or to make images that can be enlarged infinitely without influencing the image quality into images cut up into a grid to save game resource space. To create a stretchable sprite, double click on the **SpriteFrame** from the **Asset Panel** to open the **Editor Window**. After opening **Sprite Editor**, you will see there is a green line around the image, which indicates the position of the current 9-sliced split line. Drag the mouse to the split line, you will see the shape of the cursor change, then you can press down and drag the mouse to modify the position of the split line. We click and drag the four split lines at the left/right/upper/lower side respectively and cut the image into a 9-slicing. The nine areas will apply different zooming in/out strategies when the Sprite size changes, which is as illustrated in Figure 5-2:

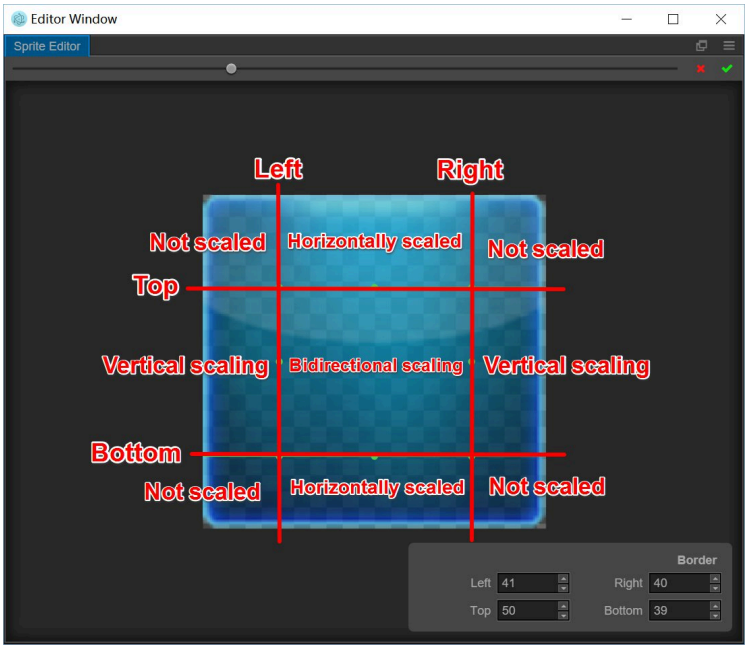
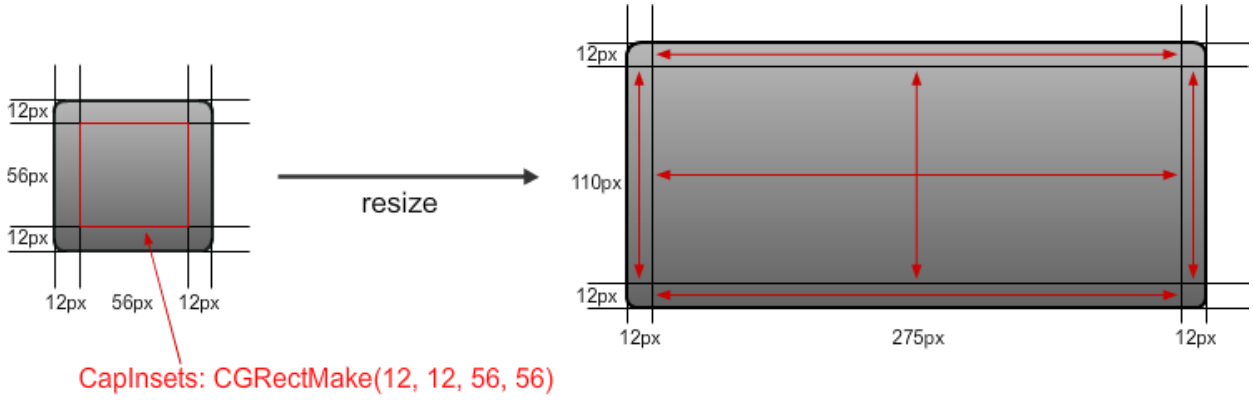



**Figure 5-2: Sprite's Editor Window**

And the following picture illustrates the state of zooming in/out in different areas:




After cutting, don't forget to click the green check mark on the upper right corner of **Sprite's Editor Window** to save modifications to the resource.

3. `Tiled mode`: The image will be repeated to fit the size of the Sprite. If the SpriteFrame is 9-sliced, when the image is tiled, the surrounding width will remain unchanged while the other sections will be repeated.
4. `Filled mode`: draws a portion of the original picture in a certain direction and scale, based on the origin and fill mode settings. Often used for dynamic display of progress bars. For the properties meaning of the filling mode, see Table 5-1:

| Properties | Function Explanation |
| --- | --- |
| Fill Type | Fill type selection, including `HORIZONTAL`, `VERTICAL`, and `RADIAL`. |
| Fill Start | Normalized values for filling starting position (from 0 ~ 1, denoting the percentage of total population), when you select a horizontal fill, the `Fill Start` is set to 0, and it is populated from the leftmost edge of the image. |
| Fill Range | Normalized values for padding ranges (same from 0 ~ 1). When set to 1, it fills up the entire range of the original image. |
| Fill Center | Fill center point, this property appears only if the `Radial` type is selected. Determines which point on the Sprite is used as pivot when the FillType is set to RADIAL, the coordinate system used is the same as the Anchor. |

5. `Mesh mode`: Only support **.plist** file which is built by **TexturePacker** (4.0 or higher version) with polygon algorithm.

*Sprite API ref: [https://docs.cocos.com/creator/api/en/classes/Sprite.html](https://docs.cocos.com/creator/api/en/classes/Sprite.html)*

- **Label Component**

The Label component is used to display a paragraph of text. The text can use a system font, a TrueType font or a Bitmap font. The Label component arranges and renders the text. Click the Add component button under the property inspector, and then you can select Label in the rendering component and add the Label component to the node. The label's property editing interface is shown in Figure 5-3.

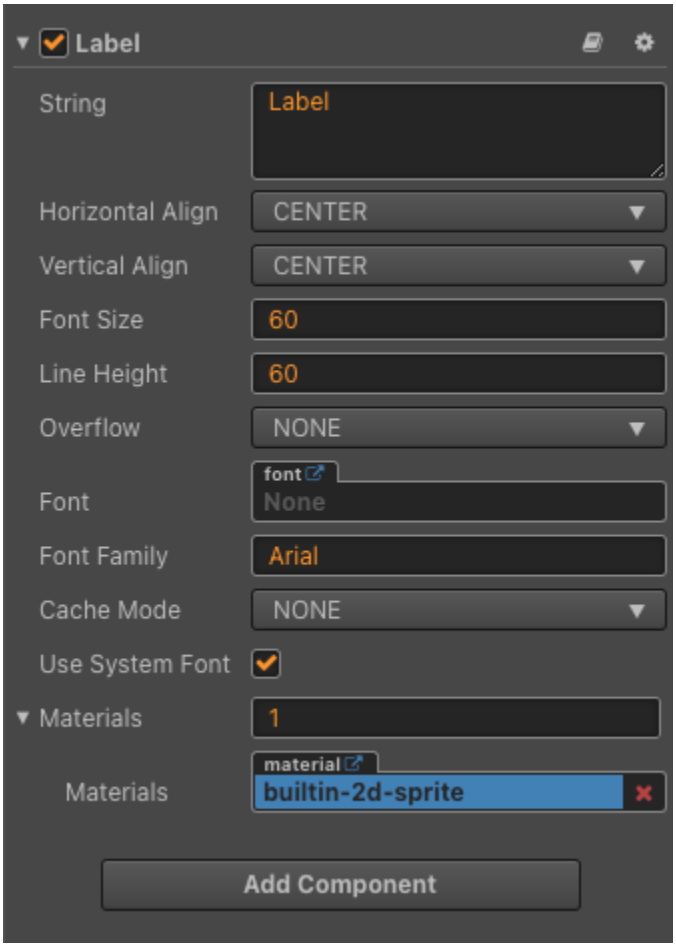



**Figure 5-3: Label Component Properties**

Description of Label Component's properties is in Table 5-2:

| Properties | Function Explanation |
| --- | --- |
| String | Text content character string. |
| Horizontal Align | Horizontal alignment pattern of the text. The options are LEFT, CENTER and RIGHT. |
| Vertical Align | Vertical alignment pattern of the text. The options are TOP, CENTER and BOTTOM. |
| Font Size | Font size of the text. |
| SpacingX | The spacing between font characters, only available in BMFont. |
| Line Height | Line height of the text. |
| Overflow | Layout pattern of the text. Currently supports CLAMP, SHRINK and RESIZE_HEIGHT. |
| Enable Wrap Text | Enable or disable the text line feed. (which takes effect when the Overflow is set to CLAMP or SHRINK) |
| Font | Designate the font file needed for rendering the text. If the system font is used, then this attribute can be set to null. |
| Font Family | Font family name, takes effect when using system font. |
| Cache Mode | The text cache mode. Takes effect only for **system font** or **ttf** font, BMFont does not require this optimization. And includes **NONE**, **BITMAP**, **CHAR** three modes. |
| Use System Font | Boolean value, choose whether to use the system font or not. |

There are 3 ways to layout the string:

1. CLAMP: The text size won't zoom in or out as the Bounding Box size changes. When Wrap Text is disabled, parts exceeding the Bounding Box won't be shown according to the normal character layout. When Wrap Text is enabled, it will try to wrap the text exceeding the boundaries to the next line. If the vertical space is not enough, any not completely visible text will also be hidden.
2. SHRINK: The text size will zoom in or out (it won't zoom out automatically, the maximum size that will show is specified by Font Size) as the Bounding Box size changes. When Wrap Text is enabled, if the width is not enough, it will try to wrap the text to the next line before automatically adapting the Bounding Box's size to make the text show completely. If Wrap Text is disabled, then it will compose according to the current text and zoom automatically if it exceeds the boundaries. **Note**: This mode may take up more CPU resources when the label is refreshed.
3. RESIZE_HEIGHT: The text Bounding Box will adapt to the layout of the text. The user cannot manually change the height of text in this status; it is automatically calculated by the internal algorithm.

#### Cache Mode

| Properties | Function Explanation |
| --- | --- |
| NONE | Defaults, the entire text in label will generate a bitmap |
| BITMAP | After selection, the entire text in the Label will still generate a bitmap, but will try to participate in Dynamic Atlas. As long as the requirements of Dynamic Atlas are met, the Draw Call will be merged with the other Sprite or Label in the Dynamic Atlas. Because Dynamic Atlas consumes more memory, **this mode can only be used for Label with infrequently updated text**. **Note**: Similar to NONE, BITMAP will force a bitmap to be generated for each Label component, regardless of whether the text content is equivalent. If there are a lot of Labels with the same text in the scene, it is recommended to use CHAR to reuse the memory space. |

| | |
|---|---|
| CHAR | The principle of CHAR is similar to BMFont, Label will cache text to the global shared bitmap in "word" units, each character of the same font style and font size will share a cache globally. Can support frequent modification of text, the most friendly to performance and memory. However, there are currently restrictions on this model, which we will optimize in subsequent releases:<br><br>1. **This mode can only be used for font style and fixed font size (by recording the fontSize, fontFamily, color, and outline of the font as key information for repetitive use of characters, other users who use special custom text formats need to be aware). And will not frequently appear with a huge amount of unused characters of Label.** This is to save the cache, because the global shared bitmap size is 2048*2048, it will only be cleared when the scene is switched. Once the bitmap is full, the newly appearing characters will not be rendered.<br><br>2. Overflow does not support SHRINK.<br><br>3. Cannot participate in dynamic mapping (multiple labels with CHAR mode enabled can still merge Draw Call in the case of without interrupting the rendering sequence). |

**Note**:

- Cache Mode has an optimized effect for all platforms.
- The BITMAP mode replaces the original Batch As Bitmap option, and old projects will automatically migrate to this option if Batch As Bitmap is enabled.
- The **RenderTexture** module in the **Project -> Project Settings -> Module Config** panel cannot be removed when using the CHAR mode.

*Label API ref: [https://docs.cocos.com/creator/api/en/classes/Label.html](https://docs.cocos.com/creator/api/en/classes/Label.html)*

# II. Commonly used UI components

## 1. Layout

Layout is a kind of container, which can turn on the automatic layout function, automatically arrange objects according to certain specifications, and make it convenient for users to make tabs and lists. The layout container is divided into three types: **HORIZONTAL**, **VERTICAL** and **GRID**. Just like adding other components, you can click the Add Component button under the Property inspector, and then select Layout from Add UI Components. After adding UI components, the default layout type is NONE. When manually placing objects, the container will use the smallest rectangular area that can accommodate all sub-objects as its own size, without changing the position and size of any sub-objects. By modifying the Type The type can be switched between the three layout container types, as shown in Figure 5-4.

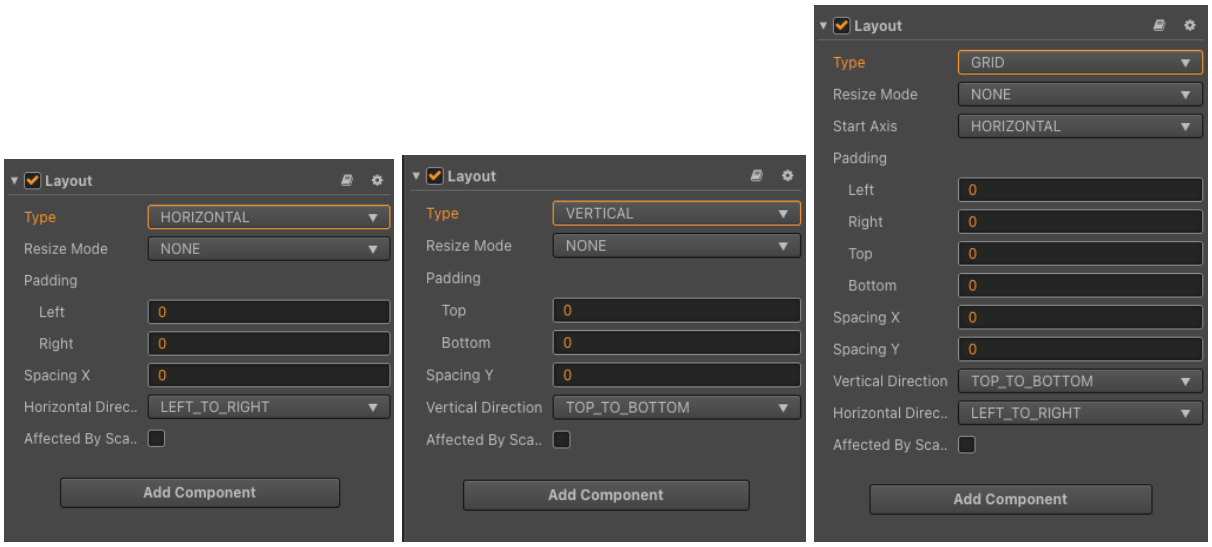



**Figure 5-4: 3 Container types of Layout Component**

Layout property

| Property | Function Explanation |
| --- | --- |
| Type | Layout type, currently has NONE, HORIZONTAL, VERTICAL and Grid. |
| Resize Mode | Resize strategies , currently have NONE, CHILDREN and CONTAINER. |
| Padding Left | The left padding between the sub-object and the container frame in the layout. |
| Padding Right | The right padding between the sub-object and the container frame in the layout. |
| Padding Top | The top padding between the sub-object and the container frame in the layout. |
| Padding Bottom | The bottom padding between the sub-object and the container frame in the layout. |
| Spacing X | The separation distance between sub-objects in the horizontal layout. NONE mode doesn't have this attribute. |
| Spacing Y | The separation distance between sub-objects in the vertical layout. NONE mode doesn't have this attribute. |
| Horizontal Direction | When it is designated as a horizontal layout, which side does the first child node start in the layout? The left or the right? |
| Vertical Direction | When it is designated as a vertical layout, which side does the first child node start in the layout? The upside or the downside? |
| Cell Size | This option is only available in Grid layout, Children resize mode. The size of each child element. |
| Start Axis | This option is only available in Grid layout, the arrangement direction of children elements. |
| Affected By Scale | Whether the scaling of the child node affects the layout. |

It should be noted that the layout component does not affect the scaling and rotation of the child nodes. At the same time, when you set the layout through code, the result of the setting needs to be updated in the next frame. You can also call the `updateLayout` function to update the layout.

## 2. Button

Button is the most commonly used UI component. It can respond to the user's click operation. By setting the clicked callback button, the corresponding logic can be called. The properties of the Button component are shown in Figure 5-5.

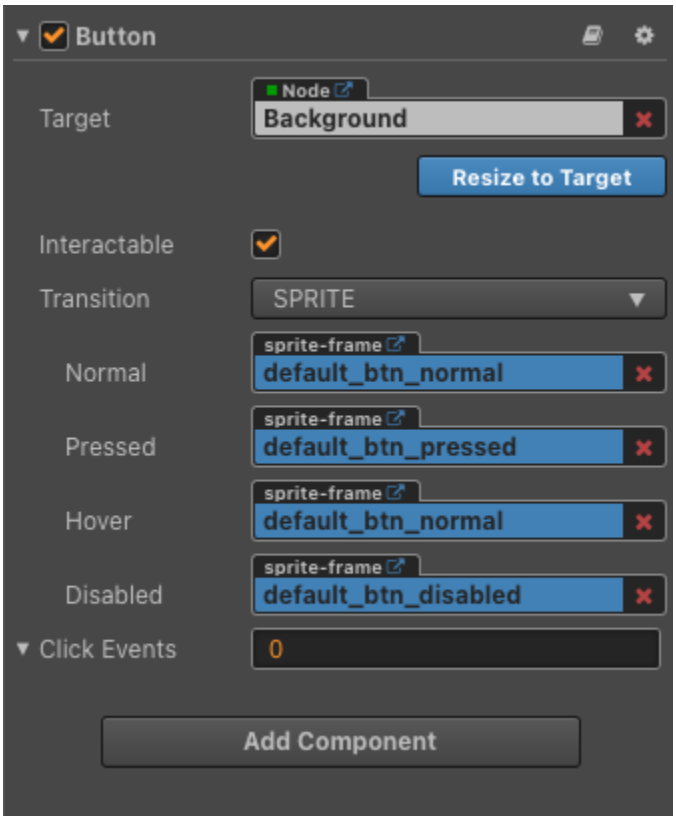



**Figure 5-5: Button Component**

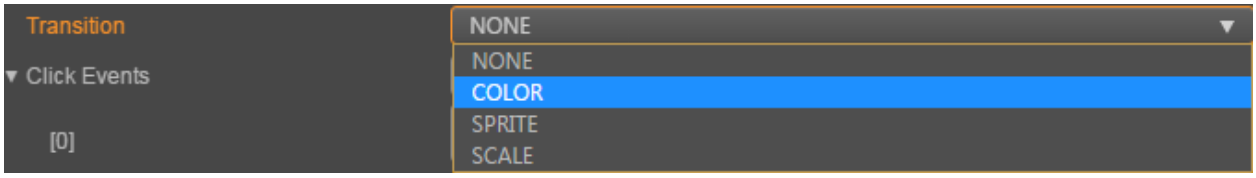
**Button Property**

| Property | Function explanation |
| --- | --- |
| Interactable | Boolean type, if set to false then the Button component enters the forbidden state. |
| Enable Auto Gray Effect | Boolean type, if set to true, the Button's target sprite will turn gray when interactable is false. |
| Transition | Enumeration type, including NONE, COLOR and SPRITE. Each type corresponds to a different Transition setting. Please see the Button Transition section below for details. |
| Click Event | Default list type is null. Each event added by the user is composed of the node reference, component name and a response function. Please see the Button Event section below for details. |

Note: When `Transition` is `SPRITE` and the `disabledSprite` property has a `spriteFrame` associated with it, the `Enable Auto Gray Effect` property is ignored at this time.

**Button Transition**

Button Transition is used to indicate the status of the Button when clicked by the user. Currently the types available are NONE, COLOR, SPRITE and SCALE.




| Property | Function Explanation |
| --- | --- |
| Normal | Color or Sprite of Button under Normal status. Available in Transition **COLOR** and **SPRITE** |
| Pressed | Color or Sprite of Button under Pressed status. Available in Transition **COLOR** and **SPRITE** |
| Hover | Color or Sprite of Button under Hover status. Available in Transition **COLOR** and **SPRITE** |
| Disabled | Color or Sprite of Button under Disabled status. Available in Transition **COLOR** and **SPRITE** |
| Duration | Time interval needed for Button status switching. Available in Transition **COLOR** and **SCALE** |
| ZoomScale | When user press the button, the button will zoom to a scale.The final scale of the button equals (button original scale * zoomScale), zoomScale could be a negative value. Available in Transition **SCALE** |

#### Button Click Event

The Button can additionally add a Click event to respond to the player's click action. There are two ways to do this: Add a callback through the **Properties** or through the **Script.**

**Add a callback through the Properties.**

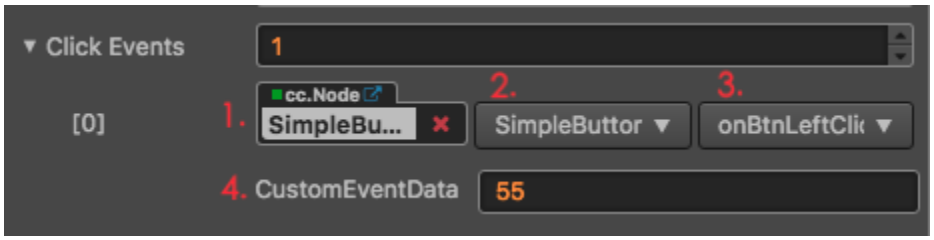



**Figure 5-6: Add button's callback through the Properties**

| No. | Property | Function Explanation |
| --- | --- | --- |
| 1 | Target | Node with the script component. |
| 2 | Component | Script component name. |
| 3 | Handler | Assign a callback function which will be triggered when the user clicks the Button. |
| 4 | customEventData | A user-defined string value passed as the last event argument of the event callback. |

**Add a callback through the script.**

There are two ways to add a callback through the script.

1. The event callback added by this method is the same as the event callback added by the editor, all added by the ***Button component***. First you need to construct a `cc.Component.EventHandler` object, and then set the corresponding `target`, `component`, `handler` and `customEventData` parameters.

```
// here is your component file, file name = MyComponent.js
cc.Class({
   extends: cc.Component,
   properties: {},

   onLoad: function () {
       var clickEventHandler = new cc.Component.EventHandler();
       clickEventHandler.target = this.node; // This node is the node to which your event
handler code component belongs
       clickEventHandler.component = "MyComponent";// This is the code file name
       clickEventHandler.handler = "callback";
       clickEventHandler.customEventData = "foobar";

       var button = this.node.getComponent(cc.Button);
       button.clickEvents.push(clickEventHandler);
   },

   callback: function (event, customEventData) {
       // here event is a Event object, you can get events sent to the event node node
       var node = event.target;
       var button = node.getComponent(cc.Button);
       // here the customEventData parameter is equal to the one you set before the "foobar"
   }
});
```

**2.** By `button.node.on ('click', ...)` way to add, this is a very simple way, but there are some limitations in the event callback which can not get the screen coordinate point of the current click button.

```
cc.Class({
     extends: cc.Component,

     properties: {
         button: cc.Button
     },

     onLoad: function () {
         this.button.node.on('click', this.callback, this);
     },

     callback: function (button) {
         // do whatever you want with button
         // In addition, attention to this way registered events, can not pass
customEventData
     }
 });
```

### 3. Editbox

EditBox is a text input component, you could use this component to gather user input easily.

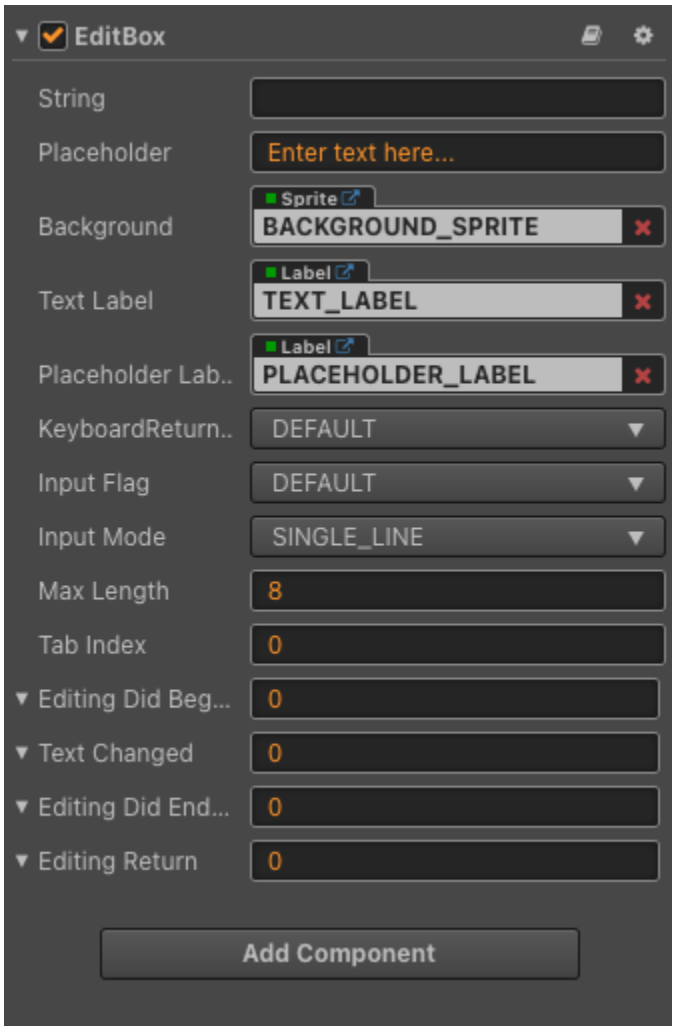



**Figure 5-7: EditBox Properties**

**Editbox property**

| Property | Function Explanation |
| --- | --- |
| String | The initial input text of EditBox. |
| Placeholder | The content string of the placeholder. |
| Background | The Sprite component attached to the node for EditBox's background. |
| Text Label | The Label component attached to the node for EditBox's input text label. |
| Placeholder Label | The Label component attached to the node for EditBox's placeholder text label. |
| KeyboardReturnType | The keyboard return type of EditBox. This is useful for keyboard of mobile device. |
| Input Flag | Specify the input flag: password or capitalize word. (Only supports Android platform) |
| Input Mode | Specify the input mode: multiline or single line. |
| Max Length | The maximize input characters. |
| Tab Index | Set the tabIndex of the DOM input element, only useful on Web. |
| Editing Did Began | The event handler to be called when EditBox began to edit text. Please refer to the Editing Did Began event below for details. |
| Text Changed | The event handler to be called when EditBox text changes. Please refer to the Text Changed event below for details. |
| Editing Did Ended | The event handler to be called when EditBox edit ends. Please refer to the Editing Did Ended event below for details. |
| Editing Return | The event handler to be called when return key is pressed, Currently does not support windows platform. Please refer to the Editing Return event below for details. |

| Event Name | Built-in Event | Function Explanation |
| --- | --- | --- |
| EditingDidBegan | editing-did-began | This event will be triggered when the user clicks on EditBox. |
| TextChanged | text-changed | This event will be triggered each time when the content in EditBox is changed. |
| EditingDidEnded | editing-did-ended | This event will be triggered when the EditBox loses focus.<br>Usually when in single line input mode, it's triggered after user presses Return key or clicks the area outside of EditBox.<br>When in multiline input mode, it's triggered only after user click the area outside of EditBox. |
| EditingReturn | editing-return | This event will be triggered when the user presses the Return key or presses the Done button on soft keyboard on the mobile.<br>In single line mode, EditBox may lose its focus if users press Return. |

Like the Button Component, Editbox can additionally add a Click event to respond to the player's click action. There are two ways to do this: Add a callback through the **Properties** or through the **Script. (HOMEWORK)**

## 4. Richtext

Sometimes Label Component do not meet all of what we need, for example mixing images and texts, or displaying multiple colors in one sentence. Cocos provides a component that can adapt these requirements, called RichText. RichText component could be used for displaying a string with multiple styles. You could customize the text style of each text segment with a few simple BBCode.

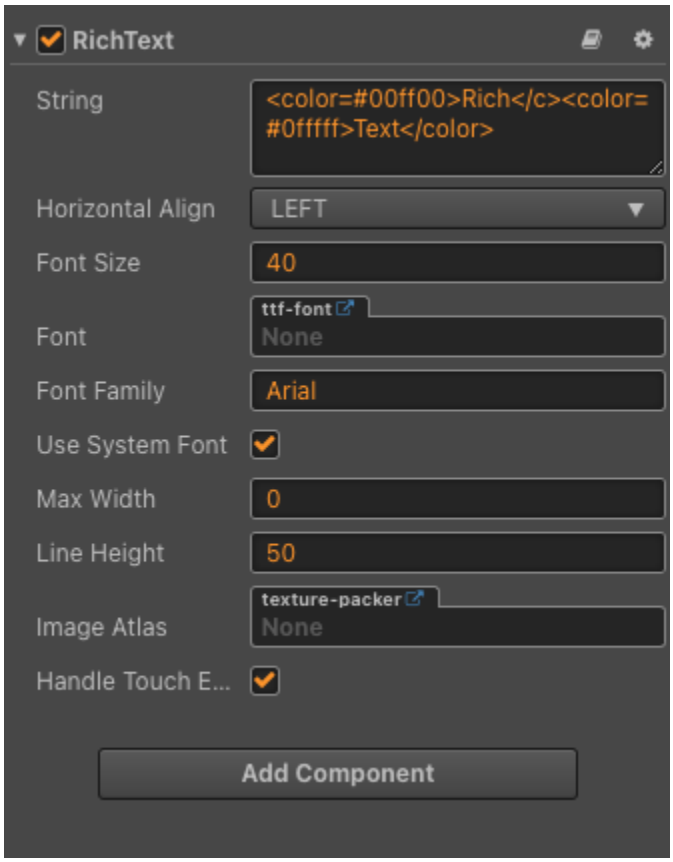



Figure 5-8: RichText Component

RichText Component Properties are described in Table below:

| Property | Function Explanation |
| --- | --- |
| String | Text of the RichText, you could use BBcode in the string |
| Horizontal Align | Horizontal alignment |
| Font Size | Font size, in points |
| Font | Custom TTF font of RichText, all the label segments will use the same custom ttf font. |
| Font Family | Custom system font of RichText. |
| Use System Font | Whether to use the system default font. |
| Max Width | The maximum width of RichText, pass 0 means not limit the maximum width. |
| Line Height | Line height, in points |
| Image Atlas | The image atlas for the img tag. For each src value in the img tag, there should be a valid spriteFrame in the image atlas. |
| Handle Touch Event | Once checked, the RichText will block all input events (mouse and touch) within the bounding box of the node, preventing the input from penetrating into the underlying node. |

**BBCode Format:**
Currently the supported tag list is: size, color, b, i, u, img and on. There are used for customizing the font size, font color, bold, italic, underline, image and click event. Every tag should have a begin tag and an end tag. The tag name and property assignment should be all lowercase. It will check the start tag name, but the end tag name restrict is loose, it only requires a </> tag, the end tag name doesn't matter.

*Supported tags*

Note: all tag names should be lowercase and the property assignment should use `=` sign.

| Name | Description | Example | Note |
|---|---|---|---|
| color | Specify the font rendering color, the color value could be a built-in value or a hex value. eg, use #ff0000 for red. | `<color=#ff0000>Red Text</color>` | For built-in color, please refer to cc.Color |
| size | Specify the font rendering size, the size should be an integer. | `<size=30>enlarge me</size>` | |
| outline | Specify the font outline, you could customize the outline color and width by using the `color` and `width` attribute. | `<outline color=red width=4>A label with outline</outline>` | If you don't specify the color and width attribute, the default color value is #ffffff and the default width is 1. |
| b | Render text as bold font | `<b>This text will be rendered as bold</b>` | The tag name must be lowercase and tag name `bold` is not supported. |
| i | Render text as italic font | `<i>This text will be rendered as italic</i>` | The tag name must be lowercase and tag name `italic` is not supported. |
| u | Add a underline to the text | `<u>This text will have a underline</u>` | The tag name must be lowercase and tag name `underline` is not supported. |
| on | Specify an event callback to a text node, when you click the node, the callback will be triggered. | `<on click="handler"> click me! </on>` | Every valid tag could also add another click event attribute. eg. `<size=10 click="handler2">click me</size>` |

| | | | |
|---|---|---|---|
| param | When the click event is triggered, the value can be obtained in the second parameter of the callback function. | `<on click="handler" param="test"> click me! </on>` | Depends on the click event |
| br | Insert a empty line | `<br/>` | Note: `<br></br>` and `<br>` are both invalid tags. |
| img | Add image emoji support to your RichText. The emoji name should be a valid spriteframe name in the ImageAtlas property. | `<img src='emoji1' click='handler' />` | Note: Only `<img src='foo' click='bar' />` is a valid img tag. If you specify a large emoji image, it will scale the sprite height to the line height of the RichText together with the sprite width. |

## 5. ProgressBar

ProgressBar is usually used to show the progress of a certain operation in the game. Add the ProgressBar component to a node and associate a Bar Sprite to this component. The Bar Sprite can then be controlled to show progress in the scene, or HP etc… Figure 5-9 shows main properties of ProgressBar Component

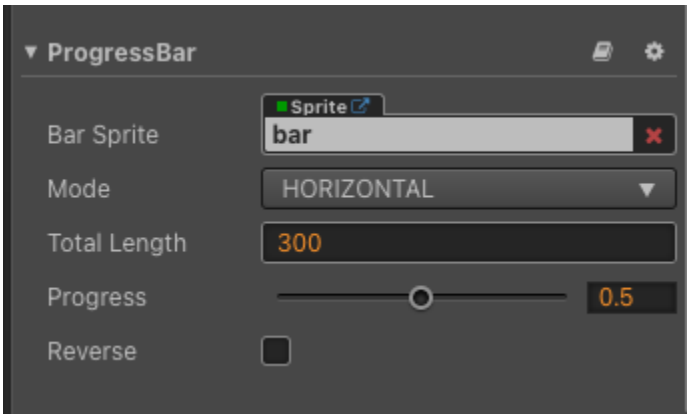



**Figure 5-9: ProgressBar Component**

ProgressBar Component Properties are described in Table below:

| Property | Function Explanation |
| --- | --- |
| Bar Sprite | The Sprite component needed for rendering ProgressBar. It can be linked by dragging a node with the **Sprite** component to this attribute. |
| Mode | Currently supports the **HORIZONTAL**, **VERTICAL** and **FILLED** modes. The initial direction can be changed by cooperating with the **Reverse** attribute. |
| Total Length | The total length/total width of the Bar Sprite when the ProgressBar is at 100%. In **FILLED** mode, **Total Length** represents the percentage of the total display range for Bar Sprite, with values ranging from 0 to 1. |
| Progress | Floating point. The value range is 0~1, and values outside the range are not allowed. |
| Reverse | Boolean value. The default fill direction is from left to right/bottom to top, and when enabled, it becomes right to left/top to bottom. |

You can directly set the value of the Progress property and set the position of the progress bar. The code is shown below.

```
//Set the progress bar
_updateProgressBar(progressBar,dt) {
   let progress = progressBar.progress;
   if( progress < 1.0){
       progress += dt*this.speed;
   } else{
       progress = dt*this.speed;
   }
   progressBar.progress = progress;
}
```

## 6. Slider

Slider is a slider component, For the production of UI components such as volume adjustment. Figure 5-10 shows main properties of Slider Component

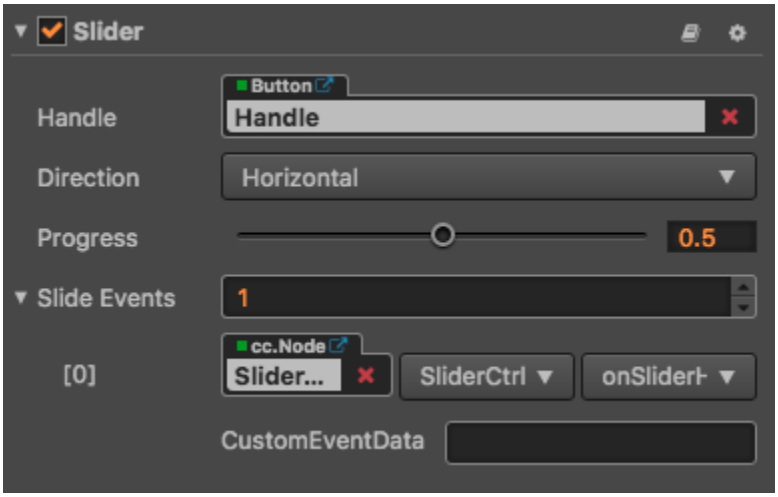



**Figure 5-10: Slider Component**

Slider Component Properties are described in Table below:

| Property | Function explanation |
|---|---|
| handle | Slider button parts can be adjusted through the button to adjust the size of the Slider value |
| direction | The direction of the slider is divided into horizontal and vertical |
| progress | Current progress value, the value of the interval is between 0-1 |
| slideEvents | Slider component event callback function |

There are also two ways to bind slider events. The specific code is shown below:

```
//Method One
var sliderEventHandler = new cc.Component.EventHandler();
sliderEventHandler.target = this.node; // Is the node of your event handling code
component
sliderEventHandler.component = "cc.MyComponent"
sliderEventHandler.handler = "callback";
sliderEventHandler.customEventData = "foobar";

slider.slideEvents.push(sliderEventHandler);

// here is your component file
cc.Class({
    name: 'cc.MyComponent'
    extends: cc.Component,

    properties: {
    },

    callback: function(slider, customEventData) {
        // Where slider is a cc.Slider object
        // Where the customEventData parameter is equal to the "foobar"
    }
});

//Method Two
      this.slider.node.on('slide', this.callback, this);
```

## 7. PageView

On smartphones, due to the limited size of the screen, it is often necessary to display some information in pages. At this time, the page container component group is needed. It is said to be a component group because it consists of two components, which are the PageView, and the PageViewIndicator. The PageView component properties are shown in Figure 5-11:

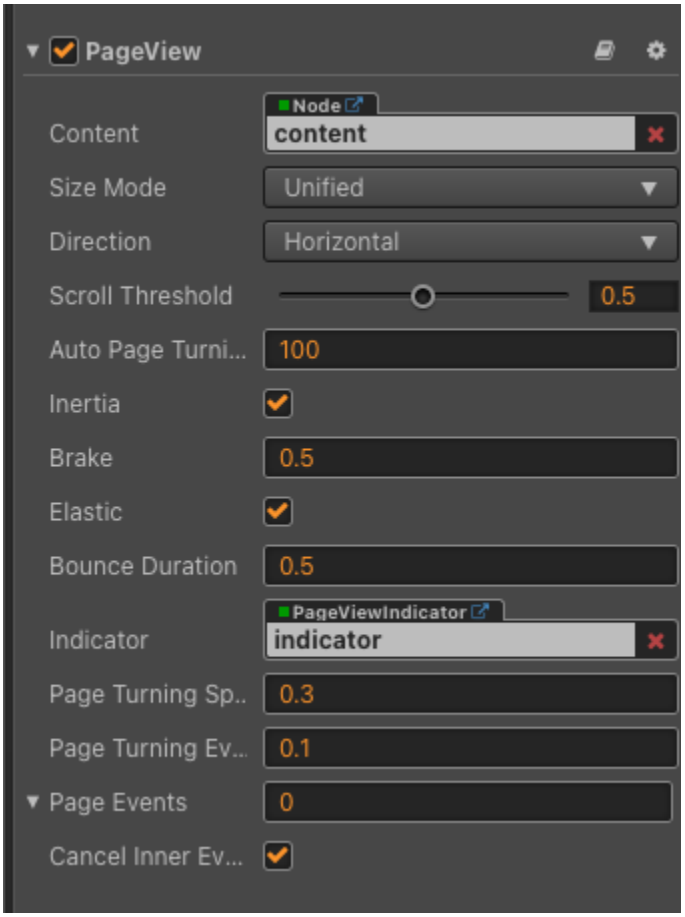



**Figure 5-11: PageView component**

PageView Component Properties are described in Table below:

| Property | Function description |
| --- | --- |
| Content | It is a node reference that is used to contain the contents of the PageView |
| Size Mode | Specify the size type of each page in PageView, currently has Unified type and Free type. |
| Direction | The page view direction |
| Scroll Threshold | This value will be multiplied with the distance between two pages, to get the threshold distance. If user scroll distance is larger than this threshold distance, the page will turn immediately |
| Auto Page Turning Threshold | Auto page turning velocity threshold. When users swipe the PageView quickly, it will calculate a velocity based on the scroll distance and time, if the calculated velocity is larger than the threshold, then it will trigger page turning. |
| Inertia | When inertia is set, the content will continue to move when touch ended |
| Brake | It determines how quickly the content stops moving. A value of 1 will stop the movement immediately. A value of 0 will never stop the movement until it reaches the boundary of scrollview. |
| Elastic | When elastic is set, the content will be bounce back when move out of boundary |
| Bounce Duration | The elapse time of bouncing back. A value of 0 will bounce back immediately |
| Indicator | The Page View Indicator, please refer to **CCPageViewIndicator Set Up** below for details. |
| Page Turning Speed | The time required to turn over a page. |
| Page Turning Event Timing | Change the PageTurning event timing of PageView |
| Page Events | PageView events callback |
| Cancel Inner Events | If cancelInnerEvents is set to true, the scroll behavior will cancel touch events on inner content nodes. It's set to true by default. |

The PageView component must have the specified content node to work, Each child node in content is a separate page, The size of each page is the size of the PageView node, The operation effect is divided into two kinds:

- Slow sliding, by dragging the page in the view to reach the specified ScrollThreshold value (the value is the percentage of page size) after the release will automatically slide to the next page.
- Fast sliding, fast to a direction to drag, automatically slide the next page, each slide up to only one page.

PageViewIndicator is optional, the component is used to display the number of pages and the current page.

The association can be done by dragging a node with a PageViewIndicator component into the **Indicator** property of the PageView component in the **Node Tree**.


Figure 5-12: PageViewIndicator Component

### PageviewIndicator property

| Property | Function description |
| --- | --- |
| Sprite Frame | The spriteFrame for each element |
| Direction | The location direction of PageViewIndicator, currently has HORIZONTAL, VERTICAL |
| Cell Size | The cellSize for each element |
| Spacing | The distance between each element |

### Add callback via script code

#### Method one

The event callback added by this method is the same as the event callback added by the editor, all added by code. First you need to construct a `cc.Component.EventHandler` object, and then set the corresponding `target`, `component`, `handler` and `customEventData` parameters.

```
var pageViewEventHandler = new cc.Component.EventHandler();
pageViewEventHandler.target = this.node; // Is the node of your event handling code component
pageViewEventHandler.component = "cc.MyComponent"
pageViewEventHandler.handler = "callback";
pageViewEventHandler.customEventData = "foobar";

pageView.pageEvents.push(pageViewEventHandler);

// here is your component file
cc.Class({
    name: 'cc.MyComponent'

    extends: cc.Component,
```

```
    properties: {
    },

    // Note: that the order and type of parameters are fixed
    callback: function(pageView, eventType, customEventData) {
        // Where pageView is a cc.PageView object
        // Where eventType === cc.PageView.EventType.PAGE_TURNING
        // Where the customEventData parameter is equal to the "foobar"
    }
});
```

**Method two**

By pageView.node.on('page-turning', ...) way to add

```
// Suppose we add event handling callbacks to the onLoad method of a
component and perform event handling in the callback function:


cc.Class({
    extends: cc.Component,

    properties: {
       pageView: cc.PageView
    },

    onLoad: function () {
       this.pageView.node.on('page-turning', this.callback, this);
    },

    callback: function (pageView) {
       // The parameter of the callback is the pageView component.
       // do whatever you want with pageView
    }
});
```

## 8. ToggleContainer

The select box is also a relatively special component, which is often implemented to meet the user's requirements for selecting several kinds of information. It can be composed of Toggle and ToggleGroup to form a check box.
The toggle component is a CheckBox, when it is used together with a ToggleGroup, it could be treated as a RadioButton.

The attributes of Toggle are shown in Figure 5-13.

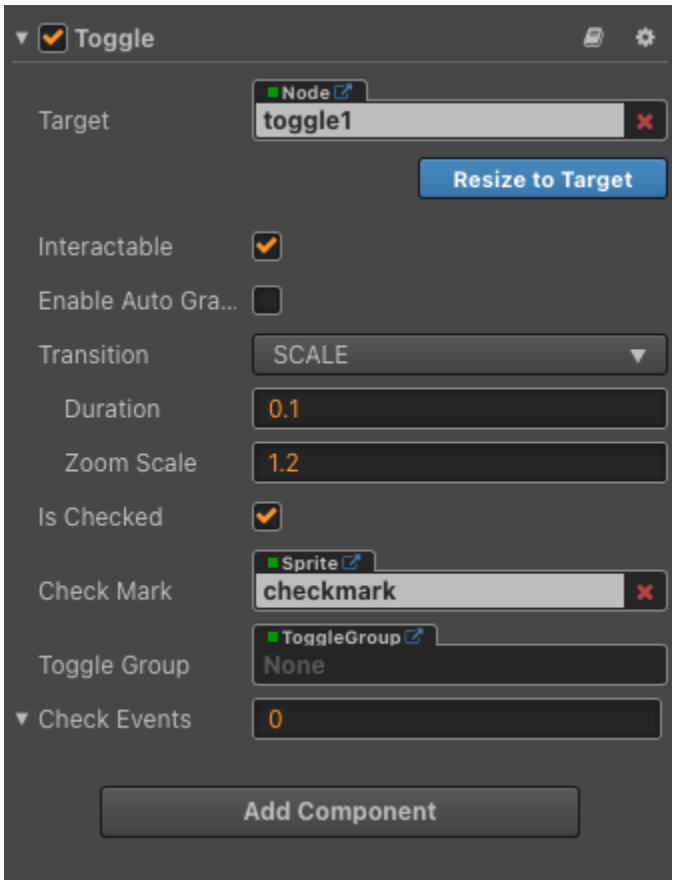



**Figure 5-13 Toggle Component**

**Toggle properties**

| Properties | Function Explanation |
| --- | --- |
| isChecked | Boolean type, When this value is true, the check mark component will be enabled, otherwise the check mark component will be disabled. |
| checkMark | cc.Sprite type, The image used for the checkmark. |
| toggleGroup | cc.ToggleGroup type, The toggle group which the toggle belongs to, when it is null, the toggle is a CheckBox. Otherwise, the toggle is a RadioButton. |
| Check Events | Default list type is null. Each event added by the user is composed of the node reference, component name and a response function. Please check more detailed information in the chapter `Button Event`. |

Note: Because Toggle is inherited from Button, so the attributes that exist in Button also apply to Toggle.

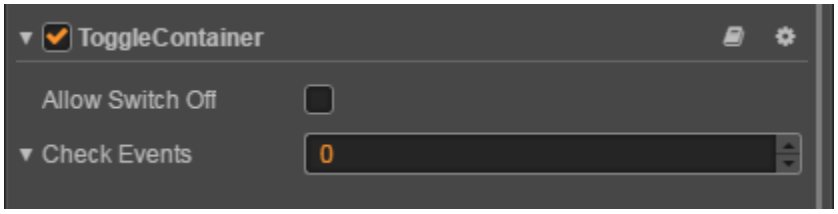
**Toggle Event**

| Properties | Function Explanation |
| --- | --- |
| Target | Node with the script component. |
| Component | Script component name. |
| Handler | Assign a callback function which will be triggered when the user clicks and releases the Toggle. |
| customEventData | A user-defined string value passed as the last event argument of the event callback. |

The Toggle event callback has two parameters, the first one is the Toggle itself and the second argument is the customEventData.

ToggleContainer is not a visible UI component but a way to modify the behavior of a set of Toggles. Toggles that belong to the same group could only have one of them to be switched on at a time.

**Note**: All the first layer child node containing the toggle component will auto be added to the container




Click the **Add component** button at the bottom of the **Properties** panel and select **ToggleContainer** from **UI Component**. You can then add the ToggleContainer component to the node.

The API reference of ToggleContainer is here: ToggleContainer API.

ToggleContainer property

| Property | Functions Explanation |
| --- | --- |
| Allow Switch Off | If this setting is true, a toggle could be switched off and on when pressed. If it is false, it will make sure there is always only one toggle that could be switched on and the already switched on toggle can't be switched off. |
| Click Event | Default list type is null. Each event added by the user is composed of the node reference, component name and a response function. Please see the **ToggleContainer Event** section below for details. |

ToggleContainer Click Event

| Property | Function Explanation |
| --- | --- |
| Target | Node with the script component. |
| Component | Script component name. |
| Handler | Assign a callback function which will be triggered when the user presses Return key. |
| customEventData | A user-defined string value passed as the last event argument of the event callback. |

## 9. ScrollView

Like paging, the UI component that is often used in mobile platform games is the scrolling list. The scrolling list is a container with scrolling function. It provides a way for users to browse more content in a limited display area. . It contains three components, ScrollView, Mask and ScrollBar. It must contain a Content node to be used. The properties of the ScrollView component are shown in Figure 5-14.

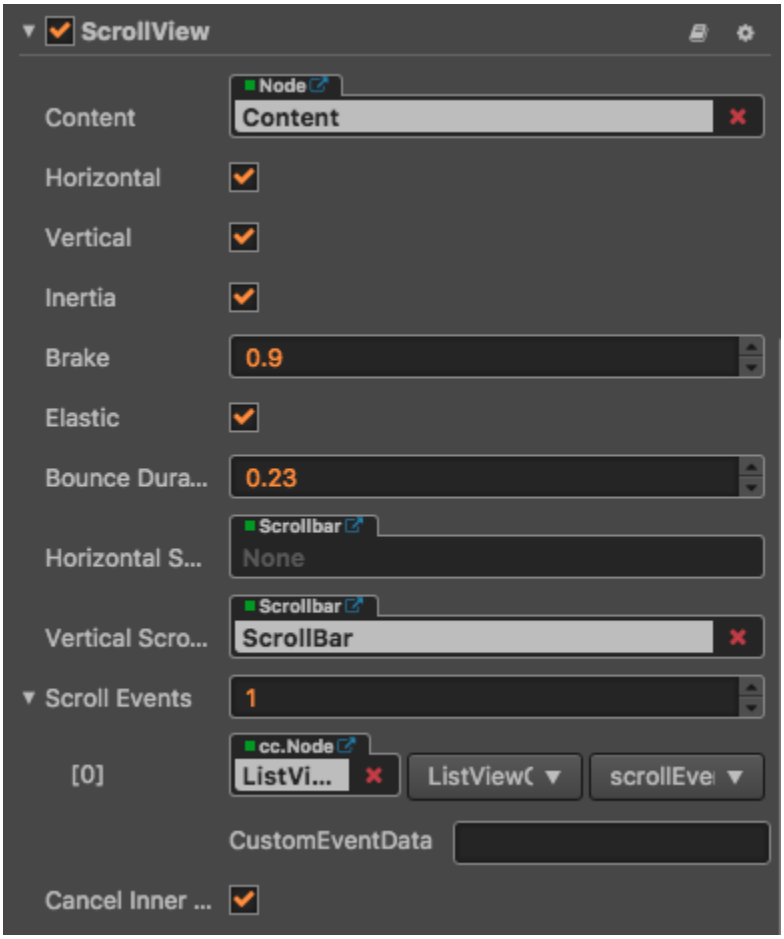



Figure 5-14: ScrollView Component

**ScrollView property**

| Property | Function Explanation |
| --- | --- |
| Content | A reference node for creating scrollable content from ScrollView. It could be a node containing a very large picture. |
| Horizontal | Boolean value, whether horizontal scroll is allowed or not. |
| Vertical | Boolean value, whether vertical scroll is allowed or not. |
| Inertia | Is there an accelerating velocity when scrolling |
| Brake | Floating point number, the deceleration coefficient after scrolling. The value range is 0-1 where if set to 1, then the scroll will stop immediately; if set to 0, then the scroll will continue until the content border. |
| Elastic | Boolean value, whether to bounce back or not. |
| Bounce Duration | Floating point number, the time duration for bounce back. The calue range is 0-10. |
| Horizontal ScrollBar | A reference node for creating a scroll bar showing the horizontal position of the contents. |
| Vertical ScrollBar | A reference node for creating a scroll bar showing vertical position of the contents. |
| Scroll Events | Default list type is null. Each event added by the user is composed of the node reference, component name and a response function. Please see the **ScrollView Event** section below for details. |
| CancelInner Events | If cancelInnerEvents is set to true, the scroll behavior will cancel touch events on inner content nodes. |

## Summary of this chapter

This chapter introduces the UI system of Cocos Creator. As the most important system in 2D mobile games, the UI system will occupy a large proportion of your learning time and actual use. Therefore, it is no exaggeration to say that this chapter is the most basic knowledge and the most important chapter. The UI system of Cocos Creator includes the basic rendering component-Sprite, text Label and complex UI components. Before introducing complex UI components, this chapter first introduces the related knowledge of screen adaptation in CocosCreator, which is the basis of UI components. UI components have always been the most important function of various mobile engines. After several generations of product evolution, the UI components of Cocos Creator are very easy to use and can also contain all the functions of game UI on the market. To learn the content of this chapter, you need to learn more on the basis of basic knowledge and practice more. You can find some UI examples of the game to imitate and make, and strengthen your proficiency in the use of UI components.

# Chapter 6: Event system, builtin Event and Event Emitter

## I.Event System

Event system is processed in `cc.Node`. Components of Nodes can register and monitor events by visiting node with this.node.

### Listen to events

Script component can register events by the function `this.node.on(..)` . Example are as below:

```
//ScriptA.js
cc.Class({
  extends: cc.Component,

  properties: {
  },

  onLoad: function () {
    this.node.on('mousedown', function ( event ) {
      console.log('Hello!');
    });
  },
});
```

In this example, scriptA registered event `'mousedown'` and print out Hello to console.

What's worth mentioning is that the event listener function `'on'` can pass to the third parameter target to bind the caller of the response function. The following two calling methods have the same effect:

```
// bind using the function
this.node.on('mousedown', function ( event ) {
  this.enabled = false;
}.bind(this));

// use the third parameter
```

```
this.node.on('mousedown', function (event) {
  this.enabled = false;
}, this);
```

### Shut Event Listener

We can shut the corresponding event listener using off when we don't care about a certain event anymore. One thing to note is that the parameter of off must be in one-to-one correspondence with the parameter of on in order to shut it.

Below are what we recommend you to put in:

```
cc.Class({
  extends: cc.Component,

  _sayHello: function () {
    console.log('Hello World');
  },

  onEnable: function () {
    this.node.on('foobar', this._sayHello, this);
  },

  onDisable: function () {
    this.node.off('foobar', this._sayHello, this);
  },
});
```

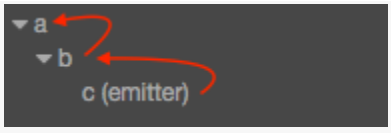
### Launch event

There are 2 ways to launch event: through `emit` and `dispatchEvent`.
Emit can contain at most 5 arguments, and cannot deliver to upper level in node hierarchy.

```
cc.Class({
  extends: cc.Component,

  onLoad () {
    // args are optional param.
    this.node.on('say-hello', function (msg) {
      console.log(msg);
    });
  },

  start () {
    // At most 5 args could be emit.
    this.node.emit('say-hello', 'Hello, this is Cocos Creator');
  },
});
```

### Event delivery

Events launched by the `dispatchEvent` method mentioned above would enter the event delivery stage. In Cocos Creator's event delivery system, we use bubble delivery. Bubble delivery will pass the event from the initiating node continually on to its parent node until it gets to the root node or is interrupted processed by `event.stopPropagation()` in the response function of some node.




As shown in the picture above, when we send the event “foobar” from node c, if both node a and b listen to the event“foobar”, the event will pass to node b and a from c. For example:

```
// In the component script of node c

this.node.dispatchEvent( new cc.Event.EventCustom('foobar', true) );
```

If we want to stop the event delivery after node b intercepts the event, we can call the function `event.stopPropagation()` to do this. Detailed methods are as follows:

```
// In the component script of node b

this.node.on('foobar', function (event) {

  event.stopPropagation();

});
```

Be noted, when you want to dispatch a custom event, please do not use `cc.Event` because it's an abstract class, instead, you should use `cc.Event.EventCustom` to dispatch a custom event.

### Event object

In the call-back of the event listener, the developer will receive an event object `event` of the `cc.Event` type. `stopPropagation` is the standard API of `cc.Event`, other important API include:

| API name | type | meaning |
| --- | --- | --- |
| `type` | `String` | type of the event (event name) |
| `target` | `cc.Node` | primary object received by the event |
| `currentTarget` | `cc.Node` | current object receiving the event; current object of the event in the bubble stage may be different from the primary object |
| `getType` | `Function` | get the type of the event |
| `stopPropagation` | `Function` | stop the bubble stage, the event will no longer pass on to the parent node while the rest of the listeners of the current node will still receive the event |

| | | |
|---|---|---|
| `stopPropagationImmediate` | `Function` | stop delivering the event. The event will not pass on to the parent node and the rest of the listeners of the current node |
| `getCurrentTarget` | `Function` | get the target node that is currently receiving the event |
| `detail` | `Function` | custom event information (belongs to `cc.Event.EventCustom`) |
| `setUserData` | `Function` | set custom event information (belongs to `cc.Event.EventCustom`) |
| `getUserData` | `Function` | get custom event information (belongs to `cc.Event.EventCustom`) |

You can refer to the `cc.Event` and API files of its child category for a complete API list.

# II. Builtin Event

- **Node System Events**

As stated in the last file, `cc.Node` has a whole set of event listener and dispatch mechanisms. Based on this mechanism, we provide some basic system events. This part will introduce the system events related to Node hierarchy.

Cocos Creator supports four types of system events: **mouse**, **touch**, **keyboard**, **device motion**. This part will mainly discuss the usage of touch and mouse events which is dispatched by related `cc.Node`. For keyboard and device motion events, they are dispatched as Global System Events by `cc.systemEvent`.

System events follow the general register method, developers can register event listener not only by using the enumeration type but also by using the event name directly, the definition for the event name follows DOM event standards.

```
// Use enumeration type to register
node.on(cc.Node.EventType.MOUSE_DOWN, function (event) {
  console.log('Mouse down');
}, this);

// Use event name to register
node.on('mousedown', function (event) {
  console.log('Mouse down');
}, this);
```

### Mouse event type and event object

The Mouse event will only be triggered on desktop platforms, the event types the system provides are as follows:

| Enumeration object definition | Corresponding event name | Event trigger timing |
| --- | --- | --- |
| `cc.Node.EventType.MOUSE_DOWN` | `mousedown` | trigger once when mouse down |
| `cc.Node.EventType.MOUSE_ENTER` | `mouseenter` | when the mouse enters the target node region, regardless if it is down |
| `cc.Node.EventType.MOUSE_MOVE` | `mousemove` | when the mouse moves in the target node region, regardless if it is down |
| `cc.Node.EventType.MOUSE_LEAVE` | `mouseleave` | when the mouse leaves the target node region, regardless if it is down |
| `cc.Node.EventType.MOUSE_UP` | `mouseup` | trigger once when the mouse is released from the down state |
| `cc.Node.EventType.MOUSE_WHEEL` | `mousewheel` | when the mouse wheel rolls |

The important APIs of mouse events (`cc.Event.EventMouse`) are as follows (`cc.Event` standard events API excluded):

| Function name | Returned value type | Meaning |
| --- | --- | --- |
| `getScrollY` | `Number` | get the y axis distance wheel scrolled, effective only when scrolling |
| `getLocation` | `Object` | get mouse location object which includes x and y properties |
| `getLocationX` | `Number` | get X axis location of the mouse |
| `getLocationY` | `Number` | get Y axis location of the mouse |
| `getDelta` | `Object` | get the distance object the mouse moves since last event, which includes x and y properties |
| `getButton` | `Number` | `cc.Event.EventMouse.BUTTON_LEFT` or `cc.Event.EventMouse.BUTTON_RIGHT` or `cc.Event.EventMouse.BUTTON_MIDDLE` |

### Touch event types and event objects

Touch event can be triggered in both mobile platforms and desktop platforms. This is designed to serve developers to debug on desktop platforms better, all you need to do is listen to touch events, touch events from mobile platforms and mouse events from desktop can be responded to at the same time. Touch event types that the system provides are as follows:

| Enumeration object definition | Corresponding event name | Event trigger timing |
| --- | --- | --- |
| `cc.Node.EventType.TOUCH_START` | 'touchstart' | when the finger touches the screen |
| `cc.Node.EventType.TOUCH_MOVE` | 'touchmove' | when the finger moves in the target node region on screen |
| `cc.Node.EventType.TOUCH_END` | 'touchend' | when the finger leaves screen in the target node region |
| `cc.Node.EventType.TOUCH_CANCEL` | 'touchcancel' | when the finger leaves screen outside the target node region |

The important APIs of touch event (`cc.Event.EventTouch`) are as follows (`cc.Event` standard event API excluded):

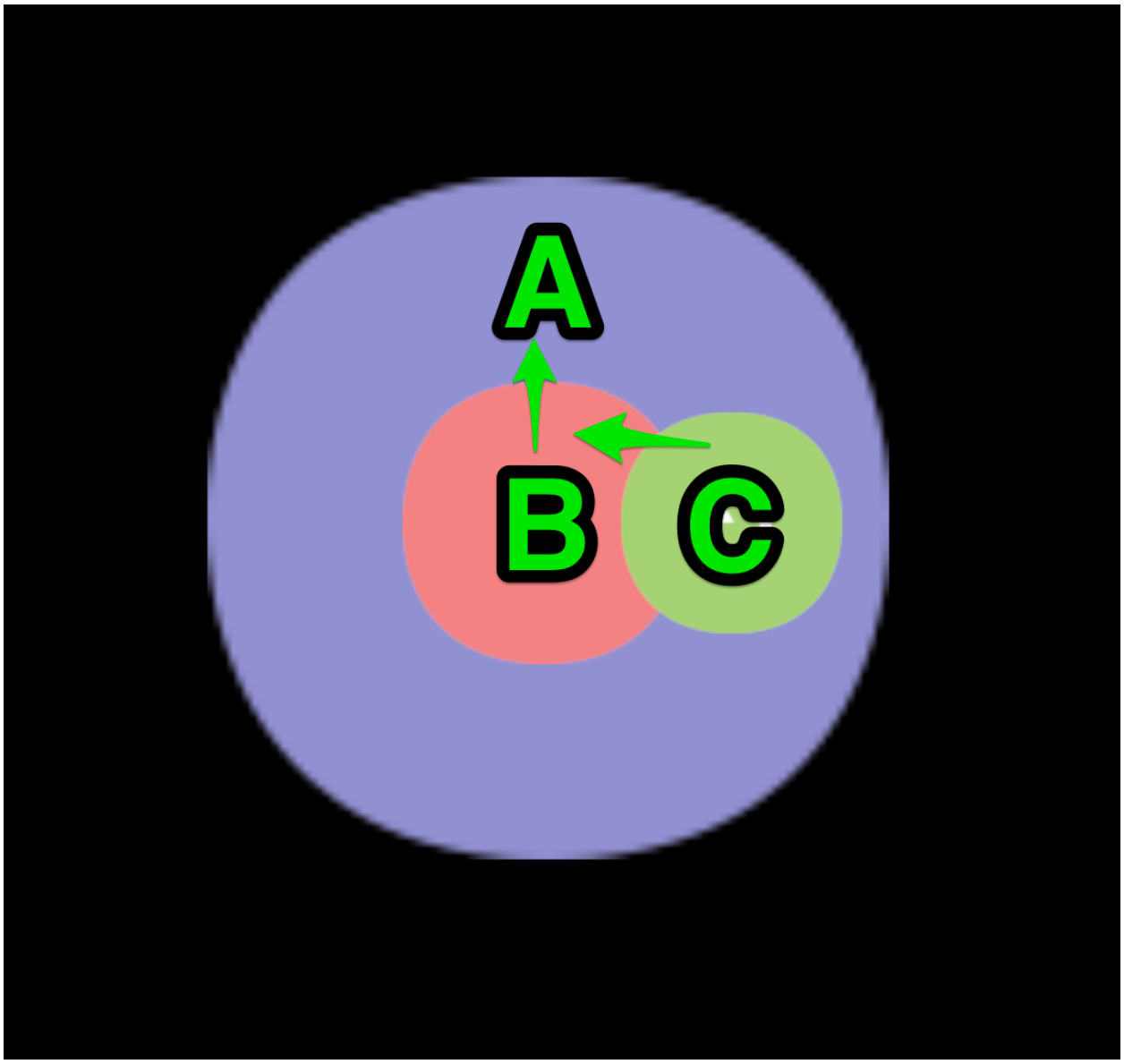
| API name | Type | Meaning |
| --- | --- | --- |
| `touch` | `cc.Touch` | contact object related to the current event |

Note, touch events support multi-touch, each touch spot will send one event to the event listener. You can get all the information of touch spot from `cc.Touch` API:

| Function name | Return value type | Meaning |
| --- | --- | --- |
| `getID` | `Number` | identification ID of the touch spot, can be used in multi-touch to track the touch spot |
| `getLocation` | `Object` | get location object of the touch spot which includes x and y properties |
| `getLocationX` | `Number` | get X axis location of the touch spot |
| `getLocationY` | `Number` | get Y axis location of the touch spot |
| `getDelta` | `Object` | get the distance object the touch spot moves since the last event, which includes x and y properties |
| `getStartLocation` | `Object` | get the location object the where touch spot gets down which includes x and y properties |
| `getPreviousLocation` | `Object` | get the location object of the touch spot at the last event which includes x and y properties |

### Touch event propagation

Touch events support the event bubbles on the node tree, take the pictures below as an example:




In the scene shown in the picture, suppose node A has a child node B which has a child node C. The developer set the touch event listeners for all these three nodes (each node has a touch event listener in examples below by default).

When the mouse or finger was applied in the node C region, the event will be triggered at node C first and the node C listener will receive the event. Then the node C will pass this event to its parent node, so the node B listener will receive this event. Similarly the node B will also pass the event to its parent node A. This is a basic event bubbling phase. It needs to be emphasized

that there is no hit test in parent nodes in the bubbling phase, which means that the node A and B can receive touch events even though the touch location is out of their node region.

The bubbling phase of touch events is no different than the general events. So, calling `event.stopPropagation()` can force to stop the bubbling phase.

#### Ownership of touch points among brother nodes

Suppose the node B and C in the picture above are brother nodes, while C partly covers over B. Now if C receives a touch event, it is announced that the touch point belongs to C, which means that the brother node B won't receive the touch event any more, even though the touch location is also inside its node region. The touch point belongs to the top one among brother nodes.

At the same time, if C has a parent node, it will also pass the touch event to its parent node through the event bubble mechanism.

#### Register touch or mouse events in the capturing phase

Sometimes we need to dispatch the touch or mouse events to parent node event listeners before dispatching to any child nodes beneath it in hierarchy, like the design of CCScrollView component.

Now the event bubbling can't meet our demand, so that we need to register the parent node event listeners in the capturing phase.

To achieve this goal, we can pass the fourth parameter `true` when registering touch or mouse event on node, which means `useCapture`. For example:

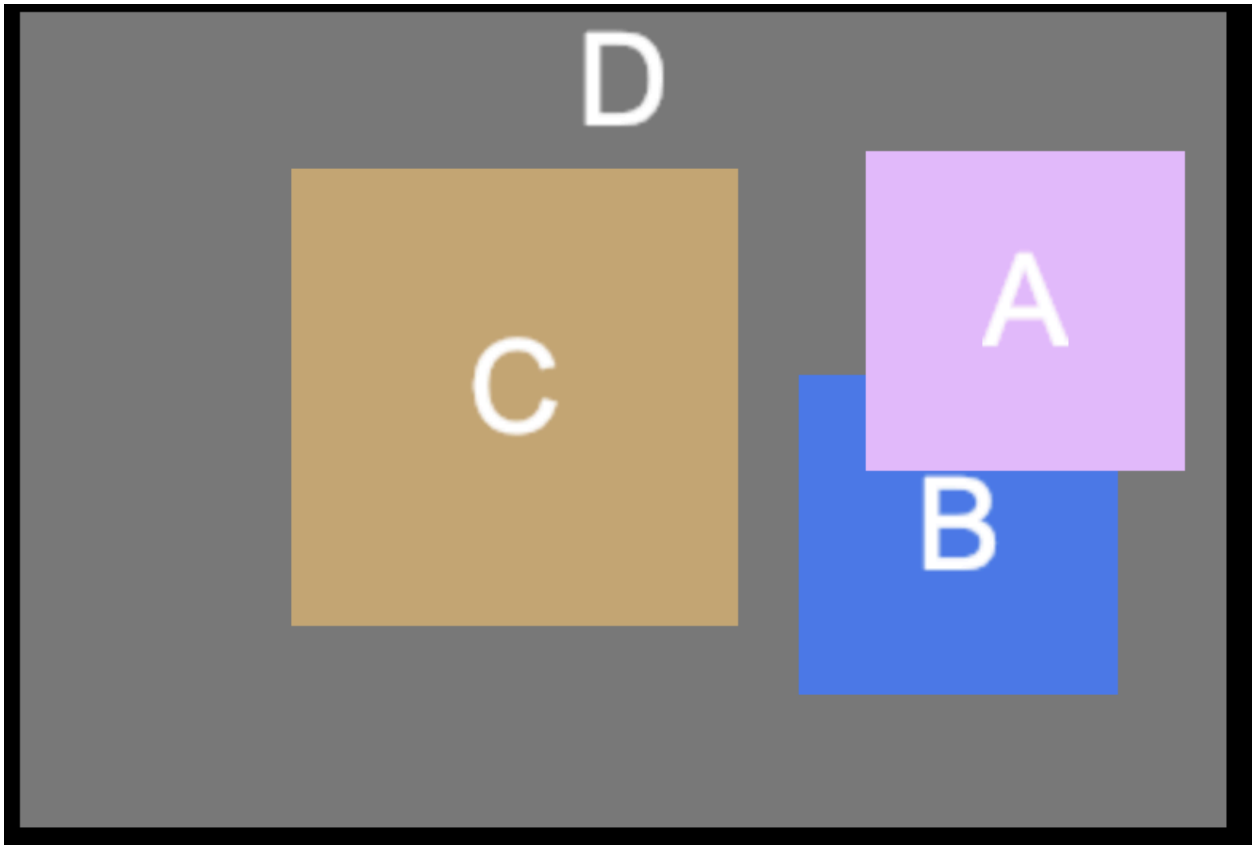
```
this.node.on(cc.Node.EventType.TOUCH_START, this.onTouchStartCallback, this, true);
```

When node fires `touchstart` event, the `touchstart` event will be firstly dispatched to all the parent node event listeners registered in the capturing phase, then dispatched to the node itself, and finally comes the event bubbling phase.

Only touch or mouse events can be registered in the capturing phase, while the other events can't be.

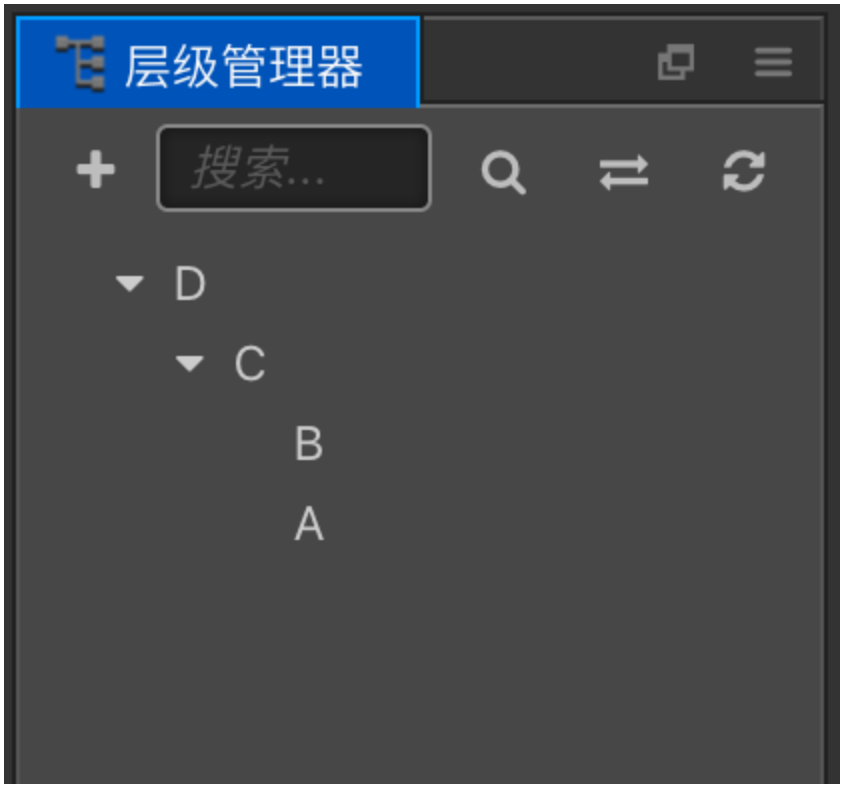
**Example for touch events**

Let's make a summary of touch event propagation with the example below:




There are four nodes A, B, C and D in the picture above, where A and B are brother nodes.

The specific hierarchical relationship should be like this:




1. If one touch is applied in the overlapping area between A and B, now B won't receive the touch event, so that propagating order of the touch event should be **A -> C -> D**
2. If the touch location is in node B ( the visible blue area ), the order should be **B -> C -> D**
3. If the touch location is in node C, the order should be **C -> D**
4. As a precondition to the second case, we register touch events on C D node in the capturing phase, then the order should be **D -> C -> B**

Other events of `cc.Node`

| Enumeration object definition | Corresponding event name | Event trigger timing |
| --- | --- | --- |
| null | `position-changed` | when the location property is changed |
| null | `rotation-changed` | when the rotation property is changed |
| null | `scale-changed` | when the scale property is changed |
| null | `size-changed` | when the size property is changed |
| null | `anchor-changed` | when the anchor property is changed |

- **Global System Events**

In this section, we will introduce the global system events of Cocos Creator.

Global system events are irrelevant with the node hierarchy, so they are dispatched globally by `cc.systemEvent`, currently supported:

- Keyboard
- DeviceMotion

If you are searching for any information about touch or mouse events, please refer to Node System Events documentation.

#### How to define the input events

You can use `cc.systemEvent.on(type, callback, target)` to register Keyboard and DeviceMotion event listeners.

Event types included:

1. cc.SystemEvent.EventType.KEY_DOWN

2. cc.SystemEvent.EventType.KEY_UP
3. cc.SystemEvent.EventType.DEVICEMOTION

Keyboard Events

- Type: `cc.SystemEvent.EventType.KEY_DOWN` and `cc.SystemEvent.EventType.KEY_UP`
- Call Back: Custom Event: `callback(event);`
- Call Back Parameter:
    - KeyCode: API Reference
    - Event: API Reference

```
cc.Class({
    extends: cc.Component,
    onLoad: function () {
        // add key down and key up event
        cc.systemEvent.on(cc.SystemEvent.EventType.KEY_DOWN, this.onKeyDown, this);
        cc.systemEvent.on(cc.SystemEvent.EventType.KEY_UP, this.onKeyUp, this);
    },

    destroy () {
        cc.systemEvent.off(cc.SystemEvent.EventType.KEY_DOWN, this.onKeyDown, this);
        cc.systemEvent.off(cc.SystemEvent.EventType.KEY_UP, this.onKeyUp, this);
    },

    onKeyDown: function (event) {
        switch(event.keyCode) {
            case cc.macro.KEY.a:
                console.log('Press a key');
                break;
        }
    },

    onKeyUp: function (event) {
        switch(event.keyCode) {
            case cc.macro.KEY.a:
                console.log('release a key');
                break;
        }
    }
});
```

**DEVICE MOTION**

- Type: `cc.SystemEvent.EventType.DEVICEMOTION`
- Call Back: Custom Event: callback(event);
- Call Back Parameter:
    - Event: API reference

```
cc.Class({
    extends: cc.Component,
    onLoad () {
        // open Accelerometer
        cc.systemEvent.setAccelerometerEnabled(true);
                                cc.systemEvent.on(cc.SystemEvent.EventType.DEVICEMOTION,
this.onDeviceMotionEvent, this);
    },

    destroy () {
                               cc.systemEvent.off(cc.SystemEvent.EventType.DEVICEMOTION,
this.onDeviceMotionEvent, this);
    },

    onDeviceMotionEvent (event) {
        cc.log(event.acc.x + "    " + event.acc.y);
    },
});
```

# III. Event Emitter

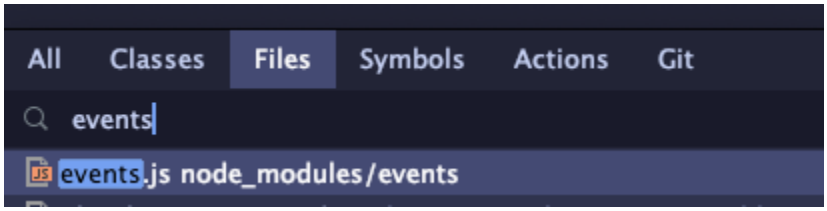
Generally, we will find it difficult to use event system between nodes to nodes, components to components if they do not have the same parent node, or are brother nodes. So, I will introduce you about Event Emitter of NodeJS

## 1. Installation Node Modules

Open **terminal** in root directory of project, and then type `npm init` and then follow instruction (or just simply enter till end)

```
package name: (bunnyproject)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/duytran/Documents/cocos-training/BunnyProject/package.json:

{
  "name": "bunnyproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) yes
```

After initializing nodeJS packages to our project, type `npm install events` to install the `'events'` module. Voila, events module is ready to use. You can find `events.js` in node_modules/events



## 2. Create a simple Singleton Event Emitter

We may need to create an instance of Event Emitter to use it across all components. Here's code example:

```
const EventEmitter = require('events');
class mEmitter {
   constructor() {
       this._emiter = new EventEmitter();
       this._emiter.setMaxListeners(100);
   }

   emit(...args)
   {
       this._emiter.emit(...args);
   }

   registerEvent(event, listener) {
       this._emiter.on(event, listener);
   }
   registerOnce(event, listener){
       this._emiter.once(event, listener);
   }

   removeEvent(event, listener) {
       this._emiter.removeListener(event, listener);
   }

   destroy()
   {
       this._emiter.removeAllListeners();
       this._emiter = null;
       mEmitter.instance = null;
   }
}
mEmitter.instance = null;
module.exports = mEmitter;
```

### 3. Practice time!

First, we will create controlling script for main node, and create instance of mEmitter, and listen to event ‘HELLO’ permanently, and listen to event ‘WELCOME’ once:

```
const Emitter = require('mEmitter');
cc.Class({
    extends: cc.Component,

    properties: {
    },

    onLoad () {
        Emitter.instance = new Emitter();
        Emitter.instance.registerEvent("HELLO", this.onHello.bind(this));
        Emitter.instance.registerOnce("WELCOME", this.onWelcome.bind(this));


    },
    onHello(data){
        cc.log('hello', data);
    },
    onWelcome(data){
        cc.log('welcome', data);
    },

    start () {
    },

});
```

Let say we create another child node, with 2 button HELLO and WELCOME, that will emit related events from function clickHello and clickWelcome:

```
const Emitter = require('mEmitter');

cc.Class({
    extends: cc.Component,

    start () {

    },
    onHello(){
        Emitter.instance.emit('HELLO', "hellooooooo");
    },
    onWelcome(){
        Emitter.instance.emit('HELLO', "Welcomeeeee");
    },
   });
```

Try bind click events and run code yourself!

## Summary of this chapter

Event emitters and listeners are crucial to NodeJS development, and many other programming languages development. They are very useful when you have some function that needs to execute “whenever this other thing happens”, without requiring that function to finish or even work for that matter.

# Chapter 7: Cocos Creator's Animation System

In the early game development, game engineers and designers used simple techniques to develop animations. The animations of early era were produced by continuously and quickly displaying a series of sequential pictures. These sequential pictures are called frames. This kind of animation is called frame by frame (fbf) animation. With the improvement of equipment hardware technology and the continuous improvement of development tools, more advanced animation skills and skeletal animation have been used. Cocos Creator includes a simple animation editing system that can make simple animations. In addition, skeletal animation is widely used in 2D games. The skeleton animation tools supported by Cocos Creator includes **Spine** and **DragonBones**. **Spine** and **DragonBones** can make more complex animations and special effects. This chapter will introduce the animation system of Cocos Creator. In addition to basic functions and animations, a game also needs to give players more visual excitement. At this time, it needs to be decorated with special effects. For example, the famous fighting game ***The King of Fighters*** series often accompanied by a large number of special effects when the characters move, making the game's screen more dazzling and more attractive to the players, enhancing the player's sense of substitution and the expressive power of the game.

## I. Cocos Creator’s Animation System

### 1. Action System

The action system that Cocos Creator provides originates from Cocos2d-x with both the API and usage having been inherited. The action system can complete displacement, zoom, rotate and all the other actions of the node within a designated time.

What needs to be noted is that the action system cannot replace the animation system. What the action system provides is an API interface for programmers, while the animation system is designed within the editor. Meanwhile, they serve different usages — the action system is more suitable for making simple deformation and displacement animation while the animation system is much more powerful, you can make animation supporting all kinds of properties with the editor, including motion track and complicated animation in slow motion.

The action system is easy to use, supporting the following API in `cc.Node`:

```
// create a moving action
var action = cc.moveTo(2, 100, 100);
// execute the action
node.runAction(action);
// stop one action
node.stopAction(action);
// stop all actions
node.stopAllActions();
```

Cocos Creator supports various kinds of actions which can be divided into following several categories.

### Basic action

Basic action is the action to achieve all kinds of deformation and displacement animation, for example using `cc.moveTo` to move the node to a certain location; using `cc.rotateBy` to rotate the node by a certain angle; using `cc.scaleTo` to zoom in and out on the node.

Basic action can be divided into interval action and free action. Interval action is a gradual change action that is done in a certain time interval. The actions mentioned above are all interval actions which are inherited from `cc.ActionInterval`. Whereas free actions happen immediately and are all inherited from `cc.ActionInstant`, for instance, `cc.callFunc` is used to call the callback function; `cc.hide` is used to hide the node.

### Container action

The container action can organize actions in different ways, below are several of the container action's usages:

1. Sequential action `cc.sequence` Sequential action makes a series of child actions run one by one in sequence. For example:

```
// make the node move back and forth
var seq = cc.sequence(cc.moveBy(0.5, 200, 0), cc.moveBy(0.5, -200, 0));
node.runAction(seq);
```

2. Synchronization action `cc.spawn` Synchronization action synchronises the execution of a series of child actions. The result of the execution of these child actions will gather together to alter the properties of the node. For example:

```
// make the node zoom while it moves upwards
var spawn = cc.spawn(cc.moveBy(0.5, 0, 50), cc.scaleTo(0.5, 0.8, 1.4));
node.runAction(spawn);
```

3. Repetitive action `cc.repeat` Repetitive action is used to repeat one action several times. For example:

```
// make the node move back and forth 5 times
var seq = cc.repeat(
            cc.sequence(
                cc.moveBy(2, 200, 0),
                cc.moveBy(2, -200, 0)
            ), 5);
node.runAction(seq);
```

4. Repeat forever action `cc.repeatForever` As its name, this action container can make the target action repeat forever until it is stopped manually.

```
// move the node back and forth and keep repeating
var seq = cc.repeatForever(
            cc.sequence(
                cc.moveBy(2, 200, 0),
                cc.moveBy(2, -200, 0)
            ));
```

5. Speed action `cc.speed` Speed action can alter the execution rate of the target action to make it quicker or slower.

```
// double the speed of the target action which means the action that took 2
seconds  before now can be done in 1 second
var action = cc.speed(
                cc.spawn(
                    cc.moveBy(2, 0, 50),
                    cc.scaleTo(2, 0.8, 1.4)
                ), 2);
node.runAction(action);
```

You can see from the above example, different container types can be combined. Besides, we provide a more convenient link-form API for the container type actions. The action objects support these three API - `repeat`, `repeatForever`, `speed` - which will return to the action object itself and support the continued link-form call. Let's see a more complicated action example:

```
// a complicated jump animation
this.jumpAction = cc.sequence(
    cc.spawn(
        cc.scaleTo(0.1, 0.8, 1.2),
        cc.moveTo(0.1, 0, 10)
    ),
    cc.spawn(
        cc.scaleTo(0.2, 1, 1),
        cc.moveTo(0.2, 0, 0)
    ),
    cc.delayTime(0.5),
    cc.spawn(
        cc.scaleTo(0.1, 1.2, 0.8),
```

```
        cc.moveTo(0.1, 0, -10)
    ),
    cc.spawn(
        cc.scaleTo(0.2, 1, 1),
        cc.moveTo(0.2, 0, 0)
    )
// play the animation at 1/2 speed and repeat 5 times
).speed(2).repeat(5);
```

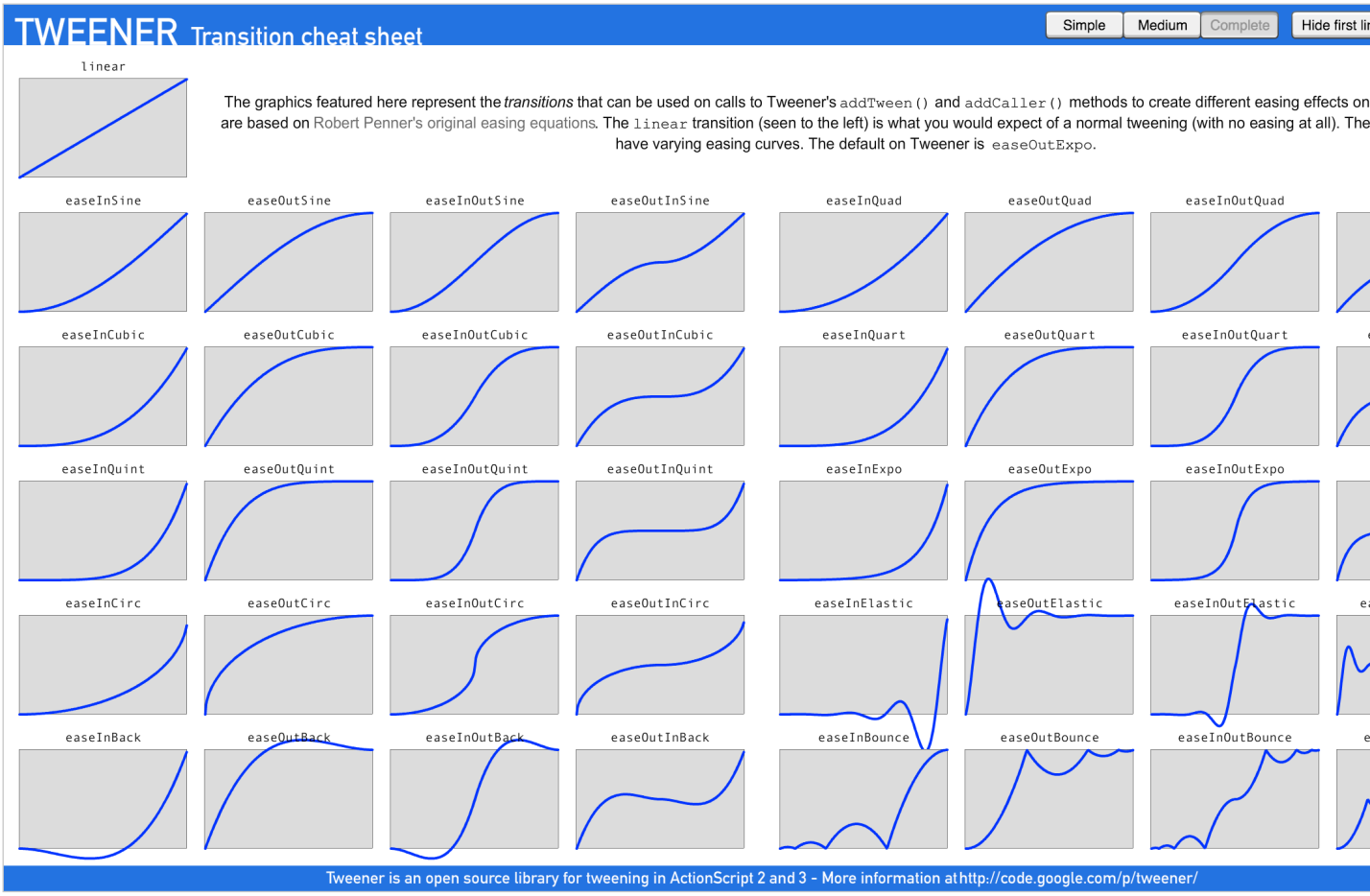
Note: In cc.callFunc should not stop its own action, because the action can not be immediately deleted, if the action in the callback pause its own action will lead to a series of traversal problems, leading to more serious bug.

### Slow motion

Slow motion cannot exist alone; it always exists to modify a basic action. It can be used to alter the time curve of the basic action to give the action fast in/out, ease in or other more complicated special effects. One thing we need to note is that only interval actions support slow motion:

```
var action = cc.scaleTo(0.5, 2, 2);
action.easing(cc.easeIn(3.0));
```

The basic slow motion category is `cc.ActionEase`. You can refer to the picture below for the time curves of different slow motions:



## 2. Tween

Cocos Creator provides a new set of APIs in v2.0.9 -- `cc.tween`. `cc.tween` can easing any property of an object, similar to the cc.Action. But `cc.tween` is much easier to use than `cc.Action`, because `cc.tween` provides a chain-created method that can manipulate any object, and easing any of the object's properties.

`cc.Action` is migrated from Cocos2d-x to Cocos Creator. Providing an API that is cumbersome, only supports use on the node properties, and requires adding a new action if you want to support new properties. In order to provide a better API, `cc.tween` on the basis of `cc.Action` made a layer of API encapsulation. Here's a comparison between `cc.Action` and `cc.tween` in use:

**cc.Action**:

```
this.node.runAction(
    cc.sequence(
        cc.spawn(
            cc.moveTo(1, 100, 100),
            cc.rotateTo(1, 360),
        ),
        cc.scale(1, 2)
    )
)
```

**cc.tween**:

```
cc.tween(this.node)
    .to(1, { position: cc.v2(100, 100), rotation: 360 })
    .to(1, { scale: 2 })
    .start()
```

### Chain APIs

Each API of `cc.tween` generates an action internally and adds this action to the internal queue. After the API is called, it returns its own instance, so that the code can be organized by chain call.

When `cc.tween` calls `start`, a cc.sequence queue is generated that combines the previously generated action queue. So the chain structure of `cc.tween` is to execute each API in turn, that is, it will execute an API and then execute the next API.

```
cc.tween(this.node)
    // at 0s, node's scale is still 1.
    .to(1, { scale: 2 })
    // at 1s, after executing the first action, scale is 2
    .to(1, { scale: 3 })
    // at 2s, after executing the second action, scale is 3
    .start()
    // Call start and start executing cc.tween
```

### Set the properties of cc.tween

`cc.tween` provides two APIs for setting properties:

- `to`: Calculate the absolute value of the property. And the final run result is the property value that is set.
- `by`: Calculate the relative value of the property. And the final run result is the property value that is set, plus the property value of the node at the start of the run.

```
cc.tween(node)
  .to(1, {scale: 2})      // node.scale === 2
  .by(1, {scale: 2})      // node.scale === 4 (2+2)
  .by(1, {scale: 1})      // node.scale === 5
  .to(1, {scale: 2})      // node.scale === 2
  .start()
```

### Support for easing any property of any object

```
let obj = { a: 0 }
cc.tween(obj)
  .to(1, { a: 100 })
  .start()
```

### Execute multiple properties simultaneously

```
cc.tween(this.node)
    // Easing three properties a, b, c at the same time
    .to(1, { scale: 2, position: cc.v2(100, 100), rotation: 90 })
    .start()
```

### easing

You can use `easing` to make the easing more vivid. `cc.tween` provides a variety of ways to use it for different situations.

```
// Pass in the easing name and use the built-in easing function directly
cc.tween().to(1, { scale: 2 }, { easing: 'sineOutIn'})

// Using custom easing functions
cc.tween().to(1, { scale: 2 }, { easing: t => t*t; })

// Use the easing function only for a single property
// value must be used with easing or progress
cc.tween().to(1, { scale: 2, position: { value: cc.v3(100, 100, 100), easing:
'sineOutIn' } })
```

### Custom progress

Compared to easing, custom progress function has more freedom to control the easing process.

```
// Customize the progress for all properties
cc.tween().to(1, { scale: 2, rotation: 90 }, {
  progress: (start, end, current, ratio) => {
    return start + (end - start) * ratio;
  }
})

// Customize the progress for a single property
cc.tween().to(1, {
  scale: 2,
```

```
  position: {
    value: cc.v3(),
    progress: (start, end, current, t) => {
      // Note that the passed in property is cc.Vec3, so you need to use Vec3.lerp
for interpolation calculations
      return start.lerp(end, t, current);
    }
  }
})
```

### Replication easing

The `clone` function will clone a current easing and accept a target as a parameter

```
// First create a easing as a template
let tween = cc.tween().to(4, { scale: 2 })

// Copy tween and use node Canvas/cocos as target
tween.clone(cc.find('Canvas/cocos')).start()
// Copy tween and use node Canvas/cocos2 as target
tween.clone(cc.find('Canvas/cocos2')).start()
```

### Insert other easing into the queue

You can create some fixed easing in advance, and then reduce the writing of your code by combining these easing to form a new easing.

```
let scale = cc.tween().to(1, { scale: 2 })
let rotate = cc.tween().to(1, { rotation: 90})
let move = cc.tween().to(1, { position: cc.v3(100, 100, 100)})

// Zoom first, then rotate
cc.tween(this.node).then(scale).then(rotate)
// Zoom first, then move
cc.tween(this.node).then(scale).then(move)
```

### Parallel execution easing

`cc.tween` is executed in the form of a sequence when it is executed in a chain. However, when writing complex easing, you may need to execute multiple queues in parallel at the same time. So `cc.tween` provides the parallel interface to meet this requirement.

```
let t = cc.tween;
t(this.node)
    // Execute two cc.tween at the same time
    .parallel(
        t().to(1, { scale: 2 }),
        t().to(2, { position: cc.v2(100, 100) })
    )
    .call(() => {
        console.log('All tweens finished.')
    })
    .start()
```

### Callback

```
cc.tween(this.node)
    .to(2, { rotation: 90})
    .to(1, { scale: 2})
    // This callback function is not called until the preceding action has been
performed
    .call(() => { cc.log('This is a callback') })
    .start()
```

### Repeat execution

The repeat/repeatForever function will use the previous action as the object of action. However, if there are parameters that provide additional action or tween, the repeat/repeatForever function will use the incoming action or tween as the object of action.

```
cc.tween(this.node)
    .by(1, { scale: 1 })
    // Repeat 10 times for the previous by
    .repeat(10)
```

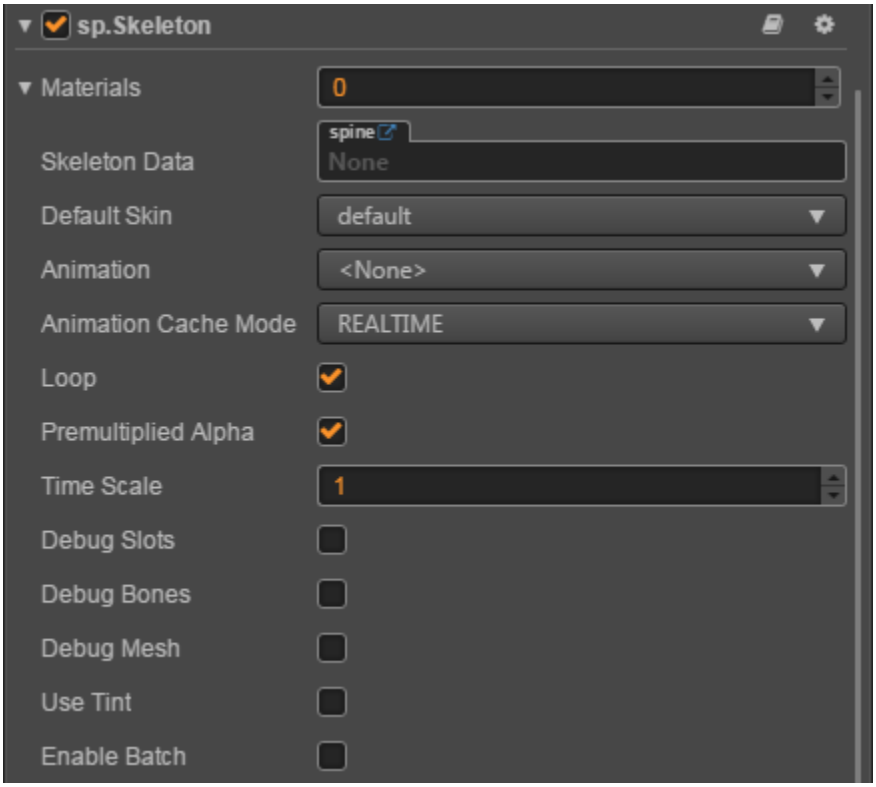
```
    // Finally node.scale === 11
    .start()

// Can also be used like this
cc.tween(this.node)
    .repeat(10,
        cc.tween().by(1, { scale: 1 })
    )
    .start()

// Keep it going over and over again.
cc.tween(this.node)
    .by(1, { scale: 1 })
    .repeatForever()
    .start()
```

### Delayed execution

```
cc.tween(this.node)
    // Delay 1s
    .delay(1)
    .to(1, { scale: 2 })
    // Another delay of 1s
    .delay(1)
    .to(1, { scale: 3 })
    .start()
```

# II. Spine Animation

The Spine component supports the data formats exported by Spine, and renders and plays Spine resources.




Click the **Add Component** at the bottom of **Properties** and select **Spine Skeleton** from **Renderer Component** in order to add the **Spine** component to the node.

About the Spine's scripting interface please refer to Skeleton API

**Spine Properties**

| Properties | Function explanation |
| --- | --- |
| Skeleton Data | The skeleton data contains the skeleton information, drag the bone resources exported from Spine into this property. |
| Default Skin | Choose the default skin texture |
| Animation | The name of current playing animation |
| Animation Cache Mode | Render mode, default is `REALTIME` mode. (new in v2.0.9)<br><br>1. **REALTIME** model, realtime calculate, support all functions of Spine.<br><br>2. **SHARED_CACHE** mode, caching and sharing animation data, the equivalent of pre baked skeletal animation, have high performance, does not support the action blend and superposition, only supports the start and end events, as for memory, when creating some same bones and the same action of animation, can present advantages of memory, the greater the amount of skeleton, the more obvious advantages, in conclusion `SHARED_CACHE` mode is suitable for the scene animation, special effects, monster, NPC and so on, can greatly increase the frame rate and reduce memory.<br><br>3. **PRIVATE_CACHE** mode, similar to `SHARED_CACHE`, but not share animation and texture data, so there is no advantage in memory, there is only a performance advantage, when trying to take advantage of caching pattern of high performance, but there is a change of texture, so you can't share the map data, then `PRIVATE_CACHE` is suitable for you. |
| Loop | Whether loop current animation |
| Premultiplied Alpha | Indicates whether to enable premultiplied alpha, default is True.<br><br>You should disable this option when the image's transparent area appears to have opaque pixels, or enable this option |

| | |
|---|---|
| | when the image's half transparent area appears to be darkened. |
| Time Scale | The time scale of animation of this skeleton |
| Debug Slots | Indicates whether show debug slots |
| Debug Bones | Indicates whether show debug bones |
| Debug Mesh | Indicates whether show debug mesh |
| Use Tint | Indicates whether open tint, default is close. (New in v2.0.9) |
| Enable Batch | Whether to enable animation batch, default is disabled. (New in v2.0.9)<br><br>When enabled, drawcall will reduce, which is suitable for a large number of simple animations to play at the same time.<br><br>When disabled, drawcall will rise, but it can reduce the computational burden of the CPU. Suitable for complex animations. |

**Note**: when using the Spine component, the `Anchor` and `Size` properties on the Node component in the **properties** panel are invalid.

## Summary of this chapter

In this Chapter, you have learned about Action and Tweening, which is the methods to detransform and modify transformable properties of Node, and know about skeletal animation called Spine.

# Chapter 8: Cocos Creator's collision System

The main purpose of the collision system in the game engine is to determine whether objects in the game world are in contact. The logical objects of each game are represented by one or more geometric shapes. These graphics are usually relatively simple, such as rectangles, circles, and polygons. You can also edit more complex shapes by yourself. The collision system judges that these are at a certain time, whether the graphics intersect or overlap, so in fact, the physical collision system can also be called a geometric intersection detector. For simpler physics requirements, it is recommended that users directly use the collision component, which can avoid the runtime overhead of loading the physics engine and building the physics world. The physics engine provides a more complete interactive interface and already preset components such as rigid bodies and joints. You can choose the physical system that suits you according to your needs.

## I. Cocos Creator's Collision System

We will introduce the Collision System in Cocos Creator in this chapter. Cocos Creator provides a simple and easy to use built-in **Collision Detection System**, which supports **Circle**, **Box** and **Polygon** Collision Detection.

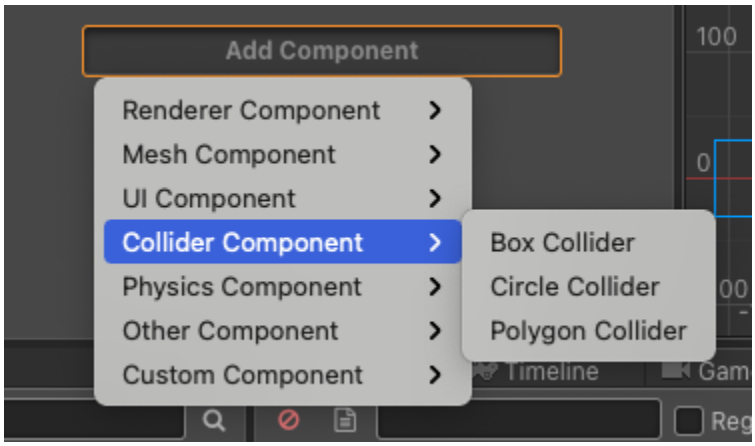
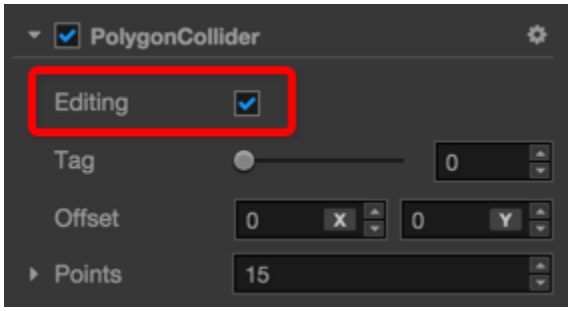



**Figure 8-1: Adding Collider Component**

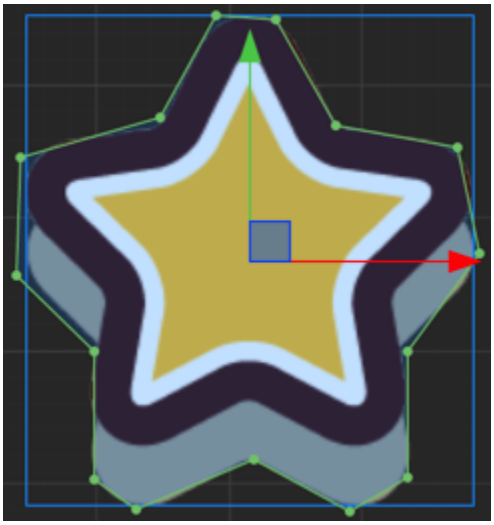
# II. Editing Collision Components

You can click **Editing** checkbox of a collider component to edit the collider shape freely.



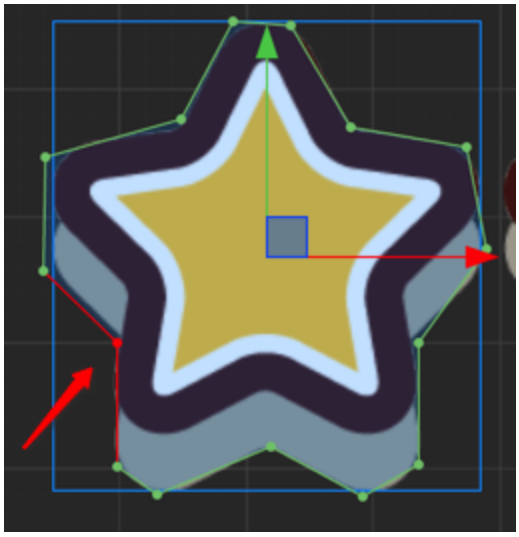

### Polygon Collider

For editing **Polygon Collider** all points of the collider can be moved freely by dragging. All changes to the points can be seen in **Points** property of Polygon Collider.



If you move the mouse over the line between two points, the mouse pointer changes to **Add** style. Then clicking the mouse to add a new point to the Polygon Collider.

If you move the mouse over a point while holding **Ctrl** or **Command** key, you'll find the point and two lines connecting to it become red. Click the mouse to remove the point from this Polygon Collider.



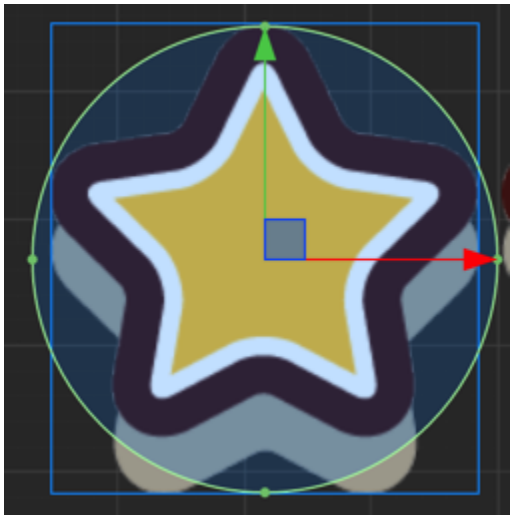
In CocosCreator v1.5, **Regenerate Points** have been added to the Polygon Collider component. It can automatically generate the points of the corresponding outline based on the texture pixels of the Sprite component on the node which the component is attached to.

**Threshold** indicates the minimum distance between the points of the generated texture outline. The larger the value, the fewer points are generated, which can be adjusted according to the requirements.


### Circle Collider

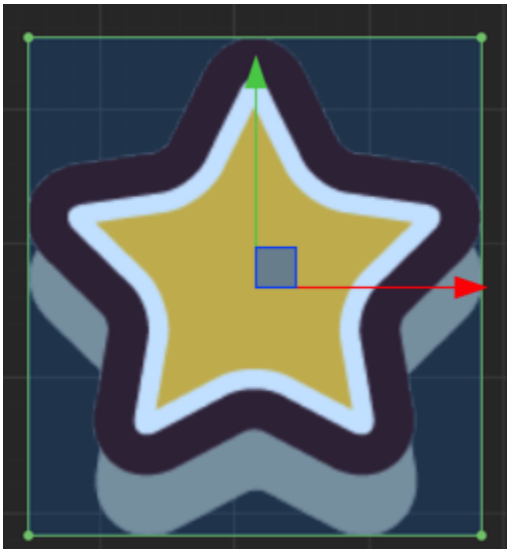
Enable editing for a **Circle Collider** will show the circle editing area like below:

Move the mouse over the circle edge and drag will modify the radius of Circle Collider.



### Box Collider

Enable editing for a **Box Collider** will show the box editing area like below:



When the mouse is hovering over the points of the box collider, clicking the left mouse button and drag to modify the length and width of the box collider component.

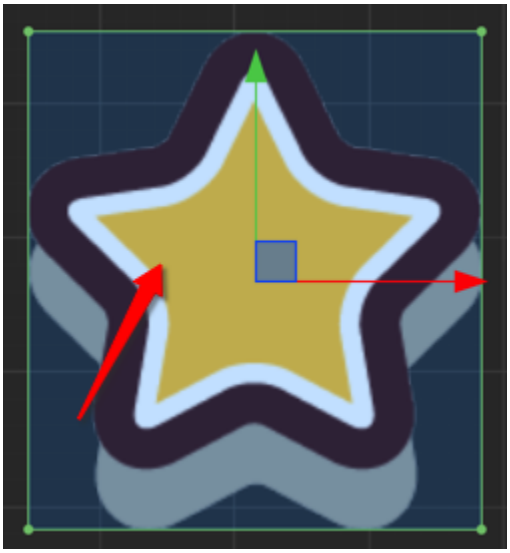
When the mouse is hovering over the edge line of the box collider, clicking the left mouse button will modify one of the length or width of the box collider component.

Drag while holding **Shift** to keep the aspect ratio of the box.

Drag while holding **Alt** to keep the center of the box unchanged.

### Change the Collider Offset

In the editing mode of all kinds of Colliders you can drag the center of the collider to move it off the center of the node. The **Offset** property of the collider will change as well.



# III. Collision Group Management

### Collision Group Management

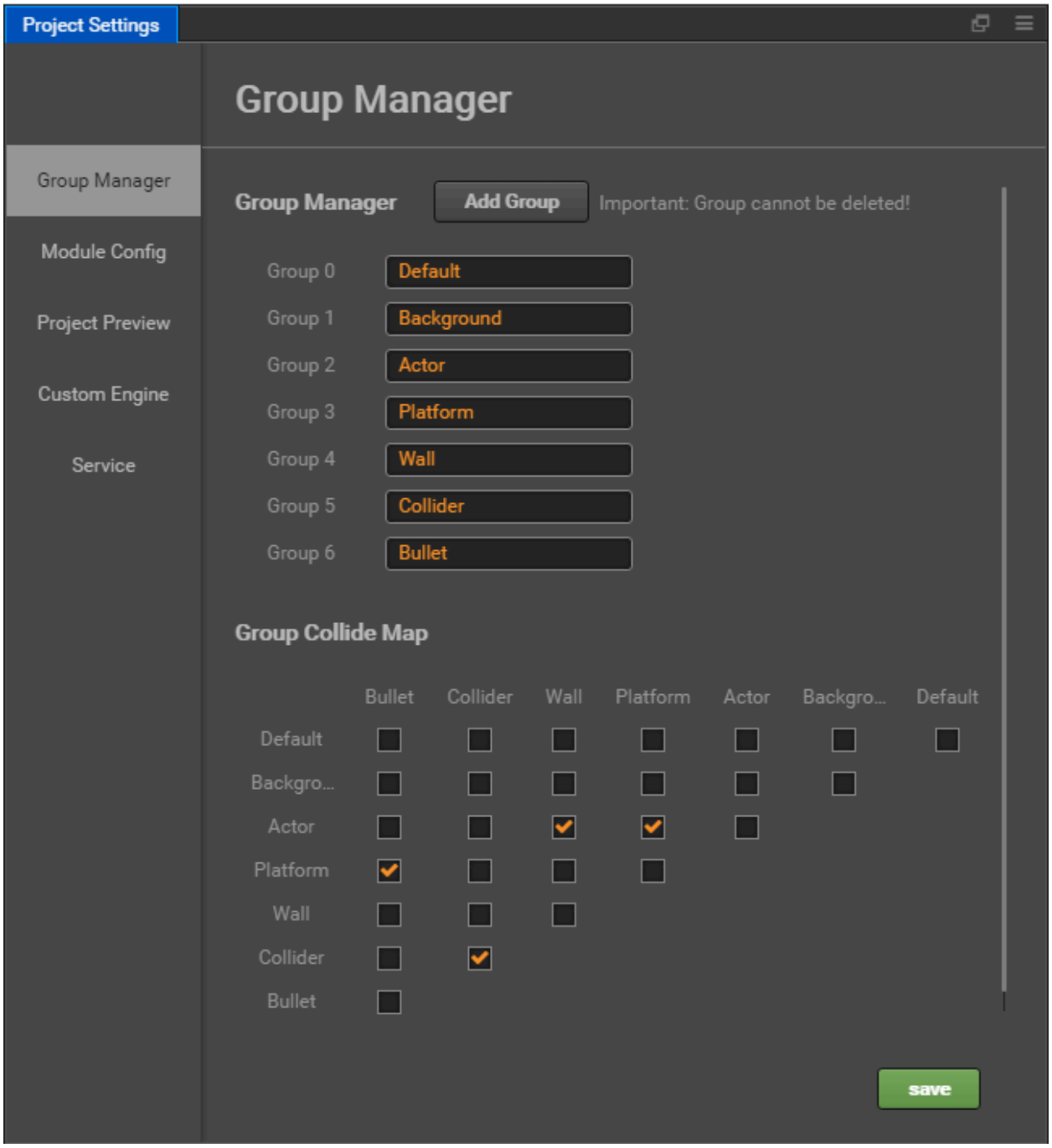
Ideally we want to control which collides with which, to reduce unnecessary collision callbacks (for both performance and gameplay). We can control the collision relationship with **Group Manager** panel.

This panel has two sections:

### Group Management

We use node groups to categorize colliders. Set a group for a node will mark all colliders on this node with this specific group.

Let's select the main menu's **Project -> Project Settings...** and open **Project Settings** panel. Then you can see the configuration items of the group list in the **Group Manager**, as shown below:

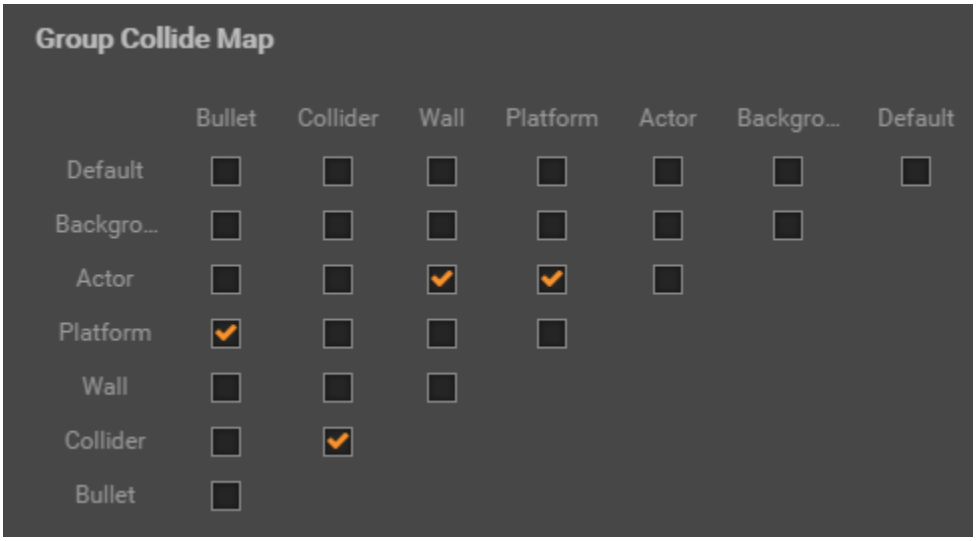



In the **Group Manager** section we can add new groups by click **Add Group** button, and there will be a **Default** group by default.

**Notice**: After the group is added, it cannot be deleted, but you can arbitrarily modify the name of the group.

### Group Collide Map

In the **Group Collide Map** section we can control whether collide is allowed for each group with any other groups. The Collide map looks like this:




Each column and row of this table has all the group listed. If you make modification to the **Group List** this table will be updated accordingly. Each checkbox in the table represents whether the group from the column will collide with the group from the row.

After the runtime modifies the node's group, it is necessary to invoke collider apply for the modification to take effect.

So as the checkboxes stated, we have following groups that can collide with each other:

Platform - Bullet
Collider - Collider
Actor - Wall
Actor - Platform

# IV. Using Scripts to Process Collision System

Cocos Creator provides a simple and easy way to use built-in Collision Detection System, it will do collision detection according to the added colliders. When a Collider Component is enabled, this Collider Component will be auto added to Collision Detection System, and Collision Manager will search other Collider Components which can generate a Collision Pair with it.

**Note: The Collider Components in the same node, will not do collision detection with each other forever.**

### Usage of the Collision Manager

### Collision Manager Interface

Collision Manager is disabled by default, if you need to use it, you must enable it. Optionally, you can also enable the debug draw if you need to see the debug info :

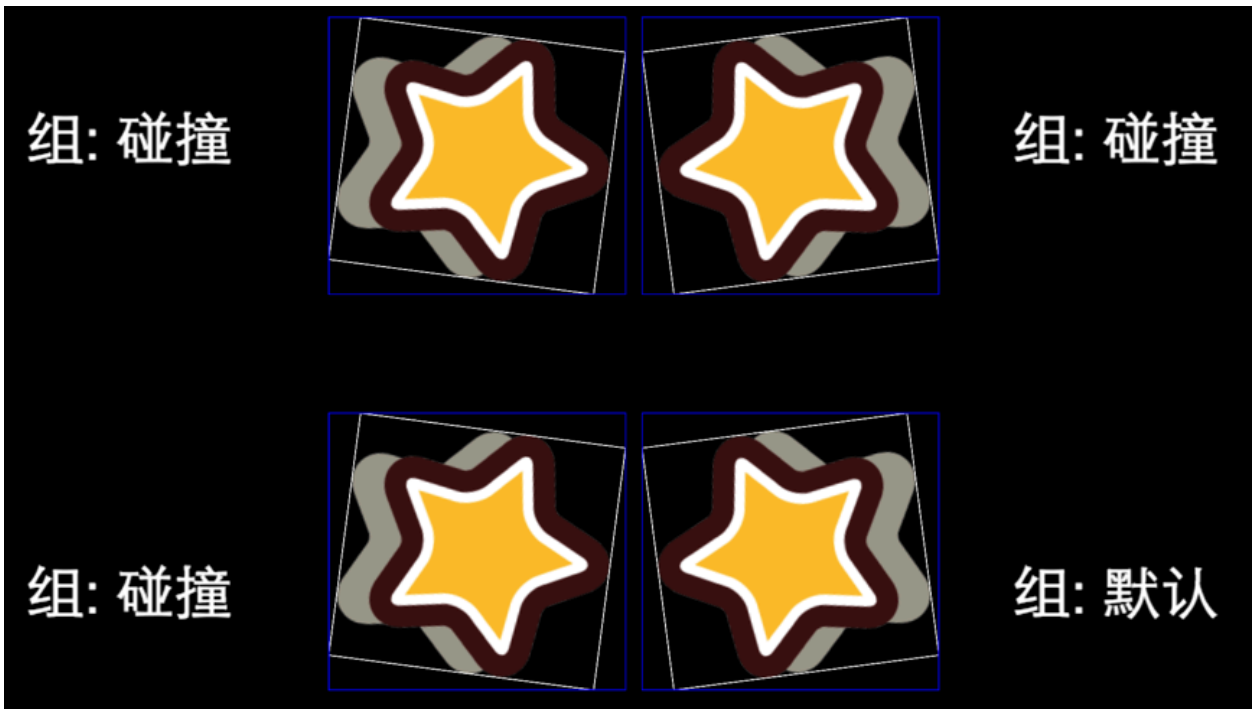
```
let manager = cc.director.getCollisionManager();
manager.enabled = true;
manager.enabledDebugDraw = true;
```

After enable **Debug Draw**, you can see the Collider area in runtime:

If you want to display the bounding box of the Collider Component, please set `enabledDrawBoundingBox` to `true`:

```
manager.enabledDrawBoundingBox = true;
```

Result:



### Collision Manager Callback

When a collision is detected, Collision Manager will call the callback to notify users. If the script which belongs to the same node with the Collider Component implements below functions, Collision will call the functions as the callback.

```
/**
 * Call when a collision is detected
 * @param  {Collider} other The other Collider Component
 * @param  {Collider} self  Self Collider Component
 */
onCollisionEnter: function (other, self) {
    console.log('on collision enter');

    // Collider Manager will calculate the value in world coordinate system, and
put them into the world property
    var world = self.world;
    // Collider Component aabb bounding box
    var aabb = world.aabb;
    // The position of the aabb collision frame before the node collision
    var preAabb = world.preAabb;
    // world transform
    var t = world.transform;
    // Circle Collider Component world properties
    var r = world.radius;
    var p = world.position;
    // Rect and Polygon Collider Component world properties
    var ps = world.points;
},

/**
 * Call after enter collision, before end collision, and after every time calculate
the collision result.
 * @param  {Collider} other The other Collider Component
 * @param  {Collider} self  Self Collider Component
 */
onCollisionStay: function (other, self) {
    console.log('on collision stay');
},
/**
 * Call after end collision
 * @param  {Collider} other The other Collider Component
 * @param  {Collider} self  Self Collider Component
 */
onCollisionExit: function (other, self) {
    console.log('on collision exit');
}
```

**Click**

```
properties: {
    collider: cc.BoxCollider
},

start () {
    // Open the collision manager, without this part statement you will not detect
any collision.
    cc.director.getCollisionManager().enabled = true;
    // cc.director.getCollisionManager().enabledDebugDraw = true;

    this.collider.node.on(cc.Node.EventType.TOUCH_START, function (touch, event) {
        // return the touch point with world coordinates
        let touchLoc = touch.getLocation();
        // https://docs.cocos.com/creator/api/en/classes/Intersection.html
Intersection
        if (cc.Intersection.pointInPolygon(touchLoc, this.collider.world.points)) {
            console.log("Hit!");
        }
        else {
            console.log("No hit");
        }
    }, this);
}
```

# Summary of this chapter

In this chapter you have learned about Collider Component, the simplest and easiest way to detect collision. The physics engine provides a more complete interactive interface and already preset components such as rigid bodies and joints, but can cause the runtime overhead of loading the physics engine and building the physics world.

# Chapter 9: Commonly used design patterns in Game Developing

*Design patterns are advanced object-oriented solutions to commonly occurring software problems. Patterns are about reusable designs and interactions of objects. Each pattern has a name and becomes part of a vocabulary when discussing complex design solutions.*

*In this Chapter we provide JavaScript examples for 6 commonly used design patterns in Game Developing:* ***Command****,* ***Flyweight****,* ***Observer****,* ***State****,* ***Singleton****. Mostly, they follow the structure and intent of the original pattern designs. These examples demonstrate the principles behind each pattern, but are not optimized for JavaScript.*

## I. Command Pattern

### 1. Summary

The Command pattern encapsulates actions as objects. Command objects allow for loosely coupled systems by separating the objects that issue a request from the objects that actually process the request. These requests are called events and the code that processes the requests are called event handlers.

Suppose you are building an application that supports the Cut, Copy, and Paste clipboard actions. These actions can be triggered in different ways throughout the app: by a menu system, a context menu (e.g. by right clicking on a textbox), or by a keyboard shortcut.
Command objects allow you to centralize the processing of these actions, one for each operation. So, you need only one Command for processing all Cut requests, one for all Copy requests, and one for all Paste requests.

Because commands centralize all processing, they are also frequently involved in handling Undo functionality for the entire application.

### 2. Diagram

### 3. Participants

The objects participating in this pattern are:

- **Client** -- In sample code: the `run()` function
  - references the Receiver object
- **Receiver** -- In sample code: **Calculator**
  - knows how to carry out the operation associated with the command
  - (optionally) maintains a history of executed commands
- **Command** -- In sample code: **Command**
  - maintains information about the action to be taken
- **Invoker** -- In our sample code: the user pushing the buttons
  - asks to carry out the request

### 4. Sample code in JavaScript

In our example we have a calculator with 4 basic operations: add, subtract, multiply and divide. Each operation is encapsulated by a Command object.
The Calculator maintains a stack of commands. Each new command is executed and pushed onto the stack. When an undo request arrives, it simply pops the last command from the stack and executes the reverse action.
The log function is a helper which collects and displays results.

```
function add(x, y) { return x + y; }
function sub(x, y) { return x - y; }
function mul(x, y) { return x * y; }
function div(x, y) { return x / y; }

var Command = function (execute, undo, value) {
   this.execute = execute;
   this.undo = undo;
   this.value = value;
}

var AddCommand = function (value) {
   return new Command(add, sub, value);
};

var SubCommand = function (value) {
   return new Command(sub, add, value);
};

var MulCommand = function (value) {
   return new Command(mul, div, value);
};

var DivCommand = function (value) {
   return new Command(div, mul, value);
};

var Calculator = function () {
```

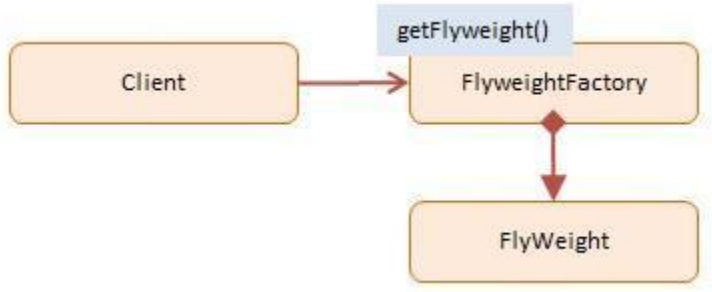
```
    var current = 0;
    var commands = [];

    function action(command) {
        var name = command.execute.toString().substr(9, 3);
        return name.charAt(0).toUpperCase() + name.slice(1);
    }

    return {
        execute: function (command) {
            current = command.execute(current, command.value);
            commands.push(command);
            log.add(action(command) + ": " + command.value);
        },

        undo: function () {
            var command = commands.pop();
            current = command.undo(current, command.value);
            log.add("Undo " + action(command) + ": " + command.value);
        },

        getCurrentValue: function () {
            return current;
        }
    }
}

// log helper

var log = (function () {
    var log = "";

    return {
        add: function (msg) { log += msg + "\n"; },
        show: function () { alert(log); log = ""; }
    }
})();

function run() {
    var calculator = new Calculator();
    // issue commands
    calculator.execute(new AddCommand(100));
    calculator.execute(new SubCommand(24));
    calculator.execute(new MulCommand(6));
    calculator.execute(new DivCommand(2));
    // reverse last two commands
    calculator.undo();
    calculator.undo();
    log.add("\nValue: " + calculator.getCurrentValue());
    log.show();
}
```

# II. Flyweight Pattern

## 1. Summary

The Flyweight pattern conserves memory by sharing large numbers of fine-grained objects efficiently. Shared flyweight objects are immutable, that is, they cannot be changed as they represent the characteristics that are shared with other objects.
Essentially Flyweight is an 'object normalization technique' in which common properties are factored out into shared flyweight objects. (Note: the idea is similar to data model normalization, a process in which the modeler attempts to minimize redundancy).
An example of the Flyweight Pattern is within the JavaScript engine itself which maintains a list of immutable strings that are shared across the application.
Other examples include characters and line-styles in a word processor, or 'digit receivers' in a public switched telephone network application. You will find flyweights mostly in utility type applications such as word processors, graphics programs, and network apps; they are less often used in data-driven business type applications.

## 2. Diagram




## 3. Participants

The objects participating in this pattern are:

- **Client** -- In sample code: **Computer**
    - calls into FlyweightFactory to obtain flyweight objects
- **FlyweightFactory** -- In sample code: **FlyweightFactory**
    - creates and manages flyweight objects
    - if requested, and a flyweight does not exist, it will create one
    - stores newly created flyweights for future requests
- **Flyweight** -- In sample code: **Flyweight**
    - maintains intrinsic data to be shared across the application

## 4. Sample code in JavaScript

In our example code we are building computers. Many models, makes, and processor combinations are common, so these characteristics are factored out and shared by Flyweight objects.

The FlyweightFactory maintains a pool of Flyweight objects. When requested for a Flyweight object the FlyweightFactory will check if one already exists; if not a new one will be created and stored for future reference. All subsequent requests for that same computer will return the stored Flyweight object.

Several different computers are added to a ComputerCollection. At the end we have a list of 7 Computer objects that share only 2 Flyweights. These are small savings, but with large datasets the memory savings can be significant.

The log function is a helper which collects and displays results.

```
function Flyweight (make, model, processor) {
   this.make = make;
   this.model = model;
   this.processor = processor;
};

var FlyWeightFactory = (function () {
   var flyweights = {};

   return {

       get: function (make, model, processor) {
           if (!flyweights[make + model]) {
               flyweights[make + model] =
                   new Flyweight(make, model, processor);
           }
           return flyweights[make + model];
       },

       getCount: function () {
           var count = 0;
           for (var f in flyweights) count++;
           return count;
```

```
        }
    }
})();

function ComputerCollection () {
    var computers = {};
    var count = 0;

    return {
        add: function (make, model, processor, memory, tag) {
            computers[tag] =
                new Computer(make, model, processor, memory, tag);
            count++;
        },

        get: function (tag) {
            return computers[tag];
        },

        getCount: function () {
            return count;
        }
    };
}

var Computer = function (make, model, processor, memory, tag) {
    this.flyweight = FlyWeightFactory.get(make, model, processor);
    this.memory = memory;
    this.tag = tag;
    this.getMake = function () {
        return this.flyweight.make;
    }
    // ...
}

// log helper

var log = (function () {
    var log = "";

    return {
        add: function (msg) { log += msg + "\n"; },
        show: function () { alert(log); log = ""; }
    }
})();

function run() {
    var computers = new ComputerCollection();

    computers.add("Dell", "Studio XPS", "Intel", "5G", "Y755P");
    computers.add("Dell", "Studio XPS", "Intel", "6G", "X997T");
    computers.add("Dell", "Studio XPS", "Intel", "2G", "U8U80");
```

```
    computers.add("Dell", "Studio XPS", "Intel", "2G", "NT777");
    computers.add("Dell", "Studio XPS", "Intel", "2G", "0J88A");
    computers.add("HP", "Envy", "Intel", "4G", "CNU883701");
    computers.add("HP", "Envy", "Intel", "2G", "TXU003283");

    log.add("Computers: " + computers.getCount());
    log.add("Flyweights: " + FlyWeightFactory.getCount());
    log.show();
}
```

# III. Observer Pattern

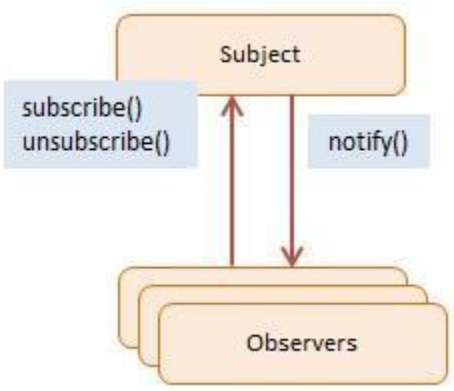
## 1. Summary

The Observer pattern offers a subscription model in which objects subscribe to an event and get notified when the event occurs. This pattern is the cornerstone of event driven programming, including JavaScript. The Observer pattern facilitates good object-oriented design and promotes loose coupling.

When building web apps you end up writing many event handlers. Event handlers are functions that will be notified when a certain event fires. These notifications optionally receive an event argument with details about the event (for example the x and y position of the mouse at a click event).

The event and event-handler paradigm in JavaScript is the manifestation of the Observer design pattern. Another name for the Observer pattern is Pub/Sub, short for Publication/Subscription.

## 2. Diagram




## 3. Participants

The objects participating in this pattern are:

- **Subject** -- In sample code: Click
    - maintains a list of observers. Any number of Observer objects may observe a Subject
    - implements an interface that lets observer objects subscribe or unsubscribe
    - sends a notification to its observers when its state changes
- **Observers** -- In sample code: clickHandler
    - has a function signature that can be invoked when Subject changes (i.e. event occurs)

## 4. Sample code in JavaScript

The Click object represents the Subject. The **clickHandler** function is the subscribing Observer. This handler subscribes, unsubscribes, and then subscribes itself while events are firing. It gets notified only of events #1 and #3.
Notice that the fire method accepts two arguments. The first one has details about the event and the second one is the context, that is, the value for when the eventhandlers are called. If no context is provided this will be bound to the global object (window).
The log function is a helper which collects and displays results.

```
Click.prototype = {

    subscribe: function(fn) {
        this.handlers.push(fn);
    },

    unsubscribe: function(fn) {
        this.handlers = this.handlers.filter(
            function(item) {
                if (item !== fn) {
                    return item;
                }
            }
        );
    },

    fire: function(o, thisObj) {
        var scope = thisObj || window;
        this.handlers.forEach(function(item) {
            item.call(scope, o);
        });
    }
}

// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
})();

function run() {
    var clickHandler = function(item) {
        log.add("fired: " + item);
    };

    var click = new Click();

    click.subscribe(clickHandler);
    click.fire('event #1');
    click.unsubscribe(clickHandler);
    click.fire('event #2');
    click.subscribe(clickHandler);
    click.fire('event #3');

    log.show();
}
```

# IV. State Pattern

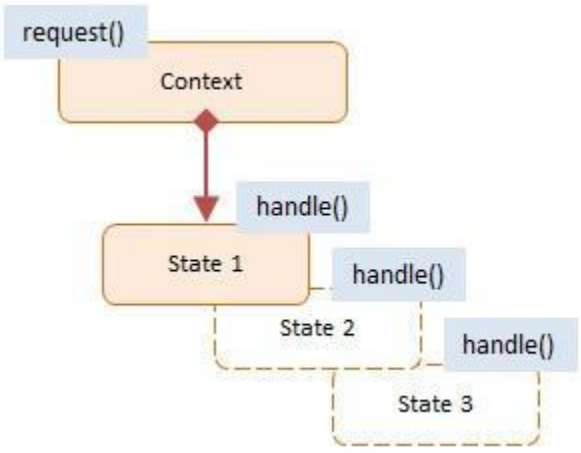
## 1. Summary

The State pattern provides state-specific logic to a limited set of objects in which each object represents a particular state. This is best explained with an example.

Say a customer places an online order for a TV. While this order is being processed it can in one of many states: New, Approved, Packed, Pending, Hold, Shipping, Completed, or Canceled. If all goes well the sequence will be New, Approved, Packed, Shipped, and Completed. However, at any point something unpredictable may happen, such as no inventory, breakage, or customer cancelation. If that happens the order needs to be handled appropriately. Applying the State pattern to this scenario will provide you with 8 State objects, each with its own set of properties (state) and methods (i.e. the rules of acceptable state transitions). State machines are often implemented using the State pattern. These state machines simply have their State objects swapped out with another one when a state transition takes place.

Two other examples where the State pattern is useful include: vending machines that dispense products when a correct combination of coins is entered, and elevator logic which moves riders up or down depending on certain complex rules that attempt to minimize wait and ride times.

## 2. Diagram



## 3. Participants

The objects participating in this pattern are:

- **Context** -- In sample code: TrafficLight
    - exposes an interface that supports clients of the service
    - maintains a reference to a state object that defines the current state
    - allows State objects to change its current state to a different state
- **State** -- In sample code: Red, Yellow, Green
    - encapsulates the state values and associated behavior of the state

## 4. Sample code in JavaScript

Our example is a traffic light (i.e. TrafficLight object) with 3 different states: Red, Yellow and Green, each with its own set of rules. The rules go like this: Say the traffic light is Red. After a delay the Red state changes to the Green state. Then, after another delay, the Green state changes to the Yellow state. After a very brief delay the Yellow state is changed to Red. And on and on.

It is important to note that it is the State object that determines the transition to the next state. And it is also the State object that changes the current state in the TrafficLight -- not the TrafficLight itself.

For demonstration purposes, a built-in counter limits the number of state changes, or else the program would run indefinitely.

The log function is a helper which collects and displays results.

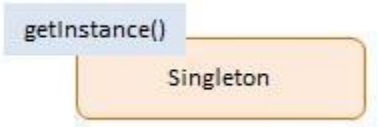
```
var TrafficLight = function () {
   var count = 0;
   var currentState = new Red(this);
   this.change = function (state) {
       // limits number of changes
       if (count++ >= 10) return;
       currentState = state;
       currentState.go();
   };
   this.start = function () {
       currentState.go();
   };
}

var Red = function (light) {
   this.light = light;

   this.go = function () {
       log.add("Red --> for 1 minute");
       light.change(new Green(light));
   }
};

var Yellow = function (light) {
   this.light = light;
   this.go = function () {
       log.add("Yellow --> for 10 seconds");
       light.change(new Red(light));
   }
};

var Green = function (light) {
   this.light = light;
   this.go = function () {
       log.add("Green --> for 1 minute");
       light.change(new Yellow(light));
   }
};
var log = (function () {
   var log = "";

   return {
       add: function (msg) { log += msg + "\n"; },
       show: function () { alert(log); log = ""; }
   }
})();
function run() {
   var light = new TrafficLight();
   light.start();
   log.show();
}
```

# V. Singleton Pattern

## 1. Summary

The Singleton Pattern limits the number of instances of a particular object to just one. This single instance is called the singleton.
Singletons are useful in situations where system-wide actions need to be coordinated from a single central place. An example is a database connection pool. The pool manages the creation, destruction, and lifetime of all database connections for the entire application ensuring that no connections are 'lost'.
Singletons reduce the need for global variables which is particularly important in JavaScript because it limits namespace pollution and associated risk of name collisions.

## 2. Diagram




## 3. Participants

The objects participating in this pattern are:

- Singleton -- In sample code: Singleton
    - defines getInstance() which returns the unique instance.
    - responsible for creating and managing the instance object.

## 4. Sample code in JavaScript

The Singleton object is implemented as an immediate anonymous function. The function executes immediately by wrapping it in brackets followed by two additional brackets. It is called anonymous because it doesn't have a name.
The getInstance method is Singleton's gatekeeper. It returns the one and only instance of the object while maintaining a private reference to it which is not accessible to the outside world.
The getInstance method demonstrates another design pattern called Lazy Load. Lazy Load checks if an instance has already been created; if not, it creates one and stores it for future reference. All subsequent calls will receive the stored instance. Lazy loading is a CPU and memory saving technique by creating objects only when absolutely necessary.

```
var Singleton = (function () {
   var instance;

   function createInstance() {
       var object = new Object("I am the instance");
       return object;
   }

   return {
       getInstance: function () {
           if (!instance) {
               instance = createInstance();
           }
           return instance;
       }
   };
})();

function run() {

   var instance1 = Singleton.getInstance();
   var instance2 = Singleton.getInstance();

   alert("Same instance? " + (instance1 === instance2));
}
```

## Summary of this Chapter

//@TODO

# Chapter 10 : Advanced CCClass

Compared to other JavaScript class implementations, the specialty of CCClass lies more within the metadata which has strong augmentability and can be plentifully defined. CCClass has a lot of details that you can get familiar with developing games. This article will list these specifications.

## Glossary

- CCClass: Class declared with `cc.Class`.
- Prototype Object: the object literals passed in when declaring class with `cc.Class`.
- Instance Member: including **member variable** and **instance method**.
- Static Member: including **static variable** and **static method**.
- Runtime: when a project runs in preview or production environment, compare to scripts run in editor.
- Serialization: Parse object in memory and output its data to a certain format of string for storage or transferring.

## Prototype Object Literals

All listed keys can be emitted, just declare the ones that needed:

```
cc.Class({

    // Class name for serialization
    // value type: String
    name: "Character",

    // base class, can be any cc.Class
    // value type: Function
    extends: cc.Component,

    // constructor
    // value type: Function
    ctor: function () {},

    // property definition (method one: object)
    properties: {
```

```
        text: ""
    },

    // property definition (method two: arrow function)
    properties: () => ({
        text: ""
    }),

    // instance method
    print: function () {
        cc.log(this.text);
    },

    // static members
    // value type: Object
    statics: {
        _count: 0,
        getCount: function () {}
    },

    // this property is for Component only
    // value type: Object
    editor: {
        disallowMultiple: true
    }
});
```

### Class name

Class name can be any string, but should be identical. You can use `cc.js.getClassName` to get class name and `cc.js.getClassByName` to lookup the corresponding class. For components defined in user scripts you can omit the class name since the editor will use filename for component script for serialization. For other CCClass if you don't need to serialize class you can omit the name.

## Constructor

### Declare by `ctor`

Use `ctor` to declare constructor of CCClass, for deserialization to work, constructor of CCClass **does NOT allow** parameter.

If the developer absolutely needs to pass parameters, `arguments` can be used to capture parameters, but do make sure you can finish the instance creation process event if no parameter is passed.

### Declare by `__ctor__`

`__ctor__` is the same as `ctor`, but it can receive constructor parameters, and will **NOT** call the constructor of parent class automatically, so you can call the constructor of parent by yourself. `__ctor__` is not the standard way to define constructor, always use the `ctor` unless you have a specific need.

## Type checks

### Check instance type

Native JavaScript's `instanceof` is enough for this:

```
var Sub = cc.Class({
    extends: Base
});

var sub = new Sub();
cc.log(sub instanceof Sub);       // true
cc.log(sub instanceof Base);      // true

var base = new Base();
cc.log(base instanceof Sub);      // false
```

### Check class type

Use `cc.isChildClassOf` to check if a class inherits from another:

```
var Texture = cc.Class();
var Texture2D = cc.Class({
    extends: Texture
});

cc.log(cc.isChildClassOf(Texture2D, Texture));   // true
```

The two parameters passed in should both be constructor function of the class instead of class instance. If two classes are the same, `isChildClassOf` will also return true.

## Members

### Member variable

The member variable defined inside the constructor cannot be serialized and not visible in the **Properties** panel.

```
var Sprite = cc.Class({
    ctor: function () {
        // Declare member variable and assign default value.
        this.url = "";
        this.id = 0;
    }
});
```

*We recommend add `_` prefix to variable name for private members.*

### Instance Method

Instance method please declare in the Prototype Object:

```
var Sprite = cc.Class({
    ctor: function () {
        this.text = "this is sprite";
    },
    // declare an instance method called "print"
```

```
    print: function () {
        cc.log(this.text);
    }
});

var obj = new Sprite();
// call the instance method
obj.print();
```

### Static variable and static method

Static variables and static methods can be defined in `statics` property of the Prototype Object.

```
var Sprite = cc.Class({
    statics: {
        // static variable
        count: 0,
        // static method
        getBounds: function (spriteList) {
            // ...
        }
    }
});
```

The above code equals:

```
var Sprite = cc.Class({ ... });

// static variable
Sprite.count = 0;
// static method
Sprite.getBounds = function (spriteList) {
    // ...
};
```

Static members will be inherited by child class, it will **shallow copy** the static variable of the parent class to child class.

```
var Object = cc.Class({
    statics: {
        count: 11,
        range: { w: 100, h: 100 }
    }
});

var Sprite = cc.Class({
    extends: Object
});

cc.log(Sprite.count);    // 11, for count is inherited from Object class

Sprite.range.w = 200;
cc.log(Object.range.w);  // 200, for Sprite.range and Object.range references the same
object.
```

If no inheritance is considered, private static member can also be defined outside of the class:

```
// local method
function doLoad (sprite) {
    // ...
};

// local variable
var url = "foo.png";

var Sprite = cc.Class({
    load: function () {
        this.url = url;
        doLoad(this);
    };
});
```

## Inheritance

### Constructor of parent class

**Notice** no matter if child class has defined its own constructor, the constructor of parent class will be called before child class instantiation.

```
var Node = cc.Class({
    ctor: function () {
        this.name = "node";
    }
});

var Sprite = cc.Class({
    extends: Node,
```

```
    ctor: function () {
        // this.name is already initialized in parent class constructor
        cc.log(this.name);    // "node"
        // re-set this.name
        this.name = "sprite";
    }
});

var obj = new Sprite();
cc.log(obj.name);    // "sprite"
```

So you shouldn't call parent's constructor explicitly, or it will be called twice.

```
var Node = cc.Class({
    ctor: function () {
        this.name = "node";
    }
});

var Sprite = cc.Class({
    extends: Node,
    ctor: function () {
        Node.call(this);        // Don't!
        this._super();          // Don't either!

        this.name = "sprite";
    }
});
```

In special cases, arguments of the parent's constructor may not be compatible with the child's constructor. Then you have to call the parent's constructor and pass the needed arguments into it manually. At this point you should declare the child's constructor in `__ctor__`.

**Override**

All member methods are virtual methods, so child class can override parent method:

```
var Shape = cc.Class({
    getName: function () {
        return "shape";
    }
});

var Rect = cc.Class({
    extends: Shape,
    getName: function () {
        return "rect";
    }
});

var obj = new Rect();
cc.log(obj.getName());    // "rect"
```

Different from constructor, overridden method from parent class will not be called automatically, so you need to call explicitly if needed:

Method one: use `this._super` provided by CCClass:

```
var Shape = cc.Class({
    getName: function () {
        return "shape";
    }
});

var Rect = cc.Class({
    extends: Shape,
    getName: function () {
        var baseName = this._super();
        return baseName + " (rect)";
    }
});

var obj = new Rect();
cc.log(obj.getName());    // "shape (rect)"
```

Method two: use native JavaScript language feature:

```
var Shape = cc.Class({
```

```
    getName: function () {
        return "shape";
    }
});

var Rect = cc.Class({
    extends: Shape,
    getName: function () {
        var baseName = Shape.prototype.getName.call(this);
        return baseName + " (rect)";
    }
});

var obj = new Rect();
cc.log(obj.getName());    // "shape (rect)"
```

If the parent and child class are neither CCClass, you can use lower level API `cc.js.extend` to do inheritance.

## Property

Properties are special instance members that can be serialized and show in **Properties** panel.

### Property and constructor

No need to define property in the constructor, the initialization of properties is even before the constructor is called. So you can use those properties in the constructor function. If a default value of a property is not available in CCClass definition, you can also reassign **default** value in constructor.

```
var Sprite = cc.Class({
    ctor: function () {
        this.img = LoadImage();
    },

    properties: {
        img: {
            default: null,
            type: Image
        }
    }
});
```

It should be noted that the process of the deserialization of the property occurs immediately **after** the constructor's execution, so that only the default values of the properties can be obtained and modified in the constructor, and the previously saved (serialized) values can not be obtained and modified.

**Attribute**

All attributes are optional, but at least one of the `default`,`get`, `set` attributes must be declared.

## `default` attribute

`default` is used to declare the default value of the property, and the property that declares the default value is implemented as a member variable by the CCClass. The default value is only used when the object is **created for the first time**, that is, when the default value is modified, the current value of the component that has been added to the scene is not changed.

When you add a component in the editor, and then return to the script to modify a default value, you will not see the change in **Properties**. Because the current value of the property has been serialized to the scene, is no longer used to create the default value of the first time. If you want to force all properties back to their default values, you can select Reset in the component menu of the **Properties**.

`default` allows you to set the following types of values:

- Any value of type, string, or boolean
- `null` or `undefined`

Object which instance of subclasses inherited from `cc.ValueType`, such as `cc.Vec2`, `cc.Color` or `cc.Rect`

```
properties: {
    pos: {
        default: new cc.Vec2(),
    }
}
```

- Empty array `[]` or empty object `{}`

- A function that allows you to return any type of value, which is called again each time the class is instantiated and takes the return value as the new default:

```
properties: {
    pos: {
        default: function () {
            return [1, 2, 3];
        },
    }
}
```

## `visible` attribute

By default, whether or not a property is displayed in the **Properties** depends on whether the property name begins with an underscore `_`. If you start with an underscore, the **Properties** is not displayed by default, otherwise it is displayed by default.

If you want to force the display in **Properties**, you can set the `visible` attribute to true:

```
properties: {
    _id: {      // begins with an underscore would have been hidden
        default: 0,
        visible: true
    }
}
```

If you want to force the hidden, you can set the `visible` parameter to false:

```
properties: {
    id: {       // not begins with an underscore would have been displayed
        default: 0,
        visible: false
    }
}
```

## `serializable` attribute

Properties that specify the `default` value are serialized by default. After serialization, the settings in the editor will be saved to the resource file such as the scene, and automatically restore the previously set value when loading the scene. If you do not want to serialize, you can set `serializable: false`.

```
temp_url: {
    default: "",
    serializable: false
}
```

## `type` attribute

When the `default` can not provide sufficient detailed type information, in order to be able to display the correct input control in **Properties**, it is necessary to use the `type` to declare the specific type explicitly:

- When the default value is null, the type is set to the specified type of constructor, so that the **Properties** can know that a Node control should be displayed.

```
enemy: {
    default: null,
    type: cc.Node
}
```

- When the default value is a number type, set type to `cc.Integer` to indicate that it is an integer, so that the property can not enter a decimal point in the **Properties**.

```
score: {
    default: 0,
    type: cc.Integer
}
```

- When the default value is an enum (`cc.Enum`), since the enumeration value itself is actually a number, so set the type to enum type to display in **Properties** as a drop-down enum box.

```
wrap: {
      default: Texture.WrapMode.Clamp,
      type: Texture.WrapMode
  }
```

## `override` attribute

All properties will be inherited by subclasses, if the subclass wants to override the parent property with the same name, you need to explicitly set the override attribute, otherwise there will be a warning:

```
_id: {
    default: "",
    tooltip: "my id",
    override: true
},

name: {
    get: function () {
        return this._name;
    },

    displayName: "Name",
    override: true
}
```

### Property delay definition

If two classes refer to each other, in the script load phase there will be a circular reference, the circular reference will lead to script loading error:

- Game.js

```
var Item = require("Item");

 var Game = cc.Class({
     properties: {
         item: {
             default: null,
             type: Item
         }
     }
```

```
});

module.exports = Game;
```

- Item.js

```
var Game = require("Game");

var Item = cc.Class({
    properties: {
        game: {
            default: null,
            type: Game
        }
    }
});

module.exports = Item;
```

When the above two scripts are loaded, because they form a closed loop during the requiring process, so the load will appear a circular reference error, the type will become undefined when circular referenced.

So we advocate the use of the following property definition:

- Game.js

```
var Game = cc.Class({
    properties: () => ({
        item: {
            default: null,
            type: require("Item")
        }
    })
});

module.exports = Game;
```

- Item.js

```
var Item = cc.Class({
    properties: () => ({
```

```
        game: {
            default: null,
            type: require("Game")
        }
    })
});

module.exports = Item;
```

This is done by specifying properties as an arrow function (lambda expression) of ES6. The contents of the arrow function are not executed synchronously during the script load, but are loaded by the CCClass asynchronously after all scripts have been loaded. So the cycle of reference does not occur during loading, properties can be normal initialization.

The arrow function is used in conjunction with the ES6 standard for JavaScript, and Creator automatically compiles ES6 to ES5, and users do not have to worry about browser compatibility issues.

You can understand the arrow function like this:

## GetSet method

After the get or set is set in the property, the predefined get or set method can be triggered when the property is accessed.

**get**

Set the get method in the property:

```
properties: {
    width: {
        get: function () {
            return this.__width;
        }
    }
}
```

The get method can return any type of value.

This property can also be displayed in the **Properties** and can be accessed directly in all the code including within the constructor.

```
var Sprite = cc.Class({
    ctor: function () {
        this.__width = 128;
        cc.log(this.width);    // 128
    },

    properties: {
        width: {
            get: function () {
                return this.__width;
            }
        }
    }
});
```

Please note:

1. After setting get, this property can not be serialized, nor can it specify a default value, but it can still include most of the attributes except `default`, `serializable`.

```
width: {
    get: function () {
        return this.__width;
    },

    type: cc.Integer,
    tooltip: "The width of sprite"
}
```

2. The get attribute itself is read-only, but the returned object is not read-only. Users can still use the code to modify the object's internal properties, such as:

```
var Sprite = cc.Class({
    ...
    position: {
        get: function () {
            return this._position;
        },
    }
    ...
```

```
});

var obj = new Sprite();
obj.position = new cc.Vec2(10, 20);   // Failure! position is read only!
obj.position.x = 100;                 // Allow! _position object itself returned by
position can be modified!
```

**set**

Set the set method in the property:

```
width: {
    set: function (value) {
        this._width = value;
    }
}
```

The set method receives an incoming parameter, which can be of any type.

Set generally used with get:

```
width: {
    get: function () {
        return this._width;
    },

    set: function (value) {
        this._width = value;
    },

    type: cc.Integer,
    tooltip: "The width of sprite"
}
```

If there is no definition with get, then set itself can not be attached with any attributes.

As get, once set declared, this property can not be serialized, can not specify the default value.

# Attribute reference

The attribute is used to append metadata to a defined property, similar to the Decorator of the scripting language or the Attribute of C#.

**Properties panel corresponding attributes**

| Parameter name | Explanation | Type | Default | Remarks |
|---|---|---|---|---|
| type | Restrict the data type for property | (Any) | undefined | See type attribute |
| visible | Show or hide in the **Properties** | boolean | (note 1) | See visible attribute |
| displayName | Show another name in the **Properties** | string | undefined | - |
| tooltip | Add Tooltip for property in the **Properties** | string | undefined | - |
| multiline | Use multiple lined text box in the **Properties** | boolean | false | - |
| readonly | Read-only in the **Properties** | boolean | false | - |
| min | Restrict the minimum value in **Properties** | number | undefined | - |

| | | | | |
|---|---|---|---|---|
| max | Restrict the maximum value in **Properties** | number | undefined | - |
| step | Restrict the step value in **Properties** | number | undefined | - |
| range | One-time setup for min, max, step | [min, max, step] | undefined | step is optional |
| slide | Show a slider in the **Properties** | boolean | false | - |

#### Serialization associated attributes

These attributes cannot be used for the get method.

| Parameter name | Explanation | Type | Default | Remarks |
|---|---|---|---|---|
| serializable | Serialize this property | boolean | true | See serializable attribute |
| formerlySerializedAs | Specify the name of the field used in formerly serialization | string | undefined | Use this attribute to rename a property without losing its serialized value. |
| editorOnly | Reject this property before exporting the project | boolean | false | - |

# Other attributes

| Parameter name | Explanation | Type | Default | Remark |
| --- | --- | --- | --- | --- |
| default | Define default for the property | (Any) | undefined | See default attribute |
| notify | Trigger a specific method when assigning the property | `function (oldValue) {}` | undefined | The default property is needed to define and is not available for array.<br><br>Not support ES6 Classes. |
| override | When reloading the super class property, this parameter needs to be defined as true | boolean | false | See override attribute |
| animatable | Whether this property can be altered by the **Timeline** panel | boolean | undefined | - |

**Note 1**: The default value of `visible` is determined by the property name. When the property name starts with an underscore `_`, then the default is set to hide, otherwise it is by default set to show.