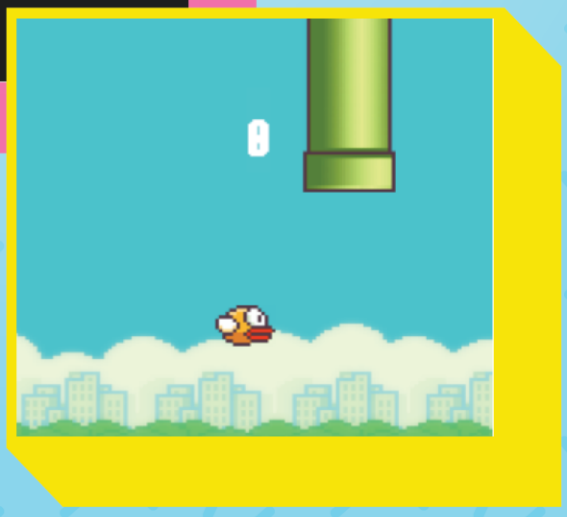
BUILD YOUR FIRST GAME


```ts
@ccclass('Bird')
export class Bird extends Component {

    @property({
        type: CCFloat,
        tooltip: 'how high does he fly?'
    })
    public jumpHeight: number = 1.5;

    @property({
        type: CCFloat,
        tooltip: 'how long does he fly?'
    })
    public jumpDuration: number = 1.5;

    //Animation property of the bird
    public birdAnimation: Animation;

    //Temporary location of the bird
    public birdLocation: Vec3;

    //hit detection call
    public hitSomething:boolean;
```



WITH COCOS CREATOR

Building Your First Cocos Creator Game – From Start To Finish

By Luke Stapley

Special Thanks To The Cocos Ecological Team for their support and help

Contents

Introduction

This tutorial is an introduction to people who are building games for Cocos Creator. This is the most intensive tutorial we have offered for those entering Cocos Creator version 3.0. We will share many of the new features that come with this version. We have updated the engine to include many great 3D offerings that were not available in 2.x.

But to help those who want to stay within the 2D world, this tutorial is dedicated to you. We hope you learn a lot from this, which motivates you to start building more games in 2D and 3D in the future.

Before you begin, we highly recommend you look at the first tutorial for Cocos Creator 3.x on YouTube. This can help you to understand node-based coding for Cocos Creator and prepare you a bit before you take on this game.

It's also recommended to study your TypeScript, so you understand more of what we are doing with the coding. Here are a few videos to watch:

Cocos Creator 3.x YouTube Video Series – https://www.youtube.com/watch?v=JSOXYPqZ1-8&list=PLbvpmJKjO3NA4dlW43GzhJUMaXyIp3xpc

Learning TypeScript for Gaming – https://www.youtube.com/watch?v=kpiO5-BtX4I

We also recommend reading up on both the documentation and APIs for Cocos Creator to give you a better understanding of how they work

Documentation for Cocos Creator – https://docs.cocos.com/creator/manual/

API calls for Cocos Creator – https://docs.cocos.com/creator/api/en/

So why Flappy Birds?

This was a good question I was asked when I decided to build this tutorial. One of the big reasons is that we continue to have people come to the website looking for tutorials, and this one, in particular, continues to be the most viewed. But due to it being initially built for Cocos Creator 2.0, it is far too old to be used and needs to be replaced.

So for beginners, this is an excellent opportunity to learn how to add assets, move them, create new nodes, have collision detection, add sound effects, and understand how to build the inner workings of a game. So get ready because we have made this course as detailed as possible, and the code you will be building as easy to understand as possible.

We note that this might not be the most efficient code. Still, we'd rather turn towards a better understanding first and then move you to optimization in future video tutorials.

So let's get flappin' and start building.

Part 1 - Preparation

Before we even sit down and start coding, we need to figure out how this game will be built and what parts are necessary. This is an essential part of each project because it's the cheapest to work from as a developer compared to the painstaking and costly work it takes to build new updates to an already prepared game.

Tools needed

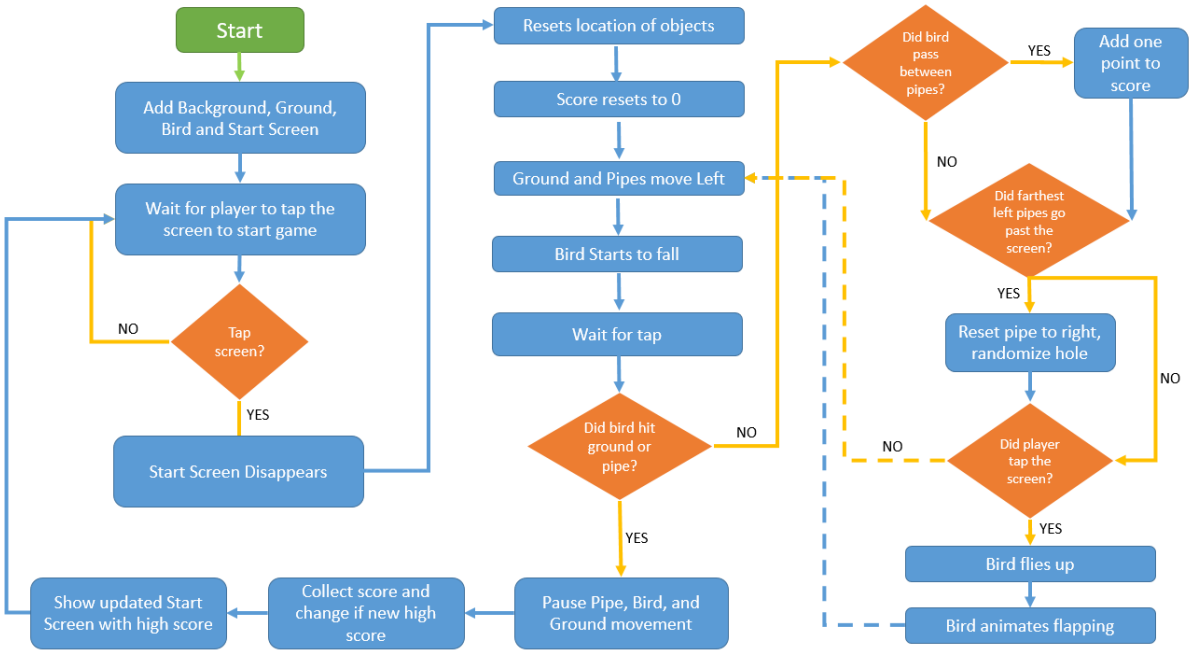
So first, what tools will we need?

- Cocos Creator 3.6 or better (find at www.cocos.com/en/creator)
- Visual Studio Code by Microsoft
- Google Chrome

Cocos Creator is a given, as we need a game engine. Next is Visual Studio Code for editing code as well as bug fixing. Finally, we need Google Chrome to test the game to find bugs. We assume you have some familiarity with all of these. So we won't be giving you a basic introduction to each software.

Game logic

Next, we need to think about the logic of the game. The game is built with this in mind:



It's always good to start with the logic tree of the game. It helps you to understand what happens when and how. Then you can understand what pieces we need.

As we can see, we will need a start screen, ground, background, bird, pipes, and a score to make this game work. Sound isn't included in this logic, but we'll add this as we go through the game.

After tapping to leave the starting screen, the game continues to play as the ground and pipes go left, and as long as the bird doesn't hit a pipe or ground, it keeps playing. Tapping the screen makes the bird fly up as it falls when we don't tap. Also, if we move past two pipes, we get a point. The game ends when we do hit something, and the game will hold on to the score if it's the high score and reset the score to zero when we start the game again. The screen also pauses after we hit something, and everything resets back to its original place once we tap the screen to start the game again. The high score screen also disappears when the game starts over.

From this, we can see we'll need some logic to talk to all the parts:

- Overall game logic that controls all other logic
- Ground moving logic
- Bird logic
- Pipes logic
- Scoring/high score logic.

Now that we figured it out let's keep this in mind as we start to build our game.

Get assets

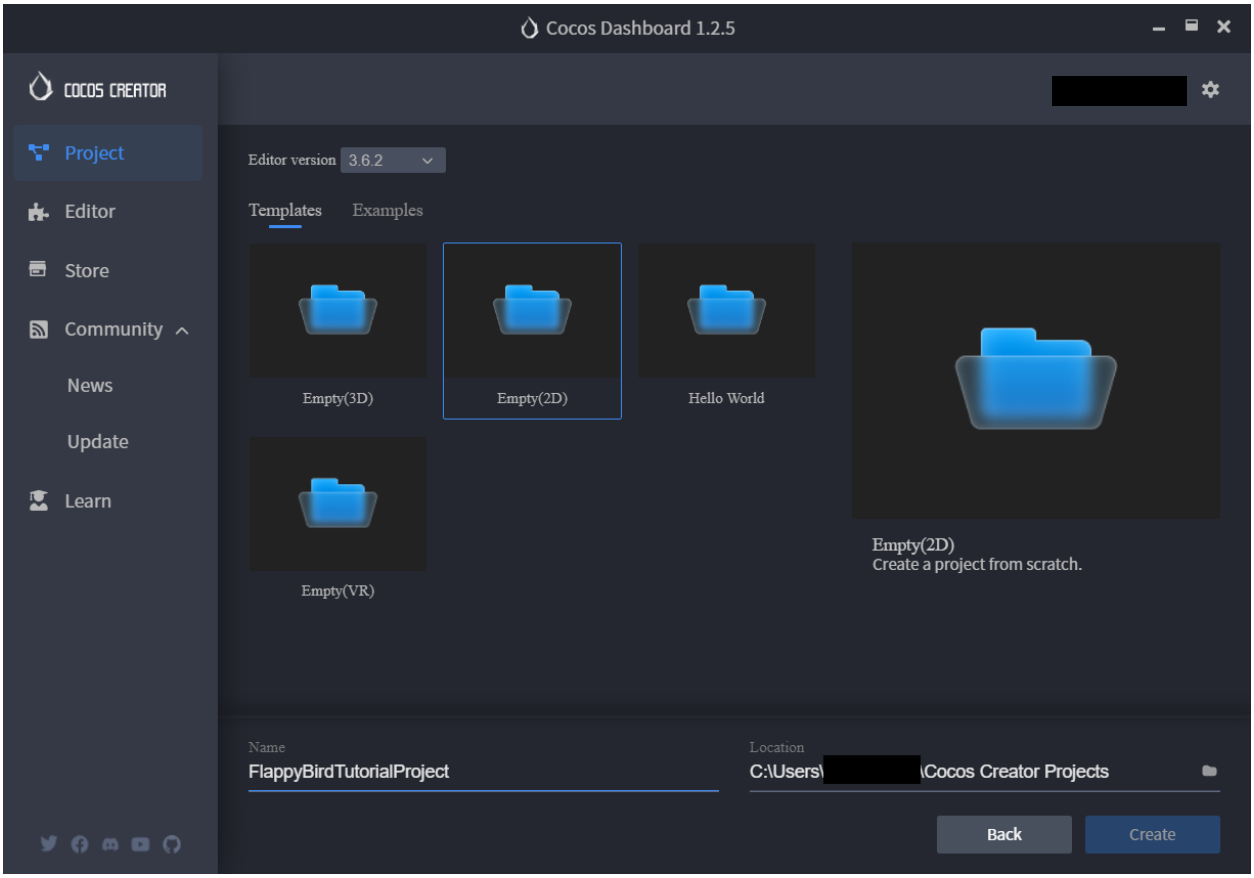
Now that we have the logic, let's get our assets. Luckily, someone on GitHub has almost all of them for us. Go to https://github.com/samuelcust/flappy-bird-assets and download the zip file and unzip it.

Even though the assets in this package already have numbers, they are images. I'd rather we use text instead. So, you can get the font at https://www.fontspace.com/flappy-bird-font-f21349 as freeware. You don't need to install the font on your computer. Just keep the tff file with the other assets.

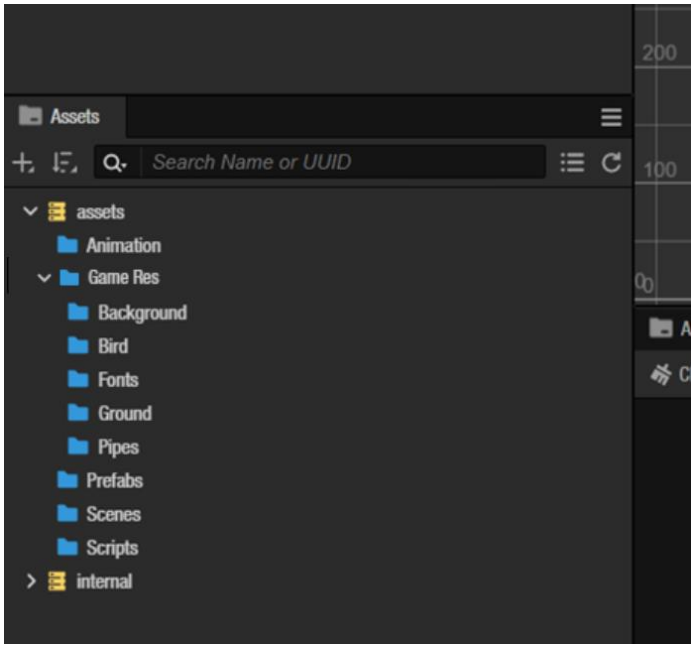
Now that our preparations are taken care of, we can finally start building the game.

Part 2 - Building The Game's Look

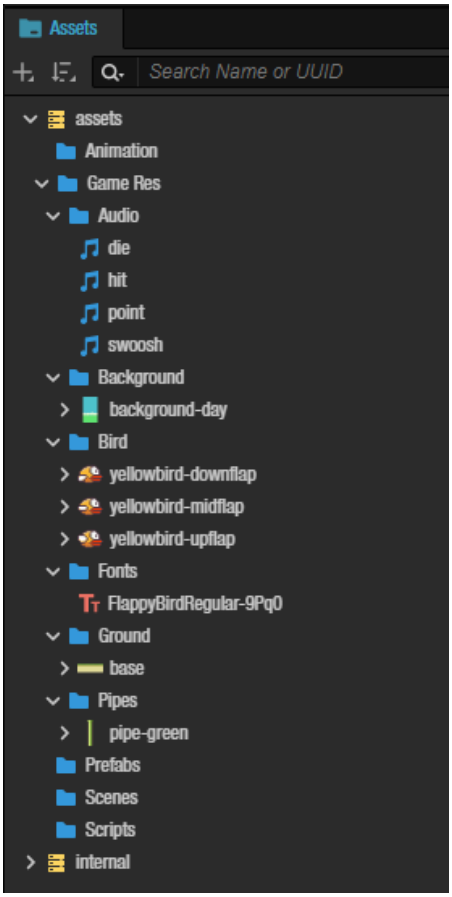
Let's go ahead and open Cocos Dashboard and create a new project. Make sure it's a blank project, name something easy to remember, and have it in a suitable directory.



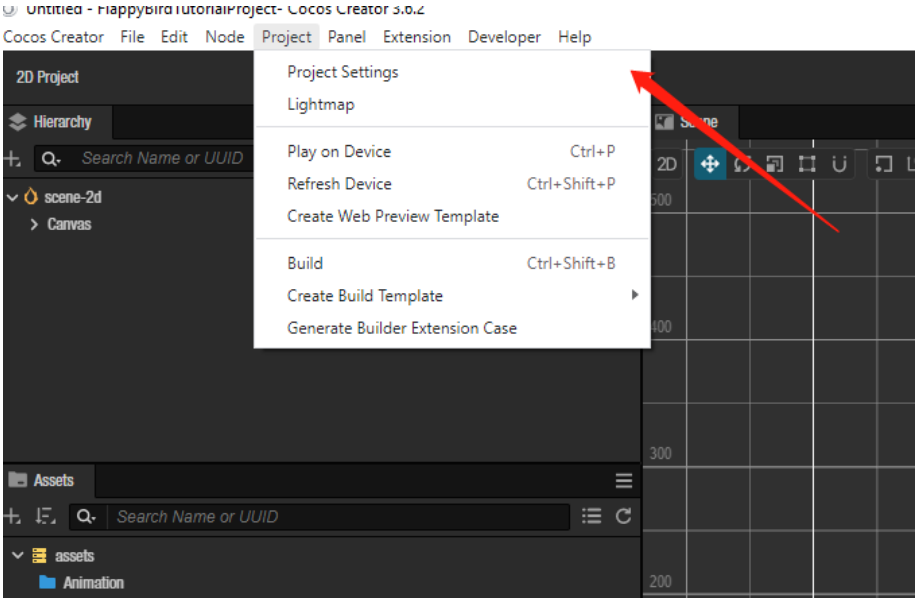
As you open Cocos Creator, add different folders for our images, scripts, font, prefabs, and sounds into our assets section. Make sure it looks similar to what I have in my assets window. Also, add a folder for Animation, Scenes, Prefabs, and Scripts. We'll be adding these later in the tutorial.



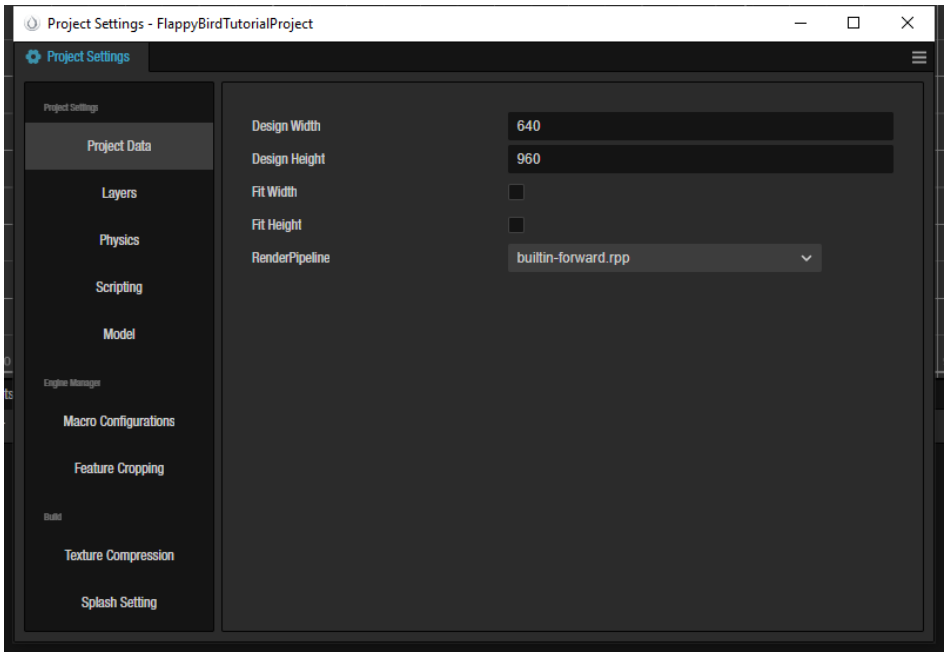
Add your files from GitHub into their respective places and your font from the font website. Keep the others empty for now.



Before we begin, we need to make this game a size that would fit most mobile screens in case someone wants to play this game on their phone instead of on the web. A good size that fits most phones is 640 x 960. So let's go to **Project -> Project Settings**



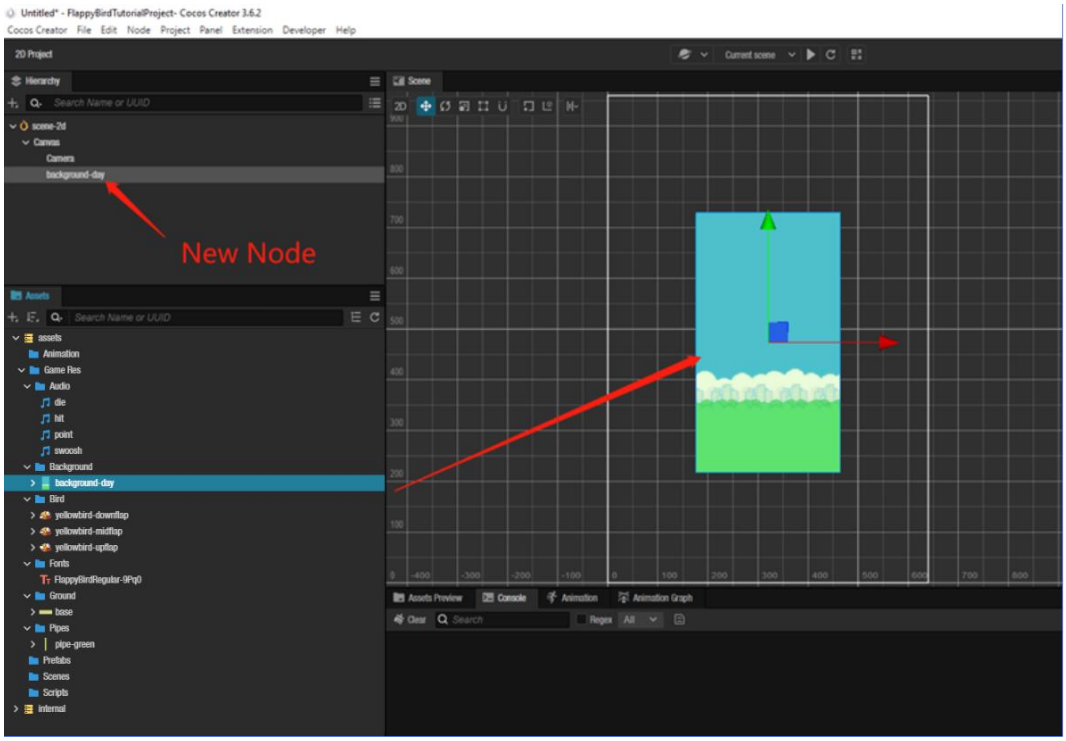
In the **Project Data** area, change the **Design Width** to 640 and the **Design Height** to 960. Uncheck **Fit Width** and **Fit Height** as well. This makes it so the width and height don't stretch out to the screen size.



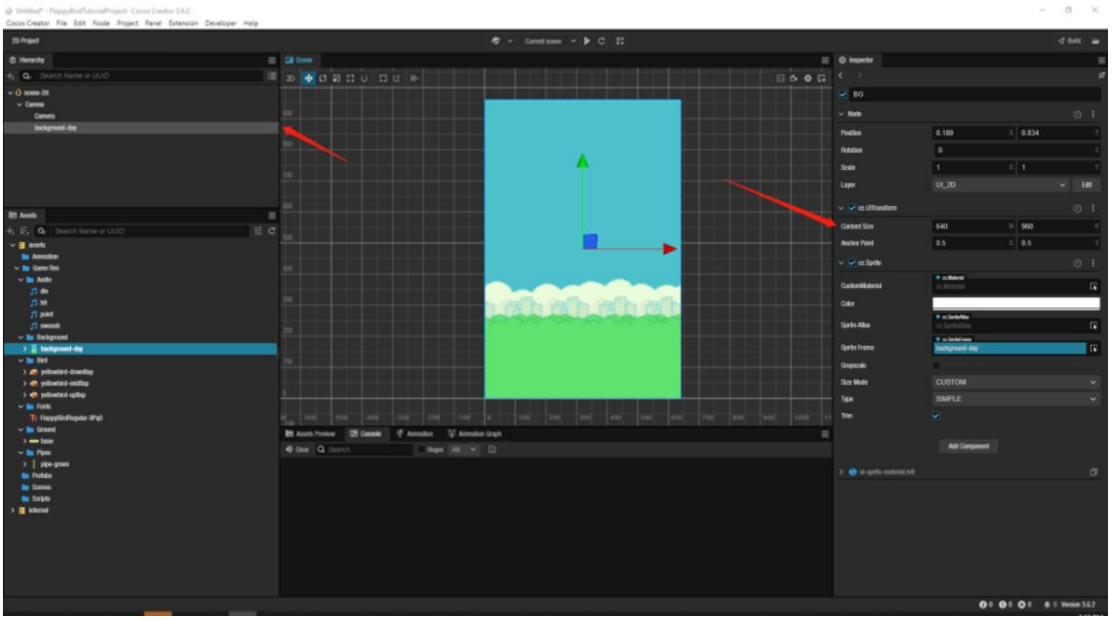
Adding the Background and the Bird

Now we can prototype the look of the game. Drag the background from your assets and place it into the

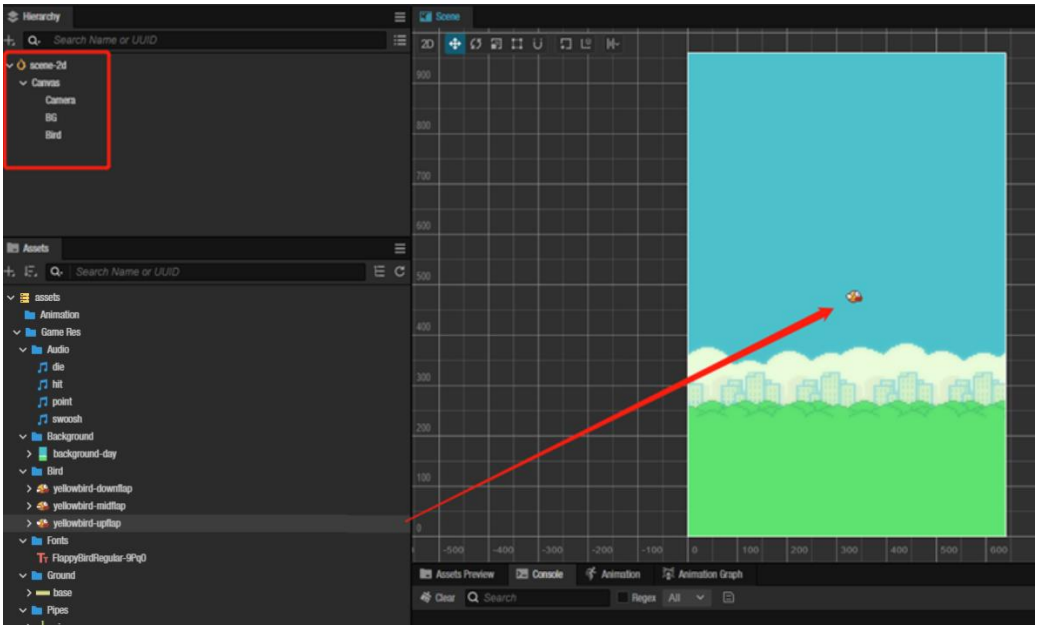
scene inside the **Canvas**. The asset will now become a node. The **Canvas** shows everything you'll see in a 2D game. A white box is used to display what will be on the screen.



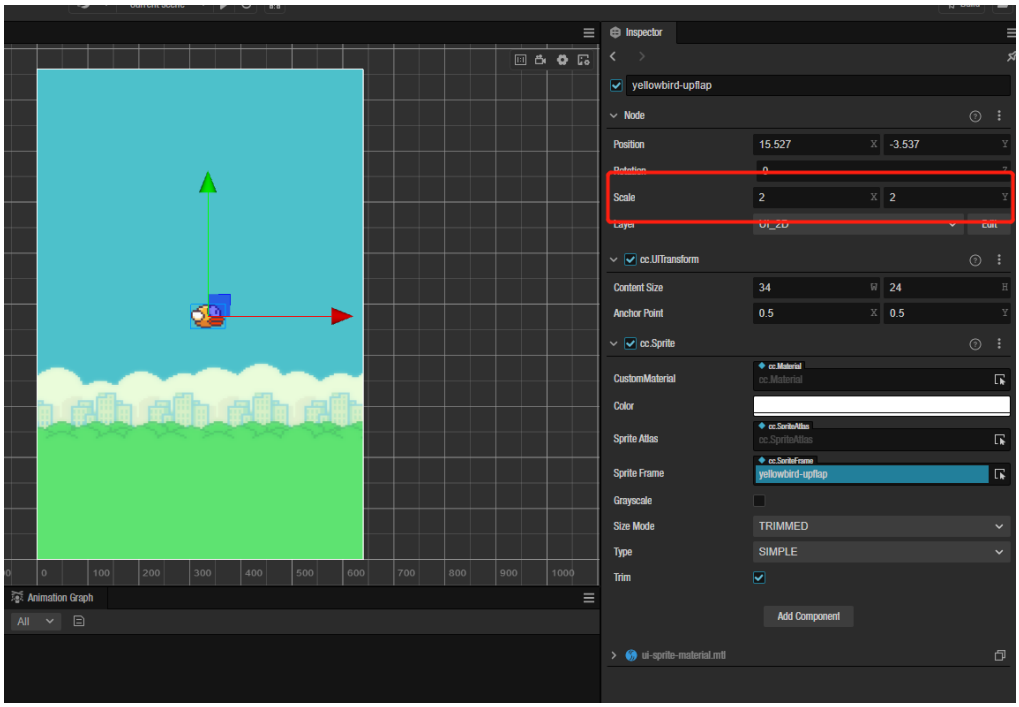
Change the size of the background to fit the **Canvas**. The **Inspector** can do this. Select the background node and go to **cc.UITransform** inside the **Inspector** and make the **Content Size** 640 x 960. We won't worry about stretching it as it's close to the original look. Also, make sure it fits the entire **Canvas**



Add the bird in the same way you added the background. Add the **yellowbird-upflap** image into the **Canvas**. Make sure to keep your node names tidy. You can rename both by selecting the node and pressing the `enter` key. I changed the background to **BG** and the bird to **Bird**.



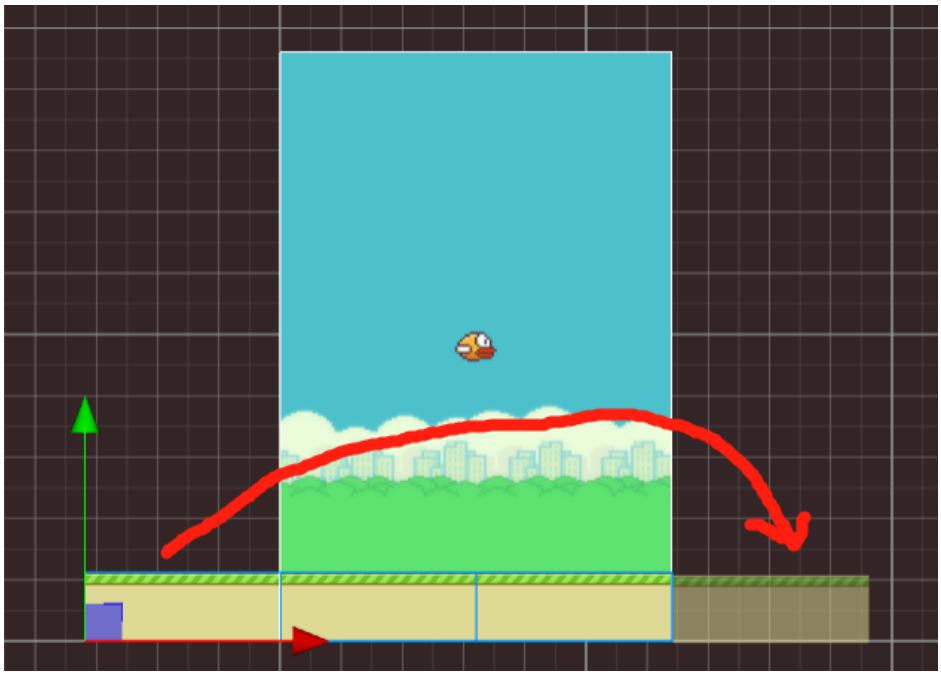
The bird is pretty tiny, so let's double the size. Go to the Inspector and increase the scale by 2.



Now that we're done prototyping the look and added our first nodes, we can start the first challenge.

Part 3 - Adding A Moving Ground

Now we are going to add the ground. We want the ground to move quickly to show that the bird is flying. So we have a problem. The ground asset provided to us will not fill the entire bottom of the screen. So we will have to reset the grounds every time the ground goes beyond the **Canvas**. Here's a picture to help explain:



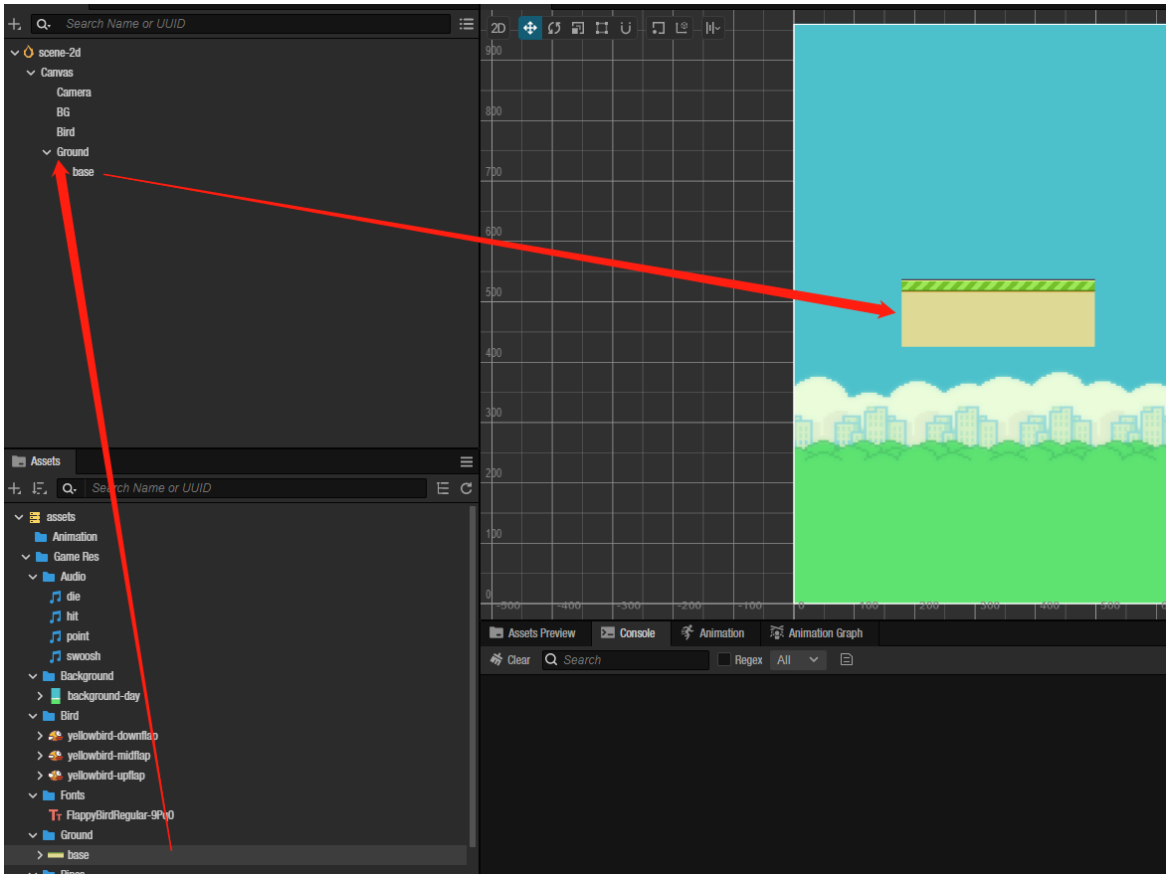
So we have enough room to fit three equally sized grounds, and as the game plays, they all move left. If the ground goes off the screen, it resets to the back of the line, and the next one does the same thing repeatedly until the game ends. It's an assembly line of ground! Doing this means we don't have to worry about making a long ground asset. Just make one coding request.

Note: there are better ways to build this, but since this is a beginner course, we'll use the most obvious and most straightforward way

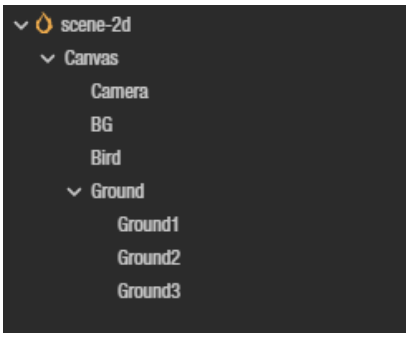
Adding the ground

To make this work, we need to put all three ground pieces inside a node that will control the ground pieces. So click on the canvas and then the "+" button to add an 'Empty Node'. From there, we can name it **Ground** and could give it a child node by clicking the "+" while highlighting **Ground**.

But instead, it’s best to drag the ground sprite "base" into the **Ground** node to make it a child of **Ground.** Make the first one Content Size of 320 * 112 (half of the screen width), just like we did with the background.

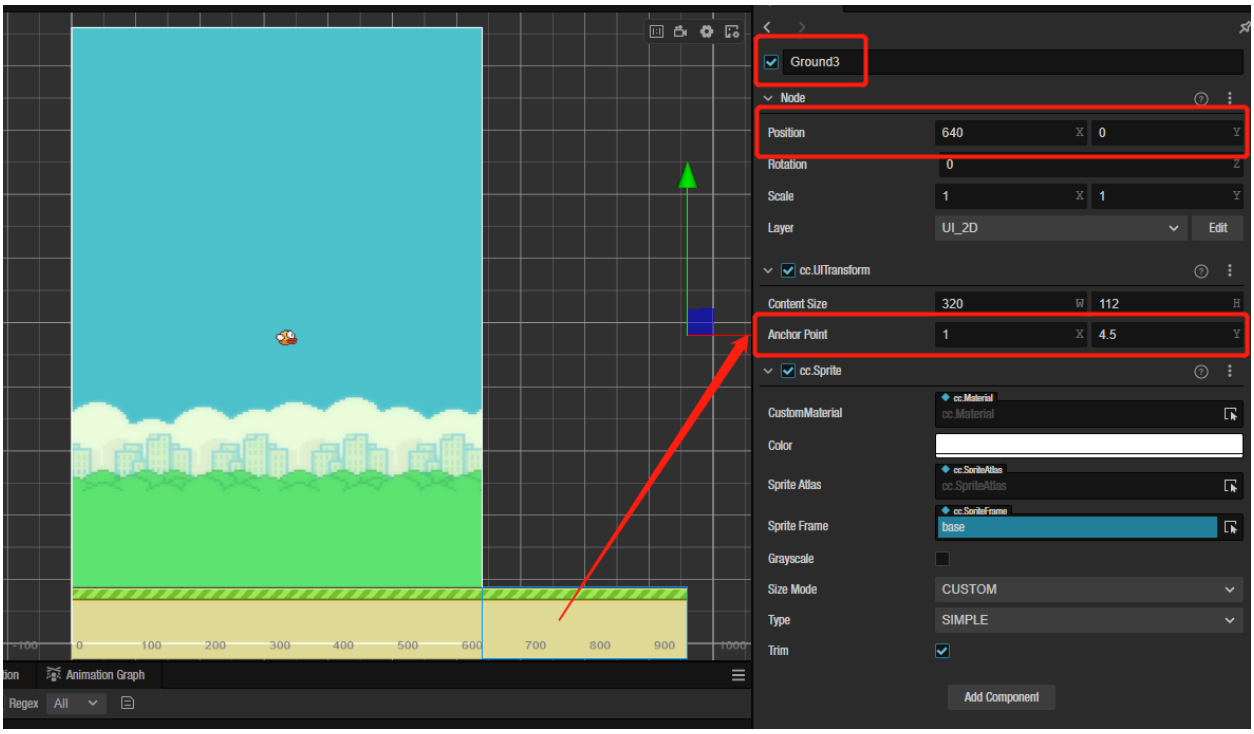


Make copies of this node. Please give them the names **Ground1**, **Ground2**, and **Ground3**.



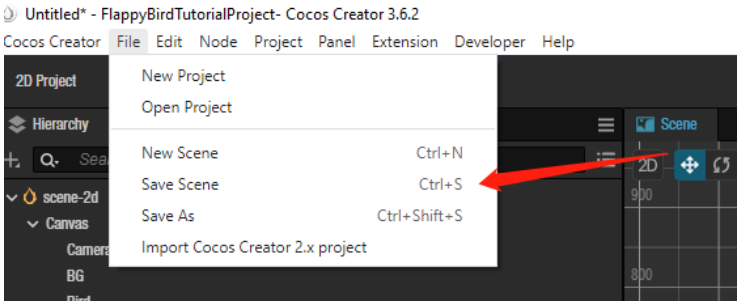
The best next thing is to space them out and ensure they are anchored to one specific location. This will help us to create good equations for the assembly line of ground assets. Here's what you should have in the properties of each node:

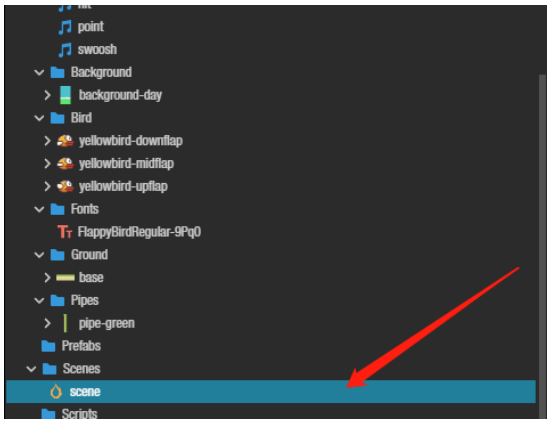
- ground1: position is (0,0), and anchor point is at (1, 4.5)
- ground2: position is (320,0), and anchor point is at (1, 4.5)
- ground3: position is (640,0), and anchor point is at (1, 4.5)



Now that they are set up correctly, the logic for the coding will be easy. When the position is more than the ground's total length, we need to move it back to the other end of the screen.

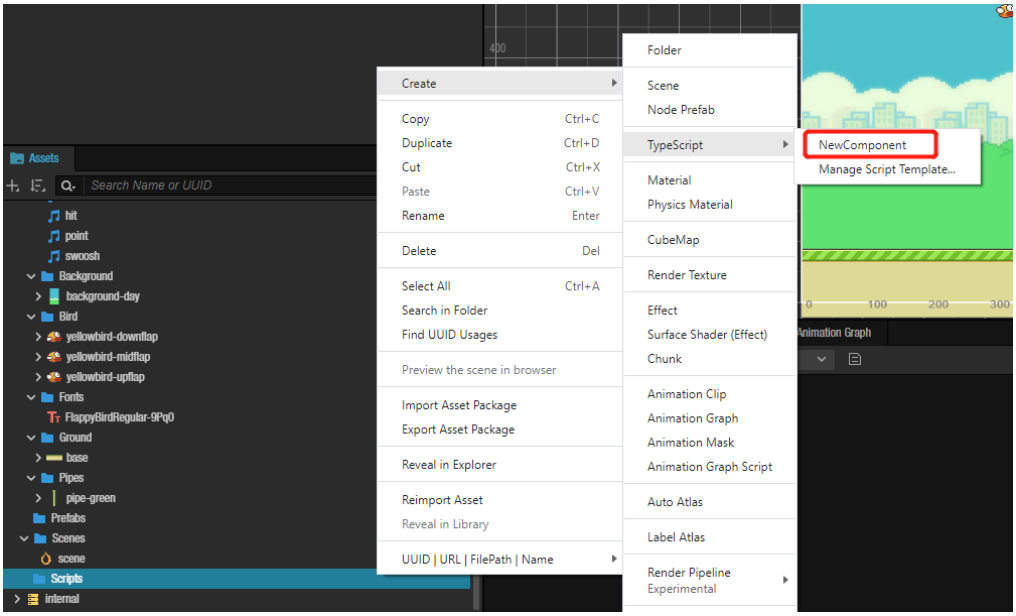
Now that this is complete, we should save the project and add the saved scene to the scene folder. Let's take the next step and start coding to move the ground. Don't worry about adding the pipes to the scene. We will set them up in the future.



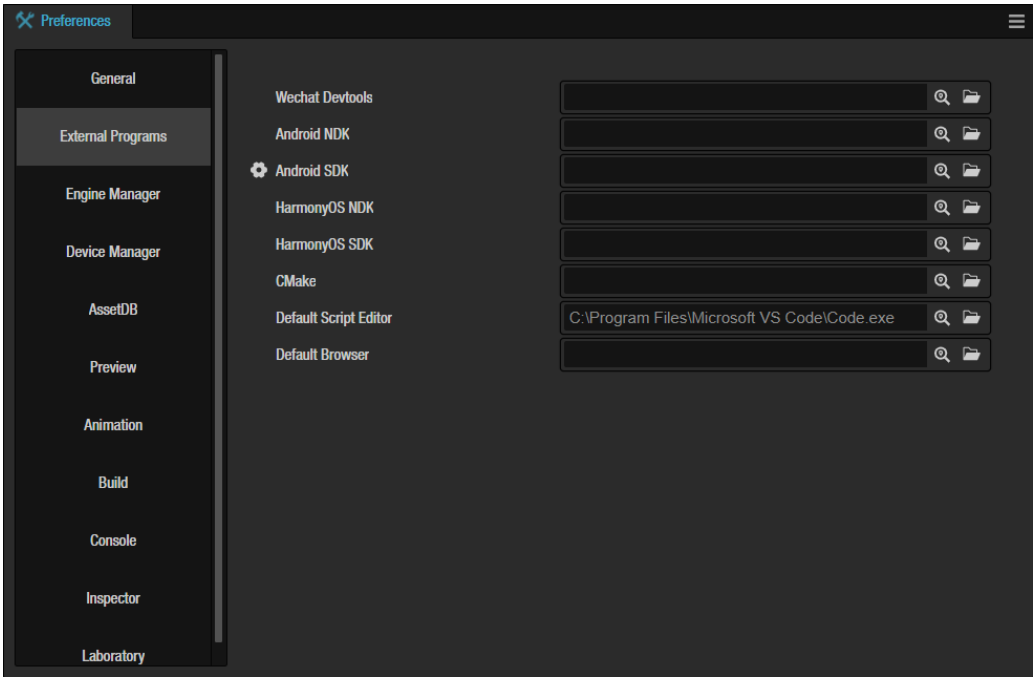


Making the ground move

Now that we have part of the node tree ready, we need to add the code to get it moving. In the 'scripts' folder, create a new TypeScript component. Name it 'Ground'.



Double-click on the new asset Ground.ts to open up Visual Studio Code. If it doesn't open, ensure the software is connected to your version of Cocos Creator. You can check by going to Cocos Creator -> Preferences -> External Programs and see if "Default Script Editor" is being used.



In every .ts file that is built with Cocos Creator, you'll see three items:

- **Import {}**: Add all the Cocos Creator APIs parts into your code.
- **Const {}**: This area tells the code to use decorators in the editor so we can edit components inside the editor instead of in the code. You won't be editing this.
- **@ccclass {}**: This is where most of the work will be done. This will be initiated when the component is added to a node.

To start, delete all the code inside the @ccclass and make sure your class is the file's name. This will help you import this code into other code, like the audio, so it's easier to find and work with later on.

NOTE: Most of the work will be done within ccclass. Unless told differently, please put the code you see in examples inside ccclass.

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

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


}
```

Now that we cleaned it up, let's add some properties. This allows us to edit whatever we want without going back into the code. We have many types: fonts, nodes, sprites, numbers, and more.

For now, we want to ensure which nodes are the ground nodes and are moving in an assembly line-like manner. So we need only to add three properties. I also add a tooltip to help people know what each property does. These are not necessary, but they help when building with others.

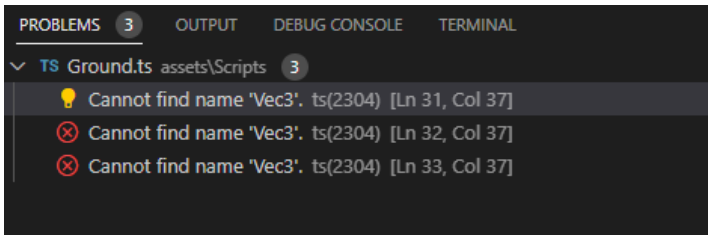
```
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

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

    @property({
        type: Node,
        tooltip: 'First ground'
    })
    public ground1: Node;

    @property({
        type: Node,
        tooltip: 'Second ground'
    })
    public ground2: Node;

    @property({
        type: Node,
        tooltip: 'Third ground'
    })
    public ground3: Node;
```

We also need a few values that we will need at multiple places in the code. This includes the width of the ground, the starting locations of the ground, and the speed at which they go. They are public, so

every method can use them if needed. Also, we need to make a temporary number for gameSpeed. We'll make it 50 for now and make a change later.

```
//Create ground width variables
public groundWidth1:number;
public groundWidth2:number;
public groundWidth3:number;

//make temporary starting locations
public tempStartLocation1 = new Vec3;
public tempStartLocation2 = new Vec3;
public tempStartLocation3 = new Vec3;

//get the gamespeeds
public gameSpeed: number = 50;
```

As you can see in the editor, there is an error with this code. The software probably showed you this:



That's because we need to tell it that **Vec3** is a part of Cocos Creator's API we'll use in the code. So at the import section at the top of the code, add **Vec3** so that the software knows this class.

```
import { _decorator, Component, Node, Vec3 } from 'cc';
const { ccclass, property } = _decorator;
```

Note: What is **Vec3**? It's the node's location in the x, y, and z axes. Though we won't use z in the project, most of the methods we'll use requires using the Vec3 class, so best to use and not worry about editing the z location in this tutorial.

Now we can go to the methods that will be used for ground. We only need three:

- **onLoad**: The first code read when the node is created. It's only used once during the node's life and starts up without you needing to call it.
- **startUp**: How everything should look when we start a new game. Ground nodes should go back to their original location.
- **update**: Every time the game updates a frame, these items must be done. This section will add the logic of moving the ground.

For onLoad, we are only doing a call to startUp. The startUp will also be called after the player restarts the game. Since **onLoad** is only called once when the node is built, we should make a new method to do the resetting.

```
//all the things we want to happen when we start the script
onLoad(){
    this.startUp()

}
```

For **startUp**, we need to get the ground's width to use that number for our movement update equation. We also need to reset the locations of the x coordinates of the ground images. The node's width is found in the **UITransform** properties of the node.

```
//preparing the ground locations
startUp(){

   //get ground width
   this.groundWidth1 = this.ground1.getComponent(UITransform).width;
   this.groundWidth2 = this.ground2.getComponent(UITransform).width;
   this.groundWidth3 = this.ground3.getComponent(UITransform).width;

   //set temporary starting locations of ground
   this.tempStartLocation1.x = 0;
   this.tempStartLocation2.x = this.groundWidth1;
   this.tempStartLocation3.x = this.groundWidth1 + this.groundWidth2;
```

Once again, we get a message that **UITransform** is not seen. So back to the top and add to the list of APIs.

```
import { _decorator, Component, Node, Vec3, UITransform } from 'cc';
const { ccclass, property } = _decorator;
```

Looking at the code above, you may be asking why we can't just directly update the positions for the ground and are using temporary properties. We can't because we don't have real access to the node's location. Some parts of a node are read-only and need another method call to change it. So by making a temporary copy of part of the node and editing that copy, we can use the copy on a method call to edit the location. Here's the final look at **startUp**. You can see we call setPosition to set the new position:

```
//preparing the ground locations
startUp(){

   //get ground width
   this.groundWidth1 = this.ground1.getComponent(UITransform).width;
   this.groundWidth2 = this.ground2.getComponent(UITransform).width;
   this.groundWidth3 = this.ground3.getComponent(UITransform).width;

   //set temporary starting locations of ground
   this.tempStartLocation1.x = 0;
   this.tempStartLocation2.x = this.groundWidth1;
   this.tempStartLocation3.x = this.groundWidth1 + this.groundWidth2;

   //update position to final starting locations
   this.ground1.setPosition(this.tempStartLocation1);
   this.ground2.setPosition(this.tempStartLocation2);
   this.ground3.setPosition(this.tempStartLocation3);

}
```

You might be asking why putting the grounds in a new setPosition even after doing all that work in the editor to put them in that position. This is because we want all the grounds to return to their original spots when the game resets. Also, doing it in the editor helped us to visualize the look before doing it in code and trying to guess what they should look like when we run it.

For **update**, we need to make a copy of the location of the ground's position, just like we did with the previous example. We'll be editing the temporary locations and putting the new location properties back onto ground nodes.

```
update(deltaTime: number) {

    //place real location data into temp locations
    this.tempStartLocation1 = this.ground1.position;
    this.tempStartLocation2 = this.ground2.position;
    this.tempStartLocation3 = this.ground3.position;

}
```

We also need to find out what the ground's moving speed is. That's found using **deltaTime** (the time between each update function call) and **gameSpeed** to help us tweak the final speed of the ground. The actual **gameSpeed** will be added later in our tutorial and be editable.

We can now move the x position of the temporary locations by the **gameSpeed** * **deltaTime** and finish the method.

```
update(deltaTime: number) {

    //place real location data into temp locations
    this.tempStartLocation1 = this.ground1.position;
    this.tempStartLocation2 = this.ground2.position;
    this.tempStartLocation3 = this.ground3.position;

    //get speed and subtract location on x axis
    this.tempStartLocation1.x -= this.gameSpeed * deltaTime;
    this.tempStartLocation2.x -= this.gameSpeed * deltaTime;
    this.tempStartLocation3.x -= this.gameSpeed * deltaTime;

}
```

Now we can add the logic of ensuring the ground resets to the back of the line when it goes off the screen. This requires us to know the size of the canvas. We are doing this with properties because in case later we need to change the size of the ground or screen size, we don't need to adjust all the numbers manually. It will do it all for us. This is done by talking to the director, who controls the game.

```
//get speed and subtract location on x axis
this.tempStartLocation1.x -= this.gameSpeed * deltaTime;
this.tempStartLocation2.x -= this.gameSpeed * deltaTime;
this.tempStartLocation3.x -= this.gameSpeed * deltaTime;


//get the canvas size prepared
const scene = director.getScene();
const canvas = scene.getComponentInChildren(Canvas);
```

Once again, we get an error saying it doesn't know the **director**. So we need to add it to the others on the top. The same goes for **Canvas**.

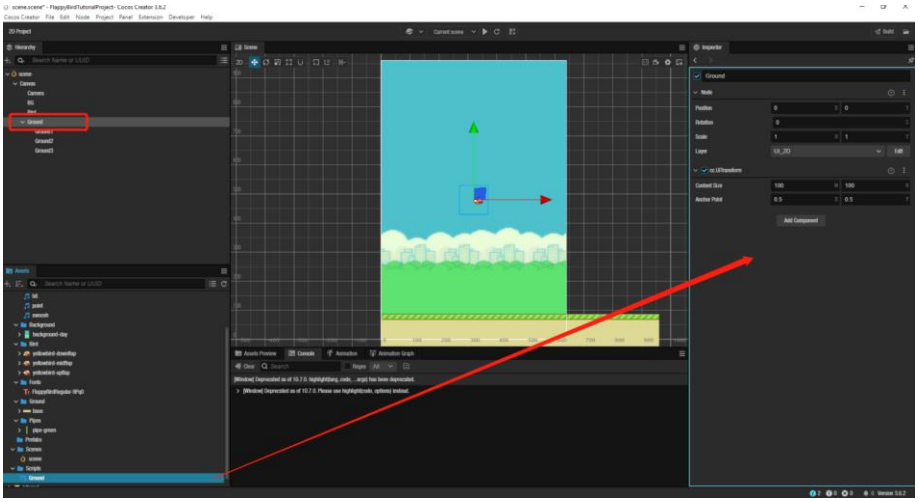
```
import { _decorator, Component, Node, Vec3, UITransform, director, Canvas} from 'cc';
const { ccclass, property } = _decorator;
```

Finally, we can set the positions of the actual ground nodes from the temporary locations using the **setPosition** method. Make sure this is all still in the **update** method.

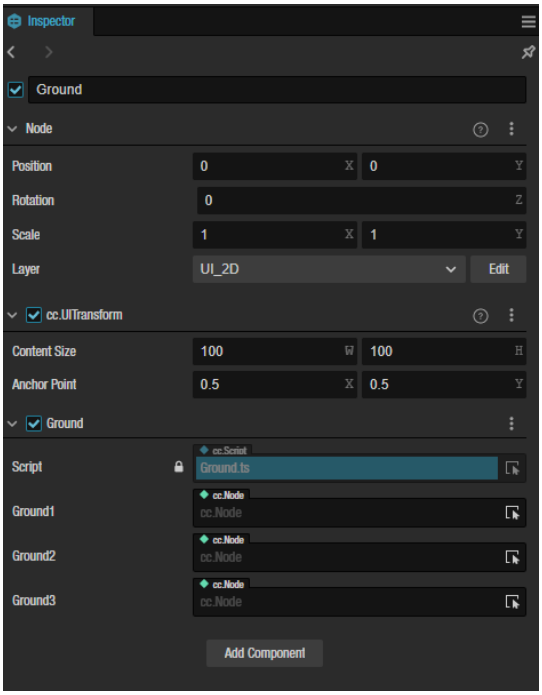
```
    //check if ground1 went out of bounds. If so, return to the end of the line.
    if (this.tempStartLocation1.x <= (0 - this.groundWidth1)) {
        this.tempStartLocation1.x = canvas.getComponent(UITransform).width;
    }

    // same with ground2
    if (this.tempStartLocation2.x <= (0 - this.groundWidth2)) {
        this.tempStartLocation2.x = canvas.getComponent(UITransform).width;
    }

    // same with ground3
    if (this.tempStartLocation3.x <= (0 - this.groundWidth3)) {
        this.tempStartLocation3.x = canvas.getComponent(UITransform).width;
    }

    //place new locations back into ground nodes
    this.ground1.setPosition(this.tempStartLocation1);
    this.ground2.setPosition(this.tempStartLocation2);
    this.ground3.setPosition(this.tempStartLocation3);

}
```

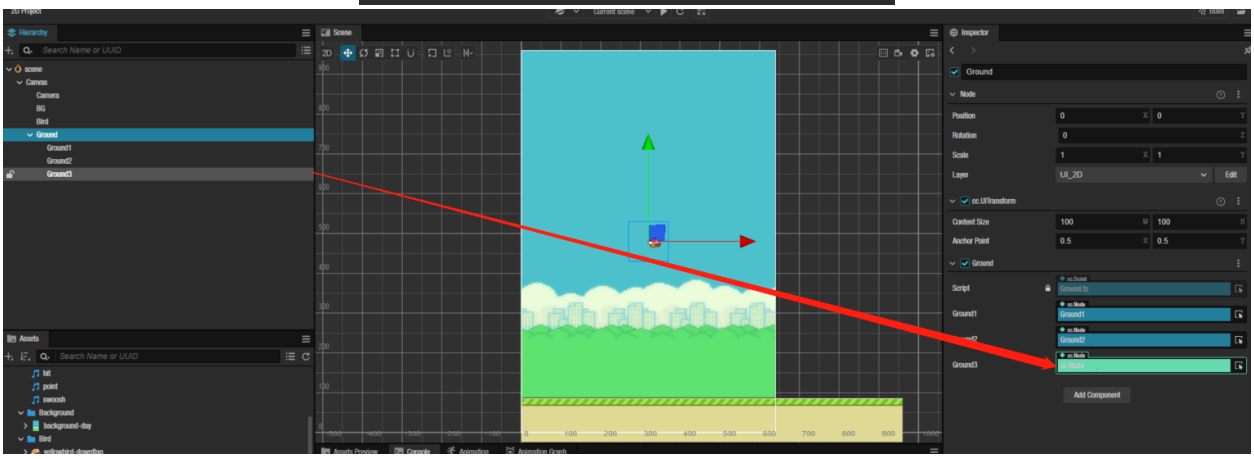
And there you go. That should be **Ground.ts** all complete except for the temporary speed we'll fix in a moment. Save it, and let's move to the next stage.

Back in Cocos Creator, we need to add this code to the **Ground** parent node. You can select the Ground node and drag the **Ground.ts** into its Inspector. This adds the code as a component of the node.

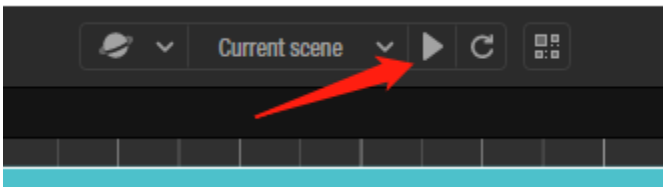


Look at your Inspector with Ground highlighted. It asks for three properties we built in the code: **Ground1**, **Ground2**, and **Ground3**. Add the three nodes inside the ground node to their corresponding place.



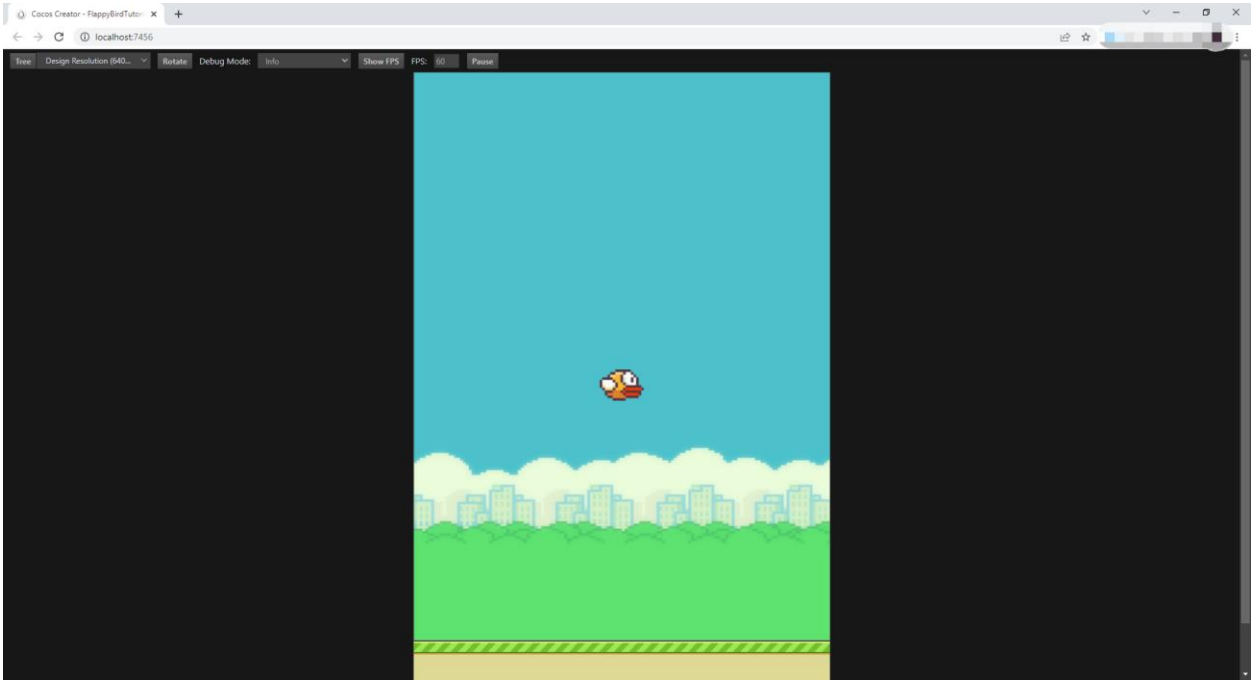


Let's now save our project by saving the **scene**. We have now completed the first part. Let's press the play button on the top-middle area of Cocos Creator and see if it runs.



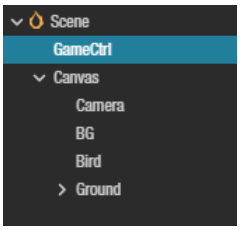
When you press this, it should open a new Google Chrome tab. This tab will have a localhost, meaning it runs on your personal computer, not the internet.

When you look at it this first time, it should have a moving background and the bird in the middle of the screen. Let's move to the next challenge.

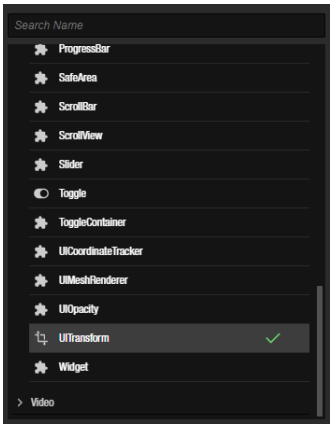


Part 4 - Making The Brains

We have our first piece of code in, but now we need a primary location that does checks on the ground and everything else. We need a brain in the project to manage all the actions happening. So let's make one. We can go back into Cocos Creator, create a new node, and call it **GameCtrl**. Make sure it's not part of the Canvas because it has no visual elements.



Let's give it a **UITransform** component by going into the Inspector clicking on "**New Component**" and selecting UI -> UITransform. We'll use this later for clicking or tapping on the screen.



Right-click on the scripts folder and add a new TypeScript component. Call it GameCtrl.

GameCtrl.ts job is to talk to all the parts of the game and tell them what to do and when. So it has many jobs:

- Start the game.
- Listen for mouse clicks.
- Tell the ground when to start, stop, and reset movement when the bird hits pipes or the ground or a new game starts.
- Tell the bird it's on the next frame so it can continue to fly or drop.
- Tell the bird if there is a tap so it can start to fly.
- Tell the pipes to start moving on the next frame or not.

- Ask if the pipes are off the screen to delete them and bring a new set of pipes at random gaps and locations.
- Ask if the bird passed through the pipes.
- Tell the score if the bird passed the pipes so it can give it a point.
- Stop everything and display the game over and high score.
- Reset the game after a mouse tap.
- Hold all the editable parts of the game (speeds)

Though it looks like a lot, we won't do all of it now. Just go through the list one at a time. Let's start by telling the ground to start, stop, or reset movement.

So let's once again go into the code and delete everything in the ccclass.

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

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

}
```

From here, we need to add a property that will represent the ground as well as the speed of the ground.

```
@ccclass('GameCtrl')
export class GameCtrl extends Component {

    @property({
        type: Component,
        tooltip: 'Add ground prefab owner here'
    })
    public ground: Ground;

    @property({
        type: CCInteger,
        tooltip: 'Change the speed of ground'
    })
    public speed: number = 200;

}
```

We need to fix a few issues before writing the code. First, the easy one. It doesn't know what **CCInteger** is. We built an integer-only number system that you can use for editing the speed. We can fix this like last time by adding the API to the top.

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

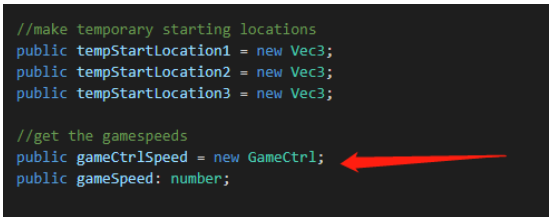
The second issue is that we are asking it to make a component of the **Ground** node. We are doing this to allow **gameCtrl.ts** to communicate with the ground node to reset the ground when the game is over from the **gameCtrl.ts** code as well as the ground speed. You will need to do something special to add knowledge of this code. We'll have to import the code. This is done above the ccclass.

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

import { Ground } from './Ground';

@ccclass('GameCtrl')
export class GameCtrl extends Component {
```

This says to add the **Ground** ccclass code from the **Ground.ts** file found at this location. After doing that, you should see no more issues with making a call to **Ground** methods. We can now do the same thing for the **Ground.ts**, so it recognizes **GameCtrl.ts**. Now we can change that temporary number we had earlier.

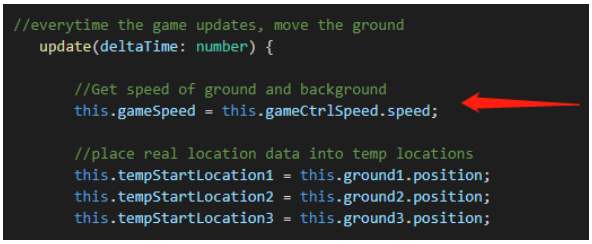
Open **Ground.ts** and add **GameCtrl.ts** in the same way.

Let's go back to the properties we built in **Ground.ts** and add a **gameCtrlSpeed** property next to **gameSpeed**.

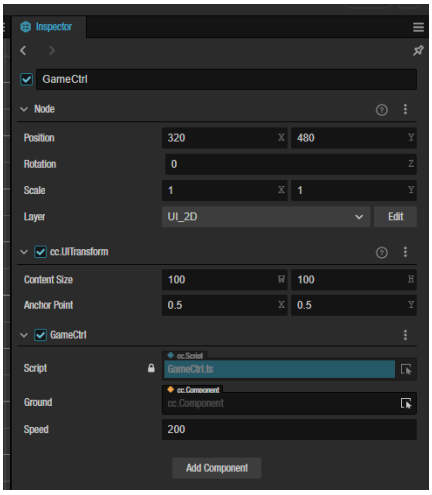


We are asking to build a new **GameCtrl** because this is a copy of **GameCtrl**. Because we need a container that holds the properties of the **GameCtrl**.

Now change the **gameSpeed** in the update method to use the speed from the **GameCtrl** properties. Remember, speed is the **CCInetger** we just added to the **GameCtrl**.



Save all your code. Now when we go back to Cocos Creator, add the **GameCtrl.ts** component to the **GameCtrl** node just like we did with the **Ground** node. Now we can edit the speed of the ground in the Inspector and not have to change it from the code.



Now that this is built, let's quickly go back to **GameCtrl.ts** and add to the code. We will add **pipeSpeed** for now and use it when we get to the pipes.

```
@property({
  type: CCInteger,
  tooltip: "Change the speed of pipes",
})
public pipeSpeed: number = 200;
```

We can now add these three methods to prepare for the next steps: **onLoad**, **initListener**, and **startGame**.

- **onLoad:** Loads all the instructions when the game starts. Since **GameCtrl** is controlling everything, it should be initializing many things not already initialized here.
- **initListener:** will be for listening for the keyboard/mouse inputs.
- **startGame**: What needs to be done to reset everything to the start of a new game and starts the game.

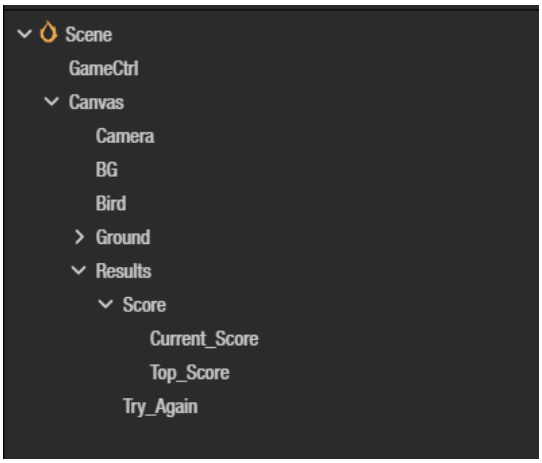
```
 //Things to do when the game loads
 onLoad(){

 }

//listener for the mouse clicks and keyboard
initListener() {

}

//What to do when the game is starting.
startGame() {

}
```

For now, we don't need to fill out anything in these. But we can work on something that allows you to press a button and add points to your bird. This will help us to know that the **initListener** is working and let us start a new game. Also, we can start getting the framework on points scoring in the process of building this.

Make sure to save your code and the scene now. Now let’s get to those points.

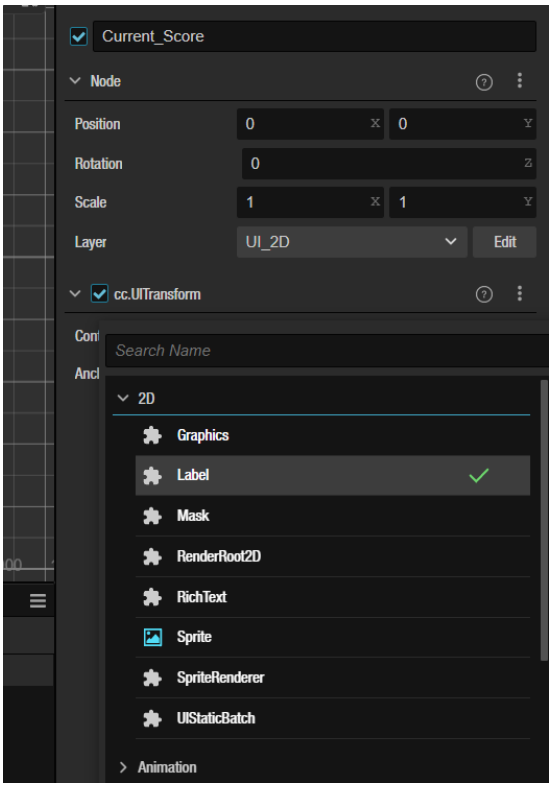
Part 5 - Building Points

One of the easier things we can do now is to add the points label and the text you get when it's game over. To start, we need to build the visuals in the editor. In Cocos Creator, let's add a node called "Results" in the **Canvas**. Within it, we have a few items, the current score, the high score, and the text that goes along with the game over. Check how I set it up in the image below:

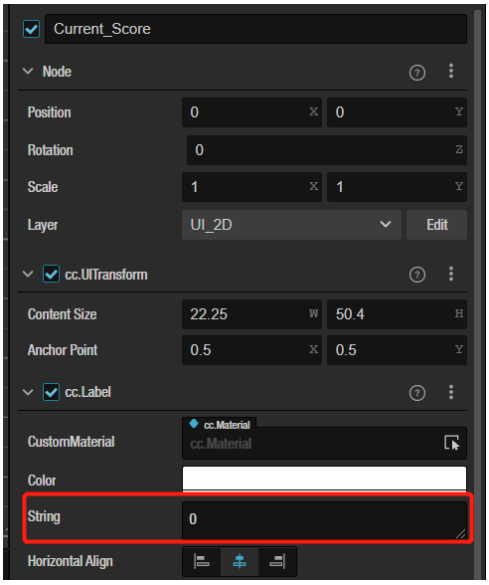


The Results contain four items: **Score** (holding the two nodes **Current_Score** and **Top_Score**) and **Try_Again.**

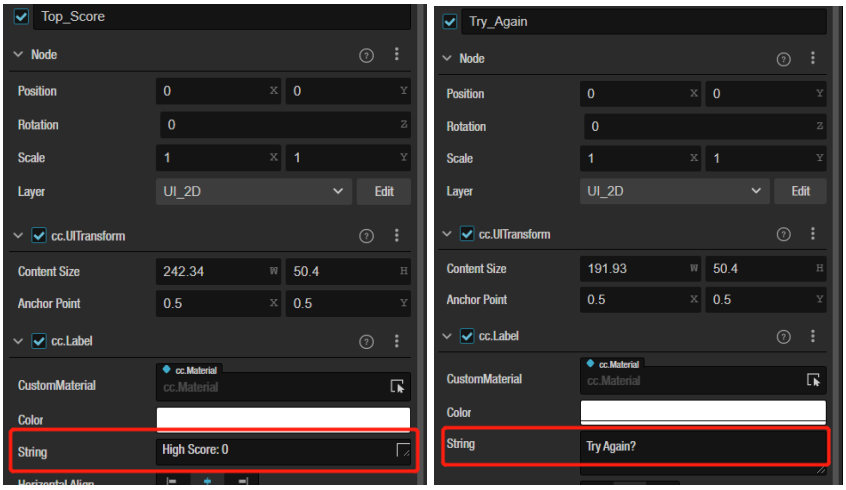
Let's make all of them, except **Score**, labels by adding a component, 2D -> Label.



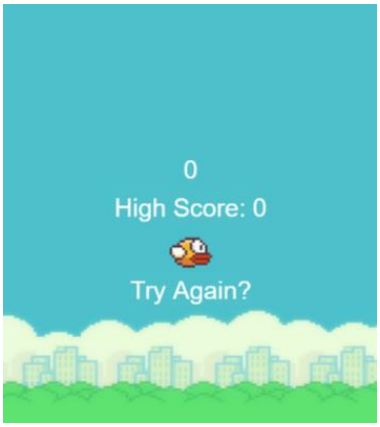
In the String section of **cc.Label**, we can go ahead and place "0" for the text of **Current_Score**.



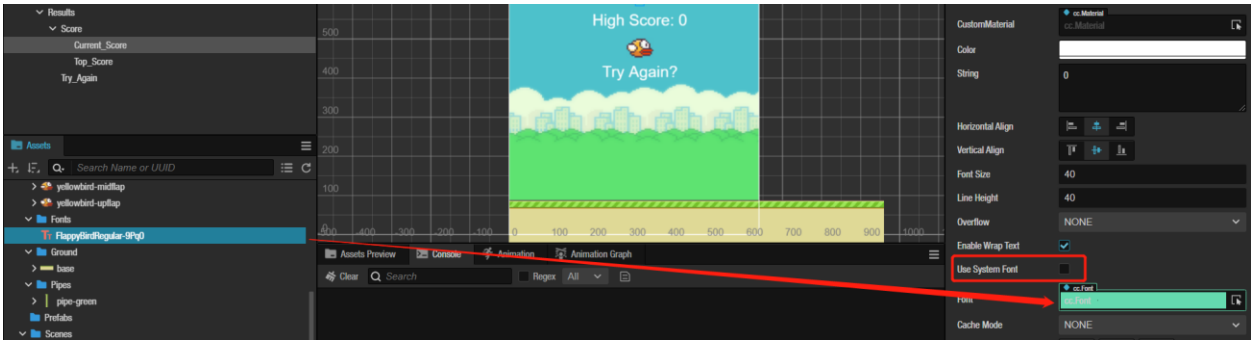
We can also have the String "High Score: 0" for the **Top_Score**, and "Play Again?" for **Try_Again**.



Move the nodes to make them look like they would if it was the actual game.

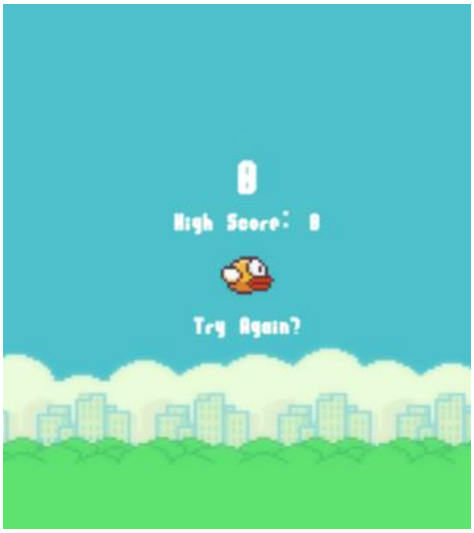


The next issue is that the font is wrong. To get the font to look like the original, you can disable "Use System Font" in the **cc.Label** component and drag in the font we grabbed earlier when gathering our assets. Do this for all of the labels.

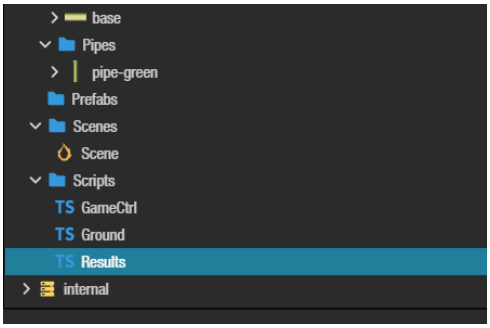


It should look like this, but you can play around with the Color, Size, and other items to make it look the way you want.

Note: Best to change the Line Height to be the same number as the Font size for no clipping.



Now we need a script to make the high score and "try again?" appear, disappear, and change the high score during gameplay. So we need to build a Results.ts file in the script directory.



Let's open it up, clean it up like the other two and add a few methods and properties.

The first thing is making three properties for the score, high score, and try again labels.

```
@ccclass('Results')
export class Results extends Component {

    @property({
        type: Label,
        tooltip: 'Current Score'
    })
    public scoreLabel: Label;

    @property({
        type: Label,
        tooltip: 'High Score'
    })
    public highScore: Label;

    @property({
        type: Label,
        tooltip: 'Try Again?'
    })
    public resultEnd: Label;

}
```

Once again, we need to add **Label** to the top of the code as the code doesn't recognize the Label property.

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

We also need to setup a variable for **maxScore** that saves the high score, and **currentScore** that is the score during the gameplay.

```
@property({
    type: Label,
    tooltip: 'Current Score'
})
public scoreLabel: Label;
@property({
    type: Label,
    tooltip: 'High Score'
})
public highScore: Label;
@property({
    type: Label,
    tooltip: 'Try Again?'
})
public resultEnd: Label;

//variables needed for the scores
maxScore: number = 0; //saved high score
currentScore: number; // current score while playing
```

Now we can start to create our methods. We will need at least five methods:

- **updateScore**: in charge of updating and displaying the current score.
- **resetScore**: Resets the score back to 0.
- **addScore**: add one point to the score.
- **showResults**: Show the results after a game is over.
- **hideResults**: Hide the results when you start a new game.

So let's look at the code. First, **updateScore** will check what the current score is and display it on the screen. It will ask you to bring a number to the method. This number is the newly displayed score. This is useful as addScore will add a point to the currentScore property when the bird passes the pipes and tell updateScore to display the new score number in currentScore.

We also have the **scoreLabel** change the text automatically by calling the **string** call. We add an empty ellipse because we can't add the property alone. Now it will automatically change for the player to see.

```
//change current score to new score or back to zero then display the new score
   updateScore(num:number){

        //update the score to the new number on the screen
        this.currentScore = num;

        //display new score
        this.scoreLabel.string = ('' + this.currentScore);
    }
```

Now we can reset the score with **resetScore**. Resetting the score asks the **updateScore** to bring the score back to 0, hide the "game over" items, and display the current score as 0.

```
//resets the score back to 0 and hides game over information
resetScore(){

    //reset score to 0
    this.updateScore(0);

    //hide high score and try again request
    this.hideResult();

    //reset current score label
    this.scoreLabel.string = ('' + this.currentScore);

}
```

addScore adds one point to the **currentScore** and updates the score in one quick line of code. Nice!

```
//add a point to the score
addScore(){

    //add a point to the score
    this.updateScore(this.currentScore + 1);
}
```

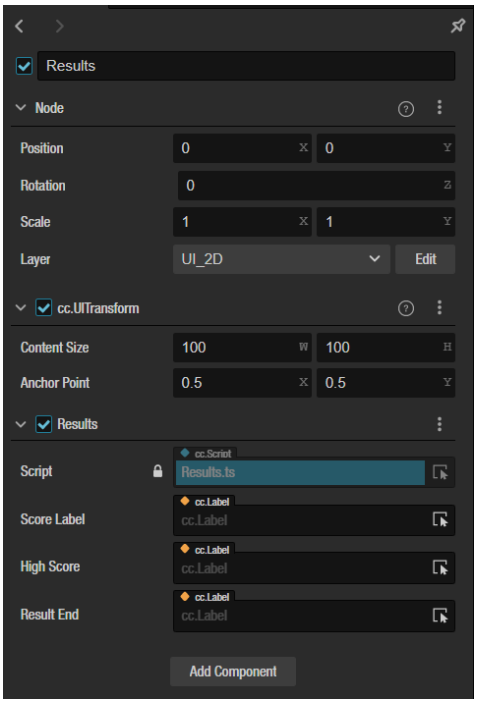
showResults will be called when the game is over. It first checks to see if this new score is higher than the **maxScore** using **Math.max**. We then update the label for the high score, display it, and that tells the player if they got the high score. It also activates our "label" for "Try Again?"

```
//show the score results when the game ends.
showResult(){

    //Check if it's the high score
    this.maxScore = Math.max(this.maxScore, this.currentScore);

    //Activate high score label
    this.highScore.string = 'High Score is:' + this.maxScore;
    this.highScore.node.active = true;

    //Activate try again label
    this.resultEnd.node.active = true;

}
```

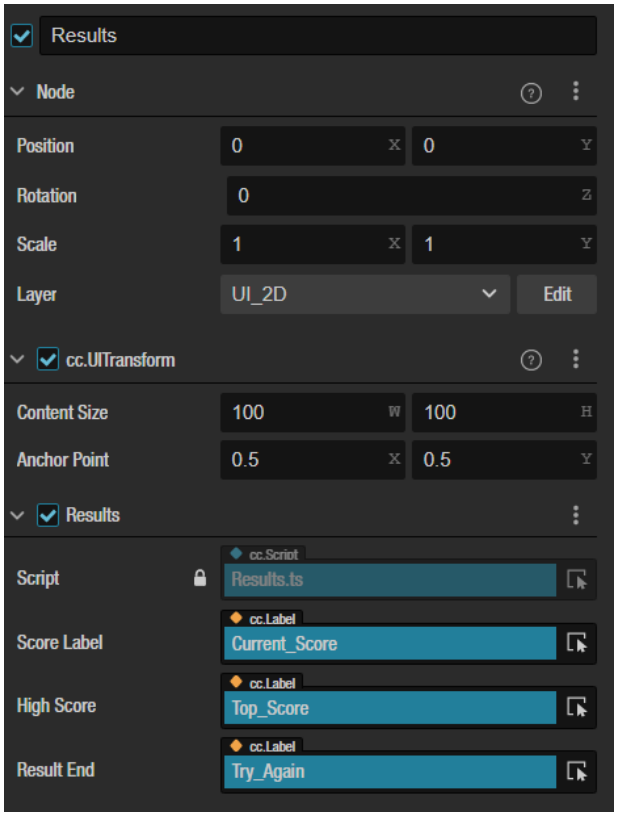
We then end with **hideResults** which hides the **highScore** and **resultsEnd** nodes.

```
//hide results and request for a new game when the new game starts
hideResult(){

    //hide the high score and try again label.
    this.highScore.node.active = false;
    this.resultEnd.node.active = false;
}
```

And that's it. We have our script. Save it, and let's return to Cocos Creator and add this script to the results node as a component.



Now drag the Current_Score and Top_Score nodes into the script and the Try_Again node.



Add Results.ts To GameCtrl.ts

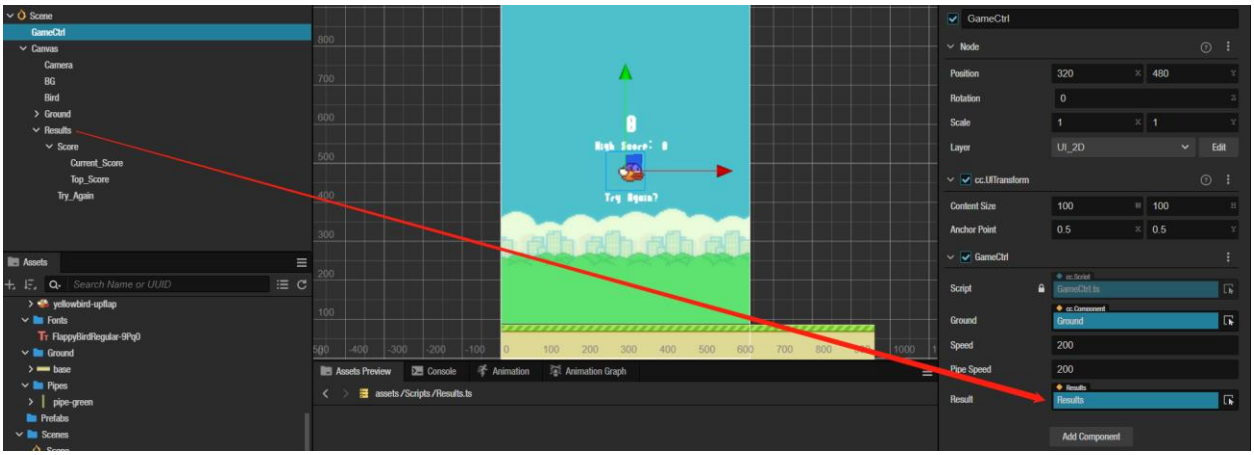
Now that the first part is done let's add it to **GameCtrl**. Go back to **GameCtrl.ts,** and let's make a new property of the type **Results** called **result**.

```
@property({
  type: Results,
  tooltip: "Add results here",
})
public result: Results;
```

We also need to add the code of results to GameCtrl just like we did with the ground, so go to the top and add it the same way we did the **Ground**.

```
import { Ground } from './Ground';
import { Results } from './Results';
```

Let's save now and add the Results to our **GameCtrl** node in Cocos Creator and save it.



Now we can go back to the code and to the initListener we talked about and build it to test the score. First, let's use the input call that watches all inputs. We'll have it look for a key to be pushed down, and if it is, call a method that tells it what to do.

```
//listener for the mouse clicks and keyboard
initListener() {

    //if keyboard key goes down, go to onKeyDown
    input.on(Input.EventType.KEY_DOWN, this.onKeyDown, this)

}
```

Again, we need to add some APIs from cc, so go back to the very top and add it.

```
import { _decorator, Component, Node, CCInteger, Input, input } from 'cc';
const { ccclass, property } = _decorator;
```

Input is the full-functioning listener of the input events. In contrast, input is a singleton (single instance) of the Input class.

Looking back at initListener, we can see we are looking for an input of a **KEY_DOWN** being turned on. When a key is pressed, we want the method **onKeyDown** to initiate and tell it which key it was.

In the **onKeyDown** method, we build after **initListener**, add the event, and do a keyCode check using a Switch/case method. We'll set the key "A" for game over, "P" to add points automatically, and "Q" to reset the game.

```
//for testing purposes, we use this. But hide as comments after you are successful
onKeyDown(event: EventKeyboard) {

  switch (event.keyCode) {
    case KeyCode.KEY_A:
      //end game
      this.gameOver();
      break;
    case KeyCode.KEY_P:
      //add one point
      this.result.addScore();
      break;
    case KeyCode.KEY_Q:
      //reset score to zero
      this.resetGame();
  }

}
```

***PLEASE NOTE*: After the game is complete, delete or comment out this section so gamers can't cheat your game.**

Once again, we are using the new API **KeyCode** and **EventKeyboard**. So add them to the top.

```
import { _decorator, Component, Node, CCInteger, Input, input, EventKeyboard, KeyCode } from 'cc';
const { ccclass, property } = _decorator;
```

EventKeyboard tells us this event is a Keyboard event. **KeyCode** is an enum of all the keys on a keyboard. So we'll need it to say which key is pressed from the **EventKeyboard** class.

We now have two problems. We don't have a **gameOver** or **resetGame** method. So let's build them now.

For the **gameOver**, let's make one that shows the results. This also should pause the game, so nothing is moving until we reset the game. So let's call for the **director** (The controller of the entire game looping) to pause everything.

```
//When the bird hits something, run this
gameOver() {

    //Show the results
    this.result.showResult();
    //pause the game
    director.pause();
}
```

Once again, we get an error that the code doesn't know what the **director** is. So back to the cc APIs at the top to add the director. We'll do this much more, so don't get discouraged. This can happen with many games that use many APIs in one set of code.

```
import { _decorator, Component, Node, CCInteger, Input, input, EventKeyboard, KeyCode, director } from 'cc';
const { ccclass, property } = _decorator;
```

Now if you press Q, it should call the **resetGame**. But we also need to build this. Let's add it after the gameOver method and have it call the results to reset the score and start the game again.

```
//When the game starts again, do these things.
resetGame() {
    //Reset score, bird, and pipes
    this.result.resetScore();

    //Get objects moving again
    this.startGame();

}
```

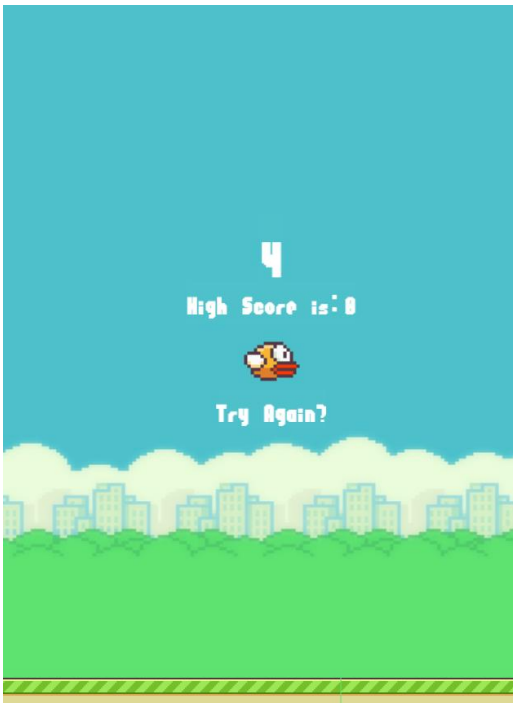
You can recall we built a **startGame** method but with nothing inside. So let's add some code to it, have it hide the results, and tell the director to resume the game.

```
//What to do when the game is starting.
startGame() {

     //hide high score and other text
     this.result.hideResult();

     //resume game
     director.resume();

}
```

Finally, we need to have the **initListener** running when we start the game. So go back to the **onLoad** method in **GameCtrl.ts** and add it, reset the score to zero, and have the **director** pause the game so we can start from a paused position.

```
//Things to do when the game loads
onLoad() {

  //get listener started
  this.initListener();

  //reset score to zero
  this.result.resetScore;

  //pause the game
  director.pause;

}
```

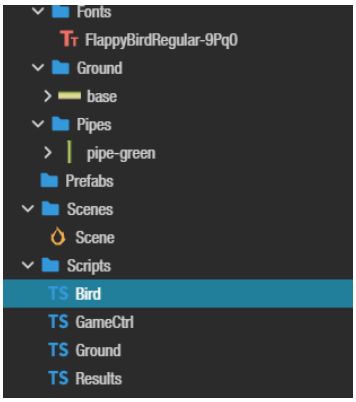
Now let's save this and run it to see if the scores are being counted. Go ahead and run it, click on the game, and press the P key. You should see that the scores are going up, and pressing the A key brings a high score and a "try again?" label. Pressing Q will reset the score to 0 and start the game again while hiding the other text. This can be done again and again with no issue. If there is, go back and check your code.



Part 6 - Moving the Bird

Now we start to get into the more intermediate coding. This first part is getting the bird to actually fall, fly, and have animations for the bird. So let's start with the flying first because if we let it fall, we can't see where it is after a few seconds of testing.

Let's build a **Bird.ts** file in the assets folder, just like the others, and clean it out.



We need to know a few things about our bird: How high it flies and for how long. Let's make properties for these and make them **CCFloat**, as we will be using decimals and need float numbers. Let's also give each property a number. We'll change in Cocos Creator.

We need a variable to hold the animation of the bird. (We'll explain later) and we also need to make a temporary location for the bird, just like we did with the ground. Also, import both **CCFloat** and **Vec3** APIs into 'cc' like usual.

```
import { _decorator, Component, Node, CCFloat, Vec3 } from 'cc';
const { ccclass, property } = _decorator;

import { GameCtrl } from './GameCtrl';

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

    @property({
        type: CCFloat,
        tooltip: 'how high does he fly?'
    })
    public jumpHeight: number = 1.5;

    @property({
        type: CCFloat,
        tooltip: 'how long does he fly?'
    })
    public jumpDuration: number = 1.5;

    @property({
        type: GameCtrl,
        tooltip:'Add GameCtrl here'
    })
    public game: GameCtrl;

    //Animation property of the bird
    public birdAnimation: Animation;

    //Temporary location of the bird
    public birdLocation: Vec3;

}
```

Now we can build the methods we will need for this part:

- **onLoad**: Things the bird does when the game starts
- **resetBird**: All the things that happen when we reset the game.
- **fly**: What happens when a bird flies

We've discussed what **onLoad** does, so we'll talk about what's happening inside. We want the bird to reset its original position, and we want to add animation. Again, we do this so we can make a temporary version of the animation to play with, as the component is primarily read-only.

```
//all the actions we want done when we start the script.
onLoad(){

    //Restart the bird
    this.resetBird();

    //Get the initial animation information
    this.birdAnimation = this.getComponent(Animation);

}
```

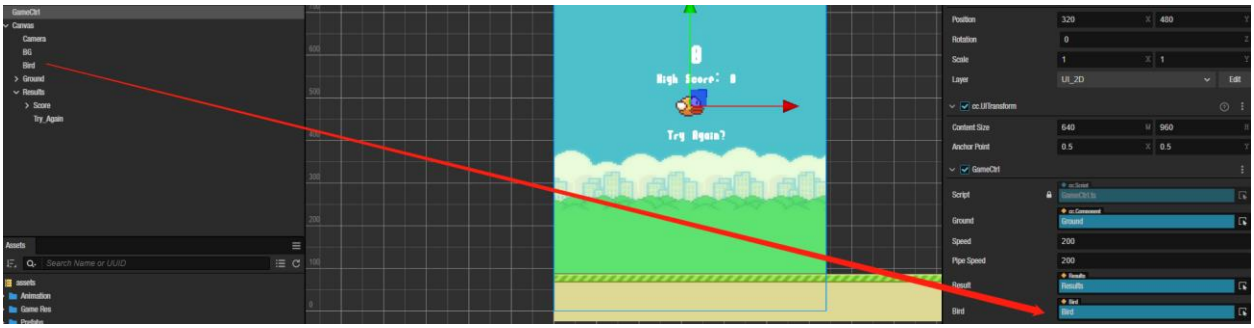
We are asking for the game's bird is because we are adding the **Bird** node itself to the **GameCtrl** so it can control the bird for us.

Let's go to **GameCtrl.ts** and finalize that. We'll add the import **Bird.ts** and add a component node to the code.

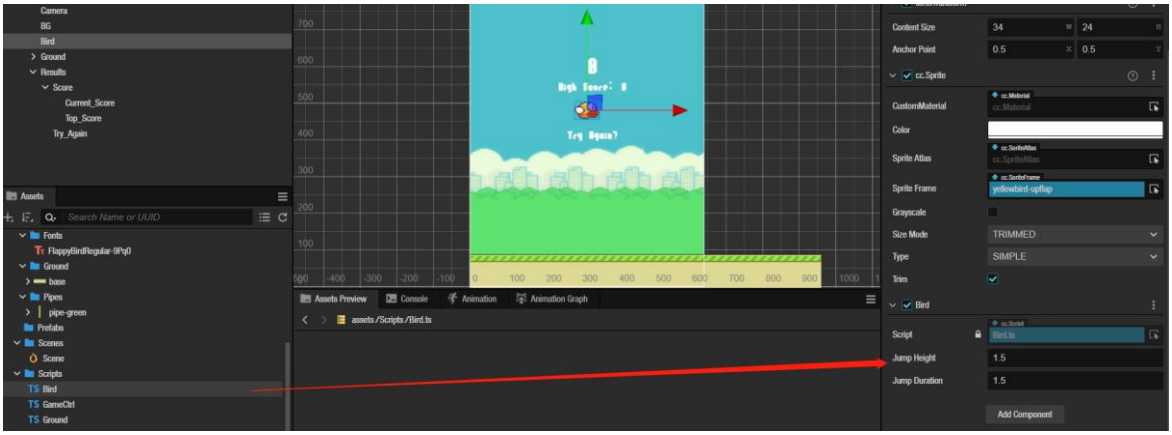
```
import { Ground } from './Ground';
import { Results } from './Results';
import { Bird } from './Bird';
```

```
@property({
  type: Bird,
  tooltip: "Add Bird node",
})
public bird: Bird;
```

Save all of this code, and let's add the Bird node to the GameCtrl component.



Let's also add the **Bird.ts** as a component of the **Bird** node.



Let's go to the **resetBird** in **Bird.ts.** We can put the bird back in its original place by giving the temporary location a new Vec3 variable in the middle of the game(0,0,0) and then set the bird's node location back to that when we reset the game.

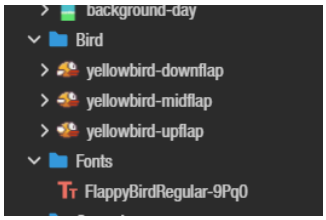
```
resetBird(){

    //create original bird location
    this.birdLocation = new Vec3(0,0,0);

    //place bird in location
    this.node.setPosition(this.birdLocation);

}
```

Before we can take a deeper look at how we add some animation to this bird, let's add the **Animation** API needed for it to work.

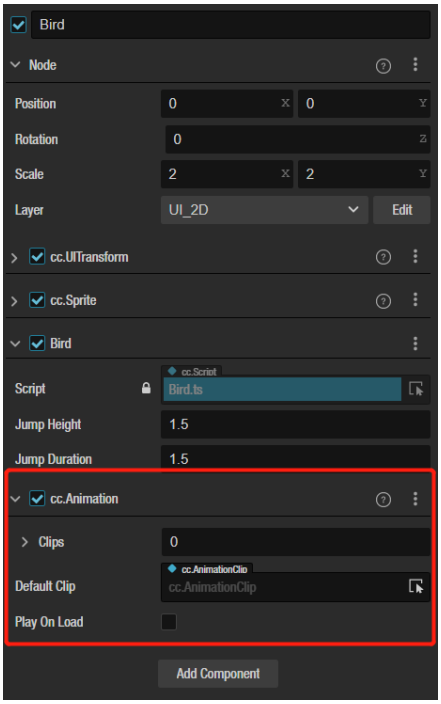
```
import { _decorator, Component, Node, CCFloat, Vec3, Animation } from 'cc';
const { ccclass, property } = _decorator;
```

Adding animation to a node

So we don't want to have the bird fly without some animation to show off how it's flying. We also can see in our assets that there are a few images for flying.



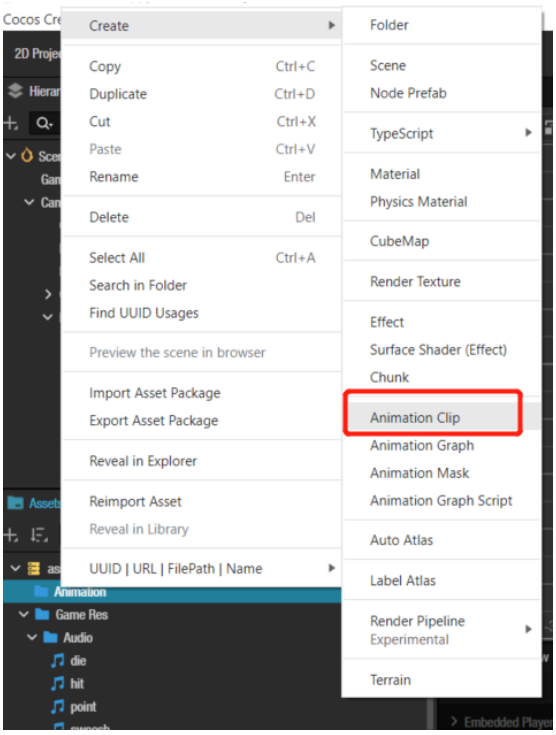
So let's build it. In Cocos Creator, highlight the bird and add a component to it. Let's add **Animation** -> **Animation**.



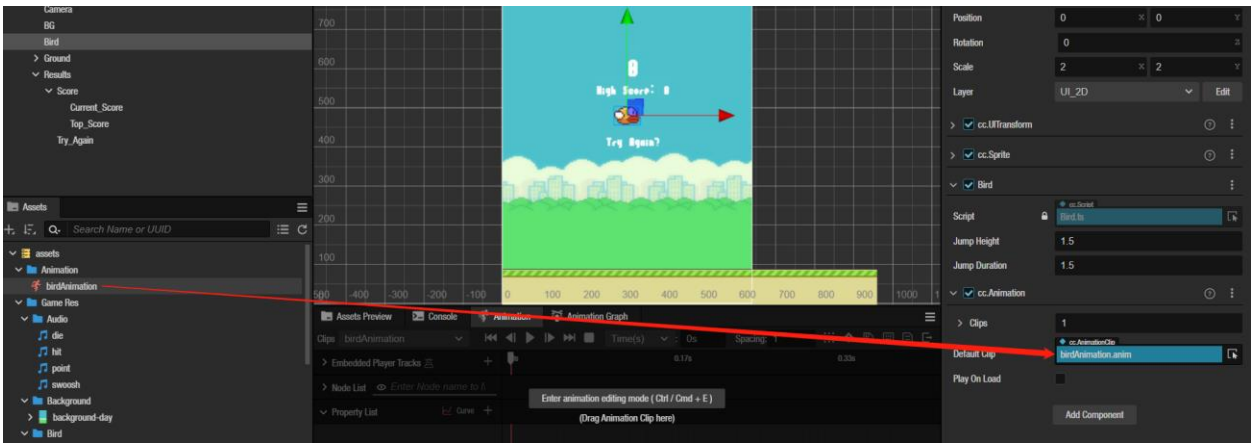
As you can see, it asks you for how animation clips you want (fly, die, explode, etc.) and the default clip. Because we just want to fly in this tutorial, we will only use the Default Clip. We'll now have to make an animation clip.

In the middle of Cocos Creator, there is a tab called Animation. Click on it and enter the editing mode.

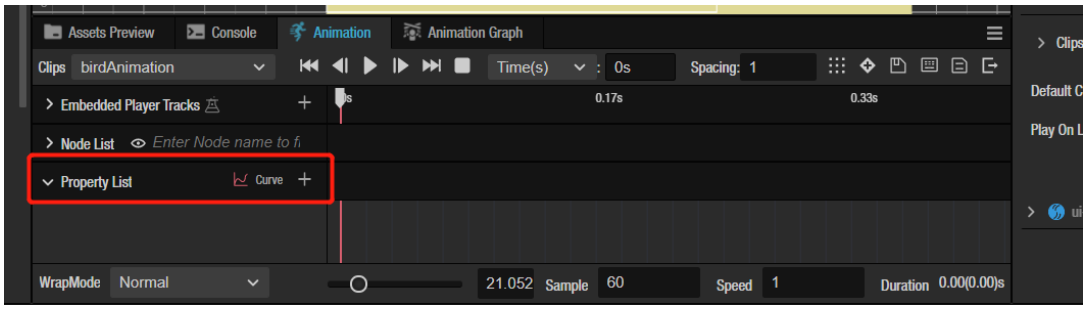
We currently don't have a clip, so in our assets area, let's add an **AnimationClip** in the Animation folder. Make sure to name it '**birdAnimation'**.



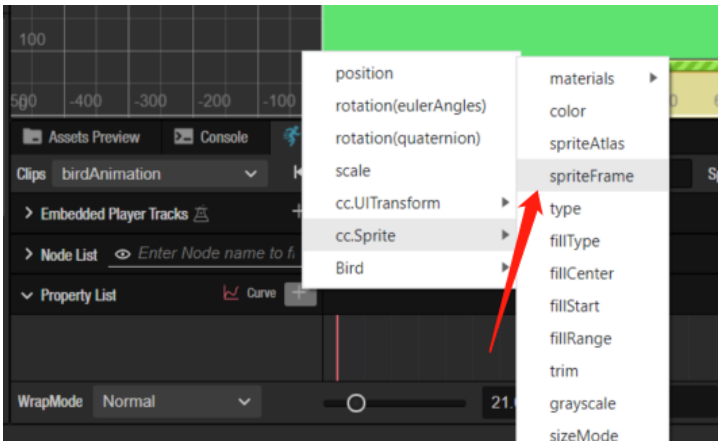
Now click on the Bird node, and let's move **birdAnimation** to the Animation component.



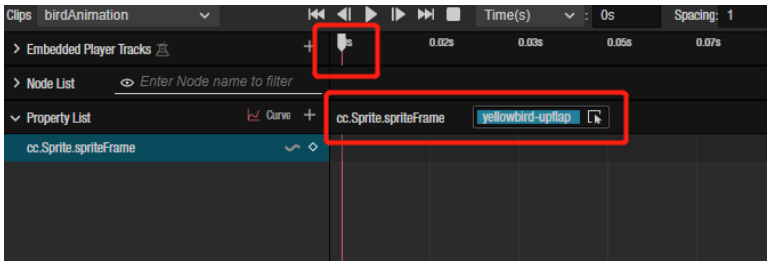
We can now enter the animation area. We could play in a few places, but we want to go to the **Property List**.



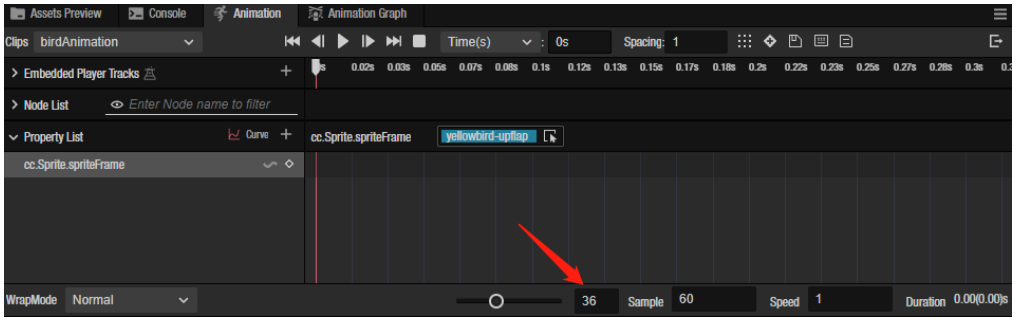
We want to change the sprite-frame. The sprite frame is what is holding the image of the node. So if we go to the Property List, click the "+" button, and choose the cc.Sprite -> **spriteFrame**



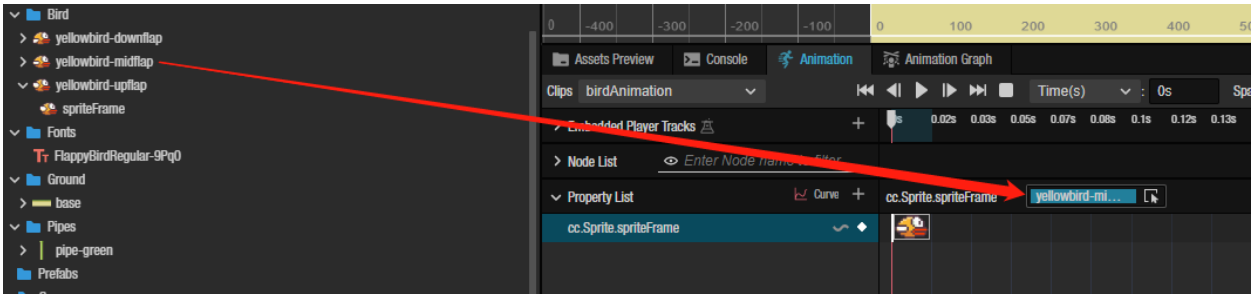
When you click on the first **cc.Sprite.spriteFrame**, you should see a section above that lets you see which spriteFrame is currently in use and a progress bar above it that enables you to move the timeline.



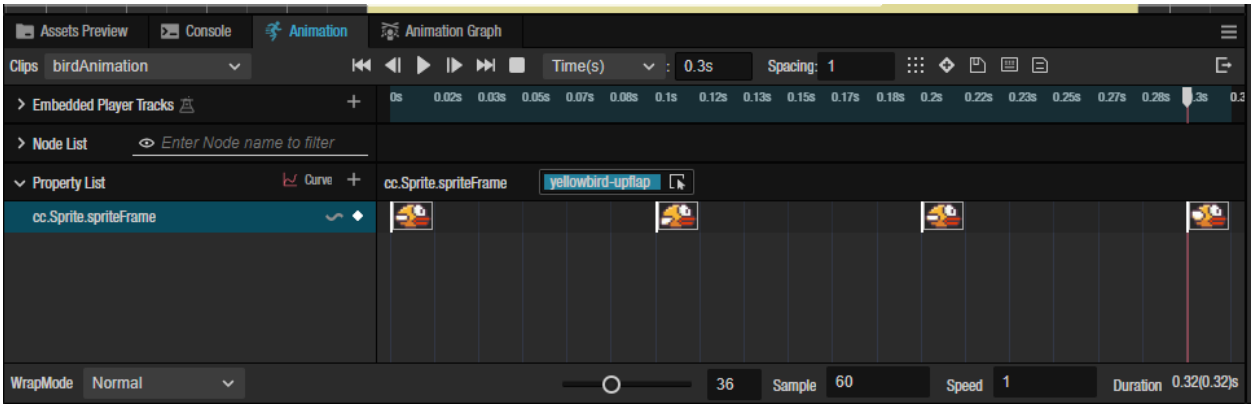
Because this is a fast flap, we only need a few milliseconds. Mine is timed to .3 seconds. So on the bottom, change the zooming number to 36.



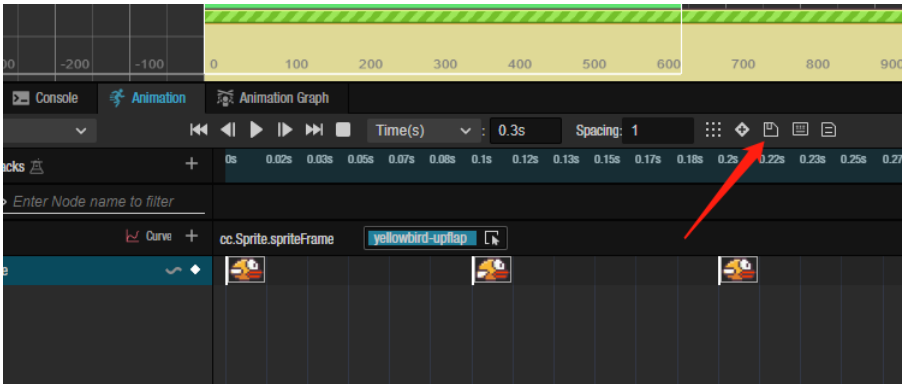
Drag the mid-flap image onto the decorator area of **cc.Sprite.spriteFrame**. You can see it gives you a small image to help you follow along with what you are doing.



Next, move the progress bar a few frames and add the down-flap to the same place we placed the first **spriteFrame**. Again with the midflap and end with the upflap again. This should take a total of 0.3 seconds. Your's should look similar to this.



Go ahead and save this by clicking on the save button in the upper right. Also, save the entire scene.



If you want to see it, you can press the play button, which will show on the **Scene**. It is pretty fast, so press it a few times to get the look. We could do more to the animation, but that's for another tutorial.

You can exit the animation by pressing the exit door on the top right.

Now back to **Bird.ts**, let's work on the fly method. This will not only move the bird but also call for animation. First, we want to call on the animation to stop so that we know that every time we click, there won't be multiple animations going on at once. We also want it to start over.

```
fly(){

    //stop the bird animation immediately
    this.birdAnimation.stop();

    //play the bird animation
    this.birdAnimation.play();

}
```

Adding Tween to the bird

Next, we'll make the bird move position. We want the movement to be pretty beautiful, not just linear movement going up. To do this, we're going to be using a **Tween**. You can search about it online, but it means how fast and slow you go from one point to the other, and there are many ways to do so. You can check the Cocos documentation for a few examples on a time graph. Not all are supported, but a lot are.

https://docs.cocos.com/creator/3.6/manual/en/tween/tween-function.html

Before we begin, adding **tween** to the API calls we have for **Bird.ts**.

```
import { _decorator, Component, Node, CCFloat, Vec3, Animation, tween } from 'cc';
const { ccclass, property } = _decorator;
```

Let me show you the final product first so you can understand what we will be talking about:

```
//start the movement of the bird
tween(this.node.position)
    .to(this.jumpDuration, new Vec3(this.node.position.x, this.node.position.y + this.jumpHeight, 0), {easing: "smooth",
        onUpdate: (target:Vec3, ratio:number) => {
            this.node.position = target;
        }
    })
    .start();
```

To do a tween, we need to get the bird's position first. Then we can use a few methods to make the movement happen within the tween. For this tutorial, we only need two, **.to** and **.start**.

.to will tell it where we want to go. It asks for two things by default but can ask for three. For our tutorial, we'll ask for these three things.

1. Jump duration
2. New location
3. How you ease into the final location

The first one is the **jumpDuration** we made earlier in the decorator section. The second is the new location for the bird. We can use the same x position since flappy doesn't move left or right, add the jump height to y, and set z to 0 since we don't use z in this game. Next, we'll be asked how we ease into the end location. I'm choosing “smooth”, but you can select any of your choosing.

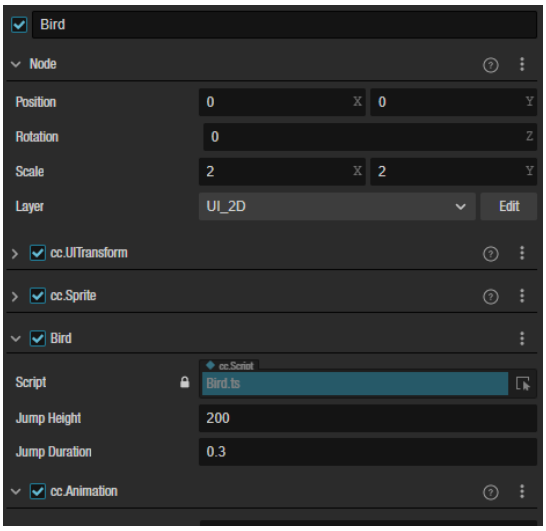
Don’t worry about the **onUpdate** area. This is just telling the tween where we were initially. Very rarely do you need to edit this part when doing your tweens.

Finally, we'll end with a **.start** method to tell the tween it can start.

Now we can add this to the **fly** method.

```
fly(){

    //stop the bird animation immediately
    this.birdAnimation.stop();

    //start the movement of the bird
    tween(this.node.position)
        .to(this.jumpDuration, new Vec3(this.node.position.x, this.node.position.y + this.jumpHeight, 0), {easing: "smooth",
            onUpdate: (target:Vec3, ratio:number) => {
                this.node.position = target;
            }
        })
        .start();

    //play the bird animation
    this.birdAnimation.play();

}
```

Now let's test if this works. Go back to Cocos Creator, and in the Bird node, set the height as 200 and duration as 0.3.



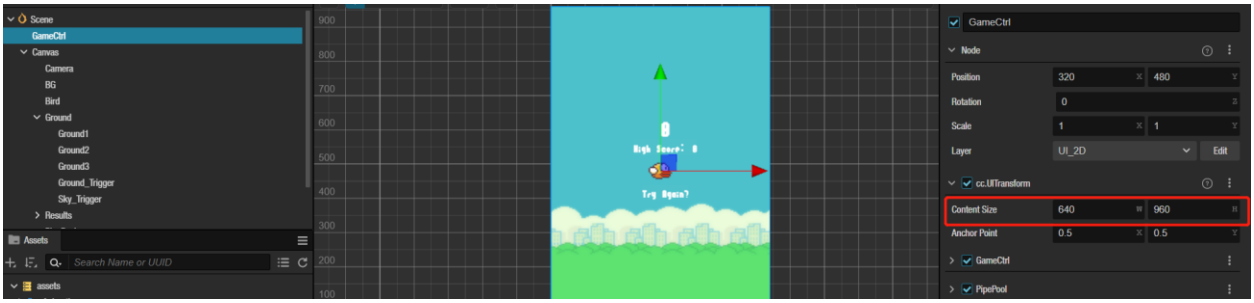
We need to call a finger touch or mouse click to have the bird go up. Let's go back to **GameCtrl.ts** and go into the **initListner**. Let's add an **EventType.TOUCH_START**. Inside, we will reset the game, start the game, and call the **bird.fly** if the game isn't over. We'll program the logic for the game over later, so let's have it so that anytime we click the mouse button, it will float up.

```
//listener for the mouse clicks and keyboard
initListener() {

  //if keyboard key goes down, go to onKeyDown
  input.on(Input.EventType.KEY_DOWN, this.onKeyDown, this);

  //if an mouse or finger goes down, do this
  this.node.on(Node.EventType.TOUCH_START, () => {

    //have the bird fly
    this.bird.fly();

  })

}
```

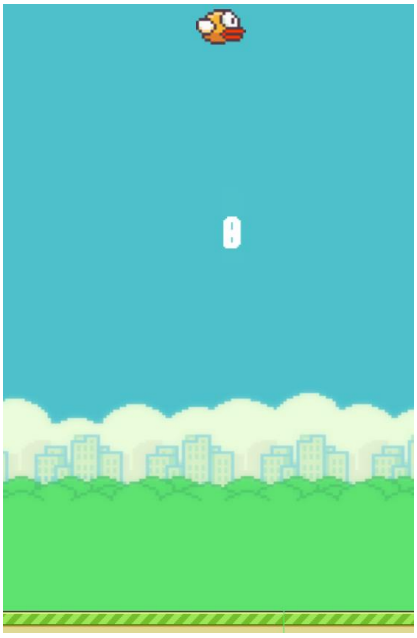
Now let's tell it to reset the bird when we reset the game in **onKeyDown**

```
switch (event.keyCode) {
  case KeyCode.KEY_A:
    //end game
    this.gameOver();
    break;
  case KeyCode.KEY_P:
    //add one point
    this.result.addScore();
    break;
  case KeyCode.KEY_Q:
    //reset score to zero
    this.resetGame();
    this.bird.resetBird();
}
```

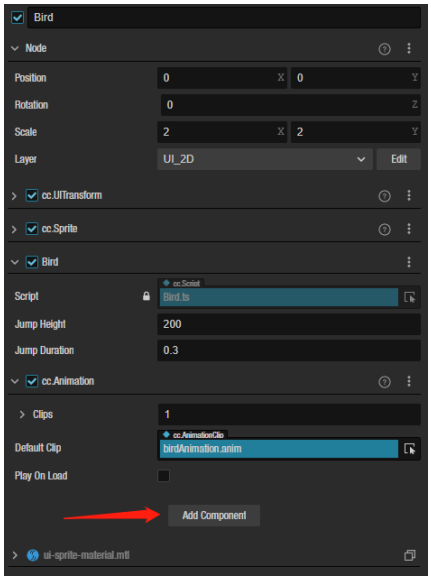
Before we finish this, we also need to ensure the **TOUCH_START** will work everywhere in the game. Remember the **UITransform** we added when building **GameCtrl**? Currently, the **GameCtrl** is only covering a small square on the canvas, so in Cocos Creator, let's make the content size the same as the canvas.

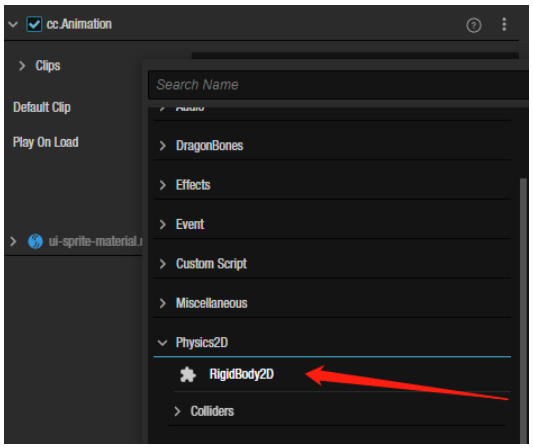


Let's save everything, and let's test this out. Press Q to start the game and then touch the game area. If working correctly while the ground is moving, you should see the bird going up as you click the scene with your mouse.

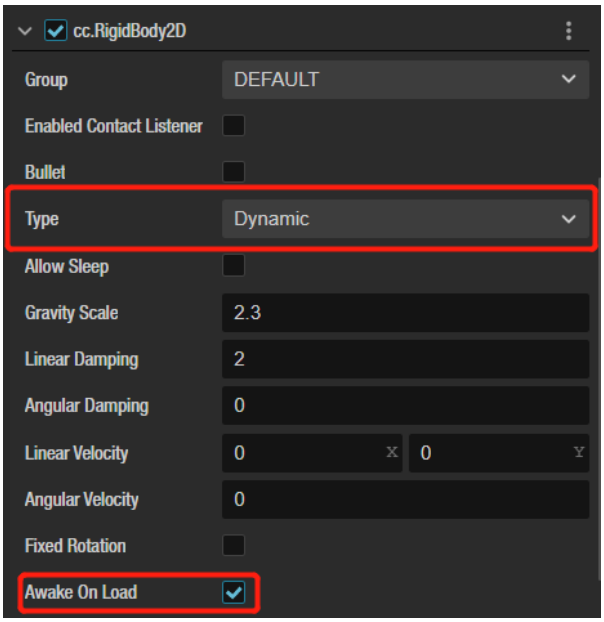


Now we need to add some physics to get the bird to fall. This is easy by adding a **RigidBody2D** component to the bird. This gives an object its physics. In Cocos Creator, click on the **Bird** node, and in the **Inspector**, click add Component and add, **Physics2D** -> **RigidBody2D**





We want to add **RigidBody2D** and change the type to “Dynamic”. This allows for the body to have a mass allowing for gravity. Also, make sure the “Awake On Load” is activated. We can play around with the Gravity Scale and Linear Dampening for a while and find what feels right. This is my settings, but you can try it on your own.



Save everything again and test it out. The bird should fly and fall. We have the bird items done. We'll return for the collision detection and game logic later, but for now, we go to the most challenging part…

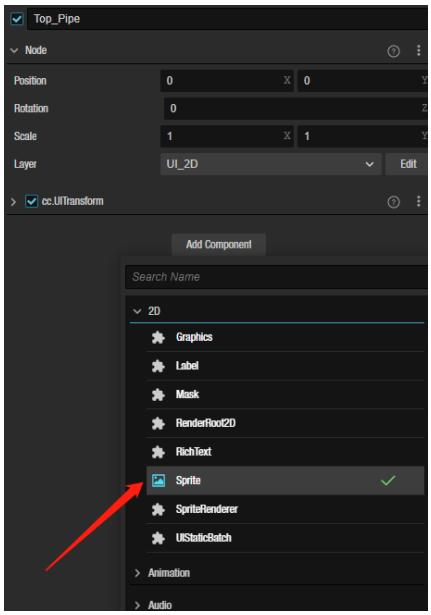
Pipes.

Part 7 – Building Pipes With Prefabs

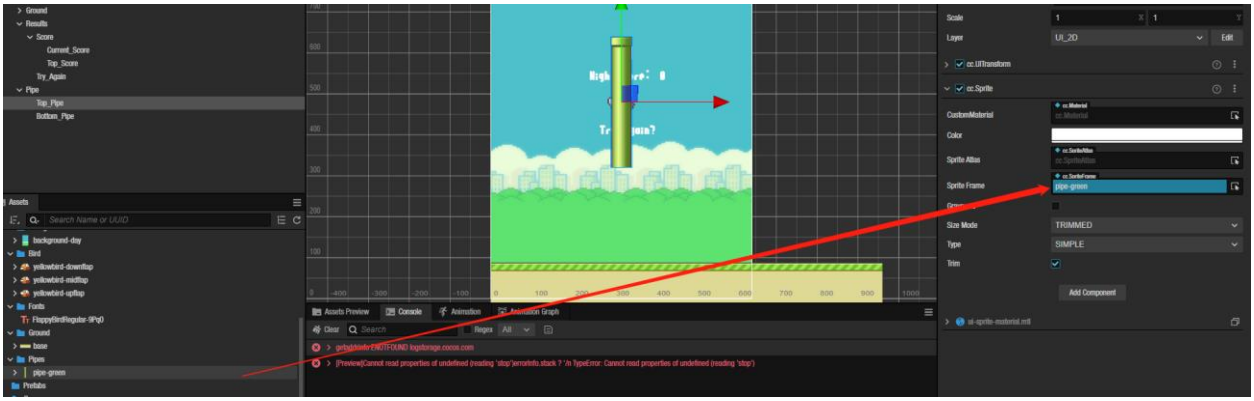
Now we need to do something with the pipes. If you look at the assets we were given, we were given one pipe with different styles. So we need to make a bottom pipe and a top pipe with one sprite. We must also have them enter the game simultaneously but with randomized locations and gaps. So we have to have a node that has many parts within it. This is why we have to build a prefab. A prefab will allow us to have two nodes, the top and bottom pipes, within it.

Let's start working on making it by creating a node in the node tree called **Pipe**. Within this node, add the pipe image inside it so it becomes a child of the **Pipe** node, just like our Ground node. Please copy and paste it, so we have two identical nodes in the **Pipe** node. Name each **Top_Pipe** and **Bottom_Pipe**.

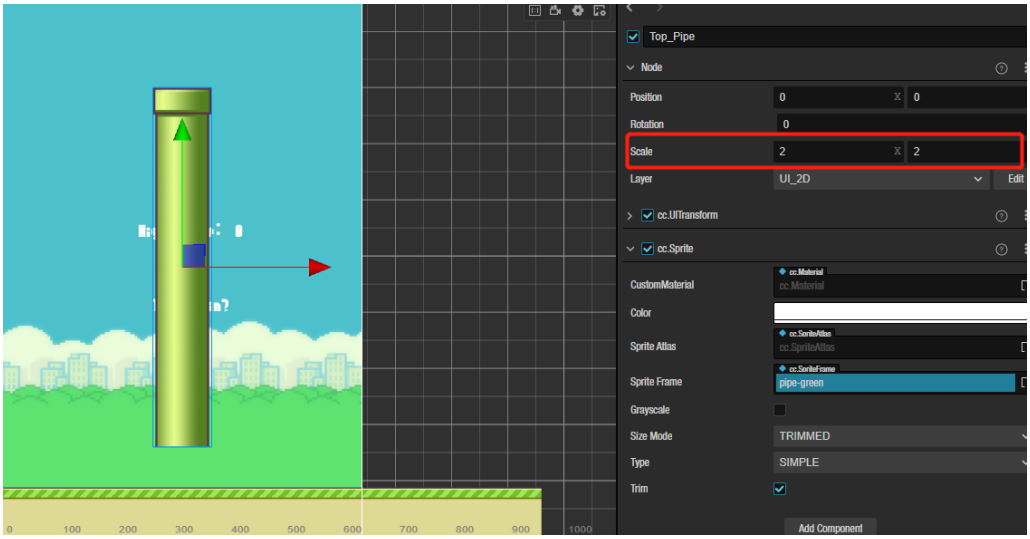
Let's go ahead and add the sprites to the nodes by adding a sprite Component to the **Top_Pipe** and **Bottom_Pipe**. You can find it at **2D** -> **Sprite**.



Add the pipe-green image from our Pipes folder in the assets window into the **spriteFrame**.



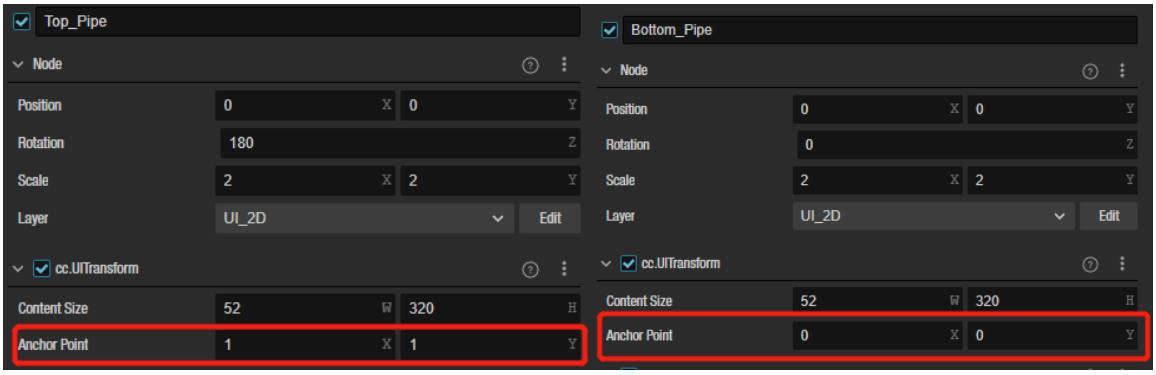
Once again, the pipe is too small. So let's increase the scale of x and y to 2.



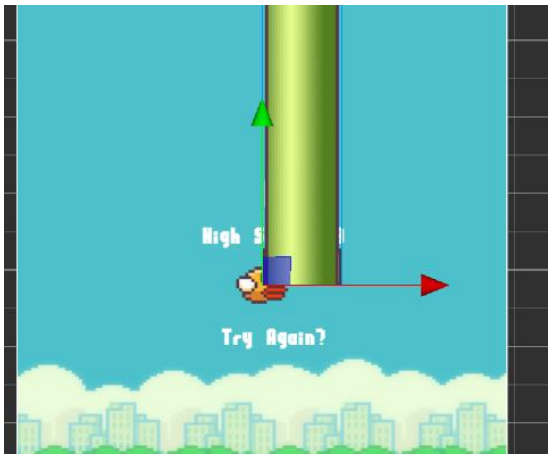
Do the same for the **Bottom_Pipe**.

Now go back to the **Top_Pipe,** and let's rotate the top pipe, so it's in the correct position. This can be done with the "Rotation" option in the Inspector. Move it to 180 degrees.

Another issue is just like the ground. We want the pipes to come in just outside the canvas. This requires us to do a lot of math, which would make it harder if the Anchor Points were in the way they are now. To fix this, we will need to change their anchor points to be the same for both pipes. For the **Top_Pipe**, we need it to be 1,1, and for the **Bottom_Pipe**, we need it to be 0,0.



It should look something like this in your Scene window. Don't worry, **Down_Pipe** is hiding the **Up_Pipe**:



Now we have to give the instructions for the pipes. Let's build a **Pipes.ts**, clean it up, and get it working.

We need to ensure the pipes are at the right end of the screen, at random locations and gaps, and move to the left and add a point to the score when the bird passes it.

We also want the pipe to destroy itself after leaving the screen because we no longer need it but tell the game to build a new pipe before it's destroyed.

NOTE: We could do the same as we did with the ground nodes, return them to the right edge of the screen, and reset the pipe heights. But in this tutorial, we want to share another way you can add nodes to a game and have them leave to save you memory. Trust me, this will be fun!

Building Pipes.ts

Same procedure as all other scripts we built. Clean up, and let's add two properties: **topPipe** and bottomPipe. This will be where we add the two pipes we just made.

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

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

    @property({
        type: Node,
        tooltip: 'Top Pipe'
     })
     public topPipe: Node;

     @property({
        type: Node,
        tooltip: 'Bottom Pipe'
     })
     public bottomPipe: Node;
}
```

Let's make a few temporary locations to help move our pipes just as we did for the ground and get the screen size.

```
//temporary Locations
public tempStartLocationUp:Vec3 = new Vec3(0,0,0);  //Temporary location of the up pipe
public tempStartLocationDown:Vec3 = new Vec3(0,0,0); //Temporary location of the bottom pipe
public scene = screen.windowSize; //get the size of the screen in case we decide to change the content size

//get the pipe speeds
public game; //get the pipe speed from GameCtrl
public pipeSpeed:number; //use as a final speed number
public tempSpeed:number; //use as the moving pipe speed
```

Once again, we need to add a few APIs: **Vec3** and **screen**

```
import { _decorator, Component, Node, Vec3, screen } from 'cc';
const { ccclass, property } = _decorator;
```

Now we only have three methods to build:

- **OnLoad**: Preparing the pipes
- **initPos**: The starting location of the pipes
- **Update**: Move the pipes on every frame

OnLoad will require us to get the speed of the pipes. Because this is a temporary node, it won't be able to find the component and add it every time it's created. So we'll ask the code to "find" the **GameCtrl** node and add the components to the temporary version to find the pipe speed.

```
//What to do when the pipes load
onLoad (){

    this.game = find("GameCtrl").getComponent("GameCtrl")
    this.pipeSpeed = this.game.pipeSpeed; // add pipespeed to temporary method
    this.initPos(); //work on original position

}
```

This won't work, of course, without the API call to **find**.

```
import { _decorator, Component, Node, Vec3, screen, find } from 'cc';
const { ccclass, property } = _decorator;
```

initPos will set up the initial position of the pipes. Start by getting the temporary starting locations of the x-axis. We first need to have them off the screen, and we can do that by adding the screen's width plus the width of the pipe.

```
initPos() {

    //Start with the initial position of x for both pipes
    this.tempStartLocationUp.x = (this.topPipe.getComponent(UITransform).width + this.scene.width);
    this.tempStartLocationDown.x = (this.bottomPipe.getComponent(UITransform).width + this.scene.width);
}
```

Again, we need to add the API call for **UITransform**.

```
import { _decorator, Component, Node, Vec3, screen, UITransform } from 'cc';
const { ccclass, property } = _decorator;
```

We then need to set up the **Top_Pipe's** y-axis location and the gap between the two. To do this, we need to add a random number generator. Go to the very top of the code and add one above the ccclass.

```
const { ccclass, property } = _decorator;

//Make a random number generator for the gap
const random = (min,max) => {
    return Math.random() * (max - min) + min
}

@ccclass('Pipes')
export class Pipes extends Component {
```

We can now build a random gap and **topHeight**. The **topHeight** tells us where on the y-axis the pipe can be located, and the **gap** will tell us the random size between the pipes. We can then work on adding the numbers to get the **bottomPipe** location. The exact numbers may be hard to get, so play around with it. Or try with what I have in my example.

Note: random can only do numbers 0 – 999, so we multiply it by 10 to get the correct gap size.

```
initPos() {

    //start with the initial position of x for both pipes
    this.tempStartLocationUp.x = (this.topPipe.getComponent(UITransform).width + this.scene.width);
    this.tempStartLocationDown.x = (this.bottomPipe.getComponent(UITransform).width + this.scene.width);

    //random variables for the gaps
    let gap = random(90,100);  //passable area randomized
    let topHeight = random(0,450);   //The height of the top pipe

    //set the top pipe initial position of y
    this.tempStartLocationUp.y = topHeight;

    //set the bottom pipe initial position of y
    this.tempStartLocationDown.y = (topHeight - (gap * 10));

    //set temp locations to real ones
    this.topPipe.setPosition(this.tempStartLocationUp.x, this.tempStartLocationUp.y);
    this.bottomPipe.setPosition(this.tempStartLocationDown.x, this.tempStartLocationDown.y);
}
```

Our final method is **update,** and we'll update the speed of how the pipe goes just as we did with the ground.

```
update(deltaTime: number){

    //get the pipe speed
    this.tempSpeed = this.pipeSpeed * deltaTime;

    //Make temporary pipe locations
    this.tempStartLocationDown = this.bottomPipe.position;
    this.tempStartLocationUp = this.topPipe.position;

    //move temporary pipe locations
    this.tempStartLocationDown.x -= this.tempSpeed;
    this.tempStartLocationUp.x -= this.tempSpeed;

    //place new positions of the pipes from temporary pipe locations
    this.bottomPipe.setPosition(this.tempStartLocationDown);
    this.topPipe.setPosition(this.tempStartLocationUp);
}
```

Add scoring

One last thing before we finish **Pipes.ts**. We need to tell the game the bird past the pipes and give the player a score. So the best way is to have two things happen:

1. If the pipes go past a certain distance, send a message to gameCtrl that a point needs to be scored.
2. Make sure it only sends this message once, so it does score every frame before being destroyed.

To do this, let's add a Boolean of **isPass** with the other temporary properties. Let's make it false when the node is created.

```
//Scoring mechanism
isPass: boolean; //Did the pipe pass the bird?

//What to do when the pipes load
onLoad (){

    this.game = find("GameCtrl").getComponent("GameCtrl")
    this.pipeSpeed = this.game.pipeSpeed; // add pipespeed to temporary method
    this.initPos(); //work on original position
    this.isPass = false; //set the scoring mechanism
}
```

In the **update**, let's make an if/then statement saying that if the **isPass** is false and the position is past the middle, we call on a method called **passPipe** inside **gameCtrl.ts** to score the point. We'll also change **isPass** to true, so it can't get back into the if/then statement afterward.

```
//find out if bird past a pipe, add to the score
if (this.isPass == false && this.topPipe.position.x <= 0)
{

    //make sure it is only counted once
    this.isPass = true;

    //add a point to the score
    this.game.passPipe();
};
```

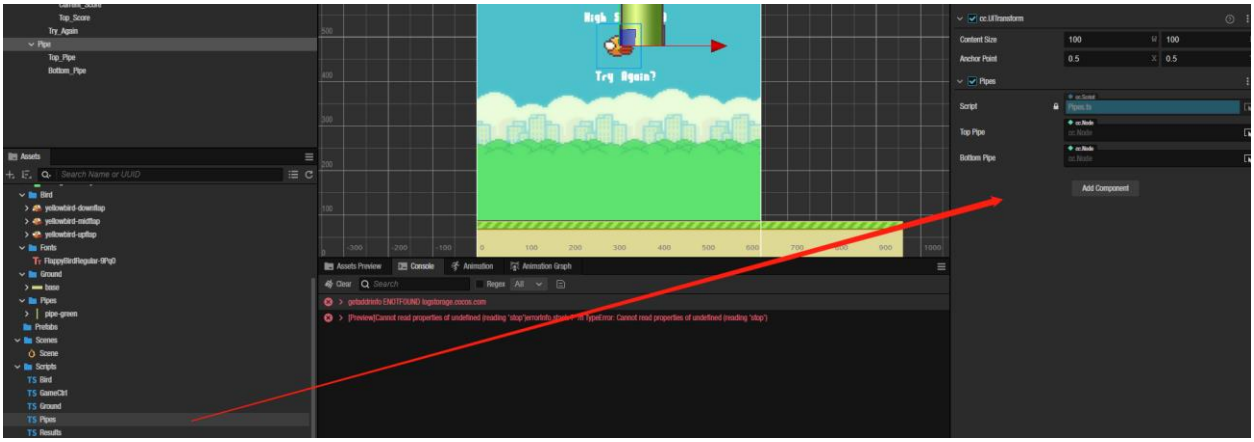
Let's go back to **gameCtrl.ts** and make the method **passPipe**. All we need to do is call on the **result.addScore()** like we did with the keyboard testing. And it should work.

```
//when a pipe passes the bird, do this
passPipe() {

  //passed a pipe, get a point
  this.result.addScore();
}
```

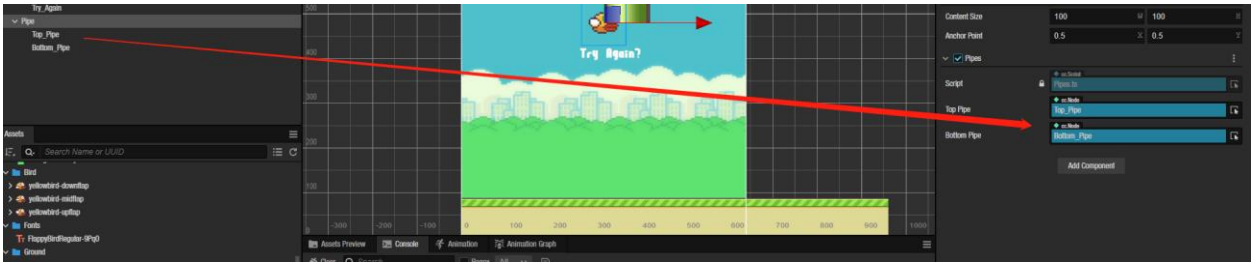
Finally, in **Pipe.ts**, one more if/then statement that says if the pipe is off the screen, destroy this node. Just ask if the pipe's x-position is less than half the canvas width and the pipe's width. If so, tell **GameCtrl.ts** to build a new pipe, then destroy itself.

```
//If passed the screen, reset pipes to new location
if (this.topPipe.position.x < (0 - this.scene.width)){

    //create a new pipe
    this.game.createPipe();

    //delete this node for memory saving
    this.destroy();
};
```

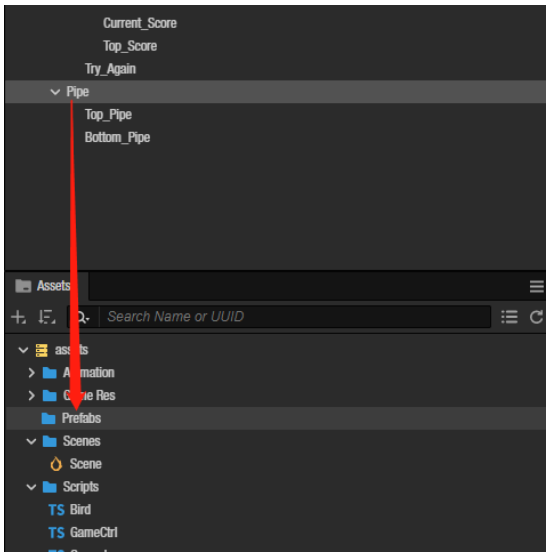
Save everything for now, and let's connect everything in Cocos Creator. Go back to the editor and add the **Pipe.ts** to the **Pipe** node.



Now add the Top_Pipe and Bottom_Pipe to the component.



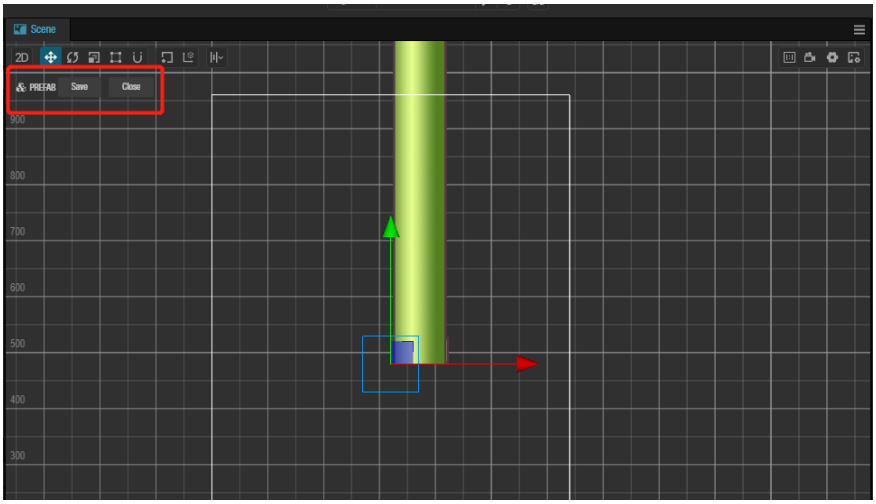
Now we can make it a prefab. Take the **Pipe** node and drag it to the Assets area and into the Prefab folder



Now you should see the **Pipe** prefab there.

You can now delete the **Pipe** node in the **Node Tree**. Because we have it as a prefab, we don't need the original node there, so delete it. We will show you in the next part how to have this prefab added repeatedly. Now save your code and scene.

NOTE: If you want to see the nodes inside the prefab, double-click the prefab Pipe, which will take you to the prefab area. You can edit everything here. Just make sure to hit the "Save" button to save your work and the "Close" button to return to the full project.



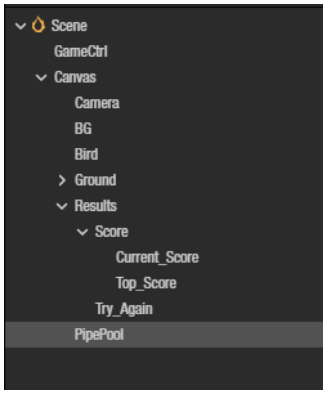
Part 8 – Making New Pipes With A NodePool

This part took me a bit of time to figure out the first time. So I hope to get you educated on it much faster than it took me.

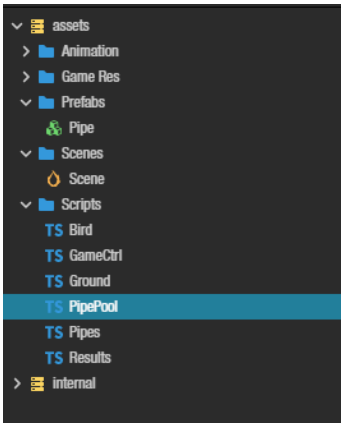
A node pool is a way to place nodes on a waiting list. When you call on the node pool, it will send you the first node it made. Like people in a line, first in line goes first, and people behind you wait. This is good because we can then put a bunch of pipes in the node pool and have them ready to show up on the screen when we need them. The other good thing about the pool is that it can be grown if required.

For this to work, we will need to create the nodes first and place them in the node pool. Once it's full, we can take one out at a time and make them a child of a node we have in the scene. This continues as the pipes move, and a new pipe will be placed after the **Pipe** node destroys itself, as discussed in the previous section.

In Cocos Creator. Let's make the parent of these new **Pipe** nodes. Let's call it **PipePool**. We won't have to add anything to it. It just is there waiting to add new children to be placed inside it.



Now we need to build a node pool script. So in the Assets, build a **PipePool.ts**.



Let's go ahead and clean it up and import **Pipe.ts** into the code since we're making pipe nodes.

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

import { Pipes } from './Pipes';

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



}
```

For properties, we need just two. The prefab of the two pipes we built and the node for pipe nodes to go into (i.e., The parent node for the pipe children).

```
@ccclass('PipePool')
export class PipePool extends Component {

    @property({
        type: Prefab,
        tooltip: 'The prefab of pipes'

    })
    public prefabPipes = null;

    @property({
        type: Node,
        tooltip: 'Where the new pipes go'
    })
    public pipePoolHome;

}
```

Another error may come up saying it doesn't know what a prefab is. Let's go ahead and add the **Prefab** API to the top.

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

Next, add initializing properties. We need a node pool holder and a temporary pipe node to add new pipes that are created to the pool.

```
//Create initial properties
public pool = new NodePool;
public createPipe:Node = null;
```

Again, we need to add another API. This time it's **NodePool**.

```
import { _decorator, Component, Node, Prefab, NodePool } from 'cc';
const { ccclass, property } = _decorator;
```

Now we need three methods to get this to work:

- **initPool** – starting up the node pool system.
- **addPool** – Add a node from the pool to the game.
- **reset** – clean everything up for the new game.

For **initPool**, we need to tell it the number of nodes we want as a maximum amount. So we build an **initCount**.

```
//build the amount of nodes needed at a time
let initCount = 3;
```

Next, we need to fill the pool. We need to send the first pipes when the game starts. So we need to add some additional information. We now add the first Pipe node and the other pipes into the node pool.

```
initPool(){

    //build the amount of nodes needed at a time
    let initCount = 3;

    //fill up the node pool
    for (let i = 0; i < initCount; i++) {
        // create the new node
        let createPipe = instantiate(this.prefabPipes); //instantiate means make a copy of the orginal

        // Put first one on the screen. So make it a child of the canvas.
        if (i == 0) {
            this.pipePoolHome.addChild(createPipe);
        } else {
            //Put others into the nodePool
            this.pool.put(createPipe);
        }
    }

}
```

To create a node, we need to instantiate the pipe prefab node. Instantiating just means creating a new copy of the original node. We make a new copy and place it in the createPipe node. We also need to add another API call to allow this to happen. So let's add **instantiate**.

```
import { _decorator, Component, Node, Prefab, NodePool, instantiate } from 'cc';
const { ccclass, property } = _decorator;
```

The last part of the loop tells the game that if it's the first one in the pool, add it as a child of pipePoolHome, which will become an enabled node viewable to the player. Items in the node pool aren't real yet because we haven't placed them in the canvas. They are just in the code. Making them a child makes them a real nodes on the node tree. We finally tell the computer to fill the node pool if it's not the first one.

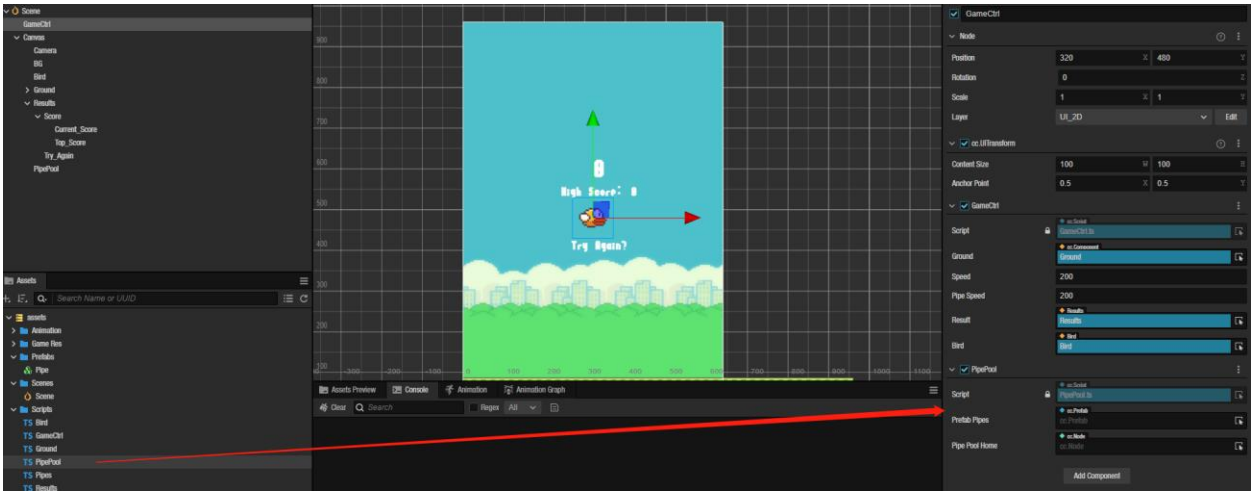
Let's move to **addPool**. We have two things we need to do here. If there are no nodes in the node pool, fill it with a new node and instantiate it. If there are, get the node and add it as a child to **pipePoolHome**. This can be accomplished with a simple if/then statement and an **addChild** method.

```
addPool() {
    //If the pool is not full add a new one, else get the first one in the pool
      if (this.pool.size() > 0) {
          //get from the pool
          this.createPipe = this.pool.get();
      } else {
          //build a new one
          this.createPipe = instantiate(this.prefabPipes);
      }
      //add pipe to game as a node
      this.pipePoolHome.addChild(this.createPipe);

  }
```

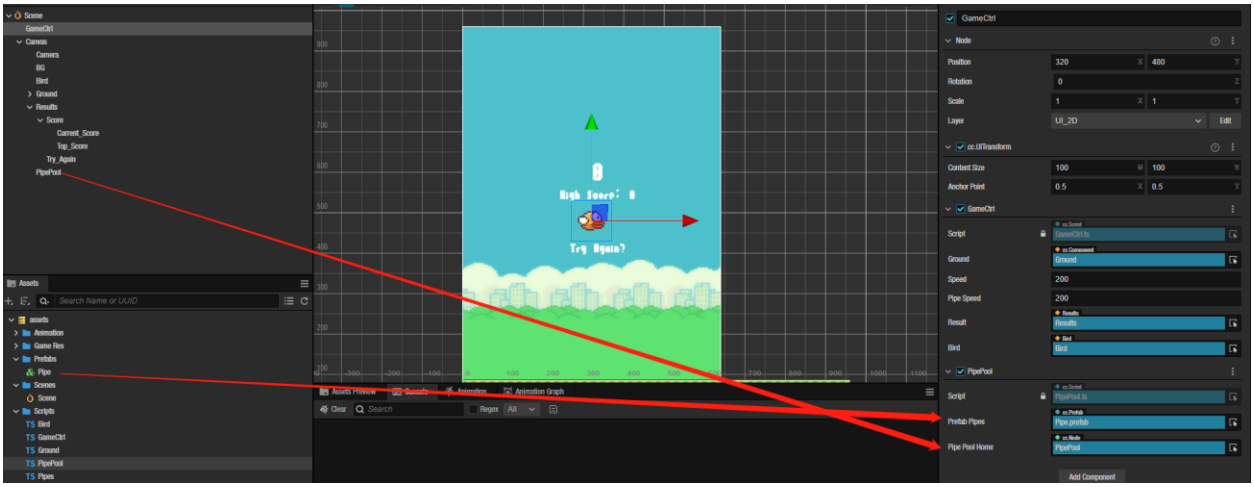
Finally, we can clean out the pipePoolHome of all children for **reset**. Hence, all pipes disappear from the screen, clear out the pool, and reinitialize it with **initPool**.

```
reset() {

    //clear pool and reinitialize
    this.pipePoolHome.removeAllChildren();
    this.pool.clear();
    this.initPool();

}
```

Save this and go to Cocos Creator. Now let's add the **PipePool.ts** to the **GameCtrl** node.



Let's fill this up with the **Pipe** prefab and the **PipePool** node.



We need to ask **GameCtrl** to call **initPool** when the game starts and when we reset the game. So back to **GameCtrl.ts,** we go.

First, let's add **pipePool** to the rest of the files.

```
import { Ground } from './Ground';
import { Results } from './Results';
import { Bird } from './Bird';
import { PipePool } from './PipePool';
```

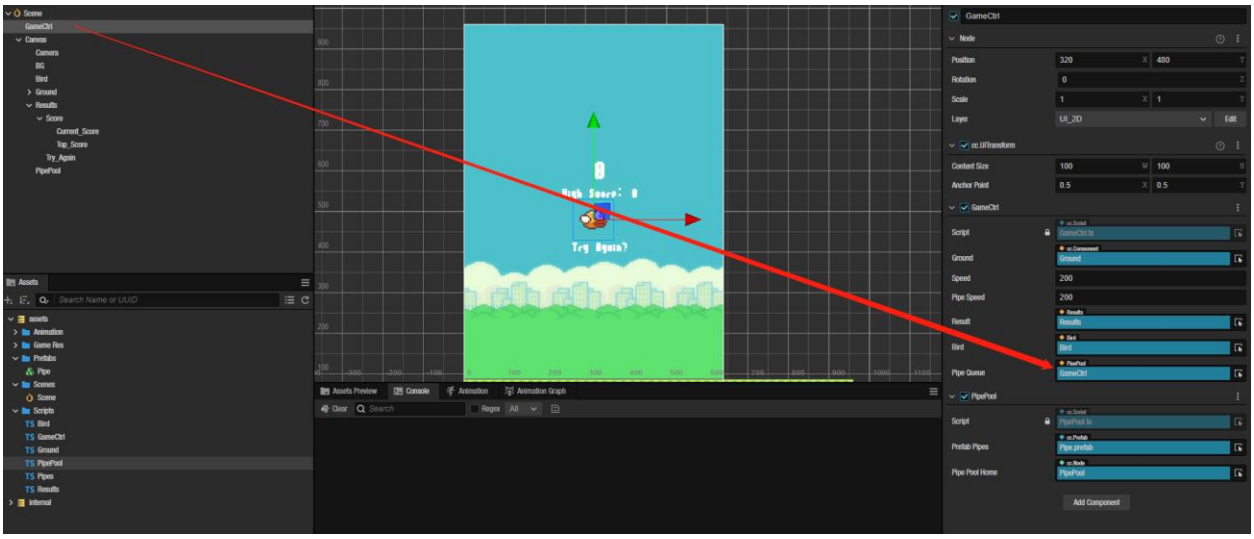
We also need to add a new property for the **PipePool**. So let's do that next.

```
@property({
  type: PipePool,
  tooltip: 'Add canvas here'
})
public pipeQueue: PipePool
```

Now add reset to the **resetGame**.

```
//When the game starts again, do these things.
resetGame() {

  //Reset score, bird, and pipes
  this.result.resetScore();

  //Reset the pipes
  this.pipeQueue.reset();

  //Get objects moving again
  this.startGame();

}
```

Save and return to Cocos Creator. Add the **GameCtrl** to the Pipe Queue. This is because it will be the one controlling the **PipePool**.

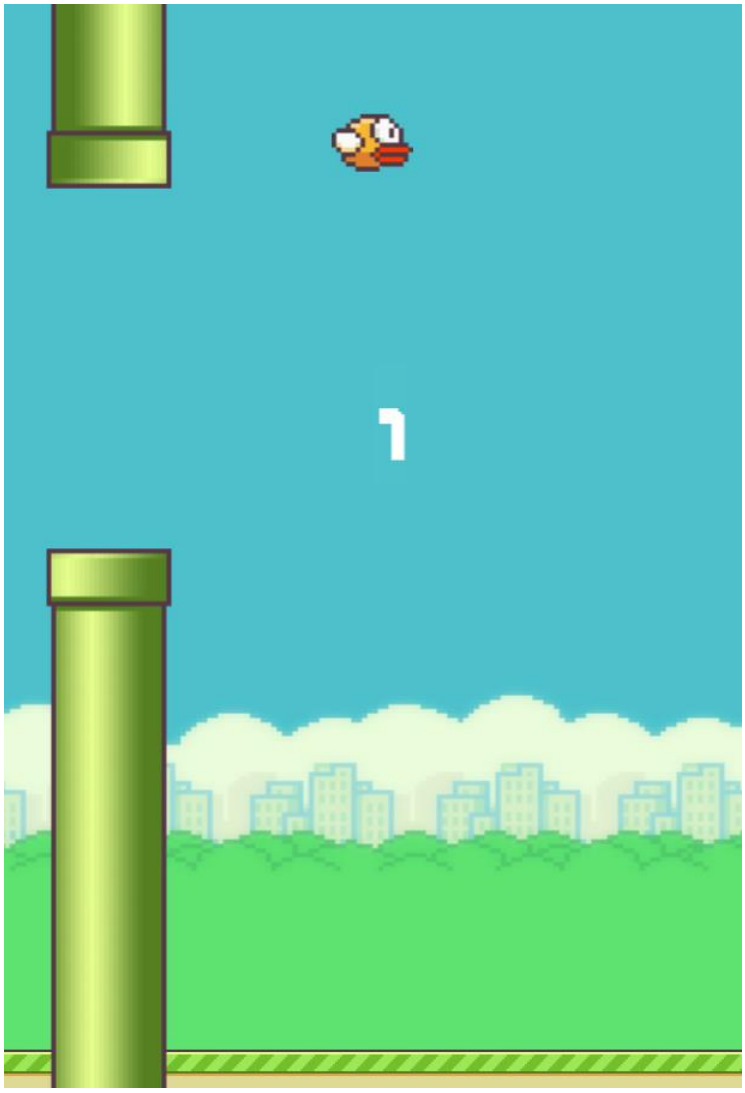


Finally, one last thing we didn't finish from the previous chapter. We didn't add **createPipe()** to **GameCtrl**. We added it to **Pipes.ts** but never what to do with it. So go back to **GameCtrl,** and let's call on the **pipeQueue** to add from the pool.

```
//When the old pipe goes away, create a new pipe
createPipe() {

  //Add a new pipe to the pipe pool
  this.pipeQueue.addPool();

}
```

And that's it. It wasn't as complicated as you thought, and we could optimize this better, like having more pipes coming out at once and filling the pool all the time. So play around with it some more. Node pools are perfect for objects that come and go quickly, like bullets or items. So it's good we got some practice here.

Go ahead and save and test out the game, since there isn't any hit detection, the pipes should move past you, and you score points as they do. So let's work on hit detection before adding the last few things to the game logic.

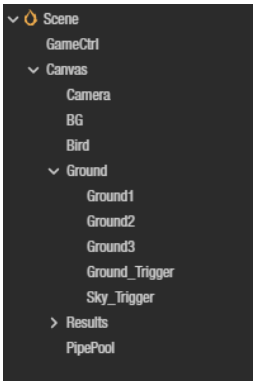


Part 9 – Adding Hit Detection And Final Game Logic

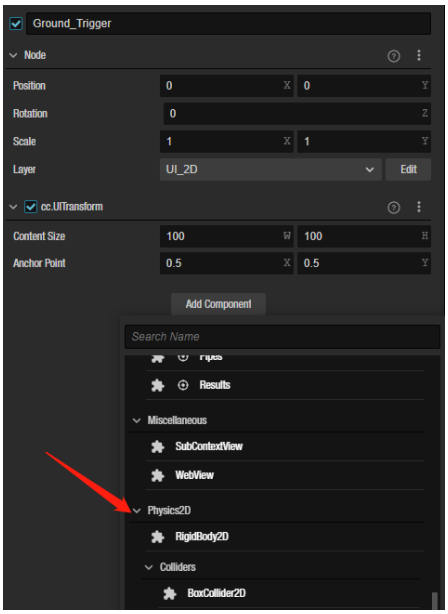
Now that we have all the items on the screen (bird, ground, background, score, and pipes), we can get to how you end the game: hitting the pipe or the ground. If you tested a bit, you'd see your bird can fall right off the screen if you don't press the mouse key. So let's first address this.

Hit detection

In Cocos Creator, let's go back to the ground group and add two nodes. Name them **Ground_Trigger** and **Sky_Trigger**.

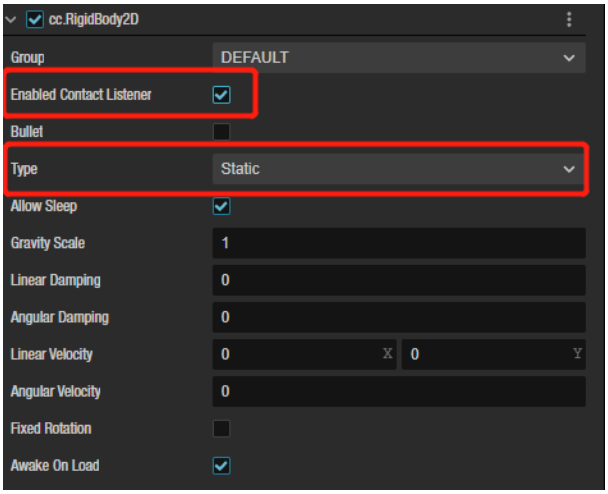


I want the hit detection boxes to stay where they are and detect if something hits them, so first, we'll give these nodes a **RigidBody2D** component to detect. This can be found in **Physics2D.**

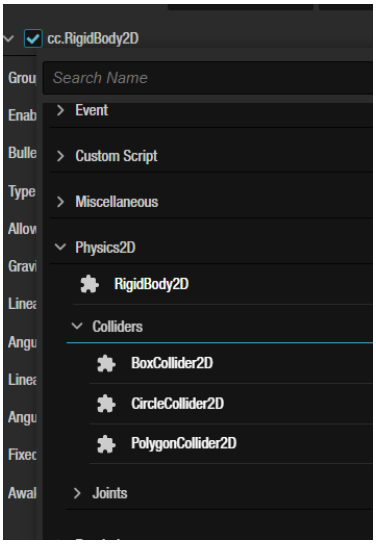


Make sure to do the same for the **Sky_Trigger** node.

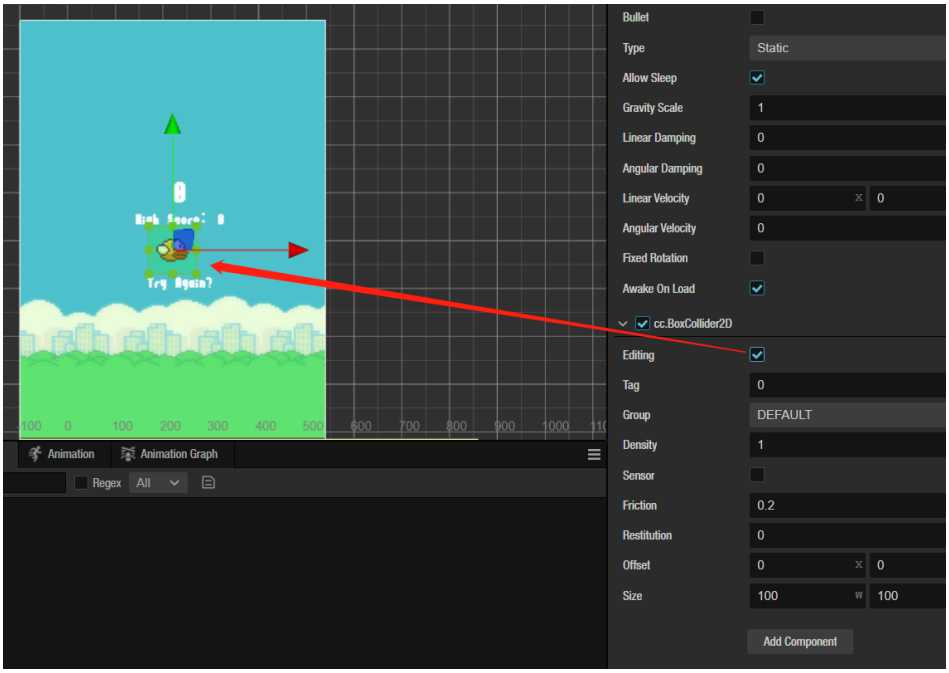
Now we need to configure these RigidBodies, so they don't have mass and stay where they are. To do this, we'll keep their Type to “Static”. This means they can't move with physics. Let's also turn on the listener so that we know they are listening for contact.



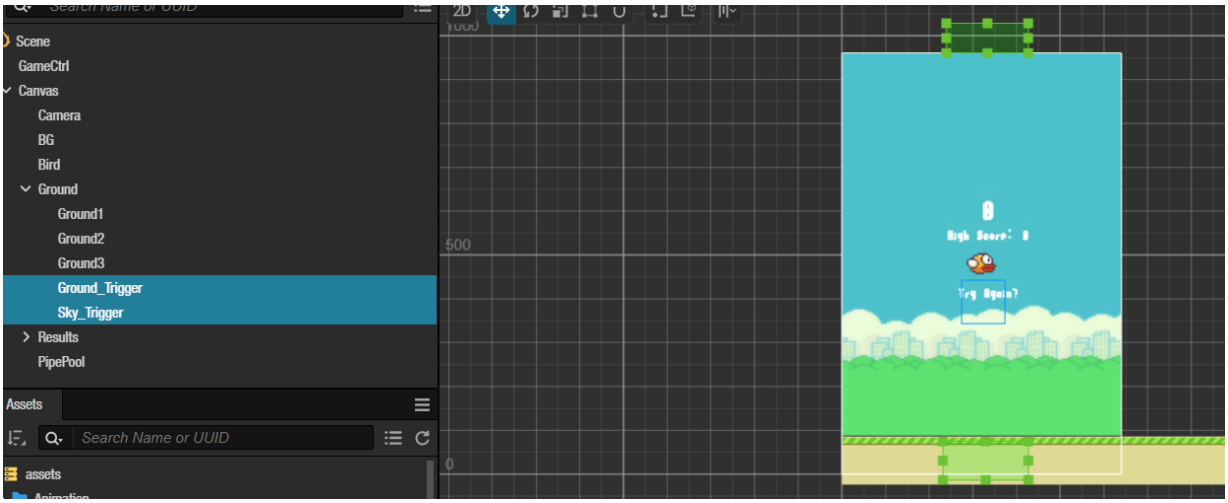
Now we need to add something to make the actual contact. We need a new item called a **BoxCollider2D**. This will tell us if something hits us and send it to the rigid body for detection. So add this to both nodes as well. You can find this in **Physics2D** -> **Colliders**



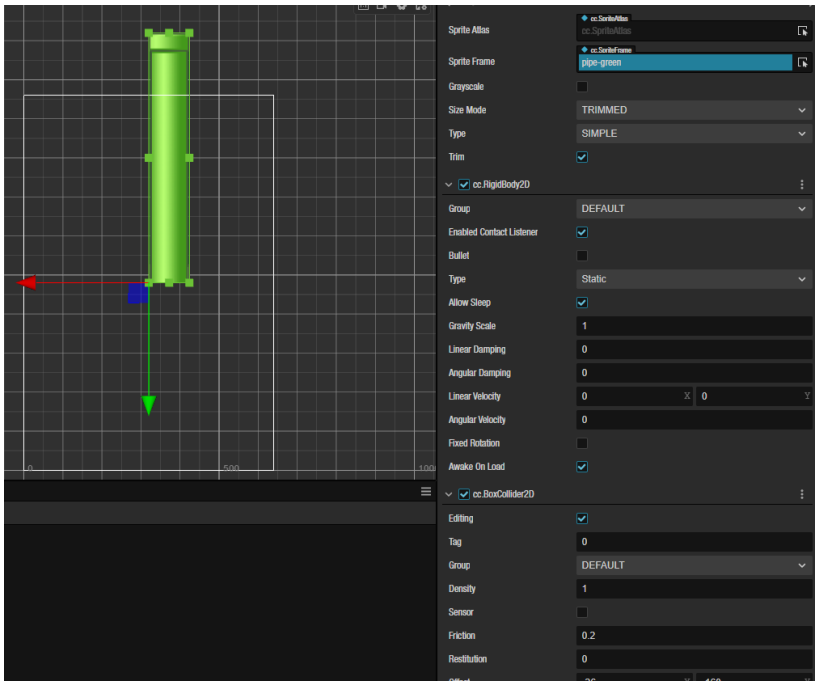
Now we need to edit the location of the collider. This is easy as clicking the Editing button and seeing the collider's location on the screen. Because this is a BoxCollider, it will be a green rectangle with nine anchor points.



Since the bird can't go left or right, we just need to place both colliders above the canvas and below the bird on the ground.

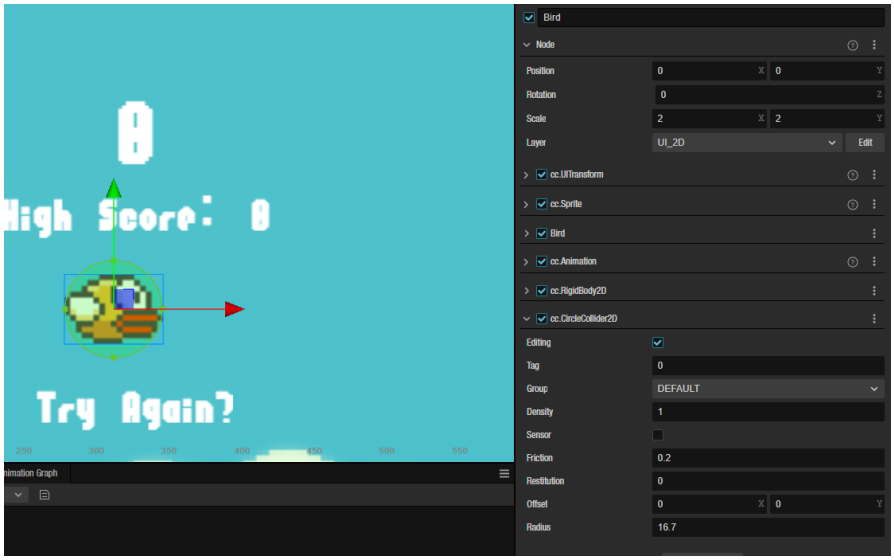


Now we need to do the same thing for the pipes. So double-click on the pipes prefab to open the Prefab area. We can now add a click on each pipe node and add a **RigidBody2D** (As a static type and enable listener) and a **boxCollider2D** on each. We want these to be static because if we didn't, they'd fall off the screen just like our bird.

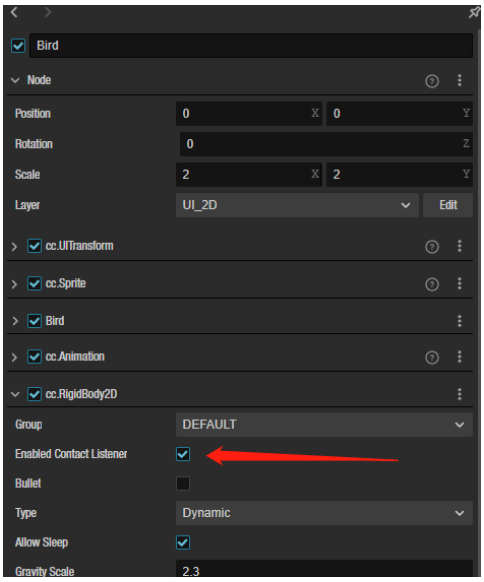


Make sure the boxCollider covers the entire pipe. Save the prefab and close it.

Now we need to add a collider to the bird. But because it's more circular, let's instead add a **circularCollider2D**. Follow the same steps as we did for the previous items. Make sure it covers it all.



One last thing. Make sure to turn on the listener in the Rigidbody2D already inside the bird.



Great, we got all the parts in the editor done. Now let's connect everything with code. We'll need to have the game watching for contact, and when it does, it needs to tell us the bird hit something, and it's time to call the game over. We do this because we want the bird.ts to be updated constantly to check on it.

Let's first open the **gameCtrl.ts** and add the APIs we'll need for this: **Contact2DType**, **Collider2D**, and **IPhysics2DContact**

```
import { _decorator, Component, Node, CCInteger, Input, input, EventKeyboard, KeyCode, director, Contact2DType, Collider2D, IPhysics2DContact } from 'cc';
const { ccclass, property } = _decorator;
```

Now we can add two new methods:

- **contactGroundPipe**: Checks if there was a collision
- **onBeginContact**: If it hits, tell the bird.ts that it did to start the game over sequence

This will check if there is contact from the contact listeners we added on the **rigidBodies2D** we made for the pipes, ground, sky, and bird. With **contactGroundPipe**, we make a temporary collider and copy the component from the bird, so we are only looking at collisions with the bird. We can then tell the collider to do an action when turned on and it sees a collision. We will be asking it to contact **onBeginContact**.

```
//Check if there was contact with the bird and objects
contactGroundPipe(){

  //make a collider call from the bird's collider2D component
  let collider = this.bird.getComponent(Collider2D);

  //check if the collider hit other colliders
  if(collider){
    collider.on(Contact2DType.BEGIN_CONTACT, this.onBeginContact, this);
    }
}
```

We have it on **BEGIN_CONTACT**, but we could have done a few others, like after contact or while in contact, but we only need to do the actions at the start of contact. You can learn more from the link below on types of contact callbacks.

https://docs.cocos.com/creator/manual/en/physics-2d/physics-2d-contact-callback.html

Now let's look at the **onBeginContact**,

```
//if you hit something, tell the bird you did
onBeginContact (selfCollider: Collider2D, otherCollider: Collider2D, contact: IPhysics2DContact | null) {

    // will be called once when two colliders begin to contact
    this.bird.hitSomething = true;
}
```

It should see that two items hit each other and now calls for **this.bird.hitSomething**. We haven't built this yet. So let's do this next.

Go to **Bird.ts**, and let's add a Boolean called **hitSomething**

```
//hit detection call
public hitSomething:boolean;


//all the actions we want done when we start the script.
onLoad(){
```

On the **restetBird** method, let's make **hitSomething** false because, at the start, it hasn't hit anything.

```
resetBird(){

    //create original bird location
    this.birdLocation = new Vec3(0,0,0);

    //place bird in location
    this.node.setPosition(this.birdLocation);

    this.hitSomething = false;

}
```

Now we can come to the final few items of logic that we need.

Last logic items

Now we need to build two new methods to help start the game over a sequence in **GameCtrl.ts**:

- **birdStruck**: If it hits something, call the **gameOver**
- **update**: check every frame it hit something

For **birdStruck**, we want it first to check if there was contact. If there was, then it will change the **hitSomething** as true. This will then trigger an if/then statement to tell the GameCtrl

to end the game. So let's have it call the contactGroundPipe to see if it's been updated to say it hit something.

```
//hit detection call
birdStruck() {
  //make a call to the gameBrain to see if it hit something.
  this.contactGroundPipe()

  //if we hit it, tell the game to call game over.
  if (this.bird.hitSomething == true)
  {
      this.gameOver();
  }

}
```

In the **update**, we ask it to check **birdStruck**. We don't need deltatime because it won't be used.

```
//every time the game updates, do this
update(){

  //If the game is still going, check if the bird hit something
  if (this.isOver == false){
      this.birdStruck();
  }

}
```

You'll see we have an if/then statement to **isOver**. We need this one item so that everything stops when it's game over and when to start up again. To do this, add a Boolean called **isOver**. When it's false, the game plays on, but when it's true, the game stops and waits for you to reset it.

```
//properties needed for GameCtrl
public isOver: boolean;
```

So we need to call on this property in three places: **onLoad** as true, **gameOver** as true, **resetGame** as false, and one more place.

```
//Things to do when the game loads
onLoad() {

  //get listener started
  this.initListener();

  //reset score to zero
  this.result.resetScore();

  //game is over
  this.isOver = true;

  //pause the game
  director.pause();

}
```

```
//When the bird hits something, run this
gameOver() {

  //Show the results
  this.result.showResult();

  //game is over
  this.isOver = true;

  //pause the game
  director.pause();

}
```

```
//When the game starts again, do these things.
resetGame() {

  //Reset score, bird, and pipes
  this.result.resetScore();

  //Reset the pipes
  this.pipeQueue.reset();

  //game is starting
  this.isOver = false;

  //Get objects moving again
  this.startGame();

}
```

The final place needs a bit of work. In the **initListener**, we need to ensure that if you click the button, if it's not game over, you fly, but if it is, it resets the game. This can be done with an if/then statement.

```
//listener for the mouse clicks and keyboard
initListener() {
  //if keyboard key goes down, go to onKeyDown
  input.on(Input.EventType.KEY_DOWN, this.onKeyDown, this);

  //if an mouse or finger goes down, do this
  this.node.on(Node.EventType.TOUCH_START, () => {

    //if we are starting a new game
    if (this.isOver == true) {

      //reset everything and start the game again
      this.resetGame();
      this.bird.resetBird();
      this.startGame();

    }

    //if the game is playing
    if (this.isOver == false) {

      //have Bird fly
      this.bird.fly();

      //make the bird go swoosh
      this.clip.onAudioQueue(0);
    }

  })
}
```

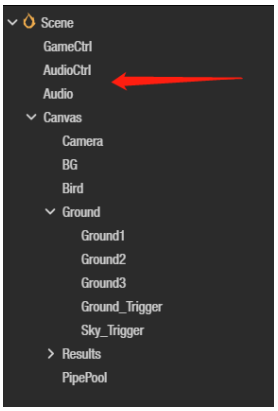
You can see that if we have reset the game, the bird, pipes, and ground reset and start over again.

That's it. Now we have a complete game. Save it and play it. Everything should be working. But we're missing something…. Sound effects. Let's get to it.

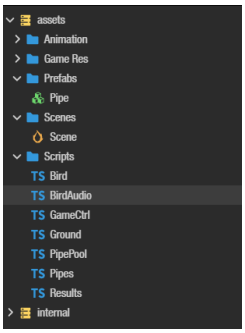
Part 10 - Sound Effects For The Game

Now for the fun part, adding sounds to the game. We got a lot from the downloaded assets, but we only need three: Scoring a point, flapping the wings, getting hit, and game over.

So we need to add a new node for sound in the node tree. Go back to Cocos Creator and add one to the node tree named **AudioCntrl**. This node will control all the sound effects we add. We also need a node to store the audio in so it can play it. So add another and name it **Audio**.



Now we need to add a script. So let's add a new TypeScript file and name it **BirdAudio.ts**. Open it up, and let's edit.



This one is pretty easy. We first cleaned up the script and added a few extra APIs. First, there is **AudioClip**, which allows you to add clips to the **AudioCntrl**, and **AudioSource**, which plays the clips in the **AudioCntrl**.

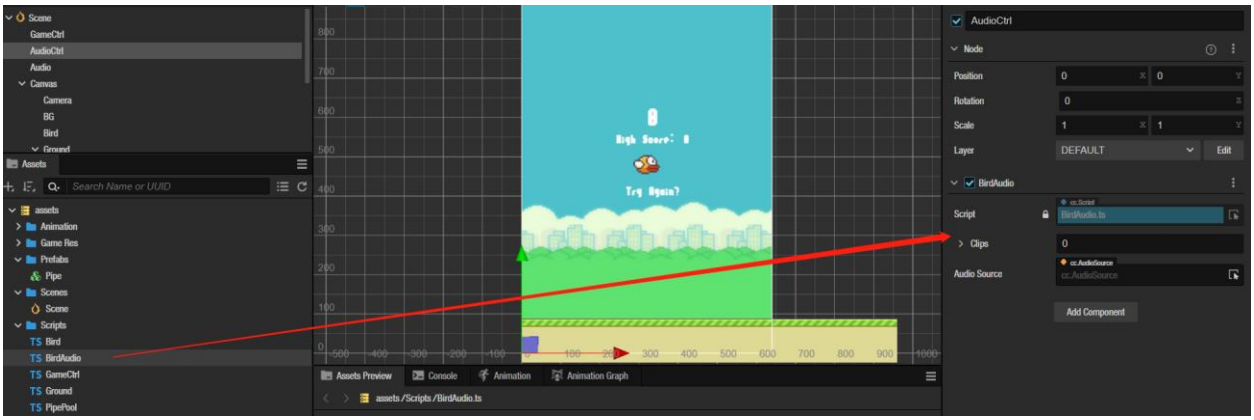
```
import { _decorator, Component, Node, AudioClip, AudioSource } from 'cc';
const { ccclass, property } = _decorator;
```

Now we can go into ccclass and add two properties:

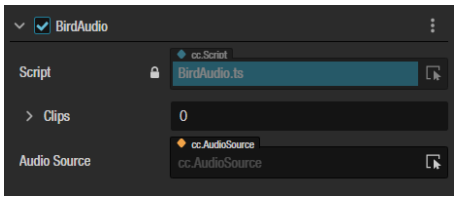
- **Audioclip**: where the array of clips goes.
- **AudioSource**: where the audio goes to be played.

```
@ccclass('BirdAudio')
export class BirdAudio extends Component {

    @property({
        type: [AudioClip],
        tooltip:'place audio clip here'
    })
    public clips: AudioClip[] = [];

    @property({
        type: AudioSource,
        tooltip: 'place audio node here'
    })
    public audioSource: AudioSource = null!;


}
```

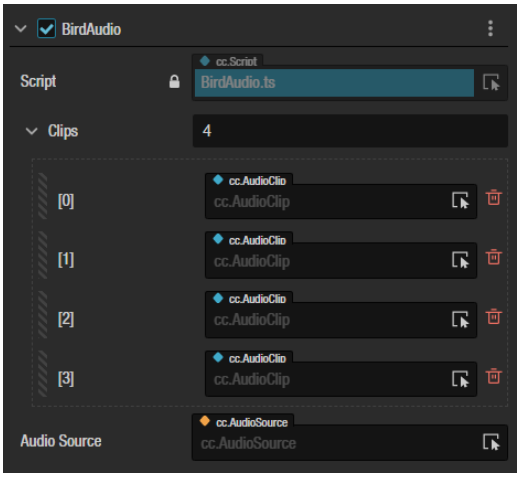
We're having AudioSource not be null by using the null! to tell it not to call that. Save this, and if we go back to Cocos Creator, drag the **BirdAudio.ts** script into **AudioCntrl**.



You can see that it has two options in the Inspector.



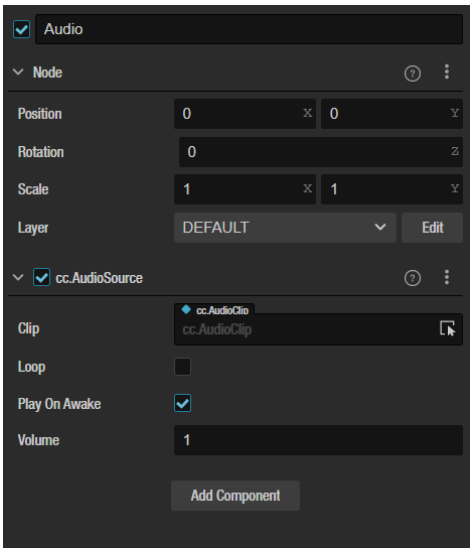
You can change the number of clips to 4. If you do that, you'll see it asking for four **AudioClips**.



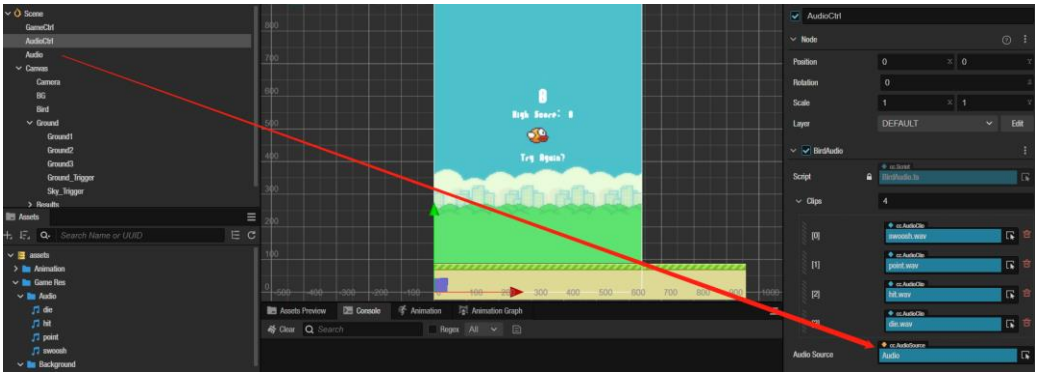
This is where we're going to add the four clips:

- 0: swoosh.wav
- 1: point.wav
- 2: hit.wav
- 3: die.wav

We also want to add the **Audio** node to the Audio Source section of the component. But to do that, we must first choose the **Audio** node and add the **AudioSource** component. It can be found at **Audio** -> **AudioSource**.



You don’t have to add anything here. We’ll be adding the clips through code. Now let's go back to **AudioSource** and add the Audio Node.



Let's go back to **Audio.ts** and add one more thing. We need to add one method:

- **onAudioQueue**: places the correct audio clip into the audio source

Here we ask for one parameter, which coincides with the clips we have 0-3. We make a temporary audio clip and then add the correct sound from the clip array into it. Then we place this clip into the **audioSource** and play it using **playOneShot()**. This means playing once without looping.

```
//place correct audio clip in the audio player and play the audio
onAudioQueue(index: number) {

  //place audio clip into the the player
  let clip: AudioClip = this.clips[index];

  //play the audio just once
  this.audioSource.playOneShot(clip);

}
```

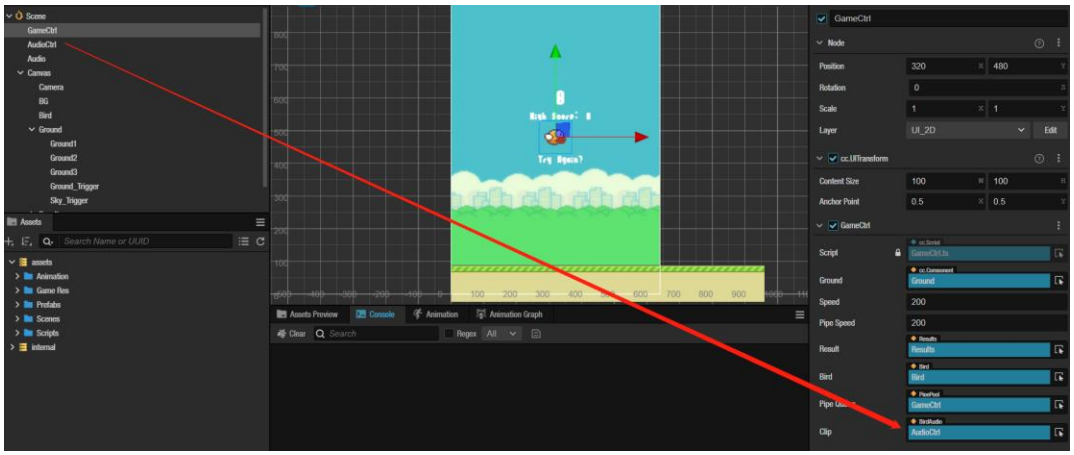
Let's save this, and now we just need to find the four spots where we need sound effects. Luckily they are all in the **GameCtrl.ts**. So just add the **BirdAudio.ts** file on the top of the code.

```
import { Ground } from './Ground';
import { Results } from './Results';
import { Bird } from './Bird';
import { PipePool } from './PipePool';
import { BirdAudio } from './BirdAudio';
```

We need to add a new property for the audio in ccclass under the **PipePool** named **BirdAudio**.

```
@property({
  type: BirdAudio,
  tooltip: "add audio controller",
})
public clip: BirdAudio;
```

Let's return to Cocos Creator and add the **AudioCtrl** to **GameCtrl** Component.



Now we can add all four. Remember to put the correct number that corresponds with the sound you put in the component:

Swoosh goes after **bird.fly** in the **initListener** method.

```
//listener for the mouse clicks and keyboard
initListener() {
  //if keyboard key goes down, go to onKeyDown
  input.on(Input.EventType.KEY_DOWN, this.onKeyDown, this);

  //if an mouse or finger goes down, do this
  this.node.on(Node.EventType.TOUCH_START, () => {

    //if we are starting a new game
    if (this.isOver == true) {

      this.resetGame();
      this.startGame();

    }

    //if the game is playing
    if (this.isOver == false) {

      //Have Bird fly
      this.bird.fly();

      //make the bird go swoosh
      this.clip.onAudioQueue(0);
    }

  })

}
```

The Point sound goes after the **addScore** call in **passPipe.**

```
//when a pipe passes the bird, do this
passPipe() {

  //passed a pipe, get a point
  this.result.addScore();

  //make the point ring
  this.clip.onAudioQueue(1);
}
```

Hit goes in the **onBeginContact** method.

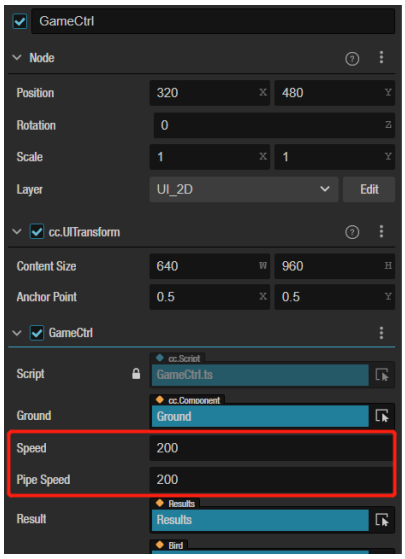
```
//if you hit something, tell the bird you did
onBeginContact(
  selfCollider: Collider2D,
  otherCollider: Collider2D,
  contact: IPhysics2DContact | null
) {

  // will be called once when two colliders begin to contact
  this.bird.hitSomething = true;

  //make the hit sound
  this.clip.onAudioQueue(2);

}
```

Die goes in the **gameOver** method before the director pauses.

```
//When the bird hits something, run this
gameOver() {

  //show the results
  this.result.showResult();

  //game is over
  this.isOver = true;

  //make the game over sound
  this.clip.onAudioQueue(3);

  //pause the game
  director.pause();
}
```

Go ahead and save and let's try it out. If all is working well, the game is playing very similarly to the original Flappy Bird game.

Now you can go back to the **GameCtrl** and **Bird** nodes and play with your speed and jump height to make the gameplay the way you want. Just make sure to save the scene after you edit it.



And that’s it! Congratulations on building your very own Flappy Bird game.

Conclusion

I hope you enjoyed making your first game and hope it was as fun making it as it was playing it. If you have issues, you can always look at the completed project on GitHub, available here:

LINK

This will allow you to look at the nodes and code to see if you missed something or placed it incorrectly. If you want to change anything to make it more like the game or add your own variety, it's all up to you. Go build the best game you and your friends can enjoy.

You can also check out our YouTube for a video version of this project and other videos that help you build your games on our channel. We hope this is the start of many years of game development and that this gives you the starting point to look deeper into the API and code to start building your dream game.

Have Fun!

Source Code

Ground.ts

```
import { _decorator, Component, Node, Vec3, UITransform, director, Canvas} from
'cc';
const { ccclass, property } = _decorator;

import { GameCtrl } from './GameCtrl';

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

    @property({
        type: Node,
        tooltip: 'First ground'
    })
    public ground1: Node;

    @property({
        type: Node,
        tooltip: 'Second ground'
    })
    public ground2: Node;

    @property({
        type: Node,
        tooltip: 'Third ground'
    })
    public ground3: Node;

    //Create ground width variables
    public groundWidth1:number;
    public groundWidth2:number;
    public groundWidth3:number;

    //make temporary starting locations
    public tempStartLocation1 = new Vec3;
    public tempStartLocation2 = new Vec3;
    public tempStartLocation3 = new Vec3;


    //get the gamespeeds
    public gameCtrlSpeed = new GameCtrl;
    public gameSpeed: number;

    //all the things we want to happen when we start the script
    onLoad(){

        this.startUp()

    }

     //preparing the ground locations
     startUp(){

        //get ground width
        this.groundWidth1 = this.ground1.getComponent(UITransform).width;
        this.groundWidth2 = this.ground2.getComponent(UITransform).width;
        this.groundWidth3 = this.ground3.getComponent(UITransform).width;

        //set temporary starting locations of ground
        this.tempStartLocation1.x = 0;
        this.tempStartLocation2.x = this.groundWidth1;
        this.tempStartLocation3.x = this.groundWidth1 + this.groundWidth2;

        //update position to final starting locations
        this.ground1.setPosition(this.tempStartLocation1);
        this.ground2.setPosition(this.tempStartLocation2);
        this.ground3.setPosition(this.tempStartLocation3);

    }

    //everytime the game updates, move the ground
    update(deltaTime: number) {

        //get speed of ground and background
        this.gameSpeed = this.gameCtrlSpeed.speed;

        //place real location data into temp locations
        this.tempStartLocation1 = this.ground1.position;
        this.tempStartLocation2 = this.ground2.position;
        this.tempStartLocation3 = this.ground3.position;

        //get speed and subtract location on x axis
        this.tempStartLocation1.x -= this.gameSpeed * deltaTime;
        this.tempStartLocation2.x -= this.gameSpeed * deltaTime;
        this.tempStartLocation3.x -= this.gameSpeed * deltaTime;


        //get the canvas size prepared
        const scene = director.getScene();
        const canvas = scene.getComponentInChildren(Canvas);


//check if ground1 went out of bounds. If so, return to the end of the line.
        if (this.tempStartLocation1.x <= (0 - this.groundWidth1)) {
            this.tempStartLocation1.x = canvas.getComponent(UITransform).width;
        }

        //same with ground2
        if (this.tempStartLocation2.x <= (0 - this.groundWidth2)) {
            this.tempStartLocation2.x = canvas.getComponent(UITransform).width;
        }

        //same with ground3
        if (this.tempStartLocation3.x <= (0 - this.groundWidth3)) {
            this.tempStartLocation3.x = canvas.getComponent(UITransform).width;
        }

        //place new locations back into ground nodes
        this.ground1.setPosition(this.tempStartLocation1);
        this.ground2.setPosition(this.tempStartLocation2);
        this.ground3.setPosition(this.tempStartLocation3);

    }
}
```

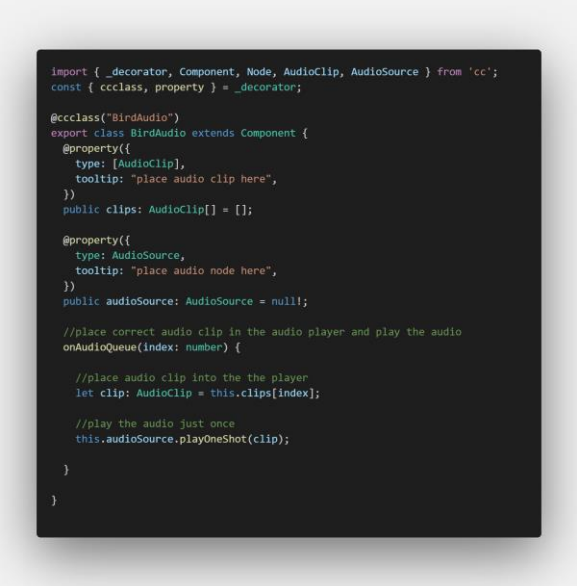
Results.ts

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

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

    @property({
        type: Label,
        tooltip: 'Current Score'
    })
    public scoreLabel: Label;

    @property({
        type: Label,
        tooltip: 'High Score'
    })
    public highScore: Label;

    @property({
        type: Label,
        tooltip: 'Try Again?'
    })
    public resultEnd: Label;

    //variables needed for the scores
    maxScore: number = 0; //saved high score
    currentScore: number; // current score while playing


//change current score to new score or back to zero then display the n
ew score
    updateScore(num:number){

        //update the score to the new number on the screen
        this.currentScore = num;

        //display new score
        this.scoreLabel.string = ('' + this.currentScore);

    }


    //resets the score back to 0 and hides game over information
    resetScore(){

        //reset score to 0
        this.updateScore(0);

        //hide high score and try again request
        this.hideResult();

        //reset current score label
        this.scoreLabel.string = ('' + this.currentScore);

    }

    //add a point to the score
    addScore(){

        //add a point to the score
        this.updateScore(this.currentScore + 1);

    }

    //show the score results when the game ends.
    showResult(){

        //check if it's the high score
        this.maxScore = Math.max(this.maxScore, this.currentScore);

        //activate high score label
        this.highScore.string = 'High Score is:' + this.maxScore;
        this.highScore.node.active = true;

        //activate try again label
        this.resultEnd.node.active = true;

    }

    //hide results and request for a new game when the new game starts
    hideResult(){

        //hide the high score and try again label.
        this.highScore.node.active = false;
        this.resultEnd.node.active = false;

    }
}
```

Bird.ts

```
import { _decorator, Component, Node, CCFloat, Vec3, Animation, tween } from 'cc';
const { ccclass, property } = _decorator;

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

    @property({
        type: CCFloat,
        tooltip: 'how high does he fly?'
    })
    public jumpHeight: number = 1.5;

    @property({
        type: CCFloat,
        tooltip: 'how long does he fly?'
    })
    public jumpDuration: number = 1.5;

    //Animation property of the bird
    public birdAnimation: Animation;

    //Temporary location of the bird
    public birdLocation: Vec3;

    //hit detection call
    public hitSomething:boolean;

    //all the actions we want done when we start the script.
    onLoad(){

        //Restart the bird
        this.resetBird();

        //Get the initial animation information
        this.birdAnimation = this.getComponent(Animation);

    }

    //reset the bird's location and hit detection
    resetBird(){

        //create original bird location
        this.birdLocation = new Vec3(0,0,0);

        //place bird in location
        this.node.setPosition(this.birdLocation);

        //reset hit detection
        this.hitSomething = false;

    }

    //have the bird fly up in the air
    fly(){

        //stop the bird animation immediately
        this.birdAnimation.stop();

        //start the movement of the bird
        tween(this.node.position)
            .to(this.jumpDuration, new Vec3(this.node.position.x, this.node.position.y + this.jumpHeight, 0), {easing: "smooth",
                onUpdate: (target:Vec3, ratio:number) => {
                    this.node.position = target;
                }
            })
            .start();

        //play the bird animation
        this.birdAnimation.play();

    }



}
```

BirdAudio.ts



```
import { _decorator, Component, Node, AudioClip, AudioSource } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("BirdAudio")
export class BirdAudio extends Component {
  @property({
    type: [AudioClip],
    tooltip: "place audio clip here",
  })
  public clips: AudioClip[] = [];

  @property({
    type: AudioSource,
    tooltip: "place audio node here",
  })
  public audioSource: AudioSource = null!;

  //place correct audio clip in the audio player and play the audio
  onAudioQueue(index: number) {

    //place audio clip into the the player
    let clip: AudioClip = this.clips[index];

    //play the audio just once
    this.audioSource.playOneShot(clip);

  }

}
```

Pipes.ts

```ts
import { _decorator, Component, Node, Vec3, screen, find, UITransform } from 'cc';
const { ccclass, property } = _decorator;

//make a random number generator for the gap
const random = (min,max) => {
    return Math.random() * (max - min) + min
}

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

    @property({
        type: Node,
        tooltip: 'Top Pipe'
    })
    public topPipe: Node;

    @property({
        type: Node,
        tooltip: 'Bottom Pipe'
    })
    public bottomPipe: Node;

    //temporary locations
    public tempStartLocationUp:Vec3 = new Vec3(0,0,0);  //Temporary location of the up pipe
    public tempStartLocationDown:Vec3 = new Vec3(0,0,0); //Temporary location of the bottom pipe
    public scene = screen.windowSize; //get the size of the screen in case we decide to change the content size

    //get the pipe speeds
    public game; //get the pipe speed from GameCtrl
    public pipeSpeed:number; //use as a final speed number
    public tempSpeed:number; //use as the moving pipe speed

    //scoring mechanism
    isPass: boolean; //Did the pipe pass the bird?

    //what to do when the pipes load
    onLoad (){

        //find GameCtrl so we can use the methods
        this.game = find("GameCtrl").getComponent("GameCtrl")

        //add pipespeed to temporary method
        this.pipeSpeed = this.game.pipeSpeed;

        //set the original position
        this.initPos();

        //set the scoring mechanism to stop activating
        this.isPass = false;

    }

    //initial positions of the grounds
    initPos() {

        //start with the initial position of x for both pipes
        this.tempStartLocationUp.x = (this.topPipe.getComponent(UITransform).width + this.scene.width);
        this.tempStartLocationDown.x = (this.bottomPipe.getComponent(UITransform).width + this.scene.width);

        //random variables for the gaps
        let gap = random(90,100);  //passable area randomized
        let topHeight = random(0,450);   //The height of the top pipe

        //set the top pipe initial position of y
        this.tempStartLocationUp.y = topHeight;

        //set the bottom pipe initial position of y
        this.tempStartLocationDown.y = (topHeight - (gap * 10));

        //set temp locations to real ones
        this.topPipe.setPosition(this.tempStartLocationUp.x, this.tempStartLocationUp.y);
        this.bottomPipe.setPosition(this.tempStartLocationDown.x, this.tempStartLocationDown.y);

    }

    //move the pipes as we update the game
    update(deltaTime: number){

        //get the pipe speed
        this.tempSpeed = this.pipeSpeed * deltaTime;

        //make temporary pipe locations
        this.tempStartLocationDown = this.bottomPipe.position;
        this.tempStartLocationUp = this.topPipe.position;

        //move temporary pipe locations
        this.tempStartLocationDown.x -= this.tempSpeed;
        this.tempStartLocationUp.x -= this.tempSpeed;

        //place new positions of the pipes from temporary pipe locations
        this.bottomPipe.setPosition(this.tempStartLocationDown);
        this.topPipe.setPosition(this.tempStartLocationUp);

        //find out if bird past a pipe, add to the score
        if (this.isPass == false && this.topPipe.position.x <= 0)
        {

            //make sure it is only counted once
            this.isPass = true;

            //add a point to the score
            this.game.passPipe();

        };

        //if passed the screen, reset pipes to new location
        if (this.topPipe.position.x < (0 - this.scene.width)){

            //create a new pipe
            this.game.createPipe();

            //delete this node for memory saving
            this.destroy();

        };

    }
}
```

PipePool.ts

```ts
import { _decorator, Component, Node, Prefab, NodePool, instantiate } from 'cc';
const { ccclass, property } = _decorator;

import { Pipes } from './Pipes';

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

    @property({
        type: Prefab,
        tooltip: 'The prefab of pipes'

    })
    public prefabPipes = null;

    @property({
        type: Node,
        tooltip: 'Where the new pipes go'
    })
    public pipePoolHome;

    //create initial properties
    public pool = new NodePool;
    public createPipe:Node = null;

    initPool(){

        //build the amount of nodes needed at a time
        let initCount = 3;

        //fill up the node pool
        for (let i = 0; i < initCount; i++) {
            // create the new node
            let createPipe = instantiate(this.prefabPipes); //instantiate means make a copy of the orginal

            // put first one on the screen. So make it a child of the canvas.
            if (i == 0) {
                this.pipePoolHome.addChild(createPipe);
            } else {
                //put others into the nodePool
                this.pool.put(createPipe);
            }
        }

    }

    addPool() {

        //if the pool is not full add a new one, else get the first one in the pool
        if (this.pool.size() > 0) {
            //get from the pool
            this.createPipe = this.pool.get();
        } else {
            //build a new one
            this.createPipe = instantiate(this.prefabPipes);
        }

        //add pipe to game as a node
        this.pipePoolHome.addChild(this.createPipe);

      }

    reset() {

        //clear pool and reinitialize
        this.pipePoolHome.removeAllChildren();
        this.pool.clear();
        this.initPool();

    }

}
```

GameCtrl.ts

```
import { _decorator, Component, Node, CCInteger, Input, input, EventKeyboard, KeyCode,
director, Contact2DType, Collider2D, IPhysics2DContact } from 'cc';
const { ccclass, property } = _decorator;

import { Ground } from './Ground';
import { Results } from './Results';
import { Bird } from './Bird';
import { PipePool } from './PipePool';
import { BirdAudio } from './BirdAudio';

@ccclass("GameCtrl")
export class GameCtrl extends Component {
  @property({
    type: Component,
    tooltip: "Add ground prefab owner here",
  })
  public ground: Ground;

  @property({
    type: CCInteger,
    tooltip: "Change the speed of ground",
  })
  public speed: number = 200;

  @property({
    type: CCInteger,
    tooltip: "Change the speed of pipes",
  })
  public pipeSpeed: number = 200;

  @property({
    type: Results,
    tooltip: "Add results here",
  })
  public result: Results;

  @property({
    type: Bird,
    tooltip: "Add Bird node",
  })
  public bird: Bird;

  @property({
    type: PipePool,
    tooltip: "Add canvas here",
  })
  public pipeQueue: PipePool;

  @property({
    type: BirdAudio,
    tooltip: "add audio controller",
  })
  public clip: BirdAudio;

  //needed to tell the game it's over
  public isOver: boolean;

  //things to do when the game loads
  onLoad() {

    //get listener started
    this.initListener();

    //reset score to zero
    this.result.resetScore();

    //game is over
    this.isOver = true;

    //pause the game
    director.pause();

  }

  //listener for the mouse clicks and keyboard
  initListener() {

    //if keyboard key goes down, go to onKeyDown
    input.on(Input.EventType.KEY_DOWN, this.onKeyDown, this);

    //if an mouse or finger goes down, do this
    this.node.on(Node.EventType.TOUCH_START, () => {

      //if we are starting a new game
      if (this.isOver == true) {

        //reset everything and start the game again
        this.resetGame();
        this.bird.resetBird();
        this.startGame();

      }

      //if the game is playing
      if (this.isOver == false) {

        //have Bird fly
        this.bird.fly();

        //make the bird go swoosh
        this.clip.onAudioQueue(0);
      }

    })

  }

  //for testing purposes, we use this. But hide as comments after you finish the game
  onKeyDown(event: EventKeyboard) {
    switch (event.keyCode) {
      case KeyCode.KEY_A:
        //end game
        this.gameOver();
        break;
      case KeyCode.KEY_P:
        //add one point
        this.result.addScore();
        break;
      case KeyCode.KEY_Q:
        //reset score to zero
        this.resetGame();
        this.bird.resetBird();
    }
  }

  //when the bird hits something, run this
  gameOver() {

    //show the results
    this.result.showResult();

    //game is over
    this.isOver = true;

    //make the game over sound
    this.clip.onAudioQueue(3);

    //pause the game
    director.pause();

  }

  //when the game starts again, do these things.
  resetGame() {

    //reset score, bird, and pipes
    this.result.resetScore();

    //reset the pipes
    this.pipeQueue.reset();

    //game is starting
    this.isOver = false;

    //get objects moving again
    this.startGame();

  }

  //what to do when the game is starting.
  startGame() {

    //hide high score and other text
    this.result.hideResult();

    //resume game
    director.resume();

  }

  //when a pipe passes the bird, do this
  passPipe() {

    //passed a pipe, get a point
    this.result.addScore();

    //make the point ring
    this.clip.onAudioQueue(1);

  }

  //when the old pipe goes away, create a new pipe
  createPipe() {

    //add a new pipe to the pipe pool
    this.pipeQueue.addPool();

  }

  //check if there was contact with the bird and objects
  contactGroundPipe() {

    //make a collider call from the bird's collider2D component
    let collider = this.bird.getComponent(Collider2D);

    //check if the collider hit other colliders
    if (collider) {
      collider.on(Contact2DType.BEGIN_CONTACT, this.onBeginContact, this);
    }

  }

  //if you hit something, tell the bird you did
  onBeginContact(selfCollider: Collider2D, otherCollider: Collider2D, contact:
IPhysics2DContact | null) {

    //will be called once when two colliders begin to contact
    this.bird.hitSomething = true;

    //make the hit sound
    this.clip.onAudioQueue(2);

  }

  //hit detection call
  birdStruck() {

    //make a call to the gameBrain to see if it hit something.
    this.contactGroundPipe()

    //if we hit it, tell the game to call game over.
    if (this.bird.hitSomething == true)
    {
        this.gameOver();
    }

  }

  //every time the game updates, do this
  update(){

    //if the game is still going, check if the bird hit something
    if (this.isOver == false){
        this.birdStruck();
    }

  }

}
```