Software Studio
軟體設計與實驗

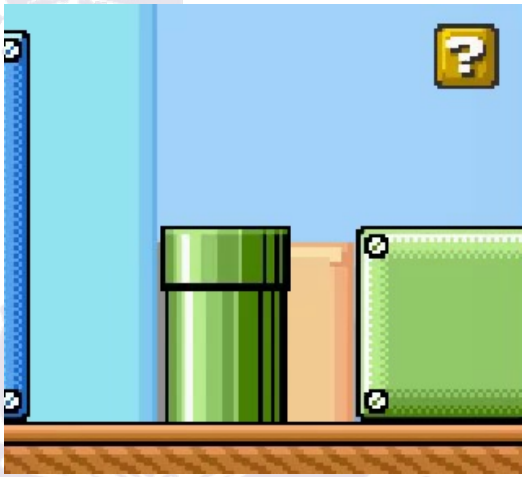
Cocos Creator : Action System & Scheduler

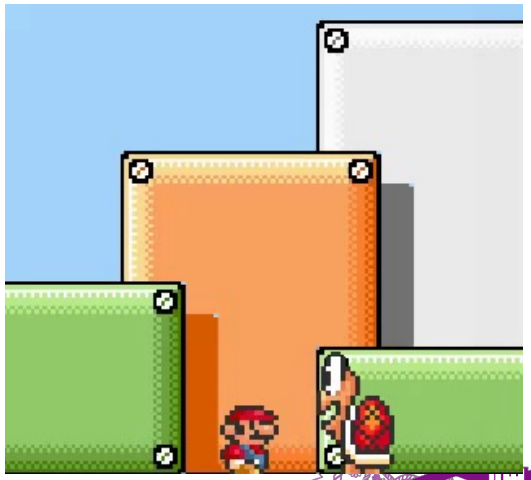
Hung-Kuo Chu
Department of Computer Science
National Tsing Hua University

CS2410

Action System

- The action system is used to apply **displacement, zoom, rotate and all the other kinds of actions** to the nodes within a designated time.





Usage

- Compared with the animation system, the action system provides a set of **API interfaces** for programmers.
- The action system is more suitable for making simple animations such as **simple deformation and displacement.**

A Glance of Action System APIs

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

```
// the action will make the node move to position(10, 10) within 2 seconds
let action = cc.moveTo(2, 10, 10);

// execute the action
this.node.runAction(action);
```

- We can also use the following APIs to stop running actions.

```
// stop one action
this.node.stopAction(action);

// stop all actions
this.node.stopActions();
```

Tags

- Programmers can also get and control the actions by setting tags for actions.

```
let action = cc.moveTo(2, 10, 10);

// set tag for the action
let actionTag = 1;
action.setTag(actionTag);

// get the action by tag
this.node.getActionByTag(actionTag);

// stop one action by tag
this.node.stopActionByTag(actionTag);
```

Action Categories

- Cocos Creator supports various kinds of actions which can be divided into several categories, including **basic action, container action, callback action, and slow motion**.
- Since there are too many action categories, we won't cover all of them. Please refer to the official document (API list) for details.

Basic Action

- Basic action is the action to achieve all kinds of **deformation and displacement animation**.
- Basic action can be divided into two classes:
  - **interval action**
  - **free action**

Interval Action

- Interval action is a gradual change action that is done in a certain time interval.

```
let action;

// the node moves to position(10, 10) within 2 seconds
action = cc.moveTo(2, 10, 10);
// the node moves (10, 10) pixels from current position within 2 seconds
action = cc.moveBy(2, 10, 10);

// the node rotates  to 60.0 degrees within 2 seconds
action = cc.rotateTo(2, 60.0);
// the node rotates 60.0 degrees from current degrees within  2 seconds
action = cc.rotateBy(2, 60.0);
```

Interval Action

```
// the node scales to 0.5 in both X and Y within 2 seconds
action = cc.scaleTo(2, 0.5);
// the node scales to 0.5 in X and 0.4 in Y within 2 seconds
action = cc.scaleTo(2, 0.5, 0.4);

// the node scales by 0.5 in both X and Y within 2 seconds
action = cc.scaleBy(2, 0.5);
// the node scales by 0.5 in X and 0.4 in Y within 2 seconds
action = cc.scaleBy(2, 0.5, 0.4);

// the node jumps to position(20, 30) with 4 times jumps within 2 seconds, jump height is 50
action = cc.jumpTo(2, 20, 30, 50, 4);
// the node jumps (20, 30) pixels from current position with 4 times jumps within 2 seconds,
jump height is 50
action = cc.jumpBy(2, 20, 30, 50, 4);

// the opacity of node fades to 0 within 2 seconds
action = cc.fadeTo(2, 0);
```

Interval Action: Examples

MoveTo

RotateTo

ScaleTo

JumpTo

FadeOut

Free Action

- Different from interval actions, free actions run immediately.

```
let action;

// show the node immediately
action = cc.show();

// hide the node immediately
action = cc.hide();

// remove the node from its parent node
action = cc.removeSelf();

// flip the node according to X-axis
action = cc.flipX();
```

Container Action

- The container action can organize actions in different ways, such as:
  - **Sequential** action
  - **Synchronization** action
  - **Repetitive** action
  - **Repeat forever** action
  - **Speed** action
  - **Combination** action

Sequential Action

- Sequential action makes a series of child actions run one by one.
- Use **cc.sequence** to create a sequential action.

```
// the action will make the node move back and forth
let action = cc.sequence(cc.moveBy(1, 200, 0), cc.moveBy(1, -200, 0));
this.node.runAction(action);
```

Synchronization Action

- Synchronization action synchronizes the execution of a series of child actions.
- Use **cc.spawn** to create a synchronization action.

```
// the action will make the node zoom in twice while it moves upwards
let action = cc.spawn(cc.moveBy(1, 0, 100), cc.scaleTo(1, 2));
this.node.runAction(action);
```

Repetitive Action

- Repetitive action is used to repeat one action several times.
- Use **cc.repeat** to create a repetitive action.

```
// the action will make the node move back and forth 5 times
let action = cc.repeat(
    cc.sequence(cc.moveBy(1, 200, 0), cc.moveBy(1, -200, 0)) , 5);

this.node.runAction(action);
```

Repeat Forever Action

- Repeat forever action can make the target action repeat forever until it is stopped manually.
- Use **cc.repeatForever** to create a repeat forever action.

```
// the action will make the node move back and forth and keep repeating
let action = cc.repeatForever(
    cc.sequence(cc.moveBy(1, 200, 0), cc.moveBy(1, -200, 0)) );

this.node.runAction(action);
```

Speed Action

- Speed action can alter the execution rate of the target action to make it quicker or slower.
- Use **cc.speed** to create a speed action.

```
// the action will make the node zoom in twice while it moves upwards within 0.5 seconds
let action = cc.speed(
    cc.spawn(cc.moveBy(1, 0, 100), cc.scaleTo(1, 2)) , 2);

this.node.runAction(action);
```

Combination Action

- Different container types can be combined.
- Cocos Creator also provides **link-form** API for the container type actions, including **repeat, repeatForever, speed** actions

Combination Action

```
// the action will make the node do a complicated heart-beating animation
let action;
action = cc.sequence(
    cc.spawn(cc.scaleTo(0.1, 0.8, 1.2), cc.moveTo(0.1, 0, 10)),
    cc.spawn(cc.scaleTo(0.2, 1, 1), cc.moveTo(0.2, 0, 0)),
    cc.delayTime(0.2),
    cc.spawn(cc.scaleTo(0.1, 1.2, 0.8), cc.moveTo(0.1, 0, -10)),
    cc.spawn(cc.scaleTo(0.2, 1, 1), cc.moveTo(0.2, 0, 0))
    ).speed(1.5).repeat(3); // the link-form API makes the animation play five
                                times faster and repeat 3 times
```

Callback Action

- Callback action belongs to free action, which executes after a series of actions is finished.
- Callback action can be declared as follows:

```
let finished = cc.callFunc(this.myMethod, this, opt);
```

Callback Action

- The first parameter in a callback action is the callback function used to deal with callback event.
- The function can be **anonymous**.
- For example, the two piece of codes are the same:

```
public myMethod() {
    cc.log("Hello world!");
}

let finished = cc.callFunc(this.myMethod,
                this, opt);
```

```
let finished = cc.callFunc(function() {
    cc.log("Hello world!");
}, this, opt);
```

Callback Action

- The second parameter specifies the context of the callback method, that is, we usually take **this** as value.

```
let finished = cc.callFunc(this.myMethod, this, opt);
```

Callback Action

- The third parameter is used to pass the parameters which callback function needs.

```
let score = 50;

// the score will add 100 points after the action is finished
let finished = cc.callFunc(function(target, score) {
    this.score += score;
}, this, 100);
```

Combine with cc.sequence

- After declaring a callback action, we can use **cc.sequence** to execute a series of actions and executing a callback action in the end.

```
let score = 50;
let finished = cc.callFunc(function(target, score) {
    this.score += score;
}, this, 100);

let action = cc.sequence(cc.moveBy(1, 100, 0), cc.fadeOut(1), finished);
```

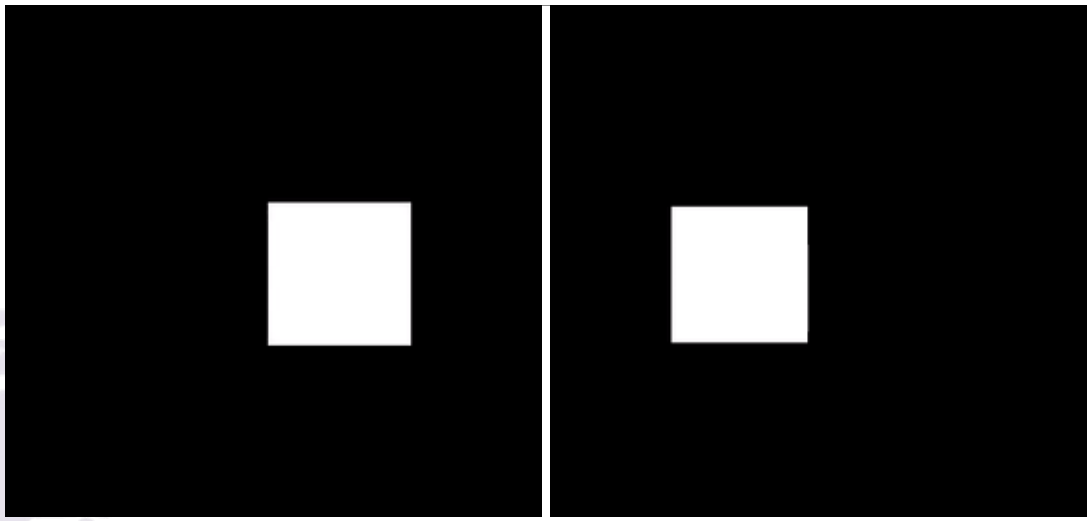
Slow Motion

- Slow motion is used to alter the time curve of the basic action to give the action fast in/out, ease in or other complicated special effects.
- Slow motion **cannot exist alone**.
- **Only interval actions** support slow motion.

Slow Motion: Example

- We can modify **scaleTo** action by:

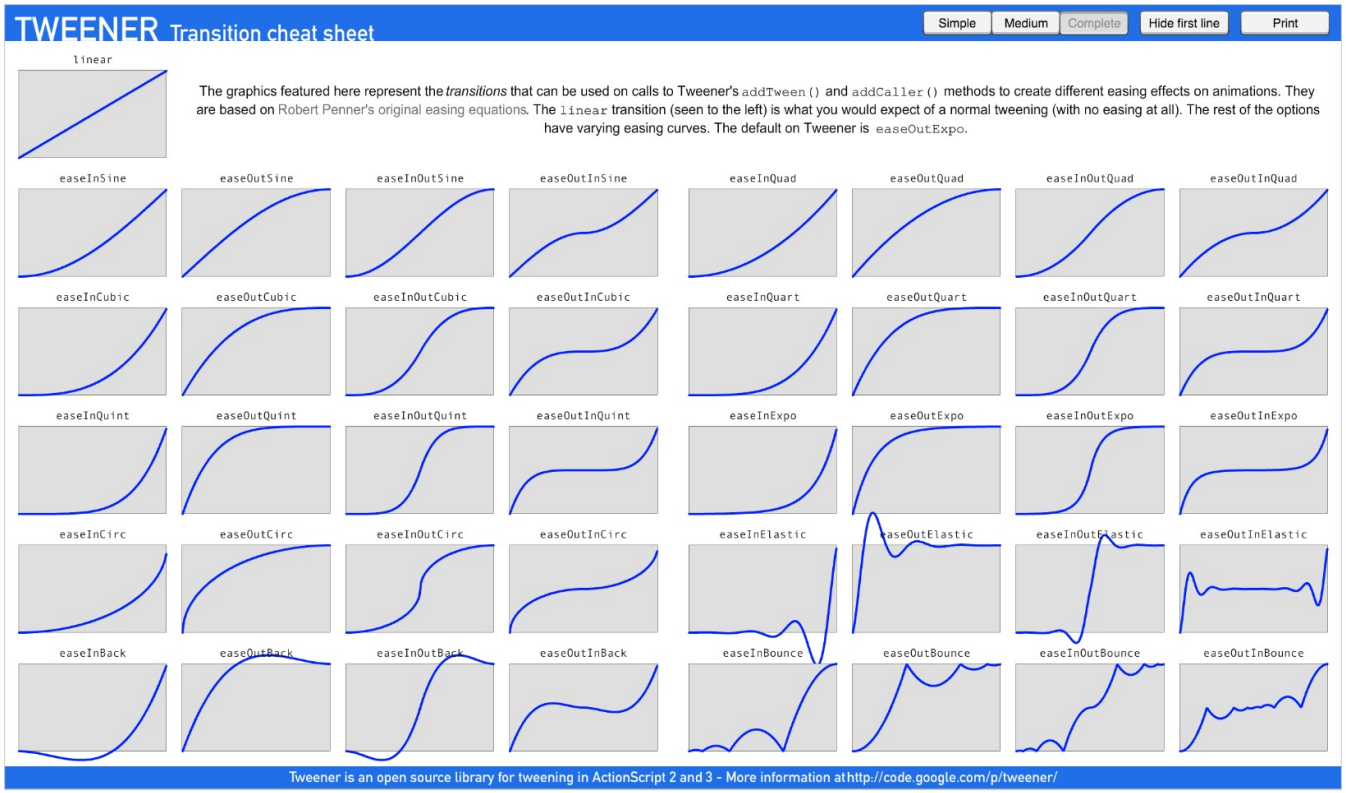
```
let scaleUp = cc.scaleTo(1, 2);
let scaleDown = cc.scaleTo(1, 1);
scaleUp.easing(cc.easeIn(3.0));
this.node.runAction(cc.sequence(scaleUp, scaleDown)).repeatForever();
```



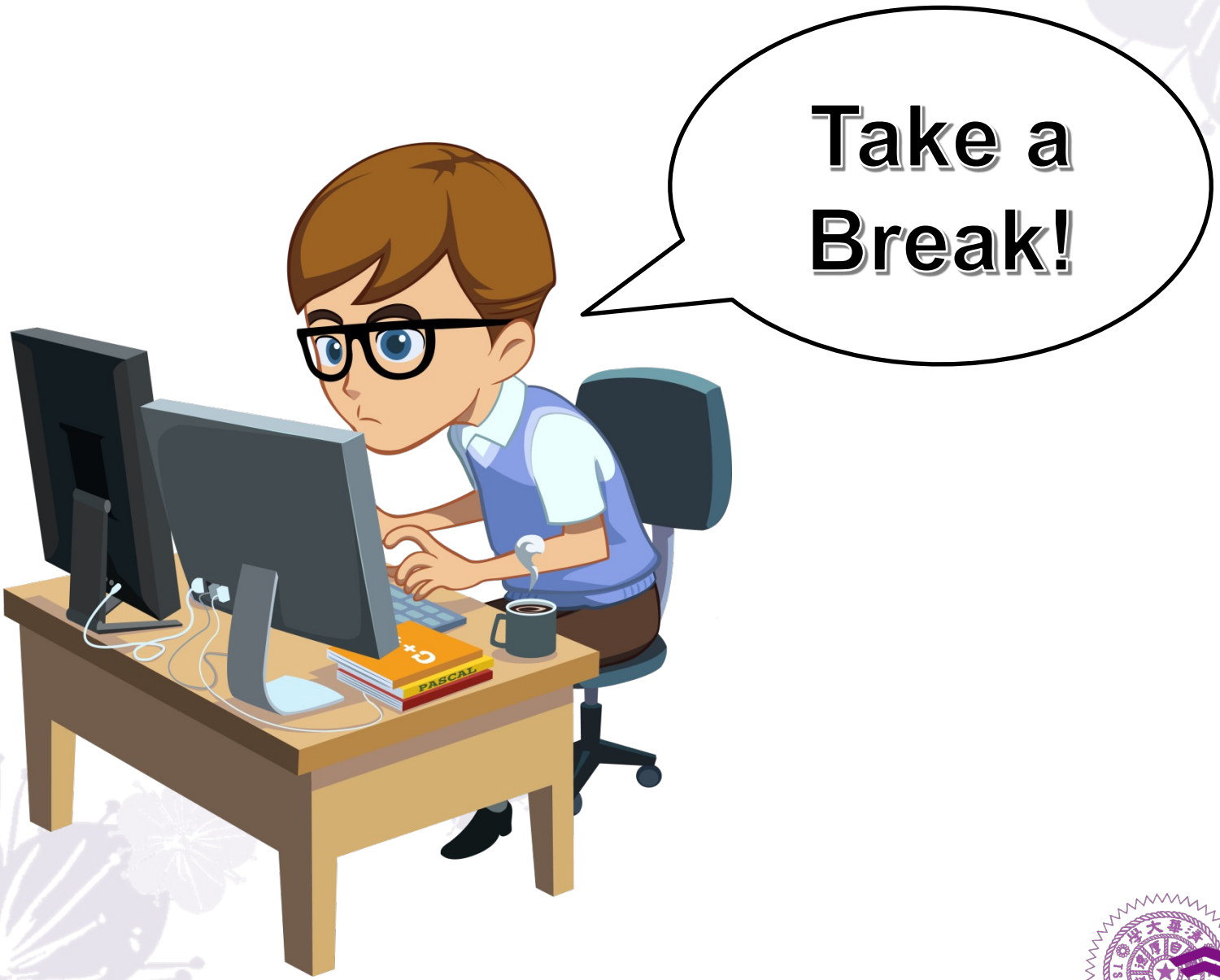
wo/ slow motion w/ slow motion

Slow Motion: Time Curves

- The picture below shows the time curves of different slow motions:



Ref: Slow Motion


Take a
Break!
C++
PASCAL

Scheduler

- Scheduler is a **timer** component for programmers to design time-related functions.
- Compared to javascript timing events, such as **setTimeout** and **setInterval**, scheduler is preferred because it is more powerful, and it combines better with other components in Cocos Creator.

Start a Timer

- We can easily use scheduler by calling **this.schedule:**

```
public myMethod() {
    cc.log("Hello world!");
}

// the scheduler will execute once every 3 seconds
this.schedule(this.myMethod, 3);
```

Start a Timer

- Like action system, the function also can be **anonymous**.
- For example, the two piece of codes are the same:

```
public myMethod() {
    cc.log("Hello world!");
}

// the scheduler will execute once
every 3 seconds
this.schedule(this.myMethod, 3);
```

```
// the scheduler will execute once
every 3 seconds
this.schedule(function() {
    cc.log("Hello world!");
    }, 3);
```

Control Timer Event

- Besides execution interval, we can also control the **time of repetition** and **schedule start delay.**

```
let interval = 2; // time interval in the unit of second
let repeat = 3; // time of repetition
let delay = 5; // start delay

// the schedule will execute 3+1 times every 2 seconds after 5 seconds
this.schedule(function() {
    cc.log("Hello world!"); }, interval, repeat, delay);
```

Schedule Once

- If we only want to execute an event once, we can use **scheduleOnce**.
- For example, the two piece of codes are the same:

```
// the schedule will execute once after 2 seconds
this.schedule(function() { cc.log("Hello world!"); }, 0, 0, 2);

// the schedule will execute once after 2 seconds
this.scheduleOnce(function() { cc.log("Hello world!"); }, 2);
```

Cancel Schedule

- We can use **unschedule** to cancel a schedule or use **unscheduleAllCallbacks** to cancel all the schedules of this component.

```
this.count = 0;
this.callback = function() {
    if(this.count == 3) {
        this.unschedule(this.callback);
    }
    cc.log("Hello world!");
    this.count += 1;
}
this.schedule(this.callback, 1); // the schedule will be cancelled after executing 4 times
```

Practice

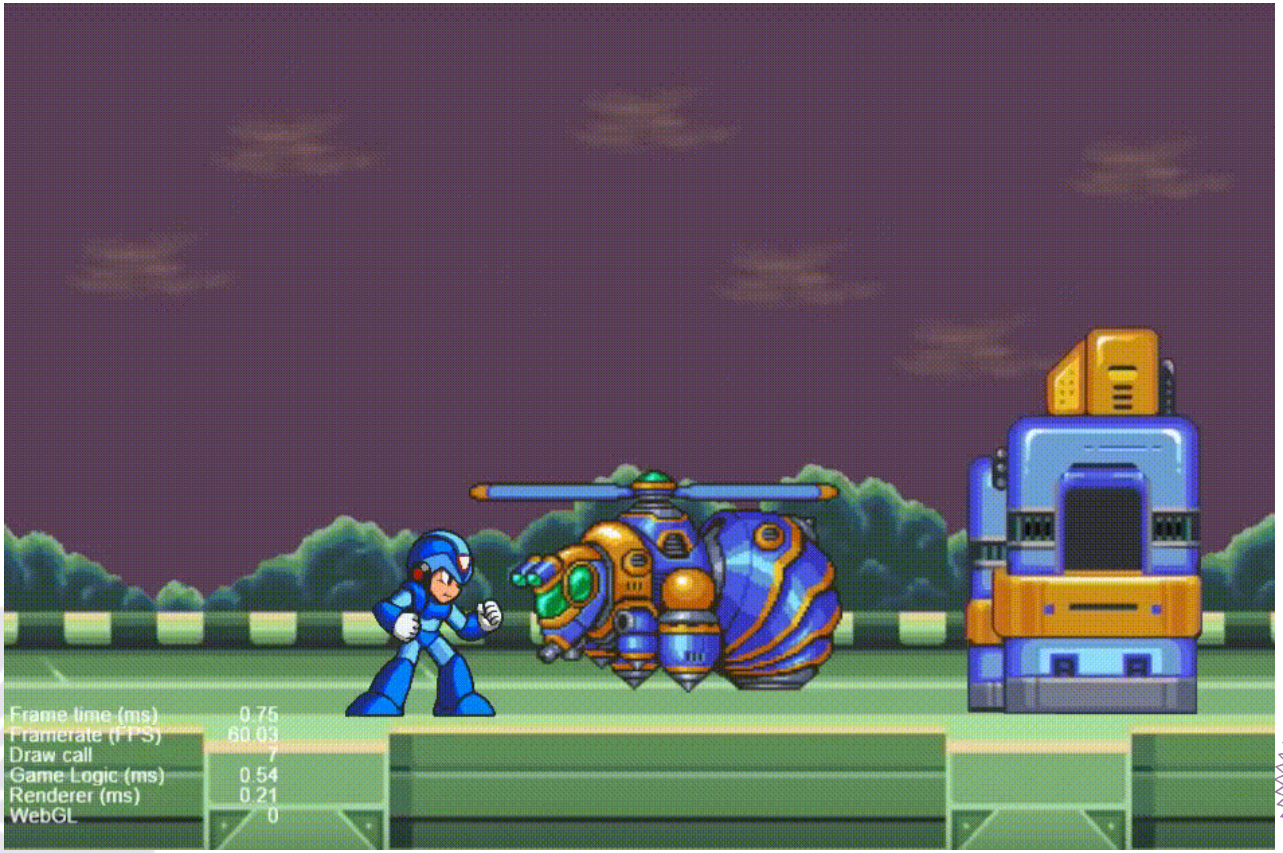
- We are going to add fancy actions to the RockMan example in Physics lecture.

Reborn Event: Original

- In the original version, the player will be killed and reborn frequently as below:

Reborn Event: Revised

- We can use **scheduler** to set time interval to improve the effect as below.



Reborn Event: How?

- Edit the Player.ts script as follows:
  - Add two new variables **isReborn** & **rebornTime**
  - Modify **onBeginContact** function
  - Modify **playerMovement** function

Reborn Event: Revised

```
private isReborn: boolean = false; // use the flag to check whether the player reborn
private rebornTime: number = 0.4; // cool-down time

onBeginContact(contact, self, other) {
    if(other.node.name == "ground") {
        cc.log("Rockman hits the ground");
        this.onGround = true;
    } else if(other.node.name == "block") {
        cc.log("Rockman hits the block");
        this.onGround = true;
    // keep the player in the invincible mode when she is in the reborn state
    } else if(other.node.name == "enemy" && !this.isReborn) {
        cc.log("Rockman hits the enemy");
        this.isDead = true;
    }
}
```

Player.ts

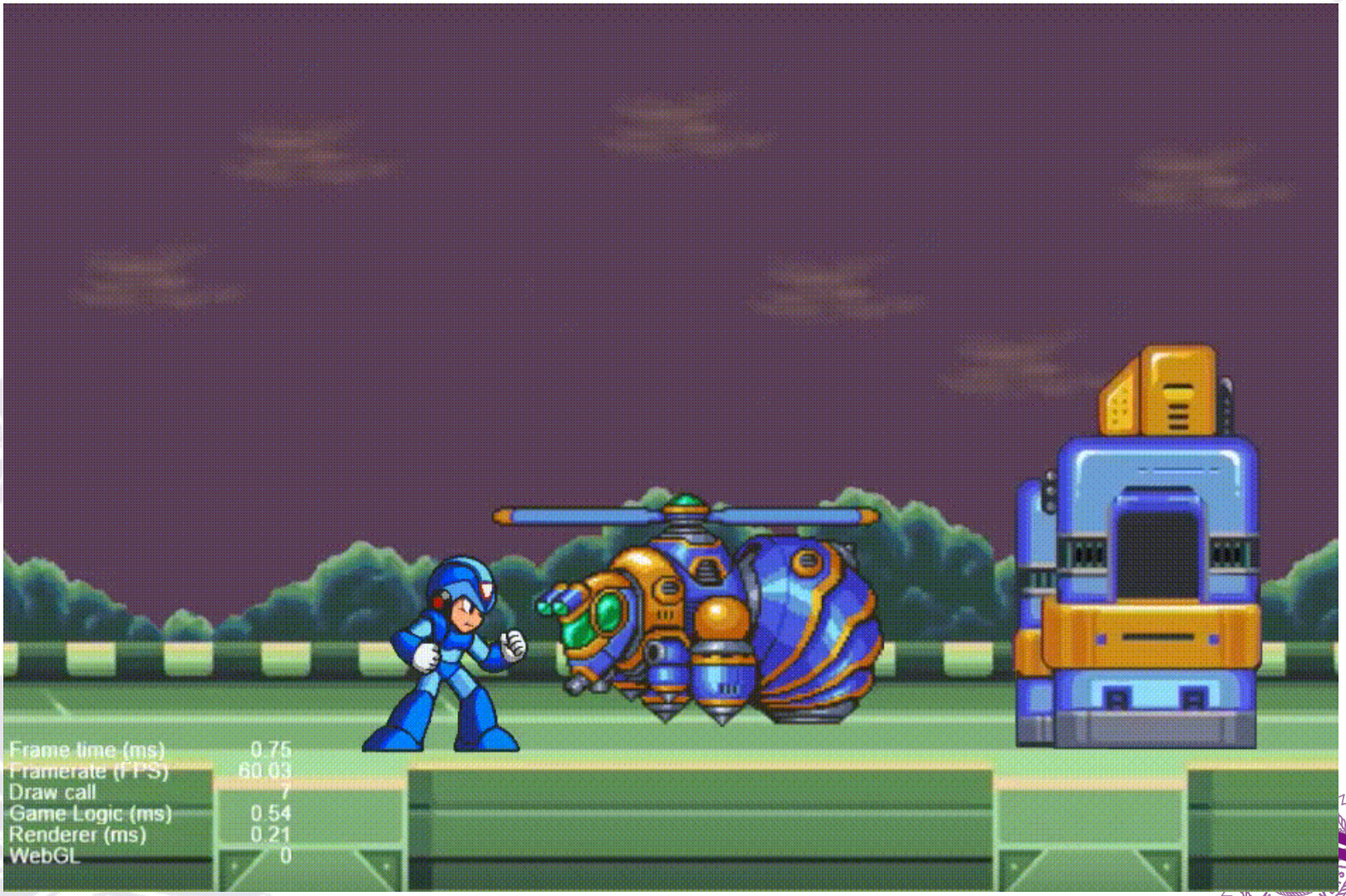
Reborn Event: Revised

```
private playerMovement(dt) {
    if(this.isDead && !this.isReborn) {
        this.node.getComponent(cc.RigidBody).linearVelocity = cc.v2(0, 0);
        this.node.position = cc.v2(-192, 255);
        this.isReborn = true;

        // reset the player's reborn state in a period (invisible period)
        this.scheduleOnce(function(){
            this.isDead = false;
            this.isReborn = false;
        }, this.rebornTime);
        return;
    }
}
```

Player.ts

Let’s Play!


Frame time (ms) 0.75
Framerate (FPS) 60.03
Draw call 7
Game Logic (ms) 0.54
Renderer (ms) 0.21
WebGL 0

Bullet Event: Original

- In the original version, the player can shoot as many bullets as the shooting key is pressed.

Bullet Event: Revised

- We can use **scheduler** to set time interval to improve the effect as below.

Bullet Event: How?

- Edit the Player.ts script as follows:
  - Add two new variables **canCreateBullet** & **bulletInterval**
  - Modify **onKeyDown** function
  - Modify **playerMovement** function
  - Modify **createBullet** function

Bullet Event: Revised

```
private canCreateBullet: boolean = true;
private bulletInterval: number = 0.2; // timer interval for creating bullet

onKeyDown(event) {
    cc.log("Key Down: " + event.keyCode);
    if(event.keyCode == cc.KEY.z) { // press key z to turn left
        this.zDown = true;
        this.xDown = false;
    } else if(event.keyCode == cc.KEY.x) { // press key x to turn right
        this.xDown = true;
        this.zDown = false;
    } else if(event.keyCode == cc.KEY.k) { // press key k to jump
        this.kDown = true;
    } else if(event.keyCode == cc.KEY.j) { // press key j to shoot
        this.jDown = true;
    }
}
```

Player.ts

Bullet Event: Revised

```
private playerMovement(dt) {
    // a bullet can be created only when canCreateBullet is True
    if(this.jDown && this.canCreateBullet) {
        this.createBullet();
    }
}

private createBullet() {
    this.canCreateBullet = false;
    // using the scheduleOnce and this.bulletInterval to implement a cool-down timer
    // for the next shooting event
    this.scheduleOnce(function(){
        this.canCreateBullet = true;
    }, this.bulletInterval);
    let bullet = cc.instantiate(this.bulletPrefab);
    bullet.getComponent('Bullet').init(this.node);
}
```

Player.ts

Let’s Play!


Frame time (ms) 0.74
Framerate (FPS) 59.99
Draw call 7
Game Logic (ms) 0.49
Renderer (ms) 0.25
WebGL 0

Destroy the Bullets: Original

- In the original version, we destroy the bullet when it touches the left/right boundary.

Destroy the Bullets: Revised

- We would like to create an effect where each bullet has a fixed lifetime.

Bullet Attenuation Effect

- Edit the Bullet.ts script as follows:
  - Modify **bulletMove** function
  - Use **cc.fadeout** to create a fade out effect
  - Use **cc.moveBy** to move the bullet.
  - Use **cc.spawn** to simultaneously move the bullet and fade it out.
  - Use **cc.callFunc** to set a callback function for destruction
  - Use **cc.sequence** to call the callback function after the bullet movement is finished.

Bullet Attenuation Effect

```
private bulletMove()
{
    let moveDir = null;
    // move bullet to 300 far from current position in 0.8 seconds
    if(this.node.scaleX > 0) moveDir = cc.moveBy(0.8, 300, 0);
    else moveDir = cc.moveBy(0.8, -300, 0);

    // bullet will fade out and move simultaneously
    let action = cc.spawn(moveDir, cc.fadeOut(0.8));

    // after moving finished, it will destroy itself
    let finished = cc.callFunc(function() {
        this.node.destroy();
    });
    // use cc.sequence to call actions in order
    this.node.runAction(cc.sequence(action, finished));
}
```

Bullet.ts

Let’s Play!


Frame time (ms) 0.8
Framerate (FPS) 60.01
Draw call 6
Game Logic (ms) 0.5
Renderer (ms) 0.3
WebGL 0


thank you!
Question?