



# Phaser, Part II

Understanding more about Phaser



## Today we'll learn about:

- How to use game states
- Animating objects
- Adding interactivity to your game
- Using variables to store important information



# Game States



# Game States

Most games have at least 2 states.

**Main menu**

**Gameplay**



# Game States

In JavaScript, we can separate each game state into its own file. This makes our code easier to manage.

**MainMenu.js**

**GamePlay.js**



# Game States

In some games, you might want to separate the win and lose states from gameplay.

MainMenu.js

GamePlay.js

WinState.js

LoseState.js



## Creating a Game State

To create a main menu state, **add** it to the **state** list. This should be done near your game instance code so it applies globally.

```
var game = new Phaser.Game(GAME_WIDTH, GAME_HEIGHT, Phaser.AUTO);
```

```
//state is added below the game instantiation
```

```
game.state.add('MainMenu', myGame.MainMenu);
```

State name

JavaScript object  
to reference



# Starting your game with a state

Command your game to begin with a certain state using the **state.start()** function.

```
game.state.start ( 'MainMenu' ) ;
```

State name





# Switching game states

You can also start another game state after the game begins.

```
myGame.mainMenu = function(game) {}

myGame.mainMenu.prototype = {
  preload: function() {

  },
  create: function() {
    this.state.start('GamePlay');
  }
}
```



# Loading images



## The `preload()` function

Phaser needs to know what images to prepare before the game can be displayed.

**in a game with states:**

```
preload: function() {  
    //commands go here  
}
```

**in a game without states:**

```
function preload() {  
    //commands go here  
}
```



## Loading Images

There are several types of images in Phaser:

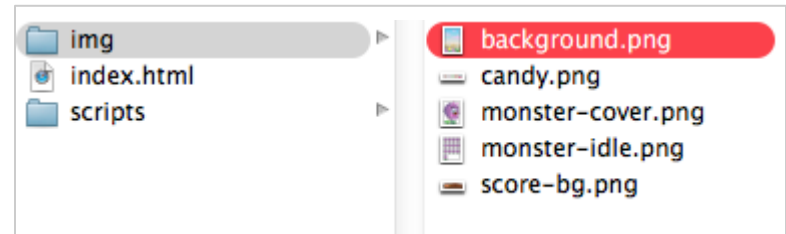
- **image** - static, no animation
- **spritesheet** - sprite with animation
- **tilemap** - environment objects



## Locating image files

All files should be referenced from the **root**, the main folder where your project is located.

`'img/background.png'` →





## Static images

To load a static image, tell your **game** object to load an **image**. Within the parentheses, name the image so it can be referenced later, then tell Phaser where to find it in your folder.

```
this.load.image('background', 'img/background.png');
```

Name of image                      Location of image file



## Why do we use **this** instead of **game**?

Loading and adding images should always be done in relation to the **game** object, which is referred to in myGame's **GamePlay** state function.

```
var myGame = {  
  Gameplay: function(game) {}  
};
```



## Why do we use **this** instead of myGame?

JavaScript's **this** allows you to create, reference, and modify an object's properties within the scope of a function. In this case, that object is **game**.

```
var myGame = {  
  GamePlay: function(game) {}  
};
```





# Why do we use **this** instead of myGame?

In games without multiple states, you can reference the **game** object directly.

```
var game = new Phaser.Game(640, 960, Phaser.AUTO,
    '{
    preload: preload,
    create: create,
    update: update
    });

function preload() {
    game.load.image('sky', img/background.png);
}
```



## Loading sprites

Sprites require widths and heights since they might have multiple animation frames. The last two numbers are the sprite's **width** and **height**.

```
game.load.spritesheet('player', 'img/player.png', 32, 64);
```



# Using images



## The `create()` function

Once the preload function is complete, Phaser needs you determine how the game will start.

**in a game with states:**

```
create: function() {  
    //commands go here  
}
```

**in a game without states:**

```
function create() {  
    //commands go here  
}
```



## The **create()** function

The **create()** function lets you set up variables, objects, and the look of your game.

```
function create() {  
    myGame.score = 0;  
}
```



## Drawing objects

You can draw, or place, objects onscreen using Phaser's **add.sprite()** function.

```
game.add.sprite(0, 0, 'background');
```

X, Y

Name of image  
to use



## Drawing objects

For important things like your player character, you can define a **global variable** that can be referenced throughout the game's functions.

```
myGame.player = game.add.sprite(30, 60, 'player');
```



# Placing objects in dynamic locations

Want to reference a location that might change?  
Use variables or JavaScript's **Math** functions.

## Using variables:

```
GAME_WIDTH: 640;  
GAME_HEIGHT: 960;  
game.add.sprite(GAME_WIDTH,  
GAME_HEIGHT, 'player');
```

## Using Math:

```
game.add.sprite(100, Math.floor  
(Math.random() * 640), 'player');
```





## Referencing an object's dimensions

You can also reference the dimensions of your object when placing it in your game.

```
game.add.sprite(24, this.height - 64, 'player');
```

'this' refers to  
the sprite object



# Animating objects

Animate an object by adding to its **animations** list.

```
myGame.character.animations.add('walk');
```

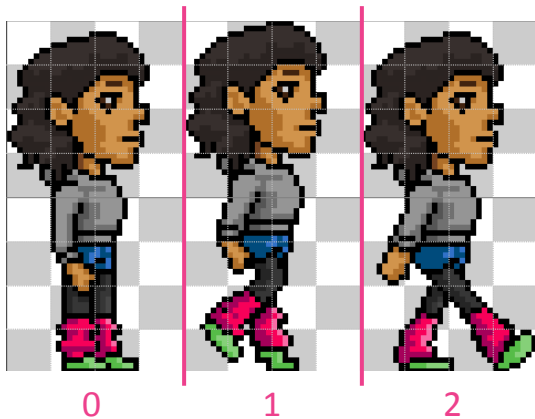
└─  
Animation  
name



# Animating objects

You can choose animation frames using brackets.

```
myGame.character.animations.add('walk', [0, 1, 2]);
```

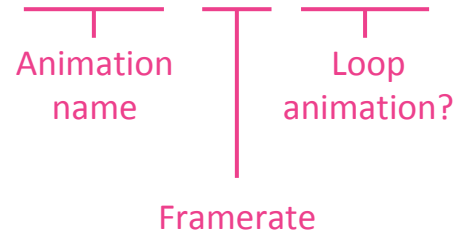




# Animating objects

To trigger an animation, use the play command.

```
myGame.character.animations.play('walk', 30, false);
```





# Physics and collision



# Physics

Phaser has a set of systems called **Physics** that allow you to check when objects touch. You must enable physics for each object that will be checked.

```
game.physics.enable(object, Phaser.Physics.ARCADE);
```

Object  
name

Type of physics – must be in CAPS



# Physics

Phaser has 3 types of physics.

- **Arcade:** `Phaser.Physics.ARCADE`
- **Ninja:** `Phaser.Physics.NINJA`
- **P2:** `Phaser.Physics.P2JS`



# Arcade Physics

Treats all objects as rectangles. Quickest to load because it only has one type of shape.







## Ninja Physics

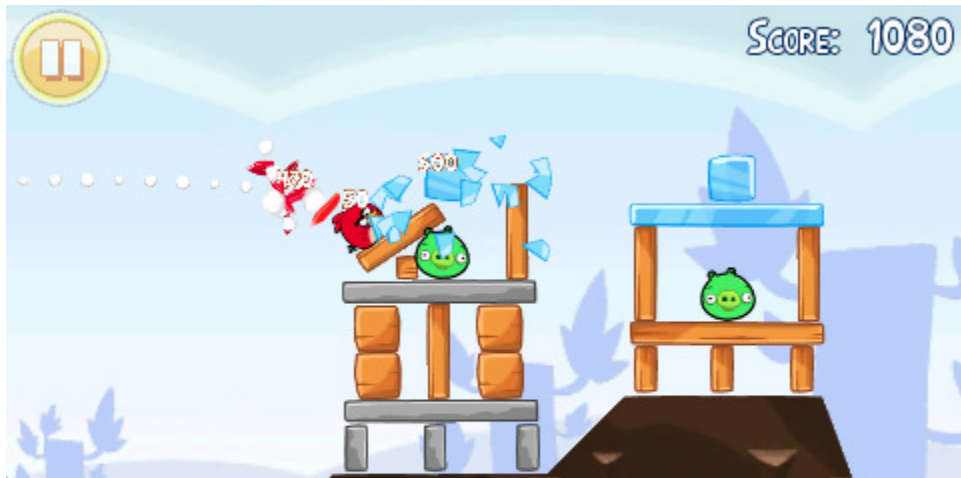
Allows for slopes and rotation. This means you can create curved shapes and ellipses.





## P2 Physics

You can make a full-fledged physics game with polygons, angles, and swinging like Angry Birds.





## Using gravity in your game

Want your game to have gravity? Enable a physics system for your entire game in the **create()** function. You can have horizontal & vertical gravity.

```
game.physics.startSystem(Phaser.Physics.ARCADE);
```

```
game.physics.arcade.gravity.y = 250;
```

Physics type  
(lowercase)



## Body settings

All objects with enabled physics have a **body**, which allows you to modify physics-related properties.

You can add bodies to objects without physics, too:

```
myGame.object.enableBody = true;
```



## Body settings

Want to move an object? Use the **body.velocity.x** or **y** property. In Phaser, the **velocity** defines movement in **pixels per second**.

```
myGame.player.body.velocity.x = 150;
```



## Body settings

Want an object to ignore the physics system you enabled? Use the **body.immovable** setting.

```
myGame.player.body.immovable = true;
```



## Body settings

To stop objects from moving off screen, use the **body.collideWorldBounds** setting.

```
myGame.player.body.collideWorldBounds = true;
```



# Adding interactivity





# Updating the Game

Unlike preload and create, which only run once each, the **update()** function runs every frame.

in a game with states:

```
update: function() {  
    //commands go here  
}
```

in a game without states:

```
function update() {  
    //commands go here  
}
```



## Updating the Game

`update()` is where your player is told to move, the score is updated, and text changes.

```
function update() {  
    myGame.score += 1;  
}
```



## Adding interactivity

You can add interactivity to your game using a variety of **input** types:

- Keyboard
- Mouse
- Touch
- Gamepad



## Keyboard input

The **keyboard** input object allows you to create interactivity using keys. Enable the keys you want to use in the **create()** function using **addKey()**.

```
var pauseKey = game.input.keyboard.addKey(Phaser.Keyboard.P);
```

Key to add from Phaser's  
list of keys



# Keyboard input

You can use the **onDown** event listener to trigger a function when a button is pressed.

```
pauseKey.onDown.add(pauseGame, this);
```

Diagram illustrating the arguments for the `add` method:

- `pauseGame` is identified as the **Function**.
- `this` is identified as the **Object to reference in function**.



# Keyboard input

This code creates a global object that contains arrow keys:

- up
- right
- down
- left

**isDown** checks if a key is down and returns a boolean value.

```
var cursors;

function create() {
    cursors = game.input.keyboard.
    createCursorKeys();
}

function update() {
    if (cursors.left.isDown) {
        player.body.velocity.x -= 150;
        player.animations.play('left');
    }
}
```



# Mouse/touch input

Phaser checks if mouse or touch interactivity is available when **this.input** is used.

```
game.input.onDown.add(startGame, this);
```

Function      Object to  
reference  
in function  
(game)



## Mouse/touch location

You can reference the location of the mouse or touch event using **input.x** and **input.y**.

```
myGame.player.x = game.input.x;  
myGame.player.y = game.input.y;
```





## Gamepad input

Browser-based gamepad input is currently in the infancy stage. At the moment, Phaser supports **Xbox 360 gamepad** input in Chrome.





## Gamepad input

Phaser supports up to 4 gamepads. You can reference each using the **input.gamepad** object.



pad1



pad2



pad3



pad4



# Gamepad input

This code creates the **pad1** object & checks if its **left d-pad button** on is down.

When the left button is down, **player** moves to the left.

```
var pad1;

function create() {
    pad1 = game.input.gamepad.pad1;
}

function update() {
    if(pad1.isDown(Phaser.Gamepad.XBOX360_DPAD_LEFT)) {
        player.x--;
    }
}
```



# Functions & Collision



## Checking collision

Using Phaser's physics, you can trigger a function when two objects overlap:

```
game.physics.arcade.overlap(player, enemy, playerDies);
```

Physics type  
(lowercase)

Object 1

Object 2

Function to run



# Global functions

**Global functions**, or functions defined outside the scope of all objects & functions, can be called anywhere in your code.

When a global function is called, JavaScript jumps to the definition and runs that code.

## GamePlay.js

### Game object

- preload
- create
  - `globalFunction();`
- update
  - `globalFunction();`

### `globalFunction()` definition

- `do stuff;`



# Global functions

This global function adds 1 to the score & destroys the **meat** object when it is touched by **player**.

The objects that collide are passed through via the function parameters.

```
var eatMeat = function(player, meat) {  
    // If meat is hit, remove it!  
    meat.kill();  
  
    //increase score  
    myGame.score += 1;  
  
};
```



## Question

How would you make the following scenarios from Super Mario Bros with functions and physics?



1. Jump



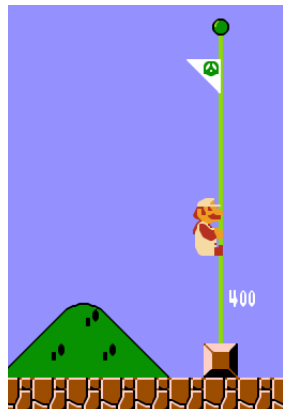
2. Touch enemy



3. Hit brick block



4. Hit "?" box



5. Touch flagpole





# Groups



## Groups

Have an object you want to repeat onscreen and give the same properties? Make a group.

```
myGame.myGroup = game.add.group();
```



## Using Groups

You can instantiate objects and add them to a group in multiple ways.

### Creating and adding a new object:

```
// Create an object, name it, and add to group  
var item = myGame.myGroup.create(0, 0,  
  'item');
```

### Adding an existing object:

```
// Add an existing object to  
a group  
myGame.myGroup.add  
  (groupItem);
```



## Collision with groups

You can trigger functions when an object collides with a group. The function below is triggered when **player** collides with an object in **myGroup**.

```
game.physics.arcade.collide(player, myGroup,  
hurtPlayer);
```

Group  
name



## Question

- Why are groups useful in games?
- What kinds of purposes could you use groups for in a game?



# Text



## Displaying text

To draw text on the screen, **add** it to the game. The text should be instantiated in **create()**.

```
myGame.scoreText = game.add.text(90, 24, '0');
```

X, Y      Text  
content



## Styling text

Style your text using the optional fourth parameter, the **style** object, which can have many properties.

### Inline:

```
game.add.text(90, 24, '0', {  
    fill: "#ccc"  
});
```

### With a variable:

```
var fontStyle = {fill: "#ccc"};  
game.add.text(90, 24, '0', fontStyle);
```





## Styling text

Types of styles you can add to your text:

- **font:** "bold 32px Arial" / "20pt Times New Roman"
- **fill:** "#000" / "#000000" / "red"
- **align:** "center" / "left" / "right"
- **stroke:** "#000" / "#000000" / "red"
- **strokeThickness:** 1
- **wordWrap:** true
- **wordWrapWidth:** 100



## Using variables with text

You can use variables in text by replacing (or adding to) the third parameter.

```
myGame.scoreText = game.add.text(90, 24, myGame.score);
```



## Updating text

In the **update()** function, you can change text using the **setText()** function.

### With a string:

```
myGame.scoreText.setText("1");
```

### With a variable:

```
myGame.scoreText.setText(myGame.score);
```



# Using variables with strings

JavaScript literally lets you add variables to strings.

```
myGame.scoreText.setText ( "Score: " + myGame.score );
```

string                      variable



## To do:

Make a small interactive game in Phaser. Use the things you learned this week:

- States
- Art & animation
- Inputs
- Collision
- Groups
- Text



# Thanks! Questions?

@cattsmall  
catt@code liberation.org