Adding keyboard input

Here we set the paddle movement velocity at a 600 pixels. Next, add the following code to the `mainState` object:

```
mainStatefunction

backgroundGraphics

ballSprite

paddleLeftSprite

paddleRightSprite

paddleLeft_up

paddleLeft_down

paddleRight_up

paddleRight_down
```

Lines 45-48 will be used to add both the left and right paddles controls with its own up and down input controls. We will add keyboard input to these next by creating a new function called `initKeyboard` after the `initPhysics` function.

```
initPhysicsfunction

physicsstartSystemPhaserPhysicsARCADE
```

```
physicsenableballSpritePhaserPhysicsARCADE

ballSpritecheckWorldBounds

ballSpritecollideWorldBounds

ballSpriteimmovable

ballSpritebounce

initKeyboardfunction
```

Add the following code in the `initKeyboard` function:

```
initKeyboardfunction

paddleLeft_upinputkeyboardaddKeyPhaserKeyboard

paddleLeft_downinputkeyboardaddKeyPhaserKeyboard

paddleRight_upinputkeyboardaddKeyPhaserKeyboard

paddleRight_downinputkeyboardaddKeyPhaserKeyboard
```

For the left paddle, we will assign the A and Z keys to control the up and down movement. For the right paddle, the UP and DOWN arrow keys will be used instead.

The `game.input.keyboard.addKey` function creates a new Key object with a specific keyCode argument to keep track of when that key is pressed. You can find the complete list of keyCodes in the `Phaser.Keyboard` class file from lines 579 to 678.

To enable the keyboard when the game starts and disable it during demo mode, add the following code to the `enablePaddles` function:

```
enablePaddlesfunctionenabled
```

```
paddleLeftSpritevisibleenabled

paddleRightSpritevisibleenabled

paddleLeft_upenabledenabled

paddleLeft_downenabledenabled

paddleRight_upenabledenabled

paddleRight_downenabledenabled
```

Next, we need to call the `initKeyboard` function from the `create` function to intialise the keyboard controls:

```
createfunction

initGraphics

initPhysics

initKeyboard

startDemo
```

Adding the arcade physics body

Going back to the `gameProperties` object, we'll set the constant movement speed for both paddles. Add the following highlighted code:

```
paddleLeft_x

paddleRight_x

paddleVelocity
```

Since both paddles function exactly the same, we'll create a grouped object to manage both paddles simultaneously. Let's call it `paddleGroup` and add it to the `mainState` function:

```
mainStatefunction

backgroundGraphics

ballSprite

paddleLeftSprite

paddleRightSprite

paddleGroup

paddleLeft_up

paddleLeft_down

paddleRight_up

paddleRight_down
```

Now we'll add the arcade physics to the paddles. Go to the `initPhysics` function and add the following code:

```
initPhysicsfunction

physicsstartSystemPhaserPhysicsARCADE

physicsenableballSpritePhaserPhysicsARCADE

ballSpritecheckWorldBounds
```

```
ballSpritecollideWorldBounds

ballSpriteimmovable

ballSpritebounce

paddleGroupgroup

paddleGroupenableBody

paddleGroupphysicsBodyTypePhaserPhysicsARCADE

paddleGrouppaddleLeftSprite

paddleGrouppaddleRightSprite

paddleGroupsetAll'checkWorldBounds'

paddleGroupsetAll'body.collideWorldBounds'

paddleGroupsetAll'body.immovable'
```

Line 102 creates a new group object. Next, we add some default properties to the group so that any new object added to the group will inherit these properties.

By default, we will want all newly added objects to have its physics body enabled and use the arcade physics system. We do that in lines 103-104.

At lines 106-107, we add our left and right paddle sprites to the group.

Next, we'll set the same property for all objects in the group by calling the `setAll` function:

- Line 109: the `checkWorldBounds` property is set to `true`. This will enable collision checking between the paddles and the game world boundaries.
- Line 110: to prevent both paddles from going outside the world boundaries, we

set the physics body property `body.collideWorldBounds` to `true`.
- Line 111: to prevent the paddles from being pushed away when the ball collides with either paddle, we set the physics body property `body.immovable` property to `true`.

Looking again at the `enablePaddles` function, let's make a minor change to our paddle visibility code and add a new line of code:

```
enablePaddlesfunctionenabled

paddleGroupsetAll'visible'enabled

paddleGroupsetAll'body.enable'enabled

paddleLeft_upenabledenabled

paddleLeft_downenabledenabled

paddleRight_upenabledenabled

paddleRight_downenabledenabled
```

By using the `setAll` function, we can easily set the `visible` and `body.enable` properties of both paddles to use the `enabled` parameter.

Moving the paddles

Next, add the following code to the `moveLeftPaddle` function:

```
moveLeftPaddlefunction

paddleLeft_upisDown

paddleLeftSpritevelocitygamePropertiespaddleVelocity
```

```
paddleLeft_downisDown

paddleLeftSpritevelocitygamePropertiespaddleVelocity

paddleLeftSpritevelocity
```

Add the following code to the `moveRightPaddle` function:

```
moveRightPaddlefunction

paddleRight_upisDown

paddleRightSpritevelocitygamePropertiespaddleVelocity

paddleRight_downisDown

paddleRightSpritevelocitygamePropertiespaddleVelocity

paddleRightSpritevelocity
```

Notice how both the `moveLeftPaddle` and `moveRightPaddle` functions are almost identical. Here how it works:

- First, we check if the up key `isDown`. If the up key is currently being pressed, we set the y velocity of the paddle physics body to a negative value. A negative y velocity value moves the sprite upwards.
- If the up key is not being pressed, we then check if the down key is being pressed instead. If the down key is being pressed, we set the y velocity to a positive value which moves the sprite downwards.
- If neither up nor down keys are being pressed, the last `else` condition code block will run. This will sets our paddle y velocity to 0 and causes it stop completely.

To get this working, we need to call
both the `moveLeftPaddle` and `moveRightPaddle` functions within the `update` function:

```
updatefunction
```

moveLeftPaddle

moveRightPaddle

Click here[1] to download the source codes up to this point. Here is our current work in progress:

Hitting the ball

In part 1[2] of this series, I mentioned that the original Pong paddles were divided into 8 segments that determine the return angle of the ball when it collided with the paddles.

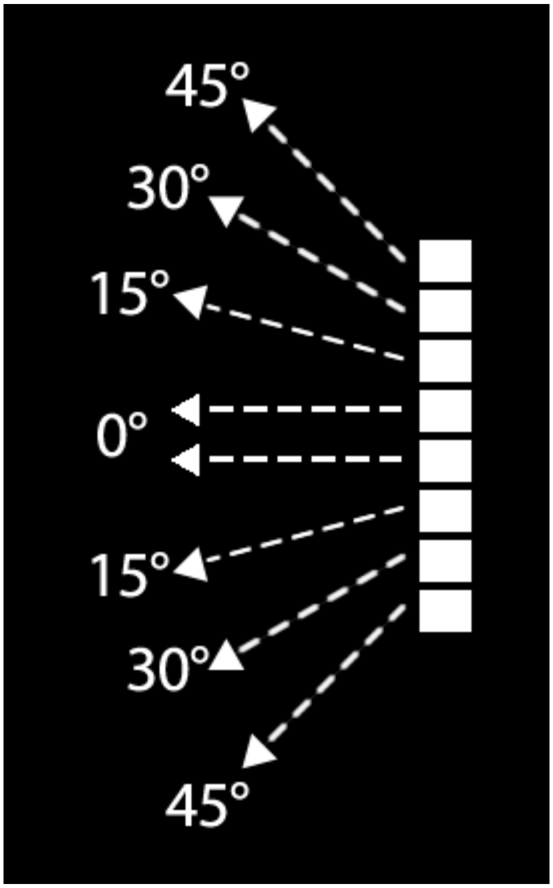
Let’s add the following code to the `gameProperties` object:

```
paddleLeft_x

paddleRight_x

paddleVelocity

paddleSegmentsMax

paddleSegmentHeight

paddleSegmentAngle
```

Here, at line 10, we set the maximum segments to a value of 4. This will be used to divide the top half into 4 segments and bottom half into another 4 segments.

Our current paddle graphic height is 32 pixels. To ensure that each segment is the same height, I have divided the 32 pixels into 8 segments with each segment being 4 pixels.

Here is the illustration used from part 1 to shows each segment and its return angle.



For each segment, there will be an increment of 15 degrees as the ball collides towards to outer edges of the paddles. The centre 2 segments will return the ball at a perfect 0 degree angle while the outer edge will return the ball at a 45 degree angle.

We need another function to perform collision detection between the paddles and balls. Add a new function called `collideWithPaddle` after the `moveRightPaddle` function:

```
moveRightPaddlefunction

paddleRight_upisDown

paddleRightSpritevelocitygamePropertiespaddleVelocity

paddleRight_downisDown

paddleRightSpritevelocitygamePropertiespaddleVelocity
```

```
paddleRightSpritevelocity

collideWithPaddlefunctionpaddle
```

The `collideWithPaddle` requires 2 parameters to work:

- The `ball` parameter which is a reference to the ball sprite.
- The `paddle` parameter which is a reference to the paddle sprite.

Now to add in the code to make the `collideWithPaddle` function work:

```
collideWithPaddlefunctionpaddle

returnAngle

segmentHitfloorpaddlegamePropertiespaddleSegmentHeight

segmentHit>=gamePropertiespaddleSegmentsMax

segmentHitgamePropertiespaddleSegmentsMax

segmentHit<=gamePropertiespaddleSegmentsMax

segmentHitgamePropertiespaddleSegmentsMax

paddlegamePropertiesscreenWidth

returnAnglesegmentHitgamePropertiespaddleSegmentAngle

physicsarcadevelocityFromAnglereturnAnglegameSettingsballVelocityballSpritevelocity

returnAnglesegmentHitgamePropertiespaddleSegmentAngle

returnAngle

returnAngle
```

physicsarcadevelocityFromAnglereturnAnglegameSettingsballVelocityballSpritevelocity

Two variables are declared:

- The returnAngle variable will be used to calculate the angle the ball will be returned at.
- The segmentHit variable will be used to determine which segment on the paddle is hit.

The centre two segments will have a segmentHit value of 0 while the outer most segments will have a value of 3.

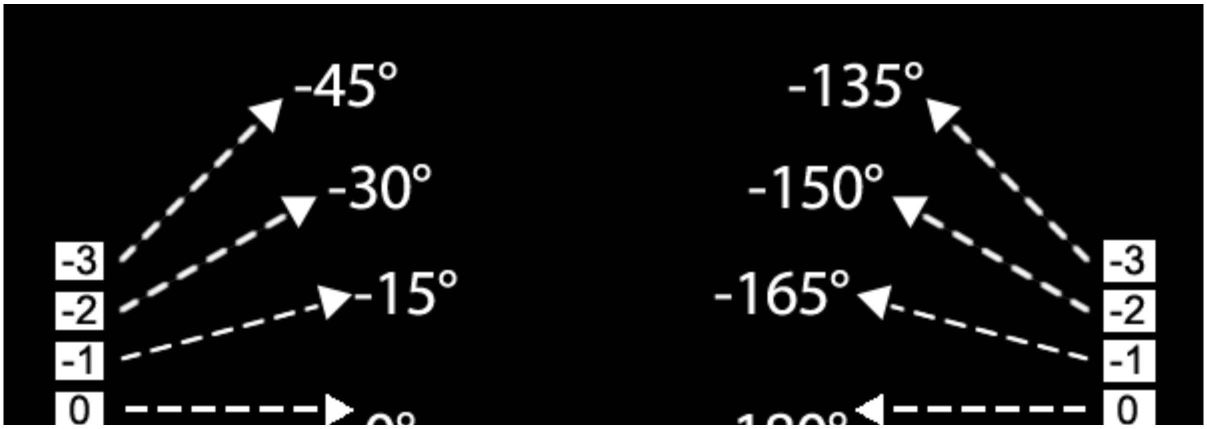
As it's possible to exceed the paddleSegmentsMax value of 4, lines 194-198 will check when that happens.

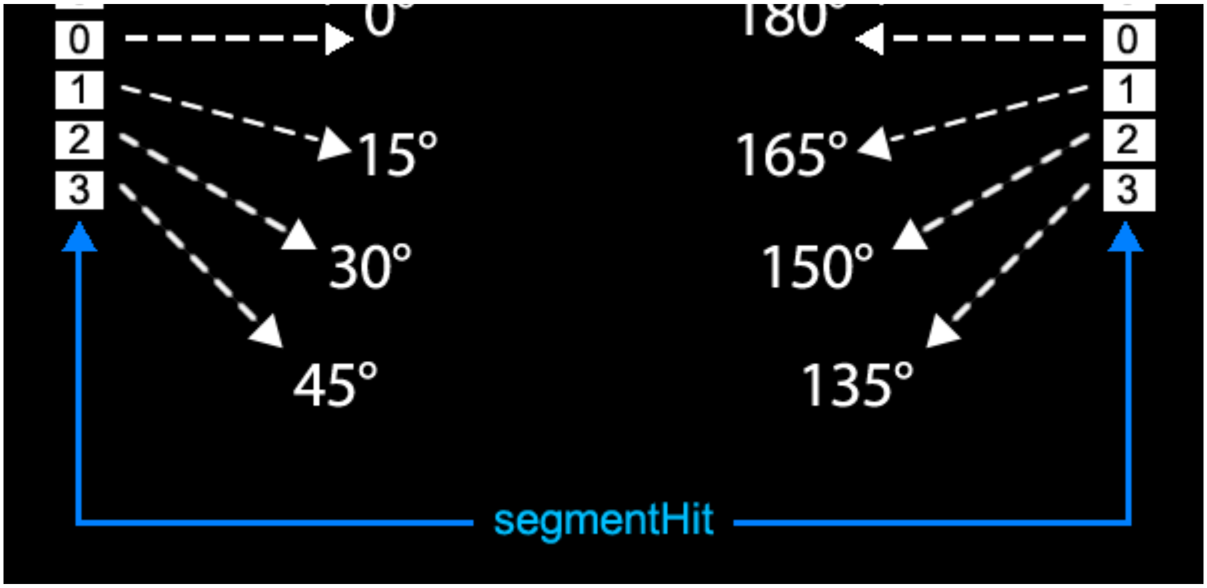
The -1 is added at the end of the calculation at lines 195 and 197 to limit the values of segmentHit to a range of -3 to 3. This final value is used to calculate the returnAngle of the ball by multiplying it by the the paddleSegmentAngle value of 15.

The next set of conditions from lines 200-210 checks whether the ball has hit the left or right paddle. The first if condition checks if the paddle is on the left side of the game world while the second condition checks if the paddle is on the right.

If the right paddle is hit, we need to offset the angle by 180 degrees to give us the correct angle in returning the ball sprite.

Here is an illustration of how to return angle is calculated:





After calculating the return angle of the ball, we need to reset the velocity and angle for the ball to bounce off the paddles. In lines 202 and 209, we use the arcade physics `velocityFromAngle` function to calculate the new velocity of the ball. Here, we use 3 arguments:

- The `angle`: value is in degrees.
- The `speed`: the velocity to move at.
- The `point`: the Point object to apply the x and y properties to. In this case we apply it to the ball's arcade physics body.

One final line of code to complete this step. In the `update` function, add this line of code:

```
updatefunction

moveLeftPaddle

moveRightPaddle

physicsarcadeoverlapballSpritepaddleGroupcollideWithPaddle
```

At line 75, we call the arcade physics `overlap` function to check if the ball sprite overlaps the paddles in the paddle group. Here, we use 5 arguments:

- The `object1`: The first object or array of objects to check. We use our ball sprite for the first object.
- The `object2`: The second object or array of objects to check. Here we use our paddle group that contains both paddles.
- The `overlapCallback`: An optional callback function that is called if the objects overlap. The two objects from the first two arguments will be passed to the `collideWithPaddle` function in the same order.
- The `processCallback`: A callback function that lets you perform additional checks against the two objects if they overlap. We only use this if we need to perform any additional verification before the `overlapCallback` function is called. As there is no need for this, we set it to `null`.
- The `callbackContext`: The context in which to run the callbacks. This will be needed later on to apply any further modifications or calculations in the `collideWithPaddle` function.

Here's our current work in progress:

Click here[3] to download the source codes.

So far so good. In the next step[4], we'll look at adding scoring and resetting the game when a player has won.

1. https://github.com/zekechan/phaser-html5-tutorial-pong/releases/download/1.0/5a-Making_the_paddles_move.zip
2. http://zekechan.net/getting-started-html5-game-development-pong1/
3. https://github.com/zekechan/phaser-html5-tutorial-pong/releases/download/1.0/5b-Adding_collision_detection.zip
4. http://zekechan.net/getting-started-html5-game-development-pong6/