Using Functions

To start, extract all the files to a Love folder called Proj06.

Because there's lots of cutting and pasting in this lesson, I've made some of the changes already in the attached main.lua file. Once we get a bit on you can start making the further changes yourself.

We've seen previously that functions are little bundles of code that perform a specific job. Functions can have arguments passed to them that affect what they do.

```
function eat(food)
    Some instructions here about how to eat food
end
```

The we can call

```
eat("carrots")
eat("hamburger")
eat("ice cream")
```

We don't have to repeat the instructions on how to eat each time. Those instructions are hidden away inside the function eat.

Sometimes a function needs to send a message back to the caller. This is called "returning a value". In our example

```
result = eat("ice cream")
```

result might contain the value true. Whereas in

```
result = eat("broccoli")
```

the result might contain the value false meaning that the command to eat broccoli failed. Go figure.

Spaceships and Other Space Junk

In our game we so far have one object that moves around the screen, our spaceship. Today we're going to add a few more. Our game needs a friendly satellite but also some menacing asteroids.

At the moment, the instructions to move the spaceship are in the love.update() function. There are also instructions that explain what to do once the spaceship reaches the edge of the window.

Imagine if we added a satellite. We'd have to duplicate all those instructions but with the table satellite rather than spaceship. That's quite a bit of work. Now image that we want to add 50 asteroids. I certainly don't want to type the instructions 50 more times.

We're going to be smart and create a function called moveObject(). This function will accept a table as an argument. As long as the table contains four keys called posX, posY, velX and velY, the function will be able to calculate a new posX and posY.

So what will our function look like from the outside? How will we call it?

```
moveObject(spaceship)
```

Here's what it might look like on the inside.

```
function moveObject(spaceObject)
   some instructions
end
```

The name spaceObject could be anything. Maybe flyingGoat. Whatever you put in the function definition is how you'll refer to the variable inside the function.

So we call moveObject(spaceship) but inside we refer to flyingGoat. Or maybe we'll stick to spaceObject since it kind of describes what the variable is.

```
function moveObject(spaceObject)
    --move an object according to its velX and velY
    spaceObject.posX = spaceObject.posX + spaceObject.velX * dt
    spaceObject.posY = spaceObject.posY + spaceObject.velY * dt
end
```

So this looks pretty good but there is one small problem.

We have to use **dt**, the time that has elapsed since the last call to love.update(), to make the spaceship move at a constant speed. Since functions can't normally see things on the outside, we have to pass the value of dt into the function.

```
moveObject(spaceship, dt)
```

```
function moveObject(spaceObject, dt)
    --move an object according to its velX and velY
    spaceObject.posX = spaceObject.posX + spaceObject.velX * dt
    spaceObject.posY = spaceObject.posY + spaceObject.velY * dt
end
```

On the inside of the function we can use a different name, maybe elapsedTime, but to keep things simple I'll keep the name dt. Whereas spaceObject could be a spaceship or a satellite or an asteroid, dt on the inside of the function will always be passed dt from the outside so having two names is just confusing.

Wrapping Up the Wrapping Code

While we're building our moveObject function we might as well include the code that wraps the objects from one side of the window to the other when our objects move out of sight.

To do this we need to know the size of the window; width and height. We also need to know how far the object should move beyond the edge of the window before we wrap it. I call this distance, marginWidth. For the spaceship we've chosen marginWidth to be 45.

Here's how we would call the moveObject function for our spaceship.

```
moveObject(spaceship, dt, 45, width, height)
```

Here's the code.

```
function moveObject(spaceObject,dt, borderWidth, width, height)
    --move an object according to its velX and velY
    --if the object reaches 'borderWidth' past the edge of the screen, wrap it

    --move the object
    spaceObject.posX = spaceObject.posX + spaceObject.velX * dt
    spaceObject.posY = spaceObject.posY + spaceObject.velY * dt

    if spaceObject.posX > width + borderWidth then
         --wrap the object from one side of the screen to the other
         spaceObject.posX = -borderWidth

    elseif spaceObject.posX < -borderWidth then
         --wrap the object from one side of the screen to the other
         spaceObject.posX = width + borderWidth
    end

    if spaceObject.posY > height + borderWidth then
         --wrap the object from one side of the screen to the other
         spaceObject.posY = -borderWidth

    elseif spaceObject.posY < -borderWidth then
         --wrap the object from one side of the screen to the other
         spaceObject.posY = height + borderWidth
    end
end
```

There's one final feature I'd like to add to our first function. Sometimes we'll have an object that once it leaves the window it should disappear. I'm thinking bullets.

The function should return true if the spaceObject wraps and false when it doesn't. This way we can choose to destroy bullets when they wrap, but leave our spaceship, satellite and asteroids alone. I've marked the code additions in orange.

```
function moveObject(spaceObject,dt, borderWidth, width, height)
    --move an object according to its velX and velY
    --if the object reaches 'borderWidth' past the edge of the screen, wrap it

    returnVal = false

    --move the object
    spaceObject.posX = spaceObject.posX + spaceObject.velX * dt
    spaceObject.posY = spaceObject.posY + spaceObject.velY * dt

    if spaceObject.posX > width + borderWidth then
         --wrap the object from one side of the screen to the other
         spaceObject.posX = -borderWidth
         returnVal = true

    elseif spaceObject.posX < -borderWidth then
         --wrap the object from one side of the screen to the other
```

```
        spaceObject.posX = width + borderWidth
        returnVal = true
    end

    if spaceObject.posY > height + borderWidth then
        --wrap the object from one side of the screen to the other
        spaceObject.posY = -borderWidth
        returnVal = true
    elseif spaceObject.posY < -borderWidth then
        --wrap the object from one side of the screen to the other
        spaceObject.posY = height + borderWidth
        returnVal = true
    end

    return returnVal
end
```

You can see that the variable returnVal is given the default value of false. Each time we reach a point where we have to wrap the object we change the variable returnVal to true. At the end of the function we have

```
return   returnVal
```

This sends the value of returnVal back to where we called the function.

```
deleteBullet = moveObject(bullet, dt, 45, width, height)
```

If the bullet has not yet wrapped then deleteBullet will be false. If it has wrapped then deleteBullet will be true. So we could write something like …

```
deleteBullet = moveObject(bullet, dt, 45, width, height)
if  deleteBullet  then
    some instructions to delete the bullet
end
```

Spaceship Control

We can also move the spaceship control code to a function. This isn't as useful as the moveObject() function because only one object, our spaceship, will ever make use of it BUT it will unclutter the love.update() function. This is helpful because before we're done we'll add a bit more code there.

In love.update() we'll call

controlSpaceship(dt)

The function is below. Basically it's all the remaining code from love.update().

```
function controlSpaceship(dt)
   --turn the ship CCW
   if love.keyboard.isDown("a") then
      spaceship.direction = spaceship.direction - spaceship.turnSpeed * dt
   end

   --turn the ship CW
   if love.keyboard.isDown("d") then
      spaceship.direction = spaceship.direction + spaceship.turnSpeed * dt
   end
   --boost the ship
   if love.keyboard.isDown("w") then
      --turn on the boosting sound if it was off
      if spaceship.sound:isStopped() then
         spaceship.sound:play()
      end

      --some trigonometry magic
      spaceship.velX = spaceship.velX + math.sin(spaceship.direction) * spaceship.acceleration * dt
      spaceship.velY = spaceship.velY + math.cos(spaceship.direction) * -spaceship.acceleration * dt

      --simple animation between two images
      if spaceship.useImage == "boost1" then
         spaceship.useImage = "boost2"
      else
         spaceship.useImage = "boost1"
      end
   else
      if not spaceship.sound:isStopped() then
         --turn off the boosting sound if it was on (actually not off)
         spaceship.sound:stop()
      end
      spaceship.useImage = "coasting"
   end
end
```

Now love.update contains only two function calls.

```
function love.update(dt)
    --update the spaceships position during each loop
    moveObject(spaceship,dt,45,width,height)
    controlSpaceship(dt)
end
```

Adding a Friendly Satellite

To add a satellite we do most of the things we did for the spaceship. The included main.lua file doesn't yet include these further changes so you'll need to add them.

Create an empty table
Define posX and posY
Define velX and velY
Load a picture
Load a sound (if any)

Before love.load() we'll add

satellite = {}

to create the empty table. In love.load() we'll add

```
--create a satellite that drifts by
satellite.image = love.graphics.newImage("satellite.gif")
satellite.posX = -500
satellite.posY = 15
satellite.velX = 25
satellite.velY = 15
satellite.sound = love.audio.newSource("beep.mp3","static")
```

I’ve chosen a posX that’s off the screen so it will drift into view. The satellite will make a beeping sound.

With the sound we’ll set a few extra commands. Don’t worry too much about these for now.

```
satellite.sound:setVolume(1)
satellite.sound:setLooping(true)

love.audio.setPosition(width/2,height/2,0)
love.audio.setDistanceModel("exponent")
satellite.sound:setDistance(100,500)

satellite.sound:play()
```

The setLooping means that the sound will play again after it’s done. The satellite will go beep, beep, beep, etc.

The three lines setPosition, setDistanceModel and setDistance are tricky. These make the satellite beeping get quieter as it moves away from a reference point which in this case is the centre of the screen.

Finally the last command starts the beeping.

Moving the Satellite

Now that we’ve created the satellite moving it is trivial.

In love.update we add

```
moveObject(satellite,dt,500,width,height)
```

and for the sound to fade in and out we add

```
--tell sound engine where satellite is so beep can change volume
satellite.sound:setPosition(satellite.posX, satellite.posY, 0)
```

Final Code

The final code is attached as final.main.lua in case you need to compare. If you want to run the final code, you’ll need to rename your main.lua to something else like test.main.lua.

Then rename finale.main.lua to simply main.lua.

Next time we’ll add asteroids.