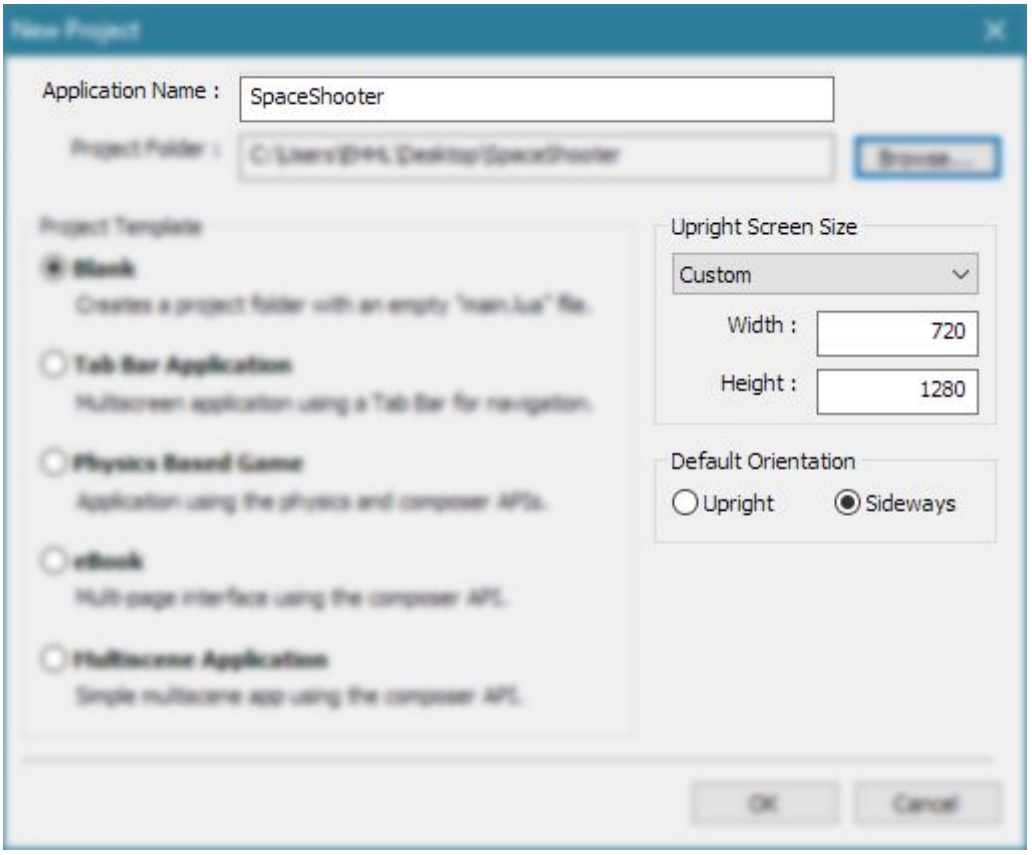
Documentação do corona:

https://docs.coronalabs.com/api/index.html

Iniciando o projeto

Usamos o Corona Simulator para criar um novo projeto. Clique em “New Project”:



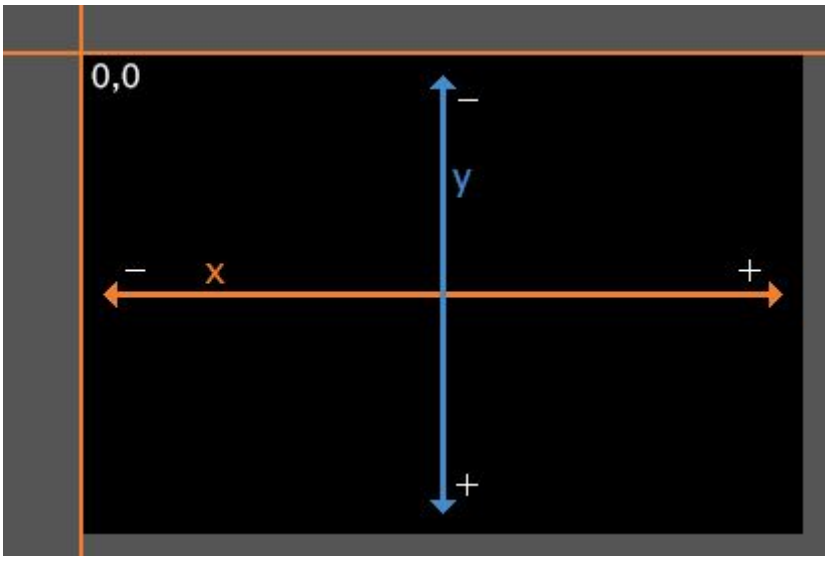
Nesta janela informe o nome do jogo, a resolução e a orientação como indicados acima. O resto pode ficar como está. Criado o projeto, o Corona vai abrir a pasta onde o jogo foi colocado, o Emulador onde poderemos executar o jogo, e o main.lua, que conterá o código do jogo. Agora é uma boa hora para copiar os recursos dados para a pasta criada para o jogo.

Carregando texturas

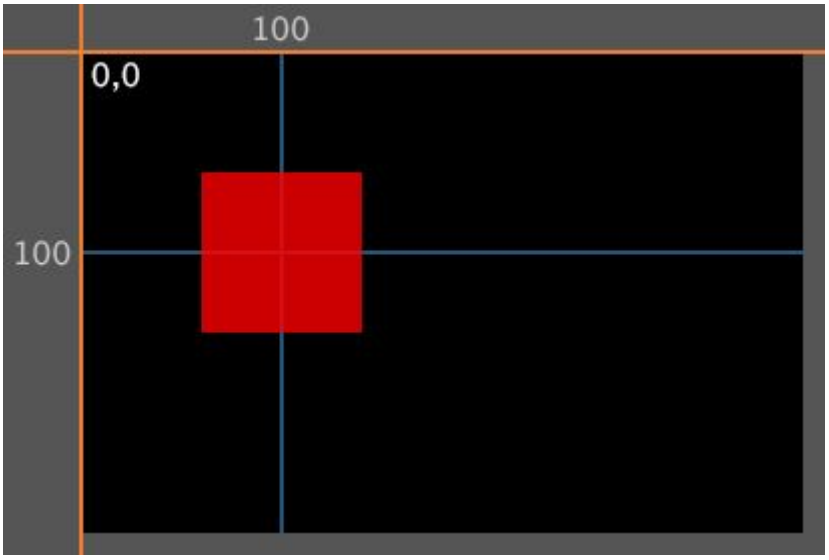
Usamos a função display.newImage para inserir uma textura no jogo.

```
player = display.newImage("textures/player.png", 100, 100)
```

Os argumentos passados são o caminho até a textura, e a posição x, y. Ela retorna um DisplayObject, que podemos usar após chamar a função para mudar suas propriedades, como sua posição, por exemplo. O sistema de coordenadas do Corona tem origem no canto superior esquerdo da janela, como na imagem:



As coordenadas passadas no display.newImage vão indicar onde ficará o centro da textura. Se passamos [100, 100], a textura será colocada assim:



Se voltarmos ao simulador, podemos ver a textura criada. Podemos usar a mesma função para adicionar o fundo:

```
display.newImage("textures/fundo.png", display.contentCenterX,
display.contentCenterY)
```

Dessa vez, como queremos que o fundo fique centralizado, usamos display.contentCenterX e display.contentCenterY, que nos indicam a posição x e y do centro da janela, respectivamente.

A ordem que essas funções são chamadas é importante. A última chamada fica por cima de todas as outras. No nosso caso, queremos que a nave fique na frente do fundo, então criamos o fundo e depois e nave.

Movimentação do jogador

Para fazer a nave se movimentar, vamos usar a biblioteca de física do Corona. Coloque a seguinte linha no início do arquivo para usar a biblioteca physics:

```
physics = require("physics")
```

A função require serve para carregar outros arquivos lua. Com ela é possível separar o jogo em vários arquivos para melhorar a organização, o que é muito importante pois jogos costumam fica com milhares de linhas de código.

Após importar a biblioteca, precisamos inicializá-la:

```
-- Inicializa a biblioteca de física
physics.start()
-- Tira a força de gravidade, já que estamos no espaço
physics.setGravity(0, 0)
```

Agora que já inicializamos a biblioteca physics, vamos tornar o jogador um objeto físico para que ele consiga interagir com o resto do jogo. Para fazer isso usamos a função physics.addBody, passando o *player* que criamos anteriormente:

```
physics.addBody(player)
```

Queremos agora movimentar o jogador com as teclas WASD. Para isso precisamos saber quando uma tecla é pressionada, usando o evento key. Para registrar um listener para o evento, usamos a função `addEventListener`, passando qual o evento e a função que vai ser chamada quando o evento acontecer. Precisamos então antes criar uma função para receber o evento. Essa função deve verificar se a tecla pressionada(`event.keyName`) é w, a, s ou d e mover o jogador de acordo, usando a função applyForce, que recebe a direção x e y para onde queremos que o objeto seja movido, e a origem dessa força, que no nosso caso será o centro da nave, ou seja, a posição x e y do `player`.

```
function keyEvent(event)
	if(event.keyName == "w") then
		player:applyForce(0, -1, player.x, player.y)
	end
	if(event.keyName == "a") then
		player:applyForce(-1, 0, player.x, player.y)
	end
	if(event.keyName == "s") then
		player:applyForce(0, 1, player.x, player.y)
	end
	if(event.keyName == "d") then
		player:applyForce(1, 0, player.x, player.y)
	end
end
-- Faz a função keyEvent ser chamada para o evento key
Runtime:addEventListener("key", keyEvent)
```

Você pode verificar no Corona Simulator que agora você já consegue controlar a nave. Porém, é preciso ficar apertando a tecla para aumentar a velocidade, e não queremos isso. Isso acontece porque o evento só acontece quando apertamos ou soltamos a tecla, mas não quando a mantemos segurada. Queremos então que a força comece a ser aplicada quando apertamos a tecla, e pare de ser aplicada quando soltamos. Para conseguir fazer isso precisamos de outro evento, o enterFrame, que é chamado a todo frame.

Vamos fazer uma nova função, `gameLoop`, que vai ter a função se aplicar a força no player, e modificar a função `keyEvent` para que ela mantenha o controle de quais teclas estão sendo pressionadas a qualquer momento em uma tabela `teclas`. Ela consegue fazer isso usando a propriedade `phase` do evento, que é `"down"` quando a tecla é pressionada e `"up"` quando a tecla é solta. Outra adição que podemos fazer é a variável `forca`, para podermos mudar a velocidade com que a nave acelera.

```
forca = 4

teclas = {}


function keyEvent(event)
	if(event.phase == "down") then
		teclas[event.keyName] = true
	else
		teclas[event.keyName] = false
	end
end
```

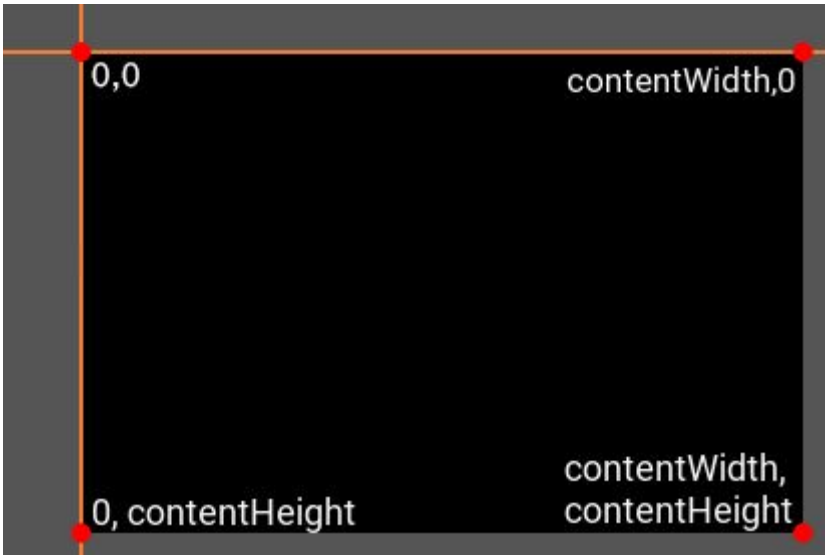
```
function gameLoop(event)
	if(teclas.w) then
		player:applyForce(0, -forca, player.x, player.y)
	end
	if(teclas.a) then
		player:applyForce(-forca, 0, player.x, player.y)
	end
	if(teclas.s) then
		player:applyForce(0, forca, player.x, player.y)
	end
	if(teclas.d) then
		player:applyForce(forca, 0, player.x, player.y)
	end
end

Runtime:addEventListener("key", keyEvent)
Runtime:addEventListener("enterFrame", gameLoop)
```

Agora conseguimos controlar melhor a nave. Porém, ela não para sozinha, e continua infinitamente seguindo em uma direção. Para evitar isso, usamos a propriedade `linearDamping` do player para que ele pare sozinho quando não há mais força sendo aplicada sobre ele.

```
physics.addBody(player)
player.linearDamping = 4
```

Aqui coloco 4, mas você pode experimentar com esse valor e decidir qual fica melhor. A mesma coisa com a força. Basicamente, quanto mais a força maior será a aceleração e velocidade máxima, e quanto maior o damping mais rápido será a desaceleração e menor será a velocidade máxima.

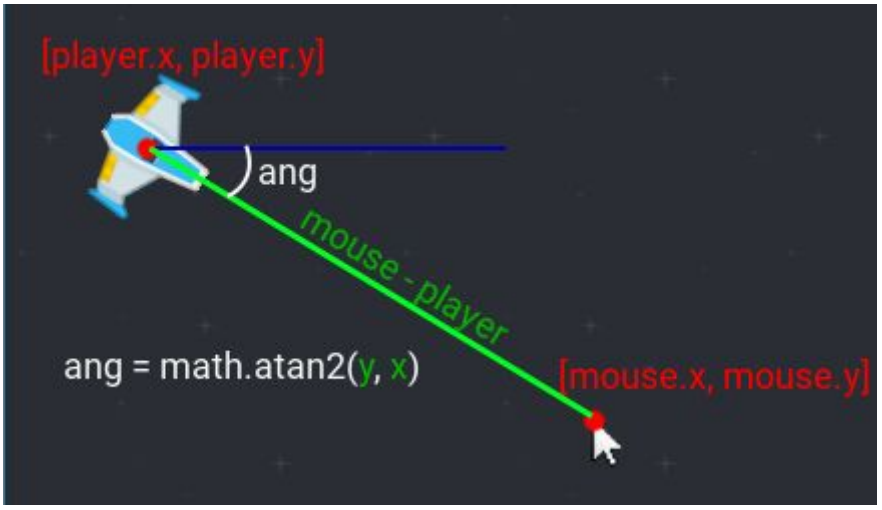
Outro problema no nosso jogo agora é que o jogador pode sair da tela. Para evitar que isso aconteça, podemos criar uma caixa ao redor da tela para que a nave fique presa dentro. Para isso vamos usar um *DisplayObject* invisível chamado `borda`, e usar o tipo de objeto físico `chain`, que recebe um conjunto de vértices e cria linhas sólidas entre eles. Se como vértices passarmos os quatro cantos da tela, como na imagem, faremos com que o jogador não consiga sair da tela.




```
local borda = display.newRect(0, 0, 0, 0)

physics.addBody(borda, "static", {
      chain = {0, 0,
               0, display.contentHeight,
               display.contentWidth, display.contentHeight,
               display.contentWidth, 0},
      connectFirstAndLastChainVertex = true
})
```

Atirando lasers

Vamos fazer com seja possível mirar e atirar com o mouse. O primeiro passo é fazer a nave do jogador sempre estar virada em direção ao mouse. Podemos mudar o rotação da nave mudando a propriedade `player.rotation`, que é a rotação da textura em graus. Mas como saber quantos graus a nave precisa estar virada para apontar para o mouse? Isso requer um pouco de matemática. Se subtrairmos a posição do mouse pela posição do player, obtemos um vetor que aponta do player em direção ao mouse. Aí podemos usar a função `math.atan2()`, que facilita nossa vida, pois transforma um vetor em um valor em radianos. Porém a rotação da nave é em graus, então é necessário transformar radianos em graus com `math.deg()`.



Como queremos que a nave sempre esteja apontando para o mouse, precisamos atualizar a rotação dela a todo frame, então podemos colocar o código na função `gameLoop` que fizemos anteriormente.

```
local playerRot = math.atan2(mouse.y-player.y, mouse.x-player.x)

player.rotation = math.deg(playerRot)
```

Queremos que a nave atire quando clicarmos com o mouse, e continue atirando se segurarmos. Podemos saber quando o jogador clica no evento `mouse`, porém assim como nas teclas, o evento só é chamado quando o botão do mouse é pressionado ou solto. Então vamos fazer a mesma coisa que antes, usar o evento do mouse para controlar quando o botão do mouse está sendo apertado, usando a mesma tabela `teclas`, verificar se o botão foi pressionado no `gameLoop`, e se sim, chamar uma função `atira()`, onde vamos escrever o código que faz a nave atirar.

```
function atira()
	-- A fazer
end

function mouseEvent(event)
	mouse.x = event.x
	mouse.y = event.y
	if (event.isPrimaryButtonDown and event.type == "down") then
		teclas.mouse = true
	else
		teclas.mouse = false
	end
end
```

```
function gameLoop(event)
	[...] -- outros códigos
	if(teclas.mouse)then
		atira()
	end
end
```

A função atira precisa primeiro criar o laser na mesma posição e rotação do jogador, e aplicar física nele. Nesse caso, não queremos que o laser colida com outros objetos, pois trataremos sua colisão manualmente mais tarde, então passamos à função addBody o parâmetro `isSensor = true`.

```
function atira()
	local laser = display.newImage("textures/laserBlue.png", player.x, player.y)
	laser.rotation = player.rotation
	physics.addBody(laser, {
		isSensor = true
	})
end
```

Agora precisamos também fazer o laser ir pra frente. Para isso vamos usar o setLinearVelocity no laser, mas essa função precisa de um vetor de direção, e o que temos é a rotação em graus do laser. Então estamos com um problema ao contrário do anterior, temos um ângulo e precisamos de um vetor. Para fazer essa transformação, podemos pegar o cosseno e seno do ângulo, que serão x e y do vetor de direção, respectivamente. O Lua tem as funções `math.cos()` e `math.sin()`, porém ambas trabalham com ângulos em radianos, então precisamos transformar os graus da rotação do laser em radianos primeiro. Podemos usar essa direção que calculamos também para fazer com que o laser apareça na frente na nave ao invés do centro dela.

```
	local dir = {
		x = math.cos(math.rad(laser.rotation)),
		y = math.sin(math.rad(laser.rotation))
	}
	-- Faz com que o laser apareça 70 pixels depois da frente da nave
	laser.x = laser.x + dir.x*70
	laser.y = laser.y + dir.y*70
	-- 1000 aqui indica a velocidade do laser, em pixels por segundo
	laser:setLinearVelocity(dir.x*1000, dir.y*1000)
```

Do jeito que está, o laser sai da tela e fica por isso. Você não vê mais ele. Mas ele continua muito além da janela, desfrutando da liberdade na sua jornada ao infinito, e gastando nossos preciosos recursos, como tempo de processamento e memória. Precisamos cortar suas asas, pois se continuar assim, depois de alguns minutos existirão milhares de lasers perdidos que podem nos causar problemas de performance.

Podemos resolver esse problema facilmente com um timer, uma função que é executada depois de um tempo depois de sua criação, por um determinado número de vezes ou indefinidamente. Para criar um timer usamos timer.performWithDelay, que recebe o intervalo em milissegundos para a função ser chamada, a função a ser chamada, e a quantidade de vezes que a função vai ser chamada. E para se desfazer do laser, usamos display.remove, que simplesmente recebe um DisplayObject e o destrói.

```
	-- Daqui a 4 segundos, remove o laser
	timer.performWithDelay(4000, function()
		display.remove(laser)
	end, 1)
```

Também precisamos limitar a velocidade com que o jogador pode atirar. Podemos fazer isso no `gameLoop`. Usando o `event.time`, que é o número de milissegundos passados desde o inicio do jogo, podemos guardar o tempo em que aconteceu o último tiro. Quando formos atirar novamente, verificamos se já passou tempo o suficiente desde o ultimo tiro, e só atiramos se passou.

```
-- Jogador só pode atirar uma vez a cada 200 milisegundos
freqTiro = 200
-- Inicializa o ultimoTiro, para que possamos atirar na primeira vez
ultimoTiro = 0

function gameLoop(event)
	[...]
	if(teclas.mouse and event.time - ultimoTiro > freqTiro)then
		ultimoTiro = event.time
		atira()
	end
end
```

Criando inimigos

Já conseguimos atirar, agora precisamos de alguma cosa para atirar em. Vamos fazer a função `criaInimigo(x, y)` para criar um inimigo na posição `x, y`. O básico é igual a como criamos o jogador no começo:

```
function criaInimigo(x, y)
	local inimigo = display.newImage("textures/enemy1.png", x, y)
	physics.addBody(inimigo)
	-- Nos indica que este displayObject é um inimigo; vai ser útil no
próximo passo
	inimigo.ehInimigo = true
end
```

```
-- Cria um inimigo na posição 300,200
criaInimigo(300,200)
```

E pronto, nosso inimigo já aparece no jogo. Mas ele não faz nada, nem é atingido pelos lasers. Vamos mudar isso. Na função `atirar()`, vamos registrar para o evento collision do laser, que acontece toda vez que o laser colide com outro objeto. Na função chamada temos acesso aos dois objetos que colidiram, `target`(o laser), e `other` (o objeto com qual o laser colidiu). Por enquanto, vamos apenas remover os dois objetos se o laser atingir um inimigo. Aqui usamos a propriedade `ehInimigo` que fizemos anteriormente, para saber se objeto que o laser colidiu é inimigo ou não.

```
laser:addEventListener("collision", function(event)
	display.remove(event.target)
	if(event.other.ehInimigo)then
		display.remove(event.other)
	end
end)
```

E assim conseguimos destruir o inimigo. Mas não tem graça se ele fica ali sem fazer nada. Vamos fazer ele reagir, ficar atirando em direção ao jogador. Assim como fizemos a nave do jogador sempre mirar no mouse, podemos fazer o inimigo sempre mirar no jogador. Mas como poderemos ter vários inimigos ao mesmo tempo, precisamos ter um controle de quais inimigos existem a qualquer momento. Podemos usar uma tabela para isso, adicionando os inimigos nela na função `criaInimigo` e retirando-os no evento de colisão do laser. Como em lua os índices das tabelas podem ser qualquer coisa, até mesmo outras tabelas, vamos usar isso a nosso favor e usar o próprio DisplayObject(que na verdade é uma tabela(tirando tipos primitivos, tudo é tabela em lua)) como índice para facilitar na hora de remover o inimigo da tabela.

```
inimigos = {}
function criaInimigo(x, y)
	[...]
	inimigos[inimigo] = true
end
function atira()
	[...]
	laser:addEventListener("collision", function(event)
		if(event.other.ehInimigo)then
			[...]
			inimigos[event.other] = nil
		end
	end
end
```

Feito isso, só precisamos percorrer a tabela `inimigos` no `gameLoop` e fazer todos apontarem para o jogador assim como fizemos para o jogador apontar para o mouse:

```
function gameLoop(event)
      [...]
      for inimigo in pairs(inimigos) do
            inimigo.rotation = math.deg(math.atan2(player.y-inimigo.y,
player.x-inimigo.x))
      end
end
```

Para fazer o inimigo se movimentar podemos usar o mesmo loop. Para cada inimigo, a todo instante, vamos modificar sua velocidade para que ele se mova em direção ao jogador, até que ele fique a uma distância de 250 pixels, e então pare.

Primeiro subtraímos a posição do jogador pela posição do inimigo, e acabamos com um vetor de diferença que indica o quanto o inimigo tem que se mover para chegar no jogador. Não queremos que ele faça isso instantâneamente, mas sim suavemente, fazendo com que ele seja rápido quando está longe e vá diminuindo sua velocidade quando chega perto. Então vamos multiplicar esse vetor de diferença por um modificador que é proporcional à distância entre as naves. A distância pode ser encontrada com o teorema de pitágoras(menos 250 pois queremos que o inimigo mantenha uma distância do jogador) e o modificador é a distância dividida por algum número que vai indicar o quão rápido vai ser o inimigo. Aqui usei 200, mas você pode brincar com esse valor.

```
local dist = math.sqrt((player.x-inimigo.x)^2 + (player.y-inimigo.y)^2)
local mod = (dist - 250) / 500
inimigo:setLinearVelocity((player.x-inimigo.x) * mod,
(player.y-inimigo.y) * mod)
```

Já está se movendo e mirando, agora falta atirar. O tiro do inimigo é muito parecido com o tiro do jogador, só vai mudar a nave de onde ele está saindo e a textura. Vamos generalizar a função atirar para que possamos utilizá-la também com os inimigos, passando a nave de origem e a textura como parâmetros. Só precisamos modificar as primeiras linhas da função:

```
function atira(nave, textura)
      local laser = display.newImage(textura, nave.x, nave.y)
      laser.rotation = nave.rotation
      [...]
end
```

E também atualizar a chamada dessa função, que acontece no gameLoop:

```
      atira(player, "textures/laserBlue.png")
```

Para fazer o inimigo atirar, podemos utilizar um timer que roda indefinidamente, atirando toda vez. Vamos inicializá-lo na função `criaInimigo`:

```
function criaInimigo(x, y)
      [...]
      inimigo.timer = timer.performWithDelay( 800, function()
            atira(inimigo, "textures/laserRed.png")
      end, 0)
end
```

E também precisamos cancelar o timer quando o inimigo é destruído, usando a função [timer.cancel] no evento de colisão do laser:

```
laser:addEventListener("collision", function(event)
      if(event.other.ehInimigo)then
[...]
timer.cancel(event.other.timer)
      end
end)
```

Tratando colisões

Vamos automatizar a criação de inimigos, para que eles apareçam de todos os lados da tela. Uma maneira simples de fazer isso é um timer que cria um inimigo fora da tela, em cima de um círculo. Assim, só precisamos gerar um ângulo aleatório e usar o seno e cosseno desse ângulo como coordenadas iniciais do novo inimigo fora da tela. Como o inimigo já segue o jogador, ele vai naturalmente entrar no jogo. Só há um porém: Cercamos o jogador com uma borda física para que ele não consiga sair da tela. E, como ninguém pode sair, ninguém pode entrar também. Então estamos com essa situação onde a borda precisa colidir com o jogador, mas não os inimigos. Isso pode ser resolvido com filtros de colisão.

Podemos separar os objetos presentes atualmente em nosso jogo em 5 categorias: A borda, o jogador, os inimigos, os tiros do jogador e os tiros dos inimigos. Para indicar para a biblioteca de física o que colide com o que, precisamos aplicar filtros nos objetos, o que indica que de que categoria ele é. Cada categoria é representada por uma potência de 2(1, 2, 4, 8, 16…), e a sua máscara(com o que ele colide) é a soma de todos as outras categorias com as quais ele colide. Para entender melhor, veja a tabela:

Filtros	1	2	4	8	16	Máscara
(1) Borda		X				2
(2) Jogador	X		X		X	21
(4) Inimigo		X	X	X		14
(8) LaserJogador			X			4
(16) LaserInimigo		X				2

O jogador, por exemplo, colide com a borda(1), os inimigos(4) e os lasers dos inimigos(16). Para chegar na sua máscara, somamos estes números para chegar em 21. Isso funciona porque na verdade estamos construindo o número em binário(10101 = 21). Vamos declarar os filtros assim:

```
local filtroBorda = { categoryBits=1, maskBits=2 }
local filtroPlayer = { categoryBits=2, maskBits=21 }
local filtroInimigo = { categoryBits=4, maskBits=14 }
local filtroLaserPlayer = { categoryBits=8, maskBits=4 }
local filtroLaserInimigo = { categoryBits=16, maskBits=2 }
```

E para aplicar os filtros, precisamos modificar todos os `physics.addBody` do jogo, adicionando o parâmetro filter com o filtro apropriado:

```
physics.addBody(borda, "static", {
      [...]
      filter = filtroBorda
})
physics.addBody(player, {filter = filtroPlayer})
physics.addBody(inimigo, {filter = filtroInimigo})
```

Como a função atirar atirar lasers tanto do player quanto dos inimigos, precisamos antes verificar qual filtros devemos usar:

```
function atira(nave, textura)
      [...]
      local filtro
      if (nave.ehInimigo) then
            filtro = filtroLaserInimigo
      else
            filtro = filtroLaserPlayer
      end
      physics.addBody( laser, {
            isSensor = true,
            filter = filtro
      })
      [...]
end
```

Agora os inimigos não conseguem se matar e também não colidem com a borda, fazendo com que eles consigam chegar de fora da tela sem problemas. Com essa situação resolvida, voltamos ao problema de criar inimigos automaticamente. Vamos criar um timer que vai criar continuamente um inimigo a cada 1500 milissegundos. Ela cria um ângulo aleatório entre 0 e 2π, e cria o inimigo em uma elipse ao redor do centro da tela, grande o suficiente para estar totalmente fora da tela.

```
timer.performWithDelay( 1500, function()
	local angulo = math.random()*math.pi*2
	criaInimigo(math.cos(angulo)*1000 + display.contentCenterX,
math.sin(angulo)*500 + display.contentCenterY)
end, 0)
```

Para colocar vidas para o jogador, precisamos apenas adicionar uma propriedade vida ao `player`, e decrementar essa propriedade quando um laser o atinge.

```
player.vida = 8
laser:addEventListener("collision", function(event)
	if(event.other.ehInimigo)then
		[...]
	else --Se não é inimigo, é o player
		event.other.vida = event.other.vida - 1
	end
end)
```

Podemos também adicionar um indicador no canto da tela usando display.newText para informar quantas vida o jogador ainda tem:

```
display.newImage("textures/playerLife.png", 50,50)
player.vidaTexto = display.newText("X"..player.vida, 100,52)
```

Porém isso não é tudo. Quando passamos `"X"..player.vida` para o `newText`, estamos criando uma string concatenando `"X"` com `player.vida`, e sua relação com o `player.vida` acaba aí. Quando o valor de player.vida mudar, o texto não vai mudar automaticamente, então precisamos atualizar o texto quando tiramos vida do jogador:

```
event.other.vidaTexto.text = "X"..event.other.vida
```

Adicionando som

Adicionar áudio no jogo é bem simples: Primeiro carregamos o arquivo de áudio com a função audio.loadSound, e depois tocamos o audio com audio.play:

```
local somLaser = audio.loadSound("audio/laser4.mp3")

function atira(nave, textura)
	audio.play(somLaser)
```