04

Identificando os estados da aplicação

Transcrição

Voltando ao código fonte vamos pensar em como identificar os diferentes estados! O primeiro nós já conhecemos, o `SETUP`, que está bem marcado. O segredo é no `loop()` entender a como mapear esses estados!

Em cada situação distinta o jogo se comportará de uma maneira diferente e isso pode causar muita confusão, já que o código pode ficar bem complicado. Por enquanto, ele ainda está pequeno pois, não está executando muitas tarefas. O que vamos fazer, primeiro, é separar a função do `for` que está junto ao `loop()`, pois isso é algo que vai acontecer várias vezes. Por isso, vamos acrescentar uma nova função `tocaLedsRodada()` e nela colocaremos o `for`:

```
void tocaLedsRodada(){
    for(int indice = 0; indice < TAMANHO SEQUENCIA; indice++){
        piscaLed(sequenciaLuzes[indice]);
    }
}
```

Feito isso, vamos aproveitar para remover o comentário do `loop()`. O primeiro objetivo é deixar o código expressivo, assim, seria ótimo se conseguíssemos marcar no código fonte que estamos percorrendo os diferentes estados. Como queremos um conjunto de valores para expressar uma determinada situação podemos declarar, usando a linguagem C, uma enumeração, portanto, escolhemos o `enum Estados`. Nós utilizaremos isso para informar quais foram os valores identificados, no caso: `PRONTO_PARA_PROX_RODADA`, `USUARIO_RESPONDENDO`, `JOGO_FINALIZADO_SUCESSO` e `JOGO_FINALIZADO_FALHA`. Teremos o seguinte:

```
enum Estados{
  PRONTO_PARA_PROX_RODADA,
  USUARIO_RESPONDENDO, JOGO_FINALIZADO_SUCESSO,
  JOGO_FINALIZADO_FALHA
};
```

É importante notar que a linguagem C trabalha de uma maneira bastante peculiar. Na verdade já vimos esse comportamento antes, quando o C usou um apelido para lidar com os diferentes LEDS. Quando usamos o `enum` estamos aplicando uma sequência numérica para estes itens, por exemplo, o `PRONTO_PARA_PROX_RODADA` é o item número 0. Temos, portanto uma numeração que vai até 3. E, isso nos permite usar coisas como:

```
void loop() {
    int estado = PRONTO_PARA_PROX_RODADA;
}
```

Compilando isso vemos que funciona! Ou seja, o `PRONTO_PARA_PROX_RODADA` também é um apelido! Assim caso quiséssemos utilizar uma `função` para conhecer o `estadoAtual()` poderíamos utilizar o apelido dele e pedir para que o `PRONTO_PARA_PROX_RODADA` fosse retornado, assim, na função `loop()` poderíamos especificar que `int estado = estadoAtual`. E incluiremos o comentário `//Computar Estado Atual`, deixaremos isso para discutir mais tarde. Teremos o seguinte:

```
void loop() {
  int estado = estadoAtual();
}

int estadoAtual(){
  //Computar Estado Atual
  return PRONTO_PARA_PROX_RODADA;
}
```

É preciso compreender que a função `loop` estará rodando indefinidamente, então, dependendo do estado que o jogo estiver, vamos ter que mostrar algo ou ler algum botão ou dizer que um jogador ganhou ou perdeu. E como faremos isso sem ter um código que acabe ficando super confuso e complicado!?

Podemos, na função `loop()` acrescentar um `switch` e também um `case` . Assim, sinalizamos que caso tudo esteja pronto para a próxima rodada, os próximos passos podem ter seguimento, `case PRONTO_PARA_PROX_RODADA` e inserimos o comentário `//executa funções próxima rodada` . Podemos ainda, completar afirmando o que ele fará em cada situação, assim, `case USUARIO_RESPONDENDO` , `case JOGO_FINALIZADO_SUCESSO` e `casa JOGO_FINALIZADO_FALHA` . Teremos o seguinte:

```
void loop() {
  switch(estadoAtual()) {
  case PRONTO_PARA_PROX_RODADA:
    //executa funções próxima rodada
  case USUARIO_RESPONDENDO:
  case JOGO_FINALIZADO_SUCESSO:
  case JOGO_FINALIZADO_FALHA:
  }
}
```

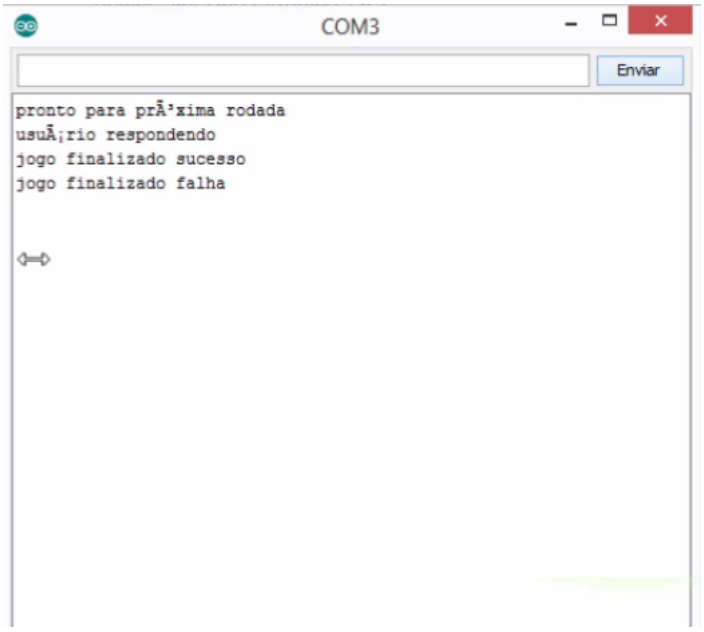
Tendo esse código o que acontece é que as tarefas serão executadas na ordem como estão escritas, de cima para baixo. Vamos ver funcionando? Escrevemos `Serial.println(" ")` acompanhado do respectivo componente:

```
void loop() {
  switch(estadoAtual()) {
  case PRONTO_PARA_PROX_RODADA:
    //executa funções próxima rodada
    Serial.println("pronto para próxima rodada");
  case USUARIO_RESPONDENDO:
    Serial.println("usuário respondendo");
  case JOGO_FINALIZADO_SUCESSO:
    Serial.println("jogo finalizado sucesso");
  case JOGO_FINALIZADO_FALHA:
    Serial.println("jogo finalizado falha");
  }
}
```

Compilamos isso e enviamos para o **Arduino**. Na verdade, estamos interessados em observar o **Console**, assim, vamos pedir para ele rodar, mas ele ficará travado em um mesmo lugar. Para fazer isso podemos utilizar a instrução `for` passando que ele deve ser nada, que checa nada e faz nada. Nós escrevemos abaixo do `loop()` :

```
for(;;);
```

Compilando isso e enviando teremos o seguinte:



Tirando um pequeno problema de acentuação, tudo foi rodado! Mas, isso não vai nos ajudar, pois queremos que ele rode apenas uma coisa toda vez que o estado for checado! É preciso que ele cheque o estado, descubra que está pronto e execute a função que prepara a próxima rodada, depois queremos que ele saiba que é a hora do usuário responder e só execute a função das funções e que ele bote o finalizado com sucesso! Note que o resultado que obtemos depois do `JOGO_FINALIZADO_SUCESSO` é `JOGO_FINALIZADO_FALHA` . Ou seja, é preciso arrumar a função `case` .

Portanto, vamos apagar o comentário que havia sido inserido e vamos acrescentar um `break` que indica que nós quebramos a execução. Observe:

```
void loop() {
  switch(estadoAtual()) {
  case PRONTO_PARA_PROX_RODADA:
    Serial.println("pronto para próxima rodada");
    break;
  case USUARIO_RESPONDENDO:
    Serial.println("usuário respondendo");
  case JOGO_FINALIZADO_SUCESSO:
    Serial.println("jogo finalizado sucesso");
  case JOGO_FINALIZADO_FALHA:
    Serial.println("jogo finalizado falha");
  }
}
```

Compilando e mandando isso teremos o seguinte resultado:

Ou seja, só uma instrução foi executada! Assim se não há o `break` as demais instruções são executadas. Portanto, vamos acrescentar o `break` aos demais pontos:

```
void loop() {
  switch(estadoAtual()) {
  case PRONTO_PARA_PROX_RODADA:
    Serial.println("pronto para próxima rodada");
    break;
  case USUARIO_RESPONDENDO:
    Serial.println("usuário respondendo");
    break;
  case JOGO_FINALIZADO_SUCESSO:
    Serial.println("jogo finalizado sucesso");
    break;
  case JOGO_FINALIZADO_FALHA:
    Serial.println("jogo finalizado falha");
    break;
  }
  for(;;);
}
```

E isso terá resolvido o problema!