INTRODUÇÃO A GML

Game Maker Language

Alex Ferreira Costa

Compilado de Tutoriais sobre a linguagem de programação do GameMaker: Studio

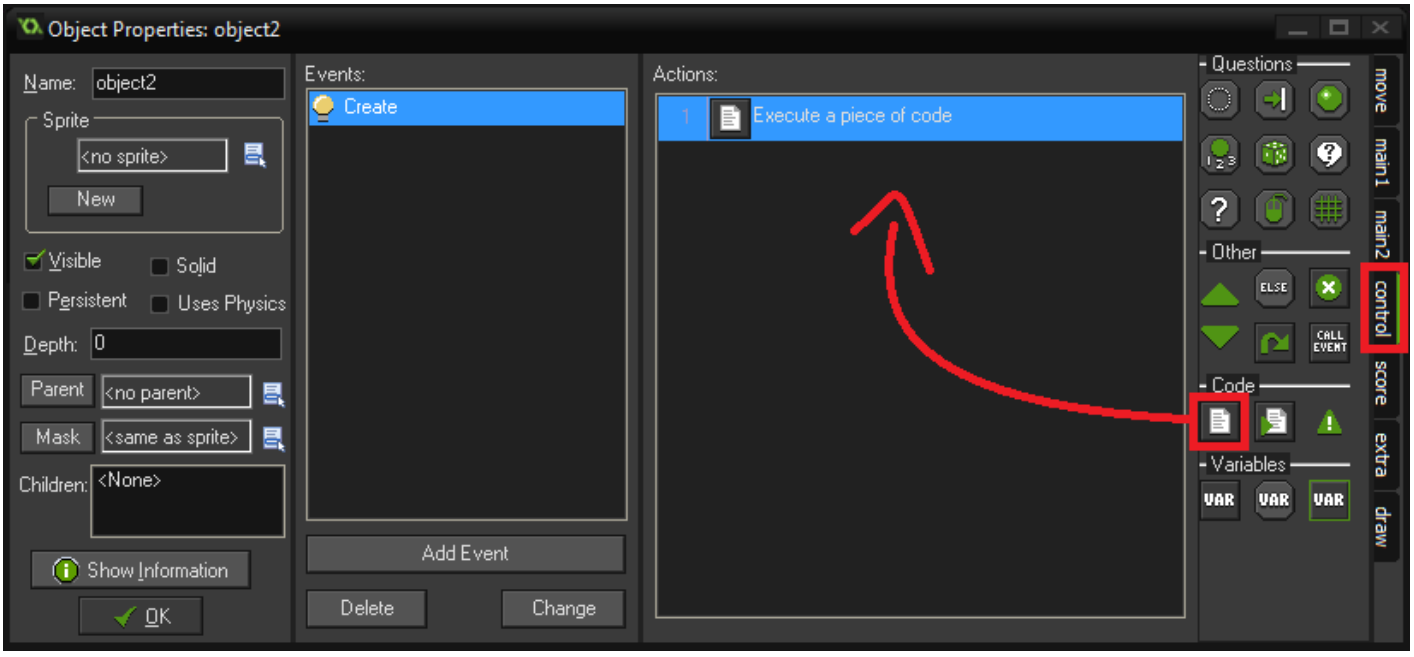
INTRODUÇÃO EM GML

Aula 01 - Editor de Script do GM:S

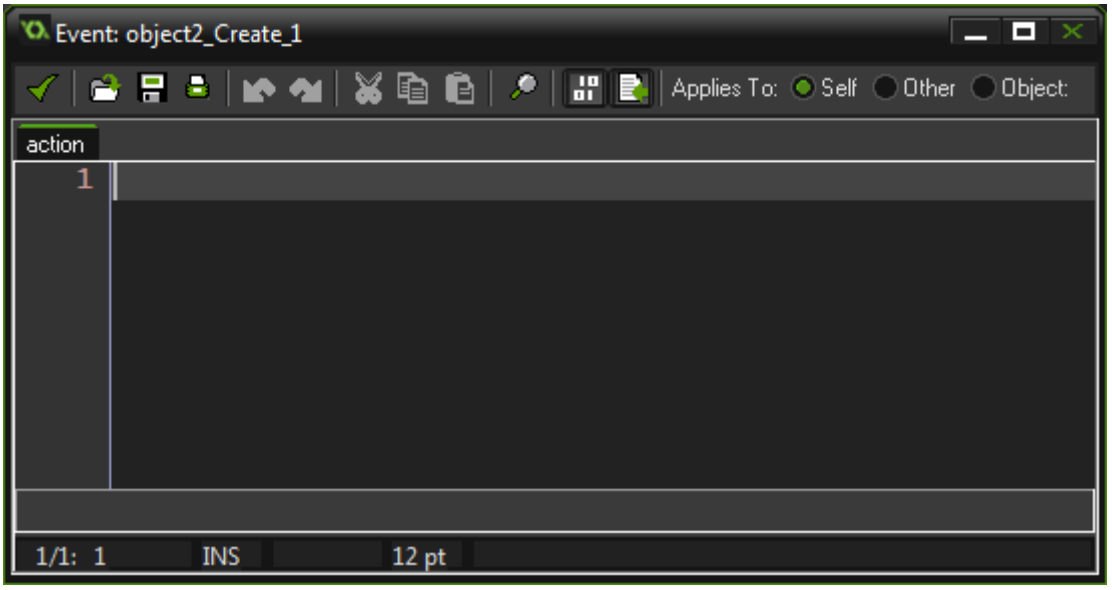
(Atualizado 08/02/2015)

Pessoal estarei postando aulas para iniciantes em GML. São aulas básicas, pra quem ainda não conseguiu dominar a GML. Então vamos à nossa 1ª aula:

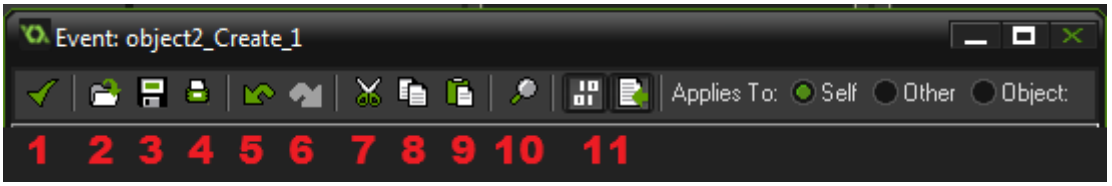
Para abrir o editor do **GM:S** abra seu objeto, adicione um evento (Create, Step, Alarm, etc) e vá à aba **control** e arraste o ícone **"Execute a piece of code"**. Como na imagem:



A seguinte janela será aberta:



Os seguintes botões executam as seguintes funções:



1- Ok: Clique nesta opção para salvar seus códigos e voltar ao objeto.

2- Load a code from a text file: Abri arquivos de texto (.txt) que contenham códigos.

3- Save the code to a text file: Salva o código aberto em um arquivo de texto (.txt).

4- Print the code: Clique nesta opção para imprimir o código.

5- Seta para trás: para desfazer uma ação no código.

6- Seta para frente: para refazer uma ação no código.

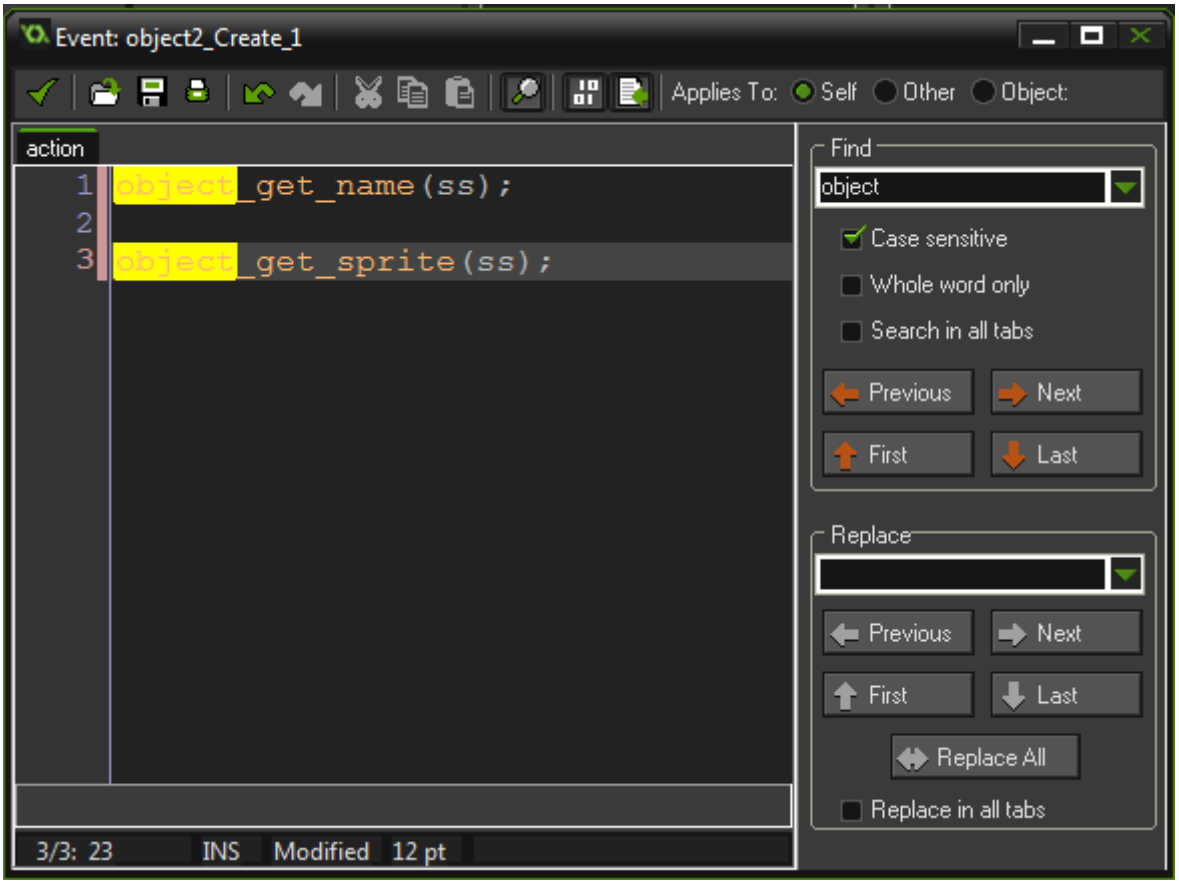
7- Cut: recortar.

8- Copy: copiar.

9- Paste: Colar.

10- Localizar: Localiza uma palavra.

Usado para encontrar e ou substituir alguma palavra ou texto por outro. Um painel será exibido à direita da janela de códigos:



Você pode digitar o texto a ser encontrado na caixa **Find**. Note que no código de todas as ocorrências do texto econtrado são imediatamente mostradas com um fundo amarelo. Digite o texto a ser substituido na caixa **Replace**.

Find - funções dos botões:

Next: Seleciona a próxima ocorrência;

Previous: Seleciona a ocorrência anterior;

First: Seleciona a primeira ocorrência;
Last: Seleciona a ultima ocorrência.

Replace - funções dos botões:

Next: Ssubstitui a próxima ocorrência;

Previous: Substitui a ocorrência anterior;

First: Substitui a primeira ocorrência;
Last: Substitui a ultima ocorrência.

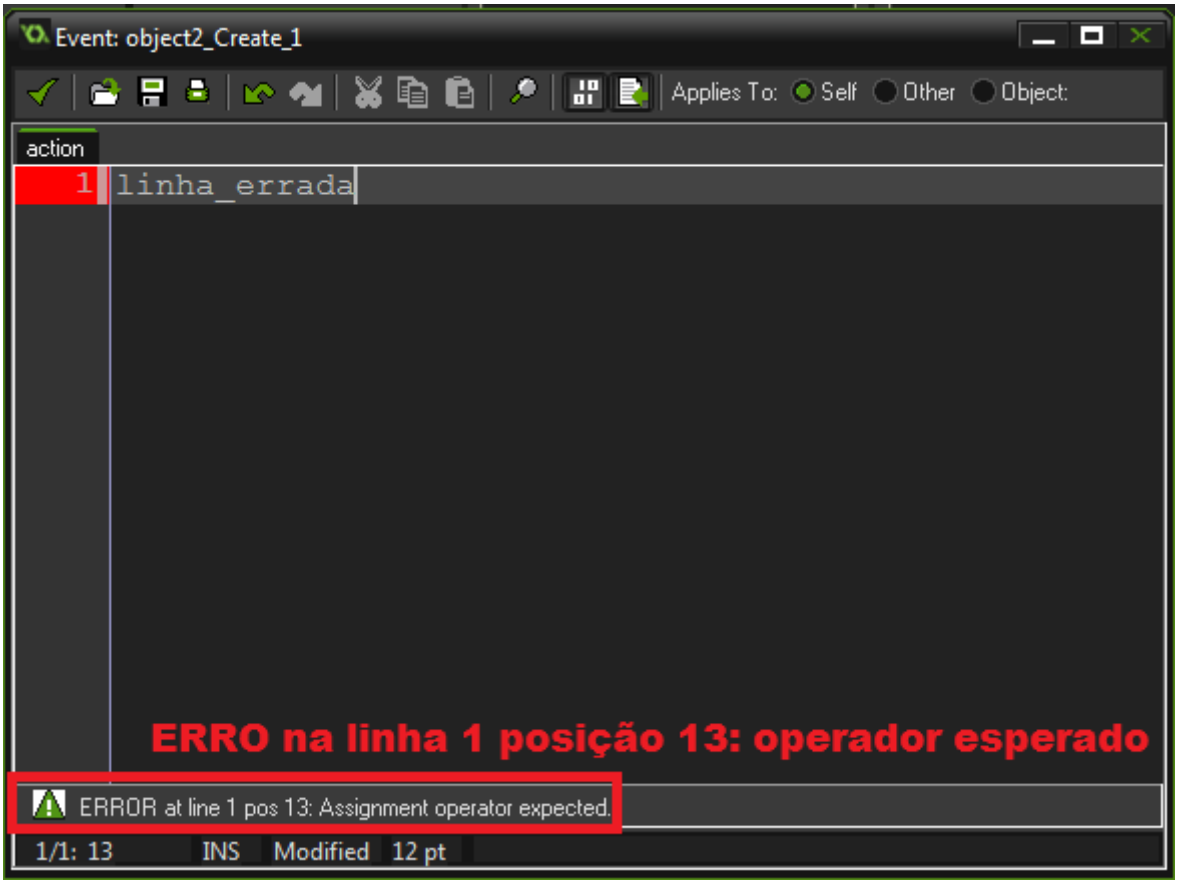
Replace All: Substitui todas as ocorrências.

11- À esquerda temos a ativação/desativação do verificador de erros e à direita temos ativação/desativação do auto-completador de códigos.

Apllies To (aplicar em):

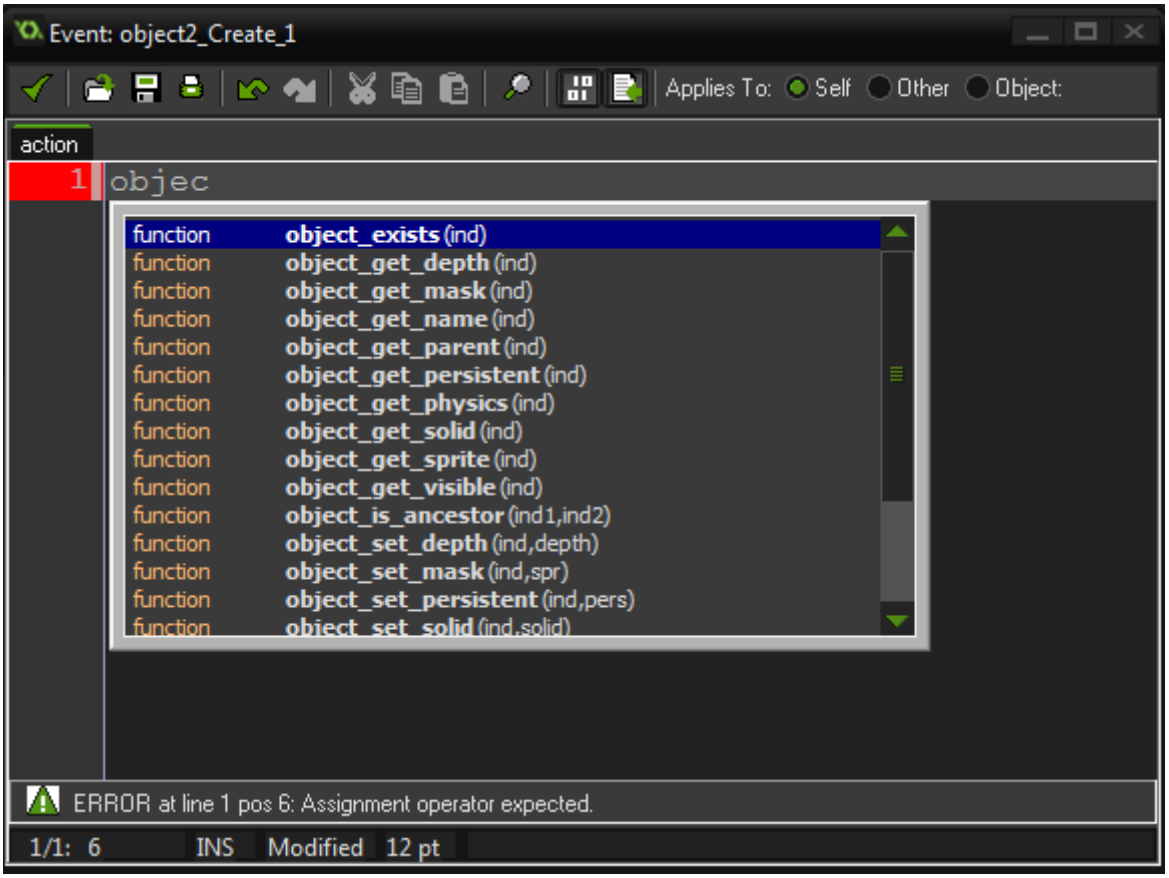
Self: Próprio objeto
Other: Objeto que está colidindo com este.
Object: Escolhe um objeto existente no jogo.

Exemplificação de erro:



Auto-completar de funções e variáveis:

Ao digitar parte de uma função ou variável nativa o GM lista todas as funções, váriaveis e nomes de resources (Sprites, backgrounds, sons, etc) em uma pequena janela:



Para mudar de opção basta usar a rodinha do mouse ou as setas direcionais cima e baixo e para selecionar a tecla **Enter** ou clique com o**botão esquerdo do mouse**.

INTRODUÇÃO EM GML

Aula 02 - Variáveis e Constantes

(Atualizado 08/02/2015)

Variáveis são utilizadas para armazenar vários tipos de valores, como números, textos e frases. Elas também são chamadas de identificadores, pois apontam um valor.

Um nome de variável pode conter letras (sem acentos gráficos e o Ç), números (Desde que não comece com um ex: 8_tiro) e alguns símbolos como o underline (_). As que armazenam números são chamadas de variáveis de valor **Real** e as que armazenam textos são chamadas de**Strings**.

Veja alguns exemplos:

- Atribui o valor **100** à variável **balas**:

> **Código:** **Selecionar todos**
>
> ```
> balas = 100
> ```

- Atribui o valor **55** à variável **BALAS**:

> **Código:** **Selecionar todos**
>
> ```
> BALAS = 55
> ```

- Atribui o valor **7.5** à variável **forca**:

> **Código:** **Selecionar todos**
>
> ```
> forca = 7.5
> ```

- Atribui o valor **3** à variável **_vidas_**:

> **Código:** **Selecionar todos**

```
_vida_ = 3
```

- Atribui o valor **"João Pedro José"** à variável **Nome3**:

Código: Selecionar todos

```
Nome3 = "João Pedro José"
```

- Atribui o valor **false** à variável **pode_atirar**:

Código: Selecionar todos

```
pode_atirar = false
```

Repare que:

- As quatro primeiras variáveis são de valor **Real**.
- A variável **forca** é de valor **Real fracionado**. Aqui usamos pontos (.) e não virgulas (,).
- As variáveis **balas** e **BALAS** são duas variáveis diferentes. Isso por causa do **case-sensitive**, ou seja mesmo que só uma das letras seja maiúscula já é uma nova variável.
- A penúltima variável é uma **String**.
- A última variável é de valor **booleano**.

Entenda melhor:

String: Textos, palavras e letras que devem vir entre apóstrofos, como "Eu venci! Então eu sou o ganhador".

Real inteiro: Números inteiros como 5, 670, 100000, e 31.

Real fracionado: Números fracionados 5.67, 100.55, e 0.5.

Booleano:true que é a mesma coisa que 1 e false que é o mesmo que 0. Tanto faz você usar 1 como usar true, ambos tem o mesmo valor.

Existem três contextos para variáveis: **global**, **local de instância** e **local de evento** que definem como e de onde as variáveis podem ser acessadas.

Variável local de instância:

O Game Maker usa objetos para criar a aplicação. E cada objeto tem suas atribuições padrão, como a posição na room (Cenário). Cada objeto e suas réplicas (instâncias) têm uma posição, ou seja as variáveis **x** e **y**. Vamos criar um exemplo:

- Crie um objeto e uma room.
- Adicione o evento **Create** (evento que acontece 1 vez, quando o objeto é criado) ao objeto.
- Neste evento insira o código:

> **Código: Selecionar todos**
>
> ```
> saude = irandom(100)
> ```

- Isso atribui o valor aleatório de 0 a **100** para a variável **saude**.

PS: a função **irandom**(**X**) nos devolve um número inteiro de **0** a **X**, no caso 100. Aprenderemos mais sobre funções mais a frente, não se preocupe em entender isso agora.

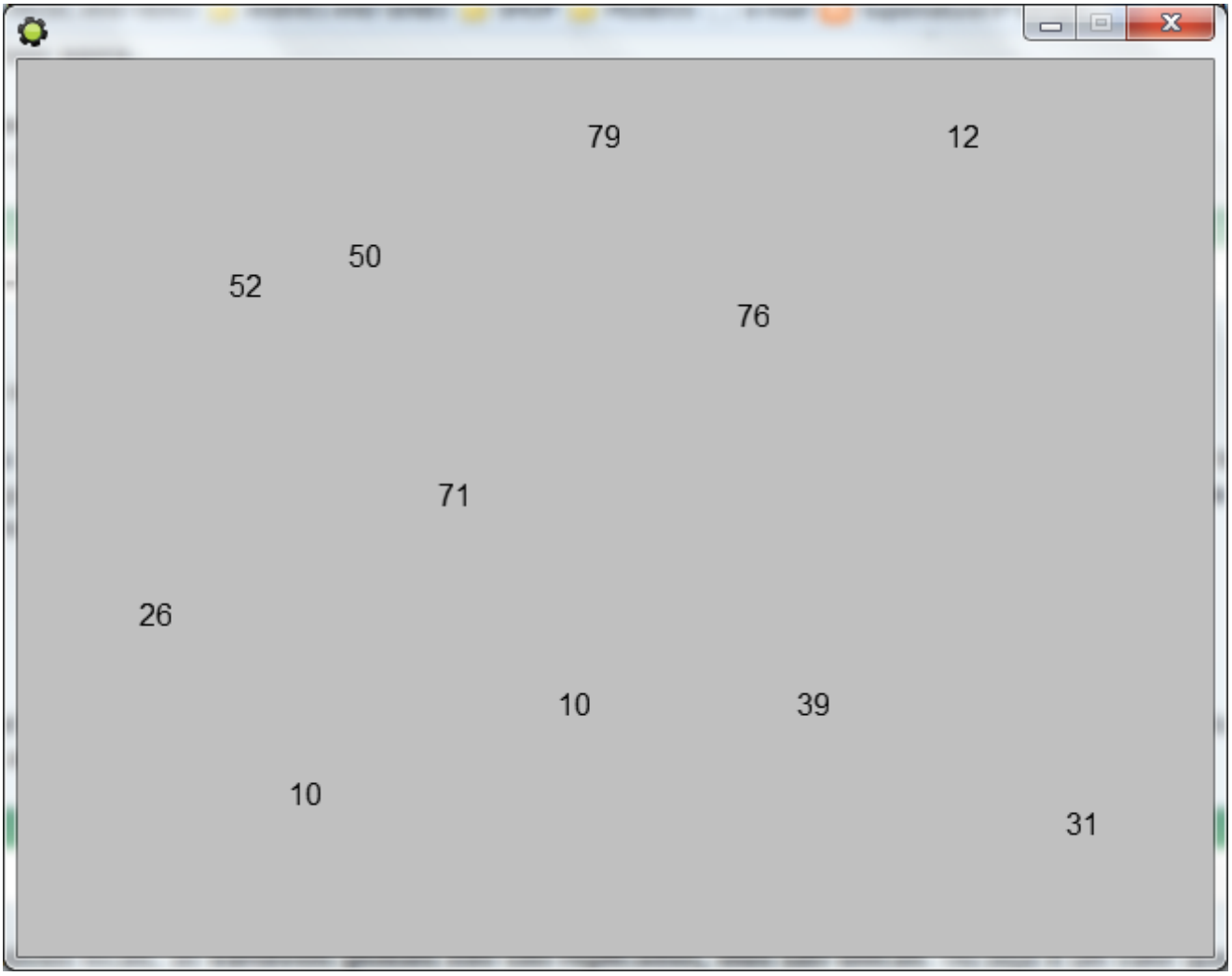
- Adicione o evento **Draw** (Evento de desenho, ocorre o tempo todo) ao objeto. Neste evento insira o código:

> **Código: Selecionar todos**
>
> ```
> draw_text(x, y, saude)
> ```

- Isso desenha o valor armazenado na variável **saude** na posição x,y do objeto.

- Adicione vários deste objeto na room. Depois execute (F5).

O resultado é:



Veja que cada instância do objeto tem uma **saude** diferente. É dessa maneira que funciona uma variável local de instância. Para ver todas as variáveis de instância padrão/nativa para os objetos, vá na barra de menu clique em **Scripts** depois selecione a opção **Show Built-in Variables**. Elas estão no fim da lista em azul com o inicio "local:".

Acesso à uma variável local a partir de outro objeto: Deixarei em um spoiler, pois pode ser que seja um pouco complexo aprender essa parte. Se quiser pular ela não há problema, depois volte pra dar uma olhada.

SPOILER: Clique para ver o conteúdo

Variável global:

Esse é o tipo de variável que pode ser utilizada por todos os objetos e em qualquer momento durante a execução da aplicação (desde que seja criada antes de modificada ou acessada).

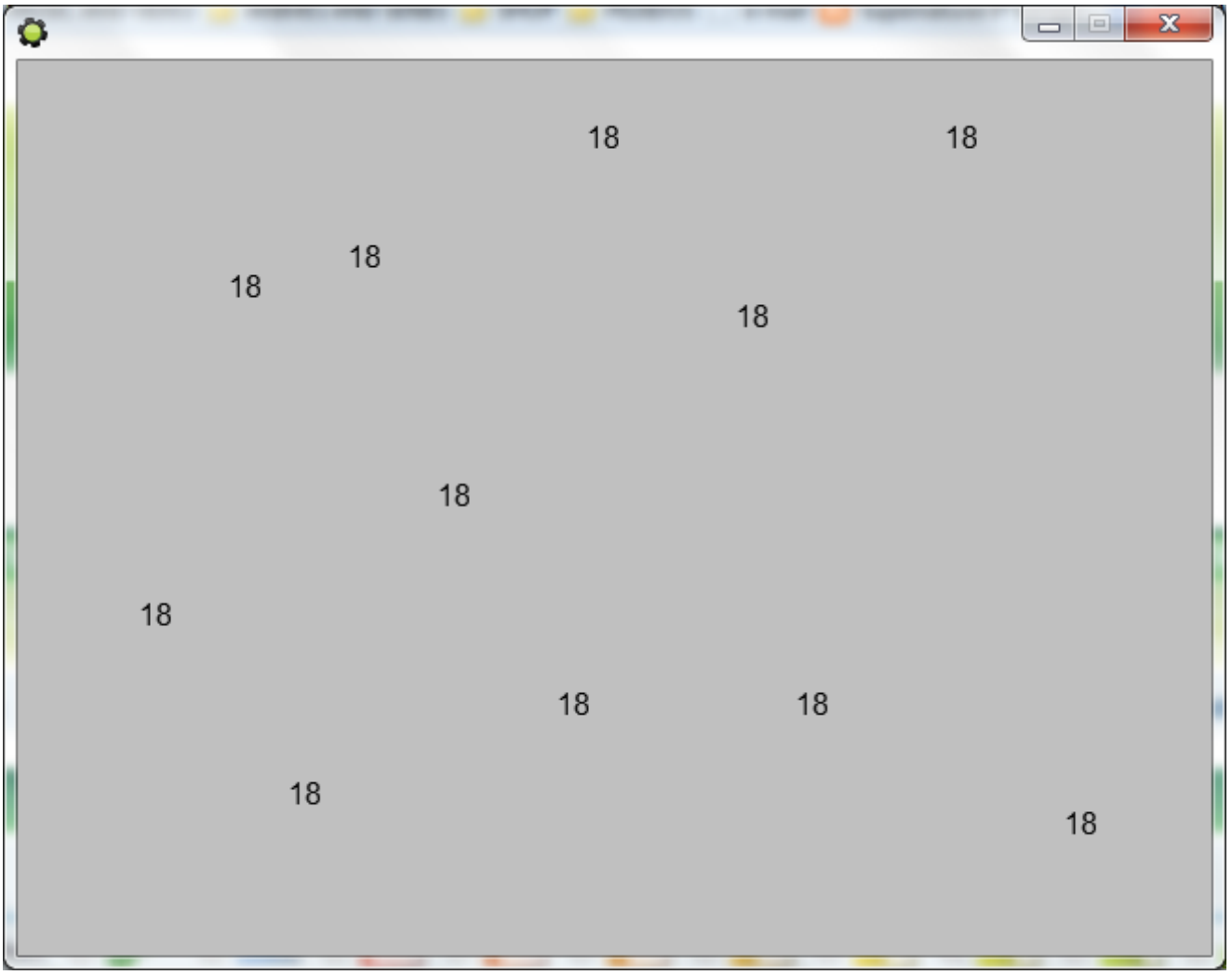
Código: Selecionar todos

```
global.saude = 100
```

Diferentemente das variáveis locais de instância, as **variáveis globais não são replicadas, elas são únicas**. Ou seja é um valor que pode ser alterado por qualquer evento ou objeto, mas é um só. Veja que colocamos **global.**antes do nome da variável. Isso que a transforma em uma variável global.

Faça o mesmo procedimento que fizemos com a variável local de instância trocando **saude** por **global.saude** (Nos dois códigos). Os resultados de todas as instancias serão os mesmos, pois estamos falando de uma mesma variável.

Resultado:



Se você criou 6 instâncias, a variável vai ser modificada 6 vezes, sendo que o valor que vai aparecer será o do último objeto adicionado a room.

Lista de algumas variáveis globais embutidas no Game Maker:

health: variável para representar a saúde do player
lives: variável para representar a quantidade de vidas
room: variável que define/representa a room atual
score: variável que representa a pontuação
mouse_x: variavel que representa a posição x do mouse
mouse_y: variavel que representa a posição y do mouse

Para ver todas as variáveis globais, vá na barra de menu e clique em**Scripts** depois selecione a opção **Show Built-in Variables**.

Repare que estas não possuem **global.**. Isso porque são do próprio game maker.

Variável local de evento:

Essa variável é comumente chamada de "variável temporária". Isso porque ela é declarada em um evento e descartada nesse mesmo evento. Vamos fazer um exemplo simples, mas antes vamos aprender a declarar essa variável temporária:

[var][nome da variável][;]

> **Código:** Selecionar todos
>
> ```
> var meu_x;
> ```

Dessa maneira você declarou a variável **meu_x**. Você coloca o construtor**var** depois coloca o **nome da variável** e por fim coloca **ponto e virgula (;)**. Para declarar mais de uma variável temporária faça o seguinte:

> **Código:** Selecionar todos
>
> ```
> var meu_x, meu_y, meu_cachorro, meu_papagaio;
> ```

Você acaba de criar quatro variáveis temporárias. Veja que você só precisou apenas colocar virgulas (,) sem a necessidade de colocar os outros construtores.

Após declarar a variável temporária você pode dar um valor à ela. É a mesma forma de uma variável comum:

> **Código:** Selecionar todos
>
> ```
> //Declaro a variável meu_x
> var meu_x;
>
> //Coloco o valor 320 na variável meu_x
> meu_x = 320
> ```

Faça um teste. Coloque esse código no evento **Create** de um objeto e coloque o na room. Agora adicione esse código no evento **Step**:

> **Código:** Selecionar todos
>
> ```
> //Muda a posição x do objeto para a posição da variável meu_x
> x = meu_x
> ```

Execute. Você receberá uma mensagem de erro. Isso porque a variável**meu_x** já não existe mais. Ela foi deletada ao fim do evento **Create** (Que ocorre antes do Step). Então, percebeu a relação da variável com o evento?

Para ter uma ideia mais clara, transfira o código para o evento **Create**, onde ela foi declarada. Veja que funciona perfeitamente.

Resumindo, a variável local de evento é usada para reduzir o consumo de memória da aplicação. Ou seja, ele não deixa a informação parada na memória sem uso. Ela mantém seu valor por apenas 1 step (frame), depois ela é deletada. Tanto que se a usássemos no evento Step, teríamos que declará-la sempre antes de a usar:

Código: **Selecionar todos**

```
//Declara variável temporária
var meu_numero;

//Guarda o valor 5 na variável
meu_numero = 5

//Move o objeto adicionando 5 a Step ao x
```

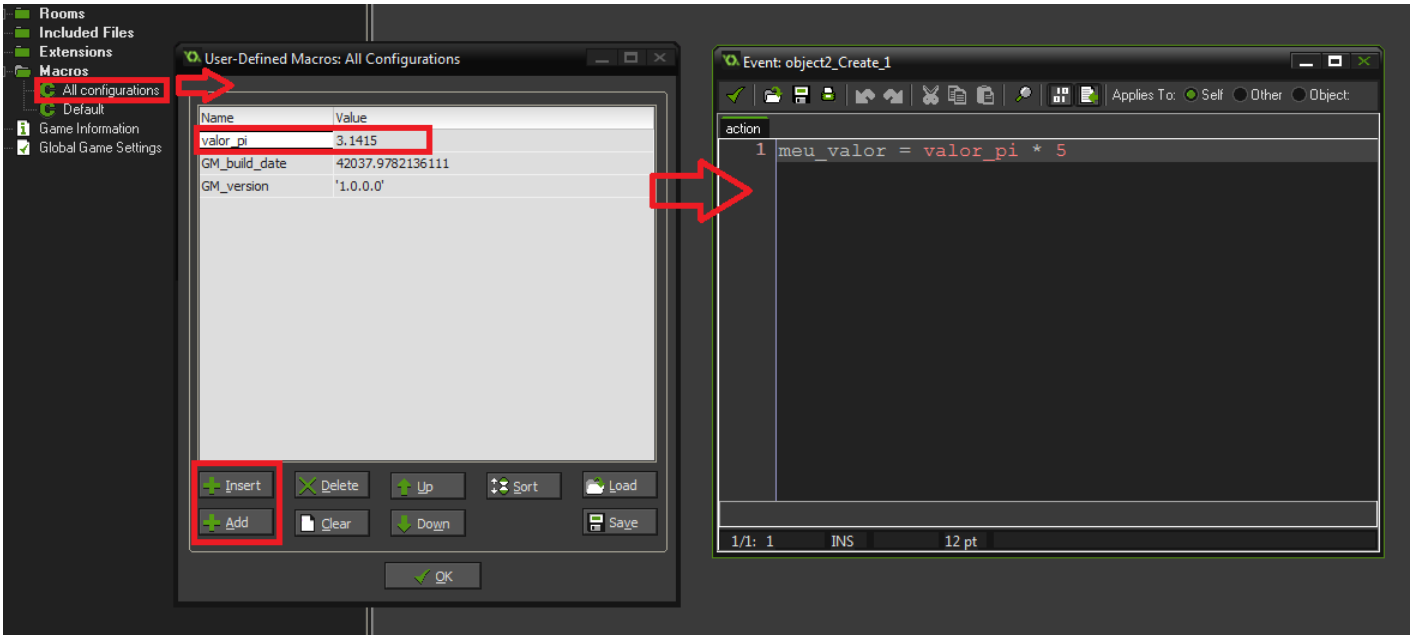
Constantes:

Uma constante é um identificador que não pode ter o valor modificado, por isso o nome de constante, valor constante, ou seja que não muda (Ao contrário das variáveis).

NOME	VALOR
True	1
False	0
pi	3.1415
c_red	255
vk_enter	13

Para ver todas as constantes, vá na barra de menu e clique em **Scripts**depois selecione a opção **Show Constants**.

E para adicionar suas próprias constantes vá na arvore de recursos e clique em **Macros** e depois selecione a opção **All Configurations**. Sempre que quiser adicionar uma nova clique em **insert** e defina o **nome** e o **valor**:



Resumindo:

Variável local de instância *só pode ser modificada pelo objeto em que foi criada (*Ou de outro objeto chamando este quando ele ainda existe).

Variável local de evento só pode ser modificada no evento em que foi criada.

Variável global pode ser modificada por qualquer objeto e evento.

E as **Constantes** não podem ter seu valor modificado

MUITO IMPORTANTE:

Não dê os mesmos nomes para variáveis globais, de instância e de evento. Ex:

> **Código:** Selecionar todos

```
carro = 5

var carro;

carro = 5

global.carro = 5
```

Isso vai causar uma série de erros. Logo, utilize nomes diferentes para cada uma dos tipos variáveis. O mesmo vale para nomes de **constantes, sprites, backgrounds, sons, paths, fonts, scripts, timelines, objetos** e**rooms**. Todos devem ter nomes diferentes.

INTRODUÇÃO EM GML

Aula 03 - Operações Matemáticas

(Atualizado 08/02/2015)

Aqui vou ensinar a usar as quatro operações matemáticas (adição, subtração, divisão e multiplicação).

A estrutura funciona da seguinte forma:

> **Código: Selecionar todos**
>
> ```
> [variável][sinal] =[variável, expressão ou numero]
> ```

Adição:

Como vimos na aula passada uma variável pode guardar números então vamos fazer o seguinte:

-Crie um objeto e em seu evento **Create** coloque:

> **Código: Selecionar todos**
>
> ```
> //pontuação igual a 0
> pontos = 0
> ```

-Agora no **step** coloque:

> **Código: Selecionar todos**
>
> ```
> //somando  1 ponto
> pontos += 1
> ```

Como o **Step** é um evento constante, ele sempre vai ocorrer enquanto este objeto existir. "**+**" é o sinal de adição, então o número aumentará.

-Mas para poder ver o que esta ocorrendo, vamos usar um código no evento de desenhos, o **Draw**:

Código: **Selecionar todos**

```
draw_text(10, 10, pontos)
```

Isso desenha um texto na tela informando o valor da variável na posição x 10 e y 10 na room.

Não procure entender esse código agora. Faça o seguinte, coloque esse objeto na room e vamos ver o que acontece. Execute.

Depois de executar você deve ter visto um número aumentando sem parar no canto esquerdo superior de sua tela. Essa é a adição em ação. Lembrando que você pode usar números menores que 1, como 0.25 e número menores que 0, como -1, -7 e etc (Fazendo o jogo se sinal). Não apague nada, usaremos esse mesmo objeto na próxima operação.

Subtração:

Substitua o código do **step** por esse:

Código: **Selecionar todos**

```
//subtraindo -1 ponto
pontos -= 1
```

Agora execute ele novamente. Dessa vez você verá o número diminuindo e ficando negativo. É a Subtração em ação seu sinal é "**-**".

Multiplicação:

Ainda no mesmo objeto substitua o código do **Create** por:

Código: **Selecionar todos**

```
pontos = 2
```

Fazemos isso pois se multiplicarmos 0 por outro número ele continuará 0 (0*5=0).

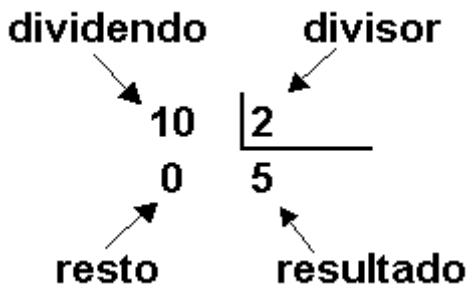
No mesmo objeto delete o evento **step**, e adicione o evento **Key press-> Space** e coloque o seguinte código:

Código: **Selecionar todos**

```
//multiplicando pontos por 2
pontos *= 2
```

Execute o jogo e não aperte espaço muitas vezes, pois o número chegará ao seu limite e o jogo travará. Nessa operação os números aumentam rapidamente, é a multiplicação em ação e seu sinal é "*".

Divisão:



Continuando no mesmo objeto, substitua o código do **create** por este:

> **Código:** **Selecionar todos**
>
> ```
> //pontos iniciais igual a 10000
> pontos = 10000
> ```

Substitua o que esta no **Key press-> Space** por:

> **Código:** **Selecionar todos**
>
> ```
> //dividindo pontos por 2
> pontos /= 2
> ```

Depois de ter executado, você deve ter percebido a divisão do número até chegar em 0, essa é a divisão em ação e seu sinal é **"/"**. E como 0 dividido por 0 é igual a 0, ele para por ai (Acontece um erro, pois nada pode ser dividido por 0).

Considerações:

Veja que usamos as operações diretamente nas variáveis, modificando-as. Mas e se quisermos que o resultado da divisão de 5 por 2 vá para nossa variável?

- Simples, fazemos isso:

> **Código:** **Selecionar todos**
>
> ```
> //Pontos terá o valor  de 5 dividido por 2, que é 2.5
> pontos = 5/2
> ```

- Também podemos usar outras variáveis e obter outros resultados:

Código: **Selecionar todos**

```
//Pontos terá o valor da divisão do valor da variável score em 2
pontos = score/2
```

Código: **Selecionar todos**

```
A = 5

B = 4

C = A + B
```

Note que não usamos o operador **"="** junto com o operador matemático. Caso usássemos aconteceria um erro.

Você pode usar parenteses para definir a ordem das operações:

Código: **Selecionar todos**

```
A = 5

B = 3

C = (A+B) / 2
```

Ou seja, primeiramente soma A + B e depois divide seu resultado por 2. Caso estivesse sem parenteses a divisão iria ser executada primeiro, ou seja B divido por 2 somado a A.

Operadores div e mod:

São operadores de divisão, mas são diferentes de **"/"**.

- div: retorna um valor inteiro da divisão:

Código: **Selecionar todos**

```
valor=5 div 2
```

O resultado seria 2,5 em uma divisão normal com **"/"**. Mas como só retorna um número inteiro o resultado é 2.

- mod: retorna o restante de uma divisão:

> **Código:** **Selecionar todos**
>
> ```
> valor= 5 mod 2
> ```

Ele retorna o que o **div** ignora, ou seja o **RESTO** da divisão que é 1. E quando a divisão é exata 0 é retornado, pois não há sobra.

É como entregar 50 balas para 20 crianças, você dá 2 para cada uma e fica com 10.

Comentários:

Como deve ter percebido nos códigos, sempre coloco "//" e uma frase explicando o código, esses são os comentários e não influenciam nada no jogo, depois de por "//" o resto da linha a ser digitado será um comentário. Há também como comentar em blocos definindo o incio com "/*" e o fim com "*/".

Exemplo:

> **Código:** **Selecionar todos**
>
> ```
> //Está linha não serve pra nada
> minha_var = 50
>
> /*Aqui eu posso pular uma linha
>
> E continuar a comentar, isso estando dentro do bloco de comentários.
>
> */
> ```

INTRODUÇÃO EM GML

Aula 04 - If/While e Expressões

(Atualizado 08/02/2015)

IF:

É uma estrutura de checagem condicional que em português significa "se". Agora explicarei uma estrutura básica de código usando **if**:

> **Código: Selecionar todos**
>
> ```
> if variavel == 0
> {
>     variavel2 = 100
> }
> ```

Tradução:

> **Código: Selecionar todos**
>
> ```
> Se variavel é igual a 0 o que esta entre chaves acontece, ou seja, variavel2 ig
> ual a 100.
> ```

Vamos fazer mais um exemplo simples. O Super Mario tem duas variáveis bem conhecidas, moedas e vidas. Para se conseguir uma vida coletando moedas, tem que conseguir 100 destas.

Baixe a seguinte **engine**:

[GM:S] DOWNLOAD DA ENGINE

[GM8] DOWNLOAD DA ENGINE

Nela a movimentação do personagem está pronta. Não se preocupe em entender esses códigos de movimento e gravidade, pois nas próximas aulas estarei ensinando a movimentação.

No evento **Other>>Game Start** do **controle** coloque:

Código: **Selecionar todos**

```
//Número de vidas
global.vidas = 3

//Número de moedas
global.moedas = 0
```

Ou seja, sempre que o jogo iniciar o personagem terá 3 vidas e 0 moedas.

Agora no **step** vamos fazer uma checagem de variáveis, e como o **step** é constante ele sempre estará checando:

Código: **Selecionar todos**

```
//Se moedas for maior ou igual a 100
if global.moedas >= 100
{
    //Retira 100 moedas
    global.moedas -= 100

    //Ganha uma vida
    global.vidas += 1
}
```

Você deve ter percebido o uso do ">" neste código, em matemática significa maior que, também tem o "<" (menor que), e adicionamos este símbolo antes do sinal de igual. Explicarei melhor isso mais abaixo, em **Expressões**.

Colocamos esse operador ali porque se você aumentar o número de moedas muito rápido, **global.moedas** pode passar de 100. Se fosse "if global.moeda =100..." o código não funcionaria bem, pois só se fosse exatos 100 que a outra expressão aconteceria.

Na engine tem um objeto chamado "**moeda**", cuidado!!! Nunca de nomes de variáveis e objetos e qualquer outro elemento(background, sprite, sons, rooms, fonts, paths e etc) iguais, senão acontecerão erros. Ex: você não pode dar o mesmo nome a uma sprite e a um objeto.

Então você percebeu agora, porque a variável se chama "**global.moedas**" e o objeto "**moeda**"?

Nesse objeto **moeda** adicione o evento **Collision** e selecione o objeto**player** e coloque o seguinte código:

Código: **Selecionar todos**

```
global.moedas += 1
```

```
instance_destroy()
```

Ou seja, ganhamos 1 moeda e a instancia de objeto **moeda** será destruída quando colidir com o **player**.

Agora para vermos as variáveis vamos ao objeto **controle** para desenhar as nossas variáveis. *Sempre use um objeto separado para desenhar coisas como pontuação, vidas e etc.*

No evento **Draw** do objeto **controle** abra o editor de códigos e coloque o seguinte código:

Código: Selecionar todos

```
draw_text(10,10,"Vidas: " + string(global.vidas) + "#" + "Moedas: "+string(glob
al.moedas))
```

Esse código desenha as variáveis **global.vidas** e **global.moedas**.

Quando for desenhar textos sempre os coloque-os entre aspas **"texto"**. E ja que estamos desenhando textos, tivemos que converter valores reais (números) em strings (palavras) usando a função **string(str)**.

"#" serve para pular uma linha. E sempre que for adicionar mais de um texto e mais de uma variável intercalados usa-se o somando seus valores para formar um novo texto.

Agora testem suas engines, para ver como saíram os resultados. Caso tenha saído algo errado baixe esta outra engine que esta completa:

[GM:S] DOWNLOAD DA ENGINE

[GM8] DOWNLOAD DA ENGINE

WHILE:

while é **quase** igual ao **if**, só há uma diferença:

Imagine todo o jogo como um ciclo que ocorre infinitamente. Quando esse ciclo passa pelo **if** ele faz a checagem normalmente, independente se a condição for verdadeira ou não e continua sem interromper esse ciclo.

Já quando o ciclo passa pelo **while** (que é um ciclo também), o ciclo pausa até que a condição vinculada a ele seja falsa. Exemplo:

- Crie um novo projeto.
- Adicione um novo objeto e um nova room.
- No **Create** do objeto coloque:

Código: **Selecionar todos**

```
moeda=0

if moeda < 100
{
    moeda += 1
}
```

-No **Draw** coloque (para sabermos o valor final da variável moeda):

Código: **Selecionar todos**

```
draw_text(10, 10, moeda)
```

- Insira o objeto na room e execute.

O valor da variável **moeda** mostrado é 1. Isso porque o **Create** ocorre apenas uma vez no inicio do ciclo do objeto e só deu tempo de checar a condição uma única vez.

- Agora troque o **IF** por **WHILE** e execute novamente.

Repare que:

- A tela fica preta por alguns instantes (Essa é a pausa no ciclo principal do jogo, aguardandoo ciclo while acabar).
- O valor de **moeda** agora é 100.

Ou seja, o **while** segurou o **Create** até que a variável **moeda** não fosse mais menor que 100.

Muito cuidado ao usar o while, se a condição nunca ficar falsa o jogo vai ficar travado.

EXPRESSÕES:

Vimos que para **if** e **while** funcionarem, antes eles checam uma condição para que o que está dentro das chaves ocorra.

Essa condição é uma expressão. As que usamos até agora foram simples, mas há formas de checar mais de uma condição em uma mesma expressão, e ainda usar uma expressão em uma operação aritmética.

Para que fique tudo mais organizado, pegue o costume de usar parenteses nas suas condições. Ex:

if (moedas >= 100) ...

A nossa expressão é (**moedas >= 100)**, ela retorna 1 (true) ou 0 (false) para o **if**. Ou seja um valor booleano.

Operadores de comparação:

> maior que
< menor que
>= maior ou igual que
<= menor ou igual que
== igual a (É diferente de apenas =, que significa atribuição)
!= diferente de

Agora vamos as expressões compostas:

AND:

Código: Selecionar todos

```
if ( (energia > 50) and (arma == 3) )
{
    energia -= 50
}
```

Nesse código a ação só acontece se duas condições forem satisfeitas, ter mais de 50 de **energia** e a **arma** ser 3. Se uma das duas for falsa a ação não ocorre. Usamos o operador **and** que faz com que dois lados de uma condição tenham que ser verdadeiros para a ação ocorrer.

OR:

Código: Selecionar todos

```
if ( (cor == c_red) or (cor == c_orange) )
{
    cor_quente = true
}
```

Nessa expressão é necessário que apenas uma condição seja verdadeira para que a ação ocorra. Usamos o operador **or** que faz com que pelo menos um dos lados de uma expressão tenha que ser obrigatoriamente verdadeiro para a ação ocorrer.

Como vimos expressões nos devolvem 1 ou 0 como valor. Podemos usar isso para simplificar códigos:

Código: Selecionar todos

```
x += (anda == true) * 5;
```

Ou seja, caso anda ter o valor 1 o objeto vai mudar de posição em +5. Repare também que estou usando "== true" nas expressões. Na verdade se eu quero verificar se ela é verdadeira, não preciso adicionar nada. Dai o código seria:

> **Código: Selecionar todos**
>
> ```
> x += (anda) * 5;
> ```

Obteremos o mesmo resultado. É a mesma coisa com o exemplo das moedas. Se eu quisesse poderia ter feito assim:

> **Código: Selecionar todos**
>
> ```
> if ( moedas >= 100 ) == true
> {
>
> }
> ```

Só que é desnecessário checar se ela é "verdade é verdadeira", sacaram? Mas quando queremos checar se a expressão é falsa não devemos omitir o "== false".

Enfim terminamos mais essa!

INTRODUÇÃO EM GML

Aula 05 - Else e Switch

(Atualizado 08/02/2015)

ELSE:

É uma palavra em inglês que significa "**senão**". No Game Maker ela é usada em conjunto com o "**if**".

Ex:

Código: Selecionar todos

```
//se moedas for maior ou igual a 100
if moedas >= 100
{
    moedas = 0
    vidas += 1
}
else //senão
{
    moedas += 2
}
```

Ou seja se **moedas** não for maior ou igual a **100** adicionará mais duas**moedas**. Para ver o resultado disso coloque o código acima no **step** e isso no **create**:

Código: Selecionar todos

```
vidas = 0
moedas = 0
```

E isso no **draw**:

Código: Selecionar todos

```
draw_text(10, 10, "Vidas: " + string(vidas) + "#" + "Moedas: " +
string(moedas))
```

Teste e veja o resultado.

Você também pode adicionar vários **elses** seguidos em conjunto com **if**:

Código: Selecionar todos

```
if tipo == 0       //se for 0
{
    forca=10
}
else if tipo == 1  //senão for  0 e for 1
{
    forca=20
}
else if tipo == 2  //senão for 0 ou 1 e for 2
{
    forca=30
}
else            //senão for nenhum
{
    forca=0
}
```

Uma vez usado **else if** na estrutura o **else** sozinho só pode aparecer na última condição.

SWITCH:

Quando usamos muitos **else** e **if** seguidos podemos ficar um pouco perdidos. A estrutura **switch** é uma forma simples e dinâmica de organizar e otimizar o código:

Código: Selecionar todos

```
switch tipo         //Expressão checada será a variável tipo
{

case 0:             //caso for 0

    forca=10

break;

case 1:             //caso for 1

    forca=20
```

```
break;

case 2:              //caso for 2

    forca=30

break;

default: forca=0;   //caso for nenhum

}
```

case marca o inicio de um bloco e **break** o fim.

Exemplo do GM:

Código: Selecionar todos

```
switch (keyboard_key)
{

  case vk_left:
  case vk_numpad4:
    x -= 4
  break

  case vk_right:
  case vk_numpad6:
    x += 4
  break

}
```

Repare que:

- Os dois pontos (**:**) são necessários, já o ponto virgula (**;**) não é obrigatório.
- O **default** (padrão) é usado caso nenhum dos casos ocorrer.
- Você pode usar 2 casos ou mais para apenas um resultado.
- Suas declarações devem ficar entre os dois pontos (**:**) o **break**.

INTRODUÇÃO EM GML

Aula 06 - For, do, until e repeat

(Atualizado 08/02/2015)

FOR:

É um ciclo assim como o **while**. Mas este é configurável. Ele pode realizar várias rotinas simultaneamente.

Essa estrutura é usada para poupar o tamanho do código. Mas não só para isso, serve também para ter controle sobre vários elementos usando um curto código.

Ex:

Se quisermos desenhar 10 círculos alinhados horizontalmente com espaços de 64 pixels entre seus centros na tela, faríamos o seguinte código:

> **Código:** Selecionar todos

```
//Escolhe a cor vermelha
draw_set_color(c_red)

//Desenha circulos
draw_circle(0,32,32,false)
draw_circle(64,32,32,false)
draw_circle(128,32,32,false)
draw_circle(192,32,32,false)
draw_circle(256,32,32,false)
draw_circle(320,32,32,false)
draw_circle(384,32,32,false)
draw_circle(448,32,32,false)
draw_circle(512,32,32,false)
draw_circle(640,32,32,false)
```

Como falei antes o **for** poupa o tamanho do código e cria rotinas simultâneas, então se quiséssemos usar o **for** no código seria simplesmente isso:

> **Código:** Selecionar todos

```
//Escolhe a cor vermelha
draw_set_color(c_red)

//Cria ciclo
for (var xx=0; xx < 640; xx+=64)
{
    //Desenha circulos
    draw_circle(xx, 32, 32, false)
}
```

Entendeu? Não né.

A estrutura do **for** é dividida em 3 partes:

- Inicialização: declaramos um valor inicial pra uma variável (var xx = 0);
- Condição para continuação do ciclo: define uma condição para fim para o ciclo (xx < 640);
- incremento da variável: Aumenta ou diminui o valor da variável para que em algum momento ela não satisfaça mais a condição (xx += 64).

Em todas as posições em que xx passou (0, 64, 128, 192, 256, 320, 384, 448 e 640) será desenhado o circulo vermelho.

Resumindo: enquanto **xx** não **for menor que 640**, serão adicionados mais**64** ao **xx** até que ele não seja **menor que 640**. E em cada passo que ele fez para chegar la será desenhado um circulo vermelho.

Desenhando uma linha usando pontos:

Código: Selecionar todos

```
//Cria ciclo
for(var yy = 0; yy < 100; yy += 1)
{
    //Desenha pixel
    draw_point(10, yy)
}
```

Ele fez uma linha vertical de 100 pixels certo?

Agora ficou mais simples né? Espero que tenham aprendido, porque é um pouco complicado explicar o **For**, até mesmo pra quem sabe usa-lo! XD.

Lembrando que o **for** é um ciclo também, então quando ele for executado ele vai pausar o ciclo do objeto até que sua expressão seja falsa.

DO e UNTIL:

Esses são sempre usados em conjunto e também fazem um ciclo. Exemplo:

> **Código: Selecionar todos**
>
> ```
> do {moeda+=1} until moeda>=100
> ```

Traduzindo: ***faça*** *{moeda+=1}* ***até que*** *moeda>=100*

Isso faz com que seja adicionado 1 até que a variável moeda seja maior ou igual a 100. Lembrando que a ação ocorre pelo menos uma vez já que a condição é checada depois da mesma. Logo, independente de moedas ser maior que 100, o moedas+=1 ocorrerá ao menos uma vez.

REPEAT:

Muito simples. Essa expressão repete um bloco de código o número de vezes que você desejar:

> **Código: Selecionar todos**
>
> ```
> //Repete 10 vezes
> repeat(10)
> {
>     //Cria uma instancia de tiro
>     instance_create(x,y,tiro)
> }
> ```

repete 10 vezes o código entre colchetes, criando 10 instancias de objeto**tiro** de uma vez.

INTRODUÇÃO EM GML

Aula 07 - Funções e Scripts

(Atualizado 08/02/2015)

Uma função é chamada por um nome, e contém zero ou mais argumentos dentro de parênteses e separados por vírgulas.

Exemplo, **instance_create** (função que tem varios argumentos):

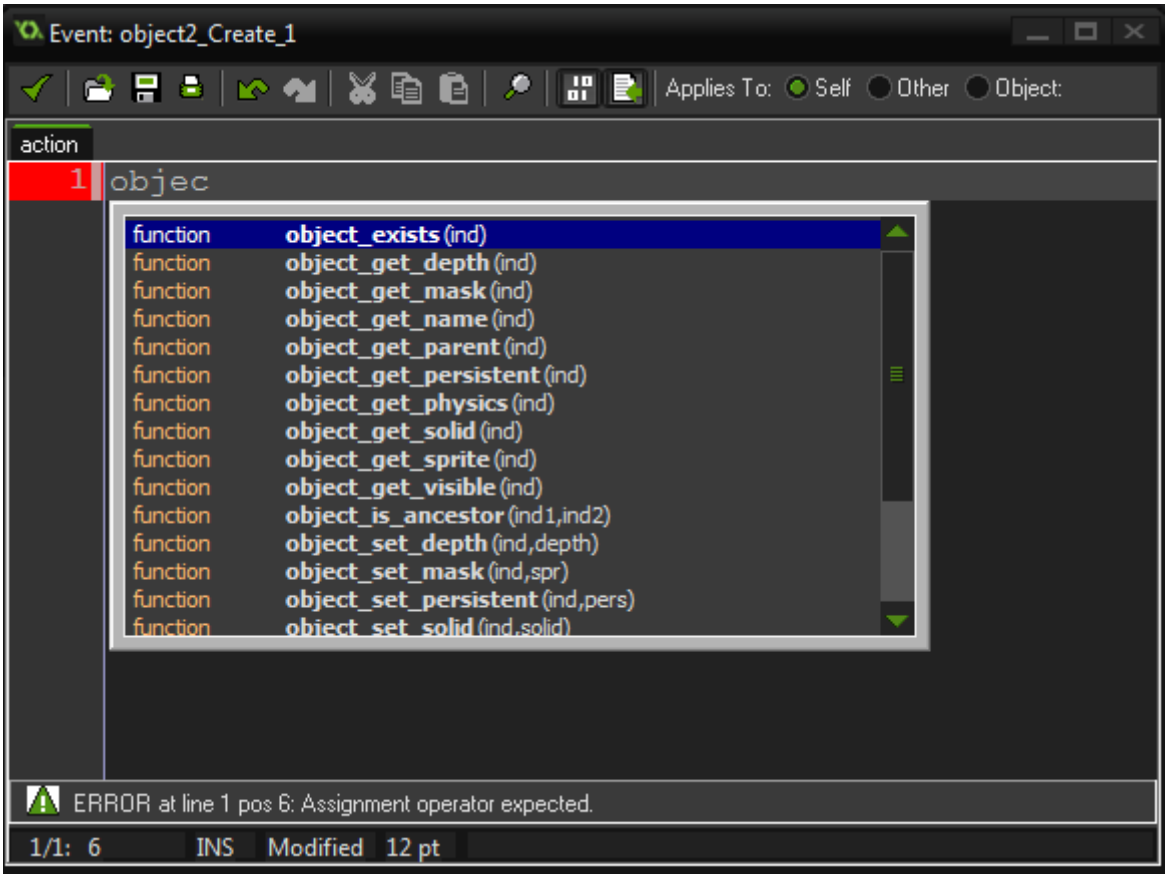
> **Código:** Selecionar todos
>
> ```
> //nome_da_função(argumento 1, argumento 2, argumento 3)
>
> //criando_objeto(posição x, posição y, nome do objeto)
>
> instance_create(x,y,obj)
> ```

Exemplo, **instance_destroy** (função que não tem argumentos):

> **Código:** Selecionar todos
>
> ```
> //nome_da_função()
>
> //destruir_instância()
>
> instance_destroy()
> ```

O Game Maker possui varias funções prontas para uso. Para ver todas, va no menu **scripts-> Show all Built-in Functions**.

Quando você digita parte de uma função do GM, ele busca as funções mais próximas do que você digitou, como na imagem abaixo:



Digitei somente **object_** e ele ja procurou e listou todas as funções que começam com **object_**. Caso a função que vai usar ja tenha aparecido la embaixo, de clique nela e ela é completada no seu código. Assim você poupa mais tempo digitando.

Scripts - criando funções:

Qualquer script que você crie é uma função. Os scripts são necessários para poupar rotinas extensas e otimizar o código.

Ao criar um script, dê um nome à ele. Exemplo, vamos calcular o IMC (Índice de massa corporal) que é o peso/altura².

> **Código:** Selecionar todos
>
> ```
> //calcula_imc(peso,altura)
>
> imc=argument0/(argument1*argument1)
>
> return imc
> ```

Repare que:

- **argument0** (representa o peso) é o primeiro argumento (peso) da função e **argument1** (altura) é o segundo argumento.
- Guardamos o resultado na variável **imc**.

- Retornamos o valor de **imc** para a função.

O **return** serve para que possamos utilizar a função como se fosse uma "**variável**":

> **Código: Selecionar todos**
>
> ```
> Meu_IMC=calcula_imc(82,1.80)
> ```

O valor da variável **Meu_IMC** será o que foi retornado na função. Ou seja, o valor da variável **imc**.

Há também as funções que não retornam um valor significativo (Retorna 0 ou -1, que nunca mudam). Por exemplo, você tem uma rotina onde há muito código que é usado em varias situações, mas não muda muito. Ele não calcula nada e serve somente para criar uma ação:

> **Código: Selecionar todos**
>
> ```
> //cria_explosao(som,objeto)
>
> //Toca o som de explosão
> sound_play(argument0)
>
> //Cria o objeto explosão
> instance_create(x,y,argument1)
> ```

Usaria assim:

> **Código: Selecionar todos**
>
> ```
> cria_explosao(sd_explosao,obj_explosao)
> ```

Não retornamos nenhum valor (Porque não precisavamos) e toda situação que em que criarmos uma explosão o código será menor e personalizável, podendo escolher o som e o objeto da explosão.

-Você pode usar até **16 argumentos** dentro de um script (de argument0 à argument15).

Nas próximas aulas veremos muitas outras funções. E para saber o que as funções embutidas no Game Maker retornam, veja o **HELP (F1)**.

INTRODUÇÃO EM GML

Aula 08 - Objetos e Instâncias

Atualizada(25/02/2015)

Objetos:

Rapaz, essa é uma aula essencial, presta atenção nela! É um tanto longa, então não precisa ver tudo de uma vez.

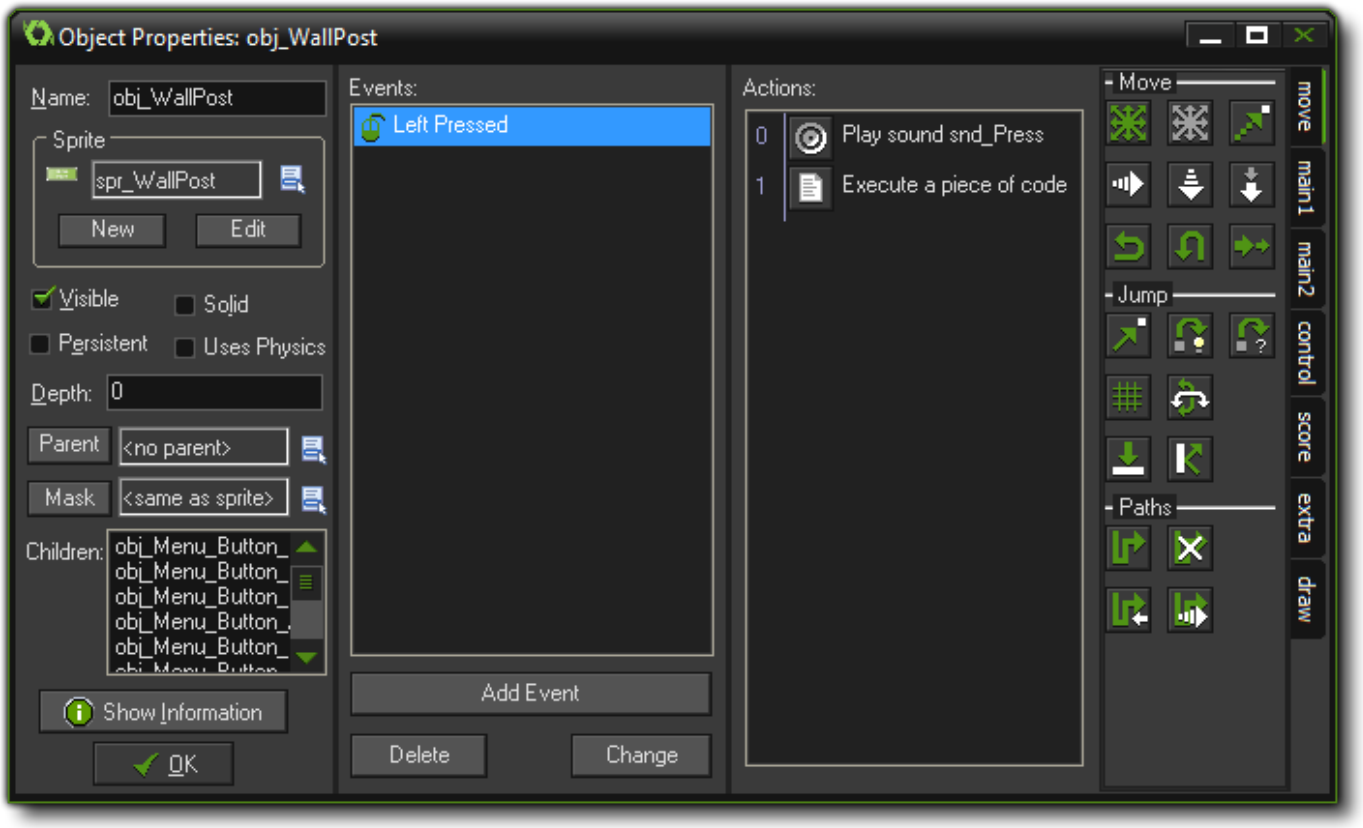
Hoje iremos aprender o sistema base em que são inseridos nossos códigos, os objetos.

Sua definição pode variar dependendo da linguagem, mas o essencial é você saber que um objeto controla os aspectos lógicos do nosso jogo, tanto visíveis, como invisíveis.

Exemplos do que pode ser um objeto:

Mario
Inimigo
Moeda
Controlador que escolhe que música deve tocar em qual fase.

Ao criar um objeto na nossa arvore de recursos nos deparamos com o seguinte:



Name: aqui damos um nome ao objeto.

Sprite: definimos uma imagem da pasta Sprites para o objeto. Os botões NEW e EDIT são atalhos para criar uma nova sprite ou editar a que está selecionada.

Visible: Se este objeto irá aparecer durante a execução do jogo. Isso inclui os eventos Draw do objeto.

Solid: Se este é um bloco que não se move, como chão ou parede, marque esta opção. Assim a física básica do jogo tratará esse objeto de forma diferente, tentando impedir que outros o atravessem.

Persistent: Isso muda o comportamento do objeto durante a passagem entre uma room e outra. Quando trocamos de room, todos os objetos da room anterior são deletados e os da nova room aparecem. A persistência, faz com que o objeto não seja deletado nessa passagem de room. Todas suas características são preservadas, como sua posição por exemplo. Ele é um objeto "teimoso" xD.

Use Physics: Selecione isso se for usar o Sistema de física do GMS.

Depth: Aqui se define a profundidade em que é desenhado o objeto. Todos tem por padrão o valor 0, e o que vai definir se um objeto é desenhado sobre outro é a ordem de criação:

É por isso que existe essa opção de "camada". Colocando uma valor menor o objeto será desenhado por cima, colocando um valor maior ele será desenhado atrás:

Parent: Aqui você escolhe um pai para seu objeto. Esse objeto irá herdar todas as configurações e códigos do seu pai. Logo ele é um filho. Vamos dizer que tu tem um inimigo e quer criar outro, mas este tem apenas a sprite diferente. Em vez de escrever todo o código novamente, tu simplesmente define o novo objeto como filho do inimigo, mudando apenas a sprite.

Mask: Quando você cria a sprite que irá usar no objeto, ao lado já pode configurar a mascara de colisão. Por padrão a mascara é a área total da imagem, ou seja um retângulo. Mas por muitas vezes você tem um personagem com várias sprites em diversos movimentos, e todas elas de dimensões (altura x largura) diferentes. Isso se torna um problema, pois na hora da mudança das sprites o personagem pode ficar preso ao chão, pois a sprite de pulo era mais alta que a normal. Resolvemos isso usando uma Sprite como máscara de colisão. Escolhemos geralmente uma retangular não animada.

Eventos:

Ao centro temos o controle de eventos do objeto, trataremos disso amplamente na próxima aula. Mas fique com esse resumo fixo na cabeça:

- ***Create**:* ocorre apenas uma vez, quando objeto é criado;
- ***Step**:* ocorre o tempo todo enquanto o objeto existir;
- ***Draw**:* ocorre o tempo todo, mas é usado para inserir funções de desenho (**draw_**);

Ações:

Ao lado temos ações que podemos adicionar ao game. Há muitos bloquinhos **Drag & Drop**, mas usaremos somente o da aba **control -> Execute a piece of code**. É aqui onde colocamos nosso código e podemos fazer tudo que é feito com os outros bloquinhos de ação e ainda muito mais.

Funções:

> **Código:** **Selecionar todos**
>
> ```
> object_exists(obj);
> ```

obj: nome do objeto.

Verifica se um objeto existe ou não na arvore de recursos (Não na room).

Exemplo:

> **Código:** **Selecionar todos**
>
> ```
> //Se o obj_lutador existe
> if object_exists(obJ_lutador)
> {
>     //Muda a posição horizontal do obj_lutador para 32
>     obj_lutador.x = 32
> }
> ```

> **Código:** **Selecionar todos**
>
> ```
> object_get_depth(obj);
> ```

obj: nome do objeto.

Pega o valor de profundidade de desenho de um objeto.

Exemplo:

> **Código:** **Selecionar todos**

```
//Se a profundidade do player for maior que -50
if object_get_depth(obj_player) > -50
{
    //A profundidade do player será -50
    obj_set_depth(-50, player);
}
```

Código: Selecionar todos

```
object_get_mask(obj);
```

obj: nome do objeto.

Retorna a sprite que está sendo usada como mascara pelo objeto.

Exemplo:

Código: Selecionar todos

```
//Pega a máscara do objeto luigi
mask_index = object_get_mask(luigi)
```

Código: Selecionar todos

```
object_get_name(obj);
```

obj: nome do objeto.

Retorna o nome do objeto selecionado em uma string.

Exemplo:

Código: Selecionar todos

```
//Guarda a string "player" na variável meu_nome
meu_nome = object_get_name(player);
```

Código: **Selecionar todos**

```
object_get_parent(obj);
```

obj: nome do objeto.

Retorna o objeto que é pai do objeto selecionado.

Exemplo:

Código: **Selecionar todos**

```
//Pega o pai do obj_inimigo_3
pai_do_outro = object_get_parent(obj_inimigo_3)
```

Código: **Selecionar todos**

```
object_get_persistent(obj);
```

obj: nome do objeto.

Retorna se o objeto é persistente (true) ou não (false).

Código: **Selecionar todos**

```
object_get_solid(obj);
```

obj: nome do objeto.

Retorna se o objeto é sólido (true) ou não (false).

Código: **Selecionar todos**

```
object_get_sprite(obj);
```

obj: nome do objeto.

Retorna a sprite escolhida inicialmente pelo objeto selecionado. Sim aquela da janela de propriedades do objeto diferntemente do sprite_index que pega a sprite usada no momento.

Código: **Selecionar todos**

```
object_get_visible(obj);
```

obj: nome do objeto.

Retorna se o objeto é visível (true) ou não (false).

Código: **Selecionar todos**

```
object_get_physics(obj);
```

Retorna se o objeto usa física (true) ou não (false).

Código: **Selecionar todos**

```
object_is_ancestor(obj);
```

obj: nome do objeto.

Retorna se o objeto é pai de outros objetos (true) ou não (false).

As funções do tipo _set_, ao invés de pegar um informação atribuem. Sendo o primeiro argumento o valor, e o segundo o objeto selecionado.

Exemplo:

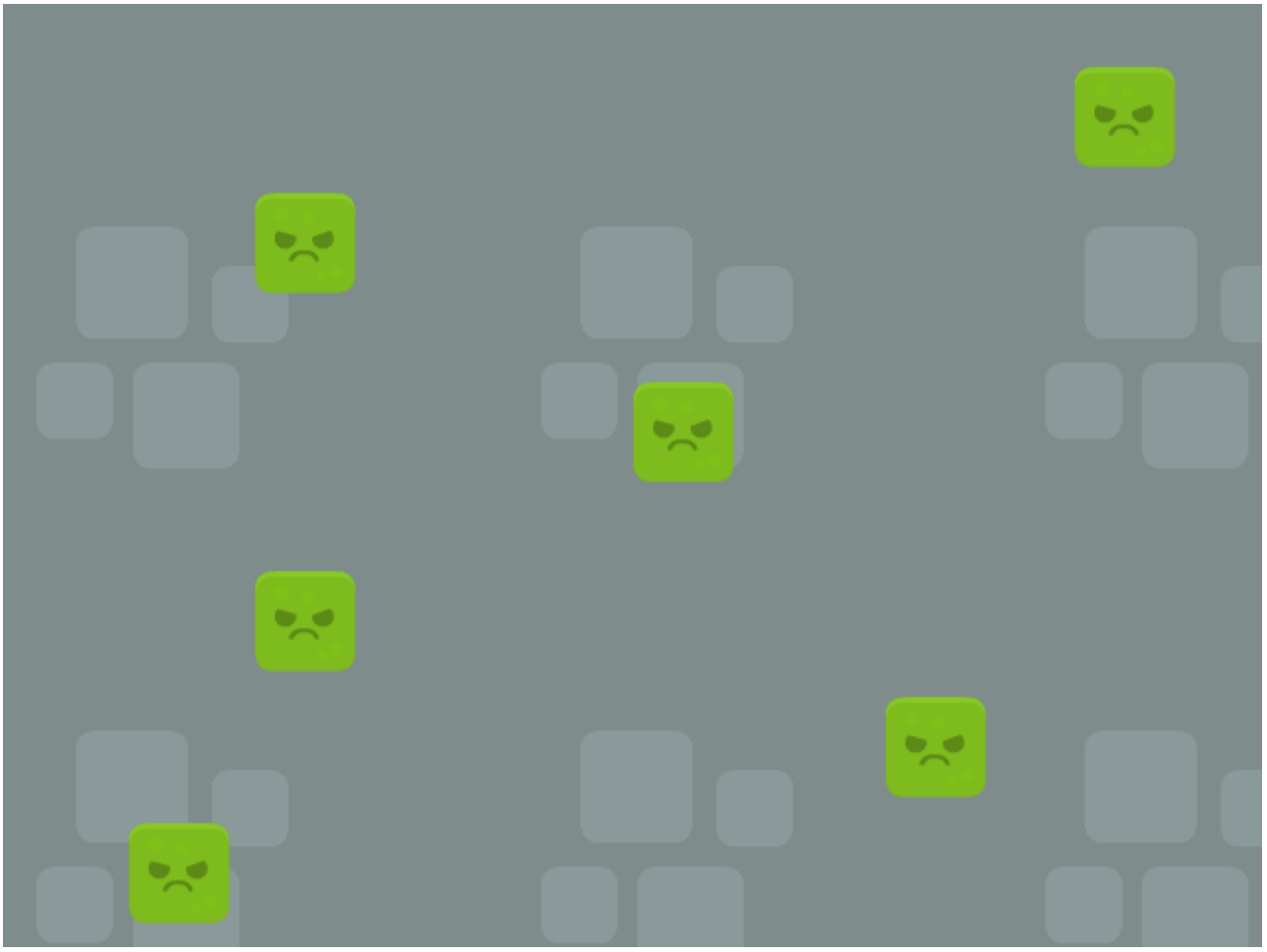
Código: **Selecionar todos**

```
if desaparecer == true
{
     object_set_visible(false, obj_bloco);
}
```

Lembrando que essas funções alteram as caracteristicas de todas as réplicas desse objeto. Ou seja as instancias, que veremos abaixo.

Instancias:

Você deve ter reparado que podemos adicionar quantas réplicas quisermos de nossos objetos no editor de rooms do GMS, certo? O nome dado a essas réplicas é instancia de objeto.



Vejamos as funções:

Código: **Selecionar todos**

```
instance_change(obj, perf);
```

obj: objeto a ser trocado.
perf: Se irá executar o evento Destroy do antigo objeto e o Create do novo.

Troca o objeto atual por outro.

Exemplo:

Código: Selecionar todos

```
/*Troca para o objeto player_morrendo sem executar o
evento Destroy do objeto anterior e o Create do objeto
novo.*/
instance_change(player_morrendo, false);
```

Código: Selecionar todos

```
instance_copy(perf);
```

perf: Irá executar o create da nova cópia (true ou false).

Essa função cria uma cópia exata da instancia atual, com todos seus valores (Incluindo a posição). Você pode escolher se quer ou não que o create da cópia aconteça.

Retorna o ID da cópia criada.

Código: Selecionar todos

```
instance_count
```

Esta variável retorna a quantidade de instâncias de qualquer objeto ativas.

Código: Selecionar todos

```
instance_create(x, y, obj);
```

x: posição horizontal onde será criado o novo objeto.
y: posição vertical onde será criado o novo objeto.
obj: objeto a ser criado.

Cria uma nova instância de objeto em uma determinada posição.

Retorna o ID da instância criada.

Exemplo:

Código: **Selecionar todos**

```
//Cria uma instancia de obj_bala e lhe atribui velocidade horizontal 10.
var tt = instance_create(x, y, obj_bala);
tt.hspeed = 10;
```

Código: **Selecionar todos**

```
instance_destroy();
```

Destrói a instância.

Código: **Selecionar todos**

```
instance_exists(obj);
```

obj: objeto ou instância a ser checado.

Checa se existe um objeto ou uma instancia de objeto. Retorna true caso sim e false caso não.

Exemplo:

Código: **Selecionar todos**

```
//Se não existir pelo menos um instância do obj_iminigo
if not instance_exists(obj_inimigo)
{
   room_goto(fase_10);
}
```

Código: **Selecionar todos**

```
instance_find(obj, n);
```

obj: nome do objeto.
n: numero da ordem.

Procura uma instancia de objeto dada a sua ordem (Do mais velho ao mais novo) começando por zero e terminado no [total de instancias -1].

Retorna a ID da instancia encontrada.

Exemplo:

Código: **Selecionar todos**

```
/*Procura todos os objetos e verifica se passou da parte inferior da room,
caso alguma sim ele vai pra cima*/
for (var i = 0; i < instance_number(obj_inimigo); i += 1)
{
   var o = instance_find(obj_Enemy,i);

   if o.y > room_height { o.vspeed = -15; }
}
```

Código: **Selecionar todos**

```
instance_furthest(x, y, obj);
```

x: posição horizontal onde será checado a distância do objeto.
y: posição vertical onde será checado a distância do objeto.
obj: objeto a ser checado.

Retorna a ID da instância de objeto mais longe da posição dada.

Exemplo:

Código: **Selecionar todos**

```
var longe = instance_furthest(x, y, obj_inimigo);

//Destrói a instancia de obj_inimigo mais longe
with (longe)
{
```

```
    instance_destroy();
}
```

Código: Selecionar todos

```
instance_id[0...]
```

Vetor/Array que guarda todas as instancias de todos objetos ativas na room.

Exemplo:

Código: Selecionar todos

```
for (var i = 0; i < instance_count; i += 1;)
{
    with (instance_id[i]) { speed += 5; }
}
```

Código: Selecionar todos

```
instance_nearest(x, y, obj);
```

x: posição horizontal onde será checado a distância do objeto.
y: posição vertical onde será checado a distância do objeto.
obj: objeto a ser checado.

Retorna a ID da instância do objeto mais perto.

Exemplo:

Código: Selecionar todos

```
var perto = instance_furthest(x, y, obj_inimigo);

//Segue instancia de obj_inimigo mais perto
mp_potential_step(perto.x, perto.y, 5, 0);
```

Código: Selecionar todos

```
instance_number(obj);
```

obj: obj a ser checado.

Retorna o número de instancias de determinado objeto.