Game Maker Tutorial

A First Person Shooter

Written by Mark Overmars

Last changed: December 23, 2009
Uses: Game Maker 8.0, Pro Edition, Advanced Mode
Level: Advanced

In this tutorial we are going to explore the 3D drawing functions in *Game Maker*. "But I thought *Game Maker* was for 2-dimensional games?" you might ask. Well, yes, it is meant for 2-dimensional games. But there are functions for 3D graphics. And many 3-dimensional looking games actually are 2-dimensional. In this tutorial we will create a first person shooter. Even though all the graphics will look 3-dimensional, the game actually takes place in a 2-dimensional world. So we will use the standard machinery of *Game Maker* to create this 2-dimensional game, but rather than drawing the 2-dimensional room we will create 3-dimensional graphics instead. As we will see, this is not very difficult. But you do need to understand GML pretty well and must not be afraid of writing quite some pieces of code. Hence, this tutorial is for advanced users only. And don't forget, the 3D graphics functions are only available in the Pro Edition of *Game Maker*.

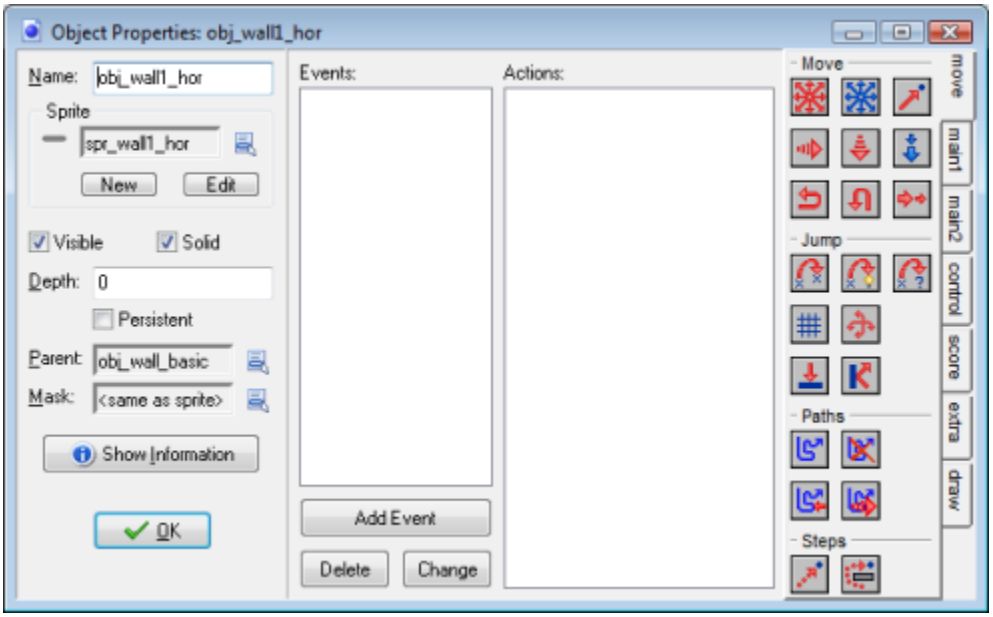
We will create the game in a number of steps, starting with a simple 2-dimensional version and then adding the 3D graphics. All partial games are provided in the folder `Examples` that comes with this tutorial and can be loaded into *Game Maker*.

A First 2-Dimensional Game

As indicated above, the actual game play of the first person shooter will be 2-dimensional. So we first have to create the 2-dimensional game, which we will then later convert to 3-dimensional graphics. Because graphics are not important at this stage, we do not need fancy sprites. All objects (player, enemies, bullets, etc.) will be represented by simple colored disks. And there are some wall objects that will be represented by horizontal and vertical blocks. In this section we will make a simple 2-dimensional scene with rooms and a player character. Other items will be added later. The game can be found in the file `fps0.gmk` in the `Examples` folder.

We will create two wall objects: a horizontal one and a vertical one. We also create one base wall object, called `obj_wall_basic`. This object will become the parent of all other wall objects (later we will create more). This will make aspects like collision detection and drawing a lot easier as we will see. Wall

objects will have no behavior. All wall objects will be solid. So the horizontal wall object will look something like this:



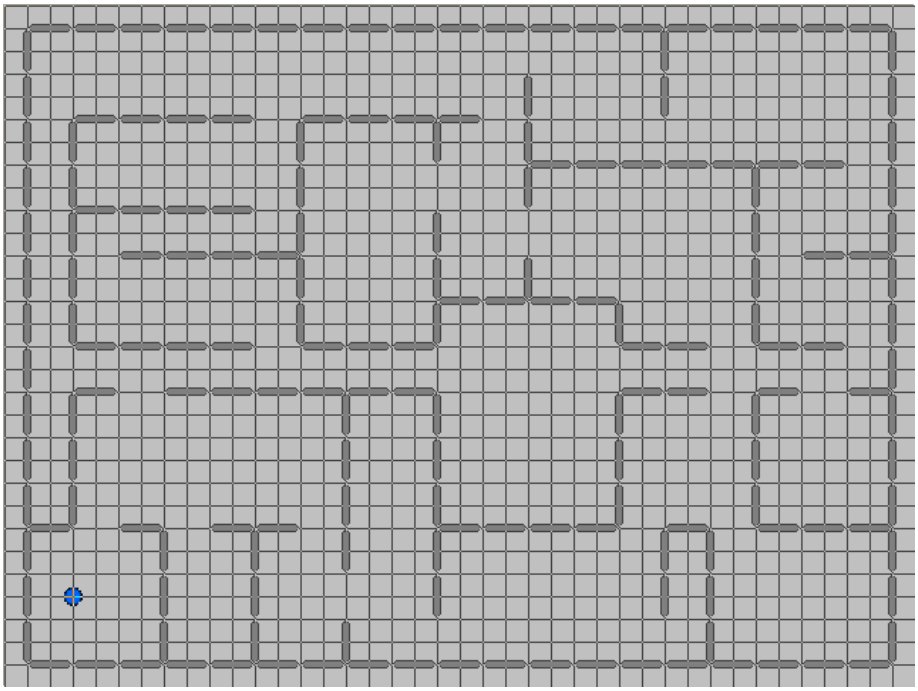
So it is pretty boring. Only the sprite and parent are filled in.

The next object we must create is the player object. We represent it with a small blue disk. We give it a red dot on one side to be able to see the direction it is moving in. This is only important in the 2-dimensional version and is irrelevant in the 3D version. In its **End Step** event we include a **Set Variable** action in which we set the variable `image_angle` to `direction`, to let the red dot indeed point in the correct direction. At the moment we only need to define the motion. To get a bit of smooth motion we do not want the motion to start and abruptly. This will look quite bad in the 3D version. So we gradually let it start moving and stop moving. To this end, in the **Keyboard** event of the <Up> key we include an **Execute Code** action with the following code:

```
{
  if (speed < 2) speed = min(2,speed+0.4);
}
```

So it will slowly gain speed until a maximum of 2 is reached. In the <Down> **Keyboard** event we do the same but in the opposite direction. In the <Left> and <Right> **Keyboard** event we simply increase or decrease the direction of motion. In the **Create** event we set the variable `friction` to 0.2 such that, once the player releases the <Up> key the speed decreases again. (You might want to play a bit with the maximum speed, speed increase, and friction to get the effect you want.) Finally, in the **Collision** event with the `obj_wall_basic` object we stop the motion (this is rather ugly and we will see later how to improve this). Because all other walls objects have the `obj_wall_basic` as parent we only need to define one collision event.

Finally we need to create a level with various regions. Careful level design will be important to create interesting game play. For the time being we restrict ourselves to the weird looking room below.



Better load the game `fps0.gmk` into *Game Maker* and play a bit with it. Looks rather boring doesn't it?

Turning It into a 3-Dimensional Game

This is probably the most important section. Here we are going to turn our boring 2-dimensional game into a much better looking (but actually still boring) 3-dimensional game.

Our player object becomes the most important object. In its creation event we initialize the 3D mode by using the following piece of code:

```
{
  d3d_start();
  d3d_set_hidden(true);
  d3d_set_lighting(false);
  d3d_set_culling(false);
  texture_set_interpolation(true);
}
```

The first line starts 3D mode. The second line indicates that hidden surface removal is turned on. This means that objects that lie behind other objects are not visible. Normally this is what you want in a 3-dimensional game. The next line indicates that we are not going to use light sources. Finally, the fourth line indicates that no culling should happen. This is slightly more complicated to understand. When

drawing a polygon in space it has two sides. When culling is on only one of the two sides is drawn (defined by the order of the vertices). This saves drawing time when carefully used. But as our walls will be viewed from both sides, this is not what we want. (Actually, these three lines are not required because these are the default settings but we included them such that you understand their meaning.) Finally we set the texture interpolation. This makes the texture we use look nicer when you get close to them. (You can also set this in the **Global Game Settings**.)

To make sure that the events in the player object are performed before those in other objects, we give it a depth of 100 (you should remember that objects are treating in decreasing depth order). We also remove the **End Step** event that we added in our 2D version of the game to indicate the direction. This is no longer required and would actually lead to problems later on.

The next thing to do is that the walls should become nice looking and are drawn in the correct way. To this end we will need to use some textures. In this tutorial we use a number of textures, some of them created by David Gurrea that can be found on the site http://www.davegh.com/blade/davegh.htm. (Please read the conditions of use stated there. Because of these conditions, the textures are not included with this tutorial. You should download them yourself.) We reduced them all in size to 128x128. It is very important that the sizes of textures are powers of 2 (e.g. 64, 128, 256) otherwise they won't match up nicely. Here is the wall, ceiling, and floor texture we will use:

We add these three images as background resources to the game. To let each wall object draw a wall with this background texture we need to know what the coordinates of the wall should be. We set this in the **Create** event of each individual wall object (because it will be different for each). For example, for the horizontal wall we put the following code in the **Create** event:

```
{
  x1 = x-16;
  x2 = x+16;
  y1 = y;
  y2 = y;
  z1 = 32;
  z2 = 0;
  tex = background_get_texture(texture_wall);
}
```

The last line requires some explanation. The function `background_get_texture()` returns the index of the texture corresponding to the indicated background resource. We will use this later when indicating the texture used to draw the wall. We put a similar piece of code in the creation event for all other wall objects.

To actually draw the wall will happen in the **Draw** event of the `obj_wall_basic` object. Here we include a **Code Action** with the following piece of code:

```
{
  d3d_draw_wall(x1,y1,z1,x2,y2,z2,tex,1,1);
}
```

It draws a wall, using the values that we set in the creation events. The last two parameters indicate that the texture must be repeated just once in both directions, so it fills the whole wall. Repeated textures can be used to fill large areas.

We are almost done. We still need to indicate how we look at the world and we must also draw the floor and ceiling of the rooms. This we will do in the draw event of the player object. The player object will from now on serve the role of the camera that moves through the world. Here we include an **Execute Code** action with the following piece of code:

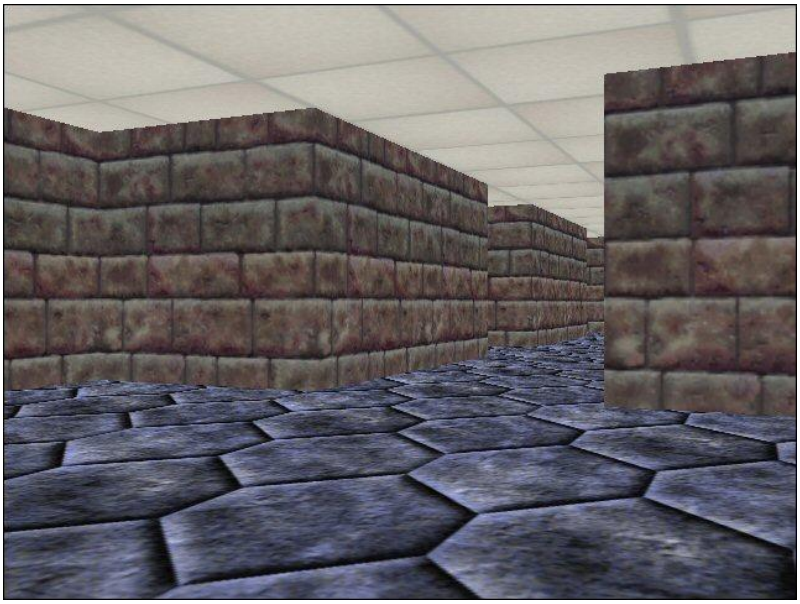
```
{
  // set the projection
  d3d_set_projection(x,y,10, x+cos(direction*pi/180),
                             y-sin(direction*pi/180),10, 0,0,1);
  // set color and transparency
  draw_set_alpha(1);
  draw_set_color(c_white);
  // draw floor and ceiling
  d3d_draw_floor(0,0,0,640,480,0,
                 background_get_texture(texture_floor),24,18);
  d3d_draw_floor(0,0,32,640,480,32,
                 background_get_texture(texture_ceiling),16,12);
}
```

There are three parts here. In the first part we set the projection through which we look at the room. The function `d3d_set_projection()` takes as its first three parameters the point from which we look, as the next three parameters the point we look towards, and as the last three parameters the upward direction (can be used to give the camera a twist). We look from position (`x`,`y`,10), that is, the position were the camera is but a bit higher in the air. The point we look towards looks rather complicated but this is simply a point that lies one step in the direction of the player. Finally we indicate that the upward direction of the camera is the z-direction (0,0,1).

The second part simply sets alpha to 1 meaning that all objects are solid. It also sets the color to white. This is important. Textures are actually blended with the current color. This is a useful feature (you can for example make the ceiling red by setting the drawing color to red before drawing the ceiling). But most of the time you want a white blending color. (The default drawing color in *Game Maker* is black so if you do not change it, the whole world will look black.)

In the third part we draw the floor and ceiling (at height 32) using the correct textures. As you see we indicate that the texture must be repeated many times over the ceiling rather than being stretched all over it.

That is all. The game can be found in the file `fps1.gmk` in the `Examples` folder. You can now run the game and walk around in your world. Suddenly our game looks completely different, even though it is in fact still our 2-dimensional game. Only the graphics has become 3D. The game should look something like this:



Improving the Experience

In this section we are going to improve on the "game" created in the previous section. First of all we will add some more variation, by using different kinds of walls. Secondly we will improve the level design. To improve the visual appearance (and add some creepiness) we will add fog to the world. And finally we will improve the motion through the world.

More variation in walls

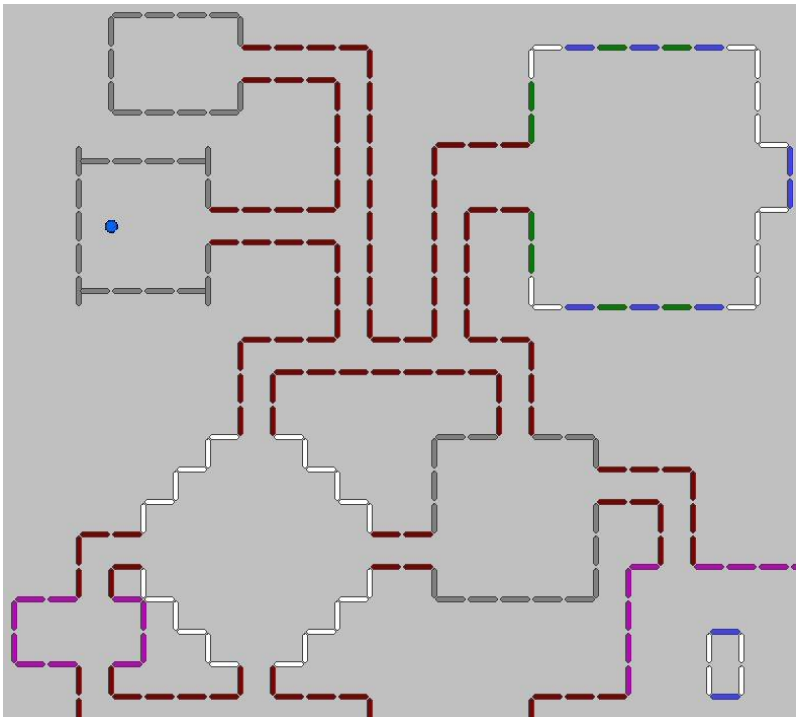
The world as we designed it up to now looks rather boring. All walls look the same. Not only is this boring, it makes it also difficult for the player to orient himself in the world. This is bad game design (unless it is the intention of the game). It is more user-friendly if the player can more easily remember where he is and where he came from.

The solution to this problem is rather trivial. We add a number of additional textures in the game and make a number of additional wall objects. These will be exactly the same as the other ones, except that in their **Create** event we assign a different texture to them. Remember to give all the wall objects the `obj_wall_basic` wall as their parent. By using different colored sprites for the different wall it is easier to create the levels. (Note that these sprites are not used in the game but they are shown in the levels.)

Better level design

The current level has a number of problems. First of all, there are wall that look very flat because you can see both sides of them. This should be avoided. Secondly, we tried to fill up the whole level with regions. Even though this is normal for real buildings, it is not very good for games. To get interesting surprises we better have a sparser set of areas with corridors between them. By adding turns in the corridors we reduce the part of the world that the player can see at any one time. This increases the feeling of danger. You never know what will lie behind the next corner. (As we will see below there is another reason for laying out the level in a more sparse way. It enables us to make the graphics faster by drawing only part of the level.)

To create a sparser level we will need more space. So we will increase the size of the room quite a bit. To avoid that this also changes the size of the window, we use a single view of the window size we want. The position of the view in the room does not matter because we anyway set the projection ourselves. Here is an image of part of the new level. The different colored walls correspond to different textures. You can find the new game in the file `fps2.gmk`.



Adding fog

To add a scarier feel you can add fog to your game. The effect of fog is that objects in the distance look darker than objects that are nearby (or lighter, depending on the color of the fog). When an object is really far away it becomes invisible. Fog adds atmosphere to the game, gives a better feeling of depth and distance, and makes it possible to avoid drawing objects that are far away, making things faster. To enable fog, just add the following line to the **Execute Code** action in the **Create** event of the player object (after enabling 3D).

```
d3d_set_fog(true,c_black,10,300);
```

If enables the fog, with a black color, starting at distance 10, and becoming completely black at distance 300. (Note that on some graphics cards fog is not supported. So better make sure that your game does not depend on it.) With fog enabled, the world looks as follows:

Better motion

What remains to be done is to improve the motion control. When you hit an object you stop moving at the moment. This is not very nice and makes walking through corridors more difficult. When you hit a wall under a small angle, you should slide along the wall rather than stop. To achieve this we need to make some changes. First of all, we no longer make the walls solid. When objects are solid we cannot precisely control what happens in case of a collision because the system does this for us. But we want all the control. So all wall objects are made non-solid. In the collision event we include an **Execute Code** action with the following piece of code:

```
{
  x = xprevious;
  y = yprevious;
  if (abs(hspeed) >= abs(vspeed) &&
      not place_meeting(x+hspeed,y,obj_wall_basic))
    { x += hspeed; exit;}
  if (abs(vspeed) >= abs(hspeed) &&
      not place_meeting(x,y+vspeed,obj_wall_basic))
    { y += vspeed; exit;}
  speed = 0;
}
```

You should read this code as follows. In the first two lines we set the player back to the previous position (to undo the collision). Next we can check whether we can move the player just horizontal (sliding along horizontal walls). We only do this when the horizontal speed is larger than the vertical speed. If the horizontal slide position is collision free with the walls we place it there. Next we try the same with sliding vertically. If both fail we set the speed to 0 to stop moving.

A second change we make is that we allow for both walking and running. When the player presses the <Shift> key we make the player move faster. This is achieved as follows: In the <Up> and <Down> **Keyboard** events we test whether the <Shift> key is pressed and, if so, allow for a faster speed, as follows:

```
{
  var maxspeed;
  if keyboard_check(vk_shift) maxspeed = 3 else maxspeed = 1.5;
  if (speed < maxspeed ) speed = min(maxspeed ,speed+0.4);
}
```

A last issue is that in most First Person Shooter games the player is also able to strafe, that is, move left and right sideways. For this we use the <z> and <x> key. The motion should perpendicular to the current direction the player is facing. For the <z> key the code becomes:

```
{
  var xn,yn;
  xn = x - sin(direction*pi/180);
  yn = y - cos(direction*pi/180);
  if not place_meeting(xn,yn,obj_wall_basic)
    { x = xn; y = yn; }
}
```

A similar piece of code is required for the <x> key. You can try out the game in the file `fps2.gmk` for the effect.

Adding Objects in the World

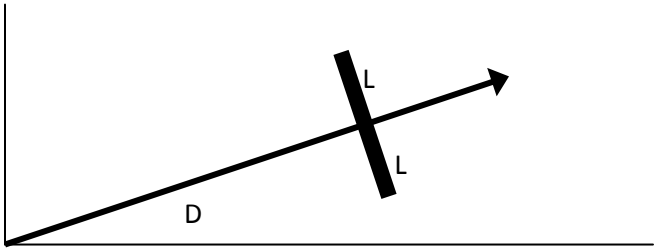
Our world is still rather empty. There are only some walls. In this section we are going to add some objects to our world. These are mainly for decoration but the player (and opponents) can also hide behind them. There are globally speaking two ways of adding objects to the world. The first way is to create a 3-dimensional object consisting of texture mapped triangles. This gives the nicest effect but is rather time consuming, both to create and to draw. Instead, we will simply use sprites to represent these objects in our world. We will use this technique for everything in our world: the bullets, plants, weapons, explosions, etc. But things are slightly more difficult than it seems. As sprites are flat, if we look from the wrong direction, they will not be visible. We will solve this by keeping sprites facing the player.

Facing sprites

As an example, we are going to create a plant object that we can place in the rooms to make the world look more interesting. Like the player and the wall segments we use a very simple sprite to represent the plant in the room. This is easy for designing the room and will be used for collision checking such that the player cannot walk through the plant. We create a plant object and give it as parent again our basic wall object. So for the player (and later for bullets) the plant will behave like a wall. We will though overwrite the draw event because we must draw something else. For drawing the plant we need a nice sprite. This sprite will be partially transparent and to make the edges look better we switch on the option to smooth the edges. In the drawing event we use this sprite image to draw the plant on a vertical wall. Because the sprite is partially transparent you only see that particular part of the wall.

There is though one important issue. The sprite is a flat image. If you would look at it from the side it becomes very thin and even disappears. To solve this we can use a simple trick that is used in many other games as well. We let the sprite always face the camera. So independent of the direction from which you look, the sprite will look the same. It kind of rotates with you. Even though this might sound unnatural the effect is pretty good. So how do we do this? We need a little bit of arithmetic for this. The following picture shows the situation. The arrow indicated the direction the player is looking, denoted with D. The black rectangle represents the sprite. Assuming the sprite has a length of 2L, the positions of the two corners are:

- `(x-L.sin(D*pi/180),y-L.cos(D*pi/180))`
- `(x+L.sin(D*pi/180),y+L.cos(D*pi/180))`



The following piece of code can e.g. be placed in the drawing event for the plant object.

```
{
  var ss,cc,tex;
  tex = sprite_get_texture(spr_plant,0);
  ss = sin(obj_player.direction*pi/180);
  cc = cos(obj_player.direction*pi/180);
  d3d_draw_wall(x-7*ss,y-7*cc,20,x+7*ss,y+7*cc,0,tex,1,1);
}
```

In the code we determine the correct sprite texture, compute the two values indicated above, and then draw the sprite on a 14x20 size wall that is standing on the floor, rotated in the correct direction. Clearly the precise size and position depend on the actual object that must be drawn.

If there are multiple sprites that must be drawn this way it is actually easier to store the sine and cosine in global variables, updated by the player object, rather than recomputing them for each sprite that must be drawn. This is the way we will do it in the game we are creating. In the **End Step** event of the player object we store the two values in global variables `camsin` and `camcos`.

There is one important issue. As the edges of the sprite are partially transparent (by smoothing them) we have to be careful with the order in which objects are drawn. Partially transparent (alpha blended) sprites are only blended with objects that have been drawn earlier. To get the required effect **alpha blended sprites must be drawn after all other objects are drawn**. This can easily be achieved by giving the plant objects (and others ones) a negative depth. The result of adding some plants looks like this:

In this way you can create many other objects in your game that you can place in the rooms. (But don't overdo it. This might make the game slow and hamper game play.)

Animated objects

We can also create animated objects in a similar way. There are just two important issues. First of all, we need a sprite that consists of several subimages for the animation. We must make sure that the actual sprite we use to represent the object has the same number of subimages, otherwise we cannot use the built-in `image_index` variable. Secondly, in the drawing event we must pick the texture corresponding to the correct subimage using the `image_index` variable. Here is a typical piece of code that could be executed in the draw event of some explosion object. As can be seen it also uses alpha settings.

```
{
  var ss,cc,tex;
  tex = sprite_get_texture(spr_explosion,image_index);
  ss = sin(obj_player.direction*pi/180);
  cc = cos(obj_player.direction*pi/180);
  draw_set_alpha(0.7);
  draw_set_color(c_white);
  d3d_draw_wall(x-8*ss,y-8*cc,2,x+8*ss,y+8*cc,18,ttt,1,1);
  draw_set_alpha(1);
}
```

The game we created so far can be found in the file `fps3.gmk`.

Shooting

Because this is supposed to be a shooter we better add the possibility to shoot. In this section we will only allow the shooting of barrels that we will place in the game world. In the next section we will add monsters.

Adding barrels

To shoot some barrels we need an image of a barrel. We actually need two, one when the barrel is standing still, and one when it is exploding. We borrow some images for this from Doom. These can be found on http://www.cslab.ece.ntua.gr/~phib/doom1.htm. These are both animated sprites. We also need two sprites to represent them in the room. To get a correct animation, it is important that these sprites have the same number of subimages as the actual animations. We create a barrel object and an exploding barrel object. Both we give the basic wall object as parent such that we cannot walk through them. The barrel object we place in the room at different locations. Because it has only two subimages we set the variable `image_speed` to 0.1 in the **Create** event to slow down the animation. In the draw event we draw the correct subimage on a facing wall as was indicated above.

When the barrel is destroyed (so in the **Destroy** event) we create an exploding barrel object at the same place. For this object we again set a slower animation speed. Because the barrel does not immediately explode we set an alarm clock such that the explosion sound is only played after a short while. We draw it in the same way as above, except that we use one trick. We slowly decrease the alpha value such that the explosion becomes more translucent over time. In the draw event we add the following line for this:

```
draw_set_alpha(1-0.05*image_index);
```

Finally, in the **Animation End** event we destroy the exploding barrel. The game can be found in the file `fps4.gmk`.

You could easily add some extra stuff, e.g. that nearby barrels explode as well and that the explosion hurt the player when he is nearby, but we did not implement such features in this simple game.

The gun

For the gun we will again use an animated sprite. This sprite has images for the stationary gun (subimage 0) and for the shooting and reloading of the gun. Normally we only draw subimage 0 (so we set the `image_speed` to 0) but when a shot is fired we must go once through the whole animation. We want to put the gun as an overlay on the game. To achieve this a few things are required. First of all, we must make sure that the gun object is drawn last. This is done by giving the gun object a depth of -100. Secondly, we must no longer use the perspective projection but change the projection in the normal orthographic projection. And finally we must temporarily switch off hidden surface removal. The code for the **Draw** event for the gun object looks as follows:

```
{
  d3d_set_projection_ortho(0,0,640,480,0);
  d3d_set_hidden(false);
  draw_sprite_ext(sprite_shotgun,-1,0,224,2,2,0,c_white,1);
  d3d_set_hidden(true);
}
```

Note that we scale the sprite with a factor 2 and both directions. Otherwise it is too small. As in the next step the **Draw** event of the player object sets the projection back to a perspective projection, we do not need to change it back here.

A similar technique can be used for all overlays you need. Below we will use it also for displaying the health of the player and you can for example use it to give instructions, display statistics, etc. You can also make the overlay partially translucent by changing the alpha value before drawing it.

The shooting

We now have the barrel to shoot and the gun, but we still do not have the shooting mechanism implemented. Because this is a shotgun, bullets go very fast. So it is not an option to create a bullet

object. Instead, at the moment the player presses the <Space> key, we must determine the object that is hit and, if it is a barrel, let it explode.

First of all we introduce a variable `can_shoot` in the gun object that indicates whether the player can shoot a bullet. We set it to true in the **Create** event of the gun object. When the player presses the <Space> key we check it. If the player can shoot we set it to false, and start the animation of the gun by changing the image speed. In the **Animation End** event we set `can_shoot` again to true and set both `image_index` and `image_speed` to 0. As a result, the player cannot shoot continuously.

To determine what the bullet hits we proceed as follows. We take small steps from the current position of the player in the current direction. For each position we check whether an object is hit. As all interesting objects are children of the basic wall object we only check whether there is such an instance at the location. If so, we check whether it is a barrel. If so we destroy the barrel. Whenever we hit a basic wall object we stop the loop because this will be the end of the path of the bullet. The code we place in the **Keyboard** event for the <Space> key looks as follows:

```
{
  var xx, yy, dx, dy, ii;
  xx = obj_player.x;
  yy = obj_player.y;
  dx = 4*cos(obj_player.direction*pi/180);
  dy = -4*sin(obj_player.direction*pi/180);
  repeat (100)
  {
    xx += dx;
    yy += dy;
    ii = instance_position(xx,yy,obj_wall_basic);
    if (ii == noone) continue;
    if (ii.object_index == obj_barrel)
      with (ii) instance_destroy();
    break;
  }
}
```

In the file `fps4.gmk` the result can be found. (The code is slightly different as we store the camera's position and the required sin and cos in global variables to speed up some calculations, but the idea is exactly the same.)

Adding Opponents

It is nice that we can shoot some barrels but the fun of that is soon gone. We will need some opponents. In this tutorial we will add just one type of monster but once you understand how this works, it is easy to add more. For the monster we need two animated sprites, one for when it is alive and one for when it

is dying. Both we took from the Doom site mentioned above. We also need a sprite to represent the monster in the room that is used for collision detection.

We make two objects, one for the monster that is alive and one for the monster that is dying. For both we set the animation speed correctly. When the monster dies we create a dying monster at the same position. This all works exactly the way as for the barrel. There is just one difference. At the end of the dying animation we do not destroy the dead monster but we keep it lying around, just showing the last subimage of the animation. To this end in the **Animation End** event we include a **Change Sprite** action and set the correct subimage (7) and set the speed to 0.

To be able to shoot the monsters we can again proceed in the same way as for the barrel. But we have to make a few changes. First of all, we do not want monsters to be children of our basic wall object. This will give problems when they walk around. So we create a new object `obj_monster_basic`. This will be the parent object for all monsters (even though we have just one now). A second change is that we want to be able to shoot through plants. So we also create a basic plant object. (The basic plant object will have the basic wall as parent because we do not want to be able to walk through the plants.) The shooting code is now adapted as follows:

```
{
  var xx, yy, dx, dy, ii;
  xx = obj_player.x;
  yy = obj_player.y;
  dx = 4*cos(obj_player.direction*pi/180);
  dy = -4*sin(obj_player.direction*pi/180);
  repeat (50)
  {
    xx += dx;
    yy += dy;
    ii = instance_position(xx,yy,obj_wall_basic);
    if (ii == noone)
    {
      ii = instance_position(xx,yy,obj_monster_basic);
      if (ii == noone) continue;
      with (ii) instance_destroy();
      break;
    }
    if object_is_ancestor(ii.object_index,obj_plant_basic) continue;
    if (ii.object_index == obj_barrel)
       with (ii) instance_destroy();
    break;
  }
}
```

It is largely the same as above. We first check whether a wall is hit. If not, we check whether a monster is hit and, if so, destroy it. If a wall was hit we check whether the instance is a type of plant and if so continue (so the bullet will fly through the plant).

(The function `object_is_ancestor(obj1,obj2)` returns whether `obj2` is an ancestor of `obj1`.) Finally, if we hit the barrel we will let it explode. You can find the game in the file `fps5.gmk`.

The final thing to do is to let the monster attack the player. This monster will not shoot but simply walk towards the player. When it hits the player the player should lose some health. A health mechanism is built into *Game Maker* as you should know. In the creation event of the player we set the health to 100. In the gun object, that also functions as our overlay, we also draw the health bar, slightly transparent, as follows:

```
draw_set_alpha(0.4);
draw_healthbar(5,460,100,475,health,c_black,c_red,c_lime,
                             0,true,true);
draw_set_alpha(1);
```

And in the **No More Health** event we simply restart the game (a bit boring; think of something better).

To let the monsters attack the player we put the following code in the **Begin Step** event of the monster object:

```
{
  if (point_distance(x,y,obj_player.x,obj_player.y) > 200)
    { speed = 0; exit; }
  if (collision_line(x,y, obj_player.x,obj_player.y,
                               obj_wall_basic,false,false))
    { speed = 0; exit; }
  if (point_distance(x,y, obj_player.x,obj_player.y) < 12)
  {
    speed = 0; health -= 2;
    if not sound_isplaying(snd_ow) sound_play(snd_ow);
    exit;
  }
  move_towards_point(obj_player.x,obj_player.y,1.4);
}
```

This does the following: If the distance from the monster to the player is large then we do nothing and simply set the speed to 0. If the monster cannot see the player (the line between them collides with a wall) we also do nothing. Only when the monster can see the player it starts moving towards it. If the distance is less than 12 we assume that it hits the player so we stop it and start subtracting health and play a sound. Otherwise we move towards the player position with a given speed. (Note that the first test could be removed but the function to test for a collision with a line is rather expensive so we only

want to do this if the monster is nearby.) To avoid walls, the monster can use the same sliding mechanism that we gave the player. (We actually put this in the basic monster because it is most likely the same for all monsters.)

Now we must place some monsters at vital places in our world and we are done. We have our first person shooter ready. The game can be found in the file `fps5.gmk`. Clearly, to make it more interesting you should create an exit to the next room, guarded by monsters, and then create a number of additional rooms. You should also create some additional monsters, and add some other objects in the world, like health packs. Also you could give monsters a health, requiring multiple shots, and you could maybe find better guns and ammunition. It is up to you to extent it. Some basic additional ideas though will be discussed below.

Other Issues

In this section we will discuss a few other issues that you might want to incorporate in your game.

Clipping

When walking through our 3-dimensional world you see only a small part of it. Unfortunately, the system does not know what part you see. So all objects (at least, those in front of you) are drawn. This is quite a waste of time and even on fast graphics cards this can lead to a slow frame-rate, especially when the room is large. All 3-dimensional games must consider this issue and avoid the drawing of unnecessary objects as much as possible.

One way to do this is to create small levels. You can then use special points such as teleporters to move from one level to the next. You can also do this in ways in which the user does not notice it. For example, you can make a winding corridor. You put this corridor in both levels. Halfway the corridor you jump

from one level to the next. As the player cannot see the rest of the level, he will not notice. (You should use persistent rooms for this to make it possible for the player to return to the previous place. You can achieve similar effect by using doors, or elevators.

Another way is to use clipping. When using clipping we only draw the objects that are close to the player. Whenever we draw something we first check whether the object is close enough. To make this fast we store the current position of the player (the camera) in two global variables `camx` and `camy`. Next, in the **Draw** event of every wall, plant, monster, etc. we add the following test:

```
if (point_distance(x,y,global.camx,global.camy) > 240) exit;
```

So if the distance to the camera is large we don't do anything. The value of 240 depends on your room design. Better make sure that are no places where you can look through long corridors. Also give the room a black background color such that if there is a place where you can look further it will simply be black, which fits nicely with our black fog. You can find the game with this and some of the other changes mentioned below in the file `fps6.gmk`. If you have a slow graphics card it should run much better than the previous one. (If you have a fast graphics card you might want to change the room speed to see the effect.)

When rooms get larger and have more objects in them the effect will become even clearer. You could even more cleverly try to determine which instances must be drawn. Commercial games use portal techniques in which they store for each area in the level which other areas are potential visible and only draw those. This can improve the speed even further.

Doors

It is a nice additional feature to put doors between rooms and corridors. This can create addition tension when playing the game. What is behind the door? Will there be monsters on the other side? Also you can force the player to follow the level in a particular way by providing buttons that open certain doors.

Here we will add some simple doors to our game: doors that slide sideways when hit by a bullet. Doors that slide up or down are also easy. Rotating doors are a bit more complicated because you must handle the situation in which there is a monster or player blocking the door.

To add the door we first need a texture for it. We add this as a background, just like a wall. We also need a sprite to represent it in our room design and for collision detection, like the other walls. We create two objects (that is easier although you could also do it with one). The first object is the closed door. It looks exactly the same as the horizontal wall object, except that in its **Destroy** event it creates an instance of the sliding wall object. Again we give it the basic wall as the parent to make sure the player and monsters cannot walk through it.

The second object is the sliding door. It looks like the closed door but in its **Create** event we give it a horizontal speed of 1. Also we set an alarm clock to 32 and in the **Alarm** event we set the speed to 0 (we

could actually also destroy the object). Finally, in its step event we adapt the `x1` and `x2` positions to make sure the texture is drawn at the correct place. This finishes the sliding wall.

We also must make sure the door can be opened. We decided that we open the door when the player shoots it. To this end, in the <Space> event of the overlay object (where the shooting is handled) we change the code at the end as follows:

```
if (ii.object_index == obj_barrel) || (ii.object_index == obj_door)
   with (ii) instance_destroy();
break;
```

That is all. Place a few strategic doors in the room. Make sure they can safely slide into the wall, so don't place them at the end of the walls. The result can be found in the file `fps6.gmk`. Of course, to make it all more interesting you should also add vertical sliding doors and maybe door that move left and right.

Floors and ceilings

In the game we created so far we made one large floor and one large ceiling. Even though this is easy it is also rather boring. This can easily be solved. You can create a number of different textures for different floors and ceilings and then you can make some different floor and ceiling objects that draw part of this. They are very simple objects as they have no further behavior (unless you want floors and ceilings to move). You place these at the appropriate positions in the room and you are done. Better make the floor and ceiling segments rather large, otherwise you need too many of them.

You can even give the ceiling pieces different heights. In this way you can create high and low areas in your world. Make sure though that the walls are also high enough. Giving different heights to the floor is also possible but more complicated. You must also adapt the height of the player and of the monsters and other objects, based on the floor they stand on. This requires some additional calculations. And you must decide, based on the height difference, whether the play can walk from one surface to the next.

One you have created all this it is also easy to create forbidden parts of the floor. For example, you can give a floor part a lava texture (you can even make it animated). To avoid the player to walk onto the lava the easiest solution is to place invisible wall segments around it.

We did not implement all these options in the example game but you should by now be able to experiment with it yourself.

Making a Game Out of It

Clearly, what we created so far is not really a game. But it gives you the basis to make one. You should add many more different monsters with different behavior, different textures to make the world look more interesting, different weapons, different sound effects, and various additional objects. Also careful level design is very important. Note that adding all of this might require too many textures to store at the same moment. But if you don't use all of them at once you can easily use the functions or actions to

replace sprites and backgrounds from a file. This has the additional advantage that for example the number of different wall objects you need is limited. In the next level you can simply use the same but, because you replace the textures, they look differently. You might want to consult the forums and other help at our website http://www.yoyogames.com for more ideas and techniques.