

Tutorial: Three-Dimensional Games

Copyright 2003, Mark Overmars

Last changed: April 18, 2003

Uses: version 5.0, advanced mode

Level: Intermediate

Real games are 3-dimensional, aren't they? Well, no. Actually, all games produce a 2-dimensional image on the screen of your computer. They might look 3-dimensional but that does not mean they are 3-dimensional. So what does it mean when a game claims to be 3-dimensional? This answer is difficult to answer. It is a combination of the way the game is created, the visual image the player sees, and the way the player controls the game.

Modern 3-dimensional games model the game world with 3-dimensional objects. They use the 3-dimensional graphics hardware to render such scenes. Given a viewpoint, that is, the position from which the player inspects the game world, the hardware creates the correct image taking into account aspects like lighting, hidden surface removal (objects behind other objects cannot be seen and should not be drawn) and atmospheric effects. This can with the current hardware lead to spectacular experiences but there is a price to pay for this. Modeling a 3-dimensional world with 3-dimensional objects is very difficult and time consuming. And creating the game logics, including aspects like behavior of the game entities, collision checking, camera movement, etc., is a very difficult task. As a result the production of a modern 3-dimensional game is a huge undertaking that involves a large team of specialized workers and typically costs millions of dollars. And still many of these games have errors, in particular in the handling of the camera and the character motions and collision checking.

Does this mean that 3-dimensional games are out of reach for you? Well, yes and no. *Game Maker* does not support 3-dimensional game worlds and has no possibilities for using 3-dimensional models. So don't expect to make a full 3-dimensional FPS with *Game Maker*. But actually all 3-dimensional looking games produced up to 1995 and many games since don't use 3-dimensional models. For example games like *Age of Empires* and the *Command and Conquer* series (except for the last) do not use a 3-dimensional game world. They are so-called isometric games. They use sprites to represent the game world and the characters and use some relatively simple tricks to give the user a feeling of a 3-dimensional world.



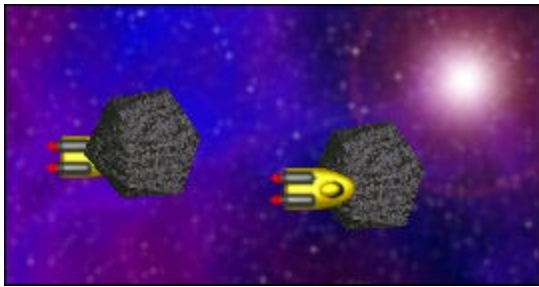
In this tutorial we will treat a number of techniques that can be used to create 3-dimensional looking games in *Game Maker*. Realize that these games are not the easiest to make. So you better be an experience *Game Maker* user before you start with this. Also we will need some code to make certain aspects work.

Fooling your brain

How do we make something look 3-dimensional? The basic idea is that we have to fool our brain into thinking that the image we see is 3-dimensions. So why do we think that an image we see is 3-dimensional. A number of aspects play a role here.

Objects that lie behind other objects are (partially) invisible

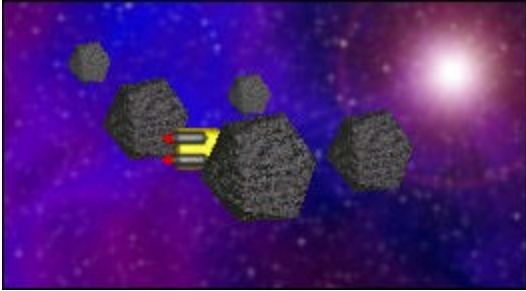
This is an important aspect. You normally cannot look through solid objects. Also, when for example a character is partially hidden by a tree the viewer immediately knows that the character is behind the tree. In this way the viewer gets a better insight in the 3-dimensional relationship between the objects. So make a world look 3-dimensional we should only display the objects that are actually visible. This process is normally called *hidden-surface removal*.



3-dimensional graphics hardware does hidden-surface removal automatically but because *Game Maker* deals with a 2-dimensional world we have to do this ourselves. The way to achieve this is to draw the sprites that correspond to the objects in the correct order. The object that lies furthest away is drawn first. The next object is drawn on top of it, and so on. Because closer objects are drawn on top of object behind them, they will (partially) hide the other objects. In *Game Maker* we can achieve this by setting the depth for the different object instances. Instances with larger depth are drawn before instances with a smaller depth.

Objects that lie further away appear smaller

Clearly, the further away objects are the smaller they appear. This gives a very strong notion of distance and, hence, a better 3-dimensional feeling.



This can be achieved by changing the size of the sprites depending on the distance. You can either make sprites with different sizes or draw the sprites with a scale factor. (We will discuss this further below.) When an object moves towards the viewer it should become larger. This gives a very strong feeling of moving through a 3-dimensional world. Be careful though. When the scaling is wrong it gives the exact opposite effect.

Objects that lie further away appear vaguer

When objects lie further away you cannot see them so well anymore. This also works the other way around. When an image appears vague we interpret it as lying further away. Hence, adding a feeling of mist and blurring images that lie further away a bit gives a good sense of distance. This can be achieved by using different sprites for the same instance, depending on its depth.

Objects that lie further away appear to move slower

An object that lies far away takes long to traverse from the left to the right through your view while an object that lies close by moves much faster through your view. You can use this to give the viewer a feeling about the depth of the objects. One particular aspect of this is parallax scrolling that we will describe below. Also, objects that move away from you or towards you hardly change position on the screen. So normally the speed in the vertical direction on the screen (which often corresponds to movement towards you or away from you) should be slower than horizontal motion.

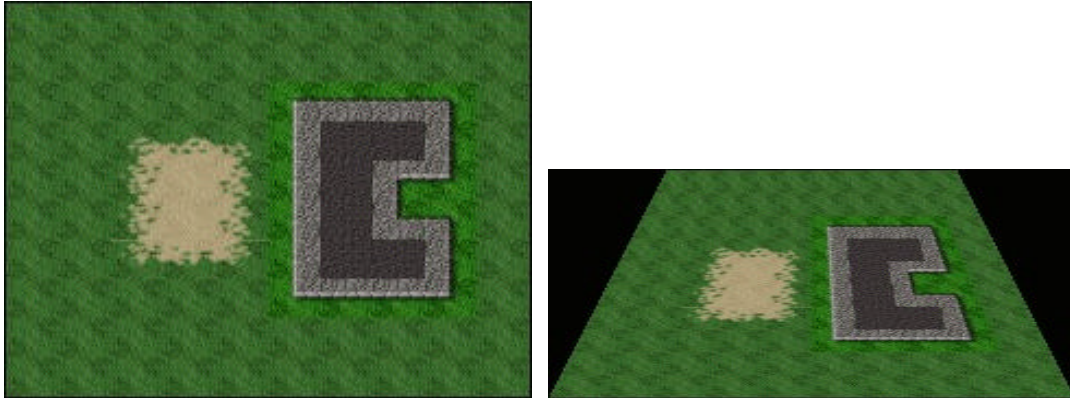
Objects tend to have a shadow

Objects cast shadows. It is difficult to compute shadows on other objects but it is relatively easy to have shadows on the floor. And even if there is no shadow it is good to have a little dark spot around the bottom of an object. This gives the impression that the object stands on the floor. Make sure the shadows are consistent, that is, all objects have shadows (or none) and that they have the same size and are in the same direction. For example, in the picture below, for the left character it is unclear where she stands. The middle character clearly stands on the ground while the right one jumps above it.

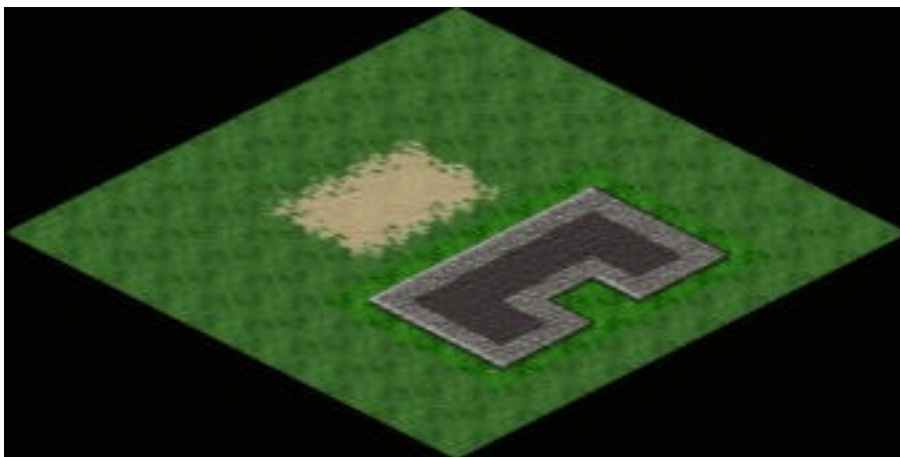


Parallel lines appear to meet in a point

As you probably all know, parallel lines in a 3-dimensional world meet in the distance in a point. For example, the two sides of a road meet in the distance. Stated more precisely, our view generates a perspective projection. This in some sense deforms the objects depending on their depth but this deformation is exactly what gives the 3-dimensional feeling. Consider for example the following two images:



The left image shows a projection without perspective. The vertical lines are parallel. In the right image we use a perspective transformation and as a result the lines will meet somewhere in the distance. It gives a much better 3-dimensional feeling. Perspective projections are important for 3-dimensional games but unfortunately they are not easy to achieve with sprites. So sprite based games normally don't use perspective transformations. To limit the effect of this they normally use a projection under an angle of 45 degrees. That is, they don't look straight from the front but a bit from the side. As a result the effect of parallel lines is less prominent and, hence, the viewer is less distracted by them. This is the basis of isometric games. The following image shows the same scene but now under an isometric projection:



If you look careful the lines are indeed still parallel (which is wrong) but still it looks a lot more 3-dimensional than the left image above.

Creating isometric game is possible with *Game Maker* and we will show how to do this below.

Parallax scrolling

As was indicated above, object that lie far away move slower than objects that lie close by. In this section we will use that fact, but only dealing with side scrolling backgrounds and static objects.

As you should know you can add backgrounds to your rooms in *Game Maker*. Very often a room has just one background but it is possible to have multiple backgrounds. Clearly, to see those backgrounds they need to be partially transparent or cover just part of the room. By giving the lower backgrounds a slower speed than the higher backgrounds you give the player the impression that those backgrounds are further away. In this section we will create a small example of a car driving sideways on a road. We don't aim for graphic quality; we just want to demonstrate the effect.

First of all we use two backgrounds. The first one indicates some mountains at the distance:



It will be placed at the top of the room and scrolls left with a very small speed (1). We tile it horizontally but not vertically. The second background represents the road. It is very small, is placed at the bottom of the room and moves much faster (4). Again we tile it only horizontal. We place a car on the road but it does not need to move. Because the road moves under it we already get an illusion of motion. Because the road moves faster than the mountains you get the feeling that the mountains are far away.

To add a bit more depth to this demo we are going to include some trees. We use threes of three different sizes:



For each of them we make an object. To make sure that the small tree is behind the middle tree which again must be behind the large tree we give the objects different

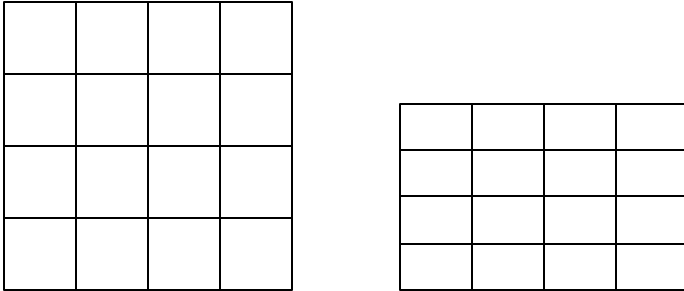
depths. (Below we will talk more about depths.) The large tree will be placed close to the road and we give it a speed of 4, like the road. The middle tree will be placed a bit further away and we give it a speed of 3. Finally, the small tree we give a speed of 2. We place a couple of them at appropriate places in the room and make sure that they reappear at the right if they disappear to the left (with a bit of randomness to make it all not too regular). The result looks as follows:



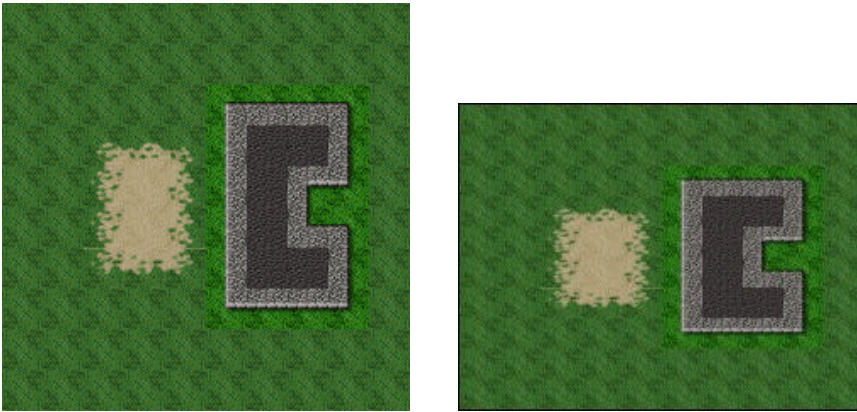
You can find the demo in the file `parallax.gmd`. Clearly it is not a game. To turn it into a game you must give the player the opportunity to steer the car and e.g. add other traffic to avoid.

Parallel projections

Even though true 3-dimensional games should use perspective projects we will first deal with parallel projections. In a parallel projection the size of the objects stays the same independent of the distance. Also parallel lines remain parallel. This makes it a lot easier to use sprites because sprites have a fixed size. Clearly this gives a distortion of the view. To limit this distortion we will make sure that the player cannot see objects that are far away. So he will not really notice the error in image sizes. We achieve this by looking down on the world under an angle (normally about 45 degrees are used). So a world that consists of squares will not look like rectangles:



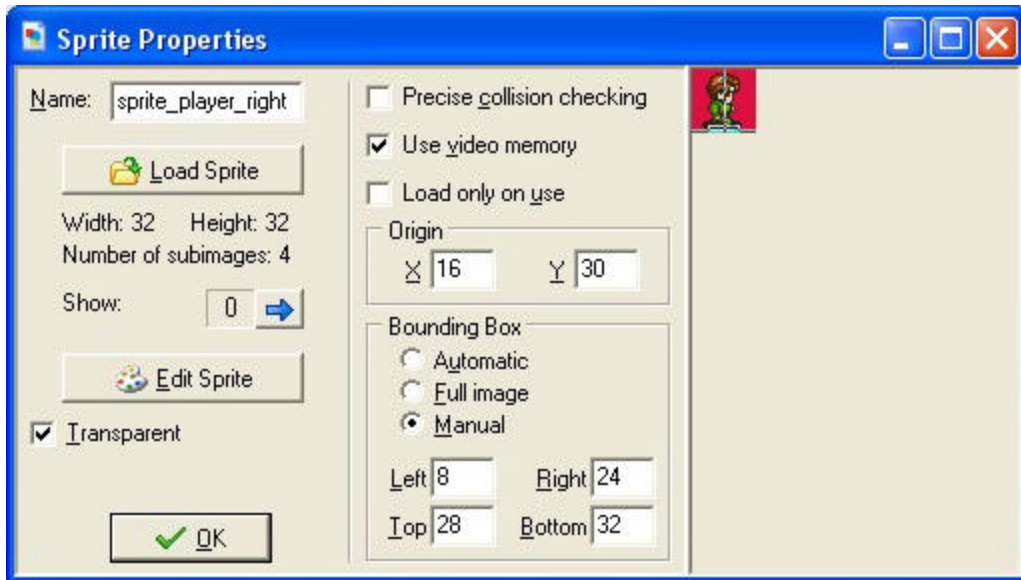
For a background image this will look as follows:



It is easy to create such an image. We simply scale the image in the vertical direction only.

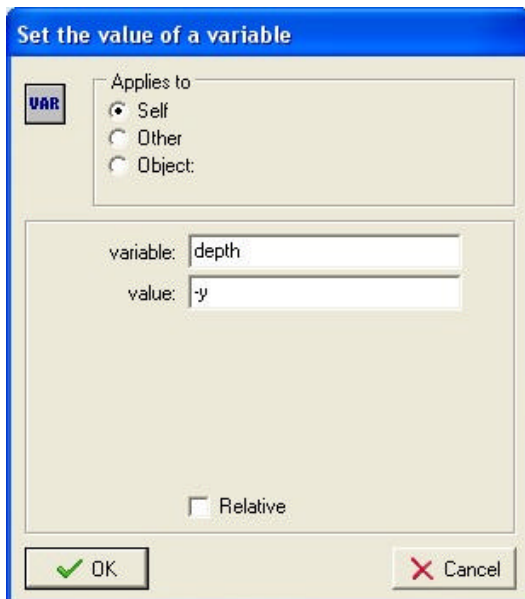
We will demonstrate this approach with a small demo in which the player controls a character that can walk through a world with some trees. There are three things we must handle in an application like this: We must make sure that hidden-surface removal is done correctly. So when the character is behind a tree it should be drawn before the tree is drawn to make it partially invisible. Secondly, we have to correctly deal with the speed of the character. Moving back and forth must be slower than moving left and right. Finally, and the most difficult, we must handle the collisions correctly.

Let us start with the hidden-surface removal. As you should know every sprite has an origin. Default this is the top-left corner of the sprite image. When we place an instance of an object at a particular position we actually put the corresponding sprite with its origin at that position. When we deal with a 3-dimensional world it is easiest to take as the origin of the objects the position where they stand on the ground. So for all sprites we set the origin to that position. So for example for the character that walks through the forest we set the sprite as follows:



Note that we also set a small manual bounding box. We will use that below. Our character will be represented by 4 different sprites depending on the direction in which he walks. Clearly for each we should set the same origin (and the same bounding box).

Now what should the depth be? In most situations the order in which the objects must be drawn is the same as the order in which they stand on the ground. So by setting the origin as above the y-coordinate of the instance determines the order. Instance with a low y-coordinate, that are close to the top of the window, must be draw before the ones with a higher y-coordinate. So a higher y-coordinate must correspond to a smaller depth. This can be achieved by setting the depth to $-y$ using the following action:



For the static objects, e.g. the trees in our example, we need to do this just once in the creation event. For the dynamic objects, the character in our example, we must do this in every step. You best use the end step event for this to make sure that all other events are processed before the depth is adapted.

Secondly we must deal with the motion of the character. We can do this in the usual way using the arrow keys. In each of the arrow keys event we set the motion in the correct direction and set the correct sprite for the instance. Also, to get an animation of the sprite while moving and no animation when the character is not moving we use the variable `image_single`. This variable indicates which single image in the animation must be shown. Setting it to -1 plays the animation. So when ever the user presses an arrow key we set this variable to -1 and when he releases a key or when there is a collision we set it to 0 to just show the first image. This all has nothing to do with 3-dimensional games. It is the usual way to handle this in all games. The only thing we do have to take care of in a 3-dimensional game is the difference in speed. For the horizontal motion we use a speed of 3 while for the vertical motion we use a lower speed of 2. You must play with these numbers to get the best effect.

Finally we must deal with collisions. Normally a collision occurs when the sprites partially overlap. This is fine for 2-dimensional game but not for 3-dimensional games. When the character stands in front of a tree the sprites overlap but there is no collision. So we need another area that must define the collision. Normally this is the area of the instance on the ground. Only when the feet of the character overlap with the bottom of the trunk of the tree there is a real collision. The easy way to achieve this is to define a very small bounding box for all the sprites at the bottom and not use precise collision checking. A collision is only reported if the bounding boxes partially overlap, which is exactly what we want.

For more complex shapes a small rectangle at the bottom might not indicate the correct collision area. In that case we can use another feature of *Game Maker*. An object can have a different mask than the sprite used to represent the instance. So we can make a second sprite that has the shape of the required collision areas and use that as collision mask. We will use this approach below when creating an isometric game.

We now have all the ingredients for the 3-dimensional game we want to make. To make it all look a bit nicer we use a tile set of background tiles. We nicely scale them as indicated above to obtain 48x24 tiles. With this we fill the background. We also add some walls as tiles to the background. To create collisions with these we use a standard technique in which we define an invisible solid wall object and place instances of this object on top of the walls in the background. So the character will stop because it collides with these invisible instances while the player only sees the nice background behind them.

Finally we must make the room. We will use a view in the room such that the player sees only part of the world. To enhance the notion of depth we make the view 400x200, so again with a scaling with a factor 2 in the vertical direction. In many games this

proportion is achieved by adding some information panel below the scrolling world. (This is the reason why you better put the information panel at the bottom of the screen rather than at the side for this type of games.) The final demo can be found in the file `parallel.gmd` and looks as follows.



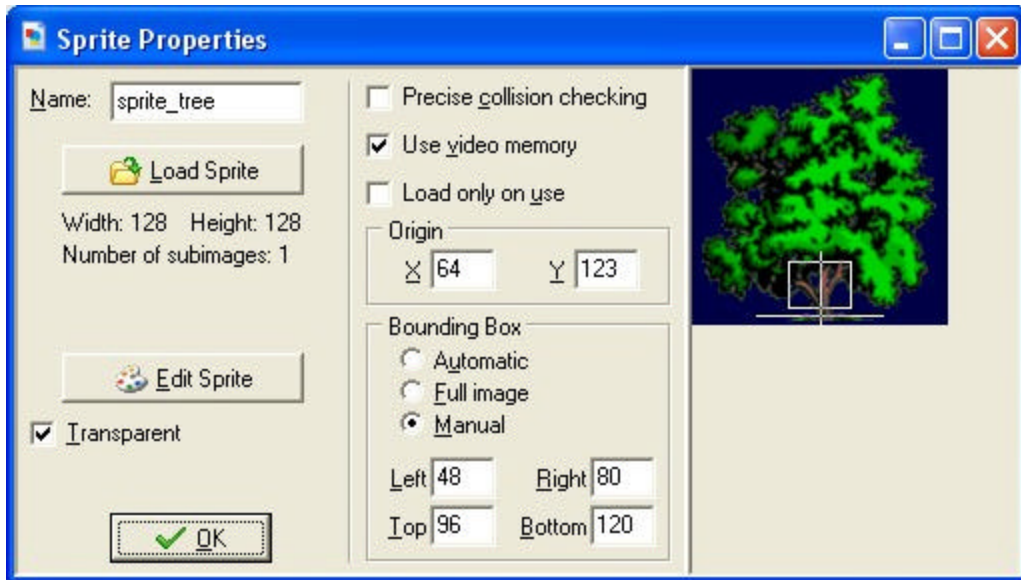
Scaling images

In the previous section we used a parallel projection. If we keep the view high enough from the ground in a fixed orientation, and we show just a small part of the world this is good enough. But if we want to get a first person view or a view closer to the ground we need perspective projection because in that case the size of objects must be smaller when the objects are further away. But how do we achieve that? In the first parallax scrolling example we used three different tree sizes. This is fine when the trees stay at fixed distances but in a real 3D game the player might approach them and, hence, the size should change continuously. We could create a huge number of different sprites, one for each size, but this will use a lot of memory. You might though want to do this for really important objects because it gives the best quality. For the other instances we can use the scaling of sprites. In this section we will show how to do this using a simple example in which the player must avoid trees while moving through a forest.

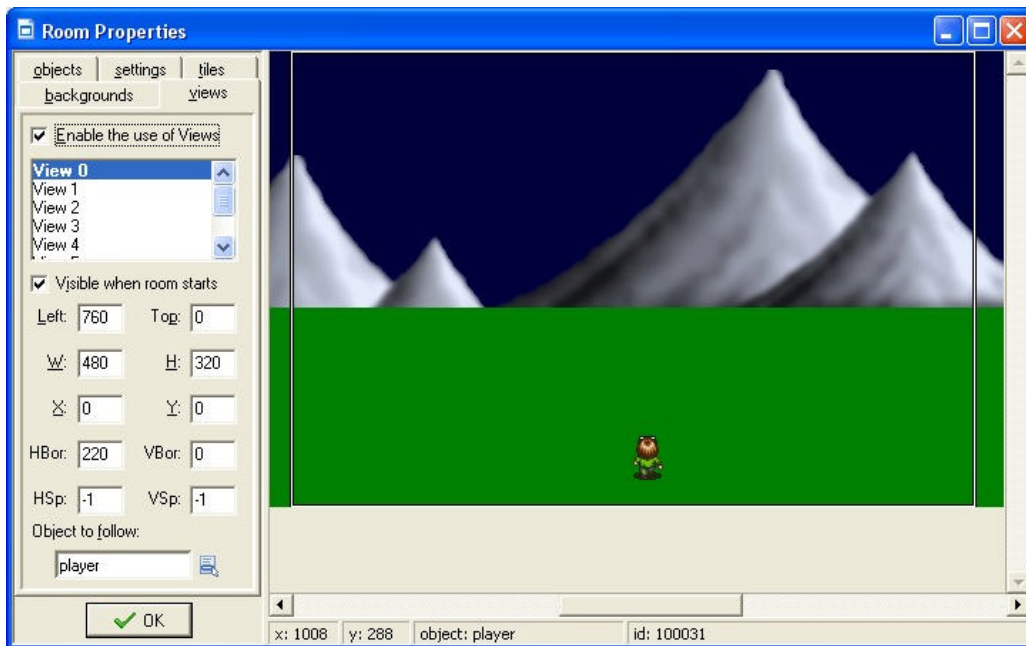
So we first need a sprite for the tree. We use one sprite here.

WARNING: For some graphics cards, sprite sizes must be a multiple of 4 to scale correctly. To be on the safe side you are recommended to give your sprite always sizes that are a multiple of 16! Also some graphics cards require the sprite to use video memory to be able to scale it!

For the sprite we manually set a small bounding box at the trunk of the tree as above. This is the only part where we are interested with collisions with the player. Also we place the origin at the bottom of the tree. So the tree sprite properties will look as follows:

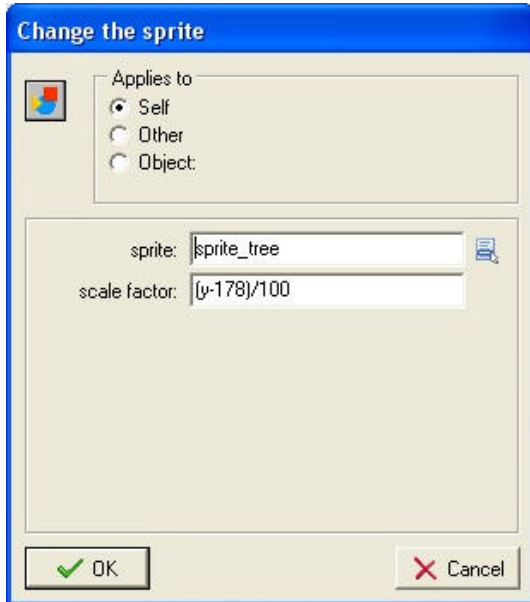


We will again use the mountains for the background, but this time non-scrolling. This time we use a wide room of width 2000. We place the main character, for which we also need a sprite, in the middle at the bottom of the room. We will use a view of size 480x320 and keep the character basically in the middle by setting the view borders and tell it to follow the character as follows:



The character will move left and right when the player presses the left or right arrow key. Clearly we must avoid it to move too far to the left or too far to the right. As scoring we use the time the player survives. So in the step event of the player we add 1 to the score. When the player collides with a tree we show the high-score list and restart the game.

What remains is the tree. There are three aspects we have to deal with: the size, the position on the screen, and the depth. Trees will appear at y-position 180. In the step event we use the action to set a sprite. We always use the same sprite but in this action we can also indicate the scale. We use a scale factor that is close to 0 when the y-position is 180 and increases when the tree moves down, as follows:



You can play a bit with the scale factor if you like (make sure it is always positive).

The second thing is the depth. Again in the step event we set the depth to $-y$. This means that trees closer to the bottom of the screen (with larger y-coordinate and closer to the viewer) have a smaller depth and are drawn on top of the others, as we like. We also need to give the player an appropriate depth to keep it at the right position among the trees.

Finally we must control the movement. This is harder than it may seem. We can let the trees move down with constant speed but, as you will notice, this will visually slow them down (this is because object close by seem to move faster than objects far away). So we must increase the vertical speed. Also, object move towards the sides if they get closer (because parallel lines meet in a point at infinity). So, based on the position of the tree relative to the middle of the view, we must adapt its horizontal speed. We do this using the following piece of code in the step event:

```
Execute a piece of code
[Icons] Applies To: Self Other Object

{
  var xoff,yoff;
  xoff = x - (view_left + view_width/2); // offset from the view center
  yoff = y - ystart; // offset from start position
  x += xoff * 0.015; // Play with these values
  y += 0.2 + yoff * 0.02;
}

5/7: 63 INS
```

(Note the use of local variables `xoff` and `yoff`.) The numbers have been determined experimentally. You might want to change them a bit.

To finish the game we a few other things, like destroying trees when they disappear below the screen and adding a controller object that controls the appearance of trees (more when the score gets higher). You can find the demo game in the file `scaling.gmd`. You can use it as a basis for your own games. Please realize that scaling sprites can be slow, especially on older machines. So use this with care.

You can actually use the same mechanism to create a full first person shooter game. Here the player object has a position and an orientation in the game world. For all other instances in the world you must compute their position relative to the direction the player is looking in, and their distance. The distance determines the drawing order and the size as above. (To avoid drawing too many instances you normally only draw instances if they are close enough.) I will not go into detail here.

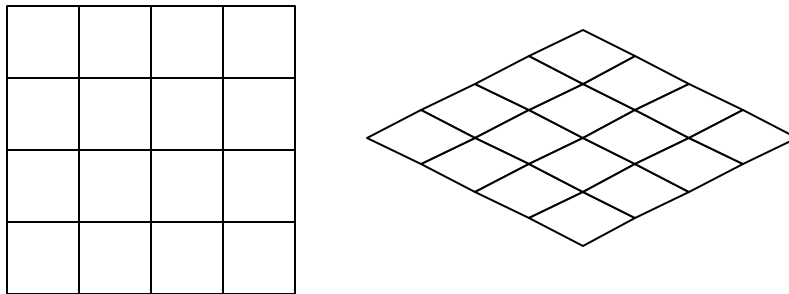
Isometric games

We will finally look at a very popular type of game and create an isometric game. Many of the currently produced games are isometric. Examples are Age of Empires, Diablo, Command and Conquer (except the last on), SimCity, and many other strategy and simulation games (see e.g the image below).

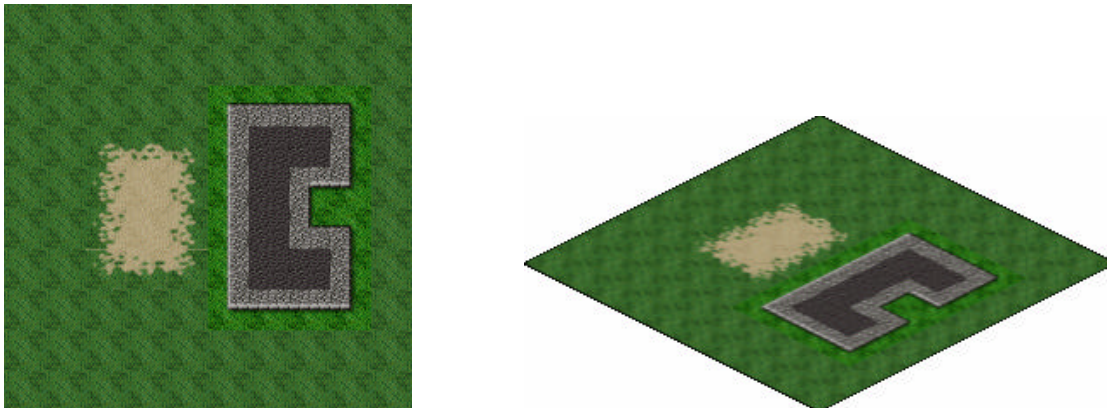


Such games are in principle possible with *Game Maker* (but realize that these commercial games took years to create by dedicated teams of workers).

The principle of an isometric game is that the 3D world is viewed under a fixed 45-degree angle. Again parallel projection is used, that is, there is no perspective, and so objects in the distance will not become smaller. Assume the world consists of square cells as in the left of the picture below. An isometric view will look like the right picture. Each square has become a diamond shape. To make lining up cells easy we typically use a 2x1 diamond, e.g. 32x16 pixels.



Doing the same with a background image we get the following pair of images:



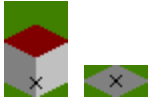
Such background images can be made rather easily with *Game Maker*. First make a tiled background. Next edit the background, enlarge the canvas, rotate it by -45 degrees, and stretch (or actually shrink) it vertically to 50%.

The advantage of an isometric project over a normal projection as we did before is that our mind immediately interprets the diamond as squares viewed under an angle. This gives a 3-dimensional image. Clearly we will still get perspective distortions because the size of objects does not change with the distance. This perspective distortion can again be limited by showing only a small part of the world. So most isometric games use a limiting view on the world.

Once you have a nice background we need to place objects on it. These objects should obviously have a 3D look from the correct 45 degree angle. Creating such isometric

objects is not easy and is normally the most time consuming task when creating an isometric game.

Each object has a "shadow" on the floor. This shadow is crucial. It will be used for collision detection, or stated differently, it will be used as collision mask. So for an object you will need to make two sprites: the image of the actual object and the shadow mask. It is best to always make the origin of the sprite and the mask the center of the shadow. For example, below you see an image of a wall element we are going to use in our isometric maze, and the corresponding shadow mask (the cross is the origin).



The mask in this case has precisely the size of one tile in the image.

By doing collision detection on the shadows the sprites are allowed to overlap, as long as their shadows do not overlap. This is precisely what we want. When e.g. a ball runs among the blocks the sprites can overlap. But a collision with a wall occurs once the shadow of the ball hits the shadow of the wall.

As in the parallel case we set the depth of each instance to $-y$. In this way the instances are drawn in the correct order.

There is a problem though. When creating your rooms the sprites are not drawn in this order. This might make it difficult to create the right scenes. One advice is to always start at the top and work your way down. Also, in the objects tab there is a button to sort the instances by y -coordinate, which is just what you want.

A final word about motion. Because of the projection, horizontal motion should be twice as fast as vertical motion. Also, when e.g. using arrow keys to steer the motion, it is common to let them correspond to diagonal motion (along the sides of the tiles) rather than horizontal and vertical motion.

A maze example

As an example we are going to make a simple isometric maze game. It is not very sophisticated but it will show you the basic ideas. The game can be found in the file `isometric.gmd`.

As a first step you better design the maze on paper. It will be very difficult to do this while creating the room. We next create a background floor for the maze that fills the whole area covered by the maze.

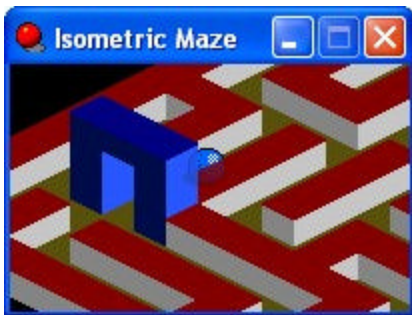
Next we create the sprites. As indicated above we need two sprites for each object: the actual image and the mask used for collision checking. (Actually, for many objects we can use the same mask.) Make sure that each sprite is nicely drawn on the floor and that the origin is in the middle of the shadow mask.

To make things more interesting I also added a gate. The gate consists of three parts. The left and right poles are simply a bit higher walls in a different color. They use the same mask as the normal walls and are exactly the same (so we gave them the wall object as parent to give them the same behavior). The middle piece of the gate is different. It consists of a block higher in the air. It does not need a mask because the ball can move under it. But we need to make sure that its origin is still in the center of its shadow (otherwise it would not be drawn at the correct depth).

For each fixed element we set its depth in the creation event. For the ball we set it in the end step event. For the rest the game is largely the same as a simple maze game. When you press an arrow key we move the ball a little bit in the correct diagonal direction, assuming the new position is collision free. As indicated above the horizontal distance is twice as large as the vertical distance. When the ball (that is, its shadow) collides with the flag (that is, again its shadow), the maze has been solved.

Creating the rooms is a bit more work this time. You have to place all wall elements correctly. Also, as indicated above, best work your way from the top downwards. Use the button in the objects tab to sort the instances by y-coordinate if you made a mistake. Use a 16x8 grid to align the instances correctly. You might want to uncheck the box to delete underlying instances because this makes it easier to place the wall elements.

There is one more trick added to the demo. When you hold the <Space> key you can see the ball through the wall. This is useful when the moving object can completely disappear behind other objects. This is done as follows. A new object called `transparent_ball` is created. It is given a depth of `-100000`, that is, in front of everything else. In its drawing event we place a little piece of code that draws the ball sprite partially transparent at the place of the ball. This has the effect as if we can partially see the ball through the walls.



Try to play with the demo. Add some nicer looking wall and gates, add some levels, etc.

More advanced isometric games

Isometric games are very well suited for many types of games. The major complication though is that you need nice looking tiles. There are though lots of resources for this on the web, including some programs that help you. Use e.g. Google and type as keywords "isometric tiles" and you will find many interesting pages. For the rest, as I hopefully indicated above, it is easier than it might seem. The only two things you have to keep in

mind are the collision masks and setting the depth correctly. Here are a few more things to keep in mind:

- When you cannot walk behind certain objects (like mountains or woods) you best draw them on the background. Next you can use some invisible objects to avoid that the moving objects can stand on top of them.
- When an object spans different tiles (like e.g. a gate) split it into several pieces to avoid problems with the depth.
- Remember to always put the origin at the center of the vertical projection of the object. So e.g. a flying bird will have its center quite a distance below it. Note that it is not necessary to make the sprite this long. The center can lie outside the sprite.
- In rare cases, the depth relation between two objects is more complicated than I indicated above. In this case you will have to set it in a more complicated way but this can normally be done using a little piece of code.

Conclusions

I hope that this tutorial has convinced you that it is possible to make many different types of games with *Game Maker* that have a clear 3-dimensional look. As always, only your imagination is the limit.