

Introduction

With this set of tutorials I hope to create a complete 3d video game. I intend to show the process I go through when creating the game.

I find the best way to learn is to work things out for yourself. If you can work out and understand my code then you will have learnt something (I hope). I will not be explaining every single command/ line of code. If you want to know what a particular command does then look it up in the DarkBASIC Pro help files or the basic tutorials that can also be found on this website.

What game is it going to be then?

I spent quite a time thinking about the sort of game I wanted to make for this tutorial. I wanted something fairly simple that I knew I would be able to explain, yet it also had to be interesting enough for everyone to want to at least have a play.

I considered writing a tank combat game, but decided I didn't fancy writing the AI for it. This evolved to a simple capture the flag game which is what I have decided to stay with.

The game (Limit Rush) will be an arena based chase game. The hovercraft (the controllable vehicles) have to chase a sequence of coloured lights around a playing field. The lights will be stationary, and will turn on and off randomly. The idea is that the first person to touch 10 lights is the winner. A player touching a light will turn it off, and a new light will ignite starting the chase to the next 'flag'.

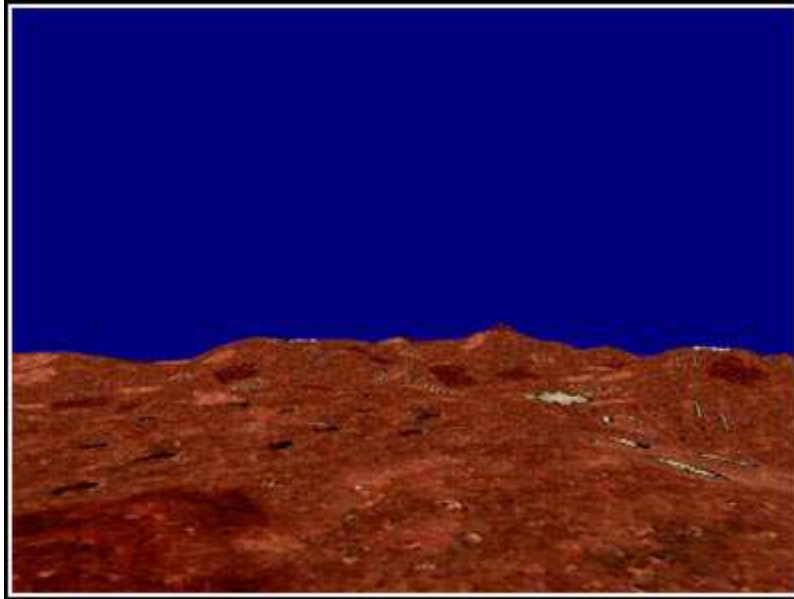
Why did I choose this game?

To start with I thought about the technical aspects of the game. I wanted to stick to something simple that I knew I could complete. I always think it is best if you know what you can and cannot do.

- With this game I can use Matrices for my level.
- The arena will be square which will mean nice simple collision.
- Hovercraft vehicle means I don't have to worry about wheels turning - also they are not used a great deal so it will make things a little more interesting.
- The AI will be fairly straightforward. It's just a case of pointing the vehicle in the right direction then moving

On with the tutorials then...

Lesson 1 - The Terrain



To make the terrain for this game I need a level editor. I could code my own or I could use one of the ones that is freely available on the Internet. To save time I have decided to use one of DarkBASIC Pros best know matrix editor MatEdit.

Using MatEdit is a fairly simple and painless exercise. The controls are a little clunky but you get used to them fairly quickly. The amount of features though, should be enough for everyone.

To save you time I have created the level. I do however recommend you have a play with MatEdit to see how it works and what you can do.

When creating the level I made a note of the size of the terrain. Doing this means that I can use these sizes in my 3d program to make an environment that fits exactly. I want to be able to create stands that surround the level to make it have more of an arena look.

Loading the terrain

Before you can load a MatEdit matrix you have to set up a few things. First you have to copy all of the variables into your program. This is because Arrays can not be stored in included files. You also have to include the MatEdit include file. This gives you access to all for the commands within MatEdit. There are more commands in this file than just ones that load the terrain and we will be using some of these later.

You should save this file in a folder all of its own. Then you should place the 'LoadMatrix.dbf' file in the same folder. This file, along with the array file, can be found in the Support folder from the MatEdit installation.

Once all this is done you should end up with a file that looks like this.

NOTE : This will not actually do anything

```
`include the MatEdit LoadMatrix files
#include "LoadMatrix.dbi"

`declare the MatEdit variables
Dim BigMatrix(600,600,1)
Dim StartLoc_X(1): Dim StartLoc_Z(1):Dim Info(2)
Dim TArrayX(1): Dim TArrayZ(1): Dim FKey(10,1)
Dim ColData(100): Dim ZoneData(100): Dim Tiles(500,500)
Dim OverTexture$(100): Dim OverName$(20): Dim ReplaceTex(100)
Dim MOffsetX(25): Dim MOffsetY(25)
Dim MWire(20): Dim MGhost(20): Dim Lock(20)
Dim MatX$(20): Dim MatY$(20): Dim MatZ$(20)
Dim MatWidth$(20): Dim MatHeight$(20)
Dim TilesX(20): Dim TilesZ(20)
Dim MatHi$(20): Dim MatLo$(20)
```

After copying the Matrix data from the Projects folder (where MatEdit saves it) to your game folder you can start thinking about loading the matrix. To do this you use the LoadMatrix function that is in the include file.

LoadMatrix(*MatrixName*\$, id)

To load the matrix all you have to do is specify the name of the matrix and the number you want to give it. As there is only going to be one matrix in this game I have decided to number the matrix '1'.

Notice the map name does not need a file extension on the end.

```
LoadMatrix("map",1)
```

Lets look at the matrix then...

So that we can have a look at the matrix I have added some simple movement code to the program. I have also added simple setup instructions such as turning the sync on, and setting the sync rate.

The 'do loop' is added to control the game flow. This loop makes the code inside it repeat until the user quits the game, or a piece of code tells it to stop looping.

The sync is placed at the end of the loop. This makes the screen refresh after the objects (camera, vehicles etc.) have been positioned and rotated. It updates all of the lighting effects, and works out what should be drawn to the screen.

I have used bits of the camera tutorial to allow me to move around the terrain. This will change for the finished game but it lets us see the world and gives you something to play with.

The file now looks like this:

```
`-----
`INCLUDES
`-----

`include the MatEdit LoadMatrix files
#include "LoadMatrix.dbi"

`-----
`ARRAYS
`-----

`declare the MatEdit variables
Dim BigMatrix(600,600,1)
Dim StartLoc_X(1): Dim StartLoc_Z(1):Dim Info(2)
Dim TArrayX(1): Dim TArrayZ(1): Dim FKey(10,1)
Dim ColData(100): Dim ZoneData(100): Dim Tiles(500,500)
Dim OverTexture$(100): Dim OverName$(20): Dim ReplaceTex(100)
Dim MOffsetX(25): Dim MOffsetY(25)
Dim MWire(20): Dim MGhost(20): Dim Lock(20)
Dim MatX$(20): Dim MatY$(20): Dim MatZ$(20)
Dim MatWidth$(20): Dim MatHeight$(20)
Dim TilesX(20): Dim TilesZ(20)
Dim MatHi$(20): Dim MatLo$(20)

`set up the program
sync on
sync rate 40
hide mouse

`load the matrix
LoadMatrix("map",1)

`-----
`MAIN LOOP
`-----

main:
do
  `the following is temporary. There will be more but it will made later
  `get keyboard input for movement
  if upkey()=1 then move camera 4
  if downkey()=1 then move camera -4
  if leftkey()=1 then yrotate camera wrapvalue(camera angle y()-4)
  if rightkey()=1 then yrotate camera wrapvalue(camera angle y()+4)

  `sort out the camera height
  x#=camera position x()
  z#=camera position z()
  y#=get ground height(1,x#,z#)+10

  `position the camera
  position camera x#,y#,z#

  `update the screen
  sync
loop
```

Lesson 2 – Loading The Environment



This lesson details the loading the game environment.

To load the environment the 'load object command' was used. Creating a game that had different objects in each section/ level then it is suggested to develop a custom loading system that loads data from a text file. This would make things a lot easier in the long run as it would basically take care of itself. However this game does not need something this complicated so the game environment object are numbered starting form 100.

```
`temporary load level info  
load object "media/arena.x",100  
load object "media/arena_light.x",101
```

Notice two separate objects are loaded in the section. One of these is the level itself and the other contains the light beams for the spotlights. Both of these models were created in a modeling package then exported separately. When creating models try to keep the amount of limbs (individual objects) to a minimum. This helps the game to run a lot faster.

```
`temporary load level info  
load object "media/arena.x",100  
load object "media/arena_light.x",101
```

Adding the previous two sections of code to the tutorial is enough to load the environment. However, the models are too small. Actually you can barely see them. So, the following sections details how to scale the game environment to fit the terrain.

Since we are dealing with a 3D object, the object requires scaling on all three axis's (x,y,z). The "scale object" command is used to scale the object to the desired size. When examining both the "arena" and the "arena_light" objects closely, we notice that they are not the same size. In actuality, the "arena_light" object is 46% smaller on the X-axis and Z-axis and 55% smaller on the Y-axis than the "arena" object. To compensate for the size discrepancy of the two models three variables are created to be used with the "scale object" command:

ArenaXYZ_SF – Scaling factor for all three axis's for the "arena" model

LightXZ_SF# – Scaling factor used for the X-axis and Z-axis for "arena_light" model

LightY_SF# – Scaling factor used for the Y-axis for the "arena_light" model

```
\
-----
\ Initialize Arena Scaling Variables
\
-----
ArenaXYZ_SF = 15000
LightXZ_SF# = .46
LightY_SF# = .55
```

Because we know the size ratio between the two models, we can scale both models simultaneously using the scale factors. Setting up the scale object command in this manner allows us to adjust the game environment by just changing one variable (ArenaXYZ_SF).

```
\scale the arena
scale object 100, ArenaXYZ_SF, ArenaXYZ_SF, ArenaXYZ_SF
scale object 101, ArenaXYZ_SF*LightXZ_SF#, ArenaXYZ_SF*LightY_SF#, ArenaXYZ_SF*LightXZ_SF#
```

Once the arena is loaded, it needs to be aligned with the terrain. The "position object" command allows for just this task.

```
\position arena
position object 100,247,189,247
position object 101,247,189,247
```

Lastly, set the fog and lighting properties. These can change each time the level increases. This functionality will be added later. For now lets see how the game looks and set everything up. So we now have code that looks like this:

```
\
-----
\ Limit Rush
\ Lesson 02
\
-----
\ http://www.binarymoon.co.uk
\ Ben aka Mop
\
-----

\
\ INCLUDES
\
-----
\ include the MatEdit LoadMatrix files
#include "LoadMatrix.db"
```

```

\-----
\ARRAYS
\-----
\declare the MatEdit variables
Dim BigMatrix(600,600,1)
Dim StartLoc_X(1): Dim StartLoc_Z(1):Dim Info(2)
Dim TArrayX(1): Dim TArrayZ(1): Dim FKey(10,1)
Dim ColData(100): Dim ZoneData(100): Dim Tiles(500,500)
Dim OverTexture$(100): Dim OverName$(20): Dim ReplaceTex(100)
Dim MOffsetX(25): Dim MOffsetY(25)
Dim MWire(20): Dim MGhost(20): Dim Lock(20)
Dim MatX#(20): Dim MatY#(20): Dim MatZ#(20)
Dim MatWidth#(20): Dim MatHeight#(20)
Dim TilesX(20): Dim TilesZ(20)
Dim MatHi#(20): Dim MatLo#(20)

\-----
\Initialize Arena Scaling Variables
\-----
ArenaXYZ_SF = 15000
LightXZ_SF# = .46
LightY_SF# = .55

\set up the program
sync on
sync rate 40
hide mouse
autocam off

\load the matrix
LoadMatrix("map",1)

\temporary load level info
load object "media/arena.x",100
load object "media/arena_light.x",101

\scale the arena
scale object 100, ArenaXYZ_SF, ArenaXYZ_SF, ArenaXYZ_SF
scale object 101, ArenaXYZ_SF*LightXZ_SF#, ArenaXYZ_SF*LightY_SF#, ArenaXYZ_SF*LightXZ_SF#

\position arena
position object 100,247,188,247
position object 101,247,188,247

\add mip-mapping
set matrix texture 1,2,1
set object texture 100,0,1
set object texture 101,2,1

\set fake light properties
set object 101,1,1,0,1,0,0,1
ghost object on 101

\set fog properties
fog on
fog distance 2000
fog color RGB(128,0,0)

\set ambient light amount
set ambient light 10

\colour main light
color light 0,RGB(0,0,160)

\make a light
make light 1

```

```

set point light 1,250,200,250
color light 1,RGB(255,255,100)

`-----
`MAIN LOOP
`-----
main:
do
  `the following is temporary. There will be more but it will made later
  `get keyboard input for movement
  if upkey()=1 then move camera 4
  if downkey()=1 then move camera -4
  if leftkey()=1 then yrotate camera wrapvalue(camera angle y()-4)
  if rightkey()=1 then yrotate camera wrapvalue(camera angle y()+4)

  `sort out the camera height
  x#=camera position x()
  z#=camera position z()
  y#=get ground height(1,x#,z#)+10

  ` debug code - show the position of the camera
  position camera x#,y#,z#
  text 5,5, "X Position = " + str$(x#)
  text 5,25, "Y Position = " + str$(y#)
  text 5,45, "Z Position = " + str$(z#)

  `update the screen
  sync
loop

```

Since the code is starting to get long, the remaining sections will on show the code that is added and or changed in the snippet sections. The complete source shall be included in the zip file that is included with this tutorial.

Lesson 3 – Adding the Chase Camera



Before we get into the game physics, lets add a camera that will follow us around the world. To do this, we need an object to look at. This will eventually be our hovercraft since we haven't made one yet, I will use a cube. We will use a cube rather than a sphere because we can see it rotating.

Some of you may be wondering why we are creating our own function instead of using the built in 'set camera to follow' command. I did intend to use the set "camera to follow" command in the beginning, but once I tried it out, I found that I developed all of the same things whether I used that function or my own function. The only advantage of using the 'set camera to follow' command is that it has built in collision for static collision boxes. This may be useful in the future, so I may still use it. And it will only require a few simple changes to my custom function to add it.

The following code gets placed just above the 'do loop'. It could actually get placed anywhere before the 'do loop'. It is good coding practice to bundle things that are similar together. Therefore, all of the players will be loaded in a section of code together. Because mixing them with the level loading code would possibly make things confusing.

```
`make a temporary player object  
make object cube 1,5  
position object 1,250,1,250
```

For the chase cam, we will make a function called *chase_cam*. We will want to use a function so that we can tell the computer which object to chase. If we decide to add a vehicle select screen in the future we can just tell the computer

to chase a different vehicle during the game. I like to add a comment describing the function so in the future I will remember what the function does.

```
\-----  
\chase cam  
\-----  
function chase_cam()  
  
endfunction
```

As discussed earlier, we want to be able to tell the function what object is being chased. So lets add the argument 'id' to the function. This argument will represent the number of the object that is being chased. The function now looks like this:

```
\-----  
\chase cam  
\-----  
function chase_cam(id)  
  
endfunction
```

The next thing to do is get the position variables of the object. This will be done inside the function. The variables are needed in order to work out where the camera should be placed.

```
\work out the angle of the object being chased  
yAng#=wrapvalue(object angle y(id)+180)  
  
\grab the objects current position  
xPos#=object position x(id)  
yPos#=object position y(id)  
zPos#=object position z(id)
```

NOTE : 180 is being added to the angle of the object. This is so that the camera chases along behind the object rather than in front. This could be turned into a variable for some rather cool effects (matrix type bullet cam, backward facing cam, sideways cam...)

Now lets declare the other variables that we will need in our function. We could just use numbers in the mathematic operations that we will be using in this function. However, using variables gives us the flexibility to change the properties of the mathematical operations with one change versus locating and changing all of the mathematical operations containing the property individually.

The two variables we will want to add are the camera height from the ground ('camHeight') and the camera distance from the vehicle ('camDist')

```
`other variables  
camDist=15  
camHeight=3
```

Now comes the slightly confusing math's part. What we are going to do is use the “**newxvalue**” and “**newzvalue**” commands. These commands work out the position of a point based on its distance and angle from another position. In other words, we are going to place the chase camera based on the current position of the object that is being chased. We shall provide the commands with the chase objects current position, angle and distance we want the camera to be from the object.

```
`Work out new position  
xCamPos#=newxvalue(xPos#,yAng#,camDist)  
zCamPos#=newzvalue(zPos#,yAng#,camDist)
```

To work out the height of the camera, we simply get the 'ground height' of the positions that were returned from the “**newxvalue**” and “**newzvalue**” commands then add on the 'camHeight' variable we created earlier.

```
`Work out camera height  
yCamPos#=get ground height(1,xCamPos#,zCamPos#)+camHeight
```

Now all that remains is for the camera to be repositioned according to the new coordinates that have been worked out. To do this, we'll just use the “**position camera**” command. So that we can the Chase Object, we will point the camera at the object that is being chased using the “**point camera**” command.

```
`update camera position  
position camera xCamPos#,yCamPos#,zCamPos#  
point camera xPos#,yPos#+camHeight,zPos#
```

NOTE : The camera target has the camHeight variable added as well. This places the camera above the Chase Object to give us a clearer view of the level ahead (third person view).

To ensure that the camera is always above the Chase Object, add this line.

```
`Work out camera height  
yCamPos#=get ground height(1,xCamPos#,zCamPos#)+camHeight  
if yCamPos#<yPos#+camHeight then yCamPos#=yPos#+camHeight
```

What it does is check to see if the camera height ('yCamPos#') is below the object height added to the camHeight ('yPos#+camHeight'). Why add the

camHeight? Well we want to be able to determine if the camera is below its minimum height, which is the lowest level that the camera will be placed. If the camera is below 'yPos#+camHeight' then the camera height gets set to that value.

The final function looks something like this:

```
\-----
\chase cam
\-----
function chase_cam(id)

  \work out the angle of the object being chased
  yAng#=wrapvalue(object angle y(id)+180)

  \grab the objects current position
  xPos#=object position x(id)
  yPos#=object position y(id)
  zPos#=object position z(id)

  \other variables
  camDist=15
  camHeight=3

  \work out new position
  xCamPos#=newxvalue(xPos#,yAng#,camDist)
  zCamPos#=newzvalue(zPos#,yAng#,camDist)

  \work out camera height
  yCamPos#=get ground height(1,xCamPos#,zCamPos#)+camHeight
  if yCamPos# < yPos#+camHeight then yCamPos#=yPos#+camHeight

  \update camera position
  position camera xCamPos#,yCamPos#,zCamPos#
  point camera xPos#,yPos#+camHeight,zPos#

endfunction
```

Making it all work.

Lets make it follow the vehicle (cube)

To use this function, all you have to do is call the *chase_cam* function before calling the sync command in the main “do loop”. Remember to put the number of the object that is being chased into the function (in this case the number 1)

```
chase_cam(1)
```

To see the car (our block) traveling through our world, we have to get rid of the camera movement code and replace it with object movement code. To do this, we'll need to replace the word 'camera' with 'object 1' in all of the commands that move the camera. After adding Object movement commands and the chase_cam commands the main loop looks like this.

```
\-----
\MAIN LOOP
\-----
```

```
main:
do

    yAng=object angle y(1)

    `the following is temporary. There will be more but it will made later
    `get keyboard input for movement
    if upkey()=1 then move object 1,4
    if downkey()=1 then move object 1,-4
    if leftkey()=1 then yrotate object 1,wrapvalue(yAng-4)
    if rightkey()=1 then yrotate object 1,wrapvalue(yAng+4)

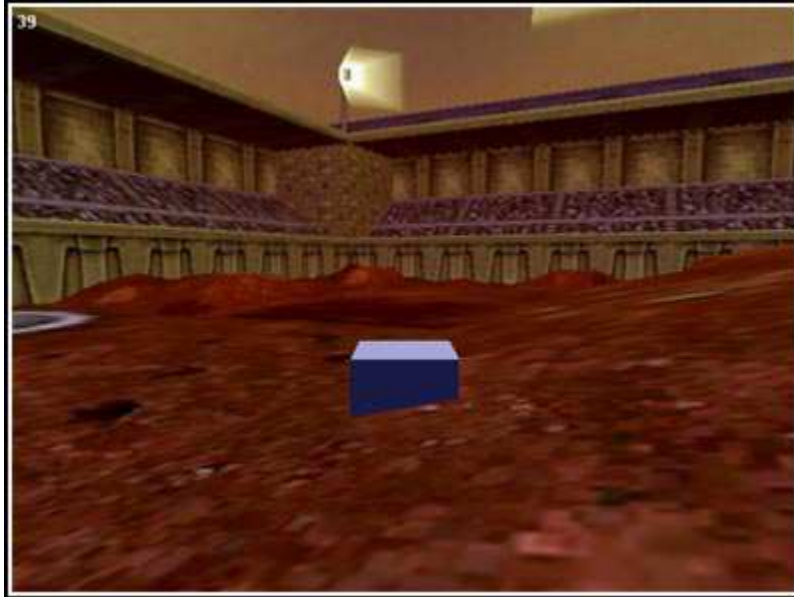
    `work out the height of the character
    xPos#=object position x(1)
    zPos#=object position z(1)
    yPos#=get ground height(1,xPos#,zPos#)

    `update the objects position
    position object 1,xPos#,yPos#,zPos#

    `update the camera
    chase_cam(1)

    `update the screen
    sync
loop
```

Lesson 4 - Developing Controls



Now it is time to develop a new control system for our game. In this lesson we will develop simple physics effects to make the game controls more interesting. We will not create true physic effects having always felt that as long as the game seems to be doing what it is meant to, and it is fun to play then it should be good enough.

We are going to do something slightly unconventional with the control system in this game. We are going to maintain normal the keyboard entry, but the actual movement code will be placed in a function. Later, we will control opposing players for this game in the future using the same function. Using the function for the main player and the opponents saves time and makes the code easier to update. It also makes the game play fair as the opposition will only be able to do what the player can do (and not cheat... too much).

For this function we need the ability to store the different characters properties (speed, friction etc.) in variables. The easiest way to do this is to use global arrays to store the different properties. Using global arrays, we have the ability to access the values from any section in the program.

The arrays required for the basic movement are:

```
dim xSpeed#(4)
dim zSpeed#(4)
dim friction#(4)
dim moveDist#(4)
```

Creating an arrays with 4 elements, gives us the ability to track four different characters using the same variables. In this case, we will use the first element in each array for the basic movement of the player. The remaining elements in the arrays we will use later.

We shall create two variables which we will store the speed of the vehicles traveling in their perspective X and Z directions: xSpeed and zSpeed. In addition a third variable called Friction shall contain the amount of friction that each vehicle will have. We should make the values unique which will give each vehicle unique handling characteristics. Lastly, the MoveDist variable shall contain the distance that each vehicle can accelerate in each loop.

Why use Global Arrays instead of standard Variables?

If we use global arrays then we can access them from any part of the program since they are 'global'. More importantly, we can obtain the properties for the different players quickly and easily. For example, we could loop through the player friction values and print them to the screen like so:

```
for player=1 to 4
  print friction#(player)
next player
```

This method is far simpler and tidier than having individual variables for each player (friction1#, friction2# friction3#... etc.).

Initializing values

Before we create the function lets initialize the variables for the test vehicle (the cube). After creating our temporary vehicle, we will assign the required values to variables as shown. Remember, these values can be changed at any time and once you have a working movement function you can adjust them to your liking.

```
`-----
`player loading
`-----
`make a temporary player object
make object cube 1,3
position object 1,250,1,250
friction#(1)=0.97
moveDist#(1)=0.065
xSpeed#(1)=0
zSpeed#(1)=0
```

NOTE: The values shown above are the ones I decided to use after I had finished writing the function you are about to start making.

Creating the movement function

Now lets create the function that the movement code will be placed in. I have decided to call this function 'move_player()'. We want this function to have the ability to control all of players forward and backwards motion and left and right directions. This makes the empty function look like this

```
\-----  
\move the specified player  
\-----  
function move_player(id,forward, backward, left, right)  
  
endfunction
```

As with the chase_cam function the first thing I want to do is get the variables we need in order to work out the new positions.

```
\-----  
\get the required object properties  
\-----  
xPos#=object position x(id)  
yPos#=object position y(id)  
zPos#=object position z(id)  
yAng#=object angle y(id)
```

NOTE : The commands shown above are have not been shown before. What they do is assign the value of the position/ angle of the object that is specified in the brackets. So if you wanted to assign the position of object 32 on the x axis to the variable xPos# then you would use:

```
xPos#=object position x(32)
```

The way the function will work is that if forward=1 then the vehicle is moving forward (accelerating). If forward=0 then the vehicle has no forward acceleration. The same goes for all of the other variables. Backward=1 means the vehicle is moving backwards, whilst left=1 means the vehicle is turning left. To do this we simply use a series of if statements.

```
\-----  
\sort out basic movement  
\-----  
\apply forward movement  
if forward=1  
  \move forward code here  
endif  
  
\apply backward movement  
if backward=1  
  \move backward code here  
endif  
  
\apply left rotation  
if left=1  
  \turn left code here  
endif  
  
\apply right rotation  
if right=1
```



```
`turn right code here
endif
```

Rather than using the 'move object' command to work out where the object is positioned we are going to use a mathematical based routine. This will let us add friction to the game, which makes the game vehicles appear as if they are floating like hovercrafts do in the real world.

To work out what the xSpeed, and zSpeed are I could use sin/ cos math's. To make this easier for people who don't know the math's. I am going to use the newxvalue and newzvalue commands again. This time because I just want to know what the distance moved is I am going to use a start point of 0 rather than the current start point. The rotation (turning) is going to stay the same and has been added.

```
`-----
`sort out basic movement
`-----
`apply forward movement
if forward=1
  xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id))
  zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id))
endif

`apply backward movement
if backward=1
  xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id)*-1)
  zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id)*-1)
endif

`apply left rotation
if left=1
  yrotate object id,wrapvalue(yAng#-4)
endif

`apply right rotation
if right=1
  yrotate object id,wrapvalue(yAng#+4)
endif
```

NOTE : Remember that angle values have to be between 0 and 360 degrees which explains the need for the wrapvalue command above.

Now to add the friction. This is actually really easy, all I have to do is multiply the xSpeed and zSpeed by the friction value. The friction value is less than 1 so multiplying the speed values by it make the numbers smaller. So even if the player is not moving forwards it will continue moving, gradually getting slower and slower until it comes to a standstill.

```
`-----
`sort out friction and other physics related things
`-----
`work out value with friction
xSpeed#(id)=xSpeed#(id)*friction#(id)
zSpeed#(id)=zSpeed#(id)*friction#(id)
```

To work out the new position is just as easy. All that needs to be done is for the xSpeed and zSpeed to be added to the xPos and zPos of the player. After this I can use these positions to work out the height of the character on the terrain. Then the final thing to do is to update the players position.

```
`work out the new position
xPos#=xPos#+xSpeed#(id)
zPos#=zPos#+zSpeed#(id)

`work out the height of the character
yPos#=get ground height(1,xPos#,zPos#)

`reposition the player object
position object id,xPos#,yPos#,zPos#
```

All going well the finished function should look like this:

```
`-----
`move the specified player
`-----
function move_player(id,forward, backward, left, right)

  `-----
  `get the required object properties
  `-----
  xPos#=object position x(id)
  yPos#=object position y(id)
  zPos#=object position z(id)
  yAng#=object angle y(id)

  `-----
  `sort out basic movement
  `-----
  `apply forward movement
  if forward=1
    xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id))
    zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id))
  endif

  `apply backward movement
  if backward=1
    xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id)*-1)
    zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id)*-1)
  endif

  `apply left rotation
  if left=1
    yrotate object id,wrapvalue(yAng#-4)
  endif

  `apply right rotation
  if right=1
    yrotate object id,wrapvalue(yAng#+4)
  endif

  `-----
  `sort out friction and other physics related things
  `-----
  `work out value with friction
  xSpeed#(id)=xSpeed#(id)*friction#(id)
  zSpeed#(id)=zSpeed#(id)*friction#(id)

  `work out the new position
  xPos#=xPos#+xSpeed#(id)
```

```

zPos#=zPos#+zSpeed#(id)

`work out the height of the character
yPos#=get_ground_height(1,xPos#,zPos#)

`reposition the player object
position object id,xPos#,yPos#,zPos#

endfunction

```

Now what needs to be done is the function needs to be called by the main loop. Otherwise all this hard work will have been for nothing :(

Where before we moved the object/ camera when the up key was pressed we will now be setting the value of the forward variable in the function. The same goes for the other controls. The down key controls the backward variable, left key controls the left variable and the right key controls the value of right.

Then we can just call the function the way you would normally.

```

\-----
\MAIN LOOP
\-----
main:
do

  `get keyboard input for movement
  if upkey()=1 then forward=1
  if downkey()=1 then backward=1
  if leftkey()=1 then left=1
  if rightkey()=1 then right=1

  `update the player
  move_player(1,forward,backward,left,right)

  `update the camera
  chase_cam(1)

  `update the screen
  sync
loop

```

If you change the main loop to this then you will notice that once the object starts moving it doesn't stop. There are two ways of sorting this out.

1. Set the values of the variables at the start of the loop
2. Set the value of the variables if the key is not being pressed (use the 'else' command)

I have decided to go with the second option. This means that each variable is being assigned to only once per loop rather than twice which could happen using the other method.

```
`-----  
`MAIN LOOP  
`-----  
main:  
do  
  
    `get keyboard input for movement  
    if upkey()=1 then forward=1 else forward=0  
    if downkey()=1 then backward=1 else backward=0  
    if leftkey()=1 then left=1 else left=0  
    if rightkey()=1 then right=1 else right=0  
  
    `update the player  
    move_player(1,forward,backward,left,right)  
  
    `update the camera  
    chase_cam(1)  
  
    text 10,10,str$(screen fps())  
  
    `update the screen  
    sync  
loop
```

Is that it?

You will be pleased to hear yes... that's it for this tutorial. It is a fairly confusing tutorial and a technique I have made up myself, although I am sure others use it (at least I hope they do). The next tutorial will be adding to the movement function to make the vehicles launch themselves off of the jumps and bumps in the terrain.

Lesson 5 - Gravity and Jumping



This is something that is asked about quite a bit on the DarkBASIC Pro forums so is something I thought I should have a go at. This is actually something I would have liked to have added to Hover Racer but at the time I didn't know how to go about it. Thanks to Dr Av I now have a better idea of what to do, and with a bit of playing I got something fairly nice working.

Lets start then shall we?

As normal we have to specify the variables that will be needed for this example. The reason I know what variables are needed is I write the code before I explain it. When coding I actually add the variables as they are needed but it is easier for me to tell you what they are at the start.

There are two extra variables for this tutorial. One of these I will call a constant. Constants are actually a type of variable used in other programming languages. The idea behind them is that they are numbers that are used in more than one place that do not change. The advantage of them is that you can assign them values at the start of the program and then refer to them whenever you need to. If you then want to change the value you can just edit it at the start of the program and it will be adjusted everywhere else. This saves you having to remember where you used the variable in your code.

The gravity variable is the amount of downward force that will be applied to the vehicles. This value is positive as it gets subtracted from the ySpeed making the vehicle move down.

```
`gravity  
dim gravity#(0) : gravity#(0)=0.1
```

NOTE : The colon (:) is used to place two pieces of code on the same line. I use this here so that I know where the 'constant' has been set.

I also need a new variable for the character properties. This is the ySpeed variable that sits alongside the xSpeed and zSpeed variables. This is used to tell you how fast the character is dropping/ jumping. I have included all of the other character property variables to show how it has been added.

```
`player arrays  
dim xSpeed#(4)  
dim ySpeed#(4)  
dim zSpeed#(4)  
dim friction#(4)  
dim movedist#(4)
```

Now for the gravity...

To add the gravity code I am going to adjust the move_player function. First thing to do is work out the ySpeed value.

```
`add gravity value  
ySpeed#(id)=ySpeed#(id)+gravity#(0)
```

The reason this value has been added is that when working out the position we subtract the ySpeed from the current y position (yPos). I added this code just after the space where the xSpeed and zSpeed values have friction applied.

This on its own does nothing. What you need to do now is subtract the ySpeed from the yPos. This code gets added below the space where the xPos and zPos are worked out.

```
yPos#=yPos#-ySpeed#(id)
```

Run this and you will find it does nothing. That's because the yPos is being reassigned to the height of the ground at the current position. So DELETE the following line:

```
yPos#=get ground height(1,xPos#,zPos#)
```

Run the program now and you will see that the cube falls through the ground. This is because the gravity has taken control. What needs to be done now is to stop the vehicle from falling through the floor.

The complicated bit

The following is what does all of the work. To start with we need to work out whether or not the characters new position is below the ground height. If it is above the ground height we don't have to do anything - the gravity will take care of the falling. So all we have to do is reposition when below the ground height.

```
`work out the height of the character
if yPos#<get ground height(1,xPos#,zPos#)

endif
```

To reposition the characters when they are below the ground we just have to get the ground height at the vehicles current position.

```
`work out the height of the character
if yPos#<get ground height(1,xPos#,zPos#)
    yPos#=get ground height(1,xPos#,zPos#)
endif
```

Run this and you will notice that it looks exactly the same as it was before you started this tutorial. To make the gravity take effect we have to change the ySpeed of the character.

```
`work out the height of the character
if yPos#<get ground height(1,xPos#,zPos#)
    ySpeed#(id)=0
    yPos#=get ground height(1,xPos#,zPos#)
endif
```

Run this code and you will notice that whilst the vehicle doesn't jump it does fall if you go off of a high ledge. Now you aren't stuck to the ground as much as before. To make the character jump you should add the difference between the characters actual position and the height of the ground.

```
`Work out the height of the character
if yPos#<get ground height(1,xPos#,zPos#)
    ySpeed#(id)=ySpeed#(id)+(yPos#-get ground height(1,xPos#,zPos#))
    yPos#=get ground height(1,xPos#,zPos#)
endif
```

This 'should' make your character jump. If not then somewhere something has gone wrong. To help you out the function I have looks like this.

```
`-----
`move the specified player
`-----
function move_player(id,forward, backward, left, right)

    `-----
    `get the required object properties
    `-----
    xPos#=object position x(id)
    yPos#=object position y(id)
    zPos#=object position z(id)
    yAng#=object angle y(id)
```

```

\-----
\sort out basic movement
\-----
\apply forward movement
if forward=1
    xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id))
    zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id))
endif

\apply backward movement
if backward=1
    xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id)*-1)
    zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id)*-1)
endif

\apply left rotation
if left=1
    yrotate object id,wrapvalue(yAng#-4)
endif

\apply right rotation
if right=1
    yrotate object id,wrapvalue(yAng#+4)
endif

\-----
\sort out friction and other physics related things
\-----
\work out value with friction
xSpeed#(id)=xSpeed#(id)*friction#(id)
zSpeed#(id)=zSpeed#(id)*friction#(id)

\add gravity value
ySpeed#(id)=ySpeed#(id)+gravity#(0)

\work out the new position
xPos#=xPos#+xSpeed#(id)
zPos#=zPos#+zSpeed#(id)
yPos#=yPos#-ySpeed#(id)

\work out the height of the character
if yPos#<get ground height(1,xPos#,zPos#)
    ySpeed#(id)=ySpeed#(id)+(yPos#-get ground height(1,xPos#,zPos#))
    yPos#=get ground height(1,xPos#,zPos#)
endif

\reposition the player object
position object id,xPos#,yPos#,zPos#

endfunction

```


Lesson 6 - Arena Collision



You may have noticed that it is possible to go through the arena. This is beginning to annoy me so I have decided it is time to sort it out. There are a number of ways that you could do this. I decided to use nice simple area collision. Because the arena is, and always will be, square then this is the quickest and simplest method.

The way this collision works is quite simply to check if the position of the object is outside of the specified area. If it is then the player gets repositioned. There are two things that need collision added. One is the player, and the other is the camera.

Player to Arena Collision

The simplest, and most obvious, place to add the collision is the move_player function. This way the collision will automatically be done for each player. We know that the arena is 500 units square, and that the bottom corner is positioned at 0,0 and that the top corner is positioned 500,500.

In the function we work out what the new position of the player is going to be and these are the values that can be used to check if the character is outside of the bounding square.

After the new player values are calculated we have the variables xPos# and zPos#. These are what we will check for collision.

```
if xPos#<0 then xPos#=0
```

Simple eh? What this is doing is seeing if the player position is less than the minimum position of the arena. If we were to add this though you would see that the player gets stuck halfway through the wall. This is because the size of the player has not been taken into account. So 'if xPos#<0 then xPos#=0' becomes:

```
if xPos#<5 then xPos#=-5
if zPos#<5 then zPos#=-5
```

NOTE : I chose the value 5 because it looked good. This means that the player does not quite reach the wall but it is good enough. This is the value that will be used for all of the future player collision calculations.

Below I have also added the code for the maximum position value. The maximum value was worked out by subtracting 5 from the width of the arena (500). This is what we end up with:

```
`collision
if xPos#<5 then xPos#=-5
if zPos#<5 then zPos#=-5
if xPos#>495 then xPos#=495
if zPos#>495 then zPos#=495
```

The move_player function with working collision now looks like this:

```
`-----
`move the specified player
`-----
function move_player(id,forward, backward, left, right)

`-----
`get the required object properties
`-----
xPos#=object position x(id)
yPos#=object position y(id)
zPos#=object position z(id)
yAng#=object angle y(id)

`-----
`sort out basic movement
`-----
`apply forward movement
if forward=1
    xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id))
    zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id))
endif

`apply backward movement
if backward=1
    xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id)*-1)
    zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id)*-1)
endif

`apply left rotation
if left=1
    yrotate object id,wrapvalue(yAng#-4)
endif

`apply right rotation
if right=1
    yrotate object id,wrapvalue(yAng#+4)
```

```

endif

\-----
\sort out friction and other physics related things
\-----

\work out value with friction
xSpeed#(id)=xSpeed#(id)*friction#(id)
zSpeed#(id)=zSpeed#(id)*friction#(id)

\add gravity value
ySpeed#(id)=ySpeed#(id)+gravity#(0)

\work out the new position
xPos#=xPos#+xSpeed#(id)
zPos#=zPos#+zSpeed#(id)
yPos#=yPos#-ySpeed#(id)

\collision
if xPos#>495 then xPos#=495
if zPos#>495 then zPos#=495
if xPos#<5 then xPos#=5
if zPos#<5 then zPos#=5

\work out the height of the character
if yPos#<get ground height(1,xPos#,zPos#)
    ySpeed#(id)=ySpeed#(id)+(yPos#-get ground height(1,xPos#,zPos#))
    yPos#=get ground height(1,xPos#,zPos#)
endif

\reposition the player object
position object id,xPos#,yPos#,zPos#

endfunction

```

Camera Collision

The camera collision can be done with exactly the same technique. The only difference is the name of the position variables and the amount that the camera has to stay from the side of the arena. So you can simply copy and paste the collision code.

To make it work for the camera you have to swap xPos# for xCamPos# and zPos# for zCamPos#. The camera will not go through the walls anymore. I changed the minimum collision values to 1 and the maximum collision values to 499 but this is entirely up to you.

Once completed the camera function looks something like this:

```

\-----
\chase cam
\-----
function chase_cam(id)

\work out the angle of the object being chased
yAng#=wrapvalue(object angle y(id)+180)

\grab the cameras current position
xPos#=object position x(id)
yPos#=object position y(id)
zPos#=object position z(id)

\other variables

```

```

camDist=15
camHeight=3

`work out new position
xCamPos#=newxvalue(xPos#,yAng#,camDist)
zCamPos#=newzvalue(zPos#,yAng#,camDist)

`camera collision
if xCamPos#>499 then xCamPos#=498
if zCamPos#>499 then zCamPos#=498
if xCamPos#<1 then xCamPos#=2
if zCamPos#<1 then zCamPos#=2

`work out camera height
yCamPos#=get ground height(1,xCamPos#,zCamPos#)+camHeight
if yCamPos#<yPos#+camHeight then yCamPos#=yPos#+camHeight

`update camera position
position camera xCamPos#,yCamPos#,zCamPos#
point camera xPos#,yPos#+camHeight,zPos#

endfunction

```

NOTE : The camera collision is added to the chase_cam function rather than the move_player function.

Lesson 7 - A Few Changes



During the course of a games production things often change. You may find a new way of doing things or you might decide that what you have does not work as you would like. One of the things I have decided that I don't like whilst playing this game is the camera. I don't like the way the camera is 'glued' to the back of the vehicle. I want it to be more organic, so this is what I am going to do now.

Camera problems?

Smoothing movement out is actually relatively straightforward in DarkBASIC Pro. To make the movement smoother I am going to use the **curvevalue** command in the chase_cam function.

To use the **curvevalue** function you need to know the current value and the new value of the things you want to move smoothly between. In this case it will be the different positions of the camera. I have already worked out the new values of the camera and stored them in the camPos variables (xCamPos#, yCamPos#, and zCamPos#). To find out the camera position the **camera position** commands can be used. A speed is needed for the smoothness of the movement that I have decided is 4.

```
xCamPos#=curvevalue(xCamPos#,camera position x(),4)
yCamPos#=curvevalue(yCamPos#,camera position y(),4)
zCamPos#=curvevalue(zCamPos#,camera position z(),4)
```

What this is doing is setting the camera position variables to equal the smooth value that has been calculated by the **curvevalue** command.

This code should be placed just before the camera is position with the 'position camera' command.

The vehicle...

Something else I want to sort out now is the vehicle. First thing that needs to be done is to make the vehicle.

Once we have a vehicle to use it has to be loaded. This is done with the **Load Object** command. All you have to do is specify the object and the object number you want to give it. Because I know what I will have to do in the next lesson I am going to use a new object number (rather than the object number 1 that we have already) and then 'glue' the vehicle to the cube that is being used. The reason for doing it this way is simply that it will make some of the math's a lot easier in the future.

So I am going to load the hovercraft as object number 11.

```
Load object "hovercraft",11
```

At the moment the object has no texture so we now need to load this. I decided not to apply the texture in the modeling program but through code so that I could use the same model for all the vehicles and then just change the texture for the opponents that will be added later.

Loading a texture is as easy as loading an image, in fact it uses the same command. To apply the texture to the object we will use the **'texture object'** command.

```
load object "media/hovercraft.x",11  
load image "media/hovercraft.bmp",1  
texture object 11,1  
scale object 11,95,95,95
```

NOTE: I scaled the object as well. I decided that it was too small when I tested the model in the game and rather than scale it in the modeling program I decided that it was easier to scale it through code. It won't add any noticeable time to the game loading so it doesn't really matter.

To make the object look nicer I want to add mipmapping. This smooths the look of the texture at long distances. The command to do this is the 'set object texture' command.

```
set object texture 11,0,1
```

If you play the game now you will see that the vehicle has been loaded but it does not move. This is because we have not glued the hovercraft to the object yet.

```
glue object to limb 11,1,0
```

What this command is doing is gluing object 11 to limb 0 on object 1, the limbs are numbered from 0 on up, if there was more than 1 limb then the next limb would be 1, then 2 etc.

Play the game now and the hovercraft moves but you can see the cube showing through it. There are two ways to sort this out. You could scale the cube so that it is so small you can not see it, or you could hide the cube.

Hiding the cube would be the best idea but doing this would stop the vehicle from being glued to it (the vehicle can not see a hidden cube). What has to be done then is to hide the cube limb. Then the object will be able to follow.

```
hide limb 1,0
```

Lesson 8 – Landscape Tilting



Now that the vehicle is loaded it has become very obvious that it isn't tilting to the landscape as it would in real life.

How do we know what angle the vehicle is at then?

Well we don't. There are a number of math's ways that could work it out for us but I don't want to do any confusing math's tilting the object so that it looks right is more than enough.

What I am going to so is work out the height of four points around the hover craft. Then from these I can work out the difference in height between the front and back, and the left and right sides of the vehicle. Once we have the difference we can 'fake' the rotation of the vehicle using the difference values.

Getting started...

All of this code will be placed in the movement function. This is so that it will be done automatically for each vehicle in the game. This code will be placed in the section that checks to see if the vehicle is on the floor or not. This is because we don't want the hovercraft to tilt when it is in the air.

The first thing to do is to work out the coordinates of the points that the heights will be taken from. These points will be in front of, behind, and to the left and right of the hovercraft. This can be done easily with the **newxvalue** and **newzvalue** commands.


```

\-----
\tilt the vehicle to the ground
\-----
distVal#=1

\work out the positions of the front, back, left and right of the vehicle
ang#=yAng#
frontX#=newxvalue(xPos#,ang#,distVal#)
frontZ#=newzvalue(zPos#,ang#,distVal#)
ang#=yAng#+180
backX#=newxvalue(xPos#,ang#,distVal#)
backZ#=newzvalue(zPos#,ang#,distVal#)
ang#=yAng#+90
leftX#=newxvalue(xPos#,ang#,distVal#)
leftZ#=newzvalue(zPos#,ang#,distVal#)
ang#=yAng#-90
rightX#=newxvalue(xPos#,ang#,distVal#)
rightZ#=newzvalue(zPos#,ang#,distVal#)

```

NOTE : I have added the 'distVal#' variable. This is the distance out from the vehicle that the heights should be taken from. I have added it as a variable to make it easier to change values later if I should want to. Also notice the use of the colon (:). This is used to place more than one command on the same line. It can also be used to add remarks.

Now the heights need to be worked out. This is nice and simple using the 'get ground height' command.

```

\work out the different heights
frontHeight#=get ground height(1,frontX#,frontZ#)
backHeight#=get ground height(1,backX#,backZ#)
leftHeight#=get ground height(1,leftX#,leftZ#)
rightHeight#=get ground height(1,rightX#,rightZ#)

```

Now that we have these values all that needs to be done is to work out what the angle should be. To do this we take the difference between the heights and multiply them by a number to make them big enough for a visible effect.

```

\Work out tilt values
xAng#=wrapvalue((backHeight#-frontHeight#)*30)
zAng#=wrapvalue((leftHeight#-rightHeight#)*30)

```

NOTE : Remember to use the **wrapvalue** command as objects can not be rotated by values greater than 360 or less than 0.

How do you work out the multiplier number? Well I just guessed. After about five goes I got it to a number that looked good. How did I know that the number looked good? Simple, I rotated the vehicle.

```

\update the vehicle rotation
rotate object id+10,xAng#,0,zAng#

```

The id has 10 added to it. This is because we are rotating the vehicle object not the pivot object (the cube). The cube gets rotated in earlier code, and getting rid of the plus ten makes everything rotate wrong.

Is that it?

Well normally yes. However after playing with it for a bit I decided to smooth out the rotation of the vehicle to make the ground affect the vehicle gradually, adding some damping and making the movement more realistic. To do this I will use the '**curveangle**' command. This works much like the '**curvevalue**' command only it makes the returned value a valid angle for a DarkBASIC rotation.

When using the **curveangle** command we need to know the current angle of the thing that is being rotated.

```
`Work out tilt values
```

```
xAng#=curveangle((backHeight#-frontHeight#)*30,object angle x(id+10),5)
```

```
zAng#=curveangle((leftHeight#-rightHeight#)*30,object angle z(id+10),5)
```

Remember to add the number 10 to the id of the object you are getting the rotation from. This is because we are rotating the vehicle not the pivot.

Lesson 9 – Slippy Slidey - Sliding Down Hills



To complete the physics, and to make the game more interesting to play I think that the character should be affected by the slope of the hill. If the vehicle is going up the hill then it should be pulled back down (because of gravity).

Like everything so far this is going to be faked using simple math's. We need to affect the vehicle on both its X and Z axis (forward, backwards, left, and right). This will make the vehicle slide down sideways making everything seem more realistic.

As with all the code that affects the movement of the vehicle this code should be placed in the movement function. It should also only work when the vehicle is only on the ground As some of the variables that were worked out in the previous tutorial will be needed the best place to add the sliding code is just after the tilt code.

The basic movement code will be the same as the movement code earlier. All that needs to be done is for a few variables to be changed. What needs to be done is for the xSpeed and zSpeed variables to be adjusted, all we have to do is work out how much to adjust them.

```
`adjust forward/ backward momentum  
xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,xMoveDist#)  
zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,xMoveDist#)
```

xMoveDist# is a variable we need to work out ourselves. This could be done a number of ways but the simplest is to use the difference between the heights that were worked out earlier

```

xMoveDist#=(backHeight#-frontHeight#)/30

`adjust forward/ backward momentum
xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,xMoveDist#)
zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,xMoveDist#)

```

Working out the left and right movement is done in just the same way, however there need to be a couple of small changes. The zMoveDist variable needs to be set with the left and right heights instead of the front and back. More importantly to affect the sideways motion the angle specified in the 'newxvalue' command needs to have 90 subtracted from it. This is so that the value worked out is 90 degrees from the forward motion.

```

`-----
`slide down slopes
`-----
xMoveDist#=(backHeight#-frontHeight#)/30
zMoveDist#=(leftHeight#-rightHeight#)/30

`adjust forward/ backward momentum
xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,xMoveDist#)
zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,xMoveDist#)

`adjust left/ right momentum
xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#-90,zMoveDist#)
zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#-90,zMoveDist#)

```

NOTE : If you add 90 instead of subtract it then the vehicle slides sideways up hills instead of down.

Now that the physics is complete the movement function should look like this:-

```

`-----
`move the specified player
`-----
function move_player(id,forward, backward, left, right)

`-----
`get the required object properties
`-----
xPos#=object position x(id)
yPos#=object position y(id)
zPos#=object position z(id)
yAng#=object angle y(id)

`-----
`sort out basic movement
`-----
`apply forward movement
if forward=1
  xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id))
  zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id))
endif

`apply backward movement
if backward=1
  xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,moveDist#(id)*-1)
  zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,moveDist#(id)*-1)
endif

```

```

`apply left rotation
if left=1
    yrotate object id,wrapvalue(yAng#-4)
endif

`apply right rotation
if right=1
    yrotate object id,wrapvalue(yAng#+4)
endif

`-----
`sort out friction and other physics related things
`-----

`work out value with friction
xSpeed#(id)=xSpeed#(id)*friction#(id)
zSpeed#(id)=zSpeed#(id)*friction#(id)

`add gravity value
ySpeed#(id)=ySpeed#(id)+gravity#(0)

`work out the new position
xPos#=xPos#+xSpeed#(id)
zPos#=zPos#+zSpeed#(id)
yPos#=yPos#-ySpeed#(id)

`collision
if xPos#>495 then xPos#:=495
if zPos#>495 then zPos#:=495
if xPos#<5 then xPos#:=5
if zPos#<5 then zPos#:=5

`work out the height of the character
if yPos#<get ground height(1,xPos#,zPos#)

    ySpeed#(id)=ySpeed#(id)+(yPos#-get ground height(1,xPos#,zPos#))
    yPos#:=get ground height(1,xPos#,zPos#)

`-----
`tilt the vehicle to the ground
`-----
distVal#:=1.2

`work out the positions of the front, back, left and right of the vehicle
ang#=-yAng#    : frontX#:=newxvalue(xPos#,ang#,distVal#) : frontZ#:=newzvalue(zPos#,ang#,distVal#)
ang#=-yAng#+180 : backX#:=newxvalue(xPos#,ang#,distVal#) : backZ#:=newzvalue(zPos#,ang#,distVal#)
ang#=-yAng#+90  : leftX#:=newxvalue(xPos#,ang#,distVal#) : leftZ#:=newzvalue(zPos#,ang#,distVal#)
ang#=-yAng#-90  : rightX#:=newxvalue(xPos#,ang#,distVal#) : rightZ#:=newzvalue(zPos#,ang#,distVal#)

`work out the different heights
frontHeight#:=get ground height(1,frontX#,frontZ#)
backHeight#:=get ground height(1,backX#,backZ#)
leftHeight#:=get ground height(1,leftX#,leftZ#)
rightHeight#:=get ground height(1,rightX#,rightZ#)

`work out tilt values
xAng#:=curveangle((backHeight#-frontHeight#)*30,object angle x(id+10),5)
zAng#:=curveangle((leftHeight#-rightHeight#)*30,object angle z(id+10),5)

`update the vehicle rotation
rotate object id+10,xAng#,0,zAng#

`-----
`slide down slopes
`-----
xMoveDist#:=(backHeight#-frontHeight#)/30
zMoveDist#:=(leftHeight#-rightHeight#)/30

`adjust forward/ backward momentum
xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#,xMoveDist#)
zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#,xMoveDist#)

```

```
`adjust left/ right momentum
xSpeed#(id)=xSpeed#(id)+newxvalue(0,yAng#-90,zMoveDist#)
zSpeed#(id)=zSpeed#(id)+newzvalue(0,yAng#-90,zMoveDist#)

endif

`reposition the player object
position object id,xPos#,yPos#,zPos#

endfunction
```

Lesson 10 – HotSpots



In this game I need to know whether or not the player is on a tile that contains the currently lit light (the object being chased). To do this I have flagged each light tile as a hotspot in MatEdit. For those who don't know how to do this you use the F keys along the top of your keyboard. There are 8 spots on the level that could be lit up and so I used the keys 1 to 8.

However just doing this does not tell me whether or not the character is on a valid square or not. To do this another function is in order.
checkHotspot()

To keep in with the style of the MatEdit commands I have decided to call this function 'checkHotspot'. What I want the function to do is tell me what hotspot (if any) the specified object (player) is on.

```
\-----  
\work out what hotspot the current object is on (if any)  
\-----  
function CheckHotspot(id)  
endfunction currentHotSpot
```

If the player isn't on a valid tile then I want to be told that the current hotSpot is 0. This is a standard value in programming often used to mean False (1=true). The first thing to do is to reset the currentHotspot.

```
currentHotspot=0
```

Now we need to work out what the current tile coordinates are. This is so that we can check them in the array that stores the hotspot data. To work out the x tile (tileX) position of the player we can take the player position and divide it by the x size of the tiles on the matrix (Info(0)). This value needs to be turned into an integer as there aren't any fractional tiles. Finally the number 1 needs to be added to the tile value (I don't know why - it just does). After this has been done the same has to be done for the z tile position (tileZ).

```
`work out current tile position  
tileX=int(object position x(id)/Info(0))+1  
tileZ=int(object position z(id)/Info(1))+1
```

Once all this has been done we need to check whether the current tile position matches the position of any of the hotspots that were specified in the matEdit file.

After a bit of reading through the loadMatrix code I worked out that the hotspots are stored in the 'FKey' array. The FKey array has two dimensions. The first stores the value of the hotSpot and the second stores the x (value 0) and z (value 1) position of the hotspot.

There are ten possible positions so all of these are checked with a for loop:

```
For                hotspot=1                to                10  
Next hotspot
```

Next what needs to be done is to check if the positions match. To do this I will use an if statement that checks two values. I will check both the tileX and tileZ variables in one go by using the 'And' operator. Using 'And' means that both of the statements used must be true.

```
if tileX=FKey(hotspot,0) and tileZ=FKey(hotspot,1) then ...
```

So if the x tile position (tileX) equals the x position for the current hotspot (FKey(hotspot,0)) and the z tile position (tileZ) equals the z position for the current hotspot (FKey(hotspot,1)) then we know that this is the current Hotspot.

This means we can set currentHotspot to the same value as hotspot.

```
If tileX=FKey(hotspot,0) and tileZ=FKey(hotspot,1) then currentHotspot=hotspot : Exit
```

NOTE : The exit command is used at the end of the line. This command exits from the current loop which in this case means that if the currentHotspot is found the values after are not checked, thus saving a few loops.

The end...

The final function looks like this.

```
`-----  
`work out what hotspot the current object is on (if any)  
`-----  
function CheckHotspot(id)  
  
  `reset current hotspot  
  currentHotspot=0  
  
  `work out current tile position  
  tileX=int(object position x(id)/Info(0))+1  
  tileZ=int(object position z(id)/Info(1))+1  
  
  `find current hotspot  
  For hotspot=1 to 10  
    if tileX=FKey(hotspot,0) and tileZ=FKey(hotspot,1) then currentHotspot=hotspot : Exit  
  Next hotspot  
  
endfunction currentHotspot
```

Extras

The function on its own does nothing. So that I can test whether this works or not an extra line of code needs to be added to the main loop.

```
text 10,30,str$(CheckHotspot(1))
```

All this does is print the current hotspot for the specified player to the screen. In the next tutorial we will have to do something with this information.

Lesson 11 – New Target



In the game I want to have a target object. This is going to be a beam of light that is emitted from the ground where the tile is. This makes the target tile a lot easier to see and makes the game a bit prettier (adds some colour).

The first thing to do is load an object to use as the light and set its properties. I have actually decided to load two objects so that I can animate their rotation adding a bit of movement to the level.

Near the start of the program the object need to be loaded. I decided to add it just before the character is loaded but it could be added pretty much anywhere.

```
\-----  
\load a target object  
\-----  
load object "media/light_beam.3ds",200  
load object "media/light_beam2.3ds",201  
  
ghost object on 200  
ghost object on 201  
  
set object 200,1,1,0,1,0,0,1  
set object 201,1,1,0,1,0,0,1
```

Now that the object is loaded it needs to be positioned. This positioning will eventually be called from different functions so the best idea would be to place it in a function.

new_target()

This function is going to be called new_target.

```
`-----  
`pick a new target location  
`-----  
function new_target()  
  
endfunction
```

The first thing to do is to pick a hotspot. This hotspot value needs to be placed into a global variable as it will need to be accessed from a number of different places in the game. In DarkBASIC this means creating a 0 dimensioned array. I also want to be able to store the old hotspot value so that I can stop the same hotspot from being used two times in a row.

At the top of the game where the arrays are declared add this code.

```
`hotspot data  
dim hotspot(0)  
dim oldHotspot(0)
```

Back to the function then. The first thing to do is to pick a new hotspot. This is as simple as picking a random number. However I have already mentioned that I don't want the same hotspot to be picked two times in a row. To prevent this I have added a small loop. The idea is that the loop will keep repeating until the newhotspot and oldhotspot values are different. I have already said how this will be done in describing how it will work (reread the last line). For those who aren't sure I am going to use a 'repeat until' loop:

```
repeat  
    hotSpot(0)=rnd(7)+1  
until hotspot(0)<>oldHotspot(0)  
  
oldHotspot(0)=hotspot(0)
```

NOTE : After the new hotspot has been picked the oldhotspot is given the chosen value. This is so that the next time the value is checked it uses the 'oldhotspot' value.

Now we know what hotspot is going to be used the position of the light beams need to be worked out.

This is actually fairly simple but it looks nasty in code form. The x and z positions need to be worked out as two separate variables (obviously). First you have to take the x tile position of the chosen hotspot (FKey(hotspot(0),0)) and multiply it by the width of the tiles (info(0)). This will give you the corner of the chosen tile. To place the object in the middle we need to subtract half the width of the tiles (info(0)/2). This gives us:

```
xPos=(FKey(hotspot(0),0)*info(0))-(info(0)/2)
```

This also needs to be done for the z axis. Once these two have been worked out the yPos can be calculated using the get ground height command, and the two objects can be positioned using the position object commands giving us this as the completed function:

```
\-----  
\pick a new target location  
\-----  
function new_target()  
  
  repeat  
    hotSpot(0)=rnd(7)+1  
  until hotspot(0)<>oldHotspot(0)  
  
  oldHotspot(0)=hotspot(0)  
  
  xPos=(FKey(hotspot(0),0)*info(0))-(info(0)/2)  
  zPos=(Fkey(hotspot(0),1)*info(1))-(info(1)/2)  
  
  yPos=get ground height(1,xPos,zPos)  
  
  position object 200,xPos,yPos,zPos  
  position object 201,xPos,yPos,zPos  
  
endfunction
```

Finishing it off

To start the game a new hotspot needs to be picked. Just before the main loop starts call the function 'new_target()'. Also once the player is in the current hotspot a new target needs to be picked so in the main loop you can remove the line added at the end of the last tutorial and add this one:

```
if checkHotspot(1)=hotSpot(0) then new_target()
```

What this is doing is checking to see if the players hotspot is the same as the target hotspot. If it is then a new target is picked.

Lesson 12 – Special Effects



To make the game look nicer I want to add some special effects to the target objects. First off I want to make them rotate in opposite directions. This was mentioned earlier and will just add some movement to a static world. This is going to be separated from the main loop by being placed in a sub routine. A subroutine is specified by typing the name of the routine followed by a colon (:).

So that the code knows where it came from when the 'return' command is used.

```
update_target:  
  
  yrotate object 200,wrapvalue(object angle y(200)+2)  
  yrotate object 201,wrapvalue(object angle y(201)-2)  
  
return
```

The yrotate object command is the same as used earlier to make the players rotate when they turn.

To call the sub routine you use the command 'gosub subroutineName'. The following code should be placed in the main loop (just before the sync).

```
gosub update_target
```

Glowing lights

Something else I thought would look nice is adding a huge glow around the light. This would actually be seen as a circle of light being emitted from the light source. First you need to make the flare object (a plain) texture it, and set its properties. I done this in the same space as the rest of the target loading stuff.

The new load target files code looks like this

```
\-----  
\load a target object  
\-----  
load object "media/light_beam.3ds",200  
load object "media/light_beam2.3ds",201  
  
make object plain 202,150,150  
load image "media/light_3.bmp",1  
texture object 202,1  
set object rotation zyx 202  
  
ghost object on 200  
ghost object on 201  
ghost object on 202  
  
set object 200,1,1,0,1,0,0,1  
set object 201,1,1,0,1,0,0,1  
set object 202,1,1,0,1,0,0,1
```

NOTE : The new flare object is number 202 so that should help you when adding in the code.

Something else that needs to be done is for the flare to be positioned at the same time and in the same location as the other light objects. A small change to the new_target() function sees this line of code added at the bottom:

```
position object 202,xPos,yPos,zPos
```

Lighting

The other thing I wanted to do is make a light that also gets positioned at the location of the target. This will help improve the level of realism making it look like the light is actually turned on.

First a new light needs to be made and its properties should be set:

```
\target light  
make light 3  
set point light 3,0,0,0  
color light 3,RGB(0,128,255)  
set light range 3,1000
```

Also the light needs to be positioned when the other objects get repositioned. Like all the other objects this gets placed in the new_target() function.

```
position light 3,xPos,yPos+5,zPos
```

NOTE : I am adding 5 to the height of the light so that it covers a wider range of objects.

Lesson 13 – Level Complete



The target of the game is to get to 10 of the lights before any of the other players. To do this we need to keep track of how many of the light shave been reached by each player. In theory each player could reach 9 different lights before someone wins.

A variable is needed to store the number of times each player reaches the target.

```
dim targetCount(4)
```

Every loop the computer must check if the player has reached a new target. This can be done with the code that was used in previous tutorials that checks if you are in the currently selected hotspot. If you are in the currently selected hotspot then the targetCount needs to be increased by 1.

```
if checkHotspot(1)=hotSpot(0)
    new_target()
    targetCount(1)=targetCount(1)+1
endif
```

Then all that needs to be done is for a check to be added to see if the new targetCount for the character=10. If it does then the code should move to a new loop that does the end sequence for the level.

```
if checkHotspot(1)=hotSpot(0)
    new_target()
    targetCount(1)=targetCount(1)+1
    if targetCount(1)=10 then end_game()
endif
```

Above you can see that a new function is being used that is called end_game(). this function will play a loop saying congratulation for winning.

```
\-----  
\End game loop  
\-----  
function end_game()  
  
    repeat  
        text 10,30,"Congratulations"  
        text 10,50,"Press Space Key to end"  
        sync  
    until spacekey()=1  
  
    end  
  
endfunction
```

For now all the loop does is print congratulations to the screen repeatedly until the space bar (spacekey) is pressed. When this is done the game ends. A better end game sequence will be made later but this will do for now.

The next thing to do is add other players.

Lesson 14 – Opposition



Now that I have a game to play I want someone to play against. Loading the players is simple - the code is already there. It just needs to be adjusted to load more than one vehicle.

This is what we have currently.

```
make object cube 1,1  
hide limb 1,0  
  
load object "media/hovercraft.3ds",11  
load image "media/hovercraft.bmp",1  
texture object 11,1  
scale object 11,175,175,175  
set object texture 11,0,1  
  
glue object to limb 11,1,0  
  
position object 1,250,1,250  
friction#(1)=0.97  
moveDist#(1)=0.065  
xSpeed#(1)=0  
zSpeed#(1)=0
```

If this is turned into a for loop that loops four times making a new player each time then it will more than do what is wanted. Each loop the player 'id' will increase by 1, and this will be used to work out the object numbers. The pivot objects (the cubes the players are glued to) will be given the same number as the player id. The actual vehicle objects will be given the value 10+id.

```

for id=1 to 4
  make object cube id,1
  hide limb id,0

  load object "media/hovercraft.3ds",10+id
  load image "media/hovercraft.bmp",1
  texture object 10+id,1
  scale object 10+id,175,175,175
  set object texture 10+id,0,1

  glue object to limb 10+id,id,0

  position object id,250,1,250
  friction#(id)=0.97
  moveDist#(id)=0.065
  xSpeed#(id)=0
  zSpeed#(id)=0

  targetCount(id)=0
next id

```

NOTE : The player currently all have the same texture. This will be sorted out later.

That is all there is to it. However if you run the game now they will just sit there. The reason for this is that they have not been told to move.

The following is 'not' Artificial intelligence however it is enough to make the players move randomly around the arena and show you that all the work that has been done up to now was worth it.

```

for id=2 to 4
  forward=1
  backward=0
  left=rnd(1)
  right=rnd(1)

  move_player(id,forward,backward,left,right)
next id

```

All this is doing is picking a random direction for the player to turn and then moving them. Notice the loop is from 2 to 4 not 1 to 4. The reason for this is that player 1 is being controlled by you, the game player.

Lesson 15 – Positioning the Players



This may sound fairly simple to do but to teach you some new commands I am going to use data statements to position the characters. I want the characters to start on the gray panels and they should all face into the middle of the arena. This means I need to set the position and angle of the players. The height is not needed as this can be worked out with the 'get ground height' command.

The first thing to do is work out the positions and angles of the characters. This wasn't too hard to do. I added the following code to the main loop.

```
text 10,30,str$(object position x(1))+ " : "+str$(object position z(1))
text 10,50,str$(object angle y(1))
```

Then I just wrote down the positions and angles of the different locations. This information got placed into data statements, like this:

```
\-----
\data statements
\-----
data_player_positions:

data 262.5,212.5,0
data 212.5,262.5,90
data 262.5,312.5,180
data 312.5,262.5,270
```

The first value in each data statement is the xPosition, the second is the zPosition and the third is the yAngle. Notice the label at the top of the data statements, this is used to set the data that gets read. You can have more than one set of data for different things and this label lets you use the data whenever

you want. I don't really need it at the moment but may later so decided to add it in. Notice the name of the label begins with the word data. This isn't essential but it does tell me what the label is doing. It is there for ease of use rather than being something that has to be done.

Where do these data statements go? It doesn't really matter where you position them but I like to put them all at the very bottom of the program.

Reading the data

Before we read the data we should tell DarkBASIC what data is being used. This is what the label is used for. The command used is 'restore dataname'. Dataname is the name of the label that is just before the data statements - in this case the following is used:

```
restore data_player_positions
```

NOTE : This gets placed just before the players are created.

To position the players we have to 'read' the data values. The easiest place to do this is in the player creation loop. After the player objects have been created (just before the position object id part) we can read the data. All you have to do is use the read command followed by the variable you want to assign the data to.

```
`work out position
read xPos#
read zPos#
yPos#=get ground height(1,xPos#,zPos#)

`work out angle
read yAng

`update the player position
position object id,xPos#,yPos#,zPos#
yrotate object id,yAng
```

And that's it. The players are now positioned in the correct places and are almost ready to be raced against. Something that needs to be done first is to sort out collision between the players which is covered in the next lesson.

Lesson 16 – Player Collision



So that the players do not overlap each other some player to player collision is needed. This is actually fairly simple if using DarkBASICs object commands. Something that needs to be done first though is to turn off the collision on all the objects that we don't want to collide with (the arena, and light objects).

```
\-----  
\turn off collision  
\-----  
set object collision off 200  
set object collision off 201  
set object collision off 202  
  
set object collision off 100  
set object collision off 101
```

The reason for turning off the collision for these objects is firstly that turning it off speeds up the game, and secondly that being inside an objects bounding box counts as a collision. This would mean that being in the arena would count as a collision and would mean that the players would be unable to collide.

Setting up the player collision

As already mentioned the object collision commands are going to be used for this game. These let you set collision boxes that surround objects. They also do moving collision so if you have two moving objects the collision boxes follow and can be made to collide. By default the collision boxes are turned on but they use the size of the object before it is scaled so this needs to be set manually.

```
make object collision box id,-3,0,-3,3,4,3,0
```

NOTE : This gets placed at the end of the player creation loop.

What is this doing?

The command make object collision box makes an invisible box that surrounds the player specified by the first value. The next six values specify the size of the box. To work out the values you should imagine the object you are sizing is at the position 0,0,0 and then work out what the minimum and maximum values for each coordinate are. In this case I want the box to be 6 units wide (minimum =-3 maximum =3) and 4 units high (minimum =0 maximum=4).

The 0 at the end of the command tells DarkBASIC that box doesn't rotate with the player (a value of 1 would mean it does rotate). The obvious thing to do for player to player collision would be to use rotating boxes. This would still tell me whether there is a collision or not but I would be unable to find out how where to reposition the player so non rotated boxes it is. Once you play the game you will see that it is actually fairly hard to tell that the boxes don't rotate.

Now it's all setup...

...we need to do something with the collision. I decided to create a subroutine that controls all of the collision updates.

```
`update the collision data  
update_collision:  
  
return
```

To check for collision with every player you need to loop through all of the players checking collision (simple really). To do this use a for loop.

```
for id=1 to 4  
  
next id
```

At the start of the loop we need to grab the player position, this is used later to reposition the player after a collision has occurred. The next thing to do is check for a collision using the command 'collision=object collision(id,0)'.

In this case the the id refers to the object we are checking for a collision with. The 0 means that it should check collision against all objects in the world. Using a different number would check collision against the object specified by the number used. The value returned is assigned to the variable collision. This value is either a 0 for no collision or, if there is a collision, the value is the number of the object that is being collided with.

```
for id=1 to 4
```

```

xPos#=object position x(id)
yPos#=object position y(id)
zPos#=object position z(id)

collision=object collision(id,0)
next id

```

Now all that needs to be done is to check if a collision has happened. If it has then the player gets repositioned, if it hasn't then nothing happens.

```

For id=1 to 4
  xPos#=object position x(id)
  yPos#=object position y(id)
  zPos#=object position z(id)

  collision=object collision(id,0)

  if collision>0
    xPos#=xPos#-(get object collision x()/2)
    zPos#=zPos#-(get object collision z()/2)
  endif

  Position object id,xPos#,yPos#,zPos#
next id

```

The 'get object collision x()' and 'get object collision z()' commands tell you the amount to reposition the player so that they are not colliding. I have divided this value by 2 so that when the player you are colliding with checks collision they get repositioned as well. This isn't perfect but it gives the impression of player to player collision.

The finished subroutine looks like this:

```

`update the collision data
update_collision:

for id=1 to 4
  xPos#=object position x(id)
  yPos#=object position y(id)
  zPos#=object position z(id)

  collision=object collision(id,0)

  if collision>0
    xPos#=xPos#-(get object collision x()/2)
    zPos#=zPos#-(get object collision z()/2)
  endif

  Position object id,xPos#,yPos#,zPos#
next id

return

```

You now need to call this command from the main loop and you have player to player collision. Try driving into the other players and you will see that you actually push them across the arena, which is quite cool.

Lesson 17 – Artificial Intelligence



The artificial Intelligence (AI) for this game is 'incredibly' simple. It can be modified for just about any way point based game (racing games are a pretty good example). I was actually dreading writing the code as I couldn't see a way to make the code small and easy to understand, yet in the end it took me about three minutes to write the 30 lines of code that are the AI (of that 8 lines are spaces, 5 are comments, and 8 set the values of variables).

Anyway the first thing to do is to create the function the AI will be placed in. I chose to call the function `control_player()` because that is what it does :). The first few lines of the function are used to get the position and angle of the player.

You should recognise the variables `forward`, `backward`, `left` and `right` as the ones used for the `move_player` function. The reason these are used is that this function is basically working out what angle to turn the player. At the end of the function the `move_player` function is called to update the player position.

```
\-----  
\artificial intelligence  
\-----  
function control_player(id)  
  
    \get positions  
    xPos#=object position x(id)  
    zPos#=object position z(id)  
  
    yAng=object angle y(id)  
  
    \set movement values  
    left=0  
    right=0  
    forward=1  
    backward=0  
  
    move_player(id,forward,backward,left,right)
```



```
endfunction
```

The next thing to do is to work out the position of the target object. Once we have this we can work out the angle between the player and the target. To work out the target position I simply copied and pasted and the code that works out the position of the target in the new_target function.

```
`work out target position  
targetXPos=(FKey(hotspot(0),0)*info(0))-(info(0)/2)  
targetZPos=(Fkey(hotspot(0),1)*info(1))-(info(1)/2)
```

To work out the angle between the player and the target the math's command ATanFull is needed. To tell the truth I don't really understand what this is doing or how it works. However I do know that it does work which is all that matters. What you do is supply the difference between the x positions of the player and the target, and the difference between the z positions, and the command works out the angle separating the two. From this angle you subtract the angle of the player.

This works out what angle the player has to turn to reach the target.

```
`Work out angle between payer and target  
angle#=-atanfull(xPos#-targetXPos,zPos#-targetZPos)-yAng
```

Printing the angle to the screen told me that when the angle was -180 degrees the player faced the target, when the angle was less than -180 then the player had to turn left to reach the target, and when the value was more the player should turn right.

So with a simple if statement I can tell the player what direction to turn.

```
`Work out direction to turn  
if angle#<-180  
    left=1  
else  
    right=1  
endif
```

With all of the code put together the completed function looks like this:

```
`-----  
`artificial intelligence  
`-----  
function control_player(id)  
  
  `get positions  
  xPos#=object position x(id)  
  zPos#=object position z(id)  
  
  yAng=object angle y(id)  
  
  `set movement values  
  left=0  
  right=0  
  forward=1  
  backward=0  
  
  `work out target position  
  targetXPos=(FKey(hotspot(0),0)*info(0))-(info(0)/2)  
  targetZPos=(Fkey(hotspot(0),1)*info(1))-(info(1)/2)  
  
  `work out angle between payer and target  
  angle#=atanfull(xPos#-targetXPos,zPos#-targetZPos)-yAng  
  
  `work out direction to turn  
  if angle#<-180  
    left=1  
  else  
    right=1  
  endif  
  
  move_player(id,forward,backward,left,right)  
  
endfunction
```

To make the function update the other players you need to call the control function in the main loop. The loop:

```
for id=2 to 4  
  move_player(id,0,0,0,0)  
next id
```

Should be replaced with:

```
for id=2 to 4  
  control_player(id)  
next id
```

The players now rush towards the current active targets. You may notice that the targets do not go out when the opposition reach the target but this is easily solved in a future lesson.

Other things could be done to make the players cleverer and to make the game harder but at the moment I think the game is perhaps a little too hard anyway so

for now I will leave it. Once I have played the complete game I will be able to decide better whether the AI should be changed at all or not.

Lesson 18 – Putting it all together



Now I need to turn it into a proper game. There are still a number of things that I want to do to make the game more fun and easier to play but this lesson completes a number of things that need tidying up before I get there.

Update targets

At the moment the opponents do not create a new target when they reach the current one. This is simple to change and makes the game come one step closer to completion. At the moment I have this piece of code:

```
if checkHotspot(1)=hotSpot(0)
  new_target()
  targetCount(1)=targetCount(1)+1
  if targetCount(1)=10 then end_game()
endif
```

This is checking if player 1 is in the target hotspot. What I want it to do is to check all of the players and see if any of them are in the current target. If they are then I want to create a new target.

```
for id=1 to 4
  if checkHotspot(id)=hotSpot(0)
    new_target()
    targetCount(id)=targetCount(id)+1
    if targetCount(id)=10 then end_game()
  endif
next id
```

Now every time an opponent reaches the target a new target is picked and the opponents' score goes up by 1.

Individual players

At the moment all of the players use the same texture. This is OK as it is obvious what player you control but it would be nice to be able to have different textures for the different players if only to make the game more interesting visually.

I adjusted the basic hovercraft texture so that there were four different ones with different colours and different numbers. I considered doing totally different textures for each vehicle but I decided the game is fast enough that it doesn't really matter as no one would notice. I then named each of the textures hovercraft_1.bmp, hovercraft_2.bmp, hovercraft_3.bmp and hovercraft_4.bmp. The idea with this is that in the player load loop the textures get loaded and applied to the texture according to the player number.

I created a variable called textureName\$ and assign it a value each loop. The texture loading in the player load loop now looks like this:

```
textureName$="media/hovercraft_"+str$(id)+".bmp"

load object "media/hovercraft.3ds",10+id
load image textureName$,1
texture object 10+id,1
scale object 10+id,200,200,200
set object texture 10+id,0,1
```

Something else that would make the players more individual (and a little easier to beat) is giving them different speeds, and frictions.

Currently the data statements look like this:

```
\-----
\data statements
\-----
data_player_positions:

data 262.5,212.5,0
data 212.5,262.5,90
data 262.5,312.5,180
data 312.5,262.5,270
```

What I want to do is add the speed and friction values onto the end of these data statements. Then the part where the speed and friction values are set I will swap for read commands.

The data statements are starting to get a little confusing so I have also added a comment telling me what each number means. At the moment it is easy for me to remember the value but in the future it may be hard for me to remember what is what.

```
\-----  
\data statements  
\-----  
data_player_positions:  
  
\value 1 = x position  
\value 2 = z position  
\value 3 = y angle  
\value 4 = friction  
\value 5 = move distance  
  
data 262.5,212.5,0,0.97,0.065  
data 212.5,262.5,90,0.98,0.055  
data 262.5,312.5,180,0.975,0.06  
data 312.5,262.5,270,0.965,0.07
```

Running this without adjusting the player load code will get some strange results (the players are in unusual places).

All you need to change is the variable assignment in the player load loop. The bit where you set the player friction and move distance should be changed to this:

```
read friction#(id)  
read moveDist#(id)
```

This is a small change but you should now find that the race to the next check point is a lot closer than it was before.

Lesson 19 – The Score



So that we can work out who has won the competition it is necessary to know what the score is. The score is already stored in the `targetCount()` variable so all that really needs to be done is for the score to be displayed. Naturally this has to look nice and work fast. The best way (in my opinion) to display 2d stuff on top of 3d stuff is actually to use 3d objects. 2d on top of 3d is known to slow games down by quite a bit so it is worth doing.

Saving Time

To make things easier I am going to use a set of functions originally written by Magellan. I modified these commands to make them a bit faster and to add extra functionality. You will find that these commands have been included in a nice little include file.

The first thing to do is to include the file. At the top of the program I now have this:

```
\-----  
\ INCLUDES  
\-----  
\include the 3d sprite library  
\include "sprite.dba"  
\include the MatEdit LoadMatrix files  
\include "LoadMatrix.dba"
```

NOTE : For some reason I got an error when the sprite include was included after the loadmatrix include. I don't know why this is, and if anyone can work out why I would be interested to hear.

The way these commands work is to create a plane object and lock it to the camera. It then uses some fairly simple math's to position the sprite in 3d space. This means you can use 2d coordinates to position the objects and not worry about where the objects should be in the world.

The coordinates should all be based on the resolution 640 by 480 even if you choose to use a higher resolution. The proportions will stay the same no matter what resolution you use which is actually rather cool. What this means is that if you use a high resolution the 2d interface will still look exactly the same.

To create a 2d sprite you simply use the command `create_2d_sprite(id,width,height)`. The id is the object id and works in the same way as all the other object id's work. The width and height are the width and height of the sprite on the screen.

Positioning the sprite is just as simple. `Position_2d_sprite(id,xPos,yPos)` position the specified sprite at the specified location.

I want my player score sprites to be placed in the top left corner. So am using the following code to position my player score sprites. There is one sprite for each character so that when playing the gamer knows what score each character has.

```
make_3d_sprite(31,64,64) : position_3d_sprite(31,10,42)
make_3d_sprite(32,64,64) : position_3d_sprite(32,84,42)
make_3d_sprite(33,64,64) : position_3d_sprite(33,158,42)
make_3d_sprite(34,64,64) : position_3d_sprite(34,232,42)
```

Textures

Now the sprites need to be textured. There is a command that does this as well but first the score images need to be loaded.

I have created a series of images named `score#.bmp` where # is the value of the score. So if the players score is 0 then they will use image `score0.bmp`, and if the player score is 5 they will use `score5.bmp`.

These files need to be loaded and assigned a number. This will be done in a similar way to the way the hovercraft textures were loaded earlier.

```
for i=0 to 10
  load image "media/score"+str$(i)+".bmp",30+i
next i
```

To texture the 2d sprite you use the texture object command also I have used the ghost object command to make the sprite transparent adding an extra graphical effect to make the game look a little nicer.


```
texture object 31,30
texture object 32,30
texture object 33,30
texture object 34,30

ghost object on 31
ghost object on 32
ghost object on 33
ghost object on 34
```

NOTE : Once you have set the sprite properties you can use the normal texture object command to set the sprite texture.

Changing the score

Now that it is all loaded we need to adjust the relevant sprite when a players score increases. This takes one whole line of code that gets placed in the main loop:

```
texture object 30+id,30+targetCount(id)
```

If you remember the sprites created were given numbers that corresponded to the player ids. So player 1 has a 2d sprite numbered 30+1 (31) this makes working out what sprite to texture very easy. The same goes for the score textures. These are numbered 30+score so if score = 4 then the texture to use is 30+4=34. As targetCount stores the score value this gets added to 30 to work out what texture to use for the players score.

You what?

At the moment you know what the score is but anyone else playing the game will not know which sprite represents which character. To make it nice and easy to understand an extra set of sprites have been added. These tell the game player which colour opponent has what score.

The code is very similar to the code above so to save time you can just copy and paste the stuff from below.

```
`name tags
make_3d_sprite(41,64,32) : position_3d_sprite(41,10,10)
make_3d_sprite(42,64,32) : position_3d_sprite(42,84,10)
make_3d_sprite(43,64,32) : position_3d_sprite(43,158,10)
make_3d_sprite(44,64,32) : position_3d_sprite(44,232,10)

for i=1 to 4
  load image "media/name"+str$(i)+".bmp",40+i
  texture object 40+i,40+i
  ghost object on 40+i
next i
```