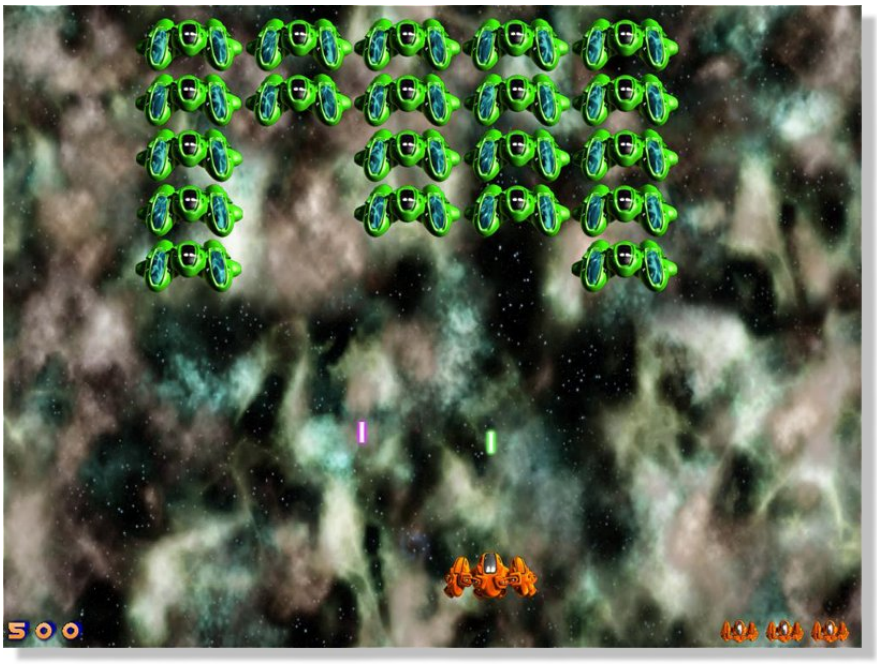
Dark Invaders

A Full Game DarkGDK Tutorial

By Dave “Ravey” Milton



Contents

The Planning Stage 2
Introduction to the Code 3
Part 1: Game Setup 4
Part2: Game Title 7
Part3: Game Levels 17
Part4: Game Play 20
Part5: Player Update 21
Part6: Enemies Update 26
Part7: Game Over Sequence 37
Part8: Customising the game further 40

As you may have deduced from the title and opening screen shot, in this tutorial we are going to make a game about alien spacecraft invading our beloved Earth. We are mankind's last line of defense – the fate of the world is held in your hands. Okay enough with selling the concept, I think most (if not all) of us are familiar with this particular style of game. The question is, how do we go about making it using the Dark GDK? let us find out...

The Planning Stage

Before we head in to the code, let us first break the game down into the various components that will build up to the finished game. First lets consider the game itself, what will it play like, what are the objectives etc.

Game Summary

The game will be set in outer space, with a scrolling backdrop.

The player will be a spaceship at the bottom of the screen and restricted to moving left and right. You will be able to fire, but only allowed to fire again when either the bullet hits it's mark and destroys an enemy, or the bullet reaches the top of the screen.

The enemies (aliens) will be in formation (5 rows of 5 ships) and will start in the top left of the screen, moving all the way to the right, at which point they will move down before heading back left and down again, then repeat. This movement will get faster as time goes on (top stop the player taking too long to finish the level) and will stop when either the player has no lives left, or they reach the bottom of the screen in which case the defense of Earth by the player has failed.
Once the enemies have moved down the screen enough, every so often a flying saucer will fly past the top of the screen from right to left offering a bonus score to the player for killing it before it goes off the left edge of the screen.

Scoring will be as follows:
Enemy Spacecraft destroyed: 100 points
Enemy Flying Saucer destroyed: 2000 points

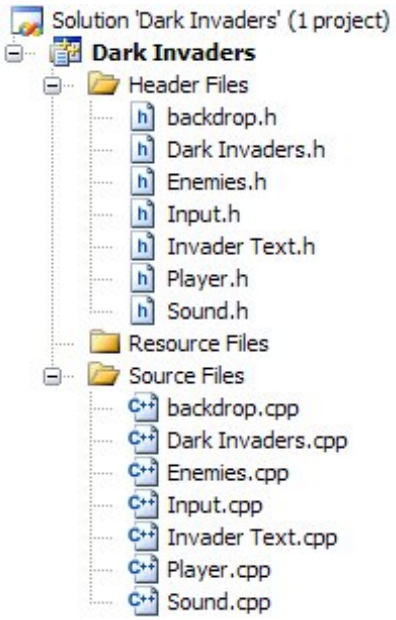
The level will end when all enemy craft have been destroyed (not including the bonus saucers). The next level will increase the initial speed of the enemies, although this could end up with enemies traveling too fast at later levels so we will cap we discover how fast is too fast.

The game will end when the player has no lives left.

Introduction to the Code

I am going to assume here that you have followed earlier tutorials and are already familiar with the basics of the Dark GDK and have a good understanding of C/C++. I won't be explaining every single line of the code, instead I will be picking out the most relevant points and introducing new commands and techniques. Also feel free to study the code itself as much as you like, the code is thoroughly documented throughout.

Before we jump into the code here is an explanation of the source files used in the project:



The files are as follows:

Dark Invaders.cpp / .h	-	Our main project files including our GDK startup code and main game loop.
Backdrop.cpp / .h	-	Handles the scrolling backdrop (well 3 scrolling backdrops as we are going to create a nice layered effect to give the illusion of depth).
Enemies.cpp / .h	-	Takes care of loading in the enemies, setting them up and updating them throughout the game.
Input.cpp / .h	-	handles all the keyboard input for our game (which is essentially left, right and fire).
Invader Text.cpp / .h	-	Our text routines that will enable us to display messages to the player on screen using our own style especially for the game.
Player.cpp / .h	-	Here we handle the players control, shooting, scoring and dying.
Sound.cpp / .h	-	Handles the loading in of all sounds used in the game.

Now we have an understanding of the basic game and what the purpose of the various files in the project are, lets think about “game modes”, in other words which different parts will make up the game experience.

Lets firstly examine the GDK loop. Open up Dark Invaders.cpp and check out the GDK function:

```
void DarkGDK ( void )
{

	// our main gdk loop
```

```
	while ( LoopGDK() )
	{
		// the game itself
		game();
		// dbSync will wait for the screen to finish drawing, and then draw our whole scene
for us
		dbSync();
	}

}
```

This is nice and simple, just a call to game() which takes care of everything that happens in the game, then our all important call to dbSync() which draws everything for us.

Above this function you can see the following lines:

```
enum  eMode { eGameSetup , eGameReset , eGameTitle , eGameWaitForFire , eGameLevel , eGameLevelWait
, eGameLevelWait2 , eGamePlay , eGameDie , eGameWin , eGameOver };
eMode g_eGameMode = eGameSetup;

int g_iLevel = 1;

int g_iLevelStartTime = 0;
```

The enum is the most interesting here, this details every mode our game will go through and is used in the game() function as follows:

```
void game ( void )
{

	// lets find out which game mode we are in and call the appropiate routine
	// from here we can easily see the different parts that make up the game
	switch ( g_eGameMode )
	{
	// initial setting up of the game
	case eGameSetup:
		gameSetup(); break;

	// resetting the game after the player looses a life, or before a new game starts
	case eGameReset:
		gameReset(); break;

	// display the main title screen
	case eGameTitle:
		gameTitle(); break;

	// wait for the player to press fire to start
	case eGameWaitForFire:
		gameWaitForFire(); break;

	// display the new level screen
	case eGameLevel:
		gameLevel(); break;

	// wait for player to press fire to start the level
	case eGameLevelWait:
		gameLevelWait(); break;

	// animate the ship flying off to the new level
	case eGameLevelWait2:
		gameLevelWait2(); break;

	// the game itself!
	case eGamePlay:
		gamePlay(); break;

	// player has completed a level!
	case eGameWin:
		gameWin(); break;

	// player has lost a life
	case eGameDie:
		gameDie(); break;

	// player has lost all lives, Game Over!
	case eGameOver:
		gameOver(); break;
	}

	// call our scrolling routine to update the backdrops
	backdropUpdate();

}
```

Let’s look at what each of these game modes will do in the final game step by step:

Part 1: gameSetup()

Part 2: gameTitle() and gameWaitForFire()

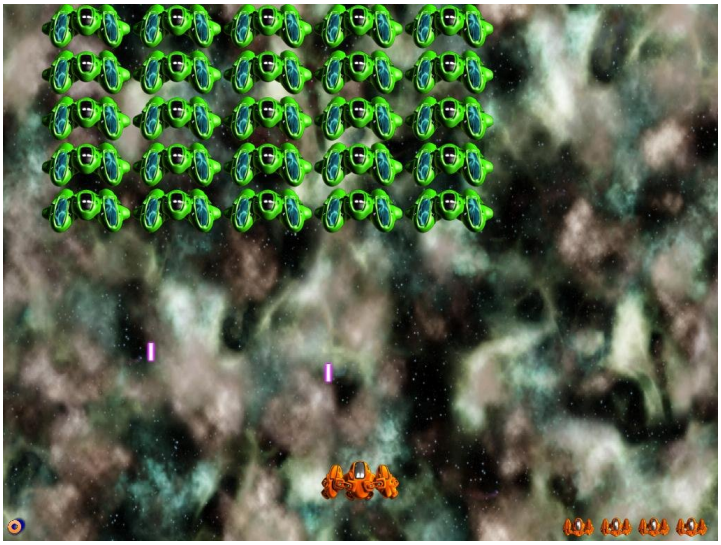
Part 3: gameLevel(), gameLevelWait() and gameLevelWait2()

Part 4: GamePlay() Introduction

Part 5: gamePlay() - updatePlayer()

Part 6: gamePlay() - updateEnemies()

Part 7: gamePlay () - gameWin(), gameDie() and gameReset()



Part 1: Game Setup

gameSetup()

Here is the code involved:

```
	// set the display to 1024 x 768 with 32bit depth
	dbSetDisplayMode ( 1024 , 768 , 32 );
	// turn sync on, so our game waits for the screen to finish drawing before displaying, also
handles backbuffer    automatically for us
	dbSyncOn();
	// lets run at a nice 60 frames per second - automatically!
	dbSyncRate(60);
	// full screen windowed mode
	dbMaximiseWindow();

	// set up our text drawing routine
	invaderTextSetup();
	// set up out backdrop routine
	backdropSetup();
	// set up the player graphics
	playerSetup();
	// set up the sounds we will use
	soundSetup();

	// switch to game title mode
	g_eGameMode = eGameTitle;
```

Firstly we are setting the display mode to the resolution we want to use for the game – dbSetDisplayMode (width, height , bit depth). All parameters for display mode are intergers. We turn Sync to on and set a frame rate of 60 using dbSyncRate (60);.

Next up we Maximise the window so it covers our full display – I always comment this line out while developing (and usually lower the resolution too) so I can debug easier rather than the game taking up every inch of display space.

The next four function calls are to our own game setup functions, lets take a closer look at those:

InvaderTextSetup()
Not much to this routine, one call to dbLoadImage which loads in a bitmap image that we will use to display our text on screen:

```
dbLoadImage ( "Assets\\font.png" , 100 );
```

The format for dbLoadImage is: dbLoadImage (char * filename , int imageID);

The image ID is the number we want associated with that image, so anytime we wish to use it we use its associated ID number. The reason I have chosen 100 for this will become apparant as we go on (valid image ID numbers start from 1).

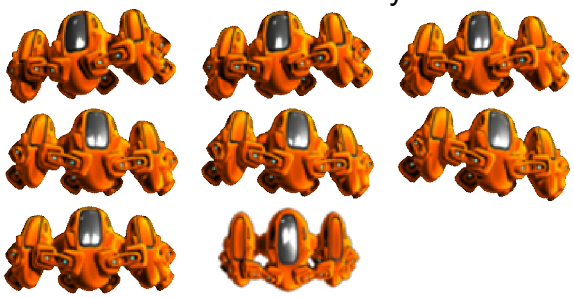
backdropSetup()
With our backdrop, we are going to have 3 images laid over one another, all travelling at different speeds to give a nice “parallax” scrolling effect. This routine loads in the 3 images we need and stores them in image ID's 50, 52 and 54. The reason I am choosing these strange numbers is to keep a relationship between the graphics on screen and the images. In this case to create a screen that scrolls down, we need to draw one big image moving down the screen, and a duplicate image above it, when the first image reaches the bottom the second will be exactly where the first image was at the very start, so at that point we can simple reposition both images back and start again. I used the term “image” there but what we will actually be using to display these images are “sprites” which can be thought of as picture frames that we can move (size and rotate also) around the screen. The

image is the picture displayed in the picture frame.

PlayerSetup()
The bottom half of this function is loading in some images for the explosions, bullets and lives display, but the top half of function has some interesting new commands:

```
dbCreateAnimatedSprite ( 1 , "Assets\\ship_anim.png" , 3 , 3 , 1 );
dbSprite ( 1 , iPlayerX , iPlayerY , 1 );
dbSetSpriteFrame ( 1 , iPlayerFrame );
dbSetSpritePriority ( 1 , 2 );
```

dbCreateAnimatedSprite (int SpriteID , char* filename , int width , int height , int imageID)
This command does a lot of work for us. It loads in an image (filename), assigns it an ID (imageID) then “cuts” the image into animation frames for us. How does it do this? The parameters we passed in for width and height reveal how many rows and collums are in our picture – in the case of our “ship_anim.png” we have 3 rows by 3 collums. Here is the image we loaded in so you can individual frames for yourself:



ship_anim.png

The next line is the stanard line for displaying a sprite:

```
dbSprite ( 1 , iPlayerX , iPlayerY , 1 );
```

Which creates (or repositions) a sprite to the passed position (X and Y coordinates). The last paremeter is the image that the sprite will show (remember my previous example of how the sprite was the frame and the image is the picture in that frame. In this case, we have created sprite 1 (the player) as an animated sprite, so setting it to use image 1, will not show the whole image but a frame of it. We can tell it which frame to display by using:

```
dbSetSpriteFrame ( 1 , iPlayerFrame );
```

Frame numbers count from 1 upwards. In our game, iPlayerFrame is initially set to 7, so this will display the ship centered in the middle (the last frame is blank as you can see from the picture above, frame 7 is the penultimate frame).

The last command we use here is

```
dbSetSpritePriority ( 1 , 2 );
```

This affects the order the sprite is drawn. By default all sprites are given the priority of 0, and will be drawn in the order that they were initially created. In our game we have 3 scrolling backdrops thats we want to be drawn first, so we set the ship to priority 2, to be above the scrolling backdrops, the bullets we will use for the player and enemies and the enemies themselves.

SoundSetup()
This routine loads in all the sounds we will be using throughout the game using the same command for each sound:

```
dbLoadSound ( char* filename , int iSoundID );
```

Sounds, like images and sprites are assigned and referenced by an ID number, which as usual can be upwards from 1.

The last part of the routine switches the game mode to gameTitle() so lets tackle that one next

```
	// switch to game title mode
	g_eGameMode = eGameTitle;
```

Part 2: Game Title

gameTitle()
This is the part of the game where we simply display our title and play a sound

```
	// call our custom text routine to display our title text
	InvaderText ( 0 , 340 , 32 , "DARK INVADERS" , true , false );
	InvaderText ( 0 , 400 , 16 , "PRESS FIRE TO START" , true , false );
	// change the speed of background sound to give a different sound
	dbSetSoundSpeed ( 1 , 2000 );
	dbPlaySound ( 1 );
	// change the speed of backdrop scrolling again
	g_fBackdropSpeed = 2.0f;
	// set players starting lives to 5
	g_iPlayerLives = 5;

	// reset player statistics and reset the level to 1 for a new game
	g_bPlayerWin = false;
	g_bEnemiesSpawned = false;
	playerReset();
	g_iPlayerScore = 0;
	g_iLevel = 1;

	// wait for the user to press fire
	g_eGameMode = eGameWaitForFire;
```

First couple of lines here are using our InvaderText routine to display some text on the screen. If you recall we loaded in an image earlier:

```
dbLoadImage ( "Assets\\font.png" , 100 );
```

now it's time to put it to good use!

Lets have a look in Invader Text.cpp and see what this text business is all about.

```
void InvaderText ( int iX , int iY , int iSize , char* czText , bool bHorizonalCenter = false , bool
bVerticalCenter = false )
```

iX and *iY* are the position we want to display the text on screen.
iSize is the size in pixels we want each letter to be (each character is square in our font so its iSize * iSize).
czText is the text we would like to display.
bHorizontalCenter – if this is set to true then the text will be centered horizontally.
bVerticalCenter – same as above but for the vertical center.

Let's have a look at the routine then:

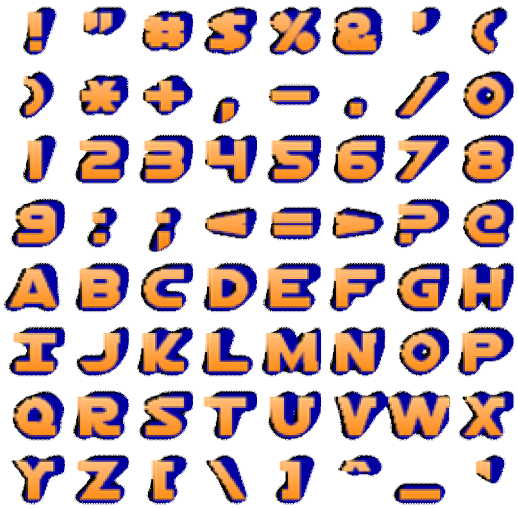
```
void InvaderText ( int iX , int iY , int iSize , char* czText , bool bHorizontalCenter = false ,
bool bVerticalCenter = false )
{
	// if bHorizonalCenter is true then work out the x coordinate to make the text centered
	if ( bHorizonalCenter )
	{
		iX = (int)(( dbScreenWidth() / 2 ) - ( ( strlen( czText ) * iSize ) / 2 ));
	}

	// if bHorizonalCenter is true then work out the y coordinate to make the text centered
	if ( bVerticalCenter )
	{
		iY = ( dbScreenHeight() / 2 ) - ( iSize / 2 );
	}

	// loop through and draw each character
	for ( int i = 0 ; i < (int)strlen( czText ) ; i++ )
	{
		if ( czText[i] == 32 )
			iX += iSize;
		else
		{
```

```
                        dbSprite (  iNextTextSprite , iX , iY , 100 );
                        // call our routine to change the texture coordinates of the sprite manually
to show the correct                         character
                        fixUV( czText[i]);
                        dbSizeSprite ( iNextTextSprite , iSize , iSize );
                        iX += iSize;
                        iNextTextSprite++;
                }
        }
}
```

First part of the routine is seeing if we need to center horizontally and/or vertically and using new commands for us to work out where the center is.

```
int dbScreenWidth()
int dbScreenHeight()
```

Both these commands return the width and height respectively of our game screen. If you game is running full windowed at a resolution of 640 x 480 and your desktop is 1920x1200 dbScreenWidth() and dbScreenHeight() will still return a resolution of 640x480. I could of just used 1024 and 768 because I know thats the resolution we are running the game at – but that may change in the future (maybe we will give the player the ability to change resolutions) so it is safer to use dbScreenWidth() and dbScreenHeight() to be safe.
The second part of the routine is looping through our string, firstly checking if it is ASCII character 32 which is a space – if it is, we just increase iX (the variable looking after our current x position) to make the gap. If the character is not a space however we create a new sprite, position it at the current x and y position and assign the font image we loaded earlier to it which is 100.



Lets have a look at the image we are using for displaying text:
Ah okay, so from what we did earlier we could of just made an animated sprite..right?
Right! However, we have done that already, so I thought I would use a different technique this time to show you an alternative way to poke around with what the final sprite will look like. After we created the sprite there is a line underneath called:

```
                        fixUV( czText[i]);
```

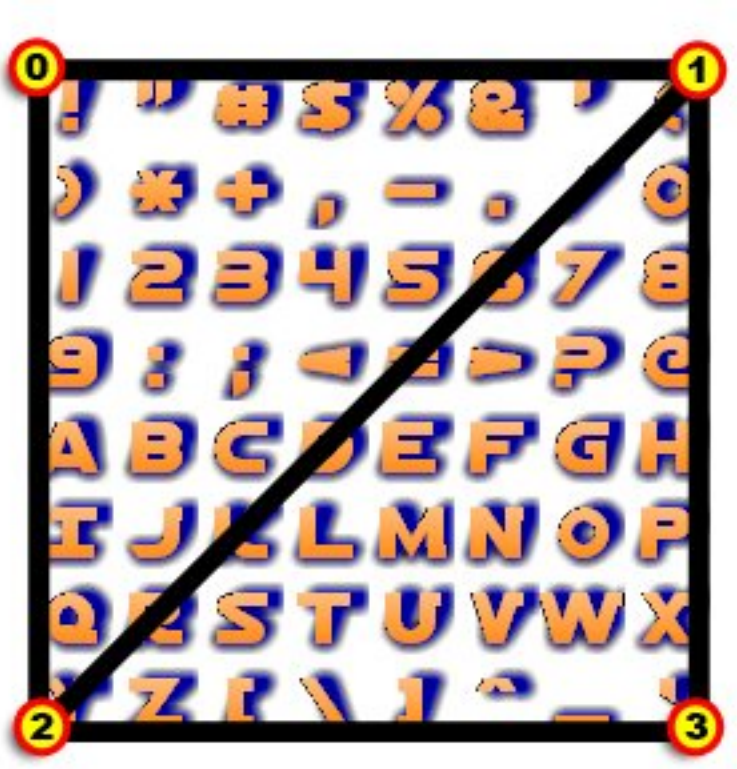
Lets have a look at the code for this function:

```
void fixUV( char c )
{
        // we have the ascii code of the letter to draw, we dont have the entire character set in
our font image however
        // so we offset it to line up with our image
        c -= 33;
```

```
        float U = 0.0f;
        float V = 0.0f;

        int iY = c / 8;
        int iX = c % 8;

        float fOffset = 1.0f / 8.0f;

        U = iX * fOffset;
        V = iY * fOffset;

        // 4 vertices in a sprites (quad), we adjust the U and V ( think x and y) coordinates of the
texture to show only the letter we want
        // rather than the whole texture
        dbSetSpriteTextureCoord ( iNextTextSprite , 0 , U , V );
        dbSetSpriteTextureCoord ( iNextTextSprite , 1 , U + fOffset , V );
        dbSetSpriteTextureCoord ( iNextTextSprite , 2 , U , V + fOffset );
        dbSetSpriteTextureCoord ( iNextTextSprite , 3 , U + fOffset , V + fOffset );
}
```

This introduces us to UV Coordinates. What are UV Coordinates you are probably wondering.
Firstly, one thing I should explain, is that although our game is what would be called “2D”, our sprites in GDK are actually 3D models, very flat looking models granted, but 3D none the less. Each sprite is infact made up of two triangles forming a square which makes our sprite. A triangle is made up of three points, or “vertices” which describe the triangle. In the case of our sprites, 4 vertices are used which the 2 triangles share.

Below is a picture of our font image to make it all become clearer:



The black lines show where our two triangles are. The yellow numbers circles are our vertices, numbered 0 through to 3. What has this got to do with UV Coordinates, I was just getting to that, but first an example to illustrate the point:

We know that 3D models are made up of triangles, and also that they are “textured”, in other words they have an image that appears to wrap around them, in many ways like our skin wraps around our bodies. Think of an easter egg. The chocolate egg has a foil

wrapper around it. Carefulling peel off that wrapper and you can flatten it out to show a 2 dimensional picture. That picture was once wrapped around the egg and become the same shape as the egg itself. That is the way it works in 3D, we take our 3D mesh (the egg – although nowhere near as fattening) and wrap our image around it so it takes on the shape of the mesh. How do we do this? We give each vertices an X and Y coordinate which relates to a part of the image. These X and Y coordinates are infact called UV coordinates, where the U is the X and the V is the Y. They are called UV to make it clear we are talking about texture coordinates and not any other type of coordinates.

One main differences with UV Coordinates and standard coordinates is, to overcome the problem that the image could be any size, and we could change the image at any time, both U and V go from 0 to 1 (so they are naturally floating point values). With U – 0 would be the left of the image and 1 would be the right. Similaly with V the 0 would represent the top and 1 the bottom.

We don't have to worry about this with sprites most of the time – all this UV business is automatically set up for us. Looking at the top example we can see that the vertices are set up to cover the entire image, so the UV coordinate of the above image could be described as:

Vertex 0: *U = 0.0 , V = 0.0*
Vertex 1: *U = 1.0 , V = 0.0*
Vertex 2: *U = 0.0, V = 1.0*
Vertex 3: *U = 1.0, V = 1.0*

Using this knowledge and the fact we know which character we need to display, we could change these values for each sprite to display a specific portion of the image (and even do crazy things like show it back to front and upside down). The FixUV() function does exactly that, and changes the UV coordinates of the current sprite we are using for text to display only the character we want.

How do we change the UV Coordinates? Like this:

```
dbSetSpriteTextureCoord ( iNextTextSprite , 0 , U , V );
dbSetSpriteTextureCoord ( iNextTextSprite , 1 , U + fOffset , V );
dbSetSpriteTextureCoord ( iNextTextSprite , 2 , U , V + fOffset );
dbSetSpriteTextureCoord ( iNextTextSprite , 3 , U + fOffset , V + fOffset );
```

dbSetSpriteTextureCoord() allows us to change the texture coordinates and is defined as follows:

dbSetSpriteTextureCoord(int SpriteID , int Vertex, float U , float V)

Go grab a cup of tea if you need to, when you are ready will will head back out to continue with our GameTitle() function:

```
// change the speed of background sound to give a different sound
dbSetSoundSpeed ( 1 , 2000 );
dbPlaySound ( 1 );
// change the speed of backdrop scrolling again
g_fBackdropSpeed = 2.0f;
// set players starting lives to 5
g_iPlayerLives = 5;

// reset player statistics and reset the level to 1 for a new game
g_bPlayerWin = false;
g_bEnemiesSpawned = false;
playerReset();
g_iPlayerScore = 0;
g_iLevel = 1;
```

```
        // wait for the user to press fire
        g_eGameMode = eGameWaitForFire;
```

Welcome back! Next in gameTitle() we are going to play sound 1 which we loaded in earlier, but for the title we are going to slow the sound right down to give a different effect (and also appear to have more sounds than we actually have).

dbSetSoundSpeed (int soundID , int Frequency);

soundID is the ID of the sound we loaded in earlier (sound 1 in our case) and frequency is the sound “speed” we want to play at (in hz).

We are also changing the speed of scrolling here for the backdrop (which we will come onto later) and setting various properties for the play and enemies alike (for example making sure the starting level is 1).

PlayerReset() is also called here which does the following:

```
void playerReset ( void )
{
        iPlayerX = 448;
        iPlayerY = 650;
        iPlayerSpeed = 5;
        iPlayerFrame = 7;
        iFrameAnimateDelay = 0;
        dbSprite ( 1 , iPlayerX , iPlayerY , 1 );
        dbSetSpriteFrame ( 1 , iPlayerFrame );
        g_bPlayerHit = false;
}
```

What we are doing here is resettting the player back to the starting position, setting the players frame to 7 (our facing forward frame) and setting other variables relevant to the player.

Back to gameTitle and all that is left there is passing control over to the next function which simply waits for the “fire” button to be pressed:

```
        // wait for the user to press fire
        g_eGameMode = eGameWaitForFire;
```

Which brings us nicely (if predictably) onto gameWaitForFire()

gameWaitForFire()

Below is the code for the function:

```
// wait for fire to be pressed
void gameWaitForFire ( void )
{

        if ( checkFire() )
        {
                // once fire is pressed update the player score using our custom text
                playerUpdateScore();
                // switch to display the level screen
                g_eGameMode = eGameLevel;
        }
}
```

Very small function that calls one of our input functions checkFire(). All the input functions are very similar so lets cover them all now. The player is able to perform 3 tasks: move left, move right and fire. In Input.cpp these three tasks can be checked for, here is the

code for them all:

```
// check the keys we have assigned to be the fire key
bool checkFire ( void )
{

	if ( dbKeyState ( 57 ) || dbKeyState ( 29 ) || dbKeyState ( 157 ) )
		return true;

	return false;
}

// check the keys we have assigned to be the left key
bool checkLeft ( void )
{

	if ( dbKeyState ( 203 ) || dbKeyState ( 30 ) )
		return true;

	return false;
}

// check the keys we have assigned to be the right key
bool checkRight ( void )
{

	if ( dbKeyState ( 205 ) || dbKeyState ( 32 )  )
		return true;

	return false;
}
```

The main command that is used here is dbKeyState (int iKey). What this does here is check if the key is currenly depressed and return true if it is, or false if it is not. The parameter than dbKeyState() accepts is a scancode rather than an ASCII value. In our routines for the keys we have allowed some flexibility of which keys can be used to perform the tasks.

For “fire” we have allowed “space”, “left control” and “right control”
For “left” we have allowed “a” or “left arrow key”
for “right” we have allowed “d” or “right arrow key”

These three functions will return true if any of the keys allowed are currently pressed, or false if not, allowing us to easily customize which keys perform what operation if we wanted to give the player the option to change the control setup.

Back to gameWaitForFire() and once “fire” has been pressed we update the player score (which when we start wont be a huge score, infact it will be zero) and pass control over to gameLevel().

Before we deal with gameLevel() lets look at how we update the player score:

```
void playerUpdateScore ( void )
{
	// first clear up any text shown
	invaderTextClear();
	char czScore[20];
	sprintf ( czScore , "%d" , g_iPlayerScore );
	// display the score
	InvaderText ( 0 , 730 , 32 , czScore , false , false );
}
```

We firstly call a new routine invaderTextClear() which cycles through all the sprites we have for text display. Then we show a new text line which displays the players current score.

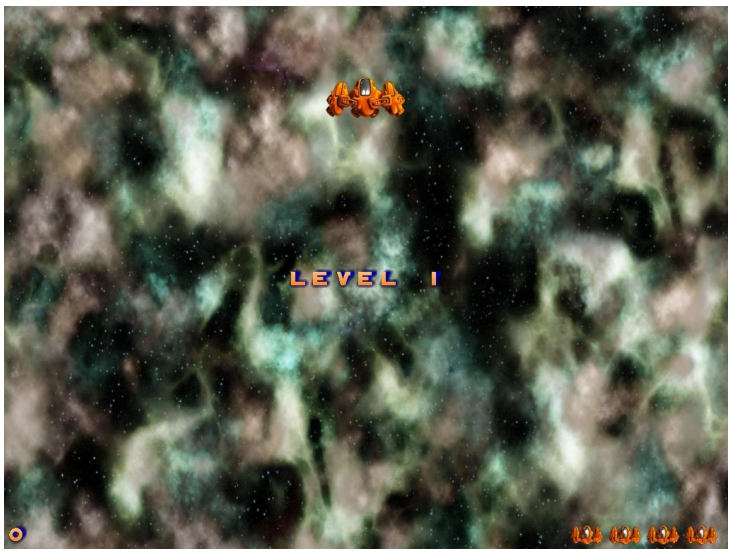
Here is the invaderTextClear() routine:

```
// delete all text sprites we have made
void invaderTextClear()
{
	for ( int i = 100 ; i < iNextTextSprite ; i++ )
	{
		if ( dbSpriteExist ( i ) )
			dbDeleteSprite ( i );
	}

	iNextTextSprite = 100;
}
```

You may recall I mentioned earlier that there was a reason I loaded in the image for text using image ID 100? Well this was the reason: Because we are using ID number we need to make sure we don't ever overlap or use ID numbers that are already in use. An easy way to do this is to think ahead about how many sprites you will need, what their different relationships are, and group together the ones that are related using a certain ID range. For this game, all the text is from sprite 100 onwards, up until iNextTextSprite which we are keeping updated so we don't have to remember. I loaded in the image for the font at 100 also just to keep it uniform. I normally try and match my image ID numbers to the same (or start of the range of) ID numbers as the sprites.

Anyway, back to the function, we are simply looping through, and checking if each sprite does indeed exist (dbSpriteExist(int iID)) before deleting it with dbDeleteSprite(). Once we have deleted all our next we can reset iNextTextSprite back to 100.

with control passed over to gameLevel() we better head over there next.



Part 3: Game Levels

gameLevel()

Here is the code for gameLevel:

```
// display the level screen
void gameLevel ( void )
{
	// use our custom text routine to display the level
	char czLevel[20];
	sprintf ( czLevel , "LEVEL %d" , g_iLevel );
	InvaderText ( 0 , 340 , 32 , czLevel , true , true );
	// change the sound speed of sound 1 again to give a different twist on the sound
	dbSetSoundSpeed ( 1 , 8000 );
	dbPlaySound ( 1 );
	// change the scrolling speed of the backdrops
	g_fBackdropSpeed = 8.0f;
	// show the player lives on screen
	playerShowLives();

	// wait for the user to press fire
	g_eGameMode = eGameLevelWait;
}
```

Looks like we have seen everything before in this routine apart from playerShowLives(). What we are doing is displaying which level we are on, changing the speed of sound 1 yet again for another effect and changing the backdrop scrolling speed again. Let us have a look at playerShowLives() and maybe after that we can finally find out what this backdrop scrolling is all about.

```
// display how many lives the player has remaining on screen
void playerShowLives ( void )
{
	// firstly clean up any sprites we have already used for this
	for ( int i = 0 ; i < 5 ; i++ )
	{
		if ( dbSpriteExist ( 40 + i ) )
			dbDeleteSprite ( 40 + i );
	}

	// now make the right amount of sprites for how many lives we have
	int iX = 1024-70;
	for ( int iLife = 0 ; iLife < g_iPlayerLives-1 ; iLife++ )
	{
		dbSprite ( iLife + 40 , iX , 768-35 , 40 );
		iX -= 54;
	}
}
```

What we are doing in this function is firstly checking if we have any existing sprites displaying how many lives are remaining for the player (using sprite ID 40 and upwards for this) and deleting them, then creating the right number of sprites compared to lives – but minus 1, because the display is showing how many lives we have in “reserve” so once we are down to the very last life none should be displayed.

Now lets have a look at the backdrop scrolling at last:

```
void backdropUpdate( void )
{
	// first the bottom one - we draw it first so it appears underneath the others
	// first lets increase the scroll value
	fBackDropBottomScrollY += 0.25f*g_fBackdropSpeed;

	// if we have reached the bottom of the screen, reset it to the top
	if ( fBackDropBottomScrollY >= 768.0f )
		fBackDropBottomScrollY = 0.0f;

	// the main sprite
```

```
	dbSprite ( 50 , 0 , (int)fBackDropBottomScrollY , 50 );
	// second sprite is draw above the first, so when the sprite scrolls down we still see the
backdrop following on
	dbSprite ( 51 , 0 , (int)fBackDropBottomScrollY-768 , 50 );

	// now the middle layer
	// first lets increase the scroll value
	fBackDropMiddleScrollY += 1.0f*g_fBackdropSpeed;

	// if we have reached the bottom of the screen, reset it to the top
	if ( fBackDropMiddleScrollY >= 768.0f )
		fBackDropMiddleScrollY = 0.0f;

	// the main sprite
	dbSprite ( 52 , 0 , (int)fBackDropMiddleScrollY , 52 );
	// second sprite is draw above the first, so when the sprite scrolls down we still see the
backdrop following on
	dbSprite ( 53 , 0 , (int)fBackDropMiddleScrollY-768 , 52 );

	// now the top layer
	// first lets increase the scroll value
	fBackDropTopScrollY += 2.0f*g_fBackdropSpeed;

	// if we have reached the bottom of the screen, reset it to the top
	if ( fBackDropTopScrollY >= 768.0f )
		fBackDropTopScrollY = 0.0f;

	// the main sprite
	dbSprite ( 54 , 0 , (int)fBackDropTopScrollY , 54 );
	// second sprite is draw above the first, so when the sprite scrolls down we still see the
backdrop following on
	dbSprite ( 55 , 0 , (int)fBackDropTopScrollY-768 , 54 );
}
```

All we are doing here is pairing sprites together for each of the three layers and moving them down the screen at their respective speeds. The very back is the slowest, the middle medium and the top layer is the fastest to achieve that parrallax scrolling look. Notice the speed of each sprite is multiplied by g_fBackdropSpeed. If you refer back to functions like gameTitle() these functions changed the variable to give us a different speed of scrolling dependant on what mode of the game we were in. For instance in this function we are current going through – gameLevel(), we have set the speed to 8.0f.

Next we move control to gameLevelWait() so let's go straight there.

gameLevelWait()

In this function we are waiting on one thing, the player ship which was sitting at the bottom of the screen now flies up and beyond the top of the screen. Once the ship has gone out of view, control passes to gameLevelWait2()

The code:

```
// move the player up the screen until it passes out of view
void gameLevelWait ( void )
{
	// has the player gone out of view?
	if ( playerLevelStart() == true )
	{
		// yes so switch to second stage
		g_eGameMode = eGameLevelWait2;
	}
}
```

playerLevelStart() is the only function called here, so lets see what it does:

```
// move player ship past the top of the screen
bool playerLevelStart ( void )
{
	dbMoveSprite ( 1 , 8 );
```

```
        if ( dbSpriteY( 1 ) <= 0 )
        {
                dbSprite ( 1 , iPlayerX , 768 , 1 );
                return true;
        }
        else
                return false;

}
```

As you can see, this is moving our trusty player 1 sprite from its normal position all the way past the top of the screen and returning true once it has done so.

Swiftly on to gameLevelWait2()...

gameLevelWait2()
This is a similar function to the previous; instead of moving the spaceship off the top of the screen, this time we move it from beneath the screen back to the players normal position all ready to let the fight commence.

Here is the code:

for gameLevelWait2()

```
// move the player from the bottom back to its normal position
void gameLevelWait2 ( void )
{
        // is the player in position?
        if ( playerLevelStart2() == true )
        {
                // yes so change the backdrop scrolling for the speed we want "in game"
                g_fBackdropSpeed = 1.0f;
                // update player score
                playerUpdateScore();
                // loop our atompsheric sound effect
                dbLoopSound ( 5 );
                // Lets store when the level starts so we can speed things up after a while to punish
them for taking              their time
                g_iLevelStartTime = dbTimer();
                // lets go play
                g_eGameMode = eGamePlay;
        }
}
```

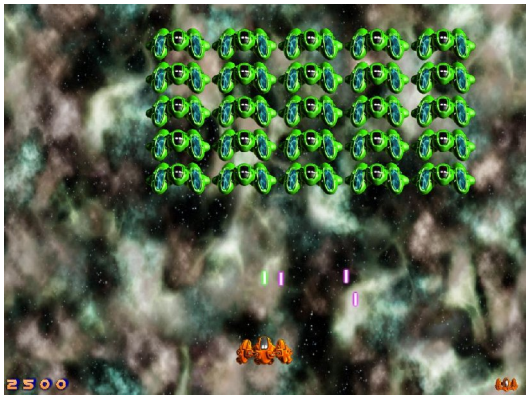
Notice we are updating the players score also, and starting our “in game” atmospheric sound. Why are we displaying the players score when we did it earlier? Once the player completes a level, the functions gameLevel(), gameLevelWait and gameLevelWait2() will be called upon once more to perform, and next time the players score will have increased.

Notice the new command near the bottom:

```
                g_iLevelStartTime = dbTimer();
```

dbTimer() returns how much time has passed since the computer has been on. The number returned is an integer and represents the time in miliseconds (1000 miliseconds = 1 second). We can use this later on to gauge how much time has passed in the currently playing level.

Once playerLevelWait2() has concluded it passes on to the where the real stuff happens – the actual game! Hold on to your seats, we are heading into gamePlay().



Part 4: Game Play

gamePlay()

So here we are in gamePlay(), it all happens in here! Firstly lets look at the code for the gamePlay() function:

```
void gamePlay ( void )
{

	// update the player
	playerUpdate();

	// update the enemies
	enemiesUpdate();

	// if the player has killed all enemies go to the level win screen
	if ( g_bPlayerWin )
		g_eGameMode = eGameWin;

	// if the player has been hit, hide the player sprite and go to the die routine to show the
player exploding
	if ( g_bPlayerHit == true )
	{
		// delete any enemy bullets so the player doesnt return to the game and get killed
instantly
		for ( int iBulletSprite = 3 ; iBulletSprite < 6 ; iBulletSprite++ )
		{
			if ( dbSpriteExist ( iBulletSprite ) )
				dbDeleteSprite ( iBulletSprite );
		}
		dbHideSprite ( 1 );
		g_eGameMode = eGameDie;
	}
}
```

So looking through the function we can see we are updating the player, then the enemies, checking to see if the player has cleared the level of all enemies (if so handing over control to gameWin()) and finally checking if the player has been hit. If the player has indeed been hit we delete all enemy bullets (we will come to those shortly), hide the player sprite and switch to gameDie() where we will animate an explosion at the location of our now deceased player.

Now we have cleared up what the bottom part of the function does, let's have a look at what the more interesting functions playerUpdate() and enemiesUpdate() do (We will cover gameWin() and gameDie() a bit later.):

Part 5: Player Update

playerUpdate()

Right then, let's have a look at playerUpdate() and see what that is all about:

```
// update the player in game
void playerUpdate ( void )
{
	// see if the player ship needs to try and go left
	if ( checkLeft() )
	{
		iPlayerX -= iPlayerSpeed;
		if ( iPlayerX < 0 )
			iPlayerX = 0;
		iPlayerDirection = LEFT;
	}
	// see if the player ship needs to try and go right
	else if ( checkRight() )
	{
		iPlayerX += iPlayerSpeed ;
		if ( iPlayerX > 1024 - 128 )
			iPlayerX = 1024 - 128;
		iPlayerDirection = RIGHT;
	}
	// not left or right, so lets face straight
	else
		iPlayerDirection = STRAIGHT;

	// animate the player
	playerAnimate();
	// check for player firing and update firing if a bullet is already in action
	playerUpdateFire();

	// place the player sprite at it's correct location
	dbSprite ( 1 , iPlayerX , iPlayerY , 1 );
	// set the frame the player sprite should show
	dbSetSpriteFrame ( 1 , iPlayerFrame );
}
```

Firstly we are checking if the player has pressed “left” by a call to our input routine we discussed earlier checkLeft(), if we have indeed pressed “left” then we deduct the player speed variable from the current X position of the player. If we haven run out of screen to head left, we simply reset the X position of the player to the minimum allowed for the game which is 0.

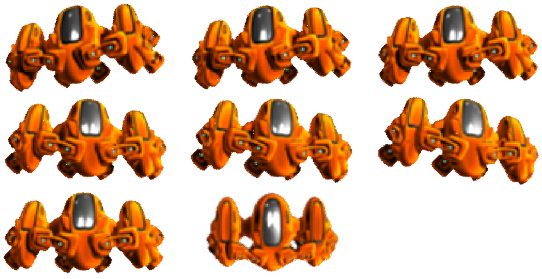
Below is the routine for checking to the right, very similar, although one difference to not is when we are checking if the ship has reached the edge of the screen we are subtracting the width of the ship from the check because a sprites X and Y positions are taken from their top left, not their center.

If we are not heading left or right we must be standing still (makes sense right) so we set the player to face STRAIGHT.

Now for two important routines for the player: playerAnimate() and playerUpdateFire().

playerAnimate()

```
void playerAnimate ( void )
{
	// firstly increment the animation delay
	iFrameAnimateDelay++;
		if ( iFrameAnimateDelay > 2 )
		{
			// reset the delay and lets go animate
			iFrameAnimateDelay = 0;
			switch ( iPlayerDirection )
			{
			case STRAIGHT:
				{
					// if we were heading left
					if ( iPlayerFrame < 3 )
						iPlayerFrame++;
					// if we were heading right
					else if ( iPlayerFrame > 4 && iPlayerFrame < 7 )
						iPlayerFrame--;
					else iPlayerFrame = 7;

				} break;
			case LEFT:
				{
					// lets animate to lean to the left
					if ( iPlayerFrame == 7 )
						iPlayerFrame = 3;
					else if ( iPlayerFrame > 1 )
						iPlayerFrame--;
				} break;
			case RIGHT:
				{
					// lean the ship to the right
					if ( iPlayerFrame == 7 )
						iPlayerFrame = 4;
					else if ( iPlayerFrame < 6 )
						iPlayerFrame++;
				} break;
			}
		}
```



```
}
```

To understand this easier lets refresh ourselves of the player ship image:

So STRAIGHT is frame 7 (since frame numbers start from one), so to straighten up we need to either add to the frame if its less than 3 to return from the left, then switch to frame 7 – or if the frame is greater than 4 and less than 7 we need to subtract the frame count until we reach 4 and then switch to 7 to level the ship up from previously heading right.
To head LEFT from straight we change the from to 3 from 7 and subtract until we reach 1.
To head LEFT from RIGHT we need to just subtract until we reach 1.

To head right from STRAIGHT we change the frame to 4 and add to the frame until we

reach 6. To head RIGHT from previously being left we simply add to the frame until we get to 6.

Notice we don"t animate the ship every single frame of the game itself. At the very start of the function we increment a delay counter. So the animation of the ship happens every 4 frames (any longer and it would start looking very jerky, much smaller and due to the amount of frames we are using the animation would be so quick you would not appreicate it).

So that's player movement sortid out, lets head to playerUpdateFire():

playerUpdateFire()

```
void playerUpdateFire ( void )
{
	// are we already firing? if so lets update
	if ( bPlayerIsFiring )
	{
		iPlayerBulletY -= 8;
		dbSprite ( 2 , dbSpriteX ( 2 ) , iPlayerBulletY , 2 );
		// have we hit anything?
		playerCheckHitEnemies();
		if ( iPlayerBulletY < 1 )
		{
			dbDeleteSprite ( 2 );
			bPlayerIsFiring = false;
		}
	}

	// if we arent currently shooting and have pressed fire - lets shoot a bullet
	if ( checkFire() == true && bPlayerIsFiring == false )
	{
		bPlayerIsFiring = true;
		dbPlaySound ( 2 );
		iPlayerBulletY = iPlayerY-32;
		dbSprite ( 2 , iPlayerX + 48 , iPlayerBulletY , 2 );
		dbSetSpritePriority ( 2 , 1 );
	}
}
```

At the bottom of the function we check if we are not already firing and have pressed fire – we can spawn a bullet. We play a sound (ID 2) to mark the arrival of our new bullet and set it off from the players middle position. We give the sprite a priority of 1 so it will be underneath the players ship, but still above the scrolling backdrops. We are now flagged for firing so next time round the function will be running the top code which moves the player bullet progressively up the screen, once it reaches the top it is deleted and we are set to not firing anymore. Ahh – but there is another routine that is called just before then I

hear you say, playerCheckHitEnemies() - it sounds like a fun routine so lets go examine it:

playerCheckHitEnemies()

Firstly the code

```
// has out bullet hit any enemies?
void playerCheckHitEnemies ( void )
{
	int iBulletX = dbSpriteX ( 2 ) + 12;
	int iBulletY = dbSpriteY ( 2 );
	int iEnemyX;
	int iEnemyY;
	int iEnemySprite = 9;

	// check bullet against all enemies
	for ( int iY = 0 ; iY < 5 ; iY++ )
	{
		for ( int iX = 0; iX < 5 ; iX++ )
		{
			iEnemySprite++;
			if ( g_bEnemiesAlive[iX][iY] == false )
				continue;
			iEnemyX = dbSpriteX ( iEnemySprite );
			iEnemyY = dbSpriteY ( iEnemySprite );

			// have we scored a hit?
			if ( ( iBulletX > iEnemyX && iBulletX < iEnemyX + dbSpriteWidth ( iEnemySprite
) ) ||		( iBulletX + 8 > iEnemyX && iBulletX + 8 < iEnemyX + dbSpriteWidth (
iEnemySprite ) ) )
			{
				if ( ( iBulletY > iEnemyY && iBulletY < iEnemyY + dbSpriteHeight   (
iEnemySprite ) ) || ( iBulletY + 32 > iEnemyY && iBulletY + 32 < iEnemyY + dbSpriteHeight (
iEnemySprite ) ) )
				{
					// yes! we have hit an enemy, lets increase the score and blow
that enemy up
					dbPlaySound ( 3 );
					// delete the sprite
					dbDeleteSprite ( iEnemySprite );
					// make a new sprite to show the explosion
					dbSprite ( iEnemySprite , iEnemyX , iEnemyY , 11 );
					iPlayerBulletY = -1;
					g_bEnemiesAlive[iX][iY] = false;
					// add and display the new score
					g_iPlayerScore += 100;
					playerUpdateScore();
					return;
				}
			}
		}
	}

	// lastly check if we hit the saucer if it's flying past
	if ( dbSpriteVisible ( 60 ) == 1 )
	{
		iEnemySprite = 60;
		iEnemyX = dbSpriteX ( iEnemySprite );
		iEnemyY = dbSpriteY ( iEnemySprite );

		// have we hit the saucer?
		if ( ( iBulletX > iEnemyX && iBulletX < iEnemyX + dbSpriteWidth ( iEnemySprite ) ) ||
( iBulletX +		8 > iEnemyX && iBulletX + 8 < iEnemyX + dbSpriteWidth ( iEnemySprite ) ) )
		{
			if ( ( iBulletY > iEnemyY && iBulletY < iEnemyY + dbSpriteHeight (
iEnemySprite ) ) || ( iBulletY + 32 > iEnemyY && iBulletY + 32 < iEnemyY + dbSpriteHeight (
iEnemySprite ) ) )
			{
				// yes to lets blow it up and update score
				dbPlaySound ( 3 );
				dbHideSprite ( iEnemySprite );
				dbSprite ( iEnemySprite+1 , iEnemyX , iEnemyY , 11 );
				iPlayerBulletY = -1;
				// update and display the new score
				g_iPlayerScore += 2000;
				playerUpdateScore();
				return;
			}
		}
	}
```

```
}
```

What we are doing in this routine is in two parts, the first part is we are checking the enemy bullet position x and y and also the x and y plus the width and height of the bullet to see if it has enetered into the a square occupied by an enemy ship. We have 5 rows of 5 collums of enemies – potentially, some may be already dead and so out of the fight, to check this we first look at the array g_bEnemiesAlive[iX][iY] which will be set to true if the enemy is still alive – otherwise we don't need to check the bullet against it. If we have scored a hit then we will set the array to reflect the enemy has been destroyed and not need to check that one again.

Some interesting commands here:
dbSpriteX (int iID) - returns the x position of a sprite
dbSpriteY (int iID) - returns the y position of a sprite
dbSpriteWidth (int iID) - returns the width of a sprite
dbSpriteHeight (int iID) - returns the height of a sprite
dbHideSprite (int iID) - makes a sprite invisible, it still exists but we just cannot see it
dbSpriteVisible (int iID) - returns true of the sprite is visible, false if it is hidden

For completeness I'll also add a further command we will use later on:

dbShowSprite (int iID) - shows a sprite that was previously hidden

One important thing to mention about this routine is this section:

```
// delete the sprite
dbDeleteSprite ( iEnemySprite );
// make a new sprite to show the explosion
dbSprite ( iEnemySprite , iEnemyX , iEnemyY , 11 );
```

Why are we deleting the sprite only to remake it after with a different image..surely we can just change the image it's using? Yes, normally we can, but the enemies, like the player (as you will see shortly) are intitally created as animated sprites with all the frames loaded in one image. Our explision however is all seperate images, animated sprites made from an image containing all its frames and “normal” sprites cannot be interchanged. With this in mind the simple solution is to simply delete the animated sprite and make it again as a standard sprite.

The bottom section of the function is similar to the top, although it deals purely with the flying saucer. When the flying saucer is hit, it is one of a kind, so rather than delete the sprite we just hide it and make another one to show the explosion as a very short lived sprite.

Right so that is the player side taken care of for now. Next up are the enemies.

Part 6: Enemies Update

enemiesUpdate()

Firstly lets have a look at the code – just for a change:

```
void enemiesUpdate ( void )
{
	// check if we have created the enemies, if now lets create them
	if ( g_bEnemiesSpawned == false )
		enemiesSpawn();

	// see how much time has passed this level and speed up enemies if needs be
	enemiesTimeCheck();

	// animate enemies
	enemiesAnimate();
	// move enemies
	enemiesMove();
	// decide if the enemies should fire and update any bullets fired
	enemiesFire();
	// check if its time for the saucer / update the saucer
	enemiesSaucer();
}
```

As you can see at the top of the function, the enemies are checked against being spawned, if this is a new level then they have not been spawned so we called enemiesSpawn() to kick off the process. Why are we doing it here? We make the player once and thats that – what is the deal with the enemies you ask? Remember back to the last section, where we check if an enemy is hit by the player, if the enemy is we deleted the enemy sprite because we wanted to show the explosion animation, which is all seperate images and so not compatible with an animated sprite. Since the sprites will all be deleted at the start of a level (or if we are playing for the first time, they wont exist anyway) then we need to make some new aliens for us to kill.

The enemiesSpawn() routine looks like this:

enemiesSpawn()

```
// create the enemies and set up their initial positions and properties
void enemiesSpawn ( void )
{

	// reset variables
	g_iEnemyOldY = 0;
	g_EnemiesDeadCount = 0;
	g_iExplosionDelay = 0;

	// delete any sprites - "dead" enemies have been swapped to a different image and so loose
their "animImage" capability, so we can delete and remake them to turn them back
	for ( int i = 9 ; i < 35 ; i++ )
	{
		if ( dbSpriteExist ( i ) )
			dbDeleteSprite ( i );
	}
	// Delete the image we loaded automatically with dbCreateAnimatedSprite, because we are
going to make them again
	if ( dbImageExist ( 10 ) )
		dbDeleteImage ( 10 );

	// setup the enemies, load the images and sprites
```

```
	enemiesSetup();

	// delete the saucer's explosion sprite
	if ( dbSpriteExist ( 61 ) == 1 )
		dbDeleteSprite ( 61 );

	// set the initial position of all enemy sprites
	// loop through the 5 rows of enemies, 5 in each row
	int iSprite = 10;
	for ( int iY = 0 ; iY < 5 ; iY++ )
	{
		for ( int iX = 0; iX < 5 ; iX++ )
		{
			dbSetSpriteFrame ( iSprite , dbRnd ( 7 ) + 1 );
			dbSprite ( iSprite , iX*130 , iY*66 , 10 );
			dbSetSpritePriority ( iSprite++ , 3 );
			// set each enemy to "alive"
			g_bEnemiesAlive[iX][iY] = true;
		}
	}

	// get our initial top left and top right posititon of all the enemies as a group, we will
from now on update these numbers ourselves
	g_iEnemiesX = dbSpriteX ( 10 );
	g_iEnemiesY = dbSpriteY ( 10 );

	g_bEnemiesSpawned = true;

	g_iEnemySpeedX = g_iLevel;
	// Lets cap the enemies speed at 10 so it doesnt get too mental
	if ( g_iEnemySpeedX > 10 )
		g_iEnemySpeedX = 10;
	// no speed boost yet
	g_iEnemySpeedBoost = 0;
}
```

First part of thisis resetting the enemies variables to what they should be at the start of the game/level. We then cycle through all the enemy sprites and delete them if they exist – they may be explosions rather than the anim sprites so we need to delete and remake them. Next we call enemiesSetup() which I will run through afterwards – this is similar to playerSetup in that it is setting up anim sprites, but there is one difference worth showing so we will go back to it shortly. Carrying on we see we are deleting the saucers explosion sprite. We don't have to worry about the saucer because that is hidden when it explodes so it doesnt loose it's anim sprite properties.

The next part of the function:

```
	int iSprite = 10;
	for ( int iY = 0 ; iY < 5 ; iY++ )
	{
		for ( int iX = 0; iX < 5 ; iX++ )
		{
			dbSetSpriteFrame ( iSprite , dbRnd ( 7 ) + 1 );
			dbSprite ( iSprite , iX*130 , iY*66 , 10 );
			dbSetSpritePriority ( iSprite++ , 3 );
			// set each enemy to "alive"
			g_bEnemiesAlive[iX][iY] = true;
		}
	}
```

Sets the initial position for each enemy, and sets them to be “alive” in the array g_bEnemiesAlive[iX][iY].

Finally underneath we set a few more variables, two important ones to point out are:

```
	g_iEnemiesX = dbSpriteX ( 10 );
	g_iEnemiesY = dbSpriteY ( 10 );
```

As the enemies move across the screen, they reach an edge, move down and head back the other way. That is simple at the start but what if the player takes out the last two collums.. this means the remaining ships have to now move further to reach the right edge.

The variables above take care of helping us work that out easily. Enemies start at sprite ID 10, so storing the position of the very first enemy gives us a base to work from, we can just update these “offsets” as we go along.

Right that's the spawning taken care of, next on the list was enemyTimeCheck() so I guess it's time we checked on that.

enemyTimeCheck()

```
// check how much time has elapsed this level
// 1000 miliseconds = 1 second.
void enemiesTimeCheck ( void )
{
	int iTimeElapsed = dbTimer() - g_iLevelStartTime;

	if ( iTimeElapsed > 30*1000 )
		g_iEnemySpeedBoost = 1;

	if ( iTimeElapsed > 60*1000 )
		g_iEnemySpeedBoost = 2;

	if ( iTimeElapsed > 90*1000 )
		g_iEnemySpeedBoost = 4;

	if ( iTimeElapsed > 120*1000 )
		g_iEnemySpeedBoost = 10;

}
```

We initially set the variable iLevelStartTime to the current time when we started the level. In this routine we check the time stored in that against dbTimer() again (the current time). If the level has been playing 30 seconds with give the enemies a little speed boost, after 60 seconds another boost, after 90 seconds a bigger boost and after 120 seconds a really silly boost.

That's all for enemyTimeCheck() lets proceed to the next one enemiesAnimate()

enemiesAnimate()

```
// animate our enemies
void enemiesAnimate ( void )
{
	// 10 is the first sprite in our enemies list
	int iSprite = 10;
	// this will hold the enemies currently displayed frame number
	int iFrame;

	// increment the explosion delay, this allows us to slow down the explosion animation
	g_iExplosionDelay++;

	// loop through the 5 rows of enemies, 5 in each row
	for ( int iY = 0 ; iY < 5 ; iY++ )
	{
		for ( int iX = 0; iX < 5 ; iX++ )
		{
			// if they are alive lets animate them using the animated image we loaded and
appied to each 				one
			if ( g_bEnemiesAlive[iX][iY] )
			{
				iFrame = dbSpriteFrame ( iSprite );
				if ( iFrame < 8 )
					iFrame++;
				else
					iFrame = 1;
				dbSetSpriteFrame ( iSprite , iFrame );
			}
			else
			{
				// they are dead so we texture them with the next frame of the
explosion animation 				which is seperate images
				if ( g_iExplosionDelay > 2 )
				{
					if ( dbSpriteImage ( iSprite ) < 23 )
					{
						dbSprite ( iSprite , dbSpriteX ( iSprite ) , dbSpriteY (
iSprite ) ,  dbSpriteImage ( iSprite
)+1 );

					}
					else if ( dbSpriteVisible ( iSprite ) )
					{
						// once we finished the animation we can hide the
departed alien
						dbHideSprite ( iSprite );
						g_EnemiesDeadCount++;
						// if 25 enemies (5 rows of 5) have all been killed, the
level is 					completed
						if ( g_EnemiesDeadCount == 25 )
							g_bPlayerWin = true;
					}
				}
			}
			iSprite++;

		}
	}

	// check the explosion delay, if it is over 2 set it to zero to start the delay again
	if ( g_iExplosionDelay > 2 )
		g_iExplosionDelay = 0;

}
```

Thankfully we have seen all this before with the player. The enemies are cycles through (5x5), checked if they are alive, if they are they are animated using their anim sprite properties (a simple incrementing through the frames for the enemies). If they are dead, their sprite would have been deleted as the player hit them (remember playerCheckHitEnemies) and a new sprite made ready for texturing with the images that make up the explosion (image Ids 11-22). Once the dead enemy has reached the end of the explosion animation the sprite is hidden with dbHideSprite (int iID). We use an explosion delay variable to slow down the animation to a nice speed rather than switching

images every single frame of the game.

EnemiesMove()

This function and its sibling functions handle the movement of the enemies as they make their way (albiet at a somewhat leisurely pace – well initially anyway) to the bottom of the screen and their goal of invading Earth.

```
// all the moving of enemies is called from this routine
void enemiesMove ( void )
{
	// pick the direction and call the routine for it
	switch ( g_eEnemiesMoveMode )
	{
	case eEnemiesMoveRight:
		enemiesMoveRight(); break;
	case eEnemiesMoveDown:
		enemiesMoveDown(); break;
	case eEnemiesMoveLeft:
		enemiesMoveLeft(); break;
	}
}
```

As you can see we have a mode for the movement (g_eEnemiesMoveMode) which points to one of 3 possible functions:

enemiesMoveRight()

```
void enemiesMoveRight ( void )
{
	// loop through the 5 rows of enemies, 5 in each row
	int iSprite = 10;
	for ( int iY = 0 ; iY < 5 ; iY++ )
	{
		for ( int iX = 0; iX < 5 ; iX++ )
		{
			// only move if the enemy is still alive
			if ( g_bEnemiesAlive[iX][iY] )
				dbSprite ( iSprite , dbSpriteX ( iSprite ) + (g_iEnemySpeedX +
					g_iEnemySpeedBoost) , dbSpriteY ( iSprite ) ,
dbSpriteImage ( iSprite ) );
			iSprite++;
		}
	}

	// update our stored position of the top left of the enemy group as a whole
	g_iEnemiesX += g_iEnemySpeedX + g_iEnemySpeedBoost;

	if ( g_iEnemiesX >= 1024-( 5*130 )-8 + enemiesGetRightOffset() )
	{
		g_iEnemyOldY = g_iEnemiesY;
		g_eEnemiesMoveMode = eEnemiesMoveDown;
	}
}
```

What we are doing here is cycling through every enemy, checking if they are still alive and if so moving them to the right. In the bottom half of the function we update our variable holding the position of the top left of the group, then checking it (adding in the offset – enemiesGetRightOffset() - to deal with collums that have been destroyed on the right side). If the enemies have reached the edge of the screen then movement control is passed to enemiesMoveDown().

enemiesMoveDown()

```
void enemiesMoveDown ( void )
{
	// loop through the 5 rows of enemies, 5 in each row
	int iSprite = 10;
	for ( int iY = 0 ; iY < 5 ; iY++ )
	{
		for ( int iX = 0; iX < 5 ; iX++ )
		{
			// only move if the enemy is still alive
			if ( g_bEnemiesAlive[iX][iY] )
				dbSprite ( iSprite , dbSpriteX ( iSprite ) , dbSpriteY ( iSprite ) +
(g_iEnemySpeedY                                    + g_iEnemySpeedBoost) , dbSpriteImage ( iSprite
) );
			// check if any have made it low enough to end the game
			if ( dbSpriteY ( iSprite ) > 650 - 64 )
			{
				// make sure they are alive first
				if ( g_bEnemiesAlive[iX][iY] == true )
				{
					g_iPlayerLives = 1;
					g_bPlayerHit = true;
				}
			}
			iSprite++;
		}
	}

	// update our stored position of the top left of the enemy group as a whole
	g_iEnemiesY += g_iEnemySpeedY + g_iEnemySpeedBoost;

	if ( g_iEnemiesY - g_iEnemyOldY >= 16 )
	{
		if ( g_iEnemiesX >= 1024-( 5*130 )-8 + enemiesGetRightOffset() )
			g_eEnemiesMoveMode = eEnemiesMoveLeft;
		else
			g_eEnemiesMoveMode = eEnemiesMoveRight;
	}
```

```
}
```

enemiesMoveDown() is much the same as the previous enemiesMoveRight() apart from it is obviously moving the enemies down, and checking if any enemies have reached the player at which point the game will end (we set the players lives to 1 and kill him, thus ending the game. Once the enemies have moved down we check which edge they are at (left or right) and switch the mode to the relevant movement (left or right).

enemiesMoveLeft()

```
void enemiesMoveLeft ( void )
{
	// loop through the 5 rows of enemies, 5 in each row
	int iSprite = 10;
	for ( int iY = 0 ; iY < 5 ; iY++ )
	{
		for ( int iX = 0; iX < 5 ; iX++ )
		{
			// make sure they are alive first
			if ( g_bEnemiesAlive[iX][iY] )
				dbSprite ( iSprite , dbSpriteX ( iSprite ) - (g_iEnemySpeedX +
					g_iEnemySpeedBoost) , dbSpriteY ( iSprite ) ,
dbSpriteImage ( iSprite ) );
			iSprite++;
		}
	}

	// update our stored position of the top left of the enemy group as a whole
	g_iEnemiesX -= g_iEnemySpeedX + g_iEnemySpeedBoost;

	if ( g_iEnemiesX <= 8 - enemiesGetLeftOffset() )
	{
		g_iEnemyOldY = g_iEnemiesY;
		g_eEnemiesMoveMode = eEnemiesMoveDown;
	}
}
```

This function is close to the right movement apart from heading left and calling enemiesGetLeftOffset() to see if the enemy craft have reached the left edge before handing over to enemiesMoveDown()

The two offset functions:

```
// we need the enemies to go tightly to the left edge, so if some commumns have been destroyed we
create an offset to show us how much further we need to move the enemies
int enemiesGetLeftOffset ( void )
{

	int iLeftOffset = 0;

	for ( int iX = 0 ; iX < 5 ; iX++ )
	{
		for ( int iY = 0; iY < 5 ; iY++ )
		{
			// if an enemy is alive in that row we can return from this function
			if ( g_bEnemiesAlive[iX][iY] == true )
				return iLeftOffset;
		}
		// whole collumn is dead so add the width of a collum to the offset
		iLeftOffset += 130;
	}
	return iLeftOffset;
}

// we need the enemies to go tightly to the right edge, so if some commumns have been destroyed we
create an offset to show us how much further we need to move the enemies
int enemiesGetRightOffset ( void )
{

	int iRightOffset = 0;

	for ( int iX = 4 ; iX > -1 ; iX-- )
	{
```

```
                for ( int iY = 0; iY < 5 ; iY++ )
                {
                        // if an enemy is alive in that row we can return from this function
                        if ( g_bEnemiesAlive[iX][iY] == true )
                                return iRightOffset;
                }
                // whole collumn is dead so add the width of a collum to the offset
                iRightOffset += 130;
        }

        return iRightOffset;
}
```

These check if any enemies in a row exist, if they do not then and offset is applied, once all collums have been checked the offset (either left or right) is returned.

EnemiesFire()

```
void enemiesFire ( void )
{
        // if the player is in the process of being blown up then we wont fire until the player
returns to the game
        if ( g_bPlayerHit )
                return;
        // go through each enemy bullet - 3 to 5
        for ( int iBulletSprite = 3 ; iBulletSprite < 6 ; iBulletSprite++ )
        {
                // if the sprite exists then its in the game and needs to be updated
                if ( dbSpriteExist ( iBulletSprite ) )
                {
                        dbSprite ( iBulletSprite , dbSpriteX ( iBulletSprite ) , dbSpriteY (
iBulletSprite ) +
        g_iEnemyBulletSpeed , 3 );
                        if ( dbSpriteY ( iBulletSprite ) > 768-32 )
                                dbDeleteSprite ( iBulletSprite );
                }
                else
                {
                        // if the bullet sprite doesnt exist, lets pick a random number and see if its
time to shoot it
                        int iRnd = dbRnd ( 60 );
                        if ( iRnd == 0 )
                                // lets add it
                                enemyAddBullet( iBulletSprite );
                }
        }

        // check if any bullets have hit the player
        enemyCheckBullets();

}
```

This routine firstly checks if the player is in the process of exploding, if he is it returns straight away. If not it loops through from 3 to 5 which are allocated for the enemies bullets (so they can have 3 on the go at once) and if the sprites exist that means they are in use, so they are updated until they reach the bottom of the screen at which time they are deleted. If any of the bullets don't exist then there is a chance one of the enemies may fire. To check for this chance we use a new function:

dbRnd (int iRnd) - return a random number between 0 and iRnd.

If we wanted to generate a random number between 10 and 20 we could do that by using:
int random = dbRnd (10) + 10;

If the random number in our game happens to be zero, then it means its time for a new bullet and we call enemyAddBullet(). Lastly we call enemyCheckBullets() to see if the enemies have hit the player.

Here are the two functions we just mentioned:

enemyAddBullet()

```
void enemyAddBullet ( int iBulletSprite )
{
       // we need to pick a collumn
       int iX = dbRnd ( 4 );

       // then work from the bottom up, the lowest enemy on that collum will be the none that
fires, if none are alive in   that collumn then no bullet will be added
      for ( int iY = 4 ; iY > -1 ; iY-- )
     {
       if ( g_bEnemiesAlive[iX][iY] )
       {
              int iEnemySprite = (( iY * 5 ) + iX) + 10;
              dbSprite ( iBulletSprite , dbSpriteX ( iEnemySprite ) + 48 , dbSpriteY ( iEnemySprite
) + 32 , 3 );
                     return;
       }
      }
}
```

this routines picks a random colum (0-4) and finds the bottom enemy that is alive in that collum (so we traverse the loop backwards in this case to find the first available enemy that is still alive). If no enemies are found on that collum the bullet sprite is made, otherwise the function returns and no bullet is created (poor enemies, try again next time).

enemyCheckBullets()

```
// check if we have hit the player
void enemyCheckBullets ( void )
{
	// check all bullets ( 3 - 5 )
	for ( int iBulletSprite = 3 ; iBulletSprite < 6 ; iBulletSprite++ )
	{
		if ( dbSpriteExist ( iBulletSprite ) )
		{
			int iPlayerX = dbSpriteX ( 1 ) + 8; // shave a bit off the side of the player
to make it more                                                    exciting
			int iPlayerY = dbSpriteY ( 1 );
			int iBulletX = dbSpriteX ( iBulletSprite ) + 12;
			int iBulletY = dbSpriteY ( iBulletSprite );

			if ( ( iBulletX > iPlayerX && iBulletX < iPlayerX + dbSpriteWidth ( 1 ) - 16 )
|| ( iBulletX +                                8 > iPlayerX && iBulletX + 8 < iPlayerX + dbSpriteWidth
( 1 ) - 16 ) )
			{
				if ( ( iBulletY > iPlayerY && iBulletY < iPlayerY + dbSpriteHeight ( 1
) ) ||                                  ( iBulletY + 32 > iPlayerY && iBulletY + 32 < iPlayerY +
dbSpriteHeight ( 1 ) ) )
				{
					// yes we have, play the explosion sound and set the player to
be hit
					dbPlaySound ( 3 );
					g_bPlayerHit = true;
				}
			}

		}
	}

	// if enemy has shot the player we delete all bullets so the player has a "safe" evironment
to come back to
	if ( g_bPlayerHit == true )
	{
		for ( int iBulletSprite = 3 ; iBulletSprite < 6 ; iBulletSprite++ )
		{
			if ( dbSpriteExist ( iBulletSprite ) )
				dbDeleteSprite ( iBulletSprite );
		}
	}

}
```

This function is very similar to the player version we looked at earlier (playerCheckHitEnemies()) and loops through every enemy bullet that exists to see if it has hit the player. We give the player a slight edge over the enemies in that we give the player a little safety zone, in that we ignoring the left and right 8 pixels to give the player that “just scraped past” feel sometimes. If they player is hit then sound ID 3 is played (the fatal explosion sound) and g_bPlayerHit is set to true which will trigger off the players death back in our main game loop (gamePlay()). Lastly, if the player has indeed been hit we delete any enemy bullets that are still in action. We do not want the player to come back

after loosing a life to land straight on a bullet again.

Lastly for the enemies we come to the flying Saucer:

```
void enemiesSaucer ( void )
{
	// if the enemies are down the screen past 64 then there is a chance the saucer can show up
	// it will only show up if it is not currently visible, and isnt in the process of blowing
up
	if ( g_iEnemiesY > 64 && dbSpriteVisible ( 60 ) == 0 && dbSpriteExist ( 61 ) == 0 )
	{
```

```
                if ( dbRnd(800) == 0 )
                {
                        // time to show and move the saucer
                        dbSprite ( 60 , 1024 , 0 , 60 );
                        dbShowSprite ( 60 );
                        dbLoopSound ( 4 );
                }
        }

        // if the saucer is shown, lets move it
        if ( dbSpriteVisible ( 60 ) == 1 )
        {
                dbSprite ( 60 , dbSpriteX ( 60 ) - 2 , dbSpriteY ( 60 ) , 60 );
                dbSetSpriteFrame ( 60 , dbSpriteFrame ( 60 ) + 1 );
                if ( dbSpriteFrame ( 60 ) > 4 )
                        dbSetSpriteFrame ( 60 , 1 );

                if ( dbSpriteX ( 60 ) < -128 )
                {
                        dbStopSound ( 4 );
                        dbHideSprite ( 60 );
                }
        }

        // if sprite 61 exists that means the saucer has been shot by the player, so we animate the
explosion
        if ( dbSpriteExist ( 61 ) == 1 )
        {
                if ( dbSpriteVisible ( 61 ) == 1 )
                {
                        if ( dbSpriteImage ( 61 ) < 23 )
                        {
                                dbSprite ( 61 , dbSpriteX ( 61 ) , dbSpriteY ( 61 ) , dbSpriteImage (
61 )+1 );
                        }
                        else
                        {
                                dbStopSound ( 4 );
                                dbDeleteSprite ( 61 );
                        }
                }
        }

}
```

There are three parts to this function. The top part checks initially that there is enough room for the saucer to fly accross without the other enemies getting in its way. If the path is clear then a random number is picked (dbRnd(800)) which if returns zero will set the saucer off (playing out nice saucer sound – sound ID 4 and also using dbShowSprite (60) to show the previously hidden sprite). We use dbLoopSound (4) here to enable the sound to keep playing until we stop it, once it reaches the end of playing it “loops” back and starts playing from the begining again.

The second part to this function moves the saucer across the screen right to left until it passes off the edge of the screen (we check the position of the sprite with dbSpriteX (60)).Once the saucer is off the screen we stop the sound with dbStopSound(4) and also hide the saucer sprite once more with dbHideSprite(60).

The final section is for when the saucer has been hit by the players bullet. If this is the case the playerCheckHitEnemies() function will have created sprite ID 61 for us. We check if this exists, if so we update the image being show for the explosion until it finishes, then stop the saucer sound and delete sprite 61 as it has fufilled its purpose.

Thats the main game loop delt with as far as the player and enemies are concerned, let's now move on to what happens with the player has killed all enemmies: gameWin() and then afterwards look at gameDie().

Part 7: Game Over Sequence

gameWin()

```
// player has cleared the level
void gameWin( void )
{
	// hide the saucer
	dbHideSprite ( 60 );
	// stop the saucer sound
	dbStopSound ( 4 );
	// delete all bullets both player (2) and enemies (3-5)
	for ( int iBulletSprite = 2 ; iBulletSprite < 6 ; iBulletSprite++ )
	{
		if ( dbSpriteExist ( iBulletSprite ) )
			dbDeleteSprite ( iBulletSprite );
	}
	// use our custom text routine to show the level is completed
	InvaderText ( 0 , 340 , 32 , "LEVEL COMPLETE!" , true , false );
	InvaderText ( 0 , 400 , 16 , "PRESS FIRE" , true , false );

	// increase the level by 1
	g_iLevel++;

	// reset the game
	g_eGameMode = eGameReset;
}
```

First thing we do here is hide the saucer sprite and stop the saucer sound – just incase the saucer was in action when we cleared the level (completing a level is triggered by killing the enemy formation, the saucer is merely a bonus to the players score).

Once we have the saucer taken care of we cycle through all bullets – both the player and enemies (players is sprite 2, enemies are sprites 3-5 so we can delete them all in one go).

We then use our nice text routine to display the fact the level is complete, increase the level variable
g_iLevel and set the game mode to use gameReset() which will prepare us for a new level.

gameDie()

```
// player has lost a life
void gameDie ( void )
{
	// player can keep shooting
	playerUpdateFire();
	// update enemies (they will not fire while the player is out of action
	enemiesUpdate();
	// has the explosion of the player completed?
	if ( playerDie() == true )
	{
		// decrease players lives by 1
		g_iPlayerLives--;
		// update how many lives the player has left
		playerShowLives();
		// show the player sprite again
		dbShowSprite ( 1 );
		// reset the fact he was hit
		g_bPlayerHit = false;
		// if the player still has a life lets carry on, if not its game over
		if ( g_iPlayerLives > 0 )
			g_eGameMode = eGamePlay;
		else
			g_eGameMode = eGameOver;
	}
}
```

The player has been killed and his explosion has been viewed, we continue to call playerUpdateFire() and enemiesUpdate() so the players bullet keeps moving (if fired) and the enemies continue to descend. We then subtract a life from the player, call playerShowLives() and reset the player by showing his sprite again (sprite ID 1) and setting him back to not being hit (g_bPlayerHit). All we then need to do is check where to pass the flow of the game too, if the player still has lives in reserve we head back to gamePlay() and carry on, alternatively if the player has now lost all his lives we head to gameOver()

gameOver()

```
// game over
void gameOver ( void )
{
	// hide the saucer
	dbHideSprite ( 60 );
	// stop the saucer sound
	dbStopSound ( 4 );
	// stop atmospheric sound
	dbStopSound ( 5 );
	// updaye the player score on screen using our custom text routine
	playerUpdateScore();
	// show the game over text and the players final score
	char szScore[20];
	sprintf ( szScore , "YOU SCORED: %d" , g_iPlayerScore );
	InvaderText ( 0 , 340 , 32 , "GAME OVER!" , true , false );
	InvaderText ( 0 , 400 , 32 , szScore , true , false );
	// change the speed of our title screen sound and player it
	dbSetSoundSpeed ( 1 , 2000 );
	dbPlaySound ( 1 );
	// clean up enemies
	for ( int iEnemy = 10 ; iEnemy < 35 ; iEnemy++ )
	{
		if ( dbSpriteExist ( iEnemy ) )
			dbDeleteSprite ( iEnemy );
	}
	// clean up all bullets
	for ( int iBullet = 2 ; iBullet < 6 ; iBullet++ )
	{
		if ( dbSpriteExist ( iBullet ) )
			dbDeleteSprite ( iBullet );
	}
	// call sync to display the screen (we call it here because we are going to wait for a while
before going back to the title screen)
	dbSync ();
	// wait for 5 seconds (1000 milliseconds = 1 second)
	dbWait ( 5000 );
```

```
        // clear all custom text
        invaderTextClear();
        // head back to the title screen
        g_eGameMode = eGameTitle;
}
```

Here again we hid the saucer and stop its' sound just incase it is flying across as the game ends. We stop playing our “in game” atmospheric sound also (dbStopSound (5);). We update the score one more time and display text informing the player the game is over for them and what their final score was. We reuse sound 1 yet again, changing the speed it plays back. Finally we delete all enemies and bullets, call dbSync() to draw the screen for us, then use a new command dbWait (int iMilliseconds) which causes the program to half for however long we choose (we have opted for 5 seconds).

Once the five seconds have past we clear all text with a call to invaderTextClear() and return to gameTitle().

gameReset()

```
// reset the game after a life is lost or level complete
void gameReset ( void )
{
        // stop the background sound playing
        dbStopSound ( 5 );
        // start the title screen sound and change the speed it plays back for a different effect
        dbSetSoundSpeed ( 1 , 2000 );
        dbPlaySound ( 1 );
        // change the speed the backdrops scroll at for a different visual effect
        g_fBackdropSpeed = 2.0f;

        // reset the player and enemies
        g_bPlayerWin = false;
        g_bEnemiesSpawned = false;
        playerReset();

        // set the mode to wait for the player to press fire
        g_eGameMode = eGameWaitForFire;

}
```

Our last function. gameReset() stops the background sound playing (our atmospheric sound ID 5), then plays our trusty sound 1 (after changing its' playback speed) and changes the speed of the backdrops to a +2 modifier.

Finally gameReset() resets the g_bPlayerWin and g_bEnemiesSpawned flagged before called playerReser() and passing over the gameWaitForFire() so we can set off once again to face the enemy.

Part 8: Customising the game further

That is the basic game completed. There are many things we could add to improve the game. Why not set yourself the task of adding some extra features to the game such as:

- High Score Table
- A bonus score for completing a level
- A bonus life after a certain amount of points
- More than one saucer appearing on later levels with an extra bonus for getting them all

I hope this tutorial has been of some use in your quest to get to grips with the power that is now in your hands with Visual Studio coupled with Dark GDK.