

Angelscript script tutorial

Introduction:

This is a tutorial on using Angelscript more than learning it. If you haven't little knowledge of Angelscript then I have written another (my first) tutorial on Angelscript that gives an incite on the basics, please read and understand it first.

I will start Chapter 1 with some basic code. We will develop a bones animation script for the character object then turn it onto a function so it can be more readily re-used in other characters.

In Chapter 2 we will setup a Multi cam chase. We start with a basic 2 key activation script then I introduce a toggle key system which uses 1 key to switch cameras.

In Chapter 3 we learn how to control the vehicle dust tracks and tyre marks. First part will be the dust control and second will be the how to control the tyre marks and skid marks.

Chapter 4 we develop a basic 4 selection menu with key and mouse selection. We will use Sprites for the menu with the option of mouse interaction.

Next Chapter 5 we will remove the 3DRAD character object and develop our own scripted player character so you can move around the scene. We will use keys to direct the movement for the player character.

Then Chapter 6 we will build the scripted NPC as a passive player in your games. It will have minimal interaction with the player. The NPC will follow a path, and walk to the first node then the next and so-on until it reaches the last and then start over.

In Chapter 7 We will give the player a weapon script so you can shoot the crap out of the NPC. This is where I will introduce the object oriented Class system.

Chapter 8 we will also include the weapon script so the NPC can defend its self. The major difference between the Player and the NPC is, it will have to make its own decision. Again following our simple format it will only have two decisions to make 1 to patrol or 2 to attack.

In Chapter 9 we setup an Orbit cam control scripts. Here I will build it with the Class script. The Class will allow you to easily re-use the script in other programs.

Chapter 10 we will do basic scoring system. I want to show some methods to how you can use the class as a data base for your game.

The last Chapter 11 we theorize about Programing AI code methods that will control your NPC character. This is a test for you to read then develop your own Ai character control.

In this tutorial have including complete scripts and a 3DRAD project so you can check them out before you start writing. When you read this tutorial the best way to learn is to start at the beginning of the script and wright each line of code in one by one as you go, doing it this way will gives you a better understanding of how each line works and how it fits with in the whole code. When you complete your code do the error check and correct any mistakes. Make sure you edit all Object Handles like 'OBJ_####' and IN_#### put and OUT_#### put reference name. Coping and pasting is quicker to get the job done but it will take longer to learn as you will make mistakes and it makes it harder to find them. Once you become better at coding you will be able to copy and paste chunks of code and modify it as you want.

Errors:

3DRAD has an error catch that when it comes across an error it pops up an error dialogue. If you run your project and the program pops up an error code take note of the script name and the line number. Open the script and go to the line and check it for errors. Check the line for obvious errors like missing matching brackets, spell errors, text case errors, missing semicolon at the end of a statement etc. If you fail to find the error or you think you may have fixed it, and you run it again, it doesn't always repeat the pop up error dialogue and the program will still runs on regardless. You may notice it is not working correctly, this generally indicates that the error is still there. To find the error save the project and then re-load it, if the error is still there it will re-occur with the popup a dialogue. So what this means is if you think you fixed an error but things are not working then re-load the project and run it again.

If it crashes 3DRAD you must then go over it line by line thoroughly to find the error. It is always good to give every script object a unique name so you can identify it when an error occurs.

At the end of player animation script in chapter 1, I have a paragraph about error checking and will include more about that as things develop. I may repeat my self about some of error checking processes but you get to know these processes very well as you develop your skills.

Now often on the forums there has been many times someone cried out calling Angel Script bugged, this is not so. There has never been any officially recognized bug found in Angel Script especially the last versions of 3DRAD7.22 and 3DRAD6.50.

Understand my code terms for this book

In this tutorial I will use a set of terms that may sound foreign to you so that's what this paragraph is all about.

Terms: **Meaning:**

Script. Code used in the script object

Code. Code refers to the written lines of script

Statement. The functions that have been made available from 3DRAD game engine which I will refer as Statement.

Function. Function is something that you create within your script to handle a special coded process.

Process. Process refers to how the game engine deals with each line of code.

Main(). Main() will refer to the Main(){} which encompasses the script and forms the basis of the script. This is the lead in point (start) where the code is processed. There are set standards for what code can go inside or outside.

Global Variables. Global variables value persists when the game engine process leaves the script object to process other data from other objects in the game.

Class. A Class has a similar function to the Main() except all the code must be inside the class brackets 'class{}' I will cover how to write class objects later.

Method A class Methods is the same as a Function (described above) but is inside the class.

Properties. Class Properties are the global variables for the class. They can be accessed from anywhere in the script, including the Main() functions and also from other classes.

Argument. A code argument describes a Boolean code and is found inside an 'if()', 'while()' or 'for()' statement etc, for example if(boolean or code argument){}

If you are unfamiliar with Angelscript statements then refer to my last tutorial in Angelscript. It covers all the common statements and gives some examples of how to use them.

Hint: I have included hints in this tutorial and they are scattered everywhere so if some are situated in a code block remove them before you test it. There may also be un-commented text so remember to remove it as well.

Index

Chapter 1 - Basic Animation

Chapter 2 - Multi cam chase

Chapter 3 - Vehicle dust tracks and tyre marks

Chapter 4 - Basic 4 selection menu

Chapter 5 - Building a scripted Player character

Chapter 6 - Basic NPC Character

Chapter 7 - Player Weapon Script Class

Chapter 8 - NPC weapon script class

Chapter 9 - Orbit cam control

Chapter 10 - Scoring

Chapter 11 - Programing AI code methods theory

Chapter 1 - Basic Animation

First we will start with a basic Bones animation script. Now load some objects like the terrain, gravity, skybox, sphere RB, andro SM, Character object, script object and cam chase.

Link the sphere to the terrain, gravity and character. Link the Script to the SkinMesh object. Setup the character with the skinmesh and sphere then test.

You should see Andro walking around but it will only have the default hand wave animation. Andro has 3 Bones animations the default 0 wave animation, 1 walk animation and 2 the sit animation.

3DRAD processes the script 60 times every second like all other objects in your game. Starting with the first line of Main() and through till the end, sometimes with diversions to functions etc. It first sets up global variables if some exist, this is done only once the first time the script is run. Then it runs the code inside the Main(), first it sets up any local variables you may want initialize, then starts with the first line of code and continues processing every line until the end. The Main() {code goes here}, left brace indicates the start and the right brace is the end. The braces are used throughout script statements and functions for the same reason. I am not going into a detailed description of each statement because I have covered the most common code statements in the Angelscript tutorial, here I am going to cover each line and explain how it works. I intend to use Angelscript class object methods as well but will describe the benefits and usage when I get to them.

This is a framework of the Angelscript Main().

```
// Global Variables go here
Main()
{
    // your script goes here
}
// your functions go here
void myFunctions(){
}
```

First write the following script in between the braces.

The following lines (with // and a description) are comments about the script. You use these to describe what this script is about and how to use it.

To start we must have 2 global variables. These go at the top outside of the Main(){}

```
// Scripted bones animation Chapter 1 Tutorial Script V1 2017
// global animations variables
int setAnimation = 0;
int setAnimationNew = 0;
Main(){
    // if SkinMesh speed is greater than 0.2 meters per second then animate
    if(IN_### > 0.2){
        setAnimationNew = 1; // walk animation
    }else{
        // else if the speed is less than 0.2 then change to wave animation
        setAnimationNew = 0; // wave animation
    }
    if(setAnimation != setAnimationNew){ // test if the variables are not equal
        OUT_### = setAnimationNew; // update new SkinMesh animation
        setAnimation = setAnimationNew; // re-assign the new animation
    }
}
```

Now bones animations like all animations have a start and an end with some time (animation) between so if we set animation to, for example 'OUT_### = setAnimationNew' every process loop, it would start the animation every loop at 60 times a second and this would result in no animation, Andro would appear to be frozen. This is why we have included the code argument 'if(setAnimation != setAnimationNew)'. If Andro's animation doesn't equal the new animation then change it. So this is a fundamental piece of code you will need for most animations.

The base code was originally provided to the community by 3DRAD

Because this code is an important piece of code we should improve it and make it so you can re-use it in other characters or projects. To do this we will turn it into a separate function. What this does is compartmentalize the code making it easier to use and understand. To make it even more useful we will add some other feature that is the animation speed. Below is the new function that will show how we will handle this.

Chapter 1 Continued

```
// Scripted bones animation function Tutorial Script V1 2017
// bones animation function
void bonesAnimation(int newAnimation, float animSpeed){
    if(setAnimation != newAnimation){ // test if the variables are not equal
        OUT_### = newAnimation; // update new SkinMesh animation
        setAnimation = newAnimation; // re-assign the new animation
        OUT_### = animSpeed; // set animation speed
    }
}
```

We still require the global variable setAnimation and we must also call this function from the Main() script below is the examples.

```
// Bones animation Function Chapter 1 Tutorial Script V1 2017
// initialize global animations variables
int setAnimation = 0;

void Main()
{
    int setAnimationNew = 0;
    // if SkinMesh speed is greater then 0.2 meters per second then animate
    if(IN_### > 0.2){
        setAnimationNew = 1; // walk animation
    }
    bonesAnimation(setAnimationNew, 1.00); //here we call the animation function
}

void bonesAnimation(){
// Scripted bones animation function Tutorial Script V1 2017
// bones animation function
void bonesAnimation(int newAnimation, float animSpeed){
    if(setAnimation != newAnimation){ // test if the variables are not equal
        OUT_### = newAnimation; // update new SkinMesh animation
        setAnimation = newAnimation; // re-assign the new animation
        OUT_### = animSpeed; // set animation speed
    }
}
}
```

What you may have noticed we have compacted the code so we can now copy it and paste it to other scripts and or other projects. What happens now is the setAnimationNew is set every process loop to the default zero and then reset only if the speed is more than 0.2. It all works the same as before but is a little more efficient and easy to reuse.

Hint: when you have and code snippets like this one save it from the script object for later use. If you do this with other examples you will build up quite a collection of useful script snippets.

Hint: with your code snippets save them into a separate folder inside your main script folder under categories like animation, key codes, shaders etc.

Hint: A general rule is when you have a piece of code like this, that is repeating, put it into a function. Repeated code in a function can be easy to error check, updated and re-use. Large chunks of code like the weapon script it is best to put in a Class that can have its own variables and methods, but more about this later.

Error testing: Now you have the script written in always test it by clicking the Check Script button. If you don't have errors you will have done well, else read the message of the first warning and note the line number, press Ok, then go to that line number and check the line for a syntax error. Correct it and test it again. When you have no more errors press Space bar and run the program. A word about the error checking. This only test for syntax Errors and not errors you may have made for example you may have used an incorrect variable in the wrong place. Your program may run but erratically or even crash. Sometimes an error message might report the line number after the error so if you can't find the error check the line before. You could write a book about error checking it is a very complex issue and Angelscript checker is very good so I don't want any talk about it being broken or full of bugs. When your program crashes it can always come back to something you have done. In most cases you will be able to find the error or an alternative to cure the problem.

If you can't find an error just after writing a block of code the best way to eliminate the new code without erasing it, is to hide it and this is done with a comment tag `//` for a single line or `/*` block of new code here `*/` for a block of code.

Hint: when writing code it is very important to format it so you will be able to understand it and error check it. Formatting code is simple and if you look at some of the example code in this tutorial you will see what I mean. The function name statement and it's curly brackets are left justified then any other statements inside is 1 tab spacing, any internal statements that have internal code are 2 tab spacing and so on. A carriage return between sections of code also help identify code.

Hint: If you want to look at the bones structure get a copy of MVIEW.exe and open Andro in it and press play animation. To change the animation select from the menu. Do not save it when you are finished looking it.

Some example errors:

Find errors in the following lines

```
// this will put up an error message in editor
int number = 0
```

```
// this will put up an error message in editor
```

```
int while = 1;
```

```
// this will put up an error message, only an run time
int []handle(10);
for(int i = 0; i <= 10; i++){
    int hNum = handle[i];
}
```

Chapter 2 - Multi cam chase

This Chapter is how to control the CamChase object. First we will script the multi cam chase that will provide 2 different views of your player character or car. Load the objects that will make up your project. Position the camera 1 behind the player or car object and camera 2 inside the head of the player character or inside the car objects. Include a Script object then link the cameras and the character or car skinmesh to it. Wright in the following code into the script object.

```
// Scripted Cam Chase Control Chapter 2 Tutorial V1 2017
// timer global variable
int timer = -1;
void Main(){
  // function key camera control V1 - chapter 8
  if(!initializing()){
    iObjectShow(OBJ_###); // camera 1
    iObjectHide(OBJ_###); // camera 2
  }
  if(iKeyDown(iKeyCode("DIK_F1")) && timer == -1){ // press function key 1 turn on camera 1
    iObjectShow(OBJ_###); // camera 1
    iObjectHide(OBJ_###); // camera 2
    timer = 30; //start timer
  }else if(iKeyDown(iKeyCode("DIK_F2")) && timer == -1){ // press function key 2 turn on camera 2
    iObjectHide(OBJ_###); // camera 1
    iObjectShow(OBJ_###); // camera 2
    timer = 30; //start timer
  }
  if(timer > 0){
    timer--;
    if(timer == 0 ){
      timer = -1;
    }
  }
}
```

How it works: First we setup 1 global variable then initialize the cameras, this basically turns on the default camera in your game. You might want camera 2 as the default that's up to you. Next the if() statement argument checks if you have pressed the F1 key, if true it turns on camera 1 and then turns off camera 2. Now when I say turn on/off the camera in 3DRAD you actually show or hide the camera as the 3DRAD statement reads. This is the preferred method because to start or stop the camera it would have to be initialized each time. As you can see from the script the F2 key is only a reverse setup to the F1 key code. It is very simple and works well.

Hint: You may have noticed I haven't suggested to make this script a class. It is best to have it in separate script object and not combined it with any other script so there is no need to make it a class. This is because it will be a stand along object and there is no need to combine it with other scripts. This also goes for the following script for the Orbit cam control.

Hint: remember to save this because you will re-use it many times over.

Chapter 2 continued

Single Button cam swap:

This script has the same function as the above script but is operated by one button, push on push off. As you can see the script needs two global variables this time one for the timer and the other for the boolean swap variable.

```
// Scripted toggle key camera control V1 Tutorial V1 2017
// timer global variable
int timer = -1;
bool swap = true;
void Main(){
  // toggle key camera control V1 - chapter 8
  if(!initializing()){
    iObjectShow(OBJ_###); // camera 1
    iObjectHide(OBJ_###); // camera 2.
  }
  if(iKeyDown(iKeyCode("DIK_F1")) && timer == -1){
    if(swap){ // press function key 1 turn on camera 1
      swap = false;
      iObjectShow(OBJ_###); // camera 1
      iObjectHide(OBJ_###); // camera 2.
    }else{ // press function key 2 turn on camera 2
      swap = true;
      iObjectHide(OBJ_###); // camera 1
      iObjectShow(OBJ_###); // camera 2.
    }
    timer = 30;      //start timer
  }
  if(timer > 0){
    timer--;
    if( timer == 0 ){
      timer = -1;
    }
  }
}
```

How it works: This works same as the previous 2 button script but instead of two key to monitor it only monitors one key. The global swap variable is set to true and the int timer variable is set to -1. The !initializing() statement sets the default cameras on or off. When the F1 key is pressed the argument (you can use any key you want) it is true, it then processes the next line. The next if() argument is also true and then processed the next line which sets the bool swap to false. Next the cameras are shown or hidden. The following code is jumped by the 'else' statement and the key delay timer is set to 30. Next it starts the timer count back. Now when you press the key again it will find the first swap argument in the 'if()' statement is 'false' then fall to the 'else' and reverse the camera show and hide settings. The timer will again be set and will start counting back.

Hint: Click the Save button and save it to your script/settings folder (default or sub folder). Do this with all of these scripts so you will build an excellent collection of scripts for the future.

Chapter 3 - Vehicle dust tracks and tyre marks

In this chapter we will script the dust tracks as long as your car is in contact with the ground. This is very good and realistic special effect. To do this you will have to set up a project with the car object and first get it working all correctly. Next include a particle object, event on contact and a script. Place the particles between and a little forward of the front wheels. Set it to about 200 per sec. and 5 sec. life span, you can make adjustments to them once you get it working. Connect the car and terrain to the event on contact, set the car as monitor and terrain as monitor. Now test to see the effect while driving. Next connect the event on contact and the particles to the script, open it and write in the following code.

```
// Scripted car particles control Chapter 3 Tutorial V1 2017
// car particles control
// if in contact is greater than zero and speed is greater than 0.2mps
if(IN_### > 100 && IN_### > 0.2){
    OUT_### = 10; // show particles per second
    OUT_### = 10; // show particles per second
}else{
    OUT_22 = 0; // hide particles per second
    OUT_### = 0; // show particles per second
}
```

How it works: This is a simple script as you can see. If (event on contact) IN_### contact is greater than zero and speed is greater than 0.2mps then start the particles by putting 200p/sec. in the (particle object) OUT_### else zero when the vehicle is stopped. Now test to see the effects.

When the vehicle is driving slow it won't pickup much dust and as the speed increases there will be more dust so if we add a little math to it, it will change. Where the OUT_### = 200 change it to $OUT_### = (20 * (IN_### * 0.5))$. This will multiply the 20 by the speed of the car IN_### by the 0.5, which you can adjust the value to what ever suits you.

You will have noticed by now that I have only used 1 particle object, this will be up to you, as the particle object is heavy on the processor and depending on your game may result in frame rate issues so be aware of that. You may be able to use 2 particle objects per vehicle but this may be the limit. Another way to control frame rates is to turn off those not seen by player, have less on NPC vehicles or only 1 particle object per NPC vehicles.

Vehicle tyre tracks/skid marks:

Here we will build a script to control Vehicle tyre tracks/skid marks. The implementation is the similar as the dust control except for the when the vehicle is under acceleration or braking. To control this we will have a different section in the code to read the key controls and adjust the skid mark color and opacity. All the code for skid mark control is below just write it into the script object. As before we will only have this code in it and not share it and will not share the script object with other codes or controls. You will still need the Event On Contact to get the tracks to work. You can use the same EOC as used for the dust script as I have in the demo project. Just set the EOCt relationships to S/H SWITCH ON CONTACT.

```
// Scripted car particles control Chapter 3 Tutorial V1 2017
// car particles control
// if in contact is greater than zero and speed is greater than 0.2mps
if(IN_#### > 100 && IN_#### > 0.2){
    OUT_#### = 10; // show particles per second
    OUT_#### = 10; // show particles per second
}else{
    OUT_#### = 0; // hide particles per second
    OUT_#### = 0; // show particles per second
}
// Scripted Vehicle tyre track and skid marks Chapter 3 Tutorial V1 2017
// Vehicle tyre track and skid marks
float trackOpacity = 0.0;
if(IN_#### > 0.2){ // if in acceleration is greater than 0
    trackOpacity = 0.95; // acceleration
}else if(IN_#### > 0){
    trackOpacity = 0.95; // braking
}else{
    trackOpacity = 0.25; // normal
}
// if contact is greater than zero and speed is greater than 0.2mps
if(IN_#### > 10 && IN_#### > 0.2){
    OUT_#### = trackOpacity; // show track opacity
    OUT_#### = trackOpacity; // show track opacity
}else{
    OUT_#### = 0.05; // hide track opacity
    OUT_#### = 0.05; // show track opacity
}
```

Hint: You may have noticed I have not given instruction on where you put this code. By now you should be a little more familiar with where the code should go.

Chapter 4 - Basic 4 selection menu

This script is a 4 selection menu, including key and mouse click activation. This is a stand alone project that loads you first game level. Before you start you will have to create 5 menu sprites. These are for the game name (heading), 1st menu item (play), 2nd menu item (help), 3rd menu item (credits) and last (exit game). You will also need a sprite for the pop-up help and credit pop-up window. Load a script object and 6 sprite items, open them and change the default sprites to the new menu images you have created, also check the 'Detect mouse-over event' box on the menu sprites only. Set out your menu display positioning and resizing the sprites per your design. Load 2 Exit fade object, link them to the script object. Rename one as Exit Fade - Exit and the other as Exit Fade - Level 1 and load and link a Text object. In the Exit Fade - level 1 address field enter the address and name of the first game level. In your game levels you will need an exit fade to return the player to the menu, if they want to end the game or if they loose. Don't forget to enter the correct handle where you see the OBJ_### reference.

```
// Chapter 4 Selection Menu Chapter 4 Tutorial V1 2017
// timer global variable
// mouse and key operation
int timer;
bool help;
bool credits;
void Main()
{
    // 4 selection menu
    if(!initializing()){
        timer = -1;
        credits = false;
        help = false;
    }
    // get key P or get mouse click on sprite play to start game
    if(iKeyDown(iKeyCode("DIK_P")) && timer == -1 || IN_### > 0 && iMouseButtonDown(0)){
        iObjectStart(OBJ_###); // start exit fade to level one of the game
        timer = 30; //start timer

    // get key H to display help info
    }else if(iKeyDown(iKeyCode("DIK_H")) && timer == -1 || IN_### > 0 && iMouseButtonDown(0)){
        help = true; // show help
        timer = 30; //start timer
        iObjectShow(OBJ_###); // show pop up menu background

    // get key C to display credits
    }else if(iKeyDown(iKeyCode("DIK_C")) && timer == -1 || IN_### > 0 && iMouseButtonDown(0)){
        credits = true; // show credits
        timer = 30; //start timer
        iObjectShow(OBJ_###); // show pop up menu background

    // get key or mouse click to exit game
    }else if(iKeyDown(iKeyCode("DIK_E")) && timer == -1 || IN_### > 0 && iMouseButtonDown(0)){
        iObjectStart(OBJ_###); // start game exit fade
        timer = 30; //start timer
    }
    if(timer > 0){
        timer--;
        if( timer == 0 ){
            timer = -1;
        }
    }
    // key help and other information
    if( help ){
        string txt = "Game key Help"; // start here with the text heading
        txt += "\r\r Forward - W"; // continue repeating this text with the +=
        txt += "\r\r Left - A + "; // The '\r' is a carriage return
        txt += " Right - D"; // this has no \r because I want it to follow the previous line
        txt += "\r\rExit game press - Esc";
        txt += "\r\rPress Return key to continue";
        iPrint(txt,-4,4,OBJ_###); // depending of the sprite you will have to realign the text print
    }
```

```

if(iKeyDown(iKeyCode("DIK_RETURN")) && timer == -1){ // wait for return key
    help = false;           // close display
    timer = 30;             // set key delay timer
    iObjectHide(OBJ_###);   // hide pop up menu background
}
}
// credits information
if( credits ){
    string txt = "Game Credits";
    txt += "\r\r3DRAD development team";
    txt += "\r\rPress Return key to continue";
    iPrint(txt,-4,4,OBJ_###); // depending of the sprite you will have to realign the text print
    if(iKeyDown(iKeyCode("DIK_RETURN")) && timer == -1){
        credits = false;
        timer = 30;
        iObjectHide(OBJ_###); // hide pop up menu background
    }
}
}
}

```

How it works: First it initialize some global variables. Then in the first 'if()' it initializes the value of these variables. In the next 'if()' it looks for a key input or a mouse click. If either input is true it would start the Exit Fade to the first game level. For the next 'if()' argument when true it makes the bool variable 'help' true and this results in the 'if()' argument following the comment about the '// key help and other information' becoming true as a result it will show the pop up background sprite then display all the help information as text. If the player reacts to the message 'Press Return key to continue' the 'if()' argument will be true and set the 'help' bool to false, this will close the help then set the key delay timer to 30 and hide the background sprite. The Credits is the same as the the Help with the obvious, different text. The 'iPrint' statement inside Help and Credit arguments will only work while it is true.

Hint: The address in the exit fade can be changed by using the iObjectTextSet() statement.

Hint: In most menus when the mouse is over the sprite it will change color. To do this you use a series of if(){}else{} statements to switch the color of the sprite. For example the first 'if(IN_### > 0){change RGB}else{change back RGB}' the IN_### is the mouse over and if true changes the RGB color of the sprite background and if else it changes it back to the default color. There are 3 OUT_###'s, for each one of the RGB. Look for them in the left of the script object.

Hint: The address format for the event on fade must have double back slash '\\' between folder names. Any address by default, begins at the 3DRAD game root folder.

Example ".\\myfolder\\myfilename.exe" - The dot, full stop or period sets the starting point or current folder for the address. It will look for the exe in a folder call myfolder.

".\\mystartfolder\\myfilename.exe" - This address would look for the exe in a folder before the root folder. not normally used.

".\\myfilename.exe" if the same as "myfilename.exe"

Don't set full address from the current drive root folder Example "C:\\Program Files\\3DRAD\\etc\\etc". If someone wants to install into a different drive it will not find what its looking for.

Chapter 5 - Building a scripted Player character

We will start with a player 3rd person character so you can move around and observe how the player character is moving and how well you can interact with the terrain and other rigid bodies. This is a step toward learning the basic on how build any player controlled object and how some objects can share information.

First we will load some objects like the terrain, gravity, skybox, sphere RB, andro SM, force, script object and chase cam. Link the sphere to the terrain and gravity. Link the script to the sphere and the sphere to the force. SkinMesh, force, and text object. Do not link the sphere to the skinmesh. Open the force object and set the force to 0.

Open the script and those objects should now show in the list to the left. The right panel will already contain some script.

```
// Scripted Character Tutorial Chapter 5 V1 2017
// this character is not perfect and may not
// work as well as intended. But I will later add
// some script to improve it, this will help explain
// how they will work.

// here we initialize the global variables
float headings;
float forward;
void Main(){

// the following comment describes what the next 12 lines do
// get key commands from the player
If(ikeyGet(Dik_W)){ // this line gets a W key click and sets the forward movement
    // 6 is the walk speed when the W key is pressed
    OUT_### = 6; // this is the output value for the force object
}else{
    // else zero forward movement when no key is pressed
    OUT_### = 0; // this is the output value for the force object
}
If(ikeyGet(Dik_A)){ // turn left in degrees
    headings += 0.01;
}
If(ikeyGet(Dik_D)){ // turn right in degrees
    headings -= 0.01;
}

// initialize some variables
Vector3 objRBvector;
Quantneion objOrientation;
RigidBody objRBody = OBJ_###; //alter to suit
SkinMesh objSMesh = OBJ_###; //alter to suit

iObjectLocation(objRBody,objRBvector); // get player RB position
iQuantneionFromEulia(ObjOrientation,Vector3(headings,0.5, 0.0); // calculate orientation form degrees (key input
from player)

iObjectLocationSet(objSMesh,ObjRBvector); // set new location of SkinMesh
iObjectOrientationSet(objSMesh,objOrientation); // set SkinMesh orientation

iObjectOrientationSet( ObjForce,objOrientation); // set force orientation
// all the variables now die as the processor leaves the Main(){ loop except // for the globals
}
```

Chapter 5 Continued

Error testing: Now you have the script written in always test it by clicking the Check Script button. If you don't have errors you will have done well else read the message of the first warning and note the line number, Ok, then go to that line number and check the line for a syntax error. Correct it and test it again.. When you have no more errors exit the script and press Space bar and run the program.

Back to the script

When you run this you will see andro wandering around waving his hands so now we will include the new animation function. We place the variables in the key catch argument so all is explained below.

```
// To start we need only 1 global variable.
// add global animation variables to script
int setAnimation = 0;

// Main(){ //at the top just after the Main() }
int setAnimationNew = 0; // initialize variable

// add this variable deceleration to the if key W pressed script
setAnimationNew = 1; // walk animation

// add this variable deceleration to the if key W pressed else script
setAnimationNew = 0; // wave animation

// add this function call to the bottom of the Main(){ just before the end bracket }
    bonesAnimation(setAnimationNew, 1.00);

// bones animation function, put outside tha Main(){ brackets
void bonesAnimation(int newAnimation, float animSpeed){
    if(setAnimation != newAnimation){ // test if the variables are not equal
        OUT_### = newAnimation; // update new SkinMesh animation
        setAnimation = newAnimation; // re-assign the new animation
        OUT_### = animSpeed; // set animation speed
    }
}
```

Test your script for errors and close the script, now run 3DRAD and should have a wave animation at idle and a walk as he moves about. This demonstrates how easy it is to reuse this function.

To design a game you must think how each actor should work. For example the player actor should handle all of it's functions like animations, key movements and score etc.. The NPC actor should do the same, pathfinding, animation, movement and score. Now this seems simple but when you think it's not quite as simple as you might think. For example when the player moves around the scene, especially if there is more than one NPC it will want to know where the player is, so it can have some chance of interacting with the player. We can use distance or visual but how will the NPC know how close the player without any direct connection. A vector can be placed into the global string but more about that later.

Now we have a complete basic player script including animation function.

```
// here we initialize the global variables
float headings;
float forward;
// global animations variables to script
int setAnimation = 0;

void Main(){
// Scripted Character Tutorial V1 2017
// this character is not perfect and may not
// work as well as intended. But I will later add
// some script to improve it, this will help explain
// how they will work.
// the following comment describes what the next 12 lines do
// get key commands from the player

int setAnimationNew = 0;
```

```

If(ikeyGet(Dik_W)){ // this line gets a W key click and increases forward movement
    // 6 is the walk speed when the W key is pressed
    OUT_### = 6; // this is the output value for the force object
    // add this variable deceleration to the if key W pressed script
    setAnimationNew = 1; // walk animation
}else{
    // zero forward movement when no key is pressed
    OUT_### = 0; // this is the output value for the force object
    // add this variable deceleration to the if key W pressed else script
    setAnimationNew = 0; // walk animation
}
If(ikeyGet(Dik_A)){ // turn left in degrees
    headings += 0.01;
}
If(ikeyGet(Dik_D)){ // turn right in degrees
    headings -= 0.01;
}

// initialize some variables
Vector3 objRBvector;
Quaterneion objOrientation;
RigidBody objRBody;
SkinMesh objSMesh;

iObjectLocation(objRBody,objRBvector); // get player RB position
iQuantneionFromEulia(objOrientation,Vector3(headings,0.5, 0.0)); // calculate orientation from degrees (key input
from player)
iObjectLocationSet(objSMesh,objRBvector); // set new location of SkinMesh
iObjectOrientationSet(objSMesh,objOrientation); // set SkinMesh orientation
iObjectOrientationSet( ObjForce,objOrientation); // set force orientation
// all the variables now die as the processor leaves the Main(){ loop except // for the globals

// add this function call to the bottom of the Main(){ just before the end bracket }
    bonesAnimation(setAnimationNew, 1.00);
}

// bones animation function, put outside tha Main(){ brackets
void bonesAnimation(int newAnimation, float animSpeed){
    if(setAnimation != newAnimation){ // test if the variables are not equal
        OUT_### = newAnimation; // update new SkinMesh animation
        setAnimation = newAnimation; // re-assign the new animation
        OUT_### = animSpeed; // set animation speed
    }
}

// ----- END of Main() and all it's FUNCTION ----- //

```

Hint: the player character will still move around a bit awkwardly slipping and sliding. What it needs now is adjustments to the rigidBody. The rigidBody has to act like a mono-cycle, where the wheel will roll forward easy but the not sideways so set the friction to 50000, the angular dampening to 0.75 and leave the liner damping at 0.999. These settings will make it slow down so it may be necessary to increase the value for the force object for normal walk speed (experiment).

Hint: In the above code you will see a Vector(), that is part of the iQuantneionFromEulia () 3DRAD statement. Vector3(headings,0.5, 0.0) // calculate orientation from degrees (key input from player). Notice the first variable 'headings' this is from the key input argument of the player, then next is 0.5 and the last is 0.0. What I want to discuss is the 0.5 value. This is to help the character stick to the ground. If this value were zero the force applied to the character would be straight ahead and if the character ran over an edge it would appear to fly until it came in contact with the terrain again. So it is necessary to set this value to 0.5 to keep the character in contact with the terrain. This applies to the player character and the NPC character as well.

Chapter 6 - Basic NPC Character

Now we have the Basic Player working we will build the NPC. Instead of building it in pieces like I did with the player character I will include all of the basic code necessary to complete it. There some things to consider when you build any NPC and that is how it will move around, make decisions and how it will react with the player. Because we have taken the simplest approach we will use the path method for moving around its territory. In this case we will use objects to mark out the path nodes. Using them this way you can have as many different path as NPC you create. All you have to do is position each object in the scene to mark out its path. The objects could be a building, drum, crates or tree this is up to you and your game.

The next part is how the NPC will interact with the player. A simple way to do this we will use the distance between the player and the NPC, for example when the player is less than a set distance, say 100metres the NPC will stop for a moment then face the player. To simplify the code we will include the movement code as a separate function. To make a NPC smart we will have to include a little extra code for decision making. Here we will keep it as simple as we can get. Basically it will check if the player is close and if not continue on patrol. If the player is close enough it will stop patrol and face the player. If the player becomes out of range again it will return to the patrol.

```
// NPC script
// here we initialize the global variables
// global animations variables
int setAnimation = 0;
int pathIndex = 0;
// 3 DRAD Main() function
void Main(){

// Scripted NPC Character Tutorial V1 2017

    int setAnimationNew = 0; // initialize new animation variable
    // build a path array with 4 objects
    // this array contains obj-1, obj-2, obj-3, obj-4 path object handles
    int []pathObjects={OBJ_###, OBJ_###, OBJ_###, OBJ_###}; // change object handles to suit
    Vector3 objRBvector;
    Vector3 nodeVector;

    iObjectLocation(OBJ_###, objRBvector); // player object vector
    iObjectLocation(pathObjects[pathIndex], nodeVector); // path node vector
    // here we have the NPC decision making process
    // the distance between player and npc
    float distance = iVectorLength(nodeVector-objRBvector);
    if(distance > 100){
        // continue on patrol
        // 6 is the walk speed
        OUT_### = 6; // this is the output value for the force object
        setAnimationNew = 1; // walk animation
        // get distance from npc to the next node
        float nodeDist = iVectorLength(objRBvector - nodeVector);
        if(nodeDist < 2){ // if distance to node is less than 2 meters
            pathIndex++; // next node
            // we check if the NPC has reached the last node then restart the path
            if(pathIndex >= pathObjects.length()){pathIndex = 0;}
        }
        npcMovement(nodeVector); // call npc Movement function
        animation(); // call animation function
    }else{
        //stop npc and face player
        if(distance < 101){
            // 0 speed to stop NPC
            OUT_### = 0; // this is the output value for the force object
            // we send the player vector to the movement function so
            // the NPC will look at the player instead of the path node
            npcMovement(objRBvector); // call npc Movement function
            // npc and look at player - animation idle
            setAnimationNew = 0; // idle animation
            animation(); // call animation function
        }else{
            setAnimationNew = 0; // idle animation
        }
    }
}
```

```

        animation(); // call animation function
        // return to patrol
        npcMovement(objRBvector); // call npc Movement function
    }
}
bonesAnimation(setAnimationNew, 1.00);
}

// the variables now die as the processor leaves the Main(){} loop except
// for the globals
}

// npc movement function
void npcMovement(Vector3 nodeVector){
    // initialize some variables for inside this function only
    Vector3 objRBvector;
    Quantneion objOrientation;
    RigidBody objRBody;
    SkinMesh objSMesh;

    iObjectLocation(objRBody,objRBvector); // get npc RB position
    iObjectLookAt(objOrientation, objRBvector, nodeVector); // orientation from look at function

    iObjectLocationSet(objSMesh,ObjRBvector); // set new location of SkinMesh
    iObjectOrientationSet(objSMesh,objOrientation); // set SkinMesh orientation

    iObjectOrientationSet( ObjForce,objOrientation); // set force orientation
}

// bones animation function, put outside tha Main(){} brackets
void bonesAnimation(int newAnimation, float animSpeed){
    if(setAnimation != newAnimation){ // test if the variables are not equal
        OUT_### = newAnimation; // update new SkinMesh animation
        setAnimation = newAnimation; // re-assign the new animation
        OUT_### = animSpeed; // set animation speed
    }
}
// ----- END of Main() and all it's FUNCTION ----- //
}

```

Hint: set the ridigBody with the same values as we discussed for the player ridigBody and the force objects.

Chapter 7 - Player Weapon Script Class

As you build any object you will find that the object may have many parts like, player input, movement, animation, weapon control etc. Each part can be constructed as separate scripted objects or one large script but this is not the best solutions. This is where the class object can dominate because they can be separate objects contained in the same script. This may still seem to be a lot of script in one script object but it is easier to debug as each object is working independently. Class object can be easily re-used so easily, if designed well they contain all there own code and variables.

Here is the frame work for a class. This frame work is very similar to the c and c++ frame work but I won't get into them here because they don't suit 3DRAD.

Class framework:

```
// class name and instance  
className instanceName;
```

```
class className{  
    // class properties go here  
  
    // class constructor goes here  
    className(){  
    }  
  
    // class methods go here  
    void classMethod(){  
    }  
}
```

Hint: when I develop a new class I just copy this framework into the script and it helps me remember what I need to complete the task.

Class description:

As you can see from the above framework, unlike the Main() the class name has no following parentheses only the two curly brackets that encompass all the class elements.

All the class properties variables and methods must be inside these curly brackets. If you want a class property variables put them at the top after the class name but do not initialize them with any value, only the type like, int, float, string etc. they must be outside of any class method.

To initialize the value you can do it in the class constructor which is shown next in the framework. This is a function of the class and automatically processes the variables in it, once only when the script is first run. The constructor name is the same as the class name. You can not use any 3drad statements in the constructor to initialize a variable. To do this you can however have a initialize method and call it from 3drads iInitialize() argument. If you do use them inside the constructor it will not recognize it as an error and the script will run but not process the statements.

A method is what does all the work in a class, this is where we will place all the script code. The class global variables are generally call properties. With the weapon class you will be able to provide all the values like float distance, vector normals and more for other scripts to use.

One of the reason for a class is, if designed properly you can reuse in other scripts that may have different purpose, like a NPC weapon script. In that case you just copy from the player script and paste it to the NPC script, modify some of the code to suit and change the object handles etc. I will detail the changes in a later chapter.

To declare an instance/ handle of a class you declare as shown at the top of the class framework above and here.

```
// class name and instance  
className instanceName;
```

The class name is the name you call your class and the instance name is the name for each instance of the class. What this means is you can have more than one instance of the class and each instance must have its own handle/name. For example you would declare multiple class by,

```
weaponControl npc1Weapon;  
weaponControl npc2Weapon;  
weaponControl npc3Weapon;
```

For example you can control more than one NPC. If you develop a class for each the major part like movement, path finding, animation and weapon control the script will be able to control as many NPCs as you want, depending on 3drads ability to handle it. This is done by having, for example 4 rigid bodies, 4 skin mesh and 4 scanners for the weapon all attached to one script. Then declaring 4 separate instances for the class names for each NPC and calling them from the Main(). Each NPC will act independently as if it where 4 separate scripts objects.

Chapter 7 Continued

With other codes like c++, c, JavaScript and more you have the option of making all variables Private or Public, but I have not yet been able to implement this in Angelscript. This feature is for security reasons and is not a real issue for 3DRAD. With Angelscript all class properties are Public and variables declared inside a method are Private. A Public variable would be accessible from anywhere in the script object and a Private variable is only available from within the class, function or method it was declared in.

Class Methods:

A class method is similar to a Angelscript function. The method can be called from within it's own class, from another method or from the Main() and a Main() function. You can pass a variable to it and it can also return a variable.

To call a method you use the class instance name followed by the dot and the name method name and brackets.

Example:

```
instanceName.methodName();
```

To use a class property you use the class instance name followed by the dot and property name.

Example:

```
InstanceName.propertyName;
```

Hint: It can be very helpful to have a set of method names so it is easy to identify sections of your class code. For example an initialize method use either Create or Initialize to initialize variables and Destroy for cleaning up and closing variables and statements. Update is another commonly used name to call the class's main method from 3DRAD Main(). You can also use descriptive names like I have used in the examples.

Next we will add a weapon script, this will take the form of an Angelscript Class. You could describe the class as another Main(){} with its own set of variables and functions. The Class is the root of all Object Orientated development. AngelScripts Class is a reference type, what this means is there can be multiply instances or handles of the same class. For example the character has more than one part, ai, movement, animations, weapon etc. in one script we can have more than one copy of each class objects giving the programmer the ability to produce more than one game object from the same script.

Below I have the complete code for the weapon class so wright it in after the last Main(){} bracket including the class deceleration just below.

```
// class deceleration
weaponControl weapon;

class weaponControl
{
    // class properties go here
    Vector3 hitNormal;
    Vector3 hitFaceing;
    float hitDistance;
    int delay;

    // class constructor goes here
    weaponControl(){
        delay = -1;
    }

    // class methods go here
    bool playerShoot(){ // we will return a boolean value to the argument calling it
        // initialize a temporary variables
        Vector3 npcTargetV // initialize target vector
        bool bHit = false;
        if(delay == -1 && iGetKey(DIK_SPACE)){
            Vector3 playerV; // initialize player vector
            Quaterneion objOrientation;
            iObjectLocation(OBJ_###,playerV); // get player location
            iObjectOrientation(OBJ_###, objOrientation); //player SkinMesh orientation
            playerV.x += 0.2; playerV.y += 1.2; playerV.z += 0.0; // reposition scanner in relationship to player
            position.
            iObjectLocationSet(OBJ_###,playerV); // set scanner location
            iObjectOrientationSet(OBJ_###,objOrientation); //set scanner orientation
            iObjectStart(OBJ_###); // start scanner
            delay = 120; // 2 second weapon reload delay
            // you can start the shoot sound and show some muzzle flash
            // you will also need a separate timer to hide the flash
```

```

    }
    if(delay > 0){ // delay - weapon repeat timer
        delay--;
        if(delay == 0){
            delay = -1;
        }
    }
    // you will also need a separate timer to hide the flash
    hitDistance = 0;
    // if scanner hits a target then the distance will be greater than zero
    if(IN_### > 0){ //compare scanner distance input
        bHit = true; // return true if a object is hit;
        hitNormal = IN_###; // get vector where the shot hit target
        hitFaceing = IN_###; // get vector where shot hit target angle of face
        hitDistance = IN_###; // get Distance from the target
        // if the player hits the NPC then we would send it to the data base
        // Score hits class call goes here
        // keep its own score. When it gets the score it will clear this memory
    }
    return bHit; // return boolean
} // end method
} // end class

```

Include this class into the player script object and edit all the object handle names including the input and output handles.

To access the class you must put in code to call the class playerShoot() method from the Main(). To do this we use the following code.

```

weapon.playerShoot(); // call class method

```

Place this code into the player script just below the get key code arguments of the Main().

This will call the class playerShoot() method every process loop, there are some processes in the weapon class that require processing every loop.

How it works: First the processor define the class properties variables, then it initialize the value for the delay variable inside the constructor, which is only processed once when the game first starts. Next is the playerShoot() method, and then the processor initialize the methods variables. The first if() statement is processed next, the delay timer and the key input from the player. The first part of the argument is the delay variable, this is for the weapon repeat timer. If the delay variable is -1 and the space keying pressed then weapon is fired, both conditions are true. Then it will continue initializing the vector and orientation variables. Next it gets the player SkinMesh location and orientation. On the next line we need to make adjustments for the weapon position so its relative to the player SkinMesh position. We then set the scanner at the new location and then the scanner orientation so the weapon is pointing in relation to the player SkinMesh orientation. Next we start the scanner, this will activate it. Finally for the first if() statement we set the weapon repeat variable, delay to 120 which is 2 seconds. This is done so the weapon will only shoot (start the scanner every 2seconds) for a more realistic weapon simulation. You can adjust this variable to suit what ever weapon you using. Now that the first if() statement is complete the weapon repeat timer will start counting back and while the value is still greater than zero you won't be able to shoot off another round until it reaches zero, when it sets the delay variable to minus one allowing the player to shoot another round. Remember 3drad processes the script once every process so the delay timer will only minus one from delay every process, so 120 will take two seconds to be processed. Next we set the hitDistance to zero and if there was no target hit by the scanner it will return zero or if there was a target hit it the argument would return true in the second if() statement. Next we get in all the variables we will need to set any special effects like blood splat and distance. The vectors are used for setting the position of the blood splat etc.. To get a variable you would, for example use float dist = weapon.hitDistance; from the Main(). I haven't implement any code for these variables in our simple player sample.

You could use hitDistance value to simulate the weapon range by limiting the distance it can shoot so it would simulate different weapon types.

Hint: for better accuracy attach a ridge body to simulate shoots that hit the body.

Chapter 8 - NPC weapon script class

Below I have the complete code for the NPC weapon script class so write it in after the last Main(){} bracket including the class deceleration just below.

```
// class deceleration
npcWeaponControl npcWeapon;

class npcWeaponControl
{
    // class properties go here
    Vector3 hitNormal;
    Vector3 hitFaceing;
    float hitDistance;
    int delay;

    // class constructor goes here
    npcWeaponControl(){
        delay = -1;
    }

    // class methods go here
    bool npcShoot(){ // we will return a boolean value to the argument calling it
        // initialize a temporary variables
        Vector3 playerVector // initialize target vector
        Vector3 npcVector // initialize npc vector
        bool bHit = false;

        iObjectLocation(OBJ_###, playerVector); // player object vector
        iObjectLocation(OBJ_###, npcVector); // npc vector
        float playerDist = iVectorLength(playerVector - npcVector);

        if(delay == -1 && playerDist < 20){ // if distance to player is less than 20 meters
            Quaterneion objOrientation;
            iObjectOrientation(OBJ_###, objOrientation); //npc SkinMesh orientation
            npcVector.x += 0.2; npcVector.y += 1.2; npcVector.z += 0.0; // reposition scanner in relationship to player
            position.
            iObjectLocationSet(OBJ_###,npcVector); // set scanner location
            iObjectOrientationSet(OBJ_###,objOrientation);//set scanner orientation
            iObjectStart(OBJ_###); // start scanner
            delay = 120; // 2 second weapon reload delay
            // you can start the shoot sound and show some muzzle flash
            // you will also need a separate timer to hide the flash
        }
        if(delay > 0){ // delay - weapon repeat timer
            delay--;
            if(delay == 0){
                delay = -1;
            }
        }
        // you will also need a separate timer to hide the muzel flash
        hitDistance = 0;
        // if scanner hits a target then the distance will be greater than zero
        if(IN_### > 0){ //compare scanner distance input
            bHit = true; // return true if a object is hit
            hitNormal = IN_###; // get vector where the shot hit target
            hitFaceing = IN_###; // get vector where shot hit target angle of face
            hitDistance = IN_###; // get Distance from the target
            // if the NPC hits the player then we would send it to the data base
            // Score hits class call goes here
            // keep their own score. When it gets the score it will clear this memory
        }
        return bHit; // return boolean
    }
}
```

Chapter 8 Continued

Include this class into the npc script object and edit all the object handle names including the input and output handles.

To access the class you must put in code to call the class npcShoot() method from the Main(). To do this we use the following code.

```
npcWeapon.npcShoot(); // call class method
```

Place this code into the npc script just before the delay weapon repeat timer.

This will call the class npcShoot() method every process loop, there are some processes in the weapon class that require processing every loop.

How it works: First the processor define the class properties variables, then it initialize the value for the delay variable inside the constructor, this is only processed once when the game first starts. Next is the npcShoot() method, there the processor initialize the methods variables. It will then get the player location and npc location, so we can calculate the distance between them using the iVectorLenght(). The argument in the first if() statement is processed next. The first part of this argument in the if() is the delay variable, this is for the weapon repeat timer. If the delay variable is -1 and the distance between the npc and the player is less than 20 the weapon is fired, both conditions are true. It then will continue and initializing the orientation variable after the if() statement. We already have the NPC SkinMesh location so we don't need to do that again but we want the npc orientation. We need to make adjustments for the weapon position in relationship to the npc so we do this on the next line. We then set the scanner location and position it at the new location. Followed by the scanner orientation so the weapon is pointing in relation to the NPC SkinMesh orientation. I have used this method of aiming the weapon because it is less accurate than the iObjectLookAt() which can be deadly every time. Next we start the scanner, this activates it. Finally for the first if() statement we set the weapon repeat variable, delay to 120 which is 2 seconds. This is done so the weapon will only shoot (start the scanner every 2seconds) for a more realistic weapon simulation. You can adjust this variable to suit what ever weapon you using. Now that the first if() statement is complete the weapon repeat timer will start counting back and while the value is still greater than zero you won't be able to shoot off another round until it reaches zero, when it sets the delay variable to minus one. Remember 3drad processes the script once every process so the delay timer will only minus one from delay variable every process, so 120 will take two seconds to be processed. Next we set the hitDistance to zero and if there was no target hit by the scanner it will return zero or if there was a hit it will return a positive value. If there was a target hit it would return true in the second if() statement. Next we get in all the variables we will need to set any special effects like blood splat and distance. These vectors are used for setting the position of the blood splat etc.. To get a properties variable you would, for example use float dist = weapon.hitDistance; from the Main(). I haven't implement any code for these variables in our simple NPC sample.

You could use this hitDistance value to simulate the weapon range by limiting the distance it can shoot so it would simulate different weapon types.

Chapter 9 - Orbit cam control

The next camera rotating around the player is a little more complex. This script has only 1 camera and using the keys or mouse you can rotate around the central character, move up and down and inward and outward from the character. When the camera is close to the ground it stops and won't go any further..

```
///Rotate a CamChase object about a specified target object
///USAGE: link this script object to the CamChase object in your
//      project and to the object you want to rotate about.
//
//REMARKS: in the CamChase property dialog, in the relationships
//      window, set the target object to 'CHASE' and the
//      Script object to 'IGNORE'.

//      In the code below, make sure the following variable names
//      are consistent with the declarations in the windows to left:
//
// OBJ_### - CamChase object to control
// OBJ_### - Object to take
// OBJ_### - Text print object
//

// Orbit Camera Control V2
OrbitCameraControl orbit;

// Orbit Camera mouse or key
class OrbitCameraControl
{
    float orbitSpeed;
    float setSpeed;
    float Radius;
    float Altitude;
    float setAltitude;
    float minAltitude;
    float height;
    float Time;
    float lastMouseZ;
    float screenX;
    float screenY;
    int timer;
    bool swap;
    bool useMouse;
    string message;

    OrbitCameraControl(){
        setSpeed  = 0.75; /// orbit set speed
        Radius    = 5;    /// maximum orbit radius
        setAltitude = 5;   /// altitude speed
        minAltitude = 2;   /// minium altitude
        screenX    = 7;    /// screen max and min X co-ords
        screenY    = 6;    /// screen max and min Y co-ords
        // don't alter the variables below
        orbitSpeed = 0.0;
        Altitude   = 0.0;
        Time       = 0.0;
        timer      = -1;
        swap       = true;
        useMouse   = true;
        initMouse();
    }
    void initMouse(){
        iMouseLookSet(0.0,0.0); // set initial position
    }
}
```

```

void CameraControl(){
    ///      Camera object
    int theOrbitCam = OBJ_66;
    ///      the target object (may be change as required )
    int theTargetObject = OBJ_0;
    ///      Text print
    int txtPrint = OBJ_88;
    // press F2 for optional key control or Mouse control
    if(iKeyDown(iKeyCode("DIK_F2")) && timer == -1){
        if(swap){
            swap = false;
            useMouse = false;
            timer = 120;
            message = "F2 - Keys";
        }else{
            swap = true;
            useMouse = true;
            timer = 120;
            message = "F2 - Mouse";
        }
    }
}

///// increment x and y orbit angle
float x = iMouseX() * 32.0 - 16.0;
float y = 12.0 - iMouseY() * 24.0;

if(useMouse){
    if(iMouseX() > x)orbitSpeed = setSpeed; // check mouse screen location
    if(iMouseX() < x)orbitSpeed = setSpeed; // check mouse screen location

    if(x > screenX){
        Time += orbitSpeed; // add orbit speed to time to rotate right
    }else if(x < -screenX){
        Time -= orbitSpeed; // add orbit speed to time to rotate left
    }else{
        orbitSpeed = 0;    // add orbit speed to zero to stop rotate
    }

    ///// pan up or down
    if(iMouseY() > y)setAltitude = setSpeed; //check mouse screen location;
    if(iMouseY() < y)setAltitude = setSpeed; //check mouse screen location
    if(y > screenY){
        Altitude += setAltitude * 0.05; // add altitude to raise cam
    }else if(y < -screenY){
        Altitude -= setAltitude * 0.05; // minus altitude to lower cam
    }else{
        setAltitude = 0;    // stop altitude movement
    }
    if(Altitude < minAltitude)Altitude = minAltitude; // keep camera above ground

    if(iMouseZ(false) > lastMouseZ)Radius -= 0.4;    // increase radius to move away
    if(iMouseZ(false) < lastMouseZ)Radius += 0.4;;    // decrease radius to move closer
    lastMouseZ = iMouseZ(false);
}else{ ///// key board controls
    orbitSpeed = setSpeed;
    if(iKeyDown(iKeyCode("DIK_A")))Time += orbitSpeed;    // add orbit speed to time to rotate right
    if(iKeyDown(iKeyCode("DIK_D")))Time -= orbitSpeed;    // minus orbit speed to time to rotate left
    setAltitude = setSpeed;
    if(iKeyDown(iKeyCode("DIK_Q")))Altitude += setAltitude; // add altitude to raise cam
    if(iKeyDown(iKeyCode("DIK_Z")))Altitude -= setAltitude; // minus altitude to lower cam

    if(Altitude < minAltitude)Altitude = minAltitude;    // stop altitude at minium altitude
    ///// zoom camera in or out
    if(iKeyDown(iKeyCode("DIK_E")))Radius += 0.4;    // increase radius to move away
    if(iKeyDown(iKeyCode("DIK_C")))Radius -= 0.4;    // decrease radius to move closer
}

```

```
    if(Radius < 4)Radius = 4;
}
// set camera rotation location
Vector3 camLocation;
Vector3 targetLocation;
iObjectLocation(theTargetObject,targetLocation);
camLocation.x = targetLocation.x + iFloatCos(Time)*Radius;
camLocation.y = targetLocation.y + Altitude;
camLocation.z = targetLocation.z + iFloatSin(Time)*Radius;
iObjectLocationSet(theOrbitCam,camLocation);

// delay timer
if(timer > 0){
    timer--;
    if(timer == 0){
        timer = -1;
        message = "";
    }
}
iPrint(message,-5,10,txtPrint);
}
```

Chapter 10 - Scoring

In this chapter we will setup a scoring system using the class data base.

As I said earlier the player script should control it's own behavior so that it will be independent and the same for the NPC. But the scanner of the player returns all the data for the NPC and the NPC returns player data. So if the player has run out of lives how is it going to know how many times it has been hit. To store the data we will use the 3DRAD iGlobalString() statement. The global string can be accessed from both player and NPC scripts and will remain in memory if you close a game level and load the next level and only dies when the game is finally closed.

I have included 2 methods, one to save the properties value to the memory and two for loading the memory. For displaying the score and finally saving or loading from a file you may want a separate script to control this it may be the menu script.

The data base class is simple and will be required in both player and NPC scripts. Below is the code for the class.

```
Class framework:
// class name and instance
scoreDataBase playerData;

class scoreDataBase{
    // class properties go here
    int scoreHits;
    int scoreLives
    // class constructor goes here
    scoreDataBase(){
        scoreHits = 0;
        scoreLives = 0;
    }
    void saveToMemory(){
        string getGS;    // create a string variable
        iGlobalStringGet(getGS,2);    // get the content of string memory 2
        if(getGS != ("" + scoreHits)){    // memory is not equal to score hit True
            iGlobalStringSet("" + scoreHits, 2);    // then save new value to 2
        }

        iGlobalStringGet(getGS,3);    // get the content of string memory 3
        if(getGS != ("" + scoreLives)){    // memory is not equal to score hit True
            iGlobalStringSet("" + scoreLives, 3);    // then save new value to 3
        }
    }
    void loadFromMemory(){
        string getGS;    // create a string variable
        iGlobalStringGet(getGS,2);    // get the content of string memory 2
        scoreHits = iStringVal(getGS); // initialize hits value

        iGlobalStringGet(getGS,3);    // get the content of string memory 3
        scoreLives = iStringVal(getGS); // initialize Lives value
    }
}
```

To set a class data base property and save it to memory you use the following code.

```
playerData.scoreHits += 1;    // set class scoreHits value
playerData.saveToMemory();    // send to memory
```

In the weapon class you will find the line below, replace it with the 2 lines above.

// Score hits class call goes here

The same for the NPC class except for the instance name.

The NPC code will be the same as the player except for the instance name and the memory location otherwise it would overwrite the player data.

Chapter 10 Continued

The class hit properties would be updated every time the scanner hits a target in the weapon script. The player would keep its own hit score which it would get from the NPC via the global string, by calling the memory location. When the hits are equal to the maximum hits the player character will begin end life procedures. I will detail this later. All these procedures are the same on the NPC except for the name and memory changes.

To get the global string data you use the following.

```
playerData.loadFromMemory();
```

Now the above line must go in the player Main() so it can keep a track of the NPC hits on the player and it's own life. When the hits equal the maximum number of lives then the player script will end it's own life. To do this it will get the values from the NPCs allocated memory for the hits property. Remember the player gets its hits from the NPC and the NPC gets its hits from the player allocated memory. So when you setup your player and NPC then it's up to you to allocate the memory and modify the scripts to suit.

How it works:

The data base classes are generally simple just a bunch of property variables that can be accessed from the scripts or saved to global string. You could implement load and save to file methods as part of this class. This would be easy to expand with more variables depending on your game.

The class properties are first below the class name declaration. Next the constructor initializes the properties values to zero. Followed by the save to memory method. First we initialize the string variable to use in the 3DRAD iGlobalStringGet() statement. This will get the variable from the global string memory location. Then with the if() statement we compare it with the current value in the Hits property. But before we can do this we must compare it as a string like "" + scoreHits. Putting the double quotes with the plus argument allows it to be compared as a string. If its not equal then it is true and will then sets the new value in the global string. It then follows through to the next variable and does the same. As you can see doing it this way if only one variable has been changed it will not alter the other. When you call the method the variable must be altered first.

Everything in this chapter is the same for NPC as the player except for the player allocations and names I am leaving it up to you, it's time for you to start thinking for yourself.

Hint: Strings can be joined by using the plus '+' operator for example "" + scoreHits. You can also do this "score " + scoreHits but plus operator won't add numbers that are strings like "1" + "2" does not equal 3 it will print 12.. But you can add ints and floats by doing this "score " + (scoreHits + damage)

Chapter 11 - Programing AI code methods theory

Here I will talk about different programing methods used in making an AI character. I am still going to stay with the KISS principles because this book is aimed at new programers.

First we will look at the basic sense, decide, act. Sense is when the NPC must have basic knowledge about its surroundings and be aware of intruders. Decide would then make a decision on what to do if the NPC was alarmed. Act, the NPC take the appropriate action. Sounds simple, if you look at modern games you would know the NPC are very smart as they rely on sight, movement and sound for their sensors as we humans do. All of these principles can be duplicated in 3DRAD script but as we all know rad can become bogged down so forcing us to look for more basic sensors.

I have already demonstrated distance as a sensor, this is when the player comes within a set distance of the NPC then it will react. This is the most basic and in many ways can be reasonably effective. There is also sight, well not exactly sight but a position that is in a field of view and depth of field of an NPC. This method compares the position of the NPC with the player position and if it is within say 15deg to the left or right of the center (overall 30deg) of the NPCs current path and within the depth of field. This is a very good method of sight for the NPC. There are other method but I won't gone into them here.

Presudo code for basic NPC AI.

Decision will turn out to be a series of if() statements that have different arguments that will direct the process to the required Action. I have some presudo code below that will be easily converted to Angelscript or other code languages.

```
if (enemy shoot at NPC) then. // sensor - sound
    If(hits greater than max hits)then
        NPC dead
    Else
        If( npc hit) hitsNPC +1; npc hit false; // sensor - sound
        Run for cover
    End if.
Else
    If(not see enemy) then // sensor - sight
        Continue on patrol
    Else
        Attack enemy function
    End if
End if

Function attack enemy()
    If(not see enemy) then // sensor - sight
        Return to patrol
    Else
        If (not close to enemy)then // sensor - distance
            Move closer
        Else
            Shoot enemy
        End if
    End if
End function

Function Continue on Patrol(){
    If (close to node) // sensor - sight
        Get next node
    Else
        Continue movement and animation
    End if
End function
```

Like I said this is basic AI but should provide good behavior from your NPC.

Chapter 11 Continued

How it Works:

The first if() argument checks (enemy shoot at NPC) and if true then checks if (hits greater than max hits) if this is also true then it will go into the NPC die routine. If its false then it would check if NPC is hit then increase the hits score for the NPC and set the 'npc hit' to false so it doesn't keep increasing the score. Next it looks for the nearest cover position then run to it. All the following arguments are skipped by the Else statement. The argument 'npc hit' would be set by the player/enemy because they have the weapon scanner.

Now if (enemy shoot at NPC) is false it would skip the first set of arguments and go to the Else statement were the if() would check (not see enemy) and if this is true then it would continue on patrol. Else if it did see the enemy it would be false so then it will go to Attack enemy function. In the attack function we first check if (not see enemy) can't see the enemy, if that's true we return to patrol else we check if (not close to enemy) is close enough, true so we move the NPC closer to the player else shoot at the enemy.

The last function Continue on Patrol looks simple but would contain all movement functions and statements to get the NPC moving and following the path nodes. The if() statement argument (close to node) checks if the NPC is close to the next node of path it will follow. If true it will index the path array so the NPC will continue on the path. If (close to node) is false it will continue moving to the next node.
