Programming for Artists and Designers

Tutorial 8: The Pawn

Aims: To introduce you to the pawn class and the concept of controllers

Introduction:
The pawn and controllers like many things in UT are massive areas, this tutorial aims to give you some insight in to the pawn class, how to include pawns in games and altering some functionality. We'll also be going over how to include you own variables and functions.
As the pawn class is massive it is impossible to cover everything, especially as a lot of documentation isn't available, that said there is a lot of information available, so the solution to this is to read – look at the pawn class and classes derived from it, look at definitions of the data and methods presents in these classes – such definitions can be found at places like the unreal wiki.

Note:
A UT2003 player is composed of a number of classes that all work together to accept and process input from the player and handle collision with the world. Bots use much the same system, so processing of AI also takes place (for Bot-specific subclasses; more on that in a minute) within the same code. What follows is an attempt to explain the different classes involved and how they interact.
Controller = the base interface to a pawn.

- Receives events from the engine that are destined for the Pawn,
- Handles AI,
- Acts as an animation interface,
- Controllers are partially implemented using native code and have native replication,
- Most of the time, each Controller has a Pawn attached to it,
- A good way to visualise the Controller/Pawn interface is to consider the Controller the "brain" of a Pawn (which is the "body").

Assessment Note:
It may be possible to implement some of the more difficult ideas through inclusion of a pawn or controller class for the game type.

First a note on GameInfo

The GameInfo in this example is very simple. Its only purpose is to specify the HUDType, the ScoreBoardType , the PlayerControllerClassName, the DefaultPlayerName, and the GameName. This game info does not create any new game type or store any new state about the current game.

The Class

As mentioned above, this GameInfo is very simple.

```
class TopKillerGameInfo extends GameInfo;
```

```
defaultproperties
{
   HUDType="MyPackage.MyHUD"
   // You could add a score board here if you wanted
   // ScoreBoardType=" MyPackage.MyScoreboard"
   PlayerControllerClassName=" MyPackage.MyController"
   DefaultPlayerName="Player"
   GameName="TopKiller"
}
```

Note: The Warfare version of this code additionally has the line: "bDelayedStart=false" which starts the game with no delay or waiting for the user to click. This line is not necessary for the Runtime or the UDNBuild because they already have bDelayedStart set to false in GameInfo.uc.

Once the simple game info that specifies these things has been created, the game INI file must be updated to use this new GameInfo by default. To do this we need to alter our packages ini file to look like this:

```
Object=(Class=Class,MetaClass=MyPackage.TopKillerGameInfo,Name
=XGame.xDeathMatch,Description="DM|DeathMatch|xinterface.Tab_I
ADeathMatch|xinterface.MapListDeathMatch|false")
```

The underlined bit should be the only bit that has changed.

What is a Pawn

First a note on the classes leading up to pawn:
Object is the parent class of all objects in Unreal. All of the functions in the Object class are accessible everywhere, because everything derives from Object. Object is an abstract base class, in that it doesn't do anything useful. All functionality is provided by subclasses, such as Texture (a texture map), TextBuffer (a chunk of text), and Class (which describes the class of other objects).

Actor (extends Object) is the parent class of all standalone game objects in Unreal. The Actor class contains all of the functionality needed for an actor to move around, interact with other actors, affect the environment, and do other useful game-related things.

Pawn (extends Actor) is the parent class of all creatures and players in Unreal, which are capable of high-level AI and player controls.

For many of the things you wish to do for assessment 1 you will be accessing and altering pawn properties.

Why Extend Pawn

We extend the functionality of pawn, rather than actor, as often we need to inherit many of the pawn class functions, such as shield strength and many dealing with animation of the model.

Variables such as HealthMax, GroundSpeed and AirSpeed are also declared in the pawn class.

These links give you good insight into the pawn class:
http://wiki.beyondunreal.com/wiki/Pawn
http://wiki.beyondunreal.com/wiki/Actor

Open up and have a look at the pawn and actor classes, there is a lot inside them. Don't be disheartened though, by working through the file slowly it is possible to a least have a good idea of what the majority of the functions and variables do.
You can even use the wiki links above to help identify what certain parts do.

Creating Our Pawn
The first thing to do is declare our class:

```
////////////////////////////////////////////////////////////
// Pawn class for our game - TopKiller
////////////////////////////////////////////////////////////

class CVGPawn extends xPawn
      config(user);
```

All we are doing is here is extending our pawn from the xPawn class. The xPawn class is the final line of inheritance in the pawn classes:

Object :: Actor :: Pawn :: UnrealPawn :: xPawn

Declaring Our Variables
The original xpawn as it stands doesn't need much more doing it, most often what it needs is user defined meshes and skins or different functionality entirely.

```
var float       Age;
var float       AgeMax;

var float       Weight;
var float       WeightMax;
```

Here we are adding an entirely new set of variables so we can tell how old our pawn is or how much he weighs.

Functions for our Variables
What we are doing below is declaring an entirely new set of functions that make use of our variables.
You will notice that when pawns take damage or get given health, shield, and adrenaline and even when pick-ups get activated, such as double damage, variables get altered and functions get called. These are functions and variables are all part of the pawn class.
Look at the functions: **TakeDamage, KilledBy** and **AddVelocity**
Look at the variables **bInvis** and **bUdamage**

```
//====================================
// Increase or decrease pawn variables


function AwardWeight(float amount)
{
    Weight += amount;
    Weight = Clamp( Weight, 0, WeightMax );
}

function AwardAge(float amount)
{
    Age += amount;
    Age = Clamp( Age, 0, AgeMax );
}



//---------------------------------
```

These functions are very similar to those declared in the other pawn classes. They are called to award an **amount** of **Age** or **Weight** to the pawn.

We see how they can be used later on.

*Note: Sometimes it is best to declare variables in the controller. AwardAdrenaline is actually part of the controller class, yet AddShieldStrength is in the xPawn class.

Using our Variables

Below we use the function **Tick**, which we covered in another tutorial. Many classes commonly use this function. Every **Tick** of time it will perform the following actions.

```
//---------------------------------
//---------------------------------

simulated function Tick(float DeltaTime)
{
      local float headscale;


      headscale = Weight/15;

      SetHeadScale(headscale);


    TickFX(DeltaTime);

      Super.Tick(DeltaTime);  //Call previous instances

}
```

```
//--------------------------------

//--------------------------------
```

The idea of this is to see if we can make our models head look bigger in proportion to our **Weight**. This is a very basic script and in reality we may have something more complex where the actual full model and mesh used is based on the **Weight**.
With this the model doesn't increase its collision size or any other factors that would be a part of growing really big.

DefaultProperties
Below are the default properties of our pawn.

```
DefaultProperties
{

    Skins(0)=Texture'PlayerSkins.NightFemaleABodyA'
    Skins(1)=Texture'PlayerSkins.NightFemaleAHeadA'

   HealthMax=400.000000
   Health=400

   Weight = 0; // set starting weight

// mod. Mark Walsh
    AgeMax = 110
    WeightMax = 250

// mod. Mathias Fuchs
    //RequiredEquipment(0)="None"
    //RequiredEquipment(1)="None"
    GroundSpeed = 1000.000000
    WaterSpeed = 1000.000000
    AirSpeed = 1000.000000
    JumpZ = 1000.000000
}
```

The first thing we have done with our pawn is give it some default skins that are different to those specified in the original **xPawn** class.
Next we have specified the maximum limit for our characteristics – pretty simple.
Then I've shown you how to deprive a pawn of starting weapons, before moving on to showing a very crude way of making the pawn permanently fast. It is possible to alter these values for periods of time in game. The speed combo (when you use adrenaline) multiplies the base speed. Pawns such as monster ones in invasion may add to the **xPawn** class by increasing such things as the base speeds and adding additional AI functions.

Testing the Pawn

The easiest way of testing a pawn.
The simplest way to test it is to open up your User.ini file, found in the system folder. The top of it should something look like:

```
[DefaultPlayer]
Name=MyCharacterHandle
Class=Engine.Pawn
Character=Nebri
team=0
```

By changing the `Class` to your pawn when you load up a game you will have all the relevant values.
Example:
`Class=MyPackage.MyPawn` or `Class=CVGpackage.CVGPawn`

Final Touches

Try the code by downloading the **WeightPickup** file from my website; incorporate this into the .int file using the following code:

Load up a game and use the console to summon these pickups:

```
Summon CVGpackage.WeightPickup
```

You should watch your weight increase as you go over them.

Also Download the HUD from my website.
Don't forget to out this HUD in your controller and game type.
You will also need to do a find-replace for every instance of CVGPawn and replace it with something else, that's if you have called your pawn something else.
You will also need to copy the .utx file to the textures directory.

Simple Third-Person Camera

The easiest way to get a third person camera is to type "behindview 1" at the console when you are in the game. This will switch the camera to a third-person camera behind the pawn. To switch back to a first-person view type "behindview 0" at the console. Some builds of the engine (2226 code-drop and builds based on 2226) have an additional function ToggleBehindView which is bound to 'b', so all you have to do to switch between first-person and third-person is to press the 'b' key.

This may prove useful for when you want to test if your code is doing what you want it to. I.e. are you invisible.

Controllers

Controllers are non-physical actors that can be attached to a pawn to control its actions. PlayerControllers are used by human players to control pawns, while AI Controllers implement the artificial intelligence for the pawns they control. Controllers take control of a pawn using their Possess() method, and relinquish control of the pawn by calling UnPossess().

Controllers receive notifications for many of the events occurring for the Pawn they are controlling. This gives the controller the opportunity to implement the behaviour in response to this event, intercepting the event and superceding the Pawn's default behaviour.

The controller class is a handle to a pawn: Pawns being the physical player characters and the controller being the controlling entity moving it around. There are two subclasses from this class, both are important: AIController and PlayerController.

Coding a Flying/Walking Controller

Flying and Walking

To easily switch between walking and flying there are three exec functions: Fly, Walk, and ToggleFlyWalk. These functions are the same functions that are in RTPlayerController in the runtime. These functions are not cheating functions because they are defined in the controller (not in CheatManager.uc) and can therefore be used in net games. To work in net games properly these function also need to be replicated to the server properly. Replication and net games is coming up near the end of the module, as replication is considered difficult.

```
exec function Fly()
{
   if ( Pawn != None )
   {
      Pawn.UnderWaterTime = Pawn.Default.UnderWaterTime;
      ClientMessage("You feel much lighter");
      Pawn.SetCollision(true, true , true);
      Pawn.bCollideWorld = true;
      GotoState('PlayerFlying');
   }
}

exec function Walk()
{
   if ( Pawn != None )
   {
      Pawn.UnderWaterTime = Pawn.Default.UnderWaterTime;
      Pawn.SetCollision(true, true , true);
      Pawn.SetPhysics(PHYS_Walking);
      Pawn.bCollideWorld = true;
      GotoState('PlayerWalking');
   }
}

exec function ToggleFlyWalk()
{
   if(IsInState('PlayerFlying'))
      Walk();
   else
      Fly();
```

```
}
```

What does this Exec Mean

Exec Functions are functions that a player or user can execute by typing its name in the console. Basically, they provide a way to define new console commands in UnrealScript code.

Valid Places for Exec Functions

Exec functions can be placed anywhere, but they only actually work in the following classes:

GameInfo – only server-side (since the GameInfo actor isn't replicated to clients)
PlayerController
CheatManager? – natively supported extension within PlayerController
PlayerInput – same as CheatManager?
Pawn
Weapon – only for the currently held weapon, not all weapons in the player's inventory
HUD
Console

Note that an exec function must be simulated to work client-side.

This means that for your Controller functions to work you will need to bring down the console in game and then type “fly” or “walk”.

Additional Code

The "Set" Command

The "Set" command is a general and very useful debugging tool that works for all actors, not just pawns. The idea is that you can change variables of actors mid-game simply by typing something at the console. The format of the "Set" command is as follows:

SET <ClassName> <VariableName> <NewValue>

The first parameter string a class name, the second string a variable name, and the third string is a value. All objects of the given class (including subclasses) will have the given variable set to the given value. For example “SET PAWN JUMPZ 4000" will set the jump velocity of all pawns, including your character, to 4000. Below are some more examples of the "Set" in use. (Note: Capitalization does not matter.)

set Pawn CollsionRadius 10 -- sets the collision radius of all pawns to 10.

set ZoneInfo bDistanceFog false -- turns off distance fog for all zones.

set Pawn Mesh SkeletalMesh'UDN_CharacterModels_K.GenericFemale' -- sets the mesh of all pawns to the UDN example girl model.

set Pawn ConstantAcceleration (X=0,Y=0,Z=1300) -- sets the additional falling acceleration of all pawns. This is added on top of gravity. 1300 is still less than normal gravity so pawns will still fall, just much more slowly.

All values are entered in to the form they are entered in for defaultproperties for the given variable type.

Important Links:

This is a very important link that will also help you immensely with your project: http://udn.epicgames.com/Technical/UnrealScriptReference
It has been put in a couple of places with indications you should read it, but here it is again.

A good look at some important aspects of the game:
http://wiki.beyondunreal.com/wiki/Unreal_Engine_And_Game_Code_Overview