Programming for Artists and Designers

Tutorial 4: Continuing with Game Types

Aims
This tutorial aims to give a greater understanding game types and to allow us to script our own more complicated game type.

Note
As you have a go at writing each piece of code read the notes that go with it, they not only describe what your doing but also certain describe what each line means to unreal script

Links
http://wiki.beyondunreal.com/wiki/GameInfo
http://wiki.beyondunreal.com/wiki/TeamGame

The Power of Game Types
Mutators can do some cool stuff. They are pretty easy to understand and they can do a lot of things by interacting with the game. They can be mixed and matched to get even cooler effects, but they are not very powerful. If you want to make a new type of game (say a Jailbreak style mod) you can't do it with mutators. You need to have complete control over the game rules. That's where the GameInfo series of classes come into play.

GameInfo is a class located in Engine. It is created by the game engine and is the core of the game play rules. Unreal Tournament 2003 makes use of a series of GameInfo. The inheritance generally follows this eries: *Actor >> Info >> GameInfo >> UnrealMPGameInfo* with further inheritance following.

UnrealMPGameInfo and DeathMatch contain code that is universal to all of Unreal Tournament's game types. *xDeathMatch* contains the code for running a normal death match. *TeamGame* contains code for code that is inherited by team games such as bombing run and CTF (capture the flag).

The first step in writing your new game type is to determine which class to subclass. If you are writing a team game, you'll want to subclass TeamGame or CTFGame. If you are writing a game without teams, use xDeathMatch. If you are writing a game that departs significantly from any previously styled game type, use *UnrealMPGameInfo*. Subclassing is very beneficial as you immediately inherit all of the code in your parent classes.

Despite the potential power of game types we will be aiming to keep it relatively simple, we are not trying to design a complete new mod like invasion.

The GameInfo class

As we saw last week, the GameInfo class was the root class of the game type.
The GameInfo defines the game being played: the game rules, scoring, what actors are allowed to exist in this game type, and who may enter the game. While the GameInfo class is the public interface, much of this functionality is delegated to several classes to allow easy modification of specific game components. These classes include GameInfo, AccessControl, Mutator, BroadcastHandler, and GameRules. A GameInfo actor is instantiated when the level is initialized for gameplay (in C++ UGameEngine::LoadMap( )).

The class of this GameInfo actor is determined by (in order):

- the DefaultGameType if specified in the LevelInfo
- the DefaultGame entry in the game ini file (in the Engine.Engine section), unless it's a network game in which case the DefaultServerGame entry is used.

Team Game

Last we looked at, and coded, a simple game type that involved basic one player games. This time we will be coding a game where we alter the properties of a team game.

If we open up unreal editor and look at the actor classes (remembering to un-tick the “Placeable actor class” selection) we see that after the DeathMatch class there is a split and into standard single player death match type games and team games.

Inheritance: *UT2003 :: Actor >> Info >> GameInfo >> UnrealMPGameInfo >> DeathMatch >> TeamGame*

Team games account for a lot more code and dealing with comes with its own complexities.

Coding a Team Game

Inheritance: *UT2003 :: Actor >> Info >> GameInfo >> UnrealMPGameInfo >> DeathMatch >> TeamGame >> CTFGame >> xTeamGame*

We will start by extending the xTeamGame class: xTeamGame just adds the precaching functions to TeamGame.
CTFGame adds a few other functions, in the whole they add very little functionality to TeamGame.

```
//=============================================================================
//    Game type called SpeedGame, both teams move quickly and
//    have an extra edge
//=============================================================================


class SpeedGame extends xTeamgame;
```

With our game, this time, we do not need to define any variables as there is no information we actually need to store.

```
//=============================================================================

function SetPlayerDefaults(Pawn PlayerPawn)
{

       if (PlayerPawn.PlayerReplicationInfo.Team == Teams[1])
       {
              PlayerPawn.AirControl = 660.000000;
              PlayerPawn.GroundSpeed = 880.000000;
              PlayerPawn.WaterSpeed = 880.000000;
              PlayerPawn.AirSpeed = 1000.000000;
              PlayerPawn.Acceleration = PlayerPawn.Default.Acceleration;
              PlayerPawn.JumpZ = PlayerPawn.Default.JumpZ;
              BaseMutator.ModifyPlayer(PlayerPawn);
              PlayerPawn.AddShieldStrength(200);
       }
      else
       {
              PlayerPawn.AirControl = 460.000000;
              PlayerPawn.GroundSpeed = 480.000000;
              PlayerPawn.WaterSpeed = 480.000000;
              PlayerPawn.AirSpeed = 600.000000;
              PlayerPawn.Acceleration = PlayerPawn.Default.Acceleration;
              PlayerPawn.JumpZ = PlayerPawn.Default.JumpZ;
              BaseMutator.ModifyPlayer(PlayerPawn);
              PlayerPawn.EnableUDamage(100);
       }

}
```

This is the meat of the game where we alter the properties of the players in the game in an attempt to give it a new feel.

The `SetPlayerDefaults` method sets the default characteristics of the PlayerPawn. In this case they sit AirControl, GroundSpeed, WaterSpeed, AirSpeed, Acceleration, JumpZ. Then it allows the Mutator list to augment these values - `BaseMutator.ModifyPlayer(PlayerPawn);`
We have used these functions to alter the properties of the player pawn, it is possible to make a pawn faster by using the actual pawns default speeds – most games just load the pawns own defaults, here however we are altering speeds of the pawn ourselves then calling the mutators that have been selected, finally we are altering the players ourselves – we are giving one team shield strength and the other team doubledamage.

It is important to note that the team values are assigned to a player – his team value is equal to either that stored in `Teams[1]` or `Teams[0]`. The Team 0 value refers to the first team, generally red; The Team 1 value refers to the second team, generally blue.

We simply compare the players team value (`PlayerPawn.PlayerReplicationInfo.Team`) against the team values and then alter the teams accordingly. One team, the second team, is much faster than normal and has a shield. The second team is slightly faster than normal and has double damage enabled.

As you may have noticed there is no call to super – the base methods just calls `BaseMutator.ModifyPlayer(PlayerPawn);` therefore there is no need for us to call the previous method, as it is simple for us to include that functionality.

```
// When player starts - give them weapons according to team

function AddGameSpecificInventory(Pawn p)
{
     if (p.PlayerReplicationInfo.Team == Teams[0])
     {
               Super.AddGameSpecificInventory(p);

                p.CreateInventory("XWeapons.LinkGun");

     }
     else
     {
               Super.AddGameSpecificInventory(p);

               p.CreateInventory("XWeapons.BioRifle");
     }
}
```

There are two methods designed for us to add items, read weapons for most games, to a player's inventory:

AddDefaultInventory (Pawn PlayerPawn)
Add to the Pawn specific Inventory items required for the game. In this case, the base mutator is queried for the default weapon and it is added to the inventory. Then it calls SetPlayerDefaults.

AddGameSpecificInventory (Pawn p)
Add to the Pawn specific Inventory items required for the game. This is the same as AddDefaultInventory method except that it does not call SetPlayerDefaults when it is done.

As we are altering the air speed etc of the pawns we wanted to avoid calling AddDefaultInventory as SetPlayerDefaults would then be called, potentially resetting the pawns speeds.

With our call to AddGameSpecificInventory we simply check the player's team and then add a weapon based on what team the player is a part off.

Again there is no call to super; most games do not use this option – so we just define our own functionality.

```
//////////////////////////////////////////

defaultproperties
{
    DefaultEnemyRosterClass="xGame.xTeamRoster"
    MapListType="XInterface.MapListTeamDeathMatch"
    ScoreBoardType="XInterface.ScoreBoardTeamDeathMatch"
    bPlayersBalanceTeams=True
    bTeamGame=True
    bAllowTrans=true
    bBalanceTeams=True
    bScoreTeamKills=True
}
```

This sets the default properties of our game, the map list, score board type etc. Much of this can be inherited from previous classes, but here we have explicitly added to our game.

Finally, Altering the INT File:
We have to add different things to our INT file this time, although don't forget you can still add it to the same INT file.

The code we have to add before it all works is this:

```
[Public]

Object=(Class=Class,MetaClass=Engine.GameInfo,Name=CVGPackage.SpeedGa
me,Description="DM| SpeedGame
|xinterface.Tab_IADeathMatch|xinterface.MapListDeathMatch|false")
```

Conclusion
Team games are more complicated than single player's games, we can still have a lot of control over the game however there are just other considerations that need to be looked at before scripting the game.

Your Own Game
As part of the assessment - which incidentally if you have missed me saying so, is online and has been for over a week - you will be required to produce your own working gametype, as well as a mutator, to get into coding your own game type for the rest of this lesson have a go a scripting your won game type – it can be single or team based, think of what you want to do open up the hood and have a look.