Programming for Artists and Designers

Tutorial 3: Making Our Own Game Type

Aims:
This tutorial aims to get us programming our own game type. It is a more advanced tutorial than those attempted so far and again it builds upon what have learnt so far. If you are still struggling with what we have done so far look online and get up to speed.
It will also look more at the file structure of unreal and inheritance in the actor and object classes.

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

The Plan:
The idea this week is to script a basic game type that is called TopKiller. The idea is that whoever draws first blood will get the damage doubler and will keep it until death does them part. Whoever slays the player with the damage doubler will receive the damage doubler.

Start Coding:

Inheritance: UT2003 :: Actor >> Info >> GameInfo >> UnrealMPGameInfo >> DeathMatch >> xDeathMatch (Package: xGame)

```
/*******************************************************
    Game type called TopKiller – The best killer stays on top.
*******************************************************/

class TopKiller extends xDeathMatch;

// A reference to an actor in the Pawn class.
var pawn HasUDam;
```

- Here we are deriving TopKiller from the class xDeathMatch.
- Below this are the variables we are using – the variable is a reference to a player pawn.
- The variable "P" above is a reference to an actor in the Pawn class. Such a variable can refer to any actor that belongs to a subclass of Pawn. For example, P might refer to a Brute, or a Skaarj, or a Manta. It can be any kind of Pawn. However, P can never refer to a Trigger actor (because Trigger is not a subclass of Pawn).

- One example of where it's handy to have a variable referring to an actor is the Enemy variable in the Pawn class, which refers to the actor that the Pawn is trying to attack.
- When you have a variable that refers to an actor, you can access that actor's variables, and call its functions.

```
function Killed( Controller Killer, Controller Killed, Pawn
KilledPawn, class<DamageType> damageType )
{

      if(HasUDam==none)
      {
            Killer.Pawn.EnableUDamage(100);
            //which pawn has the doubler
            HasUDam = Killer.Pawn;
      }
      else if(Killed.pawn.HasUDamage() &&
!Killer.pawn.HasUDamage())
      {

            Killer.Pawn.EnableUDamage(100);
            HasUDam=Killer.Pawn;
      HasUDam.PlaySound(Sound'AnnouncerMain.HolyShit_F',,255);

      }
      else if(killer.pawm.HasUDamage())
      {
          if(Killed.pawn != Killer.pawn) // not a suicide
          {
            Killer.Pawn.EnableUDamage(100);
    HHasUDam.PlaySound(Sound'AnnouncerMain.FlackMonkey',,255);
           }
           else
           {
             HasUDam=none;
           }
      }

    super.Killed( Killer, Killed, KilledPawn, damageType );
}
```

- What we are doing initially is extending the Killed function with our own code and then calling the previous instances of the killed function.
- First a note about the super keyword: **Super**. With the Super keyword you can call functions in the same state of a superclass of this object – See activehelix US_SuperKeyword for a better break down of the super keyword.
- The function Killed is found in the base death match type, it deals with killers and killed. It is what calls the first blood, killing spree messages and sounds and deals with handing out adrenaline for kills. The function does quite a few important game things…open it up and have a look.

- The Deathmatch game itself doesn't have a hold on any of the actors directly, but they do get processed by xDeathmatch.
- Looking at the function signature - Killed(Controller Killer, Controller Killed, Pawn KilledPawn, class<DamageType> damageType) – it should hopefully be a little clearer that we are passing the player that was killed and the player who did the killing….notice the word controller there….description from the Controller class….

Controller, the base class of players or AI.
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 AIControFllers 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 occuring for the Pawn they are controlling. This gives the controller the opportunity to implement the behavior in response to this event, intercepting the event and superceding the Pawn's default behavior.

- What this means is that the controller has access to the player, the player is known as a pawn.
- Due to unreal scripts object oriented nature we just use, for this function killer.pawn e.g. Killer.Pawn.EnableUDamage(100); denables the damage doubler on the killer for 100 seconds.

The Code Above Broken Down

```
if(HasUDam==none)
{
      Killer.Pawn.EnableUDamage(100);
      //which pawn has the doubler
      HasUDam = Killer.Pawn;
}
```

- If no one has damage doubler yet, i.e first blood, then they get the doubler.
- We are setting the variable HasUDam to the Pawn who gto the first kill.

```
else if(Killed.pawn.HasUDamage() && !Killer.pawn.HasUDamage())
{
      Killer.Pawn.EnableUDamage(100);
      HasUDam=Killer.Pawn;
      HasUDam.PlaySound(Sound'AnnouncerMain.HolyShit_F',,255);
}
```

- Else If the killer doesn't have the damage doubler and the killed does…
- Then give damage doubler to him and set the killer as the reference for the HasUDam variable.
- PlaySound functions, well plays a sound effect and it look something like this - PlaySound(Sound Sound, optional ESoundSlot Slot, optional

float Volume, optional bool bNoOverride, optional float Radius, optional float Pitch, optional bool Attenuate) [native, final]

- It is found in the actor class.

```
else if(killer.pawm.HasUDamage())
{
     if(Killed.pawn != Killer.pawn) // not a suicide
     {
         Killer.Pawn.EnableUDamage(100);
    HHasUDam.PlaySound(Sound'AnnouncerMain.FlackMonkey',,255);
     }
     else
     {
        HasUDam=none;
     }
}
```

- This says that if the killer has UDamage..
- Then check to see if he killed himself..
- If he didn't give him some more time on his Damage Doubler and play a sound.
- Else he did commit suicide and HasUDam is set to none, so the next player to kill someone gets it.

```
defaultproperties
{
    bAllowTrans=true
}
```

This is the last line to add to our game type, this basically says that the game type allows translocators.

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:

```
Object=(Class=Class,MetaClass=Engine.GameInfo,Name=MyPackage.T
opKiller,Description="DM|Top Killer
|xinterface.Tab_IADeathMatch|xinterface.MapListDeathMatch|fals
e")
```

A Note on the Game Info and Level Info:
Inheritance: UT :: Object >> Actor >> Info (UT) >> GameInfo

A GameInfo subclass contains the rules for a gametype. This includes timelimit, fraglimit, scoring, difficulty, game speed, allowed playerpawn classes, default inventory assignment, and if appropriate, team assignment and general bot behavior.

This is assigned to a map in the Level Properties window, under LevelInfo → DefaultGameType.

The GameInfo for the current game can be retrieved from every actor in that game via Level.Game, but only on the server. GameInfo is not replicated to network clients; only GameReplicationInfo is.

*Note: It has a lot of functions!!!

Inheritance: UT2003 :: Actor >> Info >> ZoneInfo >> LevelInfo (Package: Engine)

LevelInfo holds:
information about the map (default gamemode, author, music, etc).
information about the zones in the map that don't have a ZoneInfo (UT).

UnrealEd will automatically place a LevelInfo when you create a new map. It's located at the origin (0,0,0), but its Advanced → bHiddenEd flag is set so you can't see it.

It's in fact the properties for an actor of the class LevelInfo, which added automatically but UnrealEd to a new map, but made invisible. AFAIK, the only property group that's relevant is LevelInfo.

The Audio section is also useful for adding music to a level

Have a Play:
By this I do not mean have a play on UT2K3, that isn't going to help you pass the module is it.
Alter the code, either for this game type to attempt to make it do something else.
Check out links such as:
http://udn.epicgames.com/Technical/WebHome
http://udn.epicgames.com/Technical/GameAndAIHandout
For code reference

One more thing to look at….TEXTURES:
I am mentioning this now as in the future you may need to load your own textures.
The UDN claims to support 5 texture formats
1) DirectX Texture Compression
DXTC is the native, compressed texture format used in DirectX 8. In many cases, DXTC reduces texture memory usage by 50% or more… Three DXTC formats are available
DXT1
DXT3
DXT5
Note: you'll need to download a converter for NVIDIA to create these
Importing Textures

2) Eight-bit Palettized (BMP's)
   Eight-bit Palettized with Alpha (BMP's)
3) 32-bit RGBA
4) Pcx's
5) Tga's,

Importing Textures
UT2003 is not quite as specific as its predecessor where all the images had to be square. UT2003 will support rectangular shaped textures. So long as the dimensions are *powers of 2*.
If you forget this then you may end up with errors.

Photoshop is ideal for altering such images.

You wish to try and make an image with the above specs, because at some point we will be loading images onto a HUD.
Don't worry if you don't as I will be supplying one.