Tutorial 9:PAR in Unreal

November 8, 2015

Before Beginning

This tutorial shows how to add PAR to the unreal game engine (4.9 at the time of this writing). This tutorial assumes you are familiar with the Unreal Development environment, and understand terms used in the environment such as *blueprints*. To get started with the unreal development environment, please refer to the wiki page. Assets used in this tutorial come from "*Animation Starter Pack*"

Setting up the Environment

To start this tutorial, create a blank project. This will display a floor and player control, and can be used to create the necessary classes and blueprints. Then, import the *Animation Starter Pack* into the project. After that, create a new c++ class (see below) which will allow PAR access to the underlying code.

Creating the Agent Blueprint

There are two necessary c++ class templates to get a basic PAR agent working in the environment, a game mode class and a character class. When creating a new c++ project, the unreal engine also creates several other important files. First, we will work on adding PAR (a third party library) into Unreal. This is done in a method mirroring the tutorial on adding third party libraries, found here. In Tutorial9.build.cs, add the following public function to the overall class, which essentially sets up the library paths:

```
    public bool LoadPAR(TargetInfo Target)
    {
        bool isLibrarySupported = false;

        if ((Target.Platform == UnrealTargetPlatform.Win64) ||
                (Target.Platform == UnrealTargetPlatform.Win32))
        {
            isLibrarySupported = true;

            string PlatformString = (Target.Platform == UnrealTargetPlatform.Win64) ? "" : "d";
            string PythonString = (Target.Platform == UnrealTargetPlatform.Win64) ? "" : "_d";
            string ConnPath = (Target.Platform == UnrealTargetPlatform.Win64) ? "opt" : "debug";
            string LibrariesPath = Path.Combine(PARPath, "libs");

            PublicAdditionalLibraries.Add(Path.Combine(LibrariesPath, "lwnet" + PlatformString + ".lib"));
            PublicAdditionalLibraries.Add(Path.Combine(LibrariesPath, "database" + PlatformString + ".lib"));
            PublicAdditionalLibraries.Add(Path.Combine(LibrariesPath, "agentProc" + PlatformString + ".lib"));
            PublicAdditionalLibraries.Add(Path.Combine(LibrariesPath, "Python27" + PythonString + ".lib"));
            string connector_path = Environment.GetEnvironmentVariable("CONNECTOR_ROOT");
            PublicAdditionalLibraries.Add(Path.Combine(connector_path,"lib", ConnPath, "mysqlcppconn.lib"));
        }

        if (isLibrarySupported)
        {
            // Include path
            //PublicIncludePaths.Add("C:\\Python27\\include");
            PublicIncludePaths.Add(Path.Combine(PARPath, "agentProc"));
            PublicIncludePaths.Add(Path.Combine(PARPath, "database"));
            PublicIncludePaths.Add(Path.Combine(PARPath, "lwnets"));

        }

        Definitions.Add(string.Format("WITH_PAR_BINDING={0}", isLibrarySupported ? 1 : 0));

        return isLibrarySupported;
    }
```

This function should be called from the class's constructor. Also note that we use a string called *PARPath*. This string is created from the following function, which should also be added to the class.

```
    private string PARPath
    {
        get { return Path.GetFullPath(Path.Combine(
                                        Path.GetDirectoryName(
                                                RulesCompiler.GetModuleFilename(this.GetType().Name)),
                                                        "../../../../../../../PAR/"));
                }
    }
```

Unreal also creates a default header class, with which we can add in the necessary headers for PAR. In tutorial9.cpp, we define several global variables to be used by the system. These are:

```
parTime *partime;
char *actionLocation = "Location of PAR Actions"; /*! <Using the unreal editor may require an absolute path*/
TArray<AHumanoid*> all_agents; /*! <This is the array that holds all of our PAR agents*/
TArray<UAnimMontage*> all_montages; /*! <This array holds all of the used montages connected to PAR*/
```

Game Mode Class

The game mode class primarily controls the roles of a game, and a primer can be found here. We add PAR code to two primary functions, *PreInitializeComponents* and *Tick*, as well as adding in three global variables, seen below.

```
extern parTime *partime;
extern Actionary *actionary;
extern TArray<AHumanoid*> all_agents;
extern TArray<UAnimMontage*> all_montages;
extern ActionTable actionTable; //Holds the mapping of actions to real world code
```

We use GameMode's constructor to find references to all of our animations, and store them in the array *all_montages*. Fill out the initialization function as:

```
extern parTime *partime;
extern Actionary *actionary;
extern TArray<AHumanoid*> all_agents;
extern TArray<UAnimMontage*> all_montages;
extern ActionTable actionTable; //Holds the mapping of actions to real world code
```

Next, we will need to fill out the function *PreInitializeComponents*, which is called before any characters are created. This will allow us to set up the actionary. Fill in the function with the following information:

```
        error = 0;
        static ConstructorHelpers::FObjectFinder<UAnimMontage> TestMontage(TEXT("AnimMontage'/Game/Actions/TestMontage'"));
        all_montages.Add(TestMontage.Object);
```

And the tick function, which runs on every frame.

```
        Super::Tick(DeltaSeconds);
        LWNetList::advance(&(this->error));
```

Character Class

This class is the base agent class that PAR's agentProc is connected to. When creating this from the parent class, four functions are automatically created (the Initalization function, BeginPlay, Tick, and SetupPlayerInputComponent). It is important to fill out BeginPlay and Tick. We also need to create a new function that allows us to play animations. For this tutorial, we assume the class is called AHumanoid (unreal adds A in front of the class automatically). First, we add in three variables to the character class, seen below.

```
private:
        AgentProc *par_agent; /*! <A pointer to the par agent
        class, and what all par information goes through */
        USkeletalMeshComponent *mesh; /*! <A pointer to the mesh.
        Useful as a shorthand for getting the animation*/
        int idle;/*! <The idle counter that we use to
        provide new animations*/
```

And the function name as a public function:

```
        /*! Adds an action to the current actors animation */
        void AddAction(UAnimMontage*);
```

Next, we fill in the BeginPlay function. This function initializes the human agent with a an agent proc. Fill in the BeginPlay function with the following code:

```
        Super::BeginPlay();
        par_agent = new AgentProc(std::string(TCHAR_TO_ANSI(*this->GetName())).c_str());
        par_agent->setCapability("Nod");
        this->mesh = this->GetMesh();
        all_agents.Emplace(this);
```

Next, we need to fill in the Tick Function. This runs on every frame, and is similar to the agent update function in Tutorial 4. Here, we run an idle check, and update the MetaObject with information on the character's position and orientation. Enter the following code into the Tick Function:

```
        Super::Tick( DeltaTime );
        //For now, we only care about the position and orientation of the character
        FVector trans = this->GetActorLocation();
        Vector<3> *vec = new Vector < 3 >();
        vec->v[0] = trans.X;
        vec->v[1] = trans.Y;
        vec->v[2] = trans.Z;
        this->par_agent->getObject()->setPosition(vec);
        trans = this->GetActorQuat().Euler();
        vec->v[0] = trans.X;
        vec->v[1] = trans.Y;
        vec->v[2] = trans.Z;
        this->par_agent->getObject()->setOrientation(vec);

        //This is our idle checker, also seen in Tutorial 4
        if (this->par_agent->emptyQueue() && !this->par_agent->activeAction()){
                idle++;
        }
        if (idle == 10){
                iPAR *par = new iPAR("Nod",this->par_agent->getName());
                par->setStartTime(partime->getCurrentTime());
                par->setFinished(false);
                this->par_agent->addAction(par);
        }
```

Finally, we fill out addAction to have our character play Animation Montages. A primer on Animation Montages can be found here. Add the following code to the class:

```
//!This function starts playing an animation montage on our character, and is useful for displaying animation
/*!
        \param montage The montage we wish to play on the agent.
*/
void AHumanoid::AddAction(UAnimMontage *montage){
        if (mesh != NULL && montage != NULL){
                mesh->AnimScriptInstance->Montage_Play(montage, 1.0f);
        }
}
```

Allowing Agents to Perform Animations

Animations can be played on a Character class if that character has an associated Animation Blueprint. While it is most likely possible to forgo this step, creating a simple animation blueprint is the easiest method playing animations on a character. Specifically, we will be playing an animation montage (like fire) on the full body of our character.

First, open the editor and create an animation blueprint. Make sure to give the animation a **Full Body Slot**, which will cause the action to run over all the joints. Next, create a new animation blueprint, and navigate to the anim graph tab, as seen in Figure 1. Add a slot node that connects to the final animation of the character. Be sure to set the slot node to **Full Body** as well. The resulting graph should look like Figure 1.

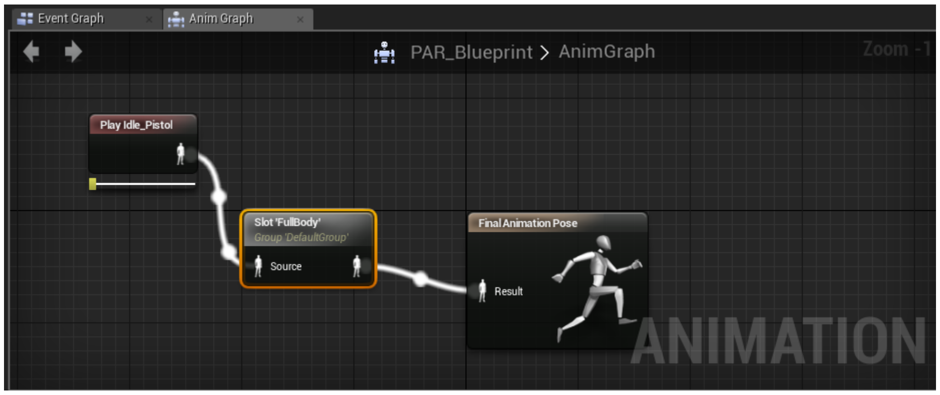

Figure 1: The animation graph blueprint

Adding a PAR agent to the Scene

After creating the animation blueprint and sample animation, all the components should be in place to add a character to the scene. Navigate to the directory in the editor that the Character blueprint resides in, and drag one into the scene. Provide a name for the character (such as Male_0) and attach a skeletal mesh to the character. Next, attach the animation blueprint to the character, so that the character can perform animations.

Finally, before running the program, navigate to the settings, and make sure that PAR's game mode is selected as the current game mode. Failure to do so will result in an error.