Développement d'un boss : Argaloth

Galathil

Important : Ce tutoriel a été rédigé par Nobodie pour le site wow-emu.fr. Je le recopie ici à l'identique, Il se peut que certaines informations soient obsolètes à ce jour. Bonne lecture !

Développement d'un boss : Argaloth

Bonjour,

aujourd'hui nous allons voir ensemble, en pratique, comment scripter un boss. Pour notre exemple, nous allons choisir un boss dont le script est simple : Argaloth. Il s'agit du premier boss du Bastion de Baradin, premier donjon de l'extension Cataclysm. De plus, celui-ci a déjà été codé quasiment parfaitement par Trinity et nous pouvons donc nous en servir comme base d'apprentissage. Entrons dès à présent dans le vif du sujet.

Le code complet de Trinity (attention, quelques modifications sont apportées par mes soins dans ce tutoriel) est visible à **cette adresse**.

1) La recherche d'informations

Le développement de boss ressemble à ce qu'on appelle de la programmation événementielle. On définit un certain nombre d'événements qui s'exécutent, et à chacun de ces événements on associe une fonction qui contient le code chargée de gérer l'événement. La priorité, lorsque l'on souhaite développer un boss, c'est donc de trouver le maximum d'informations à son sujet.

Pour ce faire, de nombreux sites sont à votre disposition, tels que wowhead, judgehype ou même youtube pour visualiser des vidéos du combat sur officiel. Pour ma part, j'utilise principalement wowhead et youtube.

Pour ce qui est de wowhead, n'hésitez pas à regarder les pages dans plusieurs langues. Si on compare la version **Argaloth** et **Argaloth** d'argaloth, on constate que la version française est beaucoup plus riche et contient une description de l'ensemble de la stratégie à appliquer contre ce boss. C'est donc la page française qui va nous servir de référence.

Après lecture de ces paragraphes et visualisation de vidéos sur Youtube, on parvient à en extraire 4 événements majeurs :

- Lancement de **Attaque météorique** environ 15 secondes après le début du combat, puis toutes les 15 à 20 secondes.
- Lancement de **Ténèbres consumantes** environ 20 secondes après le début du combat, puis toutes les 20 à 25 secondes.
- Lancement de **Tempête de feu gangrenée** lorsque la vie du boss atteint 66% puis 33% de vie.
- Lancement de **Berserk** au bout de 5 minutes de combat.

Nous avons identifié tous nos événements, il est temps de créer le fichier cpp qui va contenir le script du boss !

2) Ou créer un script pour le boss et comment le compiler

L'ensemble des fichiers script du serveur se situent dans le sous-dossier script situé dans l'arborescence suivante : `src/server/scripts/` .

Les scripts sont majoritairement rangés dans des sous-dossiers de la forme `Continent/InstanceName/bossName.cpp` . N'hésitez pas à garder cette habitude pour maintenir un dossier de scripts propre.

Les dossiers sont bien constitués et vous trouverez donc (ou créerez) le fichier sous le dossier `EasternKingdoms/BaradinHold` et aura pour nom `boss_pit_lord_argaloth.cpp` . On aurait également pu l'appeler boss_argaloth, évidemment, mais puisque Trinity l'a appelé comme ça, autant le garder sous la même forme □

La chose suivante à effectuer est d'integrer ce nouveau fichier aux fichiers de configuration que l'on appelle les fichiers `CMakeLists.txt` . Pour cela, ouvrez le fichier dans l'arborescence suivante : `src/server/scripts/EasternKingdoms/CMakeLists.txt` et ajoutez (si elle n'existe pas encore) la ligne suivante :

```
EasternKingdoms/BaradinHold/boss_pit_lord_argaloth.cpp
```

En faisant cela, votre fichier sera désormais integré à votre projet et sera compilé avec vos autres fichiers.

Maintenant que le fichier est créé, il va nous falloir le remplir avec une classe qui va nous permettre de scripter le boss. Voici le code général que nous allons y intégrer :

```
#include "ScriptMgr.h"
#include "ScriptedCreature.h"

class boss_pit_lord_argaloth : public CreatureScript
{
public:
boss_pit_lord_argaloth() : CreatureScript("boss_pit_lord_argaloth") { }

struct boss_pit_lord_argalothAI : public ScriptedAI
{
boss_pit_lord_argalothAI(Creature* creature) : ScriptedAI(creature) { }

};

CreatureAI* GetAI(Creature* creature) const override
{
return GetBaradinHoldAI<boss_pit_lord_argalothAI>(creature);
}
};

void AddSC_boss_pit_lord_argaloth()
{
new boss_pit_lord_argaloth();
}
```

ce script minimal contient 3 choses primordiales :

- Premièrement, la classe **boss_pit_lord_argaloth** hérite de CreatureScript (définie dans `src/server/game/Scripting/ScriptMgr.h`). La définition du constructeur de la classe est importante car il appelle le constructeur de CreatureScript qui prend en paramètre une chaîne de caractère (ici "**boss_pit_lord_argaloth**" qui sera utilisé plus tard dans la base de données pour relier le creature_template d'argaloth à ce script). C'est donc cette ligne qui vous permet de définir le "nom" du script.
- Deuxièmement, cette classe possède une structure **boss_pit_lord_argalothAI** qui hérite de ScriptedAI (AI signifiant **A**rtificial **I**ntelligency). C'est dans cette structure que nous coderons la totalité de notre boss par la suite (attention donc à bien écrire le code futur à l'intérieur des accolades de cette structure). Il y a également une fonction GetAI qui est une fonction surchargée (comme on le voit avec la constante override à la fin de son prototype) qui sert à relier l'Intelligence Artificielle (définie dans la structure) à notre classe **boss_pit_lord_argaloth**. Il est important de garder un prototype identique à celui présenté ici, sinon l'appel à la fonction ne se fera pas et votre créature ne fera absolument rien.
- Enfin, il y a une ultime fonction définie tout en bas du script qui va permettre de relier notre CreatureScript aux fonctions de chargement des scripts (que nous allons voir juste après). Par convention, toutes ces fonctions commencent par le préfixe AddSC (pour Add Script, en anglais) et crééent (via l'opérateur new une nouvelle occurence de notre script **boss_pit_lord_argaloth**.

Certains auront peut-être remarqué que notre structure boss_pit_lord_argalothAI hérite de ScriptedAI tandis que le script officiel de Trinity hérite de BossAI. Pour des raisons de simplification de ce tutoriel, j'ai choisi de ne pas utiliser BossAI car elle nécessite un script d'instance qui ne rentre pas dans le cadre de ce tutoriel. Néanmoins, sachez simplement qu'un script d'instance permet de sauvegarder l'état de l'instance (boss tués, principalement) afin de pouvoir interagir avec elle et effectuer des modifications sur l'instance (par exemple gérer l'ouverture de portes menant à la suite de l'instance).

Si vous avez eu du mal à comprendre certains des termes utilisés juste au dessus (héritage, surcharge, constructeur ...) je vous invite grandement à stopper la lecture de ce tutoriel et à ~~partir élever des chèvres en Ariège~~ (re)lire un cours sur la programmation orientée objet avec C++ **ICI** !. Vous ne pourrez pas suivre le reste du tutorial sans cela.

Si vous compilez votre projet dans l'état actuel des choses, votre script ne fera rien car non seulement il ne possède encore aucun code

spécifique au boss, mais aussi et surtout parce que aucune système interne au core n'appelle la fonction AddSC_boss_pit_lord_argaloth() que nous avons défini juste avant. Pour ce faire, nous allons ouvrir le fichier suivant dans l'arborescence `src/server/game/Scripting/ScriptLoader.cpp` et ajoutez deux lignes qui feront le lien avec notre script :

Premièrement la définition de notre fonction :

```
void AddSC_boss_pit_lord_argaloth();
```

Puis l'appel à la fonction dans la fonction **void AddEasternKingdomsScripts():**

```
AddSC_boss_pit_lord_argaloth();
```

Je vous laisse prendre exemple sur le fichier pour trouver ou écrire ces deux lignes

Voilà qui est fait, l'ultime étape reste à modifier la DB pour associer notre creature_template d'Argaloth (entry #47120) au ScriptName défini un peu plus tôt : **boss_pit_lord_argaloth**.

Une fois que tout ceci est fait, notre script est prêt à coder !

3) Définition des énumérations nécessaires à notre boss

Par convention, l'ensemble des scripts sont écrits en langue anglaise. Aussi je ne saurais que trop vous inviter à vous adapter à la langue de Shakespeare

Nous l'avons dit tout à l'heure, il y a en tout 4 spells à lancer. Nous allons donc commencer simplement en définissant une énumération contenant l'ensemble de ces sorts (afin d'avoir un script bien lisible par la suite). En dehors de toute fonction/classe, il suffit d'écrire la chose suivante :

```
enum Spells
{
SPELL_METEOR_SLASH = 88942,
SPELL_CONSUMING_DARKNESS = 88954,
SPELL_FEL_FIRESTORM = 88972,
SPELL_BERSERK = 47008
};
```

Les ID des sorts sont disponibles via wowhead.

Nous allons également créer une seconde énumération correspondant à nos événements. Comme on l'a dit, l'événement déclenchant l'envoi du sort **Tempête de feu gangrenée** ne sera pas fait. Il y a donc seulement 3 événements à définir :

```
enum Events
{
EVENT_METEOR_SLASH = 1,
EVENT_CONSUMING_DARKNESS = 2,
EVENT_BERSERK = 3
};
```

Attention, si la valeur numérique définissant un événement importe peu, il est par contre **primordial** que le premier événement défini commence avec l'ID 1 et non 0. Sinon, celui-ci ne sera pas exécuté ensuite lorsqu'on arrivera à la gestion des événements.

voilà qui est tout pour la définition des énumérations nécessaires à notre boss

4) Surcharge des fonctions du script

Nous allons reparler ici des fonctions surchargées. En effet, l'ensemble du système de script repose sur cette notion de la programmation orientée objet. L'intelligence artificielle de notre boss hérite d'une structure définie par le core. Cette structure possède différentes fonctions appelées à des moments clé de l'exécution d'un boss. Par exemple, la fonction **EnterCombat** est appelée lorsque le groupe de joueurs entre

en combat avec le boss. la fonction **DamageTaken** est appelée ... chaque fois que le boss prend des dégats. Et la fonction **UpdateAI** est appelée à chaque tick d'horloge du serveur. Ces trois fonctions nous seront d'ailleurs nécessaires pour coder notre boss. Pour rappel, ces fonctions sont à définir à l'intérieur de notre structure héritée de ScriptedAI. Ne vous trompez pas !

Faites très attention au prototype de votre fonction. Si celle-ci ne correspond pas exactement au prototype de la fonction mère, c'est la fonction mère qui sera appelée en lieu et place de la votre, et votre boss ne fera donc que la moitié des actions que vous lui auriez programmé. Cette erreur est souvent commise par mégarde et peut ensuite faire l'objet de nombreuses heures de recherches.

a) Surcharge de EnterCombat

Nous allons donc commencer par EnterCombat, qui est appelé dès lors que le combat avec le boss commence. Voici le code de la fonction sur Trinity :

```
void EnterCombat(Unit* /*who*/) override
{
_EnterCombat();
//instance->SendEncounterUnit(ENCOUNTER_FRAME_ENGAGE, me);
events.ScheduleEvent(EVENT_METEOR_SLASH, urand(10 * IN_MILLISECONDS, 20 * IN_MILLISECONDS));
events.ScheduleEvent(EVENT_CONSUMING_DARKNESS, urand(20 * IN_MILLISECONDS, 25 * IN_MILLISECONDS));
events.ScheduleEvent(EVENT_BERSERK, 5 * MINUTE * IN_MILLISECONDS);
}
```

On peut voir qu'elle fait 5 lignes que nous allons développer juste après :

- **_EnterCombat();** : Il s'agit de l'appel à une sous fonction EnterCombat définie dans la classe mère. Elle permet de ne pas avoir à initialiser un certain nombre de choses dans notre code.
- **instance->SendEncounterUnit(ENCOUNTER_FRAME_ENGAGE, me)**; : Cette ligne (que j'ai volontairement commentée) sert à interagir avec notre éventuel script d'instance afin de pouvoir sauvegarder son état d'avancement.
- **events.ScheduleEvent(EVENT_METEOR_SLASH, urand(10 * IN_MILLISECONDS, 20 * IN_MILLISECONDS));** : Cette ligne nous permet de créer un premier événement à partir de notre énumération. On lui attribue un temps d'exécution entre 10 et 20 secondes.
- **events.ScheduleEvent(EVENT_CONSUMING_DARKNESS, urand(20 * IN_MILLISECONDS, 25 * IN_MILLISECONDS));** : Cette ligne nous permet de créer un second événement à partir de notre énumération. On lui attribue un temps d'exécution aléatoire entre 20 et 25 secondes.
- **events.ScheduleEvent(EVENT_BERSERK, 5 * MINUTE * IN_MILLISECONDS);** : Cette ligne nous permet de créer un troisième événement à partir de notre énumération. On lui attribue un temps d'exécution de 5 minutes.

Il faut bien saisir que nous ne faison ici qu'enregistrer les trois événements. Il seront traités par la suite.

b) Surcharge de EnterEvadeMode

Voici le code source de cette fonction :

```
void EnterEvadeMode() override
{
me->GetMotionMaster()->MoveTargetedHome();
//instance->SendEncounterUnit(ENCOUNTER_FRAME_DISENGAGE, me);
_DespawnAtEvade();
}
```

Il n'y a pas grand chose à expliquer, la première ligne génère un mouvement de la créature (identifiée par le pointeur interne me) vers sa position initiale. La seconde ligne est à nouveau commentée car inutile dans ce tutoriel, et la troisième fait appel à la fonction interne gérant le despawn de la créature.

c) Surcharge de JustDied

```
void JustDied(Unit* /*killer*/) override
{
_JustDied();
```

```
//instance->SendEncounterUnit(ENCOUNTER_FRAME_DISENGAGE, me);
}
```

Rien de particulier à expliquer ici.

d) Surcharge de UpdateAI

Ici vient la principale fonction, et la plus intéressante, dont voici le code :

```
void UpdateAI(uint32 diff) override
{
if (!UpdateVictim())
return;

events.Update(diff);

if (me->HasUnitState(UNIT_STATE_CASTING))
return;

while (uint32 eventId = events.ExecuteEvent())
{
switch (eventId)
{
case EVENT_METEOR_SLASH:
DoCastAOE(SPELL_METEOR_SLASH);
events.ScheduleEvent(EVENT_METEOR_SLASH, urand(15 * IN_MILLISECONDS, 20 * IN_MILLISECONDS));
break;
case EVENT_CONSUMING_DARKNESS:
DoCastAOE(SPELL_CONSUMING_DARKNESS, true);
events.ScheduleEvent(EVENT_CONSUMING_DARKNESS, urand(20 * IN_MILLISECONDS, 25 * IN_MILLISECONDS));
break;
case EVENT_BERSERK:
DoCast(me, SPELL_BERSERK, true);
break;
default:
break;
}
}

DoMeleeAttackIfReady();
}
```

Cette fonction est divisée en plusieurs parties que nous allons voir dès à présent :

- Premièrement, on vérifie qu'on possède bien une victime potentielle. S'il n'y en a aucune (et que donc tout le monde est mort) il n'y a plus aucune raison de faire s'exécuter le boss. On quitte donc via return.
- Ensuite, on met à jour les événements via la procédure Update. Celle-ci prend en paramaètre la variable diff qui est en fait le temps (en millisecondes) depuis le précédent appel à UpdateAI. Cella va donc mettre à jour les timers de tous les événements définis via la fonction EnterCombat vue précédemment.
- Puis, si le boss est déjà en train de lancer un sort, il faut le laisser poursuivre son cast. Nous quittons donc notre boucle.
- Enfin arrive une boucle while qui peut se traduire littéralement par "Tant qu'il reste des événements à exécuter". C'est à cause de ce while que nos événements auparavant définis dans l'énumération devaient être différents de 0, car sinon cette condition serait considérée comme fausse et l'on sortirait alors de la boucle sans jamais entrer dedans. Chacun des événements possède alors sont propre code. On peut voir qu'il y a en général deux lignes :
    - Le lancement du spell correspondant (lui aussi indiqué précédemment via une énumération). Ici ce sont des AoE, mais il pourrait y avoir d'autres systèmes pour récupérer une cible aléatoire ou le tank du groupe, par exemple.
    - L'appel à nouveau au ScheduleEvent qui permet de redéfinir le prochain temps permettant de relancer le sort ensuite.
      Dals le cas de Berserk, évidemment, il n'y a pas de ScheduleEvent du fait que si le groupe ne parvient pas à tuer le boss en moins

de 5 minutes, celui-ci entre en mode Berserk jusqu'à la fin du combat.

- Finalement, on appelle la fonction **DoMeleeAttackIfReady()** qui est en général la dernière appelée dans UpdateAI et qui permet au boss de tapper au corps à corps si son état le lui permet (c'est-à-dire s'il n'est pas en train de caster un sort, s'il n'est pas entravé, etc).

5) Pour aller plus loin

Comme nous l'avons vu ensemble, il reste un sort qui n'a pas été lancé. Il s'agit de **Tempête de feu gangrenée**. Pour cela, il vous faut surcharger la fonction DamageTaken afin d'envoyer ce sort **exactement** lorsque le boss passe sous la barre des 66% puis des 33%.

Le code proposé par trinity est faux, la condition est vraie dès lors que le boss passe sous la barre des 66% de vie et enverra dès lors la tempête à chaque appel à DamageTaken. A vous de changer son comportement en integrant par exemple un booléen permettant de connaitre l'ancien état du boss.

Enfin, nous avons ici traité l'aspect purement technique du boss, et ne nous sommes donc pas penchés sur le script des sorts en eux-mêmes. Il est probable que ceux-ci puissent nécessiter des modifications afin de correspondre au système officiel. Nous ne traiterons pas dans ce tutoriel de comment scripter des sorts, mais cela fera peut-être l'objet d'un futur tutoriel. Libre à vous également de vous renseigner de votre côté pour savoir comment faire

6) Conclusion

Voilà que se termine ce tutoriel. Après correction des erreurs de compilation, votre script devrait pouvoir s'exécuter et vous aurez alors devant vous un boss Argaloth prêt à en découdre !

Gardez à l'esprit que ce boss est parmi les plus simples de Wow et que, là ou ce script fait moins d'une centaine de lignes, d'autres en font près de 6000 Préparez-vous donc à souffrir, mais n'oubliez pas pour autant que tout est réalisable et que seule votre volonté vous permettra d'arriver à vos fins. Alors n'abandonnez pas à la moindre difficulté et poursuivez vos efforts !

~~Je reste disponible pour toutes questions/améliorations proposées.~~