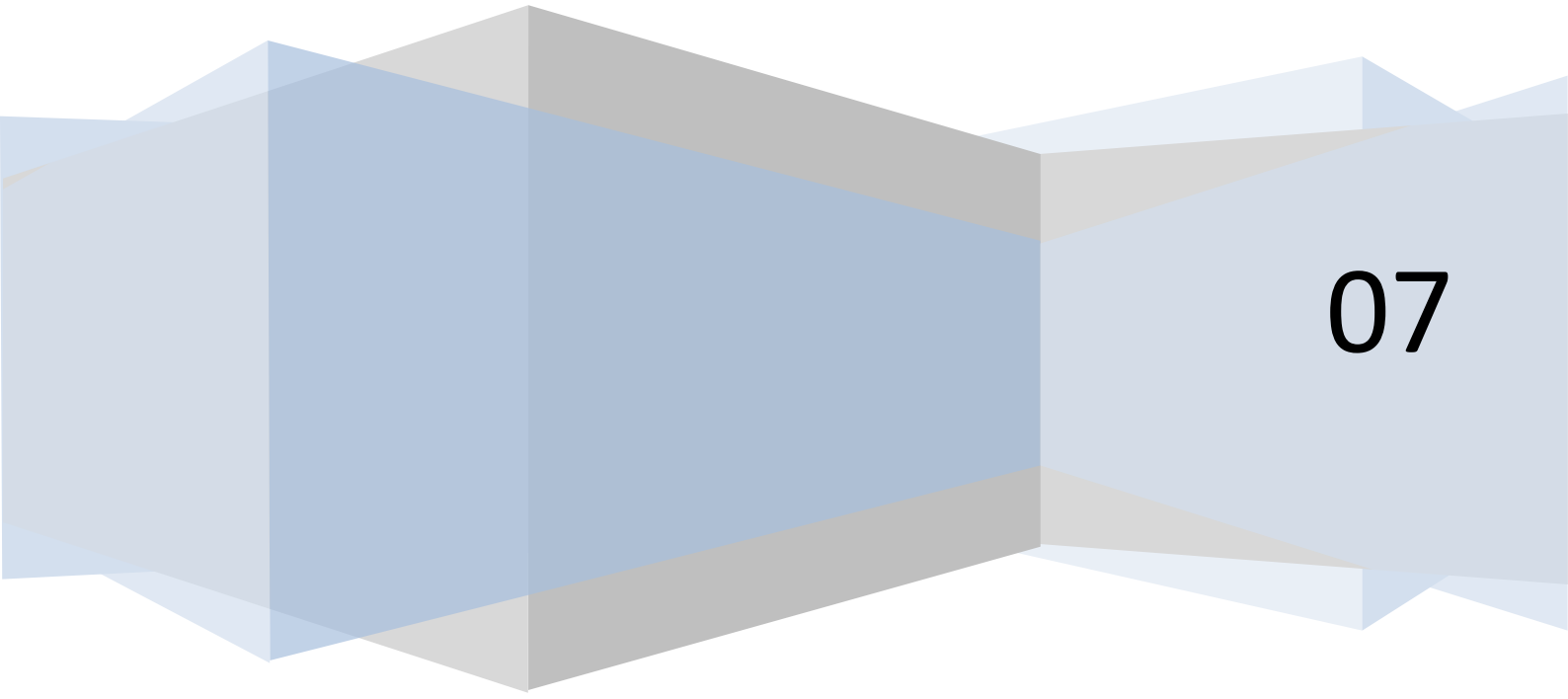
Corona Bytes .NET

CCI-Tutorial

How to create a complex Character

Raven



07

CCI Tutorial: How to create a complex character?

Evolution Class eXtensions RC2

Corona-Bytes.NET

Introduction:

How to create a complex character? I guess this is the kind of questions a lot of people have been asking lately. I will try to light up the way for you guys. However this is not a Tutorial for N00bs, you have to use your brain and you should at least have some clue about coding, about variables or other basic coding elements.

What we will be trying here is to create a complex character. For this reason I choose Son Goku and I will use basic ECX resources for this Tutorial so that you guys won't have to get any additional files.

Character Specs:

First we will think about the character specs, which is in my opinion the most important part. You should think about the specs before doing anything else. It is very important to keep a character in balance to the other characters in order to create a decent fair addition.

Let's start:

Now it's time to fire up the CCI, as we will do the most basic config settings using the CCI.Once you have the CCI open click on the star icon and the CXW (Class eXtension Wizard) will pop up.

1. Chose a name: Goku
2. Type in your version: 1.0
3. Author: That's you
4. Description: The Bio of your Character.

Basic Elements:

Now it's time that we choose what we want to implement, since I this is supposed to be a complex character.

Please check the following boxes

- Sound
- Weapon
- Charge
- Descend Class

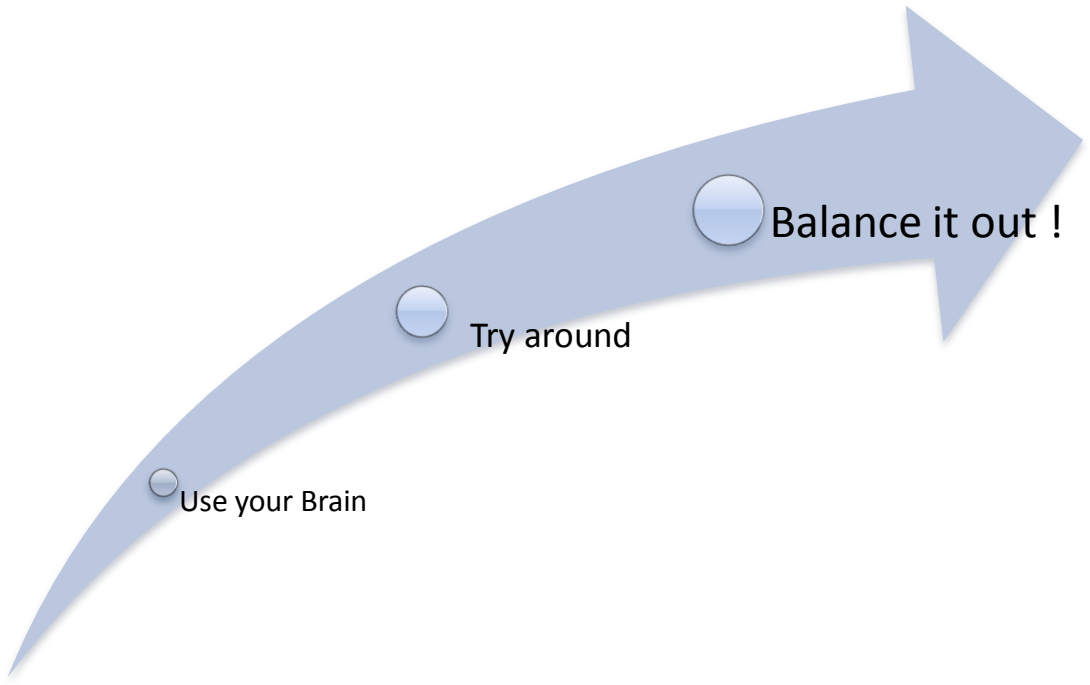
Since we want to modify the sound, the weapons, the charge and we want to declare it a Descend Class since saiyans can descend.

Now use the Support Level Bar

- Raise to bar to: "Support 2 Levels"

Configuring Levels:

This is the part that most of you will understand best I guess, however I will try to go a little bit more into depth. It is very important that you think about the values all together than just giving it 5sec than bust in a random value that you come up with.



On the next page you will find a page of suggested values for our Tutorial Class.

After configuring your character press Create and your base files will be created in a folder names after your character.

Level 0	Level 1

• Name: Saiyan
• Model: ecx.goku-ts
• Ascend Time: 1.0
• Perfect Ascend Time: 1.0
• Power Level: 850000
• Perfect Power Level: 1
• Power Level Multiplier: 1.0
• Speed: 400
• Health: 100
 	
• Teleport Range: 200
• Teleport Ki Cost: 100
• Ki Rate: 1.0
• Booster: 2.0
 	
• ATK: 100%
• DEF 0%
 	
• Attacks:
  • Melee
  • KiBlast
  • Genericbeam
  • Kamehameha
	• Name: Super Saiyan
	• Model: ecx.goku-ts.ssj
	• Ascend Time: 10.0
	• Perfect Ascend Time: 1.5
	• Power Level: 2000000
	• Perfect Power Level: 2500000
	• Power Level Multiplier: 2.0
	• Speed: 450
	• Health: 130
	
	• Teleport Range: 280
	• Teleport Ki Cost: 90
	• Ki Rate: 3.0
	• Booster: 3.0
	
	• ATK: 110%
	• DEF 10%
	
	• Attacks:
	  • Melee
	  • KiBlast
	  • Genericbeam
	  • Kamehameha

Manual Configuration:

As some of your might have noticed I didn't change the powerup and aura color in the CCI, why? Because it's just as easy manually and I find it easier by code since you are able to the fine the RGB values which I'm more comfortable with.

Here is a list of what we will be doing:

.ClassExtesion.core

- Change the Powerup Color
- Define Transformation Effect Controls
- Precache nessesary files
- Code erverything that is important for the character core

FX.core

- Create the FX functions you need for your Transformation
- Code Anything that is neccesary for Effects

MOD.Sound.core

- Define pain sounds
- Define weapon Sounds
- Define custom special sounds

MOD.Charge.core

- Define the Charge Sprite for default weapon replacements

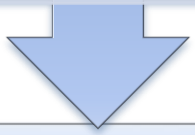
MOD.Weapon.core

- Precache Weapon Sprites
- Modify Weapon Icons
- Modify Weapon Charges
- Modify Weapon Explosions
- Modify Weapon Damage
- Substitude Weapon Sprites
- Modify Weapon Sizes
- Modify Weapon Names

Change the Powerup Color:

Find Line 43[...]

```
addClassEffect( "models/evolution/Auras/shape_01.mdl", Float:{ 255.0, 255.0, 255.0, 50.0 }, 0, kRenderTransAdd,{ 255, 255, 255 }, { 0, 0, 0, 0 }, Float:{ 0.0, 0.0, 0.0, 0.0 } );
```



Change to

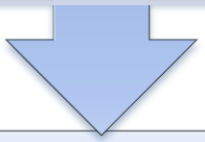
```
addClassEffect( "models/evolution/Auras/shape_01.mdl", Float:{ 70.0, 70.0, 70.0, 50.0 }, 0, kRenderTransAdd,{ 70, 70, 70 }, { 0, 0, 0, 0 }, Float:{ 0.0, 0.0, 0.0, 0.0 } );
```

What we did here is pretty simple we changed the values that were responsible for the Aura Color and the PowerUp Color. Why? Because it looks better that way, let's keep the colors smooth and a little transparent.

Next step ahead is doing exactly the same thing on the Super Saiyan (Level 1), but his time let's change it into a bright super-saiyan-yellow.

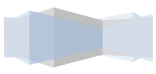
Find Line 49[...]

```
addClassEffect( "models/evolution/Auras/shape_01.mdl", Float:{ 255.0, 255.0, 255.0, 50.0 }, 0, kRenderTransAdd,{ 255, 255, 255 }, { 0, 0, 0, 0 }, Float:{ 0.0, 0.0, 0.0, 0.0 } );
```



Change to

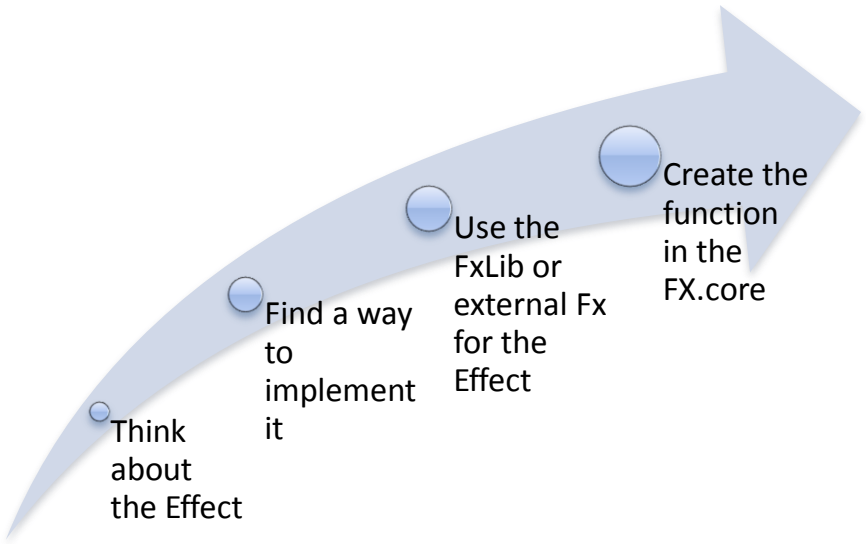
```
addClassEffect( "models/evolution/Auras/shape_01.mdl", Float:{ 160.0, 150.0, 20.0, 50.0 }, 0, kRenderTransAdd,{ 160, 150, 70}, { 0, 0, 0, 0 }, Float:{ 0.0, 0.0, 0.0, 0.0 } );
```

That's it for now.

Define Transformation Effect Controls:

Now we start on the part that probably most of you are interested in Transformation Effects. Doing Transformation Effects we'll be following a simple process.



Now what you do is, you scroll down in your .ClassExtestion.core and locate the "@ClassInitTransformation" function which will be probably at line 81. Watching the basic concept of this function most of you will already have a clue how it will work, besides let's not forget about two other very important functions. @ClassFinishTransformation and @ClassCancelTransformation, which are responsible for, who would have guessed it, the calls right after the transformation finishes and when the Transformation is canceled.

Let's summarize that:

@ClassInitTransformation

- This is the place where we will start our tasks and call anything we want that is related to the Transformation Effect.

@ClassFinishTransformation

- Here will be all the calls that have to be made when the Transformation finished. E.g. cancel the Transformation tasks etc.

@ClassCancelTransformation

- This is what we call when a Transformation is canceled. Here you have to cancel e.g. the Transformation Effects and related tasks.

What we are going to add as an FX Effect is pretty simple, just to keep it simple. For the long transformation we will add an avatar that will mimic the transformation animation for us. Then we will add a sound using the emit_sound() function we will add a model aura and a shockwave at the end. Our perfect trans will be even shorter and we will leave it by using a single shockwave at the end as Effect.

First create a one-dimensional array. Just like this:

```
@ClassInitiateTransformation
new CORE[ 1 ];
CORE[ 0 ] = Client;
```

This is very important, because later on our tasks need to transfer the client's ID.

Now let's do the perfect transformation fist. Find the first case of the perfect transformation level switch, which should be around line 88.

into case 1: add (line 88)

```
emit_sound( Client, CHAN_ITEM, "ecx.goku-ts/tp_ssj.wav", 1.0, ATTN_NORM, 0, PITCH_NORM );
AddFx( Client, "fxAvatar", "create", 102, 1.0 );
set_task( 1.3, "fxScream", uniqueTaskID( Client,100 ), CORE, 1, "a",1 );
```

into second case 1: add (line 92)

```
emit_sound( Client, CHAN_ITEM, "ecx.goku-ts/t_ssj.wav", 1.0, ATTN_NORM, 0, PITCH_NORM );
AddFx( Client, "fxAvatar", "create", 101, 0.46041 );
set_task( 0.1, "fxAura", uniqueTaskID( Client,100 ), CORE, 1, "a",1 );
```

Now if you paid attention you will notice that we didn't precache any files. Of course we need to do that. So jump into the "PluginPreCache ()" function and add the following lines:

PluginPreCache ()

```
precache_sound( "ecx.goku-ts/t_ssj.wav" );
precache_sound( "ecx.goku-ts/tp_ssj.wav" );
precache_sound( "ecx.goku-ts/t_scream.wav" );
precache_model( "models/evolution/Auras/shape_05.mdl" );
precache_model( "models/evolution/Auras/default.mdl" );
```

Precaching your files is very important, if you fail to do so you will crash the game on loading.

Now let us continue on the two last functions. Please add what follows.

```
@ClassFinishTransformation
RemFx( Client, "fxAvatar", 0);
AddFx( Client, "fxBlow" );
FX_off(Client);
remove_task( uniqueTaskID( Client, 100 ) );
```

What we did here is to remove the Avatar which made our animation or whatever we want it to do while we were transforming. We added a shockwave on transformation's end. Calling a function that will shut down all running Fx Effects. And we removed the transformation FX tasks we created earlier and prevent so any more things to happen when the transformation finished.

```
@ClassCancelTransformation
RemFx( Client, "fxAvatar", 0);
FX_off(Client);
remove_task( uniqueTaskID( Client, 100 ) );
```


Basically we did the same thing this time around, with the slight different that we don't call for a shockwave, because we don't what that on a transformation cancelation.

FX.core

Now let's move to another File. This is the file you've been waiting for. In this file we will define the functions that call the actually Effects from the FxLib. Since we want to keep this tutorial as short as possible we just have to create 4 functions:

```
public fxBlow( Core[] )
AddFx( Core[0], "fxBlow" );
return 1;
```

This is the Shockwave.

```
public fxAura( Core[] )
AddFx( Core[0], "fxModelEntity", "models/evolution/Auras/shape_05.mdl", 0, 0, 50.0, 0, 0, 0, 1.0, 0 3, 0, 4 );
AddFx( Core[0], "fxModelEntity", "models/evolution/Auras/default.mdl", 0, 0, 10.0, 0, 0, 0, 1.0, 0.3, 0, 4 );
return 1;
```

This is a double model aura, in other words two model entities.

```
public fxScream( Core[] )
AddFx( Core[0], "fxWorldLight", 40, 60, 55, 10, 2.1, 0.5);
emit_sound( Core[0], CHAN_ITEM, "ecx.goku-ts/t_scream.wav", 1.0, ATTN_NORM, 0, PITCH_NORM );
return 1;
```

This is a world light effect combines with a scream of our character.

```
public FX_off( Client )
RemFx( Client, "fxModelEntity", 0);
RemFx( Client, "fxModelEntity", 0);
return 1;
```

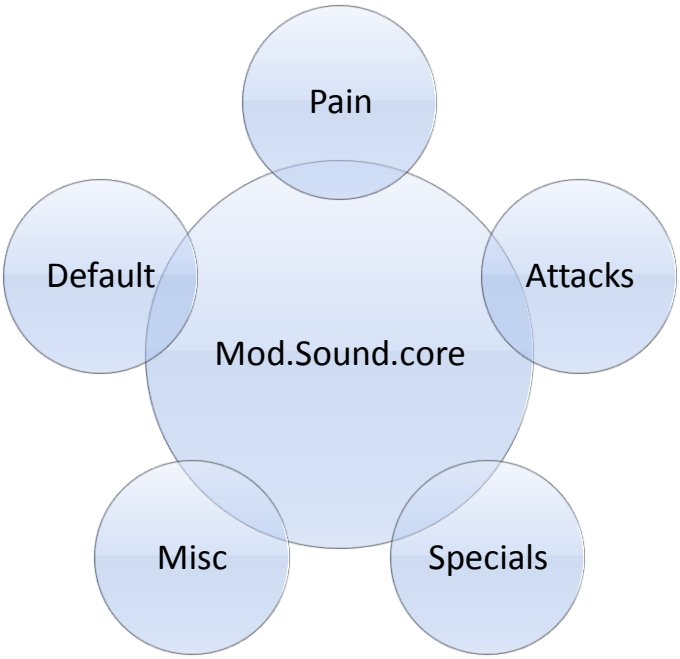
This is the most important function, because it deleted constant effects again, since we create two model entities with fxAura we need to remove them again after we used them.

FxLib Commands will be explained elsewhere as soon as I find the time.

MOD.Sound.core

Now that we finished our FX stuff it's time to move to the sounds. The MOD.Sound file is basically just a three dimensional array that the ECX will handle. Your task is now filling that array in a specific way so ECX can use it.

First off MOD.Sound sound files do not have to be precached elsewhere ECX will do that for you.



MOD.Sound will handle any incoming sound file by ESF and give you the chance to bypass it with one of your own sound files e.g. pain sounds or any other sound for that matter.

```
new MOD_SOUND_CORE[][][] =
{
        // E.g. Pain sound
        {
                "*/pain1.wav", // e.g. catches every pain1.wav
                "ecx.goku-ts/p_pain1.wav", // On level 0 substitutes original with [...]
                "ecx.goku-ts/p_pain1.wav"  // On level 1 substitutes original with [...]
        }
}
```

I understand this might not come so easy on you, but it's very simple, really. And if you have some coding experience it will come to you even easier, since you probably already know how to handle an array. However let's take a look at our finished MOD.Sound.core

```
new MOD_SOUND_CORE[][][] =
{
	// << WEAPON >>
	{
		"*/kamehame.wav",
		"ecx.goku-ts/wc_kamehame.wav",
		"ecx.goku-ts/wc_kamehame.wav"
	},
	{
		"*/ha.wav",
		"ecx.goku-ts/w_kamehame.wav",
		"ecx.goku-ts/w_kamehame.wav"
	},

	// << Pain >>
	{
		"*/pain1.wav",
		"ecx.goku-ts/p_pain1.wav",
		"ecx.goku-ts/p_pain1.wav"
	},
	{
		"*/pain2.wav",
		"ecx.goku-ts/p_pain2.wav",
		"ecx.goku-ts/p_pain2.wav"
	},
	{
		"*/pain3.wav",
		"ecx.goku-ts/p_pain3.wav",
		"ecx.goku-ts/p_pain3.wav"
	},

	{
		"*/pain4.wav",
		"ecx.goku-ts/p_pain4.wav",
		"ecx.goku-ts/p_pain4.wav"
	},

	// << Death >>
	{
		"*/death.wav",
		"ecx.goku-ts/p_death.wav",
		"ecx.goku-ts/p_death.wav"
	}
};
```

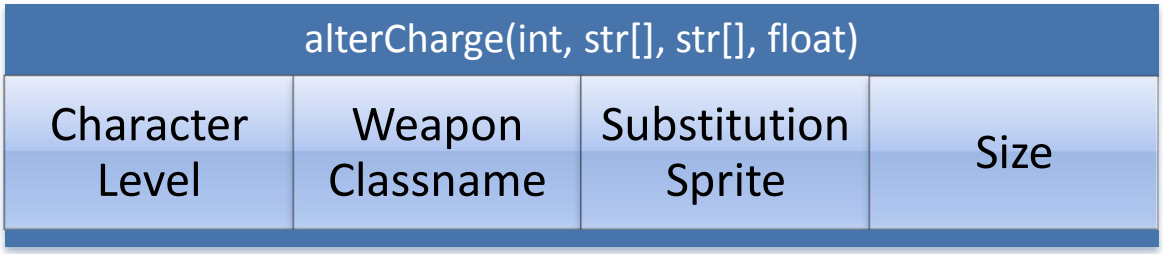
I hope you guys are still with me and you got the concept. We will leave it like that for now and move on to the next file, which is a pretty easy task, believe me.

MOD.Charge.core

The MOD.Charge gives you the ability to change weapon's charge sprite. Again just like Mod.Sound – MOD.Charge will precache its sprites automatically. The default file gives you something like this:

```
ModChargeInit ()
{
	//alterCharge( 0, "genericbeam", "sprites/w_gb_b_s.spr", 0.3 );
}
```

Let's have a closer look on the alter Charge function. Although this example gives you kind of a clue what it's supposed to do.



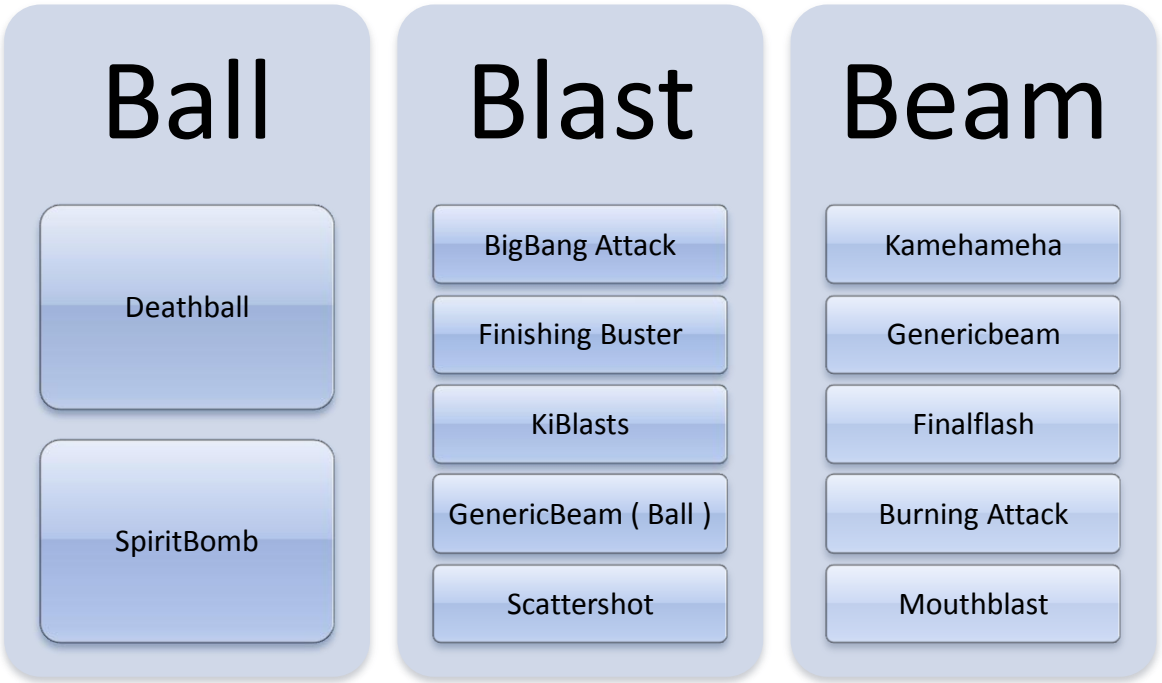
What we are going to do now is to set a charge replacement for the weapons we gave our character. The result will look like this:

```
ModChargeInit ()
{
	alterCharge( 0, "kamehameha", "sprites/w_kamehameha_c.spr", 0.3 );
	alterCharge( 1, "kamehameha", "sprites/w_kamehameha_c.spr", 0.4 );

	alterCharge( 0, "genericbeam", "sprites/w_gb_b_s.spr", 0.3 );
	alterCharge( 1, "genericbeam", "sprites/w_gb_y_s.spr", 0.3 );
}
```

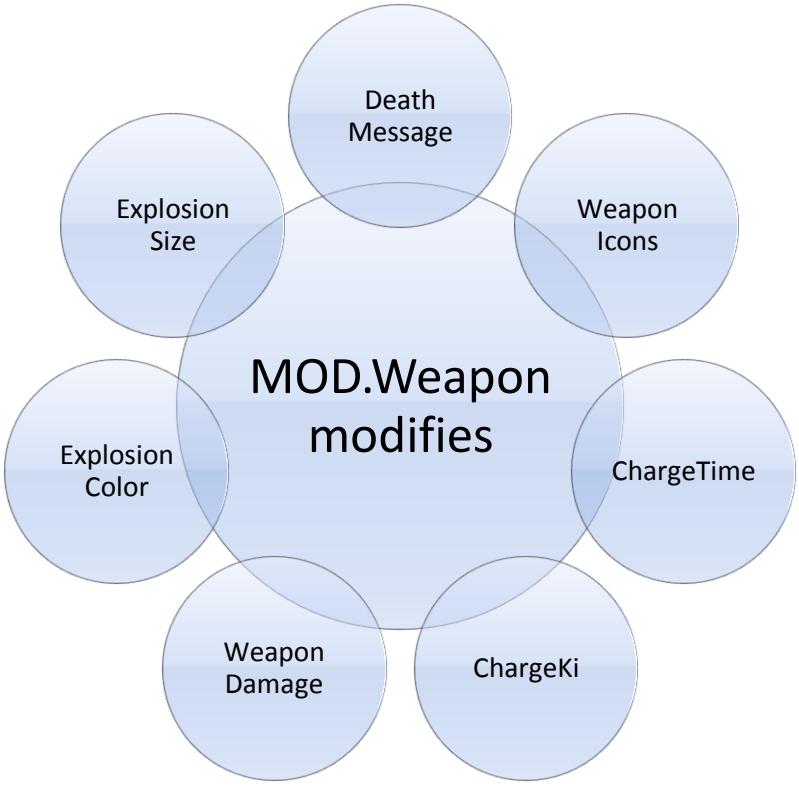
You will notice the slight different e.g. in the last alterCharge where we're using the yellow charge sprite because we are in level 1. All together I think this is the simplest part of this tutorial ☺

MOD.Weapon.core

MOD.Weapon is probably the most advanced part of a character. MOD.Weapon lets you modify any default weapon. However you will have to differentiate between some weapon types right from the start. We won't support discs or SBC here, because that is a little too advanced for now.



What MOD.Weapon can do as well is:



Now let's go through the individual functions:

global

- Declare variables for sprite precache / icon indices

stock MOD_Weapon_PreCache ()

- precache Sprites into the declared Variables
- get icon index into vars
- create custom icons

public @IconUpdate (Client, Level)

- replaces default icons with new ones

public @BaseWeaponCreation (Client, BaseWeapon, const Name[])

- sets weapon charge time
- sets weapons charge ki

public @WeaponAdjust (Client, Weapon, const Class[], PowerLevel, Charge)

- sets weapon damage
- sets weapon explosion size
- sets weapon explosion color
- add trail

public @WeaponBall (Client, Weapon)

- set Ball style
  - size
  - sprite

public @WeaponBlast (Client, Weapon, const Class[], Size)

- change sprites

public @WeaponBeam (Client, Weapon, const Class[], Size)

- change style
  - head, trail
  - size

public @WeaponDeath (Client, Killer, const Weapon[])

- alter death message
- "kamehameha" > "superkamehameha"

Ok now that we know what's going on with those functions we will consider what to use and what not to use. Since our character got three weapons, we don't have that many options to adjust, however I will try my best to give you as much insight as possible.

We will do what follows:

global
- define variables
  - genericbeam
  - kamehameha
  - superkamehameha icon

MOD_Weapon_PreCache ()
- precache
  - genericbeam spites
  - kamehameha sprite
- get Icon Index
  - weapon_kamehameha
- create custom icon
  - superkamehameha

@IconUpdate (Client, Level)
- replace kamehameha icon with superkamehameha icon on level 1

@BaseWeaponCreation (Client, BaseWeapon, const Name[])
- lenghten charge time for super kamehameha
- do more ki drain for superkamehameha

@WeaponAdjust (Client, Weapon, const Class[])
- change explosion color
  - genericbeam
- change explosion size
  - superkamehameha
- change weapon damage
  - superkamehameha

@WeaponBeam (Client, Weapon, const Class[], Size)
- on level 0 blue genericbeam
- on level 1 yellow genericbea,
- new non-default kamehameha sprites

@WeaponBlast (Client, Weapon, const Class[], Size)
- genericbeam (ball) blue on level 0
- genericbeam (ball) blue on level 1

@WeaponDeath (Client, Killer, const Weapon[])
- blueattack for genericbeam on level 0
- yellowattack for genericbeam on level 1
- superkamehameha for kamehameha on level 1

Global

Let's define some variables that is going to be easy:

```
stock SPR_SHINE;            // Shine

stock SPR_YELLOW_S;         // Yellow Beam Head
stock SPR_YELLOW_T;         // Yellow Beam Trail

stock SPR_BLUE_S;           // Blue Beam Head
stock SPR_BLUE_T;           // Blue Beam Head

stock SPR_KAME_S;           // Kamehameha Beam Head
stock SPR_KAME_E;           // Kamehameha Beam End
stock SPR_KAME_T;           // Kamehameha Beam Trail

stock SPR_SUKAME_S;         // Super Kamehameha Beam Head
stock SPR_SUKAME_E;         // Super Kamehameha Beam End
stock SPR_SUKAME_T;         // Super Kamehameha Beam Trail

stock ICON_KAMEHAMEHA;
stock C_ICON_SUKAME;
```

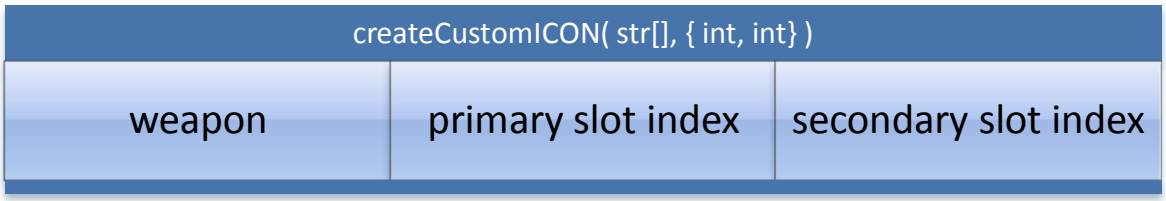
I think no explanation is needed for that.

Precache

```
stock MOD_Weapon_PreCache ()
{
	SPR_SHINE = precache_model( "sprites/ecx.shine.spr" );

	SPR_YELLOW_S = precache_model( "sprites/w_gb_y_s.spr" );
	SPR_YELLOW_T = precache_model( "sprites/w_gb_y_t.spr" );

	SPR_BLUE_S = precache_model( "sprites/w_gb_b_s.spr" );
	SPR_BLUE_T = precache_model( "sprites/w_gb_b_t.spr" );

	SPR_KAME_S = precache_model( "sprites/w_kamehameha_s.spr" );
	SPR_KAME_E = precache_model( "sprites/w_kamehameha_e.spr" );
	SPR_KAME_T = precache_model( "sprites/w_kamehameha_t.spr" );

	SPR_SUKAME_S = precache_model( "sprites/w_sukame_s.spr" );
	SPR_SUKAME_E = precache_model( "sprites/w_sukame_e.spr" );
	SPR_SUKAME_T = precache_model( "sprites/w_sukame_t.spr" );

	ICON_KAMEHAMEHA = getIconINDEX( "weapon_kamehameha" );
	C_ICON_SUKAME  = createCustomICON( "c_weapon_sukame", { 3, 1 } );

}
```

That was easy two except the last two lines right. I bet you haven't seen them before ☺

Weapon is the name of the textfile in your /ESF/sprites Directory, this is the Info we need later.



Again weapon is the name of the textfile in your /ESF/sprites Directory. The p.s.i. is the slot index of the key you press to access the first weapon in line of this specific key. The s.l.i. is the slot index of right this line. E.g. if there are two weapon both with an p.s.i. of 3 then you can define the order of these two with the s.l.i.

Icon Update

Now we're going to replace the kamehameha icon with the superkamehameha icon when our character is in level 1:

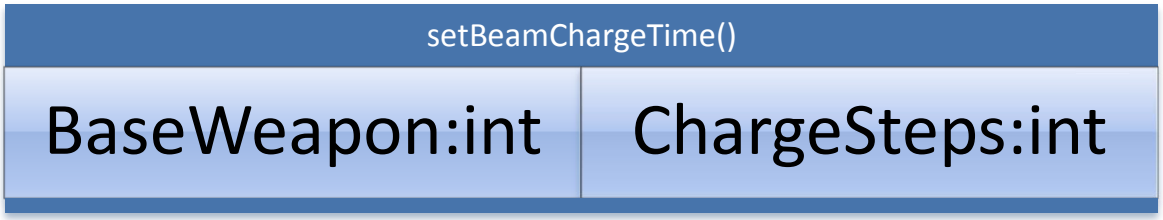
```
public @IconUpdate ( Client, Level )
{
        replaceClientICON( Client, C_ICON_FINALSHINE, ICON_KAMEHAMEHA );
}
```

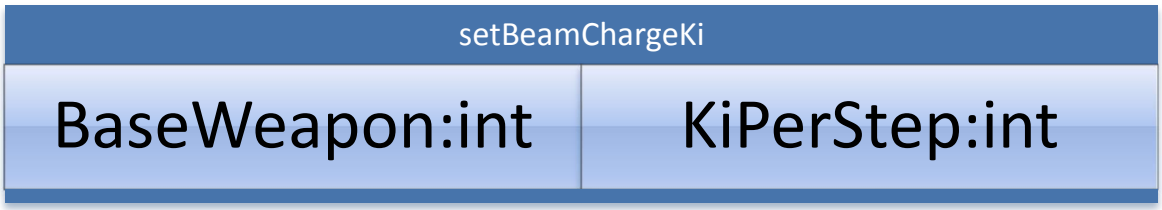
Client is the Client that is for sure. The new icon is the var in which you loaded the custom icon. The old icon is the var which contains the icon index of the weapon icon that is supposed to be replaced.

Base Weapon Creation

This is the kind of function that modifies the weapon while it's still in the charge.

```
public @BaseWeaponCreation ( Client, BaseWeapon, const Name[] )
{
	if ( equal( Name, "weapon_kamehameha" )  &&  getClientLEVEL(Client) == 1 )
	{
		setBeamChargeTime( BaseWeapon, 30 );
		setBeamChargeKi( BaseWeapon, 20 );
	}
}
```



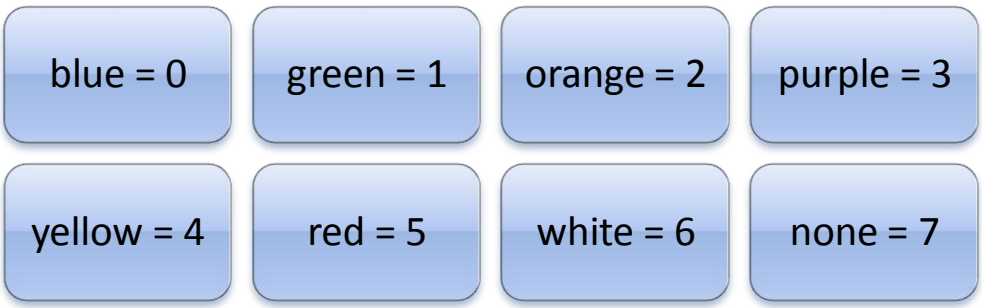


The total amount of required Ki for a full Charge would be 30*20= 600 Ki

Weapon Adjust

Weapon Adjust is the place where we modify our weapon after it's been created.

```
public @WeaponAdjust ( Client, Weapon, const Class[] )
{
	if ( equal( Class, "greenattack" ) && getClientLEVEL(Client) ==0  )
		setWeaponXPC( 0 );

	else if ( equal( Class, "greenattack" )  && getClientLEVEL(Client) ==1 )
		setWeaponXPC( 4 );

	else if ( equal( Class, "kamehameha" ) && getClientLEVEL(Client) ==1  )
	{
		// Super Kamehameha
		setWeaponDMG( 30.0, 200.0, 0.04 );
		setWeaponXPM( 10.0 );
	}
}
```



setWeaponXPM()
size: float

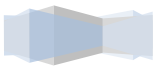
setWeaponXPM is the Weapon explosion multiplier. It just affects the size, not the damage!

setWeaponDMG()		
BaseDamage:float	MaxDamage: float	k: float

$$Damage = (MaxDamage + (BaseDamage - MaxDamage) * e^{\left(-\frac{-1}{Maxdamage * k}\right)*\left(\frac{PL}{1000000}\right)*\left(\frac{Charge}{100}\right)}$$

If you are not familiar with that kind of math formula don't bother. Leave k at 0.03

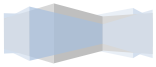
Please keep in mind that most people do NOT appreciate overpowered weapons or characters, as always Success lies in keeping the balance!



Weapon Beam

Here we are going to adjust the beam sprites as well as beam sizes.

```
public @WeaponBeam ( Client, Weapon, const Class[], Size )
{
	if ( equal( Class, "greenattack" ) && getClientLEVEL(Client) == 1 )
	{
		set_msg_arg_int( 4, ARG_SHORT, SPR_SHINE );
		set_msg_arg_int( 5, ARG_SHORT, SPR_YELLOW_S );
		set_msg_arg_int( 6, ARG_SHORT, SPR_YELLOW_S );
		set_msg_arg_int( 7, ARG_SHORT, SPR_YELLOW_S );
		set_msg_arg_int( 8, ARG_SHORT, SPR_YELLOW_T );
	}

	else if ( equal( Class, "greenattack" )  && getClientLEVEL(Client) == 0 )
	{
		set_msg_arg_int( 4, ARG_SHORT, SPR_SHINE );
		set_msg_arg_int( 5, ARG_SHORT, SPR_BLUE_S );
		set_msg_arg_int( 6, ARG_SHORT, SPR_BLUE_S );
		set_msg_arg_int( 7, ARG_SHORT, SPR_BLUE_S );
		set_msg_arg_int( 8, ARG_SHORT, SPR_BLUE_T );

		//set_msg_arg_int( 9, ARG_BYTE, 12 );
	}

	else if ( equal( Class, "kamehameha" )  && getClientLEVEL(Client) == 0)
	{
		set_msg_arg_int( 4, ARG_SHORT, SPR_SHINE );
		set_msg_arg_int( 5, ARG_SHORT, SPR_KAME_S );
		set_msg_arg_int( 6, ARG_SHORT, SPR_KAME_E );
		set_msg_arg_int( 7, ARG_SHORT, SPR_KAME_E );
		set_msg_arg_int( 8, ARG_SHORT, SPR_KAME_T );

		//set_msg_arg_int( 9, ARG_BYTE, 15 );
	}

	else if ( equal( Class, "kamehameha" )  && getClientLEVEL(Client) == 1)
	{
		set_msg_arg_int( 4, ARG_SHORT, SPR_SHINE );
		set_msg_arg_int( 5, ARG_SHORT, SPR_SUKAME_S );
		set_msg_arg_int( 6, ARG_SHORT, SPR_SUKAME_E );
		set_msg_arg_int( 7, ARG_SHORT, SPR_SUKAME_E );
		set_msg_arg_int( 8, ARG_SHORT, SPR_SUKAME_T );
		set_msg_arg_int( 9, ARG_BYTE, 15 ); // This is the size
	}
}
```

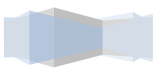
This looks more complicated than it actually is. All we do here is sprite replacements. I don't think further explanation is needed.

Weapon Blast

Here we are going to adjust the blast. In our case it's the blast that the genericbeam gives you with its secondary attack.

```
public @WeaponBlast ( Client, Weapon, const Class[], Size )
{
	if ( equal( Class, "greenattack" ) && getClientLEVEL( Client ) == 1 )
	{
		set_msg_arg_int( 2, ARG_SHORT, SPR_YELLOW_S );
		set_msg_arg_int( 3, ARG_SHORT, SPR_YELLOW_S );
		set_msg_arg_int( 5, ARG_SHORT, SPR_YELLOW_T );
		set_msg_arg_int( 6, ARG_SHORT, SPR_YELLOW_T );
	}

	else if ( equal( Class, "greenattack" ) && getClientLEVEL( Client ) == 0)
	{
		set_msg_arg_int( 2, ARG_SHORT, SPR_BLUE_S );
		set_msg_arg_int( 3, ARG_SHORT, SPR_BLUE_S );
		set_msg_arg_int( 5, ARG_SHORT, SPR_BLUE_T );
		set_msg_arg_int( 6, ARG_SHORT, SPR_BLUE_T );
	}
}
```

It's basically the same as Weapon Beam.



Weapon Death

Weapon Death is to modify the Death Message of your weapons. Please pay attention to the arguments, because this time there is a Client and a Killer. Client is the Victim!

```
public @WeaponDeath ( Client, Killer, const Weapon[] )
{
	if ( equal( Weapon, "greenattack" ) && getClientLEVEL( Killer ) == 0 )
		set_msg_arg_string( 3, "blueattack" );

	else if ( equal( Weapon, "greenattack" ) && getClientLEVEL( Killer ) == 1 )
		set_msg_arg_string( 3, "yellowattack" );

	else if ( equal( Weapon, "kamehameha" )  && getClientLEVEL( Killer ) == 1)
		set_msg_arg_string( 3, "Super Kamehameha" );
}
```

That's it you did it,

and I'm tired as hell, because I really hate playing the Personal Trainer.