Diesel Engine

Category for Diesel Engine related docs

- SystemFS
- Bundle File
- Unit File
- Skin map textures (df_cc)
- Animated Models
- Material Config XML
- Object Xml
- Sequence Manager XML
- Animation State Machine

SystemFS

SystemFS is a class to read and write files and more.

Currently, it works only in the Windows version of the game.

I'll add more info later right now I'll just list the functions and their parameters.

Functions:

open(path, flags)

close(file)

exists(path)

is_dir(path)

system_path(path)

parse_xml(path)

make_dir(path)

delete_file(path)

can delete a folder also.

list(dir, directories)

returns a table of files by default and folders if directories parameter is true

copy_file(from, to)

copy_files_async(files, callback)

each file needs to be inside the files table so for example if I want to move file x to path y it's gonna be: SystemFS:copy_files_async({"x", "y"}) the callback has two parameters - success and message if the copy failed then success will be false and message should say something.

can_write_to(path)

checksum(path)

Bundle File

*__Originally written by Simon W.__

Bundle Database (BLB)

The Bundle Database file, also known as “bundle_db.blb” and “all.blb”, defines all file entries with their hashed path, hashed extension, language, and a unique ID. Special things to note: “idstring_lookup.idstring_lookup” has an ID that is equal to the count of file entries.

File structure of the .blb format

Header

```
uint32 languages_tag //Payday 2 = "8C F2 18 00" Payday: The Heist = "9C 2B F2 00"
uint32 languages_count
uint32 languages_count //should be same value as previous uint32
uint32 languages_offset //offset to the beginning of languages section
uint32 unknown
uint32 unknown
uint32 unknown
uint32 file_entries_count
uint32 file_entries_count //should be same value as previous uint32
uint32 file_entries_offset //offset to the beginning of file entries section
uint32 unknown
uint32 unknown
uint32 unknown
uint32 unknown
```

Languages section

```
FOREACH( languages_count )
 uint64 language_hash
 uint32 language_representation
```

```
 uint32 unknown
END FOREACH
```

File entries section

```
FOREACH( file_entries_count )
 uint64 file_entry_extension_hash
 uint64 file_entry_path_hash
 uint32 file_entry_language //one of the language representations
 uint32 unknown
 uint32 file_entry_ID
 uint32 unknown
END FOREACH
```

Packages (bundles)

Bundles are diesel's packages containing files with corresponding IDs. The package name can be looked up with Bundle Modder by using the hash converter on the bundle name with “use hex” and “swap endianness” (this does not work with all_x bundles).

Which then can be used in lua to load packages. Bundles are split into two files, header and data. The header files end with “*_h.bundle” while data files end with “*.bundle”.

File structure of the _h.bundle (Header)

Header section

```
 uint32 section_size
 uint32 section_tag //Since update 70, this tag is present in every file. Payday: The Heist is
present only if bundle is all_x
 uint32 file_entries_count
 uint32 file_entries_count //should be same value as previous uint32
 uint32 file_entries_offset //if section_tag == "00 00 00 00" then file_entries_offset += 4

 uint32 file_entries_tag //Payday 2 = "E8 EB 18 00" Payday: The Heist = "88 EE 18 00"
 FOREACH( file_entries_count )
```

```
 uint32 file_entry_ID
 uint32 file_entry_address //address within the *.bundle file
 IF bundle is all_x then
  uint32 file_entry_length //the length of this file entry
 END IF
END FOREACH
```

Footer section

Please note that footer items correspond to the header file_entries in same order.

```
uint32 section_tag //Payday 2 and Payday: The Heist = "F8 C5 FC EB"
uint32 section_size
uint32 section_item_count
uint32 unknown
uint32 unknown
uint32 unknown //tag?
FOREACH( section_item_count )
 uint64 item_extension_hash
 uint64 item_path_hash
 uint32 unknown //end tag?
END FOREACH
uint32 zero //end
```

File structure of the h.bundle (Data)

This file consists of a collection of files. Each file entry starts at a specified address in header with a specified length, if length is not specified then it is calculated as a difference between current entry address and next.

```
FOREACH( file_entries_count )
 byte[] file_entry
END FOREACH
```

Unit File

This is an explanation of how units work in Payday 2. This may not be 100% perfect, but it should give the basic idea of what's going on.

A “unit” in Payday 2 is an object like a mask or an enemy. It basically sets properties, applies effects, and specifies textures for the units.

Files

- **.unit** - Points to a *.object file for further loading data about the unit. Specifies dependencies like sounds, effects, or animations. Establishes the extensions, which consist of properties for the unit like inventory, AI, damage, movement, sounds, or interactions. Sometimes can specify a path for the .unit file that is supposed to be loaded when the unit is spawned over network. And finally can specify the location of sound sources for the unit.
- **.object** - Points to a **.material_config** file, **.sequence_manager** file and the animations. For some units, there are “bodies” that establish which parts of bodies are enabled, and sets properties for them like friction, collision class, and a template. Then, there are “constraints”, which mostly establish how far limbs can rotate. After that, “decal_surfaces” are established, which basically assigns a specific decal material to parts of bodies. And finally, “graphics” are specified, they set which parts of model are enabled, and sometimes sets a LOD for them depending on distance from the unit.
- **.model** - This is the model for the unit.
- **.material_config** - Contains a list of materials with specified templates that are applied to the textures. Points to *.texture files to use for reflections, diffused textures and normal maps.
- **.cooked_physics** - Not much is known about this file but it is related to physics of the unit.
- **.sequence_manager** - This file contains sequences which can be run on the unit.
- **.texture** - This is a renamed **.dds** texture file.

Loading Order

Units in Payday are loaded in a specific order. If a file in this chain has an error, the unit will not work correctly and potentially crash the game.

The loading order is as follows: .unit file is loaded first (this file has to point to the .object file to load it next), next the .object file is loaded (this file often points to the .material_config file, which

contains textures for the model to use), next either .model or .material_config is loaded (.material_config has specified texture files that it loads and applies a specific template with effects to them. And .model contains the model and retrieves materials from the .material_config).

.unit Details

The .unit file is the first to get loaded when accessing the unit. This file is in .xml format, with a fairly simple structure. Most of the things in these .unit files are pretty self explanatory. However, the weapon units, mask units, and character units are all different. As they are obviously none of them are the same at all! Here is a sample of a .unit file.

Sample Breakdown of a .unit file

As an example, I will be looking at the cloaker's (spook's) unit files. First, the .unit file of a cloaker are located at (units\payday2\characters\ene_spook_1\ene_spook_1.unit).

```
<unit type="being" slot="12">
  <anim_state_machine name="anims/units/enemies/cop/cop_machine" />
  <object file="units/payday2/characters/ene_spook_1/ene_spook_1" />

  <dependencies>
      <depends_on animation_state_machine="anims/units/enemies/cop/cop_machine"
animation_def="anims/units/enemies/cop/cop_def" />
      <depends_on bnk="soundbanks/regular_vox" />
      <depends_on effect="effects/particles/character/cloaker_goggle" />
      <depends_on unit="units/payday2/characters/ene_acc_baton/ene_acc_baton" />
  </dependencies>

  <extensions>
      <extension name="unit_data" class="ScriptUnitData" />
      <extension name="base" class="CopBase" >
          <var name="_tweak_table" value="spooc" />
          <var name="_default_weapon_id" value="mp5_tactical" />
      </extension>
      <extension name="inventory" class="CopInventory" />
      <extension name="brain" class="CopBrain" />
      <extension name="anim_data" class="PlayerAnimationData" />
      <extension name="character_damage" class="CopDamage">
          <var name="_head_body_name" value="head" />
```

```
            <var name="_death_sequence" value="kill_spook_lights" />
        </extension>
        <extension name="movement" class="CopMovement" >
            <var name="_footwear" value="boots" />
            <var name="_anim_global" value="cop" />
        </extension>
        <extension name="interaction" class="IntimitateInteractionExt" >
            <var name="tweak_data" value="intimidate" />
        </extension>
        <extension name="network" class="NetworkBaseExtension" />
        <extension name="damage" class="UnitDamage" >
            <var name="_skip_save_anim_state_machine" value="true" />
        </extension>
        <extension name="contour" class="ContourExt" />
        <extension name="sound" class="CopSound" />
    </extensions>

    <network sync="spawn" remote_unit="units/payday2/characters/ene_spook_1/ene_spook_1_husk"/>

    <sounds>
        <default_soundsource source="Hips"/>
    </sounds>
</unit>
```

Please notice the structure: anim_state_machine, then object, then dependencies, then extensions, then network, then sounds. Sometimes, if this structure is not followed, the game will crash. (Note, Payday: The Heist sometimes does not follow this structure, so PD1 .unit files would often result in a crash).

Let's breakdown this file by sections.

```
<unit type="being" slot="12">
```

The first section states that this unit is a human being (I am assuming when you hit it, blood will come out). And that it's in slot 12 (I believe slot number can be ignored).

```
<anim_state_machine name="anims/units/enemies/cop/cop_machine" />
```

The second section establishes the animations that this unit will use. In this case, the cloaker will be using cop animations.

```
<object file="units/payday2/characters/ene_spook_1/ene_spook_1" />
```

The third line establishes where the next file is, the .object file. This file will be loaded after the .unit (Please note that the path does not contain an extension, the game already knows that you pointed at a .object file.

```
<dependencies>
    <depends_on animation_state_machine="anims/units/enemies/cop/cop_machine"
animation_def="anims/units/enemies/cop/cop_def" />
    <depends_on bnk="soundbanks/regular_vox" />
    <depends_on effect="effects/particles/character/cloaker_goggle" />
    <depends_on unit="units/payday2/characters/ene_acc_baton/ene_acc_baton" />
</dependencies>
```

This block of code establishes the dependencies that this unit has. This unit is dependent on the cop animations using the cop animation definitions. Next, this unit is dependent on the sound bank “soundbanks/regular_vox” (this soundbank is related to speech). Next, this unit is dependent on an effect, the “effects/particles/character/cloaker_goggle”, in the .object file this effect will be assigned to a specific location. And finally, this unit is dependent on another unit “units/payday2/characters/ene_acc_baton/ene_acc_baton” (this counts as an enemy accessory, thus the name “ene_acc”). And once again, not a single path has an extension, the game knows.

```
<extensions>
    <extension name="unit_data" class="ScriptUnitData" />
    <extension name="base" class="CopBase" >
        <var name="_tweak_table" value="spooc" />
        <var name="_default_weapon_id" value="mp5_tactical" />
    </extension>
    <extension name="inventory" class="CopInventory" />
    <extension name="brain" class="CopBrain" />
    <extension name="anim_data" class="PlayerAnimationData" />
    <extension name="character_damage" class="CopDamage">
        <var name="_head_body_name" value="head" />
        <var name="_death_sequence" value="kill_spook_lights" />
    </extension>
    <extension name="movement" class="CopMovement" >
        <var name="_footwear" value="boots" />
        <var name="_anim_global" value="cop" />
    </extension>
    <extension name="interaction" class="IntimitateInteractionExt" >
        <var name="tweak_data" value="intimidate" />
    </extension>
    <extension name="network" class="NetworkBaseExtension" />
```

```
    <extension name="damage" class="UnitDamage" >
        <var name="_skip_save_anim_state_machine" value="true" />
    </extension>
    <extension name="contour" class="ContourExt" />
    <extension name="sound" class="CopSound" />
</extensions>
```

This chunk of code assigns basic unit things, like AI, inventory, sounds, etc. Most of them stay the same, but variables change. The “unit_data” extension is present practically in every .unit file and does not seem to change. The “base” extension is usually present on characters or usable objects. In “base” the class changes depending on the unit, and the variables in the extension also change. For this unit, the variables set the identity of this unit as “spooc” and assigns it “mp5_tactical” as a default weapon. The “inventory” seems to be only present on units that can carry items (like ammo or other weapons). The class usually stays as “CopInventory”, but there could be multiple kinds of “inventory”. The “brain” assigns AI to the unit, in this case “CopBrain” is assigned. The “anim_data” is currently unknown to me, but I am guessing that these are the kinds of animations a unit can perform, “PlayerAnimationData” is assigned. The “character_damage” assigns various things regarding the damage the unit will take, “CopDamage” is assigned. Two variables are assigned as well, “_head_body_name” signifies what part of body (according to model) is considered to be the head, the “_deadh_sequence” signifies what sequence this unit will perform at death. The “movement” assigns what kind of movement this unit will perform, “CopMovement” is assigned, as well as two other variables. The first variable “_footwear” states what kind of shoes the unit will have, “boots” are assigned. Second variable “_anim_global” states what kind of movement animations this unit will perform, “cop” animations are assigned. The “interaction” is usually present with units that can be interacted with, I am lacking detail as to what kind of interaction, “IntimateInteractionExt” is assigned, with one variable. The variable “tweak_data” is pretty much always present with “interaction”, “intimate” is assigned. The “network” extension is usually present on units that can be spawned or changed during the game, “NetworkBaseExtension” is assigned. The “damage” extension is usually present with units that can deal damage, “UnitDamage” is assigned with one variable. The variable “_skip_save_anim_state_machine” is related to animations and I am unsure about the exact usage of this, variable is set to “true”. The “contour” extension is usually present with units that can have an outline, class of “ContourExt” is assigned. The “sound” extension determines what kind of sounds this unit can make, “CopSound” is assigned.

```
<network sync="spawn" remote_unit="units/payday2/characters/ene_spook_1/ene_spook_1_husk"/>
```

This section is usually present with units that can be spawned or changed during the game. This is responsible for syncing the spawn, by specifying the .unit file to be loaded on the clients. (Lobby host does not use this, only the client).

```
<sounds>
    <default_soundsource source="Hips"/>
</sounds>
```

This section specifies the source of sound for the unit. Apparently the cloaker (and all other enemies) make sounds from their “Hips”.

After the .unit file is loaded, the .object gets loaded. Not all commands were listed in this example, so other commands will be listed below in the “Other .unit commands” section with their explanations.

Other .unit commands

Other .unit commands will be added here as research progresses.

.object Details

The .object file is second to get loaded when accessing the unit. This file is in .xml format, with a fairly simple structure. Most of the things in these .object files are pretty self explanatory, and about 80% unique to the unit, as it heavily replies on the model and the names of body parts in the model. The .object file usually contains properties of model parts like materials, sequence_manager, bodies, constraints, effects, graphics, and lights. Here is a continuation of the unit breakdown from previous section.

Sample Breakdown of a .object file

Following from the previous section, the .object file path of a cloaker was assigned as “units/payday2/characters/ene_spook_1/ene_spook_1” (that's without the .object at the end). The .object files tend to be repetitive, as they assign each “body” in a model, specific settings. And for sake of space, the .object file will be cut down to include as little repetition as possible.

```
<dynamic_object>
  <diesel materials="units/payday2/characters/ene_spook_1/ene_spook_1"
orientation_object="root_point" />
  <sequence_manager file="units/payday2/characters/ragdoll" /> <animation_def
name="anims/units/enemies/cop/cop_def" />

  <bodies>
      <body name="body" enabled="true" template="character" friction="0.6"
collision_class="ragdoll">
          <object name="Spine1"/>
          <object name="c_capsule_body" collision_type="capsule"/>
      </body>
```

```
      <body name="mover_blocker" enabled="true" template="mover_blocker" keyframed="true"
collision_class="ragdoll">
          <object name="root_point"/>
          <object name="c_capsule_mover_blocker" collision_type="capsule"/>
      </body>
              ***PART OF THE FILE WAS SNIPPED HERE***

<!-- RAGDOLL -->
      <body name="rag_Head" enabled="false" template="corpse" friction="0.01" sweep="true"
collision_class="ragdoll" keyframed="false" collision_script_quiet_time="0.5"
collision_script_tag="small" ray="block" lin_damping="0.6" ang_damping="20" collides_with="0"
tag="flesh" restitution="0">
          <object name="Neck" />
          <object collision_type="sphere" mass="4" padding="-15"
name="c_sphere_head_ragdoll"/>
      </body>

      <body name="rag_Hips" enabled="false" template="corpse" friction="0.6" sweep="true"
collision_class="ragdoll" keyframed="false" collision_script_quiet_time="0.5"
collision_script_tag="large" ray="block" lin_damping="0.4" ang_damping="20"
collision_group="1" collides_with="0" tag="flesh" restitution="0">
          <object name="Hips" />
          <object collision_type="capsule" mass="22" padding="-5" name="c_sphere_Hips" />
      </body>
      ***PART OF THE FILE WAS SNIPPED HERE***
  </bodies>

  <constraints>
      <constraint type="ragdoll" name="RightArm" enabled="false">
          <param body_a="rag_Spine2" body_b="rag_RightArm"/>
          <param pivot="position:RightArm"/>
          <param twist_axis="yaxis:RightArm" twist_min="-60" twist_max="70"
twist_freedom="20"/><!-- X axis -->
          <param plane_axis="xaxis:RightArm"/><!-- Y axis -->
          <param cone_y="35" cone_z="40" cone_freedom="10"/>
          <param damping="1" spring_constant="200" min_restitution="0"/>
      </constraint>

      <constraint type="limited_hinge" name="RightForeArm" enabled="false">
          <param body_a="rag_RightArm" body_b="rag_RightForeArm"/>
```

```
          <param pivot="position:RightForeArm"/>
          <param min_angle="-60" max_angle="60" axis="yaxis:RightForeArm" twist_freedom="5"/>
<!-- X axis -->
          <param plane_axis="xaxis:RightForeArm"/> <!-- Y axis -->
          <param damping="1" spring_constant="200" min_restitution="0"/>
      </constraint>
      ***PART OF THE FILE WAS SNIPPED HERE***
  </constraints>

  <decal_surfaces default_material="flesh" />

  <effects>
      <effect_spawner name="es_light" enabled="false" object="e_light"
effect="effects/particles/character/cloaker_goggle" />
  </effects>

  <graphics>
      <graphic_group name="character" enabled="true" culling_object="g_body">

          <lod_object name="lod_body">
              <object name="g_body" □□enabled="true"  max_distance="3000"  max_draw_lod="0" />
              <object name="g_body_lod1" □enabled="true" lod="1" />
          </lod_object>

          <object name="s_body" enabled="true" shadow_caster="true"/>

          <object name="g_il" □□enabled="false" />

      </graphic_group>
  </graphics>

  <lights>
      <light name="point_light" enabled="false" multiplier="reddot" far_range="25"
near_range="1" falloff="4.0" type="omni|specular" />
  </lights>

</dynamic_object>
```

Please notice the structure: diesel materials, then sequence_manager, then animation_def, then bodies, then constraints, then decal_surfaces, then effects, then graphics, and then lights.

Sometimes, if this structure is not followed, the game will crash. (Note, Payday: The Heist sometimes does not follow this structure, so PD1 .object files would often result in a crash).

Like before, Let's breakdown this file by sections.

```
<diesel materials="units/payday2/characters/ene_spook_1/ene_spook_1"
orientation_object="root_point" />
```

This section specifies the location of the the .material_config file, along with the orientation position. This is present in most units that have a model (some weapons don't seem to specify this). The .material_config file is specified to be “units/payday2/characters/ene_spook_1/ene_spook_1” (once again, no .material_config, game knows) with the orientation at “root_point” of the model.

```
<sequence_manager file="units/payday2/characters/ragdoll" />` `<animation_def
name="anims/units/enemies/cop/cop_def" />
```

This section specifies what sequence_manager file to use. And what animations to use. (Sometimes these are separated into two lines). The sequence_manager file is specified to be located at “units/payday2/characters/ragdoll” (no .sequence_manager extension). And the “anims/units/enemies/cop/cop_def” animation definition is set to be used.

```
<bodies>
  □□<body name="body" enabled="true" template="character" friction="0.6"
collision_class="ragdoll">
  □□□<object name="Spine1"/>
  □□□<object name="c_capsule_body" collision_type="capsule"/>
  □□</body>
  □□<body name="mover_blocker" enabled="true" template="mover_blocker" keyframed="true"
collision_class="ragdoll">
  □□□<object name="root_point"/>
  □□□<object name="c_capsule_mover_blocker" collision_type="capsule"/>
  □□</body>
               ***PART OF THE FILE WAS SNIPPED HERE***

  <!-- RAGDOLL -->
  □□<body name="rag_Head" enabled="false" template="corpse" friction="0.01" sweep="true"
collision_class="ragdoll" keyframed="false" collision_script_quiet_time="0.5"
collision_script_tag="small" ray="block" lin_damping="0.6" ang_damping="20" collides_with="0"
tag="flesh" restitution="0">
  □□□<object name="Neck" />
  □□□<object collision_type="sphere" mass="4" padding="-15" name="c_sphere_head_ragdoll"/>
```

```
  □□</body>

  □□<body name="rag_Hips" enabled="false" template="corpse" friction="0.6" sweep="true"
collision_class="ragdoll" keyframed="false" collision_script_quiet_time="0.5"
collision_script_tag="large" ray="block" lin_damping="0.4" ang_damping="20"
collision_group="1" collides_with="0" tag="flesh" restitution="0">
  □□□<object name="Hips" />
  □□□<object collision_type="capsule" mass="22" padding="-5" name="c_sphere_Hips" />
  □□</body>
               ***PART OF THE FILE WAS SNIPPED HERE***
  □</bodies>
```

This section pretty much defines collisions and ragdolls per body parts in the model. For the first “body” is enabled (as it's set to true), the templace for “character” is used with friction of “0.6”, and a collision class of “ragdoll”. This seems to include the object “Spine1” and “c_capsule_body” of collision type “capsule”. Both of those objects are most likely defined in the .model file. **THIS IS NOT FINISHED!!!**

```
<constraints>
    <constraint type="ragdoll" name="RightArm" enabled="false">
        <param body_a="rag_Spine2" body_b="rag_RightArm"/>
        <param pivot="position:RightArm"/>
        <param twist_axis="yaxis:RightArm" twist_min="-60" twist_max="70"
twist_freedom="20"/><!-- X axis -->
        <param plane_axis="xaxis:RightArm"/><!-- Y axis -->
        <param cone_y="35" cone_z="40" cone_freedom="10"/>
        <param damping="1" spring_constant="200" min_restitution="0"/>
    </constraint>

    <constraint type="limited_hinge" name="RightForeArm" enabled="false">
        <param body_a="rag_RightArm" body_b="rag_RightForeArm"/>
        <param pivot="position:RightForeArm"/>
        <param min_angle="-60" max_angle="60" axis="yaxis:RightForeArm" twist_freedom="5"/>
<!-- X axis -->
        <param plane_axis="xaxis:RightForeArm"/> <!-- Y axis -->
        <param damping="1" spring_constant="200" min_restitution="0"/>
    </constraint>
    ***PART OF THE FILE WAS SNIPPED HERE***
</constraints>
```

This section deals with constraints of rotations and movement. **THIS SECTION NEEDS MORE EXPLANATION, BUT IS SELF EXPLANATORY!!!**

```
<decal_surfaces default_material="flesh" />
```

This little section states that the default material for the unit is “flesh”. There are a few other default materials besides flesh, and sometimes they're even applied per body part in this section.

```
<effects>
<effect_spawner name="es_light" enabled="false" object="e_light"
effect="effects/particles/character/cloaker_goggle" />
</effects>
```

This section applies effects to the unit. I am not 100% certain on the application process. I believe that the effect under the name “es_light” is being applied to the object “e_light” (probably stated in .model) from effect file “effects/particles/character/cloaker_goggle” (once again, .effect extension is not needed).

The name of the effect does not matter; it can be set to anything you want. It seems to be only for referential purposes. The effect does not necessarily need to be applied to an "e_light" object, as other objects in the file will work as well (such as "g_body" or "root_point").

```
<graphics>
    <graphic_group name="character" enabled="true" culling_object="g_body">

        <lod_object name="lod_body">
            <object name="g_body" enabled="true"  max_distance="3000"  max_draw_lod="0" />
            <object name="g_body_lod1" enabled="true" lod="1" />
        </lod_object>

        <object name="s_body" enabled="true" shadow_caster="true"/>

        <object name="g_il" enabled="false" />

    </graphic_group>
</graphics>
```

In this section, there are two things happening. First, the LOD is being set per distance (in centimeters). So at distance < 3000 cm the default LOD model with be drawn to screen. If distance > 3000 cm then the LOD1 will be drawn. Second, some elements of the model are enabled/disabled in this section. As you can see, the “s_body” is enabled (with “true”) and is set to cast a shadow with shadow_caster set to “true”. And then there is “g_il” which is disabled. Please note that not all elements of the .model can be disabled here. Only the ones you know (i.e. the

ones already listed in this .object file) or the ones you know from the model (currently there is no way of looking up element names from models.

```
<lights>
    <light name="point_light" enabled="false" multiplier="reddot" far_range="25"
near_range="1" falloff="4.0" type="omni|specular" />
</lights>
```

This section is for creating a light on the unit. For this unit in particular, it creates a glow around them that gets enabled via the sequence_manager. This “point_light” has a range of 1 - 25 with the falloff of “4.0” (I think the falloff is for the type) and type of “omni|specular”. I am not certain about what this type specifically does, but it certainly acts as an effect on this “point_light”.

After the .object file is loaded, either the .model or the .material_config file get loaded. Not all commands were listed in this example, so other commands will be listed below in the “Other .object commands” section with their explanations.

Other .object commands

Other .object commands will be added here as research progresses.

.model Details

Currently there are no details on the .models, as the filetype has not been completely reverse engineered.

Research Notes:

`.model contains hashed names of objects using Hash64, uint64. (Currently looking into editing elements)`

`Bones have been redone since Payday:`` ``The`` ``Heist , they now don't include 4th elements of fingers.`
`Each bone is specified as a 3D Object, which contains rotation matrix, position, and a parent ID.`

`Observations:`
`*It's near impossible to find model names, as they are hashed and unhashing them would be near impossible. They are not part of idstring.`
`*I'm assuming .material_config hashes the name of material, and applies it to the model. If it doesn't exist, it still applies (to nothing).`
`*It would be easier to create models from scratch, as you will know the names of all objects and materials, so you would have full control over the model.`

.material_config Details

The .material_config file is loaded sometime after the .object file. This file is in .xml format, with a fairly simple structure. Most of the things in these .material_config files are pretty self explanatory, and is unique to the unit, as it heavily replies on the model and the names of body parts in the model. The .material_config file contains texture paths (diffused and bump map textures), sometimes reflection textures, sometimes material_textures, and sometimes some variables for the render_template. Here is a continuation of the unit breakdown from previous section.

Sample Breakdown of a .material_config file

In this example we will be using the cloaker. The .material_config file path of a cloaker was assigned as “units/payday2/characters/ene_spook_1/ene_spook_1” (that's without the .material_config at the end). The .material_config files tend to be repetitive, as they assign each the requested material names in the model, specific textures and effects. And for sake of space, the .material_config file will be cut down to include as little repetition as possible.

```
<materials version="3" group="ene_spook_1">
  <material name="mtr_body"
render_template="generic:DIFFUSE_TEXTURE:NORMALMAP:RL_COPS:SKINNED_3WEIGHTS" version="2">
      <bump_normal_texture □□□file="units/payday2/characters/shared_textures/spook_heavy_nm"/>
      <diffuse_texture □□□□file="units/payday2/characters/shared_textures/spook_heavy_df"/>
  </material>
  <material name="mtr_il"
render_template="generic:ALPHA_MASKED:DIFFUSE_TEXTURE:OPACITY_TEXTURE:RL_COPS:SELF_ILLUMINATIO
N" version="2">
      <diffuse_texture □□□□file="units/payday2/characters/shared_textures/spook_il"/>
      <self_illumination_texture □□file="units/payday2/characters/shared_textures/spook_il"/>
      <opacity_texture □□□□file="units/payday2/characters/shared_textures/spook_il"/>
      <variable □□□□□□value="reddot" type="scalar" name="il_multiplier"/>
  </material>
    <material name="shadow_caster" render_template="shadow_caster_only:SKINNED_1WEIGHT"
version="2"/>
</materials>
```

There is no specific structure to follow. This file seems to be a list of materials with some variables attached. The only real problems that can occur are incorrect textures, broken model, no model at

all (but a floating blob of gray).

Let's breakdown this file by sections.

```
<materials version="3" group="ene_spook_1">
```

This establishes the group that these materials belong to. I am not sure as to what group names can be given, but it's best to keep them similar to the original model names. The version does not seem to matter.

```
<material name="mtr_body"
render_template="generic:DIFFUSE_TEXTURE:NORMALMAP:RL_COPS:SKINNED_3WEIGHTS" version="2">
    <bump_normal_texture □□□file="units/payday2/characters/shared_textures/spook_heavy_nm"/>
    <diffuse_texture □□□□file="units/payday2/characters/shared_textures/spook_heavy_df"/>
</material>
```

This section identifies a material “mtr_body” with a render_template of “generic:DIFFUSE_TEXTURE:NORMALMAP:RL_COPS:SKINNED_3WEIGHTS” and version of “2” (once again, does not seem to matter). The material name has to match the one listed in the .model file, otherwise the model will be broken. The render_template is a predefined template, and **CANNOT SIMPLY BE APPENDED**, there is a list of available render_templates with explanations HERE. This material has two variables included (these are present with almost every material). The “bump_normal_texture” specifies where the bump map of this material is, for this example it's located at “units/payday2/characters/shared_textures/spook_heavy_nm” (once again, the .texture extension is not needed). Followed by a “diffuse_texture”, which is the actual texture of the material, located at “units/payday2/characters/shared_textures/spook_heavy_df”. Both the “bump_normal_texture” and the “diffuse_texture” are passed to the render_template to be handled.

```
<material name="mtr_il"
render_template="generic:ALPHA_MASKED:DIFFUSE_TEXTURE:OPACITY_TEXTURE:RL_COPS:SELF_ILLUMINATIO
N" version="2">
    <diffuse_texture □□□□file="units/payday2/characters/shared_textures/spook_il"/>
    <self_illumination_texture □□file="units/payday2/characters/shared_textures/spook_il"/>
    <opacity_texture □□□□file="units/payday2/characters/shared_textures/spook_il"/>
    <variable □□□□□□value="reddot" type="scalar" name="il_multiplier"/>
</material>
```

This section right here is almost identical to the previously viewed material. To differentiate this new material, it has a different, uses a different render_template, and has a few new variables. The diffuse texture serves the same purpose as before. The new, “self_illumination_texture” points to the path of “units/payday2/characters/shared_textures/spook_il”. This “self_illumination_texture” is related to the render_template. Same with "opacity_texture " and the “variable”. All of them are

passed to to the render_template to be handled.

```
<material name="shadow_caster" render_template="shadow_caster_only:SKINNED_1WEIGHT"
version="2"/>
```

This last section is responsible for casting shadows. With name “shadow_caster” and render_template of “shadow_caster_only:SKINNED_1WEIGHT”. A list of available render_templates with explanations HERE.

After the .material_config file is loaded, nothing else loads. Not all commands were listed in this example, so other commands will be listed below in the “Other .material_config commands” section with their explanations.

Other .material_config commands

Other .material_config commands will be added here as research progresses.

Skin map textures (df_cc)

Skin map texture

The _cc_df textures are used instead of diffuse on models when skin is applied and each channel controls specific aspect of it:

Red

Red channel is used as material map for 6 defined types of it.

Materials shown in overkill base gradient template.

Example of PD2 Aimpoint material map layer.

Green

Green channel seems to be mix of various maps with diffuse as main goal of this channel is to give skin a shape and details like a diffuse but with colors controlled by skin.

Example of PD2 EOTech sight.

Blue

Blue channel is used to create "wear and tear" damage to skin as quality of skin lowers and gradient timeline progresses.

Example of wear and tear on PD2 EOTech sight.

Creating skin textures for custom models

Red

Material map can be either created by manually painting parts on channel or using exported UV layouts of selected meshes. Painted parts must only use solid color as any form of gradients or blending will results in artifacts.

Use these RGB color values to assign specific material:

- Metal - #000000
- Plastic - #525252
- Wood/Rubber - #5a5a5a
- Sec. Metal - #848484
- Cloth - #adadad
- Details - #efefef

Green

Diffuse texture in grayscale can be reused for this channel. Image must be in edited be around light gray colors (Too dark image will results in missing details on models).

Blue

For wear and tear layer any texture of scratches with transparency or on white background can be used.

Example of random texture with scratches. For best transition between skin quality scratches should have some black/white range can go fully from 0 to 255.

Black color = Part of texture that will get damaged/White = Solid. Optional: Very subtle outlines of AO map for easier visualization what parts of texture get damaged.

Preview of example texture in-game.

Note about Payday 2 Model Tool

For GLTF/GLB format: New objects and models can include more UV layers.

- UV0 (Default "UVMap" in blender) will be used as the "pattern UV" if a second UVMap is not present.
- UV1 (Second listed UVMap in blender) will be used as the "pattern UV" for patterns and stickers when present.

For OBJ format: Not recommended to use but new objects or models are free of any problems with skins but in case of replacing existing objects in PD2 models remember to use "Pattern UV" option to prevent patterns being missing/broken in most cases.

Additional notes about UV coordinates

Some CC textures have parts that do not utilise the skin materials and will show their original textures (Example, Commando 101 rocket launcher), this effect can be achieved by moving the pattern UVs outside of the 0-1 region.

Animated Models

(WIP PAGE ON ANIMATED MODELS USED IN PAYDAY 2)

Page Notes:

- Animations missing X Y or Z rotations dont seem to work properly. (tested on 32-Bit Floats with Discard)
  - Animations missing X Y or Z loations work fine.
  - Animations using scale likely dont work.

Importing Animations

wip

Exporting Animations

wip

Building an animated model

wip

Object File Portion:

```
<!--Storing and Grouping Animations-->
<animations>
<animation_group name="animation_group_name">
<object name="anim_part_a" />
<object name="anim_part_b" />
</animation_group>
</animations>
```

```
<!--Collision/Hitboxes-->
<bodies>
<body name="body_anim_part_a" enabled="true" template="animated">
<object name="anim_part_a" />
<object name="c_c" collision_type="box" padding="-2.5"/>
</body>
</bodies>
```

Sequence Manager

`animation_group` is the type of sequence you want to play animations in the model.

- enabled: Is it playing or paused (true/false)
- name: The animation group name you want to use ("'string'")
- from: The start of where you are playing
- to: where you want to end (not using this will make it continue until the end of the animation)

Example:

```
<sequence editable_state="true" name="'play_animation'" triggable="true">
<animation_group enabled="true" name="'animation_group_name'" from="0/30" to="20/30"/>
</sequence>
```

Material Config XML

Example XML:

```
<materials version="3">
    <material name="mat_name" render_template="generic:DIFFUSE_TEXTURE:NORMALMAP" version="2">
        <diffuse_texture  file="texture/file/path/my_texture_df"/>
        <bump_normal_texture file="texture/file/path/my_texture_nm"/>
    </material>
</materials>
```

TEMPLATES ARE **NOT** MODULAR!
A full list of template variables can be found in `shaders/base.render_template_database`

- `<materials/>` contains all of the material xml for the file.
  - Typically uses `version="3"`
- `<material/>` contains the information of a material.
  - Typically uses `version="2"`
- `<diffuse_texture/>` a variable that often uses a texture with the identifier _df at the end.
- `<bump_normal_texture/>` a variable that often uses a texture with the identifier _nm at the end.
- `<material_texture/>` a variable that often uses a texture with the identifier _gsma at the end. (Gloss, Specular, Metalness, Alpha)
- `<reflection_texture/>` is a variable that loads a cubemap to a material that accepts cubemaps.
  - `<reflection_texture type="cubemap" global_texture="current_global_texture/>"` will use the Environment set cubemap.
  - `<reflection_texture type="cubemap" file="texture/file/path/my_cubemap"/>` will use a locked cubemap and wont change from environments.

Special Use Cases

- `<material/>` control options:
  - `name="mat_name"` give your material a name to go with your models material name.
  - `render_template="template_name"` a full list of templates can be found in `shaders/base.render_template_database`
  - `unique="true"` is usually used for materials that will change per-unit. Unit contours for example.
  - `src="name"` will clone the values from an existing `<material>` to your current material.

    - cloned materials can be modified further by including variables or textures to override the source copy.
  - `decal_material="id"` graphic mesh faces with the material will use the `decal_material` variable for hit effects.
    - List of decal IDs can be found in `lib\tweak_data\tweakdata.lua` `self.materials = {list}`
  - `diffuse_color="255 255 255 255"` potentially unused. (Red, Green, Blue, Alpha)
  - `version="2"` most if not all templates use version 2.
- `<variable/>` control options:
  - Animations can control some variables using the listener type.
    - Standard variable: `<variable name="il_tint" type="vector3" value="1 1 1"/>`
    - Listener variable: `<variable name="il_tint" type="listener" value="light::color" object="lo_lightobject"/>`

Object Xml

Example XML:

```
<dynamic_object>
	<diesel materials="units/path/material_config" orientation_object="rp_rootpoint_object" />
	<sequence_manager file="units/path/sequence_manager" />
	<bodies>
		<body name="body_static" enabled="true" template="static">
			<object name="c_collision" collision_type="box" padding="-2.5" />
		</body>
	</bodies>

	<decal_surfaces default_material="stone">
		<decal_mesh name="dm_decalmesh" enabled="true" material="steel" />
	</decal_surfaces>

	<graphics>
		<object name="g_graphics" enabled="true" shadow_caster="true" />
	</graphics>
</dynamic_object>
```

- `<diesel/>` is a required table that holds a path to the materials and root object.
  - `materials="path"` leads to the material config the unit will use.
  - `orientation_object="rp_rootpoint"` the object that acts as the origin of the unit, must be an Empty object.
- `<sequence_manager file="path"/>` is its own table defining the sequence manager the unit will use.
  - **Warning!** `<extension name="damage" class="UnitDamage" />` is required in the units extensions!
- `<bodies/>` is the table that holds collision information.
- `<body/>` is the table that holds the objects that will be used as collision.
  - `name` is the name of the body you are making.
  - `enabled` is a toggle for if collision should be enabled.
  - `template` is the physics template of the body.
    - `static` is solid collision.
    - `editor` is editor only collision.
    - a full list of templates can be found in `settings/physics_settings.physics_settings`.

    - templates can be edited in object by adding properties seem in the physics_settings file.
- `<object/>` is the table that holds the referenced object. (The same object can be used more than once)
    - `name` is the name of the object being used.
        - Using empties will bind the body to the empty for constraints.
    - `collision_type` is the shape/type of collision.
        - `box` is a box.
        - `sphere` is a sphere.
        - `capsule` is a pill shape.
        - `convex` is a convex shape based on your used object mesh.
        - `mesh_mopp` is a paper thin collision that is 1:1 the shape of your objects mesh.
    - `two_sided` is a toggle for if your collision should be double sided. (Very buggy!)
    - `padding` is the additional thickness in cm of your object.
        - Default 0 leaves models 2.5cm thicker, use -2.5 to correct it.
        - Does not apply to mesh_mopp collision.
- `<decal_surfaces/>` is the table that holds a list of meshes to act as decal surfaces. (Blankets that allow decals to spawn)
    - `default_material` is used if no decal material information is avaliable when hit.
- `<decal_mesh/>` is a table that holds a specific object to act as a decal surface.
    - `name` is the name of the object being used as the decal mesh.
    - `enabled` is a toggle for if decal_mesh should be enabled.
    - `material` to be used when shot/hit.
- `<graphics/>` is the table that holds all of the graphics objects.
    - `<object/>` is a table that is used to control the selected graphic.
        - `name` is the name of the object being used as the graphic.
        - `enabled` is a toggle for if the graphics should be visible.
        - `shadow_caster` is a toggle for if the graphics object should cast a shadow. (typically used by shadow_caster material objects. `s_shadowcaster`)
- `<constraints/>` is a table that holds all of the constraint information.
- `<constraint/>` is a table that constrains objects together.
    - `enabled` is a toggle for if the constraint should be active.
    - `type=""` sets the type of constrant.
        - `static`
        - `ragdoll`
    - `<param body_a="body_a" body_b="body_b"/>` Attaches A to B.
        - Attaching to world requires `@world` to be used as `body_a`.
    - `<param pivot="position:a_object"/>` Sets the position to pivot from.
    - `<param twist_axis="yaxis:a_object" twist_min="-10" twist_max="10" twist_freedom="1"/>` Sets the twist axis and control.
    - `<param plane_axis="xaxis:a_object"/>`
    - `<param cone_y="60" cone_z="60" cone_freedom="12"/>` Limit the angle the constrant can rotate.
    - `<param damping="8" spring_constant="600" min_restitution="0"/>` Spring settings.

Sequence Manager XML

Unit extension

```
<extension name="damage" class="UnitDamage" />
```

Include in object file

```
<sequence_manager file="path to your sequence manager" />
```

Example XML:

```
<table>
  <unit>
    <sequence editable_state="true/false" name="'" triggable="true/false">


    </sequence>
  </unit>
</table>
```

Important: Strings need to have `'` at the beginning and end, otherwise they do not work.

Sequences

- `<sequence/>`
  - `name` The name of your sequence. This will show up in the mesh variation dropdown and UnitSequence/UnitSequenceTrigger elements.
  - `editable_state` Makes the sequence show up in the mesh variation dropdown and UnitSequence element. *
  - `triggable` Makes the sequence show up in the UnitSequenceTrigger element *
    - Both editable_state and triggable are optional. Sequences will run and trigger without them, you just have to type in the sequence name manually into the elements.
    - ***That's probably how it's supposed to work, however testing showed that having either of them set to true will make it show in both UnitSequence and UnitSequenceTrigger, as well as the mesh variation dropdown.***

- `once` true / false | Sequence can only run once.
- `<object/>` Used to toggle individual graphic objects.
  - `name` Name of the object. Usually start with "g_".
  - `enabled` true / false
- `<graphic_group/>` Used to toggle graphic groups.
  - `name` Name of the graphic group.
  - `visibility` true / false
- `<body/>` Used to toggle bodies/collisions.
  - `name` Name of your body.
  - `enabled` true / false
- `<decal_mesh/>` Used to toggle decal meshes.
  - `name` Name of your decal mesh. Usually start with "dm_".
  - `enabled` true / false
- `<light/>` Used to toggle lights.
  - `name` Name of your light object.
  - `enabled` true / false
- `<material_config/>` Used to change the material config of your unit.
  - `name` path to the new .material_config you want to apply.
- `<sound/>` Used to play sounds from your unit.
  - `action` play / stop
  - `event` ID of the sound you want to play.
  - `object` Source object the sound is played from.
  - `source` Alternative to `object`, use a soundsource which has been defined in the .unit file.
- `<effect/>` Used to play effects from your unit.
  - `name` Path to the effect you want to play. Example: `name="'effects/particles/explosions/explosion_grenade'"`
  - `parent` object you want to play the effect from. Example: `parent="object( 'smoke' )"`
  - `position` (Not sure what it does exactly, every effect I found had `position="v()"` in it too)
- `<animation_group/>` Plays an animation.
  - `enabled` true / false
  - `name` Name of your animation group.
  - `from` The frame that this animation should start on. Example: `from="0/30` will make the animation play at 30fps.
  - `to` End frame of your animation. Example: `to="64/30`
  - `speed` How fast your animation is playing. 1 is normal speed, can be a negative value to play backwards.
- `<run_sequence/>` Used to run another sequence.
  - `name` The name of the sequence you want to run.
- `<spawn_unit/>` Spawns a new unit.
  - `name` Path to the unit you want to spawn.
  - `position` Object position, usually an empty, that the unit will be spawned on. Example: `position="object_pos('spawn_doors')"`

    - `rotation` Same as position, but for rotation. Example: `rotation="object_rot('spawn_doors')"`
  - `<interaction/>` Toggle interactions on your unit.
    - `enabled` true / false

    - `start_time` can be used on pretty much everything to add delays.
    - `startup` true / false

Hitboxes

You can take any body from your .object and use it as a hitbox to trigger sequences.

```
<table>
  <unit>
    <body name="''">
      <endurance>
        <run_sequence name="''"/>
      </endurance>
    </body>
  </unit>
</table>
```

- `<body/>` This defines a body from your .object as a hitbox.
  - `name` The name of the body.
  - `<endurance/>` Controls how much damage the hitbox can take before it triggers the sequences inside.
    - `bullet` How many times you need to shoot it.
    - `explosions` How many explosions you need.
    - `melee` How many melee hits you need.
    - `lock` Doors, deposit boxes and basically anything you can use a saw on use this. (For Example: `lock="15"` on deposit boxes.)

Filters and Variables

You can define variables and filter sequences based on the value of a variable.

```
<table>
  <unit>
    <variables>
    <your_variable_name value="#"/>
    </variables>
```

```
    <filter name="'your_filter_name'">
      <check value="vars.your_variable_name == #"/>
    </filter>
  </unit>
</table>
```

Add `filter="your_filter_name"` to a sequence to make it only run if the variable and the filter have the same value.

Use `<set_variables your_variable_name="#"/>` to change variable values.

Example:

```
<table>
  <unit>
    <variables>
    <var_loot_type value="0"/>
    </variables>

    <filter name="'loot_money'">
      <check value="vars.var_loot_type == 1"/>
    </filter>

    <filter name="'loot_gold'">
      <check value="vars.var_loot_type == 2"/>
    </filter>

    <sequence editable_state="true" name="'set_loot_money'" triggable="true">
      <set_variables var_loot_type="1"/>
    </sequence>
    <sequence editable_state="true" name="'set_loot_gold'" triggable="true">
      <set_variables var_loot_type="2"/>
    </sequence>

    <sequence editable_state="true" name="'spawn_loot'" triggable="true">
      <spawn_unit filter="'loot_money'"
name="'units/pd2_dlc1/vehicles/str_vehicle_truck_gensec_transport/spawn_deposit/spawn_money'".
../>
      <spawn_unit filter="'loot_gold'"
name="'units/pd2_dlc1/vehicles/str_vehicle_truck_gensec_transport/spawn_deposit/spawn_gold'"..
```

```
./>
    </sequence>

  </unit>
</table>
```

To be continued...

(old notes from rex)

Vector3 form in sequence manager: `v(0, 0, 0)`

MaterialElement

- `<material/>` material sequences affect materials of the functioning unit. `Unique="true"` is required to only affect this unit.
  - `name` is the name of the material you are modifying.
  - `time` set the time of animated materials, Joys mask uses this feature.
  - `state` sets a state argument, Joys mask uses this feature to pause the UV scrolling.
  - `render_template` sets the render template.
  - `glossiness` sets the glossiness value.
  - If you use a custom `key="value"` you can affect material config elements on your own.
    - Example: `<material name="'mat_mat'" il_multiplier="10"/>`
    - This will modify the il_multiplier of an illuminated material.
    - Vector3 values need to be written like so `key="v(1, 2, 3)"`

---

pick random sequence thingy `<run_sequence name="'sequence_'..pick('1','2','3')"`

Animation State Machine

While baking animations inside `.model` files might be enough for simple, continuous animations*, when it comes to actually having complex movement occur (such as a randomized escort path, or a helicopter that simultaneously hovers and shoots missiles, like in Hotline Miami day 2), you will want to make use of the **Animation State Machine** (will be referred to as ASM).

For any unit to make use of the ASM, it will require four additional files with the following extensions:

- `animation_state_machine`
- `animation_states`
- `animation_def`
- `animation_subset`

** There are some exceptions to this, as in, for instance, car door opening/closing animations, which are actually a single animation where certain ranges of keyframes correspond to the animation. The keyframes that play can be specified in `animation_group` sequences; see Sequence Manager XML.*

But what exactly is a "state"?

An animation state is a way to control **what** action an unit is doing, what happens **during** it, and what it does **when it finishes**. Take, for instance, a `fps/melee_hit` state; the animation is determined through *weights*, sounds are played accordingly, and, when it finishes, it exits to the `fps/idle` state, that is, the "weapon idle" FPS animation. Compare this to a baked animation, where you can only start it, pause it, and be forced to time your sound design around the finished animation instead of using *animation markers*.

Initial `.unit` setup

The `animation_state_machine` and `animation_def` files **need** to be directly referenced by the `.unit` file:

```
<unit type="str" slot="1">
  ... network syncs ...
  <anim_state_machine name="anims/units/helicopter/helicopter" />
```

```
  <dependencies>
    <depends_on animation_def="anims/units/helicopter/helicopter"/>
    <depends_on animation_state_machine="anims/units/helicopter/helicopter"/>
  </dependencies>

  ... the rest of the unit definition ...
</unit>
```

Initial `.object` setup

```
<dynamic_object>
  ... material_config and sequence_manager ...
  <animation_def name="path/to/animation/definition" />
  ... the rest of the object definition ...
</dynamic_object>
```

Additionally, make sure that animated collisions follow the `animated` template, otherwise they will not move.

Files

`animation_state_machine`

This file needs to be referenced inside the unit's `.unit` file. Let's break down a sample `animation_state_machine` file. *Note that the purpose of some attributes is currently unknown.*

```
<state_machine name="a_cool_name_for_the_asm" timebase="frames">
  <segment name="base"/>
  <segment name="secondary_actions"/>

  <global name="a_variable_name" value="0"/>

  <default state="std/empty"/>
□
  <states file="path/to/the/states/file"/>
</state_machine>
```

- **Segments** are used to allow for multiple states to be active at once; say, a medic can be crouching while playing the upper body "heal" animation. Every state will need to be assigned to a segment, and only one state can play at once per segment. **HOWEVER**, you can play multiple animations together from separate states in a single segment by using blend sets; see the section on `animation_def`.
- Globals allow for **weights**: a single state can have multiple animations (if it is of the "mix" type), however, it will only play the animation with the highest weight value. These can be controlled through scripting (see the corresponding section).
- A **default empty state**, with no animations, is set when the unit is loaded. This is standard on any `animation_state_machine` file unless you want the unit to start doing stuff as soon as it loads. If a invalid state is specified, no other states will work.
- An **`animation_states`** file is defined. *There exists the possibility of defining the states directly in this file; however, it is recommended to still create individual files.*

`animation_states`

This file needs to be referenced inside the `animation_state_machine` file. It contains all of the playable states for the ASM. A valid animation state file resembles the following:

```
<xml>
  <state name="std" type="template_state" segment="a_segment_for_the_state"
mute="an_optional_segment_to_mute">
    <to name="a_generic_name_for_this_animation" redirect="name/of/the/corresponding/state"/>
  </state>

  <state name="std/empty" type="emptyloop" segment="a_segment_for_the_state">
  </state>

  ... ALL OTHER STATES ...
</xml>
```

- A state of the type `template_state`: here, you define **the names of the animation redirects to call from the sequence manager**, that is, any name of your choice, and a reference to the state that will be played, for each state. **You can't trigger states from the sequence manager if you use their `name/formatted/like/this` directly**.
- The default empty state we specified in the `animation_state_machine` file.

State types

- **`template_state`**: a state that serves as a property template for other states under the hierarchy. See the section on state groups. **For it to apply to children states, it needs to be defined before them.**
- **`emptyloop`**: a state that does not play an animation until it is overriden by another state.

    - A special blending effect occurs when entering a state of this type when using multiple segments; see the `from` action description below.
- `once`: a state that plays a single, **pre-determined** animation, once. If given multiple animations, it will always pick the same one, regardless of weights. For the correct behaviour in that case, use `mix`.
- `loop`: a state that plays a single, **pre-determined** animation, repeatedly, until another state is triggered.
- `mix`: a state that plays the animation with the most weight from a list of possible animations, once.
- `mixloop`: a state that plays the animation with the most weight from a list of possible animations, repeatedly, until another state is triggered.
- `timeblend`: a state that "plays" the position of the bones defined in its animation at a time `t`, which is an implicit parameter of the state, settable from Lua. The bones won't move from that position until another state is played or this one is re-entered with another `t` value. When using this state, animations will need to define a `time=` parameter alongside them for reference purposes. This state also allows weights. E.g. `<anim name="falcogini_wheel_turn_left" time="1" weight="falcogini"/>`:
    - When `t = 0`, the state will place the bones at their **starting** position in the animation.
    - When `t = 1` (the value specified in the XML), the state will place the bones at their **ending** position in the animation.
    - When `0 < t < 1`, the state will place the bone at that position in the animation given the reference `time` parameter.
- `snapshot`: unknown. Used once in `core/units/locator/locator.animation_state_machine`.

The following states are referenced in engine decompilations, but never used in the game:

- `sequence`: untested, but should play its animations in a sequence as specified.
- `empty`: unknown.
- `offset_once`: unknown.
- `offset_loop`: unknown.

State actions

- `to`: define a redirect to a state. E.g. `<to name="reload_exit" redirect="fps/reload/r870_shotgun/exit"/>`
- `anim`: play an animation with the specified name, which should have been defined in the `animation_subset` files. If the state is of the type `mix` or `mixloop`, you also need to include a `weight=` parameter. E.g. `<anim name="throw_concussion" weight="projectile_concussion"/>`
- `exit`: when the animation that the state played finishes, enter the named state. This will never trigger on `loop` and `mixloop` states. E.g. `<exit name="fps/idle"/>`. *If exiting a state in a particular segment to a state in another segment, the second state will break.*
- `block`: Block the specified state from triggering during this state. E.g. `<block name="upper_body/recoil/crouch/auto_exit"/>`
- `unblock`
- `block_group`
- `unblock_group`

- `from`: blend the specified "amount" between the animation that the specified state was playing when exited and the animation that will now be played by this state. E.g. `<from name="fps/idle" blend="1"/>`.
  - *`from` only works for same-segment state switching; to blend finished animations on a segment with playing animations on another segment, exit on the finished segment to an emptyloop state on that same segment which uses `from`. See the video resource on blending for more info.*
- `from_group`: same as with `from`, applying instead to an entire group.
- `keys`: a table of `key` elements. More information below.
- `param`: similar to `animation_state_machine` globals, except these are defined and stored on a per-state basis. They can also be used to set weights, just like globals, and controlled through Lua. E.g. `<param name="generic_stance" value="0"/>`
- `default`: set default properties for states under the hierarchy. E.g. `<default blend="1.5" blend_out="1.5"/>`.
  - `blend`: amount of blend-in; see the video resource on blending for more info.
  - `blend_out`: unknown... pretty sure that it doesn't do what it's supposed to. If you want to actually blend out, use `from_group`.
- `modifier`: Enable the specified modifier. E.g. `<modifier name="ik" blend="1" blend_out="1"/>`.
- `remove_modifier`

State groups

You might have noticed that state names follow `a/structure/like/this`. Think of them as paths; this naming scheme allows for states deeper in the tree to inherit nodes from **template states** higher up.

Some examples of how you might use this in a state structure:

- `std` is a template state, and has a `from_group` and `exit` node for proper blending and clearing.
- `std/death`, another template state, will be interpreted as if the `from_group` and `exit` nodes were defined inside it; additionally, it has keys for toggling an `animation_is_playing` value in an extension class.
- `std/death/animation_1` will play as if the `from_group`, `exit` and keys were defined explicitly in it.

Keys

What should happen at certain points in the state. There are two types of keys, callbacks and key setters:

```
<key at="key_type" callback="the_method_of_that_extension_class_to_call"
class_name="the_name_of_the_extension_class" param1="first_param" param2="second_param"
param3="moar_params" param4="you_get_the_point"/>
```

The former will result in the callback function being called as `callback_function(param1, param2, param3, param4, ...)` when the condition specified in `at` is met. Parameters are optional.

```
<key at="key_type" set_extension="the_name_of_the_extension_class"
extension_key="the_value_of_that_extension_class_to_set" extension_value="value"/>
```

The former will make the key `extension_key` have the value `extension_value` inside the extension class `extension_class`.

For the `at` parameter, you can use:

- **`enter`**: when the state is entered.
- **`exit`**: when the state is exited, **not when the animation finishes**.
- **`loop`**: when the state loops. This will only trigger on `loop` and `mixloop`.
- **`trigger`**: when the animation reaches a certain marker. E.g. `<key at="trigger" trigger="wp_g3_grab_end" callback="play_sound" class_name="base" param1="wp_g3_grab_end"/>`
- **`full_blend`**: when a blend-in finishes. Does not work on emptyloop states.
- *Any number*: when the animation is at that keyframe.

Example

A base-game example of a state, with comments indicating what its doing. This particular one is the state for reloading the Reinfeld 880:

```
<state name="fps/reload/r870_shotgun/loop" type="loop" segment="base" speed="1.0"> <!-- Note
the possibility of changing the speed of animations here -->
  <!-- Set redirects for stopping the reload shell insert loop. These would be called from Lua
in this case. -->
  <to name="reload_exit" redirect="fps/reload/r870_shotgun/exit"/>
  <to name="reload_not_empty_exit" redirect="fps/reload/r870_shotgun/not_empty_exit"/>

  <!-- Play the animation -->
  <anim name="r870_shotgun_reload_loop"/>

  <!-- Do stuff at certain markers/events -->
  <keys>
      <key at="enter" callback="anim_clbk_spawn_shotgun_shell" class_name="base"/>
      <key at="loop" callback="anim_clbk_spawn_shotgun_shell" class_name="base"/>
      <key at="enter" callback="enter_shotgun_reload_loop" class_name="base"
param1="fps/reload/r870_shotgun/loop"/>
      <key at="trigger" trigger="wp_reinbeck_reload_cock" callback="play_sound"
class_name="base" param1="wp_reinbeck_reload_cock"/>
```

```
      <key at="trigger" trigger="wp_reinbeck_shell_insert" callback="play_sound"
class_name="base" param1="wp_reinbeck_shell_insert"/>
      <key at="trigger" trigger="wp_foley_generic_lever_pull" callback="play_sound"
class_name="base" param1="wp_foley_generic_lever_pull"/>
      <key at="trigger" trigger="wp_foley_generic_back_in_hand" callback="play_sound"
class_name="base" param1="wp_foley_generic_back_in_hand"/>
      <key at="trigger" trigger="anim_act_01" callback="anim_clbk_unspawn_shotgun_shell"
class_name="base"/>
      <key at="exit" callback="anim_clbk_unspawn_shotgun_shell" class_name="base"/>
  </keys>
</state>
```

`animation_def`

This file needs to be referenced inside the unit's `.object` file.

The following is a simplified version of `anims/units/enemies/cop/cop_def.animation_def`, which I reckon showcases this file well.

```
<xml>
  <!-- Animatable sets -->
  <animatable_set name="cop">
      <!-- Root -->
      <bone name="root_point" root="true" alignment="true" alias="all root"/>

      <!-- Spine -->
      <bone name="Hips" alias="all legs"/>
      <bone name="Spine" alias="all upper upper_l upper_r lod"/>
      <bone name="Spine1" alias="all upper upper_l upper_r lod"/>
      <bone name="Spine2" alias="all upper upper_l upper_r lod"/>

      <!-- Left leg -->
      <bone name="LeftUpLeg" alias="all legs lod"/>
      <bone name="LeftLeg" alias="all legs lod"/>
      <bone name="LeftFoot" alias="all legs lod"/>

      <!-- Right leg -->
      <bone name="RightUpLeg" alias="all legs lod"/>
      <bone name="RightLeg" alias="all legs lod"/>
      <bone name="RightFoot" alias="all legs lod"/>
```

```
  </animatable_set>

  <!-- Blend sets -->
  <blend_set name="all" animatable_set="cop">
      <blend alias="all" weight="1.0"/>
  </blend_set>

  <blend_set name="upper_body" animatable_set="cop">
      <blend alias="all" weight="1"/>
      <blend name="Spine2" weight="0.85"/>
      <blend name="Spine1" weight="0.4"/>
      <blend name="Spine" weight="0.25"/>
      <blend alias="legs" weight="0"/>
      <blend alias="root" weight="0"/>
  </blend_set>

  <!-- IK Modifiers -->
  <modifier name="look_upper_body" type="ik" animatable_set="cop" iterations="1"
blend_in="0.5" blend_out="0.3">
      ... modifier params ...
  </modifier>

  <!-- Animation sets -->
  <animation_set name="cop" animatable_set="cop">
      <subset file="path/to/the/subset/file"/>
      <subset file="any/other/subset/file"/>
      <subset file="you/can/have/as/many/as/you/want"/>
  </animation_set>
</xml>
```

- Animatable sets are where you **directly reference the bones in your model**. You can also give them aliases, separated by spaces, in the `alias` key. Take `root_point`, which not only is declared as the root bone, but also assigned both `all` and `root` aliases.

For every animatable set you will need:

- Blend sets, which define how two animations **playing at once** blend together. This does not refer to switching between animations, but rather, how much weight the animations have over the bones; every animation needs to be specified a blend set to use. *See the video resource on blending for more info.*

    - When you don't want to blend (that is, play your animation with full bone influence), you can use an "all" set which applies a weight of 1 to all the bones. You use these in your animation subset.
- Animation sets, which **need to define one or more `animation_subset` files.**

Make sure that when you name your animatable set and your animation set, they don't match a segment's name! Otherwise, you'll crash when changing states.

Modifiers

For procedural animation, such as those seen with VR arm movements, or peer upper body rotation when they move their camera. **They don't work on empty states.**

Types: group, stand, **ik**, shoulder, displacement, offset, link, mirror.

The IK type requires either a `rotation` or a `position` node. These can be of type `script`, `lock` or `animation` (this last one might be of interest for animated IK, however, I haven't managed to make it work);

- When using `script`, they are controllable from Lua.
- When using `animation`, the `rotation/position` node also needs to include a `target` attribute, otherwise the game will crash when enabling the modifier. `rotation` can also use an `axis` (x, y or z) attribute when set to this type.

```
<!-- The modifier in charge of keeping up with a peer's camera direction and updating their ;
it's controlled through HuskPlayerMovement -->
<modifier name="action_upper_body" type="ik" animatable_set="cop" iterations="1"
blend_in="0.5" blend_out="0.3">
    <rotation type="script"/>
    <target name="aim"/> <!-- This bone needs to exist in the model, otherwise crash. -->
    <!--          ^^ Also of note is that this bone needs to be parented to whatever bones it
will move as a result of its transformations -->

    <!-- IK bone chain -->
    <bone name="Spine2" rotate="1"/>
    <bone name="Spine1" rotate="0.35"/>
    <bone name="Spine" rotate="0.35"/>
</modifier>
```

The mirror type requires a `config` attribute which can only be `biped`. For each of its entries, it needs a `first` and a `second` attribute, with an optional `transform` attribute which can either be `inverse` or `root`.

`animation_subset`

This file needs to be referenced inside the `animation_def` file. It's arguably the easiest to get a grip on.

```
<xml>
  <anim name="referencable_anim_name" file="path/to/the/animation/file" blend_set="all"/>
</xml>
```

- For each animation file, you define an animation name and a blend set to use. **This animation name is what animation states reference to play the animation file.**

The folks that worked with these files actually left us some extra info on making animations:

`If the animation finishes inside the viewable area and does not finish outside of the viewable area, you must animate the root point. Or else the animation will not sync at drop in. For eample the police car arrives and stop in the viewable area, here the root point shoul be animated. The heicopter arrives and leaves the the viewable area, here all_no_root is allowed.`

Make sure to follow that guideline in order to avoid sync issues!

Controlling animations through Sequence Managers

Use `animation_redirect`. Example:

```
<sequence name=" 'heli_suburbia_hover' " triggable="true">
    <animation_redirect name=" 'suburbia_hover' "/>
</sequence>
```

Remember that this is the name you specify in `to` state actions that redirects you to a state.

Video resources

Animation Blending

Common crash-causers

When working with these files, it is especially frustrating to try to track down the source of access violations. Below is a list of issues to check:

- Paths are misspelled on any of the files
- An XML structure was not properly closed
- `animation_def` or `animation_state_machine` are missing from the unit's dependencies list
- `animation_def` was not defined in the `object` file
- A bone specified in `animation_def` is not actually a bone, or is the root bone and does not exist
- An animation was specified in an `emptyloop` state
- An animation specified in a state does not exist

If none of these solved your issue, read this page carefully again, from top to bottom.

Appendix A: Lua methods

Animation State Machine

The ASM of an unit can be referenced through the unit's `anim_state_machine()` method.

- `ASM:set_global(name, value)`: sets the global to the specified **number** value.
- `ASM:set_global_soft(name, value)`
- `ASM:get_global(name)`: get the value of the global.
- `ASM:set_parameter(state, parameter, value)`: Set the parameter inside the specified state to a **number** value.
- `ASM:set_parameter_soft(state, parameter, value)`
- `ASM:set_speed(animation_idstring, speed)`: sets the speed of the animation.
- `ASM:set_speed_soft(animation_idstring, speed)`
- `ASM:get_speed(state)`: get the speed of the state.
- `ASM:is_playing(animation_idstring)`: return whether the animation is playing or not.
- `ASM:segment_real_time(segment_idstring)`: return the current time of the segment.
- `ASM:segment_relative_time(segment_idstring)`
- `ASM:segment_muted(segment_idstring)`
- `ASM:stop_segment(segment_idstring)`
- `ASM:segment_state(segment_idstring)`: get the playing state on the specified segment.
- `ASM:segment_state_object(segment_idstring)`
- `ASM:config():states()`: return a list of states in the ASM.
  - `State:name():s()`: return the state name.
- `ASM:state_in_group(state)`
- `ASM:set_enabled(enabled)`
- `ASM:set_animation_time_all_segments(time)`
- `ASM:play_raw(animation)`
- `ASM:play(animation)`
- `ASM:root_blending()`
- `ASM:set_root_blending(bool)`
- `ASM:instant_blending()`

- `ASM:set_instant_blending(bool)`
- `ASM:set_callback_object(lua_object)`
- `ASM:index_to_state_name(index)`
- `ASM:state_name_to_index(state)`
- `ASM:enabled_modifiers()`
- `ASM:force_modifier(modifier)`
- `ASM:allow_modifier(modifier)`
- `ASM:forbid_modifier(modifier)`
- `ASM:get_modifier(modifier)`
- `ASM:set_modifier_blend(modifier, blend)`
- `ASM:reset()`

To play a ASM state, use the unit's `play_redirect(redirect_name, time_offset)` method. This returns an Idstring identifier for the animation. Alternatively, you can also use the unit's `play_state(state_idstring, time_offset)`.

Modifiers

IK

IK modifiers can be accessed through `ASM:get_modifier(idstring_modifier_name)`. If the rotation/position type is `script`, they are controllable through:

Directly set the target bone's position/rotation:

- `IKModifierInstance:set_target_position(vector3)`
- `IKModifierInstance:set_target_rotation(rotation)`

Set the rotation of the target bone by making one of its axis have the direction of the provided **normalized** vector, taking the bone's position as the point of reference:

- `IKModifierInstance:set_target_x(vector3)`
- `IKModifierInstance:set_target_y(vector3)`
- `IKModifierInstance:set_target_z(vector3)`

Set the rotation of the target bone by making one of its axis point towards the provided point in world space:

- `IKModifierInstance:set_target_x_look_at(vector3)`
- `IKModifierInstance:set_target_y_look_at(vector3)`
- `IKModifierInstance:set_target_z_look_at(vector3)`

Unknown:

- `IKModifierInstance:set_object(?)`
- `IKModifierInstance:blend()`