Deviant Pictures Films

Grigory Revzin
Alex Portman

A Guide to HWM Elizabeth Source Files

with a brief HWM editing introduction

Model Version 1.5

PDF timestamp:

Wednesday 19th February, 2014

This is a work-in-progress PDF and it will change over time.

Russia, 2014

Contents

Typeset with MiKTEX

Legal disclamer

Elizabeth is a character from Bioshock: Infinite, written, voiced, modeled, textured etc. by Irrational Games. We, Grigory Revzin and Alex Portman (Vechyaslav Alexeev), do not claim her image/model/textures/voice lines/whatever else as our IP (we aren't that stupid or bitchy).

We declare our work on her, namely preparing her model assets for use in Source Filmmaker & preparing an advanced facial rig, a form of fan art (not really different from drawing a 2D model sheet to help out 2D artists on DeviantArt). We do not financially profit from this work, we aren't selling anything, and all this stuff was done out of our love for the fine game and the fine character.

And no, we aren't the guys behind the pornographic Source Filmmaker GIFs. The Elizabeth model described here wasn't technically available to any porn makers at the time of, um, porn making. Please don't blame us, Mr Levine.

Useful links

Blender Source Tools (*Blender SMD Tools*)	`http://steamreview.org/BlenderSourceTools/`
DMX file format	`http://developer.valvesoftware.com/wiki/DMX`
DMX model format	`http://developer.valvesoftware.com/wiki/DMX_model`
`studiomdl` model compiler	`http://developer.valvesoftware.com/wiki/Studiomdl`
VMT, VTF and Source materials	`http://developer.valvesoftware.com/wiki/Material_System`
QC files	`http://developer.valvesoftware.com/wiki/QC_File`
Procedural bones	`http://developer.valvesoftware.com/wiki/$proceduralbones`
DMX vertex animation	`http://developer.valvesoftware.com/wiki/Flex_animation#DMX_format`

1 Credits

Irrational Games The original game model and character, duh

Grigory Revzin All 3D, programming and documentation work

Alex Portman Materials and textures

Dmitry Norpo, Ilya Trofimov, Nikita Plyushov Testing

Figure 1: An appropriate still from **Magic Shmagic**

We thank **Tom "Artfunkel" Edwards** for the brilliant **Blender Source Tools** which proved themselves by handling five hi-poly HWM models; huge thanks to **Konstantin Nosov** aka **Gildor** for the **umodel** extractor; to **Grub** from `Facepunch.com` for general assistance and support; **theoneman** from `Facepunch.com` for providing some of the raw meshes.

Many thanks to **Callegos Yavolitak**, who've made a very atmospheric **Magic Shmagic** video starring this model.

Special thanks to

Irrational Games & Mr Ken Levine,

of course. *If it wasn't for their smart and affective writing, thoughtful and brilliant art, spectacular and refined animation and amazing voice acting,* ~~we would be porting dogs from Call of Duty.~~

And

Valve Software,

naturally.

We used **Blender** & **Adobe Photoshop** with **VTF plugins**. Huge props to everyone behind these.

2 A brief HWM introduction

HWM is a Valve facial rig standard which is used through post-2006 Valve games and engine versions, including our beloved Source Filmmaker. The first title featuring it was Team Fortress 2 which acts as 6 years didn't pass at all.

Very misleadingly, HWM expands to *Hardware Morphs*, which seemingly reflects the fact the mesh' vertices' positions are computed on your GPU — you won't believe how fast this thing adds `float3`s together.

Hardware usage isn't the only feature though. HWM introduces five interesting things:

1. Easy control grouping
2. Corrective shapes
3. Domination rules
4. Wrinkle maps (though Liz doesn't have one :( )
5. Stereo controls

Let's go over these.

2.1 Control grouping

HWM (and Source in general) distinguishes between *shapes*[1] and *controllers*, or *sliders*. Controllers are the things you pull in `hlmv`, `hlfaceposer`, Garry's Mod and Source Filmmaker. When a controller is pulled, it fades in a certain *shape*.

This is, of course, no different from the SMD/VTA system where one would open the QC file and then declare the shape:

```
flexfile "myflexanim.vta" {
    defaultflex frame 0
    flex "LipsOpened" frame 1 // Declaring the shape
                              // and naming it "LipsOpened"
    flex "OpenJaw" frame 2
}
```

then describe a flex controller:

```
flexcontroller Mouth "OpenLips"
```

and finally map `LipsOpened` to `OpenLips`:

```
%LipsOpened = OpenLips
```

In HWM, the same thing is done with embedding special data in the model's source DMX files.

DMX files are a topic on their own, unfortunately. They're a powerful container, which, in case of models, contains the mesh, the shapes and the controllers[2]. What matters to us is that Blender Source Tools allow the user to hand-edit this special data in text editing programs (for example, in a famed Notepad competitor called **Blender**) and embed it in the exported DMX. This data looks a little more complicated:

[1] By "shapes" I mean this thing that's called **Shape Keys** in Blender, **Morph Targets** in 3DS Max and **Blendshapes** in Maya.

[2] Previously these were stored in an SMD, a VTA and a QC respectively — things really did improve!

```
1  "DmeCombinationInputControl"
2  {
3      "id" "elementid" "4c4b3440-d611-da4a-beeb-90877351fb02"
4      "name" "string" "OuterSquint"
5      "rawControlNames" "string_array"
6              [
7                      "OuterSquint"
8              ]
9      "stereo" "bool" "1"
10     "eyelid" "bool" "0"
11     "wrinkleScales" "float_array"
12     [
13         "-2"
14     ]
15 }
```

But never trust the looks. These actually are a little simpler than their SMD/VTA analogue. Let's break down this snippet: we

declare a controller (line 1),

assign a unique ID to it (line 3)[3],

assign a name to it — this name will show up in SFM, `hlfaceposer`, etc (line 4),

specify the shape that will be driven by the controller (line 7),

mark the controller as a stereo controller — it will use the **SFM L/R slider** (line 9)[4],

specify the controller isn't an eyelid (line 10),

and finally, tell Source that we want it to draw a compressed wrinklemap when the `OuterSquint` shape is active (line 13).

The reason this data looks so complicated is that it wasn't meant to be edited by hand but rather by a utility program `dmxedit`. For various technical reasons reasons, it's not really suitable for Blender modeling in its current state so we'll have to edit these by hand for some time.

The really powerful part is that you can easily **group controls** by specifying more `rawControlNames`:

```
1  "DmeCombinationInputControl"
2  {
3          "id" "elementid" "88594ccd-99ce-7148-bc10-507844bdb1ae"
4          "name" "string" "LipsV"
5          "rawControlNames" "string_array"
6          [
7                  "CompressLips",
8                  "OpenLips"
9          ]
10         "stereo" "bool" "0"
11         "eyelid" "bool" "0"
12         "wrinkleScales" "float_array"
13         [
14                 "0",
15                 "0"
16         ]
17 }
```

[3] Unique IDs are technical fields required by the DMX files.

[4] **Do not** enable on jaw controls.

Now the *controller/slider* `LipsV` will fade in the `CompressLips` *shape* when pulled left and the `OpenLips` *shape* when pulled right (fig. 2).

Figure 2: A high-class noire mouth with `LipsV` pushed to the left, exhibiting `CompressLips` (*left third*); `LipsV` in a neutral position (*middle third*), exhibiting the rest shape and `LipsV` pushed to the far right, exhibiting `OpenLips` (*right third*).

You can add as many shapes to a slider as you like. For example, this is how smiling is handled with TF2 mercs:

```
"DmeCombinationInputControl"
{
	"id" "elementid" "2f96be1a-73c9-1c40-af37-7fcc1d2c21aa"
	"name" "string" "Smile"
	"rawControlNames" "string_array"
	[
		"SmileFlat",
		"SmileFull",
		"SmileSharp"
	]
	"stereo" "bool" "1"
	"eyelid" "bool" "0"
	"wrinkleScales" "float_array"
	[
		"1",
		"1",
		"1"
	]
}
```

If there're more than 2 shapes on a controller, Source will *automatically give you an extra controller* called "`multi_YourControllerName`" to fade between the three shapes, while controlling the intensity will still be done through the main controller. And yes, this will transparently support stereo controls (fig. 3).

Eyelids. As you see, you can declare the controller being an eyelid by setting the `"eyelid"` to `"1"`.

If the controller is an eyelid, the `_multi` slider will appear even on a two-way controller. Instead of fading through the shapes, `_multi` will set the convergence point for the shapes' intensity.

For example, if the `CloseLid_multi` slider is at 0.5, then the `CloseLidUp` and `CloseLidLo` shapes will fade in with the intensity of 0.5 when `CloseLid` slider is 1. If `CloseLid_multi` =

Figure 3: Triple-shape `Smile` controller setup

0.8 & `CloseLid` = 1, then the shapes' intensities will be `CloseLidUp` = 0.8 and `CloseLidLo` = 0.2. This is how eyelids are done on TF2 characters and on Liz.

All in all, isn't this just plain better than writing complicated math in QC files?

```
// A nice excerpt from \humans_sdk\facerules_xsi.qci
%upper_right_raiser = right_lid_raiser * (1 - right_lid_droop * 0.8) * (1 - right_lid_closer) * (1 - blink)
%upper_right_neutral = (1 - right_lid_droop * 0.8) * (1 - right_lid_raiser) * (1 - right_lid_closer) * (1 - blink)
%upper_right_lowerer = right_lid_closer + blink * (1 - right_lid_closer)
%upper_left_raiser = left_lid_raiser * (1 - left_lid_droop * 0.8) * (1 - left_lid_closer) * (1 - blink)
%upper_left_neutral = (1 - left_lid_droop * 0.8) * (1 - left_lid_raiser) * (1 - left_lid_closer) * (1 - blink)
%upper_left_lowerer = left_lid_closer + blink * (1 - left_lid_closer)
```

Good-bye, confusing QC shit, and don't come back — just die in peace.

Embedding controllers. When working on a human-like model, your best bet is to use the controllers from the TF2 models source files, available in `sourcesdk_content`. Gollum-creating-ly cool guy Bay Raitt, who apparently designed them, created a very solid system that doesn't need to be modified without any kind of a special cause. To embed a controllers block, use the **Advanced** flex mode in Blender Source Tools, like on fig. 4 (observe the controllers file being hand edited on the left part!).

You can specify any DMX HWM model file as a source of these controllers. Ideally, you'd want to have your own copy of the controller file. Blender Source Tools ignore everything that isn't the controller block while embedding so you can delete all the unnecessary stuff (which will take 99% of the poor file — who knew storing large geometry as plain-text float arrays ain't that compact).

Elizabeth, for example, has her controllers in the `head_controllers.dmx` file in the root folder. In addition to the standard TF2 controllers, `EyesWideOpen`, `EyesSealAttn` & `PuckerSizeAdj` are added. You can use this file as a template for your model's controllers. Another good template is in the HWM Femscout's source files, available from `source.maxofs2d.net`.

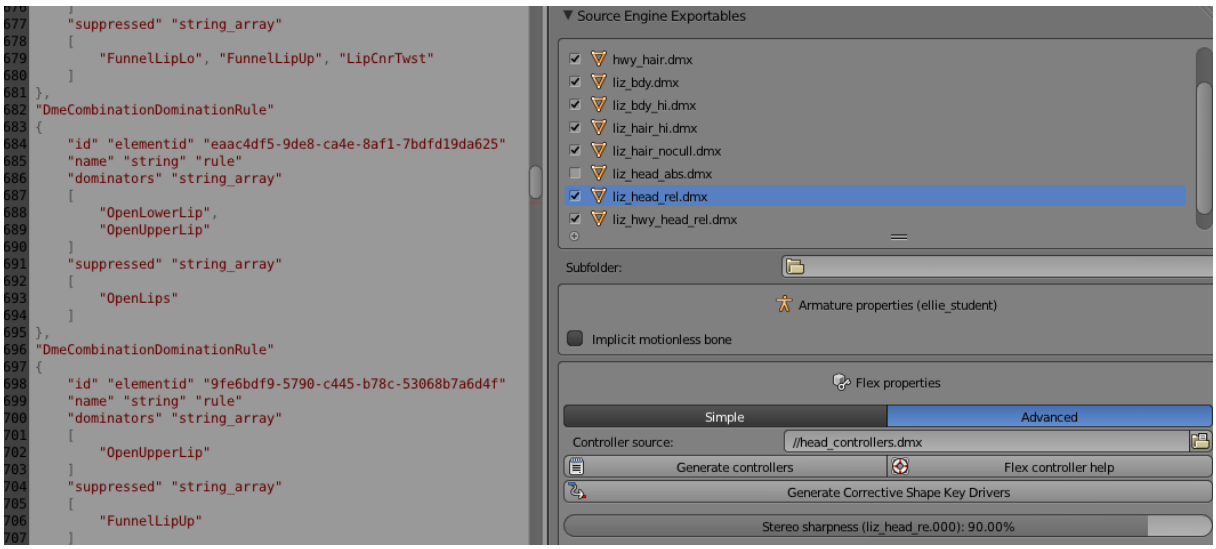

Figure 4: Choosing the controller source

! If hand-editing controller blocks (and any other DMX file for that matter), always use a unique UUID in the `id` field or the file won't work.

You can use the Blender console to generate these IDs. Type `import uuid` and `uuid.uuid4()` in the Blender console and it will print a nice universally unique ID for you to use (fig. 5).

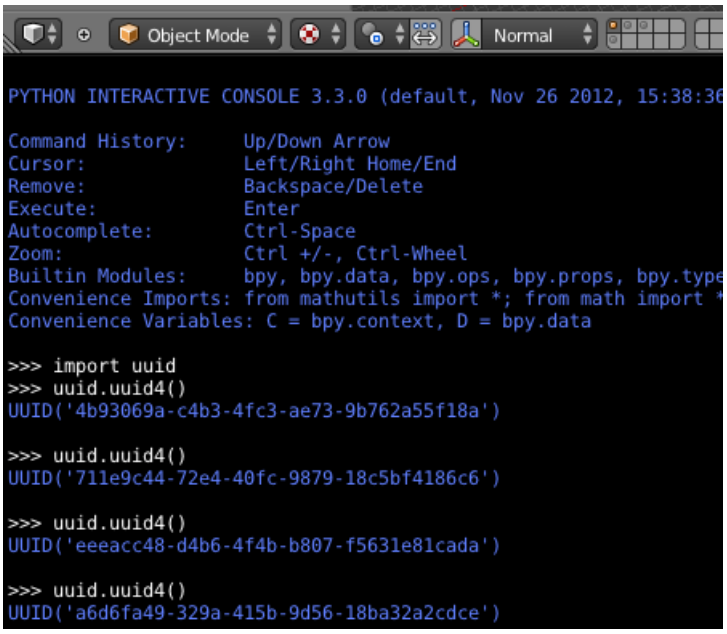

Figure 5: Generating UUIDs with Python

2.2 Corrective shapes

Introduction. Suppose you have tho shapes — `OpenUpperLip` and `OpenJaw`, which look perfectly fine on their own.

Figure 6: `OpenJaw` & `OpenUpperLip`

Unfortunately, humans are prone to opening their lips *and* the jaw at the same time! To represent this vicious action, you apply both `OpenUpperLip` and `OpenJaw`. That makes the Source engine to mathematically add them together. What you get isn't horrible (fig. 7), but doesn't show the bottom teeth; moreover, there isn't any way to show the bottom teeth using these shapes.

Figure 7: `OpenJaw` + `OpenUpperLip`: mathematical addition

Corrective shapes offer an elegant solution to the problem. If you create a shape with the expected outcome (fig. 8) and name it "`OpenUpperLip_OpenJaw`"[5], the Source engine will fade in this new shape whenever `OpenUpperLip` and `OpenJaw` are active at the same time, *correcting* the facial shape. This helps the animator, who can now think about what the shape *means* instead of how the shape *looks*: it might look very different depending on the context.

You can create corrective shapes for any shape combinations. The more corrective shapes, the better. The Heavy has 250 corrective shapes!

[5]Or "`OpenJaw_OpenUpperLip`" — thankfully, won't make any difference.

Figure 8: `OpenJaw` + `OpenUpperLip`: expected outcome

Don't create corrective shapes for areas that don't depend on each other, like the eyes & mouth. You don't need to create corrective shapes for something that can be solved by using Domination Rules (see section 2.3 for them).

Here's a list of the most necessary corrective shapes for a talking and emoting human head. These were deduced by some HWM modeling practice and aren't by any means definitive or complete, but without these shapes your character's face will break down just too much.

```
FunnelLipLo_FunnelLipUp
PuckerLipUp_PuckerLipLo
PuffLipUp_PuffLipLo
OpenJaw_OpenLips
OpenJaw_OpenUpperLip_OpenLowerLip
OpenJaw_OpenUpperLip_OpenLowerLip_Platysmus_SmileFull
OpenJaw_OpenLowerLip_Platysmus_SmileFull
OpenJaw_OpenUpperLip_Platysmus_SmileFull
OpenJaw_OpenLips_Platysmus_SmileFull
OpenJaw_OpenLowerLip
OpenJaw_OpenUpperLip
OpenJaw_Platysmus_SmileFull
OpenJaw_Platysmus
OpenJaw_SmileFull
OpenJaw_FunnelLipLo_FunnelLipUp
RaiseChin_CompressLips
OpenLowerLip_PuckerLipUp
OpenUpperLip_PuckerLipLo
OpenLowerLip_PuckerLipUp_OpenJaw
CheekV_CloseLidUp
InnerSquint_CloseLidUp
InnerSquint_OuterSquint
InnerSquint_OuterSquint_CloseLidUp
InnerSquint_OuterSquint_CloseLidLo
SmileFull_Platysmus
```

As you may have noticed from the list, **corrective shapes are specified for shape combinations**, not controller combinations!

You don't need to declare controllers for the corrective shapes, they just have to be in the DMX file (and consequentially in the compiled model). In fact, adding a corrective shape to a controller will **prevent it from being applied**.

One last thing. In Source, corrective shapes are applied "recursively". This is easily illustrated by table 1.

Active base shapes	Active corrective shapes[6]
A, B	A_B
A, B, C	A_B, C_A, B_C and A_B_C[7]

[6] Of course, if some lower-level corrective shapes don't exist, they won't be applied.

[7] In Source, the ordering of sub-shapes in the name doesn't make any difference: `InnerSquint_OuterSquint` = `OuterSquint_InnerSquint` etc.

Table 1: Application of corrective shapes.

Corrective shapes are insanely powerful and easy to use. They're applied absolutely transparently for the animator and the modeler, supporting grouped controls, stereo controls and wrinklemaps. This is a very major step up from the SMD/VTA subsystem and it is every Source modeler's duty to spread the knowledge.

Technical aspects. Unfortunately, there's a small confusing aspect of corrective shapes that often prevents everyone from grasping HWM instantly. Corrective shapes can exist in two flavors:

***Absolute* corrective shape** is the target shape for a certain combination; the shape that you want to *fade towards* when the combination is active.

***Relative* corrective shape** is a delta[8] that should be *added* to the underlying mix rather than being faded towards.

The relationship between a relative corrective shape and an absolute corrective shape is shown on fig. 9.

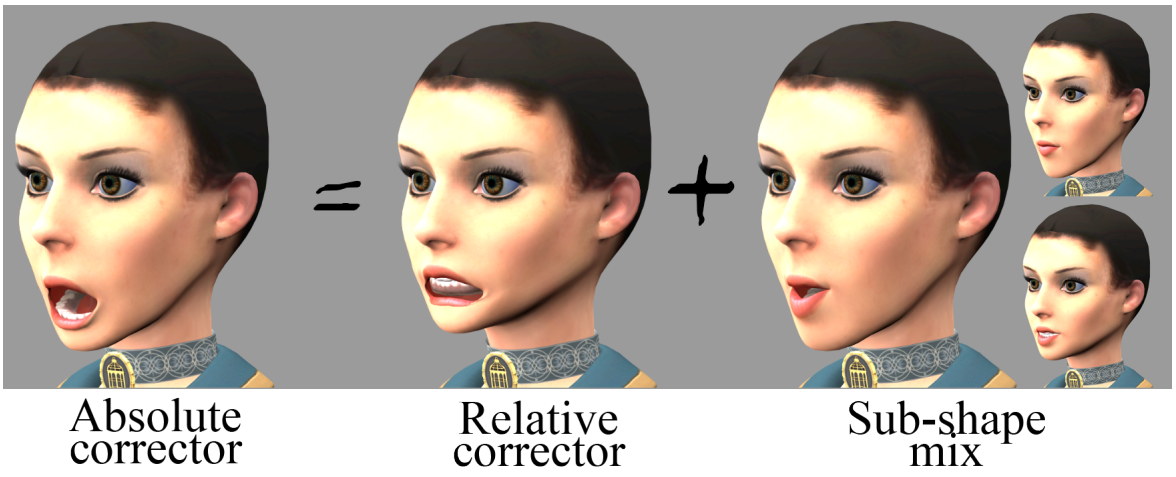

Figure 9: The relationship between absolute and relative corrective shapes

Absolute corrective shapes are easier to work with. Unlike relative correctors, they're not dependent on the underlying shapes, which means you won't trash them by modifying any of the underlying shapes and they do make sense on their own (or dare to explain, how the hell the middle face on fig. 9 is "open lips and open jaw").

[8] "Delta" means that a shape is stored as a list of displacements (3D vectors) from the base mesh shape (usually "Basis" in Blender) rather than a set of absolute coordinates.

However, relative corrective shapes are more computationally efficient: applying them is essentially adding a lot of 3D vectors together which is a nice task for any modern GPU. And the `studiomdl` model compiler wants relative corrective shapes, or the stuff you'll in see `hlmv` or SFM will be your very last nightmare.

Conversion between absolute and relative is pretty straightforward. Unfortunately, major 3D packages obviously don't have give a shit about Valve's internal standards and don't support this. The solution is writing custom code; below you can check out a Python function `Corr_AbsToRel()` from the Blender HWM Toolkit packaged with the HWM Elizabeth.

```
def Corr_AbsToRel(mesh_in, mesh_out, shapekey_in_abs, shapekey_out_rel):
    ''' Purpose: saves this corrector as a rel shape,
                appropriately checks for sub-shapes on the mesh_in
                under the assumption they all are RELATIVE (to keep things simple)
    '''
    if len(mesh_in.data.vertices) != len(mesh_out.data.vertices):
        raise ValueError('Different meshes specified.')
    subKeys = []
    for subKeyName in YeildSubShapeNames(shapekey_in_abs.name):
        key = FindShapeKey(mesh_out, subKeyName)
        if key:
            subKeys.append(key)
    subMix_co = []
    # Can loop on key-vtx, or on vtx-key. This is the former
    for i in range(len(shapekey_in_abs.data)):
        subMix_co.append(Vector((0,0,0)))
    for k in subKeys:
        delta_co = GetDeltaCoords(mesh_in, k)
        for i in range(len(shapekey_in_abs.data)):
            subMix_co[i] += delta_co[i]
    CopyShapeKey(shapekey_in_abs, shapekey_out_rel)
    for i in range(len(shapekey_out_rel.data)):
        shapekey_out_rel.data[i].co -= subMix_co[i]
```

HWM Elizabeth sources (and all the other DP Films HWM models, including the main cast from Adam Palmer's Cloud Odyssey) consequentially have two heads in their Blender source files: the mesh with the absolute corrective shapes, which we edit, and the mesh with relative corrective shapes, which we export and compile. The latter is automatically preprocessed by the Blender HWM Toolkit.

To avoid confusion, the HWM Toolkit requires the "absolute" mesh's name to end with `_abs` postfix.

2.3 Domination rules

Try inflating cheeks with your mouth wide open. You, The Heavy, The Soldier, The Coach or even HWM Elizabeth all can't handle this very simple[9] task.

This restriction is utilizing the Domination Rules concept. Opened mouth *dominates* inflated cheeks. In a HWM model, you can specify such a relationship with the following DMX code:

You've probably already guessed you can write string arrays in the place, of, well, string arrays; what I meant is that you can specify more `suppressed` shapes than one.

[9]For the cripplingly overspecialized species of *Inflatedcheekswithopendmouths-aliens Vulgaris*, with whom the author is currently affiliated.

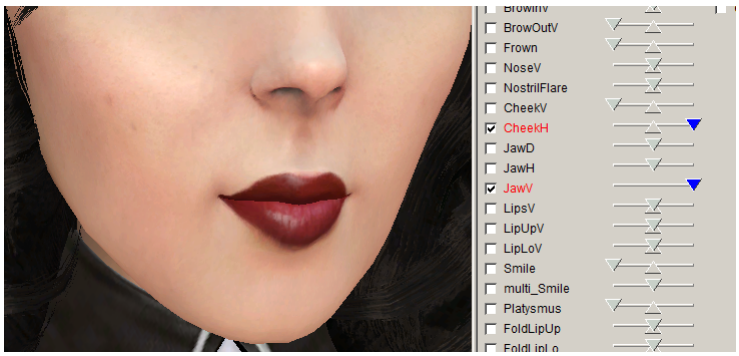

Figure 10: Even though `CheekH` = 1 (trying to fade in `InflateCheeks`), it's no use, because `JawV` = 1 (which already faded in `OpenJaw`). Puny humens.

```
"DmeCombinationDominationRule"
{
	"id" "elementid" "ab961fe8-2ea7-5c46-a671-d7ccc5716814"
	"name" "string" "rule"
	"dominators" "string_array"
	[
		"Platysmus"
	]
	"suppressed" "string_array"
	[
		"FunnelLipLo", "FunnelLipUp", "LipCnrTwst"
	]
}
```

This code sets the `Platysmus` shape to dominate `FunnelLipLo`, `FunnelLipUp`, `LipCnrTwst`. Whenever `Platysmus` fades in, these three will proportionally fade out if they're active at the moment.

Vice versa, you can set more than one shape to be the dominator.

```
"DmeCombinationDominationRule"
{
	"id" "elementid" "eaac4df5-9de8-ca4e-8af1-7bdfd19da625"
	"name" "string" "rule"
	"dominators" "string_array"
	[
		"OpenLowerLip",
		"OpenUpperLip"
	]
	"suppressed" "string_array"
	[
		"OpenLips"
	]
}
```

Somewhat counter-symmetrically, this code means `OpenLips` will be dominated if and only if `OpenLowerLip` & `OpenUpperLip` are *both applied*. This means you can still have corrective shapes `OpenUpperLip_OpenLips` & `OpenLowerLip_OpenLips`.

As with the controller structure, it's usually not necessary to edit the domination

rules. Just, once again, use the ones Bay Raitt did, embedding the domination rules from the Source SDK TF2 model sources.

Domination rules drastically reduce the number of needed corrective shapes. You don't need to create a corrective shape for a combination that's covered by a domination rule.

2.4 Wrinklemaps

Intro. Wrinklemaps are a cheap way to simulate wrinkles that appear when we compress or stretch our faces. These wrinkles, unfortunately, are three to five times smaller than the common mesh resolution, so we have to imitate them with some sort of trick — as usual, this kind of stuff really complicates 3D. There're actually two such diffuse textures: the *compress map* and the *stretch map*.

What wrinkles should we paint on the stretch map and what — on the compress map? Well I found a superb guide from the real pros unlike us (fig. 11)[10].

Figure 11: Bob Parr aka Mr Incredible's facial expressions. © Disney/Pixar. An amazing study by Pixar's Greg Dykstra. A compressed, grinning face on the *left*; a stretched, shocked face on the *right*. Now I believe you can guess where to paint which wrinkles.

A compressed face, and the *compress map*, generally will have wrinkles around the upper nose crest (there's even a shape in the standard HWM TF2 set called `WrinkleNose`) and cheeks/mouth corners pulled by the *zygomaticus major* muscle.

The most important thing in the *stretch map* are forehead wrinkles; and the secondary important thing are neck/chin area wrinkles. In fact, why I'm even talking about this? Launch Photoshop, grab a mirror and imitate Bob's faces from the figurines!

Don't go over-excited on wrinklemaps, especially on NPR models. Young stylized women especially should be handled with extra care or they'll turn out old (still stylized and women though).

[10]Scanned from a Pixar artbook I was lucky enough to buy at the Pixar Expo in Amsterdam.

The Source Engine lets you to specify these textures in a VMT file using the special tags: `$compress $stretch $bumpcompress $bumpstretech`. Correct, Source supports seperate bump maps for compressed and stretched states too. An example from L42D's Coach head VMT:

```
"VertexLitGeneric"
{
        "$baseTexture" "models\survivors\coach\coach_head"
        "$compress"  "models\survivors\coach\coach_head_compress"
        "$stretch"  "models\survivors\coach\coach_head_stretch"

        "$bumpmap"  "models\survivors\coach\coach_head_normal"
        "$bumpcompress" "models\survivors\coach\coach_head_normal_compress"
        "$bumpstretch" "models\survivors\coach\coach_head_normal_stretch"
    ...
```

If such a texture (or both) is specified, and the model was compiled from a DMX file with wrinkle data embedded, Source will proportionally blend the wrinkle texture around the vertices that are displaced from the base state. This is an absolutely transparent process (fig. 12).

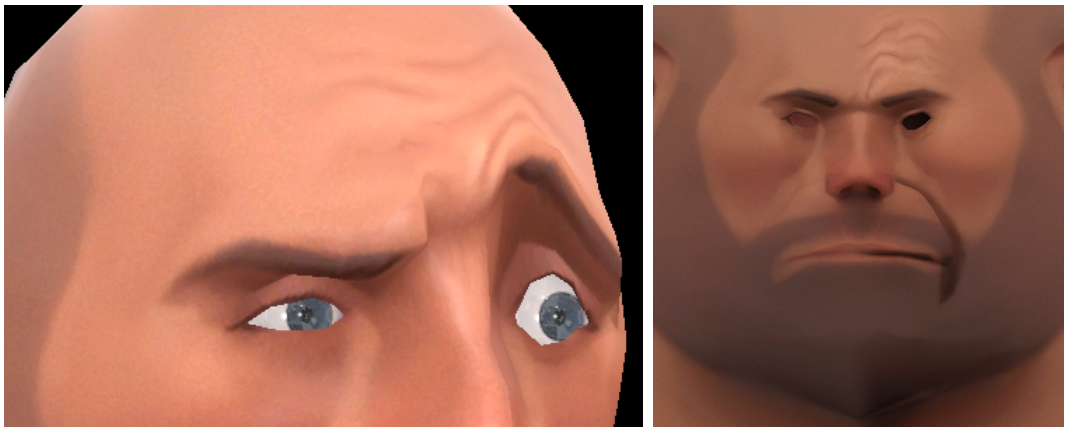

Figure 12: Wrinklemaps kicking in on the right!

How to? Blender Source Tools make adding wrinklemaps really easy for you (if you have the texture, of course — for instance, the author can't paint). They embed all necessary wrinkle data in the mesh's DMX automatically.[11]

But wait, asks the reader, **how does the Source engine know when to blend the compress map and when to blend the stretch map?** Does it have a sort of a super-sophisticated inhumane heuristic algorithm for sorting this out?

Well, it doesn't. The reader, however, hopefully *does* — or we're not getting anywhere. Fortunately for humens, they're quite good at having such algorithms hardwired somewhere in their puny brains. Do you remember the `wrinkleScales` parameter in the controller/slider definition on, say, page 5?

```
1 "DmeCombinationInputControl"
2 {
3     "id" "elementid" "4c4b3440-d611-da4a-beeb-90877351fb02"
```

[11] If the corresponding `wrinkleScale` isn't 0 in the controller block

```
4       "name" "string" "OuterSquint"
5       "rawControlNames" "string_array"
6               [
7                       "OuterSquint"
8               ]
9       "stereo" "bool" "1"
10      "eyelid" "bool" "0"
11      "wrinkleScales" "float_array"
12      [
13          "-2"
14      ]
15  }
```

Yep, this number on line 13 tells the Source engine what to do. Let's call it A for convenience. If

1. $A < 0$, the *stretch map* will be blended in whenever the *shape* fades in;
2. $A > 0$, the *compress map* will be blended in whenever the *shape* fades in;
3. $A = 0$, wrinkles won't appear when the shape fades in. (This doesn't prohibit them from being applied by some other shape though.)

The bigger the $|A|$ value is, the more prominent the wrinkles will appear when the corresponding *shape* fades in. I emphasize, *the shape*: as the corrective shapes & domination rules, wrinkle scales are specified *per-shape* and **not** *per-controller*.

There's a frequent case with two and more shapes assigned to a controller, like in the following snippet:

```
1   "DmeCombinationInputControl"
2   {
3           "id" "elementid" "2f96be1a-73c9-1c40-af37-7fcc1d2c21aa"
4           "name" "string" "Smile"
5           "rawControlNames" "string_array"
6           [
7                   "SmileFlat",
8                   "SmileFull",
9                   "SmileSharp"
10          ]
11          "stereo" "bool" "1"
12          "eyelid" "bool" "0"
13          "wrinkleScales" "float_array"
14          [
15                  "0.5",
16                  "1.0",
17                  "1.5"
18          ]
19  }
```

As you can see, there're 3 wrinkle scales and 3 shapes assigned to the controller. They are sequentially matched up by their indices: the *first* shape uses the *first* `wrinkleScale`, the *second* shape uses the *second* `wrinkleScale` etc.

In the `Smile` controller example,

`SmileFlat`**'s intensity** is 0.5,

`SmileFull`**'s intensity** is 1.0,

`SmileSharp`**'s intensity** is 1.5.

Once again, the `wrinkleScales` setup you'll find in the Source SDK TF2 samples is probably the best, but don't hesitate to experiment, especially with something that isn't a batshit crazy mercenary male. There can be a batshit crazy mercenary *female*!

2.5 Stereo controllers

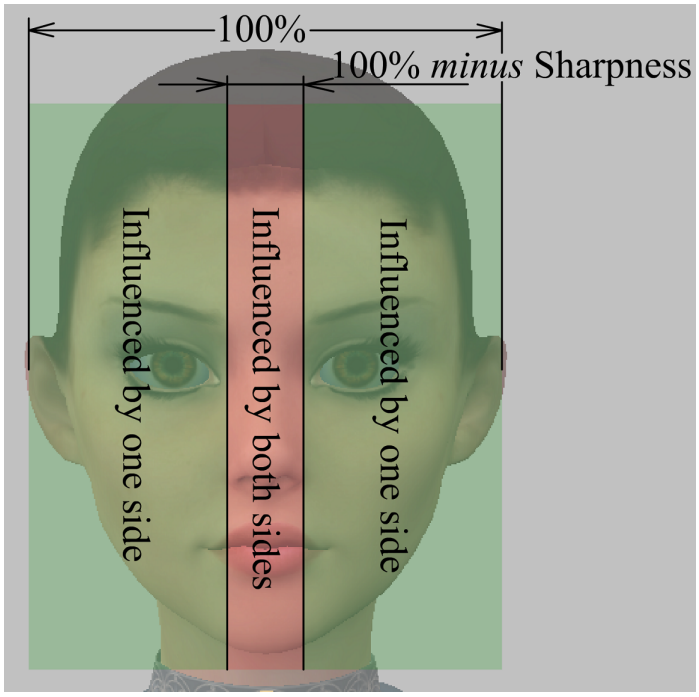

Figure 13: Setting up stereo controllers in Blender

Actually, we've successfully covered stereo controllers during the course of previous sections. To make one, set `stereo` to 1 in the controller definition in the DMX file (see section 2.1).

The only part we've left untouched is how Source tells apart the left part from the right part. This is done by a special `balance` value in the DMX file, which determines how left or how right the vertex is. Maya users can paint this value over the face mesh to define the left and right parts.

Blender users are more limited in that regard as of now. In fig. 4 you can see a **Stereo sharpness** slider in the lower-right corner. It defines how much of the face is going to be affected by both sides' "muscles" (fig. 13). As a rule of thumb, set the sharpness as low as you can without losing the ability to control the eyes separately. The values usually lie between 75 and 90 percent.

This is not an ideal system. Generally one would want the mouth corners to be influenced by both sides of the face and still have the eyes separated. That's not a problem too big though!

3 The model sources

> **!** **As of time of this writing, the model has been tested to work with Blender 2.69 and Blender SMD Tools 2.X *only*!**

3.1 The files and folders

`sfm_liz_source/`
: is the root folder ("`//`").

`//sfm_liz.blend`
: is the **main Blender file** with the meshes. Bet you didn't figure that out!

`//head_contollers.dmx`
: contains Liz's facial animation rules, the DMX Flex Controller Block (see previous section for explanations). It's a KeyValues2 DMX that can be edited by hand or by script of some kind. It's referenced in the .blend file as a text block. The file is meant to be embedded in the exported head mesh via **Advanced Mode** in **Source Engine Export panel** in Blender SMD Tools on export.

`//liz_sources_readme.tex`
: is the LaTeX source for this document.

`//liz_readme.tex`
: is the LaTeX source for engine files' readme.

`//parts/dmx/`
: is the folder where the **DMX meshes** are stored for `studiomdl`.

`//scripts/`
: is the folder with the **QC files** that `studiomdl` uses to compile the model.

  - `liz_pristine.qc, liz_bas.qc etc.` are the main QCs that get compiled into respective models.
  - Include (`$include`) `liz_head.qci`, `liz_head_e3.qci`, `liz_head_bas.qci` in your QC file when you want to use the normal head, the E3 2012 head or the pale Burial at Sea one *respectively*.
  - `liz_head_commons.qci` has some common values for various eye-related SFM specific `$attachment` QC commands.
  - `liz_hlp.vrd` sets up the procedural wrist (`bip_lowerArm_X`) rotation. It's included in the model via the `$proceduralbones` QC command. The VRD file format is documented in the Valve Developer Community.

  NB: when you reference a DMX model from in your QC file and the QC is in the `//scripts/` folder, ensure you are referring to the correct relative folder, e. g.

```
$model body "body.dmx" // Incorrect, studiomdl won't find the file
$model body "../../parts/dmx/body.dmx" // Correct
```

`//supplemental/` contains some pictures and SFM sessions used for this manual and the "promotional" material, as well as...

`//supplemental/extra_models.7z` contains some custom SFM models (the Maltese Falcon book and the adjustable light gobo) that are used in "The 1950 Update" banner

`//supplemental/sfm_hwm_sample.7z` A sample HWM project to use with the "How To: Do a HWM model in Blender from scratch" tutorial

`//supplemental/sfm_hwm_sample.7z/tf2_head_controllers.dmx` A standard TF2 Controller Block for you to embed in your characters.

`//texture/tga/` contains the **TGA textures** (and some opaque PNGs as well, ha!). They're referenced in the `sfm_liz.blend` file 'cause Blender cannot into VTFs.

`//blenpy/` contains the Python scripts for Blender that are used in authoring the model.

`//blenpy/hwmtk/` contains a premature[12] version of the HWM Toolkit, documented later in this file.

3.2 The sfm_liz.blend file and how to work with it

Elizabeth's source files behave just like any other Blender model and all of your Blender knowledge applies to them. However, the HWM stuff adds some non-intuitive and non-standard complexity that needs to be explained before using it. We're including some guidance on how the .blend file is organized and how to edit the model.

This info and instructions can be applied to any HWM character model in Blender so listen closely, an aspiring HWM modeler!

General notes. The file is separated into different scenes by Elizabeth's variations. Armatures and attachment points are distributed to layers. The materials reference the textures in `texture/tga`.

3.2.1 Opening the .blend file properly

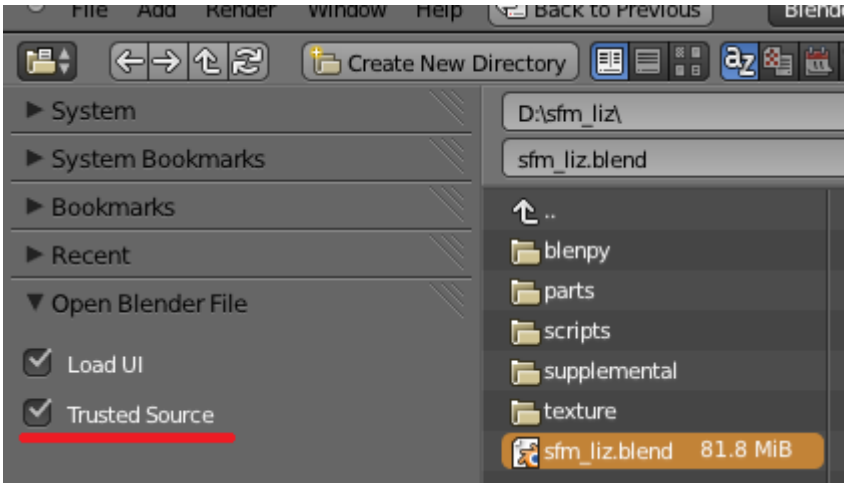

Figure 14: Do you trust us?

Make sure you've loaded the .blend file as a **Trusted** file (fig. 14). Blender can run user-specified Python scripts when a particular file loads. In Blender 2.68 Blender devs

[12]More like a 5-month-old aborted fetus that survived with satanic powers. Still works though!

realized that shitheads may embed harmful scripts in the .blend file and register them to be run on startup.

Just imagine the horror: you download a Blender model of a fine naked Quarian (supposedly), open the .blend file and suddenly My Documents is gone! To prevent this, Blender since version 2.68 doesn't run these startup scripts unless permitted to.

So why I'm telling you this shady stuff? In Liz's .blend file there's a startup script that does important stuff that needs to be done on startup or you won't have important and useful tools enabled (fig. 15). So yeah, you'd want to load the file as trusted.

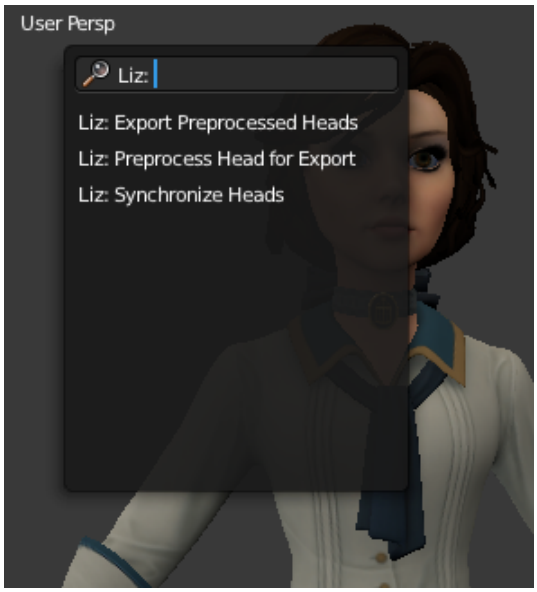

Figure 15: Liz-specific tools

3.2.2 Setting up the Source Filmmaker path

Blender Source Tools need to know where your Source Filmmaker is to compile the model files. Obviously SFM isn't installed in the same folder on every machine so you'll have to adjust the Engine Paths for your machine's SFM installation.

To save yourself from doing it manually (in all scenes!), open the Script Editor and open the `resetExports.py`. Running this script (press Alt-P in the Text Editor window or a big **Run Script** button in the bottom-right corner) will reset the Engine Path to your SFM's location in all scenes of your file.[13] If the location is still incorrect even after running the script, close Blender, and run SFM once, then repeat.

Once this is done, save the file. You have to do this operation only once, Blender will save the path in the file.

3.2.3 Custom Blender Operators

Head Management. We mentioned that this Elizabeth model has a few specific Blender operators (fig. 15) to aid working with it.

1. *Liz: Pre-process Head for Export.* Creates a copy of the `liz_head_abs` mesh (named `liz_head_rel`) and prepares it for export and `studiomdl`, switching all corrective shapes to relative.

[13] In the pre-release BST 2.0 the Source Export parameters aren't stored in the same place as in old 1.8.5 so this script fixes this up too.

2. *Liz: Synchronize Heads.* The Burial at Sea head and the E3 head have exactly the same topology as the head that's used for the main game Elizabeths, but with a different material. Of course, we want to exploit that to re-use the facial shapes between the head meshes. This operator updates the `liz_head_bas_rel` head and `liz_head_e3_rel` head with the shapes from `liz_head_rel`, adding new shapekeys, modifying existing shapekeys and deleting removed shapekeys.[14]
3. *Liz: Export Heads Only.* Exports the three heads with relative corrective shapes with their settings via the Blender Source Tools. **Warning: buggy at the moment.**

They're all implemented in `init_importHwmtk.py` using the HWM Toolkit's functions. Refer to this file to get an idea on how to code stuff using the HWM Toolkit.

Shapekey Authoring. The HWM Toolkit provides an *Soft Blend from Shape* operator, which is quite useful in working with complex shape keys.

This operator performs identically to the *Blend from Shape* built-in operator, but it supports soft selection[15], making it possible to blend in facial shapes with much more control, which is highly useful when authoring corrective shapes.

You can adjust the blending amount, the falloff distance, and the shape of the falloff cone (fig. 16 and fig. 27) to get different results to satisfy your artistic requirements.

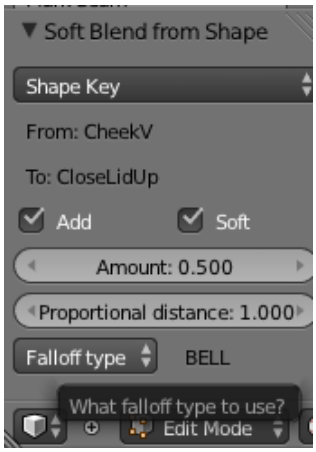

Figure 16: *Soft Blend form Shape* in action

3.2.4 Head meshes

There're four Elizabeth heads in the .blend file — which one's which?

Absolute and relative. As discussed in the HWM Introduction section, absolute corrective shapes are easier to edit, but `studiomdl` wants relative corrective shapes. The solution is to have two head meshes — the first with absolute corrective shapes, and the second — with relative corrective shapes, `liz_head_abs` and `liz_head_rel` respectively.

So, to edit/add/update a facial expression, **you do it on** `liz_head_abs` and then update `liz_head_rel`. Updating is done internally with the HWM Toolkit, via the

[14] If you want to make a change to a facial shape, do it on `liz_head_abs`, run *Liz — Pre-process Head* and finally *Liz — Synchronize Heads* to update all relative head meshes.

[15] aka Proportional editing

`hwm.PreprocessMesh(meshName)` function. In the `sfm_liz.blend` file, the aforementioned *Liz: Pre-process Head for Export* shortcuts to it.

Head variations. E3 and BaS Elizabeths have their own heads, which are geometrically identical to the regular Elizabeth's head. The difference is the linked material file. Of course, we could change the materials on the head mesh every time we would want to compile the other variation, but that's highly counterproductive.

The productive way is to set up the materials once on the duplicate meshes and then synchronize them to the "main" relative head (we obviously want all of our Elizabeths having the same facial controls). Then we just export all heads and compile them into different models. This is done via the aforementioned *Liz: Synchronize Heads* Blender operator. The synchronization is using some of the HWM Toolkit's functions for shapekey management.

3.2.5 How To: Editing a facial expression

So, to edit/add/update a facial expression, **you do it on** `liz_head_abs` and then update **the other relative meshes**. Updating is done in two steps:

1. Update the Pristine Liz's relative head mesh with *Liz: Preprocess Head for Export* (you can access all Blender operators by hitting Space).
2. Update E3 & BaS heads' relative meshes with *Liz: Synchronize Heads*.

Now you're ready to export and compile! Don't forget it's the mesh with *relative* corrective shapes that you want to get compiled.

3.2.6 How To: Adding a corrective shape

What makes HWM really powerful is the total absence of any special settings to enable the corrective shapes. Everything is straightforward here.

1. Figure out the combination you want to correct (e. g. `CheekV` & `CloseLidUp`)
2. Create a new shapekey for your corrective shape on the absolute mesh. You can use **New Shape from Mix** button in the Shape Keys Blender panel to make a solid base for your corrective shape.
3. Name it after the combination you're correcting. E. g. for `CheekV` & `CloseLidUp` the corrective shape name would be `CloseLidUp_CheekV`.
4. Sculpt the desired result in the new shapekey.

Done! Update the relative mesh (in Liz's case, use the operator), export and compile. It will work in the engine right away.

If you're doing a higher-order corrective shape (for example, `InnerSquint_OuterSquint_CheekV_CloseLidUp`), it's usually easier to start with the lower-order shapes and then mix and adjust them to get a higher-order corrective shape.

3.2.7 How To: Adding a new slider

We discussed sliders (controllers) in the HWM Introduction. You (I hope) remember, that

1. Shape keys are grouped into sliders/controllers.
2. The grouping is done in the Controller Block file.

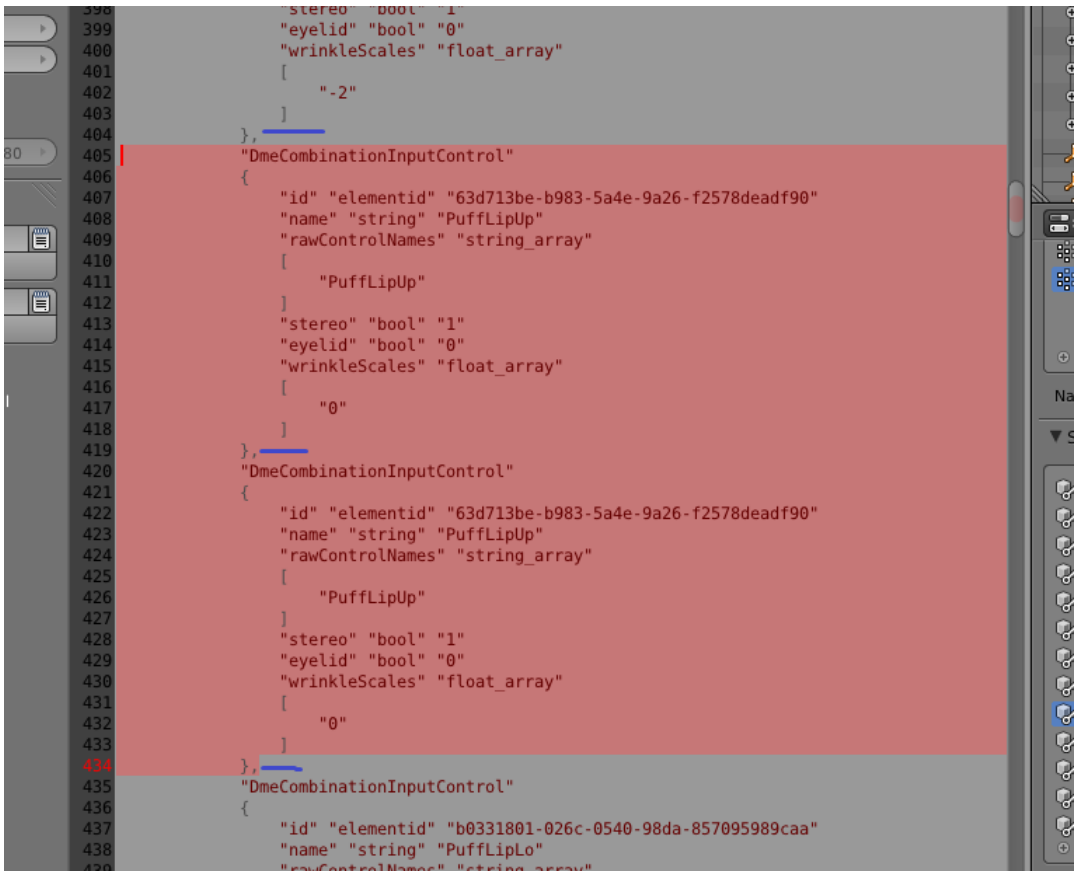

Figure 17: Duplicating a controller declaration

3. Blender Source Tools can embed these controllers blocks into the mesh when exporting.

Liz's Controller Block File is `head_controllers.dmx`. You can open it with Notepad or in Blender's Text Editor.

Let's pretend we need to add a have a control on Liz's face with her eyes jumping out of their sockets, `EyesOut`. Assuming you've already created a new unpleasant shape key called `EyesJumpingOut`, we now need to declare that controller in the Controller Block File.

To do that, open the `head_controllers.dmx` and scroll to an array of `DmeCombinationInputControl` blocks.

The easiest way to add a new slider is by duplicating an existing slider declaration (fig. 17). Note the commas seperating the controller declaration blocks, highlighted with blue marks.

After duplicating, replace the name of the new controller as you want.

```
...
"DmeCombinationInputControl"
{
        "id" "elementid" "63d713be-b983-5a4e-9a26-f2578deadf90"
        "name" "string" "EyesOut"
    ...
```

Now add in the shape keys you want to assign to the controller.

```
    ...
```

```
    "name" "string" "EyesOut"
    "rawControlNames" "string_array"
    [
        "EyesJumpingOut"
    ]
    ...
```

And finally, generate a new UUID (see page 9) and replace the old one.

```
...
"DmeCombinationInputControl"
{
        "id" "elementid" "dfcc0415-d30e-4e80-beb0-2f9ac9cc63fd"
        "name" "string" "EyesOut"
    ...
```

In the the new declaration will look like that:

```
...
"DmeCombinationInputControl"
{
        "id" "elementid" "dfcc0415-d30e-4e80-beb0-2f9ac9cc63fd"
        "name" "string" "EyesOut"
        "rawControlNames" "string_array"
        [
                "EyesJumpingOut"
        ]
        "stereo" "bool" "1"
        "eyelid" "bool" "0"
        "wrinkleScales" "float_array"
        [
                "0"
        ]
},
...
```

Always make sure you're preserving the DMX file encoding syntax. It's a good idea to have a backup of the controller file before actually changing anything as finding mistakes by hand is really tedious.

Now select your controller block file as a controller source in the Advanced flex mode in Blender and you're good to go. Controller declarations are described in sect. 2.1, lurk there if you have any questions.

3.2.8 How To: Adding a domination rule

Domination rules are stored in the same Controller Block File, just a little further. Their data elements are called `DmeCombinationDominationRule`. You'll see a big array of these in around the second half of the file. Once again, the best way to add a new domination rule is copy-pasting an existing one an then adjusting it to suit your needs.

For reference, this is a domination rule with Liz's custom[16] `EyesWideOpen` shape.

```
"DmeCombinationDominationRule"
{
        "id" "elementid" "e135a3a2-f178-fe43-8b43-47275db12b21"
```

[16] By "custom" I mean, not found on TF2 models.

```
        "name" "string" "rule"
        "dominators" "string_array"
        [
                "WrinkleNose"
        ]
        "suppressed" "string_array"
        [
                "EyesWideOpen"
        ]
},
```

Refer to section 2.3 to learn how domination rules work.

3.3 Python

3.3.1 `init_importHwmtk.py`

This script

1. appends the path to the HWM Toolkit to Python's `PATH`, exposing the HWM Toolkit to Python's import system,
2. imports the *Soft Blend from Shape* operator to Blender
3. adds the shortcut Blender operators to preprocessing and synchronizing the head meshes

As you can see in the lower part of Blender's Text Editor, it's registered to be run at Blender startup so we can have the needed HWM Toolkit functions right away.

3.3.2 `armature_Bio3BonesToTF2Bones.py`

Somehow the Bioshock Infinite human biped skeletons[17] have exactly the same topology and hierarchy as TF2 skeletons do (some of the bone orientations are different though). This wonderful fact is commemorated by the titular script, that renames the bones to comply with the TF2 names on the selected armature.

3.3.3 `tools_publishLiz.py`

An incredible invention that sped up productivity by a half a dozen times. In fact, we stole this script from an alternate reality where we are cool.

One day we were feeling terrible. You know this feeling:

1. Ever felt yourself terrible packing up a zip with your model (and materials) to publish? This is a pain in the ass indeed because of the folder structure which needs to remain intact, but you can't just copy the whole `models` folder etc. This script does it for you (packing a 7z-archive, but still). Even grabs the needed rig script and the readme PDF.
2. Ever had some extra/testing/beta/garbage models or even files in the model folder which you don't want to get published? You omitted them by hand when packing your zip? This one omits them for you, only allowing the files you specify to pass.
3. Ever had a lot of extra shit in your source files that you don't need to publish? We do (fig. 18): Blender backups, Dropbox crap, LaTeX intermediate files, testing stuff

[17] Skeletons = armatures, bones = joints.

and more. This script filters out all the un-needed crap and assembles a 7z archive to publish.

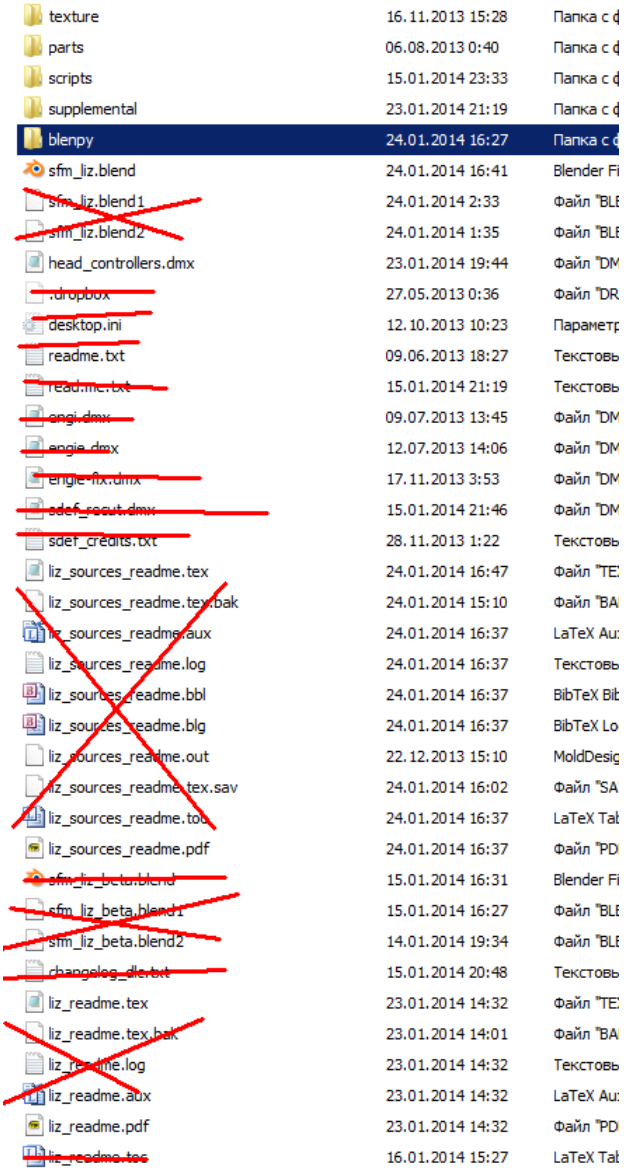

Figure 18: Don't need these in the published archive!

One can specify the files to include and omit in the published 7z. By default, everything is ommited.

```
allowSourceFiles = [
    "*.qc","*.qci", "*.dmx", "*.vrd",
    "*.py","*.blend",
    "*.tga", "*.png", "*.jpg",
    "*.pdf", "*.tex", "*.bat", "*.psd", "*.7z"
]

# Top sicrit stuff we're not pyblising
disallowSourceFiles = [
    "liz_prtmn.qc",
    'sfm_liz_beta.blend',
```

```
    "comb.qc",
    "engie-fix.dmx",
    "engie.dmx",
    "engi.dmx",
    "blender_medal.blend",
    "*.blend?", # Backups
    "*.blend??",
    "book.blend",
    "book",
    ".dropbox", # Dropbox crap
    "sdef_*",
    "liz_readme.pdf"
]
```

You would not believe how much better our lives became when we started using this.

3.3.4 `rig_biped_simple_hwmliz.py`

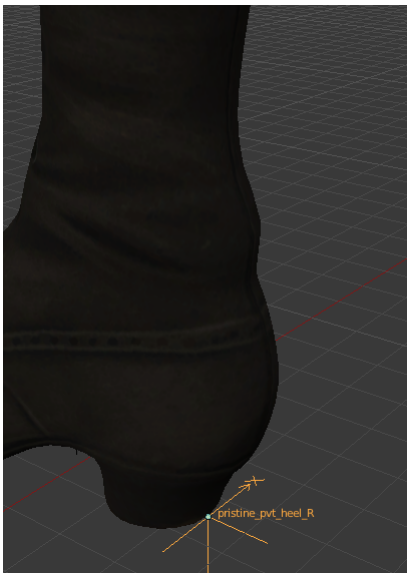

Figure 19: The heel pivot attachment position for `rig_footRoll`

You probably tried that nice `rig_footRoll` control on the updated Liz. You did like it, because it's awesome. Whoever at Valve decided to include this is really cool.

Everything you need to have on you model to enable this `rig_footRoll` is a attachment marking where the heel is (fig. 19) and `rig_biped_simple.py` will do the work for you. Unfortunately, Liz historically is rotated 180 degrees around Z in relation to TF2 models, and her foot bone orientation doesn't match up with TF2 models. This has led to `rig_biped_simple`'s `rig_footRoll` to break her gentle feet.

The changes made to `rig_biped_simple.py` were switching the foot rotation axis in `CreateReverseFoot`:

```
26      ...
27      footRollDefault = 0.5
28      rotationAxis = vs.Vector( 0, 0, 1 ) # Was 1, 0, 0 in original
29      ...
```

And reversing the slider's effect with sign changes:

```
78      ...
79      # Create the expressions to re-map the footroll slider value for use in the
            constraint and rotation operators
80      toeOrientExprName = "expr_toeOrientEnable_" + sideName
81      toeOrientExpr = sfmUtils.CreateExpression( toeOrientExprName, "inrange( footRoll,
            0.5001, 1.0 )", animSet )
82      toeOrientExpr.SetValue( "footRoll", footRollDefault )
83
84      toeRotateExprName = "expr_toeRotation_" + sideName
85      toeRotateExpr = sfmUtils.CreateExpression( toeRotateExprName, "max( 0, (footRoll -
            0.5) ) * (-140)", animSet ) # Was (140) in original
86      toeRotateExpr.SetValue( "footRoll", footRollDefault )
87
88      heelRotateExprName = "expr_heelRotation_" + sideName
89      heelRotateExpr = sfmUtils.CreateExpression( heelRotateExprName, "max( 0, (0.5 -
            footRoll) ) * 100", animSet ) # Was (-100) in original
90      heelRotateExpr.SetValue( "footRoll", footRollDefault )
91      ...
```

This has fixed the leg breaking. Definitely an improvement (fig. 20).

Figure 20: This is one bad FRX

4 How To: Do a HWM model in Blender from scratch

4.1 A warning

This HWM pipeline is in its early stage. Setting it up is a little rough on the edges. You'll have to use some special scripts and make basic[18] adjustments to them. You'll also have to read this PDF a lot. Make sure you

1. Have a good grasp of Blender
2. Have an idea on how to compile for Source

[18]Basic even for someone who sees programming for the first time, do not worry.

3. Have enough time since HWM is a lot of work

before entering.

It is worth mentioning **there is another pipeline utilizing Valve's `dmxedit` tool**. It was used for the HWM Femscout, for example. I don't really like that other pipeline because of its un-opened-ness and a good number of serious mindfucks with proprietary tools mysteriously crashing and piss-poor documentation, but it has some appeals. If you're still interested, contact **Gamerman12** and **Narry Gewman** on Steam.

The pipeline presented here is completely open-source and provides more control. It was tested on quite a bunch of HWM models, including the Cloud Odyssey characters, some of the Insurgency release trailer's characters, and Liz, of course. I'm much easier to get in contact with than Valve, so I strongly advise using this one, especially if you're a Blender user.

May the Bay be with you!

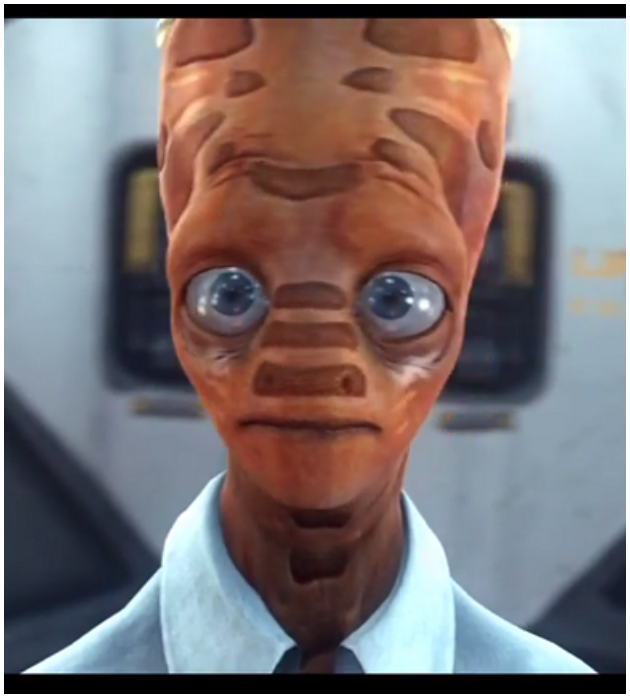

Figure 21: ~~You need a head this big to proceed!~~ This face is using the HWM Toolkit.

4.2 Preparations

You need Blender Source Tools 2.0 to proceed.

4.2.1 Planning

First of all, you must ask yourself two questions:

1. *"Do I need corrective shapes?"*
2. *"Do I need controls grouping and domination rules?"*

Answering "no" to one of them makes your life a little easier. Answering "no" to both means you don't have to do anything special at all, just export your meshes as DMX files with Blender Source Tools.

If you want to make a TF2/L4D2 style facial controller setup for your character, both answers are automatically *"Yes!"*.

Do I need corrective shapes? If your character is going to fully articulate in close-ups, you do need corrective shapes. However, if he's gonna be far away or do just simple facial movements, then you probably don't need them.

Just to guide on what corrective shapes can do, all TF2 HWM characters have lots of corrective shapes. Elizabeth has corrective shapes. Femscout and Femsniper have corrective shapes, etc.

If you use corrective shapes, you'll need the HWM Toolkit.[19]

Do I need controls grouping and domination rules? Grouped helps the animator, who can go from compressed lips to the opened mouth with a single slider. Or `CloseLid` on TF2 models — it's composed of two shapes (`CloseLidUp` and `CloseLidLo`) that are superimposed in a convenient (for the animator) way. So ask your animator.[20]

Domination rules usually come with the corrective shapes, if you're creating a more-or-less humanoid face, as they help to reduce the number of needed corrective shapes — you don't need a corrective shape for a combination that's covered by a domination rule.

4.2.2 Setting up your file

From now on, we suppose you said *"Yes!"* to corrective shapes. That means we'll need to use the HWM Toolkit.

There's a sample HWM project featuring Revzin's copyrighted character The Lazy HWM Blarbface (fig. 22) in the Liz source files in `//supplemental/sfm_sample_hwm.7z`. Check with it if you're not sure what to do. And with Liz, of course.

Adding the HWM Toolkit to your Blender file is a little complicated at the moment, but nothing *too* complicated.

1. Create a new Blender file or open the one with your character.
2. Copy the `//blenpy/hwmtk` folder from HWM Elizabeth's sources next to your `.blend` file.
3. Create a new script file and name it `hwminit.py` (or whatever else). Unzip (un-Seven-zip) the sample HWM project and within it find the same-named `hwminit.py` file with these contents:

```
absoluteMeshName = "head_abs"
# Adjust this line if your character's head mesh isn't named 'head_abs'

import sys, bpy

# Expose the HWMTK pyfiles to the interpreter
# Path to the HWMTK folder relative to the .blend file
sys.path.append(bpy.path.abspath('//hwmtk\\'))
```

[19] There is a way to sculpt them with bare Blender by sculpting the corrective shape on top of the sub-shape mix. This is, however, a very calamitous way. Changing any of the sub-shapes will change the corrective shape too, effectively preventing you from making any adjustments.

[20] Or your inner animator.

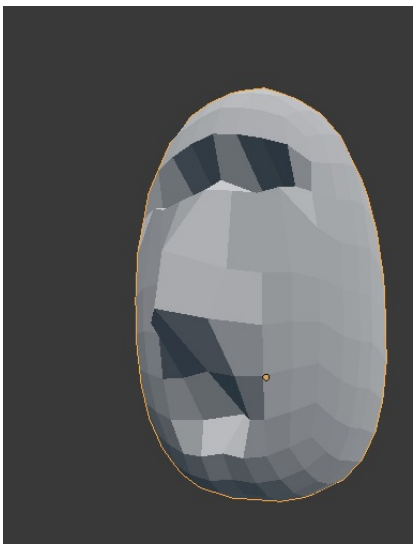

Figure 22: The Lazy HWM Blarbface © 2001–2100 Revzin and Associates

```
# Enable the Soft Blend from Shape op
import op_softblend
# Import the main toolkit module
import hwm

class HwmOps_PreprocessHeadOp(bpy.types.Operator):
    """Preprocess abs correctors to rel correctors"""
    bl_idname = "object.hwm_hwmpreproc"
    bl_label = "HWM: Preprocess Head for Export"

    @classmethod
    def poll(cls, context):
        return bpy.context.mode == 'OBJECT'
    def execute(self, context):
        relMesh = hwm.PreprocessMesh(absoluteMeshName)
        if (not relMesh):
            self.report({'WARNING'}, 'Failed to preprocess the %s mesh. Look in the
                console for now...' % absoluteMeshName)
        else:
            self.report({'INFO'}, "Successfully updated %s..." % relMesh.name)
        return {'FINISHED'}

bpy.utils.register_class(HwmOps_PreprocessHeadOp)
```

Copy-paste this code from the sample project into your project's `hwminit.py`.

4. Adjust the `absoluteMeshName` line if your character's head mesh is not named `'head_abs'`. Or rename your character's head mesh! Remember that the HWM Toolkit requires the mesh name to end in `'_abs'` for safety reasons.[21]
5. Register this script to be run on Blender startup by checking "**Register**" at the bottom-right of the text editor.
6. Run the script. If you did place the HWM Toolkit files correctly, you'll see `'Hello from hwm.py!'` in the Blender console.

[21] Data safety is valuable around here.

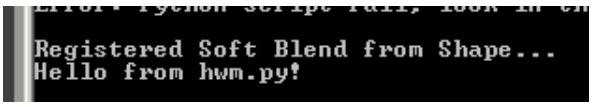

Figure 23: A successful HWM Toolkit import

Now you're able to use the HWM Toolkit. Most importantly, the *HWM: Preprocess Mesh for Export* operator.

4.2.3 Getting the controller block

This part is needed if you need controls grouping and/or domination rules. Or if you want to have the same facial sliders as TF2/L4D2 models do.

As discussed earlier (two times), the controller block sets up your character's sliders and domination rules. To embed a controller block to your character's head,

1. Get a controller block file either from the sample HWM project (`tf2_head_controllers.dmx`).
2. Save it next to your character's .blend.
3. Select your character's absolute head mesh and in the Blender Source Tools' **Source Engine Exportables** panel, switch the **Flex Properties** to **Advanced**. Select your controller block file as the **Controller source**.

Now whenever you'll export the head, it will have the flex controllers defined in the controller block file.

If you skip this step, Blender Source Tools will autogenerate a controller for each base shape on your mesh (and no domination rules).

4.2.4 Some QC stuff

Since all controller and shape data is embedded in the model's DMX file, there's no need for anything flex related in the QC. Here's the QC file for the sample project:

```
$modelname "test_hwm_head.mdl"

// Studiomdl wants relative correctors!
$model head "head_rel.dmx"

$sequence idle "head_rel.dmx"
```

If you have eyes on your character, declaring them in the QC file is similar to SMDs/VTAs:

```
$model head "../parts/dmx/liz_head_bas_rel.dmx" {
        eyeball righteye bip_head 1.42489 3.57888 67.01199 bas_eye_r 0.868 2 liz_bas_skin 0.8
        eyeball lefteye bip_head -1.42489 3.55873 67.01199 bas_eye_l 0.868 -2 liz_bas_skin 0.8
        localvar %dummy_eyelid_flex
        flexcontroller eyes range -45 45 eyes_updown
        flexcontroller eyes range -45 45 eyes_rightleft
}
```

For further reference use Elizabeth's source QC files.

4.2.5 Exporting and compiling

Set the **Export Path** and the **Engine Path** as usual. Add a few shapekeys on your character's head mesh.

If set up correctly, the Blender Source Tools' UI panel will look like fig. 24.

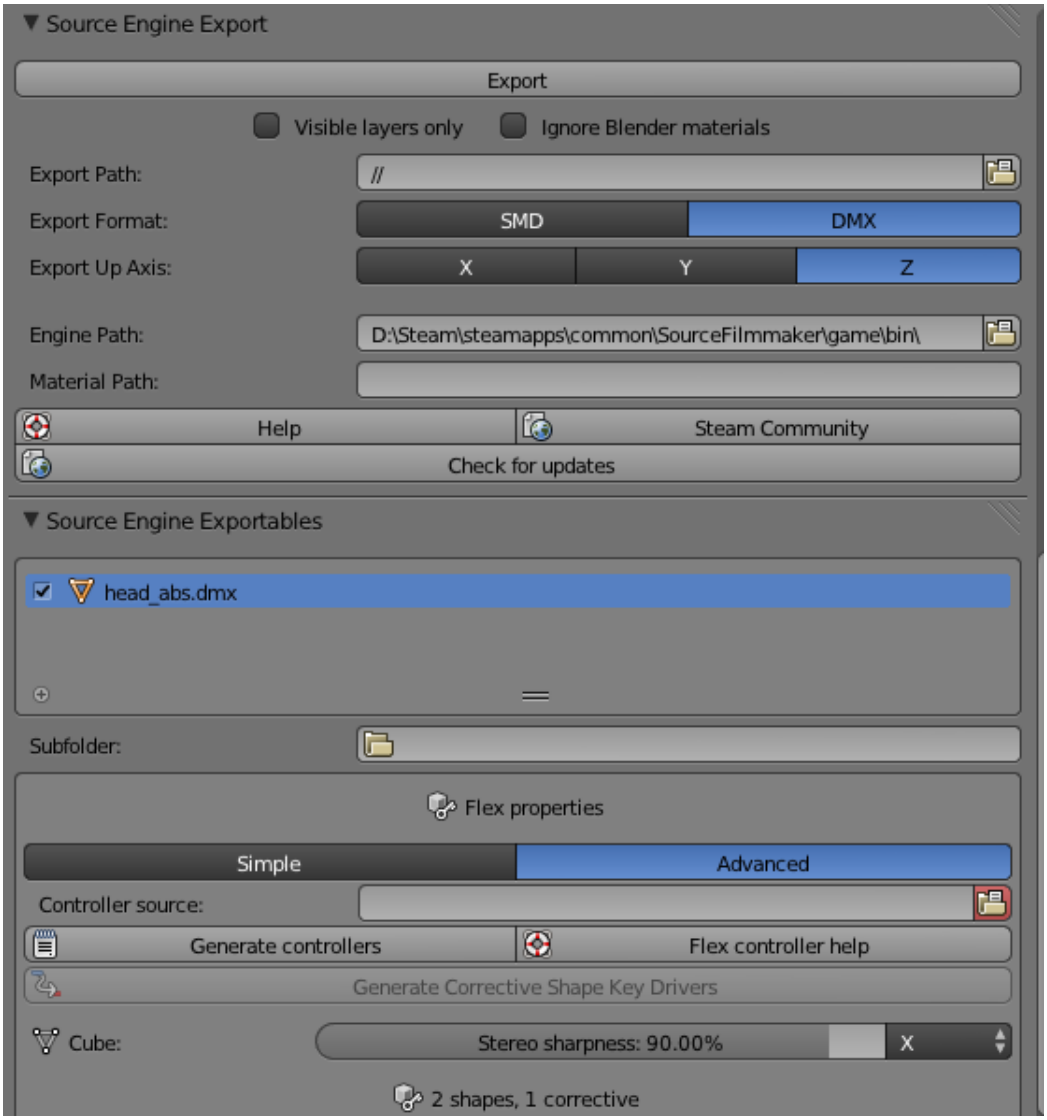

Figure 24: Blender Source Tools initial setup for HWM exporting (the sample project)

Now we're ready to preprocess and export your character's mesh for the first time!

1. From the Spacebar-menu, run the *HWM: Preprocess Mesh for Export* operator. If the setup was correct, Blender will inform you it *"Successfully updated head_rel..."* If an error occurs, check the Blender console for the cause. The HWM Toolkit discards the meshes without shapekeys, so be sure to add at least one.
2. You'll see a new `head_rel` mesh appearing in your file. **Export it to DMX** with Blender Source Tools.
3. Create a basic QC for your character. You must compile the mesh with relative correctors, not the other way around!

```
$model head "head_rel.dmx"
```

4. Compile the QC file.

If everything went smoothly, your character is now compiled, and the pipeline is working. Now you can start sculpting shapes.

4.3 Sculpting the shapes

Where to start?

Well, I don't know how many times I've said this, but once again, start by selecting the mesh with **absolute** correctors. Always do all your shape work on the absolute mesh and then update the relative mesh.

Start with the base shapes. All base shapes (for TF2/L4D2-style facial controllers) you'll need to sculpt are listed below.

BlowNostril
BrowOutV
CheekV
ClenchJaw
CloseLidLo
CloseLidUp
CompressLips
DeflateCheek
Dimple
Frown
FunnelLipLo
FunnelLipUp
InflateCheek
InnerSquint
JutJaw
JutUpperLip
LipCnrTwst
OpenJaw
OpenLips
OpenLowerLip
OpenUpperLip
OuterSquint
Platysmus
PressNose
PuckerLipLo
PuckerLipUp
PuffLipLo
PuffLipUp
RaiseBrowIn
RaiseChin
ScalpBack
ScalpForward
SlideJawL
SlideJawR
SmileFlat
SmileFull
SmileSharp
SneerNose
SuckJaw
SuckLipLo
SuckLipUp
SuckNostril
TongueBack
TongueCurlDown
TongueCurlUp
TongueLeft
TongueOut
TongueRight
TongueV
WrinkleNose

When the base shapes are ready, you can move on to corrective shapes, mixing them and adjusting the mixes. Don't forget to use *Soft Blend from Shape*, the *W-O Vertex Smooth* and all your Blender knowledge.

The Other `dmxedit` Pipeline puts a lot of pressure on you when doing base shapes — if they're bad, most corrective shapes will be bad too. No such thing will happen here, so feel free to make un-ideal shapes. You will be 100% able to adjust each one later whenever you want without breaking anything at all. Liz's shapes have been polished and adjusted and added constantly from May to Febraury and no shape or animation was ever lost.

Continue with the corrective shapes. For the sake of usability, here's the Important Corrective Shape List from earlier:

FunnelLipLo_FunnelLipUp
PuckerLipUp_PuckerLipLo
PuffLipUp_PuffLipLo
OpenJaw_OpenLips
OpenJaw_OpenUpperLip_OpenLowerLip
OpenJaw_OpenUpperLip_OpenLowerLip_Platysmus_SmileFull
OpenJaw_OpenLowerLip_Platysmus_SmileFull
OpenJaw_OpenUpperLip_Platysmus_SmileFull
OpenJaw_OpenLips_Platysmus_SmileFull
OpenJaw_OpenLowerLip
OpenJaw_OpenUpperLip
OpenJaw_Platysmus_SmileFull
OpenJaw_Platysmus
OpenJaw_SmileFull
OpenJaw_FunnelLipLo_FunnelLipUp
RaiseChin_CompressLips
OpenLowerLip_PuckerLipUp
OpenUpperLip_PuckerLipLo
OpenLowerLip_PuckerLipUp_OpenJaw
CheekV_CloseLidUp
InnerSquint_CloseLidUp
InnerSquint_OuterSquint
InnerSquint_OuterSquint_CloseLidUp
InnerSquint_OuterSquint_CloseLidLo
SmileFull_Platysmus

These shapes are your second priority task. After sculpting them, you can continue to any other necessary corrective shape. There is no (sane[22]) upper limit on corrective shapes.

4.3.1 Using reference

Your best reference for the base shapes are the TF2 models. Import the closest-looking mercenary to your scene and try replicating his shape keys (the base keys only — mercs' corrective shapes are in relative mode) on your character. Liz, for example, has followed the Heavy in his expressive facial expressions.

Of course, copying blindly is a bad idea (the heads and art styles and character's personalities can be different). Think about what do these shapes *mean*. If you do that, you'll end up with a good outcome around 99% of the time. Put a mirror near your PC and make funny faces. This way, it's easy to figure out where to put the limits.

Sculpting corrective shapes is mostly based around "fix whatever looks shitty, repeat forever". If you're really unsure about how a particular shape should look, launch `hlmv` and open up an existing model, and make it show the needed shape by adjusting sliders. Think about what the shape is trying to convey.

I advise to use Elizabeth's sources as a technical reference[23] and the Mercenaries as an art reference. Obviously, the facial shapes created by professional Valve-tier 3D artists are better than ones created by an second-year electronics engineering student.

4.3.2 The iterative pipeline

After all we've learned, let's recap the HWM shape sculpting process.

[22] Presumably around 16000 shapes.
[23] Don't forget about The Lazy HWM Blarbface, too!

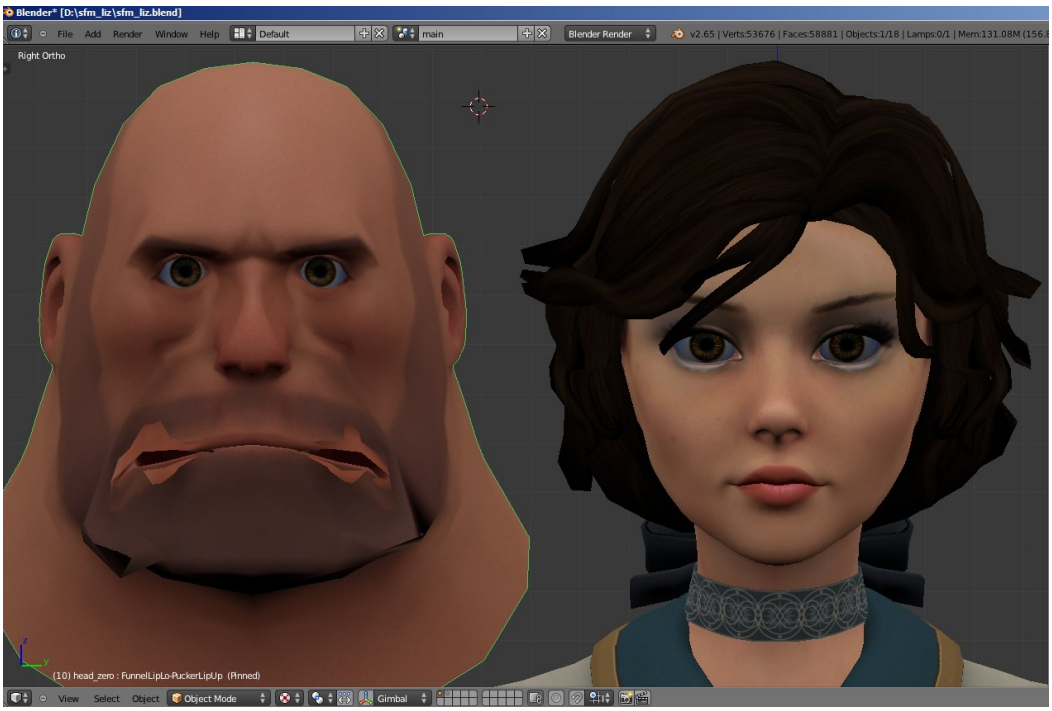

Figure 25: Aren't they just totally similar? (A 2.65-old screenshot!)

Preparations

1. Import the HWM Toolkit to your scene with the script from earlier. Don't forget to **Register** the script to run at Blender startup.
2. Name your character's head mesh `'head_abs'`. Don't forget to have at least one shape key on your character's head.
3. Set up the custom Controller Block.
4. Make the QC file, setup the export paths, etc.
5. Do a test export and compile.

Now, being sure everything's cool, you can move to iterating your character.

The pipeline

1. Sculpt, sculpt, sculpt the base shapes on the absolute mesh. Use TF2 models as the reference.
2. Sculpt corrective shapes on the absolute mesh. Use *New Shape from Mix* and *Soft Blend from Shape* operators. Start with the More Important corrective shapes. Don't forget you can mix corrective shapes to create new corrective shapes too.
3. Add new domination rules or sliders as you wish. Like `EyesWideOpen` on Liz or `hideCig` on The Spy. Refer to the earlier sections of this document for examples and instructions
4. Want to test? Hit Spacebar and run the *HWM: Preprocess Mesh for Export*. This will update the mesh with relative corrective shapes.
5. Export the relative corrective shapes mesh and compile the model. Run `hlmv` and play with the sliders.
6. Write down the combinations that "break" the face (fig. 26) and fix them with a corrective shape or a domination rule, where possible. Don't forget how controllers

Figure 26: `Platysmus` + `OpenJaw` + `OpenUpperLip` + `OpenLowerLip` + `SmileFull`. Could use a corrective shape here...

relate to the shapes. A bunch of Controller-To-Shape "conversion" examples are in the table 2.

7. Go to 1. Repeat until you[24] are satisfied.
8. Are you satisfied? Release the model, we want to animate faces too!

Sliders' values	**Active shapes**
`LipsV` $= -1$, `LipLoV` $= +1$.	`CompressLips` + `OpenLowerLip`
`LipsV` $= +1$, `LipUpV` $= -1$, `Smile` $= 1$, `Smile_multi` $= -1$	`OpenLips` + `JutUpperLip` + `SmileFlat`
`FoldLipLo` $= -1$, `FoldLipUp` $= +1$, `Platysmus` $= 1$	`SuckLipLo` + `FunnelLipUp` + `Platysmus`
`BrowOutV` $= 1$, `BrowInV` $= -1$	`BrowOutV` + `RaiseBrowIn`
etc	

$+1$ means "right" for two-way sliders.

Table 2: Sliders' values and active shapes on a TF2-style face

4.4 The HWM Toolkit short reference

The HWM Toolkit is very incomplete at the moment, only allowing for minimal HWM authoring support. However, some of the more important (and stable) features can be used and are documented here.

Toolkit is separated into a bunch of Python modules. Interesting functions from them are documented below.

Unfortunately, I didn't have a lot of time to spend with this and it shows.

[24] Your (inner) animators.

4.4.1 `op_softblend.py` — The *Soft Blend from Shape* operator

This module provides the *Soft Blend from Shape* operator. It uses a half of the HWM Toolkit's powers so take a look at its code to get acquainted.

The UI needs polish. Still a usable and useful operator.

4.4.2 `hwm.py` — High-Level HWM Tools

`PreprocessMesh(meshName, preprocess_script_file = None)` This function takes the `mesnName` object and preprocesses it into the form digestible by `studiomdl` (after exporting from Blender). Doesn't do anything if there aren't shapekeys on it. If supplied with `preprocess_script_file`, will execute its contents. If not, will just convert all corrective shapes to relative.

The idea behind the preprocess script is that you can generate some of the corrective shapekeys algorithmically, using the commands from `shapescripting.py`. This isn't the case with Liz — all her shapes were done manually — so while preprocessing scripts are implemented, this feature isn't tested at all.

To save the user from data corruption by launching this function on an incorrect mesh (*there is no algorithmic way to distinguish between an absolute and relative corrective shape!*), `hwm.PreprocessMesh` requires that the absolute mesh's name ends with `'_abs`', e. g. `liz_head_abs`.

`RebuildAbsoluteMesh(mesh_in)` Switches the corrective shapes to absolute on a given mesh (by handle). Useful for working with meshes from `sourcesdk_content/tf/modelsrc` — mercs' heads have the corrective shapes in relative mode.

This module uses the other half of the HWM Toolkit's power.

4.4.3 `shapetools.py` — Shapekey Management

`FindShapeKey(mesh, name, exact_mode = False)` — Finds a shapekey named `name` on the mesh `mesh` (by handle). If `exact_mode = True`, `FindShapeKey` will distinguish `A_B` and `B_A`. If none is found, returns `None`.

`AddShapeKey(mesh, name, overwrite = False)` — Adds a shapekey named `name` on the mesh `mesh` (by handle). If `overwrite = True`, `AddShapeKey` will delete the old shapekey if it had existed. Uses non-exact matching internally.

`CopyShapeKey(shape_in, shape_out)` — Copies the `shape_in` shape key to `shape_out` by indexes. Won't work if their meshes aren't duplicates or the same.

`RemovesShapeKey(mesh, name)` — Finds a shapekey named `name` on the mesh `mesh` (by handle) and removes it, if it existed. Uses non-exact matching internally.

`YeildSubShapeNames(name)` — Generates sub-shape for the specified shape name.

`name` value	Generated strings
`"A"`	`"A"`
`"A_B"`	`"A"`, `"B"`
`"B_A"`	`"A"`, `"B"`
`"A_B_C"`	`"A"`, `"B"`, `"C"`, `"A_B"`, `"B_C"`, , `"A_C"`
etc	

`Interp(mesh, vtx_weight_dict, shapekey_in, shapekey_out, amount)` — Takes the `shapekey_out` on `mesh` and interprets the mesh vertices pointed by the `vtx_weight_dict` towards `shapekey_in` by a normalized `amount` (everything is referenced by handles).

In `vtx_weight_dict`, the keys must be the vertex indexes and the values must be normalized "selection" weights. 1.0 equals to a "hard" selected vertex which will be fully affected, 0.0 equals to a completely non-affected vertex.

To better grasp what `Interp` does, here's two formulas:
`Diff = TgtVertexPos - OldVertexPos`,
`NewVertexPos = OldVertexPos + Diff * amount * SelectionWeight`

Selections can be created and edited using `selections.py`.

`Add(mesh, vtx_weight_dict, shapekey_in, shapekey_out, amount)` — same as before, but the formula is a little different:
`Diff = OldVertexPos - BaseVertexPos`,
`NewVertexPos = OldVertexPos + Diff * amount * SelectionWeight`

`BaseVertexPos` means the vertex's position in the base state (*Basis*).

`Corr_AbsToRel(mesh_in, mesh_out, shapekey_in_abs, shapekey_out_rel)` — Converts an absolute corrective shape pointed by `shapekey_in_abs` to relative mode, writing it to `shapekey_out_rel`, appropriately checking for underlying corrective shapes on the `mesh_in` mesh. The underlying corrective shapes are assumed to be relative.

`Corr_RelToAbs(mesh_in, mesh_out, shapekey_in_rel, shapekey_out_abs)` — Converts a relative corrective shape pointed by `shapekey_in_rel` to absolute mode, writing it to `shapekey_out_abs`, appropriately checking for underlying corrective shapes on the `mesh_out` mesh. The underlying corrective shapes are assumed to be relative.

ValidateShapeNames(mesh) — Validates that the mesh's shape key names are compatible with Source Engine. Alphanumerics and underscores are permitted.

CheckForRedundantCorrectives(mesh) — In Source, there's no difference between `CloseLidLo_CheekV` and `CheekV_CloseLidLo`. This function checks for ambiguous shapekey names on the mesh.

4.4.4 `obtools.py` — Object Tools

`FindObject(Name)` — Returns an object of name `Name`, `None` if doesn't exist.

`DeleteObject(Name)` — Deletes a specified object, wiping out data and removing users. Has some issues. For example, it will fail to delete an object that is a target of a camera in the scene etc.

`DuplicateObject(fromName, toName, overwrite = True)` — Duplicates the `fromName` object, if one exists, names the new object `toName`. If `overwrite` is specified and there is a `toName` object, deletes it before duplicating.

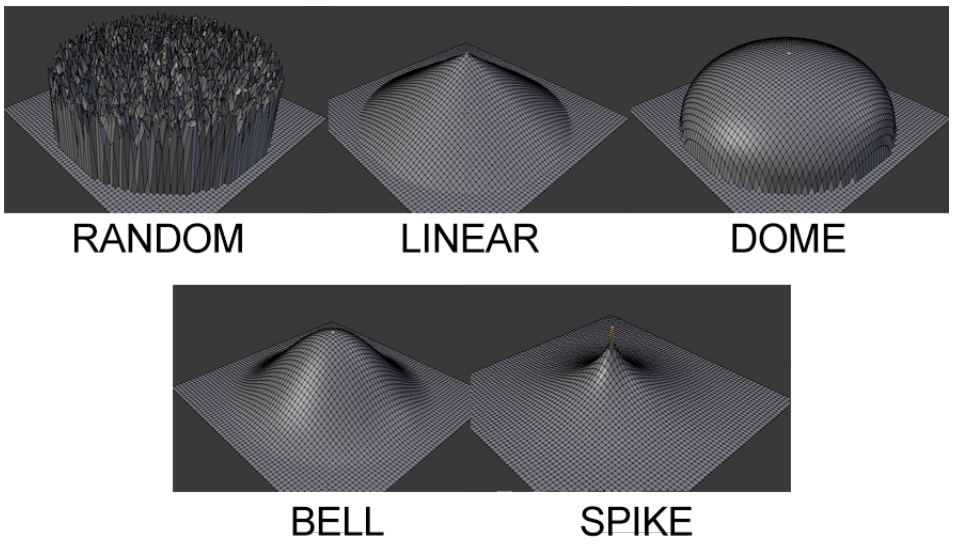

Figure 27: Different soft selections built with `BuildSoftSelection` using different interpolators. The central highlighted vertex is "hard-selected"

4.4.5 `selections.py` — Vertex selection tools

These functions are for building selections to use in `shapetools`.

A selection is a dictionary. Keys are vertex indices, values are normalized selection weights. A vertex with weight 1.0 is "hard-selected", a vertex with 0.0 (or absent from the keys) is not affected. This is for implementing soft selection.[25]

Remember this has nothing with Blender editmode selections.

These functions work on `BMesh` objects. To get a `BMesh` from a regular mesh, use this snippet:

```
bm = bmesh.new()
bm = bmesh.from_edit_mesh(regular_mesh.data)
```

`Select(vtx_index_list)` — returns a `vtx_weight_dict` with vertices in `vtx_index_list` hard-selected.

`SelectIntersect(vtx_weight_dict_a, vtx_weight_dict_b)` — intersects the given selections (discarding any soft-selected vertices) and returns the new selection.

`SelectAdd(vtx_weight_dict_a, vtx_weight_dict_b)` — adds the the given selections (discarding any soft-selected vertices) and returns the new selection.

`SelectSubtract(vtx_weight_dict_a, vtx_weight_dict_b)` — subtracts (`vtx_weight_dict_a - vtx_weight_dict_b`) the given selections (discarding any soft-selected vertices) and returns the new selection.

`SelectAll(bmesh_in)` — sugary sugar that returns the selection dictionary of all vertices in the `bmesh_in`.

`BuildSoftSelection(bmesh_in, vtx_weight_dict, falloff_distance, falloff_type)` — makes a soft selection around the hard-selected vertices specified by `vtx_weight_dict`, with falloff proportional to `falloff_distance`. Different falloff interpolation (`falloff_type`) can be selected: `'SPIKE'`, `'BELL'`, `'DOME'`, `'RANDOM'`, `'LINEAR'` (fig. 27).

[25] aka Proportional editing once again.

`SelectMore(bmesh_in, vtx_weight_dict) & SelectLess(bmesh_in, vtx_weight_dict)` — analogous to using **Grow Selection** and **Shrink Selection** in Blender.

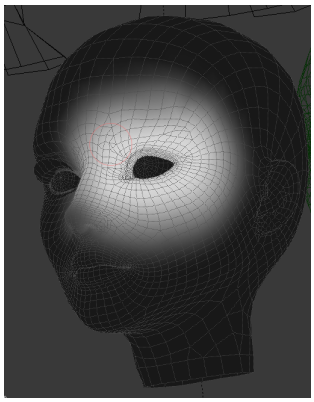

Figure 28: A mesh colored proportional to the soft-selection weight. A loop around the eye was hard-selected as a start `BuildSoftSelection`

4.4.6 `facerules.py` — Controller Block editing tools

Not implemented yet. Supposed to make DMX Flex Controller Block editing less stupid.

4.4.7 `shapescripting.py` — the `dmxedit` emulator

These functions are designed to emulate Valve's `dmxedit`. Since Liz doesn't use them in any way, they're far from being tested, but they are implemented and can sort of work.

Since they're identical to `dmxedit`'s Lua interface, you can learn about them by launching `dmxedit` with `-h` command line parameter.

5 Typical problems and their resolutions

Problem	Resoultion
Blender closes after exporting a mesh	This is a known bug for pre-1.10 versions of Blender Source Tools. Fear not though: your file has been exported correctly! Update to Blender Source Tools 2.0.
`studiomdl` says *Too many influences per vertex, aborting!*	Remove extra influences from the meshes by selecting them and running `armature_limitVtxGroups.py` in Blender. This happens even if the offending mesh is included as a `$sequence` (so obvious, isn't it?).
`studiomdl` says *Model is too large, cutting into multiple models!* and crashes thereafter.	This is a Blender Source Tools bug. Try updating to the latest Blender Source Tools.
`studiomdl` says *Too many flex controllers, max 128!* and aborts the compilation.	By default, older versions of Blender Source Tools will generate a controller for every shape present on the mesh if no custom controllers are specified. So this means the head mesh was exported without the custom controller block. Check it in the Flex Properties tab in Blender Source Tools.
Weird flex controllers appear in the compiled model, like `OpenUpperLip`, `CloseLidLo`... Where's my `CloseLid_multi`?	Newer versions of Blender Source Tools don't generate controllers for corrective shapes by default, so your model doesn't trigger the previous problem. Nevertheless this means you didn't embed the custom controller block and Blender Source Tools just generated a controller for every shape.
Blender closes after I use the *Liz: Export Preprocessed Heads* operator.	Known bug. Your heads have been exported, just restart Blender.
Blender Source Tools throw errors about UUIDs when exporting a mesh with shape keys.	There's an error in your Controller Block File.
My shapekeys are corrupted! 	This is a bad Blender bug. It was fixed around 2.68 so update to the latest Blender. This corruption is irreversible so be sure to have a backup next time.