Head X Synthesis Engine 0.88 OSE

Head X
Open Source Edition

Capability & Interface Guide
Martin H. Luerssen @ Flinders University, Adelaide, Australia

WHAT IS HEAD X?

- It's a Talking Head
  - Head X converts text into speech and facial expressions
- It's an Embodied Conversational Agent
  - Alice XS: Optional Program D plug-in for AIMA-based dialogue management
- It's highly configurable and can be set up to suit your requirements
  - Integrate your own 3D models, change speech and behavior
- It's Open Source, under the New BSD license
  - Free for commercial and non-commercial use
- It can be directly controlled from within your own program, without any need to recompile or even see the Head X source code
  - Supports multiple interfaces for easy, flexible control
- It's a product of the Flinders University Artificial Intelligence Laboratory
  - Developed as part of the Australian ‘Thinking Head’ ARC/NHMRC Special Research Initiative (2007-2012)

RUNNING HEAD X

Download Head X from: https://www.dropbox.com/sh/r25pouic2ifn46e/JBB2WIcG_q

Head X 0.88 (R88) is contained within HeadXOSE_R88.zip. Earlier or later versions of the Head may be contained in archives with different Rxx numbers. Until a new guide is published, this guide should be applicable to R88 and later versions of the Head, although we suggest you carefully check any release notes for important changes.

No formal installation is required, as the Head X archive contains all the necessary runtime libraries. Just unzip it to a location of your choice. On Windows XP systems, please ensure that this location is one that you can subsequently write to (i.e., not a CD or DVD or write-locked drive) – the Head will not talk otherwise.

If you are looking for Head X source code, you may obtain the newest code directly from the Google code Subversion server at http://head-x.googlecode.com/svn/trunk/. It comprises a Microsoft Visual Studio 2010 SP1 solution, which relies on a number of (excluded) libraries:
  - FreeType (version 2.3.9 is assumed)
  - FTGL (2.1)
• GLEW (1.7)
• Libjpeg
You will need to obtain and compile these separately; they belong to public projects that are easy to find. Contact the author for more information and assistance on this matter, if required.

**Launching Head X**
The `.cmd` scripts in the top-level directory of the archive make it easy to launch Head X in various configurations. You may wish to edit these or add your own to suit your specific requirements.

• **Head X.cmd**
  o Launches Head X with a female character

• **Head X (Old Computer).cmd**
  o A configuration that is more compatible with old computers; please try this if you are having problems with the other configurations

• **Head X (Depth Peeling).cmd**
  o A configuration that uses Depth Peeling to achieve high-quality hair rendering. Requires a DirectX 10 capable graphics accelerator – Nvidia cards are strongly suggested, as other hardware may not be compatible

• **Head X (Test).cmd**
  o A configuration suitable for testing and debugging: a simple face, no subtitles, and no idle animation

If you have installed the *Alice XS* dialogue manager (see the next section for instructions on this), the following additional scripts will be available as well:

• **Head X +DM.cmd**
  o Launches a fully configured Head X with a female character ('Faith'), plus the Alice XS dialogue manager using a slightly edited Alice AIML database

• **DM.cmd**
  o Launches the Alice XS dialogue manager with the Alice 'Faith' AIML database

• **DM (Stelarc).cmd**
  o Launches the Alice XS dialogue manager with the Alice 'Stelarc' AIML database

Alternatively, you can start Head X directly from the command-line. The Head X executable is located in the Bin subdirectory.

**Usage:**
```bash
HXPlayer.exe [-m samples] [-d] [-l] [-s shader path] [startup config file]
```

• `-m` option: followed by the number of samples desired for multisample anti-aliasing
• `-d` option: enables Depth Peeling
• `-l` option: legacy mode, force disables shaders (for computers with old/bad drivers)
• `-s` option: followed by the path to the shader files, default is `../Data/Shaders`

following all of the above, you may additionally specify the configuration filename to load on start-up

**Head X Requirements**
• 2+ GHz single-core CPU, or 1+ GHz dual-core (or better) CPU
• 512+MB RAM
• 120+MB free hard disk space
• Discrete graphics card, or fast motherboard graphics
  o On certain, limited graphics hardware (e.g. Intel graphics) you may encounter delays in the start-up of the program or visual artifacts during execution
• Microsoft Windows XP, Vista, or 7
  o 32-bit or 64-bit
• Java runtime (1.6 or later)
  o Only required if you wish to use the Alice XS dialogue manager

**HEAD X (& ALICE XS) SUBDIRECTORIES**

- **Bin**
  - Head X + Alice XS executables
- **Data**
  - **Data/Face**
    - Essential 3D models
  - **Data/Hair**
    - Accessory models
  - **Data/Fonts**
    - Subtitle fonts
  - **Data/Detail**
    - Detail textures
  - **Data/Shaders**
    - OpenGL shaders
- **Config**
  - Head X configuration files
- **Config/AIML**
  - Alice XS dialogue scripts
- **programd**
  - Alice XS configuration + executables (Program D)

**TROUBLESHOOTING**

- If Head X crashes on start-up or crashes randomly, please ensure that you have up-to-date graphics drivers. There have been multiple reports of problems that have been traceable to outdated graphics drivers.
- If Head X is animating extremely slowly: do you have OpenGL acceleration available and enabled? This is likely to be an issue if you are within a virtual machine (VM).
- The default voice under Windows XP is quite bad; for adequate speech quality, a different voice needs to be installed.
- If Head X refuses to talk back to you in any configuration that requires the dialogue manager to be running, then the dialogue manager (AliceXS) may not have started properly (you should see a separate, minimized window for it). The likely culprit is an unusual or old (pre-1.6) Java configuration – please ensure there is a Java 1.6 or later JDK or JRE installed within the system default Program Files or Program Files (x86) directory.

**ALICE XS (DIALOGUE MANAGER)**

Alice XS is a dialogue manager included with Head X as a separate program. More precisely, it is an interface to Program D 4.6 (from aitools.org/Program_D), which in turn is a Java
implementation of an AIML interpreter (see www.alicebot.org). You can download AliceXS_R80.zip from the same place as Head X and unzip into the same directory as your Head X installation, which should add a few files to your Head X. Alice XS is licensed under the GNU GPL v2, and source code is hence also available on request.

Alice XS is automatically started together with Head X via the **Head X +DM** command script, which also ensures that Alice XS is shut down automatically with Head X (and vice versa). If the *Echo* switch in Head X is off (the menu item for this is **Process**→**Echo User Input**), all textual user input to the Head will be processed by Alice XS.

Alice XS starts Program D – and only produces a limited set of error messages if this should fail. Alice XS may also not generate any error message if Program D crashes (e.g., when encountering an infinite dialogue recursion). You should therefore ensure that Alice XS is running – it will display a minimized window with the title **Alice X chat engine** (see above). On slow computers, it can take a while for the dialogue database to be loaded; the Head will not respond to questions until this is done.

**Usage:** AliceBot.exe [-x] [port number] [program_D path]

- **-x** option: Alice XS will shutdown with Head X
  - (NB: the Alice XS version included with Head X must be run from the parent directory of where the executable is located)

Alice XS communicates with Program D via a socket. The default port number is 33333, with higher numbers automatically allocated if multiple Alice XS clients are running concurrently. You may need to tell your firewall (or will be asked by it) to let these connections through.

Alice XS comes with a modified version of the standard Alice database (scripted in AIML), representing a female Flinders University personality called 'Faith'. You can further modify the AIML files in **Config/AIML/faith**; and modify/add new ones in **Config/AIML/thead**. Additional Program D configuration files can be found in **Config/DM**, which specify some character background and the AIML files that are to be loaded.
Head X provides a diversity of ways in which to interact with it. Much of the functionality of Head X is exposed on each of these interfaces. The subsequent sections of this report provide instructions on how to access this functionality. You can quickly distinguish the interface to which each instruction applies by the colour and label of its container. **Commands** (for more on these, see page 9 and later) can be sent to Head X through multiple interfaces, so they are given a special container.

How to do it with the GUI menu

How to do it with XML
(excluding the enclosing `<headx>` tag)

How to do it with Synapse
(see Synapse section)

How to do it with Thinking Head Events

How to do it with Commands
**HEAD X GUI**

The main window of Head X offers access to many useful features of Head X, either through the various menu items or directly by mouse and keyboard interaction with the Head. Text to be processed can be sent to the Head through the Command Window (which can be opened from the menu: Process→Command Window) or the on-screen Command Prompt (which can be enabled through the menu as well: Process→Text on Screen→Command Prompt – it is enabled by default in the standard Head X configuration).

**CONTROLLING HEAD X WITH THE MOUSE**

You can move, zoom, and rotate the Head directly with the mouse. You can also change the lighting and the position of the subtitle this way. The effect of your mouse movement depends on (1) the selected item in the Interface→Mouse menu, (2) the mouse button currently held down, and (3) whether the SHIFT key is pressed while doing so (see the next page). The outcome is visualized immediately, with the status display providing detailed information about the parameters modified this way.
**CAMERA CONTROL**

<table>
<thead>
<tr>
<th>Mouse button</th>
<th>SHIFT key not pressed</th>
<th>SHIFT key pressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Rotate Head on X and Y-axes</td>
<td>Zoom</td>
</tr>
<tr>
<td>Middle</td>
<td>Rotate Head on Z-axis</td>
<td>Squash (aspect ratio/mirror image)</td>
</tr>
<tr>
<td>Right</td>
<td>Move Head on X and Y-axes</td>
<td>Move Head on Z-axis (perspective)</td>
</tr>
</tbody>
</table>

**LIGHT CONTROL**

<table>
<thead>
<tr>
<th>Mouse button</th>
<th>SHIFT key not pressed</th>
<th>SHIFT key pressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Rotate Light 1 on X and Y-axes</td>
<td>Rotate Light 2 on X and Y-axes</td>
</tr>
<tr>
<td>Middle</td>
<td>Ambient intensity of Light 1</td>
<td>Ambient intensity of Light 2</td>
</tr>
<tr>
<td>Right</td>
<td>Diffuse/specular intensity of Light 1</td>
<td>Diffuse/specular intensity of Light 2</td>
</tr>
</tbody>
</table>

**SUBTITLE POSITION**

<table>
<thead>
<tr>
<th>Mouse button</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Move the subtitle on X and Y-axes</td>
</tr>
<tr>
<td>Right</td>
<td>Move the subtitle on Z-axis (scale)</td>
</tr>
</tbody>
</table>

**WINDOW POSITION**

<table>
<thead>
<tr>
<th>Mouse button</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Move the window in the direction of mouse movement</td>
</tr>
<tr>
<td>Right</td>
<td>Resize the window along the dimensions of mouse movement</td>
</tr>
</tbody>
</table>

You can also set any of the above through the XML configuration.

```xml
<action mouse="X" shortcuts="yes"/>
```

where X can be any one of camera, light, subtitle, window, and lock (ignore any mouse input)
**Window Control**

The main Head X window can be displayed without a menu and/or full-screen, where full-screen is equivalent to a maximized (frame-less) window without a menu. Additionally, you can ensure the window is “Always on Top”, that is, never occluded by any other windows. For a widget-like experience, you can switch the window background to fully transparent, so that the desktop (or any other application) is visible behind the head. This requires a very fast graphics accelerator with up-to-date graphics drivers: with inadequate drivers, the Head may turn invisible as well!

Since you cannot access menu items while there is no menu, any maximized, transparent, and/or otherwise menu-less window can be restored back to its default form by simply pressing the Esc key.

```
<render>
  <window maximize="no" menu="yes" border="no" x="0" y="0" width="500" height="500"
          top="no" transparent="no" title="My Awesome Head"/>
</render>
```

**XML**

Maximize window size (and remove frame)

Show the main window menu (default is yes)

Show the frame of the window (default is yes)

Initial window position (system default used otherwise) (-1 to disable)

Keep the window top-most in front of all others (default is no)

Render Head X with a transparent window and background (as shown in screenshot below)

New title for window (default is "Head X")
HEAD X COMMANDS

You can send text to the Head by...

• entering it into a Command Window (multiple windows can be open simultaneously)
• or into the on-screen Command Prompt;
• by sending it to the Command Synapse of the Synapse interface,
• or by sending it as a TH_EVENT_DM_RESPONSE (for EM 1.0) or CHATBOT_ANSWER (for EM 2.0) event to the Event interface

Any such text will be translated into audiovisual speech, excluding anything in { curly brackets }, which we refer to as commands.

• E.g. {EMOTE anger} I don't like this!

Anything inside the {...} tag is interpreted. If it is unknown, the content will simply be ignored. Text outside of commands will be passed unmodified to the selected text to speech (TTS) system, and may also include SAPI and/or SSML tags to modify the resulting speech output.

For most {...}-commands, the location of the {...} in the text will control the timing of the command. The command is typically only executed once the speech output actually reaches the location of the command in the spoken text.

There are a large number of {...}-commands, covering a major part of the functionality of the Head, including most of what you can do through the GUI menu. Many of these commands are listed in subsequent sections, visually enclosed in red boxes. However, several abilities of the Head are available only as commands – these include BREAK (as well as the related PAUSE), EMOTE, and SOUND.

COMMAND: BREAK

The BREAK command “breaks” the input stream into segments, with each segment interpreted separately (as if you entered each segment on its own).

• E.g. It is {BREAK 0.2} exciting, {BR} is it not?

The TTS engine is engaged separately for each segment, and {...}-commands are also processed separately.

The use of BREAK will typically cause a noticeable pause between segments, as the speech and command processing takes time. You can deliberately extend the BREAK pause by a certain amount of time, specified as an argument to the command.

Usage: {BREAK delay} or {BR delay}
where delay specifies pause before processing next segment (in seconds; default is 0.0).

You can ask Head X to automatically insert a BREAK at each full stop (and equivalent punctuation).

Process→Auto-Break on Punctuation

GUI
**COMMAND: PAUSE**
The BREAK command can be used to add pauses to speech, but as each segment is processed separately, using BREAK in this way leads to changes in prosody that may not be desirable. If you wish to add a pause to speech without affecting prosody in any way then you should use the PAUSE command, which achieves this goal. PAUSE only works if you are using a SAPI voice that supports the <silence> tag; it is ignored for other voices, including all MARY voices.

**Usage:** {PAUSE delay}
where delay specifies the desire pause in speech (in seconds; default is 1.0).

---

**COMMAND: EMOTE**
The EMOTE command triggers a facial expression or movement.

- E.g. {EMOTE happy} Good day, friend!

An expression name (in this case happy) is provided as an argument to the command. The allowed expressions and their names are defined by the <expression> tags within the Head X XML configuration (see pages 15 and 16 for more detail on this). If the configuration contains multiple expressions with the same name, EMOTE will randomly choose one of these. The configuration file expressions.xhtml, which is located in the Config directory with all other such files, contains a default set of expression definitions and is included by all configurations in this Head X release.

A second argument can be specified to extend the peak (point of highest amplitude) of the expression by a given amount of time, assuming that the expression is set to bounce (as is default for most expressions).

**Basic Usage:** {EMOTE name} or {EM name}
where name specifies the name of the expression to animate (rise+fall of the expression).

**Extended peak:** {EM name delay}
where delay is the wait time (in seconds) at the expression peak (after rise, before fall) (default is 0.0).

**Expression Timing**
At what time the expression starts playing depends on the selected speech engine.

- **SAPI TTS:** The expression begins to play at the start of the next word after {EMOTE...} (when that word is spoken). If there is no subsequent word, the {EMOTE...} plays at the segment end (after speech).
- **MARY TTS:** If the {EMOTE...} command is specified at the start of a segment, it will play at the start of the segment (at the start of speech). Otherwise, it will play at the end of the segment (after speech).
If the use of MARY is expected or possible (and the use of SAPI is not excluded), make sure to define `{EMOTE...}` always either at the very start or the very end of a segment! The above is also true for any other commands that are word-position sensitive.

**Expression Composition**

**Switching:** `{EM +name} {EM –name} {EM =}
- Toggles expressions on (+) or off (-) *(for bounced expressions only, see page 15/16)*
  - (+) and (-) can be stacked to increase or decrease expressions
    - Negative expressions are possible
  - (=) resets all switched expressions to zero
- E.g. I am `{EM +grin} happy to see you {EM –grin}`

**Concatenation:** `{EM name delay +name -name...}
- Plays expressions in sequence
- E.g. I am `{EM +grin +wink} happy to see you {EM = blink}`

**Concurrency:** `{EM name delay | +name | -name...}
- Plays expressions in each [...] block in parallel
- E.g. `{EMOTE happy | fear | surprise}
  - animates all three expressions simultaneously

**Command:** `SOUND`

Head X can play back any WAV sound file, even while speaking, which is specified by the SOUND command. Sounds will start playing with the same timing logic as expressions; speech will continue as it normally does, irrespective of the length of the sound.

**Usage:** `{SOUND bing.wav}
where `bing.wav` could be a path to any WAV (Waveform Audio File Format) file. If no absolute path is specified, it is assumed the file exists in the current working path of Head X (see the last page of this document on how to determine the working path).

If you do not wish your custom sound and the speech to play concurrently, it is suggested that you use the PAUSE command to stop speech for sufficiently long enough to play the sound.

- E.g. This is a `{SOUND beep.wav} {PAUSE 1.5} problem.
**EXTERNAL PROCESSING**

Any command input to Head X from the GUI is either...
- Processed externally (expecting new commands from an outside source, e.g. a DM)
- Processed internally, and immediately translated into audiovisual speech

The latter mode is enabled by the **Echo User Input** option.

**PROCESSING**

- **User input**
  - **External processor** (e.g. Alice XS)
  - **External processing**
    - **Processed externally** (expecting new commands from an outside source, e.g. a DM)
    - **Processed internally**, and immediately translated into audiovisual speech

The latter mode is enabled by the **Echo User Input** option.

**QUICK COMMANDS**

Up to 10 text/command sequences can be defined within an XML configuration file. They are accessible from the GUI (via menu or keyboard shortcuts) and can also directly be triggered via a special command.

**SPECIFICATION**

```xml
<quick>
  <id0 label="Click me"
       command="Hey! \{EMOTE grin\}"/>
  <id1 ... (up to id9)
</quick>
```

**EXECUTION**

- **Action** → **Quick Commands** →...
- **Press CTRL-#** where **# is the numeric ID (0-9) of the Quick Command (as defined in the HXML file)**

**COMMAND**

```text
{quick #}
```

where **# is the numeric ID of the Quick Command**

- e.g. `{quick 1}`
HEAD X CONFIGURATIONS

Files ending in .hxml (or .hxm) files are assumed to be XML files that contain complete or partial configuration information for Head X. Every modifiable parameter of Head X can be specified in such an HXML file. When you load a configuration file, current Head X parameters are replaced with the parameter values in the HXML file. If a parameter value is not specified, the previous value is retained (not reset – you have to clear the configuration to achieve this).

A configuration can be explicitly reloaded from the last configuration file that was loaded. This is convenient, for example, if you are currently editing and testing the file.

Configuration loading, reloading, and clearing is only permitted before or after speech, not during speech. If you attempt to do so during speech, the action will be delayed until after speech.

File→Load Configuration and
File→Reload Configuration

(GUI)

{loadconfig file}
where file is the full path to an HXML file
  e.g. {loadconfig ..\Config\hxconfig.hxml}

(Command)

Instead of loading the HXML configuration from a file using the LOADCONFIG command, you can provide the configuration directly as an argument to the CONFIG command. (In doing so, you must omit the enclosing <headx> tag of your configuration, as CONFIG adds this tag automatically. This also occurs with other commands that accept XML configuration arguments.)

{config xml}
where xml is a XML configuration definition
  e.g. {config <render><window maximize="yes"/></render>}

(Parameter Loading)
You may wish to (re-)load only the parameter settings from the HXML configuration, without loading 3D models. This is much faster and can be done during speech with no (or minimal) delay. However, changes in certain 3D render options (peeling depth and transparent backgrounds) and the choice of .fg FaceGen file will be ignored as well.

File→Parameters→Load Parameters Only and
File→Parameters→Reload Parameters Only

(GUI)
As with CONFIG, the HXML configuration can also be passed directly as an argument to the PARAM command. Any file-loading (and some other) parameters will be ignored.

**RESETTING THE CONFIGURATION**
Clearing a configuration causes all parameters to be reset to internal default values (this also happens on start-up) and all files to be unloaded from memory.

**BASIC HXML STRUCTURE**

```xml
<headx>
  ...
  <import file="other.xhtml"/>
  ...
  <render>
    ...
  </render>
  ...
  <action>
    <!--INSERT COMMENTS LIKE THIS-->
    ...
  </action>
  <TTS>
    ...
  </TTS>
  <quick>
    ...
  </quick>
</headx>
```

- `<headx>` **required** enclosing tag
- `<import file="other.xhtml"/>` Insert contents of another HXML file
- `<render>` Window and 3D rendering options
- `<action>` Behaviour and interface configuration
- `<TTS>` Text to speech parameters
- `<quick>` Quick command shortcuts
**EXPRESSIONS**

Expressions that can be triggered by the `{EMOTE...}` command must first be defined in the configuration XML. For the configurations included with Head X, the basic set of expressions is recorded in the separate expressions.hxml file, but you can put them anywhere you prefer, or modify and extend the existing ones – whatever is appropriate for your particular needs.

The syntax for defining expressions is illustrated with an example below. Be aware that you can define multiple `<expressions>` within the `<action>` tag, and multiple `<bs>` and `<rotation>/<translation>/<scale>` within the `<expression>` tag. You are not limited to just one of each as in this example. (This is also true for idle expressions, as defined later.)

```
<action>
  <expression name="disappointed" bounce="yes">
    <bs name="Modifier: LookDown" amplitude="0.2" offset="0.0"
        duration="0.8" rampfunc="linear" />
    <rotation x="-0.08"
        offset="0.0"
        duration="0.8"
        rampfunc="linear"
        group="head" />
  </expression>
</action>
```

Name of the expression, as used for triggering via `{EMOTE...}`

(optional) Whether the expression should be subsequently "undone"/reversed (default is yes)

(optional) Delay before change is applied (in seconds; default is 0)

X-axis rotation in radians; `y` and `z` for `Y`- and `Z`-axis are also available

Proportion of the blendshape to apply to existing shape (default is 1, i.e. replace neutral expression completely with this one)

(optional) Duration over which change is applied (in seconds; default is 1)

(optional) Ramping (or blending) function to use (default is sigmoid sine; see below for more options)

A rotation is applied as part of this expression; `translation` and `scale` are also available

(optional) Which model group to limit this rotation to (default is empty, which denotes the root); only available for affine transforms, not blend-shapes

XML

Name of the blendshape, as defined in the model configuration

Delay before change is applied (in seconds; default is 0)
BOUNCED EXPRESSIONS
Expressions are typically assumed to include a rise (blending into the expression) and fall (blending out of the expression), i.e. the animation bounces back from its peak intensity. You can disable this behaviour by setting the bounce attribute to “no”, which can be helpful if you’re scheduling a complex, multi-stage expression animation. However, it is your responsibility in this case to ensure that the face returns to a neutral expression at the end of the expression.

If an expression is bounced, its specified duration is for the combined rising and falling phases. Otherwise, it includes only the rising phase (and may therefore seem to animate half as quickly).

RAMPING FUNCTIONS
By default, expressions are animated by blending the current expression with the new expression along a sigmoid sine trajectory. You can change this to a different pre-defined ramping function, as chosen from the following list:

- “linear” = linear blend \( x \)
- “root” = square root blend \( \sqrt{x} \)
- “square” = squared blend \( x^2 \)
- “half-sine” = sine blend \( \sin(x \times \pi/2) \)
- “inverse root” = inverse square root blend \( 1-\sqrt{1-x} \)
- “inverse square” = inverse squared blend \( 1-(1-x)^2 \)
- “inverse half-sine” = inverse sine blend \( 1-\sin((1-x) \times \pi/2) \)
- “sine” = sigmoid sine blend \( \sin((x-0.5) \times \pi+1)/2 \)
- “sinc” = sinc blend \( \sin((2x-1) \times \pi)/(2x-1) \times \pi \)
- “max” = immediate blend 1
**Idle Motion**

Head X includes a basic system for animating the Head even when it is not talking. It gives a semblance of life to the Head by randomly inserting eye blinks, eye brow movements, smiles, and head rotations into the animation stream. As with most Head X features, you have extensive parametric control over this. The default idle motion is blinking with limited head/eye rotation, as defined in `idle_expressions.hxml`.

**Process→Idle Behaviour**

```xml
<action>
  <idle motion="yes">  <!-- Enable idle motion -->
    <expression>
      name="blink"
      min_amplitude="1.0"
      max_amplitude="1.0"
      min_interval="0.0"
      max_interval="5.0"
      min_duration="0.1"
      max_duration="0.1"
      min_hold="0.0"
      max_hold="0.0"
      rampfunc="sine">
        <bs name="Modifier: Blink Left"
          amplitude="1.0"
          offset="0.0"
          duration="1.0"/>
        <bs ... />
        <rotation z="0.02"
          offset="0.0"
          duration="0.5"
          rampfunc="sine"
          group="shoulders"/>
      </expression>
      <expression name="blink" ...
      </expression>
  </idle>
</action>
```

Multiple idle expressions can be defined.

Each idle expression is defined by the blend-shapes and transforms that it animates.

Multiple idle expressions can have the same name; they will be randomly chosen as alternatives (and will never play in parallel).
3D MODELS

Head X can read meshes and materials from standard Wavefront OBJ (*.obj) 3D model files, which are easily exported from just about any 3D modeling tool. Head X will load the materials (*.mtl) file that may accompany the OBJ and will respect the texture and ambient/diffuse/specular lighting properties of each material. However, OBJ files cannot define animations. For Head X to express visemes and emotions, an OBJ file for each (blend-/morph-) shape of the Head must be loaded. The hxconfig_loadOBJs.hxml configuration offers an example of OBJ loading (of the MakeHuman model); the original OBJ files for this are available as HeadX_OBJModels.zip from wherever you downloaded Head X. They should be put in the Data/Models subfolder of your Head X installation.

Alternatively, models can be loaded from proprietary binary archives (typically with the extension .hx3) that can be exported from Head X – see the next section for more on this.
**MODEL MANAGEMENT**

You can specify 3D models as part of the main configuration, or add and remove models directly through the menu or through commands that you supply to Head X. The loadable items must still be defined as .hxml files (as with configuration loading), but only the 3D model information is extracted; all other parameters are ignored.

It is also possible to export all of the currently loaded models into a single, uncompressed .hx3 binary archive. This may simplify and accelerate model loading for complex models with many blendshapes.

---

**GUI**

`:File→Models→Add Model...`

and

`:File→Models→Remove Model→...`

and

`:File→Models→Clear All Models`

and

`:File→Models→Export All Models`

---

**Command**

`:loadmodel file`

where file is a full path to an HXML file

e.g. `{loadmodel ..\Data\Hair\HairPreppyBlonde.hxml}`

`:model xml`

where xml is an XML model definition

e.g. `{model <render> <model file="MidLengthMessyCap"/> <render/>}`

`:removemodel group`

where group is the name of a defined model group

e.g. `{removemodel hair}`

`:clearmodels`

removes all model accessories

---

**MODELS INCLUDED WITH HEAD X**

The Open Source edition of Head X comes with a 3D head generated by the Open Source tool **MakeHuman** (available from makehuman.org). OBJ files were produced for the different required facial shapes; the heads of all 3D model were then removed using the Open Source modeler **Blender** (see blender.org) and combined into a single binary file (**defmodel.hx3**), which is part of the binary release of Head X. The model is divided into a head and a hair group.

You may follow this process to obtain a different 3D head, or employ other tools and resources to animate a wide variety of human and non-human speaking characters. Please note that since
Head X only supports morph animation (and not skeletal animation) and does not import file formats that incorporate animation, it is not ideally suited for full-body animation.

The following is a list of MakeHuman blendshapes that are included (and therefore can be animated/triggered) in the default Head X model:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Modifier</th>
<th>Phoneme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amused</td>
<td>Blink</td>
<td>big aah</td>
</tr>
<tr>
<td>Frown</td>
<td>LEyeL (Left Eye Left...)</td>
<td>B,M,P</td>
</tr>
<tr>
<td>Grumpy</td>
<td>LEyeR</td>
<td>ch,J,sh</td>
</tr>
<tr>
<td>Disgust</td>
<td>LEyeU</td>
<td>D,S,T</td>
</tr>
<tr>
<td>Hopeful</td>
<td>LEyeD</td>
<td>eh</td>
</tr>
<tr>
<td>Puzzled</td>
<td>REyeL</td>
<td>F,V</td>
</tr>
<tr>
<td>Tender</td>
<td>REyeR</td>
<td>i</td>
</tr>
<tr>
<td>Anger</td>
<td>REyeU</td>
<td>K</td>
</tr>
<tr>
<td>Fear</td>
<td>REyeD</td>
<td>N</td>
</tr>
<tr>
<td>Sad</td>
<td>Head Up</td>
<td>oh</td>
</tr>
<tr>
<td>Seduce</td>
<td>Head Down</td>
<td>ooh,Q</td>
</tr>
<tr>
<td>SmileClosed</td>
<td>Head Left</td>
<td>R</td>
</tr>
<tr>
<td>SmileOpen</td>
<td>Head Right</td>
<td>th</td>
</tr>
<tr>
<td>Surprised</td>
<td></td>
<td>aah</td>
</tr>
</tbody>
</table>

A closed-source variant of Head X that incorporates FaceGen technology is available on request, but requires the user to be covered by a **FaceGen SDK** license (which is true for all contributors to the 'Thinking Head' project). FaceGen enables a huge variety of faces to be generated as parametric face coordinates and then imported into Head X. It can also automatically model the face of a real person from photos.

**SUGGESTED MODEL IMPORT PATHS**
**BACKGROUNDs**

The Head X background is a single colour by default, but some of the existing Head X configurations also load the *Head X Logo* as a background. Alternatively, you can either change the colour of the background directly or add any of your own image files to be displayed behind the Head. The image file must be of a type that is intrinsically supported by Windows (PNG, JPG, and BMP have been tested). As the image is displayed directly as a texture, it must also be of a size that is supported by your graphics driver; otherwise the background may not show or the application may even crash. Factors of 2 are suggested if you want to be safe about it.

XML

```xml
<render>
  <background file="../Data/hxlogo.png"
    red="1.0",
    green="0.3",
    blue="0.0"/>
</render>
```

GUI

*File→Background→Load* and

*File→Background→Remove*

**Command**

```
{loadbg file}
where file is full path to an image file
e.g. {loadbg ..\Data\hxlogo.png}
```

```
{clearbg}
removes the background
```
**CAMERA & LIGHTING**

Camera and lighting can be controlled directly through the GUI, as was described on pages 6 and 7. Alternatively, you can set these parameters in the HXML configuration file (you need to at least add one light, as no lighting is the default).

The configuration also permits you to apply different transformations to different objects in a hierarchical fashion, thus e.g. placing a particular object or group of objects separately from each other.

---

XML

Camera and Lighting parameters can also be modified directly through *Synapse* – see the Head-X(#) Camera *Synapse* for more on this.
**Text to Speech**

Head X makes use of external TTS engines to generate the speech output. Two distinct TTS interface are supported: MARY 4.0 and SAPI 5. (Earlier versions of these interfaces are not supported.)

**MARY 4.0**

- Open source, build your own voices
  - see [http://mary.dfki.de](http://mary.dfki.de)
- Need to run MARY server while using voices
- No tracking of current word position (i.e., EMOTE and other commands will be executed either before or after speech)
- Speech animation is not as well tuned as with SAPI 5

**Microsoft SAPI 5**

- Industry standard interface
- Many commercial voices (which you must buy and install)
- Voices preinstalled on Windows
  - Microsoft Anna (Windows Vista/7)
  - Microsoft Sam (Windows XP; poor quality)
- Experimental support for Chinese phoneme-driven speech animation – please contact the author if you are interested in this feature

You can choose either of these speech engines from the menu or within your configuration file.

**Process → Text to Speech → Engine**

Head X can then list all the voices available on the system for the selected engine.

**Process → Text to Speech → Change Voice**

You can modify the voice to a limited extent, and the mapping from voice to mouth animation to a greater extent, through the XML configuration of Head X. The parameters for this are presented on the next page.
**TTS PARAMETERS**

Set to **Chinese** for Chinese phoneme-driven speech

```
<TTS
  engine="SAPI"
  voice="ScanSoft Karen_Full_22kHz"
  language="English"
>
  rate="-3"
  khz="22"
  avsync="0.2"
  vmodulator="0.65"
  overlap="0.5"
  rampfunc="sine"
  mixframes="3"
  mixfactor="0.5"/
</TTS>
```

Audio playback frequency (should match voice; default is 22, but Windows built-in voice is better at 16)

Viseme frames to blend on each side of a viseme (default is 2)

Contribution of neighbouring viseme frames to current viseme frame (power decay) (default is 0.5)

Multiple definitions allowed

```
<viseme name="V">
  <bs name="Phoneme: aah"
      amplitude="0.5"
      offset="0.0"
      duration="1.0"
      rampfunc="sine"/>
</viseme>
```

Name of the viseme (SAPI) or phoneme (Chinese SAPI or MARY) to respond to

The name of the blendshape to animate in response to the viseme

Multiple definitions allowed

Playback rate (-10 to 10; default is 0)

Audio-video synchronization adjustment (0.1-0.2 is typically required; default is 0)

Maximum viseme amplitude (0-1; default is 0.65)

Viseme overlap (default is 0.75)

Viseme render function: **None**, **Linear**, **Half-Sine**, **Sine**, **Sinc** (default is **Sine**)

Audio playback frequency (should match voice; default is 22, but Windows built-in voice is better at 16)

Set to **Chinese** for Chinese phoneme-driven speech

Blending function (as used with expressions; default is "sine")
**Render Options**

Through the use of transparent textures or the setting of the *alpha_threshold* parameter for 3D models, various 3D surfaces may need to be rendered transparently. By default, this is achieved through an additional alpha blending pass for transparent fragments, with depth testing disabled. Visual artifacts may therefore be observed wherever multiple transparent surfaces overlap each other.

If the `-d` command-line option to Head X is specified, a high-quality, multi-pass transparency rendering technique – **Dual Depth Peeling** – becomes available, if it is supported by the graphics accelerator. You can limit the maximum rendering depth for transparent objects, as complex 3D models may otherwise take a long time to render. If the maximum layers are set to zero, Dual Depth Peeling is disabled and the standard approach is used instead.

*Note: The current implementation makes use of various nVidia OpenGL extensions which may or may not be supported by products of other manufacturers. Furthermore, while Dual Depth Peeling is enabled, no anti-aliasing will be performed. The visual quality of opaque objects is therefore superior if Dual Depth Peeling is disabled.*

Additionally, you can specify the **maximum frame rate** with which the Head is updated, thereby reducing the maximum computational load of the application but also adversely affecting the smoothness of speech animation. Note that this affects not just rendering, but also the rate at which Synapses are accessed and commands are processed.

For compatibility purposes, a slow “immediate mode” renderer and the facility to disable shader execution are also available.

```
<render fps="50" max_depth="12"
       immediate_mode="no" use_shaders="yes"/>
```

Maximum frame rate (in frames per second; default is 50)  Maximum transparent layers to render (default is 12; 0 = disabled, 1 is invalid)

Enable immediate mode renderer (default is no)  Run pixel shaders (default is yes; otherwise the fixed-function pipeline is used)
SUBTITLES

Head X supports text rendering for either a...

- …subtitle to the audiovisual speech
- …command prompt on the 3D display

**Process→Text on Screen→Show Subtitles**

and

**Process→Text on Screen→Command Prompt**

```xml
<render>
  <subtitle input_show="yes"
             output_show="no"
             input_prefix="> "
             output_prefix=""
             font="../Data/Fonts/Soutane Bold.ttf"
             font_size="20"
             red="255" blue="255" green="255"
             x="0" y="-160" z="100"
             length="40"
             blink_delay="500"/>
</render>
```

You can override any depicted subtitle with a custom subtitle of your choice through the SUB command. SUB replaces the subtitle for the currently spoken segment with its argument. Subtitles need to already be enabled for SUB to function.

```
{sub text}

where text is the custom subtitle

e.g. {sub Look, I have subtitles.}
```

Command

```
{sub Look, I have subtitles.}
```
**INTERACTION LOG**

Head X can log all text and command inputs and outputs into a file.

```xml
<interface logfile="hxlog.txt"/>
```

Each line in this file represents one of four events – an input by the user (I), a command received by Head X (C), a speech output (O), or an error (E). The type of event is marked by the header at the beginning of each line, which also includes the time of the event in seconds after application start, i.e. `@event/time@ text`

```
@E/574.749@ File "??" cannot be opened for reading.
@I/617.004@ hey
@C/619.890@ {emote grin} Hello. {break} I am a product of the Flinders University Artificial Intelligence Laboratory. {break} {emote grin 0.5}
@O/619.907@ Hello.
@O/621.082@ I am a product of the Flinders University Artificial Intelligence Laboratory.
```

**SHUTDOWN**

The Head X application can be exited by various means.

- **File→Exit** (GUI)
- Receive **TH_EVENT_KILL** (Events)
  - `{kill}` shuts down Head X immediately (Command)
**SYNAPSE**

*Synapse* is a system for sharing and synchronizing data between multiple processes. It does so through memory-mapped files, which is a very efficient approach, particularly for large chunks of data. Synapse is ideal for developing plug-ins for Head X, as it gives you access to the capabilities of Head X without requiring you to compile Head X. You can also use Synapse to communicate between components of your own creation.

Synapse was developed in and for C++ and is therefore most effective in this context, but it also directly supports Java and C#. Coding with Synapse is very easy; Doxygen code documentation for C++ and demo projects for all languages are included with the Synapse project, the code of which is freely available at [http://synapse-x.googlecode.com/svn/trunk/](http://synapse-x.googlecode.com/svn/trunk/).

**SYNKit**

Most developers will not be interested in modifying Synapse itself, but simply want to integrate it into their own program with as little fuss as possible. For this purpose, we provide the **SynKit** package, available as **SynKitR9.zip** from the same places from which you grabbed Head X (see page 1).

SynKit includes three solutions that allow a quick start in each of C++, Java, and C#.

- **SynCPP** is Synapse for C++. At the heart of Synapse is a single header file called synapse.h, which provides the complete Synapse functionality. synapse_app.cpp is a small program that demonstrates its use. Since SynCPP does not include any binaries, it is assumed that your C++ compiler can compile Synapse. Synapse is written in standard-compliant C, but it includes Windows headers, so using Visual C++ is the safest bet. A Visual Studio project file is included.

- **SynCS** is Synapse for C#. It comes with a set of include files and two DLLs, one for 32-bit programs (Synapse4CSharp.dll) and one for 64-bit programs (Synapse4CSharp64.dll). The DLL must be in the same directory as the executable, which for the included Visual C# project (with the Program.cs demo program) would be the bin\Release subdirectory. Furthermore, Synapse4CSharp64.dll must be renamed to Synapse4CSharp.dll if you are running a 64-bit program.

- **SynJava** is Synapse for Java, organized into the package au.edu.thinkinghead.synapse. It comes in 32-bit and 64-bit flavours, but unlike with SynCS you can select the correct DLL to load (Synapse4Java.dll or Synapse4Java64.dll) at the beginning of your own code. You would do so based on the VM that you are using to run your program. See the included demo program (S4JApp.java), and compile.cmd, which demonstrates how to compile it.

**LANGUAGE DIFFERENCES**

Programming with Synapse is very similar across all three languages. The main difference is that the C++ implementation is based on templates, whereas no generics support is provided for the Java and C# implementations. The C++ implementation therefore allows sharing of any
object (as long as it doesn't contain pointers to heap memory), while Java and C# only support primitives and arrays of primitives.

A code sample in C++:

```cpp
Synapse<float> mySyn; // ... creates a float-typed Synapse
mySyn.Create("DemoSynapse", 20);
    // ... names it DemoSynapse and allocates space for 20 floats
mySyn.Write(6.4f, 7);
    // ... writes 6.4 into the 8th position of the shared array
```

In Java and C#, the above would look marginally different:

```java
FloatSynapse mySyn = new FloatSynapse();
mySyn.Create("DemoSynapse", 20);
mySyn.Write(6.4f, 7);
```

The following Synapses exist in C# and Java (=== equivalent Synapses in C++):

- `BoolSynapse` === `Synapse<bool>` // currently not available in C#
- `CharSynapse` === `Synapse<signed char>`
- `UCharSynapse` === `Synapse<unsigned char>`
- `StringSynapse` === `Synapse<char>` // provides additional methods for native String handling
- `ShortSynapse` === `Synapse<short>`
- `UShortSynapse` === `Synapse<unsigned short>`
- `Int32Synapse` === `Synapse<long>`
- `UInt32Synapse` === `Synapse<unsigned long>`
- `Int64Synapse` === `Synapse<long long>`
- `UInt64Synapse` === `Synapse<unsigned long long>`
- `FloatSynapse` === `Synapse<float>`
- `DoubleSynapse` === `Synapse<double>`

Native array handling is provided in all three languages. `WriteArray` and `ReadArray` methods easily transfer data into and out of the shared space. The efficiency of the exchange is best between two C++ programs and worst between two non-C++ programs, as data needs to be copied from C space to the C# or Java space. You can generally improve efficiency to some degree by directly accessing the shared space, but you will need to lock the mutex manually in this case (using `Lock()` and `Unlock()`). Synapse offers multiple levels of functionality – for additional details on this, please refer to the Doxygen HTML documentation that is provided with SynCPP.

**ACCESSING A SYNAPSE**

The first step after instantiating a Synapse object is to either create a new memory-mapped space or open an existing one. There are four methods for this (shown in a language-neutral notation):

- **boolean Create(name, size, default-value)**
  creates a new space. It will fail and return FALSE if a space with the same name already exists.
- **boolean CreateFromArray(name, size, array)**
  does the same thing, except it initializes the space with the contents of the passed array.
• **boolean OpenNow(name)**
  opens an existing space. If this space does not exist, it returns FALSE.

• **void Open(name)**
  waits for an existing space and does not return until that space exists.

You can check whether a named space exists, without opening it, using the

• **boolean Exists(name)**
  static method.

Many of the methods listed here have default arguments, so you can actually call them with fewer parameters (e.g. size will default to 1, default-value is 0).

**Basic Reading and Writing**

Once a Synapse has been opened, you can write to and read from it. Only one write operation can occur concurrently, but read operations are allowed in parallel. The basic method set for this is:

• **boolean Write(object, index)**
  writes an object into the index position of the shared space. If the space is owned by another thread, it will wait until this is resolved. If other Synapse instances are reading from the same space, it will wait until they are finished doing so.

• **boolean Read(object, index)**
  reads the value at the index location in shared space into the object argument. If the space is owned by another thread, it will wait until this is resolved. This method (and ReadNow/ReadWhenNew) is only available in C++.

• **object ReadCopy(index)**
  fulfills the same purpose as the above, but returns the read object. This method, and all *Copy methods, are available in all supported languages.

The above methods will return FALSE (or type(0), for ReadCopy) if either the Synapse is not open or any waiting (see below) was cancelled.

For less patient programmers and programs, we furthermore provide

• **boolean WriteNow(object, index)**
• **boolean ReadNow(object, index)**
• **object ReadCopyNow(index)**

which return false (or (type)0, for ReadCopyNow) if the method cannot complete the write or read operation at this moment (no waiting!). Only one read operation is allowed at the same time if you use these methods.

**Using Time Stamps**

Each Synapse keeps track of its last successful read and write attempt, and compares this against the read and write times for the shared space. Consequently, you can check whether data has been modified or read since the last time you read or modified the data.

• **boolean WriteWhenOld(object, index)**
  will only write (and waits otherwise) when the previously written data has been read by a Synapse.
• boolean ReadWhenNew(object, index)
• object ReadCopyWhenNew(index)

will only read (and waits otherwise) when the data in the shared space is newer than the last read attempt.

You can check for the timer conditions manually through the
• boolean IsNew()
• boolean IsOld()

methods. IsNew() will check whether the last local read attempt is older than the last shared write time. Any successful write will also update the local read time, so IsNew() will only succeed if another Synapse wrote the new data. IsOld() will check whether the last local write attempt is older than the last shared read time. Please note that both IsNew() and IsOld() can be true at the same time!

WAITING
There are two cases where a method call will not return immediately - either when a thread is waiting on another thread, or when it's just not the right time to do something. In the latter case, the method will be notified by other Synapses once it is worth continuing. You can manually wait for such events through a set of waiting methods:

• boolean WaitForRead()
• boolean WaitForWrite()

wake up when another Synapse is reading from or writing to the shared space.

• boolean WaitForNew()
• boolean WaitForOld()

wake up when the IsNew() or IsOld() conditions are satisfied.

• boolean WaitForMatch(object, index, comparator)

wakes up when a shared object matches the object given as an argument to this method. The meaning of “match” is defined by the comparator function. This method is currently only supported in C++ (see SynCPP documentation).

All the above methods return a boolean. This boolean will be TRUE if the waiting ended for the expected reason. The boolean will be FALSE if the waiting was cancelled. You can cancel waiting by setting the stop variable to TRUE on the Synapse (only affects the local instance):

• void StopWaiting(boolean)

No waiting will be permitted while the stop variable is TRUE. Set the stop variable to FALSE when you wish to resume normal operations.

ARRAY READING AND WRITING
For more convenient array handling, all the above Write/Read methods also come in a form that accepts array arguments:

• boolean WriteArray(array, size)
• boolean ReadArray(array, size)
• boolean WriteArrayNow(array, size)
• boolean ReadArrayNow(array, size)
• boolean WriteArrayWhenOld(array, size)
• boolean ReadArrayWhenNew(array, size)

You need to specify the size of the array you’re passing in order to avoid potential buffer overflows.

Under Java and C# only, there exist further support functions for handling strings:

• boolean WriteString(string)
• string ReadString()
• boolean WriteStringNow(string)
• string ReadStringNow()
• boolean WriteStringWhenOld(string)
• string ReadStringWhenNew()

You can simply use normal Java and C# strings with these methods.

**Protected Writing**

It is possible, and in some cases likely, that what you write into a shared space will be overwritten by another Synapse before it is read. To ensure that your data reaches its intended recipient, you can use the Protect* write methods, which block and prevent any other write attempts until the shared space has been read. However, to safely use these methods, you must ensure that there is a proper “reader” (not a reader/writer) of the shared space, otherwise you will face deadlock.

• boolean ProtectWrite(object, index)
• boolean ProtectWriteArray(array, size)
• boolean ProtectWriteString(string)

**Direct Access**

As an alternative to all the above, you can also manually/directly access the shared space, by first obtaining ownership of the space (Lock(), or WriteLock() if you want be sure nobody is reading right now), accessing the data (Grab(), GrabCopy(), GrabPointer()), updating the time stamps (SetAsRead(), SetAsWritten()), and releasing ownership (Unlock()). This is for careful programmers who want maximum flexibility and minimum performance overhead. Check out the SynCPP documentation for more.
**Head X Synapses**

Each instance of the Head X program that is running will publish a set of Synapses of different names (also referred to as signatures). The following is a list of Synapses published by Head X.

**Head-X**

One Synapse is published by all Head X instances: **Head-X** stores the number of Head X instances currently running on this system as a single 32-bit integer. If **Head-X** doesn't exist, no Head X is running!

![Recommended Use Diagram]

**Head-X(#)**

Each running Head X publishes an identity through which it can be addressed.

# is the number of the Head...

- Starting from 1 (one)
- E.g. **Head-X(1)**
- Each Head has a different number

The Synapse stores the Head X interface version number (62 with the current release). Earlier Head X releases may not support Synapse to the same extent as the current and later releases.

Head identity is part of the signature of all the next Synapses.
- E.g. **Head-X(#) - Command**

![Command Diagram]

**Command**

Any text written to the **Head-X(#) - Command** Synapse will be processed as if it were typed into the Command Window or Prompt, i.e. text is converted into speech and commands are executed. A maximum input length of 4096 characters is currently enforced; anything above this is truncated. If you need to send more to the Head, you must do so through multiple operations.
The Command input is also accessible through the Event Interface.

**DIRECTCOMMAND**

The **Head-X(#)-DirectCommand** Synapse processes individual commands...

- e.g. **EMOTE happy 2.3** *(note: no curly delimiters)*

Processing occurs immediately, which can be used e.g. for emoting or changing backgrounds while speaking. Certain loading commands may cause interruptions to speech output if they are processed during speech. The BREAK and PAUSE commands are not supported.

**SPOKENTEXT**

The **Head-X(#)-SpokenText** Synapse stores the last segment that the Head spoke. It updates as soon as the Head starts speaking the segment. A segment is everything until the last \{BREAK\} (or stop, if *Auto-Break on Punctuation* is set).

**USERTEXT**

The **Head-X(#)-UserText** Synapse stores the last command entered via the GUI. It is observed by the Alice XS dialogue manager, which publishes its responses to the **Command** synapse.
STATUS

The **Head-X(#)-Status** Synapse records Head X state information. It is an array of numbers, each of which represents different state information about the Head. See the table below – the first row specifies the array index, the second row specifies the type of information stored at this index, and the final row describes the different values that are permitted.

<table>
<thead>
<tr>
<th>Element 0</th>
<th>Element 1</th>
<th>Element 2</th>
<th>Element 3</th>
<th>Element 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ready?</strong></td>
<td><strong>Talking?</strong></td>
<td><strong>Word Index</strong></td>
<td><strong>Last Error</strong></td>
<td><strong>Window State</strong></td>
</tr>
<tr>
<td>-1 = Dead</td>
<td>0 = Idle</td>
<td>Index of the last spoken word in the segment</td>
<td>0 = No error</td>
<td>0 = Normal</td>
</tr>
<tr>
<td>0 = Busy (executing command)</td>
<td>1 = Speaking, or Preparing to speak, or Emoting</td>
<td>1 = Error while loading</td>
<td>1 = Error while saving</td>
<td>1 = No Menu</td>
</tr>
<tr>
<td>1 = Ready</td>
<td>2 = Speaking</td>
<td>2 = Error while saving</td>
<td>2 = Maximized</td>
<td>4 = No border</td>
</tr>
<tr>
<td>Sum these for multiple states (e.g. 7 for full-screen)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**KBM**

The **Head-X(#)−KBM** Synapse records the current mouse and keyboard input to the Head X GUI.

<table>
<thead>
<tr>
<th>Element 0</th>
<th>Element 1</th>
<th>Element 2</th>
<th>Element 3</th>
<th>Element 4</th>
<th>Element 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key down?</td>
<td>Last key</td>
<td>Mouse button down?</td>
<td>Last mouse click?</td>
<td>X mouse-down</td>
<td>Y mouse-down</td>
</tr>
<tr>
<td>&gt;0 if key is down</td>
<td>Character code of last pressed key</td>
<td>1 = Left button</td>
<td>Same as element 2, but no multiple buttons</td>
<td>XY coordinates of the mouse pointer when last button was pressed</td>
<td></td>
</tr>
<tr>
<td>0 otherwise</td>
<td>2 = Right button</td>
<td>4 = Middle button</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sum these for multiple buttons</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**CAMERA**

The **Head-X(#)-Camera** Synapse records the state of the Head rotation, translation, scale and aspect ratio, as well as the lighting parameters for both supported lights. The values for each element are the same as you would use in an HXML configuration file (i.e., see above). You can write to this Synapse; any changes to values that you make will also change the Head accordingly. (Specifically, the Head compares the newly written values against the last updated values and applies the difference.)

<table>
<thead>
<tr>
<th>Element 0</th>
<th>Element 1</th>
<th>Element 2</th>
<th>Element 3</th>
<th>Element 4</th>
<th>Element 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 6</td>
<td>Element 7</td>
<td>Element 10</td>
<td>Element 11</td>
<td>Element 12</td>
<td>Element 13</td>
</tr>
<tr>
<td>Scale</td>
<td>Squash</td>
<td>Light 1 azimuth</td>
<td>Light 1 elevation</td>
<td>Light 1 ambient</td>
<td>Light 1 diffuse</td>
</tr>
<tr>
<td>Element 20</td>
<td>Element 21</td>
<td>Element 22</td>
<td>Element 23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Light 2 azimuth</td>
<td>Light 2 elevation</td>
<td>Light 2 ambient</td>
<td>Light 2 diffuse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**HEADMORPHS**

The **Head-X(#)-HeadMorphs** Synapse records the state of the Head animation morphs. The Synapse has 39 elements representing the 39 standard FaceGen morph targets (in the same order as in the Expressions File or FaceGen Modeller). You can change these as you would change e.g. rotations with the Camera Synapse.
**WORKINGPATH**

The **Head-X(#)-WorkingPath** Synapse shows the current working path of the Head X application. This is useful to know if you are providing relative paths to commands.

```
Synapse [char, 4096]
Head-X(#)-WorkingPath
```

**SYNAPSE IN HEAD X**

The content of each Synapse can be verified through the Synapse Interface dialog.

```
Interface→Synapse→Show Capabilities
```

Command, DirectCommand, Camera, and HeadMorphs Synapses are not updated automatically unless they are enabled (they are disabled by default). All other Synapses are updated automatically by Head X.

```
<action>
  <synapse commands="yes" direct="yes"
    camera="yes" headmorphs="yes"
    block="yes"/>
</action>
```

If **yes**, the Command Synapse will block, i.e. is not released by Head X until completion of the command/speech – this may simplify programming if you do not need to be able to interrupt the Head’s actions (default is **no**).