ASLe16 – Compressed Binary Encoding for American Sign Language

Jesse Jurman
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Email: jrj2703@rit.edu

Ethan Jurman
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Email: ehj2229@rit.edu

Abstract

One of the important aspects of language, or of any information, is the fundamental ability to encode and visually represent it on a digital device. All languages have had unique difficulties in their transition from written symbols to universal digital encodings, but none as vast and incomplete as American Sign Language. ASL has inherent difficulties in the unique gestures that capture information, including eye movement, hand poses, arm rotations, and even facial expressions. While these traits may be inherently difficult to encode in binary at first, our goal was to build an encoding that encompasses gestures and poses in a fundamentally small digital space. Our encoding can represent ASL that can be easily moved from computer to computer, and can generate visual representations on the receiving computer that convey the original message.

1 Introduction

ASL remains to be the one of the only universally understood forms of communication that has not found any unified binary encoding. The difficulty with ASL is that many of the gestures and poses require a full 3D model of a human to represent. While in recent years the advent of recording dictations has become more commonplace, passing videos from computer to computer is an impractical process due to the sheer size of the files, and the fixed format that they fall in.

Previous research has placed large emphasis on generating full 3D models, either by synthesizing large amounts of data that capture every gesture, or by literally translating motions one-to-one on a 3D character (Lu and Huenerfauth, 2012). These models are constricted in a variety of ways, namely in that a specific 3D character is coupled to that encoding, and that impractically large amounts of data needs to be used to render each individual pose.

The goal of ASLe16 is to compress ASL into a format which can easily be shared and streamed in a digital format. ASLe16 leverages the processing power that has become commonplace in most computers to interpolate between fixed positions and poses, allowing for synthesized renders that can be used as fundamental as black-and-white stick figures, to fully emotive 3D characters.

2 Encoding American Sign Language

Since ASLe16 is extensible to work with 128 unique parts, and 256 modifiers for each part, there are few aspects of ASL the format could fail to cover. In its current version (0.1) ASLe16 does not cover animation because of the complexity motion brings into ASL; However the format does not discourage it. With additional time and elements in our encoding, interpolation of ASLe16's current "snapshot" status of ASL could produce motion and variance needed to produce proper signs. With use of Python, ASLe16 is currently generated in a compressed format. This allows for easy modification and addition of new components when changes to the system need to be introduced as well as abstracting out complications that come with generating an expansive, complex, and reproducible format.

2.1 Positional Library of ASL

ASLe16 currently produces a multitude of handshapes, hand location, wrist rotation, facial expression, and some miscellaneous modifiers. Within each ASLe16 block – a set of 16 bits to describe a unique feature of ASL – we can give information to the rendering application.

2.2 Encoding ASL to Binary Format

The ASLe16 format is a compressed way to describe parts on an armature, and set of modifiers for each part which may or may not be unique to that part. The following is an example of three ASLe16 blocks, with an accent on the part of that we are describing to the right:

10000100 00010111	Continuation bit
1**0000111** 00110000	Part Id (7-bits)
00000010 **00000101**	Modifiers (8-bits)

Table 1 : ASLe16 Placement Legend

The first bit (the continuation bit) tells the renderer if there is going to be a following block for this frame. It is either 0 (an **ending bit**) or 1 (a **continue bit**). The next 7 bits tell the renderer what part will be modified, which allows for 128 unique parts that can be selected for modification. Any combination of 1s or 0s are valid, given that they map to a part described in the corpus. The last part of the block are 8 unique bits for that part, which can be used to describe the location for a hand, the handshape being used, the direction the character's eyes are looking, etc.

```
10000101 00001110
10000110 00010001
10000011 00000001
10001000 00100000
10000010 00000000
00000001 00000101
```

Encoding 1 : ASLe16 Encoding – the sign Help

2.3 Verification in 3D Modeling & Animation Environment

In order to verify that ASLe16 accurately described the poses available in ASL, an armature in Blender – a 3D modeling and animation program – was built and rigged. Many bones and armature rigs in the model were abstracted out or compressed to single variables as to contain a core set of ASL functions that would transform easily from the encoding. The liberties taken by the Blender, and Python – a high level scripting language – show how powerful computer interpolation can be on the most fundamental scale. Further interpolation could make use of frame-by-frame transitions between poses, and more emotive gestures. Use of interpolated values and gestures could be extremely useful in sign recognition, especially when incorporated with Hidden Markov Model which has been previously used in detect ASL words and sentences with high margins of success (Vogler and Metaxas, 2004).

2.4 Altering the 3D Environment with Python

The blender scripts themselves make use of bpy – the blender python library – which allowed us to select specific bones and move them into pre-defined positions. This means that for every modifier value, there is a unique position that has been pre-defined to work with the armature.

2.5 Interactive Builder for ASL Encoding

Screenshot 1 : Tk Application written for working with ASLe16

Building an interactive ASLe16 builder helped validate the encoding against our 3D model, as well as allow other users see how flexible the encoding was at animating a character. The interactive builder allows users to select a part described in the ASL corpus, and then modify several option-menus for redefining values specific for each part (seen on the left in Screenshot 1). These values generate a unique binary (listed at the top-left as seen in Screenshot 1) and a render (shown to the right in Screenshot 1) with our armature to visually show the user what the encoding looks like on a rendered format.

The code for the application and all resources are available here:
https://github.com/JRJurman/ASLe16

2.6 Sequence of Python Processes

In order to build and test the encoding, several scripts and processes needed to work in an interlocking way. While it is not the purpose of this paper to discuss specific algorithms or coding practices, the next sections will detail the core interactions we decided to implement.

2.6.1 Language Dictionary

In order to translate binary to meaningful data, we stored a collection of Parts inside a hash-table, each mapped to its appropriate ASLe16 string identifier. In our implementation, each Part also contained a collection of Modifier objects, which in and of themselves had a hash-table of ASLe16 string identifiers to human-readable strings.

2.6.2 Language Translation

In order to validate and translate these ASLe16 encodings, we built a class which could take an ASLe16 string, and call a special function. Our implementation allowed the user to register functions to be called, given a specific modifier-value or part name. The function would then update our model for the next rendering. Calling the function would also add a respective undo variation of the function to a stack, which was called after the render had been taken. This was required to put the figure back into a default position, without remembering any of the part-modifiers it had placed on it.

2.6.3 Posing the Armature

The function calls used a variety of techniques to re-pose the armature in unique ways. Some functions that triggered on a single value (such as the handshapes), did re-posing to the every finger in a unique way. Other functions, such as the wrist movement, passed the value of the modifier along to the figure, which understood the value of the modifier to indicate a unique position. This allowed us to avoid having to write a unique rigging position for every possible value.

2.6.4 The Hand-shake

A list of drop-down menus allow the user to build the ASLe16 without typing the binary by hand. The binary is sent to the translation script[1], which calls the series of functions that have been registered beforehand. The functions update the 3D figure to be in the correct pose, and generate a render. The undo-stack is called, putting the figure in the default pose, while the front-end searches for the final rendered image. Once loaded, the user can work again with the drop-downs to modify the render in new ways.

3 Verification

After making several renders based on beginnings ASL signs, we had several deaf students tell us what they thought the nearest translation for the render was.

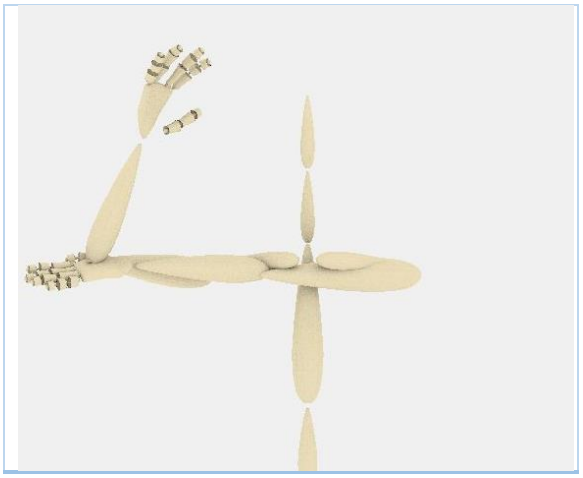
Render 1: ASL Sign for Help (ASLe16 v0.1)

This sign was immediately recognized as (the beginning of) the sign for "Help" in ASL by a number of deaf individuals. This is due to its unique position, and handshapes involved.

[1] Our implementation separated the creation and reading of ASLe16 strings by running separate python processes, one for tk_code_builder.py, and one for blender_script.py

Render 2: ASL Sign for Think (ASLe16 v0.1)

This sign was interpreted as (the beginning of) the sign for "Think" as well as "For". This ambiguity is due to the lack of motion implemented in the system, as motion distinguishes these signs from one another.



Render 3: ASL Sign for Tree (ASLe16 v0.1)

This sign was rendered to be interpreted as "Tree", however due to a lack of extensive positional controls we were not able to present the arm to be in the correct position - straight, upright, 90 degree's with the other arm. Because of this, none of the users were able to interpret the sign as desired (a common response for this sign however was "Day"). In addition, failure to rig the model correctly has allowed the left hand to be placed inside the right elbow.

Despite some ambiguities in renders, users responses to the system and the project itself was unanimously positive.

4 Conclusion

While our implementation of ASLe16 allows for a wide set of controls, there are many improvements that could be made. As mentioned above, many of the bits used to modify parts are left undefined, and for some parts, such as hand-location, 256 unique values is over-extensive (unless we planned on mapping those positions on a 3D-coordinate system). These could possibly be compressed with other parts to make an even more compressed encoding. Many aspects of ASL were ignored as well, including classifiers, and audience conversation – which might have the armature look at or point to different subjects. As mentioned before, ASLe16 is an open format which can be modified and extended by other users, be used with more dynamic renderers, and have more added to the corpus. A large component of ASL that was not implemented in this release was animation. Blender's renderer could output multiple frames to simulate animation, so future implementation of this feature is merely a matter of dedicating more time to both the renderer and the corpus.

5 Additional Authors & Acknowledgments

Sage Nagy – Domain Expert
Email: smn7543@rit.edu

6 References

Christian Vogler and Dimitris Metaxas. 2004. *Handshapes and movements: Multiple-channel ASL recognition*, Springer Verlag, Artificial Intelligence 2915, pp. 247-258.

Pengfei Lu and Matt Huenerfauth. 2012. Learning a Vector-Based Model of American Sign Language Inflecting Verbs from Motion-Capture Data

Appendix

Abstract

Lookup Table

python asl_endcoding.py

```
0000000 ________ > Face
0000000 000_____ >   EyePosX : center
0000000 001_____ >   EyePosX : focus
0000000 010_____ >   EyePosX : left
0000000 011_____ >   EyePosX : right
0000000 100_____ >   EyePosX : farLeft
0000000 101_____ >   EyePosX : farRight
0000000 ___00___ >   EyePosY : center
0000000 ___01___ >   EyePosY : focus
0000000 ___10___ >   EyePosY : down
0000000 ___11___ >   EyePosY : up
0000000 _____0__ >   Cheek : normal
0000000 _____1__ >   Cheek : out
0000000 ______00 >   EyeBrow : normal
0000000 ______01 >   EyeBrow : raised
0000000 ______10 >   EyeBrow : together

0000011 ________ > LeftHandLocation
0000011 00000000 >   Location : neutral-space
0000011 00000001 >   Location : center-space
0000011 00000010 >   Location : right-space
0000011 00000011 >   Location : left-space
0000011 00000100 >   Location : target
0000011 00000101 >   Location : hand
0000011 00000110 >   Location : forehead/brow
0000011 00000111 >   Location : mouth/chin
0000011 00001000 >   Location : eyes/nose
0000011 00001001 >   Location : left-temple
0000011 00001010 >   Location : right-temple
0000011 00001011 >   Location : left-cheek/ear
0000011 00001100 >   Location : right-cheek/ear
0000011 00001101 >   Location : face/head
0000011 00001110 >   Location : shoulder-left
0000011 00001111 >   Location : manubrium
0000011 00010000 >   Location : shoulder-right
0000011 00010001 >   Location : torso-left
0000011 00010010 >   Location : torso-center
0000011 00010011 >   Location : torso-right
0000011 00010100 >   Location : waist-left
0000011 00010101 >   Location : waist-center
0000011 00010110 >   Location : waist-right
0000011 00010111 >   Location : upper-arm
0000011 00011000 >   Location : elbow
0000011 00011001 >   Location : forearm
0000011 00011010 >   Location : back-of-wrist
0000011 00011011 >   Location : inside-of-wrist
0000011 00011100 >   Location : up
0000011 00011101 >   Location : down
0000011 00011110 >   Location : forward
0000011 00011111 >   Location : back
0000011 00100000 >   Location : left
0000011 00100001 >   Location : right

0000101 ________ > LeftHandShape
0000101 00000000 >   Handshape : relaxed
0000101 00000001 >   Handshape : 1
0000101 00000010 >   Handshape : 3
0000101 00000011 >   Handshape : bent3
0000101 00000100 >   Handshape : 4
0000101 00000101 >   Handshape : 5
0000101 00000110 >   Handshape : claw5
0000101 00000111 >   Handshape : 6
0000101 00001000 >   Handshape : 7
0000101 00001001 >   Handshape : 8
0000101 00001010 >   Handshape : open8
0000101 00001011 >   Handshape : 9
0000101 00001100 >   Handshape : flat9
0000101 00001101 >   Handshape : a
0000101 00001110 >   Handshape : openA
0000101 00001111 >   Handshape : b
0000101 00010000 >   Handshape : bentB
0000101 00010001 >   Handshape : flatB
0000101 00010010 >   Handshape : openB
0000101 00010011 >   Handshape : c
0000101 00010100 >   Handshape : flatC
0000101 00010101 >   Handshape : smallC
0000101 00010110 >   Handshape : e
0000101 00010111 >   Handshape : g
0000101 00011000 >   Handshape : h
0000101 00011001 >   Handshape : i
0000101 00011010 >   Handshape : k
0000101 00011011 >   Handshape : l
0000101 00011100 >   Handshape : m
0000101 00011101 >   Handshape : openM
0000101 00011110 >   Handshape : n
0000101 00011111 >   Handshape : openN
0000101 00100000 >   Handshape : o
0000101 00100001 >   Handshape : openO
0000101 00100010 >   Handshape : smallO
0000101 00100011 >   Handshape : r
0000101 00100100 >   Handshape : t
0000101 00100101 >   Handshape : v
0000101 00100110 >   Handshape : bentV
0000101 00100111 >   Handshape : x
0000101 00101000 >   Handshape : openX
0000101 00101001 >   Handshape : y
0000101 00101010 >   Handshape : ily
0000101 00101011 >   Handshape : corna

0000001 ________ > LeftTargetModifiers
0000001 00000000 >   Finger : none
0000001 00000001 >   Finger : thumb
0000001 00000010 >   Finger : index
0000001 00000011 >   Finger : middle
0000001 00000100 >   Finger : ring
0000001 00000101 >   Finger : pinky

0000111 ________ > LeftWrist
0000111 0000____ >   Roll : 0
0000111 0001____ >   Roll : 1
0000111 0010____ >   Roll : 2
0000111 0011____ >   Roll : 3
0000111 0100____ >   Roll : -3
0000111 0101____ >   Roll : -2
0000111 0110____ >   Roll : -1
0000111 ____00__ >   Pitch : none
0000111 ____01__ >   Pitch : toward
0000111 ____10__ >   Pitch : tilted
0000111 ____11__ >   Pitch : away
0000111 ______00 >   Yaw : none
0000111 ______01 >   Yaw : toward
0000111 ______10 >   Yaw : tilted
0000111 ______11 >   Yaw : away

0000100 ________ > RightHandLocation
0000100 00000000 >   Location : neutral-space
0000100 00000001 >   Location : center-space
0000100 00000010 >   Location : right-space
0000100 00000011 >   Location : left-space
0000100 00000100 >   Location : target
0000100 00000101 >   Location : hand
0000100 00000110 >   Location : forehead/brow
0000100 00000111 >   Location : mouth/chin
0000100 00001000 >   Location : eyes/nose
0000100 00001001 >   Location : left-temple
0000100 00001010 >   Location : right-temple
0000100 00001011 >   Location : left-cheek/ear
0000100 00001100 >   Location : right-cheek/ear
0000100 00001101 >   Location : face/head
0000100 00001110 >   Location : shoulder-left
0000100 00001111 >   Location : manubrium
0000100 00010000 >   Location : shoulder-right
0000100 00010001 >   Location : torso-left
0000100 00010010 >   Location : torso-center
0000100 00010011 >   Location : torso-right
0000100 00010100 >   Location : waist-left
0000100 00010101 >   Location : waist-center
0000100 00010110 >   Location : waist-right
0000100 00010111 >   Location : upper-arm
0000100 00011000 >   Location : elbow
0000100 00011001 >   Location : forearm
0000100 00011010 >   Location : back-of-wrist
0000100 00011011 >   Location : inside-of-wrist
0000100 00011100 >   Location : up
0000100 00011101 >   Location : down
0000100 00011110 >   Location : forward
0000100 00011111 >   Location : back
0000100 00100000 >   Location : left
0000100 00100001 >   Location : right

0000110 ________ > RightHandShape
0000110 00000000 >   Handshape : relaxed
0000110 00000001 >   Handshape : 1
0000110 00000010 >   Handshape : 3
0000110 00000011 >   Handshape : bent3
0000110 00000100 >   Handshape : 4
0000110 00000101 >   Handshape : 5
0000110 00000110 >   Handshape : claw5
0000110 00000111 >   Handshape : 6
0000110 00001000 >   Handshape : 7
0000110 00001001 >   Handshape : 8
0000110 00001010 >   Handshape : open8
0000110 00001011 >   Handshape : 9
0000110 00001100 >   Handshape : flat9
0000110 00001101 >   Handshape : a
0000110 00001110 >   Handshape : openA
0000110 00001111 >   Handshape : b
0000110 00010000 >   Handshape : bentB
0000110 00010001 >   Handshape : flatB
0000110 00010010 >   Handshape : openB
0000110 00010011 >   Handshape : c
0000110 00010100 >   Handshape : flatC
0000110 00010101 >   Handshape : smallC
0000110 00010110 >   Handshape : e
0000110 00010111 >   Handshape : g
0000110 00011000 >   Handshape : h
0000110 00011001 >   Handshape : i
0000110 00011010 >   Handshape : k
0000110 00011011 >   Handshape : l
0000110 00011100 >   Handshape : m
0000110 00011101 >   Handshape : openM
0000110 00011110 >   Handshape : n
0000110 00011111 >   Handshape : openN
0000110 00100000 >   Handshape : o
0000110 00100001 >   Handshape : openO
0000110 00100010 >   Handshape : smallO
0000110 00100011 >   Handshape : r
0000110 00100100 >   Handshape : t
0000110 00100101 >   Handshape : v
0000110 00100110 >   Handshape : bentV
0000110 00100111 >   Handshape : x
0000110 00101000 >   Handshape : openX
0000110 00101001 >   Handshape : y
0000110 00101010 >   Handshape : ily
0000110 00101011 >   Handshape : corna

0000010 ________ > RightTargetModifiers
0000010 00000000 >   Finger : none
0000010 00000001 >   Finger : thumb
0000010 00000010 >   Finger : index
0000010 00000011 >   Finger : middle
0000010 00000100 >   Finger : ring
0000010 00000101 >   Finger : pinky

0001000 ________ > RightWrist
0001000 0000____ >   Roll : 0
0001000 0001____ >   Roll : 1
0001000 0010____ >   Roll : 2
0001000 0011____ >   Roll : 3
0001000 0100____ >   Roll : -3
0001000 0101____ >   Roll : -2
0001000 0110____ >   Roll : -1
0001000 ____00__ >   Pitch : none
0001000 ____01__ >   Pitch : toward
0001000 ____10__ >   Pitch : tilted
0001000 ____11__ >   Pitch : away
0001000 ______00 >   Yaw : none
0001000 ______01 >   Yaw : toward
0001000 ______10 >   Yaw : tilted
0001000 ______11 >   Yaw : away
```

README.md from https://github.com/JRJurman/ASLe16

ASLe16

Created By Jesse Jurman and Ethan Jurman

ASLe16 is an encoding system that represents the American Sign Language using a 16 bit encoding.

What's Inside the Box!

asl_encoding.py

To generate a full listing of encodings, run (hint, you may want to append `| less` to the call to make it easier to read):

```
python asl_encoding.py
```

Python script; asl_encoding is a script that generates all the binary encodings. The script uses the PartBlock and ModifierBlock classes from Encoding.py, and is loaded into the construction of PyDecipher

ASL_Model.blend

Binary Blender file; This is the blender armature that has been rigged with various drivers to work with the blender_script.py. If you have Blender installed, you may open the file with that, and can change some render settings (although some are forced in the blender_script.py)

blender_script.py

This file can be run by itself, and takes binary strings from standard-in. The images rendered will be placed in the ./tmp/ located in this folder.

```
python blender_script.py
```

Python Script; This script works with the ASL_Model.blend file and the bpy library which allows python to interact with blender in a scriptable way. This file also makes use of the PyDecipher class, which allows registering functions to a given decipherment.

Every function in this file describes how the armature in blender should be modified to make the correct pose. Each function also appends to an `undoStack` which is a list of functions which get called after each render is made. The `undoStack` is required to put the model in the default pose, ready for the next one.

Encoding.py

Python Script; This file contains the PartBlock and ModifierBlock classes, which are created by the asl_encoding.py script, and are loaded by the PyDecipher script.

Any PartBlock contains the name for a part, it's binary encoding (in 7-bits) and a list of ModifierBlocks.

Any ModifierBlock contains the name of the modifier, the size the modifier takes in binary, and a list of values (in the order that they should be mapped to).

__init__.py

Python Script; Tells python that files located here may be used for other python scripts.

launcher.bat

Batch Script; Batch file that can be double-clicked in Windows to open the program (literally just opens tk_code_builder.py with python).

launcher.command

Bash Script; Bash script that can be double-clicked in Mac OS X to open the program (literally just opens tk_code_builder.py with python).

PyDecipher.py

Python Script; Python script that loads a hash of PartBlocks (it only reads in the values) and can make function calls or print strings based on binary-strings that map to those PartBlocks.

It contains a `decipher(self, string)` method, which takes in a string and returns a list of hashes that have Name -> PartBlock.name and Modifiers -> a list of tuples that are: (ModifierName, ModifierValue).
Also included is a list of register functions, including `setAfterRegister(self, func)`, `setBeforeRegister(self, func)`, `register(self, func, PartName=None, PartModifier=None, ModifierValue=None)`, and finally`callFunctions(self, string)`. `callFunctions(self, string)` calls a function set

in `setBeforeRegister(self, func)`, all the functions defined in `register(self, func, ...)` in order of least generic to most generic, and a function set in `setAfterRegister(self, func)`. `callFunctions(self, string)` takes in a binary-string, similar to the `decipher(self, string)` function, but instead of returning a hash of PartBlocks, it simply calls the registered functions.
At the very end of the file are several validation methods (all regexes), which check if the encoding is in a valid format: `isValid(self, string)`, `isMissingBits(self, string)`, `isMissingBytes(self,sting)` and`tooManyBytes(self, string)`.

README.md

MarkDown File; This file, a description of each file and folder required for running the application.

tk_code_builder.py

```
python tk_code_builder.py
```

Graphical User Interface for blend_script.py. This file loads a client window written in Tkinter to allow users to select PartBlocks from the asl_encoding.py and pipe binary strings to the blender_script.py (which runs in a`subprocess.Popen` call). Renders which are placed in the ./tmp/ folder are converted into `.gif` files, either with`sips` (on mac) or with `ImageMagick` (on Windows) and loaded in the right pane.