# Voc

## A vocal tract physical model implementation.

By Paul Batchelor

Git Hash: aba8c68f14e4928d85ce287ff398599c20882681

# Introduction

The following document describes *Voc*, an implementation of a vocal tract physical model.

## Motivations and Goals

The human voice is a powerful tool for any composer, second only to silence. Even an approximation of the voice can tap into the entire range of human emotion. This is why the wind howls, floorboards moan, or R2D2 pouts. For computer musicians and sound designers alike, creating sonic elements with vocal qualities can give cold digital sounds a human-like relatable quality; an excellent tool for engaging an audience.

The goal of *Voc* is to provide a low level model for producing utterances and phonemes. It will neither attempt to sing or talk, but it will babble and chatter. A program which is closely aligned with Voc's scope is Neil Thapen's web application *Pink Trombone*. [pinktrombone] In this program, vocal phonemes are generated through directly manipulating a virtual vocal tract in continuous time.

## Literate Programming

As an experiment, the author has decided to use *literate programming* for this project. Literate programming, created by Donald Knuth [knuth1992literate], is the concept of melting documentation and code together. What you are reading is also a program!

The biggest advantage of using literate programming for this project is the ability to use mathematical notation to describe concepts that are implemented. The C-language does not lend itself well for comprehensibility when it comes to DSP, even with comments. Nobody ever learned about DSP from C code alone! A very successful example of literate programming is the book *Physically Based Rendering* [pharr2016physically], which is both a textbook and software implementation of a physically accurate ray tracer.

The underlying technology used here is CWEB[knuth1994cweb], the definitive literate programming tool developed by Donald Knuth, with some minor macro adjustments for formatting.

# Overview (1)

In a literate program, it is customary (and somewhat mandatory) to provide an "overview" section. This section serves as the entry point in generating the C amalgamation file *voc.c*. Complying with the constraints of `CWEB`, the corresponding sections will appear at the bottom of this section.

## The Core Voc Components

⟨ Headers 3 ⟩ is the header section of the C file (not be confused with the separate header file ⟨ `voc.h` 63 ⟩. This is where all the system includes, macros, global data, and structs are declared.

⟨ The Glottis 31 ⟩ is the component of Voc concerned with producing the glottal excitation signal.

⟨ The Vocal Tract 43 ⟩ is implementation of the physical waveguide of the vocal tract.

⟨ Top Level Functions 11 ⟩ is the section consisting of all public functions for controlling Voc, from instantiation to parametric control.

## Supplementary Files

In addition to the main C amalgamation, there are a few other files that this literate program generates:

⟨ `debug.c` 65 ⟩ is the debug utility used extensively through out the development of Voc, used to debug and test out features.

⟨ `voc.h` 63 ⟩ is the user-facing header file that goes along with the C amalgamation. Anyone wishing to use this program will need this header file along with the C file.

⟨ `plot.c` 66 ⟩ is a program that generates dat files, which can then be fed into gnuplot for plotting. It is used to generate the plots you see in this document.

⟨ `ugen.c` 67 ⟩ provides an implementation of Voc as a Sporth unit generator, offering 5 dimensions of control. In addition the main Sporth plugin, there are also smaller unit generators implementing portions of Voc, such as the vocal tract filter.

`voc.lua` (not generated but included with the source code) contains metadata needed to implement Voc as a Soundpipe module.

⟨ `ex_voc.c` 76 ⟩ Is a small C program that uses Voc, written in the style of a classic Soundpipe example.

⟨ `t_voc.c` 77 ⟩ Is a small C program written for the Soundpipe testing utility.

⟨ `p_voc.c` 78 ⟩ Is a small C program written for the Soundpipe perfomance measurement utility.

⟨ Headers 3 ⟩
⟨ The Glottis 31 ⟩
⟨ The Vocal Tract 43 ⟩
⟨ Top Level Functions 11 ⟩

# Header Inclusion, Structs, and Macros (2)

**3.**

## Header of File

The header section consists of header inclusion, and definition of C-structs. The system-wide header files include *stdlib.h* for things like *malloc*( ). Standard math library functions from *math.h* are used. Soundpipe/Sporth specific header files are *soundpipe.h* and *sporth.h*. It should be noted that due to the implementation of Sporth, the Soundpipe header file *must* be included before the Sporth header file.

ANSI C doesn't have the constant `M_PI`, so it has to be explicitly defined.

Both `MIN` and `MAX` macros are defined.

The header file *string.h* is included so that *memset* can be used to zero arrays.

There is exactly one local header file called *voc.h*, which is generated by CTANGLE. For more information about this header file, see ⟨ `voc.h` 63 ⟩

The macro `MAX_TRANSIENTS` is the maximum number of transients at a given time.

```
⟨ Headers 3 ⟩ ≡
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "soundpipe.h"
#ifndef M_PI
#define M_PI  3.14159265358979323846
#endif
#include "voc.h"
#ifndef MIN
#define MIN(A, B)  ((A) < (B) ? (A) : (B))
#endif
#ifndef MAX
#define MAX(A, B)  ((A) > (B) ? (A) : (B))
#endif
#define EPSILON  1.0 · 10^-38
#define MAX_TRANSIENTS  4
  ⟨ Data Structures and C Structs 4 ⟩
```

This code is cited in section 1.

This code is used in section 1.

**4.**

## Structs

This subsection contains all the data structs needed by Voc.

```
⟨ Data Structures and C Structs 4 ⟩ ≡
  ⟨ Glottis Data Structure 6 ⟩
  ⟨ Transient Data 7 ⟩
  ⟨ Tract Data 10 ⟩
  ⟨ Voc Main Data Struct 5 ⟩
```

This code is used in section 3.

**5.** The top-most data structure is *sp_voc*, designed to be an opaque struct containing all the variables needed for *Voc* to work. Like all Soundpipe modules, this struct has the prefix "sp".

⟨ Voc Main Data Struct 5 ⟩ ≡
```
struct sp_voc {
    glottis glot;      /* The Glottis */
    tract tr;      /* The Vocal Tract  */
    SPFLOAT buf[512];
    int counter;
};
```
This code is used in section 4.

**6.** The glottis data structure contains all the variables used by the glottis. See ⟨ The Glottis 31 ⟩ to see the implementation of the glottal sound source.

- *freq* is the frequency
- *tenseness* is the tenseness of the glottis (more or less looks like a cross fade between voiced and unvoiced sound). It is a value in the range $[0, 1]$.
- *Rd*
- *waveform_length* provides the period length (in seconds) of the fundamental frequency, in seconds.
- The waveform position is kept track of in *time_in_waveform*, in seconds.
- *alpha*
- E0
- *epsilon*
- *shift*
- *delta*
- *Te*
- *omega*
- *T*

⟨ Glottis Data Structure 6 ⟩ ≡
```
typedef struct {
    SPFLOAT freq;
    SPFLOAT tenseness;
    SPFLOAT Rd;
    SPFLOAT waveform_length;
    SPFLOAT time_in_waveform;
    SPFLOAT alpha;
    SPFLOAT E0;
    SPFLOAT epsilon;
    SPFLOAT shift;
    SPFLOAT delta;
    SPFLOAT Te;
    SPFLOAT omega;
    SPFLOAT T;
} glottis;
```
This code is used in section 4.

**7.**

⟨ Transient Data 7 ⟩ ≡
```
⟨ A Single Transient 8 ⟩
⟨ The Transient Pool 9 ⟩
```
This code is used in section 4.

**8.** This data struct outlines the data for a single transient. A transient will act as a single entry in a linked list implementation, so there exists a *next* pointer along with the `SPFLOAT` parameters.

⟨ A Single Transient 8 ⟩ ≡
```
typedef struct transient {
  int position;
  SPFLOAT time_alive;
  SPFLOAT lifetime;
  SPFLOAT strength;
  SPFLOAT exponent;
  char is_free;
  unsigned int id;
  struct transient *next;
} transient;
```
This code is used in section 7.

**9.** A pre-allocated set of transients and other parameters are used in what will be known as a *transient pool*. A memory pool is an ideal choice for realtime systems instead of dynamic memory. Calls to *malloc* are discouraged because it adds performance overhead and possible blocking behavior, and there is a greater chance of memory leaks or segfaults if not handled properly.

⟨ The Transient Pool 9 ⟩ ≡
```
typedef struct {
  transient pool[MAX_TRANSIENTS];
  transient *root;
  int size;
  int next_free;
} transient_pool;
```
This code is used in section 7.

**10.** The Tract C struct contains all the data needed for the vocal tract filter.

```
⟨ Tract Data 10 ⟩ ≡
  typedef struct {
    int n;    n is the size, set to 44.
    SPFLOAT diameter[44];
    SPFLOAT rest_diameter[44];
    SPFLOAT target_diameter[44];
    SPFLOAT new_diameter[44];
    SPFLOAT R[44];    component going right
    SPFLOAT L[44];    component going left
    SPFLOAT reflection[45];
    SPFLOAT new_reflection[45];
    SPFLOAT junction_outL[45];
    SPFLOAT junction_outR[45];
    SPFLOAT A[44];
    int nose_length;    The original code here has it at floor(28 * n/44), and since n=44, it should be 28.
    int nose_start;    n − nose_length + 1, or 17
     tip_start is a constant set to 32
    int tip_start;
    SPFLOAT noseL[28];
    SPFLOAT noseR[28];
    SPFLOAT nose_junc_outL[29];
    SPFLOAT nose_junc_outR[29];
    SPFLOAT nose_reflection[29];
    SPFLOAT nose_diameter[28];
    SPFLOAT noseA[28];
    SPFLOAT reflection_left;
    SPFLOAT reflection_right;
    SPFLOAT reflection_nose;
    SPFLOAT new_reflection_left;
    SPFLOAT new_reflection_right;
    SPFLOAT new_reflection_nose;
    SPFLOAT velum_target;
    SPFLOAT glottal_reflection;
    SPFLOAT lip_reflection;
    int last_obstruction;
    SPFLOAT fade;
    SPFLOAT movement_speed;
     15 cm/s SPFLOAT lip_output;
    SPFLOAT nose_output;
    SPFLOAT block_time;
    transient_pool tpool;
    SPFLOAT T;
  } tract;
```

This code is used in section 4.

# Top-level Functions (11)

Broadly speaking, the top-level functions are in charge of computing samples for the DSP inner-loop before, after, and during runtime. They get their name from the fact that they are the top level of abstraction in the program. These are the functions that get called in the Sporth Unit Generator implementation ⟨ `ugen.c` 67 ⟩.

⟨ Top Level Functions 11 ⟩ ≡
- ⟨ Voc Create 12 ⟩
- ⟨ Voc Destroy 13 ⟩
- ⟨ Voc Initialization 14 ⟩
- ⟨ Voc Compute 15 ⟩
- ⟨ Voc Tract Compute 16 ⟩
- ⟨ Voc Set Frequency 17 ⟩
- ⟨ Voc Get Frequency 18 ⟩
- ⟨ Voc Get Tract Diameters 19 ⟩
- ⟨ Voc Get Current Tract Diameters 20 ⟩
- ⟨ Voc Get Tract Size 21 ⟩
- ⟨ Voc Get Nose Diameters 22 ⟩
- ⟨ Voc Get Nose Size 23 ⟩
- ⟨ Voc Set Diameters 24 ⟩
- ⟨ Voc Set Tongue Shape 25 ⟩
- ⟨ Voc Get Counter 26 ⟩
- ⟨ Voc Set Tenseness 27 ⟩
- ⟨ Voc Get Tenseness 28 ⟩
- ⟨ Voc Set Velum 29 ⟩
- ⟨ Voc Get Velum 30 ⟩

This code is cited in sections 1 and 65.

This code is used in section 1.

**12.** In the function *sp_voc_create*, an instance of Voc is created via *malloc*.

⟨ Voc Create 12 ⟩ ≡
```
int sp_voc_create(sp_voc **voc)
{
    *voc = malloc(sizeof(sp_voc));
    return SP_OK;
}
```

This code is cited in sections 49 and 68.

This code is used in section 11.

**13.** As a counterpart to *sp_voc_compute*, *sp_voc_destroy* frees all data previous allocated.

⟨ Voc Destroy 13 ⟩ ≡
```
int sp_voc_destroy(sp_voc **voc)
{
    free(*voc);
    return SP_OK;
}
```

This code is cited in section 71.

This code is used in section 11.

**14.** After data has been allocated with *sp_voc_create*, it must be initialized with *sp_voc_init*.

```
⟨ Voc Initialization 14 ⟩ ≡
  int sp_voc_init(sp_data * sp, sp_voc *voc)
  {
    glottis_init(&voc→glot, sp→sr);      /* initialize glottis */
    tract_init(sp, &voc→tr);      /* initialize vocal tract */
    voc→counter = 0;
    return SP_OK;
  }
```

This code is cited in section 69.

This code is used in section 11.

**15.** The function *sp_voc_compute* is called during runtime to generate audio. This computation function will generate a single sample of audio and store it in the SPFLOAT pointer *out.

```
⟨ Voc Compute 15 ⟩ ≡
  int sp_voc_compute(sp_data * sp, sp_voc *voc, SPFLOAT * out)
  {
    SPFLOAT vocal_output, glot;
    SPFLOAT lambda1, lambda2;
    int i;
    if (voc→counter ≡ 0) {
      tract_reshape(&voc→tr);
      tract_calculate_reflections(&voc→tr);
      for (i = 0; i < 512; i++) {
        vocal_output = 0;
        lambda1 = (SPFLOAT)i/512;
        lambda2 = (SPFLOAT)(i + 0.5)/512;
        glot = glottis_compute(sp, &voc→glot, lambda1);
        tract_compute(sp, &voc→tr, glot, lambda1);
        vocal_output += voc→tr.lip_output + voc→tr.nose_output;
        tract_compute(sp, &voc→tr, glot, lambda2);
        vocal_output += voc→tr.lip_output + voc→tr.nose_output;
        voc→buf[i] = vocal_output * 0.125;
      }
    }
    *out = voc→buf[voc→counter];
    voc→counter = (voc→counter + 1) % 512;
    return SP_OK;
  }
```

This code is cited in sections 16, 26, and 70.

This code is used in section 11.

**16.** The function *sp_voc_compute_tract* computes the vocal tract component of Voc separately from the glottis. This provides the ability to use any input signal as an glottal excitation, turning the model into a formant filter. Compared to the main implementation in ⟨ Voc Compute 15 ⟩, this function does not have the 512 sample delay.

```
⟨ Voc Tract Compute 16 ⟩ ≡
  int sp_voc_tract_compute(sp_data * sp, sp_voc *voc, SPFLOAT * in, SPFLOAT * out)
  {
    SPFLOAT vocal_output;
    SPFLOAT lambda1, lambda2;
    if (voc→counter ≡ 0) {
      tract_reshape(&voc→tr);
      tract_calculate_reflections(&voc→tr);
    }
    vocal_output = 0;
    lambda1 = (SPFLOAT)voc→counter/512;
    lambda2 = (SPFLOAT)(voc→counter + 0.5)/512;
    tract_compute(sp, &voc→tr, *in, lambda1);
    vocal_output += voc→tr.lip_output + voc→tr.nose_output;
    tract_compute(sp, &voc→tr, *in, lambda2);
    vocal_output += voc→tr.lip_output + voc→tr.nose_output;
    *out = vocal_output * 0.125;
    voc→counter = (voc→counter + 1) % 512;
    return SP_OK;
  }
```

This code is used in section 11.

**17.** The function *sp_voc_set_frequency* sets the fundamental frequency for the glottal wave.

```
⟨ Voc Set Frequency 17 ⟩ ≡
  void sp_voc_set_frequency(sp_voc *voc, SPFLOAT freq)
  {
    voc→glot.freq = freq;
  }
```

This code is used in section 11.

**18.** The function *sp_voc_get_frequency_ptr* returns a pointer to the variable holding the frequency. This allows values to be set and read directly without. The use of a helper function. This function was notably created for use in a demo using the GUI library Nuklear.

```
⟨ Voc Get Frequency 18 ⟩ ≡
  SPFLOAT * sp_voc_get_frequency_ptr(sp_voc *voc)
  {
    return &voc→glot.freq;
  }
```

This code is used in section 11.

**19.** This getter function returns the cylindrical diameters representing tract.

```
⟨ Voc Get Tract Diameters 19 ⟩ ≡
  SPFLOAT * sp_voc_get_tract_diameters(sp_voc *voc)
  {
    return voc->tr.target_diameter;
  }
```

This code is cited in section 20.

This code is used in section 11.

**20.** Similar to *sp_voc_get_tract_diameters* in ⟨ Voc Get Tract Diameters 19 ⟩, the function *sp_voc_get_current_tract_diamete* returns the diameters of the tract. The difference is that this function returns the actual slewed diameters used in ⟨ Reshape Vocal Tract 57 ⟩, rather than the target diameters.

```
⟨ Voc Get Current Tract Diameters 20 ⟩ ≡
  SPFLOAT * sp_voc_get_current_tract_diameters(sp_voc *voc)
  {
    return voc->tr.diameter;
  }
```

This code is used in section 11.

**21.** This getter function returns the size of the vocal tract.

```
⟨ Voc Get Tract Size 21 ⟩ ≡
  int sp_voc_get_tract_size(sp_voc *voc)
  {
    return voc->tr.n;
  }
```

This code is used in section 11.

**22.** This function returns the cylindrical diameters of the nasal cavity.

```
⟨ Voc Get Nose Diameters 22 ⟩ ≡
  SPFLOAT * sp_voc_get_nose_diameters(sp_voc *voc)
  {
    return voc->tr.nose_diameter;
  }
```

This code is used in section 11.

**23.** This function returns the nose size.

```
⟨ Voc Get Nose Size 23 ⟩ ≡
  int sp_voc_get_nose_size(sp_voc *voc)
  {
    return voc->tr.nose_length;
  }
```

This code is used in section 11.

**24.** The function *sp_voc_set_diameter*( ) is a function adopted from Neil Thapen's Pink Trombone in a function he called setRestDiameter. It is the main function in charge of the "tongue position" XY control. Modifications to the original function have been made in an attempt to make the function more generalized. Instead of relying on internal state, all variables used are parameters in the function. Because of this fact, there are quite a few function parameters:

- **voc**, the core Voc data struct
- **blade_start**, index where the blade (?) starts. this is set to 10 in pink trombone
- **lip_start**, index where lip starts. this constant is set to 39.
- **tip_start**, this is set to 32.
- **tongue_index**
- **tongue_diameter**
- **diameters**, the floating point array to write to

For practical use cases, it is not ideal to call this function directly. Instead, it can be indirectly called using a more sane function *sp_voc_set_tongue_shape*( ), found in the section ⟨ Voc Set Tongue Shape 25 ⟩.

⟨ Voc Set Diameters 24 ⟩ ≡

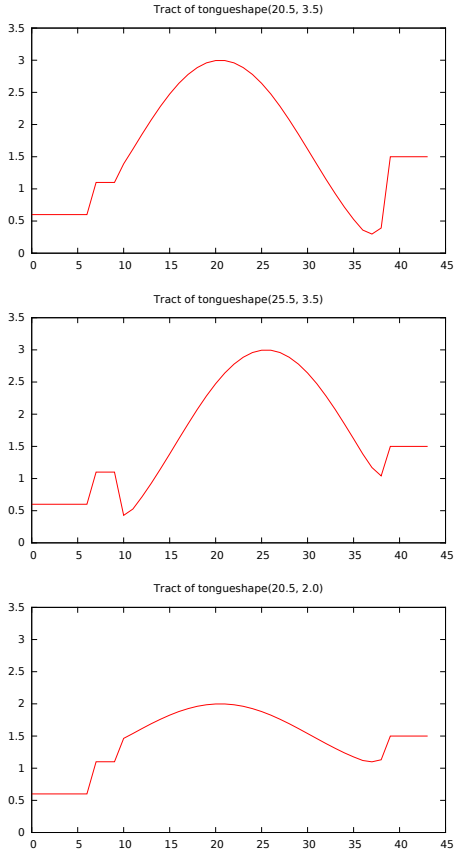
```
void sp_voc_set_diameters(sp_voc *voc,
int blade_start,
int lip_start,
int tip_start,
SPFLOAT tongue_index,
SPFLOAT tongue_diameter,
SPFLOAT * diameters)
{
  int i;
  SPFLOAT t;
  SPFLOAT fixed_tongue_diameter;
  SPFLOAT curve;
  int grid_offset = 0;
  for (i = blade_start; i < lip_start; i++) {
    t = 1.1 * M_PI * (SPFLOAT)(tongue_index - i)/(tip_start - blade_start);
    fixed_tongue_diameter = 2 + (tongue_diameter - 2)/1.5;
    curve = (1.5 - fixed_tongue_diameter + grid_offset) * cos(t);
    if (i ≡ blade_start - 2 ∨ i ≡ lip_start - 1) curve *= 0.8;
    if (i ≡ blade_start ∨ i ≡ lip_start - 2) curve *= 0.94;
    diameters[i] = 1.5 - curve;
  }
}
```

This code is cited in section 25.

This code is used in section 11.

**25.** The function *sp_voc_set_tongue_shape*( ) will set the shape of the tongue using the two primary arguments *tongue_index* and *tongue_diameter*. It is a wrapper around the function described in ⟨ Voc Set Diameters 24 ⟩, filling in the constants used, and thereby making it simpler to work with.

A few tract shapes shaped using this function have been generated below:




⟨ Voc Set Tongue Shape 25 ⟩ ≡

```
void sp_voc_set_tongue_shape(sp_voc *voc, SPFLOAT tongue_index,
SPFLOAT tongue_diameter)
{
  SPFLOAT * diameters;
  diameters = sp_voc_get_tract_diameters(voc);
  sp_voc_set_diameters(voc, 10, 39, 32, tongue_index, tongue_diameter, diameters);
}
```

This code is cited in sections 24 and 76.

This code is used in section 11.

**26.** Voc keeps an internal counter for control rate operations called inside of the audio-rate compute function in ⟨ Voc Compute 15 ⟩. The function *sp_voc_get_counter*( ) gets the current counter position. When the counter is 0, the next call to *sp_voc_compute* will compute another block of audio. Getting the counter position before the call allows control-rate variables to be set before then.

```
⟨ Voc Get Counter 26 ⟩ ≡
  int sp_voc_get_counter(sp_voc *voc)
  {
    return voc→counter;
  }
```

This code is used in section 11.

**27.** The function *sp_voc_set_tenseness* is used to set the tenseness variable, used when calculating glottal time coefficients in ⟨ Set up Glottis Waveform 34 ⟩, and is the main factor in calculating aspiration noise in ⟨ Glottis Computation 33 ⟩. Typically this is a value between 0 and 1. A value of 1 gives a full vocal sound, while a value of 0 is all breathy. It is ideal to have a little bit of aspiration noise. Empirically good values tend to be in the range of $[0.6, 0.9]$.

```
⟨ Voc Set Tenseness 27 ⟩ ≡
  void sp_voc_set_tenseness(sp_voc *voc, SPFLOAT tenseness)
  {
    voc→glot.tenseness = tenseness;
  }
```

This code is used in section 11.

**28.** The function *sp_voc_get_tenseness_ptr* returns an SPFLOAT pointer to the parameter value directly controlling tenseness. This function is useful for GUI frontends that use direct pointer manipulation like Nuklear, the cross-platform UI framework used to make a demo for Voc.

```
⟨ Voc Get Tenseness 28 ⟩ ≡
  SPFLOAT * sp_voc_get_tenseness_ptr(sp_voc *voc)
  {
    return &voc→glot.tenseness;
  }
```

This code is used in section 11.

**29.** The function *sp_voc_set_velum* sets the *velum*, or soft pallette of tract model. In the original implementation, the default value is 0.01, and set to a value of 0.04 to get a nasally sound.

```
⟨ Voc Set Velum 29 ⟩ ≡
  void sp_voc_set_velum(sp_voc *voc, SPFLOAT velum)
  {
    voc→tr.velum_target = velum;
  }
```

This code is used in section 11.

**30.** The function *sp_voc_get_velum_ptr* returns the pointer associated with the velum, allowing direct control of the velum parameter. This function was created for use with a demo requiring direct access.

```
⟨ Voc Get Velum 30 ⟩ ≡
  SPFLOAT * sp_voc_get_velum_ptr(sp_voc *voc)
  {
    return &voc→tr.velum_target;
  }
```

This code is used in section 11.

# The Glottis (31)

This is where the synthesis of the glottal source signal will be created.

While the implementation comes directly from Pink Trombone's JavaScript code, it should be noted that the glottal model is based on a modified LF-model[lu2000glottal].

⟨ The Glottis 31 ⟩ ≡
⟨ Set up Glottis Waveform 34 ⟩
⟨ Glottis Initialization 32 ⟩
⟨ Glottis Computation 33 ⟩

This code is cited in sections 1 and 6.

This code is used in section 1.

**32.** Initializiation of the glottis is done inside of *glottis_init*.

⟨ Glottis Initialization 32 ⟩ ≡

```
static void glottis_init(glottis *glot, SPFLOAT sr)
{
    glot->freq = 140;       /* 140Hz frequency by default */
    glot->tenseness = 0.6;      /* value between 0 and 1 */
    glot->T = 1.0/sr;       /* big T */
    glot->time_in_waveform = 0;
    glottis_setup_waveform(glot, 0);
}
```

This code is used in section 31.

**33.** This is where a single sample of audio is computed for the glottis

⟨ Glottis Computation 33 ⟩ ≡

```
static SPFLOAT glottis_compute(sp_data * sp, glottis *glot, SPFLOAT lambda)
{
    SPFLOAT out;
    SPFLOAT aspiration;
    SPFLOAT noise;
    SPFLOAT t;
    SPFLOAT intensity;
    out = 0;
    intensity = 1.0;
    glot->time_in_waveform += glot->T;
    if (glot->time_in_waveform > glot->waveform_length) {
        glot->time_in_waveform -= glot->waveform_length;
        glottis_setup_waveform(glot, lambda);
    }
    t = (glot->time_in_waveform / glot->waveform_length);
    if (t > glot->Te) {
        out = (-exp(-glot->epsilon * (t - glot->Te)) + glot->shift)/glot->delta;
    }
    else {
        out = glot->E0 * exp(glot->alpha * t) * sin(glot->omega * t);
    }
    noise = 2.0 * ((SPFLOAT)sp_rand(sp)/SP_RANDMAX) - 1;
    aspiration = intensity * (1 - sqrt(glot->tenseness)) * 0.3 * noise;
    aspiration *= 0.2;
    out += aspiration;
    return out;
}
```

This code is cited in sections 27 and 36.

This code is used in section 31.

**34.** The function *glottis_setup_waveform* is tasked with setting the variables needed to create the glottis waveform. The glottal model used here is known as the LF-model, as described in Lu and Smith[lu2000glottal].

⟨ Set up Glottis Waveform 34 ⟩ ≡

```
static void glottis_setup_waveform(glottis *glot, SPFLOAT lambda){⟨ Set up local variables 35 ⟩
    ⟨ Derive waveform_length and R_d 36 ⟩
    ⟨ Derive R_a, R_k, and R_g 37 ⟩
    ⟨ Derive T_a, T_p, and T_e 38 ⟩
    ⟨ Calculate epsilon, shift, and delta 39 ⟩
    ⟨ Calculate Integrals 40 ⟩
    ⟨ Calculate E_0 41 ⟩
    ⟨ Update variables in glottis data structure 42 ⟩}
```

This code is cited in section 27.

This code is used in section 31.

**35.** A number of local variables are used for intermediate calculations. They are described below.

⟨ Set up local variables 35 ⟩ ≡
```
SPFLOAT Rd;
SPFLOAT Ra;
SPFLOAT Rk;
SPFLOAT Rg;
SPFLOAT Ta;
SPFLOAT Tp;
SPFLOAT Te;
SPFLOAT epsilon;
SPFLOAT shift;
SPFLOAT delta;
SPFLOAT rhs_integral;
SPFLOAT lower_integral;
SPFLOAT upper_integral;
SPFLOAT omega;
SPFLOAT s;
SPFLOAT y;
SPFLOAT z;
SPFLOAT alpha;
SPFLOATE0;
```
This code is used in section 34.

**36.** To begin, both *waveform_length* and $R_d$ are calcuated.

The variable *waveform_length* is the period of the waveform based on the current frequency, and will be used later on in ⟨ Glottis Computation 33 ⟩.

$R_d$ is part of a set of normalized timing parameters used to calculate the time coefficients described in the LF model [fant1997voice]. The other timing parameters $R_a$, $R_g$, and $R_k$ can be computed in terms of $R_d$, which is why this gets computed first. $R_d$ is derived from the parameter *glot→tenseness*.

$R_d$ is then clamped to be in between 0.5 and 2.7, as these are good approximations[lu2000glottal].

⟨ Derive *waveform_length* and $R_d$ 36 ⟩ ≡
```
glot→Rd = 3 * (1 - glot→tenseness);
glot→waveform_length = 1.0/glot→freq;
Rd = glot→Rd;
if (Rd < 0.5) Rd = 0.5;
if (Rd > 2.7) Rd = 2.7;
```
This code is used in section 34.

**37.** $R_d$ can be used to calculate approximations for $R_a$, $R_g$, and $R_k$. The equations described below have been derived using linear regression.

$$R_{ap} = \frac{(-1 + 4.8R_d)}{100}$$

$$R_{kp} = \frac{(22.4 + 11.8R_d)}{100}$$

$R_{gp}$ is derived using the results from $R_{ap}$ and $R_{kp}$ in the following equation described in Fant 1997:

$$R_d = (1/0.11)(0.5 + 1.2R_k)(R_k/4R_g + R_a)$$

Which yields:

$$R_{gp} = \frac{(R_{kp}/4)(0.5 + 1.2R_{kp})}{(0.11R_d - R_{ap} * (0.5 + 1.2R_{kp}))}$$

⟨ Derive $R_a$, $R_k$, and $R_g$ 37 ⟩ ≡

```
Ra = -0.01 + 0.048 * Rd;
Rk = 0.224 + 0.118 * Rd;
Rg = (Rk/4) * (0.5 + 1.2 * Rk)/(0.11 * Rd - Ra * (0.5 + 1.2 * Rk));
```

This code is used in section 34.

**38.** The parameters approximating $R_a$, $R_g$, and $R_k$ can be used to calculate the timing parameters $T_a$, $T_p$, and $T_e$ in the LF model:

$$T_a = R_{ap}$$

$$T_p = 2R_{gp}^{-1}$$

$$T_e = T_p + T_pR_{kp}$$

⟨ Derive $T_a$, $T_p$, and $T_e$ 38 ⟩ ≡

```
Ta = Ra;
Tp = (SPFLOAT)1.0/(2 * Rg);
Te = Tp + Tp * Rk;
```

This code is used in section 34.

**39.** ⟨ Calculate epsilon, shift, and delta 39 ⟩ ≡

```
epsilon = (SPFLOAT)1.0/ Ta;
shift = exp(-epsilon * (1 - Te));
delta = 1 - shift;
```

This code is used in section 34.

**40.** ⟨ Calculate Integrals 40 ⟩ ≡

```
rhs_integral = (SPFLOAT)(1.0/epsilon) * (shift - 1) + (1 - Te) * shift;
rhs_integral = rhs_integral/delta;
lower_integral = -(Te - Tp)/2 + rhs_integral;
upper_integral = -lower_integral;
```

This code is used in section 34.

**41.**

$$E_0 = -\frac{E_e}{e^{\alpha T}\sin\omega_g T_e}$$

$$\omega = \frac{\pi}{T_p}$$

$$\epsilon T_a = 1 - e^{-\epsilon(T_c - T_e)}$$

⟨ Calculate $E_0$ 41 ⟩ ≡

```
omega = M_PI / Tp;
s = sin(omega * Te);
y = -M_PI * s * upper_integral / (Tp * 2);
z = log(y);
alpha = z / (Tp / 2 - Te);
E0 = -1 / (s * exp(alpha * Te));
```

This code is used in section 34.

**42.** ⟨ Update variables in glottis data structure 42 ⟩ ≡

```
glot->alpha = alpha;
glot->E0 = E0;
glot->epsilon = epsilon;
glot->shift = shift;
glot->delta = delta;
glot->Te = Te;
glot->omega = omega;
```

This code is used in section 34.

# The Vocal Tract (43)

The vocal tract is the part of the vocal model which takes the excitation signal (the glottis) and produces the vowel formants from it.

The two main functions for the vocal tract consist of of an initialization function *tract_init* called once before runtime, and a computation function *tract_compute* called at twice the sampling rate. See ⟨ Vocal Tract Initialization 44 ⟩ and ⟨ Vocal Tract Computation 49 ⟩ for more detail.

```
⟨ The Vocal Tract 43 ⟩ ≡
  ⟨ Calculate Vocal Tract Reflections 55 ⟩
  ⟨ Calculate Vocal Tract Nose Reflections 56 ⟩
  ⟨ Vocal Tract Transients 58 ⟩
  ⟨ Reshape Vocal Tract 57 ⟩
  ⟨ Vocal Tract Initialization 44 ⟩
  ⟨ Vocal Tract Computation 49 ⟩
```

This code is cited in sections 1 and 66.

This code is used in section 1.

**44.** The function *tract_init* is responsible for zeroing out variables and buffers, as well as setting up constants.

```
⟨ Vocal Tract Initialization 44 ⟩ ≡
  static void tract_init(sp_data * sp, tract *tr)
  {
    int i;
    SPFLOAT diameter, d;      /* needed to set up diameter arrays */
    ⟨ Initialize Tract Constants and Variables 45 ⟩
    ⟨ Zero Out Tract Buffers 46 ⟩
    ⟨ Set up Vocal Tract Diameters 47 ⟩
    ⟨ Set up Nose Diameters 48 ⟩
    tract_calculate_reflections(tr);
    tract_calculate_nose_reflections(tr);
    tr→nose_diameter[0] = tr→velum_target;
    tr→block_time = 512.0/(SPFLOAT)sp→sr;
    tr→T = 1.0/(SPFLOAT)sp→sr;
    ⟨ Initialize Transient Pool 59 ⟩
  }
```

This code is cited in sections 43 and 59.

This code is used in section 43.

**45.** ⟨ Initialize Tract Constants and Variables 45 ⟩ ≡

```
tr→n = 44;
tr→nose_length = 28;
tr→nose_start = 17;
tr→reflection_left = 0.0;
tr→reflection_right = 0.0;
tr→reflection_nose = 0.0;
tr→new_reflection_left = 0.0;
tr→new_reflection_right = 0.0;
tr→new_reflection_nose = 0.0;
tr→velum_target = 0.01;
tr→glottal_reflection = 0.75;
tr→lip_reflection = −0.85;
tr→last_obstruction = −1;
tr→movement_speed = 15;
tr→lip_output = 0;
tr→nose_output = 0;
tr→tip_start = 32;
```

This code is used in section 44.

**46.** Several floating-point arrays are needed for the scattering junctions. C does not zero these out by default. Below, the standard function *memset*( ) from *string.h* is used to zero out each of the blocks of memory.

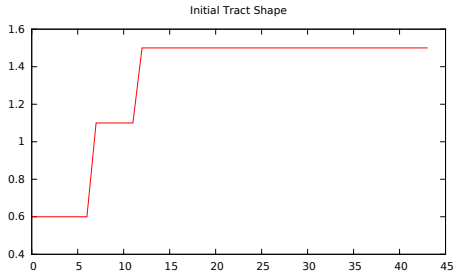
⟨ Zero Out Tract Buffers 46 ⟩ ≡

```
memset(tr→diameter, 0, tr→n * sizeof (SPFLOAT));
memset(tr→rest_diameter, 0, tr→n * sizeof (SPFLOAT));
memset(tr→target_diameter, 0, tr→n * sizeof (SPFLOAT));
memset(tr→new_diameter, 0, tr→n * sizeof (SPFLOAT));
memset(tr→L, 0, tr→n * sizeof (SPFLOAT));
memset(tr→R, 0, tr→n * sizeof (SPFLOAT));
memset(tr→reflection, 0, (tr→n + 1) * sizeof (SPFLOAT));
memset(tr→new_reflection, 0, (tr→n + 1) * sizeof (SPFLOAT));
memset(tr→junction_outL, 0, (tr→n + 1) * sizeof (SPFLOAT));
memset(tr→junction_outR, 0, (tr→n + 1) * sizeof (SPFLOAT));
memset(tr→A, 0, tr→n * sizeof (SPFLOAT));
memset(tr→noseL, 0, tr→nose_length * sizeof (SPFLOAT));
memset(tr→noseR, 0, tr→nose_length * sizeof (SPFLOAT));
memset(tr→nose_junc_outL, 0, (tr→nose_length + 1) * sizeof (SPFLOAT));
memset(tr→nose_junc_outR, 0, (tr→nose_length + 1) * sizeof (SPFLOAT));
memset(tr→nose_diameter, 0, tr→nose_length * sizeof (SPFLOAT));
memset(tr→noseA, 0, tr→nose_length * sizeof (SPFLOAT));
```

This code is used in section 44.

**47.** The cylindrical diameters approximating the vocal tract are set up below. These diameters will be modified and shaped by user control to shape the vowel sound.

The initial shape of the vocal tract is plotted below:



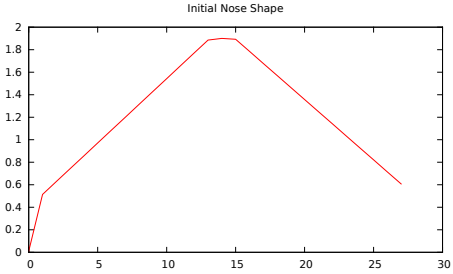

⟨ Set up Vocal Tract Diameters 47 ⟩ ≡

```
for (i = 0; i < tr→n; i++) {
  diameter = 0;
  if (i < 7 * (SPFLOAT)tr→n/44 − 0.5) {
    diameter = 0.6;
  }
  else if (i < 12 * (SPFLOAT)tr→n/44) {
    diameter = 1.1;
  }
  else {
    diameter = 1.5;
  }
  tr→diameter[i] = tr→rest_diameter[i] = tr→target_diameter[i] = tr→new_diameter[i] = diameter;
}
```

This code is used in section 44.

**48.** The cylindrical diameters representing nose are set up. These are only set once, and are immutable for the rest of the program.

The shape of the nasal passage is plotted below:




⟨ Set up Nose Diameters 48 ⟩ ≡

```
for (i = 0; i < tr→nose_length; i++) {
  d = 2 * ((SPFLOAT)i/tr→nose_length);
  if (d < 1) {
    diameter = 0.4 + 1.6 * d;
  }
  else {
    diameter = 0.5 + 1.5 * (2 − d);
  }
  diameter = MIN(diameter, 1.9);
  tr→nose_diameter[i] = diameter;
}
```

This code is used in section 44.

**49.** The vocal tract computation function computes a single sample of audio. As the original implementation describes it, this function is designed to run at twice the sampling rate. For this reason, it is called twice in the top level call back (see ⟨ Voc Create 12 ⟩).

*tract_compute* has two input arguments. The variable *in* is the glottal excitation signal. The *lambda* variable is a coefficient for a linear crossfade along the buffer block, used for parameter smoothing.

⟨ Vocal Tract Computation 49 ⟩ ≡

```
static void tract_compute(sp_data * sp, tract *tr, SPFLOAT in, SPFLOAT lambda)
{
  SPFLOAT r, w;
  int i;
  SPFLOAT amp;
  int current_size;
  transient_pool *pool;
  transient *n;
  SPFLOAT noise;
  ⟨ Process Transients 62 ⟩
  ⟨ Calculate Scattering Junctions 50 ⟩
  ⟨ Calculate Scattering for Nose 51 ⟩
  ⟨ Update Left/Right delay lines and set lip output 52 ⟩
  ⟨ Calculate Nose Scattering Junctions 53 ⟩
  ⟨ Update Nose Left/Right delay lines and set nose output 54 ⟩
}
```

This code is cited in sections 43 and 62.

This code is used in section 43.

**50.** A derivation of $w$ can be seen in section 2.5.2 of Jack Mullens PhD dissertation *Physical Modelling of the Vocal Tract with the 2D Digital Waveguide Mesh.* [mullen2006physical]

⟨ Calculate Scattering Junctions 50 ⟩ ≡
```
tr→junction_outR[0] = tr→L[0] * tr→glottal_reflection + in;
tr→junction_outL[tr→n] = tr→R[tr→n − 1] * tr→lip_reflection;
for (i = 1; i < tr→n; i++) {
  r = tr→reflection[i] * (1 − lambda) + tr→new_reflection[i] * lambda;
  w = r * (tr→R[i − 1] + tr→L[i]);
  tr→junction_outR[i] = tr→R[i − 1] − w;
  tr→junction_outL[i] = tr→L[i] + w;
}
```
This code is used in section 49.

**51.** ⟨ Calculate Scattering for Nose 51 ⟩ ≡
```
i = tr→nose_start;
r = tr→new_reflection_left * (1 − lambda) + tr→reflection_left * lambda;
tr→junction_outL[i] = r * tr→R[i − 1] + (1 + r) * (tr→noseL[0] + tr→L[i]);
r = tr→new_reflection_right * (1 − lambda) + tr→reflection_right * lambda;
tr→junction_outR[i] = r * tr→L[i] + (1 + r) * (tr→R[i − 1] + tr→noseL[0]);
r = tr→new_reflection_nose * (1 − lambda) + tr→reflection_nose * lambda;
tr→nose_junc_outR[0] = r * tr→noseL[0] + (1 + r) * (tr→L[i] + tr→R[i − 1]);
```
This code is used in section 49.

**52.** ⟨ Update Left/Right delay lines and set lip output 52 ⟩ ≡
```
for (i = 0; i < tr→n; i++) {
  tr→R[i] = tr→junction_outR[i] * 0.999;
  tr→L[i] = tr→junction_outL[i + 1] * 0.999;
}
tr→lip_output = tr→R[tr→n − 1];
```
This code is used in section 49.

**53.** ⟨ Calculate Nose Scattering Junctions 53 ⟩ ≡
```
tr→nose_junc_outL[tr→nose_length] = tr→noseR[tr→nose_length − 1] * tr→lip_reflection;
for (i = 1; i < tr→nose_length; i++) {
  w = tr→nose_reflection[i] * (tr→noseR[i − 1] + tr→noseL[i]);
  tr→nose_junc_outR[i] = tr→noseR[i − 1] − w;
  tr→nose_junc_outL[i] = tr→noseL[i] + w;
}
```
This code is used in section 49.

**54.** ⟨ Update Nose Left/Right delay lines and set nose output 54 ⟩ ≡
```
for (i = 0; i < tr→nose_length; i++) {
  tr→noseR[i] = tr→nose_junc_outR[i];
  tr→noseL[i] = tr→nose_junc_outL[i + 1];
}
tr→nose_output = tr→noseR[tr→nose_length − 1];
```
This code is used in section 49.

**55.** The function *tract_calculate_reflections* computes reflection coefficients used in the scattering junction. Because this is a rather computationally expensive function, it is called once per render block, and then smoothed.

First, the cylindrical areas of tract section are computed by squaring the diameters, they are stored in the struct variable $A$.

Using the areas calculated, the reflections are calculated using the following formula:

$$R_i = \frac{A_{i-1} - A_i}{A_{i-1} + A_i}$$

To prevent some divide-by-zero edge cases, when $A_i$ is exactly zero, it is set to be 0.999.

From there, the new coefficients are set.

⟨ Calculate Vocal Tract Reflections 55 ⟩ ≡

```
static void tract_calculate_reflections(tract *tr)
{
  int i;
  SPFLOAT sum;
  for (i = 0; i < tr→n; i++) {
    tr→A[i] = tr→diameter[i] * tr→diameter[i];      /* Calculate area from diameter squared */
  }
  for (i = 1; i < tr→n; i++) {
    tr→reflection[i] = tr→new_reflection[i];
    if (tr→A[i] == 0) {
      tr→new_reflection[i] = 0.999;      /* to prevent bad behavior if 0 */
    }
    else {
      tr→new_reflection[i] = (tr→A[i - 1] - tr→A[i])/(tr→A[i - 1] + tr→A[i]);
    }
  }
  tr→reflection_left = tr→new_reflection_left;
  tr→reflection_right = tr→new_reflection_right;
  tr→reflection_nose = tr→new_reflection_nose;
  sum = tr→A[tr→nose_start] + tr→A[tr→nose_start + 1] + tr→noseA[0];
  tr→new_reflection_left = (SPFLOAT)(2 * tr→A[tr→nose_start] - sum)/sum;
  tr→new_reflection_right = (SPFLOAT)(2 * tr→A[tr→nose_start + 1] - sum)/sum;
  tr→new_reflection_nose = (SPFLOAT)(2 * tr→noseA[0] - sum)/sum;
}
```

This code is cited in section 56.

This code is used in section 43.

**56.** Similar to *tract_calculate_reflections*, this function computes reflection coefficients for the nasal scattering junction. For more information on the math that is happening, see ⟨ Calculate Vocal Tract Reflections 55 ⟩.

⟨ Calculate Vocal Tract Nose Reflections 56 ⟩ ≡

```
static void tract_calculate_nose_reflections(tract *tr)
{
  int i;
  for (i = 0; i < tr→nose_length; i++) {
    tr→noseA[i] = tr→nose_diameter[i] * tr→nose_diameter[i];
  }
  for (i = 1; i < tr→nose_length; i++) {
    tr→nose_reflection[i] = (tr→noseA[i − 1] − tr→noseA[i])/(tr→noseA[i − 1] + tr→noseA[i]);
  }
}
```

This code is used in section 43.

**57.**

⟨ Reshape Vocal Tract 57 ⟩ ≡

```
static SPFLOAT move_towards(SPFLOAT current, SPFLOAT target, SPFLOAT amt_up, SPFLOAT amt_down)
{
  SPFLOAT tmp;
  if (current < target) {
    tmp = current + amt_up;
    return MIN(tmp, target);
  }
  else {
    tmp = current − amt_down;
    return MAX(tmp, target);
  }
  return 0.0;
}
static void tract_reshape(tract *tr)
{
  SPFLOAT amount;
  SPFLOAT slow_return;
  SPFLOAT diameter;
  SPFLOAT target_diameter;
  int i;
  int current_obstruction;
  current_obstruction = −1;
  amount = tr→block_time * tr→movement_speed;
  for (i = 0; i < tr→n; i++) {
    slow_return = 0;
    diameter = tr→diameter[i];
    target_diameter = tr→target_diameter[i];
    if (diameter < 0.001) current_obstruction = i;
    if (i < tr→nose_start) slow_return = 0.6;
    else if (i ≥ tr→tip_start) slow_return = 1.0;
    else {
      slow_return = 0.6 + 0.4 * (i − tr→nose_start)/(tr→tip_start − tr→nose_start);
    }
    tr→diameter[i] = move_towards(diameter, target_diameter, slow_return * amount, 2 * amount);
  }
  if (tr→last_obstruction > −1 ∧ current_obstruction ≡ −1 ∧ tr→noseA[0] < 0.05) {
    append_transient(&tr→tpool, tr→last_obstruction);
  }
  tr→last_obstruction = current_obstruction;
  tr→nose_diameter[0] = move_towards(tr→nose_diameter[0], tr→velum_target, amount * 0.25, amount * 0.1);
  tr→noseA[0] = tr→nose_diameter[0] * tr→nose_diameter[0];
}
```

This code is cited in sections 20, 58, and 60.

This code is used in section 43.

**58.** In Pink Trombone, there is a special handling of diameters that are exactly zero. From a physical point of view, air is completly blocked, and this obstruction of air produces a transient "click" sound. To simulate this, any obstructions are noted during the reshaping of the vocal tract (see ⟨ Reshape Vocal Tract 57 ⟩), and the latest obstruction position is noted and pushed onto a stack of transients. During the vocal tract computation, the exponential damping contributes to the overal amplitude of the left-going and right-going delay lines at that precise diameter location. This can be seen in the section ⟨ Process Transients 62 ⟩.

⟨ Vocal Tract Transients 58 ⟩ ≡
```
⟨ Append Transient 60 ⟩
⟨ Remove Transient 61 ⟩
```
This code is used in section 43.

**59.** The transient pool is initialized inside along with the entire vocal tract inside of ⟨ Vocal Tract Initialization 44 ⟩. It essentially sets the pool to a size of zero and that the first available free transient is at index "0".

The transients in the pool will all have their boolean variable *is_free*, set to be true so that they can be in line to be selected.

⟨ Initialize Transient Pool 59 ⟩ ≡
```
tr→tpool.size = 0;
tr→tpool.next_free = 0;
for (i = 0; i < MAX_TRANSIENTS; i++) {
  tr→tpool.pool[i].is_free = 1;
  tr→tpool.pool[i].id = i;
}
```
This code is used in section 44.

**60.** Any obstructions noted during ⟨ Reshape Vocal Tract 57 ⟩ must be appended to the list of previous transients. The function will return a 0 on failure, and a 1 on success.

Here is an overview of how a transient may get appended:

0. Check and see if the pool is full. If this is so, return 0.
1. If there is no recorded next free (the id is -1), search for one using brute force and check for any free transients. If none can be found, return 0. Since `MAX_TRANSIENTS` is a low N, even the worst-case searches do not pose a significant performance penalty.
2. With a transient found, assign the current root of the list to be the next value in the transient. (It does not matter if the root is NULL, because the size of the list will prevent it from ever being accessed.)
3. Increase the size of the pool by 1.
4. Toggle the *is_free* boolean of the current transient to be false.
5. Set the *position*.
6. Set the *time_alive* to be zero seconds.
7. Set the *lifetime* to be 200ms, or 0.2 seconds.
8. Set the *strength* to an amplitude 0.3.
9. Set the *exponent* parameter to be 200.
10. Set the *next_free* parameter to be $-1$.

⟨ Append Transient 60 ⟩ ≡

```
static int append_transient(transient_pool *pool, int position)
{
  int i;
  int free_id;
  transient *t;
  free_id = pool→next_free;
  if (pool→size ≡ MAX_TRANSIENTS) return 0;
  if (free_id ≡ −1) {
    for (i = 0; i < MAX_TRANSIENTS; i++) {
      if (pool→pool[i].is_free) {
        free_id = i;
        break;
      }
    }
  }
  if (free_id ≡ −1) return 0;
  t = &pool→pool[free_id];
  t→next = pool→root;
  pool→root = t;
  pool→size++;
  t→is_free = 0;
  t→time_alive = 0;
  t→lifetime = 0.15;
  t→strength = 0.6;
  t→exponent = 200;
  pool→next_free = −1;
  return 0;
}
```

This code is used in section 58.

**61.** When a transient has lived it's lifetime, it must be removed from the list of transients. To keep things sane, transients have a unique ID for identification. This is preferred to comparing pointer addresses. While more efficient, this method is prone to subtle implementation errors.

The method for removing a transient from a linked list is fairly typical:

0. If the transient *is* the root, set the root to be the next value. Decrease the size by one, and return.
1. Iterate through the list and search for the entry.
2. Once the entry has been found, decrease the pool size by 1.
3. The transient, now free for reuse, can now be toggled to be free, and it can be the next variable ready to be used again.

⟨ Remove Transient 61 ⟩ ≡

```
static void remove_transient(transient_pool *pool, unsigned int id)
{
  int i;
  transient *n;
  pool→next_free = id;
  n = pool→root;
  if (id ≡ n→id) {
    pool→root = n→next;
    pool→size−−;
    return;
  }
  for (i = 0; i < pool→size; i++) {
    if (n→next→id ≡ id) {
      pool→size−−;
      n→next→is_free = 1;
      n→next = n→next→next;
      break;
    }
    n = n→next;
  }
}
```

This code is used in section 58.

**62.** Transients are processed during ⟨ Vocal Tract Computation 49 ⟩. The transient list is iterated through, their contributions are made to the Left and Right delay lines.

In this implementation, the transients in the list are iterated through, and their contributions are calculated using the following exponential function:

$$A = s2^{-E_0 * t}$$

Where:

- $A$ is the contributing amplitude to the left and right-going components.
- $s$ is the overall strength of the transient.
- $E_0$ is the exponent variable constant.
- $t$ is the time alive.

This particular function also must check for any transients that need to be removed, and removes them. Some caution must be made to make sure that this is done properly. Because a call to *remove_transient* changes the size of the pool, a copy of the current size is copied to a variable for the for loop. Since the list iterates in order, it is presumably safe to remove values from the list while the list is iterating.

⟨ Process Transients 62 ⟩ ≡

```
pool = &tr→tpool;
current_size = pool→size;
n = pool→root;
for (i = 0; i < current_size; i++) {
  noise = (SPFLOAT)sp_rand(sp)/SP_RANDMAX;
  amp = n→strength * pow(2, −1.0 * n→exponent * n→time_alive);
  tr→L[n→position] += amp * 0.5 * noise;
  tr→R[n→position] += amp * 0.5 * noise;
  n→time_alive += tr→T * 0.5;
  if (n→time_alive > n→lifetime) {
    remove_transient(pool, n→id);
  }
  n = n→next;
}
```

This code is cited in section 58.

This code is used in section 49.

# Header File (63)

CTANGLE will end up generating two files: a single C amalgamation and this header file.

This header file exists for individuals who wish to use Voc in their own programs. Voc follows Soundpipe's hardware-agnostic design, and should be trivial to throw in any DSP inner loop.

The contents of the header is fairly minimal. Following a standard header guard, the contents consist of:

- a **typedef** around the opaque struct **sp_voc**
- function declarations which adhere to the 4-stage Soundpipe module lifecycle model.
- a collection of setter/getter functions to allow to get and set data from the opaque struct.

Since *Voc* makes use of opaque struct pointers, this header file will need to declare setter/getter functions for any user parameters.

⟨ voc.h 63 ⟩ ≡

```
#ifndef SP_VOC
#define SP_VOC
  typedef struct sp_voc sp_voc;
  int sp_voc_create(sp_voc **voc);
  int sp_voc_destroy(sp_voc **voc);
  int sp_voc_init(sp_data * sp, sp_voc *voc);
  int sp_voc_compute(sp_data * sp, sp_voc *voc, SPFLOAT * out);
  int sp_voc_tract_compute(sp_data * sp, sp_voc *voc, SPFLOAT * in, SPFLOAT * out);
  void sp_voc_set_frequency(sp_voc *voc, SPFLOAT freq);
  SPFLOAT * sp_voc_get_frequency_ptr(sp_voc *voc);
  SPFLOAT * sp_voc_get_tract_diameters(sp_voc *voc);
  SPFLOAT * sp_voc_get_current_tract_diameters(sp_voc *voc);
  int sp_voc_get_tract_size(sp_voc *voc);
  SPFLOAT * sp_voc_get_nose_diameters(sp_voc *voc);
  int sp_voc_get_nose_size(sp_voc *voc);
  void sp_voc_set_tongue_shape(sp_voc *voc, SPFLOAT tongue_index, SPFLOAT tongue_diameter);
  void sp_voc_set_tenseness(sp_voc *voc, SPFLOAT breathiness);
  SPFLOAT * sp_voc_get_tenseness_ptr(sp_voc *voc);
  void sp_voc_set_velum(sp_voc *voc, SPFLOAT velum);
  SPFLOAT * sp_voc_get_velum_ptr(sp_voc *voc);
  void sp_voc_set_diameters(sp_voc *voc, int blade_start, int lip_start, int tip_start,
      SPFLOAT tongue_index, SPFLOAT tongue_diameter, SPFLOAT * diameters);
  int sp_voc_get_counter(sp_voc *voc);
#endif
```

This code is cited in sections 1 and 3.

# Small Applications and Examples (64)

It has been fruitful investment to write small applications to assist in the debugging process. Such programs can be used to generate plots or visuals, or to act as a simple program to be used with GDB. In addition to debugging, these programs are also used to quickly try out concepts or ideas.

**65.**

## A Program for Non-Realtime Processing and Debugging

The example program below is a C program designed out of necessity to debug and test Voc. It a program with a simple commandline interface, where the user gives a "mode" along with set of optional arguments.

The following modes are as follows:

- **audio:** writes an audio file called "test.wav". You must supply a duration (in samples).
- **plot:** Uses sp_process_plot to generate a matlab/octave compatible program that plots the audio output.
- **tongue:** Will be a test program that experiments with parameters manipulating tongue position. It takes in tongue index and diameter parameters, to allow for experimentation without needing to recompile.

The functions needed to call Voc from C in this way are found in the section ⟨ Top Level Functions 11 ⟩.

⟨ debug.c 65 ⟩ ≡

```
#include <soundpipe.h>
#include <string.h>
#include <stdlib.h>
#include "voc.h"
  static void process(sp_data * sp, void *ud)
  {
    SPFLOAT out;
    sp_voc *voc = ud;
    sp_voc_compute(sp, voc, &out);
    sp_out(sp, 0, out);
  }
  static void run_voc(long len, int type)
  {
    sp_voc *voc;
    sp_data * sp;
    sp_create(&sp);
    sp->len = len;
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    if (type == 0) {
      sp_process_plot(sp, voc, process);
    }
    else {
      sp_process(sp, voc, process);
    }
    sp_voc_destroy(&voc);
    sp_destroy(&sp);
  }
  static void run_tongue(SPFLOAT tongue_index, SPFLOAT tongue_diameter)
  {
    sp_voc *voc;
    sp_data * sp;
    sp_create(&sp);
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    fprintf(stderr, "Tongue index: %g. Tongue diameter: %g\n", tongue_index, tongue_diameter);
    sp_voc_set_tongue_shape(voc, tongue_index, tongue_diameter);
    sp_process(sp, voc, process);
    sp_voc_destroy(&voc);
```

```
    sp_destroy(&sp);
  }
  int main(int argc, char *argv[])
  {
    if (argc ≡ 1) {
      fprintf(stderr, "Pick a mode!\n");
      exit(0);
    }
    if (¬strcmp(argv[1], "plot")) {
      if (argc < 3) {
        fprintf(stderr, "Usage: %s plot duration (samples)\n", argv[0]);
        exit(0);
      }
      run_voc(atoi(argv[2]), 0);
    }
    else if (¬strcmp(argv[1], "audio")) {
      if (argc < 3) {
        fprintf(stderr, "Usage: %s audio duration (samples)\n", argv[0]);
        exit(0);
      }
      run_voc(atoi(argv[2]), 1);
    }
    else if (¬strcmp(argv[1], "tongue")) {
      if (argc < 4) {
        fprintf(stderr, "Usage %s tongue tongue_index tongue_diameter\n", argv[0]);
        exit(0);
      }
      run_tongue(atof(argv[2]), atof(argv[3]));
    }
    else {
      fprintf(stderr, "Error: invalid type %s\n", argv[1]);
    }
    return 0;
  }
```

This code is cited in section 1.

**66.**

## A Utility for Plotting Data

The following program below is used to write data files to be read by GNUplot. The primary use of this program is for generating use plots in this document, such as those seen in the section ⟨ The Vocal Tract 43 ⟩.

⟨ plot.c 66 ⟩ ≡

```
#include <soundpipe.h>
#include <string.h>
#include <stdlib.h>
#include "voc.h"
  static void plot_tract()
  {
    sp_voc *voc;
    sp_data * sp;
    SPFLOAT
        *tract;
        int size;
        int i;
    sp_create(&sp);
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    tract = sp_voc_get_tract_diameters(voc);
    size = sp_voc_get_tract_size(voc);
    for (i = 0; i < size; i++) {
      printf("%i\t%g\n", i, tract[i]);
    }
    sp_voc_destroy(&voc);
    sp_destroy(&sp);
  }
  static void plot_nose()
  {
    sp_voc *voc;
    sp_data * sp;
    SPFLOAT * nose;
    int size;
    int i;
    sp_create(&sp);
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    nose = sp_voc_get_nose_diameters(voc);
    size = sp_voc_get_nose_size(voc);
    for (i = 0; i < size; i++) {
      printf("%i\t%g\n", i, nose[i]);
    }
    sp_voc_destroy(&voc);
    sp_destroy(&sp);
  }
  static void plot_tongue_shape(int num)
  {
    sp_voc *voc;
```

```
  sp_data * sp;
  SPFLOAT
      *tract;
      int size;
      int i;
  sp_create(&sp);
  sp_voc_create(&voc);
  sp_voc_init(sp, voc);
  tract = sp_voc_get_tract_diameters(voc);
  size = sp_voc_get_tract_size(voc);
  switch (num) {
  case 1: sp_voc_set_tongue_shape(voc, 20.5, 3.5);
    break;
  case 2: sp_voc_set_tongue_shape(voc, 25.5, 3.5);
    break;
  case 3: sp_voc_set_tongue_shape(voc, 20.5, 2.0);
    break;
  case 4: sp_voc_set_tongue_shape(voc, 24.8, 1.4);
    break;
  }
  for (i = 0; i < size; i++) {
    printf("%i\t%g\n", i, tract[i]);
  }
  sp_voc_destroy(&voc);
  sp_destroy(&sp);
}
int main(int argc, char **argv)
{
  if (argc < 2) {
    fprintf(stderr, "Usage: %s plots/name.dat\n", argv[0]);
    exit(1);
  }
  if (¬strncmp(argv[1], "plots/tract.dat", 100)) {
    plot_tract();
  }
  else if (¬strncmp(argv[1], "plots/nose.dat", 100)) {
    plot_nose();
  }
  else if (¬strncmp(argv[1], "plots/tongueshape1.dat", 100)) {
    plot_tongue_shape(1);
  }
  else if (¬strncmp(argv[1], "plots/tongueshape2.dat", 100)) {
    plot_tongue_shape(2);
  }
  else if (¬strncmp(argv[1], "plots/tongueshape3.dat", 100)) {
    plot_tongue_shape(3);
  }
  else if (¬strncmp(argv[1], "plots/tongueshape4.dat", 100)) {
    plot_tongue_shape(4);
  }
  else {
```

```
        fprintf(stderr, "Plot: could not find plot %s\n", argv[1]);
        exit(1);
      }
      return 0;
    }
```

This code is cited in section 1.

# External Sporth Plugins (67)

Sporth, a stack-based synthesis language, is the preferred tool of choice for sound design experimentation and prototyping with Voc. A version of Voc has been ported to Sporth as third party plugin, known as an *external Sporth Plugin*.

## Sporth Plugins as Seen from Sporth

In Sporth, one has the ability to dynamically load custom unit-generators or, *ugens*, into Sporth. Such a unit generator can be seen here in Sporth code:

```
_voc "./voc.so" fl
# frequency
170 8 1 5 jitter +
# tongue position
0 1 5 randi
# tongue diameter
0 1 20 randi
# breathiness
0.4 0.7 9 randi
# velum amount
0
_voc fe

# Add reverberation using the Zita reverberator
dup dup 1 2 8000 zrev drop -3 ampdb * +

# close the plugin
_voc fc
```

In the code above, the plugin file is loaded via `fl` (function load) and saved into the table `_voc`. An instance of `_voc` is created with `fe` (function execute). Finally, the dynamic plugin is closed with `fc` (function close).

## Sporth plugins as seen from C.

Custom unit generators are written in C using a special interface provided by the Sporth API. The functionality of an external sporth ugen is nearly identical to an internal one, with exceptions being the function definition and how custom user-data is handled. Besides that, they can be seen as equivalent.

The entirety of the Sporth unit generator is contained within a single subroutine, declared **static** so as to not clutter the global namespace. The crux of the function is a case switch outlining four unique states of operation, which define the *lifecycle* of a Sporth ugen. This design concept comes from Soundpipe, the music DSP library that Sporth is built on top of.

These states are executed in this order:

1. Create: allocates memory for the DSP module
2. Initialize: zeros out and sets up default values
3. Compute: Computes an audio-rate sample (or samples)
4. Destroy: frees all memory previously allocated in Create

Create and init are called once during runtime, compute is called as many times as needed while the program is running, and destroy is called once when the program is stopped.

The code below shows the outline for the main Sporth Ugen.

⟨ ugen.c 67 ⟩ ≡

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

```
#ifdef BUILD_SPORTH_PLUGIN
#include <soundpipe.h>
#include <sporth.h>
#include "voc.h"
#else
#include "plumber.h"
#endif
#ifdef BUILD_SPORTH_PLUGIN
  static int sporth_voc(plumber_data * pd, sporth_stack * stack, void **ud)
#else
       int sporth_voc(sporth_stack * stack, void *ud)
#endif
       {
         sp_voc *voc;
         SPFLOAT out;
         SPFLOAT freq;
         SPFLOAT pos;
         SPFLOAT diameter;
         SPFLOAT tenseness;
         SPFLOAT nasal;
#ifndef BUILD_SPORTH_PLUGIN
         plumber_data * pd;
         pd = ud;
#endif
         switch (pd->mode) {
         case PLUMBER_CREATE:
           ⟨ Creation 68 ⟩;
           break;
         case PLUMBER_INIT:
           ⟨ Initialization 69 ⟩;
           break;
         case PLUMBER_COMPUTE:
           ⟨ Computation 70 ⟩;
           break;
         case PLUMBER_DESTROY:
           ⟨ Destruction 71 ⟩;
           break;
         }
         return PLUMBER_OK;
       }
       ⟨ Return Function 72 ⟩
```

See also sections 73 and 74.

This code is cited in sections 1 and 11.

**68.** The first state executed is **creation**, denoted by the macro `PLUMBER_CREATE`. This is the state where memory is allocated, tables are created and stack arguments are checked for validity.

It is here that the top-level function ⟨ Voc Create 12 ⟩ is called.

```
⟨ Creation 68 ⟩ ≡
  sp_voc_create(&voc);
#ifdef BUILD_SPORTH_PLUGIN
  *ud = voc;
#else
  plumber_add_ugen(pd, SPORTH_VOC, voc);
#endif
  if (sporth_check_args(stack, "fffff") ≠ SPORTH_OK) {
    plumber_print(pd, "Voc: not enough arguments!\n");
  }
  nasal = sporth_stack_pop_float(stack);
  tenseness = sporth_stack_pop_float(stack);
  diameter = sporth_stack_pop_float(stack);
  pos = sporth_stack_pop_float(stack);
  freq = sporth_stack_pop_float(stack);
  sporth_stack_push_float(stack, 0.0);
```

This code is used in section 67.

**69.** The second state executed is **initialization**, denoted by the macro `PLUMBER_INIT`. This is the state where variables get initalised or zeroed out. It should be noted that auxiliary memory can allocated here for things involving delay lines with user-specified sizes. For this reason, it is typically not safe to call this twice for reinitialization. (The author admits that this is not an ideal design choice.)

It is here that the top-level function ⟨ Voc Initialization 14 ⟩ is called.

```
⟨ Initialization 69 ⟩ ≡
#ifdef BUILD_SPORTH_PLUGIN
  voc = *ud;
#else
  voc = pd->last->ud;
#endif
  sp_voc_init(pd->sp, voc);
  nasal = sporth_stack_pop_float(stack);
  tenseness = sporth_stack_pop_float(stack);
  diameter = sporth_stack_pop_float(stack);
  pos = sporth_stack_pop_float(stack);
  freq = sporth_stack_pop_float(stack);
  sporth_stack_push_float(stack, 0.0);
```

This code is used in section 67.

**70.** The third state executed is **computation**, denoted by the macro `PLUMBER_COMPUTE`. This state happens during Sporth runtime in the audio loop. Generally speaking, this is where a Ugen will process audio. In this state, strings in this callback are ignored; only floating point values are pushed and popped.

It is here that the top-level function ⟨ Voc Compute 15 ⟩ is called.

```
⟨ Computation 70 ⟩ ≡
#ifdef BUILD_SPORTH_PLUGIN
  voc = *ud;
#else
  voc = pd->last->ud;
#endif
  nasal = sporth_stack_pop_float(stack);
  tenseness = sporth_stack_pop_float(stack);
  diameter = sporth_stack_pop_float(stack);
  pos = sporth_stack_pop_float(stack);
  freq = sporth_stack_pop_float(stack);
  sp_voc_set_frequency(voc, freq);
  sp_voc_set_tenseness(voc, tenseness);
  if (sp_voc_get_counter(voc) == 0) {
    sp_voc_set_velum(voc, 0.01 + 0.8 * nasal);
    sp_voc_set_tongue_shape(voc, 12 + 16.0 * pos, diameter * 3.5);
  }
  sp_voc_compute(pd->sp, voc, &out);
  sporth_stack_push_float(stack, out);
```

This code is used in section 67.

**71.** The fourth and final state in a Sporth ugen is **Destruction**, denoted by `PLUMBER_DESTROY`. Any memory allocated in `PLUMBER_CREATE` should be consequently freed here.

It is here that the top-level function ⟨ Voc Destroy 13 ⟩ is called.

```
⟨ Destruction 71 ⟩ ≡
#ifdef BUILD_SPORTH_PLUGIN
  voc = *ud;
#else
  voc = pd->last->ud;
#endif
  sp_voc_destroy(&voc);
```

This code is used in section 67.

**72.** A dynamically loaded sporth unit-generated such as the one defined here needs to have a globally accessible function called *sporth_return_ugen*. All this function needs to do is return the ugen function, which is of type *plumber_dyn_func*.

```
⟨ Return Function 72 ⟩ ≡
#ifdef BUILD_SPORTH_PLUGIN
  plumber_dyn_func sporth_return_ugen()
  {
    return sporth_voc;
  }
#endif
```

This code is used in section 67.

**73.**

## A Ugen for the Vocal Tract Model

```
⟨ ugen.c 67 ⟩ +≡
#ifdef BUILD_SPORTH_PLUGIN
  static int sporth_tract(plumber_data * pd, sporth_stack * stack, void **ud)
  {
    sp_voc *voc;
    SPFLOAT out;
    SPFLOAT pos;
    SPFLOAT diameter;
    SPFLOAT nasal;
    SPFLOAT in;
    switch (pd->mode) {
    case PLUMBER_CREATE:
      sp_voc_create(&voc);
      *ud = voc;
      if (sporth_check_args(stack, "ffff") ≠ SPORTH_OK) {
        plumber_print(pd, "Voc: not enough arguments!\n");
      }
      nasal = sporth_stack_pop_float(stack);
      diameter = sporth_stack_pop_float(stack);
      pos = sporth_stack_pop_float(stack);
      in = sporth_stack_pop_float(stack);
      sporth_stack_push_float(stack, 0.0);
      break;
    case PLUMBER_INIT:
      voc = *ud;
      sp_voc_init(pd->sp, voc);
      nasal = sporth_stack_pop_float(stack);
      diameter = sporth_stack_pop_float(stack);
      pos = sporth_stack_pop_float(stack);
      in = sporth_stack_pop_float(stack);
      sporth_stack_push_float(stack, 0.0);
      break;
    case PLUMBER_COMPUTE:
      voc = *ud;
      nasal = sporth_stack_pop_float(stack);
      diameter = sporth_stack_pop_float(stack);
      pos = sporth_stack_pop_float(stack);
      in = sporth_stack_pop_float(stack);
      if (sp_voc_get_counter(voc) ≡ 0) {
        sp_voc_set_velum(voc, 0.01 + 0.8 * nasal);
        sp_voc_set_tongue_shape(voc, 12 + 16.0 * pos, diameter * 3.5);
      }
      sp_voc_tract_compute(pd->sp, voc, &in, &out);
      sporth_stack_push_float(stack, out);
      break;
    case PLUMBER_DESTROY:
      voc = *ud;
      sp_voc_destroy(&voc);
      break;
```

```
    }
    return PLUMBER_OK;
  }
#endif
```

**74.**

## A multi ugen plugin implementation

New Sporth developments contemporary with the creation of Voc have lead to the development of Sporth plugins with multiple ugens.

```
⟨ ugen.c 67 ⟩ +≡
#ifdef BUILD_SPORTH_PLUGIN
  static const plumber_dyn_func sporth_functions[] = {sporth_voc, sporth_tract, };
  int sporth_return_ugen_multi(int n, plumber_dyn_func * f)
  {
    if (n < 0 ∨ n > 1) {
      return PLUMBER_NOTOK;
    }
    *f = sporth_functions[n];
    return PLUMBER_OK;
  }
#endif
```

# Sporth Code Examples (75)

Here are some sporth code examples.

## Chant

*_voc "./voc.so"* **fl**

36 0.3 1 4 **jitter** + **mtof**

0.1 1 **sine** 0 1 **biscale**
0.9
0.9
0.3 1 **sine** 0 1 **biscale**
*_voc* **fe** 36 **mtof** 70 5 **eqfil**

**dup dup** 0.97 10000 **revsc drop** −14 **ampdb** ∗ **dcblk** +

*_voc* **fc**

## Rant

```
# It kind of sounds like an angry rant
```
*_voc "./voc.so"* **fl**
100
8 **metro** 0.3 **maygate** 200 ∗ + 0.1 **port**
30 1 10 **jitter** +

0 1 3 **randi**
0 1 3 20 1 **randi randi**
0.7
0
*_voc* **fe**

1 **metro** 0.7 **maygate** 0.03 **port** ∗

**dup dup** 1 2 8000 **zrev drop** −10 **ampdb** ∗ +

*_voc* **fc**

## Unya

*_voc "./voc.so"* **fl**
*_rate* **var**
*_seq "0 2 4 7 9 11 12"* **gen_vals**

15 **inv** 1 **sine** 0.3 3 **biscale** *_rate* **set**

*_rate* **get metro** 1 *_seq* **tseq** 48 + 5 6 1 **randi** 1 **sine** 0.3 ∗ + **mtof**

*_rate* **get metro** 0.1 0.01 0.1 **tenv** 0.0 0.3 **scale**
*_rate* **get metro** 0.1 0.1 0.3 **tenv** 0.0 *_rate* **get metro** 0.3 0.9 **trand scale**
0.8
*_rate* **get metro tog**
*_voc* **fe**

**dup dup** 0.9 8000 **revsc drop** −14 **ampdb** ∗ **dcblk** +

*_voc* **fc**

# Soundpipe Files (76)

This section here outlines files specifically needed to fulfill the Soundpipe the requirements for being a Soundpipe module.

The components of a fully implemented Soundpipe module consist of the following:

- The core callback code implementing create, destroy, initialize, and compute functions (the core of this document)
- An accompanying header file for the core code (see $ ( *voc.h*)
- An example file, showcase a simple usecase for the module in a small C program, using the namespace convenction ex_FOO.c.
- A metadata file in the form of a Lua table. This file is mainly used to generate documentation for Soundpipe, but it is also used to generate Sporth ugen code.
- A soundpipe test file, using the namespace t_FOO.c. This file gets included with Soundpipe's internal test utility, which implements a form of unit testing for DSP code.
- A soundpipe performance file, using the namespce p_FOO.c. This file get inslucded with Soundpipe's internal performance utiltity, used to gauge how computationally expensive a given Soundpipe module is.

## A small C Example

Each soundpipe module comes with a small example file showcasing how to use a module. This one utilizes the macro tongue control outlined in ⟨ Voc Set Tongue Shape 25 ⟩ to shape the vowel formants. In this case, a single LFO is modulating the tract position.

In addition to providing some example code, these short programs often come in handy with debugging programs like GDB and Valgrind.

```
⟨ ex_voc.c 76 ⟩ ≡
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "soundpipe.h"
  typedef struct {
    sp_voc *voc;
    sp_osc * osc;
    sp_ftbl * ft;
  } UserData;
  void process(sp_data * sp, void *udata)
  {
    UserData *ud = udata;
    SPFLOAT osc = 0, voc = 0;
    sp_osc_compute(sp, ud→osc, Λ, &osc);
    if (sp_voc_get_counter(ud→voc) ≡ 0) {
      osc = 12 + 16 * (0.5 * (osc + 1));
      sp_voc_set_tongue_shape(ud→voc, osc, 2.9);
    }
    sp_voc_compute(sp, ud→voc, &voc);
    sp→out[0] = voc;
  }
  int main( )
  {
    UserData ud;
    sp_data * sp;
    sp_create(&sp);
```

```
    sp_srand(sp, 1234567);
    sp_voc_create(&ud.voc);
    sp_osc_create(&ud.osc);
    sp_ftbl_create(sp, &ud.ft, 2048);
    sp_voc_init(sp, ud.voc);
    sp_gen_sine(sp, ud.ft);
    sp_osc_init(sp, ud.osc, ud.ft, 0);
    ud.osc→amp = 1;
    ud.osc→freq = 0.1;
    sp→len = 44100 * 5;
    sp_process(sp, &ud, process);
    sp_voc_destroy(&ud.voc);
    sp_ftbl_destroy(&ud.ft);
    sp_osc_destroy(&ud.osc);
    sp_destroy(&sp);
    return 0;
}
```

This code is cited in section 1.

**77.**

## Soundpipe Unit Test

The prototypical soundpipe unit test will fill a buffer of memory with samples. The md5 of this buffer is taken, and then compared with a reference md5. If they match, the signal is sample-accurately identical to the reference and the test passes. A test that does not pass can mean any number of things went wrong, and indicates that the module should be seriously looked at it.

```
⟨ t_voc.c 77 ⟩ ≡
#include "soundpipe.h"
#include "md5.h"
#include "tap.h"
#include "test.h"
  typedef struct {
    sp_voc *voc;
    sp_osc * osc;
    sp_ftbl * ft;
  } UserData;
  int t_voc(sp_test * tst, sp_data * sp, const char *hash)
  {
    uint32_t n;
    UserData ud;
    int fail = 0;
    SPFLOAT osc, voc;
    sp_voc_create(&ud.voc);
    sp_osc_create(&ud.osc);
    sp_ftbl_create(sp, &ud.ft, 2048);
    sp_voc_init(sp, ud.voc);
    sp_gen_sine(sp, ud.ft);
    sp_osc_init(sp, ud.osc, ud.ft, 0);
    ud.osc->amp = 1;
    ud.osc->freq = 0.1;
    for (n = 0; n < tst->size; n++) {      /* compute samples and add to test buffer */
      osc = 0;
      voc = 0;
      sp_osc_compute(sp, ud.osc, 0, &osc);
      if (sp_voc_get_counter(ud.voc) == 0) {
        osc = 12 + 16 * (0.5 * (osc + 1));
        sp_voc_set_tongue_shape(ud.voc, osc, 2.9);
      }
      sp_voc_compute(sp, ud.voc, &voc);
      sp_test_add_sample(tst, voc);
    }
    fail = sp_test_verify(tst, hash);
    sp_voc_destroy(&ud.voc);
    sp_ftbl_destroy(&ud.ft);
    sp_osc_destroy(&ud.osc);
    if (fail) return SP_NOT_OK;
    else return SP_OK;
  }
```

This code is cited in section 1.

**78.**

## Soundpipe Perfomance Test

The essence of a performance test in Soundpipe consists of running the compute function enough times so that some significant computation time is taken up. From there it is measured using a OS timing utility like `time`, and saved to a log file. The timing information from this file can be plotted against other soundpipe module times, which can be useful to see how certain modules perform relative to others.

⟨ p_voc.c 78 ⟩ ≡

```
#include <stdlib.h>
#include <stdio.h>
#include "soundpipe.h"
#include "config.h"
  int main()
  {
    sp_data *sp;
    sp_create(&sp);
    sp_srand(sp, 12345);
    sp->sr = SR;
    sp->len = sp->sr * LEN;
    uint32_t t, u;
    SPFLOAT out = 0;
    sp_voc *unit[NUM];
    for (u = 0; u < NUM; u++) {
      sp_voc_create(&unit[u]);
      sp_voc_init(sp, unit[u]);
    }
    for (t = 0; t < sp->len; t++) {
      for (u = 0; u < NUM; u++) sp_voc_compute(sp, unit[u], &out);
    }
    for (u = 0; u < NUM; u++) sp_voc_destroy(&unit[u]);
    sp_destroy(&sp);
    return 0;
  }
```

This code is cited in section 1.

# References (79)