# TabSynth 0.1: Report and Documentation

Cosmin Deaconu

March 19, 2009

**Abstract**

TabSynth provides a software framework for generating sound based on the position of a cursor within a window. The extra input options afforded by graphics tablets are supported (and recommended!). TabSynth is written in C++ using gtkmm (for the user interface) and the Synthesis Toolkit (STK) (for audio generation). TabSynth's project web site is `http://ccrma.stanford.edu/~cozzyd/tabsynth/`. This document describes version 0.1 of TabSynth.

# Contents

# 1 Background and Motivation

As a Christmas present this year, I received a Wacom Tablet of the Bamboo variety (these are the cheapest ones; they do not support detecting tilt.) After a bit of fooling around, I managed to get it working with Linux, and all was good (thanks to the people at the linuxwacom project).

When I learned that I could pursue an independent project for an extra unit for Music 420, I immediately thought of attempting to do something with my tablet. In particular, I wanted to make a theremin that I could play with my tablet. I soon realized that it would be useful to have a general framework for this sort of thing.

A little bit of research showed that I was not the first to come up with this idea. In particular, I was able to find an article [1] which lists some existing programs for tablet input. While it appears there exists an HID object for PD on Linux, there is not (to my knowledge) an easy to use theremin-like application that uses tablet input. [1]

As we were learning to use the STK in Music 420, I decided to use that for audio generation. Since the STK is written in C++, that would be the language to use. After a bit of research, I learned that GTK+ has support for tablets (which makes sense, since it was developed as a toolkit for GIMP). Since I was using C++, I looked into using gtkmm (C++ GTK+ bindings) for the user interface for my program. I had never developed a GTK+ application before, so this was a bit of a learning experience.

I also decided that I wanted to have a module-oriented synthesis system, where synthesis plugins are dynamically loaded. This was also new to me (which explains some of the poor design decisions).

Finishing the program became increasingly difficult, as every time I wanted to "test" it, I would spend a lot of time playing with it instead. I originally had grand plans to write a lot of output plugins, but in the end I ended up with just two real plugins and one pedagogical plugin. Fortunately, writing an output plugin is "easy," so additional plugins may be provided in the future.

I hope others find TabSynth as enjoyable as I do.

# 2 User Documentation

This section provides instructions for setting up and using TabSynth.

## 2.1 Installation Procedure

(Note: A more updated version of this will be maintained at `http://ccrma.stanford.edu/~cozzyd/tabsynth/#install`)

[1] http://createdigitalmusic.com/2006/06/15/use-graphics-tablets-for-music-new-and-updated-software-free-tablet-theremin/

### 2.1.1 Obtaining TabSynth

Instructions for obtaining the most current version of TabSynth are available at `http://ccrma.stanford.edu/~cozzyd/tabsynth/#dl`.

### 2.1.2 Basics

TabSynth must currently be compiled from source. This requires a C++ compiler (g++) as well as the development files for gtkmm[2] and STK [3] TabSynth is known to compile on Ubuntu 8.10 and Fedora 10.

### 2.1.3 Installing gtkmm

**gtkmm-dev** should be provided in the repository of most distros. In particular, Ubuntu 8.10 and Fedora 10 both have it easily available. The Makefile looks for gtkmm-2.4, though it's possible an older version may also work.

### 2.1.4 Installing the STK

**STK** is not uncommonly included in distro's repositories, however the version shipped with Ubuntu 8.10 is too old to work with TabSynth's provided plugins. (We need at least 4.2.1 and 8.10 provides 4.2.0). Fedora's version is recent enough.

Ubuntu users must manually install the STK libraries. Some instructions for doing that can be found at the STK web page [4] (look for the part called "Library Use"). I have outlined the important steps below:

#### Installing STK from Source

- It's probably best to get rid of any existing, older version of STK using your package manager.
- Download the newest sources from the project home page [5].
- Untar and cd into the directory just created
- `./configure`
  The default options should probably suit you, but if you want JACK output or something like that, feel free to peruse the STK documentation to look up the correct flags.
- `make`
  **This step will probably fail**. I think this is because STK was made for gcc3 but most new distributions bundle gcc4. In order to get it to compile, either use gcc3 or modify the source as follows (thanks to Nelson Lee for figuring this out):

[2] http://www.gtkmm.org
[3] http://ccrma.stanford.edu/software/stk/
[4] http://ccrma.stanford.edu/software/stk/compile.html
[5] http://ccrma.stanford.edu/software/stk/download.html

  - In stk/include/Stk.h:
    1. add `#include <stdlib.h>`
    2. change `#include <string>` to `<string.h>`
  - In stk/src/Messanger.cpp, add `#include <algorithm>`
  - In stk/src/RtAudio.cpp, add `#include <limits.h>`

- Copy src/libstk.a somewhere in your library path (/usr/local/lib is probably a good candidate). It may be necessary to run ldconfig before your system will recognize the library.
- Copy everything in include/ to a folder called stk/ somewhere in your include path (e.g. /usr/local/include/stk/).
- That should be everything. If the step below still complains about stk not being found, check your library and include environmental variables to make sure they're looking in the right place.

#### 2.1.5 Compiling TabSynth

To install, get the newest sources and run "make" in the root tabsynth directory. If you're lucky, it will compile and produce a tabsynth executable :).

### 2.2 Running TabSynth

To run TabSynth, enter the TabSynth root directory and execute `./tabsynth`. Currently, calling TabSynth from a different directory won't work due to hard-coded paths (hopefully this will be fixed in the future).

Upon opening TabSynth, two windows will spawn. The functions of these windows are detailed in the next sections.

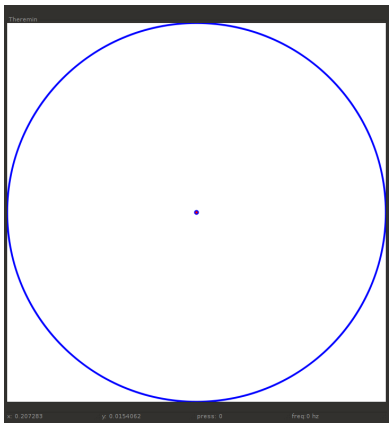
### 2.3 The Input Window

The Input Window (Fig. 1) is the heart of TabSynth. Sound is produced by moving the cursor around in the playing area (the white part surrounded by the frame). Individual output plugins are responsible for figuring out what sound to make based on the parameters in the region. Normally sound will only be produced if there is non-zero tablet pressure (or, if you don't have a tablet, if you are holding down the left mouse button).

A frame shows the name of the currently loaded plugin and the status area at the bottom shows the current position (in relative coordinates), pressure and frequency to be outputted by the plugin.

The window may be resized past its minimum sized freely, although the playing area will always maintain a square aspect ratio.

Figure 1: *The Input Window with the Theremin plugin loaded. Note that the actual appearance of the widgets will vary according to your GTK+ theme. The appearance of the canvas area can be modified by a plugin.*



## 2.4 The Control Window

The Control Window (Fig. 2) allows you to select the output plugin as well as set any other options.

The Input Devices button loads the Input Devices Dialog (see below), which allows you to configure extended input devices (e.g. tablets).

The volume slider controls the volume of TabSynth, regardless of plugin.

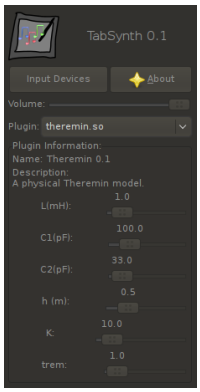
The plugin selector drop down box allows you to select the currently loaded plugin. This list is populated at start up by the current contents of the plugins/ directory.

Below that is the "Plugin Information" frame, which contains the name and description of the currently loaded plugin as well as any settable parameters the plugin may specify.

## 2.5 The Input Devices Dialog

The Input Devices dialog is the standard GTK+ way of configuring extended input devices, such as tablets. If you have a properly configured tablet (for example, using the linuxwacom projects drivers, NOT the hotplugging HID drivers which will not allow pressure detection), you will want to select the stylus device (on my system, it's called "stylus," but the actual name will vary according to X.org configuration) and select "Screen" mode. This maps the surface of the tablet to the entire display. "Window" is supposed to map the device to a specific window, though in practice, its behavior is rather ill-defined.

Figure 2: *The Control Window with the Theremin plugin loaded. The options available in the "Plugin Information" frame will vary from plugin to plugin.*




# 3 Included Plugins

Each currently included plugin is documented here.

## 3.1 Theremin

The theremin plugin is the raison d'etre of TabSynth. As its name may suggest, it is supposed to model theremin.

The theremin plugin draws an "antenna" in the middle and an inscribed blue circle on the playing area. The closer one is to the antenna, the higher the frequency. The outer circle represents a frequency of 0 hz. The instrument's volume is controlled by the pressure of the stylus (or, if no stylus is present, it will be 100 percent when the left button is pressed and 0 otherwise).

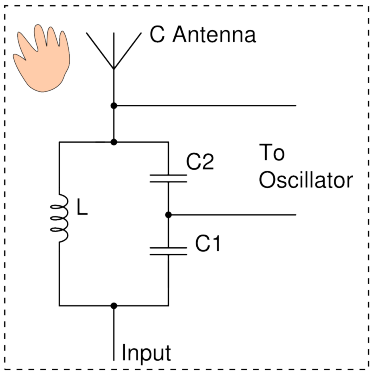
Care was taken to mimic a real analog theremin, so most of the settable parameters are somewhat opaque. The exception is **trem**, which controls the tremolo frequency. Tremolo may be induced by pressing the right mouse button.

### 3.1.1 Physical Model

The basic idea of the theremin is that my moving one's hand around an antenna, it is possible to modify the capacitance of an LC circuit. However, because the capacitance change from this action is so small compared to the components that can be reasonably used in a circuit, a theremin actually employs two oscillators operating at very high frequencies. The capacitance of one of them is slightly modified by the operator, and what one hears is the beat frequency between the two.

I used the basic model outlined in Skeldon[2]. The circuit of the oscillator hooked up to the antenna looks something like what is depicted in Fig. 3. The circuit provides the resonant frequency for a Colpitts oscillator. The following

Figure 3: *The oscillator connected to the antenna in an analog theremin*




equations are used to calculate the pitch of this oscillator, where $L$ is inductance, $C_1$, $C_2$ are specified capacitances, $h$ is height of the antenna, $d$ is diameter, $x$ is distance from hand, $k$ is a factor having to do with the distance from the ground ($\sim$0.4 for slightly above ground) and $K$ is a fudge factor.

$$f = \frac{1}{2\pi\sqrt{L}}\left(\frac{1}{C_1} + \frac{1}{(C_2 + C_A)}\right)^{(1/2)} \tag{1}$$

$$C_A \approx \frac{2\pi\epsilon_0 h}{\log(2h/d) - k} + \frac{\pi\epsilon_0 h}{K\log(4x/d)} \tag{2}$$

You may notice that the first equation is nothing more than the resonance frequency of an LC circuit. The second equation is based on empirical studies of the capacitance of an antenna near the earth's surface. The first factor represents the intrinsic capacitance of such an antenna with no hand near it. The second equation adds the effect of a nearby hand (modelled as an infinite plane). The factor K is introduced as a fudge factor because most people's hands are not of infinite extent (Rachmaninoff notwithstanding).

$C_1$, $C_2$, $h$, $L$ and $K$ are all settable parameters in the Theremin plugin. The other oscillator is tuned such that the frequencies are equal (and thus the total output is 0 hz) at the blue circle in the drawing area. $k$ is set to 0.4 and $d$ is set to 1 cm. The extent of the box is assumed to be $\sim$ 2 m.

Note that if $x$ is less than $d$, $x$ is set to $d$ (however, there are still instabilities near the antenna).

The output plugin outputs a sine wave with a frequency corresponding to the beat frequency between the oscillators.

## 3.2 Flutemin

The Flutemin is an unphysical thereminish instrument. Like the theremin, the frequency goes up as one approaches the antenna. Unlike the theremin, the

frequency is calculated according to an inverse power law. The exponent on the bottom and the maximum frequency are settable. Also, a checkbox for doubling the frequency was added (mostly so all three types of plugin parameter widgets would be displayed).

Unlike the theremin plugin, which uses a simple sine wave, the Flutemin boasts the STK Flute instrument model. However, the flute doesn't very flute-like at low frequencies, and if you get low enough, the instrument cannot produce the corresponding sound. It sounds cool though.

Once again, the volume is a function of the pen pressure (or the left mouse button provides binary volume control).

The plugin draws only an antenna in the middle.

### 3.3 Super Simple

The Super Simple plugin is meant to serve as an example of how to develop an audio output plugin. As such, it is not really meant to be used for playback, though you are welcome to regardless. The output frequency is calculated according to the completely arbitrary formula:

$$f = 1000(x^2 - y^2 + xy - P^2)^2 \tag{3}$$

where $f$ is the output frequency, $x$ and $y$ are the coordinates in the playing area and $P$ is the pressure. The pressure also controls the volume.

The settable parameter corresponds to the number of harmonics in the STK BlitSaw instrument used for output.

It does not render any graphics to the playing area at all.

## 4 Developer Documentation

This section is intended to explain the code layout of TabSynth and to provide instructions for developing plugins.

### 4.1 TabSynth Design

TabSynth was designed with the possibility of having many different output modules in mind. To accomplish this, the output plugins are compiled as shared objects and loaded at runtime by TabSynth.

Here is a brief outline of each what each file does:

- **Makefile**
  The main makefile, although all it does is call src/Makefile
- **plugins/**
  Folder created during build process that will hold the shared object files for the plugins

- **resources/icon.svg**
  The beautiful TabSynth logo. This is called absolutely in several places. This should probably be fixed in the future.

- **src/Makefile**
  Responsible for compiling the main program and for triggering the compilation of the plugins

- **src/Main.cpp**
  The point of entry for TabSynth. Initializes GTK+ and the threading system and spawns an InputWindow and a ControlWindow

- **src/control_window.[h|cpp]**
  Implements the TabSynth control window. This code is responsible for dynamically loading the plugins as well as all configuration.

- **src/input_window.[h|cpp]**
  Implements the TabSynth input window. This code is responsible for capturing input events and sending them to the plugin.
  Also, it calls the plugin's draw function to decorate the input area.

- **src/plugin_option.[h|cpp]**
  A plugin option is a data structure used to describe and store per-plugin user-settable parameters.

- **src/tabsynth_plugin.[h|cpp]**
  All plugins extend from the class defined in this file (TSPlugin) and should implement the virtual members of this class. TSPlugin implements some convenience functionality so that individual plugins don't need to, like a separate audio processing thread that constantly polls tick().

- **src/plugins/Makefile**
  Builds all .cpp files in the plugins directory as plugins and copies them to plugins/

- **src/plugins/theremin.cpp**
  Implements the theremin plugin

- **src/plugins/flutemin.cpp**
  Implements the flutemin plugin

- **src/plugins/super_simple.cpp**
  Implements the "super simple" example plugin that is outlined in the tutorial .

## 4.2 Developing Plugins

Developing a plugin is meant to be a simple process not requiring in-depth knowledge of the rest of TabSynth's code base. To write a plugin, one must

create a .cpp file in src/plugins/ and populate it accordingly. Once compiled, the plugin will appear in the drop down list in the control window. For example, if your file is named foo.cpp, foo.so will appear in the dropdown box.

### 4.2.1 Subclass methods

The majority of the logic should be done in a subclass to `TSPlugin`. The correct include file for the class is `tabsynth_plugin.h`. The following virtual methods need to be implemented in the subclass:

- ```
  setParameters(double x, double y, double pressure,
                bool button2, bool button3,
                double tiltX, double tiltY);
  ```

  This function is called by the input window whenever it processes a GTK+ event. It should be used to modify the parameters of the output. Generally, it is advisable to use the pressure as the volume since otherwise sound will be emitted all the time.

  The two coordinates are in coordinates such that the size of the playing area is 1.0. If no pressure is detected, 1 is passed if the left mouse button is held down and 0 otherwise. Note that button2 refers to the middle button and button3 to the right button. The tilt has not been tested yet as I lack a fancy enough tablet.

- ```
  StkFloat tick();
  ```

  This function is called by the audio processing thread whenever it needs a new sample (44100 times a second, in fact).

- ```
  void draw(Cairo::RefPtr<Cairo::Context> cairo_context,
            int size);
  ```

  This function is called by the input window whenever it needs to redraw the playing area. It passes along a Cairo context belonging to the playing area and an integer specifying the length of a side of the playing area. The idea is that the plugin should draw something here if it wants to).

- ```
  std::string getName();
  std::string getDescription();
  std::string getVersion();
  ```

  The (self-explanatory) return values of these functions are used to populate the plugin inspector in the control window.

### 4.2.2 Adding settable parameters

In addition to the above methods, the plugin can optionally make use of the the protected `std::vector<PluginOption*> options` member. Any added `PluginOption`s will appear as settable parameters in the control window.

To create a `PluginOption`, use the following constructor:

```
PluginOption(pluginOptionType type, std::string name,
                std::string description, double def = 0,
                double min = 0, double max = 1,
                int numSteps = 10);
```

Here, `pluginOptionType` is an enum that can be one of `OPTION_SLIDER` (a slider control), `OPTION_SPIN` (a spin entry box) or `OPTION_CHECK` (a check box). The label that appears next to the control is specified by `name` and its tooltip is dictated by `description`. The default, minimum, maximum and number of steps (when using arrow keys to access the field) can also be set (though for a check box, only 0 and 1 make sense as values).

The value of a `PluginOption` may then be obtained using the `getValue()` method.

### 4.2.3 Making the plugin loadable

Two functions are necessary to make the plugin loadable dynamically (via the dlfcn module). As dlfcn was meant for C, the compiler needs special instructions for these functions. The two necessary functions are:

```
extern "C" TSPlugin * init() {
  return new Foo;
}

extern "C" void destroy(TSPlugin * p) {
  delete p;
}
```

Obviously, you'd need to replace `Foo` with the appropriate name for your subclass of `TSPlugin`. The function `init()` is called by the plugin loader to create the plugin and `destroy(TSPlugin * p)` is called to deallocate its resources.

### 4.2.4 Getting it compiled

Adding the .cpp file to the src/plugins/ directory and running make (in the root directory) should be sufficient to compile and add the plugin to the plugins/ folder.

## 4.3 Tutorial: The Super Simple Plugin

An annotated version of a simple plugin (called super_simple.cpp) follows..

```
#include "../tabsynth_plugin.h" //This file must be included
#include <stk/BlitSaw.h>       //Use this for synthesis
#include <cmath>
```

The important thing here is the include for `../tabsynth_plugin.h`. This will allow us to subclass `TSPlugin`.

```
class SuperSimple : public TSPlugin {

 protected: //Protected Member Variables
  double vol;
  BlitSaw * blit;
```

We define a `SuperSimple` class (subclassing `TSPlugin`) and give it a few protected variables. The double `vol` will represent the output volume of this plugin and the STK generator `blit` is going to be used for audio synthesis.

```
 public:
  SuperSimple() { //Constructor
    blit = new BlitSaw(220);
    vol = 0;

    //Add an option to set the number of harmonics
    options.push_back(new PluginOption(OPTION_SPIN,
    "NumHarmonics",
    "setHarmonics is called with this number",
    10,1,100,50));
  }
```

In the constructor, we initialize the volume to 0 and we initialize our generator. As we will change the frequency later, the initial frequency doesn't matter too much.

The STK `BlitSaw` also contains a parameter for the number of harmonics it uses. We want that to be user settable, so we add a new `PluginOption *` to the `options` vector (which is a protected member of `TSPlugin`. We want the users to be able to assign any number between 1 and 100 (with a default of 10) and we also want a step size of 2 when using arrow keys in the spin button.

```
  \~SuperSimple() { //Destructor
    delete blit;
  }
```

This is the destructor. It is called when this plugin is unloaded and should free any allocated memory. This (hopefully!) does that.

```
 //Tablet event occurred. Calculate new frequency from this.
 //Return frequency for display purposes
 StkFloat setParameters(double x, double y,
                        double pressure,
                        bool button2, bool button3,
                        double tiltX, double tiltY) {

   //Some weird way of computing frequency
```

```
  double freq = 1000 * pow((x*x - y*y + x*y -
                pressure*pressure),2);

  //Use pressure as volume
  vol = pressure;
  blit->setFrequency(freq);

  //Grab the harmonics from the GUI option
  blit->setHarmonics( (unsigned int) options[0]->getValue());
  return freq;
}
```

This is the heart of the plugin. Upon receiving a new input event, we want to adjust the frequency and volume accordingly. For this toy plugin, we use an arbitrary formula for coming up with frequency. We use the pressure as volume (since that usually makes sense).

Also, we set the number of harmonics to the value currently selected by the user (in case they changed it).

The frequency is returned so that it may be displayed in the status bar.

```
//Computes next sample
StkFloat tick()  {
  return blit->tick() * vol;
}
```

This method is called whenever a new sample is needed (this happens thousands of times a second). We call the `method` on the `BlitSaw` and scale it by the current volume.

```
/* We could draw something here, but Cairo's complicated,
   so we'll skip this*/
void draw(Cairo::RefPtr<Cairo::Context> cr, int size) {}
```

A plugin can draw whatever figure it wants on the playing area of the input window. Describing how to use Cairo is outside the scope of this tutorial, however.

```
  std::string getName() {
   return "Super Simple";
  }
  std::string getDescription() {
   return "A toy plugin for demonstration";
  }
  std::string getVersion() {
   return "-1";
  }
}; //End of Class Definition
```

These methods are used to provide the display text inside the plugin inspector. We have finished the class.

```
/* In addition to containing a class inheriting from TSPlugin,
 * a plugin must have the following two methods. (Used for
 * dynamic loading) */

extern "C" TSPlugin * init() {
  return new SuperSimple;
}

extern "C" void destroy(TSPlugin * p) {
  delete p;
}
```

The above methods are needed in every plugin to allow for dynamic loading (because `dlfcn` is a C library which doesn't understand classes).

As long as the .cpp file is inside src/plugins, it should be compiled automagically when running make in the root directory and the plugin will appear in the dropdown list in the control window.

# 5 Future Plans

Here are some of the known issues in TabSynth.

- No event is triggered when the cursor leaves the drawing area (at least, none that I know about)
- Glitchy audio when volume changes too rapidly. Need to implement gradual volume changes in plugins?
- Memory leaks
- Somewhat non-standard build system (I should probably switch to autotools)
- Crash on exit (double free or something in Glib?), but only on one of my computers
- Badly organized code... too much mix up of GUI and function.
- Resource / plugin paths hard linked. The executable has to be executed from the right place to work.
- While gtkmm and stk are cross-platform, I use POSIX-specific paths and the POSIX dynamic loading functions (dlfcn.h). Also I link to -lasound in the Makefile, which would make compiling on non-Linux require a change to the Makefile.

- Not enough plugins :(

With some luck, some of these will be ameliorated in future releases.

# References

[1] Peter Kirn. Use graphics tablets for music: New and updated software, free tablet theremin, June 2006. http://createdigitalmusic.com/2006/06/15/use-graphics-tablets-for-music-new-and-updated-software-free-tablet-theremin/.

[2] K. D. Skeldon, L. M. Reid, V. McInally, B. Dougan, and C. Fulton. Physics of the Theremin. *American Journal of Physics*, 66:945–955, November 1998.