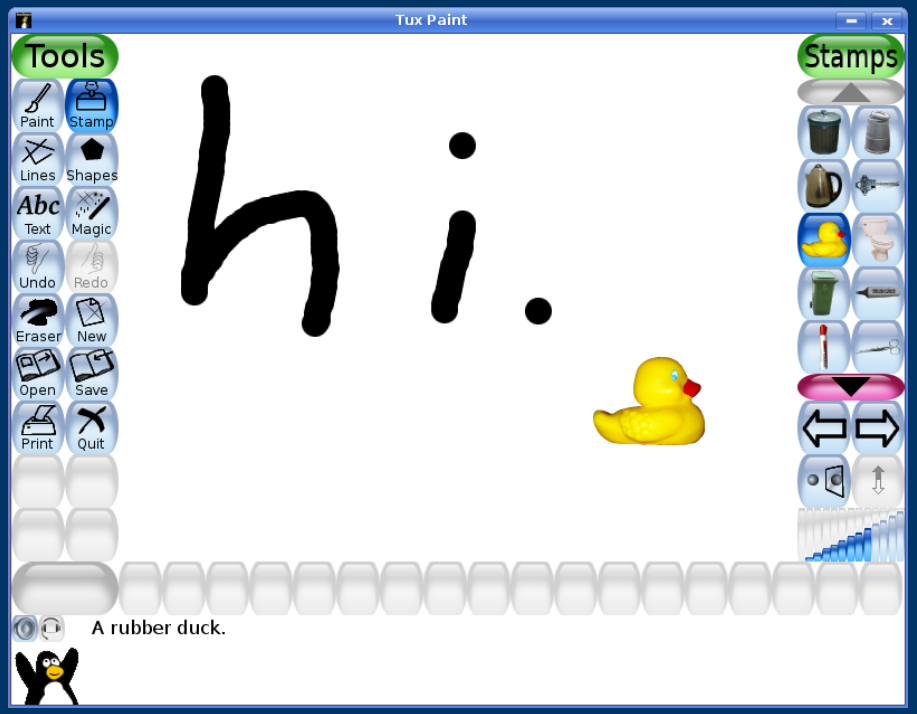
Tux Paint "Magic" Tool Plug-in API

Bill Kendrick
Lead Developer

Peninsula Linux User Group
December 13, 2007

What is Tux Paint?

- Drawing software
- For kids age 3yrs & up
- Open Source
- Linux/Windows/Mac
- Written in C
- Based on libSDL
- 2000+ downloads/day
- Too many lines of code :)



What are “Magic” Tools?

- Special drawing tools
  - Rainbow
  - Bricks
  - Grass
- Effect-applying tools
  - Blur
  - Smudge
  - Tint
- Full-image effects
  - Mirror
  - Flip

Why a Plug-in API?

- Tux Paint is a *lot* of code
- “Magic” tools were entrenched
- Had to rebuild all of Tux Paint to create & test new effects
- Making it easy makes it more fun & accessible!

- *More people will be able to contribute!*

(Idea came to me after a double-shot mocha, while biking to work...
It was just a 'sick joke' at first.
But here we are!)

What you need to know
(or will learn by doing)

- C programming language
- Simple DirectMedia Layer library
- Pointers
- Callbacks
- Event-driven programming
- Graphics programming

Current API was designed around what all of the "Magic" tools in Tux Paint version 0.9.17 needed...

How Plug-ins Get Into Tux Paint
(on Linux, as an example)

- Compile the plugins as a shared object:
  `gcc -shared {...} -o my_plugin.so`
- Put it where Tux Paint expects to find it:
  `/usr/lib/tuxpaint/plugins/`
- Tux Paint scans that directory for .so files.
- Object files are opened, and checked for all the functions Tux Paint expects you to have written:
  `my_plugin_init()`
  `my_plugin_getname()`
  `my_plugin_click()`
  etc.

Sequence of Events

- Tux Paint calls some functions you'll have written:
  - “Are you compatible with this API version?”
  - “Initialize!”
    (load your icons, sounds, any other data)
  - “How many tools do you have?”
    (Plug-ins can include more than one tool; each will appear as a separate “Magic” tool)
  - For each tool the plug-in includes, the plug-in is asked:
    - “What's the tool's name?” (for the button)
    - “Give me your icon” (for the button)
    - “What's the tool's description?” (for Tux penguin to say)
    - “Do you support colors?”
      (e.g., 'Blur' and 'Smudge' do not, but 'Bricks' and 'Grass' do)

All loaded, time to draw!

- Click the “Magic” button under “Tools”
- The bar on the right will show buttons for each tool, including name and icon
- Click a “Magic” tool's icon on the right to activate it
- The tool's description is shown at the bottom, next to Tux the penguin
- The tool is informed of the current color (if applicable)
- *... and now we wait!*

Events to Expect

- Color choice changes
- Clicks
- Drags
- Releases
- *... that's all, folks!*
  (for now)

Dealing with Events

Your functions...

(named after your plug-in, to avoid namespace collisions):

- `foo_set_color()`
- `foo_click()`
- `foo_drag()`
- `foo_release()`

...accept various arguments from TP, including:

- Pointer to a “Magic” tool API structure
- Which of the plug-in's tools is being utilized
- Snapshot of canvas prior to the last click
- Current canvas (where you draw)
- Old and New X & Y coordinates
- An “update rectangle” structure, to tell Tux Paint what part of the canvas was just updated

"Magic" Tool API Structure

- Most of your functions are sent a pointer to the "`tp_magic_api`" structure
- It contains pointers to functions *inside* Tux Paint, as well as some C macros. For example:
  - Get a pixel, put a pixel
  - Scale an entire surface
  - Test whether a coordinate is within a circle *(good for effects that want to have a round brush)*
  - Calculate a line *(this needs explaining)*
  - Draw the progress bar
  - Convert sRBG to linear & back
  - Convert HSV to RGB and back
  - min(), max(), clamp()
  - Play a sound effect
  - *etc.*

Lines & Callbacks
(I promised I'd explain)

- Mice move quickly
- Brush effects need to include everything between two mouse positions, or they'd often be dotted lines
- You can ask Tux Paint to calculate a line between (X_1, Y_1) and (X_2, Y_2)
- Tux Paint will calculate the line, and for each step position between the points, it calls a *callback* function, that you write!
- For brush-like effects, you usually just call Tux Paint's line() inside your drag() function. It, in turn, calls whatever function you wrote that actually *does* your effect.

Compiling & Installing

- Many Open Source projects include “-config” tools to help you compile & install stuff. (SDL, Gimp, etc.)
- Tux Paint “Magic” tool API does, too: `tp-magic-config`
- Use grave/backtick (`) on the shell to get what you need:
  - ```
    gcc --shared `tp-magic-config --cflags` \
    my_plugin.c -o my_plugin.so
    ```
  - ```
    sudo cp my_plugin.so \
    `tp-magic-config --pluginprefix`
    ```
  - ```
    sudo cp my_plugin_icon.png \
    `tp-magic-config --dataprefix`/images/magic
    ```
    ```

Example Code (“ex”) Plug-in

```
// So we recognize TP's Magic API
#include “tp_magic_api.h”

// For loading our PNG icon
#include “SDL_image.h”

// For loading our sound effect
#include “SDL_mixer.h”

// Place to hold sound effect:
Mix_Chunk * snd_effect;

// Place to hold current color
Uint8 ex_r, ex_g, ex_b;
```

Example Code (“ex”) Plug-in

```
// Tell Tux Paint which plug-in API we were
// built against.  (We pick up the value
// as a #define from “tp_magic_api.h”)
Uint32 ex_api_version(void)
{
  return(TP_MAGIC_API_VERSION);
}

// Our initialization routine.
// Just load our sfx from TP's data folder:
int ex_init(magic_api * api)
{
  char fname[1024];
  sprintf(fname, “%s/sounds/magic/ex.wav”,
     api->data_directory);
  snd_effect = Mix_LoadWAV(fname);
  return(snd_effect != NULL); // Success?
}
```

Example Code (“ex”) Plug-in

```
// Tell Tux Paint we have but one tool:
int ex_get_tool_count(magic_api * api)
{
  return(1);
}

// Load our tool's icon and give to Tux Paint:
// Note: We only have one tool, so are assured
// that 'which' will always be '0' (zero)
SDL_Surface * ex_get_icon(magic_api * api,
                          int which)
{
  char fname[1024];
  sprintf(fname, "%s/images/magic/ex.png",
     api->data_directory);
  return(IMG_Load(fname)); // Return the icon!
}
```

Example Code (“ex”) Plug-in

```
// Give Tux Paint our tool's name
char * ex_get_name(magic_api * api, int which)
{
  // Copying it (with 'strdup()'), because
  // Tux Paint will free it when the user quits
  return(strdup(“Example”));
}

// Give Tux Paint our tool's description
char * ex_get_description(magic_api * api,
                          int which)
{
  // Copying with 'strdup()' here, too
  return(strdup(“An example tool!”));
}
```

Example Code (“ex”) Plug-in

```
// Tell Tux Paint that we utilize colors
// (the color palette below the canvas will
// become/remain active when our tool is used)
int ex_requires_colors(magic_api * api,
                       int which)
{
  return(1); // AKA 'true'
}

// Clean up after ourselves when TP quits:
void ex_shutdown(magic_api * api)
{
  // Release RAM used by our sfx:
  Mix_FreeChunk(snd_effect);
}
```

Example Code (“ex”) Plug-in

```
// Respond to clicks (mouse button down event)
void ex_click(magic_api * api, int which,
              SDL_Surface * canvas,
              SDL_Surface * snapshot,
              int x, int y,
              SDL_Rect * update_rect)
{
  // Cheating!!!  For our effect, a click is
  // the same as a drag, so just send “x,y” as
  // the start and end points:

  ex_drag(api, which, canvas, snapshot,
          x, y, x, y, update_rect);
}
```

Example Code (“ex”) Plug-in

```
// Respond to drags (mouse motion events while
// user is clicking)
void ex_drag(magic_api * api, int which,
             SDL_Surface * canvas,
             SDL_Surface * snapshot,
             int ox, int oy, int x, int y,
             SDL_Rect * update_rect)
{
  // Tell Tux Paint to calculate a line between
  // the two points (ox,oy) and (x,y),
  // calling our callback function every step
  api->line((void *) api, which,
            canvas, snapshot,
            ox, oy, x, y, 1, // old, new, step
            ex_line_callback); // our function!

  // there's more...
```

Example Code (“ex”) Plug-in

```
// (after api->line() is called...)

// We'll want to tell Tux Paint what part
// of the canvas changed; let's make sure
// we send top/left and bottom/right:
if (ox > x) { int tmp=ox; ox=x; x=tmp; }
if (oy > y) { int tmp=oy; oy=y; y=tmp; }

// Fill in the (x,y) and (w,h) elements of
// the update rectangle for Tux Paint:
// (Our brush is 9x9, centered around (x,y))
update_rect->x = ox - 4;
update_rect->y = oy - 4;
update_rect->w = (x + 4) - update_rect->x;
update_rect->h = (y + 4) - update_rect->y;

// still a little more...
```

Example Code (“ex”) Plug-in

```
    // (after update_rect is filled in...)

    // Play our sound effect as the user drags
    // (and since we call 'ex_drag()' for clicks,
    // it plays for single clicks, too!)

    api->playsound(snd_effect,
                   (x * 255) / canvas->w,
                   255);

    // What are those values?
    // Pan (0=left, 255=right) and
    // distance (0=far, 255=near)...
    // Tux Paint “Magic” tool sound effects can
    // be in stereo!!!
}
```

Example Code (“ex”) Plug-in

```
// Respond to releases (mouse button up event)
void ex_release(magic_api * api, int which,
                SDL_Surface * canvas,
                SDL_Surface * snapshot,
                int x, int y,
                SDL_Rect * update_rect)
{} // Noone ever said we had to do anything!

// Accept colors (when tool is first selected,
// or when user picks a different color)
void ex_set_color(magic_api * api,
                  Uint8 r, Uint8 g, Uint8 b)
{
  ex_r = r;
  ex_g = g;
  ex_b = b;
}
```

Example Code (“ex”) Plug-in

```
// Finally, our own line callback routine.
// It gets called by Tux Paint via api->line(),
// which we invoked in ex_drag().
void ex_line_callback(void * ptr, int which,
                      SDL_Surface * canvas,
                      SDL_Surface * snapshot,
                      int x, int y)
{
  // Need to cast, since we get a void ptr...
  magic_api * api = (magic_api *) ptr;

  // the work happens on the next slide...
```

Example Code (“ex”) Plug-in

```
// (about to do the line callback work...)

// Quick-and-dirty 9x9 round brush
for (int yy = -4; yy <= 4; yy++)
{
  for (int xx = -4; xx <= 4; xx++)
  {
    if (api->in_circle(xx, yy, 4)) // Round?
    {
      // Ask TP to change the canvas...
      api->putpixel(canvas, x+xx, y+yy,
                    // ...to the user's color:
                    SDL_MapRGB(canvas->format,
                               ex_r, ex_g, ex_b));
    }
  }
}
```

Some Magic Types

- **click()** only:
  - Full-image effects:
    - Mirror, Flip
- **click()** and **drag()**:
  - Drawing and effect brushes:
    - Rainbow, Blur, Negative, etc.
  - Full-image effects needing some input:
    - Shift
- **click()**, **drag()** and **release()**:
  - Multi-step drawing
    - Flower
  - Full-image effects needing input *and* preview:
    - Shift

canvas vs. snapshot

- **`snapshot`** is a recording of the drawing canvas when the user first clicked with the “Magic” tool
- **`canvas`** is the *live* drawing canvas – your changes go here
- Sometimes you want to read from **`snapshot`**
  (click and scribble with “Negative”, and you negate more of the picture, without un-negating as you drag over the same spot)
- Sometimes you want to read from **`canvas`**, allowing the user to apply more of the effect on the same area without releasing and clicking
  (click and scribble with “Smudge”, and the pixels smudge around more and more)

Any more and this'd be overwhelming!

- Trying to document the API in a human-readable way
- My dream is to have Tux Paint “Magic” tool plug-in API taught to first-time comptuer graphics programmers (for example, high schoolers!)
- Help is appreciated!

http://www.tuxpaint.org/
bill@newbreedsoftware.com

Thanks!