0.1 Serialization

The main function is running a initializing routine at startup which ensures that we have a packer, tmp and trash directory located at $HOME[1]/.stopmotion/ ready to be used by the program. “Ready to be used” means that the directories exists and we have read/write permissions to them. If the initializing routine fails to create the directories, it means there is no space left on the hard drive or the user doesn’t have permission to write or execute in its own home directory (which probably never should be the fact, if so: we’re dealing with a very strange user). The program displays an error message to the user and exits if one of the above cases occurs. So, in short: the directories are checked and ready to be used if the initializing routine was run successfully.

The reason for creating the above directories is that adding of images and sounds always goes through the tmp directory before they eventually are saved to a project file. If the user wants to add an image from the hard drive it means that we’re making a copy of it in the tmp directory. Exactly the same routine is run when grabbing an image from the camera. As described in section **??** on page **??**, couple of images are written to capturedFile.[jpg|png|xxx] when the camera is turned on. So, when the user decides to grab an image, we’re making a copy of capturedFile in the tmp directory. By this way of doing it we ensures that the original image never is lost or damaged during processing in the application.

Naturally, it might happen that the user doesn’t liked the newly grabbed image, or a selection of images, and deletes it. Wait a minute. What if the user changes opinion and wants the deleted image(s) back? It’s here the trash directory comes in. The “secret” is that every image which is selected as deleted and removed from the visible widgets, are placed in the trash directory. We can then use an undo operation, explained in section **??** on page ??, to regenerate the images. The regenerated images are then moved back to the tmp directory and displayed to the user. The packer directory is used for project storage and will be explained in the following section.

0.1.1 Project storage

A typical project created by an user contains images, scenes and sounds. The user will normally spend a lot of time making the animation as good as possible, and he/she will of course want to have the ability to save it for editing at a later point in time. But how should we save all the images etc? We doesn’t want to just leave the tmp directory opened and let the user manually add everything again next time he/she wants to do more editing. Just imagine a project containing thousands of images placed in different scenes with couple of sounds. Loading of a saved project should therefore be an easy operation for the user to do. It’s not just time consuming and cumbersome to load everything manually. It’s also bigger chance to get the data in the tmp directory broken or changed in such a way that it affects the created animation.

To separate the different parts of the animation and make it more structured, we creates two directories – images and sounds – inside the packer directory. Then, when

[1] This is an environment variable pointing to the users home directory, e.g. /home/foo. This is referred to as ~/ later in this document.

```
bjoern@bjoern:~/.stopmotion/packer/tuxdance$ ls
totalt 12
drwxr-xr-x  2 bjoern bjoern 4096 2005-05-11 03:41 images
drwxr-xr-x  2 bjoern bjoern 4096 2005-05-11 03:42 sounds
-rw-r--r--  1 bjoern bjoern  929 2005-05-11 03:43 tuxdance.dat
bjoern@bjoern:~/.stopmotion/packer/tuxdance$ cat tuxdance.dat
<?xml version="1.0"?>
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 2.0//EN" "http://www.w3.org/2001/SMIL20/SMIL20.dtd">
<smil xmlns="http://www.w3.org/2001/SMIL20/Language" xml:lang="en" title="Stopmotion">
  <scenes>
    <seq>
      <images>
        <img src="000001.jpg"/>
        <img src="000002.jpg"/>
        <img src="000003.jpg">
          <sounds>
            <audio src="000003_snd_1.ogg" alt="Comfortable background music"/>
          </sounds>
        </img>
        <img src="000004.jpg"/>
        <img src="000005.jpg"/>
        <img src="000006.jpg"/>
        <img src="000007.jpg"/>
        <img src="000008.jpg"/>
        <img src="000009.jpg"/>
        <img src="000010.jpg"/>
        <img src="000011.jpg"/>
        <img src="000012.jpg"/>
        <img src="000013.jpg"/>
        <img src="000014.jpg"/>
        <img src="000015.jpg"/>
        <img src="000016.jpg"/>
      </images>
    </seq>
  </scenes>
</smil>
bjoern@bjoern:~/.stopmotion/packer/tuxdance$
```

Figure 1: The structure for a saved project

the user decides to save the project, images are moved to the “packer/<projectname>/-images” directory and sounds to the “packer/<projectname>/sounds” directory. This will not make it easier and less time consuming for the user to load a saved project, but it will be easier for us to create a function doing it. The only problem is how we could determine which images and sounds which belongs to the different scenes. It’s actually not a big problem because a XML file can do it for us. The structure, paths to images and sounds and which scenes they belong to, is therefore saved in a XML file located at packer/<projectname>/<projectname>.dat (see figure1 on page 2). Then, in our function, we just reads the XML file and recursively builds a DOM tree which represents the structure. We can then use the DOM to decide which action to do (add a new scene, image or sound). The Implementation for this is shown here:

```
void ProjectSerializer::
getAttributes(xmlNodePtr node, vector<Scene*>& sVect)
{
  xmlNodePtr currNode = NULL;
  for (currNode = node; currNode; currNode = currNode->next) {
    if (currNode->type == XML_ELEMENT_NODE) {
```

```
    char *nodeName = (char*)currNode->name;
    // We either have an image node or a sound node
    if ( strcmp(nodeName, "img")   == 0  ||
         strcmp(nodeName, "audio") == 0     ) {
      char *filename =
              (char*)xmlGetProp(currNode, BAD_CAST "src");
      if (filename != NULL) {
        char tmp[256] = {0};
        // The node is an image node
        if ( strcmp(nodeName, "img") == 0 ) {
          sprintf(tmp, "%s%s", imagePath, filename);
          // Create a new fram with the given file path
          Frame *f = new Frame(tmp);
          // and add it to the current scene
          Scene *s = sVect.back();
          s->addSavedFrame(f);
          f->markAsProjectFile();
        }
        // The node is a sound node
        else {
          sprintf(tmp, "%s%s", soundPath, filename);
          // Get the current scene
          Scene *s = sVect.back();
          // with the current frame
          Frame *f = s->getFrame(s->getSize() - 1);
          // and add the sound to this frame
          f->addSound(tmp);
          char *soundName =
                  (char*)xmlGetProp(currNode, BAD_CAST"alt");
          // If the sound has a name setted
          if (soundName != NULL) {
            unsigned int soundNum = f->getNumberOfSounds() - 1;
            f->setSoundName(soundNum,  soundName);
          }
        }
      }
    }
    // The node is a scene node
    else if ( strcmp((char*)currNode->name, "seq") == 0 ) {
      Scene *s = new Scene();
      sVect.push_back(s);
    }
  }
  // Get attributes for the children to the current node
  getAttributes(currNode->children, sVect);
}
```

```
}
```

We have now reached the point that it's easy for the user to load a saved project, but the directories are still wide open and easy to damage. It's also difficult to move the project to another location. The solution is to pack everything together into a tarball file and append it with a .sto extension to show that it is a Stopmotion project. This is the same way as openoffice.org has decided to have theirs file format. The only difference is that openoffice.org uses a zip file instead of a tarball. The implementation for packing everything into a tarball is written in pure C and uses the libtar API.

The .sto extension is added as own MIME type to the system when installing the Debian package. If the user prefer to doble click for opening programs, it will be easy to just dobbel click the .sto file to open Stopmotion with the given project file. This file can easily be moved to an another location and be loaded into Stopmotion.

Loading an already saved project works quite the same way as for saving a project. The difference is that when the user loads a project, the project-file is unpacked to the packer directory and all of the images and sounds are already placed in its respective directories. We then gets the same structure as described above. So, if the user decides to add more images to the loaded project, everything works as for creating a new project: The images will be added to the tmp directory and later on moved to the directory inside the packer directory on saving. Finally, everything is packed together in a .sto file where the name and location of the file is chosen by the user.

0.1.2 Recovery mode

You might already have been wondering about when we're deleting all of the directories created in ~/.stopmotion/. The program has registered a clean-up function to be run at normal program termination. Normal program termination means if the user has pressed quit or closed the application with pressing the cross in the upper right corner. That means, if something causes stopmotion to exit abnormally the directories containing the data will still be intact. When the user then restarts the program we can figure out, by checking if the directories still exits and contains data, if the program was exited abnormally last time it was run. It's then easy to regenerate exactly the same structure of the program as it was at the point it was exiting abnormally.

If ~/.stopmotion/packer/foo/ with its sub directories still exists it means that the user had a project called foo (stored in the file foo.sto) opened. And if there also exists images in the tmp directory it means that the user had added additional images to the foo project without saving it. As you probably remember, the deleted images are placed in the trash directory. So even though the undo history for deleting of images can be reproduced.

All of the above discussed directories are deleted if the user doesn't wants to recover the data. The initializing routine will then be run and we're back to the beginning where everything is "ready to be used".

Public Member Functions

bool	**setPreferencesFile** (const char *filePath, const char *version)
bool	**setPreference** (const char *key, const char *attribute, bool flushLater=false)
bool	**setPreference** (const char *key, const int attribute, bool flushLater=false)
const char *	**getPreference** (const char *key, const char *defaultValue)
const int	**getPreference** (const char *key, const int defaultValue)
void	**removePreference** (const char *key)
bool	**flushPreferences** ()

Figure 2: PreferencesTool Doxygen API documentation

0.1.3 Preferences

Most programs have some sort of preferences that needs to be saved. For this reason we were quite surprised when Qt didn't have a system for saving preferences, like Javas Preferences and Properties packages.

We decided to implement our own preferences system, and this was done through a singleton class in the Foundation layer called PreferencesTool. The PreferencesTool use libXML2 as this was already an project dependency and saves the preferences in key-value pairs.

Figure 2 shows the api to the PreferencesTool class. As the tool is a singleton the functions are used by calling the static get functions on the class for retrieving the singleton instance and then call its functions, eg:

```
PreferencesTool::get()->setPreference(...);
```

The *setPreferencesFile* function is used to specify where the preferences file should be saved as well as the version. In our case the preferences file is saved in ~/.stopmotion/preferences.xml.

The version is used to make the preferences for a program backward compatible. The *setPreference* functions are used for storing preferences. They take a *key*, for example "fps-setting" with and an *attribute*. The attribute can either be an int or a char. The preference will be flushed to file instantly to avoid loss of data if the program crashes, but if one wants to save allot of preferences at the same time one can set the *flushLater* attribute and then call the *flushPreferences* function when all the preferences are saved. The *getPreference* functions are obviously used for retrieving preferences.

The way the PreferencesTool works internally is by creating a XML tree which mirrors the XML file. When one ask for a preference it retrieves this preference from the three and when a preference is saved a node is attached to the three and the node is flushed to disk. We considered using hashing for retrieving nodes, but we decided this wasn't necessary because of the relative small amount of preference information almost all programs have.

The following code demonstrates how one of the the functions for adding a preference adds a node to the internal three and flushes it to disk:

```
bool PreferencesTool::setPreference(const char* key,
        const char* attribute, bool flushLater )
{
```

```
    checkInitialized();
    xmlNodePtr node = NULL;
    node = findNode(key);

    if( node == NULL ) {
        node = xmlNewChild(preferences, NULL,
                BAD_CAST "pref", NULL);
        xmlNewProp(node, BAD_CAST "key",
                BAD_CAST key);
        xmlNewProp(node, BAD_CAST "attribute",
                BAD_CAST attribute);
    }
    else {
        xmlSetProp(node, BAD_CAST "attribute",
                BAD_CAST attribute);
    }

    return (!flushLater) ? flushPreferences() : true;
}
```

The preferences tool, like all classes, in the foundation layer, is very general. It can easily be reused in other application and has been used for preferences storage in the AdminWorm application[**?**]. It can run on all platform libXML2 runs on which means most platforms.