0.1 Video import

0.1.1 GStreamer

For video import we first chose to go with gstreamer which is a framework for creating multimedia applications. The reasons for choosing gstreamer were many. It has a fairly large base of developers, it supports many input devices (v4l, DV, sound devices, etc) and it is the official multimedia backend for GNOME as well as proposed as the multimedia backend for KDE 4.0.

Some time was spent learning gstreamer and after a lot of trying and failing we managed to create a pipeline which displayed video from web-cameras to the user and allowed the user to grab images and add them to the project. This pipeline was with our application for two releases, but it soon became apparent that gstreamer was ill suited for combining video and still pictures. For our application we also needed still images to be mixed/onionskinned on the video as well as the ability to combine images to a video stream and save this video as a mpeg/avi file.

Almost 150 hours was spent fighting with bugs and quirks in gstreamer without being able to get it working properly. One issue we spent a lot of time with was that the images we imported from .png files was broken. After a lot of time someone told us that this was a bug with the Debian repository version of gstreamer. After waiting for 3 weeks a new version of the gstreamer plugins library was released and we hoped to be able to get it working now, only to discover new bugs. This, in addition to several other issues with gstreamer, caused us to abandon it and look for other solutions.

0.1.2 External programs

After a meeting with the project supervisor and the customer we decided to use external programs to import live video from the web-camera. The way this works is that the user specifies a command line which calls another program that grabs a picture from the camera and puts it on the disk. Stopmotion then repeatedly calls this command line and when the picture is saved on the disk Stopmotion imports the picture and displays it to the user, thereby creating live video. Stopmotion comes with several pre-set command-lines, and the Debian package has dependency to our default grabber program which is a small application called vgrabbj.

The advantage with this way of doing it is that our application is a lot more flexible and supports all input devices, as long as the user has a program which can grab a picture from it. The disadvantage is that it tends to consume a lot of resources.

We wanted to write a module for streaming video directly from V4L, but when we finally decided to abandon GStreamer we didn't have enough time left for both the external program and V4L importing so we chose to go with the external programs because we though this option would to be more flexible and robust.

0.1.3 Camera viewing modes

Creating Stop Motion animations is hard and time consuming. In order to create lifelike animations the animators need some tools to help them move the figures between shots.

Figure 1: Onionskinning/Image mixing in Stopmotion

0.1.3.1 Onionskinning/Image mixing

Onionskinning is perhaps the most commonly used feature by stop motion animators. As explained in the user manual in chapter **??** on page **??** Stopmotion allows the user to see the next frame in relation to the other and help them create lifelike motions. The way onionskinning is implemented in Stopmotion is through alpha channels.

When a new image is retrieved from the camera Stopmotion takes up to five of the previous frames and uses SDL to add alpha channels to them with increasingly smaller alpha values. These images are then blitted/merged together and shown to the user. As you can see from figure 1 the camera frame is clears with the frames becoming more and more faded out the further back in time they are (lower alpha values).

To increase the efficiency the images which are merged on top of the camera are buffered in the memory when using this mode. This was necessary as this mode took too much resources when retrieving them from disk.

The following lines of code shows how an alpha channel can be added to a frame image surfaces using SDL. The surface is then blitted onto the screen surface:

```
SDL_SetAlpha(frameSurfaces, SDL_SRCALPHA, 80/i);
SDL_BlitSurface(frameSurface, NULL , screen, &dst2);
```

The *i* variable is a loop counter which increases as frames backwards are onto the screen. The 80/i value which is inputed in the `SDL_SetAlpha(..)` function is the alpha value and as i increases this value decreases.

0.1.3.2 Image differentiating

Another mode, which was proposed by Øyvind Kolås, is the Image Differentiation mode. Øyvind suggested we have a way to see the difference between the camera and a given frame. This is a feature no similar program we have seen has and the initial

Figure 2: Image differentiation in Stopmotion

thought was that it would be helpful for moving an object to a previous position in case it falls or similar.

When this feature was implemented however it proved a very useful addition to the Onionskinning for moving the figures around, and one user even reported that he preferred it to the Onionskinning.

The way the image differentiation is implemented is by taking the absolute difference between the three color components red, green and blue in the two pictures:

```
deltaRed = |red1 - red2|
deltaGreen = |green1 - green2|
deltaBlue = |blue1 - blue2|
```

This will give the output shown in figure 2. The implementation of this uses SDL to access the image surface and for converting the new RGB values back to a pixel. It originally used the two functions getPixel() and putPixel() from the SDL webpage[1], but these were too general as they calculated the position of a pixel by multiplying Y with the rowstride factor(pixels per row) and then added the X value multiplied with bit per pixel. As our code never needs to access pixels randomly but always traverses the entire surface this algorithm was later optimized to access the surface pixel arrays directly.

The code isn't as clear as it was before this optimization. but according to tests it is approximately twice as fast. On a Pentium 4, 2.4 Ghz it now takes about 0.242 secs on the average to load two images and then run the differentiateSurfaces algorithm 10 times. The test for this can be found in the directory *implementation/prototypes/sdl_difference_prototype/algorithmtes*t. The following code shows how the differentiation function works:

[1] www.libsdl.org

```
SDL_Surface* FrameView::differentiateSurfaces(
        SDL_Surface *s1, SDL_Surface *s2) {
    int width = s2->w;
    int height = s2->h;
    SDL_Surface *diffSurface = SDL_CreateRGBSurface(
        SDL_SWSURFACE, width, height, 32, 0xff000000,
        0x00ff0000, 0x0000ff00, 0x000000ff);

    //Lock the surfaces before working with the pixels
    SDL_LockSurface( s1 );
    SDL_LockSurface( s2 );
    SDL_LockSurface( diffSurface );

    //Pointers to the first byte of the
    //first pixel on the input surfaces.
    Uint8 *p1 = (Uint8 *)s1->pixels;
    Uint8 *p2 = (Uint8 *)s2->pixels;

    //Pointer to the first pixel on the resulting surface
    Uint32 *pDiff =  (Uint32 *)diffSurface->pixels;

    SDL_PixelFormat fDiff = *diffSurface->format;
    Uint32  differencePixel;
    Uint8 dr, dg, db;
    int offset = 0, pixelOffset = 0;
    int i, j;

    //Goes through the surfaces as one-dimensional arrays.
    for(i=0; i<height; ++i) {
        for(j=1; j<width; ++j) {
            //px[offset] is the red value of surface x,
            //px[offset+1] the green, etc.
            dr = abs(p1[offset] - p2[offset]);
            dg = abs(p1[offset+1] - p2[offset+1]);
            db = abs(p1[offset+2] - p2[offset+2]);

            differencePixel = SDL_MapRGB(&fDiff, dr, dg, db);

            pDiff[pixelOffset++] = differencePixel;
            offset += 3;
        }
        ++pixelOffset;
        offset += 3;
    }

    //Unlock the surfaces for displaying them.
```

```
    SDL_UnlockSurface( s1 );
    SDL_UnlockSurface( s2 );
    SDL_UnlockSurface( diffSurface );

    return diffSurface;
}
```

0.1.3.3 Playback

The playback mode is a mode where the user can see a continuous playback of the last up to fifty frames with the camera as the final picture. This way the user can view the next shot together with the other shots live thereby avoiding jerky movements.

The way this is implemented is by using a starting a timer which continuously calls a function called `nextPlayBack()`. The interval between each time this function is called is set by the user as explained in the user manual. The `nextPlayBack()` will play the frames up to, and including, the selected frame. After it has played the selected frame it will call the FrameViews `redraw()` function causing the frameview to display the picture from the camera instead of a frame from the animation.