千 Line 으로 비디오플레이어 만들기 5

멀티미디어 : 2009/02/27 14:54

ffmpeg 천줄로 비디오플레이어 만들기
http://www.dranger.com/ffmpeg/ffmpeg.html
http://hybridego.net/entry/천줄로-비디오플레이어-만들기
http://hybridego.net/entry/千-Line-으로-비디오플레이어-만들기-1
http://hybridego.net/entry/千-Line-으로-비디오플레이어-만들기-2
http://hybridego.net/entry/千-Line-으로-비디오플레이어-만들기-3
http://hybridego.net/entry/千-Line-으로-비디오플레이어-만들기-4
http://hybridego.net/entry/千-Line-으로-비디오플레이어-만들기-5
http://hybridego.net/entry/千-Line으로-비디오플레이어-만들기-6
http://hybridego.net/entry/千-Line으로-비디오플레이어-만들기-7
http://hybridego.net/entry/千-Line으로-비디오플레이어-만들기-8
http://hybridego.net/entry/千-Line으로-비디오플레이어-만들기-마무리
별로 중요하지 않은 부분중 생략된 부분이 있고 의역이 많이 있습니다.
군데군데 틀린 번역이 있을수 있습니다. (영어가 후달려서)

하지만 소스코드 설명 부분은 틀리지 않도록 노력했습니다.

위의 원문을 번역한 것입니다.
잘못된 부분이 있을지도 모르겠습니다.
혹시 잘못된 부분을 발견하시면 댓글로 알려주시면 감사하겠습니다.

Tutorial 05: Synching Video

tutorial05.c

How Video Syncs

우리는 지금까지 무비플레이어로는 쓸모없는 것을 만들었습니다. 비디오와 오디오가 플레이 되기는 하지만 동영상이라고 부르기에는 아직 멀었네요. 이제 우리는 무엇을 해야 할까요?

PTS and DTS

운좋게 오디오와 비디오가 갖고 있는 정보에는 얼마나 빨리 그리고 언제 플레이를 해야할지에 대한 정보가 들어 있습니다.
오디오 스트림에는 sample rate가 있고, 비디오 스트림에는 frame per second 값이 있습니다. 하지만 만일 우리가 간단하게 다양한 frame rate에 의해 단순히 계속 Frame을 진행시켜서 비디오 싱크를 맞춘다면, 오디오 싱크와 어긋날 가능성이 있습니다. 대신에 스트림의 패킷은 아마 Deconding time stamp(DTS) 와 Presentation time stamp(PTS)라고 불리는 것을 갖고 있을 것입니다. 이 두개의 값을 이해하기 위해서 여러분은 동영상의 저장방식에 대해서 알아야 합니다. 몇몇 MPEG같은 포맷들은 'B' 프레임(B stands for "bidirectional")이라고 불리우는 것을 사용합니다. 그리고

'I'프레임과 'P'프레임("I" for "intra" and "P" for "predicted") 이라고 불리우는 것이 있습니다. I 프레임은 Full Image 를 포함하고 있고, P프레임은 앞선 I프레임과 P프레임의 차이 정보를 갖고 있습니다. B프레임은 P프레임과 유사한데 전후로 디스플레이된 프레임에 의해 계산된 프레임 입니다.
이것은 avcodec_decode_video 를 호출한 다음에 프레임이 끝나지 않는지를 설명합니다.

우리가 영상을 갖고 있고, 프레임은 다음과 같이 디스플레이 되었습니다. I B B P. 이제 우리는 B프레임을 디스플레이 하기 전에 P프레임에 대한 정보를 알아야 합니다. 때문에 프레임들은 아마 다음과 같이 저장되어 있을 것입니다. I P B B. 이것은 왜 Decoding timestamp 와 Presentaion timestamp가 각 프레임마다 있어야 하는지를 말해줍니다. Decoding timestamp는 언제 디코딩해야 하는지를 말해줍니다. 그리고 Presentation timestamp는 언제 디스플레이 해야하는지를 알려줍니다.
그래서 이경우에는 우리 스트림은 다음과 같이 됩니다.

```
  PTS: 1 4 2 3
  DTS: 1 2 3 4
Stream: I P B B
```

일반적으로 PTS와 DTS는 언제 B 프레임을 플레이 해야하는지에 대한것만 다릅니다.
우리가 av_read_frame()으로 패킷을 얻을 떄, 패킷 안에는 PTS와 DTS 값이 포함되어 있을것입니다. 하지만 우리가 정말 원하는 것은 새로 디코드된 RAW Frame의 PTS기 때문에 그것이 언제 디스플레이 하는지 압니다. 그러나 avcodec_decode_video()로 없은 프레임은 우리에게 쓸모있는 PTS값이 있는지 없는지 모르는 AVFrame를 넘겨줍니다. (※주의 : AVFrame은 PTS값을 갖고 있지만 우리가 언제 프레임을 얻기를 원하는지에 대한 정보가 항상 포함되어 있는것은 아니다.)그러나 ffmpeg는 패킷을 재 정렬하고 패킷의 DTS는 avcodec_decode_video()에 의해 처리되기 시작합니다. avcodec_decode_video()는 항상 리턴된 프레임의 PTS와 같습니다. (하지만 우리가 항상 이 정보를 원하는 것은 아닙니다.)

걱정하지 마세요. 프레임에서 PTS를 얻는 다른 방법이 있습니다. 그리고 우리는 프로그램에서 스스로 재정렬되게 할 수 있습니다. 우리는 프레임의 첫번재 패킷의 PTS를 저장합니다. 이것은 마지막 프레임의 PTS가 될것입니다. 그래서 스트림이 DTS를 넘겨주지 않을때, 우리는 그냥 저장된 PTS를 사용하면 됩니다. 우리는 avcodec_decode_video()가 알려줌으로써 프레임의 첫번째 패킷을 만들수 있습니다. 어떻게? 언제든지 한 패킷은 한 프레임을 시작합니다. avcodec_decode_video()은 우리 프레임에 한 버퍼를 할당하는 함수를 호출할 것입니다. 그리고 물론 ffmpeg는 그 할당함수를 재정의 할수 있게 해줍니다. 이제 패킷의 PTS를 저장하는 새 함수를 만들겠습니다.

우리가 알맞은 PTS를 얻지 못했더라도. 이것은 다음에 다루겠습니다.

Synching

이제 우리는 언제 각각의 비디오 프레임을 보여주어야 하는지 알게 되었습니다. 하지만 그것은 실제로 어떻게 해야 할까요?
한 프레임을 보여준 다음 언제 다음 프레임을 보여줘야 할지에 대한 방법이 여기 있습니다. 그러면 간단히 일정시간이 지나면 반복적으로 비디오를 Refresh 하기 위한 Timeout을 새롭게 설정하겠습니다.여러분이 기대하시는 것처럼 우리는 타임아웃이 얼마나 걸리는지 시스템 클럭을 보기위해 다음 프레임의 PTS의 값을 체크합니다. 이 접근 작업은 기본적으로 두가지 이슈가 있습니다.

첫번째 이슈는 언제 다음 PTS가 될것이냐 입니다. 여러분은 아마 그냥 Video rate를 현재 PTS에 추가한다고 생각할지도 모릅니다. 그렇습니다. 하지만 몇몇 종류의 비디오들은 반복되는 프레임들을 호출합니다. 이 의미는 현재 프레임을 일정시간동안 반복시켜야 한다는 것입니다. 프로그램이 다음 프레임을 너무 빨리 디스플레이 하기 때문에 그렇습니다. 우리는 그래서 그것을 계산해야 합니다.

두번째 이슈는, 프로그램은 멈춰있고 비디오와 오디오는 흘러가고 있는데 싱크가 걱정스럽습니다. 만약 모든것이 완벽하게 이루어진다면 걱정할 필요가 없지만 여러분의 컴퓨터는 그렇지 못합니다. 그리고 많은 비디오 파일들도 그렇지 않습니다. 그래서 우리는 세가지 선택을 할수 있는데 오디오 싱크를 비디오에 맞추거나, 비디오 싱크를 오디오에

맞추거나, 아니면 외부 클럭에 둘다를 맞추는 것(여러분들의 컴퓨터 처럼)입니다. 이번에는 비디오 싱크를 오디오 싱크에 맞추도록 하겠습니다.

Coding it: getting the frame PTS

이제 코드를 짜보도록 하겠습니다. 우리의 큰 구조체 안에다가 새로운 멤버들을 추가해야 합니다. 먼저 우리의 비디오 쓰레드를 보세요. 디코드 쓰레드에 의해 큐에 넣어진 패킷을 어디에서 꺼내는지 기억해야 합니다. avcodec_decode_video에 의해 우리에게 주어진 프레임의 PTS를 얻는 코드가 필요합니다. 첫번째 방법은 마지막 패킷이 처리된 DTS를 얻는 것입니다.

view plain copy to clipboard print ?

```
01.  double pts;
02.
03.    for(;;) {
04.      if(packet_queue_get(&is->videoq, packet, 1) < 0) {
05.
06.        break;
07.      }
08.      pts = 0;
09.
10.      len1 = avcodec_decode_video(is->video_st->codec,
11.                                  pFrame, &frameFinished,
12.                  packet->data, packet->size);
13.      if(packet->dts != AV_NOPTS_VALUE) {
14.        pts = packet->dts;
15.      } else {
16.        pts = 0;
17.      }
18.      pts *= av_q2d(is->video_st->time_base);
```

avcodec_decode_video(), av_q2d()
이것을 처리하지 못하면 PTS를 0으로 설정합니다.

만약 패킷의 DTS가 우리를 도와주지 못한다면 우리는 디코드된 프레임의 첫번째 패킷의 PTS를 이용해야 합니다. ffmpeg는 우리가 만든 함수로 한 프레임을 할당합니다. 다음이 그 함수 입니다.

view plain copy to clipboard print ?

```
01.  int get_buffer(struct AVCodecContext *c, AVFrame *pic);
02.  void release_buffer(struct AVCodecContext *c, AVFrame *pic);
```

get 함수는 패킷에 대한 어떤 것도 알려주지 않습니다. 그래서 패킷을 얻을때 마다 PTS를 클로벌 변수에 저장할 것입니다. 그리고 우리 get 함수는 그것을 읽기만 하면 됩니다. 그러면 우리는 AVFrame 구조체의 opaque 변수에 저장할 수 있습니다. 이것은 User-defined 변수이고 이것을 우리가 원하는대로 쓸수 있습니다. 우선 여기 함수를 보겠습니다.

view plain copy to clipboard print ?

```
01.  uint64_t global_video_pkt_pts = AV_NOPTS_VALUE;
02.
03.
04.
05.
06.
07.  int our_get_buffer(struct AVCodecContext *c, AVFrame *pic) {
08.    int ret = avcodec_default_get_buffer(c, pic);
09.    uint64_t *pts = av_malloc(sizeof(uint64_t));
10.    *pts = global_video_pkt_pts;
11.    pic->opaque = pts;
```

```
  return ret;
}
void our_release_buffer(struct AVCodecContext *c, AVFrame *pic) {
  if(pic) av_freep(&pic->opaque);
  avcodec_default_release_buffer(c, pic);
}
```

avcodec_default_get_buffer 와 avcodec_default_release_buffer 는 ffmpeg가 할당을 위해 사용하는 기본 함수 입니다. av_freep 는 메모리 관리 함수입니다. 이것은 지정되어 있는 메모리를 해제하는것 뿐만 아니라 포인터를 NULL로 설정해 줍니다.

이젠 스트림 Open 함수(stream_component_open)로 가보겠습니다. 우리는 ffmpeg에게 뭘 할지 알려주기 위해 다음 라인을 삽입합니다.

view plain copy to clipboard print ?

```
codecCtx->get_buffer = our_get_buffer;
codecCtx->release_buffer = our_release_buffer;
```

이제 전역변수에 PTS를 저장하는 코드를 추가해야 합니다. 그리고 저장된 PTS를 필요할때 사용합니다.
코드는 다음과 같습니다.

view plain copy to clipboard print ?

```
for(;;) {
    if(packet_queue_get(&is->videoq, packet, 1) < 0) {

      break;
    }
    pts = 0;


    global_video_pkt_pts = packet->pts;

    len1 = avcodec_decode_video(is->video_st->codec, pFrame, &frameFinished,
                packet->data, packet->size);
    if(packet->dts == AV_NOPTS_VALUE
       && pFrame->opaque && *(uint64_t*)pFrame->opaque != AV_NOPTS_VALUE) {
      pts = *(uint64_t *)pFrame->opaque;
    } else if(packet->dts != AV_NOPTS_VALUE) {
      pts = packet->dts;
    } else {
      pts = 0;
    }
    pts *= av_q2d(is->video_st->time_base);
```

avcodec_decode_video(), av_q2d()
우리는 PST를 int64로 사용하고있습니다. 이것은 PTS가 하나의 Interger로 저장되어 있기 때문입니다. 하나의 timestamp 값은 스트림의 time_base 에 있는 시간을 측정한것과 일치합니다. 예를들면 만약 우리가 매 초당 24프레임을 갖고 있다면 42의 PTS는 42번째의 프레임을 지정할 것입니다.
우리는 이 값을 Frame rate로 나누어서 초로 바꿀수 있습니다. 스트림의 time_base 값은 1/framerate 가 될것입니다.(fixed-fps 일 경우). 그래서 초에서 PTS를 얻으려면 우리는 time_base에 곱해야 합니다.

Coding: Synching and using the PTS

이제 PTS를 얻어서 모든 세팅을 다 하였습니다. 이제 우리는 전에 언급했던 싱크하기위한 두가지 문제점에 대해 주의를 해야합니다. 우리는 synchronize_video라는 함수를 정의할 것입니다. 이 함수는 PTS를 전체적으로 싱크되게

업데이트 합니다. 이 함수는 프레임을 위한 PTS 값을 얻을수 없는 경우에도 처리할수 있습니다. 동시에 우리는 언제 다음프레임을 보여줘야 할지를 계속 추적해야 하기때문에 우리는 refresh rate를 알맞게 설정해야 합니다. 우리는 이것을 video가 얼마나 많은 시간이 흘렀는지를 추적하는 내부 video_clock 값을 사용함으로써 완성시킬수 있습니다. 우리는 이 값을 우리의 큰 구조체에 넣겠습니다.

view plain copy to clipboard print ?

```
typedef struct VideoState {
  double          video_clock;
```

다음은 synchronize_video 함수입니다.

view plain copy to clipboard print ?

```
double synchronize_video(VideoState *is, AVFrame *src_frame, double pts) {

  double frame_delay;

  if(pts != 0) {

    is->video_clock = pts;
  } else {

    pts = is->video_clock;
  }

  frame_delay = av_q2d(is->video_st->codec->time_base);

  frame_delay += src_frame->repeat_pict * (frame_delay * 0.5);
  is->video_clock += frame_delay;
  return pts;
}
```

AVFrame, av_q2d()
여러분은 이함수에서 역시 반복되는 프레임을 세야 합니다.

이제는 적절한 PTS를 구하고 queue_picture를 사용해 queue에 frame을 올리겠습니다.

view plain copy to clipboard print ?

```

   if(frameFinished) {
     pts = synchronize_video(is, pFrame, pts);
     if(queue_picture(is, pFrame, pts) < 0) {
break;
     }
   }
```

우리가 queue_picture를 바꾸는 유일한 것은 PTS 값을 VideoPicture 구조체에 저장하는 것입니다. 그래서 PTS 변수를 구조체에 추가하고 다음 한줄을 삽입합니다.

view plain copy to clipboard print ?

```
typedef struct VideoPicture {
  ...
  double pts;
}
int queue_picture(VideoState *is, AVFrame *pFrame, double pts) {
  ... stuff ...
```

```
07.   if(vp->bmp) {
08.     ... convert picture ...
09.     vp->pts = pts;
10.     ... alert queue ...
11.   }
```

AVFrame
그래서 우리는 picture들을 picture queue에 적절한 PTS 값과 같이 순서대로 넣고, Video refresh 함수를 찾습니다. 여러분은 아마 지난번에 우리가 이부분을 속이고 refresh를 80ms로 임의 지정했던것을 기억하실 것입니다. 자. 이번에는 실제로 이것을 어떻게 하는지를 알아보겠습니다.

우리의 전략은 앞선 PTS와 이번것의 시간을 간단히 측정함으로써 다음 PTS의 시간을 예측 하는 것입니다.

동시에 비디오를 오디오에 싱크해야 합니다. 우리는 Audio Clock을 만들것입니다. Audio Clock : 오디오의 어느 부분을 플레이 하고 있는지 추적하는 내부의 값. 어느 mp3 player 에서도 디지털로 읽어낼수 있습니다. It's like the digital readout on any mp3 player. Since we're synching the video to the audio, the video thread uses this value to figure out if it's too far ahead or too far behind.

우리는 다음에 할것이고 지금은 일단 get_audio_clock 함수를 갖고 있다고 치고 audio clock의 시간을 주겠습니다. 한번 우리가 값을 갖고 있었더라도 비디오와 오디오의 싱크가 어긋나면 우리는 어떻게 해야 할까요? 멍청하게 간단히 시도하고 검색들을 통해 정확한 패킷으로 뛰어넘어가야 할것입니다. 대신 우리는 그냥 다음 refresh를 위해 계산했던 값으로 조정할 것입니다. 만일 PTS가 오디오 타임에 너무 뒤에 있으면 계산된 딜레이를 두배로 하고 만약에 PTS가 오디오 보다 너무 앞에 있다면 그냥 refresh를 빠르게 하면 됩니다. 이제 우리는 조정된 시간과 delay을 갖게 되었고, 다음으로는 우리 실행중인 frame_timer에 의한 컴퓨터의 클럭과 비교를 해야 합니다. 이 프레임 타이머는 동영상이 플레이 되는 동안의 계산된 delay를 모두 더할것입니다. 다른말로 이 frame_timer는 언제 우리가 다음 프레임을 디스플레이 할지를 정하는 것입니다. 간단히 frame timer에 새로운 delay를 더하고 우리 컴퓨터의 clock과 비교를 합니다. 그리고 그 값을 이용해서 다음 refresh 스케쥴을 짭니다. 이것은 약간 복잡하기 때문에 주의 깊게 보셔야 합니다.


```
01. void video_refresh_timer(void *userdata) {
02.
03.   VideoState *is = (VideoState *)userdata;
04.   VideoPicture *vp;
05.   double actual_delay, delay, sync_threshold, ref_clock, diff;
06.
07.   if(is->video_st) {
08.     if(is->pictq_size == 0) {
09.       schedule_refresh(is, 1);
10.     } else {
11.       vp = &is->pictq[is->pictq_rindex];
12.
13.       delay = vp->pts - is->frame_last_pts;
14.       if(delay <= 0 || delay >= 1.0) {
15.
16.     delay = is->frame_last_delay;
17.       }
18.
19.       is->frame_last_delay = delay;
20.       is->frame_last_pts = vp->pts;
21.
22.
23.       ref_clock = get_audio_clock(is);
24.       diff = vp->pts - ref_clock;
25.
26.
27.
28.       sync_threshold = (delay > AV_SYNC_THRESHOLD) ? delay : AV_SYNC_THRESHOLD;
29.       if(fabs(diff) < AV_NOSYNC_THRESHOLD) {
30.     if(diff <= -sync_threshold) {
```

```
      delay = 0;
    } else if(diff >= sync_threshold) {
      delay = 2 * delay;
    }
      }
      is->frame_timer += delay;

      actual_delay = is->frame_timer - (av_gettime() / 1000000.0);
      if(actual_delay < 0.010) {

    actual_delay = 0.010;
      }
      schedule_refresh(is, (int)(actual_delay * 1000 + 0.5));

      video_display(is);


      if(++is->pictq_rindex == VIDEO_PICTURE_QUEUE_SIZE) {
    is->pictq_rindex = 0;
      }
      SDL_LockMutex(is->pictq_mutex);
      is->pictq_size--;
      SDL_CondSignal(is->pictq_cond);
      SDL_UnlockMutex(is->pictq_mutex);
    }
  } else {
    schedule_refresh(is, 100);
  }
}
```

av_gettime(), SDL_LockMutex(), SDL_CondSignal(), SDL_UnlockMutex()
우리가 만든것을 몇개 체크해야 할게 있습니다.
첫째 PTS와 전 PTS같에 딜레이를 확실히 해야 합니다. 만일 그것이 안되면 우리는 추측하거나 마지막 delay를 사용합니다. 다음으로는 싱크가 완벽하게 맞을 수 없기 때문에 우리는 싱크의 threshold를 갖고 있어야 합니다. ffplay는 이 값으로 0.01을 사용합니다. 우리는 또한 싱크의 threshold를 PTS간의 차보다 절대로 작게 하지 않도록 해야 합니다. 마지막으로 최소 refresh 값을 10ms 로 합니다.
우리는 변수들을 큰 구조체에 추가 하고 코드를 체크하는 것을 잊지 말아야합니다. 또한 frame timer를 초기화 하고 stream_component_open 에서 이전 프레임의 delay를 초기화 것을 잊어서는 안됩니다.

```
view plain  copy to clipboard  print  ?
is->frame_timer = (double)av_gettime() / 1000000.0;
is->frame_last_delay = 40e-3;
```

av_gettime()

Synching: The Audio Clock

Audio clock을 실행할 차례 입니다. 우리는 audio_decode_frame 함수에서 어디서 오디오를 디코드 하는 지를 알려주는 clock time을 업데이트 할수 있습니다. 이제 이 함수를 호출할때 마다 새 패킷을 처리할 필요가 없어졌습니다. 그래서 clock을 업데이트 해야 하는 두 장소가 있습니다. 첫번째 장소는 새 패킷을 얻는곳 : 우리는 간단히 audio clock을 패킷의 PTS로 설정합니다. 그리고 만약 한 패킷이 여러개의 프레임을 갖고 있다면 샘플들의 수를 센것에 의해 오디오 플레이 시간을 유지합니다. 그리고 주어진 samples-per-second rate에 의해 그것들을 multiplying 합니다. 그래서 우리는 그 패킷을 얻었습니다.

```
view plain  copy to clipboard  print  ?

if(pkt->pts != AV_NOPTS_VALUE) {
  is->audio_clock = av_q2d(is->audio_st->time_base)*pkt->pts;
```

```
04.   }
```

av_q2d()
그리고 한번 패킷을 처리하고 있습니다.

```
view plain  copy to clipboard  print  ?
01.
02.   pts = is->audio_clock;
03.   *pts_ptr = pts;
04.   n = 2 * is->audio_st->codec->channels;
05.   is->audio_clock += (double)data_size /
06.   le)(n * is->audio_st->codec->sample_rate);
```

이제 드디어 get_audio_clock 함수를 실행할 수 있습니다. 이것은 is->audio_clock 값을 얻는것 만큼 간단하지 않습니다. 우리가 이것을 처리할 때마다 audio PTS를 설정해야 합니다. 그러나 우리가 audio_callback 함수를 보면 output buffer에 오디오 패킷의 모든 데이타를 이동하는데 시간이 걸립니다. 이것은 audio clock안에 있는 값이 너무 앞으로 갔다는 것을 의미합니다. 그래서 우리는 얼마나 쓰기위해 지나갔는지 체크해야 합니다.
다음이 그 코드 입니다.

```
view plain  copy to clipboard  print  ?
01.  double get_audio_clock(VideoState *is) {
02.    double pts;
03.    int hw_buf_size, bytes_per_sec, n;
04.
05.    pts = is->audio_clock;
06.    hw_buf_size = is->audio_buf_size - is->audio_buf_index;
07.    bytes_per_sec = 0;
08.    n = is->audio_st->codec->channels * 2;
09.    if(is->audio_st) {
10.      bytes_per_sec = is->audio_st->codec->sample_rate * n;
11.    }
12.    if(bytes_per_sec) {
13.      pts -= (double)hw_buf_size / bytes_per_sec;
14.    }
15.    return pts;
16.  }
```

여러분은 왜 이 함수가 지금까지 사용되었는지 말할수 있을 것입니다.

자! 이제 컴파일 해보겠습니다.

```
gcc -o tutorial05 tutorial05.c -lavutil -lavformat -lavcodec -lz -lm`sdl-config --cflags --libs`
```

드디어 여러분의 무비 플레이어로 영상을 볼수 있게 되었네요!
다음번에는 오디오 싱크에 대해서 알아보고, 튜토리을이 끝난후 탐색에 관해 이야기 해보겠습니다.

千 Line 으로 비디오플레이어 만들기 5 조회수 851

'멀티미디어' 카테고리의 다른 글

千 Line으로 비디오플레이어 만들기 6 (0) 2009/03/03
千 Line 으로 비디오플레이어 만들기 5 (0) 2009/02/27
千 Line 으로 비디오플레이어 만들기 4 (0) 2009/02/17
千 Line 으로 비디오플레이어 만들기 3 (0) 2009/02/14

Posted by Real_G