

번역자 : 전호철 (<http://hybridego.net>)

千 Line 으로 비디오플레이어 만들기

ffmpeg는 비디오 어플을 만드는데 훌륭한 라이브러리 입니다.

ffmpeg는 비디오를 decoding, encoding, muxing, demuxing 하는 어려움을 쉽게 할수 있도록 도와줍니다.

Media 어플을 매우 심플하게 만들수 있습니다.

한가지 문제점은 기초 문서가 부족하다는 것입니다.

자동생성된 doxygen 문서와 ffmpeg 기본 tutorial 뿐입니다.

그래서 제가 디지털 비디오와 오디오 어플이 어떻게 돌아가는지, ffmpeg에 대해 파보려고 결심했고 이 Tutorial을 작성하기로 했습니다.

ffplay 라는 ffmpeg를 이용한 샘플 프로그램이 있습니다.

이것은 완벽한 비디오 플레이어 도구인 ffmpeg를 이용한 간단한 C 프로그램 입니다.

이 Tutorial은 [Martin Böhme](#) 가 쓴 오리지널 Tutorial의 업데이트 버전으로 시작할 것입니다.

그리고 비디오 플레이어를 개발하는 것은 Fabrice Bellard의 ffplay.c를 기반으로 할 것입니다.

저는 한가지 새로운 아이디어를 소개하고 어떻게 사용하는지 설명할 것입니다.

각 Tutorial마다 소스파일을 다운 받을수 있습니다.

소스파일에서 실제 프로그램의 흐름을 자세히 볼수 있을 것입니다.

이 Tutorial이 끝날때 까지 1000 라인 안에 비디오 플레이어를 만들어 봅시다! ㅋㅋ

플레이어를 만들때 오디오 비디오 output을 위해 [SDL](#)을 사용할 것입니다.

SDL은 MPEG 재생기, emulator, 많은 게임들 등을 만들때 사용되는 멋진 Cross-platform multimedia library 입니다.

이 Tutorial에서 제공하는 프로그램을 Compile 하기 위해서는 SDL 개발 라이브러리를 다운받아 설치해야 합니다.

이 Tutorial을 읽는 대상은 적당한 프로그래밍 배경 지식이 있어야 합니다. 최소한 C 언어를 알아야 하고 queues, mutexes, 기타등등 에 대한 개념이 있어야 합니다.

웹이브파일 Format 같은 멀티미디어에 대한 기본 개념도 있으면 좋지만 매우 자세히 알 필요는 없습니다.

이 문서에서 설명할 것이니까요~

Tutorial 01: Making Screensaps

첫번째 Tutorial 에서 만들어 볼 것은 Video 파일에서 처음 다섯 프레임을 읽어 ppm 이미지 파일로 저장 하는 것입니다.

이 예제를 실행하기 위해서는 libavformat 과 libavcodec 이 설치되어 있어야 합니다.

테스트 하실때에는 컴파일한 파일의 첫번째 아규먼트로 테스트할 비디오 파일명을 입력하면 됩니다.

그럼 시작해볼까요?

Movie 파일은 몇가지 기본적인 요소로 구성되어 있습니다.

첫째, Container 라고 불리는 파일 자체, 그리고 파일정보가 어디에 있는지를 결정하는 Container 의 타입입니다.

AVI 나 Quicktime 같은 것을 Container 라고 부르는 것이죠.

둘째로는 Stream 의 다발 입니다. 예를 들면, 보통 오디오 스트림, 비디오 스트림 같은 것입니다. (Stream 은 '시간에 따른 연속적 데이터 요소'를 흔히 일컫는 말입니다.)

Stream 의 요소는 Frame 이라고 부르지요.

각 Stream 은 각기 다른 Codec 에 의해 인코딩되어 있습니다.

Codec 은 실제 데이터를 어떻게 COded, DECodeD 할것인지를 정의해놓은 것이어서 CODEC 이라는 이름이 붙게되었습니다.

Codec 의 예로는 DivX, MP3 같은것이 있어요

Packet 은 Stream 을 읽은 것입니다.

Packet 은 RAW Frame 으로 decode 될 비트 데이터를 갖고 있는 데이터의 조각입니다.

바로 우리의 프로그램을 위해서 우리가 다루어야 할 것이기도 하지요.

우리의 목적은 완전한 Frame 들을 갖고 있는 패킷이거나 오디오의 경우 multiple Frame 들 다루어야 하는 것입니다. ?????? 🐧

다음은 기초적인 오디오, 비디오를 다루는 내용입니다.

```
10 OPEN video_stream FROM video.avi
20 READ packet FROM video_stream INTO frame
30 IF frame NOT COMPLETE GOTO 20
40 DO SOMETHING WITH frame
50 GOTO 20
```

ffmpeg 를 이용해 멀티미디어파일을 다루는 것은 이 프로그램에서 처럼 매우 간단합니다. 몇몇 프로그램에서는 뭘 하기가 매우 복잡하겠지만.... 그래서 이 Tutorial 에서는 파일을 열고, 파일에 들어있는 비디오 스트림을 읽어서 뭔가를 해보도록 하겠습니다.(PPM 파일에 Frame 을 쓰는것)

Opening the File

첫번째 부분에서 파일을 어떻게 여는지 보도록 합시다.

ffmpeg 에서 첫째로 할일을 라이브러리를 초기화 하는 것 입니다.

(몇몇 시스템에서는 <ffmpeg/avformat.h> , <ffmpeg/avcodec.h> 라고 써야될 수도 있습니다.)

```
1. #include <avcodec.h>
2. #include <avformat.h>
3. ...
4. int main(int argc, char *argv[]) {
5.     av_register_all();
```

모든 파일 포맷과 코덱을 라이브러리와 함께 등록하고, 이것들은 대응되는 포맷이나 코덱이 열릴때 자동으로 사용됩니다.

[av_register_all\(\)](#) 는 딱 한번만 호출해야 하기때문에 우리는 main() 함수 안에 쓰기로 합니다.

자~ 이제 우리는 실제 파일을 열어봅니다.

```
1. AVFormatContext *pFormatCtx;
2.
3. // Open video file
4. if(av_open_input_file(&pFormatCtx, argv[1], NULL, 0, NULL)!=0)
5.     return -1; // Couldn't open file
```

첫번째 아규먼트에서 파일명을 얻습니다. 이 함수는 파일헤더를 읽어오고 파일포맷에 대한 정보를 우리가 지정한 [AVFormatContext](#) structure 에 저장합니다. 마지막 세개의 아규먼트들은 파일포맷, 버퍼크기, 포맷옵션을 지정하게 되어있지만 그냥 NULL 이나 0 을 세팅합니다. libavformat 이 자동으로 찾거든요.

이 함수는 단지 헤더를 찾는 일만 합니다. 그래서 우리는 파일에서 스트림 정보를 찾아야 해요~

```
1. // Retrieve stream information
2. if(av_find_stream_info(pFormatCtx)<0)
3.     return -1; // Couldn't find stream information
```

이 함수는 pFormatCtx->streams 에 고유 정보를 넣어줍니다.

다음은 다루기 쉬운 디버깅함수를 소개합니다.

```
1. // Dump information about file onto standard error
2. dump_format(pFormatCtx, 0, argv[1], 0);
```

pFormatCtx->nb_streams 는 파일에 있는 스트림의 개수 이고
pFormatCtx->streams 는 여기 저장된 각 스트림의 스트림 데이터가 들어있습니다.
우리는 이것을 통해서 video stream 을 찾아야 해요.

```
1. int i;
2. AVCodecContext *pCodecCtx;
3.
4. // Find the first video stream
5. videoStream=-1;
6. for(i=0; i<pFormatCtx->nb_streams; i++)
7.     if(pFormatCtx->streams[i]->codec->codec_type==CODEC_TYPE_VIDEO) {
8.         videoStream=i;
9.         break;
10.    }
11. if(videoStream==-1)
12.     return -1; // Didn't find a video stream
13.
```

```

14. // Get a pointer to the codec context for the video stream
15. pCodecCtx=pFormatCtx->streams[videoStream]->codec;

```

코덱에 관한 스트림 정보를 codec context 라고 부릅니다. 이것은 코덱에 관한 모든 정보를 갖고 있습니다. 우리는 그 스트림을 사용하고 있고 그것을 가리키는 포인터도 갖고 있습니다. 하지만 여전히 우리는 실제 코덱을 찾고 그것을 열어야 합니다.

```

1. AVCodec *pCodec;
2.
3. // Find the decoder for the video stream
4. pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
5. if(pCodec==NULL) {
6.     fprintf(stderr, "Unsupported codec!\n");
7.     return -1; // Codec not found
8. }
9. // Open codec
10. if(avcodec_open(pCodecCtx, pCodec)<0)
11.     return -1; // Could not open codec

```

Storing the Data

이젠 frame 을 저장할 공간이 필요합니다

```

1. AVFrame *pFrame;
2.
3. // Allocate video frame
4. pFrame=avcodec_alloc_frame();

```

우리는 PPM 파일로 출력해야 하고, 24-bit RGB 로 저장되어야 하기 때문에, native format 에서 RGB 로 프레임을 변환해야 합니다. 이 작업은 ffmpeg 가 해줄겁니다. ^_^
대부분의 프로젝트에서는 초기 프레임을 특정 포맷으로 변환해야 할것입니다.
자~ 이제 변환된 프레임으로 프레임을 할당해 봅시다.

```

1. // Allocate an AVFrame structure
2. pFrameRGB=avcodec_alloc_frame();
3. if(pFrameRGB==NULL)
4.     return -1;

```

프레임을 할당했다라도 우리는 변환시에 Raw Data 를 저장할 장소가 필요합니다. 이때 필요한 싸이즈를 얻고 수동으로 장소를 할당하기 위해 [avpicture_get_size](#) 를 사용합니다.

```

1. uint8_t *buffer;
2. int numBytes;
3. // Determine required buffer size and allocate buffer

```

```

4. numBytes=avpicture_get_size(PIX_FMT_RGB24, pCodecCtx->width,
5.                               pCodecCtx->height);
6. buffer=(uint8_t *)av_malloc(numBytes*sizeof(uint8_t));

```

[av_malloc\(\)](#) 은 ffmpeg 에서 쓰는 malloc 인데 이걸 malloc 을 간단히 wrapper 한것이라 기능역시 유사합니다.

av_malloc()은 메모리 릭 이나 double freeing, 또는 다른 malloc 문제 로부터 여러분을 보호해 줄수는 없습니다.

이제 [avpicture_fill\(\)](#)을 이용해서 Frame 과 우리가 새로 할당한 버퍼를 연결합니다.

[AVPicture](#) cast : AVPicture struct 는 [AVFrame](#) struct 의 작은 덩어리입니다. - AVFrame struct 의 시작부분은 AVPicture struct 와 동일합니다.

```

1. // Assign appropriate parts of buffer to image planes in pFrameRGB
2. // Note that pFrameRGB is an AVFrame, but AVFrame is a superset
3. // of AVPicture
4. avpicture_fill((AVPicture *)pFrameRGB, buffer, PIX_FMT_RGB24,
5.               pCodecCtx->width, pCodecCtx->height);

```

드디어! stream 을 읽을 준비가 다 되었군요~!

Reading the Data

패킷에서 읽은것을 전체 비디오스트림에 걸쳐 읽어 이것을 디코딩해서 프레임에 넣으면 하나의 프레임이 완성되고 그것을 변환하고 저장할 것입니다.

```

1. int frameFinished;
2. AVPacket packet;
3.
4. i=0;
5. while(av_read_frame(pFormatCtx, &packet)>=0) {
6.     // Is this a packet from the video stream?
7.     if(packet.stream_index==videoStream) {
8.         // Decode video frame
9.         avcodec_decode_video(pCodecCtx, pFrame, &frameFinished,
10.                             packet.data, packet.size);
11.
12.         // Did we get a video frame?
13.         if(frameFinished) {
14.             // Convert the image from its native format to RGB
15.             img_convert((AVPicture *)pFrameRGB, PIX_FMT_RGB24,
16.                       (AVPicture*)pFrame, pCodecCtx->pix_fmt,
17.                       pCodecCtx->width, pCodecCtx->height);

```

```

18.
19.     // Save the frame to disk
20.     if(++i<=5)
21.         SaveFrame(pFrameRGB, pCodecCtx->width,
22.                 pCodecCtx->height, i);
23.     }
24. }
25.
26. // Free the packet that was allocated by av_read_frame
27. av_free_packet(&packet);
28. }

```

이 반복문은 간단합니다.

[av_read_frame\(\)](#)는 패킷에서 읽어와서 [AVPacket](#) struct 에 저장합니다.

우리는 단지 packet structure 만 할당하면 됩니다. 그러면 ffmpeg 가 packet.data 에 의해 지정된 내부데이터를 할당해줍니다.

이것은 다음에 [av_free_packet\(\)](#)함수로 메모리 해제를 할수 있습니다.

[avcodec_decode_video\(\)](#)는 패킷을 프레임으로 변환합니다. 그러나 디코딩 된 프레임을 위한 모든 정보를 다 갖고 있지는 못합니다. 그래서 avcodec_decode_video() 함수는 다음 프레임으로 갈때 frameFinished 을 설정해줍니다.

드디어!! [img_convert\(\)](#) 함수를 사용하여 native format(pCodecCtx->pix_fmt)를 RGB 로 변환 합니다.

여러분은 [AVFrame](#) pointer 를 [AVPicture](#) pointer 로 할당할 수 있다는 것을 기억하세요.

우리는 이제 frame, width, height 정보를 SaveFrame 함수로 보냅니다.

SaveFrame 함수는 RGB 정보를 PPM 포맷의 파일에 기록합니다.

이제 개략적인 PPM 포맷을 설계해 봅시다.

```

1. void SaveFrame(AVFrame *pFrame, int width, int height, int iFrame) {
2.     FILE *pFile;
3.     char szFilename[32];
4.     int y;
5.
6.     // Open file
7.     sprintf(szFilename, "frame%d.ppm", iFrame);
8.     pFile=fopen(szFilename, "wb");
9.     if(pFile==NULL)
10.        return;
11.
12.    // Write header
13.    fprintf(pFile, "P6\n%d %d\n255\n", width, height);
14.
15.    // Write pixel data
16.    for(y=0; y<height; y++)

```

```

17.     fwrite(pFrame->data[0]+y*pFrame->linesize[0], 1, width*3, pFile);
18.
19.     // Close file
20.     fclose(pFile);
21. }

```

일반적 파일 열기처럼 파일을 열고 RGB 데이터를 기록합니다. 우리는 한번에 한줄씩 파일에 쓰기로 합니다. PPM 파일은 RGB 정보를 긴 문자열로 늘어놓은 간단한 파일입니다. 만일 여러분이 HTML 색상코드를 아신다면, 이것은 끝에서 끝까지, 빨간 화면이라면 #ff0000#ff0000.... 이런식으로 각각의 픽셀의 컬러값을 늘어놓은 것과 비슷하다고 보시면 됩니다. (이 것들은 binary 로 저장되고 연속되지 않습니다.) Header 는 이미지의 넓이, 높이와 RGB 값의 최대 싸이즈를 가리킵니다.

이제 main()함수로 돌아가볼까요?

일단 우린 비디오 스트림으로 부터 읽기를 완료했습니다.

그리고 메모리를 해제해 줍니다.

```

1.     // Free the RGB image
2.     av_free(buffer);
3.     av_free(pFrameRGB);
4.
5.     // Free the YUV frame
6.     av_free(pFrame);
7.
8.     // Close the codec
9.     avcodec_close(pCodecCtx);
10.
11.    // Close the video file
12.    av_close_input_file(pFormatCtx);
13.
14.    return 0;

```

여러분은 avcode_alloc_frame 과 [av_malloc](#) 으로 할당했었던 메모리를 해제하기 위해서 [av_free](#) 를 써야한다고 말씀하시겠지요?

이게 바로 그 코드 입니다.

이제 Linux 에서 다음을 실행시켜 봅시다.

```
gcc -o tutorial01 tutorial01.c -lavutil -lavformat -lavcodec -lz -lavutil -lm
```

ffmpeg 가 구버전이라면 -lavutil 은 빼고

```
gcc -o tutorial01 tutorial01.c -lavformat -lavcodec -lz -lm
```

이렇게 실행시켜 보세요

대부분의 이미지 프로그램은 PPM 파일을 열수 있을 겁니다.

몇개의 테스트용 동영상 파일로 한번 테스트 해보시기 바랍니다.

Tutorial 02: Outputting to the Screen

SDL and Video

우리는 스크린에 그림을 그리기 위해서 SDL 을 사용할 것입니다.

SDL 은 Simple Direct Layer 표준이고 멀티미디어, Cross-Platform 등을 위한 훌륭한 라이브러리 입니다.

SDL [the official website](#) 에서 개발 패키지를 다운 받으실 수 있습니다.

이번 Tutorial 을 컴파일 하기 위해서는 SDL 라이브러리가 꼭 있어야 하니까 설치해 두셔야 합니다.

SDL 은 스크린에 그림을 그리기 위한 많은 메소드를 갖고 있고, 이것으로 비디오를 출력할 수도 있습니다.

이걸 YUV 오버레이 라고 부릅니다.

[YUV \(technically not YUV but YCbCr\)](#) 는 RGB 같은 RAW 이미지 데이터를 저장하는 방법입니다.

개략적으로 말하면 Y 는 밝기, U 와 V 는 색상을 나타냅니다. (이것은 RGB 보다 좀더 복잡한데 몇몇 색갈정보는 버려지고, Y 요소 2 개당 U,V 요소는 1 개씩 들어가기 때문입니다.) SDL 의 YUV 오버레이는 YUV 데이터의 RAW 배열을 갖고 있고 이것을 디스플레이 합니다. 4 가지 종류의 YUV 포맷이 사용 가능하지만 YV12 가 가장 빠릅니다. YUV420P 이라는 다른 YUV 포맷이 있는데 이것은 YV12 에서 U 랑 V 가 바뀐것 빼고는 똑같습니다.

[Subsample](#) 된 420 이라는 것은 4:2:0 의 비율을 나타내는데 Y 성분이 4 개면 U,V 성분은 1 개가 있다는 뜻입니다.

이런 방법은 bandwidth 를 저장하는데 매우 효율적입니다. 사람의 눈은 이런 변환을 잘 구분하지 못하기 때문이지요.

이름에 있는 P 는 포맷이 planar 라는 뜻입니다. -이것은 YUV 요소가 연속적 배열이라는 의미 입니다.

ffmpeg 는 이미지를 YUV420P 로 변환할 수 있고, 그것을 다른 비디오 포맷으로 쉽게 변환 할 수 있습니다.

이제 우리는 Tutorial1 의 SaveFrame() 함수를 대체하고, 출력을 우리 화면에 해보도록 합시다.

아! 그전에 SDL Library 를 어떻게 사용해야 하는지를 먼저 봐야 겠군요

일단 라이브러리를 Include 하고 SDL 을 초기화해볼까요?

```
1. #include <SDL.h>
2. #include <SDL_thread.h>
3.
4. if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)) {
5.     fprintf(stderr, "Could not initialize SDL - %s\n", SDL_GetError());
6.     exit(1);
7. }
```

[SDL_Init\(\)](#)는 우리가 SDL 을 사용하려고 할때 꼭 써야하는 함수 입니다.

[SDL_GetError\(\)](#)는 디버깅 함수이구요

Creating a Display

이젠 이미지를 디스플레이 할곳이 필요한데 이 SDL 에서는 이 장소를 Surface 라고 부릅니다.

```
1. SDL_Surface *screen;
2.
3. screen = SDL_SetVideoMode(pCodecCtx->width, pCodecCtx->height, 0, 0);
4. if(!screen) {
5.     fprintf(stderr, "SDL: could not set video mode - exiting\n");
6.     exit(1);
7. }
```

화면의 가로, 세로를 설정합니다. 그 다음 옵션은 스크린의 depth 인데요 0 으로 하면 현재 디스플레이와 같은 depth 로 설정하는 것입니다.

이번엔 우리가 만든 screen 에 YUV overlay 를 만들어 video 를 올릴것입니다.

```
1. SDL_Overlay *bmp;
2.
3. bmp = SDL_CreateYUVOverlay(pCodecCtx->width, pCodecCtx->height,
4.                             SDL_YV12_OVERLAY, screen);
```

가로, 세로, YV12 를, screen 에~

Displaying the Image

이젠 이미지를 디스플레이 하기만 하면 됩니다. 마지막 프레임까지 주욱~

우리는 RGB 프레임을 위해 갖고 있던것들을 제거할 수 있습니다. 그리고 SaveFrame()과 디스플레이 코드를 바꿀것입니다.

이미지를 디스플레이 하기 위해서 [AVPicture](#) struct 를 만들고 TUV 에 쓸 데이터 포인터와 라인사이즈를 설정합니다.

```
1. if(frameFinished) {
2.     SDL_LockYUVOverlay(bmp);
3.
4.     AVPicture pict;
5.     pict.data[0] = bmp->pixels[0];
6.     pict.data[1] = bmp->pixels[2];
7.     pict.data[2] = bmp->pixels[1];
8.
9.     pict.linesize[0] = bmp->pitch[0];
10.    pict.linesize[1] = bmp->pitch[2];
11.    pict.linesize[2] = bmp->pitch[1];
```

```

12.
13. // Convert the image into YUV format that SDL uses
14. img_convert(&pict, PIX_FMT_YUV420P,
15.             (AVPicture *)pFrame, pCodecCtx->pix_fmt,
16.             pCodecCtx->width, pCodecCtx->height);
17.
18. SDL_UnlockYUVOverlay(bmp);
19. }

```

overlay 를 써야하기 때문에 일단 lock 을 걸어줍니다. lock 을 거는것을 습관하 하세요 그래야 나중에 뒷탈이 없습니다.

전에 봤던 [AVPicture](#) struct 는 하나의 data pointer 를 갖고 있는데 이것은 4 개의 pointer 의 array 입니다. YUV420P 를 다루기 때문에 우리는 3 개의 channel 을 갖고 있고 그래서 4 개중에 3 개만 세팅 합니다. 다른 포맷들은 아마 Alpha channel 까지 해서 네개의 포인터를 갖고 있을지도 모릅니다.

linesize 는 사운드 같은것 입니다.

YUV overlay 에 있는 비슷한 구조들은 다양한 pixel 과 pitch 입니다. ("pitches" 는 주어진 데이터 라인의 넓이를 알기 위해 사용되는 SDL 용어 입니다.)

pict 에 쓸때 overlay 에 pict.data 의 세개 배열을 지정한다. 우리는 사실 overlay 에 기록하는 것입니다. 비슷하게, overlay 로부터 직접 linesize 정보를 얻기도 합니다. 바꾼 포맷을 PIX_FMT_YUV420P 로 바꾸고, 방금 것처럼 [img_convert](#) 를 사용합니다.

Drawing the Image

이제 그리면 됩니다. 이 함수에는 비디오의 가로, 세로 사이즈를 지정하면 크기를 변환해준다. SDL 은 graphic processor 를 이용해 빠른 Scaling 을 해줍니다.

```

1.  SDL_Rect rect;
2.
3.  if(frameFinished) {
4.      /* ... code ... */
5.      // Convert the image into YUV format that SDL uses
6.      img_convert(&pict, PIX_FMT_YUV420P,
7.                 (AVPicture *)pFrame, pCodecCtx->pix_fmt,
8.                 pCodecCtx->width, pCodecCtx->height);
9.
10.     SDL_UnlockYUVOverlay(bmp);
11.     rect.x = 0;
12.     rect.y = 0;
13.     rect.w = pCodecCtx->width;
14.     rect.h = pCodecCtx->height;
15.     SDL_DisplayYUVOverlay(bmp, &rect);

```

```
16. }
```

Now our video is displayed!

이번에는 SDL 의 다른 기능을 살펴봅시다.

SDL 은 event system 입니다. 여러분이 버튼을 누르거나 마우스를 움직이거나 어떤 Signal 을 줄때 event 가 일어납니다.

여러분의 프로그램에서 이 event 들을 감지할 수 있습니다. 그리고 여러분의 프로그램에서 event 를 발생시킬 수도 있습니다.

이런 기능은 SDL 로 Multithread 프로그램을 돌릴때 유용하게 쓸수 있습니다.

지금 이 프로그램에서는 packet processing 이 끝나자마자 이벤트를 감시할 것입니다.

SDL_QUIT 이벤트를 이용해서 우리 프로그램을 종료 하도록 합시다.

```
1.  SDL_Event    event;
2.
3.  av_free_packet(&packet);
4.  SDL_PollEvent(&event);
5.  switch(event.type) {
6.  case SDL_QUIT:
7.      SDL_Quit();
8.      exit(0);
9.      break;
10. default:
11.     break;
12. }
```

이제 컴파일을 해봅시다.

여러분이 리눅스를 쓰신다면 다음이 SDL lib 을 사용한 가장 좋은 컴파일 방법입니다.

```
gcc -o tutorial02 tutorial02.c -lavutil -lavformat -lavcodec -lz -lm `
`sdl-config --cflags --libs`
```

sdl-config 는 gcc 에다가 SDL libraries 를 include 할수 있는 적당한 flag 를 출력해 줍니다.

여러분이 만약 여러분 시스템의 컴파일을 위해 약간 다른 설정을 해야 한다면 SDL Documentation 을 참고 하세요.

자! 이제 컴파일을 하고 실행을 시켜 봅시다.

어떤일이 일어나나요? 비디오가 미친거 같죠? ㅋㅋ 사실 우리는 비디오 파일에서 추출할 수 있는 한 최대속도로 비디오는 디스플레이 합니다. 우리는 지금 언제 비디오를 디스플레이 해야하는지를 coding 하지 않았어요. sync 가 전혀 안되고 있습니다.

우리는 video syncing 을 해야하고요.... 아차! 일단 Sound 플레이를 해야겠군요!

Tutorial 03 : Play Sound

Audio

이제 우리는 Sound 재생을 해야할 차례 입니다.

SDL 은 사운드 출력 메소드도 제공해줍니다. [SDL_OpenAudio\(\)](#)함수로 사운드 장치를 이용할 수 있습니다. 이 함수는 [SDL_AudioSpec](#) struct 를 아규먼트로 받는데 여기에는 우리가 출력할 모든 오디오 정보를 담고 있습니다.

이것을 어떻게 쓰는지 보기전에 컴퓨터가 오디오를 어떻게 제어하는지를 알아볼까요?

디지털 오디오는 샘플들의 긴 스트림으로 이루어져 있습니다. 각각의 샘플은 audio [waveform](#) 의 값을 나타냅니다.

Sound 는 sample rate 들의 기록입니다. 간단히 말하면 각각의 샘플을 얼마나 빨리 재생할지, 그리고 초당 샘플수가 측정된 것 입니다. 예를들면 sample rates 가 22,050 그리고 44,100 samples per second 같은것들은 라디오나 CD 에서 쓰이는 rate 들 입니다. 추가적으로 대부분의 오디오들은 surround 나 stereo 를 위해 한개 이상의 채널을 갖을수 있습니다. stereo sample 같은 경우에는 한번에 2 개의 sample 이 재생될것입니다. 우리가 비디오파일에서 데이터를 받을때, 얼마나 많은 샘플을 얻게될지 모릅니다. 하지만 ffmpeg 는 부분적인 샘플을 주지 않을것입니다. - 이것은 stereo sample 을 쪼갤수 없다는 것을 의미하기도 합니다.

SDL 의 오디오 재생 메소드 : 여러분은 여러분의 오디오 옵션을 지정해야 합니다. : sample rate(SDL structure 에서 frequencyt 를 freq 라고 부릅니다.), 채널 수, 콜백함수, 사용자데이터 들을 세팅합니다. 오디오를 플레이하기 시작할때 SDL 은 콜백 함수를 연속적으로 호출하고 오디오 버퍼와 바이트 수를 채울것을 요청할것입니다. 우리는 [SDL_AudioSpec](#) struct 의 정보를 입력한 다음에 [SDL_OpenAudio\(\)](#)함수를 호출합니다. 그러면 오디오 디바이스를 열고 AudioSpec struct 을 돌려줄것입니다. 여기서 나온 spec 들이 실제 사용하게 될 것입니다.

Setting Up the Audio

우리는 지금 정신이 몽롱 합니다.

우리는 아직 오디오 스트림에 대한 아무런 정보도 없거든요.

자~ 소스코드로 가서 우리가 비디오 스트림을 찾았던 부분에서 오디오 스트림도 찾아봅시다.

```
1. // Find the first video stream
2. videoStream=-1;
3. audioStream=-1;
4. for(i=0; i < pFormatCtx->nb_streams; i++) {
5.     if(pFormatCtx->streams[i]->codec->codec_type==CODEC_TYPE_VIDEO
6.         &&
7.         videoStream < 0) {
8.         videoStream=i;
9.     }
10.    if(pFormatCtx->streams[i]->codec->codec_type==CODEC_TYPE_AUDIO &&
11.        audioStream < 0) {
```

```

12.  audioStream=i;
13.  }
14. }
15. if(videoStream==--1)
16.     return -1; // Didn't find a video stream
17. if(audioStream==--1)
18.     return -1;

```

stream 의 [AVCodecContext](#) 에서 우리가 원하는 모든 정보를 얻을 수 있습니다.
 그냥 비디오 스트림에서 우리가 했던대로 하면 됩니다. 참~ 쉽죠잉~?

```

1.  AVCodecContext *aCodecCtx;
2.
3.  aCodecCtx=pFormatCtx->streams[audioStream]->codec;

```

다음 codec context 에 우리가 오디오 set up 에 필요한 모든 정보가 다 들어있습니다.

```

1.  wanted_spec.freq = aCodecCtx->sample_rate;
2.  wanted_spec.format = AUDIO_S16SYS;
3.  wanted_spec.channels = aCodecCtx->channels;
4.  wanted_spec.silence = 0;
5.  wanted_spec.samples = SDL_AUDIO_BUFFER_SIZE;
6.  wanted_spec.callback = audio_callback;
7.  wanted_spec.userdata = aCodecCtx;
8.
9.  if(SDL_OpenAudio(&wanted_spec, &spec) < 0) {
10.     fprintf(stderr, "SDL_OpenAudio: %s\n", SDL_GetError());
11.     return -1;
12. }

```

다음을 볼까요??

- freq : 앞에서 설명한바와 같이 sample rate 입니다.
- format : SDL 에 주어야할 포맷입니다. S16SYS 에서 S 는 "signed"를 의미하고 16 은 각각의 샘플이 16bit 이라는것을 의미합니다. 그리고 SYS 는 시스템의 endian-order 를 나타냅니다.
[avcodec_decode_audio2](#) 를 사용하면 오디오 in 의 포맷을 우리에게 알려줍니다.
- channels : 오디오 채널의 수.
- silence : silence 를 지정한 값. 오디오가 signed 이기 때문에 0 이 주로 쓰인다.
- samples : SDL 이 추가 오디오 데이터를 요청할 때를 대비한 오디오 버퍼의 사이즈. 512~ 8192 의 값을 사용하는것이 좋고 ffmpeg 에서는 1024 를 사용합니다.
- callback : 여기서 callback 함수는 건너뛴니다. 다음에 callback 함수에 대해서 알아보도록 합니다.
- userdata : SDL 은 callback 에게 어떤 user data 를 가리키고 있는 void pointer 를 돌려줍니다.

[SDL_OpenAudio](#) 으로 오디오를 열어봅시다.

여러분은 여러분의 오디오 데이터를 혼합하고 그것을 오디오 스트림으로 보내는 콜백 함수를 만들어야 합니다.. 그 다음엔, 원하는 오디오 포맷과 속도를 선택하고, 오디오 장치를 엽니다. `SDL_PauseAudio(0)` 를 호출할때까지 오디오는 실제로 재생되지 않습니다. 이 함수는 여러분의 콜백 함수가 실행되기 전에, (필요하다면) 다른 오디오 초기화작업을 수행할 수 있도록 합니다. 사운드 출력을 마친 다음, `SDL_CloseAudio()` 함수를 사용해 오디오 출력을 닫아야 합니다.

팁:

만약 여러분의 애플리케이션이 다른 오디오 포맷을 처리할 수 있다면, 두번째 `SDL_AudioSpec` 포인터를 `SDL_OpenAudio()` 에 보내 실제 하드웨어 오디오 포맷을 취하도록 해야 한다. 만약 두번째 포인터를 `NULL` 로 남겨놓는다면, 오디오 데이터는 실행시에 하드웨어 오디오 포맷으로 변환될 것이다.

전에 했던 tutorial 을 기억하신다면 오디오 코덱을 스스로 열어야할 필요가 있다는 것을 아실것입니다. 다음처럼 하면 간단합니다.

```
1. AVCodec      *aCodec;
2.
3. aCodec = avcodec_find_decoder(aCodecCtx->codec_id);
4. if(!aCodec) {
5.     fprintf(stderr, "Unsupported codec!\n");
6.     return -1;
7. }
8. avcodec_open(aCodecCtx, aCodec);
```

Queues

이제 스트림에서 오디오 정보를 얻을 준비가 다 되었습니다. 하지만 이 정보로 뭘해야 할까요? 우리는 무비파일에서 계속 패킷을 얻을 것인데 같은 시간 SDL 은 callback 함수를 호출할 것입니다. 이 방법은 어떤 종류의 전역 구조체를 생성하는 방법입니다. 우리는 여기에 오디오로부터 얻은 오디오 패킷을 채울수 있습니다. 그래서 우리는 패킷의 queue 를 만들것인데 `ffmpeg` 는 이것을 위해 [AVPacketList](#) 를 지원해 줍니다. 이것은 패킷을 위한 linked list 입니다. queue 의 구조는 다음과 같습니다.

```
1. typedef struct PacketQueue {
2.     AVPacketList *first_pkt, *last_pkt;
3.     int nb_packets;
4.     int size;
5.     SDL_mutex *mutex;
6.     SDL_cond *cond;
7. } PacketQueue;
```

우선 `nb_packets` 를 지정해야 하는데 이것은 `size` 같은것이 아닙니다. `size` 는 `packet->size` 로 부터 얻은 1byte size 를 참조합니다. 여러분은 여기서 [mutex](#) 와 [condition variable](#) 를 선언해줘야 합니다. SDL 은 audio process 를 분리된 Thread 처럼 실행하고 있기 때문이죠. 만약 여러분이 queue 에 락을 정확히 걸지 않으면

여러분의 데이터는 엉망진창이 될 것입니다. 이젠 queue 의 실행을 보겠습니다.

우선 queue 를 초기화하는 함수를 만들어 보겠습니다.

```
1. void packet_queue_init(PacketQueue *q) {
2.     memset(q, 0, sizeof(PacketQueue));
3.     q->mutex = SDL_CreateMutex();
4.     q->cond = SDL_CreateCond();
5. }
```

이번에는 만들어진 queue 에 데이터를 채우는 함수를 만들겠습니다.

```
1. int packet_queue_put(PacketQueue *q, AVPacket *pkt) {
2.
3.     AVPacketList *pkt1;
4.     if(av_dup_packet(pkt) < 0) {
5.         return -1;
6.     }
7.     pkt1 = av_malloc(sizeof(AVPacketList));
8.     if (!pkt1)
9.         return -1;
10.    pkt1->pkt = *pkt;
11.    pkt1->next = NULL;
12.
13.
14.    SDL_LockMutex(q->mutex);
15.
16.    if (!q->last_pkt)
17.        q->first_pkt = pkt1;
18.    else
19.        q->last_pkt->next = pkt1;
20.    q->last_pkt = pkt1;
21.    q->nb_packets++;
22.    q->size += pkt1->pkt.size;
23.    SDL_CondSignal(q->cond);
24.
25.    SDL_UnlockMutex(q->mutex);
26.    return 0;
27. }
```

[SDL_LockMutex\(\)](#)를 이용해서 queue 에 mutex 를 잠그고 데이터를 추가합니다. 그리고 [SDL_CondSignal\(\)](#)로 get 함수에 signal 을 보냅니다. 그런 다음 mutex 를 unlock 합니다.

다음은 `get` 함수에 대응하는 내용입니다.

어떻게 `SDL_CondWait()` 함수가 Block 을 걸게 되는지 생각해 보세요.

```
1.  int quit = 0;
2.
3.  static int
4.  packet_queue_get(PacketQueue * q, AVPacket * pkt, int block)
5.  {
6.      AVPacketList *pkt1;
7.
8.      int      ret;
9.
10.     SDL_LockMutex(q->mutex);
11.
12.     for (;;)
13.     {
14.
15.         if (quit)
16.         {
17.             ret = -1;
18.             break;
19.         }
20.
21.         pkt1 = q->first_pkt;
22.         if (pkt1)
23.         {
24.             q->first_pkt = pkt1->next;
25.             if (!q->first_pkt)
26.                 q->last_pkt = NULL;
27.             q->nb_packets--;
28.             q->size -= pkt1->pkt.size;
29.             *pkt = pkt1->pkt;
30.             av_free(pkt1);
31.             ret = 1;
32.             break;
33.         }
34.         else if (!block)
35.         {
36.             ret = 0;
37.             break;
38.         }
39.         else
```

```

40.     {
41.         SDL_CondWait(q->cond, q->mutex);
42.     }
43. }
44. SDL_UnlockMutex(q->mutex);
45. return ret;

```

보시다시피 무한루프로 함수를 돌려췄습니다. 그래서 우리가 블록을 얻길 원한다면 데이터를 확실히 얻을 수 있을 것입니다. 무한루프가 일어나는 것을 막으려면 SDL의 [SDL_CondWait\(\)](#) 함수를 사용하게 합니다. 기본적으로 SDL_CondWait는 제공된 mutex를 unlock하고 다른 스레드에서 [SDL_CondSignal\(\)](#) 또는 [SDL_CondBroadcast\(\)](#)를 호출한 signal을 기다렸다가 다시 lock을 걸어 진행합니다. 그러나 이것은 mutex로 잡혀있습니다. 만약 lock이 걸려있다면 queue에 어떠한 것도 넣을 수 없습니다.

In Case of Fire

우리는 quit 전역변수를 갖고 있는데 이것은 프로그램 종료 signal이 발생하지 않았다는 것을 확실하게 체크해 줍니다. (SDL은 자동으로 TERM signal 같은 것을 제어합니다.)

만약 그렇지 않으면 무한루프에 빠져서 우리가 kill -9로 프로그램을 죽여야 할 것입니다. 만약 몇몇 blocking 함수를 종료할 필요성이 있다면 ffmpeg는 callback을 체크하고 볼 수 있는 함수를 제공합니다.:

[url_set_interrupt_cb](#)

```

1. int decode_interrupt_cb(void) {
2.     return quit;
3. }
4. ...
5. main() {
6.     ...
7.     url_set_interrupt_cb(decode_interrupt_cb);
8.     ...
9.     SDL_PollEvent(&event);
10.    switch(event.type) {
11.        case SDL_QUIT:
12.            quit = 1;
13.    ...

```

이것은 ffmpeg 함수만을 지원합니다. 당연히 SDL의 함수는 안됩니다.

우리는 quit flag를 반드시 1로 세팅해야 합니다.

Feeding Packets

queue 셋업이 하나 남았군요

```

1. PacketQueue audioq;
2. main() {

```

```

3. ...
4. avcodec_open(aCodecCtx, aCodec);
5.
6. packet_queue_init(&audioq);
7. SDL_PauseAudio(0);

```

[SDL_PauseAudio\(\)](#) 로 오디오 디바이스를 구동합니다. 데이터를 얻지 못하면 아무소리도 안납니다.

우리는 queue 셋업을 해놨기 때문에 이제 packet 을 feeding 할 준비가 다 되었습니다.
이제 패킷을 읽어오는 루프를 실행 합니다.

```

1. while(av_read_frame(pFormatCtx, &packet)>=0) {
2.     // Is this a packet from the video stream?
3.     if(packet.stream_index==videoStream) {
4.         // Decode video frame
5.         ....
6.     }
7. } else if(packet.stream_index==audioStream) {
8.     packet_queue_put(&audioq, &packet);
9. } else {
10.     av_free_packet(&packet);
11. }

```

패킷을 큐에 넣은 다음에 메모리 해제를 하지 않습니다. 다음에 디코드 할때 해제할 것입니다.

Fetching Packets

이제 queue 의 패킷에 붙일 audio_callback 함수를 만들어 봅시다.

callback 함수는 void callback(void *userdata, Uint8 *stream, int len)의 형태여야 합니다. , userdata 는 물론 우리가 SDL 에 준 포인터 입니다. 스트림은 우리가 쓰고있는 오디오 데이터의 버퍼 이고, len 은 버퍼의 사이즈 입니다.

다음 코드를 보시죠

```

1. void audio_callback(void *userdata, Uint8 *stream, int len) {
2.
3.     AVCodecContext *aCodecCtx = (AVCodecContext *)userdata;
4.     int len1, audio_size;
5.
6.     static uint8_t audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE * 3) / 2];
7.     static unsigned int audio_buf_size = 0;
8.     static unsigned int audio_buf_index = 0;
9.
10.    while(len > 0) {

```

```

11.  if(audio_buf_index >= audio_buf_size) {
12.      /* We have already sent all our data; get more */
13.      audio_size = audio_decode_frame(aCodecCtx, audio_buf,
14.                                     sizeof(audio_buf));
15.      if(audio_size < 0) {
16.          /* If error, output silence */
17.          audio_buf_size = 1024;
18.          memset(audio_buf, 0, audio_buf_size);
19.      } else {
20.          audio_buf_size = audio_size;
21.      }
22.      audio_buf_index = 0;
23.  }
24.  len1 = audio_buf_size - audio_buf_index;
25.  if(len1 > len)
26.      len1 = len;
27.  memcpy(stream, (uint8_t *)audio_buf + audio_buf_index, len1);
28.  len -= len1;
29.  stream += len1;
30.  audio_buf_index += len1;
31.  }
32.  }

```

이것은 기본적인 다른 함수로부터 데이터를 끌어오는 간단한 루프입니다. 우리는 쓰고, `audio_decode_frame()`를 실행하고 결과를 하나의 중계 버퍼에 저장합니다. 그리고 `len` 만큼의 byte 를 `stream` 에 쓰도록 시도합니다. 그리고 아직 충분하지 못하면 데이터를 더 가져오고 데이터가 남았다면 기다렸다가 저장합니다. `audio_buf` 의 사이즈는 가장 큰 audio frame 의 크기에 1.5 배로 설정합니다. `ffmpeg` 는 우리에게 괜찬은 쿠션을 제공해 줍니다.

Finally Decoding the Audio

다음은 디코더 소스 입니다.

```

1.  int audio_decode_frame(AVCodecContext *aCodecCtx, uint8_t *audio_buf,
2.                          int buf_size) {
3.
4.      static AVPacket pkt;
5.      static uint8_t *audio_pkt_data = NULL;
6.      static int audio_pkt_size = 0;
7.
8.      int len1, data_size;
9.
10.     for(;;) {
11.         while(audio_pkt_size > 0) {

```

```

12.     data_size = buf_size;
13.     len1 = avcodec_decode_audio2(aCodecCtx, (int16_t *)audio_buf, &data_size,
14.         audio_pkt_data, audio_pkt_size);
15.     if(len1 < 0) {
16.         /* if error, skip frame */
17.         audio_pkt_size = 0;
18.         break;
19.     }
20.     audio_pkt_data += len1;
21.     audio_pkt_size -= len1;
22.     if(data_size <= 0) {
23.         /* No data yet, get more frames */
24.         continue;
25.     }
26.     /* We have data, return it and come back for more later */
27.     return data_size;
28. }
29. if(pkt.data)
30.     av_free_packet(&pkt);
31.
32. if(quit) {
33.     return -1;
34. }
35.
36. if(packet_queue_get(&audioq, &pkt, 1) < 0) {
37.     return -1;
38. }
39. audio_pkt_data = pkt.data;
40. audio_pkt_size = pkt.size;
41. }
42. }

```

모든 처리는 함수의 끝 부분에서 시작합니다. call `packet_queue_get()`를 호출한 부분. queue 에서 패킷을 집어와서 그 정보를 저장합니다. 그리고 한 패킷으로 `avcodec_decode_audio2()`를 호출합니다. 한 패킷은 한 프레임 이상을 갖고 있을 수도 있습니다. 그래서 여러분은 아마 패킷으로 부터 모든 데이터를 얻기 위해 이것을 여러번 호출해야 할지도 모릅니다.

또한 여러분은 SDL 이 8bit int buffer 를 주기 때문에 `audio_buf` 를 casting 해야 하는 것을 기억하셔야 합니다. 그리고 `ffmpeg` 는 우리에게 16bit int buffer 를 줍니다.

또한 `len1` 과 `data_size` 가 다르다는 것도 알고 있어야 합니다. `len1` 은 우리가 패킷을 얼마나 사용했는지를 나타내는 것이고 `data_size` 는 raw data 가 반환된 양을 나타냅니다.

우리가 데이터를 받았을 때, 우리가 여전히 queue 로 부터 데이터를 더 얻어야 한다면 작업이 완료 되었거나 즉시 리턴합니다. 우리가 처리할 패킷을 가지고 있다면 이것을 다음에 저장합니다. 만약 패킷 처리가 끝났다면 packet 을 해제 합니다.

이제 queue 로 루프를 돌면서 읽은 것을 모두 옮겼습니다. audio_callback 함수에 의해 읽어진 데이터를 SDL 이 여러분의 사운드 카드로 쓸 것입니다.

이제 컴파일을 해볼까요?

```
gcc -o tutorial03 tutorial03.c -lavutil -lavformat -lavcodec -lz -lm -W'sdl-config --cflags --libs`
```

얼라리~ 여전히 비디오는 후딱 지나가는군요. 하지만 오디오는 제대로 재생됩니다.

왜 그럴까요?

그것은 오디오 정보는 Sample rate 를 갖고 있기 때문입니다. - 우리는 가능한 빨리 오디오 정보를 뽑아내지만 오디오는 sample rate 에 따라 천천히 스트림으로 부터 재생 됩니다.

우리는 이제 비디오와 오디오의 싱크를 맞추면 됩니다. 그전에 우리는 프로그램을 약간 손봐야할 필요가 있습니다.

오디오를 큐에 넣는 메소드와 플레이 하는 것을 다른 쓰레드에서 하도록 분리해야 합니다. 이렇게 해야 유지보수 하기 쉽고, 더 모듈화 되기 때문입니다.

비디오와 오디오를 싱크하기 전에 우리의 코드를 더 다루기 쉽게 만들어야 합니다.

다음 Tutorial 에서는 쓰레드를 만들어 보겠습니다.!

Tutorial 04: Spawning Threads

Overview

지난번에는 SDL 의 오디오 기능을 이용해서 오디오 재생기능을 추가해봤었습니다.

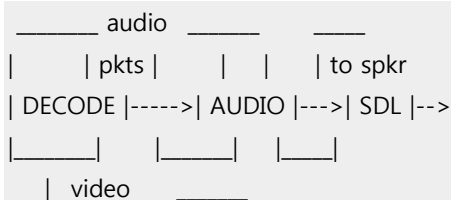
SDL 은 오디오가 필요할때 마다 콜백 함수를 정의한 쓰레드를 시작했었습니다. 이제 우리는 비디오 디스플레이 같은 종류의 작업을 할 것입니다. 지금 만들 코드는 Sync 를 적용할 때 더 작동하기 쉽게 모듈화 하는 코드 입니다.

자~ 그럼 어디서부터 시작할까요?

먼저 우리의 main 함수를 보면 완전 개판입니다. ㅋ

지금은 이벤트 루프, 패킷에서 읽어오기, 비디오 디코딩이 전부 다 들어있습니다. @_@ 그래서 이제 이것들을 다 분리해낼 것입니다. 우리는 패킷을 디코딩하는것을 담당하는 쓰레드를 만들것 입니다. 이 패킷들은 큐에 추가되고 오디오, 비디오 쓰레드에 맞게 읽어질 것입니다. 오디오 쓰레드는 이미 우리가 원하는 대로 셋업되어 있고 비디오 쓰레드는 비디오를 디스플레이 해야하기 때문에 조금더 복잡해질 것입니다. 이제 메인 루프에 실질적으로 비디오 디스플레이 하는 코드를 넣겠습니다. 그냥 루프돌면서 재생하는게 아니라 이벤트 루프에 따라서 비디오가 디스플레이 되도록 하겠습니다. 이 비디오 디코드 개념은 결과물 프레임을 다른 queue 에 넣는 것입니다. 그리고 FF_REFRESH_EVENT 라는 커스텀 이벤트를 만들어서 이벤트 시스템에 추가 합니다. 그리고 이벤트 루프에서 이벤트를 감지하면 큐에있는 다음 프레임을 디스플레이 합니다.

다음의 흐름도를 참고하세요



```

| pkts | |
+----->| VIDEO |
| | | |
| EVENT | +----->| VIDEO | to mon.
| LOOP |----->| DISP. |-->
| |<---FF_REFRESH---| |

```

[SDL_Delay](#) 쓰레드를 이용하면 스크린에 다음 비디오 프레임을 보여줄 때를 정확히 컨트롤 할 수 있습니다. 다음 튜토리얼에서는 비디오를 싱크 할때, 우리는 간단한 코드를 추가해서 다음 비디오로 refresh 할 것입니다. 그러면 화면이 제때 제때 보이게 될것지요.

Simplifying Code

우리는 오디오와 비디오 코덱 정보 전부를 갖고 있습니다. 그리고 큐와 버퍼에 추가하는것들을 알고 있습니다. 이 모든것이 하나의 logical unit 을 위한 것입니다. 바로 Movie 말이지요. 그래서 우리는 VideoState 라는 모든 정보를 담을수 있는 커다란 구조체를 정의합니다.

```

1. typedef struct VideoState {
2.
3.     AVFormatContext *pFormatCtx;
4.     int          videoStream, audioStream;
5.     AVStream     *audio_st;
6.     PacketQueue  audioq;
7.     uint8_t      audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE * 3) / 2];
8.     unsigned int  audio_buf_size;
9.     unsigned int  audio_buf_index;
10.    AVPacket      audio_pkt;
11.    uint8_t       *audio_pkt_data;
12.    int           audio_pkt_size;
13.    AVStream     *video_st;
14.    PacketQueue  videoq;
15.
16.    VideoPicture  pictq[VIDEO_PICTURE_QUEUE_SIZE];
17.    int           pictq_size, pictq_rindex, pictq_windex;
18.    SDL_mutex    *pictq_mutex;
19.    SDL_cond     *pictq_cond;
20.
21.    SDL_Thread    *parse_tid;
22.    SDL_Thread    *video_tid;
23.
24.    char          filename[1024];
25.    int           quit;
26. } VideoState;

```

여기서 잠깐 우리가 뭘 하려고 하는지 알아보까요?

우선 기본정보를 보겠습니다.

Format Context 와 오디오 비디오의 인덱스들 그리고 [AVStream](#) 오브젝트들을 보세요. 우리가 오디오 버퍼를 이 구조체에 넣었던 것을 알수가 있습니다.

audio_buf, audio_buf_size 같은 것들은 전부 오디오를 위한 정보입니다. 우리는 비디오를 위한 큐와 프레임 디코드를 위한 버퍼를 추가했었습니다. VideoPicture struct 는 우리가 만든것이구요. 별도의 두 쓰레드를 지정하는 포인터와 quit 플래그, 무비파일의 이름을 지정할 char 배열이 있습니다.

이제 main 함수로 돌아가서 우리 프로그램이 어떻게 바뀌었는지 살펴보겠습니다.

일단 VideoState 구조체를 설정 합니다.

```
1. int main(int argc, char *argv[]) {
2.
3.     SDL_Event     event;
4.
5.     VideoState    *is;
6.
7.     is = av_mallocz(sizeof(VideoState));
```

[av_mallocz\(\)](#) 함수는 메모리를 0 으로 채워서 할당해줍니다.

그리고 이벤트 루프에서 display 함수를 호출하기 때문에 디스플레이 버퍼(pictq)를 위한 lock 을 초기화 해줍니다.

다음에 우리는 qictq 에서 pre-decoded 프레임들 가져올 것입니다. 동시에 비디오 디코더는 qictq 에 정보를 넣을 것입니다. 우리는 어떤것이 먼저 실행될지 알수 없습니다. 이건 race condition 같은 것임을 알아보실 것입니다. 그래서 쓰레드가 시작하기 전에 이것을 할당하는 것입니다. 그리고 무비파일이름을 VideoState 에 카피해 넣습니다.

```
1. pstrcpy(is->filename, sizeof(is->filename), argv[1]);
2.
3. is->pictq_mutex = SDL_CreateMutex();
4. is->pictq_cond = SDL_CreateCond();
```

pstrcpy 는 ffmpeg 가 제공하는 함수 인데 strncpy 보다 정확합니다.

(저는 psyncpy 가 라이브러리에 정의되어 있지 않더군요. 그래서 인터넷에서 찾아다가 카피해 넣었습니다. by 호철)

```
1. /**
2.  * Copy the string str to buf. If str length is bigger than buf_size -
3.  * 1 then it is clamped to buf_size - 1.
4.  * NOTE: this function does what strncpy should have done to be
5.  * useful. NEVER use strncpy.
```

```

6.  *
7.  * @param buf destination buffer
8.  * @param buf_size size of destination buffer
9.  * @param str source string
10. */
11. void pstrcpy(char *buf, int buf_size, const char *str)
12. {
13.     int c;
14.     char *q = buf;
15.
16.     if (buf_size <= 0)
17.         return;
18.
19.     for(;;) {
20.         c = *str++;
21.         if (c == 0 || q >= buf + buf_size - 1)
22.             break;
23.         *q++ = c;
24.     }
25.     *q = '\0';
26. }

```

Our First Thread

이제 스레드를 돌려보겠습니다.

```

1.  schedule_refresh(is, 40);
2.
3.  is->parse_tid = SDL_CreateThread(decode_thread, is);
4.  if(!is->parse_tid) {
5.      av_free(is);
6.      return -1;
7.  }

```

schedule_refresh 함수는 다음에 정의하겠습니다.

이것은 기본적으로 지정된 millisecond 후에 FF_REFRESH_EVENT 를 시스템에 발생시키는 일을 합니다. 이것은 이벤트 큐에 이것이 나타나면 비디오 refresh 함수를 호출합니다.

[SDL_CreateThread\(\)](#)함수는 새로운 스레드를 만듭니다. 이것은 부모 프로세스의 모든 메모리에 접근 가능하고 이 함수를 호출하면서 실행되기 시작합니다. 위 예제에서는 decode_thread 에 VideoState 구조체를 붙여서 호출했습니다. main 함수의 반환 값은 바뀌었지 않습니다. 파일을 열고 오디오 비디오 스트림의 인덱스를 찾는 내용 그대로입니다. 한가지 바뀐 것은 format context 를 우리의 새로운 큰 구조체에 저장한다는 것 뿐이군요.

스트림 인덱스들을 찾은 다음 `Stream_component_open()` 이라는 함수를 정의해서 호출합니다. 이 함수는 자연적인 방법으로 이것들을 분리합니다. 그리고 비디오와 오디오 코덱의 설정에 비슷한 부분이 많기 때문에 이 함수를 재활용할 것입니다.

`stream_component_open()` 함수는 코덱 디코더에서 찾을 수 있는데 오디오 옵션을 설정하고 우리의 큰 구조체에 중요한 정보를 저장합니다. 그리고 우리의 오디오 비디오 쓰레드를 실행합니다. 이것 말고도 자동으로 코덱을 찾는 것 대신 강제로 설정하는 기능들 다른 여러 옵션을 추가할 수도 있습니다.

```
1. int stream_component_open(VideoState *is, int stream_index) {
2.
3.     AVFormatContext *pFormatCtx = is->pFormatCtx;
4.     AVCodecContext *codecCtx;
5.     AVCodec *codec;
6.     SDL_AudioSpec wanted_spec, spec;
7.
8.     if(stream_index < 0 || stream_index >= pFormatCtx->nb_streams) {
9.         return -1;
10.    }
11.
12.    // Get a pointer to the codec context for the video stream
13.    codecCtx = pFormatCtx->streams[stream_index]->codec;
14.
15.    if(codecCtx->codec_type == CODEC_TYPE_AUDIO) {
16.        // Set audio settings from codec info
17.        wanted_spec.freq = codecCtx->sample_rate;
18.        /* .... */
19.        wanted_spec.callback = audio_callback;
20.        wanted_spec.userdata = is;
21.
22.        if(SDL_OpenAudio(&wanted_spec, &spec) < 0) {
23.            fprintf(stderr, "SDL_OpenAudio: %s\n", SDL_GetError());
24.            return -1;
25.        }
26.    }
27.    codec = avcodec_find_decoder(codecCtx->codec_id);
28.    if(!codec || (avcodec_open(codecCtx, codec) < 0)) {
29.        fprintf(stderr, "Unsupported codec!\n");
30.        return -1;
31.    }
32.
33.    switch(codecCtx->codec_type) {
34.    case CODEC_TYPE_AUDIO:
35.        is->audioStream = stream_index;
```

```

36.  is->audio_st = pFormatCtx->streams[stream_index];
37.  is->audio_buf_size = 0;
38.  is->audio_buf_index = 0;
39.  memset(&is->audio_pkt, 0, sizeof(is->audio_pkt));
40.  packet_queue_init(&is->audioq);
41.  SDL_PauseAudio(0);
42.  break;
43.  case CODEC_TYPE_VIDEO:
44.  is->videoStream = stream_index;
45.  is->video_st = pFormatCtx->streams[stream_index];
46.
47.  packet_queue_init(&is->videoq);
48.  is->video_tid = SDL_CreateThread(video_thread, is);
49.  break;
50.  default:
51.  break;
52.  }
53.  }

```

이것은 오디오와 비디오를 처리하는 것만 빼고 전의 코드와 거의 같습니다. aCodecCtx 대신 우리의 큰 구조체의 오디오 콜백을 위한 userdata 를 설정합니다. 우리는 audio_st 와 video_st 스트림이 자동으로 저장되게 했습니다. 그리고 비디오 큐를 추가했고 오디오 큐도 마찬가지로 설정했습니다. 거의 모든 포인트는 비디오와 오디오 쓰레드를 실행합니다.

```

1.  SDL_PauseAudio(0);
2.  break;
3.
4.  /* ..... */
5.
6.  is->video_tid = SDL_CreateThread(video_thread, is);

```

지난번에 [SDL_PauseAudio\(\)](#) 과 [SDL_CreateThread\(\)](#)를 같은 방법으로 사용했던것을 기억 하시나요? video_thread()함수로 돌아가 봅시다.

아 그전에 decode_thread() 함수의 중간 부분으로 가봅시다. 이 부분은 패킷을 읽고 큐에 넣는 루프입니다.

```

1.  for(;;) {
2.  if(is->quit) {
3.  break;
4.  }
5.  // seek stuff goes here
6.  if(is->audioq.size > MAX_AUDIOQ_SIZE ||
7.  is->videoq.size > MAX_VIDEOQ_SIZE) {

```

```

8.     SDL_Delay(10);
9.     continue;
10.  }
11.  if(av_read_frame(is->pFormatCtx, packet) < 0) {
12.      if(url_ferror(&pFormatCtx->pb) == 0) {
13.          SDL_Delay(100); /* no error; wait for user input */
14.          continue;
15.      } else {
16.          break;
17.      }
18.  }
19.  // Is this a packet from the video stream?
20.  if(packet->stream_index == is->videoStream) {
21.      packet_queue_put(&is->videoq, packet);
22.  } else if(packet->stream_index == is->audioStream) {
23.      packet_queue_put(&is->audioq, packet);
24.  } else {
25.      av_free_packet(packet);
26.  }
27.  }

```

여기도 특별히 새로울것은 없습니다.

여기서는 오디오와 비디오 큐를 위해 MAX size 를 정해주고 Read 에러를 체크하기 위한 함수가 추가되었습니다.

Format Context 에는 pb.라고 부르는 ByteIOContext 구조체가 들어있습니다. ByteIOContext 는 low-level 파일 정보를 모두 갖고 있도록 구성되어 있습니다. [url_ferror](#) 체크는 File 에서 읽는 도중에 어떤 에러가 발생하는지 볼수 있게 되어있습니다.

루프를 실행한 후, 우리 코드는 프로그램이 끝나거나 끝을 알려주기를 기다리고 있습니다.

다음 코드는 어떻게 이벤트들을 Push 하는지 알려줍니다.

```

1.  while(!is->quit) {
2.      SDL_Delay(100);
3.  }
4.
5.  ail:
6.  if(1){
7.      SDL_Event event;
8.      event.type = FF_QUIT_EVENT;
9.      event.user.data1 = is;
10.     SDL_PushEvent(&event);
11. }
12. return 0;

```

우리는 SDL 의 SDL_USEREVENT 를 사용함으로써 User Event 를 얻을수 있습니다. 첫번째 User Event 는 SDL_USEREVENT 값에 의해 할당될 것입니다. 다음은 SDL_USEREVENT + 1 이고 쪽~~~ 이렇게 됩니다. FF_QUIT_EVENT 는 우리 프로그램에서 SDL_USEREVENT + 2 로 정의됩니다. 그리고 우리는 User Data 를 통과할 수 있습니다. 그리고 여기서는 큰 구조체를 가리키는 포인터를 통과합니다. 드디어 우리는 [SDL_PushEvent\(\)](#)를 호출했습니다. 이 이벤트 루프에서 우리는 SDL_QUIT_EVENT 섹션에 의해 단지 이것을 넣기만 할 뿐입니다. 우리는 이벤트 루프를 좀더 자세하게 볼것입니다. 우선은 언제 우리가 FF_QUIT_EVENT 를 Push 하는지 확실히 알기만 하면 됩니다. 우리는 다음에 이것을 캐치하고 quit Flag 를 발생시킬 것 입니다.

Getting the Frame : video_thread

우리 코덱이 준비된 다음에 비디오 쓰레드를 실행시킬 것입니다. 이 쓰레드는 비디오 큐의 패킷을 읽고 비디오를 프레임으로 디코드 하고 queue_picture 함수를 호출해서 처리된 프레임을 Picture 큐에 넣습니다.

```
1. int video_thread(void *arg) {
2.     VideoState *is = (VideoState *)arg;
3.     AVPacket pkt1, *packet = &pkt1;
4.     int len1, frameFinished;
5.     AVFrame *pFrame;
6.
7.     pFrame = avcodec_alloc_frame();
8.
9.     for(;;) {
10.        if(packet_queue_get(&is->videoq, packet, 1) < 0) {
11.            // means we quit getting packets
12.            break;
13.        }
14.        // Decode video frame
15.        len1 = avcodec_decode_video(is->video_st->codec, pFrame, &frameFinished,
16.            packet->data, packet->size);
17.
18.        // Did we get a video frame?
19.        if(frameFinished) {
20.            if(queue_picture(is, pFrame) < 0) {
21.                break;
22.            }
23.        }
24.        av_free_packet(packet);
25.    }
26.    av_free(pFrame);
27.    return 0;
28. }
```

이 함수는 몇가지 빼고는 거의 유사합니다. 우리는 [avcodec_decode_video](#) 함수를 여기로 옮겼습니다. 그냥 아규먼트 몇개만 바꿨을 뿐입니다. [AVStream](#) 을 큰 구조체에 넣었고 거기에서 코덱을 얻습니다. 우리는 에러를 만나거나 종료될때 까지 비디오 큐에서 패킷을 짚짚 읽기만 하면 됩니다.

Queueing the Frame

디코딩된 프레임이 저장된 함수를 보면 Picture 큐에 pFrame 가 있습니다. Picture 큐는 SDL 오버레이이기 때문에 우리 프레임을 변환해야 합니다. 우리가 만든 구조체의 Picture 큐에 그 데이터를 저장 합니다.

```
1. typedef struct VideoPicture {
2.     SDL_Overlay *bmp;
3.     int width, height; /* source height & width */
4.     int allocated;
5. } VideoPicture;
```

우리 큰 구조체는 이것들의 버퍼를 갖고 있고 그곳에 저장할 수 있습니다. 하지만 [SDL_Overlay](#) 를 할당해야 합니다.

이 큐를 이용하기 위해 우리는 두개의 포인터를 갖고 있습니다. - 쓸 인덱스와 읽을 인덱스를 가리키는 것 두개요. 그리고 버퍼안에 얼마나 많은 picture 가 있는지 계속 체크하고 있어야 합니다. 큐에 쓰기위해 우리는 버퍼가 클리어 되기를 기다려야 합니다. 그래서 VideoPicture 를 저장할 공간을 갖고 있습니다. 쓸 인덱스에 이미 오버레이를 할당했다면 그것을 체크해봐야 합니다. 그렇지 않으면 공간을 할당해줘야 합니다. 또한 윈도우의 크기가 바뀌었다면 버퍼를 재할당 해줘야 합니다. 그러나 여기서 할당을 하는것 대신 locking issues 를 피하게 하겠습니다.

```
1. int queue_picture(VideoState *is, AVFrame *pFrame) {
2.
3.     VideoPicture *vp;
4.     int dst_pix_fmt;
5.     AVPicture pict;
6.
7.     /* wait until we have space for a new pic */
8.     SDL_LockMutex(is->pictq_mutex);
9.     while(is->pictq_size >= VIDEO_PICTURE_QUEUE_SIZE &&
10.         !is->quit) {
11.         SDL_CondWait(is->pictq_cond, is->pictq_mutex);
12.     }
13.     SDL_UnlockMutex(is->pictq_mutex);
14.
15.     if(is->quit)
```

```

16.     return -1;
17.
18.     // windex is set to 0 initially
19.     vp = &is->pictq[is->pictq_windex];
20.
21.     /* allocate or resize the buffer! */
22.     if(!vp->bmp ||
23.        vp->width != is->video_st->codec->width ||
24.        vp->height != is->video_st->codec->height) {
25.         SDL_Event event;
26.
27.         vp->allocated = 0;
28.         /* we have to do it in the main thread */
29.         event.type = FF_ALLOC_EVENT;
30.         event.user.data1 = is;
31.         SDL_PushEvent(&event);
32.
33.         /* wait until we have a picture allocated */
34.         SDL_LockMutex(is->pictq_mutex);
35.         while(!vp->allocated && !is->quit) {
36.             SDL_CondWait(is->pictq_cond, is->pictq_mutex);
37.         }
38.         SDL_UnlockMutex(is->pictq_mutex);
39.         if(is->quit) {
40.             return -1;
41.         }
42.     }

```

이 이벤트 메커니즘은 우리가 이미 봤던 종료시에 했던 그것과 같습니다. 우리는 FF_ALLOC_EVENT 를 SDL_USEREVENT 처럼 정의했습니다. 이벤트를 발생시키고 할당함수를 돌리는 할당변수를 기다립니다.

우리 이벤트 루프가 어떻게 바뀌었는지 보세요

```

1.     for(;;) {
2.         SDL_WaitEvent(&event);
3.         switch(event.type) {
4.             /* ... */
5.             case FF_ALLOC_EVENT:
6.                 alloc_picture(event.user.data1);
7.                 break;

```

event.user.data1 이 우리 큰 구조체라는 것을 기억하세요.

다음 alloc_picture() 함수를 보시겠습니다.

```

1. void alloc_picture(void *userdata) {
2.
3.     VideoState *is = (VideoState *)userdata;
4.     VideoPicture *vp;
5.
6.     vp = &is->pictq[is->pictq_windex];
7.     if(vp->bmp) {
8.         // we already have one make another, bigger/smaller
9.         SDL_FreeYUVOverlay(vp->bmp);
10.    }
11.    // Allocate a place to put our YUV image on that screen
12.    vp->bmp = SDL_CreateYUVOverlay(is->video_st->codec->width,
13.                                  is->video_st->codec->height,
14.                                  SDL_YV12_OVERLAY,
15.                                  screen);
16.    vp->width = is->video_st->codec->width;
17.    vp->height = is->video_st->codec->height;
18.
19.    SDL_LockMutex(is->pictq_mutex);
20.    vp->allocated = 1;
21.    SDL_CondSignal(is->pictq_cond);
22.    SDL_UnlockMutex(is->pictq_mutex);
23. }

```

여러분은 우리 Main 루프에서 여기로 가져온 [SDL_CreateYUVOverlay\(\)](#) 함수를 기억하셔야 합니다. 이 코드는 공정하게 self-explanatory 됩니다. 비디오 크기는 변하면 안되기 때문에 VideoPicture 에 width, height 를 저장해야 하는것을 기억해야 합니다.

자 모두 세팅 되었습니다. 우리는 YUV overlay 를 할당했고 picture 를 받을 준비가 되었습니다. queue_picture 로 돌아가서 오버레이로 frame 을 복사하는 코드를 보겠습니다.

```

1. int queue_picture(VideoState *is, AVFrame *pFrame) {
2.
3.     /* Allocate a frame if we need it... */
4.     /* ... */
5.     /* We have a place to put our picture on the queue */
6.
7.     if(vp->bmp) {
8.
9.         SDL_LockYUVOverlay(vp->bmp);
10.
11.         dst_pix_fmt = PIX_FMT_YUV420P;

```

```

12.  /* point pict at the queue */
13.
14.  pict.data[0] = vp->bmp->pixels[0];
15.  pict.data[1] = vp->bmp->pixels[2];
16.  pict.data[2] = vp->bmp->pixels[1];
17.
18.  pict.linesize[0] = vp->bmp->itches[0];
19.  pict.linesize[1] = vp->bmp->itches[2];
20.  pict.linesize[2] = vp->bmp->itches[1];
21.
22.  // Convert the image into YUV format that SDL uses
23.  img_convert(&pict, dst_pix_fmt,
24.             (AVPicture *)pFrame, is->video_st->codec->pix_fmt,
25.             is->video_st->codec->width, is->video_st->codec->height);
26.
27.  SDL_UnlockYUVOverlay(vp->bmp);
28.  /* now we inform our display thread that we have a pic ready */
29.  if(++is->pictq_windex == VIDEO_PICTURE_QUEUE_SIZE) {
30.      is->pictq_windex = 0;
31.  }
32.  SDL_LockMutex(is->pictq_mutex);
33.  is->pictq_size++;
34.  SDL_UnlockMutex(is->pictq_mutex);
35.  }
36.  return 0;
37. }

```

이 부분에 중요한 것은 우리가 전에 프레임으로 YUV overlay 를 채웠던 그 것 입니다. 마지막 비트는 간단하게 우리 큐에 값을 추가합니다. 큐는 꽉찰때 까지 계속 추가했다가 큐에 무엇인가 있는 동안 계속 읽어옵니다. 그래서 모든것은 is->pictq_size 에 관련되어 있고 lock 이 필요합니다. 우리가 여기에 무엇을 하던간 write pointer 를 증가시켜야 하고 그래서 queue 에 lock 을 걸고 싸이즈를 늘려야 합니다. 그럼 Reader 는 큐에 정보가 더 있는지 알수 있고 큐가 가득차면 Writer 가 인지할 것입니다.

Displaying the Video

이것은 비디오 쓰레드를 위한 것입니다. 우리가 전에 schedule_refresh() 함수를 호출했던 것을 기억하시나요? 이게 실제로 어떻게 되어있는지 보겠습니다.

```

1.  /* schedule a video refresh in 'delay' ms */
2.  static void schedule_refresh(VideoState *is, int delay) {
3.      SDL_AddTimer(delay, sdl_refresh_timer_cb, is);
4.  }

```

[SDL_AddTimer\(\)](#) 는 SDL 에서 간단히 user-specified 한 callback 함수를 만들수 있게 해줍니다. 정확한 millisecond 를 제공하고 추가적으로 user data 를 전달할 수 있습니다. 우리는 이 함수를 이용해서 video update 를 구현할 것입니다. 이 함수가 호출될 때 마다 타이머를 세팅할것이고 event 를 발생시킬것 입니다. 이것을 main() 함수에 넣고 함수를 돌려서 Picture 에서 프레임을 땡겨올 것입니다.

일단 가장 우선으로 할 것은 이벤트를 발생시키는 것입니다.

```
1. static Uint32 sdl_refresh_timer_cb(Uint32 interval, void *opaque) {
2.     SDL_Event event;
3.     event.type = FF_REFRESH_EVENT;
4.     event.user.data1 = opaque;
5.     SDL_PushEvent(&event);
6.     return 0; /* 0 means stop timer */
7. }
```

여기는 이벤트 Push 하는것과 유사합니다. FF_REFRESH_EVENT 는 여기에서 SDL_USEREVENT + 1 로 정의됩니다. 한가지 알아야 할것은 return 0 을 하면 SDL 이 타이머를 멈추고 callback 은 다시 만들어지지 않는다는 것입니다.

FF_REFRESH_EVENT 를 push 한것을 이벤트 루프에서 제어해야 합니다.

```
1. for(;;) {
2.
3.     SDL_WaitEvent(&event);
4.     switch(event.type) {
5.         /* ... */
6.         case FF_REFRESH_EVENT:
7.             video_refresh_timer(event.user.data1);
8.             break;
```

우리의 Picture 큐에서 데이터를 끌어올 것입니다.

```
1. void video_refresh_timer(void *userdata) {
2.
3.     VideoState *is = (VideoState *)userdata;
4.     VideoPicture *vp;
5.
6.     if(is->video_st) {
7.         if(is->pictq_size == 0) {
8.             schedule_refresh(is, 1);
9.         } else {
```

```

10.   vp = &is->pictq[is->pictq_rindex];
11.   /* Timing code goes here */
12.
13.   schedule_refresh(is, 80);
14.
15.   /* show the picture! */
16.   video_display(is);
17.
18.   /* update queue for next picture! */
19.   if(++is->pictq_rindex == VIDEO_PICTURE_QUEUE_SIZE) {
20.     is->pictq_rindex = 0;
21.   }
22.   SDL_LockMutex(is->pictq_mutex);
23.   is->pictq_size--;
24.   SDL_CondSignal(is->pictq_cond);
25.   SDL_UnlockMutex(is->pictq_mutex);
26.   }
27. } else {
28.   schedule_refresh(is, 100);
29. }
30. }

```

우리가 뭔가를 갖고있을때 큐에서 가져오는 매우 간단한 함수 입니다. 다음 비디오 프레임을 보여줘야 할때를 위해 Timer 를 설정합니다. 스크린에 비디오를 보여주기 위해서 video_display 를 호출합니다. 큐에 카운터를 증가시켜주고 사이즈는 줄여줍니다. 여러분은 혹시 이 함수에서 vp 가 아무런 역할도 없다는 것을 눈치 채셨을지도 모릅니다. 우리는 다음에 비디오와 오디오를 싱크할때 이것을 이용해서 시간정보를 얻어낼 것입니다. "오옷! 타이밍 코드가 여기에?" 이 섹션에서는 다음 비디오를 얼마나 빨리 보여줄 것인지를 이해하시면 됩니다. 그리고 schedule_refresh() 함수에는 더미값으로 80 을 넣겠습니다. 여러분은 이 값을 바꾸고 재 컴파일 해서 확인해 보세요. 어떤 변화가 있을까요?

자... 이제 거의 다 되었습니다. 이제 마지막 한가지만 하면 되는데요. 비디오를 디스플레이 하는 것이죠 여기서 video_display() 함수가 등장합니다.

```

1.   void video_display(VideoState *is) {
2.
3.     SDL_Rect rect;
4.     VideoPicture *vp;
5.     AVPicture pict;
6.     float aspect_ratio;
7.     int w, h, x, y;
8.     int i;
9.
10.    vp = &is->pictq[is->pictq_rindex];

```

```

11. if(vp->bmp) {
12.     if(is->video_st->codec->sample_aspect_ratio.num == 0) {
13.         aspect_ratio = 0;
14.     } else {
15.         aspect_ratio = av_q2d(is->video_st->codec->sample_aspect_ratio) *
16.         is->video_st->codec->width / is->video_st->codec->height;
17.     }
18.     if(aspect_ratio <= 0.0) {
19.         aspect_ratio = (float)is->video_st->codec->width /
20.         (float)is->video_st->codec->height;
21.     }
22.     h = screen->h;
23.     w = ((int)rint(h * aspect_ratio)) & -3;
24.     if(w > screen->w) {
25.         w = screen->w;
26.         h = ((int)rint(w / aspect_ratio)) & -3;
27.     }
28.     x = (screen->w - w) / 2;
29.     y = (screen->h - h) / 2;
30.
31.     rect.x = x;
32.     rect.y = y;
33.     rect.w = w;
34.     rect.h = h;
35.     SDL_DisplayYUVOverlay(vp->bmp, &rect);
36. }
37. }

```

스크린 크기가 변할수 있기 때문에 우리는 동적으로 무비의 크기를 측정해야 합니다. 그래서 우리는 일단 무비의 aspect ratio(width를 height로 나눈것)를 측정해야 합니다. 몇몇 코덱은 홀수의 sample aspect ratio를 갖고 있습니다. 이것은 간단히 싱글 픽셀의 width/height 비율입니다. Codec context의 height와 width 값은 픽셀로 측정되기 때문에 실제 aspect ratio는 aspect ratio와 sample aspect ratio의 곱과 같습니다. 몇몇 코덱에서 aspect ratio가 0인것을 볼수 있는데 이것은 간단히 size가 1x1인것을 가리킵니다. 그러면 이제 우리는 영상을 우리 스크린에 맞추출수 있습니다. & -3 bit-twiddling을 하면 거의 4의 배수의 값이 나옵니다. 그러면 우리는 영상의 중심을 맞추고 [SDL_DisplayYUVOverlay\(\)](#)를 호출합니다.

새로운 VideoStruct를 이용하려면 오디오 코드를 다시 써야 합니다. 약간만 바꾸면 됩니다. 다음의 샘플 코드를 보세요

마지막으로 할것은 ffmpeg의 내부 quit 콜백함수를 바꾸는 것입니다.

```

1. VideoState *global_video_state;
2.
3. int decode_interrupt_cb(void) {

```

```
4. return (global_video_state && global_video_state->quit);
5. }
```

우리는 `global_video_state` 를 `main()` 함수의 큰 구조체로 설정했습니다.

자 다되었군요! 컴파일 해봅시다.

```
gcc -o tutorial04 tutorial04.c -lavutil -lavformat -lavcodec -lz -lm `
`sdl-config --cflags --libs`
```

자~ 싱크가 안맞는 영상을 감상해 보세요 ~~~

ㅋㅋㅋ

다음번에는 드디어 실질적인 비디오 플레이어를 만들것 입니다.

Tutorial 05: Synching Video

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

운 좋게 오디오와 비디오가 갖고 있는 정보에는 얼마나 빨리 그리고 언제 플레이를 해야할지에 대한 정보가 들어 있습니다.

오디오 스트림에는 `sample rate` 가 있고, 비디오 스트림에는 `frame per second` 값이 있습니다. 하지만 만일 우리가 간단하게 다양한 `frame rate` 에 의해 단순히 계속 `Frame` 을 진행시켜서 비디오 싱크를 맞춘다면, 오디오 싱크와 어긋날 가능성이 있습니다. 대신에 스트림의 패킷은 아마 `Decoding 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 MPEG 같은 포맷들은 'B'프레임(B stands for "bidirectional")이라고 불리는 것을 사용합니다. 그리고 'I'프레임과 'P'프레임("I" for "intra" and "P" for "predicted") 이라고 불리는 것이 있습니다. I 프레임은 Full Image 를 포함하고 있고, P 프레임은 앞선 I 프레임과 P 프레임의 차이 정보를 갖고 있습니다. B 프레임은 P 프레임과 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 B 프레임을 디스플레이 하기 전에 P 프레임에 대한 정보를 알아야 합니다. 때문에 프레임들은 아마 다음과 같이 저장되어 있을 것입니다. I P B B. 이것은 왜 `Decoding timestamp` 와 `Presentaion timestamp` 가 각 프레임마다 있어야 하는지를 말해줍니다. `Decoding timestamp` 는 언제 디코딩해야 하는지를 말해줍니다. 그리고 `Presentation timestamp` 는 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 해야 하는지에 대한 것만 다릅니다.

우리가 [av_read_frame\(\)](#)으로 패킷을 얻을 때, 패킷 안에는 PTS와 DTS 값이 포함되어 있을 것입니다. 하지만 우리가 정말 원하는 것은 새로 디코드된 RAW Frame의 PTS기 때문에 그것이 언제 디스플레이 하는지입니다. 그러나 [avcodec_decode_video\(\)](#)로 얻은 프레임은 우리에게 쓸모있는 PTS 값이 있는지 없는지 모르는 [AVFrame](#)를 넘겨줍니다. (※주의 : AVFrame은 PTS 값을 갖고 있지만 우리가 언제 프레임을 얻기를 원하는지에 대한 정보가 항상 포함되어 있는 것은 아닙니다.) 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 첫 번째 패킷의 PTS를 저장합니다. 이것은 마지막 프레임의 PTS가 될 것입니다. 그래서 스트림이 DTS를 넘겨주지 않을 때, 우리는 그냥 저장된 PTS를 사용하면 됩니다. 우리는 [avcodec_decode_video\(\)](#)가 알려줌으로써 프레임의 첫 번째 패킷을 만들 수 있습니다. 어떻게? 언제든지 한 패킷은 한 프레임을 시작합니다. [avcodec_decode_video\(\)](#)은 우리 프레임에 한 버퍼를 할당하는 함수를 호출할 것입니다. 그리고 물론 ffmpeg는 그 할당 함수를 재정의 할 수 있게 해줍니다. 이제 패킷의 PTS를 저장하는 새 함수를 만들겠습니다.

우리가 알맞은 PTS를 얻지 못했더라도 이것은 다음에 다루겠습니다.

Synching

이제 우리는 언제 각각의 비디오 프레임을 보여 주어야 하는지 알게 되었습니다. 하지만 그것은 실제로 어떻게 해야 할까요?

한 프레임을 보여준 다음 언제 다음 프레임을 보여 주어야 할지에 대한 방법이 여기 있습니다. 그러면 간단히 일정 시간이 지나면 반복적으로 비디오를 Refresh하기 위한 Timeout을 새롭게 설정하겠습니다. 여러분이 기대하시는 것처럼 우리는 타임아웃이 얼마나 걸리는지 시스템 클럭을 보기 위해 다음 프레임의 PTS의 값을 체크합니다. 이 접근 작업은 기본적으로 두 가지 이슈가 있습니다.

첫 번째 이슈는 언제 다음 PTS가 될 것이냐 입니다. 여러분은 아마 그냥 Video rate를 현재 PTS에 추가한다고 생각할 지도 모릅니다. 그렇습니다. 하지만 몇몇 종류의 비디오들은 반복되는 프레임들을 호출합니다. 이 의미는 현재 프레임을 일정 시간 동안 반복시켜야 한다는 것입니다. 프로그램이 다음 프레임을 너무 빨리 디스플레이 하기 때문에 그렇습니다. 우리는 그래서 그것을 계산해야 합니다.

두 번째 이슈는, 프로그램은 멈춰있고 비디오와 오디오는 흘러가고 있는데 싱크가 걱정스럽습니다. 만약 모든 것이 완벽하게 이루어진다면 걱정할 필요가 없지만 여러분의 컴퓨터는 그렇지 못합니다. 그리고 많은 비디오 파일들도 그렇지 않습니다. 그래서 우리는 세 가지 선택을 할 수 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 놓여진 패킷을 어디에서 꺼내는지 기억해야 합니다. [avcodec_decode_video](#) 에 의해 우리에게 주어진 프레임의 PTS 를 얻는 코드가 필요합니다. 첫번째 방법은 마지막 패킷이 처리된 DTS 를 얻는 것입니다.

```
1. double pts;
2.
3. for(;;) {
4.     if(packet_queue_get(&is->videoq, packet, 1) < 0) {
5.         // means we quit getting packets
6.         break;
7.     }
8.     pts = 0;
9.     // Decode video frame
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);
```

이것을 처리하지 못하면 PTS 를 0 으로 설정합니다.

만약 패킷의 DTS 가 우리를 도와주지 못한다면 우리는 디코드된 프레임의 첫번째 패킷의 PTS 를 이용해야 합니다. ffmpeg 는 우리가 만든 함수로 한 프레임을 할당합니다. 다음이 그 함수 입니다.

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

get 함수는 패킷에 대한 어떤 것도 알려주지 않습니다. 그래서 패킷을 얻을때 마다 PTS 를 글로벌 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.

```
1. uint64_t global_video_pkt_pts = AV_NOPTS_VALUE;
2.
3. /* These are called whenever we allocate a frame
4.  * buffer. We use this to store the global_pts in
```

```

5.  * a frame at the time it is allocated.
6.  */
7.  int our_get_buffer(struct AVCodecContext *c, AVFrame *pic) {
8.      int ret = avcodec_default_get_buffer(c, pic);
9.      uint64_t *pts = av_malloc(sizeof(uint64_t));
10.     *pts = global_video_pkt_pts;
11.     pic->opaque = pts;
12.     return ret;
13. }
14. void our_release_buffer(struct AVCodecContext *c, AVFrame *pic) {
15.     if(pic) av_freep(&pic->opaque);
16.     avcodec_default_release_buffer(c, pic);
17. }

```

[avcodec_default_get_buffer](#) 와 [avcodec_default_release_buffer](#) 는 ffmpeg 가 할당을 위해 사용하는 기본 함수 입니다. [av_freep](#) 는 메모리 관리 함수입니다. 이것은 지정되어 있는 메모리를 해제하는것 뿐만 아니라 포인터를 NULL 로 설정해 줍니다.

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

```

1.  codecCtx->get_buffer = our_get_buffer;
2.  codecCtx->release_buffer = our_release_buffer;

```

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

```

1.  for(;;) {
2.      if(packet_queue_get(&is->videoq, packet, 1) < 0) {
3.          // means we quit getting packets
4.          break;
5.      }
6.      pts = 0;
7.
8.      // Save global pts to be stored in pFrame in first call
9.      global_video_pkt_pts = packet->pts;
10.     // Decode video frame
11.     len1 = avcodec_decode_video(is->video_st->codec, pFrame, &frameFinished,
12.         packet->data, packet->size);
13.     if(packet->dts == AV_NOPTS_VALUE
14.         && pFrame->opaque && *(uint64_t*)pFrame->opaque != AV_NOPTS_VALUE) {
15.         pts = *(uint64_t *)pFrame->opaque;
16.     } else if(packet->dts != AV_NOPTS_VALUE) {

```

```

17.     pts = packet->dts;
18.   } else {
19.     pts = 0;
20.   }
21.   pts *= av_q2d(is->video_st->time_base);

```

우리는 PST 를 int64 로 사용하고있습니다. 이것은 PTS 가 하나의 Interger 로 저장되어 있기 때문입니다. 하나의 timestamp 값은 스트림의 time_base 에 있는 시간을 측정한것과 일치합니다. 예를들면 만약 우리가 매 초당 24 프레임 을 갖고 있다면 42 의 PTS 는 42 번째의 프레임 을 지정할 것입니다. 우리는 이 값을 Frame rate 로 나누어서 초로 바꿀수 있습니다. 스트림의 time_base 값은 1/framerate 가 될것입니다.(fixed-fps 일 경우). 그래서 초에서 PTS 를 얻으려면 우리는 time_base 에 곱해야 합니다.

Coding: Synching and using the PTS

이제 PTS 를 얻어서 모든 세팅을 다 하였습니다. 이제 우리는 전에 언급했던 싱크하기위한 두가지 문제점에 대해 주의를 해야합니다. 우리는 synchronize_video 라는 함수를 정의할 것입니다. 이 함수는 PTS 를 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 rate 를 알맞게 설정해야 합니다. 우리는 이것을 video 가 얼마나 많은 시간이 흘렀는지를 추적하는 내부 video_clock 값을 사용함으로써 완성시킬수 있습니다. 우리는 이 값을 우리의 큰 구조체에 넣겠습니다.

```

1.  typedef struct VideoState {
2.    double video_clock; ///

```

다음은 synchronize_video 함수입니다.

```

1.  double synchronize_video(VideoState *is, AVFrame *src_frame, double pts) {
2.
3.    double frame_delay;
4.
5.    if(pts != 0) {
6.      /* if we have pts, set video clock to it */
7.      is->video_clock = pts;
8.    } else {
9.      /* if we aren't given a pts, set it to the clock */
10.     pts = is->video_clock;
11.   }
12.   /* update the video clock */
13.   frame_delay = av_q2d(is->video_st->codec->time_base);
14.   /* if we are repeating a frame, adjust clock accordingly */
15.   frame_delay += src_frame->repeat_pict * (frame_delay * 0.5);
16.   is->video_clock += frame_delay;
17.   return pts;

```

```
18. }
```

여러분은 이함수에서 역시 반복되는 프레임을 세야 합니다.

이제는 적절한 PTS 를 구하고 queue_picture 를 사용해 queue 에 frame 을 올리겠습니다.

```
1. // Did we get a video frame?
2. if(frameFinished) {
3.     pts = synchronize_video(is, pFrame, pts);
4.     if(queue_picture(is, pFrame, pts) < 0) {
5.         break;
6.     }
7. }
```

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

```
1. typedef struct VideoPicture {
2.     ...
3.     double pts;
4. }
5. int queue_picture(VideoState *is, AVFrame *pFrame, double pts) {
6.     ... stuff ...
7.     if(vp->bmp) {
8.         ... convert picture ...
9.         vp->pts = pts;
10.        ... alert queue ...
11. }
```

그래서 우리는 picture 들을 picture queue 에 적절한 PTS 값과 같이 순서대로 넣고, Video refresh 함수를 찾습니다.

여러분은 아마 지난번에 우리가 이부분을 속이고 refresh 를 80ms 로 임의 지정했던것을 기억하실 것입니다. 자. 이번에는 실제로 이것을 어떻게 하는지를 알아보겠습니다.

우리의 전략은 앞선 PTS 와 이번것의 시간을 간단히 측정함으로써 다음 PTS 의 시간을 예측 하는 것입니다.

동시에 비디오를 오디오에 싱크해야 합니다. 우리는 **Audio Clock** 을 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 clock 의 시간을 주겠습니다.

한번 우리가 값을 갖고 있었더라도 비디오와 오디오의 싱크가 어긋나면 우리는 어떻게 해야 할까요? 멍청하게 간단히 시도하고 검색들을 통해 정확한 패킷으로 뛰어넘어가야 할 것입니다. 대신 우리는 그냥 다음 refresh 를 위해 계산했던 값으로 조정할 것입니다. 만일 PTS 가 오디오 타임에 너무 뒤에 있으면 계산된 딜레이를 두배로 하고 만약에 PTS 가 오디오 보다 너무 앞에 있다면 그냥 refresh 를 빠르게 하면 됩니다. 이제 우리는 조정된 시간과 delay 을 갖게 되었고, 다음으로는 우리 실행중인 frame_timer 에 의한 컴퓨터의 클럭과 비교를 해야 합니다. 이 프레임 타이머는 동영상이 플레이 되는 동안의 계산된 delay 를 모두 더할것입니다. 다른말로 이 frame_timer 는 언제 우리가 다음 프레임을 디스플레이 할지를 정하는 것입니다. 간단히 frame timer 에 새로운 delay 를 더하고 우리 컴퓨터의 clock 과 비교를 합니다. 그리고 그 값을 이용해서 다음 refresh 스케줄을 짭니다. 이것은 약간 복잡하기 때문에 주의 깊게 보셔야 합니다.

```

1. void video_refresh_timer(void *userdata) {
2.
3.     VideoState *is = (VideoState *)userdata;
4.     VideoPicture *vp;
5.     double actual_delay, delay, sync_threshold, ref_clock, diff;
6.
7.     if(is->video_st) {
8.         if(is->pictq_size == 0) {
9.             schedule_refresh(is, 1);
10.        } else {
11.            vp = &is->pictq[is->pictq_rindex];
12.
13.            delay = vp->pts - is->frame_last_pts; /* the pts from last time */
14.            if(delay <= 0 || delay >= 1.0) {
15.                /* if incorrect delay, use previous one */
16.                delay = is->frame_last_delay;
17.            }
18.            /* save for next time */
19.            is->frame_last_delay = delay;
20.            is->frame_last_pts = vp->pts;
21.
22.            /* update delay to sync to audio */
23.            ref_clock = get_audio_clock(is);
24.            diff = vp->pts - ref_clock;
25.
26.            /* Skip or repeat the frame. Take delay into account
27.             FFPlay still doesn't "know if this is the best guess." */
28.            sync_threshold = (delay > AV_SYNC_THRESHOLD) ? delay : AV_SYNC_THRESHOLD;
29.            if(fabs(diff) < AV_NOSYNC_THRESHOLD) {
30.                if(diff <= -sync_threshold) {
31.                    delay = 0;
32.                } else if(diff >= sync_threshold) {
33.                    delay = 2 * delay;

```

```

34.     }
35.     }
36.     is->frame_timer += delay;
37.     /* computer the REAL delay */
38.     actual_delay = is->frame_timer - (av_gettime() / 1000000.0);
39.     if(actual_delay < 0.010) {
40.         /* Really it should skip the picture instead */
41.         actual_delay = 0.010;
42.     }
43.     schedule_refresh(is, (int)(actual_delay * 1000 + 0.5));
44.     /* show the picture! */
45.     video_display(is);
46.
47.     /* update queue for next picture! */
48.     if(++is->pictq_rindex == VIDEO_PICTURE_QUEUE_SIZE) {
49.         is->pictq_rindex = 0;
50.     }
51.     SDL_LockMutex(is->pictq_mutex);
52.     is->pictq_size--;
53.     SDL_CondSignal(is->pictq_cond);
54.     SDL_UnlockMutex(is->pictq_mutex);
55.     }
56. } else {
57.     schedule_refresh(is, 100);
58. }
59. }

```

우리가 만든것을 몇개 체크해야 할게 있습니다.

첫째 PTS 와 전 PTS 같에 딜레이를 확실히 해야 합니다. 만일 그것이 안되면 우리는 추측하거나 마지막 delay 를 사용합니다. 다음으로는 싱크가 완벽하게 맞을 수 없기 때문에 우리는 싱크의 threshold 를 갖고 있어야 합니다. ffmpeg 는 이 값으로 0.01 을 사용합니다. 우리는 또한 싱크의 threshold 를 PTS 간의 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 timer 를 초기화 하고 stream_component_open 에서 이전 프레임의 delay 를 초기화 것을 잊어서는 안됩니다.

1. is->frame_timer = (double)av_gettime() / 1000000.0;
2. is->frame_last_delay = 40e-3;

Synching: The Audio Clock

Audio clock 을 실행할 차례 입니다. 우리는 audio_decode_frame 함수에서 어디서 오디오를 디코드 하는 지를 알려주는 clock time 을 업데이트 할수 있습니다. 이제 이 함수를 호출할때 마다 새 패킷을 처리할

필요가 없어졌습니다. 그래서 clock 을 업데이트 해야 하는 두 장소가 있습니다. 첫번째 장소는 새 패킷을 얻는곳 : 우리는 간단히 audio clock 을 패킷의 PTS 로 설정합니다. 그리고 만약 한 패킷이 여러개의 프레임 을 갖고 있다면 샘플들의 수를 센것에 의해 오디오 플레이 시간을 유지합니다. 그리고 주어진 samples-per-second rate 에 의해 그것들을 multiplying 합니다. 그래서 우리는 그 패킷을 얻었습니다.

```
1. /* if update, update the audio clock w/pts */
2. if(pkt->pts != AV_NOPTS_VALUE) {
3.     is->audio_clock = av_q2d(is->audio_st->time_base)*pkt->pts;
4. }
```

그리고 한번 패킷을 처리하고 있습니다.

```
1. /* Keep audio_clock up-to-date */
2. pts = is->audio_clock;
3. *pts_ptr = pts;
4. n = 2 * is->audio_st->codec->channels;
5. is->audio_clock += (double)data_size /
6. 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 PTS 를 설정해야 합니다. 그러나 우리가 audio_callback 함수를 보면 output buffer 에 오디오 패킷의 모든 데이터를 이동하는데 시간이 걸립니다. 이것은 audio clock 안에 있는 값이 너무 앞으로 갔다는 것을 의미합니다. 그래서 우리는 얼마나 쓰기위해 지나갔는지 체크해야 합니다.

다음이 그 코드 입니다.

```
1. double get_audio_clock(VideoState *is) {
2.     double pts;
3.     int hw_buf_size, bytes_per_sec, n;
4.
5.     pts = is->audio_clock; /* maintained in the audio thread */
6.     hw_buf_size = is->audio_buf_size - is->audio_buf_index;
7.     bytes_per_sec = 0;
8.     n = is->audio_st->codec->channels * 2;
9.     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 말할 수 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 볼 수 있게 되었네요!

다음번에는 오디오 싱크에 대해서 알아보고, 튜토리얼이 끝난 후 탐색에 관해 이야기 해보겠습니다.

Tutorial 06: Synching Audio

Synching Audio

자~ 이제 우리는 동영상을 볼 수 있는 어엿한 플레이어를 갖게 되었습니다.

그럼 이제 늘어져 있는 나머지 작업들을 보겠습니다. 지난번에는 약간의 싱크 차이를 그냥 얼버무렸었습니다. 즉 비디오를 오디오에 동기화 시키는 것보다 오디오를 비디오 클럭에 동기화 시켰습니다. 우리는 이것을 비디오와 같은 방법으로 하고 있습니다. : 비디오와 오디오가 얼마나 차이가 나는지를 추적하는 내부 비디오 클럭을 만드는 방법. 다음에는 오디오와 비디오를 외부 클럭에 어떻게 싱크하는지 찾아보겠습니다.

Implementing the video clock

이제 우리는 비디오 클럭을 저번에 했던 오디오 클럭처럼 실행해야 합니다. : 하나의 지역변수에 지금 비디오가 플레이 되고 있는 것의 time offset 을 줍니다. 우선 여러분은 간단하게 현재 PTS 와 타이머를 업데이트 합니다. 하지만 밀리세컨드 레벨에서 비디오 프레임의 간격은 꽤 길수도 있다는 것을 잊으면 안됩니다. 해결 방법은 다른 값을 추적하는 것입니다. 그 비디오 클럭을 마지막 프레임의 PTS 로 설정합니다. 비디오 클럭의 현재 값은 $PTS_of_last_frame + (current_time - time_elapsed_since_PTS_value_was_set)$ 입니다. 이 방법은 `get_audio_clock` 에서 우리가 했던 것과 매우 유사합니다.

그래서 우리의 큰 구조체 안에 `double video_current_pts` 와 `int64_t video_current_pts_time` 을 넣습니다. 클럭 업데이트는 `video_refresh_timer` 함수에 자리잡고 있다.

```
1. void video_refresh_timer(void *userdata) {
2.
3.     /* ... */
4.
5.     if(is->video_st) {
6.         if(is->pictq_size == 0) {
7.             schedule_refresh(is, 1);
8.         } else {
9.             vp = &is->pictq[is->pictq_rindex];
10.
11.             is->video_current_pts = vp->pts;
```

```
12.     is->video_current_pts_time = av_gettime();
```

stream_component_open 에서 초기화 하는것을 잊지 마세요.

```
is->video_current_pts_time = av_gettime();
```

이제 우리가 필요한 것은 정보를 얻는 방법입니다.

```
1.  double get_video_clock(VideoState *is) {
2.      double delta;
3.
4.      delta = (av_gettime() - is->video_current_pts_time) / 1000000.0;
5.      return is->video_current_pts + delta;
6.  }
```

Abstracting the clock

왜 우리가 video clock 을 사용하도록 강제했을까요? 우리는 비디오 싱크 코드를 변경할 것입니다. 그래서 오디오와 비디오가 서로 싱크하려 하지 않습니다. 만약 우리가 ffmpeg 에서 Command line option 을 만들라고 할때 문제점을 상상해 보세요. 개념은 이렇습니다. 우리는 새로운 wrapper function 을 만들것입니다. get_master_clock 함수는 av_sync_type 변수를 체크합니다. 그리고 get_audio_clock, get_video_clock 또는 우리가 쓰고싶은 다른 clock 들을 호출합니다. 우리는 컴퓨터의 clock 도 사용할수 있습니다. 우리는 get_external_clock 을 호출할것입니다.

```
1.  enum {
2.      AV_SYNC_AUDIO_MASTER,
3.      AV_SYNC_VIDEO_MASTER,
4.      AV_SYNC_EXTERNAL_MASTER,
5.  };
6.
7.  #define DEFAULT_AV_SYNC_TYPE AV_SYNC_VIDEO_MASTER
8.
9.  double get_master_clock(VideoState *is) {
10.     if(is->av_sync_type == AV_SYNC_VIDEO_MASTER) {
11.         return get_video_clock(is);
12.     } else if(is->av_sync_type == AV_SYNC_AUDIO_MASTER) {
13.         return get_audio_clock(is);
14.     } else {
15.         return get_external_clock(is);
16.     }
17. }
18. main() {
19.     ...
20.     is->av_sync_type = DEFAULT_AV_SYNC_TYPE;
```

- 21. ...
- 22. }

Synchronizing the Audio

여기는 어려운 부분입니다. 오디오를 비디오 클럭에 동기화 시킬 것입니다. 우리의 전략은 오디오가 어디인지를 측정하고 그것을 비디오 클럭과 비교한다음 얼마나 많은 샘플을 우리가 조정해야 하는지 알아내는 것입니다. 속도를 올려야 되는지, 샘플들을 건너뛰어야 하는지 아니면 속도를 줄여야 하는지 말이지요

우리는 `synchronize_audio` 함수를 실행할 것입니다. 매번 `audio sample` 세트를 처리하고 그것을 알맞게 늘리거나 줄여야 합니다. 그러나 우리는 매번 동기화 하는것을 원하지 않습니다. 그것은 비디오 패킷보다 훨씬 더 자주 오디오를 처리하기 때문입니다. So we're going to set a minimum number of consecutive calls to the `synchronize_audio` function that have to be out of sync before we bother doing anything. 물론 지난번과 같이 "out of sync"의 의미는 오디오클럭과 비디오클럭같은 우리의 `sync threshold` 를 벗어났다는 것을 말합니다.

그래서 우리는 `fractional coefficient` 를 사용합니다. 그리고 out of sync 된 N 개의 오디오 샘플 세트를 얻습니다. out of sync 가 많으면 많은 변화가 일어날 수 있어서 우리는 각각의 out of sync 된 것들의 평균값을 사용합니다. 예를 들면 첫번째호출은 아마 40ms 싱크가 어긋났고 다음번에는 50ms, 쪽~~ 이렇게 보일것입니다. 하지만 우리는 간단한 평균값을 갖지 않을것 입니다. 왜냐면 대부분 최근 값들은 이전 값들보다 더 중요하기 때문입니다. 그래서 우리는 `fractional coefficient`, say `c`, 그리고 다음 처럼 차이를 더한 것을 사용합니다. : $diff_sum = new_diff + diff_sum * c$ 우리가 차이값을 찾을 준비가 되면 $avg_diff = diff_sum * (1 - c)$ 로 간단히 계산합니다.

Here's what our function looks like so far:

```
1. /* Add or subtract samples to get a better sync, return new
2.    audio buffer size */
3. int synchronize_audio(VideoState *is, short *samples,
4.    int samples_size, double pts) {
5.    int n;
6.    double ref_clock;
7.
8.    n = 2 * is->audio_st->codec->channels;
9.
10.    if(is->av_sync_type != AV_SYNC_AUDIO_MASTER) {
11.        double diff, avg_diff;
12.        int wanted_size, min_size, max_size, nb_samples;
13.
14.        ref_clock = get_master_clock(is);
15.        diff = get_audio_clock(is) - ref_clock;
16.
17.        if(diff < AV_NOSYNC_THRESHOLD) {
```

```

18. // accumulate the diffs
19. is->audio_diff_cum = diff + is->audio_diff_avg_coef
20. * is->audio_diff_cum;
21. if(is->audio_diff_avg_count < AUDIO_DIFF_AVG_NB) {
22. is->audio_diff_avg_count++;
23. } else {
24. avg_diff = is->audio_diff_cum * (1.0 - is->audio_diff_avg_coef);
25.
26. /* Shrinking/expanding buffer code.... */
27.
28. }
29. } else {
30. /* difference is TOO big; reset diff stuff */
31. is->audio_diff_avg_count = 0;
32. is->audio_diff_cum = 0;
33. }
34. }
35. return samples_size;
36. }

```

우리는 꽤 잘하고 있습니다. 우리가 사용하고 있는 클럭이나 비디오로부터 오디오가 얼마나 차이나는지 대략 알고 있습니다. 그래서 이제 어느정도의 샘플을 "Shrinking/expanding buffer code"코드에서 추가하거나 빼거나 할지 계산합니다.

```

1. if(fabs(avg_diff) >= is->audio_diff_threshold) {
2. wanted_size = samples_size +
3. ((int)(diff * is->audio_st->codec->sample_rate) * n);
4. min_size = samples_size * ((100 - SAMPLE_CORRECTION_PERCENT_MAX)
5. / 100);
6. max_size = samples_size * ((100 + SAMPLE_CORRECTION_PERCENT_MAX)
7. / 100);
8. if(wanted_size < min_size) {
9. wanted_size = min_size;
10. } else if (wanted_size > max_size) {
11. wanted_size = max_size;
12. }

```

$\text{audio_length} * (\text{sample_rate} * \# \text{ of channels} * 2)$ 는 오디오의 audio_length 초 에 있는 샘플 수 라는것을 기억하세요. 그래서 샘플 수를 더하거나 빼고 오디오가 흘러간 양만큼 조정해야 합니다. 또한 크건 작건 보정한것의 한계를 설정해야 합니다. 왜냐면 버퍼가 너무 많이 바뀌면 사용자에게 혼란을 줄수 있기 때문입니다.

Correcting the number of samples

이제 우리는 오디오를 교정해야 합니다. 여러분은 아마 `synchronize_audio` 함수를 아실텐데 이 함수는 스트림으로 몇 바이트를 보내는지를 나타내는 `sample size` 를 리턴합니다. 그래서 우리는 그냥 `sample size` 를 `wanted_size` 로 조정하기만 하면 됩니다. 이일은 `sample size` 를 작게 만들기 위함입니다. 하지만 우리가 이것을 크게 만들고 싶다면 그냥 크게 만들수는 없습니다. 왜냐면 버퍼에 더이상의 데이터가 없기 때문입니다. 그래서 우리는 이것을 추가해야 합니다. 그런데 뭘 추가하죠? 이것은 바보같은 시도가 될것이고 오디오를 추측하는 것입니다. 그래서 우리는 마지막 샘플의 값으로 이미 부풀려진 버퍼의 오디오를 그냥 사용합니다.

```
1. if(wanted_size < samples_size) {
2.     /* remove samples */
3.     samples_size = wanted_size;
4. } else if(wanted_size > samples_size) {
5.     uint8_t *samples_end, *q;
6.     int nb;
7.
8.     /* add samples by copying final samples */
9.     nb = (samples_size - wanted_size);
10.    samples_end = (uint8_t *)samples + samples_size - n;
11.    q = samples_end + n;
12.    while(nb > 0) {
13.        memcpy(q, samples_end, n);
14.        q += n;
15.        nb -= n;
16.    }
17.    samples_size = wanted_size;
18. }
```

이제 우리는 샘플 사이즈를 리턴하고 함수를 끝마쳤습니다.

이제 우리는 이것들을 다 사용해야 합니다.

```
1. void audio_callback(void *userdata, Uint8 *stream, int len) {
2.
3.     VideoState *is = (VideoState *)userdata;
4.     int len1, audio_size;
5.     double pts;
6.
7.     while(len > 0) {
8.         if(is->audio_buf_index >= is->audio_buf_size) {
9.             /* We have already sent all our data; get more */
10.            audio_size = audio_decode_frame(is, is->audio_buf, sizeof(is->audio_buf), &pts);
11.            if(audio_size < 0) {
12.                /* If error, output silence */
13.                is->audio_buf_size = 1024;
```

```

14.  memset(is->audio_buf, 0, is->audio_buf_size);
15.  } else {
16.  audio_size = synchronize_audio(is, (int16_t *)is->audio_buf,
17.  audio_size, pts);
18.  is->audio_buf_size = audio_size;

```

우리가 했던 것은 synchronize_audio 를 호출함으로서 입력됩니다. (또한 어디어 우리가 변수들을 초기화 하였는지 확실히 체크해야 합니다.)

끝마치기 전에 하나가 남았는데

we need to add an if clause to make sure we don't sync the video if it is the master clock:

```

1.  if(is->av_sync_type != AV_SYNC_VIDEO_MASTER) {
2.  ref_clock = get_master_clock(is);
3.  diff = vp->pts - ref_clock;
4.
5.  /* Skip or repeat the frame. Take delay into account
6.  FFPlay still doesn't "know if this is the best guess." */
7.  sync_threshold = (delay > AV_SYNC_THRESHOLD) ? delay :
8.  AV_SYNC_THRESHOLD;
9.  if(fabs(diff) < AV_NOSYNC_THRESHOLD) {
10.  if(diff <= -sync_threshold) {
11.  delay = 0;
12.  } else if(diff >= sync_threshold) {
13.  delay = 2 * delay;
14.  }
15.  }
16.  }

```

다 되었군요!

여러분은 어떤 변수를 초기화 할때 중복정의 되었는지 중복초기화 되었는지 전체 소스를 잘 체크하셔야 합니다.

```
gcc -o tutorial06 tutorial06.c -lavutil -lavformat -lavcodec -lz -lm`SDL_CONFIG` --cflags --libs`
```

다음에는 뒤로감기와 빨리감기를 만들어 보겠습니다.

Tutorial 07: Seeking

Handling the seek command

이제 우리는 플레이어에다가 찾기 능력을 추가해보도록 하고 이것을 만들면서 [av_seek_frame](#) 함수를 사용하는 법을 익혀보겠습니다.

우리는 좌우 화살표는 영상의 앞뒤로 조금씩 움직이고 상하 화살표는 만표이 움직이게 하도록 만들겠습니다. (여기서 조금은 10 초 그리고 많이는 60 초 입니다.) 그래서 우리의 main loop 에서 키보드입력을 감지할수 있게 설정해야 합니다. 하지만 우리가 키 입력을 받았을 때, 바로 [av_seek_frame](#) 함수를 호출할수 없습니다.

그래서 이것은 main decode loop(decode_thread)에서 처리해야 합니다. 그래서 그대신에 우리 큰 구조체에다가 몇가지 값을 추가하겠습니다. 우리가 찾을 위치와 찾는중인지를 표시하는 플래그 값을요.

```
1. int seek_req;
2. int seek_flags;
3. int64_t seek_pos;
```

이제 우리 메인 루프에다가 키입력을 감지할 수 있도록 설정합니다.

```
1. for(;;) {
2.     double incr, pos;
3.
4.     SDL_WaitEvent(&event);
5.     switch(event.type) {
6.     case SDL_KEYDOWN:
7.         switch(event.key.keysym.sym) {
8.         case SDLK_LEFT:
9.             incr = -10.0;
10.            goto do_seek;
11.         case SDLK_RIGHT:
12.            incr = 10.0;
13.            goto do_seek;
14.         case SDLK_UP:
15.            incr = 60.0;
16.            goto do_seek;
17.         case SDLK_DOWN:
18.            incr = -60.0;
19.            goto do_seek;
20.         do_seek:
21.            if(global_video_state) {
22.                pos = get_master_clock(global_video_state);
23.                pos += incr;
24.                stream_seek(global_video_state,
25.                            (int64_t)(pos * AV_TIME_BASE), incr);
26.            }
27.            break;
28.         default:
29.            break;
30.         }
31.         break;
```

키입력을 감지하기 위해서는 우선 SDLK_KEYDOWN 이벤트를 받아야 합니다. 그러면 event.key.keysym.sym 을 체크해서 어떤키가 눌려졌는지 찾아냅니다. 찾는 법을 알았으니까 이제 새로운 get_master_clock 함수의

값을 증가시킨 새로운 time 을 계산해야 합니다. 그리고 seek_pos 등을 설정하기 위한 stream_seek 함수를 호출합니다. 그다음엔 우리가 얻은 새로운 time 을 avcodec 의 내부 timestamp 로 변환합니다. 스트림에서 timestamp 는 second 보다는 frame 으로 측정되는데 공식은 seconds=frames*time_base(fps) 입니다. avcodec 은 기본으로 한 값이 1,000,000fps 입니다. (2 초의 1 pos 는 2,000,000 timestamp 입니다.)

다음은 stream_seek 함수입니다. 만약 반대로 재생하고 있다면 플래그를 설정해야 합니다.

```
1. void stream_seek(VideoState *is, int64_t pos, int rel) {
2.
3.     if(!is->seek_req) {
4.         is->seek_pos = pos;
5.         is->seek_flags = rel < 0 ? AVSEEK_FLAG_BACKWARD : 0;
6.         is->seek_req = 1;
7.     }
8. }
```

이제 decode_thread 를 지나서 실제로 탐색을 수행하는 부분을 보겠습니다. 소스파일 안에 "seek stuff goes here"라고 마킹을 해두었습니다. 우리는 이것을 거기에 넣습니다.

탐색의 중심부분에는 [av_seek_frame](#) 함수가 있습니다. 이 함수는 format context, stream, timestamp, 아규먼트 같은 플래그 세트들을 얻을수 있습니다. 이 함수는 여러분이 지정한 timestamp 를 찾을것입니다. timestamp 의 유닛은 함수에 보내는 stream 의 time_base 입니다. 하지만 여러분은 이것에 스트림을 보내서는 안됩니다.(indicated by passing a value of -1). 만약 여러분이 그렇게 하면 time_base 는 avcodec 의 내부 timestamp 유닛이 되거나 1000000fps 가 될것입니다. 이것이 seek_pos 를 설정할때 AV_TIME_BASE 당 포지션을 곱했던 이유입니다.

만일 여러분이 [av_seek_frame](#) -1 을 스트림에 보내면 몇몇 파일에서 문제에 부딪힐수 있습니다. 그래서 우리는 파일에서 첫번째 스트림을 고집어 내서 이것을 ac_seek_frame 에 보낼것입니다. 우리가 timestamp 를 새로운 유닛으로 rescale 해야한다는것을 잊으면 안됩니다.

```
1. if(is->seek_req) {
2.     int stream_index= -1;
3.     int64_t seek_target = is->seek_pos;
4.
5.     if (is->videoStream >= 0) stream_index = is->videoStream;
6.     else if(is->audioStream >= 0) stream_index = is->audioStream;
7.
8.     if(stream_index>=0){
9.         seek_target= av_rescale_q(seek_target, AV_TIME_BASE_Q,
10.             pFormatCtx->streams[stream_index]->time_base);
11.     }
12.     if(av_seek_frame(is->pFormatCtx, stream_index,
13.         seek_target, is->seek_flags) < 0) {
```

```

14.     fprintf(stderr, "%s: error while seeking\n",
15.             is->pFormatCtx->filename);
16. } else {
17.     /* handle packet queues... more later... */

```

[av_rescale_q](#)(a,b,c) 는 하나의 timestamp 를 다른것으로 rescale 하는 함수 입니다. 이걸 $a*b/c$ 로 계산하는데 이함수는 그 계산이 오버플로워 될 수 있기때문에 필요합니다. AV_TIME_BASE_Q 는 AV_TIME_BASE 의 분수 버전입니다. 이것은 꽤 다른 차이가 있습니다. : $AV_TIME_BASE * time_in_seconds = avcodec_timestamp$ 그리고 $AV_TIME_BASE_Q * avcodec_timestamp = time_in_seconds$ (AV_TIME_BASE_Q 는 사실 하나의 [AVRational](#) 오브젝트 입니다. 그래서 여러분은 avcodec 에서 이것을 제어하려면 특별한 q 함수를 써야 합니다.).

Flushing our buffers

그래서 우리는 올바른 탐색을 설정합니다. 하지만 아직 끝난것이 아닙니다. 우리가 갖고 있는 하나의 큐를 packet 들을 모을수 있도록 설정해야 합니다. 지금 우리는 다른 장소에 있으면 우리는 큐를 비우거나 영상에서 찾기를 중지해야 합니다. 그뿐 아니라 avcodec 은 자체적으로 내부 버퍼를 갖고있고 그것 역시 각 쓰레드에 의해서 비워져야 합니다.

일단 우리는 패킷 큐를 비우는 함수를 작성합니다. 그리고 오디오, 비디오 쓰레드에 avcodec 의 내부 버퍼를 비우하는 명령을 하는 방법을 정의해야 합니다.

우리는 그것을 비운다음 큐에 특별한 패킷을 넣음으로써 할 수 있습니다. 그리고 그 특별한 패킷을 찾으면 그것들은 자동으로 버퍼를 비울것입니다. 다음 간단함 flush 함수를 보겠습니다.

```

1. static void packet_queue_flush(PacketQueue *q) {
2.     AVPacketList *pkt, *pkt1;
3.
4.     SDL_LockMutex(q->mutex);
5.     for(pkt = q->first_pkt; pkt != NULL; pkt = pkt1) {
6.         pkt1 = pkt->next;
7.         av_free_packet(&pkt->pkt);
8.         av_freep(&pkt);
9.     }
10.    q->last_pkt = NULL;
11.    q->first_pkt = NULL;
12.    q->nb_packets = 0;
13.    q->size = 0;
14.    SDL_UnlockMutex(q->mutex);
15. }

```

이제 큐는 비워졌습니다. 그리고 우리의 "flush packet"을 넣습니다. 이전에 일단 우리는 그게 무엇인지를 선언합니다.

```

1. AVPacket flush_pkt;
2.

```

```

3. main() {
4.   ...
5.   av_init_packet(&flush_pkt);
6.   flush_pkt.data = "FLUSH";
7.   ...
8. }

```

이제 큐에 패킷을 넣습니다.

```

1. } else {
2.   if(is->audioStream >= 0) {
3.     packet_queue_flush(&is->audioq);
4.     packet_queue_put(&is->audioq, &flush_pkt);
5.   }
6.   if(is->videoStream >= 0) {
7.     packet_queue_flush(&is->videoq);
8.     packet_queue_put(&is->videoq, &flush_pkt);
9.   }
10. }
11. is->seek_req = 0;

```

(This code snippet also continues the code snippet above for decode_thread.) 우리는 packet_queue_put 을 바꿔야 합니다. 그래서 special flush 패킷을 복사하지 않습니다.

```

1. int packet_queue_put(PacketQueue *q, AVPacket *pkt) {
2.
3.   AVPacketList *pkt1;
4.   if(pkt != &flush_pkt && av_dup_packet(pkt) < 0) {
5.     return -1;
6.   }

```

그리고 오디오쓰레드와 비디오쓰레드에서 packet_queue_get 한다음 바로 [avcodec_flush_buffers](#) 로 와서 이것을 넣습니다.

```

1. if(packet_queue_get(&is->audioq, pkt, 1) < 0) {
2.   return -1;
3. }
4. if(packet->data == flush_pkt.data) {
5.   avcodec_flush_buffers(is->audio_st->codec);
6.   continue;
7. }

```

전체 코드 조각은 정확히 비디오 쓰레드와 같습니다. 오디오가 비디오로 대체되고 있는데요. 다 되었습니다. 여러분의 플레이어를 컴파일 해보세요

```
gcc -o tutorial07 tutorial07.c -lavutil -lavformat -lavcodec -lz -lm`SDL_CONFIG --cflags --libs`
```

다음에는 약간의 수정만 해보도록 하겠습니다. ffmpeg 가 지원하는 소프트웨어 스케일링으로 작은 샘플링을 체크해보겠습니다.

Tutorial 08: Software Scaling

libswscale

ffmpeg 는 최근 새 인터페이스를 추가했습니다. libswscale 이라는 이미지 스케일을 제어하는 것어요. 사실 전에 우리가 [img_convert](#) 를 이용해서 RGB 를 YUV 로 바꾸었던것입니다. 이제 우리는 새 인터페이스로 그것을 대신 하겠습니다. 이 새로운 인터페이스는 더 모듈화되고, 더 빠르고, MMX 최적화를 해서 더 믿을만 합니다. 한마디로 스케일링하는데 적합하다는 것이죠.

우리는 그동안 기본함수인 [sws_scale](#) 을 사용할것입니다. 하지만 우선 우리는 SwsContext 라고 부르는 것을 설정해야 합니다. 이것은 우리가 원하는대로 바꾼것을 컴파일 하게 해주고 그것을 다음에 [sws_scale](#) 에 넘겨줍니다. 이것은 SQL 이나 파이선의 compiled regexp 에서 몇몇의 준비된 문장입니다. 이 context 를 준비하려면 [sws_getContext](#) 함수를 사용합니다. 이 함수는 소스의 가로 세로 값과 우리가 원하는 가로 세로 값 그리고 소스의 포맷과 원하는 포맷을 입력해야 합니다. 그리고 몇가지 다른 옵션들과 플래그도 넣어줘야 합니다. 그러면 SwsContext 에 그것을 건네주지 않고 [sws_scale](#) 을 [img_convert](#) 와 비슷한 방법으로 사용합니다.

```
1. #include <ffmpeg/swscale.h> // include the header!
2.
3. int queue_picture(VideoState *is, AVFrame *pFrame, double pts) {
4.
5.     static struct SwsContext *img_convert_ctx;
6.     ...
7.
8.     if(vp->bmp) {
9.
10.        SDL_LockYUVOverlay(vp->bmp);
11.
12.        dst_pix_fmt = PIX_FMT_YUV420P;
13.        /* point pict at the queue */
14.
15.        pict.data[0] = vp->bmp->pixels[0];
16.        pict.data[1] = vp->bmp->pixels[2];
17.        pict.data[2] = vp->bmp->pixels[1];
18.
19.        pict.linesize[0] = vp->bmp->pitch[0];
20.        pict.linesize[1] = vp->bmp->pitch[2];
21.        pict.linesize[2] = vp->bmp->pitch[1];
```

```

22.
23. // Convert the image into YUV format that SDL uses
24. if(img_convert_ctx == NULL) {
25.     int w = is->video_st->codec->width;
26.     int h = is->video_st->codec->height;
27.     img_convert_ctx = sws_getContext(w, h,
28.                                     is->video_st->codec->pix_fmt,
29.                                     w, h, dst_pix_fmt, SWS_BICUBIC,
30.                                     NULL, NULL, NULL);
31.     if(img_convert_ctx == NULL) {
32.         fprintf(stderr, "Cannot initialize the conversion context!\n");
33.         exit(1);
34.     }
35. }
36. sws_scale(img_convert_ctx, pFrame->data,
37.           pFrame->linesize, 0,
38.           is->video_st->codec->height,
39.           pict.data, pict.linesize);

```

이제 우리가 새롭게 scaling 한 것이 나옵니다.

여러분은 이것을 이용해서 libswscale 이 할수 있는 여러가지 아이디어를 얻으실 수 있을것입니다.

컴파일 해볼까요?

```
gcc -o tutorial08 tutorial08.c -lavutil -lavformat -lavcodec -lz -lm `sdl-config --cflags --libs`
```

드디어 여러분은 1,000 줄 만에 근사한 무비 플레이어를 만들게 되었습니다.

축하합니다. ☺

What now?

우리는 플레이어를 만들어 보았습니다. 하지만 완벽하게 멋진것은 아니었습니다. 우리는 많은 속임수를 사용했고 다른 기능들을 사용하였습니다.

- 에러처리. 우리코드에서 에러처리는 굉장히 후집니다. 아마 훨씬 개선될수 있을것입니다.
- 일시정지. 우리 플레이어는 일시정지 기능이 없습니다. 이 기능은 일반적으로 유용한 기능입니다. 우리는 우리 큰 구조체 안에 내부의 멈춤 변수를 이용해서 구현할 수 있습니다. 오디오와 비디오 그리고 디코드 쓰레드는 이것을 체크하고 아무것도 출력하지 않으면 됩니다. 또한 av_read_play 를 하용해서 네트워크 지원을 할수 있습니다. 이것은 간단한 설명이고 여러분 스스로 채워야할 것입니다. 더 배우고 싶다면 한번 구현하는것에 도전해 보세요 ffmpeg.c 에 힌트가 있을 것입니다.
- 비디오 하드웨어 지원. [Martin's old tutorial](#) 에 있는 Frame Grabbing 섹션을 찾아보세요
- 바이트 단위 탐색. 여러분이 만약에 초당 검색을 사용하는 것 보다 바이트단위 검색을 하고 싶다면, 이 방법은 timestamp 가 없는 비디오 파일에서 더 정확한 탐색 효과를 보여줄 것입니다. VOB 파일처럼요

- 프레임 낮추기. 비디오가 너무 뒤로 가있다면 여러분은 짧은 refresh rate 를 설정하여 다음 프레임을 드롭시키시면 됩니다.
- 네트워크 지원. 지금 만든 플레이어는 네트워크 지원을 하지 못합니다.
- YUV 파일같은 RAW 비디오를 지원. 우리 플레이어에서 YUV 파일 같이 사이즈나 time_base 를 추측할수 없는 RAW 비디오 파일을 지원하고 싶다면 몇가지 옵션이 있습니다.
- 풀 스크린
- 다양한 옵션. 예) 다른 그림 포맷: ffmpeg.c 를 보세요.
- 다른 hand-wavy things. for example, the audio buffer in our struct should be declared aligned.

ffmpeg 에 더 많은 정보를 얻고 싶으시다면, 우리는 일부분만 다뤘기 때문에 다음번에는 어떻게 multimedia encode 를 할수 있는지 공부해 보세요 ffmpeg distribution 의 output_example.c 를 보시는것이 좋은 출발점이 될 것입니다.

저는 이 튜토리얼이 재미있고 유익했기를 바랍니다. 버그나 다른 제안 또는 불만이나 칭찬은 dranget at gmail dot com 으로 보내주세요

Links:

[ffmpeg home page](#)

[Martin Bohme's original tutorial](#)

[libSDL](#)

[SDL Documentation](#)