Zagazig University
Faculty of Engineering

Computer Programming

AsciiPlayer

A Terminal-based Video Player

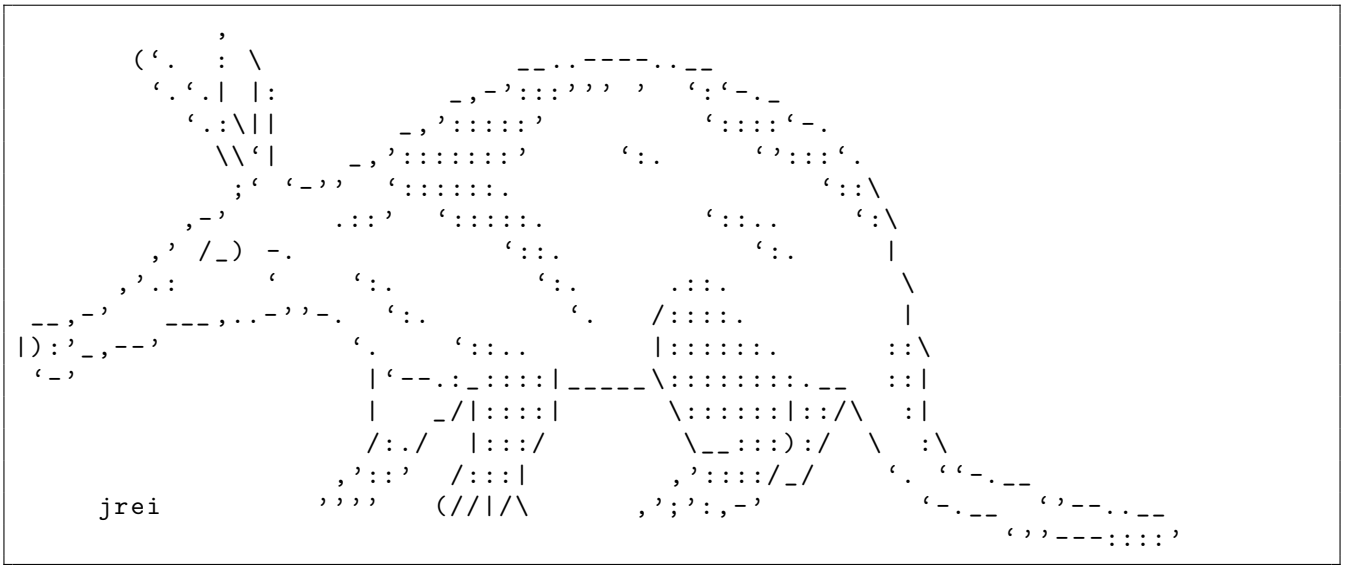
Computer and Systems Engineering
Level 100

Dec 2024

Contents

1 Introduction

ASCII art is a graphic design technique that uses computers for presentation and it consists of pictures pieced together from the 95 printable (from a total of 128) characters defined by the ASCII Standard from 1963 and ASCII compliant character sets with proprietary extended characters (beyond the 128 characters of standard 7-bit ASCII)



Listing 1: An ASCII art for Ant bear from https://www.asciiart.eu/animals/aardvarks

It's a cool idea and started from old TTY machines (or teletypewriter) as early as 1923. Nowadays, Linux machines and Unix-Like systems use the terminal in daily uses, where we cannot type or entering anything else than ASCII characters. So, if we need to see or watch something on this terminal envirommenent, will be impossible without using complex terminal emulators like 'kitty' or 'alacritty' that handle the image previewing inside it.

This is our idea, A tool that make you able to play your videos inside the terminal without need to any bloatwares or stupid things that make your machine more heavier!

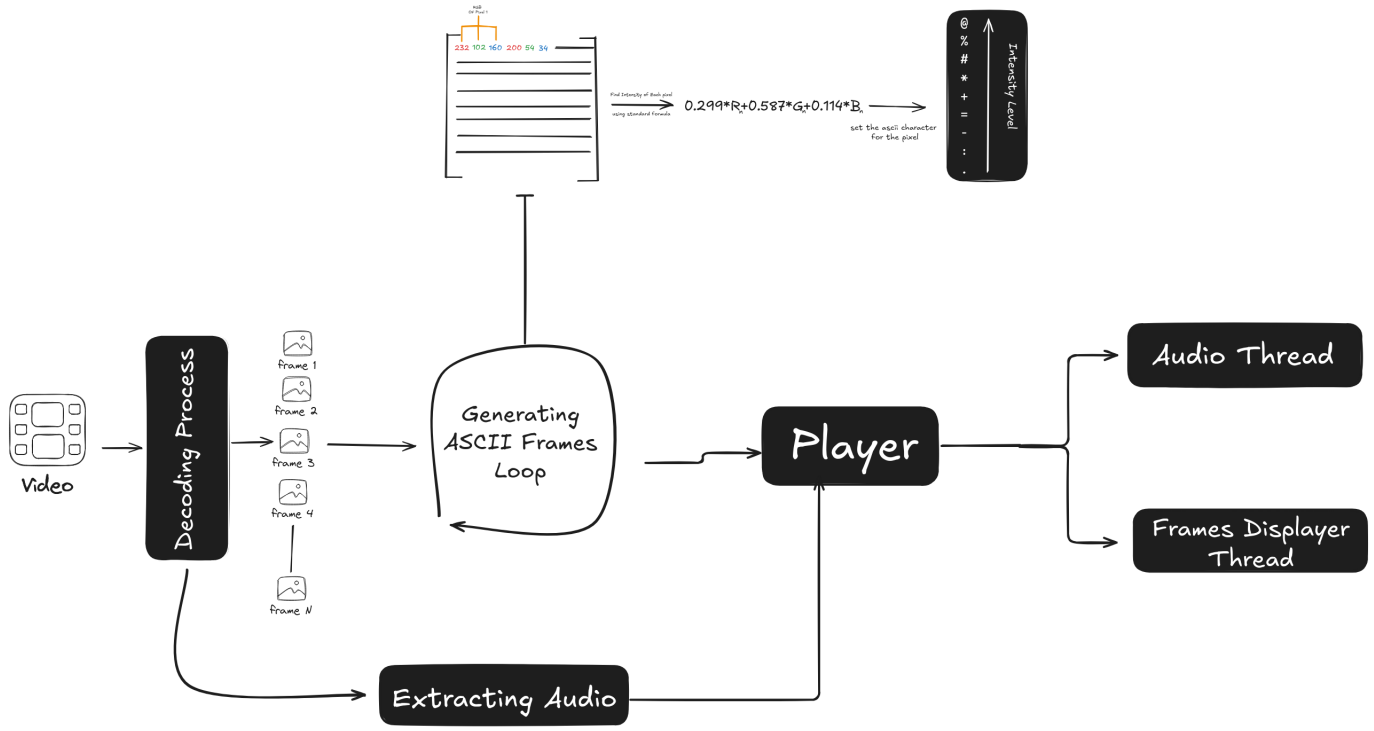
2 Project Architecture

The project architecture centers around transforming a video into an ASCII art animation while preserving audio playback. The process begins with decoding the input video, where the frames and audio are separated. The video frames are then converted into ASCII representations using a pixel intensity calculation. This intensity is derived from the RGB values of each pixel using the formula:

$$I = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

The resulting intensity values are mapped to characters, ranging from lighter ones like `.` and `:` to darker, denser symbols such as `#` and `@`. This creates a visual hierarchy where brighter parts of the frame appear "lighter" and darker regions use heavier symbols.

While this ASCII conversion happens, the audio extracted earlier is handled separately. Both the audio and the ASCII frames are sent to a player component, which synchronizes their playback. The player ensures two critical things: the audio runs smoothly, and the frames are displayed sequentially without lag, creating the effect of an animated video.



To achieve this synchronization, the system runs two threads. One thread is responsible for playing the extracted audio, while the other handles the display of ASCII frames in real time. The overall result is a cohesive ASCII-based video playback experience, complete with audio.

3 Behind the scenes

Here, We will talk about the parts of our project and how it works behind the scenes.

3.1 Pixel, The Atom of what we see

Pixel is the smallest unit of the frame of image or video that we see and the most important thing in the frame, everything in the frame contains pixels beginning from the text to the smallest drop seen in the photo.

To work with this concept in C, we need to abstract it by making a struct for it containing the important data about it. Struct pixel is used to represent RGB colors in 16-bit unsigned int representing red (r), green (g) and blue (r) colors.

```
typedef struct
{
    uint16_t r;
    uint16_t g;
    uint16_t b;
} pixel;
```

Now, We can do the important operations that we need. In first, We need to calculate the intensity level of this pixel to select the suitable ASCII character for it using this formula:

$$I = 0.299 * R + 0.587 * G + 0.114 * B$$

```
uint16_t pixel_intensity(pixel px)
{
    uint16_t intensity;

    intensity = (0.299 * px.r + 0.587 * px.g + 0.114 * px.b);

    return intensity; // return intensity as a value
}
```

Converting each pixel into ASCII characters depends on the intensity of each pixel and can be done by calculating index of the suitable:

```
char pixel_to_ascii(pixel px)
{
    static char ASCII_CHARS[] = " .,:;-=+*#%@";

    uint16_t intensity, index;

    intensity = pixel_intensity(px); // calculating intensity of the pixel

    // defining Ascii length as the string length of ASCII characters minus 1
    uint16_t ascii_length = strlen(ASCII_CHARS) - 1;

    index = intensity / 255.0 * (ascii_length); // calculating index

    return ASCII_CHARS[index];
}
```

3.2 The Frame, What we see

The frame is the buliding block of a video, as a video is just a multiple frames played one after another very quickly. We begin by giving each frame a unique id and specifying the line height of each frame. Then we define the width and height of the frames and provide each frame a buffer keeping its data.

```
typedef struct
{
    uint64_t id;
    int wrap;
    int width;
    int height;
    unsigned char *buf;
} frame;
```

After defining the frame, we need to convert it to ASCII format and save it to the disk. And this happens by a function called *frame_ascii_write_to*. This function converts each pixel in a frame to ASCII format.

```
int frame_ascii_write_to(frame *frame, char *filename)
{
    FILE *file = fopen(filename, "wb");

    for (int y = 0; y < frame->height; y++)
    {
        for (int x = 0; x < frame->width; x++)
        {
            pixel px = pixel_new(
                frame->buf[y * frame->wrap + x],
                frame->buf[y * frame->wrap + x + 1],
                frame->buf[y * frame->wrap + x + 2]);
            char ascii_char = pixel_to_ascii(px);

            fputc(ascii_char, file);
            fputc(ascii_char, file);
            fputc(ascii_char, file);
        }
        fputc('\n', file);
    }

    fclose(file);
}
```

3.3 Video decoding

After a lot of abstractions, We will start in our first step, The video decoding. We use *libavcodec* for this operation, it's a famous library, and it's used in a lot of video editors programs like *kdenlive*. Also, the famous tool in videos, *FFmpeg*.

We will define our struct to start decoding that contains important stored data for this operation.

```
typedef struct
{
    const char *src, *out;
    AVFormatContext *fmt_ctx;
    const AVCodec *codec;
    AVCodecContext *codec_ctx;
    AVStream *video_stream;
    AVStream *audio_stream;
    int video_stream_index;
    int audio_stream_index;
    AVPacket *pkt;
    AVFrame *frame;
} video;
```

Now, We will need to initilize this struct

```
video *video_new(const char *src, const char *out)
{
    video *vid = malloc(sizeof(video));

    vid->src = src;
    vid->out = out;
    vid->fmt_ctx = NULL;
    vid->codec = NULL;
    vid->codec_ctx = NULL;
    vid->video_stream = NULL;
    vid->audio_stream = NULL;
    vid->video_stream_index = -1;
    vid->audio_stream_index = -1;
    vid->pkt = NULL;
    vid->frame = NULL;

    if (avformat_open_input(&vid->fmt_ctx, vid->src, NULL, NULL) < 0)
    {
        ERROR("Could not open input file\n");
        goto fail;
    }

    if (avformat_find_stream_info(vid->fmt_ctx, NULL) < 0)
    {
        ERROR("Could not find stream information\n");
        goto fail;
    }

    if (video_set_streams(vid) == -1)
    {
        ERROR("Couldn't find video stream.");
        goto fail;
    }

    if (video_find_decoder(vid) == -1)
    {
        ERROR("Could not find the decoder");
        goto fail;
    }

    INFO("Decoder was found, dec={%s}, type={%u}", vid->codec->long_name, vid
       ->codec_ctx->codec_type);
    vid->frame = av_frame_alloc();
    vid->pkt = av_packet_alloc();
    if (!vid->frame || !vid->pkt)
    {
        ERROR("Could not allocate frame or packet\n");
        goto fail;
    }

    return vid;

fail:
    ERROR("Couldn't create the video struct");
    return NULL;
}
```

As you notice, we set the streams during this initilization using *video_set_streams*, making a loop to find the video and audio streams

```
static int video_set_streams(video *vid)
{
    for (unsigned int i = 0; i < vid->fmt_ctx->nb_streams; i++)
    {
        if (vid->fmt_ctx->streams[i]->codecpar->codec_type ==
            AVMEDIA_TYPE_VIDEO)
        {
            vid->video_stream_index = i;
            vid->video_stream = vid->fmt_ctx->streams[i];
            INFO("video stream was found, i={%i}", vid->video_stream_index);
        }
    }

    if (vid->video_stream_index == -1)
    {
        return -1;
    }
    return 0;
}
```

and find the decoder using *video_find_decoder*

```
static int video_find_decoder(video *vid)
{
    vid->codec = avcodec_find_decoder(vid->video_stream->codecpar->codec_id);
    if (vid->codec == NULL)
    {
        ERROR("Codec not found");
        return -1;
    }

    vid->codec_ctx = avcodec_alloc_context3(vid->codec);
    if (vid->codec_ctx == NULL)
    {
        ERROR("Could not allocate codec context");
        return -1;
    }

    // Copy codec parameters from the input stream to the codec context
    if (avcodec_parameters_to_context(vid->codec_ctx, vid->video_stream->
        codecpar) < 0)
    {
        ERROR("Could not copy codec parameters to context");
        return -1;
    }

    // Open codec
    if (avcodec_open2(vid->codec_ctx, vid->codec, NULL) < 0)
    {
        ERROR("Could not open codec");
        return -1;
    }
    else
    {
        INFO("Codec was opened");
    }

    return 0;
}
```

Finally, We will need to decode the video. For that, We make a function that decodes it by passing a handler that handles the decoding operation

```
int video_decode_frames(video *vid, int (*handler)(AVCodecContext *codec_ctx,
    AVFrame *frame, AVPacket *pkt, AVFormatContext *fmt_ctx, AVStream *
    video_stream, const char *out))
{

    // Read frames from the file
    while (av_read_frame(vid->fmt_ctx, vid->pkt) >= 0)
    {
        if (vid->pkt->stream_index == vid->video_stream_index)
        {
            if (handler(vid->codec_ctx, vid->frame, vid->pkt, vid->fmt_ctx,
                vid->video_stream, vid->out) == -1)
            {
                FATAL("Couldn't handle the frame #%i", vid->codec_ctx->
                    frame_num);
                goto fail;
            }
        }
        av_packet_unref(vid->pkt);
    }
    return 0;

fail:
    return -1;
}
```

3.4 It seems all are ready. Let's generate ASCII Frames!

We will start with defining a struct for that that contains the video source, output directory, and the video struct to decode it.

```
typedef struct
{
    const char  *src;
    const char  *dest;
          video *vid;
} ascii_video_gen;
```

Now, Let's generate the video. It's a simple function that runs the generator.

```
int ascii_video_gen_run(ascii_video_gen *gen)
{
    video_decode_frames(gen->vid, decode_handler);

    // Flush the decoder
    decode_handler(gen->vid->codec_ctx, gen->vid->frame, NULL, gen->vid->
        fmt_ctx, gen->vid->video_stream, gen->dest);
    return 0;
}
```

What's *deocde_handler* ? As we talked in video decoding, We need to pass a decoding handler to *video_decode_frames* to convert the video frames into ASCII format.

```
1
2  int decode_handler(AVCodecContext *dec_ctx, AVFrame *av_frame, AVPacket *
       av_pkt, AVFormatContext *av_fmt_ctx, AVStream *av_video_stream, const char
       *out_dir)
3  {
4      char buf[1024];
5      int ret;
6      ret = avcodec_send_packet(dec_ctx, av_pkt);
7      if (ret < 0)
8      {
9          ERROR("Error sending a packet for decoding, ret={%i}", ret);
10         return -1;
11     }
12
13     AVRational framerate =
14         av_guess_frame_rate(av_fmt_ctx, av_video_stream, av_frame);
15
16     double fps = framerate.num / framerate.den;
17     double duration = av_fmt_ctx->duration / (double)AV_TIME_BASE;
18     int frames_count = fps * duration;
19
20     while (ret >= 0)
21     {
22         ret = avcodec_receive_frame(dec_ctx, av_frame);
23         if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
24             goto done;
25         else if (ret < 0)
26         {
27             ERROR("Error during decoding");
28             return -1;
29         }
30
31         int frame_n = dec_ctx->frame_num;
32         int w = av_frame->width, h = av_frame->height;
33         snprintf(buf, sizeof(buf), "%s/%ld.ascii", out_dir, frame_n);
34
35         INFO("(%.1f\%) Saving frame %ld/%ld: path={\"%s\"}, res={%ix%i}",
               frame_n / (float)frames_count * 100, frame_n, frames_count, buf,
               av_frame->width, av_frame->height);
36
37         frame *fm = frame_new(frame_n, av_frame->data[0], av_frame->linesize
               [0], w, h);
38
39         if (!fm)
40         {
41             ERROR("Creating frame#%i was failed", dec_ctx->frame_num);
42         }
43
44         if (frame_ascii_write_to(fm, buf) == -1)
45         {
46             ERROR("Couldn't write frame <%i> into %s", dec_ctx->frame_num, buf
                   );
47             return -1;
48         }
49         frame_free(fm);
50     }
51 done:
52     return 0;
53 }
```

3.5 Audio Extraction

Alright, We converted the video into ASCII. Now, We need to hear something. For that, the Audio extraction is important to play the sound with the video. This happens using *FFmpeg* tool by running it as a child process.

```c
int video_extract_audio(video *vid, const char *output_file)
{
    if (!vid || !vid->src || !output_file)
    {
        ERROR("Invalid input parameters");
        return -1;
    }

    pid_t pid = fork();
    if (pid == -1)
    {
        ERROR("Failed to fork process");
        return -1;
    }
    else if (pid == 0)
    {
        execl("/usr/bin/ffmpeg",
              "ffmpeg",                // argv[0]
              "-i", vid->src,          // input file
              "-vn",                   // no video
              "-acodec", "pcm_s16le", // PCM 16-bit little-endian
              "-ar", "44100",          // 44.1kHz sample rate
              "-ac", "2",              // 2 audio channels
              output_file,             // output file
              NULL);

        ERROR("Failed to execute ffmpeg");
        exit(EXIT_FAILURE);
    }
    else
    {
        int status;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status))
        {
            int exit_status = WEXITSTATUS(status);
            if (exit_status == 0)
            {
                INFO("Audio extraction successful");
                return 0;
            }
            else
            {
                ERROR("FFmpeg process failed with exit code %d", exit_status);
                return -1;
            }
        }
        else
        {
            ERROR("FFmpeg process did not exit normally");
            return -1;
        }
    }
}
```

3.6 Specifications

Specifications is a structure designed to store data about a video. It holds various details such as

1. **Frames Count:** The total number of frames in the video.
2. **Frames Per Second:** The playback speed, indicating how many frames are shown per second.
3. **Duration:** The total length of the video in seconds.
4. **Resolution:** The dimensions of the video in pixels.
5. **Audio:** A flag indicating whether the video includes an audio track.

Here's how we defined the Specifications struct:

```
typedef struct
{
  uint32_t   frames_count;
  double     fps;
  double     duration;
  uint32_t   width;
  uint32_t   height;
  bool       audio;
} specs;
```

3.7 Player

The Player is the most important part of the whole program as it's where we see and hear all of the work we've done so far. It takes the path to the generated frames and also the specifications. In addition, we play the audio as a child process. We store its parent id in the player.

```
typedef struct
{
    const char *src;
    specs *specs;
    pid_t audio_pid;
} player;
```

The Player has control over the audio, as it can send either a **SIGSTOP** to pause the audio or a **SIGCONT** to resume it.

```
static void audio_stop(pid_t audio_pid)
{
    if (audio_pid > 0)
    {
        int result = kill(audio_pid, SIGSTOP);
        if (result == -1)
        {
            ERROR("Failed to stop audio playback: %s", strerror(errno));
        }
        else
        {
            is_audio_stopped = true;
        }
    }
}
static void audio_resume(pid_t audio_pid)
{
    if (audio_pid > 0)
    {
        int result = kill(audio_pid, SIGCONT);
        if (result == -1)
        {
            ERROR("Failed to resume audio playback: %s", strerror(errno));
        }
        else
        {
            is_audio_stopped = false;
        }
    }
}
```

It also reads ASCII art frames from a file and displays them on an ncurses pad, erasing previous content.

```
static int print_frame(char *filename, WINDOW *frame_pad, int width, int
    height)
{
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        INFO("Error: Could not open file %s\n", filename);
        return -1;
    }

    werase(frame_pad);
    char line[width + 1];
    for (int y = 0; y < height; y++)
    {
        if (fgets(line, sizeof(line), file) == NULL)
            break;

        mvwprintw(frame_pad, y, 0, "%.*s", width, line);
    }

    fclose(file);
    return 0;
}
```

We loops through frames to display them, synchronize frame timing, and handle terminal resizing dynamically using a function called *player_video_run*.

In this function, We will start with specifying some information important to render the frames like *delay*, terminal dimensions, frame dimensions, offset to make the video at the center, etc.

```
double fps = ceil(player->specs->fps);
uint32_t frames_count = player->specs->frames_count;
uint32_t duration = player->specs->duration;

int delay = 1000000 / (2 * fps);

int width = player->specs->width * 4, height = player->specs->height;

INFO("Video dimensions: width = %i, height = %i\n", width, height);

int offset_y;
int offset_x;

int term_rows, term_cols;
int prev_term_rows, prev_term_cols;

getmaxyx(stdscr, prev_term_rows, prev_term_cols);
WINDOW *frame_pad = newpad(height, width);
if (!frame_pad)
{
    ERROR(" Could not create frame pad\n");
    return -1;
}
```

After that, we will enter the displaying frames loop by printing each frame file into the *frame_pad*.

```
for (int i = 1; i <= player->specs->frames_count; i++)
{
    char filename[128];

    snprintf(filename, sizeof(filename), "%s/%i.ascii", player->src, i);
    print_frame(filename, frame_pad, width, height);
    if (term_rows != prev_term_rows || term_cols != prev_term_cols)
    {
        clear();
        refresh();
    }
    prefresh(frame_pad, 0, 0, offset_y, offset_x, offset_y + height - 1,
       offset_x + width - 1);

    usleep(delay);
    prev_term_rows = term_rows;
    prev_term_cols = term_cols;
}
```

but Before printing the frame, we need to check if the dimensions of the terminal suits video dimensions or will ask the user to resize the terminal.

```c
do
{
    getmaxyx(stdscr, term_rows, term_cols);

    offset_y = (term_rows - height) / 2;
    offset_x = (term_cols - width) / 2;

    if (offset_y >= 0 && offset_x >= 0)
    {
        if (player->specs->audio)
        audio_resume(player->audio_pid);
        break;
    }
    else
    {

        clear();
        mvprintw(term_rows / 2, (term_cols - 30) / 2, "Resize terminal to %dx%d", width, height);
        mvprintw(term_rows / 2 + 1, (term_cols - 30) / 2, "Current: %dx%d", term_cols, term_rows);
        refresh();

        usleep(10000);
        if (player->specs->audio)
        audio_stop(player->audio_pid);
    }
} while (1);
```

Now, We need to play the audio that we extracted by creating a child process, *aplay*, a command line sound player.

```c
int player_audio_run(player *player)
{
    char audio_file[1024];
    snprintf(audio_file, sizeof(audio_file), "%s/audio.wav", player->src);
    player->audio_pid = fork();

    if (player->audio_pid == -1)
    {
        ERROR("Failed to fork process");
        return -1;
    }
    else if (player->audio_pid == 0)
    {
        execl("/usr/bin/aplay",
              "aplay",     // argv[0]
              "-q",        // quiet mode (suppress output)
              audio_file, // audio file to play
              NULL);
        ERROR("Failed to execute aplay"); exit(EXIT_FAILURE);
    }
    else
    {
        int status;
        waitpid(player->audio_pid, &status, 0);
        // Some Error Handling...
    }
}
```

3.8 Let's combine all together!

In the main funciton, we receive process arguments for video source and output directory, create the video struct, generate the ASCII video, extract the audio, and specifications.

```
 const char *src, *out;

if (argc <= 2)
{
    fprintf(stderr, "Usage: %s <input file> <output file>\n", argv[0]);
    exit(0);
}
src = argv[1];
out = argv[2];

video *vid = video_new(src, out);

ascii_video_gen *gen = ascii_viden_gen_new(src, out, vid);
ascii_video_gen_run(gen);

char buf[128];
snprintf(buf, sizeof(buf), "%s/audio.wav", out);
int audio_ext_ret = video_extract_audio(vid, buf);
if (audio_ext_ret != 0)
{
    ERROR("Couldn't extract the audio and play it.");
}
AVRational framerate =
    av_guess_frame_rate(vid->fmt_ctx, vid->video_stream, vid->frame);

double fps = framerate.num / framerate.den;
double duration = vid->fmt_ctx->duration / (double)AV_TIME_BASE;

int frames_count = fps * duration;
specs *specs = specs_new(frames_count, fps, duration, vid->video_stream->
   codecpar->width, vid->video_stream->codecpar->height, audio_ext_ret == 0 ?
   true : false);
```

Now we will run the player and audio together using multi-threading by creating a thread for each one and creating a routine function that wraps the main function to pass it to the thread creator.

```
void *audio_routine(void *p)
{
    return (void *)player_audio_run((player *)p);
}
void *video_routine(void *p)
{
    return (void *)player_video_run((player *)p);
}
```

```
player *player = player_new(out, specs);

pthread_t th_video;
pthread_t th_audio;

void *ret_audio, *ret_video;

pthread_create(&th_video, NULL, video_routine, (void *)player);
audio_ext_ret == 0 &&
    pthread_create(&th_audio, NULL, audio_routine, (void *)player);

pthread_join(th_video, &ret_video);

audio_ext_ret == 0 &&
    pthread_join(th_audio, &ret_audio);

if ((int)ret_video == -1)
{
    ERROR("player failed to play the video");
}
```

4 References

1. Source code: `https://github.com/hulxv/asciivideo`
2. FFmpeg Documentation: `https://ffmpeg.org/documentation.html`
3. ncurses Documentation: `https://invisible-island.net/ncurses/`
4. asciiart.eu: `https://www.asciiart.eu`

5 Team

We completed this project with teamworking and using *git* as a version control system to work together and make a plan to done our work.

Name	Section
Mohamed Emad Elsawy	2
Mahmoud Fathallah AbdelFatah	2
Mostafa Ahmed Abdelsalam Morsi	2
Mohamed Mahmoud Ramadan	2
Eyad Eslam Eltaher	1