An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Appendix A FFmpeg Libraries

A.1 Introduction

FFmpeg has been a popular open-source video library for processing videos. One can also make use of this library to make interesting graphics as discussed below. It can be downloaded from its official site at *http://ffmpeg.org/* . FFmpeg is a complete, cross-platform solution to record, convert and stream audio and video. The code is written in C. It provides executable programs for users to process and play videos. We do not intend to present the details of using their executables as they are well documented in their web site. We are more interested to learn to use their libraries so that we can incorporate their features and functions into our own applications such as graphics or video games. In this appendix, we make use of what we have learned about graphics and the libraries to embed a simple video player in a graphical application. The materials presented here are mostly obtained from the official FFmpeg web site (http://ffmpeg.org). Interested readers may obtain further information about FFmpeg directly from the site.

A.2 Getting and Compiling FFmpeg

You may download the source code from their web site directly (http://ffmpeg.org/download.html). At the time of this writing, their version is 0.5 and you will obtain from the site the zipped file "ffmpeg-0.5.tar.bz2". You may unzip it into a directory using the command:

```
$bunzip2 -c ffmpeg-0.5.tar.bz2 | tar xvf -
```

It creates the directory "ffmpeg-0.5". You may go into the directory to configure your installation. As an example, suppose you want to install the package in the directory "/apps/mpeg". You may do the configuration using the command:

```
$./configure --prefix=/apps/mpeg --enable-cross-compile
```

After the configuration, you are ready to compile and make the executables and libraries. However, you need the GNU Make utility version 3.81 or later. (You may check your system's make version by the command "make –version".) If your system's **make** is earlier than 3.81, you need to download version 3.81 or later from the GNU web site (*http://www.gnu.org/*). If the version is correct, simply type "make" to build FFmpeg. Then type "make install" to install all binaries and libraries built into "/apps/mpeg". After "make install", you should find the three directories:

```
$/apps/mpeg/bin
$/apps/mpeg/lib
$/apps/mpeg/share
$/apps/mpeg/include
```

Directory “bin” contains the executable programs, “ffmpeg”, “ffplay”, and “ffserver”. The program “ffmpeg” is a very fast video and audio converter, allowing you to grab from a live audio/video with command-line interface. The program “ffserver” is a streaming server for both audio and video, supporting several live feeds, streaming from files and time shifting. If you want to play a video, you need to use “ffplay”, which is a very simple and portable media player built using the FFmpeg libraries and the SDL library, and is mostly used as a testbed for various FFmpeg APIs. SDL here stands for Simple DirectMedia Layer, which is a cross-platform multimedia library designed to provide low level access to 3D hardware via OpenGL. For details, please refer to its official web site at *http://www.libsdl.org/*.

The libraries “libavcodec.a”, “libavdevice.a”, “libavformat.a”, and “libavutil.a” are in the directory “lib” and the corresponding include header-files are in the directory “include”. Therefore, in building your applications, your include-path should contain “/apps/mpeg/include” and your library-path should contain “/apps/mpeg/lib”. Actually, we also need another header file, “libswscale” for building some ffmpeg applications. This header directory is not built by default in the installation process because it requires a different distribution license from other ffmpeg libraries. To create this directory in the installation process, you need to choose a special option in the configuration stage. To simplify the process, we just simply copy these files manually to the appropriate directories:

```
$mkdir /apps/mpeg/include/libswscale
$cp /apps/ffmpeg-0.5/libswscale/*.h /apps/mpeg/include/libswscale/.
```

Note that the library “libswscale.a” does **not** exist. Its functions can be found in the library “libavcodec.a”. Their usages are to replace the “imgconvert” functions of earlier ffmpeg versions. If you do not like the“libswscale” functions, simply use “imgconvert”.

For the convenience of explanation, let us make the directories “/apps/mpeg/work” and “/apps/mpeg/data”, where we shall save our project programs and sample video files respectively.

A.3 Using FFmpeg Libraries

Opening the File

To use FFmpeg libraries, we have to include their header files. Since FFmeg is written in pure C and our programs are written in C/C++ with extension “.cpp”, we have to use the keyword “extern” in specifying the include:

```
extern "C" {
  #include <libavcodec/avcodec.h>
  #include <libavformat/avformat.h>
  #include <libswscale/swscale.h>
}
```

If we do not use "extern", the compiler will generate errors. Before opening any video file, we need to use **av_register_all**() to register all available file formats and codecs so that they will be used automatically when a file with the corresponding format/codec is opened:

```
av_register_all();
```

We only have to call this function once in our application. We now can actually open the video file:

```
AVFormatContext *pFormatCtx;
char filename[100];
.....

// Open video file
if(av_open_input_file(&pFormatCtx, filename, NULL, 0, NULL)!=0)
  return -1; // Couldn't open file
```

We provide the filename as input parameter to this function, which reads the file header, saving information about the file format in the **AVFormatContext** structure. The last three arguments are used to specify the file format, buffer size, and format options, but by setting them to NULL or 0, **libavformat** will auto-detect the correct parameter values. The information is returned via *pFormatCtx* which is a pointer pointing to the AVFormatContext structure the function has created; the structure contains the miscellaneous video and audio information such as author, timestamp, packet size and bit rate of the date in the file.

Next, we may retrieve the information from the stream and may dump the information to the screen:

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

// Dump information about file onto standard error
dump_format(pFormatCtx, 0, filename, 0);
```

The field *pFormatCtx->streams* is an array of pointers, of size *pFormatCtx->nb_streams*, each pointing to an **AVStream** structure. We can search through it to find a video stream:

```
//Find the first video stream
videoStream=-1;
for(int i=0; i<pFormatCtx->nb_streams; i++) {
  if(pFormatCtx->streams[i]->codec->codec_type==CODEC_TYPE_VIDEO){
     videoStream=i;
     break;
   }
}

if(videoStream==-1)
   return -1; // Didn't find a video stream

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

The information of the codec used to enclose the video stream is referred to as "codec context", which contains all the information about the codec that the stream is using. Now we have a pointer to the "codec context", and we need to find the actual codec to open it:

```
  AVCodec *pCodec;
  // Find the decoder for the video stream
  pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
  if(pCodec==NULL) {
    fprintf(stderr, "Unsupported codec!\n");
    return -1; // Codec not found
  }
  //open codec
  if(avcodec_open(pCodecCtx, pCodec)<0)
    return -1; // Could not open codec
```

Decoding the Stream

We have found the video stream and know about its codec context. We need to allocate some memory to buffer the data of one or more frames so that we can further process them. We shall use the same producer-consumer concept, where a producer thread produces data and a consumer thread consumes data, to process the decoded data. For instance, we can allocate memory to buffer 4 frames:

```
  int width = pCodecCtx->width;
  int height = pCodecCtx->height;

  char *buf[4];
  for ( int i = 0; i <  4; ++i ) {
    buf[i] = ( char * ) malloc ( width * height * 3 );
    if ( buf[i] == NULL )
      return -2;
 }
```

In this case the decoder is our producer and the player is our consumer. The decoder thread just needs to decode data and put them in the buffer. The player thread reads the data from the buffer and displays them on the screen. We can write a class **Video** to store the relevant decoding information of the file:

```
class Video {
private:
  SDL_Thread *producer, *consumer;

public:
  SDL_Surface *screen;
  unsigned long head;
  unsigned long tail;
  int width;
  int height;
  int x0;       //play position
  int y0;
  bool quit;
  AVCodecContext *pCodecCtx;
  AVFormatContext *pFormatCtx;
 int videoStream;

  ........
};
```

We use FFmpeg function **avcodec_alloc_frame**() to allocate a frame structure to hold one frame. Also, we need the data to be stored in RGB (or BGR) format so that we can use an SDL function to blit the data on the screen:

```
AVFrame *pFrame, *pFrameRGB;
if(pFrame==NULL)
  return -1;

// Allocate video frame buffer for holding one frame of data
pFrame=avcodec_alloc_frame();

// Allocate an AVFrame structure for holding one frame of RGB data
pFrameRGB=avcodec_alloc_frame();
if(pFrameRGB==NULL)
  return -1;
```

We have to convert our frame from its native format to RGB. Our decoder makes use of the FFmpeg function **avcodec_decode_video**() to decode the video stream and it uses **sws_getContext**() and **sws_scale**() to convert the image from its native format to RGB. In older FFmpeg versions, the function **img_convert**() is supplied to make the conversion, but now this function has been deprecated. We shall also use FFmpeg function **av_malloc**() to

temporarily allocate memory to buffer one frame of data and use **avpicture_fill**() to associate the frame with the newly allocated buffer. The following shows the decoder thread:

Program Listing A-1

```
//Producer Thread
int decoder ( void *data )
{
  Video *vi = ( Video * ) data;
  SDL_Surface *screen = ( SDL_Surface * ) vi->screen;
  AVCodecContext *pCodecCtx =  vi->pCodecCtx;
  .......
  int frameFinished;
  AVPacket packet;
  AVFrame *pFrame;
  // Allocate video frame
  pFrame=avcodec_alloc_frame();
  // Allocate an AVFrame structure
  AVFrame *pFrameRGB;
  pFrameRGB=avcodec_alloc_frame();
  if(pFrameRGB==NULL)
    return -1;
  uint8_t *buffer;
  int numBytes;
  // Determine required buffer size and allocate buffer
  numBytes=avpicture_get_size(PIX_FMT_RGB24, pCodecCtx->width,
                              pCodecCtx->height);
  while ( !vi->quit ) {
    if ( vi->tail >= vi->head + 4 ) {  //buffer full
        SDL_Delay ( 30 );
        continue;
    }
    //produce data
    if (av_read_frame(pFormatCtx, &packet)>=0) {
      // Is this a packet from the video stream?
      if(packet.stream_index==videoStream) {
        // Decode video frame
        avcodec_decode_video(pCodecCtx, pFrame, &frameFinished,
                             packet.data, packet.size);
        // Did we get a video frame?
        if(frameFinished) {
          // Convert the image from its native format to RGB
          SwsContext * encoderSwsContext = sws_getContext(
           pCodecCtx->width, pCodecCtx->height, pCodecCtx->pix_fmt,
           pCodecCtx->width, pCodecCtx->height, PIX_FMT_RGB24,
           SWS_BICUBIC, NULL,NULL,NULL);

          sws_scale( encoderSwsContext, pFrame->data,
           pFrame->linesize, 0, pCodecCtx->height, pFrameRGB->data,
           pFrameRGB->linesize);

          char *pb = vi->buf[vi->tail%4]; //points to buffer slot
          int size = vi->width * 3;

          //copy decoded data to our buffer
          for ( int y = 0; y < height; y++ ){
              char *pc =(char *)
                   (pFrameRGB->data[0]+y*pFrameRGB->linesize[0]);
              int k = 0;
```

```
              char *p = pb + y * size;
              for ( int i = 0; i < vi->width; ++i ) {
                p[k] = pc[k+2];
                p[k+1] = pc[k+1];
                p[k+2] = pc[k];
                k += 3;
              }
            }
            vi->tail++;
          }
        }
       // Free the packet that was allocated by av_read_frame
        av_free_packet(&packet);
      } else
          vi->quit = true;
  } //while
  //free resources here
  .......
  return 0;
}
```

After we exited the while loop near the end of the decoder, we need to free the resources allocated by FFmpeg functions. We again call the FFmpeg functions to free them:

```
// Free the RGB image
av_free(buffer);
av_free(pFrameRGB);

// Free the YUV frame
av_free(pFrame);

// Close the codec
avcodec_close(pCodecCtx);
// Close the video file
av_close_input_file(pFormatCtx);
```

When we initialize the SDL, its better to use the double buffering feature so that we can carry out the decoding and put the decoded data at the background. We later swap the background and foreground buffers to avoid any flickering effect:

```
//initialize video system
if ( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
    fprintf(stderr, "Unable to init SDL:%s\n",SDL_GetError());
    exit(1);
}

//set video mode of  with 24-bit pixels
screen = SDL_SetVideoMode(640, 480, 24, SDL_DOUBLEBUF);
if ( screen == NULL ) {
    fprintf(stderr, "Unable to set video:%s\n",SDL_GetError());
    exit(1);
}
```

Playing the Video Data

The player thread which sends the decoded data to the screen is our consumer. It 'consumes' the data generated by the decoder. It puts a frame on the screen at the position (x0, y0) specified by the user. A background buffer is created to hold a frame of RGB data. The data are sent to the screen by **SDL_BlitSurface**() and **SDL_UpdateRect**():

Program Listing A-2

```
//Consumer Thread
int player ( void *data )
{
  Video *vi = ( Video * ) data;
  SDL_Surface *screen = ( SDL_Surface * ) vi->screen;
  Uint32 prev_time, current_time;
  current_time = SDL_GetTicks();//ms since library starts
  prev_time = SDL_GetTicks();   //ms since library starts

  SDL_Surface *background;
  background = SDL_CreateRGBSurface(SDL_SWSURFACE,
                       vi->width, vi->height, 24, 0, 0, 0, 0);
  if ( background == NULL ) {
    fprintf(stderr, "Unable to set  video: %s\n",SDL_GetError());
    return -1;
  }
  while ( !vi->quit ) {
    if ( vi->head == vi->tail ){//buffer empty (data not available)
      SDL_Delay ( 30 );         //sleep for 30 ms
      continue;
    }
    //consumes the data
    background->pixels = vi->buf[vi->head%4];
    current_time = SDL_GetTicks();    //ms since library starts
    if (current_time - prev_time < 50)//20 fps ~ 50 ms / frame
      SDL_Delay ( 50 - (current_time - prev_time) );
    prev_time = current_time;

    SDL_Rect source_rect, dest_rect;
    source_rect.x = 0;              source_rect.y = 0;
    source_rect.w = vi->width;
```

```
    source_rect.h = vi->height;
    dest_rect.x = vi->x0;           dest_rect.y = vi->y0;
    dest_rect.w = vi->width;
    dest_rect.h = vi->height;

    SDL_BlitSurface(background, &source_rect, screen, &dest_rect);
    SDL_UpdateRect ( screen, 0, 0, 0, 0 );  //update whole screen
    vi->head++;
  } //while

  return 0;
}
```

Sample Makefile

The following is a typical **Makefile** for compiling and linking programs that use ffmpeg libraries:

```
#Sample Makefile for using ffmpeg libraries.
#"video.cpp" and "vplayer.cpp" are the application files

#users may need to modify the SDL path
LIBSDL =  -L/usr/local/lib -L/usr/local/lib -lSDL  -lpthread

PROG=vplayer
CC=g++
BASE=/apps/mpeg
FI = $(BASE)/include
FLIBS = -L$(BASE)/lib -lavcodec -lavdevice -lavformat -lavutil \
        -L/usr/lib
INCLS = -I/usr/include  -I$(FI)/libavcodec  -I$(FI)/libavdevice \
        -I$(FI)/libavformat -I$(FI)/libavutil -I$(FI)
#source codes
SRCS = $(PROG).cpp
#substitute .cpp by .o to obtain object filenames
OBJS = $(SRCS:.cpp=.o) video.o
FLAGS= -D_ISOC99_SOURCE -D_POSIX_C_SOURCE=200112  -g
FLAGS2=-rdynamic -export-dynamic -Wl,--warn-common -Wl,\
       --as-needed -Wl,
PATHS=-rpath-link,-Wl,-Bsymbolic
fflibs=-lavdevice -lavformat -lavcodec -lavutil
otherlibs=-lz -lbz2 -lm  -lasound -ldl

#$< evaluates to the target's dependencies,
#$@ evaluates to the target
$(PROG): $(OBJS)
    $(CC) -o $@ $(OBJS) $(FLAGS) $(FLIBS) $(FLAGS2)$(PATHS)\
              $(fflibs) $(otherlibs) $(LIBSDL)

$(OBJS):
    $(CC) -c  $*.cpp $(INCLS)

clean:
    rm $(OBJS)
```

A.4 Using FFmpeg Functions in Graphics

In the above sections, we discuss the usage of FFmpeg libraries. In the example, the decoded data are in RGB form and saved in a buffer. We can regard each frame as an image and make further processing on it. A typical application is to incorporate a video in a graphics application. This is commonly used in video game software. As an example, we discuss how we can use the techniques we have discussed to play videos in an OpenGL (Open Graphics Library) application.

First, we have to set the SDL video mode with the "SDL_OPENGL" option:

```
SDL_Surface *screen;
screen = SDL_SetVideoMode(640, 480, 24, SDL_SWSURFACE|SDL_OPENGL);
```

We may then initialize our graphics as usual. The following is a typical initialization of a graphics application that has a yellow light source at (20, 40, 20) and the camera (view point) is located at (0, 0, 10), viewing at the origin (0, 0, 0) with the up direction along the y-axis (0, 1, 0):

```
void init_gl ( SDL_Surface *screen ) {
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
  glViewport( 0, 0, screen->w, screen->h );
  glMatrixMode( GL_PROJECTION );
  glOrtho( -2.0, 2.0, -2.0, 2.0, -20.0, 20.0 );  //world window
  //camera at (0, 0, 10), looking at (0, 0, 0) with up-axis (0, 1, 0)
  gluLookAt ( 0, 0, 10, 0, 0, 0, 0, 1, 0 );

  glMatrixMode( GL_MODELVIEW );
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  glShadeModel(GL_SMOOTH);
  glClearColor( 1.0, 1.0, 1.0, 1.0 );           //white background
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  GLfloat light[] = { 1.0, 1.0, 0 };            //yellow light
  //light source at (20,40,20)
  GLfloat light_position[]={20.0,40.0,20.0,1.0};
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light );
  glLightfv(GL_LIGHT0, GL_SPECULAR, light );
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
}
```

With the OpenGL functions available, we no longer need to use the SDL functions like **SDL_UpdateRect**() to send the image data to the screen. Instead, we can use the OpenGL functions **glDrawPixels**() to write the image data to the graphics frame buffer at a location

specified by **glRasterPos**(). The data in the frame buffer are rendered to the screen. Using this method, we can play the video at more than one position of the screen simultaneously. The following is an example of simultaneous display of a video at two positions with a lit sphere revolving around:

```
//vi points to an object of the video class
//width = image width, height = image height
while ( !vi->quit ) {
  if ( vi->head == vi->tail ) {       //buffer empty
    SDL_Delay ( 30 );                 //sleep for 30 ms
    continue;
  }
  //consumes the data
  current_time = SDL_GetTicks();      //ms since library starts
  if ( current_time - prev_time < 50) //20 fps ~ 50 ms / frame
    SDL_Delay ( 50 - (current_time - prev_time) );
  prev_time = current_time;
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glRotatef ( vi->az,0, 0, 1 );
  glTranslatef ( 1.5, 0, 0 );
  glutSolidSphere ( 0.25, 22, 16 );   //creates a solid sphere
  glPopMatrix();
  vi->az += 1.0;
  glRasterPos2i ( -2, -2 );
  //may use GL_BGR depending on how data are organized in buf[]
  glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE,
                                        vi->buf[vi->head%4]);
  glRasterPos2i ( 0, 0 );
  glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE,
                                        vi->buf[vi->head%4]);

  glFlush();
  SDL_GL_SwapBuffers();
  vi->head++;
} //while
```

As another example of graphics application we may use each video frame as a texture image and paste it on any object we want. In other words, we can play the video on any object surface. The following code listing shows how we can play the video on the 6 faces of a rotating cube. In the code, the array *texImages*[] points to the video images saved in the buffer array *buf*[] that we have discussed before.

```
if ( !tex_initialized ) {
   glShadeModel ( GL_FLAT );
   glEnable(GL_DEPTH_TEST);

   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glGenTextures(6, texName);
   for ( int i = 0; i < 6; ++i ) {
     //now we work on texName
     glBindTexture(GL_TEXTURE_2D, texName[i]);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   }
   tex_initialized = true;
}

glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
float x0 = -1.0, y0 = -1, x1 = 1, y1 = 1, z0 = 1;
//defining the 6 faces of a cube
float face[6][4][3] =
  {{{x0, y0, z0},{x1, y0, z0}, {x1, y1, z0},{x0, y1, z0}},//front
  {{x0, y1, -z0},{x1, y1, -z0},{x1, y0, -z0},{x0,y0,-z0}},//back
  {{x1, y0, z0},{x1, y0, -z0},{x1,y1,-z0}, {x1, y1, z0}},//right
  {{x0, y0, z0},{x0, y1, z0},{x0, y1, -z0}, {x0, y0, -z0}},//left
  {{x0, y1, z0},{x1, y1, z0},{x1, y1, -z0}, {x0, y1, -z0}},//top
  {{x0, y0, z0},{x0, y0, -z0},{x1, y0, -z0},{x1, y0, z0}}//bottom
 };
 glEnable( GL_CULL_FACE );
 glCullFace ( GL_BACK );
 glPushMatrix();
 glRotatef( ax, 1.0, 0.0, 0.0);//rotate the cube along x-axis
 for ( int i = 0; i < 6; ++i ){//draw cube with texture images
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width,
              height, 0, GL_RGB, GL_UNSIGNED_BYTE, texImages[i]);
    glBindTexture(GL_TEXTURE_2D, texName[i]);
    glBegin(GL_QUADS);
      glTexCoord2f(0.0, 0.0); glVertex3fv ( face[i][0] );
      glTexCoord2f(1.0, 0.0); glVertex3fv ( face[i][1] );
      glTexCoord2f(1.0, 1.0); glVertex3fv ( face[i][2] );
      glTexCoord2f(0.0, 1.0); glVertex3fv ( face[i][3] );
    glEnd();
 }
glPopMatrix();
glFlush();
glDisable(GL_TEXTURE_2D);
```

Figure A-1 is an image captured from the output of the program that plays two instances of a video with a revolving sphere and Figure A-2 shows that the video is played on a cube with a lit sphere revolving around it.

The blending of image processing and computer graphics is a new field of study. Many objects can be synthesized by computer graphics techniques. Therefore, the scene of a video may be specified by a few parameters and a computer model can reconstruct the scene based on the parameters. It can result in very high compression and pleasant presentation of the video. The international standard MPEG-4 has introduced the concept of hybrid synthetic and natural video objects for visual communication. In this method, we

code the ‘natural’ or ‘real world’ objects using the traditional coding techniques we have discussed, but we employ some tools from the 2D/3D animation community to render synthetic or computer-generated visual scenes.

Figure A-1 Graphics Playing Two Video Instances

Figure A-2 Graphics Playing Video on Rotating Cube

Using computer graphics techniques, one can generate any 3D visual objects using polygon meshes. A polygon mesh models a 3D object as a collection of polygons, which may be simply triangles. One can rotate, translate, or scale the mesh to simulate the motion of any rigid body. Moreover, by deforming a couple polygons in a mesh, we can use the mesh to synthesize non-rigid objects like human bodies. For example, MPEG-4 specifies support for animated face and body models by defining the geometric shape of a body or face model and sending animation parameters to animate the body/face model. Developers use Facial Definition Parameters (FDPs) to describe a face model and Facial Animation Parameters

(FAPs) to describe its animation. One can use the default FDPs to render a generic face at the decoder, and obtain a custom set of FDPs sent by the encoder to create a specific face. The FAPs sent by the encoder are used to animate the face model. A typical application of this is to ‘compress’ the scene of a TV news announcement where the scene is fairly static with the main motion in the picture due to lips movement of the announcer; also in such an application, the exact motion of other parts of the body of the news announcer is not very crucial. In a similar way, MPEG-4 specifies the use of Body Definition Parameters (FDPs) and Body Animation Parameters (BAPs) to render a human body. Using these parameters, one can render generic body models as well as a customized synthetic body to represent a ‘real world’ human. In general, an application for face and body animation consists of the generation of virtual scenes containing face and/or body meshes, as well as model-based coding of natural (real world) face or body scenes, where face and/or body movement is analyzed, coded and transmitted by the encoder as a set of BAPs and FAPs; upon receiving the BAPs and FAPs, the decoder utilize them to synthesize the face and body.

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273