Incorporating Video in Platform-Independent Video Games Using Open-Source Software

Tong Lai Yu, David Turner, David Stover, Arturo Concepcion
Department of Computer Science and Engineering
California State University
San Bernardino, CA 92407, USA

Abstract – In a video game, it is common to embed a prerecorded video and play it at various times during the game for the purposes of narrative, tutorials, hints, or any other goal that the developer sees fit in the game. It is a challenging task to play the embedded video in a cross-platform video game at multiple locations of the screen simultaneously and on an arbitrary surface. This paper describes the use of open-source resources to accomplish such a task.

The open-source FFmpeg library is employed to decode video saved in commonly-used video formats such as MPEG-4, AVI or MOV. The platform-independent Simple DirectMedia Layer (SDL) library is used to render video and audio and to handle threading features. The producer-consumer paradigm is employed to separate the tasks of data decoding from data rendering which are run in different threads.

I. INTRODUCTION

The improvement in computing hardware technology in recent years has sharply reduced the cost of computing. Consequently, the computing capacity of PCs has become ever more powerful. Many sophisticated video games that were only able to be run on dedicated game consoles with dedicated hardware can now be run on PCs. Moreover, by making use of open-source resources and open standards, one can significantly shorten the development time of a large-scale video game and make the code more robust.

Mythic is a video game project constructed by students and faculty under an NSF CPATH grant [1] to investigate the use of community-based video game development projects as a means to stimulate interest in computing studies and improve the development of computational thinking. It is a collaboration between the Department of Computer Science and Engineering (CSE) of California State University at San Bernardino (CSUSB), Riverside Community College, Rim of the World High School, and Broken Circle Studios. Geng is an open-source game engine developed at CSUSB to be used in Mythic and other video game projects. Geng utilizes mostly open-source tools in the development and is itself open-source. Geng programs can be compiled and run on the three popular platforms, Linux, Mac OS/X, and MS Windows. Figure 1 presents a block diagram of the architecture of Geng, showing various open-source resources that it has used; it involves a wide variety of features and has used a layered modular approach.

Basically, a contemporary video game consists of a synthesized world using computer graphics. However, there are situations that a video game has to embed a real-world video in it for the purposes of narrative, tutorials, and game play hints, or if necessary, the synthesized world could blend real-world scenes in it to create special entertaining effects. In fact, the blending of image processing of real world scenes and the creation of synthesized graphical objects has become a new field of study [9, 7, 12, 15] . In this paper, we concentrate on discussing the incorporation of videos in Geng, which is a challenging task as Geng is a full-fledged video game engine that requires rendering videos at several positions of the screen simultaneously and playing the video on an arbitrary 3D surface such as spheres, cubes, and ellipsoids; also, the problem of synchronizing video and audio has to be addressed. We solve the problems encountered using open-source software.

II. MULTITHREADED AND MULTIMEDIA GAME ENGINE

Geng is developed using C/C++. As shown in Figure 1, it employs a layered and modular approach to divide tasks into various modules. Its multithreaded and multimedia features are managed by the Simple DirectMedia Layer (SDL) [19] functions. The SDL library is written in C and can be easily integrated with OpenGL; it is a cross-platform open-source multimedia library designed to provide low level

[1] National Science Foundation grant CNS-0938964

978-1-4244-5540-9/10/$26.00 ©2010 IEEE

access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer; it has been used by MPEG playback software, emulators, and many popular games. We use the SDL libraries and Open Audio Library (OpenAL)[4] to incorporate sound and music in the game. The C/C++ standard template library (STL) containers such as vectors and queues are used to record the states of the game [17].

Geng uses threads extensively to carry out many tasks simultaneously. Using threads properly is important in embedding video and audio in a game. IEEE defines a POSIX standard API, referred to as Pthreads (IEEE 1003.1c), for thread creation and synchronization[6, 8]. Many contemporary systems, including Linux, Solaris, and Mac OS X implement Pthreads. Pthreads specification has a rich set of functions, allowing users to develop sophisticated multithreaded programs. However, in the Geng project, we find that many of the Pthread features are not needed in the project. So instead we use SDL threads in our implementations. SDL threads are much simpler and only consist of a few functions. More importantly, SDL is platform independent and can run in the three popular platforms, Linux, Mac OS/X and MS Windows without any modifications of the source code. Though simplified, SDL threads provide semaphores and locking mechanisms that allow us to do proper synchronization of various events.

III. FFMPEG LIBRARIES

Though MPEG-4 and H.264 [3, 10, 11] are the widely used and endorsed video standards, practically, we have to deal with a large variety of video formats as the videos might have been recorded in a non-standard format or the related party may not have the tool to record the video in the standard format. The problem of handling non-standard formats is solved by utilizing the open-source package FFmpeg [18] to make conversions.

There exists open-source video codecs that are available and ready to use. The most commonly used open-source video libraries may be those from FFmpeg which 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 first use FFmpeg utilities to convert any non-standard video to MPEG-4 format. In this way, we only have to study one single video format (MPEG) in detail and Geng only has to process videos in MPEG-4. Actually, we also use the FFmpeg functions to decode an MPEG-4 file; the decoded video data are sent to the screen using OpenGL or SDL functions and the decoded audio data are processed by SDL functions.

IV. VIDEO AND AUDIO SYNCHRONIZATION

MPEG uses time-division multiplexing (TDM) to combine video and audio data together into one single stream, which can be either transmitted on a single communication channel or stored in a digital storage medium (DSM). As shown in Figure 2, TDM allocates periodic time slots to each of the streams of audio and video. In general, audio and video data are prepared and edited separately prior to multiplexing, and therefore they need to be synchronized[3]. MPEG achieves synchronization through the use of clock references and metadata fields referred to as time stamps. Time stamps reference a system time clock (STC) to determine when a particular presentation unit (a video, audio or other data stream) should be decoded and presented to the output device. Clock references are 42-bit data fields that inform the demultiplexer what the STC time should be when each clock reference is received. Decoding of N elementary streams is synchronized by adjusting the decoding of streams to the system time clock, which may be one of the N decoders' clocks, the data sources clock, or some external clock.

MPEG uses bidirectionally predicted frames (B-frames) to encode data [10, 11]. Because B-frames can be predicted or interpolated from earlier and/or later frames, MPEG defines two types of time stamp, decoding time stamp (DTS) and presentation time stamp (PTS) to indicate when the presentation unit is to be decoded and when it is to be passed to the output device for display, respectively.

To achieve synchronization, the decoder uses a buffer to temporarily store the compressed data stream until the decoding time of the data is at the right instance. When the decoder STC advances to the time specified by the DTS, the corresponding presentation unit (PU) is removed from the buffer and passed to the decoder. If the PTS is later than the DTS, the decoder witholds the decoded PU until the STC advances to the PTS time. Otherwise the decoded PU is presented to the display immediately.

The FFmpeg library provides functions to read the PTS of an MPEG file in the correct order and the SDL library provides functions for setting timers. After we have shown a frame, we use the function SDL_timer() to set the timeout period for showing the next frame. We check the value of the PTS of the next frame against the system clock to determine what the timeout period should be. On the other hand, the time to send the audio samples to the audio device is calculated from the sample rate provided by the data file. We synchronize the video to audio by comparing the video timeout period to the audio output time and make any necessary adjustment for the next refresh.

V. The PRODUCER-CONSUMER PARADIGM

To create a robust playback system, we use the producer-consumer paradigm [13] to separate the display functionalities from the decoding process [17]. We use one thread to decode the compressed data (**decoder**) and another thread to process the decoded data (**player**) with a shared buffer. In

this way, the decoding process is cleanly separated from further data processing. The **decoder** can use whatever method or techniques it wants to decode the compressed data and put the decoded data in the buffer. On the other hand, the **player** can do whatever it wants with the decoded data; it can render them on the screen directly, or as discussed in the next section, it can treat the data as texture and play them on an arbitrary surface. This is a classical producer-consumer problem [13]; the **decoder** 'produces' the data while the **player** 'consumes' the data. In a more general scope, a producer-consumer problem consists of a set of *producer* threads supplying resources to a set of *consumer* threads. These threads share a common buffer pool where resources are deposited by producers and removed by consumers. All the threads are asynchronous in the sense that producers may attempt to deposit and consumers may attempt to remove resources at any instant. Since producers may outpace consumers and vice versa, two constraints must be satisfied. Consumers are forbidden to remove any resource when the buffer pool is empty and producers are forbidden to deposit any resources when the buffer pool is full. A conventional way to solve a producer-consumer problem is to use semaphores to block and wakeup the threads at appropriate moments; we use the SDL semaphore functions to achieve this.

In our implementation, we use a ring buffer that can hold two to four image frames as the buffer pool. The following C-like psuedo code illustrates the mechanism. Note that the call to **wait**() is a nonbusy wait, meaning that the thread goes to sleep and is woken up only by another thread signaling on the the corresponding semaphore.

```
unsigned long head = 0, tail = 0;  //shared variables
char buf[4][frameSize];            //buffer pool
semaphore sem_tail;            //a counting semaphore
semaphore sem_head;            //a counting semaphore
//decoder() (producer), insert data at tail of buffer
while ( there_is_data ) {
  while ( tail >= head + 4 )   //buffer full
      wait ( sem_head );       //wait on sem_head

  //produce data
  decode_data( buf[tail%4] );  //deposit decoded data
                               // in buffer
  tail++;                      //advance the tail
  signal ( sem_tail );//wake up consumer if necessary
}

//player() (consumer): delete data at head of buffer
while ( there_is_data ) {
  while (head==tail && there_is_data)//buffer empty
      wait (sem_tail);  //wait on semaphore sem_tail,
                        //  do not consume
  //consume the data
  consume_data ( buf[head%4] );
  head++;               //advance the head pointer
  signal ( sem_head );  //wake up producer if necessary
  SDL_Delay(sync_time); //synchronize with other events
}
```

VI. EMBED VIDEO IN GRAPHICS

In the Geng project we want to play a video simultaneously on several locations on the screen and on arbitrary surfaces. A convenient way to achieve this is to utilize OpenGL functions along with the SDL library. The following function call is an example of initializing the game system so that it can run OpenGL on top of SDL:

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

In the previous section, we discussed that the data of a frame are saved in a buffer and are ready for use. OpenGL provides the function **glDrawPixels**() [5, 14] that allows us to write a rectangular array of pixels from CPU memory buffer into the graphics framebuffer at a raster position specified by the function **glRasterPos**(). Therefore, we can send the data to the screen at various positions for as many times as we want by using **glDrawPixels**() and **glRasterPos**(). To avoid any flickering effect, double-buffering is employed. That is, we write the data to the graphics buffer at the background and after we have finished all the writing, we blit them to the screen using the function **SDL_GL_SwapBuffers**().

Since the decompressed video data are available in a memory buffer, we can render them to any surface using a mapping technique. Mapping algorithms can be thought of as either modifying a shading algorithm using a rectangular array (the map) or as modifying the shading by using the map to alter surface parameters, such as material properties and surface normals[1, 5]. There are three major mapping techniques:

1. Texture Mapping
2. Bump Mapping
3. Environment Mapping

All of them are still graphics research topics. Here we only consider texture mapping. One can naturally extend the technique to the other two mappings, depending on the application. Texture mapping, in its simplest from, consists of applying a graphics image or a pattern to a surface. A texture map can hold any kind of information that affects the appearance of a surface: the texture map serves as a precomputed table, and the texture mapping then consists simply of table lookup to retrieve the information affecting a particular point on the surface as it is rendered[2].

OpenGL supports a variety of texture mapping options. The following is an example that demonstrates the initialization of texture mapping and how to map the video data onto the six surfaces of a rotating cube:

```
-------------------------------------------------------
unsigned texName[6];

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glGenTextures(6, texName);
for ( int i = 0; i < 6; ++i ) {
 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);
}
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
float face[6][4][3]  = ...//defines the 6 faces of cube
glEnable( GL_CULL_FACE );
glCullFace ( GL_BACK );
.....   //Rotation here
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();
.......
-------------------------------------------------------
```

See Figure 3 for an example of the result.

VII. CONCLUSIONS

We have presented an effective method to embed a prerecorded video in a video game by making use of open-source resources. We use SDL and OpenGL to make the codes to run in cross-platform environments. Open-source FFmpeg is used to convert videos in other formats to the MPEG standard format and to decode the compressed MPEG file. Synchronization of audio and video is done with use of presentation time stamps (PTS), SDL semphores and timers. The producer-consumer paradigm is used to cleanly separate data decoding from further processing. Consequently, OpenGL functions can be utilized to present multiple instances of a video at various positions on the screen, and texture mapping can be used to play the video on arbitrary surfaces; we expect that the video mapping technique can be generalized to other common mappings such as bump mapping and environment mapping. More research can be done on the mapping techniques in the future for integration of video with 3D graphical applications.

Acknowledgments

This work is supported in part by National Science Foundation CPATH grant CNS-0938964.

References

[1] E. Angel, *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, Fourth Edition, Addison-Wesley, 2005.

[2] S. R. Buss, *3-D Computer Graphics: A Mathematical Introduction with OpenGL*, Cambridge University Press, 2003.

[3] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An introduction to MPEG-2*, Springer, 1996.

[4] John R. Hall, *Programming Linux Games*, Linux Journal Press, 2001.

[5] F.S. Hill, and S.M. Kelly, *Computer Graphics Using OpenGL*, 3rd Edition, Prentice Hall, 2007.

[6] B. Lewis, and D. Berg, *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, 1995.

[7] C. Marrin, R. Myers, J. Kent, and P. Broadwell. "Steerable media: interactive television via video synthesis," *Proceedings of the sixth international conference on 3D Web Technology: Virtual Reality Modeling Language Symposium*, ACM, p. 714, Paderbon, Germany, 2001.

[8] T.Q. Pham, and P.K. Garg, *Multithreaded Programming with Windows NT*, Prentice Hall, 1996.

[9] Editied by Igor S. Pandzic, and Robert Forchheimer, *MPEG-4 Facial Animation: The Standard, Implementation and Applications*, John Wiley & Sons, 2002.

[10] Iain E.G. Richardson, *H.264 and MPEG-4 Video Compression*, John Wiley & Sons Ltd., 2003

[11] Iain E.G. Richardson, *H,264/MPEG-4 Part 10 White Paper*, http://www.vcodex.com

[12] Edited by Nikos Sarris and Michael G. Strintzis, *3D Modeling & Animation: Synthesis and Analysis Techniques for the Human Body*, IRM Press, 2005.

[13] M. Singhal and N.G. Shivaratri, *Advanced Concepts in Operating Systems*, McGraw-Hill, 1994.

[14] D. Shriener et al., *OpenGL Programming Guide*, Sixth Edition, Addison-Wesley, 2008.

[15] T.L. Yu, "A Framework for Very High Performance Compression of Table Tennis Video Clips", *Proceedings of IASTED on Signal and Image Processing*, P.167-172, Kailua-Kona, Hawaii, August 18-20, 2008.

[16] T.L. Yu, "Implementation of Video Player for Embedded Systems", *Proceedings of IASTED on Signal and Image Processing*, p. 25-30, Honolulu, Hawaii, August 20-22, 2007.

[17] T.L. Yu, "Chess Gaming and Graphics using Open-Source Tools", *Proceedings of ICC2009*, p. 253-256, Fullerton, California, IEEE Computer Society Press, April 2-4, 2009.

[18] http://ffmpeg.org

[19] http://www.libsdl.org

[20] http://www.mesa3d.org

[21] http://www.openal.org

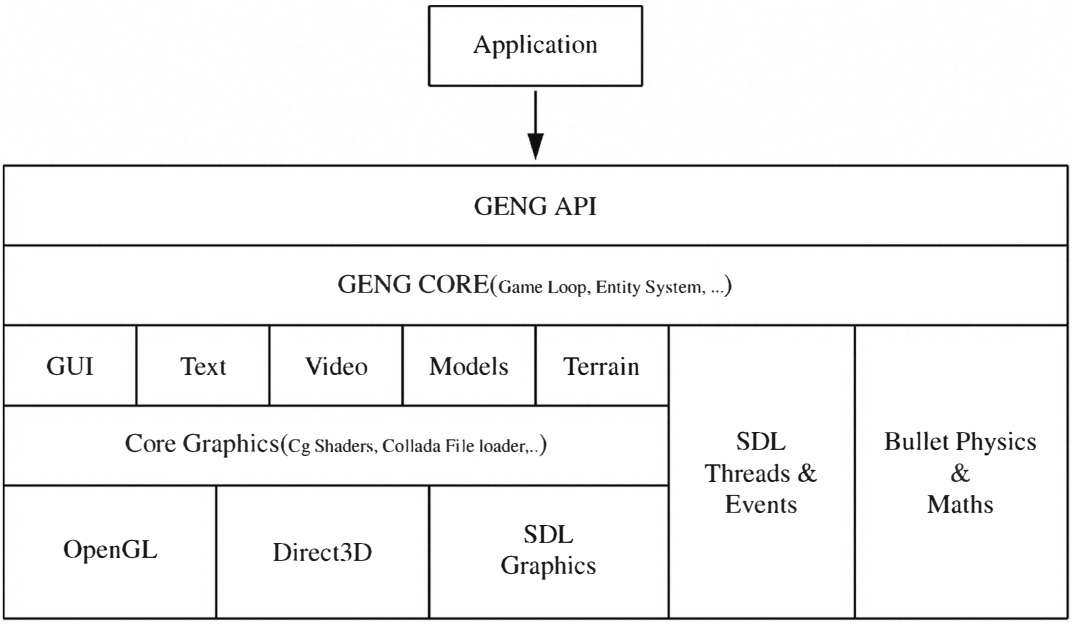

Figure 1. Geng Architecture

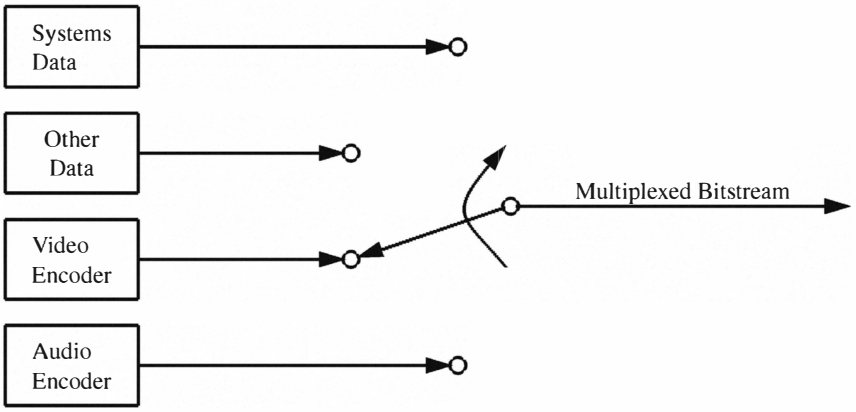

Figure 2. Time-Division Multiplexing of Data

Figure 3. Blending Graphics with Video Played on Surfaces of Rotating Cube