Implementation of an Intelligent Force Feedback Multimedia Game

Fei He[§]

Ernst & Young LLP, Kansas City, Missouri U.S.A.

Arvin Agah[†]

Department of Electrical Engineering and Computer Science
The University of Kansas, Lawrence, Kansas 66045 U.S.A.

ABSTRACT

This paper presents the design and programming of an intelligent multimedia computer game, enhanced with force feedback. The augmentation of game images and sounds with appropriate force feedback improves the quality of the game, making it more interesting and more interactive. We used the Immersion Corporation's force feedback joystick, the I-FORCE Studio computation engine, and the Microsoft DirectX Software Development Kit (SDK) to design the multimedia game, running in the Windows NT operating system. In this game, the world contains circles of different sizes and masses. When the circles hit each other, collisions take place, which are shown to, and felt by, the user. When the circles hit together, the overall score increases; the larger the size of the circle, the higher the score increase. The initial score is set to zero, and when the game ends, a lower score represents a better performance. This game was used to examine the behavior of the users under different environments through their respective scores and comments. The analysis of experimental results helps in the comparative study of different kinds of multimedia combinations.

[§]Work performed while Fei He was at the University of Kansas

[†]Corresponding author
e-mail: agah@ukans.edu
tel: +1 (785) 864-7752; fax: +1 (785) 864-0387

KEYWORDS

intelligent multimedia games, video game programming, force feedback

1. INTRODUCTION

When most individuals think of multimedia, they consider multimedia to be the combination of text, graphical animation, stereo sounds, and live video images. In current multimedia applications, the focus is on providing a good visual and audio feedback to the user, trying to improve tracking the user, and providing better resolution media to the user (Buttolo et al., 1996). In multimedia applications, the goal is to make the users believe that they are actually immersed in a computer-generated environment. To achieve this goal, text, visual feedback, and auditory feedback are often added. These elements are not enough, however. Being able to visualize and to hear these environments is only part of the equation. Individuals must be able to interact with the virtual worlds. Yet, the lack of proprioception, i.e., the impossibility of really 'touching' the entities in the world (game), makes the interaction unreal and more difficult. This shortcoming can hamper interactions for which a high level of dexterity is required. Force feedback enables the operator to manipulate the environment in a natural and effective way, enhances the sensation of 'presence', and permits a faster rate of information by including haptic information.

Thus, it can be argued that a combination of images, sounds, and forces are all required. Beyond these reasons, the objective of this research is to see how the forces, sounds, and images affect the user's perception and action during the human-computer interaction. We used a force feedback joystick and the Microsoft DirectX Software Development Kit (SDK) to design a multimedia game. The game is enhanced by images, sounds, and force feedback. In this game, the world contains circles of different sizes and masses. When the circles hit each other, collision occurs. When the circles hit together, the overall score increases; the larger the size of the circle, the higher the score increase. The initial score is set to zero, and when the experiment ends, a

lower score represents a better performance. The time length of all experiments is identical. The game was used to examine, under different environments, the behavior of the users through their respective scores and comments. Different types of multimedia interfaces were tested during the experiments, varying combinations and parameters of sound and force feedback with and without time delays (Fei, 1999; Fei & Agah, 2000). The analysis of the experimental results helps with the comparative study of different types of multimedia combinations.

This paper is organized into five sections. The related work on force feedback and multimedia games is included in Sec. 2. Section 3 provides information on Microsoft DirectX, which is the tool used for programming the game. Section 4 discusses how the game is implemented using DirectX. This section will include the game environment and main algorithms. Discussions of the limitations and the future work are presented in Sec. 5.

2. RELATED WORK

This section covers the background and related work in the areas of force feedback systems and multimedia computer games.

2.1 Force Feedback

In our research, we used the Immersion Corporation's force feedback joystick as the haptic device. A picture of the joystick is shown in Fig. 1 (Rosenberg, 1997). A typical joystick is input-only; it tracks a user's physical manipulations but provides no physical feedback representing the results of those manipulations. For example, in a video game or a virtual reality simulation, a user might be using a joystick to command the motion of a simulated racecar or spacecraft through simple physical simulations. The results of the user's manipulations are displayed visually on the screen. When the user's actions cause the spacecraft to collide with an asteroid, or cause the racecar to slam into a wall, a standard joystick has no means of conveying such information back to the user.

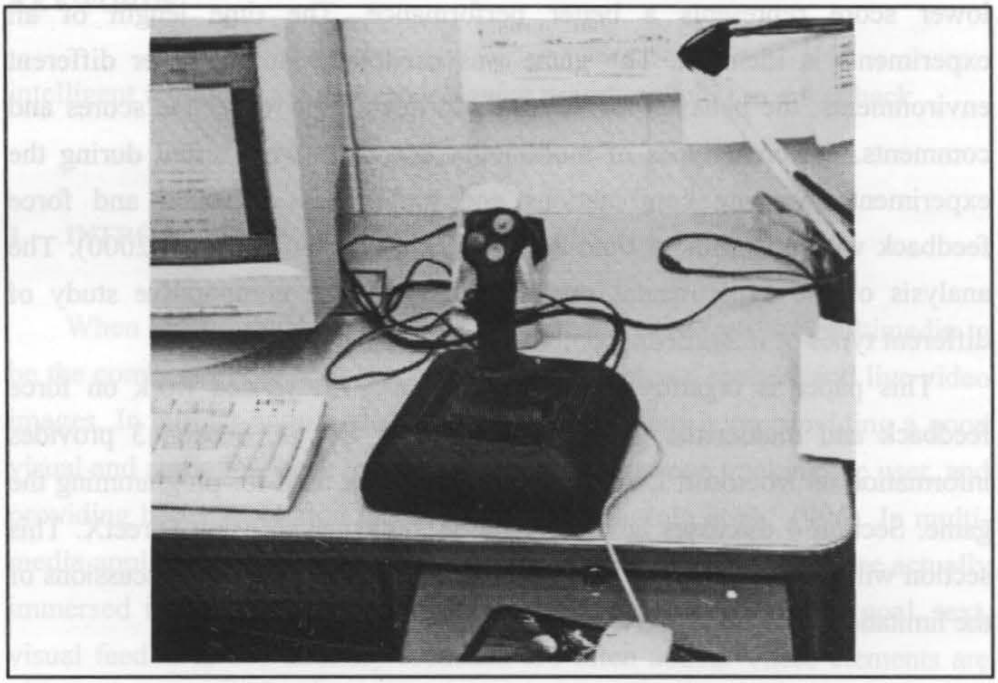

Fig. 1: An example of a force feedback joystick

The force feedback joystick is an *input-output* device. In other words, the force feedback joystick can track the user's physical feedback sensations representing the results of those manipulations. For example, when using a force feedback joystick to control the flight of a spaceship, an impact is not only shown visually on the screen but also actually displayed physically as real forces imparted on the user's hand. When equipped with force feedback, the haptic device can impart the simulated 'feel' of impacts, rigid surfaces, liquids, springs, vibrations, textures, varying masses, winds, machine engines, and other physical phenomena that have a mathematical representation.

The basic idea behind force feedback technology is quit simple—as the user manipulates the interface hardware, the actuators (motors) apply computer-modulated forces that either resist or assist the manipulations. Such forces are generated based on mathematical models that are appropriate for the desired sensations. For instance, when simulating the feel of a rigid wall with a force feedback joystick, motors within the joystick apply forces to the handle to replicate the feel of encountering a wall. In this case, the mathematical model

driving the force is as follows: as the user moves the joystick to penetrate the wall, the motors apply a force that resists the penetration. The harder the user pushes the harder the motors push back. The end result is a sensation that feels quite compelling because it truly represents a physical encounter with an obstacle—a simulated obstacle. Other sensations follow more complex mathematical models, but the paradigm is basically the same. To simplify the process of generating feel sensation, the manufacturers of force feedback hardware have taken care of all the mathematics required in the generation of different types of feel sensations. Most of the sophisticated computations are handled by dedicated hardware onboard the peripheral device. Immersion Corporation's I-FORCE Studio is a computation engine that has been licensed to many major manufacturers of gaming peripheral devices for use in their force feedback products. Such a computation engine enables a wide variety of complex feel sensations to be produced efficiently in hardware with minimal programming overhead (Rosenberg, 1997). With manufacturers handling the mathematical complexities of force feedback in dedicated hardware, we can be provided with an easy-to-use, high-level API (such as Microsoft DirectX) that abstracts the problem of feel programming to a perceptual rather than a mathematical level. API calls allow us to define and initiate feel sensations easily, using intuitive function calls with descriptive physical names such as 'Wall', 'Vibration', or 'Liquid'. These functions are highly parameterized so that we can customize the feel of each basic sensation type with great flexibility. We are free from the burden of actually controlling force as a mathematical function of time or motion (Force-feedback, 1998).

As the force feedback joystick is an input-output device, it needs to communicate with the host computer bi-directionally. Tracking information is sent from the peripheral to the host for use in controlling play. As illustrated in Fig. 2, force feedback information is sent from the host to peripheral to coordinate the feel of the corresponding events. Because force feedback devices require bi-directional communication to coordinate feel sensation with play, the communication speed has a substantial effect upon force feedback performance. The faster the communication links, the better the coordination between visual and physical events. Ideally, force feedback devices use efficient processing techniques to minimize the amount of information that

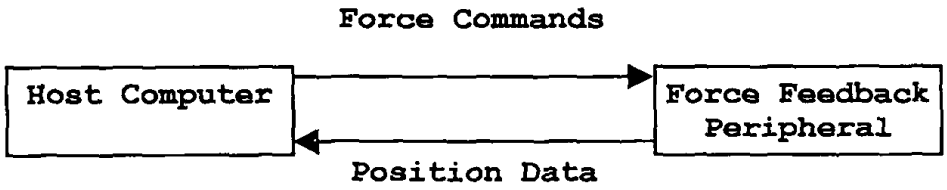

Fig. 2: Force feedback information flow

must be communicated to the host and the peripheral. For example, efficient force feedback devices use sophisticated local microprocessors to produce force feedback sensations locally in response to concise high level commands sent from the host. In our research, we use a I-FORCE processor as the computation engine (Rosenberg, 1997).

2.2 Multimedia Computer Games

In our research, we use the Microsoft DirectX Software Development Kit (SDK) to design a multimedia game (Lamothe, 1997; Clark, 1998; DirectX, 1999; Barnett, 1999; Geocities, 1998; Wksoftware, 1999; Kavach, 1997; Browsebooks, 1999). Video games have been around as long as computers have. Most computer historians would agree that the first computer games, usually text-based adventure or military-simulation games, were made for UNIX mainframes in the 1960s and 1970s. The revolution in games for personal computers came in the late 1970s, with the advent of the Atari 800 and Apple II personal computers, which were the first computers to have good color, decent sound, and reasonable graphics power. Game programmers wrote Atari and Apple games in BASIC or in pure machine language.

In 1984, IBM PC started selling very well in the personal computer marketplace. Also, the 80286 processor arrived on the market, the EGR card was available, and a VGA card was on its way. The IBM PC had some hardware that could be used by programmers to produce good games. Because IBM PC is so popular and in such common use, it is considered to be a game machine, and many programmers started writing games for the PC. In the latter part of the 1980s, the 80386 processor, the VGA card, and the Sound Blaster sound card became available to PC gamers. At this point, the

IBM PC was starting to push heavily into the game market. By 1990, few programmers wrote a game for any platform other than the PC. Nevertheless, many programmers were not happy with the performance of the PC. The early IBM PC was a difficult platform for writing games. In 1993, id Software, Inc. released Doom. It was very fast, looked good, had great graphics and a dependable game engine, had good sound effects, and was fun. Since then, the game programming society has realized that the PC can do almost anything gamewise. Both hardware and software manufacturers have built on this supposition and today, the top-of -the-line gaming PC is a 266 MHz, 3D-accelerated, wave-table synthesized, 32-bit game platform that is nothing short of a dream machine (Lamothe, 1997). Currently, gaming is a multi-billion-dollar industry that will continue to grow and move into new markets.

3. MICROSOFT DIRECTX

This section will include an introduction to Microsoft DirectX, which we used to design our multimedia computer game.

3.1 Description of DirectX

DirectX is a low-level programming API (Application Program Interface), developed by Microsoft, that is designed to give the programmer a close-to-hardware-programming environment for multimedia and game programming (Barnett, 1999). It allows multimedia and games designed for Windows 95 and Windows NT to have access to the computer hardware through a standard interface. Without a standard interface, stable compatibility between games and the computer's hardware would be virtually impossible. With DirectX, game developers can write for this interface and let the Windows DirectX compatible drivers deal with how to 'talk' to the computer hardware. An overview of DirectX is shown in Fig. 3.

As illustrated, any function that the game requires, the computer hardware to perform it is passed via the DirectX API to the computer hardware drivers. The drivers are responsible for making sure that the game's requests are properly implemented in the hardware. DirectX is actually a suite of APIs.

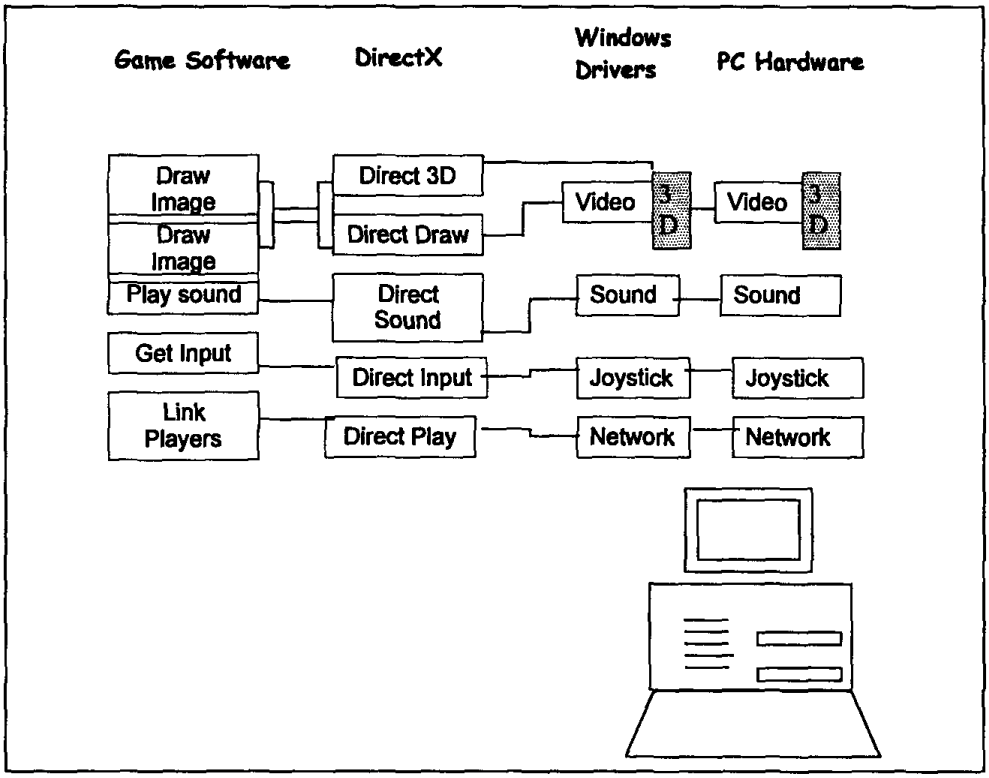

Fig. 3: An overview of DirectX

The primary individual components are DirectDraw, Direct3D, DirectSound, DirectSound3D, DirectMusic, DirectInput, DirectPlay, DirectSetup, and Auto Play. Figure 4 illustrates all the DirectX components and their relation to Win32, GDI (Graphics Device Interface), and the hardware. GDI and DirectX are on different sides of the structure, and each has access to the other and to the hardware. The blocks of DirectX are called the HAL (Hardware Abstraction Layer) and HEL (Hardware Emulation Layer).

3.2 Components of DirectX

3.2.1. *The Hardware Abstraction Layer (HAL).* The Hardware Abstraction Layer (HAL) is the lowest level of software in DirectX, consisting of the hardware drivers that are provided by the manufacturers to control the hardware directly. This layer of software provides the utmost performance because it talks directly to the hardware.

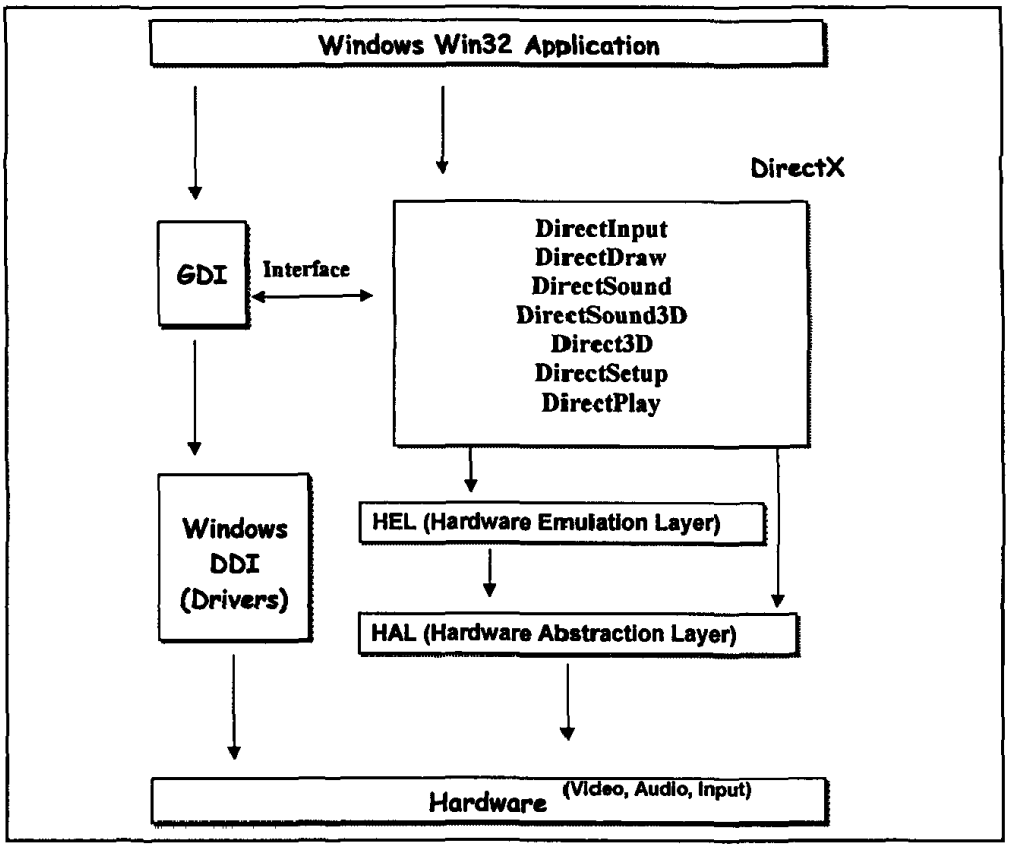

Fig. 4: The components of DirectX

3.2.2. *The Hardware Emulation Layer (HEL).* The Hardware Emulation Layer (HEL) is built on top of HAL. In general, DirectX is designed to take advantage of the hardware if the hardware is there, but DirectX still works if hardware is not available. For example, suppose that we write some graphics code, assuming that the hardware that we are running on supports bitmap rotation and scaling. We can make calls to DirectX to scale and rotate bitmaps. On hardware that supports scaling and rotation, our code runs at full speed and uses the hardware, but if we run on hardware that doesn't support scaling and rotation, we have to use HEL. HEL emulates the functionality of HAL with software algorithms so that we don't know the difference. Nevertheless, the code will run more slowly.

3.2.3. *Other Components.* Other components of DirectX include the following:

- DirectDraw: Enables access to the video card, along with hardware acceleration capabilities.
- Direct3D: Enables the programmer to use a standard API to communicate with any 3D hardware.
- DirectSound: Responsible for playing all of the digital sound effects.
- DirectSound3D: An implementation of 3D sound.
- DirectMusic: Plays MIDI files and supports dynamic music.
- DirectInput: Allows a program to acquire data from the keyboard, mouse, and joystick in a uniform manner.
- DirectPlay: Provides networking support for multi-player games.
- DirectSetup: An API for installing and setting up DirectX and the game.
- AutoPlay: The Windows support for automatic loading of CDs.

3.3 Using DirectX

In our multimedia game, we use DirectDraw, DirectSound, and Direct Input. We will give some general introduction to see how these three components work.

3.3.1. *DirectDraw.* The first step is the creation of a DirectDraw object with the DirectDrawCreate function. The DirectDraw object creates and owns all other objects in the DirectDraw API and is also the means by which we access all global functions, such as setting the display resolution or color depth. Once we have a pointer to a DirectDraw object, we have to set the cooperative level and display mode. As we will use full-screen mode, we will set cooperative level to Exclusive, Full-Screen. The next step is to create a primary DirectDrawSurface object, which represents either a display surface or an off-screen buffer and resides in either system or video memory. Their member functions allow the copying of data from one surface to another, page flipping, and much more. The last step is to create a DirectDrawPalette object, using the CreatePalette function. DirectDrawPalettes represent 16 or 256 index palettes that can be attached to a surface. When all these setting are done, we can put DirectDraw to use.

3.3.2. *DirectSound.* The first step is similar to setting up the DirectDraw, in creating the DirectSound object. In our system, we need only one Direct Sound object for the default sound card on the system. The next step is to use the IDirectSound interface. The IDirectSound interface has the ability to create and duplicate DirectSoundBuffer objects and to deal with sound card capabilities and global settings. Before we play the sound, we set the cooperative level to DSSCL_NORMAL. The last step is to copy the sampled sound data into the buffer. In our system, we will load the data from wave (*.WAV) files.

3.3.3. *DirectInput.* The starting point is to create a DirectInput object and retrieve a pointer to its IDirectInput interface. We will use the DirectInput Create function to create the high-level object. When we have the object and an interface to the object, we will use CoCreateInstance to make the first call to initialize the object. Once we have our DirectInput object, we can use it to create a DirectInputDevice object. In Windows, we usually have a system keyboard and a system mouse, but in our game, we use a joystick. Therefore, we use the EnumDevices function to enumerate the joystick. Now that we have used the CreateDevice member function to get an interface to a Direct InputDevice object, we will begin to deal with the actual physical device—the joystick. We will set the data format for the device object and cooperative level of the device. In our game, we set cooperative level to DISCL_ EXCLUSIVE and DISCL_FOREGROUND.

Before we begin retrieving input from the joystick, we have to set the property for the device. This includes such details as the range of values that we want and what portion of the joystick is the dead-zone. Next, we have to begin making force feedback effects for the joystick. Although this process begins with the DirectInputDevice object, much of the work is done with an object called DirectInputEffect. For the most part, DirectInputDevice retrieves input from a device and creates instances of the DirectInputEffect object. First, we create an instance of the DirectInputEffect object for each effect and fill out the DIEFFECT structure. In our game, we use a *constant force* and a *ramp force.* Constant force is a force in a single direction that does not change in strength. A ramp force is a constant force that changes in strength linearly over time. After filling in the DIEFFECT structure and calling

CreateEffect, we receive a pointer to an IDirectInputEffect interface. Now we can use this interface to play the effect, to change the effect, and so on.

4. THE MULTIMEDIA GAME

4.1 Game Overview

In our game, we create circles of different sizes and masses. One of the circles (MyCircle) is to be controlled by the user (player) through the force feedback joystick, while other circles (FreeCircles) move randomly on the screen. Each circle has its position indicated by the x-coordinate and the y-coordinate. When the circle controlled by the joystick hits another circle, collision takes place and each of them bounce accordingly. The final velocities of the two colliding circles are determined by the theories of *conservation of momentum* and *conservation of energy*. Also, as a collision occurs, the sensation of force feedback can be created and thereafter felt by the player. The edges of the game environment are designed as walls where circles can not penetrate. A circle will bounce back when it hits a wall, and the player will feel the force from the joystick. Our game is enhanced with images, sounds, and force feedback. Each time when two circles collide, we can hear a sound similar to a 'bounce'. We also incorporated a time delay in our game, to be used for experimentation with the effects of time delay in force feedback. The rule of the game is that the player uses the joysticks to control a circle and tries to avoid collisions with the walls and the five circles. Each game is limited to the duration of two minutes. Figure 5 includes a snapshot of the game and Fig. 6 shows a player playing the multimedia game.

4.2 Software Architecture

The video game program starts by creating and setting up a window. This requires the DirectDraw interface to the graphic device and DirectSound interface to the sound buffer. Then the force feedback device is initialized and a doubly linked-list leading by MyCircle is created. The system is then thrown into an infinite 'while' loop. The body of the while loop first peeks the

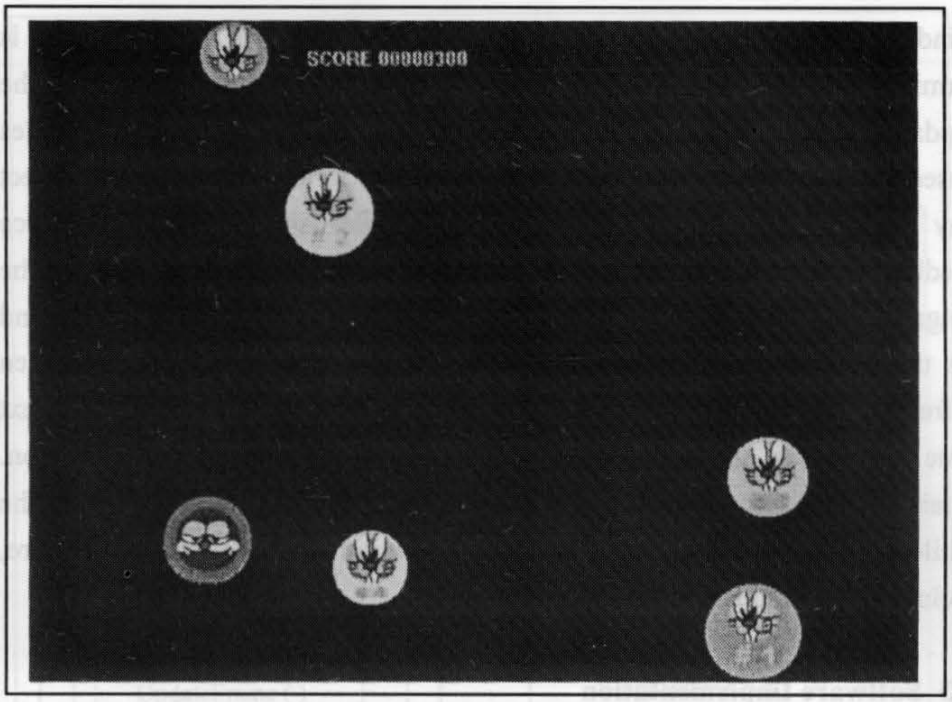

Fig. 5: A snapshot of the game.

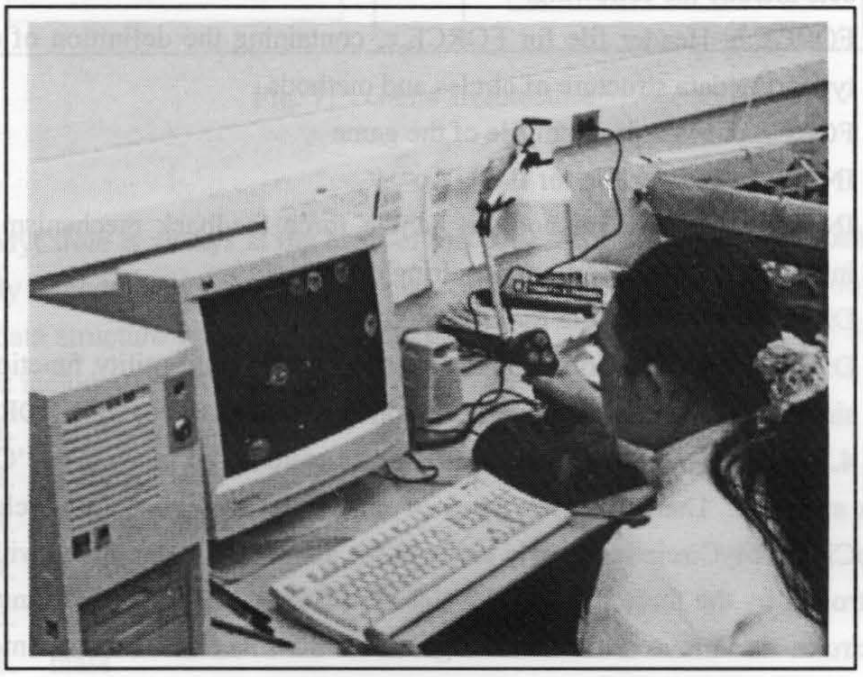

Fig. 6: A player playing the multimedia game

window message queue to detect any ESCAPE message. The program is terminated if the Escape key is pressed. If not, then the system calls the UpdateFrame() method. This function updates the positions of all circles based on their velocities. Then, CheckForHits() function is called to detect any collision that has taken place and to handle each collision. The force feedback effect is played, and the sound of a 'scream' will be heard. The magnitude and direction of the force effect that the player experiences depend on the masses and velocities of the colliding circles. The screen is then refreshed, and the graphic subsystem draws circles onto the screen. The next time the system enters the while loop is the next frame of the animation. Therefore, we can control the frame rate by introducing a delay inside the while loop. Figure 7 shows a simplified version of the game architecture, including the modules in the game software.

4.3 Software Implementation

Our multimedia game is implemented in C and C++. It consists of three sets of files, with each set composed of a header and an implementation file. The sets include the following:

- FORCE.h: Header file for FORCE.c, containing the definition of circle types, the data structure of circles and methods.
- FORCE.c: Main program file of the game
- INPUT.h: Header file for INPUT.c
- INPUT.c: Provide methods to handle force feedback mechanism (e.g. initialization, enumeration, acquiring device, etc.)
- DDUTIL.h: Header file for DDUTIL.cpp
- DDUTIL.cpp: Provide DirectDraw and DirectSound utility functions to simplify the animation implementation and synthesize sound in FORCE.c

4.3.1. *Data structures.* The major data structure in this game is the 'Circle' data structure. There are two types of circles in the game: MyCircle and FreeCircle. MyCircle is the only circle in the game that is associated with and controlled by the force feedback device. FreeCircles are an arbitrary number of circles that move around in the game environment and serve as moving obstacles to MyCircle. All the circle objects that exist in the game environment are linked together using a doubly linked list.

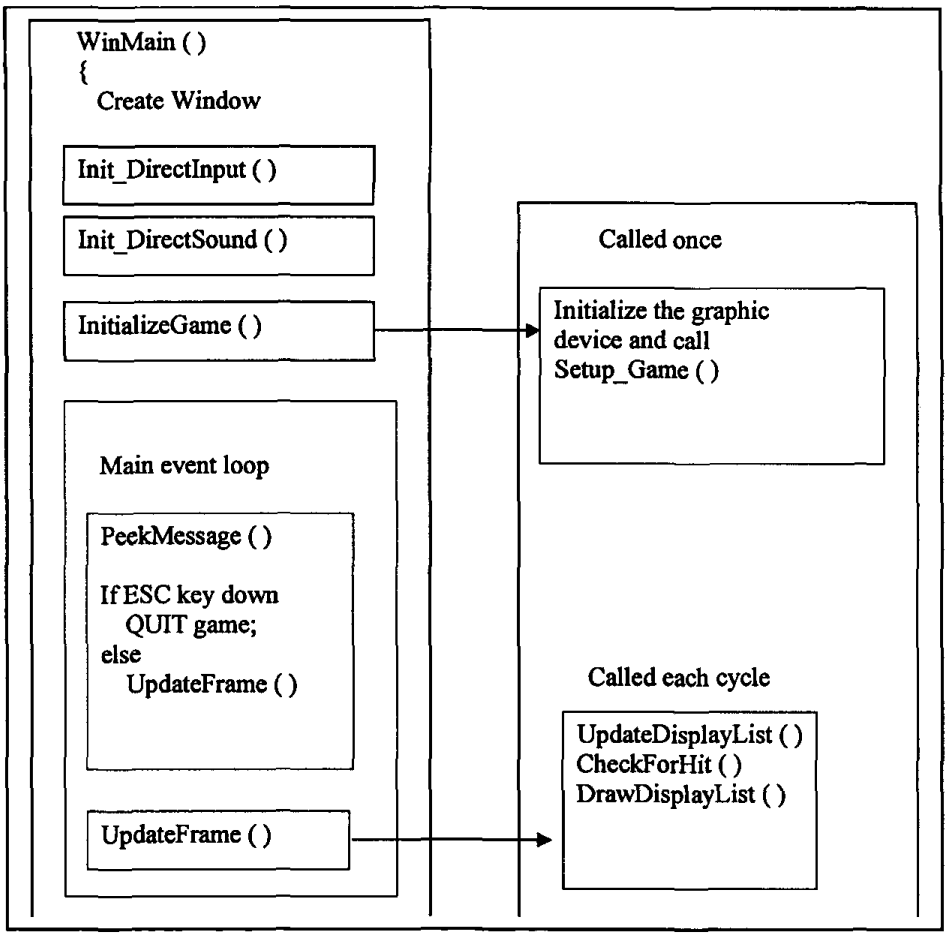

Fig. 7: Game architecture

MyCircle is always at the head of this linked-list. The reason for using a doubly linked list is that we can dynamically add or delete any circle object. The data structure for Circle contains the major attributes of a moving circle object including the following:

1. the circle type
2. x_coordinate
3. y_coordinate
4. x_velocity
5. y_velocity
6. mass
7. and other parameters related to animation such the frame position of the circle's bitmap

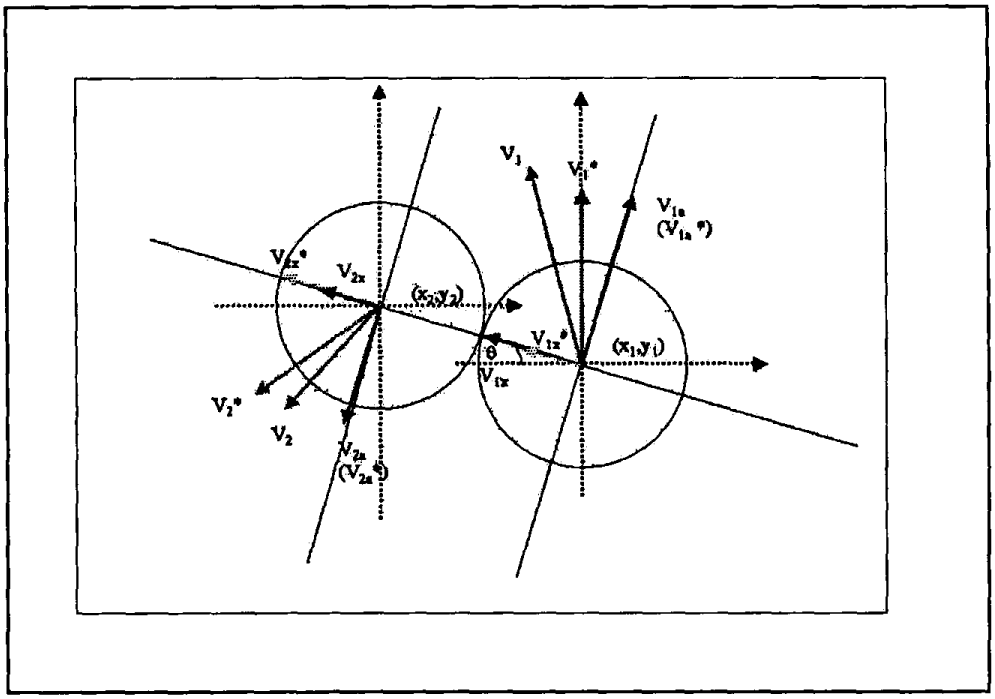

Fig. 8: Elastic collision

4.3.2. *Main Algorithms.* The two major algorithms in the game are the collision detection algorithm and the collision handling algorithm. The collision detection algorithm detects a collision when the distance of the centers of two colliding circles is less than the sum of the radii of the two circles. The collision handling algorithm computes new velocities after the occurrence of a collision. The velocities for the two colliding circles are computed using the laws of conservation of momentum and conservation of energy. The angle of collision is taken into account in the computations. We can see how to compute the final velocities of circles and the angle of collision, using Fig. 8. The collision is similar to a collision of balls observed in a billiard game. The pseudo-code of elastic collision is included in Fig. 9.

```
void Compute_Vel( LPDBLNODE target1, LPDBLNODE target2 )
{
//coefficients
    double  a, b, c, d
//final velocities
    double  velx_1f, velx_2f, vely_1f, vely_2f;
//distance and angle
    double  x, y, dist_x, dist_y, dist, sina, cosa;
//intermediate velocities
    double  v1x, v1a, v2x, v2a;
    double  v1x_f, v1a_f, v2x_f, v2a_f;

//according to Conservation of Momentum and Conservation of energy to
//calculate coefficients
    a = (target1->mass - target2->mass)/(target1->mass + target2-
>mass);
    b = (2 * target2->mass ) / ( target1->mass + target2->mass );
    c = (2 * target1->mass ) / ( target1->mass + target2->mass );
    d = (target2->mass - target1->mass)/(target1->mass + target2-
>mass);

//compute the distance between two colliding balls
    x = (target1->dst.left + target1->dst.right) / 2;
    y = (target1->dst.top + target1->dst.bottom) / 2;
    dist_x = x - (target2->dst.left + target2->dst.right)/2;
    dist_y = y - (target2->dst.top + target2->dst.bottom)/2;
    dist = sqrt(dist_x*dist_x + dist_y*dist_y);

//compute the angle of collision
    cosa = (target1->posx - target2->posx)/dist;
    sina = (target1->posy - target2->posy)/dist;

//resolve the x and y components of the balls velocities into the
//direction of collision and the direction perpendicular to it
    v1x = target1->velx * cosa + target1->vely * sina;
    v1a = target1->velx * sina - target1->vely * cosa;
    v2x = target2->velx * cosa + target2->vely * sina;
    v2a = target2->velx * sina - target2->vely * cosa;
    v1x_f = a * v1x + b * v2x;
    v2x_f = c * v1x + d * v2x;
    v1a_f = v1a;
    v2a_f = v2a;

//resolve the resulting velocities (in the direction of collision) of
//the wo balls back into x and y components
    velx_1f = v1a_f * sina + v1x_f * cosa;
    vely_1f = - v1a_f * cosa + v1x_f * sina;
    velx_2f = v2a_f * sina + v2x_f * cosa;
    vely_2f = - v2a_f * cosa + v2x_f * sina;
    target1->velx = velx_1f;
    target1->vely = vely_1f;
    target2->velx = velx_2f;
    target2->vely = vely_2f;
```

Fig. 9: Pseudo-code of elastic collision.

5. CONCLUSION

This paper has described the design and development of a multimedia game incorporating force feedback. This game was used as a test-bed to perform human subject experiments, in which different types of multimedia interfaces were tested, varying combinations, and parameters of sound and force feedback with and without time delays (Fei & Agah, 2000). The analysis of experimental results helps with the comparative study of different types of multimedia combinations.

This multimedia game had a limited precision that was due to the collision detection algorithm and thus caused certain side effects. When two circles collided too quickly, and the collision speed was not high enough, the successive two or more distances were always less than the collision distance. At this circumstance, two circles could not be split up, and collisions continuously happened. We developed work-arounds to reduce this effect into minimum. However, this is not the situation in reality. In addition, it is hard to derive the two-dimension collision algorithm, which will fit into the system. The algorithm became very complicated when the number of circles increased, and the angles of collisions were considered as huge numbers of cases had to be considered. Assumptions were made to reduce the complexity of the algorithm. One such assumption was that the chances of all the circles in the game environment colliding into each other at the same time were almost zero.

We found that another limitation in our approach was the asynchronization limitation. When the speed of a circle was too high, more collisions would happen. The bad situation was when the successive collisions happened too quickly, and the player would feel the 'shake' of the joystick but would not see the collision happen. The reason was that the computation of force and force applying through the joystick was lower than the frame update speed. In general, data conversion and computer speed limit the attainable sampling rate in force display (Minsky *et al.*, 1990; Miyasato & Akatsu, 1997). For our system, 30 frames per second performance proved sufficient for acceptable visual illusions of fluid movements in the game. Low sampling rate can make the system unstable, however. We can lower the velocity in x and y directions according to joystick's x speed and y speed to let the player easily control the circle and reduce the collision speed.

In our future work we hope to support force feedback joystick devices with I-Force 2.0 (24 force effects), which would allow us to add more features to the circles and thus more varying force effects. Moreover, we can integrate network gaming capabilities to the game for players connected through a LAN or the Internet. They even can control the circles in a common environment, bumping into each other.

Although there are some computer multimedia games that can be played through the network, incorporating the force feedback into the network is a new idea for the game industry. Through this game, players can dynamically feel the presence of their opponents. In this interactive game, the players will share the same environment, interacting through modems directly connected to each other or indirectly connected via a network. They share the information used for the interaction. With using similar joysticks, the data produced by them should be transferred mutually. Usually, the data are used to determine the player's position and the player's actions in an ordinary game. After the data are transferred to the opponent's computer, the position representing the local player's joystick is produced, and the computer can determine the process of the game. When using a force feedback joystick, the concept of the interactive game implementation is the same, except that the computer has to process much more data for representing the forces. By carefully designing the game, the data needed to represent the forces can be decreased, but the other data that should be transferred remain the same as those for using the common joystick.

REFERENCES

Barnett. 1999. http://www.barnett.sk/software/bbook/directx.

Bergamasco, M. 1997. Haptic interfaces: the study of force and tactile feedback systems, *Proceedings of IEEE International Workshop on Robot and Human Communication.*

Browsebooks. 1999. http://www.browsebooks.com/Kovach.

Buttolo, P., Oboe, R., Hannaford, B. and McNeely, B. 1996. Force feedback in shared virtual simulations, *Proceedings MICAD*, Paris, France.

Clark, J. 1998. May the force feedback be with you: grappling with DirectX and DirectInput, *Microsoft Systems Journal.*

DirectX. , 1999. http://www.directxfaq.com/general.htm.

He, Fei. 1999. *Enhanced multimedia human-computer interaction using force feed-back*, M.S. Thesis, Department of Electrical Engineering and Computer Science, The University of Kansas.

He, Fei and Agah, Arvin. 2000. Effectiveness of incorporating force feedback into multimedia interfaces, *Proceedings of The Second International Forum on Multimedia & Image Processing (IFMIP'00), World Automation Congress (WAC'00)*, Maui, Hawaii.

Force-feedback. 1998. http://www.force-feedback.com.

Geocities. 1998. http://www.geocities.com/SiliconValley/Way/3390.

Kavach, P.J. 1997. *The awesome power of Direct3D/DirectX*, Manning Publications Co.

Lamothe, A. 1997. Windows Game Programming For Dummies", IDG Books.

Minsky, M., Ouh-Young, M., Steels, O., Brooks, F.P. and Behensky, M. , 1990. Feeling and seeing: issues in force display, *ACM 1990*, 235--243.

Miyasato, T. and Nakatsu, R. 1997. Allowable delay between images and tactile information in a haptic interface, *Proceedings of IEEE International Workshop on Robot and Human Communication.*

Rosenberg, L.B. 1997. A force feedback programming primer, *Immersion Corporation.*

Wksoftware. 1999. http://www.wksoftware.com.