ISBN 978-89-6211-399-0 98560

VRJuggler: VR 응용 시스템 구축을 위한 통합 라이브러리

허 영 주 (popea@kisti.re.kr)

한 국 과 학 기 술 정 보 연 구 원
Korea Institute of Science & Technology Information

목차

그림 차례

소스 차례

1. 서론

VRJuggler는 가상현실 애플리케이션 개발에 필요한 플랫폼을 제공하는 통합 환경 라이브러리로, VRJuggler를 사용하면 사용자는 거의 모든 종류의 VR 시스템에서 애플리케이션을 실행하는 것이 가능하다. 즉, VRJuggler는 다른 Juggler 요소들 간의 "접착제" 역할을 수행하는 것으로, PC같은 단일 데스크탑 시스템에서부터 하이-엔드 워크 스테이션이나 슈퍼컴퓨터와 맞물려 돌아가는 복잡한 멀티-스크린 시스템에 이르기까지, 다양한 형태의 시스템에 적용할 수 있다.

VRJuggler 프로젝트는 1997년 아이오와 주립 대학(Iowa State University)의 VR 애플리케이션 센터에서 시작됐으며, 오픈 소스와 커뮤니티 활동을 기반으로 현재도 계속 기능이 추가되고 있다. 이렇게 프로젝트가 지속되다보니 여타의 다른 VR 환경에 대한 프레임워크보다는 그 안정성이 매우 뛰어나기 때문에 여러 다양한 분야의 VR 애플리케이션에서 사용되고 있다.

그러나 이렇게 다양한 기능을 다루다보니, 여러 복잡한 모듈이 추가돼 있으며, 사용자가 처음부터 이런 복잡한 기능을 분석해서 사용하기가 쉽지 않다. 본 문서에서는 VRJuggler의 복잡한 모듈과 기능에 대한 설명과 함께 VRJuggler에 대한 전반적인 분석 내용을 다룰 것이다.

2. 개요

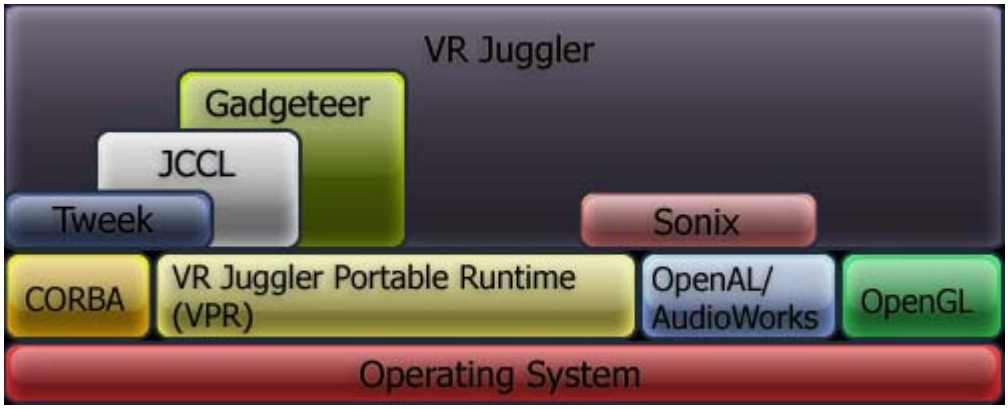
VRJuggler는 PC같은 단순한 데스크탑 시스템에서부터 클러스터나 워크스테이션, 혹은 슈퍼컴퓨터 시스템에 탑재되는 복잡한 멀티-스크린 시스템에 이르기까지 확장이 가능한 가상현실 애플리케이션 구축을 위한 플랫폼이다. VRJuggler는 매우 유연성이 높으며, 이로 인하여 데스크탑 VR, HMD, CAVE™와 유사한 디바이스, 혹은 Powerwall™같은 디바이스를 포함하는 다양한 VR 시스템에서 애플리케이션을 실행할 수 있다. 또, 이식성이 높으며 IRIX, Linux, Windows, FreeBSD, Solaris 및 Mac OS X같은 다양한 OS를 지원한다.

VRJuggler가 수행하는 기능은 매우 다양하며, 여러 다양한 툴을 기반으로 다음과 같은 기능을 지원한다.

- VRJuggler 가상 플랫폼
- Gadgeteer라는 디바이스 관리 시스템 (I/O 디바이스에 대한 로컬, 혹은 원격 접속)
- 수학 템플릿 라이브러리
- 크로스-플랫폼 쓰레드, 소켓 및 직렬 포트 프리미티브를 제공하는 런타임
- Sonix라는 사운드 매니저
- Tweek이라는 분산 모델-뷰 컨트롤러
- XML을 기반으로 하는 설정 시스템

3. VRJuggler의 구조

VRJuggler의 구조는 [그림 3-1]과 같다. VRJuggler는 Juggler뿐만 아니라 Gadgeteer, JCCL, Tweek, Sonix 등의 여러 다양한 구성요소로 구성돼 있다. VRJuggler는 다양한 Juggler 구성요소들을 연결시키는 접착체 역할을 하는 구성요소라 할 수 있으며, VR 애플리케이션 개발이 필수적인 플랫폼의 역할을 수행한다. Gadgeteer는 일종의 디바이스 관리 시스템으로 시스템 설정, 제어, 인식 부분을 다루며 VR 디바이스로부터 들어오는 데이터를 처리하는 역할을 수행한다. JCCL은 XML을 기반으로 하는 설정 시스템으로 VRJuggler 사용자들은 모든 시스템 설정에 이 모듈을 사용하게 된다. 다음에는 VRJuggler의 각 구성요소에 대해 좀더 자세히 알아보기로 한다.



[그림 3-1] VRJuggler의 구성요소

가. VRJuggler

앞서 여러번 설명했듯이 VRJuggler는 VR 애플리케이션 개발과 관련된 플랫폼을 제공하며, 애플리케이션과 VR 시스템을 연결하는 일종의 “glue" 역할을 수행한다. 확장성이 커서 PC같은 단순한 데스크탑 시스템에서부터 클러스터나 최첨단 워크스테이션, 슈퍼컴퓨터에서 수행되는 멀티-스크린 시스템에 이르기까지, 모든 다양한 종류의 디스플레이 시스템을 지원할 수 있다. 또, 데스크탑 HMD, CAVE™, 혹은 Powerwall™같은 다양한 형태의 VR 디바이스를 지원한다.

나. Gadgeteer

Gadgeteer는 일종의 하드웨어 디바이스 관리 시스템으로 시스템 설정, 제어, 인식 부분을 다루며 VR 디바이스로부터 들어오는 데이터를 처리하는 역할을 수행한다. 즉, VRJuggler 애플리케이션에서 사용할 수 있도록 물리적인 디바이스 입력을 처리하는 역할을 수행하는데, 동적으로 확장 가능한 입력 매니저를 포함하며, 이 입력 매니저는 “positional", "digital", "gesture" 등의 추상적인 개념으로 디바이스를 처리한다. 또, 컴퓨터간에 디바이스 샘플을 공유할 수 있는 원격 입력 매니저(Remote Input Manager)도 포함한다.

Gadgeteer를 사용해서 입력 디바이스 하드웨어를 한단계 추상화함으로써, 사용자는 다양한 디바이스를 이용할 수 있는 몰입형 소프트웨어를 제작하는 것이 가능하다. 몰입형 애플리케이션은 특정 디바이스와 직접 연계됨으로써, 몰입형 하드웨어 설정(immersive hardware configuration)간의 애플리케이션 이식성에 제약이 생기는 현상을 방지할 수 있으며, 특정 디바이스에 대한 상세한 지식 없이도 서로 다른 하드웨어 설정간 애플리케이션 이식을 손쉽게 수행할 수 있다.

Gadgeteer에서 사용하는 입력 디바이스의 종류는 다음과 같다.

- Analog
  - 연속 구간에 속한 입력값을 뜻하며, 디지털 컴퓨터에서 아날로그 값은 시뮬레이션을 통해 계산한다.
  - Gadgeteer에서는 이 시뮬레이션을 floating-point 형태로 수행한다.

- Digital
  - 주로 “On"과 ”Off" 상태를 갖는 버튼 디바이스에 적합한 개념이지만, 이 2가지 값 외에도 몇 가지 다른 값을 가질 수 있다.
  - On/Off/Toggle On(이전 프레임에서 Off상태였다가 On으로 바뀜)/Toggle Off(이전 프레임에서 On 상태였다가 Off 상태로 바뀜), 이 4가지 형태의 값을 가지는 디바이스
- Position
  - Polhemus Fastrak이나 Ascension MotionStar같이 6DOF 트래커로부터 정보를 얻는 디바이스
  - 일반적으로 리턴값은 특정 트래커의 위치와 방향을 나타내는 4 x 4 transformation 행렬이 된다.
- Simulator
  - 6DOF 트래커가 없는 환경에서 VR 애플리케이션을 사용할 경우, 마우스와 키보드로 6DOF 를 시뮬레이트할 수 있게 해주는 디바이스 모드
- Command
- Glove

이 카테고리 내에서는 서로 다른 벤더가 만든 디바이스라 할지라도 동일한 형태의 데이터를 리턴한다. 애플리케이션 개발자는 이 추상화된 입력 데이터에 근거해서 개발을 진행하면 된다.

다. JCCL

JCCL은 XML을 기반으로 한 설정 시스템으로, 애플리케이션에 대한 설정/성능 모니터링 툴이라 할 수 있다. JCCL로 빌드된 애플리케이션은 XML 파일로 저장된 설정 정보에 손쉽게 접근할 수 있다.

JCCL 설정 데이터는 JCCL의 이형 타입 시스템을 사용해서 접근할 수 있다. JCCL은 VJ Control이라는 자바 기반의 GUI를 제공함으로써 이런 설정 파일을 편집하는 기능을 제공한다. 또, JCCL은 애플리케이션의 성능을 모니터링하는 툴(Java GUI & C++ 클래스)도 포함한다.

JCCL GUI는 실행중인 애플리케이션의 상태를 제어할 수 있는 툴도 제공한다. JCCL은 애플리케이션 쪽에서 실행시간에 설정을 바꿀 수

있게 해주는 툴도 제공한다. 따라서 애플리케이션 사용자들은 VJ Control GUI같은 외부 툴을 사용해서 실행시간에 애플리케이션의 상태를 변경할 수도 있다.

라. VPR(VRjuggler Portable Runtime)

VPR은 일반 OS 기능에 cross-platform, 객체 지향 추상 레이어를 덧붙인 것으로, Gadgeteer, Tweek, VRJuggler 및 다른 미들웨어의 이식성에 대한 핵심 요소라고 볼 수 있다. 즉, 쓰레드, 소켓(TCP/UDP) 및 직렬 I/O 프리미티브에 대해 플랫폼에 독립적인 추상화를 제공하는 것으로, 일반적으로 VPR에서 개발된 소프트웨어는 IRIX, Linux, Windows, FreeBSD, SOlaris에서 별다른 수정 없이 컴파일하는 것이 가능하다.

내부적으로 VPR은 기본적인 유틸리티 클래스의 집합체로서, BSD 소켓, POSIX 쓰레드, Win32 쓰레드 및 Win32 overlapped I/O같은 플랫폼에 종속적인 API를 한단계 추상화하는 역할을 수행한다.

마. Sonix

Sonix는 일종의 오디오 라이브러리로서 오디오 하드웨어, 혹은 오디오 API에 대한 고급 추상화를 제공한다.

바. Tweek

플러그인으로 구성된 Java GUI와 C++ 애플리케이션간의 매개체 역할을 수행하는 툴로, C++, Java, JavaBeans, CORBA같은 여러 다양한 기술의 집합체다. Java API와 C++ API, 이 두부분으로 구성되며, C++로 정의된 오브젝트를 Java로 정의된 오브젝트에서 접근하거나 그 반대의 동작을 하는것도 가능하며, C++ 오브젝트를 Tweek Java GUI에서 수정하는 것도 가능하다. 또, 언어간 소통을 최소화했기 때문에, 각 기술에 대한 지식이 없더라도 VR 애플리케이션 내에서 Tweek을 사용하는 것이 가능하다.

1) C++ API

Tweek 소프트웨어 시스템의 핵심은 Observer 패턴으로 볼 수 있으며, 이 패턴은 Java GUI(Observer)와 C++ 애플리케이션(Subject)간

의 관계를 정의하는 데 사용된다. C++ API는 Subject/Observer 패턴과 Subject manager 및 CORBA Manager로 구성된다.

- Subject
  - C++ 애플리케이션의 일부로 구성되며, communication channel은 subject의 인터페이스로 정의된다.
  - Subject에는 observer가 붙게 되며, subject의 상태가 바뀔 때마다 그 subject에 붙어있는 objserver에게 그 사실을 공지하게 된다.
  - Tweek C++ API는 기본적인 subject 인터페이스(tweek::Subject)를 정의하며, 이 인터페이스는 subject 패턴을 정의한다.
  - Tweek C++ API를 사용하려면 기본 subject implementation (tweek::SubjectImpl)에서 클래스를 파생해서 그것을 확장해야 하는데, 이 확장에는 다음 2가지 사항을 반드시 지켜야 한다. 우선 이 인터페이스는 IDL(Interface Definition Language)를 정의해야 하며, 2번째로 인터페이스 자체는 반드시 C++ 코드로 작성해야 한다.
- Observer
  - Tweek의 observer는 Java GUI의 일부이며, 원격의 subject 상태를 감시하고 이 상태를 시각적으로 렌더링한다.
  - Tweek C++ API에서는 tweek::Observer라는 기본 observer 인터페이스를 정의한다.
  - tweek::SubjectImpl에 대응되는 표준 observer implementation은 없지만, observer는 반드시 subject에 부합돼야 하며, IDL을 사용해서 기본 observer 인터페이스를 확장할 필요는 없다.
  - Observer는 기본 observer 클래스(Java의 tweek.Observer POA나 C++의 POA_tweek::Observer)를 상속해서 update() 메소드를 구현하면 된다.
- CORBA Manager
  - CORBA 사용상의 복잡성을 낮추기 위한 방편으로 마련된 장치로, 주 임무는 localORB의 초기화이며, POA(Portable

Object Adapter)를 생성해서 Naming Service에 대한 초기 레퍼런스를 풀고 ORB에 대한 쓰레드를 시작해서 request를 처리하는 것으로 초기화 작업은 끝난다.

- local ORB가 초기화되면 Subject Manager가 생성된다. 이 작업 역시 CORBA Manager가 수행하는데, Subject Manager도 CORBA 오브젝트중 하나이기 때문.

- Subject Manager
  - CORBA의 사용을 보다 손쉽게 만들어주는 구성요소로, high-level에서 보면 CORBA 네이밍 서비스에 대한 단순화된 버전으로 볼 수 있다.
  - Tweek 사용자들은 Subject Manager를 사용해서 subject servant를 등록한다.
  - Subject Manager는 CORBA 등록과 servant들의 활성화를 담당한다. subject를 등록하고 나면 사용자가 정의한 symbolic string으로 접근할 수 있으며, Subject Manager에 등록된 subject 집합 내에서 각 subject에 대한 식별자도 볼 수 있다.

2) CORBA

- Common Object Request Broker Architectur의 약자로 분산 프로그래밍에 사용되는 강력한 툴이다.
- 어떤 하드웨어에 설치된 어떤 OS상에서 실행되는 어떤 프로그래밍 언어로 씌여진 소프트웨어건 간에 상관 없이 상호 communication을 가능하게 해주는 구조다.
- 오브젝트와 메소드 파라미터의 모든 serialization 및 de-serialization을 처리하기 때문에 프로그래머들은 시스템 호환성에 대해 고려할 필요가 없다.
- Tweek에서는 CORBA를 사용해서 C++ 애플리케이션과 Tweek Java GUI간 communication을 수행한다. local ORB에 등록된 C++ 오브젝트는 Subject Manager를 통해 Java GUI에서도 사용하는 것이 가능하며, C++, Java, Python과 같은 프로그래밍 언어를 지원한다.

4. 애플리케이션 구조

가. 애플리케이션 구조와 기능

VRJuggler는 애플리케이션 오브젝트(Application Object)를 통해 애플리케이션을 구현할 수 있다. 모든 애플리케이션은 커널이 처리할 수 있는 오브젝트 형태로 구현돼야 하므로, 따라서 오브젝트는 애플리케이션과 동일 개념으로 볼 수 있다.

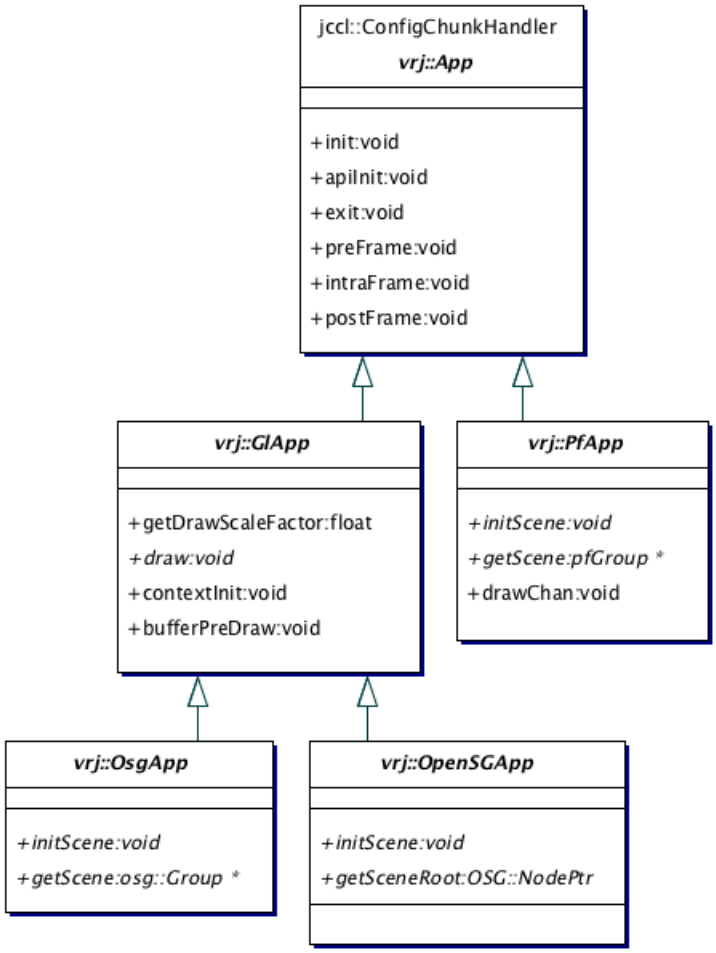
애플리케이션 오브젝트의 구현은 커널과 Draw Manager에 정의된 기본 인터페이스를 구현하는 것으로 시작된다. 커널이 요구하는 인터페이스를 위한 기본 클래스는 vrj::App이며, Draw Manager 인터페이스가 처리하는 기본 클래스로는 vrj::PfApp, vrj::GlApp 등을 들 수 있다.

VRJuggler 애플리케이션에 대한 클래스 계층 구조는 [그림 4-1]과 같다. vrj::App 클래스는 애플리케이션의 초기화, 셧다운 및 실행에 필요한 메소드로 구성돼 있으며, VRJuggler 커널이 처리할 수 있는 추상 클래스다. vrj::PfApp, vrj::GlApp, vrj::OsgApp, vrj::OpenSGApp는 애플리케이션을 가상현실 환경에서 렌더링하는데 필요한 API에 특정적인 함수들로 구성돼 있다.

VRJuggler 커널은 Python, C#, 혹은 VB.NET으로 애플리케이션 오브젝트를 작성하더라도 이를 추상 인터페이스인 vrj::App의 인스턴스로 인식한다. 따라서 VRJuggler 커널 내에서는 여러 개의 다양한 프로그래밍 언어를 섞어서 사용하는 것도 가능하다.

VRJuggler에서는 애플리케이션이 오브젝트로 취급되기 때문에 상식적으로는 main() 함수가 필요치 않다. 하지만 OS에게 프로그램의 시작점을 알려줘야 한다는 현실적인 이유로 인해, 형식적인 main() 함수가 존재한다. VRJuggler의 main() 함수는 VRJuggler 커널을 구동시킨 뒤, 커널에 실행할 애플리케이션을 넘기는 역할만 수행하며, 그런 뒤 main() 함수는 커널이 셧다운 되기를 기다리는 모드로 돌입한다.

일반적으로 main() 함수는 다음과 같은 형태로 구현한다.



[그림 4-1] VRJuggler 애플리케이션 계층구조

[소스 4-1]에서 1번은 VRJuggler 커널을 (생성하거나) 찾는 단계다. 2번에서는 사용자의 애플리케이션 오브젝트(여기서는 simpleApp)의 인스턴스를 초기화한다. 이 오브젝트는 헤더파일(여기서는 simpleApp.h)에 정의돼 있다. 3번은 커널의 loadConfigFile() 메소드로 설정 파일을 전달하기 위한 코드로, 이 설정 파일은 커맨드 라인에서 입력할 수도 있고, 다른 소스를 활용할 수도 있다. 4번은 start 구문을 실행함으로써 VRJuggler 커널을 실행한다. 이 코드로 인해 커널의 새로운 실행 쓰레드가 생성되며, 커널은 내부 프로세스를 시작하게 된다. 5번에서는 애플리케이션을 커널에 전달하는 역할을 수행한다. 이 메소드로 인해 커널은 다시 재성정되며, 애플리케이션 오브젝트의 멤버 함수를 호출하기 시작한다. 이런 과정을 거치면서 애플리케이션은 VR 시스템에서 구동되기 시작한다.

```
#include <vrj/Kernel/Kernel.h>
#include <simpleApp.h>

int main(int argc, char* argv[])
{
   1  vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel

   2  simpleApp* app     = new simpleApp();        // Create the app object

   3  kernel->loadConfigFile(...);           // Configure the kernel

   4  kernel->start();                      // Start the kernel thread

   5  kernel->setApplication(app);           // Give application to kernel
      kernel->waitForKernelStop();           // Block until kernel stops

      return 0;
}
```

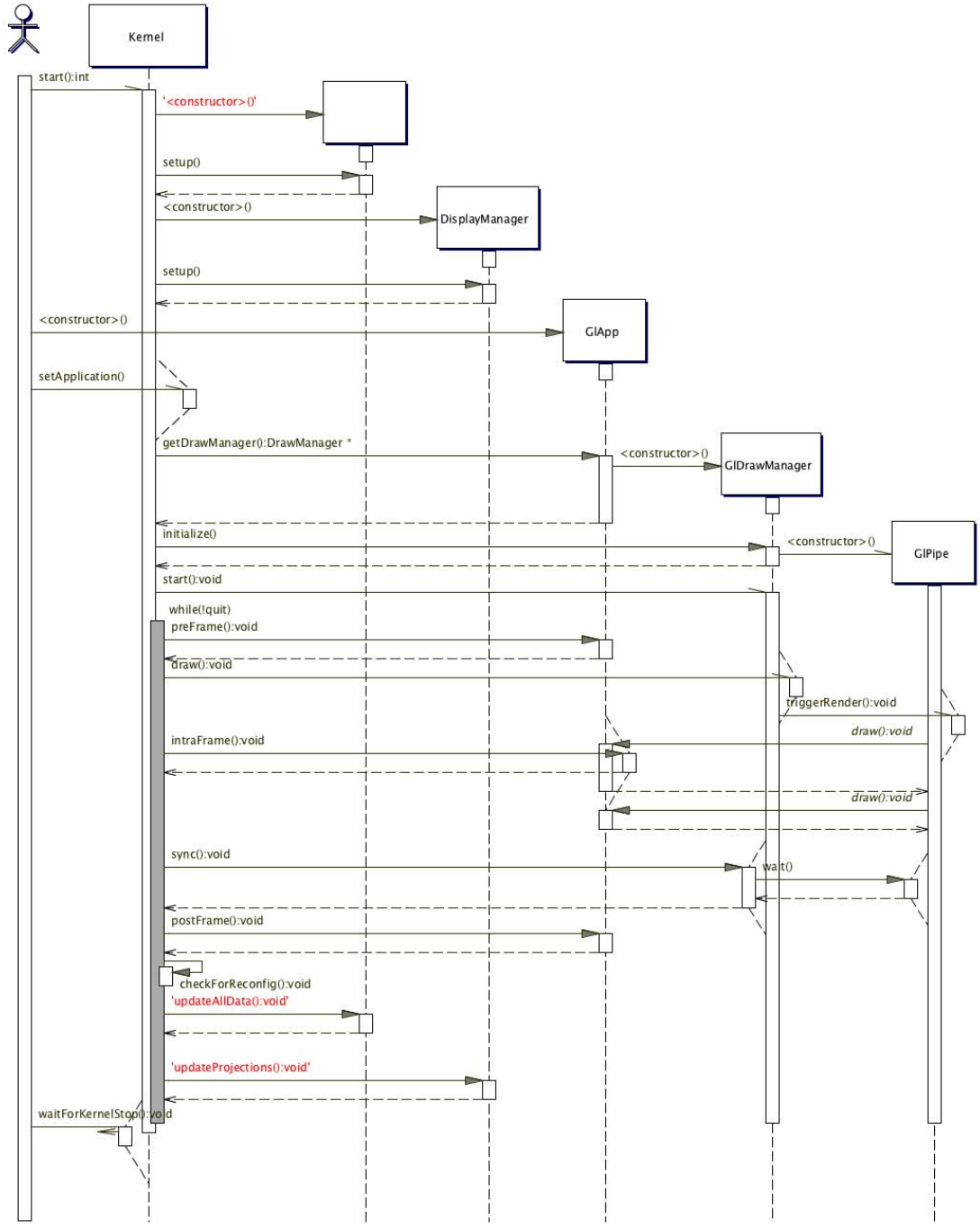
[소스 4-1] VRJuggler의 main()함수

[그림 4-2]는 VRJuggler의 커널 루프(kernel loop)을 나타낸다.

여기에서 프레임(Frame)이라는 것은 애플리케이션 오브젝트 내의 메소드를 호출하는 기본 구성을 나타내며, 그림에서 while(!quit) 구문 내의 모든 실행 코드로 구성된다. 한 frame 동안 커널은 애플리케이션 메소드를 호출하고 updateAllData() 메소드를 호출함으로써 내부적으로 업데이트를 수행한다. 프레임에 대한 제어를 갖고 있는 것은 커널이기 때문에 커널은 애플리케이션이 프로세스를 수행하지 않는 "안전한" 시간 동안에 가상 플랫폼에 대한 설정을 바꾼다던가 하는 등의 변경 작업을 수행한다. 프레임 단위는 애플리케이션을 위한 프레임 워크로도 동작하며, 예를 들자면 애플리케이션은 preFrame()이 호출될 때 디바이스 데이터에 대한 업데이트를 기대할 수도 있다.

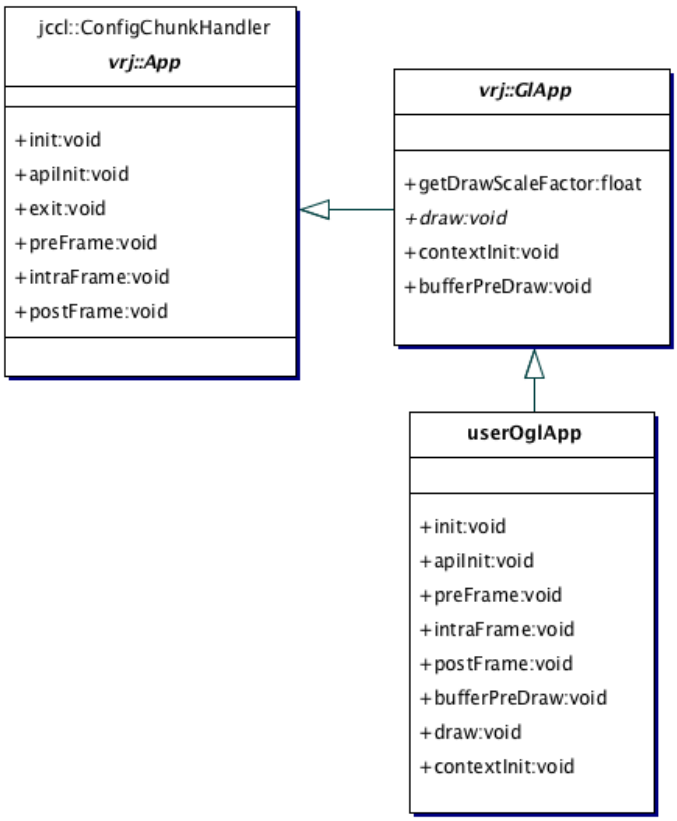
나. 기본 애플리케이션 오브젝트 인터페이스

애플리케이션 오브젝트 인터페이스란 vrj::App에 정의돼 있는 인터페이스로, init(), apiInit(), preFrame(), intraFrame(), postFrame()같은 함수를 기본적으로 정의한다. VRJuggler 커널은 컨트롤 루프에서 이 함수들을 호출함으로써 처리 시간을 할당하며, 주로 초기화와 프레임 관련 계산을 담당한다. 기본 애플리케이션 오브젝트 인터페이스는 [그림 4-3]에 정리돼 있다.



[그림 4-2] 커널 루프의 구조

VRJuggler 애플리케이션의 초기화와 관련된 인터페이스는 vrj::App::init()과 vrj::App::apiInit()이 있다. init()은 커널이 애플리케이션 데이터의 초기화를 위해 호출하는 것으로, 커널은 애플리케이션을 새로 시작할 때 이 메소드를 처음으로 호출하며, 애플리케이션에게 실행 시작이 임박했음을 알리는 역할을 수행한다. init() 메소드는 커널이 애



[그림 4-3] 기본 애플리케이션 오브젝트 인터페이스

플리케이션 실행을 시작할 것을 알아차린 뒤, VRJuggler에 의해 그래픽스 API 처리가 시작되기 전에 실행되며, 데이터 파일 로딩, 룩업 테이블 작성 등, 애플리케이션이 데이터 구조를 셋업하기 위한 사전 처리 작업을 수행하는 단계다.

apiInit() 함수는 애플리케이션이 필요로 하는 그래픽스 API에 특정적인 초기화를 수행하는 멤버 함수로, 그래픽스 API가 apiInit을 통해 초기화되기 전까지는 데이터 멤버가 초기화될 수 없다. OpenGL의 경우에는 API 초기화라는 개념이 없으므로, OpenGL 애플리케이션에 대해서는 따로 작성할 필요가 없다. apiInit() 함수는 그래픽스 API가 시작된 뒤, 커널 프레임이 시작되기 전에 호출되며, 대부분의 경우 씬 그래프 로딩이나 API와 관련된 작업을 수행하는 단계로 사용된다.

VRJuggler 커널에 의해 애플리케이션 오브젝트가 초기화되고 나면 커널 프레임 루프(kernel frame loop)가 시작된다. 프레임에는 매 프레임마다 호출되는 특정 애플리케이션 오브젝트 메소드가 존재하며, 이런 메소드의 호출 시점과 사용법에 대해 알아야만 애플리케이션의 기

능성을 높일 수 있다. 경우에 따라서는 프레임 함수의 멤버를 사용해서 애플리케이션을 최적화하는 것도 가능하다.

vrj::App::GetDrawScaleFactor() 함수는 VRJuggler가 그리는 그림의 단위로 기본값은 피트(feet)다. 이 메소드를 오버라이드하면 VRJuggler의 DrawManager에 의해 scale factor가 변경되면서, 원하는대로 단위를 변경할 수 있다. 이 함수에 대해 gadget/Position/PositionUnitConversion.h에 등록돼 있는 상수 목록은 다음과 같으며, 기본적으로는 [소스 4-2]와 같이 사용할 수 있다.

- gadget::PositionUnitConversion::ConvertToFeet
- gadget::PositionUnitConversion::ConvertToInches
- gadget::PositionUnitConversion::ConvertToMeters
- gadget::PositionUnitConversion::ConvertToCentimeters

```
float vrj::App::getDrawScaleFacto()
{
    return gadget::PositionUnitConversion::ConvertToFeet;
}
```

[소스 4-2] GetDrawScaleFactor() 함수

VRJuggler는 내부적으로 미터 단위로 계산을 수행하며, 다른 단위는 미터 단위로부터 환산된다.

vrj::App::PreFrame() 함수는 시스템이 drawing 작업을 시작할 때 호출되는 메소드로, 이 때 애플리케이션 오브젝트는 입력 디바이스 상태에 따라 데이터에 대한 마지막 업데이트 작업을 수행해야 한다. 이 메소드에서 사용되는 시간이 시스템에서 전체 디바이스의 레이턴시(latency)에 영향을 미치게 된다. PreFrame() 함수는 현재 프레임의 렌더링을 시작하기에 앞서 호출되며, 디바이스 입력에 대한 응답으로 “마지막” 데이터 업데이트를 수행한다.

vrj::App::LatePreFrame() 함수는 vrj::App::Preframe() 함수가 호출된 뒤, 그리고 클러스터 노드 사이에 공유되는, 애플리케이션에 따라 달라지는 데이터가 동기화된 뒤, 씬이 렌더링되기 전에 호출된다. 클러스터 설정에 애플리케이션에 따라 달라지는 데이터를 사용하는 씬

그래프 기반의 애플리케이션 오브젝트는 이 함수에서 가장 최근에 받은 애플리케이션 데이터의 복사본에 기반해서 씬 그래프 업데이트 작업을 수행한다. 반면, 씬 그래프를 사용하지 않는 애플리케이션 오브젝트는 이 메소드나 렌더링 메소드(ex. vrj::glApp)에서 상태를 업데이트한다. 즉, 이 메소드는 애플리케이션에 따라 달라지는 데이터라 클러스터 노드 사이에서 동기화된 뒤, 현재 프레임의 렌더링 작업이 시작되기 전에 수행되며, 클러스터 환경에서 씬 그래프 기반의 애플리케이션 오브젝트로 데이터를 공유할 때, 공유 데이터로부터 데이터를 읽어들이는 노드는 공유 데이터에 대한 최신 업데이트에 기반해서 상태 업데이트 작업을 수행하게 된다.

vrj::App::intraFrame() 함수 내의 코드는 렌더링 메소드와 함께 병렬적으로 실행된다. 즉, 현재 프레임이 그려지는 동안 실행된다는 의미다. 이 메소드는 다음 프레임에 앞서 수행돼야 하는 일을 처리하는 곳으로 사용할 수 있으며, 이 메소드 내에서 병렬 프로세싱을 수행함으로써 프레임레이트의 향상을 기대할 수도 있다. (그리는 작업과 계산 작업의 병렬화로 인한 결과.) 하지만, 렌더링이 수행되는 동안에는 렌더링에 사용되는 데이터가 변경되지 않으므로 데이터 처리에 신경을 써야 한다. 이 메소드는 렌더링이 시작된 뒤, 하지만 렌더링이 끝나기 전에 호출되며, 다음 프레임에 사용될, 시간이 많이 걸리는 계산 작업을 수행하는데 사용할 수 있다.

vrj::App::postFrame() 함수는 커널 프레임 루프의 마지막 단계에서 사용되는 메소드로 입력 데이터에 종속적이지 않거나 렌더링 작업과 겹쳐서 수행할 수 없는 데이터 업데이트 작업을 수행하게 된다. 이 함수는 렌더링 작업이 끝난 뒤, VRJuggler 디바이스나 다른 내부 데이터를 업데이트하기 전에 호출되며, 프레임이 렌더링된 뒤, 혹은 계산 작업이 완료된 뒤의 클리닝 업(cleaning up) 작업에 사용할 수 있다.

다. Draw Manager에 따라 달라지는 애플리케이션 클래스

여기에서 다룰 부분은 주어진 Draw Manager에 따라 달라지는 메소드들로, 특정 Draw Manager에서 사용할 용도로 확장된 애플리케이션 클래스들이다. vrj::App에서 파생된 그래픽스 API 애플리케이션 클래스를 통해 이 인터페이스를 확장할 수 있다. 여기에서는 OpenGL 애플리케이션 클래스와 OpenGL Performer 애플리케이션 클래스에 대

해 간략하게 설명하기로 한다.

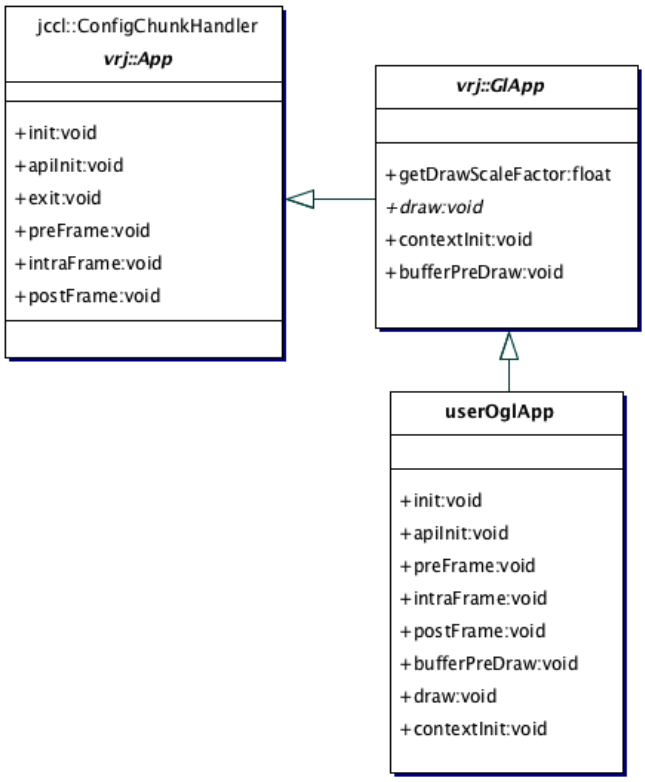
1) OpenGL 애플리케이션 클래스

OpenGL 애플리케이션 클래스는 [그림 4-4]와 같다. 애플리케이션 인터페이스에 몇몇 메소드를 추가하는 형태의 클래스로, OpenGL 그래픽스 컨텍스트를 렌더링하는 역할을 수행한다.

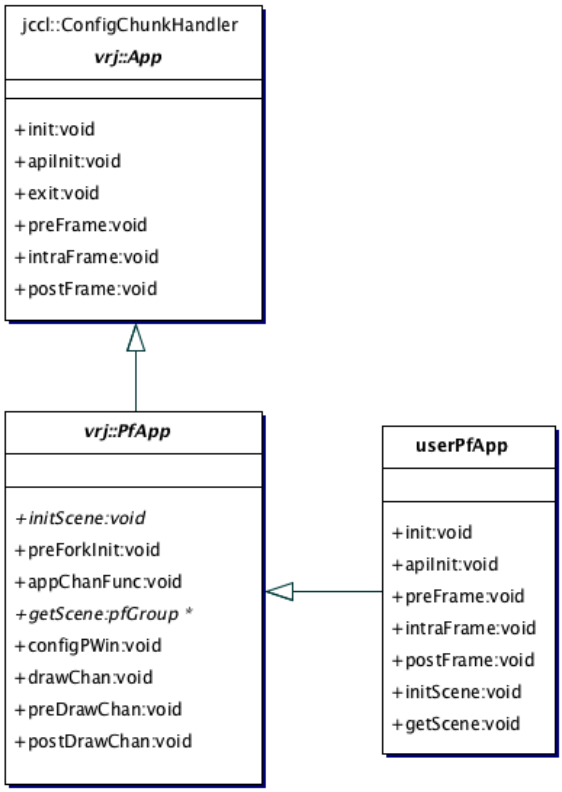
vrj::GlApp::draw() 함수는 현재 씬을 OpenGL 그래픽스 윈도우에서 렌더링할 때 OpenGL Draw Manager가 호출하는 메소드로, 각각의 OpenGL 컨텍스트(context)에 대해 호출된다.

2) OpenGL Performer 애플리케이션 클래스

vrj::PfApp 애플리케이션 클래스는 OpenGL Performer의 씬 그래프를 처리하는 인터페이스 함수로 구성돼 있으며, [그림 4-5]와 같이 구성돼 있다. 이 클래스에서 추가된 대표적인 함수로는 initScene()과 getScene() 함수를 들 수 있다. vrj::PfApp::initScene() 함수는 애플리케이션이 씬 그래프를 생성할 때 호출되는 함수이며, 이 함수는 VRJugg



[그림 4-4] vrj::GlApp 애플리케이션 클래스 구조



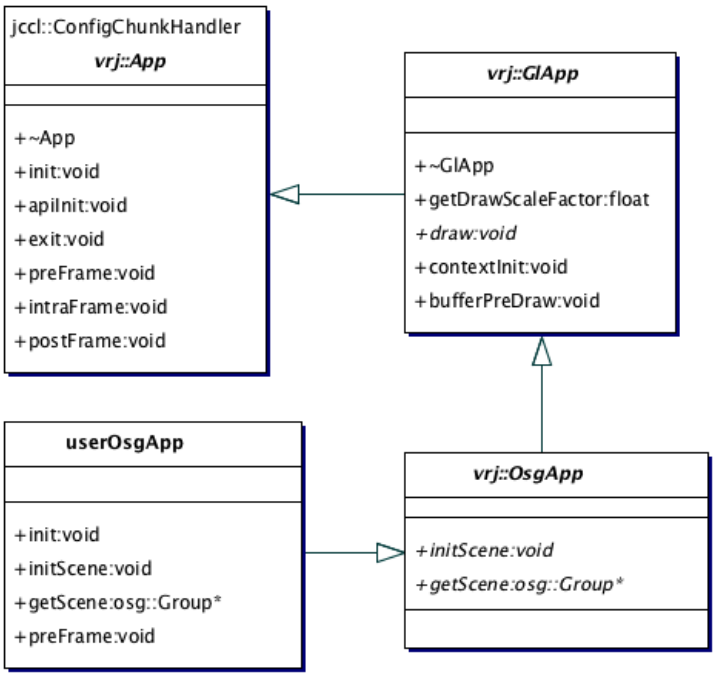
[그림 4-5] vrj::PfApp 애플리케이션 클래스

ler에 의해 OpenGL Performer가 초기화되는 동안, pfInit() 함수와 pfConfig() 함수가 호출된 뒤, vrj::App::apiInit() 함수가 실행되기 전에 호출된다. vrj::PfApp::getScene()은 어떤 씬 그래프를 렌더링할지를 결정할 때 Performer Draw Manager에 의해 호출되는 함수로, 이 함수는 씬 그래프의 루트 노드를 Performer에게 넘겨주는 역할을 수행한다.

3) Open Scene Graph(OSG) 애플리케이션 클래스

오픈 소스 씬 그래프 라이브러리인 Open Scene Graph(OSG)를 기반으로 하는 VRJuggler 애플리케이션은 반드시 vrj::OsgApp 클래스를 상속해야 한다. vrj::OsgApp 클래스는 vrj::GlApp 클래스로부터 파생되며, 씬 그래프를 다루는 데 필요한 몇몇 메소드가 추가된 것이다. OSG 애플리케이션 클래스는 [그림 4-6]과 같다.

vrj::OsgApp 클래스에서 가장 중요한 2가지 메소드는 initScene()과 getScene()이다. 이 두 함수는 OSG 애플리케이션 클래스에서 호출되며, 애플리케이션 씬 그래프를 초기화하고, 씬 그래프의 루트 노드를



[그림 4-6] vrj::OsgApp 애플리케이션 클래스

가져오는데 사용된다. 이 두 함수는 반드시 애플리케이션에서 구현돼야 한다. vrj::OsgApp::initScene() 함수는 씬 그래프를 초기화하는 데 사용되며, 이 함수 내에서는 씬 그래프의 루트 노드를 생성하고 필요한 모듈을 로딩해서 씬 그래프에 덧붙이는 작업을 수행해야 한다. vrj::OsgApp::getScene() 함수는 씬 그래프 루트에 접근하는 데 사용되는 함수로, OSG 애플리케이션 클래스 랩퍼(wrapper)에 의해 호출됨으로써 현재 씬 그래프를 접근하는 데 사용된다.

5. VRJuggler 프로그램

가. 기본 애플리케이션 프로그래밍

기본적으로 모든 VRJuggler 애플리케이션은 기본 애플리케이션 오브젝트 클래스인 vrj::App 클래스로부터 파생된다. vrj::App 클래스에는 기본 인터페이스가 정의돼 있기 때문에, 따라서 애플리케이션을 작성할 때, 사용자가 정의할 애플리케이션 오브젝트는 vrj::App로부터 상속을 받거나 vrj::App를 슈퍼클래스로 가지는 Draw Manager 애플리케이션 클래스로부터 상속을 받아야 한다. [소스 5-1]의 코드는 VRJ

ugglerㄱ 애플리케이션 오브젝트를 필요로 할 때 사용할 수 있는 애플리케이션 클래스다.

```
class userAPP : public vrj::App
{
   public:
      init();
      preFrame();
      postFrame();
 }
```

[소스 5-1] 기본적인 애플리케이션 코드의 기본 형태

대부분의 경우, 사용자의 애플리케이션 클래스는 Draw Manager에 따라 달라지는 애플리케이션 클래스로부터 상속을 받아서 사용하게 된다. [소스 5-2]는 OpenGL 애플리케이션 예제다. 애플리케이션 클래스(userGlApp)는 vrj::GlApp 애플리케이션 클래스로부터 상속된 것이다.

```
class userGlApp : public vrj::GlApp
{
    public:
      init();
      preFrame();
      postFrame();
      draw();
 }
```

[소스 5-2] OpenGL 애플리케이션 코드의 기본 형태

나. OpenGL 애플리케이션

OpenGL 애플리케이션 클래스는 [그림 4-4]를 통해 앞에서 설명했다. OpenGL 애플리케이션은 [그림 4-4]의 vrj::GlApp 클래스로부터 상속을 받아서 구현해야 한다. 함수중 draw() contextInit(), bufferPreDraw() 함수들은 vrj::GlApp 인터페이스에서 추가된 것으로, OpenGL의 drawing 루틴을 처리하고 컨텍스트에 따라 달라지는 데이터를 관리하는 역할을 수행한다.

OpenGL 애플리케이션을 구현하려면 우선 color buffer와 depth buffer를 초기화시키는 작업을 수행해야 한다. 예를 들어 타일형 디스플레이 장치같이 각각의 뷰포트(vierport)가 씬의 특정 부분을 렌더링하는

경우에는 하나의 VRJuggler 디스플레이 윈도우에서 여러개의 OpenGL 뷰포트를 렌더링하는 기능이 유용하다. OpenGL 기반의 애플리케이션에서 다중 뷰포트를 이용하려면 적절한 타이밍에 color buffer와 depth buffer를 깨끗이 지우는 기능이 필요하다. 이런 버퍼 초기화 작업은 bufferPreDraw() 함수와 draw() 함수를 통해 수행한다.

[소스 5-3]에서는 vrj::GlApp::bufferPreDraw() 함수를 오버라이드함으로써 color buffer를 초기화하는 코드를 볼 수 있다.

```
void userApp::bufferPreDraw()
{
   glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
   glClear(GL_COLOR_BUFFER_BIT);
}
```

[소스 5-3] OpenGL 애플리케이션에서의 color buffer 초기화

depth buffer의 초기화는 (stereo rendering 때문에) color buffer 초기화와는 별개로 진행된다. color buffer와는 달리 depth buffer는 애플리케이션 오브젝트의 draw() 메소드에서 초기화해야 하는데, 이 코드는 [소스 5-4]에서 볼 수 있다.

```
 void userApp::draw()
{
   glClear(GL_DEPTH_BUFFER_BIT);

   // Rendering the Scene....
}
```

[소스 5-4] OpenGL 애플리케이션에서의 depth buffer 초기화

vrj::GlApp 클래스 인터페이스의 draw() 멤버 함수는 가상 환경을 그리는 데 필요한 코드를 포함한다. 따라서 OpenGL의 draw와 관련된 코드는 draw() 함수 내에 위치해야 한다. draw() 함수는 OpenGL Draw Manager가 사용자 애플리케이션에 의해 생성된 가상 환경을 렌더링할 때마다 호출되며, multi-surface setup이나 stereo 설정 여부에 따라 프레임당 여러번 호출될 수도 있다. draw() 함수가 호출되면 OpenGL의 모델 뷰 행렬(model view matrix)와 프로젝션 행렬(projection matrix)가 재설정돼서 씬을 그리게 되며, 한 프레임 내에서 호출되는 drawing() 메소드에 대해 입력 디바이스는 동일한 상태(값, 위

치 등)를 유지하게 된다. draw() 메소드 내에서 실행돼야 하는 코드는 OpenGL drawing 루틴뿐이다. 즉, 입력 디바이스의 값을 읽어들이는 것은 허용되지만, 애플리케이션 데이터 멤버에 대한 업데이트는 발생해서는 안된다.

[소스 5-5]는 OpenGl을 이용해서 육각형을 그리는 간단한 코드 예제를 보여준다. 윗부분은 클래스 선언부분이고, 아랫 부분은 draw() 멤버 함수를 나타내고 있다.

```
class simpleApp : public GLApp
{
   public:
      simpleApp();
      virtual void init();
      virtual void draw();
   public:
      PositionInterface mWand;
      PositionInterface mHead;
      DigitalInterface mButton0;
      DigitalInterface mButton1;
}

using namespace gmtl;

void simpleApp::draw()
{
   ...
   // Create box offset matrix
   Matrix44f box_offset;       //gmtl::Matrix44f 오브젝트를
                               //생성함으로써 가상 현실에서
                               // cube의 offset 정의
   const EulerAngleXYZf euler_ang(Math::deg2Rad(-90.0f),
                                  Math::deg2Rad(0.0f),
                                  Math::deg2Rad(0.0f));

   box_offset = gmtl::makeRot<Matrix44f>(euler_ang);
   gmtl::setTrans(box_offset, Vec3f(0.0, 1.0f, 0.0f));
   ...
   glPushMatrix();
      //Push on offset
      glMultMatrixf(box_offset.getData());
      ...
      drawCube();
   glPopMatrix();
   ...
}
```

[소스 5-5] OpenGL 애플리케이션 예제

다. Open Scene Graph 애플리케이션

Open Scene Graph 애플리케이션 클래스는 [그림 4-6]을 통해 앞에서 설명했다. Open Scene Graph 애플리케이션은 vrj::OsgApp 클래스로부터 상속을 받아서 구현해야 하는데, 이 클래스는 vrj::GlApp 클래스에 씬 그래프 관련 메소드를 추가한 것이다.

vrj::OsgApp 클래스에서는 initScene()과 getScene() 메소드가 핵심이며, OSG 애플리케이션 클래스 랩퍼에 의해 호출됨으로써 애플리케이션 씬 그래프를 초기화하고, 씬 그래프의 루트에 접근하는데 사용된다. OSG를 사용하는 애플리케이션은 씬 그래프 사용에 앞서 반드시 씬 그래프를 초기화해야 하며, initScene() 함수를 통해 해당 작업을 수행한다. initScene() 함수에서는 애플리케이션 씬 그래프의 루트를 생성하고, 필요한 모델을 루트에 추가해서 로딩한다. 이렇게 로딩된 씬 그래프를 OSG가 렌더링하게 하려면 씬 그래프 루트에 접속할 수 있어야 한다. getScene() 함수는 OSG 애플리케이션 클래스 랩퍼에 의해 호출되며, 랩퍼가 필요로 할 때(ex. 렌더링이나 업데이트 작업을 할 때마다) 씬 그래프에 접근할 할 수 있게 해준다. 씬 그래프 업데이트에는 preFrame()이나 infraFrame(), 혹은 postFrame() 메소드를 이용한다.

[소스 5-6]에서 [소스 5-9]에 이르는 부분은 OSG를 이용해서 간단한 모델을 로딩해서 렌더링하는 예제다. [소스 5-6]은 클래스 선언부로, osgNav 클래스는 vrj::OsgApp 클래스로부터 상속을 받으며, initScene(), getScene(), configSceneView(), preFrame(), latePreFrame()에 대한 선언을 포함한다.

```
class OsgNav : public vrj::OpenSGApp
{
  public:
     OsgNav(vrj::Kernel* kern, int& argc, char** argv);
     virtual ~OsgNav();

     virtual void configSceneView();

     virtual void initScene();
     void myInit();
```

```
    virtual osg::Group* getScene();

    virtual void preFrame();
    virtual void latePreFrame();

    void setModelFileName(std::string filename);

  private:
    osg::Group*           mRootNode;
    osg::Group*           mNoNav;
    osg::MatrixTransform*   mNavTrans;
    osg::MatrixTransform*   mModelTrans;
    osg::Node*            mModel;

    OsgNavigator  mNavigator;

    std::string   mFileToLoad;

    vpr::Interval mLastPreFrameTime;

  public:
    gadget::PositionInterface  mWand;
    gadget::PositionInterface  mHead;
    gadget::DigitalInterface   mButton0;
    gadget::DigitalInterface   mButton1;
    gadget::DigitalInterface   mButton2;
    gadget::DigitalInterface   mButton3;
    gadget::DigitalInterface   mButton4;
    gadget::DigitalInterface   mButton5;
};
```

[소스 5-6] OSG를 이용한 애플리케이션의 선언부

[소스 5-7]은 initScene() 멤버 함수를 나타내며, 마지막으로 myInit() 함수를 호출한다는 점을 빼면, 다른 애플리케이션 오브젝트의 초기화 함수와 매우 유사하다.

```
void OsgNav::initScene()
 {
    mWand.init("VJWand");
    mHead.init("VJHead");
    mButton0.init("VJButton0");
    mButton1.init("VJButton1");
    mButton2.init("VJButton2");
    mButton3.init("VJButton3");
    mButton4.init("VJButton4");
    mButton5.init("VJButton5");
```

```
    myInit();
  }
```

[소스 5-7] OSG를 이용한 애플리케이션의 initScene() 함수

[소스 5-8]은 initScene() 함수에서 마지막으로 호출한 myInit() 함수에 대한 구현이다. myInit() 함수에서는 씬 초기화 작업을 수행한다. 즉, 씬 그래프의 루트 노드 및 라이팅(lighting) 노드, 그리고 관련 모델을 생성하는 부분이다.

```
void OsgNav::myInit()
{
   //
   //         /-- mNoNav
   // mRootNode
   //        \-- mNavTrans -- mModelTrans -- mModel

   // 트리의 최상위 레벨
   mRootNode = new osg::Group();       // 애플리케이션 씬 그래프를
                                       // 구성하는 노드를 생성
   mNoNav    = new osg::Group();       // 이 애플리케이션은 사용자의
                                       // navigation에 영향을 받는
                                       // 노드와 그렇지 않은 노드,
   mNavTrans = new osg::MatrixTransform();  // 이 2가지로 구성됨.

   mNavigator.init();

   mRootNode->addChild(mNoNav);
   mRootNode->addChild(mNavTrans);

   //Load the model
   std::cout << "Attempting to load file: "   // 앞서 호출한
                  // setModelFileName()을 통해 제공된 모델을 로딩.
             << mFileToLoad << "... " << std::flush;
   mModel = osgDB::readNodeFile(mFileToLoad);
   std::cout << "done." << std::endl;

   // 모델에 대한 transform 노드
   mModelTrans  = new osg::MatrixTransform();
   // 모델의 각도가 바꾸는 데 사용
   mModelTrans->preMult(
       osg::Matrix::rotate(gmtl::Math::deg2Rad(-90.0f),
                           1.0f, 0.0f, 0.0f));

   if(NULL == mModel)
   {
      std::cout << "ERROR: Could not load file: "
                << mFileToLoad << std::endl;
```

```
    }
    else
    {
       // transform에 모델 추가
       mModelTrans->addChild(mModel);
    }

    // 트리에 transform 추가
    mNavTrans->addChild(mModelTrans);  // model transform을
                                  //navigation 가능한 씬그래프
                                              // 브랜치에 붙임.

    // 씬 그래프에 대한 최적화
    osgUtil::Optimizer optimizer;
    optimizer.optimize(mRootNode);  // osgUtil::Optimize 메소드로
                                    // 씬 그래프를 최적화함
```

[소스 5-8] OSG를 이용한 씬 그래프 구성예

[소스 5-9]는 getScene() 함수를 나타낸다.

```
osg::Group* getScene()
{
   return mRootNode;
}
```

[소스 5-9] OSG를 이용한 애플리케이션의 getScene() 함수

모든 OSG를 이용한 VRJuggler 애플리케이션은 위의 예제와 같은 기본 형태를 가지며, 예제를 확장함으로써 다양한 애플리케이션을 구현하는 것이 가능하다.

6. 결론

VRJuggler는 가상현실 애플리케이션 개발에 필요한 플랫폼을 제공하는 통합 환경 라이브러리로, VRJuggler뿐만 아니라 Gadgeteer, JCCL, Tweek, Sonix 등, 가상현실 환경 구축에 필요한 여러 다양한 툴로 구성돼 있다. VRJuggler는 매우 유연성이 높아서 여러 다양한 가상현실 디바이스와 연동이 가능할 뿐만 아니라 설정 파일을 바꾸는 것만으로 똑같은 애플리케이션을 다양한 형태의 기기에 적용하는 것이 가능하다. 또, 디바이스 관리나 사운드 관리, 그래픽스와 관련된 수학 연산에 필요한 툴도 자체적으로 포함하고 있기 때문에, 가상현실 환경에서 필요한 계산을 수행하고 데이터를 디스플레이하는데 있어서 최

상의 유연성과 이식성을 제공하는 라이브러리라 할 수 있다.

또, OpenGL을 비롯해서 Performer, Open Scene Graph 등, 여러 다양한 그래픽스 관련 API와의 호환성을 보장하기 때문에, 가상현실 환경을 구축에 기존에 사용하던 API를 이용할 수 있다. 게다가 지속적으로 기능이 추가되고 있으며, 커뮤니티 활동을 통해 기술적인 문제를 해결할 수 있다는 장점이 더해져서 VRJuggler는 가상현실 환경 구축에 필수적인 라이브러리로 자리를 잡아가고 있다.

하지만, 다양한 환경을 지원하는 대신 그 설정 방법이 까다로울뿐만 아니라 프로그래밍 기술과 디바이스에 대한 기본 지식을 필요로 하기 때문에 진입장벽을 넘기가 쉽지 않을뿐더러, 아직은 트래킹 디바이스에 대한 지원이 부족한 실정이다. 이런 문제점을 해결한다면 VRJuggler는 가상현실 환경을 구축하는 데 있어 최상의 환경을 제공하는, 유용한 툴이 될 것이다.