가상현실 가시화 시스템의 설계 및 구현

2014. 11

KiSTi www.kisti.re.kr 한국과학기술정보연구원

목 차

표 차례

그림 차례

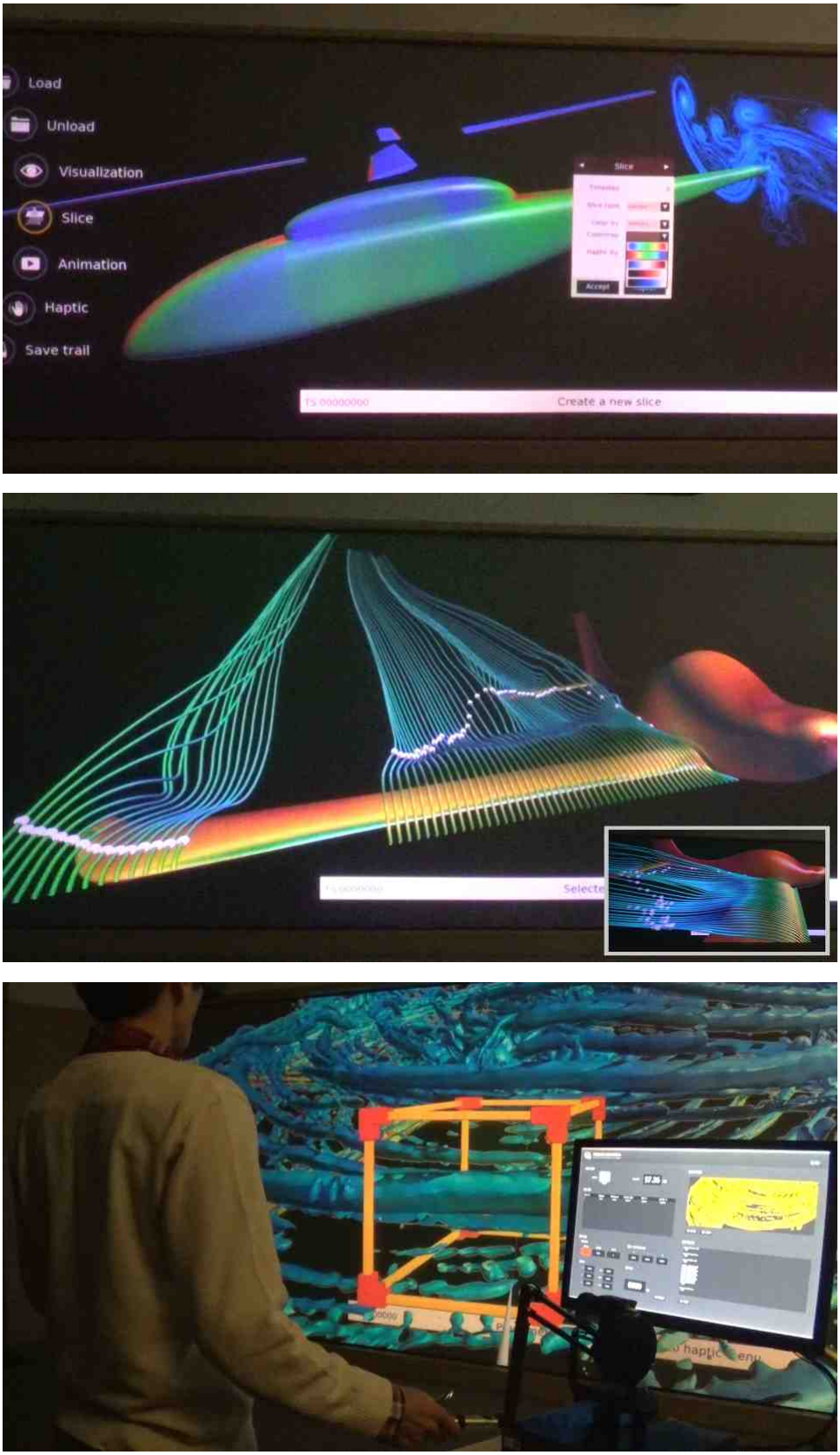
1. 개요

이 문서는 대용량 데이터 가시화 시스템인 GLOVE의 가상현실 인터페이스를 담당하는 GIVI의 내부구조에 대해 설명한다. 원래 GLOVE는 visualization engine에 해당하는 GLORE와 가상현실 인터페이스인 GIVI, 데이터 관리자인 GDM, 네트워크 통신을 담당하는 GIP 등으로 구성되어 있는데, 이 문서는 GIVI로 그 범위를 제한해서 설계/구현내용을 구체적으로 살펴본다. 2014년 11월 현재 GIVI의 소스코드는 [표 1-1]과 같이 구성되어 있다.

구분	라인 수	소스 구성
애니메이션 제어	1,247	애니메이션 IO 쓰레드, 동기화 관련 루틴
햅틱 렌더링	3,198	햅틱 프로토콜 정의, 햅틱 IO 쓰레드
이벤트 필터	3,614	입력장치의 상태에 따른 기능 호출
콘솔 입/출력	11,597	GIVI console 및 콘솔 IO 쓰레드
설정파일	1,508	Runtime config 관리
FSM	17,037	GIVI FSM (state 및 state transition)
UI	14,250	VR 사용자 인터페이스
SceneGraph 및 geometry	8,613	Geometry 관리
Object & Object DB	2,919	Visualization object 정보 관리
VR 위젯	26,924	가상현실 어플리케이션을 위한 위젯
일반	11,774	
합계	102,681	

[표 1-1] GIVI의 소스 구성

이 중 콘솔 입/출력은 독립 어플리케이션으로 작동하기도 하고, GIVI의 실행 제어 및 상태 모니터링 용도로 사용하기 때문에 이 문서에서 별도로 설명하지 않는다. 그리고 VR 위젯은 별도의 기술보고서로 다룰 예정이다.



[그림 1-1] GIVI 실행장면

2. FSM

이론적으로 모든 컴퓨터 프로그램은 FSM(finite state machine)으로 볼 수 있다. 이론적인 관점뿐만 아니라 실제로 프로그램을 개발할 때에도 FSM 엔진을 적용하면 디버깅이 쉽고, 프로그램의 실행 단계별로 구현해야 하는 기능을 구분하기 쉽다는 장점이 있다. 한편, GUI 기반 어플리케이션의 개발은 MFC, Qt, wxWidgets 등의 개발도구를 많이 이용하는데 이 도구들의 공통점은 GUI를 쉽게 개발할 수 있는 일련의 위젯(widget)과 네트워크 입/출력, 그래픽 처리 등을 위한 추상화된 인터페이스를 제공한다는 장점이 있다. 뿐만 아니라 GUI 기반 프로그램은 개별 UI 단위, 또는 이벤트 핸들러 단위로 세부기능을 구현할 수 있도록 해주기 때문에 GUI 자체가 일종의 FSM 역할도 수행한다는 특정을 갖고 있다.
하지만 GIVI의 개발에 사용했던 미들웨어인 VR Juggler는 디스플레이(출력장치)의 view frustum을 관리하거나 가상현실 입력장치로부터 전달받는 최소한의 정보(위치, 방향, 버튼 등)를 어플리케이션에게 전달할 뿐, 어플리케이션 개발을 도와주는 추상화된 인터페이스나 UI 구성을 위한 위젯의 지원이 매우 취약하다. 그렇기 때문에 GUI 어플리케이션을 개발할 때와 같이 GUI를 이용한 FSM의 간접적인 구현/활용이 불가능하고, 기본적인 FSM 엔진을 직접 구현해야 하는 부담이 존재한다. 이 장에서는 GIVI의 FSM 내용(상태 및 상태 전이 다이어그램)과 실제 FSM을 구현한 기법에 대해서 설명한다.

가. FSM 상태 정의

GIVI FSM의 상태(state)는 사용자 인터페이스의 구성과 그 내용이 비슷하다. 하지만 UI의 모든 내용을 일일이 대응시키는 방식으로 FSM state를 만들고, 그것을 기반으로 state transition을 정의하면 그 분량이 지나치게 많아지기 때문에 특정 기능에 대해서는 state를 재활용하는 방식으로 그 수를 조절했다. GIVI의 FSM state는 [표 2-1]과 같이 분류한다.

분류	내용
Initialization / Reset / Quit	프로그램 시작 및 종료
Load data / Unload data	데이터 로드 및 unload
Load trail / Save trail	Trail 로드 및 저장
Visualization (surface/isosurface)	Surface / isosurface visualization
Visualization (streamline/pathline)	Streamline / pathline visualization, update
Visualization (slice)	Slice visualization, update, data 저장
Animation / Timestep jump	애니메이션 재생 및 timestep 변경
Haptic rendering	햅틱 렌더링
Delete	Visualization object 삭제

[표 2-1] GIVI의 FSM state 분류

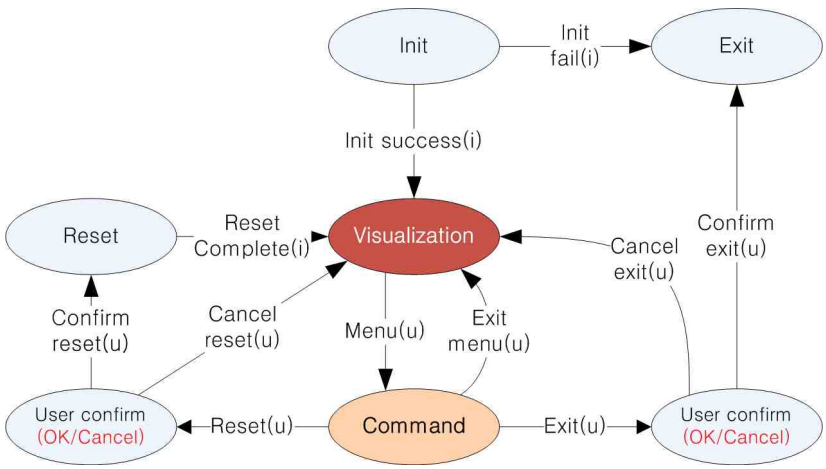
이후 개별 FSM state의 세부내용과 state transition diagram은 [표 2-1]의 분류에 따라서 설명한다.

1) Initialization / Reset / Quit

이 그룹은 프로그램의 시작과 종료, 그리고 프로그램이 실행 중인 동안 모든 작업의 시작과 종료지점이 되는 상태를 정의한다. 기본적으로 프로그램이 시작되면 GIVI는 INIT 상태에 들어가며, 일련의 초기화 과정이 정상적으로 진행되면 VISUALIZATION 상태에서 사용자의 입력을 기다린다. 즉, VISUALIZATION 상태가 모든 작업의 출발점이고, 프로그램 종료를 제외한 다른 모든 작업의 마지막 종료지점이 된다.

State	세부내용
INIT	• 프로그램 시작단계 • 각종 변수, 쓰레드 초기화 • GLORE / 햅틱 서버로의 네트워크 연결
COMMAND	• 사용자 메뉴 내비게이션 • 상황에 따라서 다이얼로그 박스를 통한 사용자 입력 처리
VISUALIZATION	• 모든 visualization 작업의 시작점 • 화면에 있는 visualization object의 조작
RESET CONFIRM	• 사용자가 메뉴에서 'Reset'을 선택했을 때 메시지 박스를 통해서 다시 한 번 사용자 확인을 묻는 상태
RESET	• Scene graph 및 화면, object DB에 존재하는 모든 visualization object 삭제 • 데이터 unload는 진행하지 않음
EXIT CONFIRM	• 사용자가 메뉴에서 'Exit'를 선택했을 때 메시지 박스를 통해서 다시 한 번 사용자 확인을 묻는 상태
EXIT	• 프로그램 종료

[표 2-2] INIT / COMMAND / VISUALIZATION 상태의 세부내용

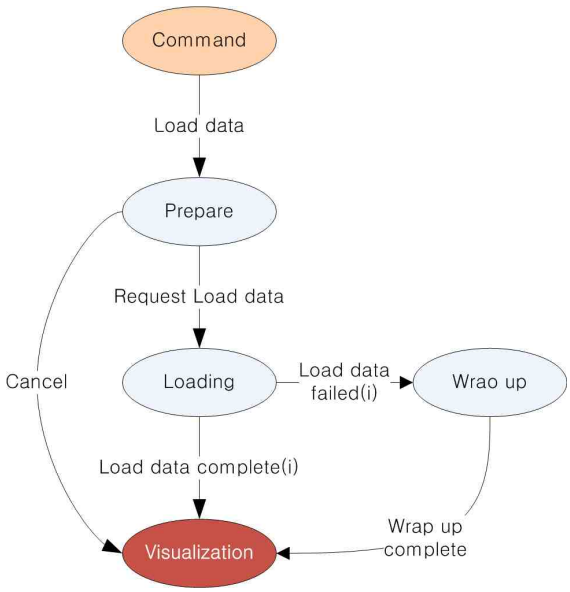


[그림 2-1] INIT, RESET, EXIT state transition diagram

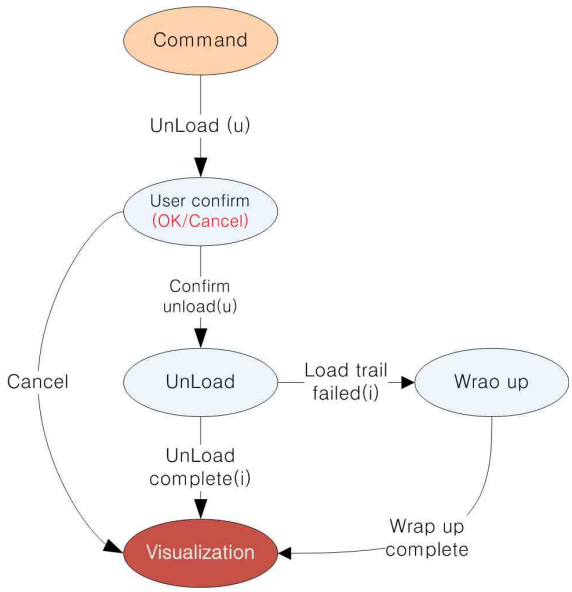
2) Load / Unload data

State	세부내용
LOAD DATA PREPARE	• 사용자 인터페이스를 통해서 GDM/GLORE가 읽을 데이터 파일(meta.xml) 선택
LOAD DATA LOADING	• GDM이 스토리지에 있는 데이터를 읽는 중 • 모든 종류의 사용자 입력을 처리하지 않음
LOAD DATA WRAP UP	• 데이터를 읽는 데에 실패했을 때 데이터를 읽기 전의 상태로 돌아가기 위해 필요한 마무리 작업 수행
UNLOAD CONFIRM	• 사용자가 메뉴에서 'Unload'를 선택했을 때 메시지 박스를 통해서 다시 한 번 사용자 확인을 묻는 상태
UNLOAD	• GLORE에 unload 명령 전송 후 unload 완료까지 대기 • 모든 종류의 사용자 입력을 처리하지 않음
UNLOAD WRAP UP	• Unload가 실패했을 때 이전의 상태로 돌아가기 위해 필요한 마무리 작업 수행

[표 2-3] LOAD / UNLOAD DATA 상태의 세부내용



[그림 2-2] LOAD DATA
state transition diagram

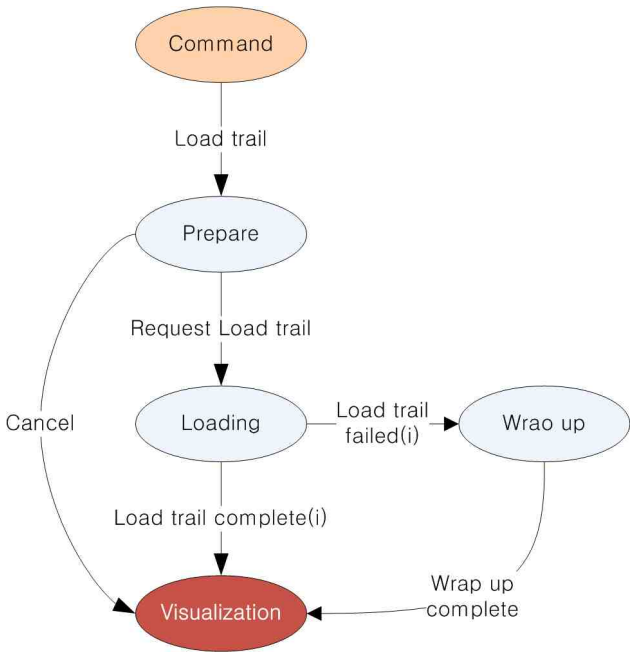


[그림 2-3] UNLOAD
state transition diagram

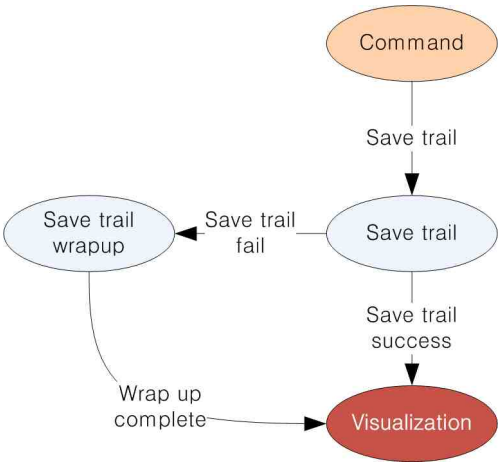
3) Load trail / Save trail

State	세부내용
LOAD TRAIL PREPARE	• 사용자 인터페이스를 통해서 GIVI가 읽을 trail 파일 (*.trail) 선택
LOAD TRAIL LOADING	• GIVI가 trail 파일을 읽어서 object DB / scene graph에 등록하는 중 • 동시에 GDM은 이후의 작업에 필요한 원본 데이터 파일을 읽음
LOAD TRAIL WRAP UP	• Trail을 읽는 데에 실패했을 때 데이터를 읽기 전의 상태로 돌아가기 위해 필요한 마무리 작업 수행
SAVE TRAIL	• Trail 저장 중 (GIVI)
SAVE TRAIL WRAP UP	• Trail 저장에 실패했을 때 이전의 상태로 돌아가기 위해 필요한 마무리 작업 수행

[표 2-4] LOAD / SAVE TRAIL 상태의 세부내용



[그림 2-4] LOAD TRAIL state transition diagram



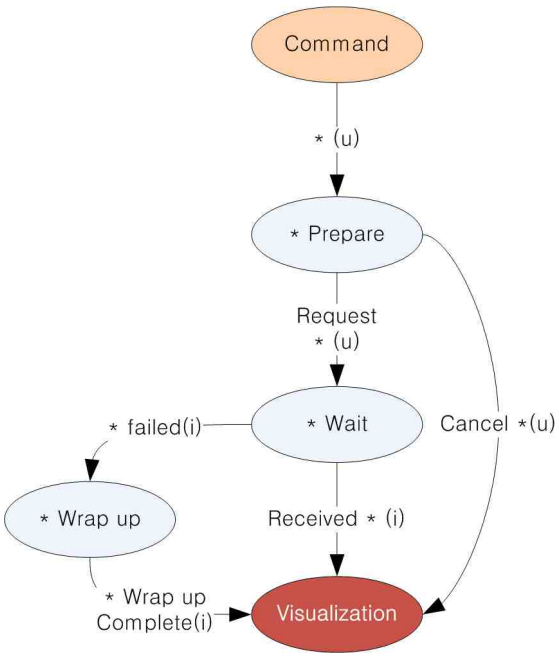
[그림 2-5] SAVE TRAIL state transition diagram

4) Visualization : Surface / Isosurface

Surface와 isosurface를 그리는 과정은 큰 차이가 없고 FSM 내에서의 state transition도 동일한 패턴을 갖고 있기 때문에 하나로 묶어서 설명한다. 하지만 실제 구현은 surface와 isosurface를 따로 분리해서 구현했다.

State	세부내용
(ISO)SURFACE PREPARE	• 새로 만들어지는 surface / isosurface의 특성을 사용자 인터페이스로 지정
(ISO)SURFACE WAIT	• GLORE에 query를 보내고 그 결과를 기다리는 상태
(ISO)SURFACE WRAP UP	• Visualization 작업 실패 • 미리 등록했던 object DB의 entry 삭제 등 query를 보내기 전의 상태로 되돌아가기 위한 작업 진행

[표 2-5] (ISO)SURFACE 상태의 세부내용



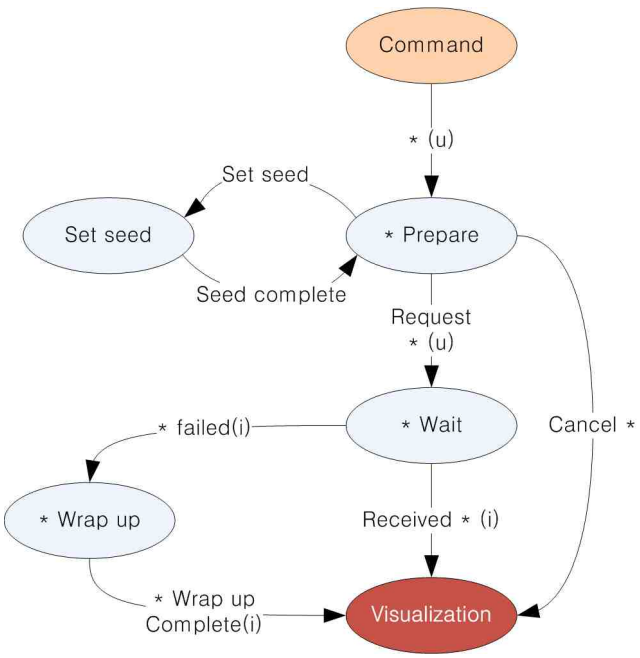
[그림 2-6] (ISO)SURFACE state transition diagram

5) Visualization : Streamline / Pathline

Surface/isosurface의 경우와 비슷하게 streamline과 pathline을 그리는 과정 사이에는 큰 차이가 없고 FSM 내에서의 state transition 역시 동일한 패턴을 갖고 있기 때문에 하나로 묶어서 설명한다. 하지만 실제 구현은 각각의 경우를 따로 분리해서 구현했다.

State	세부내용
STREAMLINE PREPARE (PATHLINE PREPARE)	• 새로 만들어지는 streamline(pathilne)의 특성을 사용자 인터페이스로 지정
STREAMLINE SET SEED (PATHLINE SET SEED)	• Streamline(pathline)의 시작점을 기술하기 위한 geometric widget의 위치/크기/방향을 지정하는 상태
STREAMLINE WAIT (PATHLINE WAIT)	• GLORE에 query를 보내고 그 결과를 기다리는 상태
STREAMLINE WRAP UP (PATHLINE WRAP UP)	• Visualization 실패 • 미리 등록했던 object DB의 entry 삭제 등 query를 보내기 전의 상태로 되돌아가기 위한 작업 진행

[표 2-6] STREAMLINE / PATHLINE 상태의 세부내용



[그림 2-7] STREAMLINE(PATHLINE)
state transition diagram

Streamline과 pathline은 최초 visualization을 위한 state에 더해서 이미 화면에 존재하는 객체의 세부내용을 수정하기 위한 state들이 존재한다. State transition의 큰 흐름은 [그림 2-7]과 거의 유사하기 때문에 별도의 transition diagram은 소개하지 않고, 각 상태의 세부내용만을 설명한다.

State	세부내용
STREAMLINE UPDATE READY	• Sub-dial menu에 streamline update에 필요한 메뉴 출력
STREAMLINE UPDATE ORIENTATION	• 현재 선택된 streamline을 만들었던 위젯(line widget, sphere widget)의 위치, 크기 등을 변경
STREAMLINE UPDATE WAIT	• 변경된 widget 내용으로 새로운 streamline을 GLORE에 요청
STREAMLINE UPDATE WRAP UP	• Streamline update가 실패했을 때 이전 상태로 되돌아가기 위한 작업 진행

[표 2-7] STREAMLINE UPDATE 상태의 세부내용

State	세부내용
PATHLINE UPDATE READY	• Sub-dial menu에 pathline update에 필요한 메뉴 출력
PATHLINE UPDATE ORIENTATION	• 현재 선택된 pathline을 만들었던 위젯(line widget, sphere widget)의 위치, 크기 등을 변경
PATHLINE UPDATE WAIT	• 변경된 widget 내용으로 새로운 pathline을 GLORE에 요청
PATHLINE UPDATE WRAP UP	• Pathline update가 실패했을 때 이전 상태로 되돌아가기 위한 작업 진행

[표 2-8] PATHLINE UPDATE 상태의 세부내용

6) Visualization : Slice

다른 visualization object와는 달리 slice는 여러 가지 부가기능을 제공해야 하는 특성이 있기 때문에 상대적으로 많은 시나리오가 존재하고, FSM state도 종류가 더 많다. 우선 slice 관련 시나리오는 아래와 같이 정리할 수 있다.

- 일반적인 visualization 대상으로서의 slice
- 이미 만들어진 slice의 내용 갱신 (위치, 크기 등)
- Slice에 존재하는 데이터의 저장
- Slice를 대상으로 하는 햅틱 렌더링

각 시나리오에 대응하는 FSM state는 [표 2-9] ~ [표 2-12]에 나눠서 정리했다.

State	세부내용
SLICE PREPARE	• 새로 만들어지는 slice의 특성을 사용자 인터페이스로 지정
SLICE ORIENTATION	• Geometric widget을 이용해서 slice의 크기, 방향을 지정
SLICE WAIT	• GLORE에 query를 보내고 그 결과를 기다리는 상태
SLICE WRAP UP	• Slice visualization 실패 • 미리 등록했던 object DB의 entry 삭제 등 slice query를 보내기 전의 상태로 되돌아가기 위한 작업 진행

[표 2-9] SLICE 상태의 세부내용

State	세부내용
SLICE UPDATE READY	• Sub-dial menu에 slice update에 필요한 메뉴 출력
SLICE UPDATE ORIENTATION	• 현재 선택된 slice의 위치, 크기 등을 변경
SLICE UPDATE WAIT	• 새로 지정한 위치/크기에 대응하는 새로운 slice를 GLORE에 요청
SLICE UPDATE WRAP UP	• Slice update가 실패했을 때 이전 상태로 되돌아가기 위한 작업 진행

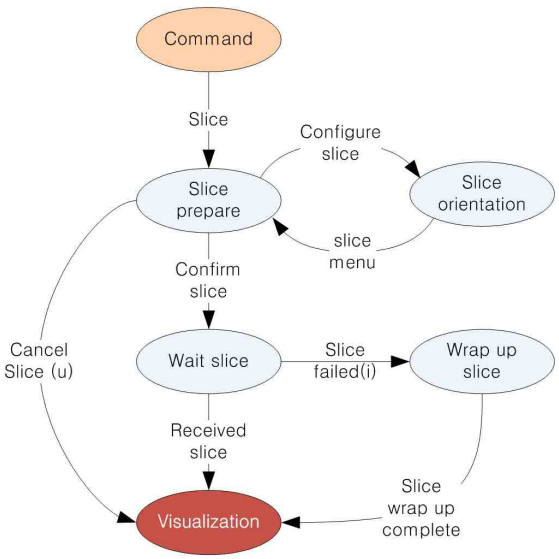
[표 2-10] SLICE UPDATE 상태의 세부내용

State	세부내용
SLICE HAPTIC COMMAND	• Slice haptic rendering 관련 명령이 시작하는 기본 대기상태
SLICE HAPTIC CALIBRATE	• Haptic device의 calibration 진행
SLICE HAPTIC RENDER	• 햅틱 렌더링 진행 • 햅틱 렌더링 영역은 현재 선택된 slice로 제한

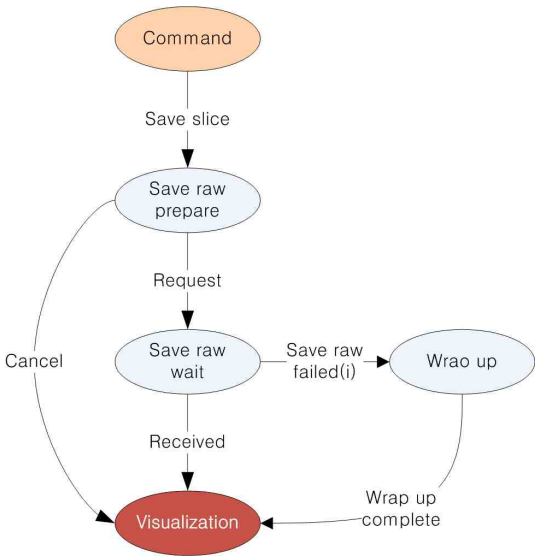
[표 2-11] SLICE HAPTIC 상태의 세부내용

State	세부내용
SLICE SAVE RAW PREPARE	• Slice에 포함된 데이터 중 실제 파일로 저장할 데이터 종류 선택 • 파일이 위치할 디렉터리 지정
SLICE SAVE RAW WAIT	• GLORE에 query를 보내고 그 결과를 기다리는 상태
SLICE SAVE RAW	• GLORE로부터 전달받은 slice data를 파일로 저장
SLICE SAVE RAW WRAP UP	• 데이터 저장 실패 • Query를 보내기 전의 상태로 되돌아가기 위한 작업 진행

[표 2-12] SLICE SAVE RAW 상태의 세부내용



[그림 2-8] SLICE state transition diagram



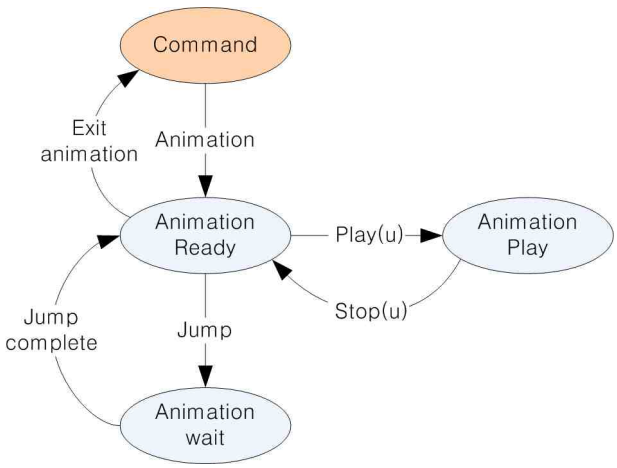
[그림 2-9] SLICE SAVE RAW state transition diagram

7) Animation / Timestep jump

애니메이션은 play, pause, stop, previous timestep, next timestep 등 다양한 기능을 제공하는 것처럼 보이지만 근본적으로는 '대기 → timestep 요청 → 데이터 수신 → 대기'의 순환과정만으로 거의 대부분의 기능을 구현할 수 있다. 따라서 애니메이션을 위한 FSM state는 [표 2-13]과 같이 단순하게 정리할 수 있다.

State	세부내용
ANIMATION READY	• 모든 animation 관련 명령이 시작하는 기본 대기상태
ANIMATION PLAY	• Animation play : stop 명령을 받기 전까지 쉬지 않고 next timestep request / receive / display 진행
ANIMATION WAIT	• 단일 timestep request를 위한 상태 • Animation ready 상태에서 특정 timestep을 요청한 후 그 결과를 기다리는 상태

[표 2-13] ANIMATION 상태의 세부내용

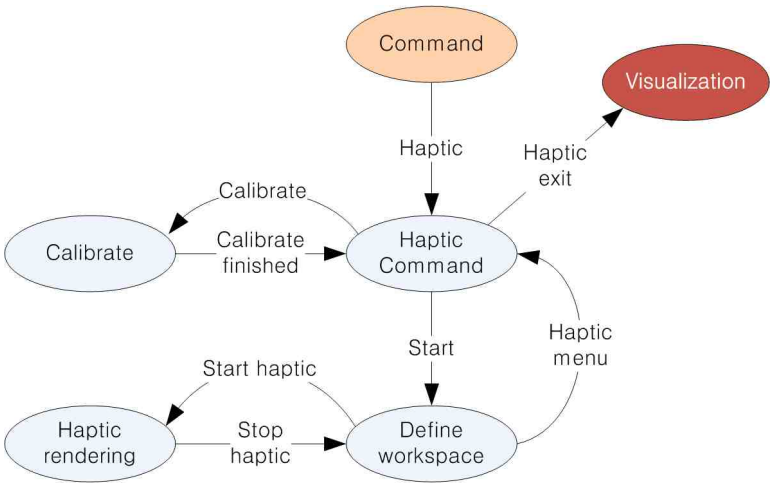


[그림 2-10] ANIMATION state transition diagram

8) Haptic

State	세부내용
HAPTIC COMMAND	• 모든 haptic rendering 관련 명령이 시작하는 기본 대기상태
HAPTIC CALIBRATE	• Haptic device의 calibration 진행
HAPTIC WORKSPACE	• Haptic rendering 영역 지정
HAPTIC RENDER	• 햅틱 렌더링 진행

[표 2-14] HAPTIC 상태의 세부내용

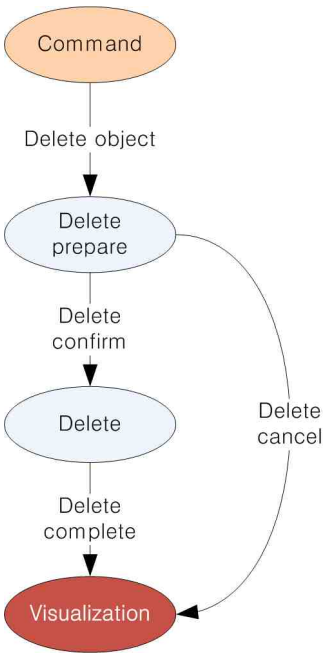


[그림 2-11] HAPTIC state transition diagram

9) Delete

State	세부내용
DELETE PREPARE	• 제거할 visualization object 선택
DELETE	• Object DB, scene graph, response DB에서 관련 정보 제거

[표 2-15] DELETE 상태의 세부내용



[그림 2-12] DELETE state transition diagram

나. FSM 엔진의 구현

1) State transition map

기본적으로 FSM은 state map과 state transition map으로 구성되어 있다. State map은 FSM을 구성하는 모든 state를 테이블 형태로 정리한 것이고, state transition map은 특정 event가 발생할 때 state의 변화를 기술하기 위해 사용한다. 우선 기본적인 데이터 타입을 [그림 2-13]과 같이 정의했다.

```
class FSM;

typedef  givi::Command  FSMEventData;

typedef bool (FSM::*StateFunction) (givi::FSMEventData *eventData);
```

[그림 2-13] givi::FSM::StateFunction의 type definition

실제 state map은 stl::map으로 구현하는데, 이에 필요한 table element와 '테이블 시작', '테이블 엔트리', '테이블 끝'을 지정하기 위한 매크로를 [그림 2-14]와 같이 정의했다.

```
struct StateStruct
{
   std::string mName;
   StateFunction mFunction;
};


#define BEGIN_STATE_MAP                          \
   public:                                       \
      const StateStruct *GetStateMap () {        \
         static const StateStruct mStateMap[] = {

#define STATE_MAP_ENTRY(name, entry)            \
    { (name), reinterpret_cast<StateFunction>(entry) },

#define END_STATE_MAP                            \
    { (std::string("")), reinterpret_cast<StateFunction>((void (givi::FSM::
*)(givi::FSMEventData *)) NULL) }               \
    };                                           \
    return &mStateMap[0]; }
```

[그림 2-14] State map을 구현하기 위한 구성요소

[그림 2-14]의 BEGIN_STATE_MAP, STATE_MAP_ENTRY, END_STATE_MAP은 결국 FSM 클래스의 GetStateMap이라는 member function을 구현하는 것이다. 그리고 그 함수 내에서 mStateMap이라는 테이블(state의 이름과 state function으로 구성된)을 정의하고, GetStateMap 함수는 그 테이블의 시작주소를 돌려준다. 이 매크로를 이용해서 FSM 클래스 내에서의 state map을 [그림 2-15]와 같이 구현했다.

```
class FSM
{
   private:

   중략

      BEGIN_STATE_MAP
         STATE_MAP_ENTRY(std::string("init"),     &FSM::ST_Init)
         STATE_MAP_ENTRY(std::string("command"), &FSM::ST_Command)

         ...

         STATE_MAP_ENTRY(std::string("Ignore"), &FSM::ST_IgnoreEvent)
      END_STATE_MAP

   후략

};
```

[그림 2-15] State map의 실제 구현 (헤더파일 내에서의 코딩)

앞에서 설명한 매크로를 적용하면 [그림 2-16]과 같이 GetStateMap이라는 함수가 만들어지는 것을 확인할 수 있다.

```
public:
   const StateStruct *GetStateMap () {
      static const StateStruct mStateMap[] = {
         { (std::string("init")),
            reinterpret_cast<StateFunction>(&FSM::ST_Init) },
         ...

         { (std::string("")),
            reinterpret_cast<StateFunction>(NULL)}
       };
      return &mStateMap[0];
    }
```

[그림 2-16] State map의 실제 구현

2) Event transition function

Event transition function은 FSM이 특정 상태에 있을 때 발생하는 event에 따라서 어떤 상태로 변화하는지를 정의한다. 우선 FSM의 각 상태를 테이블 인덱스로 사용하기 위해 [그림 2-17]과 같이 StateID를 정의한다.

```
typedef enum
{
   STATE_INIT = 0,
   STATE_COMMAND,
   STATE_VISUALIZATION,

   STATE_RESET_CONFIRM,
   STATE_RESET,
   STATE_EXIT_CONFIRM,
   STATE_EXIT,

   중략

} StateID;
```

[그림 2-17] FSM state의 정수 인덱스 정의

그리고 state map과 유사하게 transition map을 정의하기 위한 매크로를 [그림 2-18]과 같이 정의한다.

```
#define BEGIN_TRANSITION_MAP \
    bool ret;                \
    static const StateID TRANSITIONS[] = {

#define TRANSITION_MAP_ENTRY(entry)\
    (StateID) entry,

#define END_TRANSITION_MAP   \
    STATE_IGNORE_EVENT };    \
    mMutexFSM.acquire();     \
    ret = this->ProcessEvent(TRANSITIONS[mCurrentState], (data)); \
    mMutexFSM.release();     \
    return ret;
```

[그림 2-18] Transition map을 정의하기 위해 사용하는 매크로

[그림 2-18]에서 볼 수 있듯이 BEGIN_TRANSITION_MAP, TRANSITION_MAP_ENTRY, END_TRANSITION_MAP의 세 매크로는 TRANSITIONS라는 배열을 만들고, 이벤트 처리 함수(ProcessEvent)를 호출하는 루틴까지 구현한다. FSM의 현재 상태(mCurrentState)는 앞에서 설명한 StateID 중 하나의 값을 가지며, 그 값이 바로 TRANSITIONS의 인덱스로 사용된다. 이 매크로들을 이용해서 FSM에 이벤트가 발생했음을 알리는 EVT_ * 함수를 구현한다.

```
bool
givi::FSM::EVT_Menu (givi::FSMEventData *data)
{
   BEGIN_TRANSITION_MAP
      TRANSITION_MAP_ENTRY(STATE_IGNORE_EVENT)    // STATE_INIT
      TRANSITION_MAP_ENTRY(STATE_COMMAND)         // STATE_COMMAND

      중략

      TRANSITION_MAP_ENTRY(STATE_IGNORE_EVENT)    // STATE_DELETE
      TRANSITION_MAP_ENTRY(STATE_IGNORE_EVENT)    // STATE_IGNORE_EVENT
   END_TRANSITION_MAP
}
```

[그림 2-19] EVT_* 함수의 구현

[그림 2-19]는 사용자가 입력장치를 이용해서 메뉴 모드로 들어갈 때 발생하는 EVT_Menu 함수의 구현내용을 보여준다. [그림 2-1]의 state transition diagram을 보면 VISUALIZATION 상태에서 COMMAND 상태로 전환시키는 이벤트가 'Menu'라고 표기되어 있는데 바로 이 이벤트를 구현한 것이다. [그림 2-19]에서 사용한 매크로를 모두 전개하면 [그림 2-20]과 같은 내용이 된다. 이 코드를 자세히 살펴보면 TRANSITION은 결국 StateID의 배열에 불과함을 알 수 있다. 그리고 앞에서 설명한 것처럼 현재 상태(mCurrentState)를 인덱스로 사용해서 TRANSITION 배열을 참조하면(TRANSITION [mCurrentState]) 또 다른 StateID가 나오는데, 이것이 바로 '현재 상태에서 이 이벤트가 발생했을 때 바뀌는 다음 상태'를 의미한다.

```
bool
givi::FSM::EVT_Menu (givi::FSMEventData *data)
{
bool ret;
static const StateID TRANSITIONS[] =
   {
       (StateID) STATE_IGNORE_EVENT,

       중략

       (StateID) (STATE_IGNORE_EVENT),
                  STATE_IGNORE_EVENT
    };

   mMutexFSM.acquire();
   ret = this->ProcessEvent(TRANSITIONS[mCurrentState], (data));
   mMutexFSM.release();

   return ret;
}
```

[그림 2-20] EVT_Menu의 실제 구현내용

결국 TRANSITIONS는 현재 상태(mCurrentState)에서 EVT_Menu가 발생했을 때 다음 상태가 무엇인지 확인할 수 있도록 해주는 look up table의 역할을 하는 것이다. 그리고 mMutexFSM은 서로 다른 thread가 FSM의 EVT_ 계열 함수를 동시에 호출할 때 발생할 수 있는 충돌상황을 미리 방지하기 위해 사용한다.

한편 모든 EVT_* 함수는 TRANSITION 배열을 선언한 후 FSM::ProcessEvent를 호출해서 실제 FSM의 상태를 바꾼다. ProcessEvent 함수의 구현내용은 [그림 2-21]에서 확인할 수 있다. 우선 입력 파라미터 중 newState는 EVT_* 함수의 TRANSITIONS[mCurrentState] 값이다. 앞에서 설명했듯이 이 값은 '현재 상태에서 특정 이벤트가 발생했을 때 다음 상태'를 의미한다. 그렇기 때문에 newState가 STATE_IGNORE_EVENT이면 발생하지 말았어야 할 이벤트가 발생한 것을 의미하고, 이때에는 아무 작업도 수행하지 않은 상태로 false를 돌려준다. 그리고 FSM의 상태가 바뀌면, 그 상태에 있을 때 수행해야 하는 작업을 확인하는데, 이는 GetStateMap() 함수를 호출함으로써 이루어진다. 그 결과, stateMap은 다음 state에서 수행해야 할 작업을 기술한 함수를 가리키고, stateMap을 호출함으로써 state transition이 완료되는 것이다.

```
bool
givi::FSM::ProcessEvent (givi::StateID newState, givi::Command *data)
{
const StateStruct *stateMap;
givi::StateID prevState;
bool ret;

   if (newState == givi::STATE_IGNORE_EVENT)
    {
       return false;
    }

   prevState = mCurrentState;
   mCurrentState = newState;
   stateMap = this->GetStateMap();
   if (stateMap[mCurrentState].mFunction == NULL)
    {
       // For 'no need to call' state functions
       ret = true;
    }
   else
    {
       ret = (this->*stateMap[mCurrentState].mFunction)(data);
    }

   후략
```

[그림 2-21] ProcessEvent 함수의 구현

3. Thread 구성

GIVI는 다양한 모듈과 실시간으로 통신을 하는 특성이 있기 때문에 여러 개의 쓰레드를 동시에 제어해야 하는 어려움을 갖고 있다.

가. GIVI main thread

GIVI 메인 쓰레드는 간단히 말해서 VR Juggler 메인 어플리케이션 클래스를 구현한 것으로 이해할 수 있다. 이 쓰레드는 여타의 VR Juggler 어플리케이션과 마찬가지로 preFrame, latePreFrame, draw, postFrame으로 구성되어 있으며, scene graph의 관리, 사용자 입력 전달 등이 구현되어 있다.

1) preFrame

preFrame은 주로 화면에 내용을 출력하기 전의 준비과정을 포함한다.

```
void
GIVI::preFrame (void)
{
   1. 불필요한 geometry 삭제
   2. Particle animation 시작/중지
   3. 사용자 인터페이스(다이얼 메뉴, 위젯) 업데이트
   if (애니메이션 진행 중)
    {
       각 노드가 다음 타임스텝 출력을 위해 필요한 모든 geometry를 받았는지 확인
    }
   if (햅틱 렌더링을 위한 동기화)
    {
       1. 햅틱 프로브 위치를 슬레이브 노드에게 알려주기 위한 준비
       2. 햅틱 렌더링 용 데이터를 햅틱 서버에게 모두 전달했음을 슬레이브에게 알려주기 위한
          준비
       3. 햅틱 캘리브레이션이 완료됐음을 슬레이브에게 알려주기 위한 준비
    }
}
```

[그림 3-1] preFrame의 내용구성

① 불필요한 geometry 삭제

VR Juggler와 OpenSceneGraph를 같이 사용할 경우, scene graph의 내용 수정은 preFrame과 latePreFrame에서만 진행해야 한다. 따라서 어플리케이션 실행 도중 사용자가 특정 ge-

ometry를 지우는 명령을 실행할 때 실제로 geometry가 scene graph에서 지워지는 시점은 preFrame을 실행할 때이다. 그리고 object DB의 갱신도 이 시점에 이뤄진다.

② Particle animation 시작/중지
Streamline/pathline에 대한 particle animation의 시작/중지 역시 기본적으로 scene graph를 수정하는 것이기 때문에 preFrame 내에서 실행을 해야 한다.

③ 애니메이션
VR Juggler 어플리케이션이 공통으로 갖고 있는 preFrame의 중요한 역할 중 하나는 특정 노드(마스터)의 정보를 다른(디스플레이) 노드에 전달하기 위한 준비를 하는 것이다. GIVI에서는 애니메이션을 재생할 때 모든 노드(마스터 및 디스플레이)가 다음 timestep을 보여주기 위해 필요한 모든 geometry를 갖고 있는지의 여부를 확인하고, 그 결과를 디스플레이 노드에 알려주기 위한 준비를 preFrame에서 진행한다. 이 때 모든 노드가 출력에 필요한 geometry를 받았는지의 여부를 판단하는 작업과 그 판단결과를 디스플레이 노드에 알리는 역할은 마스터 노드가 수행한다.

④ 햅틱 렌더링
햅틱 렌더링 관련 정보 역시 마스터 노드가 각 슬레이브 노드에 알려준다. VR Juggler는 마스터 노드와 디스플레이 노드가 동일한 상태를 공유해야 하는지만 GIVI에서 햅틱 서버의 제어는 모두 마스터 노드가 전담하기 때문에 마스터 노드의 작업 결과를 디스플레이 노드에게 알려줄 수 있는 방법이 필요하다. GIVI의 preFrame에서 마스터 노드가 슬레이브 노드에 전달하는 정보는 다음과 같다.

- 햅틱 프로브 위치 : 디스플레이 노드가 VR 화면에 햅틱 프로브의 위치를 그릴 때 필요한 좌표 정보를 포함한다.
- 햅틱 서버로의 정상적인 데이터 전송 여부
- 햅틱 렌더링의 정상적인 시작 여부
- 햅틱 디바이스의 정상적인 캘리브레이션 여부
- 햅틱 프로브가 햅틱 데이터로부터 데이터 샘플링을 하고 있는지의 여부 : 햅틱 프로브의 색을 바꾸기 위해 사용한다.

2) latePreFrame

latePreFrame 역시 화면에 내용을 출력하기 위한 준비과정이지만 preFrame과 약간의 차이가 있다.

```
void
GIVI::latePreFrame (void)
{
   1. 컬러맵 동기화
   2. 필요한 경우 legend 내용 갱신
   if (애니메이션 타임스텝 동기화)
    {
       Scene graph의 front / back 교체
       Object DB의 front / back 교체
    }
   if (햅틱 렌더링 동기화)
    {
       햅틱 데이터 전송 여부 확인  → FSM 상태변경
       햅틱 렌더링 시작 여부 확인  → FSM 상태변경
       햅틱 캘리브레이션 여부 확인 → FSM 상태변경
       햅틱 프로브 위치 확인 → scene graph의 햅틱 프로브 위치 수정
       햅틱 렌더링 여부 확인 → scene graph의 햅틱 프로브 색 수정
    }
   5. 완드 상태 확인
   6. 이벤트 필터 실행
}
```

[그림 3-2] latePreFrame의 내용구성

① 컬러맵 동기화

컬러맵 동기화 역시 scene graph를 구성하는 노드의 DisplayList를 새로 만드는 작업이기 때문에 preFrame이나 latePreFrame에서 수행해야 한다.

② Legend 내용 갱신

Legend가 화면에 출력되어 있을 때에 한해 필요한 경우 그 내용을 갱신한다. Legend 또한 scene graph의 노드로 표현되기 때문에 색 변경, 내용 변경 등이 필요할 때에는 preFrame이나 latePreFrame에서 진행한다.

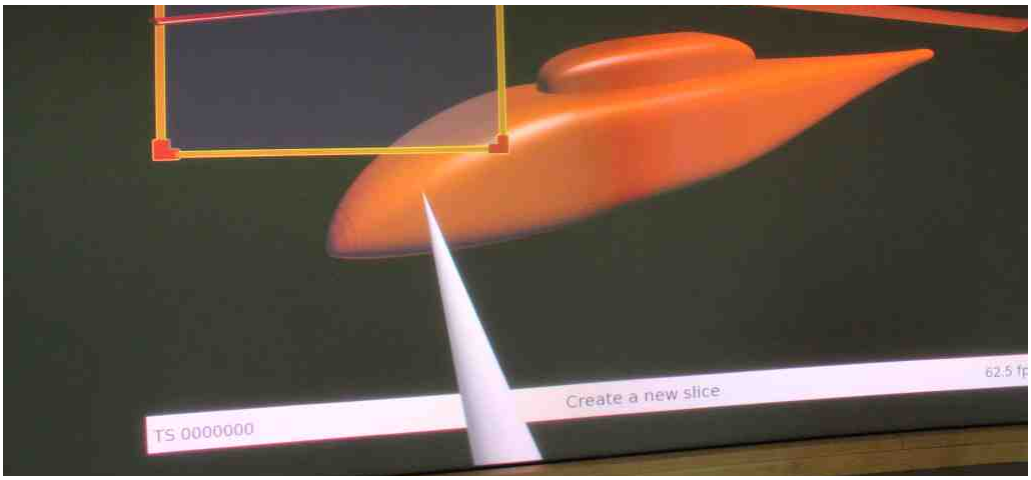
③ 애니메이션 타임스텝 동기화

preFrame에서 '모든 노드가 다음 타임스텝 출력을 위한 geometry를 다 받았음'을 마스터가 확인하고, 이를 모든 노드에 공지했을 경우에만 이 부분이 실행된다. 이 때 scene graph의

back buffer를 front buffer로 바꾸고, 이미 front buffer에 있는 노드를 삭제하고, object DB도 그에 맞춰서 내용을 갱신한다.

④ 햅틱 렌더링 동기화
햅틱 렌더링이 진행 중일 때 preFrame에서 마스터가 보낸 햅틱 렌더링 관련 정보를 여기서 활용한다.

⑤ 완드상태 확인
VR Juggler까 제공하는 자체 API를 이용해서 가상현실 입력장치의 위치와 방향, 버튼 상태 등을 가져온다. 이 정보를 이용해서 화면에 출력하는 포인터의 위치와 방향을 수정하고 그 결과를 scene graph에 반영한다.



[그림 3-3] 완드 포인터 (아래 하얀색 원뿔)

⑥ 이벤트 필터 실행
바로 직전 단계에서 갱신한 입력장치의 정보를 이용해서 인터페이스 조작, 객체 조작 등의 작업을 수행한다. 이벤트 필터의 세부 내용은 뒤에서 별도로 설명한다.

3) draw

```
void
GIVI::draw (void)
{
   glEnable(GL_DEPTH_TEST);
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|GL_STENCIL_BUFFER_BIT);

   vrj::osg::App::draw();
}
```

[그림 3-4] draw 함수의 구성

draw 함수는 scene graph의 darw traversal을 실행하는 부분이다. 여기서 실제로 그림을 그리는 작업은 모두 OpenSceneGraph가 담당하기 때문에 함수 자체는 그 내용이 간단하다. 한 가지 주의할 점은 사용자가 선택한 물체의 outline을 그릴 때 사용하는 osgFX::Outline의 정확한 작동을 보장하기 위해 스텐실 버퍼를 강제로 clear 하는 루틴이 포함되어야 한다.

4) postFrame

```
void
GIVI::postFrame (void)
{
   1. frame rate 계산
   if (애니메이션 진행 중)
    {
       다음 timestep 요청
    }
}
```

[그림 3-5] postFrame의 내용구성

postFrame은 draw가 끝난 후의 마무리 작업을 위해 수행한다. 여기서는 애니메이션이 재생 중일 때 다음 timestep을 요청하는 작업을 수행한다.

나. GLORE IO thread

1) TransactionDB / ResponseDB

GLORE IO 쓰레드는 이름그대로 GLORE와의 통신을 전담한다. GIVI의 관점에서 보면 GLORE IO 쓰레드가 네트워크를 통해서 데이터를 보내는 것으로 해석하면 되지만 실제로는 shared memory에 데이터를 쓰거나 데이터를 읽어오는 작업을 수행한다. 대신 독립적으로 실행되는 giviGipServer와 giviGipClient가 그 shared memory의 데이터를 GLORE로 보내고 GLORE로부터 전달받은 데이터를 shared memory 영역에 저장하는 역할을 담당한다.

용도	이름	내용
Message ID 저장	TransactionDB	• GLORE에게 보내는 메시지 저장
	ResponseDB	• GLORE로부터 전송받은 geometry 데이터의 저장
데이터 저장	SendQ	• GLORE에게 보내는 message ID 저장 • Message ID는 *DB의 엔트리에 대한 primary key로 사용되며 GIVI가 자체적으로 관리한다.
	ReceiveQ	• GLORE로부터 전송받은 데이터의 message ID 저장

[표 3-1] GIVI에서 사용하는 share memory 데이터베이스의 종류 및 특성

우선 message ID는 GLORE에 query를 보내고, GLORE가 해당 작업을 실행한 후의 결과를 받기 까지 그 작업을 식별하기 위한 고유번호로 사용된다.

① TransactionDB

TransactionDB는 GIVI가 보내는 메시지를 저장하기 위해 운영한다. TransactionDB에 저장하는 내용(transaction)은 GLORE에게 보내는 명령을 인코딩 한 octet stream이며, messageID로 식별할 수 있다. GLORE에 지시한 작업의 성공/실패 여부에 상관없이 답을 받은 후에는 해당 transaction을 DB에서 삭제한다.

② ResponseDB

GLORE로부터 geometry 등의 데이터를 전달받았을 때 이를 저장하기 위해 사용한다. 이 데이터베이스에 저장하는 엔트리는 모두 그 용량이 다르기 때문에 TransactionDB가 아닌, 별도의 메모리 공간에 데이터베이스를 운용한다. 주로 geometry가 저장되며 사용자가 해당 geometry를 삭제할 때 비로소 ResponseDB에서도 제거된다.

③ SendQ

giviGip에게 TransactionDB에 GLORE로 전송할 데이터가 있음을 알리기 위해 사용한다.

④ ReceiveQ

giviGip으로부터 ResponseDB에 데이터가 있는지 확인하기 위해 사용한다. ResponseDB에 새로운 데이터가 등록되어 있으면 해당 데이터의 messageID를 돌려준다.

2) Send / Receive

givi::GloreIO::Send는 서로 다른 쓰레드가 사용할 수도 있기 때문에 vpr::Mutex를 이용해서 단 한 쓰레드만 Send를 호출하도록 제어해야 한다.

```
void
givi::GloreIO::Send (gipMessage *message)
{
   mMutexSend.acquire();

   Transaction DB에 message ID 및 packet 정보 등록
   Send queue에 메시지 저장 (GIP 라이브러리가 자동 전송함)

   mMutexSend.release();
}
```

[그림 3-6] givi::GloreIO 클래스의 Send 함수

```
void
givi::GloreIO::Run (void)
{
   while (true)
    {
       GLORE로부터 전달받은 메시지 처리
       usleep(1000);    // 불필요한 CPU load 제거
    }
}
```

[그림 3-7] Glore IO 쓰레드 : 데이터 수신을 전담한다.

3) 디코드 대상 데이터의 특성

GLORE로부터 전달받은 메시지는 메시지 코드에 따라서 디코드 방법이 달라진다. 우선 [표 3-2]는 GIP 메시지의 식별자와 그에 따른 메시지 내용을 보여주고 있다.

GIP 메시지 코드	세부내용
CMD_RES_CON	• GIVI를 처음 실행할 때 GLORE와 성공적으로 연결됐을 때 받는다. • 이 메시지를 받은 후에 GIVI의 다른 기능을 사용할 수 있다.
CMD_RES_LOAD	• 데이터 로드가 정상적으로 끝났을 때에만 받는다. • 데이터 로드에 실패하면 CMD_RES_STATUS를 받는다.
CMD_RES_DATA	• 모든 종류의 geometry와 데이터가 성공적으로 만들어져서 GIVI에 전달됐을 때 받는다. - surface, isosurface, streamline, pathline, slice - 파일로의 저장을 위한 raw data
CMD_RES_STATUS	• GIVI가 GLORE에게 보낸 명령의 실행이 비정상적으로 종료됐을 경우에 받는다.

[표 3-2] 메시지 코드 별 데이터 내용

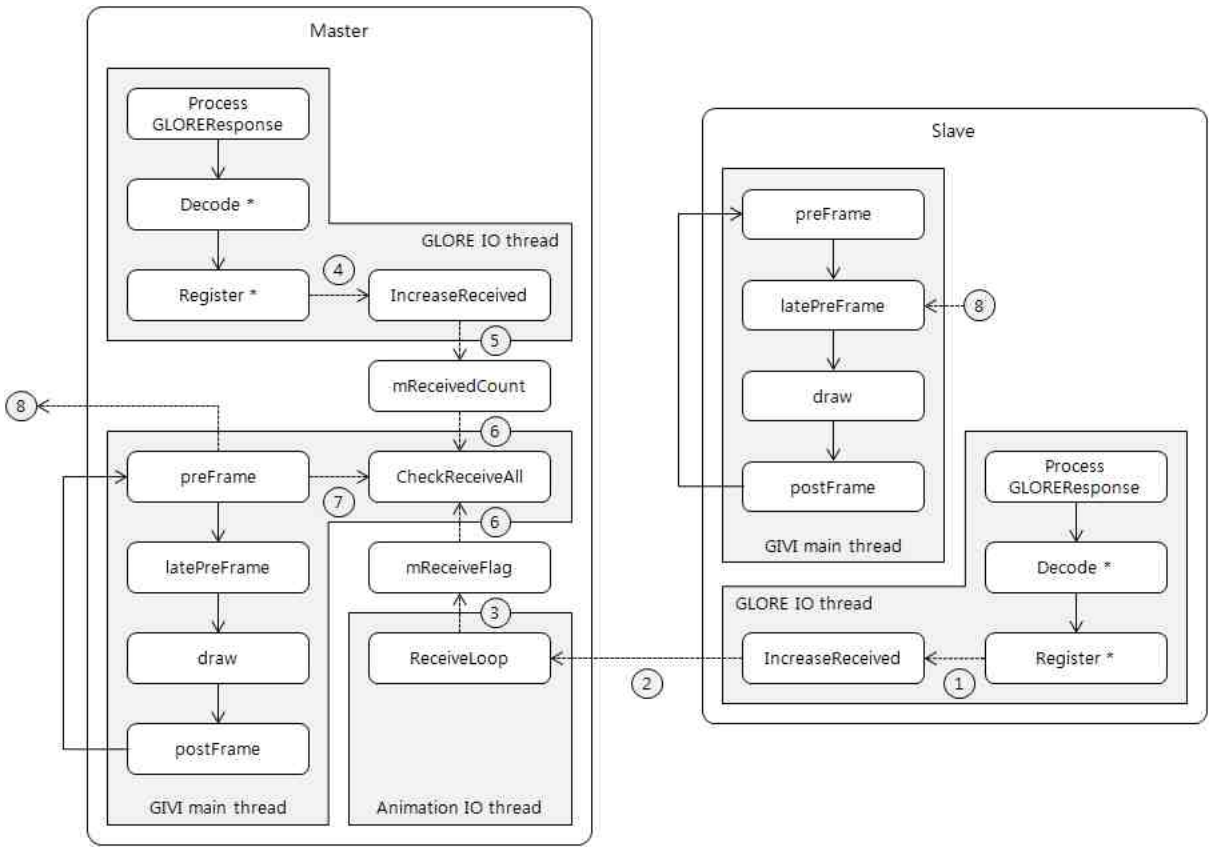
그리고 GIVI에 구현되어 있는, GIP 메시지의 종류에 따른 디코드 함수는 [표 3-3]에 정리했다.

GIP 메시지 코드	관련 member function
CMD_RES_CON	GloreIO::DecodeConnect (unsigned char *, unsigned int)
CMD_RES_LOAD	GloreIO::DecodeLoadResponse (unsigned char *, unsigned int) GloreIO::DecodeHistogram (unsigned char *, int)
CMD_RES_DATA	GloreIO::DecodeData (unsigned char *, unsigned int) GloreIO::DecodeSurface (gipCmdResData *) GloreIO::DecodeLines (gipCmdResData *) GloreIO::DecodeScalarSlice (gipCmdResData *) GloreIO::DecodeContourSlice (gipCmdResData *) GloreIO::DecodeGlyphSlice (gipCmdResData *) GloreIO::RegisterSurface (givi::Surface *, glvResPolyData *, scene::TARGET_BUFFER) GloreIO::RegisterIsosurface (givi::Isosurface *, glvResPolyData *, scene::TARGET_BUFFER target) GloreIO::RegisterStreamline (givi::Streamline*, glvResStreamPoints *, scene::TARGET_BUFFER) GloreIO::RegisterPathline (givi::Pathline *, glvResStreamPoints *, scene::TARGET_BUFFER) GloreIO::RegisterScalarSlice (givi::Slice *, glvResPolyData *, scene::TARGET_BUFFER) GloreIO::RegisterContourSlice (givi::Slice *, glvResPolyData *, scene::TARGET_BUFFER) GloreIO::RegisterContourSliceHaptic (givi::Slice *, gipCmdResData *, scene::TARGET_BUFFER) GloreIO::RegisterGlyphSlice (givi::Slice *, glvResPointSample *, scene::TARGET_BUFFER) GloreIO::RegisterGlyphSliceHaptic (givi::Slice *, gipCmdResData *, scene::TARGET_BUFFER)
CMD_RES_STATUS	GloreIO::DecodeStatus (unsigned char *, unsigned int)

[표 3-3] GIP 메시지 별 디코드 함수

다. Animation sync thread

Animation sync thread는 animation이 진행 중일 때 마스터 노드와 슬레이브 노드 사이의 동기를 맞추기 위해 사용한다. 여기서의 동기는 frame rate가 아니라 time-varying 데이터 내에서 모든 노드가 동일한 타임스텝 데이터를 보여주는 것을 보장하는 것을 의미한다. 이 작업은 기본적으로 마스터 노드의 '현재 타임스텝' 정보를 디스플레이 노드들에게 알려주고, 디스플레이 노드가 이에 맞춰서 동일한 타임스텝 정보를 화면에 출력하는 방식으로 이루어진다.



[그림 3-8] Animation sync thread의 작동방식

여기서는 이미 FSM이 animation play 상태에 있음을 전제하고, GLORE로부터 geometry를 받는 순간부터 설명한다.

① 다음 timestep의 출력을 위해 GLORE로부터 전송받은 geometry를 scene graph(back buffer)에 등록한 후 received counter를 1 증가시킨다.
② 현재 화면에 출력되어 있는 모든 geometric object에 대해서 다음 timestep에 해당하는

geometry를 받으면 이를 GIVI master에게 알려준다(multicast).

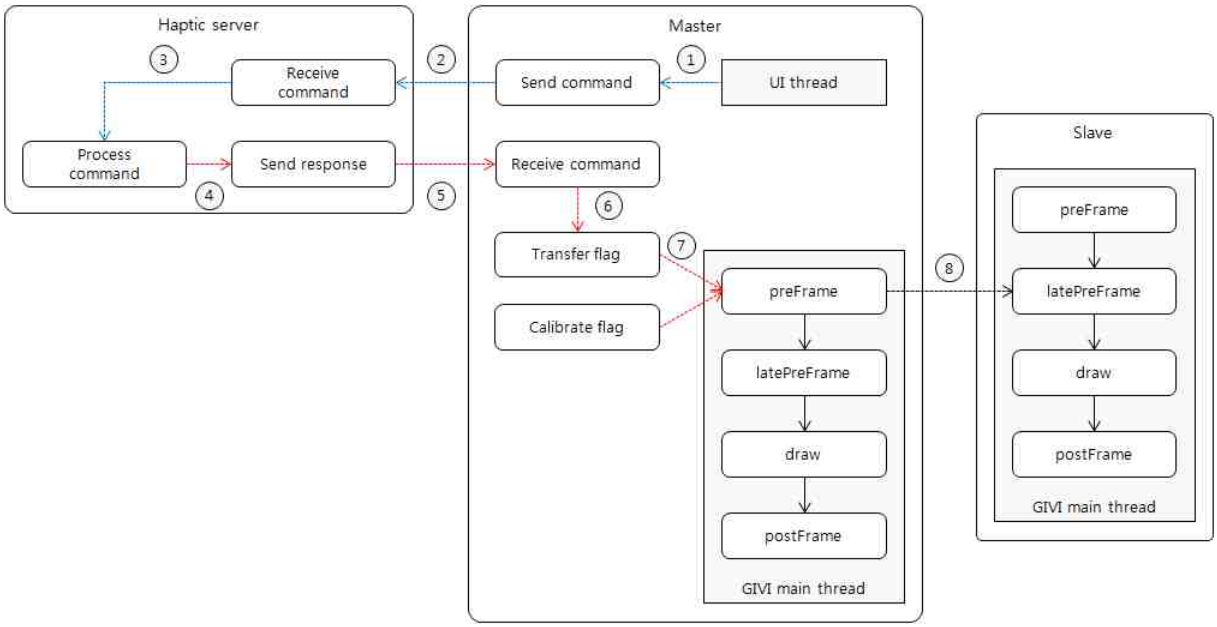
③ GIVI master는 필요한 geometry를 전달받은 디스플레이 노드(slave) 정보를 유지한다.

④ GIVI master 역시 필요한 geometry를 전송받을 때마다 received counter를 증가시킨다.

⑤~⑦ GIVI master 자신과 모든 디스플레이 필요한 geometry를 받았는지의 여부 확인은 GIVI::preFrame에서 진행한다.

⑧ 모든 노드들이 필요한 geometry를 받으면 preFrame 내에서 GIVI master가 디스플레이 노드에게 공지한다.

라. Haptic IO thread



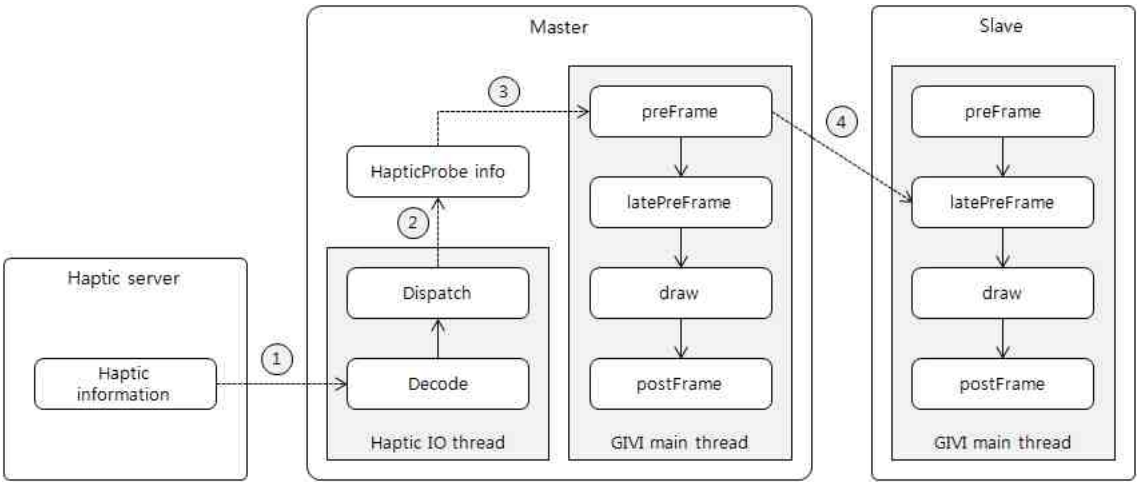
[그림 3-9] Haptic IO thread의 작동방식 (명령 수행 및 결과 확인)

① Haptic IO는 기본적으로 사용자 인터페이스에서 사용자가 햅틱 관련 명령을 선택하는 것으로부터 시작한다.

②~⑤ GIVI(마스터)로부터 명령을 전달받은 햅틱 서버는 명령에 대응하는 작업을 수행하고 그 결과를 GIVI(마스터)에게 알려준다.

⑥~⑦ 햅틱 서버의 작업결과를 전달받은 GIVI(마스터)는 그 결과를 디스플레이 노드에게 알릴 준비를 한다. 이 과정은 GIVI::preFrame에서 진행된다.

⑧ 작업 결과는 모든 노드의 GIVI::latePreFrame에서 확인할 수 있으며, 그에 따라서 FSM 변경, scene graph의 햅틱 프로브 위치/색 변경 등 필요한 작업을 수행할 수 있다.



[그림 3-10] Haptic IO thread의 작동방식 (햅틱 프로브 위치)

① 햅틱 렌더링을 하는 동안 햅틱 프로브의 위치/방향을 GIVI(마스터)에게 알려준다.

② GIVI(마스터)는 전달받은 햅틱 프로브 정보를 디코드하고, 그 결과를 HapticProbeInfo에 저장한다.

③ 동시에 GIVI::preFrame에서 햅틱 프로브 정보를 디스플레이 노드에게 알리기 위한 준비 작업도 진행한다.

④ 디스플레이 노드는 마스터로부터 전달받은 햅틱 프로브 정보에 따라서 화면의 프로브 위치와 색을 변경한다(scene graph 포함).

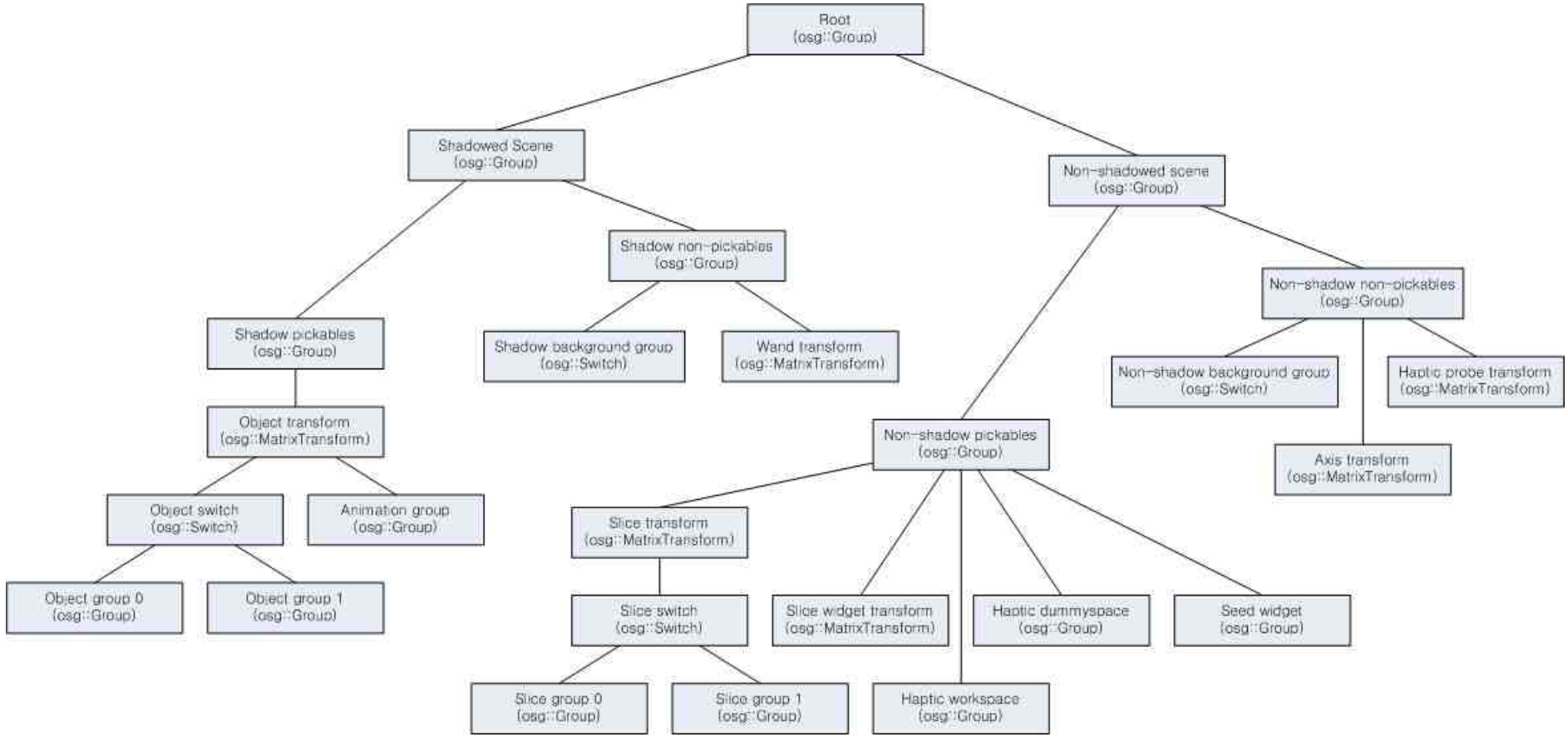
4. Scene graph의 구성

GIVI의 scene graph는 visualization object뿐만 아니라 사용자 인터페이스, 부가정보 등을 모두 관리해야 하기 때문에 비교적 복잡한 구성을 갖고 있다. [표 4-1]은 GIVI의 scene graph에서 leaf node를 제외한 나머지 노드의 목록을 보여주고 있다.

이름	용도
Root	• 전체 scene graph의 최상위 노드
Shadowed scene	• 그림자 계산이 필요한 모든 geometry의 총괄관리
Shadow pickables	•
Object transform	• Slice를 제외한 모든 visualization object의 관리
Object switch	• Animation play를 진행할 때 object group 0과 object group 1을 교대로 화면에 출력하기 위해 운용
Object group 0	• Group 0과 group 1 중 어떤 것이 'front buffer'와 'back buffer'의 역할을 하는지는 매 frame마다 달라진다.
Object group 1	
Animation group	• Streamline / pathline의 particle animation을 위해 개별 particle의 transformation matrix 관리
Shadow non-pickables	• 그림자 계산이 필요하지만 실제로 사용자가 직접 조작하지 않는 geometry의 총괄 관리
Shadow background group	• 그림자 계산이 필요한 배경(checker board 등)의 관리
Wand transform	• 가상현실 입력 장치(flystick, wand 등)의 VR 공간 내에서의 위치와 방향을 알려주기 위한 geometry 관리
Non-shadowed scene	• 그림자 계산이 필요 없는 모든 geometry의 총괄관리
Non shadow pickables	• 그림자 계산은 필요 없지만 사용자의 조작이 필요한 geometry의 총괄관리
Slice transform	• Slice data를 관리 • Visualization object와 달리 slice widget transform과 동일하게 적용
Slice switch	• Slice group 0과 Slice group 1 중 하나를 선택해서 보여주기 위해 사용
Slice group 0	• Group 0과 group 1 중 어떤 것이 'front buffer'와 'back buffer'의 역할을 하는지는 매 frame마다 달라진다.
Slice group 1	

Slice widget transform	• 화면에 보이는 slice 및 새로 만드는 slice에 대응하는 planar widget 관리
Haptic workspace	• Haptic workspace를 지정하기 위한 cube widget 관리
Haptic dummy space	• 실제 haptic rendering을 진행할 때 haptic workspace 대신 화면에 보이는 cube 관리
Seed widget	• Streamline과 pathline을 만들 때 seed point의 배열 방식을 지정하기 위한 widget (line widget, sphere widget) 관리
Non-shadowed non-pickables	• 그림자 계산과 사용자의 직접 조작을 모두 지원하지 않는 노드의 총괄관리
Non-shadow background group	• Shadow 계산이 필요 없는 배경(주로 cube map) 관리
Axis transform	• 물체의 orientation을 알려주기 위한 coordinate axis 관리
Haptic probe transform	• Haptic rendering을 할 때 화면에 나타나는 probe 관리

[표 4-1] Scene graph의 노드 구성



[그림 4-1] Scene graph 구성

5. Visualization object의 관리

가. Visualization object

GIVI 내에서 유지하는 visualization object는 기본적으로 givi::Object로 표현되며, 종류(isosurface, streamline, pathline 등)에 따라서 약간의 정보가 추가된다. GIVI가 사용하는 visualization object는 [표 5-1]에 정의되어 있다.

Visualization object		내용
Surface		멀티블록 데이터의 블록 외형
Isosurface		3차원 공간에서 동일한 값을 갖는 영역을 면(polygonal surface)으로 표현
Streamline		단일 timestep 내에서 벡터 데이터를 공간적으로 적분한 결과
Pathline		두 개 이상의 timestep 내에서 벡터 데이터를 시/공간으로 적분한 결과
Slice	Scalar value distribution	3차원 공간에 정의된 2차원 평면에서의 스칼라 값 분포
	Contour line	3차원 공간에 정의된 2차원 평면에서의 contour line
	Glyph	3차원 공간에 정의된 2차원 평면에서의 glyph

[표 5-1] GIVI에서 사용하는 visualization object의 종류

각 visualization object의 세부정보는 두 종류로 구분할 수 있다. 한 종류는 모든 visualization object가 공통으로 사용하는 정보이며, 다른 한 종류는 각 visualization object에 고유하게 정의되는 정보다. GIVI에서는 givi::Object라는 클래스가 공통정보를 관리하도록 했고, 각 visualization object에 특화된 정보는 givi::Object로부터 파생된 클래스(derived class)에 포함시켰다. 우선 [표 5-2]는 givi::Object에 포함되어 있는 기본정보를 보여준다. 여기에는 몇 가지 주의해야 할 사항이 있다. 우선 givi::Surface, givi::Isosurface, givi::Streamline, givi::Pathline의 실제 기하정보를 저장하는 mGeodeFront와 mGeodeBack은 derived class에 포함될 수도 있지만 실제로 GIVI를 구현하는 과정에서 편의를 위해 givi::Object 클래스에 포함시켰다. mOutlineFront, mOutlineBack, mGroupFront, mGroupBack 모두 동일한 이유로 givi::Object에 포함되어 있다. 그리고 mLineWidget, mSphereWidget, mPlaneWidget 역시 프로그램 구현상의 편의를 위해 givi::Object에 포함시켰다.

데이터 타입	변수명	세부내용
object::TYPE	mType	Visualization object의 형태
std::string	mName	Visualization object의 이름
unsigned short	mObjectID	Object DB에서 사용하는 primary key
unsigned short	mMessageID	GLORE에 query를 보낼 때 발급받은 message ID
unsigned short	mMessageOLD	Animation 실행 시 사용
bool	mReceived	GLORE로부터 geometry를 받았는지의 여부 확인
bool	mHidden	화면에의 출력 여부 확인
bool	mRegisteredSG	Scenegraph에의 등록 여부 확인
int	mDataSetID	GLORE에서 발급한 데이터 셋 고유번호
int	mTimestep	Visualization object가 만들어진 timestep
std::vector<int>	mElement	Visualization object를 만들 때 사용한 element ID 목록
int	mGeometryBy	기하학적 형태를 만들기 위해 사용한 데이터 값
int	mGeometryBySub	기하학적 형태를 만들기 위해 사용한 데이터 값 (벡터 데이터의 일부를 추출할 때 사용)
int	mColorBy	객체의 색을 입히기 위해 사용한 데이터 값
int	mColorBySub	객체의 색을 입히기 위해 사용한 데이터 값 (벡터 데이터의 일부를 추출할 때 사용)
int	mHapticBy	객체에 대해 햅틱 렌더링을 할 때 사용할 데이터 값
int	mHapticBySub	객체에 대해 햅틱 렌더링을 할 때 사용할 데이터 값 (벡터 데이터의 일부를 추출할 때 사용)
osg::Geode	mGeodeFront	기하학 정보를 담고 있는 OSG node (front buffer 용)
osg::Geode	mGeodeBack	기하학 정보를 담고 있는 OSG node (back buffer 용, animation 실행 시 사용)
osgFX::Outline	mOutlineFront	각 visualization object의 윤곽선
osgFX::Outline	mOutlineBack	각 visualization object의 윤곽선
givi::SliceGroup	mGroupFront	Slice 정보
givi::SliceGroup	mGroupBack	Slice 정보
vrGeoWidgetLine	mLineWidget	*Line을 만들 때 사용한 widget 정보
vrGeoWidgetSphere	mSphereWidget	*Line을 만들 때 사용한 widget 정보

데이터 타입	변수명	세부내용
vrGeoWidgetPlane	mPlaneWidget	Slice를 만들 때 사용한 widget 정보

[표 5-2] Visualization object (givi::Object) 의 기본정보

Surface는 별도의 추가정보를 필요로 하지 않기 때문에 givi::Object로부터 파생된 givi::Surface 클래스는 존재하지만 추가 member instance는 존재하지 않는다. 반면 givi::Isosurface는 3차원 볼륨 데이터로부터 isosurface를 추출하기 위한 isovalue를 기술하기 위한 데이터(mThreshod, [표 5-3])가 추가된다.

데이터 타입	변수명	세부내용
double	mThreshold	Isosurface를 추출할 때 사용한 isovalue

[표 5-3] givi::Isosurface를 위한 추가정보

givi::Streamline부터는 추가정보의 수가 상대적으로 많아진다.

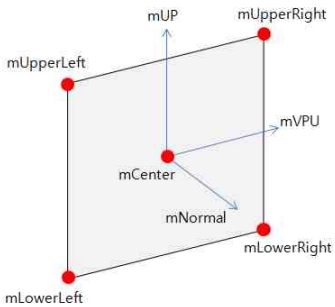
데이터 타입	변수명	세부내용
object::SEED_TYPE	mSeedType	Seed를 정의하는 widget의 형태
unsigned int	mSeedCount	Seed point 개수
std::vector<object::Seed>	mSeed	Seed point 목록
float[3]	mCenter	(Sphere widget) 중심
float	mRadius	(Sphere widget) 반지름
float[3]	mStart	(Line widget) 시작점
float[3]	mEnd	(Line widget) 끝점
object::INTEGRATION	mIntegration	적분 방향 (forward, backward, both)

[표 5-4] givi::Streamline을 위한 추가정보

givi::Pathline의 추가정보는 givi::Streamline과 거의 동일하지만 (integration의) 시작시간과 종료시간이 포함된다는 점에서 차이가 있다.

데이터 타입	변수명	세부내용
int	mStartTime	Integration 시작 시간
int	mEndTime	Integration 끝 시간
object::SEED_TYPE	mSeedType	Seed를 정의하는 widget의 형태
unsigned int	mSeedCount	Seed point 개수
std::vector<object::Seed>	mSeed	Seed point 목록
float[3]	mCenter	(Sphere widget) 중심
float	mRadius	(Sphere widget) 반지름
float[3]	mStart	(Line widget) 시작점
float[3]	mEnd	(Line widget) 끝점
object::INTEGRATION	mIntegration	적분 방향 (forward, backward, both)

[표 5-5] givi::Pathline을 위한 추가정보

데이터 타입	변수명	세부내용
float	mWidth	슬라이스의 너비
float	mHeight	슬라이스의 높이
osg::Vec3	mCenter	슬라이스의 기하학적 중점
osg::Vec3	mNormal	
osg::Vec3	mUP	
osg::Vec3	mVUP	
float[3]	mUpperLeft	
float[3]	mUpperRight	
float[3]	mLowerLeft	
float[3]	mLowerRight	
object::SLICE_TYPE	mSliceType	Slice의 형태 (scalar value distribution, contour, glyph)
int	mContourCount	(Contour slice) slice에 그려지는 contour line의 개수
int	mGlyphCountX	(Glyph slice) slice에 나타나는 glyph의 VPU 방향
int	mGlyphCountY	(Glyph slice) slice에 나타나는 glyph의 UP 방향 개수

[표 5-6] givi::Slice를 위한 추가정보

나. Object DB

Object DB는 화면에 보이는 모든 visualization object의 정보를 관리한다. 일반적으로 생각할 수 있는 데이터베이스의 매우 단순화된 버전으로 볼 수 있으며, 앞에서 설명한 givi::Object의 등록, 삭제, 내용 수정 등 데이터 관리를 위한 기본적인 API를 제공한다. 한 가지 주의할 사항은 Update 작업은 애니메이션을 위해 Front와 Back이라는 접미어를 붙여서 따로 진행할 수 있도록 했다는 점이다. 그리고 front buffer에 등록된 object와 back buffer에 등록된 object의 front/back 플래그를 한 번에 바꿀 수 있도록 했다는 특징도 존재한다. 또, object DB는 서로 독립적으로 실행되는 쓰레드가 동시에 접근하는 경우가 많기 때문에 mutex를 이용해서 concurrency control을 적용해야 했다.

작업	API
등록	Insert (unsigned short objectID, givi::Surface &surface) Insert (unsigned short objectID, givi::Isosurface &isosurface) Insert (unsigned short objectID, givi::Streamline &streamline) Insert (unsigned short objectID, givi::Pathline &pathline) Insert (unsigned short objectID, givi::Slice &slice)
검색	Retrieve (unsigned short objectID); Retrieve (std::string &name);
갱신	UpdateFront (unsigned short objectID, osg::ref_ptr<osg::Geode> geode) UpdateFront (unsigned short objectID, osg::ref_ptr<osgFX::Outline> outline) UpdateFront (unsigned short objectID, osg::ref_ptr<givi::SliceGroup> slice) UpdateBack (unsigned short objectID, osg::ref_ptr<osg::Geode> geode) UpdateBack (unsigned short objectID, osg::ref_ptr<osgFX::Outline> outline) UpdateBack (unsigned short objectID, osg::ref_ptr<givi::SliceGroup> slice)
삭제	Delete (unsigned short objectID)

[표 5-7] Object DB의 주요 인터페이스

6. Event filter

가. 이벤트 필터의 사용

이벤트 필터는 가상현실 입력 장치의 상태(버튼 누름, 조이스틱 움직임)를 어플리케이션에 전달하는 역할을 담당한다. 이벤트 필터는 GIVI의 메인 쓰레드가 사용하는데, 그 중에서도 latePreFrame에서 모든 처리가 진행된다.

```
void
GIVI::latePreFrame (void)
{

   // 중략

gmtl::Matrix44f matrix;                 // wand matrix
WandState state;

   mWand->UpdateStatus(matrix);         // update wand
   gScene->SetWandTransform(matrix);    // Wand를 화면에 출력
   mEventFilter->Dispatch(*mWand);
}
```

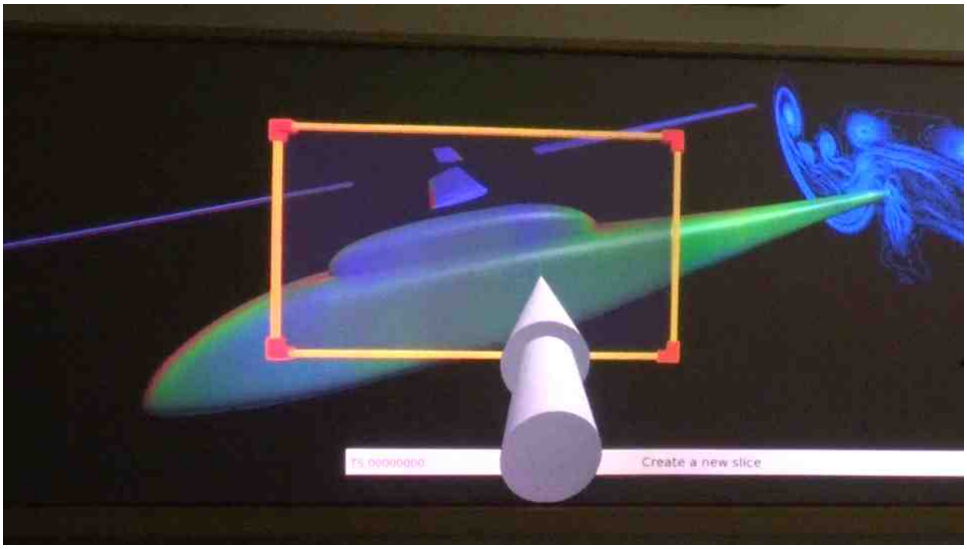
[그림 6-1] Event filter의 사용

- mWand->UpdateStatus(matrix)

완드의 상태를 GIVI의 내부 자료구조에 저장한다. 이 정보에는 wand 자체의 transformation matrix, world coordinate 상에서의 위치, 방향, 조이스틱의 움직임, 버튼 상태가 포함된다.

- gScene->SetWandTransform(matrix)

바로 직전에 갱신한 완드 상태정보에 맞춰서 완드 프로브를 화면에 출력한다.



[그림 6-2] 완드 프로브가 화면에 출력된 모습 (가운데 하얀색 화살표)

- mEventFilter->Dispatch(*mWand)

최신 완드 상태를 이용해서 이벤트(FSM의 이벤트가 아닌, 입력장치에서 발생하는 이벤트를 의미한다)를 어플리케이션의 다른 모듈에게 전달한다. 이벤트 필터 내에서 수행하는 일은 FSM의 상태에 따라서 달라진다.

이벤트 핸들러	역할	FSM state
DoNothing	사용자 입력 방지	INIT RESET EXIT LOAD_DATA_LOADING LOAD_TRAIL_LOADING UNLOAD SAVE_TRAIL SURFACE_WAIT ISOSURFACE_WAIT STREAMLINE_WAIT STREAMLINE_UPDATE_WAIT PATHLINE_WAIT PATHLINE_UPDATE_WAIT SLICE_WAIT SLICE_UPDATE_WAIT SLICE_SAVE_RAW_WAIT SLICE_SAVE_RAW HAPTIC_CALIBRATE DELETE
UserInterface	메뉴, 다이얼로그 박스 등 사용자 인터페이스 처리 모듈로 이벤트 전달	COMMAND RESET_CONFIRM EXIT_CONFIRM LOAD_DATA_PREPARE LOAD_DATA_WRAP_UP

이벤트 핸들러	역할	FSM state
		LOAD_TRAIL_PREPARE LOAD_TRAIL_WRAP_UP UNLOAD_CONFIRM UNLOAD_WRAP_UP SAVE_TRAIL_WRAP_UP SURFACE_PREPARE SURFACE_WRAP_UP ISOSURFACE_PREPARE ISOSURFACE_WRAP_UP STREAMLINE_PREPARE STREAMLINE_WRAP_UP STREAMLINE_UPDATE_READY STREAMLINE_UPDATE_WRAP_UP PATHLINE_PREPARE PATHLINE_WRAP_UP PATHLINE_UPDATE_READY PATHLINE_UPDATE_WRAP_UP SLICE_PREPARE SLICE_WRAP_UP SLICE_UPDATE_READY SLICE_UPDATE_WRAP_UP SLICE_HAPTIC_COMMAND SLICE_HAPTIC_CALIBRATE SLICE_HAPTIC_RENDER SLICE_SAVE_RAW_WRAP_UP ANIMATION_READY ANIMATION_PLAY ANIMATION_WAIT ANIMATION_WRAP_UP HAPTIC_COMMAND SET_TIMESTEP SET_TIMESTEP_WRAP_UP DELETE_PREPARE
ObjectManipulation	화면에 있는 물체의 이동/회전	VISUALIZATION
SeedManipulation	Streamline/pathline을 만들 때 seed point의 시작점을 지정하는 위젯의 위치변경/회전	STREAMLINE_SET_SEED PATHLINE_SET_SEED
SeedUpdateManipulation	이미 만들어진 streamline/pathline의 seed widget 변경	STREAMLINE_UPDATE_ORIENTATION PATHLINE_UPDATE_ORIENTATION
SliceManipulation	Slice를 만들 때 크기와 위치를 지정하는 위젯의 위치변경/회전	SLICE_ORIENTATION
SliceUpdateManipulation	이미 만들어진 slice의 위젯 위치 변경/회전	SLICE_UPDATE_ORIENTATION
HapticWorkspace	햅틱 워크스페이스를 지정하는 위젯의 위치/크기 변경	HAPTIC_WORKSPACE
HapticRender	햅틱 렌더링 도중 사용자 입력 처리	HAPTIC_RENDER

[표 6-1] FSM 상태별로 호출되는 이벤트 핸들러 및 역할

이벤트 핸들러는 매 프레임마다 호출되는 특성을 갖고 있다. 완드의 상태정보 역시 매순간 달라지기 때문에 직전 프레임의 완드 상태를 기억해야 물체의 이동, 회전 등의 작업을 정확하게 구현할 수 있다. 이를 위해서 모든 이벤트 핸들러의 마지막 부분에는 완드의 상태를 기억하는 루틴이 포함되어 있다.

```
void
givi::EventFilter::HandleUserInterface (givi::Wand &wand)
{
WandState stateWand;
givi::StateID stateFSM;

   wand.CopyStatus(stateWand);

   중략

   // otherwise, pass wand event to widget library
   mUI->updateDeviceStatus(stateWand);

   mPreviousPosition  = stateWand.mPosition;
   mPreviousDirection = stateWand.mDirection;
}
```

[그림 6-3] 이벤트 핸들러 내에서 현재의 완드 상태를 저장하는 루틴 (빨간색)

나. 이벤트 필터의 구현

이벤트 필터는 특별한 경우를 제외하고는 거의 비슷한 성격의 루틴들이 구현되어 있다.

① 입력장치의 상태 분석

입력장치의 버튼이 눌렸는지의 여부, 조이스틱의 움직임에 따라서 물체의 이동, 크기 조절 등 적절한 작업을 진행한다. 이 부분은 특정 상태(FSM state)에서 사용 가능한 입력조합을 모두 처리할 수 있어야 하므로 대략 [그림 6-4]와 같은 형태로 소스가 구성되어 있다.

```
void
givi::EventFilter::HandleSeedManipulation (givi::Wand &wand)
{
   // Default seed manipulation mode : translate
   wand.CopyStatus(state);
   if (wand.Check(wand::MENU_BUTTON) == gadget::Digital::TOGGLE_ON)
    {
      // Menu 버튼을 눌렀을 때의 처리
    }
   else if (wand.Check(wand::SELECT_BUTTON) == gadget::Digital::TOGGLE_ON)
    {
      // Select 버튼을 눌렀을 때의 처리 : 물체의 선택 등에 적용
    }
   else if (wand.Check(wand::SELECT_BUTTON) == gadget::Digital::ON)
    {
      // Select 버튼을 계속 누르고 있을 때의 처리 : 물체의 이동 등에 적용
    }
   else
    {
      // 조이스틱 등 위의 if 문에 해당하지 않는 입력 이벤트 처리
    }

   // 현재의 완드 위치 및 방향 기억

exitFilter:
   mPreviousPosition  = state.mPosition;
   mPreviousDirection = state.mDirection;
```

[그림 6-4] 이벤트 필터가 사용하는 사용자 입력 처리 루틴의 구조

② 대상물체의 조작(크기 조절, 이동, 회전 등)

대상물체의 조작 루틴은 geometry나 위젯의 크기/방향/위치 등을 바꿀 때 사용한다. 예를 들어서 seed widget(streamline/pathline의 seed point 배치를 위한 위젯)의 이동은 [그림 6-5]와 같은 형태로 구현되어 있다.

```
void
givi::EventFilter::TranslateSeedWidget (gmtl::Point3f &position,
                                        gmtl::Vec3f &direction)
{
   if (mFocusedComponent->getName() == "vrGeoWidgetProxyComponent")
    {
       // seed widget 전체를 움직이기 위한 루틴 구현
    }
   else if (mFocusedComponent->getName() == "vrGeoWidgetComponent")
    {
       // seed widget 중 특정 구성요소(점, 선 등)를 움직이기 위한 루틴 구현
    }
}
```

[그림 6-5] 이벤트 필터 내에서 seed widget을 움직이는 루틴의 구현

[그림 6-5]의 소스에서 seed widget을 움직이려면 직전 프레임과 현재 프레임에서의 위치 및 방향 정보를 이용해서 움직이는 거리와 방향을 결정한다. 그렇기 때문에 [그림 6-5]의 소스에서 현재 프레임의 위치를 저장하는 것이다.

③ 완드와 대상물체의 충돌 감지

완드 프로브와 visualization geometry, 또는 완드 프로브와 위젯의 충돌 감지 루틴은 OSG에서 일반적으로 사용하는 intersection check 루틴을 그대로 사용한다.

7. 결론

지금까지 가상현실 가시화 시스템의 설계 및 구현내용에 대해 설명했다. 가상현실 가시화 시스템은 이 한 편의 보고서만으로 모든 내용을 설명할 수는 없을 만큼 그 양이 방대하기 때문에 앞으로 지속적인 내용보강을 진행할 예정이다. 향후 가시화 시스템의 개발 방향은 크게 세 가지로 나누어 생각할 수 있다.

① 데이터 분석 기능의 강화

현재 가시화 시스템에 구현되어 있는 내용은 isosurface 추출, streamline, pathline 등 기본적인 visualization 알고리즘과 햅틱 렌더링 및 부가기능이 주를 이룬다. 하지만 이 시스템이 본격적인 데이터 분석도구의 역할을 수행하려면 causality relationship, flow structure 분석 및 시각적 표현 등 응용분야에 더욱 특화된 데이터 분석 및 visualization 기능을 추가해야 한다.

② 스마트 디바이스 지원 및 인터페이스 보강

완드나 flystick과 같은 가상현실 입력장치는 정확도의 한계가 있기 때문에 세밀한 작업이 어렵다는 단점이 있다. 이를 보완하기 위해 패드와 같은 스마트 디바이스로 사용 편의성을 높이고, 동시에 정밀한 제어가 필요한 작업을 원활하게 수행할 수 있는 방법을 제공할 계획이다.

③ 불필요한 통신 제거를 통한 효율성 향상

현재 GIVI와 giviGip*은 서로 독립적인 어플리케이션으로 실행되고 있기 때문에 shared memory와 같이 프로세스 간 통신 방법을 이용해서 데이터를 전송하고 있다. 하지만 이는 전체적인 성능을 떨어뜨리는 요인이 되기 때문에 두 어플리케이션을 통합해서 효율을 향상시키는 방식으로의 수정이 필요하다.