ISBN 978-89-6211-604-5

GIVI: GLOVE 통합 가시화 환경의 설계와 구현 (GLORE, GIP)

허영주 (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

GLOVE(GLObal Virtual Environment for research)는 고성능 컴퓨팅 환경에서 데이터를 효과적으로 분석하고 가시화하여 공유할 수 있게 하는 프레임워크로, 가상현실 인터페이스를 이용, 고성능 컴퓨터나 고해상도 디스플레이 장치에 대한 전문지식이 부족한 사용자라도 손쉽게 자신의 데이터를 가시화하고 분석할 수 있게 해주는 프레임워크다. GLOVE는 대용량 데이터를 지원하기 위해 클러스터 환경에서의 병렬 컴퓨팅 기능을 지원하며, 이런 이유로 인터페이스 부분인 GIVI와 렌더링 엔진 부분인 GLORE가 서로 분리돼 있다. GIVI와 GLORE간의 통신을 담당하는 것이 바로 GIP이며, GIVI와 GLORE는 GIP을 이용해서 request와 가시화 결과를 서로 주고받게 된다.

이 때, GLOVE의 인터페이스 부분인 GIVI(GLOVE Integrated Visualization Interface)는 GLORE와의 연동을 위해 다소 복잡한 구조를 가지게 되며, 끊임없이 사용자 인터페이스를 받아들여서 GLORE에 데이터를 요청하고 그 결과 데이터를 받아서 디스플레이하는 루틴을 반복하게 된다.

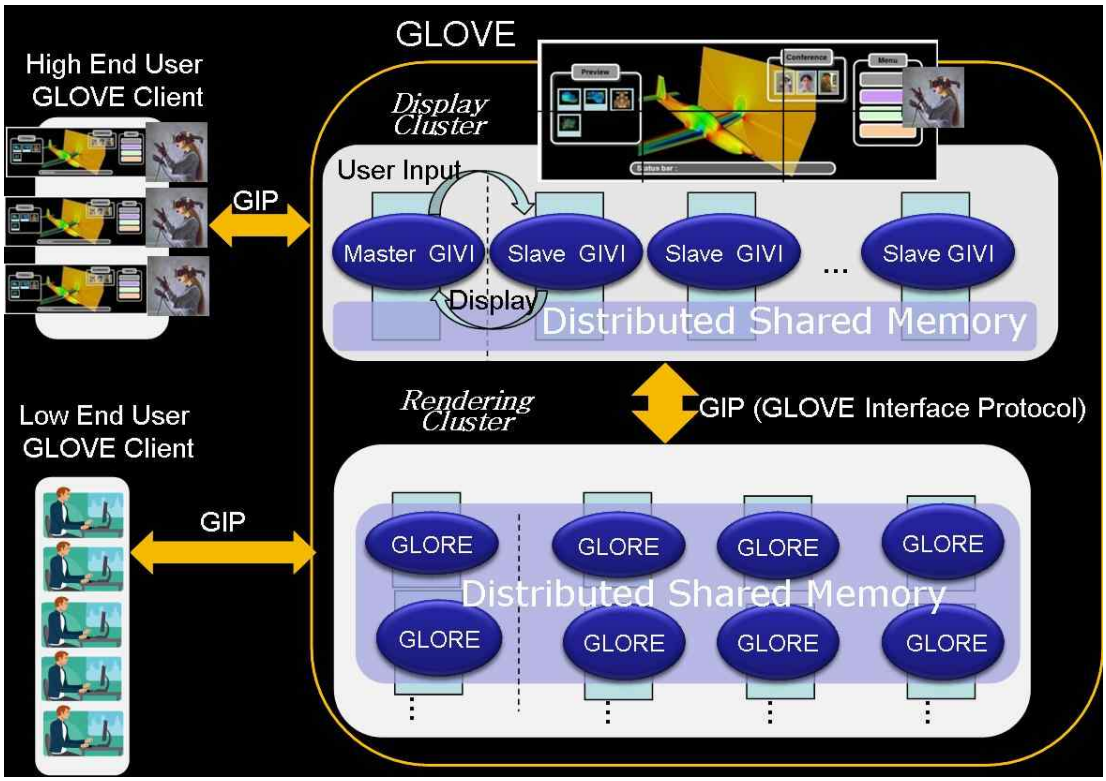
본 문서에서는 GIVI의 이런 기능에 대한 설계 및 구현 내용에 대해 설명하기로 한다. 이를 위해, GLOVE 전체의 구조에 대해 설명하고 GIVI의 동작 방식을 설명한 뒤, 데이터를 로딩하고 GLORE에 데이터를 요청하는 각각의 루틴에 대해 상세하게 설명할 것이다. 또, Probe Plane 기능과 시간에 따라 변하는 데이터에 대한 애니메이션 기능의 동작 방식과 구현 내용에 대해 설명하고, GIVI에서 전체 동작을 위해 사용하는 컨텍스트 데이터에 대해서도 설명할 것이다.

2. GLOVE (GLObal Virtual Environment for collaborative research)

GLOVE는 고성능 컴퓨팅 환경에서 대용량 시뮬레이션 데이터를 효과적으로 분석하고 가시화할 수 있게 해주는 통합 가시화 프레임워크로, 현재는 로터 동역학 시뮬레이션 데이터 분석을 타겟으로 개발되고 있다. GLOVE는 다음과 같은 특징을 갖는다.

- VR 기반 통합 환경 내에서 데이터의 정성적, 정량적 분석
- 병렬 처리를 통한 대용량 데이터의 효율성 분석 및 가시화
- 단일 모니터에서 가상현실 장비에 이르기까지의 다양한 형태의 가시화 장비 지원
- 대용량 데이터 가시화 서비스 제공

GLOVE는 이런 기본 프레임워크 뒤에 각 응용분야에 특화된 사용자 인터페이스를 구현함으로써 해당분야 연구자들이 접근하기 용이한 통합 환경을 제공한다.



[그림 2-1] GLOVE 구조

GLOVE의 기본 구조는 [그림 2-1]과 같으며, 그림에서 볼 수 있듯이 통합 사용자 인터페이스인 GIVI(GLOVE Integrated Visualization Interface)와 데이터 가공 및 렌더링을 담당하는 GLORE(GLOVE Rendering Engine), 그리고 이 둘 간의 통신을 담당하는 GIP(GLOVE Interface Protocol), 이 세 부분으로 나눠볼 수 있다.

GIVI는 가상현실 입출력장치를 총괄하며, 사용자로부터 입력을 받아서 GLORE에 전달하고, GLORE의 실행 결과를 전달받아서 화면에 출력하는 역할을 담당한다. GLORE는 대용량 데이터의 가공 및 렌더링을 위한, GIVI의 서브시스템이라 할 수 있겠다. GLOVE에서 GLORE와 GIVI는 서로 독립적으로 운영되는 클러스터 환경에서 실행되는데, 이런 구조는 GLOVE의 인터페이스 부분과 핵심 렌더링 기능을 물리적으로 분리할 수 있기 때문에 충분한 그래픽 처리 능력을 갖추지 못한 시스템에서 GIVI만 실행한 상태로 원격지의 고성능 컴퓨터에서 실행되는 GLORE를 이용해서 대용량 데이터를 처리할 수 있게 한다. 이렇게 시스템이 분산되는 경우에는 데이터의 공유를 위한 효율적인 전송이 반드시 필요한데, 이때 GLORE와 GIVI간의 통신을 담당하는 부분이 바로 GIP이다.

이제 이 각각의 구성요소에 대해 알아보기로 한다.

가. GIVI

GLOVE의 통합 사용자 인터페이스를 GIVI라고 하며, 주기능으로는 사용자의 요청을 받아들여서 GLORE에 데이터를 요청하고, GLORE가 추출한 데이터를 전송받아서 렌더링한 뒤 출력하는 기능을 들 수 있다.

또, GIVI는 개발자가 응용 분야에 적합한 인터페이스를 개발하는데 필요한 기반환경을 제공한다. 현재 GLOVE 시스템은 로터 동역학 분야를 주 응용분야로 설정하고 개발이 진행중이다. 그러나 GIVI는 기본적으로 해당 응용 분야에 따라 자유롭게 구성해서 사용할 수 있게 설계돼 있으며, 이에 따라 다양한 분야에서 각 분야에 특화된 사용자 인터페이스를 제공할 수 있다.

GIVI는 과학 데이터 가시화에 필요한 다양한 툴과 위젯(widget)을 제공함으로써 사용자 인터랙션을 받아들일 수 있는 GUI를 구성할 수

있게 한다. 이를 위해 과학 데이터 탐색에 많이 사용되는 기본적인 위젯을 3차원 형태로 제공하고 있으며, GNUPlot을 이용해서 통계 데이터에 대한 그래프를 그리고, 그 그래프를 가상현실 환경에서 렌더링된 데이터와 나란히 비교할 수 있게 하는 인터페이스도 제공한다.

GIVI는 개발상 자유도가 높은 VR Juggler를 이용, 다양한 형태의 디바이스를 지원하며, VR Juggler 애플리케이션의 동작 매커니즘에 따라 마스터 프로세스와 마스터와 동일하게 동작하는 슬레이브 프로세스로 구성된다. 여러 대의 클러스터 컴퓨터가 디스플레이를 담당하는 고해상도의 대형 가시화 시스템에서는 서로 다른 GIVI 슬레이브 프로세스가 각 디스플레이의 렌더링 프로세스를 담당하게 된다.

나. GLORE

GLORE는 GIVI의 사용자 요청을 받아 렌더링을 수행하고 그 결과를 GIVI에 전달하는 렌더링 엔진이다. GLORE는 마스터 노드가 사용자 요청을 받아 여러 대의 슬레이브 노드에서 렌더링 작업을 수행한 뒤, 그 결과를 GIVI로 전송하게 된다.

GLORE는 대용량 데이터의 처리를 위해 다수의 노드를 스케줄링하여 병렬처리를 수행하는데, 데이터 사이즈에 따라 단일 노드 내에서도 병렬 처리를 수행할 수도 있다. 이런 기능을 수행하는 GLORE는 응용 데이터에 독립적이며, GIVI에서의 사용자 요청을 전달하는 프로토콜인 GIP을 이용해서 요청을 받고, 다시 GIP을 이용해서 처리한 결과를 GIVI에 전달한다.

다. GIP

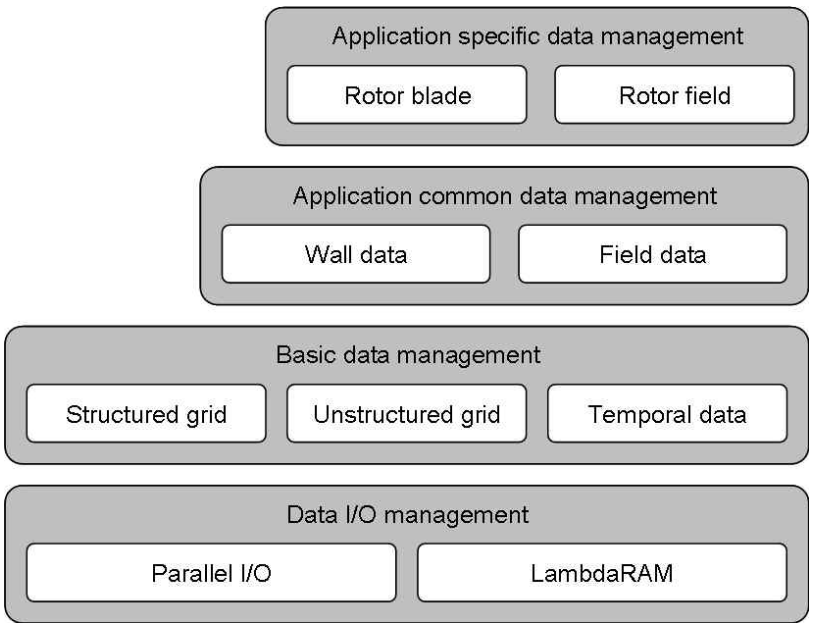
GIP은 GLOVE Interface Protocol의 약자로, 서로 원거리에 위치해 있는 GIVI와 GLORE간의 통신을 위해 설계/구현된 TCP/IP 기반의 프로토콜이다. 이 프로토콜은 request와 response로 나뉘는데, request는 응용 맞춤형 메뉴에 따르는 것이 아니라 streamline, isosurface, pathline 등의 가시화 형태에 근거해서 종류가 달라지게 된다. 따라서, 응용 데이터가 바뀐다 하더라도 GIP 자체는 변경될 필요가 없다. GIP은 GIVI와 GLORE간 통신뿐만 아니라 다수의 디스플레이 노드를 필요로 하는 고해상도 디스플레이 환경에서도 마스터 GIVI와 슬레이브 GIVI간의 통신에 사용된다.

기본적으로 GLORE는 마스터 GIVI와 통신을 하게 되며, 마스터 GIVI가 다른 슬레이브 GIVI들에게 데이터를 전달하게 된다.

라. 계층적 데이터 구조

GLOVE는 데이터를 계층적 구조로 관리함으로써, 범용 가시화를 위한 프레임워크 위해 각 애플리케이션의 특성을 반영하는 맞춤형 사용자 인터페이스를 제공할 수 있다. [그림 2-2]는 이런 GLOVE의 데이터 계층구조를 보여준다.

데이터 계층구조의 가장 아랫부분에 위치한 Data I/O management 계층에서는 데이터의 병렬 입출력 및 공유 캐시(cache)등의 메커니즘을 관리한다. Basic data management 계층에서는 가시화를 위한 structured grid, unstructured grid 같은 기본적인 메시 구조와 다중 블록(multi-block) 데이터 및 시간에 따라 변하는(time-varying) 데이터를 위한 자료구조를 관리하게 된다. Application common data management 계층에서는 GLOVE가 지원하는 응용 분야에서 공통으로 활용하는 자료 구조를 다루게 되는데, CFD 분야에 있어서는 헬기의 표면을 나타내는 Wall 데이터와 기류의 흐름을 나타내는 Field 데이터를 이 계층에서 적용할 수 있으며, 이 계층부터 응용에 따라 지원하는 자료구조라 달라지게 된다.



[그림 2-2] GLOVE의 데이터 계층 구조

Application specific data management 계층에서는 특정 응용분야에서 사용하는 자료 구조를 관리하게 되는데, 현재 GIVI가 지원하는 로터 동역학 분야에서는 Rotor blade 데이터와 Rotor field 데이터를 이 계층에서 정의할 수 있다.

마. 데이터에 따른 기능 정의

현재 GIVI는 로터 동역학 분야를 지원하도록 정의돼 있으며, 이 분야에서 정의된 데이터는 Rotor 데이터와 Field 데이터로 볼 수 있다. 여기에서는 각 데이터에 따라 구분된 GIVI의 기능 정의에 대해 설명하기로 한다.

Rotor 데이터에 대한 가시화 기능은 다음과 다.

- Pressure Distribution: Rotor 데이터 표면의 압력 분포를 보여준다.
- Graph (데이터에 대한 정량적 분석)
  - Pressure: 주어진 span position에서 회전각에 따른 압력의 변화를 보여준다.
  - Pitch angle variation: 특정 span position과 0~360도의 회전각에 대한 pitch angle의 변화를 보여준다.
  - Sectional force: 주어진 span position과 회전각에 대한 sectional normal force, sectional chord force, sectional span force 정보를 보여준다.
  - Sectional moment: span position과 회전각에 대한 sectional x-moment, sectional y-moment, sectional z-moment 그래프를 커브와 디스크 형태로 제공한다.

Field 데이터에 대한 가시화 요소는 크게 isosurface와 probe plane을 통한 컷팅 플레인, 그리고 streamline이 있다. probe plane은 다시 contour와 scalar에 대한 컷팅 플레인으로 나눌 수 있다.

- Pressure: 데이터 내의 압력 분포를 나타내는 값으로 isosurface와 probe plane을 통한 컷팅 플레인을 분석할 수 있다.
- Vorticity: 사용자가 지정한 vorticity 값에 따른 isosurface를 볼 수

있으며, 컷팅 플레인을 분석할 수도 있다.

- Velocity: 사용자가 지정한 velocity 값을 이용해 스트림라인을 가시화할 수 있다.
- Q-Criteria: vortex 영역을 Q-Criteria 기법으로 가시화한 것으로, 역시 isosurface와 컷팅 플레인 분석이 가능하다.

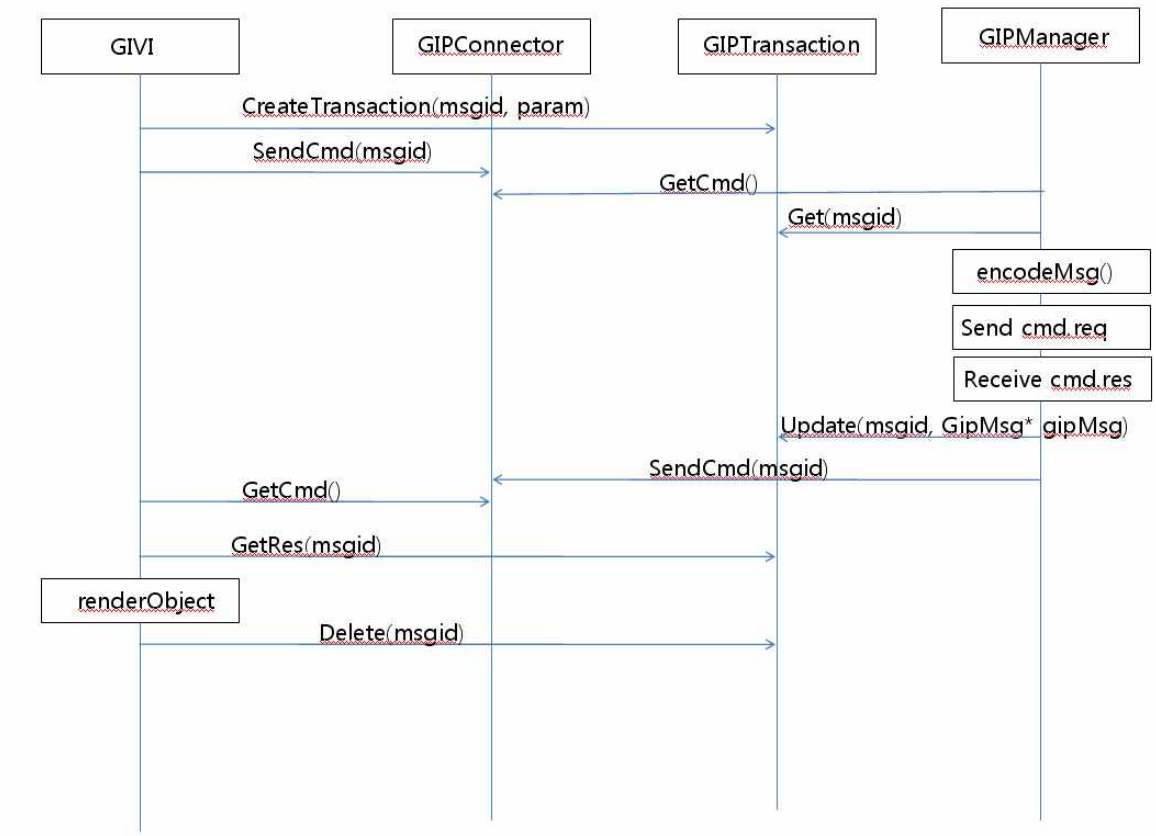
3. GIVI의 동작

3장에서는 GIVI의 기본적인 동작에 대해 설명하고, GIVI에서 데이터를 관리하는 방식에 대해 설명한 뒤,

가. 전체

GIVI는 앞서 설명했듯이 GLORE에 사용자 요청을 전달하고 그 결과를 GLORE로부터 받아서 디스플레이 장치에 전달하는 역할을 수행한다. 이 때, 모든 데이터 전송은 메시지를 기반으로 하며, GIP의 Request와 Response 메시지를 이용해서 서로 통신을 수행하게 된다. [그림 3-1]은 GIVI의 전체적인 동작을 보여주는 Sequence 다이어그램을 보여주고 있다.

[그림 3-1]에서 알 수 있듯이, GIVI는 메시지 아이디와 관련 파라미터를 보냄으로써 GIPTransaction과의 트랜잭션을 생선한다. 일단 트



[그림 3-1] GIVI의 동작을 나타내는 Sequence 다이어그램

랜잭션이 생성되고 나면 GIPConnector에 SendCmd 메시지를 전달하게 되고, SendCmd를 받은 GIPConnector는 GIPManager로부터 GetCmd를 통해 커맨드를 받을 때까지 계속 대기하게 된다. GIPManager는 메시지 아이디를 생성해서 GIPTransaction에 보낸 다음, encodeMsg() 함수로 메시지를 인코딩한 뒤, request 명령을 보내게 된다(Send cmd.req).

이렇게 전송된 request가 GLORE로부터 처리되고 나면 response를 받게 되며(Receive cmd.res), 이렇게 받은 response를 GIPConnector에 전송하게 된다(SendCmd(msgid)). GIVI는 msgid로 response를 구분, 결과값을 받아서(GetRes(msgid)) 오브젝트를 렌더링한 뒤, 해당 메시지를 삭제한다(Delete(msgid)).

전체적인 동작 방식은 이와 같으며, 사용자의 모든 인터랙션에 대해 이와같은 과정을 통해 GLORE에 request를 보내고 GLORE로부터 response를 받아서 결과를 디스플레이하게 된다.

실질적으로 GIVI에서는 처음에 트랜잭션을 생성하고 난 뒤, GLORE로부터 response가 올 때까지 계속 대기모드로 루프를 돌다가 GLORE로부터 response가 올 때마다 데이터를 처리하는 형태로 코드가 구성된다. 다음의 [소스 3-1]에서는 GIVI에서 메시지를 처리하는 방식을 볼 수 있다.

```
void GIVI::ProcessMsg(void)
{
  unsigned short msgId = 0;
  int          nRet;
  char         errMsg[80];

  nRet = gipResQ->Receive(&msgId, 0);

  if ( nRet >= 0)
  {
    sprintf(errMsg, "msgId(%u)₩n", msgId);
    givLog::Write(errMsg, 0);
    cout << "msgId : " << msgId << endl;
```

```
gipTrans *trans;
trans = gipResDB->GetTransaction(&msgId);

if (trans == NULL)
{
    printf("GetTransaction(%u) fail₩n", msgId);
    sprintf(errMsg, "GetTransaction(%u) fail₩n", msgId);
    glvLog::Write(errMsg, 0);
}
else
{
     gipMessage *gipMsg;

     switch(trans->trans[0])
     {
     case CMD_RES :
          gipMsg = ProcessCmdRes(trans->trans, trans->trLen);
          gContextData->SetButton2On();
          break;
     default :
          printf("Unknown gip primitive₩n");
          gipMsg = NULL;
          break;
          }

     nRet = gipTrDB->Delete(&msgId, sizeof(unsigned short));
     if (nRet < 0)
     {
        printf("gipTrDB->Delete(%u) fail₩n", msgId);
     }
     else
     {
        printf("gipTrDB->Delete(%u) success₩n", msgId);
     }

     if (gipMsg != NULL)
     {
```

```
                    gipMsg->SetMessage(NULL, 0);
                    delete gipMsg;
                }
            }
        }
    }
```

[소스 3-1] 평면의 Origin 좌표를 원좌표계로 환원하는 함수

[소스 3-1]의 ProcessMsg() 함수는 GIVI의 preFrame() 함수에서 실행을 하게 되며, preFrame() 함수는 프로그램이 실행되는 동안 렌더링을 하기 위한 draw()함수를 호출하기에 앞서 매번 실행된다. 즉, GIVI는 ProcessMsg() 함수를 통해 애플리케이션이 실행되는 동안 매 프레임을 그리기에 앞서 메시지를 체크하고 처리하게 된다. GIP과의 트랜잭션을 생선하는 등의 초기 작업은 GIVI를 초기화할 같이 초기화되며, 이런 이유로 GIVI의 init() 함수에서 호출된다.

소스 코드에서 볼 수 있듯이, GIVI는 우선 GetTransaction() 함수를 통해 GLORE에서 전송된 메시지를 확인하며, 만약 response 메시지가 와 있으면 ProcessCmdRes() 함수를 호출해서 GLORE로부터의 response를 처리한다.

[소스 3-2]에서는 ProcessCmdRes() 함수를 볼 수 있다.

```
gipMessage *GIVI::ProcessCmdRes(unsigned char *cpMsg, unsigned int nLen)
{
  gipCmdResPolyData *cmdResPolyData;
  gipCmdResLoad     *cmdResLoad;
  gipCmdResImage    *cmdResImage;
  gipCmdResStatus   *cmdResStatus;
  gloreHistogram    *histo;
  gipBoundingBox    *bounds;
  unsigned char     *polydata;
  unsigned int      size;
  unsigned short    msgId;
```

```
double                  boundValue[6];
vtkColorTransferFunction *tmpTF;

switch (cpMsg[CMD_RES_TYPE_POS])
{
    case CMD_RES_POLY_DATA :
        cmdResPolyData = new gipCmdResPolyData();
        cmdResPolyData->Decode(cpMsg, nLen);
        ProcessCmdResPolyData(cmdResPolyData);
        return cmdResPolyData;
    case CMD_RES_LOAD :
        cmdResLoad = new gipCmdResLoad();
        cmdResLoad->Decode(cpMsg, nLen);
        gContextData->SetEndTime(cmdResLoad->GetEndTimeStep());
        gContextData->SetDeltaAngle(cmdResLoad->GetDeltaAngle());

        polydata = cmdResLoad->GetHistogram();
        size = cmdResLoad->GetHistogramLen();
        histo = gloreHistogram::New(polydata, size);

        gContextData->SetPressureTF(histo->GenTF("Pressure"));

        tmpTF = histo->GenTF("Velocity");
        if (tmpTF == NULL)
        {
            tmpTF = histo->GenTF("Velocity Magnitude");
        }
        gContextData->SetVelTF(tmpTF);

        tmpTF = histo->GenTF("Vorticity");
        if (tmpTF == NULL)
        {
            tmpTF = histo->GenTF("Vorticity Magnitude");
        }
        gContextData->SetVorTF(tmpTF);

        tmpTF = histo->GenTF("Q-Criteria");
        if (tmpTF == NULL)
```

```
        {
            tmpTF = histo->GenTF("Q Criteria");

            if (tmpTF == NULL)
            {
                tmpTF = histo->GenTF("Q-criteria");
            }
        }
        gContextData->SetQCriteriaTF(tmpTF);

        histo->Delete();

        /// set the bounding box
        bounds = cmdResLoad->GetBounds();
        boundValue[0] = bounds->minX;
        boundValue[1] = bounds->maxX;
        boundValue[2] = bounds->minY;
        boundValue[3] = bounds->maxY;
        boundValue[4] = bounds->minZ;
        boundValue[5] = bounds->maxZ;

        gContextData->SetPolyBounds(boundValue);

        msgId = cmdResLoad->GetMsgId();
        gipResDB->Delete(&msgId, sizeof(msgId));

        /// delete loading message from dotree
        if (gRoot->SearchNode("LoadMsgPanelTransform") != NULL)
        {
            dotree::Transform *lmPanelTransform = gRoot->SearchNode("LoadM
            sgPanelTransform")->GetTransform();
            dotree::Object *lmPanelObject = lmPanelTransform->GetChild(0)->G
            etObject();
            lmPanelObject->GetPanel()->DeleteAll();
            lmPanelTransform->RemoveChildren();
            gRoot->RemoveChild(lmPanelTransform);
        }
        return cmdResLoad;
```

```
        case CMD_RES_IMAGE :
            cmdResImage = new gipCmdResImage();
            cmdResImage->Decode(cpMsg, nLen);
            cmdResImage->Print();
            ProcessCmdResImageData(cmdResImage);
            return cmdResImage;
        case CMD_RES_STATUS :
            cmdResStatus = new gipCmdResStatus();
            cmdResStatus->Decode(cpMsg, nLen);
            cmdResStatus->Print();

            msgId = cmdResStatus->GetMsgId();
            if (msgId == gContextData->GetProbByPlaneId())
            {
               // if message id is equal to probe by plane msgId
               ProcessProbeByPlane();
               gPlaneWidget->GetPlaneActor()->GetProperty()->SetRepresentationT
               oSurface();
            }
            else
            {
               msgId = cmdResStatus->GetMsgId();
            }
            gipResDB->Delete(&msgId, sizeof(msgId));

            return cmdResStatus;

        default :
            return NULL;
    }
}
```

[소스 3-2] GIVI에서의 response 처리

GIVI는 ProcessCmdRes() 함수를 통해 response 메시지를 처리한다. 이 때, 메시지의 종류에 따라 그 처리방식이 다른데, 메시지의 종류에 따른 처리 방식은 switch문을 보면 알 수 있다.

response로 온 메시지가 CMD_RES_POLY_DATA 타입의 메시지인 경우에는 메시지를 디코딩한 뒤, ProcessCmdResPolyData() 루틴을

호출해서 오브젝트에 대한 렌더링을 수행한다.

메시지가 CMD_RES_LOAD 타입의 메시지인 경우, 즉, 데이터 로딩 요청에 대한 응답인 경우에는 메시지를 디코딩한 뒤, 메시지에 들어있는 기본적인 컨텍스트 데이터를 설정하고, 히스토그램 정보를 해석해서 컨텍스트 데이터 내에 Transfer Function을 저장한다. 이 때, Transfer Function의 종류는 Pressure, Velocity, Velocity Magnitude, Vorticity, Vorticity Magnitude, Q-Criteria 등이 있으며, 이런 히스토그램 정보는 모두 GIVI의 컨텍스트 데이터에 저장되게 된다. Load 메시지를 통해 설정되는 컨텍스트 정보로는 히스토그램을 통해 생성되는 Transfer Function 외에도 End Time Step, DeltaAngle, Bounding Box 정보 등이 있다.

CMD_RES_IMG 타입의 메시지는 그래프 정보를 처리하기 위한 메시지로, 디코딩이 이뤄진 뒤, ProcessCmdResImageData() 함수를 통해 처리된다.

이외에도 CMD_RES_STATUS 타입의 메시지가 있으며, 이 메시지는 Probe by Plane 의 기능을 실행하거나, 메시지의 상태를 설정하는데 사용된다. Probe by Plane 기능은 ProcessProbeByPlane() 함수에 의해 실행된다.

나. GIP 메시지의 종류

GIVI가 사용하는 메시지는 크게 cmd.req와 cmd.res가 있다.

cmd.req 메시지는 GIVI가 메시지를 요청할 때 사용하는 메시지 유형으로, 16비트의 msgid와 8비트로 구성된 Type, 그리고 Body의 Length를 나타내는 64비트의 Len 비트열이 뒤따르고, 그 뒤에 Len 길이만큼의 Body가 따른다. Type은 GIVI에서 요청하는 동작의 유형을 가리키는 것으로 Load, Scalar Distribution, Graph, Iso-Surface, Streamline, Animation, Pathline, Probe by Plane, Transfer function, 이 9가지가 있다. cmd.req에 대한 상세 사항은 [그림 3-2]를 참조하면 된다.

cmd.res는 GIVI의 요청에 대한 결과값을 싣는 메시지 유형으로 16비트의 msgid와 8비트의 Type, 메시지의 길이를 나타내는 32비트의 Len 비트열 뒤에 실제 데이터가 뒤따르게 된다. 메시지 타입에는 Poly

Data와 Status, 그리고 Transfer function key의 이 3가지가 존재한다. [그림 3-3]은 cmd.Res 메시지를 나타낸다.

cmd.req

msgid	Type	Len	Body
16 bit	8	64	Len

Type	Value
Load	0x01
Scalar Distribution	0x02
Graph	0x03
Iso-surface	0x04
Streamline	0x05
Animation	0x06
Pathline	0x07
Probe by Plane	0x08
Transfer function	0x09

[그림 3-2] cmd.req 메시지 상세

cmd.Res Body

msgid	Type	Len	Data
16	8	32	Len

Type	Value
Poly Data	0x01
Status	0x02
Transfer function key	0x03

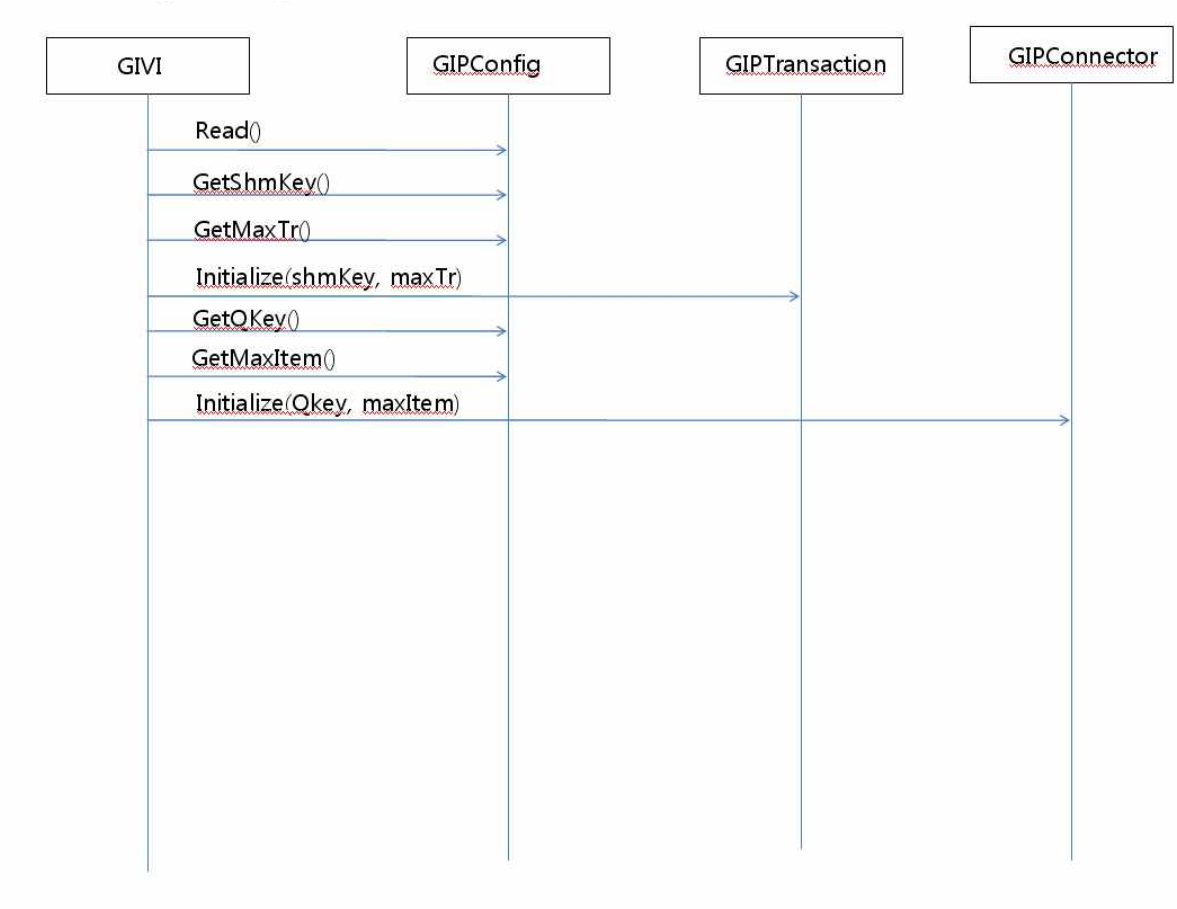
[그림 3-3] cmd.Res 메시지 상세

다. 초기화

GIVI가 GIP과의 커넥션을 초기화하는 과정은 [그림 3-4]의 Sequence 다이어그램에서 설명돼 있다.

GIVI에서는 Read()함수를 실행해서 초기화에 필요한 설정 정보를 가져온 뒤, GetShmKey(), GetMaxTr() 함수를 실행함으로써 초기화에 필요한 공유 메모리의 키와 트랜잭션 DB의 크기를 가져온다. 이 값으로 Initialize를 수행한 뒤에는 GetQKey()와 GetMaxItem() 함수를 수행함으로써 QDB의 키 값과 maxItem 값을 가져오고, 이 값을 토대로 최종적으로 GIPConnector와의 커넥션을 얻게 된다.

GIVI에서 이 과정은 GIVI 클래스의 init() 함수에서 수행하게 되는데, 이에 대한 구체적인 소스 코드는 [소스 3-3]에서 볼 수 있다.



[그림 3-4] GIVI Initialize() 함수에 대한 Sequence 다이어그램

```
void GIVI::init()
{

  int nRet;
  char errMsg[1024];
  char host[MAX_STR];

  gethostname(host, MAX_STR);

  gipTrDB = new gipTransactionDB(gipConf->GetTrDBKey());
  nRet = gipTrDB->CreateTable("GipTransaction", 250, 10, 1, sizeof(short));

  if (nRet < 0)
  {
     sprintf(errMsg, "gipTrDB->CreateTable(%x) fail₩n", gipConf->GetTrDBKey
     ());
     glvLog::Write(errMsg, 0);
  }

  sprintf(errMsg, "gipTrDB->CreateTable(%x) success₩n", gipConf->GetTrDBKey
  ());
  glvLog::Write(errMsg, 0);

  if (strcmp(host, gipConf->GetMasterHost()) == 0)
  {
     gContextData->SetMasterTrue();

     gipMsgQ = new gipMessenger();
     nRet = gipMsgQ->Create(gipConf->GetMsgQKey());

     if (nRet < 0)
     {
        printf("gipMsgQ->Create(%x) fail₩n", gipConf->GetMsgQKey());
        sprintf(errMsg, "gipMsgQ->Create(%x) fail₩n", gipConf->GetMsgQKey(
        ));
        glvLog::Write(errMsg, 0);
     }
  }
```

```
    gipResDB = new gipVariableTrDB(gipConf->GetResDBKey());

    nRet = gipResDB->CreateTable("ResTable", gipConf->GetResTrSize(), gipConf
    ->GetMaxResTrNum(), 1, sizeof(unsigned short));

    if (nRet < 0)
    {
        sprintf(errMsg, "resTrDB->CreateTable(%x) fail₩n", gipConf->GetResDBKey
        ());
        glvLog::Write(errMsg, 0);
    }

    sprintf(errMsg, "resTrDB->CreateTable(%x) success₩n", gipConf->GetResDBK
    ey());
    glvLog::Write(errMsg, 0);

    gipResQ = new gipMessenger();

    nRet = gipResQ->Create(gipConf->GetResMsgQKey());

    if (nRet < 0)
    {
        printf("msgQ->Create(%x)₩n", gipConf->GetResMsgQKey());
        sprintf(errMsg, "msgQ->Create(%x)₩n", gipConf->GetResMsgQKey());
        glvLog::Write(errMsg, 0);
    }

}
```

[소스 3-3] GIVI 커넥션 초기화

본래 init() 함수에는 트랜잭션을 초기화하는 과정 외에, VR Juggler 환경을 초기화하고 giviAnimation 및 각종 변수를 생성하고 초기화하는 과정이 포함돼 있지만, 여기에서는 GIP 커넥션을 맺는 과정만 남겨놓았다.

[소스 3-3]에서 볼 수 있듯이 GIVI는 TrDB를 생성하고 초기화한 다음, 메시지 큐를 생성하고 GLORE로부터의 response를 받는 ResDB를 생성한다. 그리고 GIP 메시지를 수신하는 response 메시지 큐를

생성하는 것으로 GIP과의 커넥션을 초기화한다.

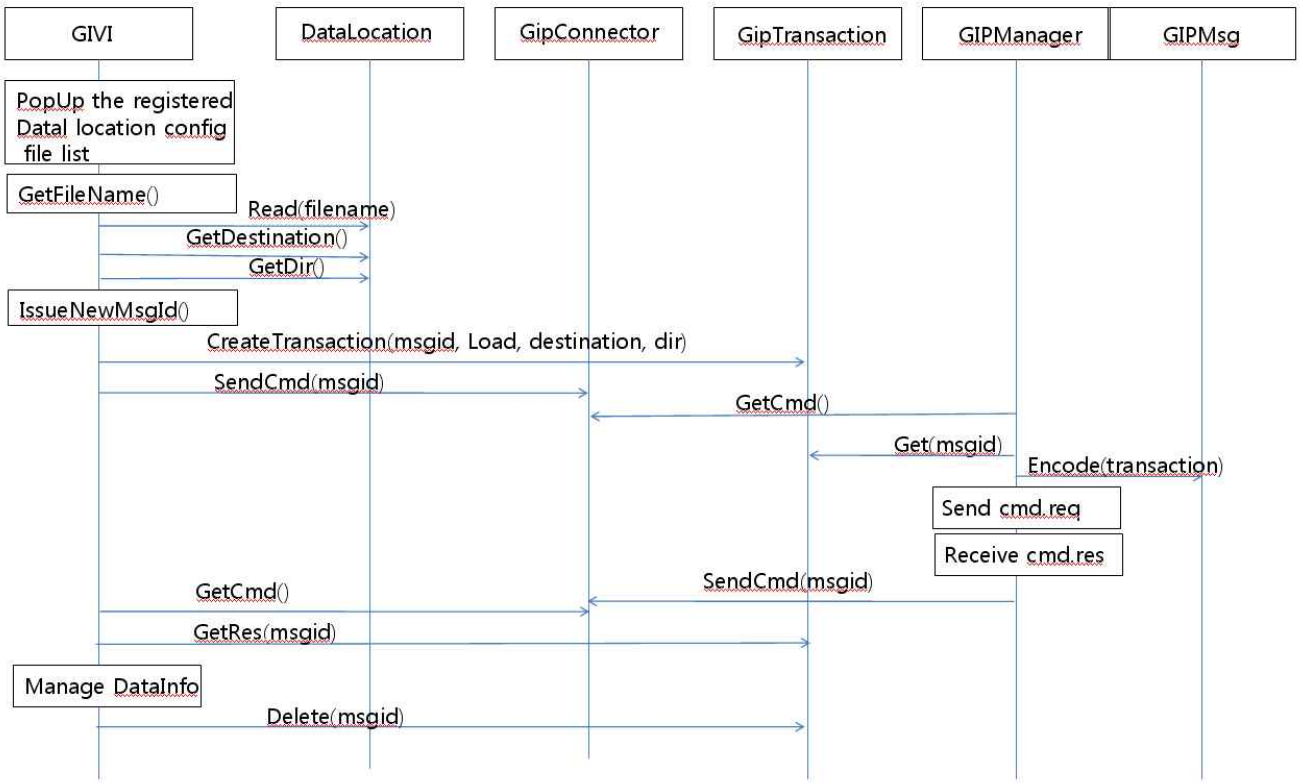
이렇게 커넥션이 초기화되고 나면 사용자와의 인터랙션에 따라 여러 동작을 수행하게 된다. 여기에서는 데이터 로딩과 폴리곤 데이터의 렌더링, 그리고 Probe By Plane의 동작 기제에 대해 설명하기로 한다.

라. 데이터 로딩

GIVI는 실행 초기 단계에서 데이터를 로딩하고 그 로딩된 데이터를 기반으로 모든 동작을 수행한다.

데이터를 로딩하는 과정은 [그림 3-5]의 Sequence 다이어그램과 같다.

GIVI에서 사용자가 Load 버튼을 클릭하면 내부적으로 LoadDataCB() 함수가 실행된다. 그러면 GIVI는 사전에 설정 파일에 설정돼 있는 데이터 디렉토리의 파일 목록을 사용자에게 보여준다. 그런 다음, 사용자가 선택한 파일명을 기반으로 트랜잭션을 생성, 데이터를 로딩하는 작업을 수행한다.



[그림 3-5] 데이터 로딩 과정

cmd.Req Load Primitive Body

filename prefix of simulation results to load

Type	Value
Load	0x01

[그림 3-6] cmdReq: Load

우선 GIVI는 사용자가 클릭한 파일명을 기반으로 파일을 읽는 Read 명령을 수행한다. 이 때, 새로운 메시지가 생성되는데, 이 메시지에는 Load 명령과 함께 데이터의 위치와 디렉토리 정보가 함께 수록된다. 이 메시지를 기반으로 request를 보내면 GIP은 GLORE로 request를 전달한 뒤, GLORE가 로딩한 데이터를 GetCmd()와 GetRes() 명령을 통해 가져와서 데이터의 초기화 정보를 관리하게 된다.

이 때, Load 작업을 수행하는 request 메시지 내에는 로딩할 데이터의 파일명이 실리게 된다. Load 작업에 대한 request 메시지는 [그림 3-6]에 나와 있다.

실질적으로 Load 메시지를 생성하고 전송하는 명령을 수행하는 코드는 다음의 [소스 3-4]와 같다.

```
void LoadFileCB(vtkObject *cObject, unsigned long eid, void *clientdata, void *
calldata)
{

    char *filename = (char*)(clientdata);

    unsigned short msgId = gContextData->GetMsgId();
    msgId++;
    gContextData->SetMsgId(msgId);

    gipCmdReqLoad *loadMsg = new gipCmdReqLoad();
```

```
    loadMsg->SetPrimitive(CMD_REQ);
    loadMsg->SetMsgType(CMD_REQ_LOAD);
    loadMsg->SetLoadPrefix(filename);
    loadMsg->SetMsgId(msgId);
    loadMsg->Encode();

    InsertAndSend(loadMsg);
    delete loadMsg;
}
```

[소스 3-4] GIVI에서의 Load 명령의 실행

LoadFileCB 함수에서는 request 메시지를 새로 생성해서 InsertAndSend 명령을 통해 GLORE에 전송한다. 이 때, 메시지에는 사용자가 선택한 파일명과 메시지 유형(CMD_REQ_LOAD) 등의 정보가 인코딩돼서 전달되며, 요청 메시지를 받은 GLORE는 해당 데이터를 로딩해서 response 메시지로 로딩한 뒤, GIVI로 전송하게 된다. 이 때, 동일한 작업에 대한 request 메시지의 message Id와 response 메시지의 message Id는 동일하며, GIVI는 이 message Id로 Response DB에 들어와 있는 데이터를 구분, 해당 작업에 대해 처리하게 된다. 이 때, GIVI에서는 response 메시지를 디코딩해서 각종 컨텍스트 데이터와 히스토그램 정보를 생성하게 된다. 이 과정은 [소스 3-5]에서 볼 수 있다.

```
cmdResLoad = new gipCmdResLoad();
cmdResLoad->Decode(cpMsg, nLen);
gContextData->SetEndTime(cmdResLoad->GetEndTimeStep());
gContextData->SetDeltaAngle(cmdResLoad->GetDeltaAngle());

polydata = cmdResLoad->GetHistogram();
size = cmdResLoad->GetHistogramLen();
histo = gloreHistogram::New(polydata, size);
```

[소스 3-5] GIVI에서의 Load 메시지에 대한 response 처리

마. 폴리곤 데이터의 처리

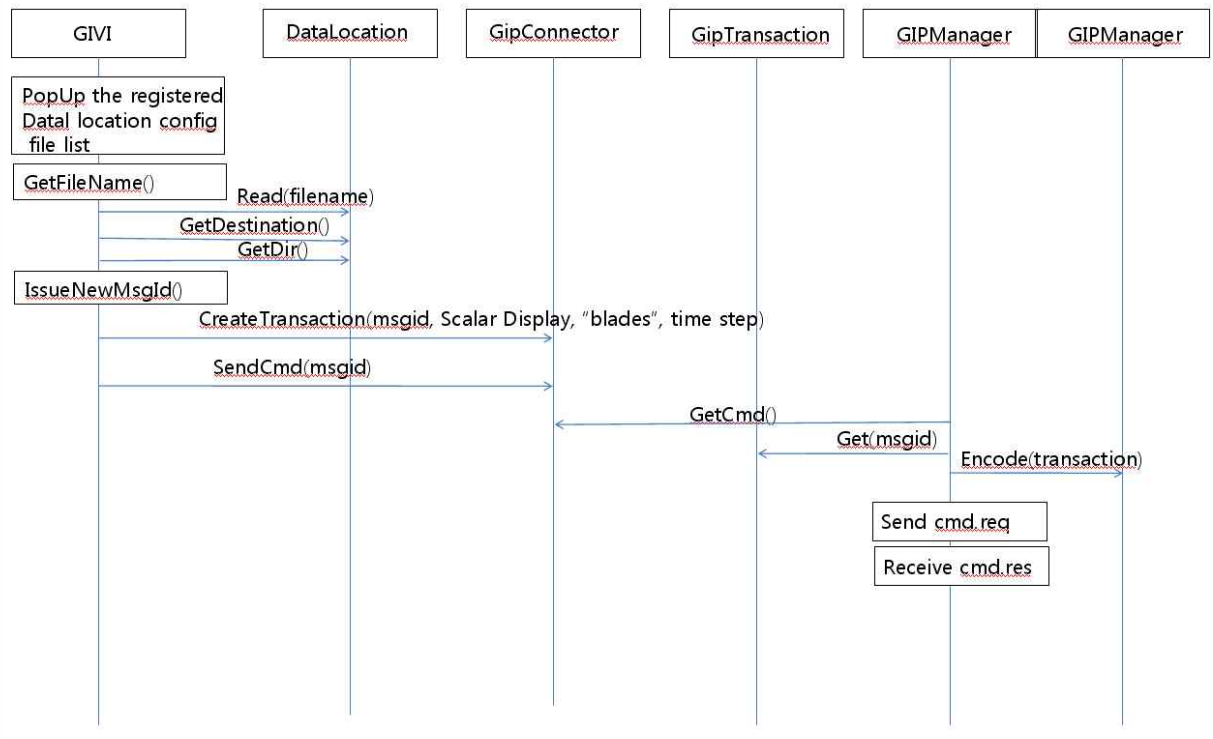
데이터 로딩이 끝나면 GIVI는 사용자의 요구사항을 GLORE로 전송하고, 그 처리 결과를 화면에 디스플레이한다. 가장 흔한 경우가 폴리곤 데이터를 처리하는 경우로, 여기에서는 Rotor 메뉴의 Pressure 버튼을 클릭했을 경우를 예제로 들어보기로 한다.

[그림 3-7]은 Pressure 메뉴를 클릭했을 때의 GIVI의 동작을 보여주는 Sequence 다이어그램이다.

GIVI는 메시지에 폴리곤 데이터를 로딩하는 데 필요한 정보(타임 스텝, 데이터의 종류 등)를 담아서 인코딩하고, GIP Manager를 통해 GLORE에 전송하게 된다.

여기에 필요한 request 메시지는 [그림 3-8]에서 볼 수 있다. 여기에는 Scalar Distribution의 타입과 현재 디스플레이되고 있는 컨텐츠의 타임스텝이 나와 있으며, GLORE는 이 값에 근거해서 적당한 데이터를 response 메시지로 전송하게 된다.

폴리곤 데이터를 요청하는 request 메시지 중에는 scalar distribution



[그림 3-7] 폴리곤 데이터 처리

Type	Time Step
8 bit	16 bit

Type	Value
Blade Pressure	0x01
Field Pressure	0x02
Vorticity	0x03
Velocity	0x04

Time Step : Current time step

[그림 3-8] cmdReq: Scalar Distribution

Type	Time Step	Value Type	Range	Start	End
8	16	8	8	64	64

Type	Value
Vorticity	0x01
Q-Criteria	0x02
Velocity	0x03

Value Type	Value
%	0x01
Physical value	0x02

Range	Value
>=Start	0x01
<=Start	0x02
>=Start and <= End	0x03
== Start	0x04

[그림 3-9] cmdReq: Iso-Surface

외에도 isosurface를 요청하는 메시지도 있다. isosurface를 요청하는 메시지는 [그림 3-9]에 나와 있다. isosurface의 종류에는 Vorticity, Q-Criteria, Velocity의 3가지 유형이 있으며, 이 메시지에는 값의 범위를 지정한 값도 함께 인코딩돼서 전송되므로, GLORE에서는 해당

범위 내에 속한 isosurface값만 response로 전송하면 된다.

실질적으로 폴리곤 데이터를 요청하는 메시지를 생성하고 전송하는 명령을 수행하는 코드는 다음의 [소스 3-6]과 같다.

```
void BladePressureCB( vtkObject* cObject, unsigned long eid, void* clientdata,
void *calldata )
{
  /// Request message setting
  gContextData->SetBladeModeOn();

  gContextData->GetReqScalar()->SetPrimitive(CMD_REQ);
  gContextData->GetReqScalar()->SetMsgType(CMD_REQ_SCALAR);
  gContextData->GetReqScalar()->SetScalarType(gipCmdReqScalar::BladePressu
  re);
  gContextData->GetReqScalar()->SetTimeStep(gContextData->GetCurrentTimeS
  tep());

  unsigned short msgId = gContextData->GetNextMsgId();
  gContextData->GetReqScalar()->SetMsgId(msgId);
  gContextData->SetMsgId(msgId);
  gContextData->GetReqScalar()->Encode();

  InsertAndSend(gContextData->GetReqScalar());

}
```

[소스 3-6] GIVI에서의 Scalar Distribution에 대한 폴리곤 데이터 요청

Scalar Distribution 데이터를 요청하는 메시지에는 메시지의 타입과 Scalar Type, 그리고 타임스텝 정보가 포함돼서 인코딩된다.

GLORE로부터 response로 받은 폴리곤 데이터를 처리하는 과정은 다음의 [소스 3-7]과 같다. Response 메시지를 체크하고 처리하는 과정에 포함되는 이 코드는 메시지를 디코딩한 뒤, ProcessCmdResPolyData 함수를 호출하는데, 이 함수에는 폴리곤 데이터를 렌더링해서 화면에 출력하는 루틴이 포함된다.

```
case CMD_RES_POLY_DATA :
    cmdResPolyData = new gipCmdResPolyData();
    cmdResPolyData->Decode(cpMsg, nLen);
    ProcessCmdResPolyData(cmdResPolyData);
```

[소스 3-7] GIVI에서의 폴리곤 데이터 요청에 대한 response 처리

바. Probe Plane

Probe Plane에 대해 request 메시지를 작성하고 응답으로 온 데이터를 로딩하는 과정은 기본적으로 폴리곤 데이터를 로딩하는 과정과 동일하다 볼 수 있다.

Probe Plane에 대한 request 메시지는 [그림 3-10]에 나와 있다.

request message에는 Presentation Type, Time Step, Plane Coordinate 및 Presentation Body가 포함된다. 이 때, Probe Plane의 타입으로는 Contour line과 Scalar distribution이 있으며, 여기에 GIVI에서 계산한 평면의 네 꼭지점의 좌표가 Plane Coordinate 부분에 인코딩된다. 만약 Presentation Type이 Scalar인 경우에는 Presentation Body 정보가 생략되며, Contour line의 경우엔 사용자가 지정한 값이 입력, 인코딩된다.

Presentation Type	Time Step	Plane coordinate	Presentation Body
8 bit	16 bit		

Presentation Type	Value
Contour line	0x01
Scalar distribution	0x02

cmd.Req "probe by plane" plane coordinate

Upper left coordinate	Upper right coordinate	Lower left coordinate	Lower right coordinate
64*3	64*3	64*3	64*3

[그림 3-10] cmdReq: Probe by Plane Body

[소스 3-8]에서는 Probe Plane에 대한 request 메시지가 인코딩되는 과정을 볼 수 있다.

```
gipCmdReqProbeByPlane  *cmdPbyPlane = new gipCmdReqProbeByPlane();

cmdPbyPlane->SetPrimitive(CMD_REQ);
cmdPbyPlane->SetMsgType(CMD_REQ_PROBE_BY_PLANE);

unsigned short msgId = gContextData->GetNextMsgId();
cmdPbyPlane->SetMsgId(msgId);

struct gipPlane plane;

plane.upperLeft.x = origin[0];
plane.upperLeft.y = origin[1];
plane.upperLeft.z = origin[2];

plane.upperRight.x = point1[0];
plane.upperRight.y = point1[1];
plane.upperRight.z = point1[2];

plane.lowerRight.x = point3[0];
plane.lowerRight.y = point3[1];
plane.lowerRight.z = point3[2];

plane.lowerLeft.x = point2[0];
plane.lowerLeft.y = point2[1];
plane.lowerLeft.z = point2[2];

cmdPbyPlane->SetPlaneCoordinate(plane);
cmdPbyPlane->SetTimeStep(gContextData->GetCurrentTimeStep());
cmdPbyPlane->SetGenMethod(gipCmdReqProbeByPlane::Auto);

switch (probMode)
{
   case PROBE_CONTOUR_VORTICITY :
          cmdPbyPlane->SetValueType(gipCmdReqProbeByPlane::Vorticity);
```

```
            cmdPbyPlane->SetPresentationType(gipCmdReqProbeByPlane::Cont
            ourLine);
            cmdPbyPlane->SetLineNum(probLineNum);
            break;
        case PROBE_CONTOUR_QCRITERIA :
            cmdPbyPlane->SetValueType(gipCmdReqProbeByPlane::QCriteria);
            cmdPbyPlane->SetPresentationType(gipCmdReqProbeByPlane::Cont
            ourLine);
            cmdPbyPlane->SetLineNum(probLineNum);
            break;
        case PROBE_CONTOUR_PRESSURE :
            cmdPbyPlane->SetValueType(gipCmdReqProbeByPlane::Pressure);
            cmdPbyPlane->SetPresentationType(gipCmdReqProbeByPlane::Cont
            ourLine);
            cmdPbyPlane->SetLineNum(probLineNum);
            break;
        case PROBE_SCALAR_VORTICITY :
            cmdPbyPlane->SetValueType(gipCmdReqProbeByPlane::Vorticity);
            cmdPbyPlane->SetPresentationType(gipCmdReqProbeByPlane::Scal
            arDistribution);
            break;
        case PROBE_SCALAR_QCRITERIA :
            cmdPbyPlane->SetValueType(gipCmdReqProbeByPlane::QCriteria);
            cmdPbyPlane->SetPresentationType(gipCmdReqProbeByPlane::Scal
            arDistribution);
            break;
        case PROBE_SCALAR_PRESSURE :
            cmdPbyPlane->SetValueType(gipCmdReqProbeByPlane::Pressure);
            cmdPbyPlane->SetPresentationType(gipCmdReqProbeByPlane::Scal
            arDistribution);
            break;
        default :
            printf("Unknown Probing Contents Type₩n");
            return;
```

[소스 3-8] GIVI에서의 Probe by Plane에 대한 request

사. GIVI의 컨텍스트 관리

GIVI는 실행시간 동안 현재 디스플레이하고 있는 컨텐츠들에 관한 정보를 유지하고 업데이트하면서 디스플레이 정보를 관리하며, 이 정보에 근간해서 GLORE에 오브젝트 관련 메시지를 요청하게 된다.

이를 위해 GIVI에서는 giviContext라는 클래스를 전역변수로 생성, GIVI가 실행되는 동안의 디스플레이 상황을 이 변수를 통해 관리한다. 이렇게 GIVI가 관리하는 정보로는

- 현재 디스플레이되고 있는 오브젝트의 종류 및 해당 오브젝트가 전송된 메시지의 메시지 아이디
- 타임스텝 관련 정보: 현재 타임스텝, Start Time, End Time
- 델타 앵글
- 바운딩 박스 정보
- 컷팅 플레인을 생성하는 Probe Plane에 관한 정보: 메시지 아이디, Contour의 경우 contour의 개수, 현재 Probe하고 있는 컨텐츠의 종류 등
- 애니메이션 관련 정보
- GIVI에서 사용하는 각종 request 메시지

등이 있다.

다음은 giviContext 클래스의 멤버 변수를 나타낸다.

- mMsgId: unsigned short
- mAniMsgId: unsigned short
- mProbByPlaneId: unsigned short
- mBladePressureId: unsigned short
- mVelIsoId: unsigned short
- mVorIsoId: unsigned short
- mVelStreamId: unsigned short
- mVorStreamId: unsigned short
- mQCriteriaId: unsigned short

- mProbByPlaneAniId: unsigned short
- mBladePressureAniId: unsigned short
- mVelIsoAniId: unsigned short
- mVorIsoAniId: unsigned short
- mVelStreamAniId: unsigned short
- mVorStreamAniId: unsigned short
- mQCriteriaAniId: unsigned short
- mCurrentTimeStep: unsigned short
- mStartTime: unsigned short
- mEndTime: unsigned short
- mDeltaAngle: float
- mBladeMode: bool
- mVelIsoMode: bool
- mVorIsoMode: bool
- mStreamlineMode: bool
- mQCriteriaMode: bool
- mGraphMode: bool
- mPbyPlaneMode: bool
- mIsAniNextFirst: bool
- mProbContext: int
- mProbLineNum: unsigned short
- mIsoValue: double
- mNumAniCache: int
- mReqScalar: gipCmdReqScalar *
- mReqIsoS: gipCmdReqIsoS *
- mReqStreamLine: gipCmdReqStreamLine *
- mReqGraph: gipCmdReqGraph *

- mReqProbeByPlane: gipCmdReqProbeByPlane *
- mTF: vtkColorTransferFunction **
- mAniMaster: bool
- mIsAniStart: bool
- mButtonStatus: bool[3]
- mPolyBounds: double[6]
- mGiviAni: giviAnimation *

다음은 giviContext 클래스의 멤버 함수다.

- SetMsgId(value: unsigned short): void
  - mMsgId 값을 지정된 값으로 설정한다. 이 mMsgId 값은 GIVI에서 메시지 아이디를 부여하기 위해 사용하는 변수로, 현재의 메시지 아이디중 최대값을 나타낸다.
- GetMsgId(value: void): unsigned short
  - mMsgId값을 리턴한다.
- GetNextMsgId(value: void): unsigned short
  - GIVI에서 메시지 아이디를 부여하기 위해 사용하는 함수로, mMsgId값에 1을 더한 값을 리턴한다.
- SetAniMsgId(value: unsigned short): void
  - mAniMsgId값을 지정된 값으로 설정한다.
- GetAniMsgId(value: void): unsigned short
  - mAniMsgId값을 리턴한다.
- GetNextAniMsgId(value: void): unsigned short
  - GIVI에서 애니메이션 메시지 아이디를 부여하기 위해 사용하는 함수로, mAniMsgId값에 1을 더한 값을 리턴한다.
- SetObjId(value: unsigned short, unsigned char, unsigned char): void
  - GIVI에서 디스플레이하고 있는 각종 오브젝트에 대한 아이디

를 설정하는 함수

- mVelIsoId, mVorIsoId, mBladePressureId, mVelStreamId, mVorStreamId, mProbByPlaneId에 대한 값을 파라미터에 따라 설정함으로써, GIVI에서 현재 디스플레이하고 있는 모든 컨텐츠에 대한 오브젝트 아이디를 관리할 수 있다.

- GetObjId(value: unsigned char, unsigned char): unsigned short
  - 파라미터에 따라 mVelIsoId, mVorIsoId, mBladePressureId, mVelStreamId, mVorStreamId, mProbByPlaneId 중 적절한 값을 리턴함으로써 GIVI에서 오브젝트 컨텐츠를 관리할 때 사용한다.
- SetProbByPlaneId(value: unsigned short): void
  - mProbByPlaneId값을 지정된 값으로 설정한다.
- GetProbByPlaneId(value: void): unsigned short
  - mProbByPlaneId값을 리턴함으로써 GIVI에서 probe plane을 관리할 수 있게 한다.
- SetBladePressureId(value: unsigned short): void
  - mBladePressureId값을 지정된 값으로 설정한다.
- GetBladePressureId(value: void): unsigned short
  - mBladePressureId값을 리턴한다.
- SetVelIsoId(value: unsigned short): void
  - mVelIsoId 값을 지정된 값으로 설정한다.
- GetVelIsoId(value: void): unsigned short
  - mVelIsoId 값을 리턴한다.
- SetVorIsoId(value: unsigned short): void
  - mVorIsoId 값을 지정된 값으로 설정한다.
- GetVorIsoId(value: void): unsigned short
  - mVorIsoId 값을 리턴한다.
- SetVelStreamId(value: unsigned short): void
  - mVelStreamId 값을 지정된 값으로 설정한다.

- GetVelStreamId(value: void); unsigned short
  - mVelSteramId 값을 리턴한다.
- SetVorStreamId(value: unsigned short): void
  - mVorSteramId 값을 지정된 값으로 설정한다.
- GetVorStreamId(value: void): unsigned short
  - mVorStreamId 값을 리턴한다.
- SetQCriteriaId(value: unsigned short): void
  - mQCriteriaId 값을 지정된 값으로 설정한다.
- GetQCriteriaId(value: void): unsigned short
  - mQCriteriaId 값을 리턴한다.
- SetProbContext(value: int): void
  - mProbContext 값을 지정된 값으로 설정한다.
- GetProbContext(value: void): int
  - mProbContext 값을 리턴한다.
- SetProbLineNum(value: unsigned short): void
  - mProbLineNum 변수를 지정된 값으로 설정한다.
- GetProbLineNum(value: void): unsigned short
  - mProbLineNum 변수를 리턴한다.
- SetCurrentTimeStep(value: unsigned short): void
  - mCurrentTimeStep, 즉 현재 타임 스텝값을 지정된 값으로 설정한다.
- SetStartTime(value: unsigned short): void
  - mStartTime, 즉 시작 타임스텝값을 지정된 값으로 설정한다.
- SetEndTime(value: unsigned short): void
  - mEndTime, 즉 마지막 타임스텝값을 지정된 값으로 설정한다.
- GetCurrentTimeStep(value: void): unsigned short
  - mCurrentTimeStep값을 리턴한다.
- GetNextTimeStep(value: void): unsigned short

- mCurrentTimeStep에 1을 더한 값을 리턴한다.

- GetPreTimeStep(value: void): unsigned short
  - mCurrentTimeStep에서 1을 뺀 값을 리턴한다.
- GetStartTime(value: void): unsigned short
  - mStartTime 값을 리턴한다.
- GetEndTime(value: void): unsigned short
  - mEndTime 값을 리턴한다.
- IncreaseCurrentTimeStpe(value: void): void
  - 현재 타임 스텝, 즉 mCurrentTimeStep값을 1 증가시킨다.
- DecreaseCurrentTimeStep(value: void): void
  - 현재 타임 스텝, 즉 mCurrentTimeStep값을 1 감소시킨다.
- SetDeltaAngle(value: float): void
  - mDeltaAngle 변수를 지정된 값으로 설정한다.
- GetDeltaAngle(value: void): float
  - mDeltaAngle 값을 리턴한다.
- SetGiviAnimation(value: int): void
  - giviAnimation 변수인 mGiviAni 변수를 새로 생성한다. 이 때, 캐쉬 개수를 지정된 값으로 설정한다.
- GetGiviAnimation(void): giviAnimation *
  - mGiviAni 변수를 리턴한다.
- SetBladeMode(value: bool): void
  - mBladeMode 변수를 지정된 값으로 설정한다. 블레이드가 화면에 그려졌을때 이 값을 true로 설정하고, 화면에서 지워지면 false로 설정한다.
- SetVelIsoMode(value: bool): void
  - mVelIsoMode 변수를 지정된 값으로 설정한다. Velocity에 대한 Isosurface가 화면에 그려질 때 이 값을 true로 설정하고, 화면에서 지워지면 false로 설정한다.

- SetVorIsoMode(value: bool): void
  - mVorIsoMode 변수를 지정된 값으로 설정한다. Vorticity에 대한 Isosurface가 화면에 그려질 때 이 값을 true로 설정하고, 화면에서 지워지면 false로 설정한다.
- SetStreamLineMode(value: bool): void
  - mStreamLineMode 변수를 지정된 값으로 설정한다. Streamline이 화면에 그려질 때 이 값을 true로 설정하고, 화면에서 지워지면 false로 설정한다.
- SetQCriteriaMode(value: bool): void
  - mQCriteriaMode 변수를 지정된 값으로 설정한다. QCriteria에 대한 Isosurface가 화면에 그려질 때 이 값을 true로 설정하고, 화면에서 지워지면 false로 설정한다.
- SetPbyPlaneMode(value: bool): void
  - mPbyPlaneMode 변수를 지정된 값으로 설정한다. 컷팅 플레인이 화면에 그려질 때 이 값을 true로 설정하고, 화면에서 지워지면 false로 설정한다.
- GetBladeMode(value: void): bool
  - mBladeMode 변수값을 리턴한다.
- GetVelIsoMode(value: void): bool
  - mVelIsoMode 변수값을 리턴한다.
- GetVorIsoMode(value: void): bool
  - mVorIsoMode 변수값을 리턴한다.
- GetStreamLineMode(value: void): bool
  - mStreamLineMode 변수값을 리턴한다.
- GetQCriteriaMode(value: void): bool
  - mQCriteriaMode 변수값을 리턴한다.
- GetPbyPlaneMode(value: void): bool
  - mPbyPlaneMode 변수값을 리턴한다.
- SetIsoValue(value: double): void

- mIsoValue 변수를 지정된 값으로 설정한다.

- GetIsoValue(value: void): double
  - mIsoValue 변수값을 리턴한다.
- SetNumAniCache(value: int): void
  - mNumAniCache 변수를 지정된 값으로 설정한다. 이 변수는 애니메이션의 캐시 개수를 지정하는데 사용된다.
- GetNumAniCache(value: void): int
  - mNumAniCache 변수값을 리턴한다.
- SetPolyBounds(value: double *): void
  - 로딩된 데이터의 boundary 정보를 설정한다.
- GetPolyBounds(value: void): double *
  - mPolyBounds 값을 리턴한다.
- SetPressureTF(value: vtkColorTransferFunction *): void
  - Pressure 데이터에 대한 Transfer Function을 설정하는 함수로, 지정된 값으로 Transfer Function을 설정한다.
- SetVelTF(value: vtkColorTransferFunction *): void
  - Velocity 데이터에 대한 Transfer Function을 설정하는 함수로, 지정된 값으로 Transfer Function을 설정한다.
- SetVorTF(value: vtkColorTransferFunction *): void
  - Vorticity 데이터에 대한 Transfer Function을 설정하는 함수로, 지정된 값으로 Transfer Function을 설정한다.
- SetQCriteriaTF(value: vtkColorTransferFunction *): void
  - QCriteria 데이터에 대한 Transfer Function을 설정하는 함수로, 지정된 값으로 Transfer Function을 설정한다.
- GetPressureTF(value: void): vtkColorTransferFunction *
  - Pressure 데이터에 대한 Transfer Function을 리턴한다.
- GetVelTF(value: void): vtkColorTransferFunction *
  - Velocity 데이터에 대한 Transfer Function을 리턴한다.

- GetVorTF(value: void): vtkColorTransferFunction *
  - Vorticity 데이터에 대한 Transfer Function을 리턴한다.
- GetQCriteriaTF(value: void): vtkColorTransferFucntion *
  - QCriteria 데이터에 대한 Transfer Function을 리턴한다.
- SetReqScalar(value: gipCmdReqScalar *): void
  - mReqScalar 변수를 지정된 값으로 설정한다.
- SetReqIsoS(value: gipCmdReqIsoS *): void
  - mReqIsoS 변수를 지정된 값으로 설정한다.
- SetReqStreamLine(value: gipCmdReqStreamLine *); void
  - mReqStreamLine 변수를 지정된 값으로 설정한다.
- SetReqProbByPlane(value: gipCmdReqStreamLine *): void
  - mReqProbByPlane 변수를 지정된 값으로 설정한다.
- GetReqScalar(value: void): gipCmdReqScalar *
  - mReqScalar 값을 리턴한다.
- GetReqIsoS(value: void): gipCmdReqIsoS *
  - mReqIsoS 값을 리턴한다.
- GetReqStreamLine(value: void): gipCmdReqStreamLine *
  - mReqStreamLine 값을 리턴한다.
- GetReqProbByPlane(value: void): gipCmdReqProbeByPlane *
  - mReqProbByPlane 값을 리턴한다.

이런 루틴을 이용, GIVI는 애플리케이션이 실행되는 동안 컨텍스트를 관리하게 된다.

4. 결론

GLOVE는 가상현실 인터페이스를 이용해서 사용자가 데이터를 가시화하고 분석할 수 있게 해주는 프레임워크로, 대용량 데이터와 고해상도 디스플레이를 지원한다. GLOVE는 인터페이스인 GIVI와 렌더링 엔진 부분인 GLORE, 그리고 GIVI와 GLORE간의 통신을 담당하는 GIP으로 나뉘어 있으며, GIVI와 GLORE는 GIP을 통해 request와 response를 주고받으며 가시화 작업을 수행하게 된다.

GLOVE는 데이터를 계층적 구조로 관리함으로써 범용 가시화를 위한 프레임워크 위에 각 애플리케이션의 특성을 반영하는 맞춤형 사용자 인터페이스를 제공한다. 데이터 계층구조의 가장 아랫부분에는 Data I/O management 계층이 있으며, 그 위에는 Basic data management 계층이, 그리고 그 윗단에는 Application common data management 계층이 존재한다. 응용에 따라 지원하는 자료 구조는 이 Application 단에서부터 달라진다.

GIVI의 모든 동작은 message를 기반으로 하며, 실질적으로 GIVI에서는 처음에 트랜잭션을 생성하고 난 뒤, GLORE로부터 response가 올 EO까지 계속 대기 모드로 루프를 돌다가 GLORE로부터 response가 올 EO마다 데이터를 처리한다. 이렇게 분리된 구조로 인해 GIVI에서는 컨텍스트 데이터를 따로 관리함으로써 디스플레이 정보를 관리하고, 이 정보를 기반으로 GLORE에 데이터를 요청한다.

GLOVE는 이런 식으로 구성돼 있기 때문에 GLORE와 GIVI를 물리적으로 분리하는 것이 가능하며, 이런 식으로 원격지의 대용량 데이터를 GIVI에서 디스플레이하는 것이 가능하다.