ISBN 978-89-6211-286-3

VTKCAVE: 분석 및 사용

(VTKCAVE: Analysis & Usage)

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

1. 개요

VTKCave는 VTK 프로그램을 CAVELib을 사용하는 가상환경에서 렌더링하는데 사용되는 라이브러리로, 3차원 몰입형 애플리케이션을 개발하는데 필요한 소프트웨어 개발 시스템이다. VTKCave는 VTK와 CAVELib을 사용하며, VTK 라이브러리로부터 상속받은 강력한 객체지향 구조를 사용한다.

VTKCave는 VTK 모델링과 같은 모델링 기법, 3차원 가상 메뉴, 교차 알고리즘, CAVE 스타일의 상호작용(interaction)과 같은 기능을 제공하는데, 사용자는 2차원 이미지와 3차원 그래픽스 알고리즘 및 데이터를 혼용해서 사용할 수 있다. 또, VTK 라이브러리에 추가 가능한 요소로 취급되기 때문에 사용자는 별도로 OS나 디스플레이 시스템, 디바이스-레벨의 프로그래밍 기법에 신경쓰지 않고 자신의 애플리케이션에 VR 기능을 덧붙일 수 있다.

그러나 VTKCave는 VTK 4.x 버전을 대상으로 작성됐기 때문에, 현재의 VTK5.0 버전과는 잘 호환되지 않는다. 이에 따라 VTK 5.0으로의 포팅 작업이 반드시 필요하다. 이 문서의 3장에서는 VTK 5.0에서 크게 달라진 점에 대해 설명하고, 4장에서 VTKCave의 구조를 설명하면서 각 클래스에서 VTK 5.0 버전으로의 포팅을 위해 수정한 사항을 설명하기로 한다.

2. 컴파일

VTKCave를 컴파일하는 방법은 다음과 같다.

1. TMake 설치

TMake는 QT 프로그램을 위한 Makefile 생성기로, Trolltech(http://trolltech.com/)에서 다운로드할 수 있다. TMake를 다운로드해서 압축

```
TMAKEPATH = /usr/local/tmake/lib/linux-g++ (tmake 내의 lib 디렉토리)
PATH = $PATH:/usr/local/tmake/bin  (tmake, progen 실행파일이 존재하는 디렉토리)
export TMAKEPATH PATH
```

[그림 2-1] TMake 디렉토리 추가

을 푼 후, .bash_profile에 다음과 같이 TMake 디렉토리를 추가하면 된다.

2. 모든 *.pro 파일에서 VTK와 CAVELib 패스를 수정한다.
3. LD_LIBRARY_PATH에 VTKCave 디렉토리 내의 lib 디렉토리를 추가한다.
4. VTKCave 디렉토리에서 gmake 명령을 실행한다.
5. 그러면 VTKCave/bin 디렉토리에 여러 example의 실행 파일이 생성된다.

3. VTK 4.x vs. VTK 5.0

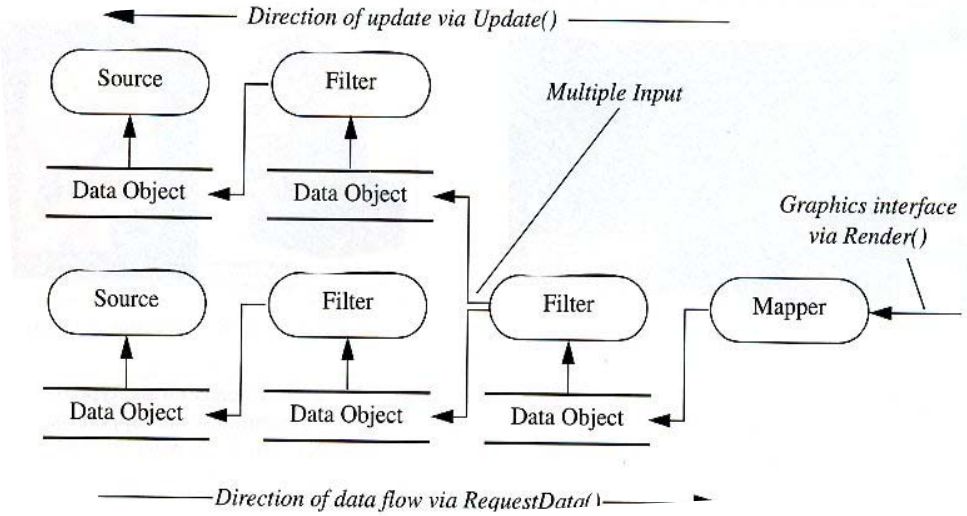
3장에서는 VTK가 버전 5.0으로 바뀌면서 크게 달라진 점에 대해 설명하기로 한다.

가. 파이프라인의 실행

VTK의 필터는 algorithm과 executive 오브젝트, 이 두 부분으로 나뉘어 있다. algorithm 오브젝트는 vtkAlgorithm에서 파생된 클래스로 데이터를 처리하는 역할을 담당한다. 반면 execuive 오브젝트는 vtkExecutive에서 파생된 클래스로 알고리즘을 언제 실행할지, 그리고 어떤 데이터를 처리할지를 algorithm 오브젝트에게 알려주는 역할을 담당한다. 필터의 executive 오브젝트 부분은 algorithm 오브젝트와는 독립적으로 생성되기 때문에 VTK 클래스의 코어 부분과는 전혀 별개로 실행된다.

필터에 의해 생성된 데이터는 하나 이상의 output port에 저장된다. Output port 하나는 논리적으로 필터의 출력 하나와 대응되는데, 예를 들어서 어떤 필터가 이미지 하나와 바이너리 마스크 이미지 하나를 생성한다고 하면 그 필터에는 2개의 output port가 정의돼 있어야 한다. 물론 각 output port는 각각의 이미지를 담게 된다.

파이프라인과 관련된 정보는 각 output port상의 vtkInformation 인스턴스에 저장된다. Output port에 대한 데이터는 vtkDataObject에서 파생된 클래스의 인스턴스에 저장된다. 필터가 필요로 하는 데이터는 하나 이상의 input port를 통해 입력된다. Input port 하나는 필터의 입

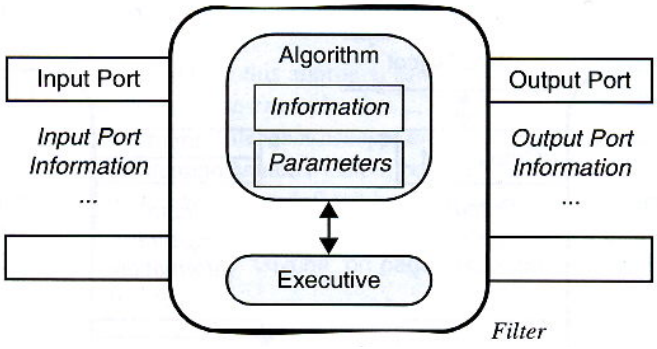


[그림 3-1] VTK의 파이프라인 실행

력 하나에 대응되는데, input port는 다른 필터의 output port를 참조하는 input connection을 저장한다. 이렇게 다른 필터의 input connection과 연결된 output port는 필터에 데이터를 제공하는 역할을 수행한다. 각각의 input connection은 하나의 데이터 오브젝트와 connection이 맺어지는 output port로부터 얻은 정보를 제공하게 된다.

[그림 3-1]은 VTK의 파이프라인 실행을 나타내고 있다. 일반적으로 파이프라인의 실행은 mapper의 Render() method가 호출되는 순간 발생하게 되는데, 이 시점은 통상 vtkActor의 Render() method가 호출되는 시점이다. 이 method가 호출되면 mapper의 input에서는 Update() method가 호출되는데, 이 프로세스는 파이프라인을 거슬러 올라가면서 계속 반복, 실행되며 데이터를 요청한다. 결국 데이터는 이 요청을 초기화한 오브젝트, 즉 mapper로 리턴된다. 실제로는 RequestData() method가 파이프라인에서 필터를 실행시키며 output data를 생성한다. Update() method의 호출은 파이프라인을 거슬러 올라가면서 발생되고, 데이터의 흐름은 파이프라인을 거슬러 내려가면서 발생한다.

각 필터의 구조는 [그림 3-2]에서 볼 수 있다. 필터는 executive와 algorithm으로 구성돼 있는데, executive 오브젝트는 algorithm의 실행을 관리하고 pipeline을 통해 전달되는 information request를 제어하는 역할을 담당한다. 이런 관점에서 보면 필터는 파이프라인과 독립적



[그림 3-2] 필터를 구성하는 algorithm, executive 및 port의 관계

이며 algorithm의 인터페이스와 관련된 모든 정보를 포함한다.

1) 파이프라인 오브젝트의 연결

VTK 5.0 이전 버전에서는 입력 데이터는 Input 인스턴스 변수로 표현하고 SetInput() method로 설정했다. 출력 데이터는 Output 인스턴스 변수로 표현하며, GetOutput() method로 접근했으며, 이런 구조에서는 다음과 같은 방식으로 필터를 서로 연결했다.

```
filter2->SetInput(filter1->GetOutput()); //Prior to VTK 5.0
```

[그림 3-3] 파이프라인 오브젝트의 연결 (버전 5.0 이전)

그러나 이런 구조에서는 다음과 같은 문제를 야기할 수 있었다.

- 컴파일시 타입 체킹이 이뤄지기 때문에 임의의 reader type이나 다른 type의 출력을 생성하는 필터를 지원하기가 어려웠다.
- 파이프라인의 업데이트와 관리가 암묵적으로 process 오브젝트 및 data 오브젝트와 상관관계가 있었다. 즉 파이프라인 관리에 data/process 오브젝트의 변경 작업이 수반됐다.
- Update() method를 전달하는 동안에는 파이프라인의 실행을 중단하기가 어려웠다. 게다가 에러 체크를 중앙에서 관리하기 어려워서 각각의 필터에 에러 체크하는 코드가 중복 존재하는 상황이 발생했다.
- 메타 데이터를 파이프라인에 내놓는 과정에서 data 및 process 오브젝트에 대한 API를 변경해야 하는 경우가 발생한다. API를 변경

하지 않고도 reader가 데이터스트림에 메타 데이터를 추가할 수 있게 함으로써 필터가 그 데이터를 받게 하는 기능이 필요했다.

이런 문제와 병렬 처리와 관련된 문제점 때문에 VTK 파이프라인의 변경이 불가피했다. 이렇게 해서 VTK 5.0에서부터는 VTK의 파이프라인 구성에 [그림 3-4]와 같이 connection과 port를 이용한다.

```
filter2->SetInputConnection(filter1->GetOutputPort()); //VTK 5.0
```

[그림 3-4] 파이프라인 오브젝트의 연결

예를 들어 보자. vtkGlyph3D는 여러개의 입력을 받아들여서 하나의 출력을 생성하는 필터로, vtkGlyph3D의 입력은 Input과 Source 인스턴스 변수로 표현된다. vtkGlyph3D의 역할은 Source 데이터에 정의된 geometry를 Input에 정의된 각 포인트로 복사하는 것이다. geometry는 Source 데이터값(ex. 스칼라 및 벡터 값)에 따라 수정된다. vtkGlyph3D 오브젝트를 사용하려면 다음과 같은 코드를 작성해야 한다.

```
Glyph = vtkGlyph3D::New();
  glyph->SetInputConnection(foo->GetOutputPort());
  glyph->SetSourceConnection(bar-.GetOutputPort());
....
```

[그림 3-5] vtkGlyph3D 필터

여기에서 foo와 bar는 적절한 형태의 출력을 생성하는 필터다. 또 다른 예로는 vtkExtractVectorComponent를 들 수 있다. 이 필터는 하나의 입력을 받아서 여러개의 출력 데이터를 생성하는 필터로, 3D 벡터의 각 요소를 3개의 스칼라 값으로 분리하는 역할을 수행한다. 3종류의 출력값은 각각 output port 0, 1, 2로 나가게 된다. 이 필터는 다음과 같이 사용할 수 있다.

```
vz = vtkExtractVectorComponents::New();
foo = vtkDataSetMapper::New();
   foo = SetInputConnection(vz->GetOutputPort(2));
```

[그림 3-6] vtkExtractVectorComponent 필터

또, vtkMergeFilter와 vtkAppendFilter를 비교해볼 수도 있다. vtkMergeFilter에는 여러개의 입력 포트가 있지만vtkAppendFilter에는 논리적으로 입력 포트가 하나밖에 없다. 그러나 이 하나의 입력 포트에 대해 여러개의 connection을 맺을 수가 있다. vtkMergeFilter의 경우에는 여러 종류의 입력이 서로 다른 용도로 사용되지만, vtkAppendFilter의 경우에는 모든 입력이 같은 의미를 지닌다. [그림 3-7]을 보면, AddInputConnection() method는 connection 목록에 추가하는 개념이지만, SetInputConnection()은 목록을 초기화하고 port에 단일 connection을 명시하는 역할을 수행한다. VTK 5.0의 파이프라인은 이런 식으로 구성된다.

```
merge = vtkMergeFilter::New();
  merge->SetGeometryConnection(foo->GetOutputPort());
  merge->SetScalarConnection(bar->GetOutputPort());

append = vtkAppendFilter::New();
  append->AddInputConnection(foo->GetOutputPort());
  append->AddInputConnection(bar->GetOutputPort());
```

[그림 3-7] vtkMergeFilter 필터와 vtkAppendFilter 필터

2) 파이프라인의 Executive와 Information 오브젝트

VTK에서는 파이프라인의 실행을 돕기 위해 메타데이타와 information 오브젝트를 사용하는데, 이 데이터는 데이터셋을 기술하는데 사용되는 데이터를 가리킨다. 이 오브젝트들은 vtkInformation의 하위 클래스다.

Pipeline information 오브젝트는 파이프라인 실행에 필요한 정보를 포함하는데, 이 오브젝트는 vtkExecutive의 인스턴스일 수도 있고 subclass일 수도 있다. 이 오브젝트는 vtkExecutive::GetOutputInformation() method를 통해 접근할 수 있다. 각 출력 포트에는 하나의 pipeline information 오브젝트가 할당되며, 대응 포트에 대해 출력 vtkDataObject에 대한 접근점이 주어진다. vtkDataObject는 대응되는 pipeline information 오브젝트에 대핸 포인터 백(pointer back)를 포함하며, vtkDataObject::GetPipelineInformation() method를 통해 접근할 수 있다. pipeline information 오브젝트는 필터가 실행돼서 출력물을 생성할 때 데이터 오브젝트에 어떤 데이터를 넣어야 하는지에 대한 정

보도 포함한다. 실제로 저장되는 정보는 출력 데이터 타입과 사용되는 실행 모델에 의해 결정된다. 입력 connection에 대한 pipeline information 오브젝트는 vtkExecutive::GetInputInformation()으로 접근 가능하다.

Port information 오브젝트는 출력 포트에서 생성되고 입력 포트에서 소비되는 데이터 유형에 대한 정보를 포함한다. 이 정보는 vtkAlgorithm의 인스턴스에 저장되는데, 입력 포트 하나당 하나의 information 오브젝트가, 그리고 출력 포트 하나에 대해서도 하나의 information 오브젝트가 존재한다. 이 정보는 vtkAlgorithm::GetInputPortInformation()과 vtkAlgorithm::GetOutputPortInformation() method를 통해 접근할 수 있다. Port information object는 필터의 인터페이스를 명시하기 위한 목적으로 vtkAlgorithm의 subclass로 생성된다.

Request information 오브젝트는 executive나 algorithm에 전달된 특정 request에 대한 정보를 포함한다. 이 information 오브젝트는 public method로는 접근할 수 없으며, 해당 request를 구현한 ProcessRequest() method로만 전달 가능하다.

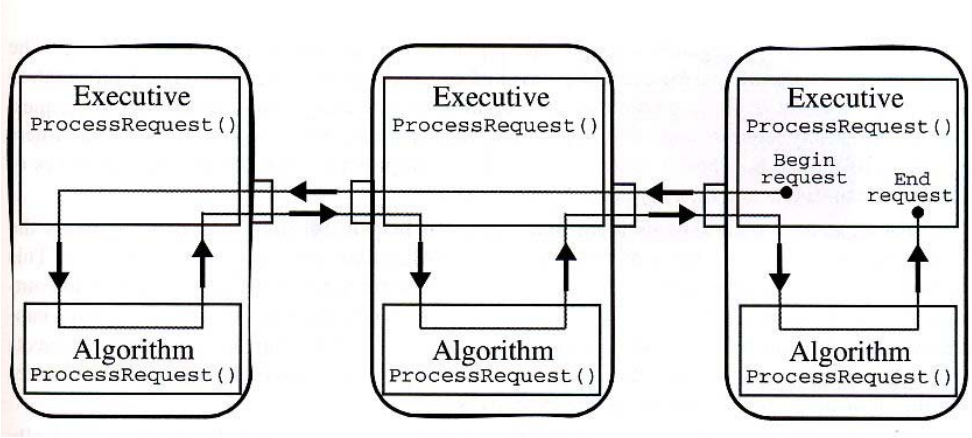
Data information 오브젝트는 vtkDataObject에 저장된 데이터에 대한 정보를 포함한다. 각각의 data 오브젝트에는 data information 오브젝트가 하나씩 존재하며, vtkDataObject::GetInformation() method로 접근할 수 있다.

Algorithm information 오브젝트는 vtkAlgorithm의 인스턴스에 대한 정보를 포함한다. Algorithm 오브젝트 하나에는 하나의 algorithm information 오브젝트가 존재하며, vtkAlgorithn::GetInformation() method로 접근 가능하다.

3) 파이프라인 실행 모델

VTK에서 파이프라인 업데이트 구조는 기본적으로 request를 기반으로 한다. request는 파이프라인을 통해 특정 정보를 전달시키는 파이프라인의 기본 동작체계로써 VTK의 실행 모델은 특정 executive에 의해 정의된 request의 집합으로 볼 수 있다.

Request는 사용자 호출에 의해 algorithm이 update를 요청함으로써 생성된다. 예를 들어 Write() method가 호출되면 algorithm 오브젝트



[그림 3-8] 파이프라인을 통해 전달되는 request의 경로

는 자신의 executive에서 파이프라인을 업데이트할 것을 요청하게 되고, this->GetExecutive->Update()를 호출함으로써 write 명령을 실행하는 것이다. 몇몇 Request는 파이프라인의 윗방향으로 전달되기도 한다.

Request는 각 필터의 executive에 의해 파이프라인을 통해 전달된다. vtkExecutive::ProcessRequest() method는 request information 오브젝트가 주어질 때 executive에서 호출된다. 이 method는 각 executive에 구현돼 있으며, request가 적절하면 그 request를 충족시키게 돼 있는데, 대부분의 필터 request는 필터에 적절한 입력을 공급하는 것으로 충족된다.

executive는 종종 algorithm 오브젝트에게 request를 수행할 것을 요청하기도 한다. 이럴 때는 vtkAlgorithm::ProcessRequest()를 호출함으로써 algorithm object에 request를 전달한다. 이 method는 request를 처리하는 루틴으로 모든 algorithm에 구현돼 있다. Input/Output pipeline information 오브젝트는 이 method에 대한 인자로 제공된다. algorithm은 자신의 필터 parameter 셋팅과 주어진 pipeline information 오브젝트만으로 request를 처리해야 하며, executive에 추가로 정보를 요청할 수 없다. 이렇게 함으로써 알고리즘은 executive에 독립적으로 동작한다. 보편적으로 request는 파이프라인의 맨 끝에 존재하는 consumer에서 발생하며, executive에 의해 파이프라인을 거슬러 올라가며 전달된다. 각 executive는 request를 처리할 것을 algorithm에 요청한다.

나. Callback 함수 설정

VTK 버전 5.0부터는 이전 버전과 callback 함수를 설정하는 방식이 달라졌다. VTK 버전 5.0부터는 기본적으로 Command/Observer 방식을 사용하는데, 모든 VTK 클래스에는 AddObserver라는 method가 있어서 callback 함수를 설정할 수 있다. 이 방식을 이용하면 사용자가 자신의 callback 함수를 손쉽게 추가할 수 있다.

Observer는 오브젝트가 호출하는 모든 이벤트를 주시하고 있다가 자신을 호출하는 이벤트가 발생하면 관련 command(혹은 callback)를 호출한다. 예를 들어, 모든 VTK 필터는 실행에 앞서 StartEvent라는 이벤트를 호출하게 돼있다. 만약 사용자가 StartEvent에 observer를 추가했다면, 필터가 실행될 때마다 observer와 관련된 command가 호출될 것이다.

오브젝트에 observer를 추가하려면 AddObserver method를 사용하면 되는데, 이 method에 추가하는 함수를 구현하는 데는 여러 가지 방식이 존재한다. vtkCommand를 직접 만들어서 추가하거나 vtkCallbackCommand 클래스로 callback 함수를 만들어서 SetCallback method로 추가하거나 하는 등의 방법을 사용할 수도 있고, 기존의 callback함수 루틴을 그대로 사용하고 싶다면 vtkOldStyleCallbackCommand 클래스를 사용할 수도 있다.

기존에는 한 오브젝트에 callback 함수를 추가하는데 [그림 3-9]와 같은 루틴을 사용했다.

```
this->RedDiskActor->SetPickMethod(RedDiskPressed, this);
 .
 .


void vtkCaveActor::RedDiskPressed(void *a)
{
  vtkCaveActor *Actor = (vtkCaveActor *)a;
  usleep(3);
  while (CAVEBUTTON2)
  {
      Actor->LocalTransform->RotateX(Actor->RotateSensitivity);
  }
  while (CAVEBUTTON1)
  {
      Actor->LocalTransform->RotateX(-(Actor->RotateSensitivity));
  }
}
```

[그림 3-9] Callback 함수 설정의 예 (VTK 5.0 이전 버전)

여기에서는 SetPickMethod 함수를 사용했는데, 이 method는 해당 Actor가 선택됐을 때 RedDiskPressed라는 함수를 this라는 인자를 가지고 호출하라는 것이다. 이 코드를 다음 각 방법을 이용해서 VTK 5.0에 맞는 방식으로 변환해보겠다.

1) vtkCommand 클래스

[그림 3-9]의 코드를 vtkCommand 클래스를 사용해서 변경한 코드는 [그림 3-10]과 같다.

RedDiskPressed라는 vtkCommand 클래스를 정의하고, 그 클래스의 Execute부분에 실제 동작 부분을 구현함으로써 callback 함수를 구현하고, 이 vtkCommand의 인스턴스를 AddObserver 함수의 인자로 설정하면 callback 함수의 구현이 끝난다.

```
RedDiskPressed *rp = RedDiskPressed::New();
this->RedDiskActor->AddObserver(vtkCommand::PickEvent, rp);
rp->Delete();
.
.
class RedDiskPressed : public vtkCommand
{
    public:
```

```
        static RedDiskPressed *New();
        {       return new RedDiskPressed; }

        virtual void Execute(vtkObject *caller, unsigned long, void *)
        {
            vtkCaveActor *Actor = (vtkCaveActor *)caller;
            usleep(3);
            while (CAVEBUTTON2)
            {
               Actor->LocalTransform->RotateX(Actor->RotateSensitivity);
            }
            .
            .
        }
}
```

[그림 3-10] vtkCommand를 이용한 callback 함수의 추가

2) vtkCallbackCommand 클래스

vtkCallbackCommand 클래스를 사용하는 경우는, 주로 일반 함수를 callback 함수로 설정하는데 유용하다. 이 때, callback 함수로 설정되는 함수는 특수한 형태로 구현돼야 하는데,

void func(vtkObject*, unsigned long eid, void* clientdata, void *calldata) 형태로 구현하면 된다. 여기에서 vtkObject는 이벤트를 호출하는 오브젝트를 나타내고, eid는 이벤트 id를, clientdata는 vtkCallbackCommand에 전달되는 인자를, 그리고 calldata는 vtkObject::InvokeEvent()가 callback함수에 전달하는 인자를 가리킨다. [그림 3-9]의 함수를 vtkCallbackCommand 클래스로 구현하면 다음과 같다.

```
vtkCallbackCommand *crd = vtkCallbackCommand::New();
crd->SetCallback(RedDiskPressed);
crd->SetCLientData((void *)this);
this->RedDiskActor->AddObserver(vtkCommand::PickEvent, crd);
crd->Delete();
.
.
 void vtkCaveActor::RedDiskPressed(vtkObject *, unsigned long, void *a, void *)
{
  vtkCaveActor *Actor = (vtkCaveActor *)a;
  usleep(3);
  while (CAVEBUTTON2)
  {
      Actor->LocalTransform->RotateX(Actor->RotateSensitivity);
```

```
  }
  while (CAVEBUTTON1)
  {
      Actor->LocalTransform->RotateX(-(Actor->RotateSensitivity));
  }
}
```

[그림 3-11] vtkCallbackCommand를 이용한 callback 함수의 추가

3) vtkOldStyleCallbackCommand 클래스

vtkOldStyleCallbackCommand 클래스는 vtkCallbackCommand 클래스와 사용법은 비슷하지만, VTK 5.0 이전 버전의 callback 루틴을 포팅하는데 보다 효과적이다. 여기에서 callback 함수는 void func(void *clientdata)의 형태를 가지는 함수다. [그림 3-12]는 vtkOldStyleCallbackCommand로 callback함수를 구현한 것이다.

```
vtkOldStyleCallbackCommand *cbc = vtkOldStyleCallbackCommand::New();
cbc->SetCallback(RedDiskPressed);
cbc->SetClientData((void *)this);
this->RedDiskActor->AddObserver(vtkCommand::PickEvent, cbc);
crd->Delete();
.
.
 void vtkCaveActor::RedDiskPressed(void *a)
{
  vtkCaveActor *Actor = (vtkCaveActor *)a;
  usleep(3);
  while (CAVEBUTTON2)
  {
      Actor->LocalTransform->RotateX(Actor->RotateSensitivity);
  }
  while (CAVEBUTTON1)
  {
      Actor->LocalTransform->RotateX(-(Actor->RotateSensitivity));
  }
}
```

[그림 3-12] vtkCallbackCommand를 이용한 callback 함수의 추가

4. VTKCave의 구조

가. vtkCaveActor

vtkCaveActor는 vtkOpenGLActor를 상속받은 클래스로 렌더링된 씬(scene) 내에 존재하는 하나의 entity를 가리키는 클래스다. 이 클래스는 vtkActor로부터 actor의 위치 및 각도와 관련된 함수를 상속받았으며, VR 환경 내에서 사용자의 상호작용을 받아서 처리하는 기능도 갖추고 있다.

vtkCaveActor에서 가능한 상호작용은 translation, rotation, scaling의 3가지다. Wand의 2번째 버튼으로는 translation, rotation, scaling의 모드를 조절할 수 있고, 각 모드에서의 실제 조작은 1번째 버튼을 누르는 동안 발생한다.

이 클래스는 vtkCave의 가장 핵심적인 클래스 중 하나인데, callback 함수에 대한 수정을 거쳐야만 VTK 5.0과 호환, 컴파일이 가능하다. 즉, 각각의 상호작용에 대해 SetPickMethod() 함수로 지정돼 있는 callback 함수를 앞의 3장에서 기술한 callback함수 지정방식으로 수정해야만 컴파일이 가능한 것이다.

예를 들어서, RedDiskActor의 경우에는 다음과 같이 수정한다.

```
변경전:
    this->RedDiskActor->SetPickMethod(RedDiskPressed, this);

변경후:
    vtkCallbackCommand *crd = vtkCallbackCommand::New();
    crd->SetCallback(RedDiskPressed);
    crd->SetClientData((void *)this);
    this->RedDiskActor->AddObserver(vtkCommand::PickEvent, crd);
    crd->Delete();
```

[그림 4-1] RedDiskActor callback 함수 설정 부분의 수정

vtkCaveActor 클래스의 rotation 및 scaling 기능은 현재 타일형 디스플레이 장치에서 제대로 동작하지 않으므로 향후 수정이 필요하다.

나. vtkCaveButton

vtkCaveButton 클래스는 위에 문자가 쓰여진 간단한 사각형 오브젝트로, 입력 장치에 의해 선택이 가능하다. 따라서 메뉴를 구성하거나 상호작용을 받는 버튼을 구성하는데 사용할 수 있으며, 따라서 버튼에 쓰일 텍스트를 지정한다거나 상호작용에 대한 callback 함수를 지정하는 method를 주로 제공한다.

이 클래스에서 callback함수를 설정하는 부분 역시 VTK5.0에서 지원하는 형태로 변경해야 한다. 새로 변경된 callback함수 설정 부분은 다음과 같다.

```
변경전:
    this->ButtonActor->SetPickMethod(f, arg);

변경후:
    vtkOldStyleCallbackCommand *cbc = vtkOldStyleCallbackCommand::New();
    cbc->SetCallback(f);
    cbc->SetClientData(arg);
    this->ButtonActor->AddObserver(vtkCommand::PickEvent, cbc);
    cbc->Delete();
```

[그림 4-2] vtkCaveButton의 callback 함수 설정 부분의 수정

vtkCaveButton 클래스에서는 SetName(), SetCallBack(), void *), GetActors(), SetPosition(), SetSize()같은 함수를 지원한다.

다. vtkCaveCamera

vtkCaveCamera 클래스는 3D 렌더링을 위한 가상 카메라로 view point와 pocal point의 위치와 각도를 조절하는 여러 함수를 제공한다. 이 클래스 역시 VTK 5.0에서는 몇가지 수정을 거쳐야 컴파일이 가능한데, 이는 VTK 5.0의 함수들의 리턴 타입이 대부분 float형에서 double형으로 바뀌었기 때문이다. 따라서 vtkCaveCamera 클래스에서 사용된 GetOrientation() 함수나 GetOrientationWXYZ() 함수의 리턴 타입을 float에서 double형으로 변환하면 컴파일이 가능하다. 그러나 VTK의 Active Camera와의 충돌 문제로 실제 사용은 불가능하다.

라. vtkCaveCellPicker

vtkCaveCellPicker 클래스는 그래픽스 윈도우로 ray를 쏴서 actor가 정의한 geometry와 교차하는 cell 하나를 선택하는데 사용되는 클래

스다. 이 클래스는 교차하는 cell의 좌표 대신, ray와 가장 가까운 곳에 위치하는 cell의 id를 리턴한다.

VTK 5.0으로 컴파일하기 위해서는 IntersectWithLine 함수에 사용된 인자를 모두 float형에서 double형으로 변경해야 한다. 따라서 이 함수의 해당 인자를 모두 double형으로 바꿔주면 문제없이 컴파일이 가능하다.

마. vtkCaveInteractorStyle

이 클래스는 VTK의 vtkInteractorStyleUser 클래스를 상속한 클래스로 주로 이벤트와 관련 인터페이스를 제공하면서 대부분의 움직임을 제어하는 루틴을 제공한다. 이 클래스는 플랫폼에 독립적인 wand/key/마우스 루틴 및 timer 컨트롤을 지원하는 vtkCaveRenderWindowInteractor에 대한 Interactor Style을 제공한다.

vtkCaveInteractorStyle에서는 'wand' 스타일의 상호작용을 제공한다. 즉, 사용자가 wand 키를 누르고 있으면 이벤트 스트림이 발생함으로써 rotate, translate, pan, zoom같은 액션이 계속 실행된다는 것이다. 이 클래스에 구현돼 있는 이벤트 동작은 다음과 같다.

- 버튼 3: 카메라 조작 모드를 변경하는데 사용된다.
  - 모드 1: Navigate 모드. Navigation에 필요한 계산을 수행한다. 이 모드에서는 조이스틱의 상태를 체크해서 움직임과 회전을 결정짓는다. 조이스틱 Y 방향의 움직임은 wand 방향에 대한 움직임의 속도를 조절하고 X 방향의 움직임은 CAVE의 Y축에 대한 회전 속도를 결정짓는다. 조이스틱의 움직임에 대한 dead zone은 −0.15~0.15이며, 이 사이의 움직임은 모두 무시된다.
  - 모드 2: 선택된 오브젝트를 중심으로 카메라를 회전(rotate)한다.
  - 모드 3: 아직 구현되지 않은 기능으로 카메라를 프로젝션 방향으로 회전(rotate)한다. 벡터 방향은 선택된 오브젝트 중심으로의 프로젝션 방향이다.
- q/Esc 키: 애플리케이션을 빠져나간다.
- l 키: pick 동작을 수행한다. vtkRenderWindowInteractor에는 vtkCellPick

er의 인스턴스가 있어서 pick 동작을 수행할 수 있다.

- s 키: 모든 actor를 surface모드로 표현한다.
- u 키: 사용자 정의 함수를 호출하는 키로, 이 키를 누르면 사용자가 명령을 입력할 수 있는 interactor가 나타나게 된다.
- e 키: 모든 actor를 wireframe 모드로 표현한다.

vtkCaveInteractorStyle 클래스에서 subclass를 구현하면 새로운 interaction style을 구현할 수 있으며, 기존에 제공되던 interaction을 override해서 새로운 동작을 구현할 수도 있다.

이 클래스 역시 VTK 5.0으로 컴파일하려면 몇몇 float형의 데이터를 double형으로 변경해야 한다.

바. vtkCaveMenu

vtkCaveMenu 클래스는 3차원 메뉴를 보다 손쉽게 만들 수 있게 위한 목적으로 제작된 클래스로, 버튼, 텍스트 및 음영 효과를 포함하고 있다. 대부분의 사용자는 메뉴 버튼을 선택하고 그 버튼에 대해 애플리케이션이 동작하는 방식에 익숙해져 있기 때문에, vtkCaveMenu 역시 이와 같은 방식의 메뉴 구성을 따른다. 메뉴는 메뉴를 구성하는 여러 개의 개별 아이템으로 구성되며, 각 아이템은 기능을 제공한다. 메뉴 요소의 상태는 active 상태와 inactive 상태가 모두 가능하며, 메뉴의 기능은 callback 형태로 정의할 수 있다.

메뉴에 구성요소를 추가하는 방법은 Add와 AddToRow, 이 2가지가 있다. Add는 새 줄을 생성하고 구성요소를 추가하는 반면, AddToRow는 구성요소를 기존 줄의 맨 오른쪽에 추가하는 방식이다. 각 구성요소의 너비와 높이는 메뉴 전체의 너비와 높이, 한 줄당 구성요소의 개수, 메뉴 내에 존재하는 줄의 개수 등에 따라 조절된다.

이 클래스에는 메뉴의 제목을 설정하거나 메뉴 구성 요소를 추가하는 등의 기능이 구현돼 있다.

VTK 5.0에서는 vtkPlaneSource의 SetPoint1(), SetPoint2() 함수에 사용되는 인자가 float형에서 double형으로 변경됐다. 따라서 vtkCaveMenu::SetSize() 함수 내에 사용된 SetPoint1, SetPoint2 함수의 인자를 모두 double형으로 변경한 뒤 컴파일해야 컴파일이 가능하다.

사. vtkCaveMenuElement

vtkCaveMenuElement 클래스는 모든 메뉴 구성요소에 대한 abstract 클래스다. 이 클래스는 메뉴 구성요소에 대한 최소한의 API를 제공한다. 따라서 이 클래스에서는 메뉴 구성요소의 x, y, z 좌표, 메뉴 구성요소의 크기, 구성요소의 경계값을 리턴하는 함수를 제공한다.

아. vtkCavePicker

vtkCavePicker 클래스는 picker에 대한 슈퍼클래스로, 그래픽스 윈도우로 ray를 쏴서 actor의 바운딩 박스와의 교차여부를 체크함으로써 vtkProp3D의 인스턴스를 선택하는데 사용되는 클래스다. Picker에 사용되는 ray는 윈도우 좌표계로 정의되며, 카메라 위치로부터 시작한다. 이 클래스는 교차하는 바운딩 박스를 하나 이상 찾아내서 여러개의 vtkProp3D를 리턴할 수도 있는데, 주로 교차되는 vtkProp3D의 목록, world 좌표계에서의 pick 좌표 및 카메라와 가장 가까운 위치에 있는 vtkProp3D와 mapper를 리턴한다. 이 때, 가장 가까운 위치에 있는 vtkProp3D란 ray에 프로젝션된 중심점이 카메라와 가장 가까운 거리에 있는 것을 가리킨다. 이 클래스는 신속한 geometry picking에 사용되며, point나 cell을 picking하는데는 vtkCavePointPicker 클래스나 vtkCaveCellPicker 클래스를 사용하면 된다.

VTK 5.0으로 이 클래스를 컴파일하기 위해서는 몇가지 데이터형을 float에서 double형으로 재선언해야 한다. VTK 5.0에서 사용하는 대부분의 데이터형이 double형으로 변경된 데 반해 CAVELib에서 사용하는 데이터는 float형이기 때문이다. 또, VTK의 이전 버전에 존재했던 vtkCell::HitBBox 함수가 5.0버전에서는 없어졌으므로, 이 함수를 코드에 추가해야 한다. 이 클래스에 추가한 HitBBox 함수의 코드는 다음과 같다.

```
#define VTK_RIGHT 0
#define VTK_LEFT 1
#define VTK_MIDDLE 2
```

```
char HitBBox(double bounds[6], double origin[3], double dir[3],
             double coord[3], double& t)
{
    char inside = 1;
    char quadrant[3];
    int  i, whichPlane=0;
    double        maxT[3], candidatePlane[3];

    //First find closest planes

    for (i=0; i<3; i++)
    {
        if (origin[i] < bounds[2*i])
        {
            quadrant[i] = VTK_LEFT;
            candidatePlane[i] = bounds[2*i];
            inside = 0;
        }
        else if (origin[i] > bounds[2*i+1])
        {
            quadrant[i] = VTK_RIGHT;
            candidatePlane[i] = bounds[2*i+1];
            inside = 0;
        }
        else
        {
             quadrant[i] = VTK_MIDDLE;
        }
    }

    // Check whether origin of ray is inside bbox
    if (inside)
    {
        coord[0] = origin[0];
        coord[1] = origin[1];
        coord[2] = origin[2];
        t = 0;
        return 1;
    }
    // Calculate parametric distances to plane
    for (i=0; i<3; i++)
    {
        if (quadrant[i] != VTK_MIDDLE && dir[i] != 0.0)
        {
            maxT[i] = (candidatePlane[i]-origin[i]) / dir[i];
        }
        else
        {
            maxT[i] = -1.0;
        }
    }
```

```
    // Find the largest parametric value of intersection
    for (i=0; i<3; i++)
    {
       if (maxT[whichPlane] < maxT[i])
       {
          whichPlane = i;
       }
    }

    // Check for valid intersection along line
    if (maxT[whichPlane] > 1.0 || maxT[whichPlane] < 0.0)
    {
         return 0;
    }
    else
    {
         t = maxT[whichPlane];
    }
    // Intersection point along line is okay. Check bbox.
    for (i=0; i<3; i++)
    {
       if (whichPlane != i)
       {
          coord[i] = origin[i] + maxT[whichPlane]*dir[i];
          if (coord[i] < bounds[2*i] || coord[i] > bounds[2*i+1])
          {
              return 0;
          }
       }
       else
       {
         coord[i] = candidatePlane[i];
       }
    }

    return 1;
}
```

[그림 4-3] 함수 HitBBox

자. vtkCavePointPicker

vtkCavePointPicker 클래스는 그래픽스 윈도우로 레이를 쏴서 actor가 정의한 geometry와 교차되는 지점, 특히 한 점(point)을 선택하는데 사용되는 클래스다. 이 클래스는 선택 지점의 좌표와 actor, 그리고 mapper를 리턴하는 대신 ray의 오차범위 내에서 가장 가까운 point의 id를 리턴한다.

VTK 5.0에서는 대부분의 float형 변수가 double형으로 변경됐기 때문에, 몇몇 변수의 형태를 double형으로 변경하면 VTK 5.0으로 컴파일

할 수 있다.

차. vtkCaveRenderer

vtkCaveRenderer 클래스는 vtkOpenGLRenderer 클래스를 다시 재구성한 하위 클래스로 OpenGL 그래픽스 라이브러리에 대한 인터페이스다.

카. vtkCaveRenderWindow

vtkCaveRenderWindow 클래스는 vtkXOpenGLRenderWindow에 대한 하위 클래스로 OpenGL 그래픽스 라이브러리에 대한 인터페이스를 제공한다. 프로그래머는 보통 이 클래스를 사용하며, AddRenderer 메쏘드로 vtkCaveRenderer를 추가할 수 있다.

타. vtkCaveRenderWindowInteractor

vtkCaveRenderWindowInteractor 클래스는 플랫품에 독립적인 interaction을 제공하는 클래스로, 마우스/키보드/시간에 대한 이벤트 메시지를 vtkInteractorStyle 클래스에 전달하는 베이스 클래스로써의 역할을 수행한다. 또, 이 클래스는 picking, 렌더링 프레임 레이트 등에 관한 컨트롤도 제공한다.

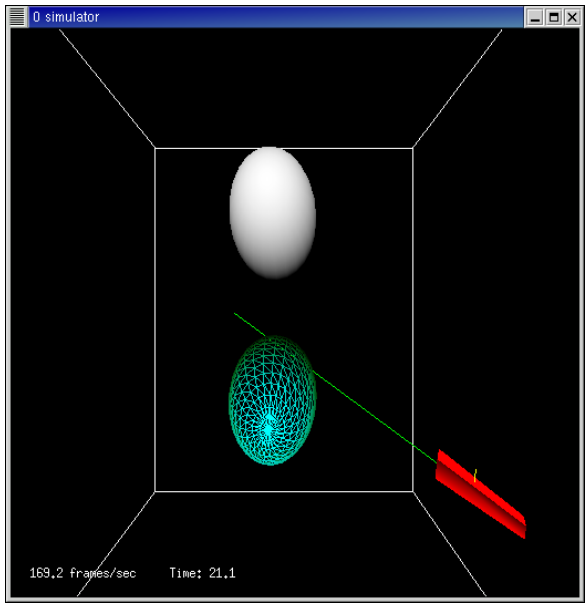
파. vtkCaveWand

vtkCaveWand 클래스는 CAVE 애플리케이션에서 wand의 각도(orientation)를 조절하는 루틴을 제공하는데, 거의 사용되지 않는 클래스다.

5. 예제

가. Callbacks

Callback 함수의 사용법을 보여주기 위한 간단한 예제 프로그램으로, 1번 버튼을 누르면 위쪽의 흰 공이 point로 표현되고 2번 버튼을 누르면 wireframe으로, 그리고 3번 버튼을 누르면 [그림 5-1]과 같이 surface로 표현된다.

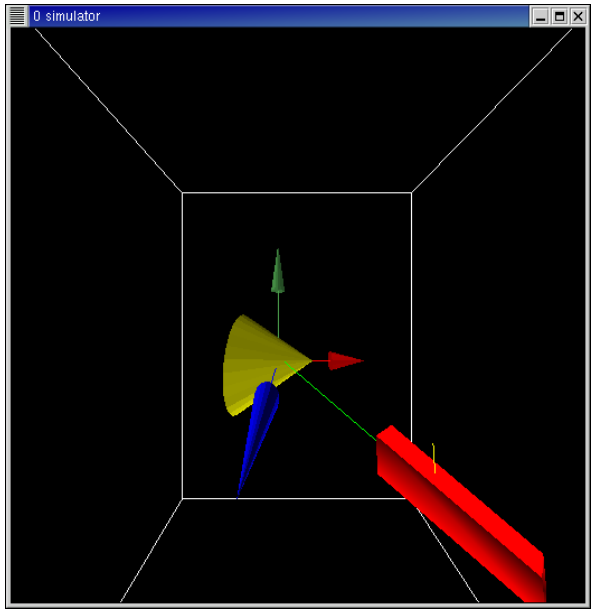


[그림 5-1] Callbacks 예제

이 예제를 컴파일하기 위해서는 각 버튼에 대한 callback 함수를 vtkCallbackCommand로 선언, AddObserver 함수를 사용해서 VTK 5.0 스타일의 callback 함수로 변환해야 한다.

나. CaveInteractor

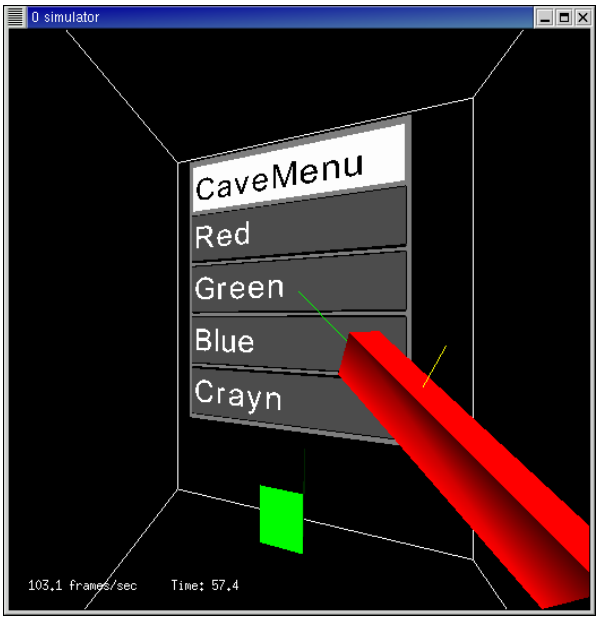
vtkCaveActor 클래스를 사용하는 간단한 예제로, vtkCaveActor의 기능을 시험할 수 있다. vtkCaveActor를 사용하는 또다른 예제는 CaveActor에서 찾아볼 수 있다.



[그림 5-2] CaveInteractor 예제

다. CaveMenu

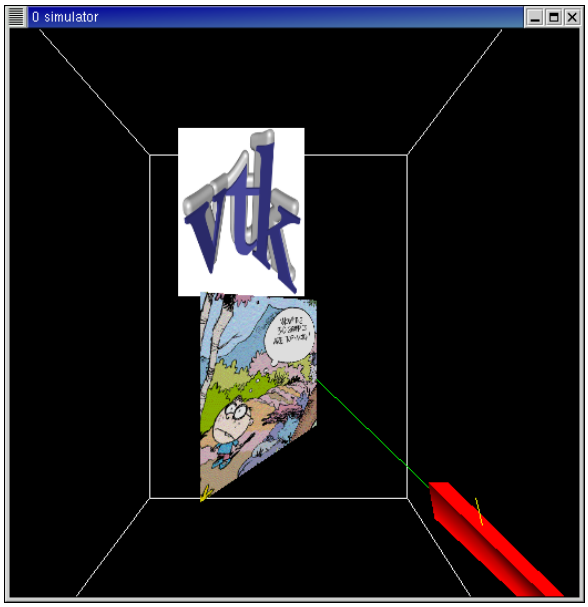
Cave 환경에서 메뉴를 생성하는 방법을 보여주는 예제다.



[그림 5-3] CaveMenu 예제

라. Texture

Cave 환경에서 이미지 파일을 읽어서 보여주는 예제다.



[그림 5-4] Texture 예제

6. 결론

VTKCave는 VTK와 CAVELib을 이용한 라이브러리로, VTK를 이용해 렌더링한 결과를 CAVE 환경에서 볼 수 있게 해주는 유용한 라이브러리다. 그러나 오래전에 개발이 중단된 상태이기 때문에 VTK의 새 version을 제대로 지원하지 못한다는 단점이 있다. 또, 가장 중요한 역할을 담당하는 클래스중 하나인 vtkCaveActor 클래스의 기능이 완벽하게 구현돼 있지 않았었다. 따라서 VTKCave 라이브러리를 사용하기 위해서는 우선 VTK 5.0과의 호환성 문제를 해결한 뒤, vtkCaveActor 클래스의 기능을 보완해야만 한다. 이런 문제점을 해결하면 VTKCave 라이브러리는 VTK의 결과물을 가상 환경에서 구현하는데 아주 유용한 툴을 제공할 것이다.