Double Raschel 인수인계

작성일 : 12. 28

소　속 : 개발부

작성자 : 신성주

Index

1. Basement

Double Raschel은 영우 CNI에서 융복합기술개발 국책과제로써 진행하는 프로젝트다.
Tricot로 이루어진 서로 다른 두 원단을 앞 · 뒷면으로 서로 연결하여 완성된 것이 더블라셀이라 한다. 더블라셀은 크게 앞면을 설계하는 Front Bar, 뒷면을 구성하는 Back Bar, 앞 · 뒷면을 이어주는 Pile Bar로 구성된다. Tricot에서는 가장 많게는 7Bar로 원단을 구성한다. Double Raschel 역시 최대 7Bar를 통해서 설계를 할 수 있다. (필자도 7Bar로 확신하진 못한다. 연구기간 동안 방문한 공장에서는 기계 설계의 한계상 7개의 Bar가 현재로썬 최대라도 들었다.)

최소 3개, 최대 7개의 Bar를 Front, Back, Pile에 분배하여 설계한다. 3가지 분류에 하나이상의 Bar가 할당되지 않으면 절대 안된다. (그래서 최소의 Bar가 3개이다)

Front, Back Bar는 자신의 면만 구성하므로 2차원적인 움직임(x, y 축)을 가지고 있다. 그러나 Pile은 Front면과 Back면을 이어야 하므로 3차원적인 움직임(x, y, z 축)을 가진다. Front, Back과 Pile은 움직임이 다르나 설계는 같은 방식으로 하되 Pile은 Front와 Back에 관여하므로 Pile Bar 하나에 Front와 Back에 작용하는 설계를 해야한다. 즉, Front : 1, Back : 1, Pile : 1 로 각각 Bar를 할당하면 설계는 Front 1번, Back 1번, Pile 2번(Front, Back) 총 4번의 설계를 해야한다.

Front, Back, Pile이 실제 공장에서 움직이는 방식이 다를 수 있으나 설계 방식은 같기 때문에 기존의 자사 프로그램 Textricot의 설계방식을 이용하여 Double Raschel을 설계하도록 하였다.
때문에 Double Rashel 프로그램은 Textricot의 확장버전으로 받아들일 수 있다.

A. 개발에 사용된 언어

C Builder 6.0이라는 컴파일러를 통해 C++로 제작된 Tricot와는 달리 유지보수와 UI개선을 위해 Visual Studio를 통해 Winform으로 제작하였다. Tricot는 유저의 입력에 따라 실시간으로 연산해야 하는 무거운 프로그램이기에 C#의 연산속도로 Tricot를 그대로 구현하는 것이 어렵다 판단되었다. 때문에 C#의 편의성과 C++의 속도를 모두 사용할 수 있는 C++/CLI를 도입하게 되었다.

Design 기능을 구현하기 위해선 Bitmap을 이용한 표현 방식이 충분하다. 그러나 시뮬레이션을 구현하는데 있어서는 Bitmap을 통해 그리기를 사용한 TexTricot는 부적절한 점이 있었다. 그것은 앞서 Pile Bar를 언급함과 같이 3차원의 움직임을 표현해야 한다는 것, 3차원으로 움직이는 Pile Bar로 인해 원단이 가지는 두께감을 표현해야 한다는 것, 두께감을 위해 Lighting(Shadow) 표현을 줘야 한다는 것, 실시간으로 원단을 돌려 볼 수 있어야 한다는 것으로 인해 기존의 Drawing이 적절하지 않다는 판단을 하게 되었다. 그래서 GPU를 이용한 Drawing을 사용하게 되었고 이에 사용된 Graphic API가 DirectX 11이다. 그런데 DirectX 11은 C++ Base 언어이기 때문에 래핑하는 과정이 필요했고 그 작업을 해주는 라이브러리가 SharpDX이다.

i. C++ / CLI

직접 C++/ CLI를 설명하기보단 더 좋은 레퍼런스를 참고하길 바란다. 회사 공유 폴더에 "Native C++ 개발자를 위한 C++/CLI"라는 문서로 공유하도록 한다.

ii. DirectX 11, SharpDX

SharpDX는 DirectX 11을 래핑하여 C#에서 사용할 수 있도록 하는 라이브러리다. DirectX 11의 렌더링 파이프라인은 반드시 숙지하길 바란다. 이를 위해 Frank D.Luna가

저술한 DirectX 11 레퍼런스를 정독하길 바란다.

3D API에 대한 개념과 렌더링 파이프라인에 익숙하다면 SharpDX는 쉽게 따라올 것이다. 관련된 국내 서적은 없으나 함수의 기능이나 핵심적인 리소스들은 그대로 반영하고 있기 때문에 구글의 검색이 SharpDX에 적응하는데 도움을 줄 것이다.

2. 개발 의도 및 방향

3. 개발 현황

A. 1차 년도

현재 2차년도의 개발에 사용되는 SharpDX가 아닌 SlimDX로 제작되었다. 필자의 전임 근무자(김태훈)의 개발을 이어받았다. 전임 개발자가 C++/CLI를 이용하여 Listener, Action History, Design Panel 등 큰 틀을 완성하였다. 필자는 여기에 Pile Bar의 개념을 추가 그에 따른 자료구조 정리 및 창 구성 변경, TexTricot의 시뮬레이션을 재구성하여 현재 프로그램에 옮겨 놓았다. (C++/CLI의 의도에 맞게 시뮬레이션 모듈은 Native에 구현해 놓았다)

TexTricot의 시뮬레이션을 옮겨온 의도는 Tricot의 시뮬레이션에서 반영된 원단 느낌을 그대로 가져오기 위함이다. 또한 개발에 주어진 시간상 새로운 시뮬레이션을 개발하는데 난항이 있을 것으로 예상되기 때문이었다.

현재 구현되 Simulation 방식은 기존 시뮬레이션 데이터를 3D Grid에 붙여서 앞면 뒷면을 Blending 효과로 보여주는 것이다. 즉 기존 시뮬레이션의 모듈에서 생성한 두장의 One Repeat 이미지로 DirectX에서 사용할 수 있는 Resource로 변경한 뒤 앞면 뒷면 그리드에 텍스쳐 매핑을 한 것이다.

시뮬레이션을 위해 필요한 메쉬는 많은 Vertex로 이루어진 Grid 2개이다. 2개의 물체중에 카메라에서 멀리 있는 물체를 먼저 그리고 가까이 있는 물체를 나중에 그리면서 Alpha Blending, Alpha Test 효과를 이용한다.

가장 단순한 기하 모델에 텍스쳐 매핑을 한 시뮬레이션을 구현 한 것이 1차년도 프로그램이다.

B. 2차 년도 개발 현황

2차 년도의 개발의 요는 시뮬레이션의 Full 3D화다. TexTricot의 시뮬레이션에서 실을 표현한 원통 모양의 텍스쳐는 실이 향하는 벡터에서 그려질 픽셀의 위치를 거리로 환산하여 광택을 주면서 표현하고 있다. 이 원통 부분을 3차원 Vertex로 이루어진 Cylinder로 바꾸는게 목표였다.

현재 시뮬레이션의 알고리즘은 물리 공식에 따라 실의 방향을 결정해주나 여러 Bar에 의해 서로 얽혀 있는 부분을 계산하지 못한다(이 부분이 위에서 이야기했던 개발시간이 오래 걸리는 부분이다). 그러나 Bottom, Low, Mid, High, Front로 총 5단계의 실의 가상의 z축 위치를 구함으로 실의 얽힘을 표현해주고 있다. 부드럽게 Cylinder를 연결시켜 표현하지는 못하나 원단의 복잡한 얽힘의 느낌을 충분히 줄 수 있을 것으로 예상한다.

1차년도에서 사용하였던 SlimDX라는 라이브러리는 여러 하드웨어 환경에 호환이 되지 않는 현상이 발생하였고 호환이 된다고 하여도 개발하는데 세팅이 반드시 필요한 불편함이

있었다. 그래서 SlimDX라는 라이브러리보단 Visual Studio의 Nuget을 통해 쉽게 설치 할 수 있고 DLL을 통째로 프로젝트에 자동으로 복사해주는 SharpDX로 교체하기로 하였다. (실제로 마이크로 소프트에서도 SlimDX보단 SharpDX를 더 지원해주는 듯 하다)

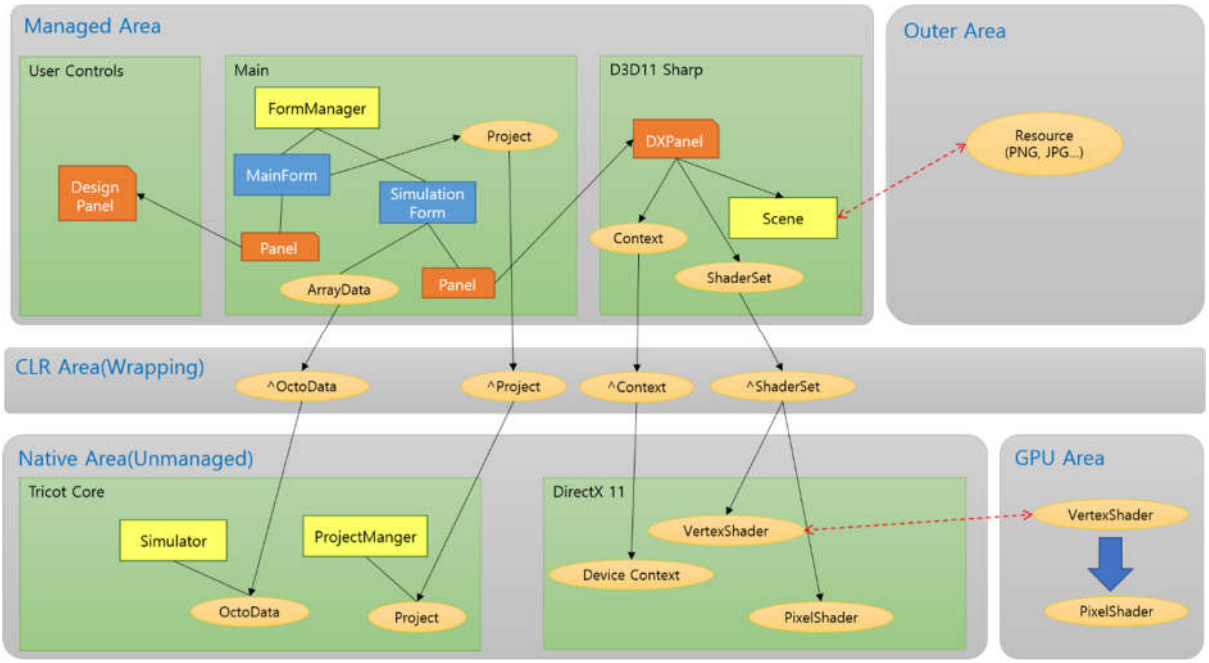
현재까지는 SharpDX를 이용한 Rendering 부분의 설계를 마쳤고 구현 중에 있다. 설계에 대한 부분은 뒤에서 설명하도록 한다.

∵ 설계를 다시 할 수 밖에 없었던 것은 1차년도의 개발과정이 짧았기 때문에 30% 이상을 하드코딩으로 구현하였다. 더 이상... 손 댈 수가 없기 때문이다.

4. 프로그램 설계도

A. Native / Managed, Area

C++ / CLI로 이루어진 Managed, Unmanaged, DirectX 영역의 상관관계를 표현한 것이다. 1차년도에 작성된 것이긴 하지만 2차년도의 큰 설계는 여기에서 벗어나지 않는다.



구조도는 크게 5개 영역으로 나뉘어져 있다.

① Winform(C#)의 Managed Area

② GC(Garbage Collector)가 아닌 메모리에 직접 접근하여 많은 양의 계산을 담당해주는 Native(Unmanaged) Area

③ Native Code를 Managed Area에 Wrapping하여 올려주는 CLR Area,

④ Shader Code를 통해 Swap Buffer에 Drawing 역할을 담당하는 GPU Area(꼭 그리기만 이 아닌 Compute Shader를 통해 빠른 계산 결과를 받을 수도 있다.)

⑤ 그림, INI 파일 등을 저장하는 Outer Area.

5가지 중에서 1, 2, 3번의 영역에 설명을 더하고자 한다.

i. Managed Area

프로그램의 Entry Point와 프로그램에 포함된 모든 Form이 여기에서 작성되어 있다. 일반적으로 Winform의 프로젝트를 시작하면 자동으로 Program.cs에 Main() 함수 내용이 자동으로 작성이 된다. 그러나 현재 프로젝트에선 C# 영역에 있는 SharpDX와의 이벤

트 관계가 필요하므로 다음과 같이 변경하였다.

```
5.  namespace TexTricot
6.  {
7.      static class Program
8.      {
9.          /// <summary>
10.         /// The main entry point for the application.
11.         /// </summary>
12.         [STAThread]
13.         static void Main()
14.         {
15.             DirectXSharp.D3DApp.CreateInst();
16.             DirectXSharp.D3DApp.Inst.Init();
17.
18.
19.             Application.EnableVisualStyles();
20.             Application.SetCompatibleTextRenderingDefault(false);
21.
22.
23.             MainForm main = new Forms.MainForm();
24.             main.m_MyWndProc += DirectXSharp.D3DApp.Inst.WndProc;
25.
26.             main.Show();
27.             main.Update();
28.             main.Run();
29.             DirectXSharp.D3DApp.Inst.Dispose();
30.
31.             //Application.Run(new Forms.MainForm());
32.         }
33.     }
34. }
```

코드 내용을 보면 MainForm을 직접 생성하고 show(), Update(), Run()을 통해 직접 실행하는 부분을 볼 수 있고

main.m_MyWndProc+= DirectXSharp.D3DApp.Inst.WndProc;

과 같이 DirectX를 총괄하는 Procedure 함수를 main의 Procedure에 등록하는 과정을 볼 수 있다. 이는 Winform에 포함된 DXPanel(User Control 패널)에 그래픽을 표현하기 위해선 반드시 등록이 되어야한다.

모든 초기화가 마치면 main의 run()을 통해 종료신호가 발생할 때까지 무한 루프에 들어간다. 다음이 그 과정이다.

```
public void Run()
 {
     while(DirectXSharp.D3DApp.Inst.m_isRunning)
     {
         DirectXSharp.Timer.Inst.TickTimer();
         Application.DoEvents();
         DirectXSharp.D3DApp.Inst.ProgessAllPanel((float)DirectXSharp.Timer.Inst.DeltaTime
);
         DirectXSharp.D3DApp.Inst.RenderAllpanel();
```

```
    }
    Dispose();

}
```

많은 Form이 있지만 충분히 구성된 UI나 프로그램이 동작하는 것으로 어떻게 구현되어 있는지 추측할 수 있기에 Form에 대한 일일이 설명하는 것은 생략하도록 한다. DXPanel이 등록된 Panel에 대해만 설명하겠다

DXPanel을 포함한 Form이 생성될 시 DXPanel의 핸들을 DirectX를 관리하는 영역에 등록해야한다.

```
public SimulationForm()
{
    InitializeComponent();
    InitializeListener();
    UpdateComponentAll();
}
public void UpdateComponentAll()
{
     m_SceneMNG = new SceneMNG();
     m_SceneMNG.Init(m_DxPanel_Index);
     m_BackColor = SystemColors.ControlDark;
     D3DApp.Inst.RegisterControl(d3D11Control1, m_DxPanel_Index, m_BackColor);
     m_SceneMNG.SetComponent(new Scene_Simulation_Loading(m_DxPanel_Index, null), d3D11Co
ntrol1.Size.Width, d3D11Control1.Size.Height);

     UpdateComponentProjectSetting();
     m_fSimWidth = ProjectManager.Inst.Project.SimEnvironment.GetWidth();


}
```

D3DApp.Inst.RegisterControl(d3D11Control1, m_DxPanel_Index, m_BackColor);

위의 함수가 바로 그 함수이다.

Double Raschel에서 DirectX를 사용하는 부분은 3군데이다. Simulation Form의 패널, Color Form의 패널, Yarn Maker의 패널이다. 때에 따라 동시에 DirectX가 사용된 창이 활성화 될 수 있기 때문에 Multi Rendering이 필요하다. 그래서 DirectX 관리자 클래스는 여러 패널을 등록하고 사용할 수 있도록 특별히 구조체로 관리하고 있다.

```
public class ViewportSet
{
    public SwapChain              m_SwapChain;
    public RenderTargetView       m_RenderTargetView;
    public Viewport               m_ViewPort;
    public Size                   m_ClientSize;
    public UserControl            m_UserControl;
    public RenderMNG               m_RenderManager;
    public ObjectMNG               m_ObjectManager;
    public Color                m_BackColor;

}
```

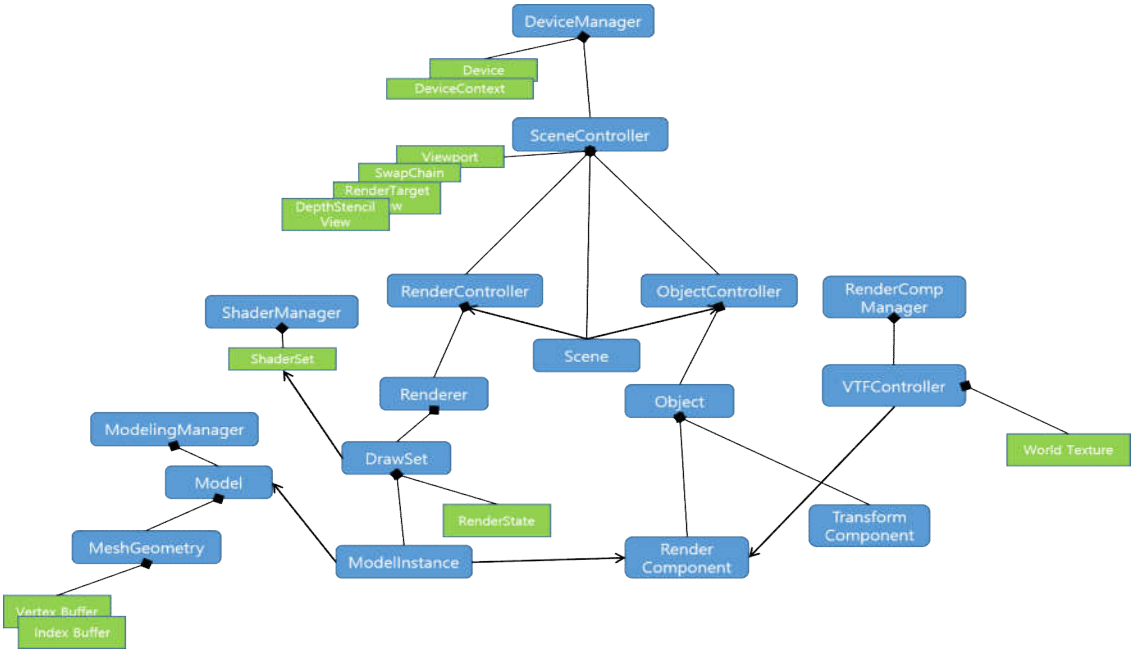
이 구조체에서 m_UserControl이 Panel을 등록하는 과정에서 불렀던 함수의 인자로 전달된다.
위와 같은 구조체를 List로 관리하고 있는데 DXPanel을 사용하는 Form마다 enum으로 미리 지정된 Index(m_DxPanel_Index)를 가지고 있다. 이를 통해 Clear Buffer Color를 변경하거나 Resize 이벤트가 발생할 때 Panel을 Resize할 수 있도록 한다.
DXPanel이 기능이 DirectX Rendering을 보여주는 역할이므로 Form에서 특별히 컨트롤할 부분은 없다.
DirectX가 설계된 부분은 설명을 생략하기로 한다. 위에서 언급했듯이 1차년도에 구현된 DXPanel은 유지보수하기 어려워 SharpDX로 대체할 예정이므로 2차년도를 위해 설계된 부분을 설명하는 것으로 대신하겠다.

i. Native Area
Guide Bar, Simualation Environment Data, Yarn Data 등 Project에 관련된 모든 핵심 자료들이 여기에서 관리된다.
Simulation Data를 생산하는 핵심 모듈도 여기에 구현되어 있다.
Octopus
실이 꼬일 때 매듭이 표현되는 모양을 통해서 네이밍이 된 자료형이다. Head의 넓이 들어오는 실과 나가는 실의 방향의 데이터를 적재할 수 있다.
Movement
Octopus의 본래 위치를 가지고 있으며 다른 코스(Course)나 다른 실들에 의해 변경된 위치를 가지고 있다.
OctopusSet
Front, Pile, Back 각각 Simulation을 위해 가진 구조체이며 Octopus의 데이터를 적재한다.
TricotSimulator
Project의 모든 Guide bar의 Course Data를 조회하여 Octopus 데이터로 변경하며 OctopusSet을 생성한다.
SimulationPainter
One Repeat을 표현할 이미지의 크기만큼 확보된 2차원 배열에 OctopuSet의 자료를 기초하여 Pixel Data를 만들어낸다.

ii. CLR Area
Project Data 와 Simulation의 결과를 Managed Area에 전달하는 역할을 한다. Native의 Project와 Simulation Data에 접근하기 위해 TCProject, TCSimulator를 가지고 Form에서 직접 컨트롤한다.

A. 3D Structure based on SharpDX

(프로젝트 위치 E:₩Source₩TestEngine)
1차년도에 설계된 SlimDX를 대체할 SharpDX를 이용한 2차년도 DirectX Rendering 부분의 설계이다. (Native, Managed Area에 대한 설계는 윗부분과 동일하다)



각 클래스에 대해 설명을 이어가도록 하겠다.

- DeviceManager

  DirectX 11에서 사용하는 Device와 DeviceContext를 생성 및 관리하는 클래스다. DXPanel이 Form에서 등록된다면 여기에 Register되며 하위 SceneContoller에 전달된다.

- SceneController

  DXPanel의 Handle을 관리하며 Scene을 관리하는 클래스이다. (Scene이란 장면을 컴포넌트화 시킨 것이라 봐도 무방하다) 패널을 Clear하고 Swap Chain하는 클래스이므로 Viewport, SwapChain, RenderTargetView, DepthStencilView가 여기에서 생성 · 관리된다.

- RenderController

  Renderer를 관리하는 클래스이다. 그리기 모듈의 가장 최상위 관리클래스이다. 활성화된 Scene에 참조되며 Scene에서 직접 컨트롤한다.

- ObjectController

  Visualize한 객체, 그렇지 않은 객체(Camera 객체가 여기 속한다)를 모두 생성 · 관리해준다.

- Renderer

  Renderer는 Visualize한 객체를 그리기 분류에 따라 분류하는 컨테이너 역할과 동시에 Shader Setting의 횟수를 최소화하기 위해 ShaderSet(Vertex, Pixel, 등등 Shader의 리소스를 보관)에 따라 물체를 분류하여 주는 역할을 한다. 그리기 분류이란 애니메이션 정보의 유무, Particle Effect, Dynamic Vertex 등 객체를 그리는데 있어 공통 필수조건들에 의의 분류한 것을 말한다.

- DrawSet

  ShaderSet을 ShaderManager에서 참조하여 관리하고 ShaderSet에 맞게 DepthStencil

State, SamplerState, RasterizerState의 정보를 보관하여 DrawCall 때 세팅하여 그리기 명령을 ModelInstance들에게 할당한다.

- ModelInstance
  ModelingManager에서 특정 Model 클래스를 참조하여 Visualize한 객체들 중에 같은 Mesh로 분류된 객체들의 RenderComponent를 등록 · 관리한다.

- RenderCompManager
  Draw Instancing을 고려하여 만든 클래스이다. 많은 객체가 Scene에 등록 될 시 World Matrix의 정보를 담을 수 있는 Raw Texture를 이용하여 빠른 그리기를 하기 위함이다. 자세한 사항은 VTF Instancing의 자료를 검색해 보길 권한다.

5. 2차년도 개발 방향

EAT의 Procad와 같은 정밀한 물리연산으로 원단을 구성하는 시뮬레이션은 현재 여건상 어렵다 판단된다. 적어도 전담팀이 꾸려져 년 단위의 연구가 집중적으로 이루어져야 한다고 예상된다. 실의 꼬이는 방식에 대해 실제 원단 업체의 어느 전문가도 알지 못하는 상황이니 더욱 어렵다.

때문에 국책과제 수준을 맞추기 위해 Tricot의 알고리즘을 이식한 Double Raschel의 시뮬레이션 모듈을 사용할 수 밖에 없다. 위에서 언급하였지만 2차원의 드로잉으로 현재 시뮬레이션을 하고 있지만 시뮬레이션 알고리즘 상에선 총 5단계의 깊이 정렬을 한다. Bottom, Low, Middle, Hight, Top이 바로 그것이다. 이 5단계를 DirectX 상에서 z좌표 값을 할당하여 깊이 값을 주고 2차원 원통 모양으로 표현한 부분을 3차원 Cylinder로 교체하여 3D 시뮬레이션을 구현하는 계획을 구상하였다.

Rendering의 구조부분을 어느정도 완성하였고 여기에 카메라 인터페이스만 완성을 한다면 2차년도 시뮬레이션은 구상했던 단계에 도달할 것으로 보인다.