게임알고리즘심화과정

과목명 : 게임응용프로그래밍

능력단위 : 게임플랫폼응용프로그래밍

제출일자 : 2021년 10월 11일

포트폴리오 : 동적 및 정적 지형 LOD 구현,

윈도우 기반 3D게임엔진 라이브러리

작성자 : 최지원

(제출내역)

1. 동적 및 정적 지형 LOD , 윈도우 기반 3D게임 엔진 프로젝트

2. 동적 및 정적 지형 LOD, 윈도우 기반 3D게임 엔진 구현 분석 및 세부 문서

목차

- **UML 다이어그램 시스템 설명**

선택2. 공간 분할된 게임 지형에 동적 및 정적 지형 LOD를 구현하여 렌더링한다.

A. 기능 구현에 따른 분석내용 및 자체 포맷의 설계도.

A-1. 동적 및 정적 지형 LOD의 클래스 다이어그램

그림 1

Sample 클래스	3D 엔진 라이브러리, 인스턴스 관리 기본 클래스

KCamera 클래스	3D Camera 기본 클래스
KDebugCamera 클래스	Debug용 자유 이동 카메라 클래스
KModel 클래스	버퍼 생성, 쉐이더 관리, 오브젝트 렌더 클래스
KMap 클래스	버텍스 데이터, 인덱스 데이터 생성 및 맵(지형) 로드 클래스
KQuadtree 클래스	리프노드 렌더, 동적 정적 LOD 클래스
KRState 클래스	RasterizerState 관리 클래스 디버깅용 wireframe
TCore 클래스	Input, Timer, Write, Window, Device 등 게임 엔진 라이브러리 기본 클래스

표 1

A-2. 동적 및 정적 지형 LOD 의 시퀀스 다이어그램

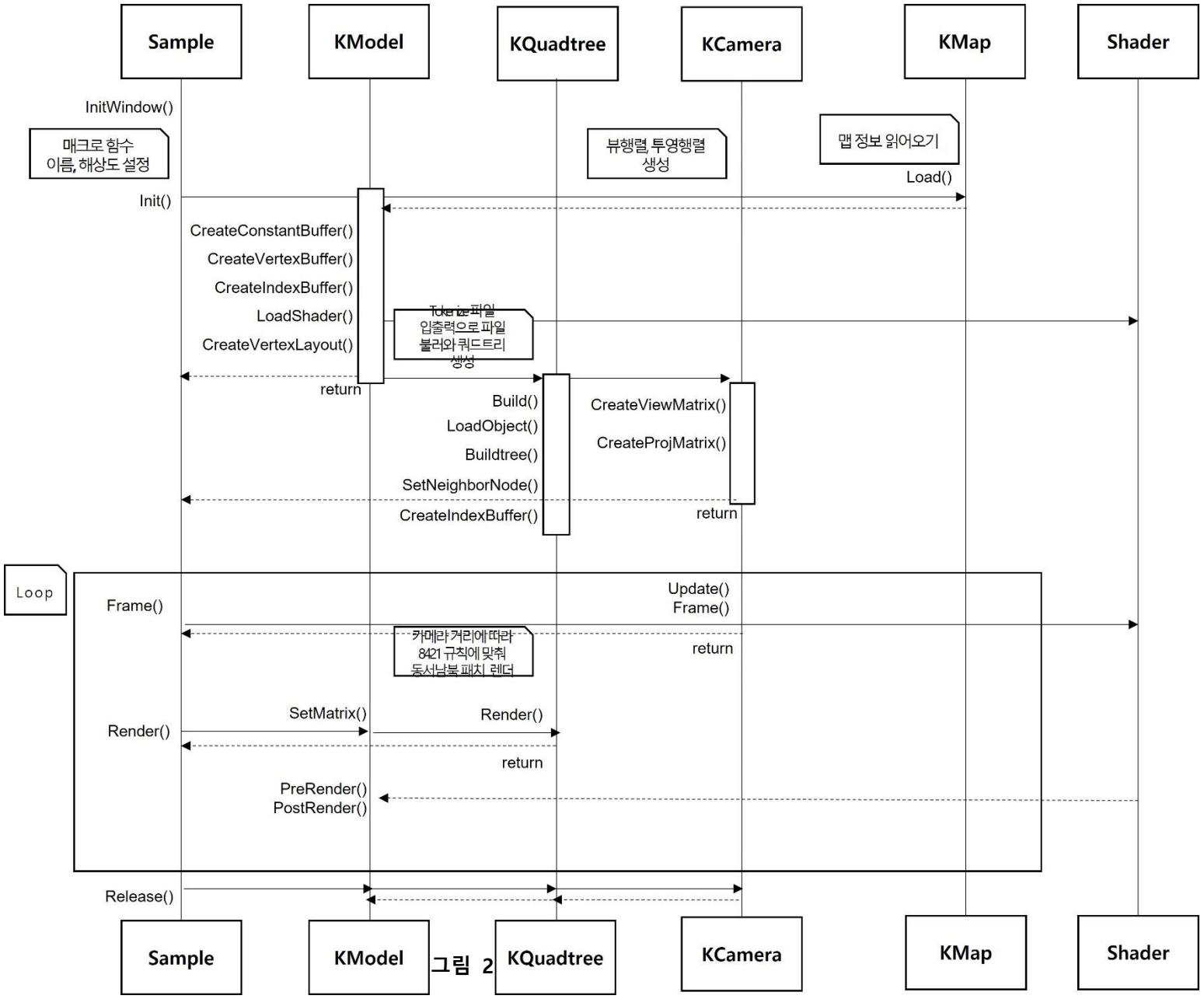


그림 2

1	맵 정보를 읽어온다.
2	정적인 방사형 구조 LOD 정보를 불러와 공유 인덱스버퍼와 리프노드 당 정점 버퍼가 있는 쿼드트리를 만든다.
3	카메라 뷰행렬, 투영행렬 생성
4	카메라 거리에 따른 8421 규칙을 적용해 리프노드 구간에 LOD 구현
5	공유되는 인덱스버퍼, 리프노드당 인덱스 버퍼, 쉐이더에 정보 넘겨줘서 렌더링
6	동적 메모리 해제

표 2

공통. 윈도우 기반의 3D게임엔진 라이브러리를 제작한다.

A-3. (공통) 윈도우 기반의 3D 게임엔진 라이브러리 클래스 다이어그램

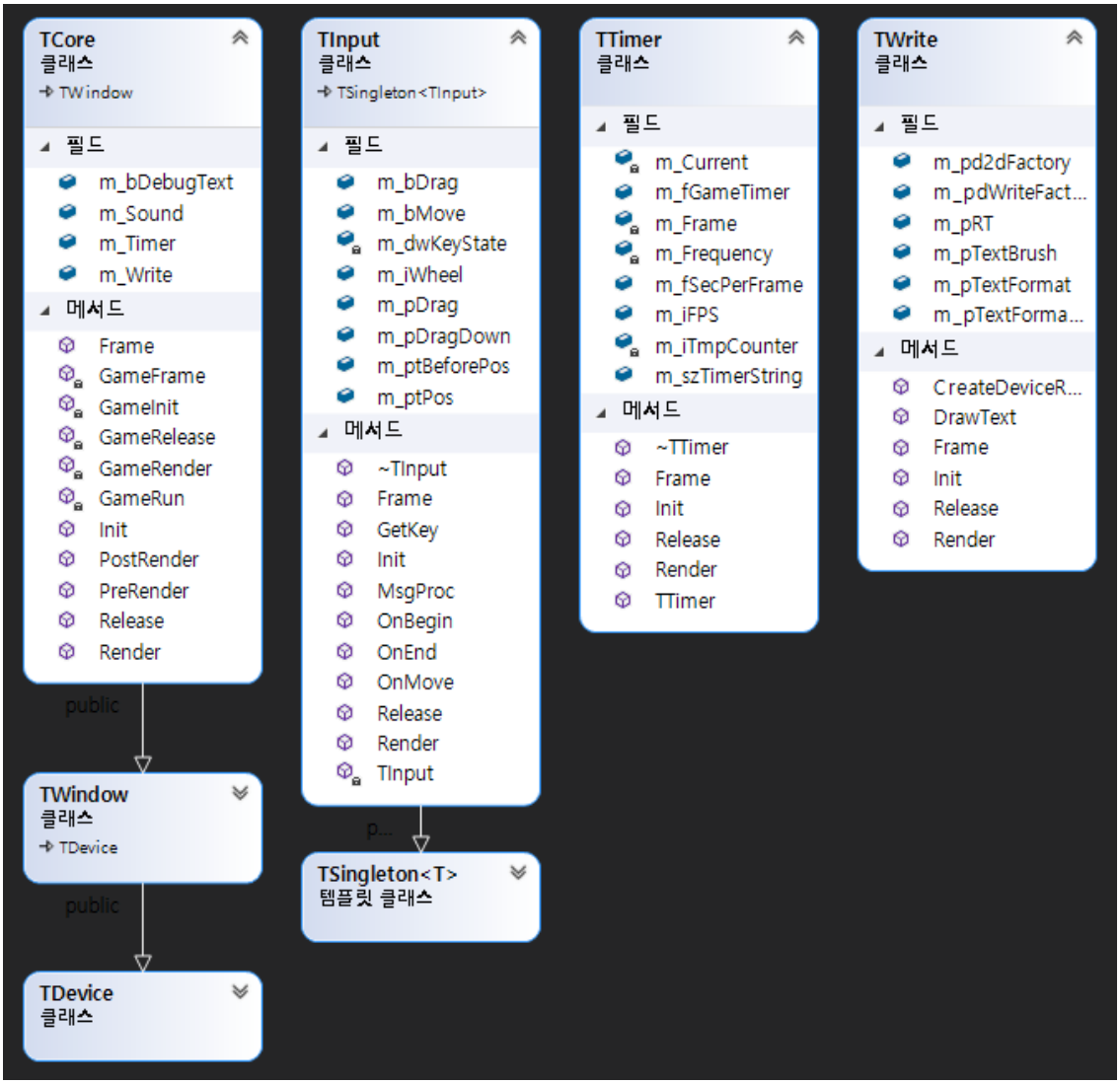


그림 3

TCore 클래스	3D게임엔진 프로젝트 인스턴스 관리 클래스
TWindow 클래스	윈도우 기반 클래스
TDevice 클래스	DirectX 기반 클래스
TInput 클래스	싱글톤으로 구현된 마우스, 키보드의 입력 정보 처리 클래스
TTimer 클래스	주파수, 시간, 프레임 등 시간 관리 클래스
TWrite 클래스	2D기반 문자 출력 클래스

표 3

A-4. (공통) 윈도우 기반의 3D 게임엔진 라이브러리 시퀀스 다이어그램

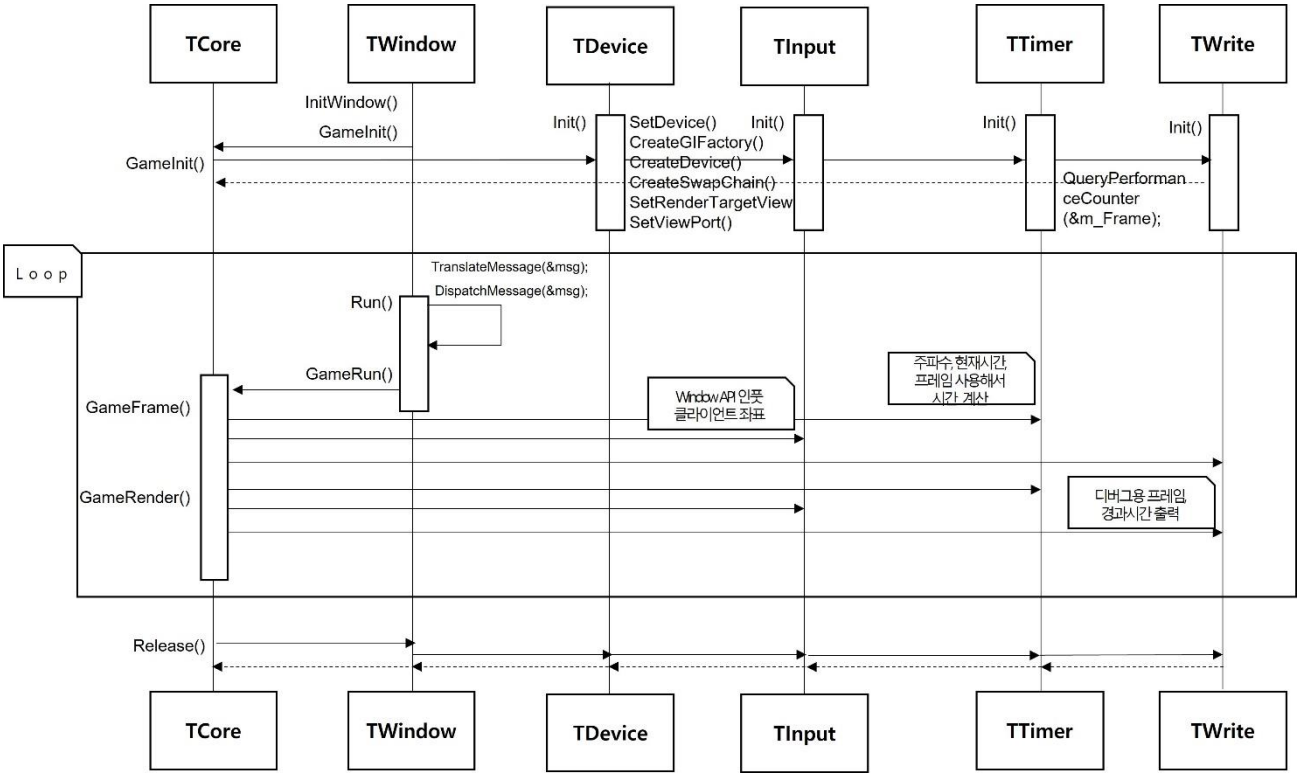


그림 4

1	wWinmain 함수로 윈도우 초기화, 윈도우 생성
2	GameInit() 다른 객체들 초기화 및 생성, DX Device, Context 생성
3	PeekMessageW로 윈도우 메시지 처리, 없을 시 GameRun() 반복
4	코어의 GameFrame으로 Timer, Write, Input 등 연산 반복
5	코어의 GameRender으로 Timer, Write, Input 등 렌더링
6	동적 메모리 해제

표 4

- **LOD 설계**

B- LOD 의 개요

LOD 는 Level Of Detail 으로 단계에 따라서 메시의 모델링 데이터의 정밀도를 조절하는 것이다. 게임에서 최적화를 위해서 거리에 대응해 LOD 를 사용한다. 크게 두 가지로 나뉜다.

첫 번째, 정적 LOD 는 처음부터 메시의 정밀도가 정해져 있고 이를 카메라와의 거리에 따라서 단계별로 인덱스 버퍼의 교체를 통해 출력하는 것이다. 이미 메시의 정밀도가 이미 정해져 있어서 연산이 간단해 속도가 빠르다. 하지만 메시를 추가적인 메모리를 가지고 있어야 하며 단계가 급격히 변화해 튀는 현상이 있다.

두 번째, 동적 LOD는 실시간으로 메시의 정밀도를 변화시키는 기법이다. 자연스럽게 LOD 가 이루어지기 때문에 튀는 현상이 적고, 낭비되는 메모리도 없지만 계속해서 연산을 해야 되기 때문에 상대적으로 속도가 느리다는 단점이 있다.

B-1 LOD 버퍼의 균열 방지

LOD는 다른 여러 단계의 메시를 단일 메시에 적용한다. 단계별 메시의 단위를 패치라고 한다. 패치는 카메라로부터의 거리를 기반으로 적용되기 때문에 이웃 노드들과의 LOD 레벨이 다를 수밖에 없다. 이웃 정점들과 패치의 정점이 공유되지 못하면서 균열이 일어나게 된다.

이를 방지하기 위해서 균열이 일어날 경우의 수를 알아야 한다. 그림 5 와 같다.

코드	조합
0	대상 노드의 4방향 이웃 노드의 LOD레벨 값이 같을 경우에 해당됨.
1	Upper(0001)=0001
2	Right(0010)=0010
3	Upper(0001)+Right(0010)=0011
4	Lower(0100)=0100
5	Upper(0001)+Lower(0100)=0101
6	Lower(0100)+Right(0010)=0110
7	Lower(0100)+Right(0010)+Upper(0001)=0111
8	Left(1000)=1000
9	Left(1000)+Upper(0001)=1001
10	Left(1000)+ Right(0010)=1010
11	Left(1000)+ Right(0010)+Upper(0001)=1011
12	Left(1000)+ Lower(0100)=1100
13	Left(1000)+ Lower(0100)+Upper(0001)=1101
14	Left(1000)+ Lower(0100)+Right(0010)=1110
15	Left(1000)+ Lower(0100)+Right(0100)+Upper(0001)=1111 대상 노드의 4 방향의 이웃 노드 LOD값이 큰 경우에 해당됨.

그림 5

총 16 개의 경우의 수로 이 프로젝트에서는 한 개의 정수형 변수에 저장하기 위해서 8421 코드를 사용하였다. 8421 코드는 4 비트의 2 진수로 0000~1111 로 최대 16 개의 경우의 수를 표현해주는 역할이다. 해당 노드의 LOD 레벨이 이웃레벨의 LOD 레벨보다 크다면 각 방향에 해당하는 8421 코드를 더해서 조합을 한다. 그리고 해당하는 버퍼에 넣어주면 되는 로직이다.

이 프로젝트는 두개의 패치(각 16 개의 정점버퍼)가 포함된 외부파일과 공유되는 인덱스 버퍼로

총 3 단계의 LOD 를 구현했다.

B-2 LOD 주요 함수 설계

<시스템 구성도에서 제시한 모듈에 포함된 클래스나 함수에 대한 구체적인 설계>

void KQuadtree::Build(KMap* pMap)

- 정적 LOD 인 StaticLOD.txt 를 받아서 쿼드 트리를 생성하고 이웃 노드를 설정해주며 공유 인덱스 버퍼를 업데이트해준다.

- 셀 계수, 패치 계수를 구하는 공식 : m_iNumCell = (m_iNumCol-1) / pow(2.0f, m_iMaxDepth); m_iNumPatch = (log(m_iNumCell) / log(2.0f));

위 공식을 이용해서 LODPATCH 가 몇 개인지 알 수 있고 패치개수 만큼 resize 해준다.

bool KQuadtree::LoadObject(std::wstring filename)

- 파일 입출력으로 StaticLod.txt 를 읽어와 패치 크기대로 Tokenize, 토큰으로 잘라서 벡터에 넣어준다. 패치 크기만큼 인덱스 버퍼를 생성한다.

void KQuadtree::Tokenize(const std::wstring& text, const std::wstring& delimiters, OutputIterator first)

- 콤마로 구분하기 위해서 토큰화를 사용한다

- Find_first_not_of 는 토큰이 아닌 첫 번째 것을 불러오는 것이다.

- Find_first_of 는 콤마를 찾는 것이다.

void KQuadtree::Buildtree(KNode* pNode)

- Subdivide 함수를 사용해서 간격이 4 보다 크면 쪼갤 수 없냐 있냐를 판단해서 재귀 함수를 사용해 LOD 쿼드 트리를 생성한다. 쪼개지지 않으면 리프 노드이며 리프 노드의 버텍스 버퍼 생성한다.

void KQuadtree::SetNeighborNode()

- 모든 노드의 4 방향(상하좌우)의 이웃노드를 얻는다

bool KQuadtree::UpdateIndexList(KNode* pNode)

- 공유되는 인덱스 버퍼를 업데이트해준다. CreateIndexBuffer(m_pLeafList[0]) 인덱스 버퍼 생성

bool KQuadtree::UpdateVertexList(KNode* pNode)

- 리프노드 렌더링으로 Lod 는 패치단위로 16 개의 정점 인덱스 버퍼가 있어야하기 때문에 리프노드 당 정점 버퍼를 업데이트 해준다. CreateVertexBuffer(KNode* pNode)로 버텍스 버퍼 생성

bool KQuadtree::Render(ID3D11DeviceContext* pContext, KVector3* vCamera)

- 이 프로젝트에서는 한 개의 정수형 변수에 저장하기 위해서 8421 코드를 사용하였다. 8421 코드는 4 비트의 2 진수로 0000~1111 로 최대 16 개의 경우의 수를 표현해주는 역할이다. 해당 노드의 LOD 레벨이 이웃레벨의 LOD 레벨보다 크다면 각 방향에 해당하는 8421 코드를 더해서 조합을 한다. 그리고 카메라를 인자로 받아와 거리에 따른 LOD 를 D3D Context 로 렌더한다.

bool KRState::Render(ID3D11DeviceContext* context)

- ID3D11RasterizerState* m_pRSSolid, m_pRSWireFrame 를 이용해 디버깅에 필요한 와이어프레임으로 렌더링 할 수 있다.

KMatrix TDebugCamera::Update(KVector4 vValue)

- 디버깅용 카메라 업데이트 함수이다. 쿼터니언, 즉 사원수를 이용해 회전 행렬로 변환한다. 사원수는 짐벌락을 방지한다는 장점이 있다. 카메라는 오브젝트 회전과 반대이므로 역행렬로 한번 더 변환해서 적용한다.

- **3D 게임엔진 라이브러리 설계**

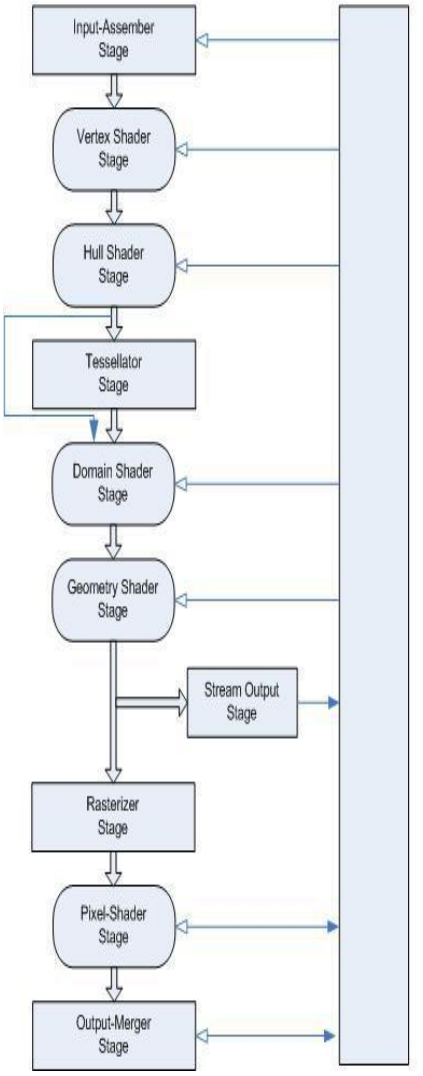
C. 윈도우 기반의 3D 게임엔진 라이브러리의 개요

윈도우 시스템의 모든 애플리케이션은 메시지(또는 이벤트)를 기반으로 구동된다. 윈도우 프로그래밍은 애플리케이션에서 사용자가 발생시키는 메시지에 대한 처리 "루틴"을 만들어 주는 것이라고 할 수 있다.

윈도우는 WinMain() 함수의 원형을 쓰는데 전형적으로 (1. 윈도우 클래스 생성 2. 윈도우 클래스 등록 3. 윈도우 생성 4. 윈도우 화면에 표시 5. 메시지 큐로 메시지 받아 윈도우 프로시저로 보냄) 의 루틴으로 진행된다.

해당 프로젝트는 3D 게임 엔진 라이브러리로 구축되었으며 DirectX를 사용한다. 다이렉트X는 그래픽카드에 직접 명령하는 API로 빠른 처리가 가능하다.

그림 6



왼쪽 그림 6은 DirectX11 그래픽스 파이프라인이다.

1	user-filled buffer로부터 원시 데이터를 읽어들여 그 후 파이프라인 단계에서 사용될 primitive types(line list, triangle strips 등등)데이터로 assemble하는 역할을 수행한다.
2	input assembler 단계를 통과한 vertices들을 처리하는 단계. transformation, skinning, morphing, and per-vertex lighting과 같은 작업을 수행. vertex shader는 단일 input vertex을 받고 output으로 단일 vertex를 출력한다.
3	다이렉트11은 테셀레이션(GPU에서 low-detail subdivision surface를 higher-detail primitves로 바꾸는)을 구현하기위한 3가지의 새로운 단계를 support합니다. Hull shader, tesselator, domain shader 단계가 그것입니다.
4	도메인(quad, tri or line)을 더 많은 작은 오브젝트(triangles, points or lines)로 잘게 나누는 작업을 수행한다.
5	잘게 나누어진 포인트의 vertex position를 계산하는 단계.
6	정점을 input으로 받아 application-specified 쉐이더를 적용하고 생성된 정점들을 출력한다. 버텍스 쉐이더와는 다르게 기하쉐이더는 입력으로 full primitive의 정점을 받는다. 기하쉐이더는 edge 인접 원시 데이터들을 입력을 받을 수 있다.
7	기하 쉐이더단계(또는 정점쉐이더 단계)에서 메모리에 있는 하나 이상의 버퍼로 정점 데이터를 연속적으로 출력하는 단계. 메모리에 출력된 데이터들은 pipeline에서 다시 읽어들일 수 있다.
8	vector 정보를 레스터 이미지(픽셀)로 변환하는 단계.
9	픽셀 라이팅이나 post-processing과 같은 쉐이딩을 가능하게하는 단계.
10	최종출력되는 픽셀의 색상을 생성한다. depth/stencil 테스트를 수행하여 실제로 렌더링여부를 결정하고 최종색상을 blend 한다.

윈도우를 생성하고 DirectX 를 활용하기위해 스왑체인, 렌더타겟 뷰, 뷰포트세팅, 디바이스/디바이스컨텍스트 객체 생성 등 작업이 필요하다. 이 프로젝트에서는 윈도우를 상속받은 TCore 에서 Device 객체와 그 외 게임에 필요한 인풋, 타임, 2D Write 의 객체를 생성해 관리한다.

C-1. 윈도우 기반의 3D 게임엔진 라이브러리 주요 함수 설계

bool TWindow::InitWindows

- 윈도우 클래스 등록과 생성을 담당한다. 화면 좌표와 클라이언트 좌표를 받아온다.

LRESULT TWindow::WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

- 윈도우 프로시저로 메시지에 대한 처리를 한다. 종료하기 같은 기본적인 윈도우 메시지를 처리하고 마우스의 움직임도 해당 함수에 메시지를 받아서 활용한다.

bool TWindow::Run()

- 반복문을 사용해 계속해서 peekmessageW 로 모든 메시지를 받아온다.

bool TDevice::SetDevice()

- D3D 의 설정을 담당한다. CreateGIFactory()함수로 그래픽카드를 설정하고 CreateDevice 로 생성을 담당하는 Device,ImmediateContext 객체를 생성한다. CreateSwapChain 은 프론트버퍼와 백버퍼의 플립핑를 구현한다. SetRenderTargetView(), SetViewPort()은 각각 렌더타겟을 설정, 뷰포트를 설정한다.

LRESULT TInput::MsgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

- 마우스 인풋을 윈도우 프로시저로 받아온다. setCapture 함수는 화면 밖에서도 작동한다. 마우스의 움직임, 왼쪽 마우스 클릭 누름과 땜을 감지하여 드래그를 구현하였다.

bool TInput::Frame()

- Window API 로 키보드, 마우스 인풋을 구현했다. 키상태를 열거형으로 등록하여 눌려진 키를 반환한다. 마우스는 클라이언트 좌표계에서 작동하게 되었다. TInput 은 gof 디자인 패턴의 생성패턴인 싱글톤으로 구현되어 어느 클래스에서든 인스턴스에 접근 가능하다.

bool TTimer::Frame()

- QueryPerformanceCounter 함수로 Performance Counter Frequency 라는 주파수에 따라 1 초당 진행되는 틱 수 계산하여 속도를 측정할 수 있다. Init 과 Frame 에서 왔다갔다 하는 틱을 이용해 지금 프레임과 이전 프레임을 빼고 주파수를 나눠서 프레임 초를 구할 수 있다. 프레임을 더 해 타이머로 구현하였다.

bool TWrite::DrawText(RECT rt, const TCHAR* data, D2D1::ColorF color, IDWriteTextFormat* pTextFormat)

- Write 글자는 2D 이기 때문에 3D 와의 연동이 필요하다. CreateDeviceResource ()로 백버퍼를 넘겨서 연동이 되어야한다. 또한 Device 에서도 준비작업이 요구된다. D3D11_CREATE_DEVICE_BGRA_SUPPORT 로 설정하면 된다. 위 함수는 rect 와 글자, 색, 텍스트 포맷을 받아와 출력을 해주는 함수이다. 디버그용으로 프레임과 타이머를 출럭한다.

- **최종 결과물**

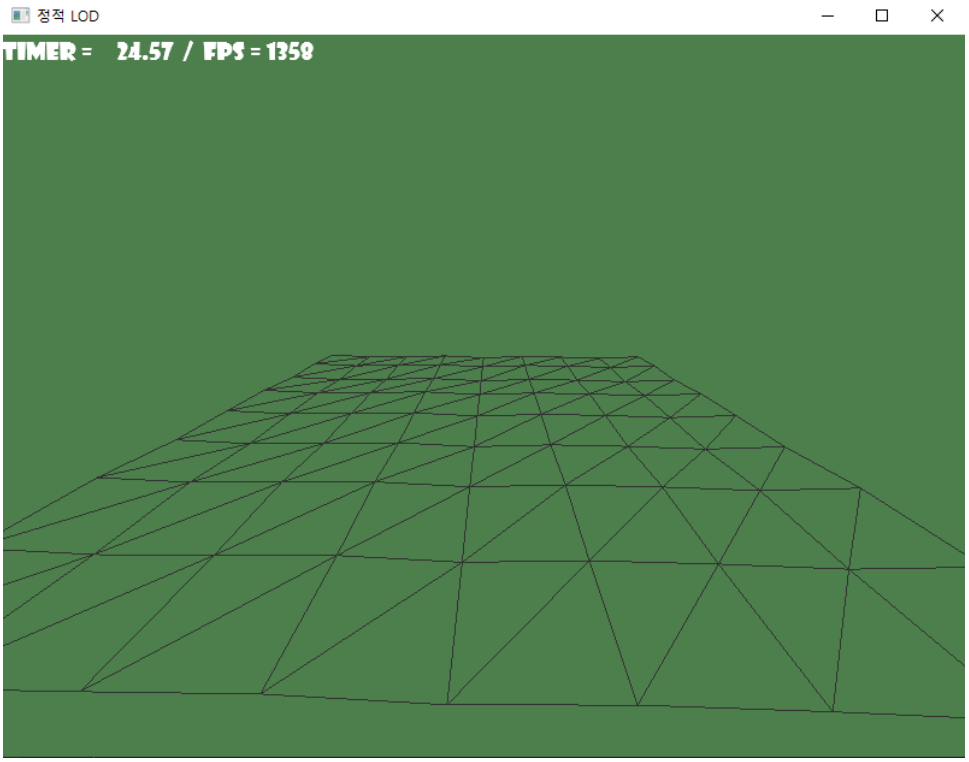


그림 7 카메라에 거리가 멀어졌을 때 최상단의 패치를 보여준다.

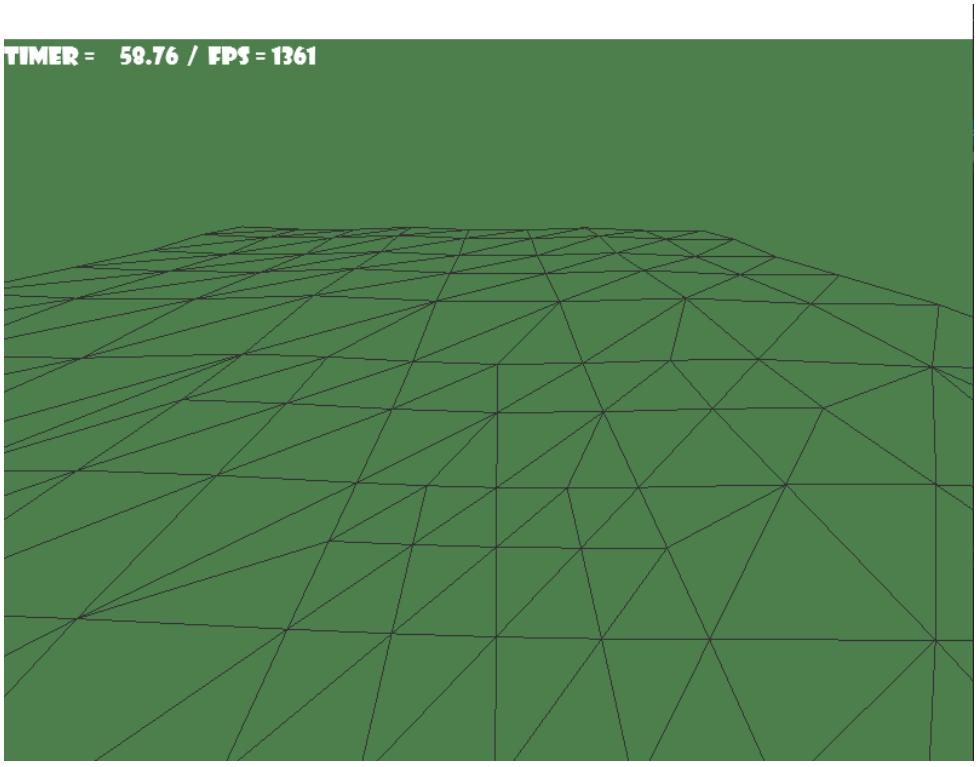


그림 8 거리에 따라서 두개의 패치(각 16개의 정점버퍼)가 포함된 외부파일과 공유되는 인덱스 버퍼로 총 3단계의 LOD를 보여주고 있음

D-1. 제작된 기능/라이브러리를 수정/보완해야할 사항들을 기술

게임 엔진 라이브러리로 방사형 정적 LOD 를 구현했다.

거리에 따라 폴리곤수가 갑작스럽게 줄어들거나 늘어나면 그래픽이 튀어보이는 현상인 Popping 현상을 막기 위해 거리에 따라 자연스럽게 LOD 가 이루어지는 동적 LOD 를 구현하여 보완한다.

공간 분할 알고리즘은 주로 실시간 렌더링 데이터 검출과 실시간 충돌 데이터 검출을 위해 사용한다. 그 중, Octree 는 Quadtree 에서 높이에 대한 분할까지 고려하는 것이다. 따라서, 최적화를 위해서 넓은 공간(야외)이고 날 수 있는 환경에서는 Octree 를 적용하고, 평면 하나밖에 없는 환경에서는 Quadtree 를 사용하도록, 각 환경에 적합한 알고리즘은 골라 적용하게 병행하는 알고리즘을 추가할 수 있다.

D-2. 범용성/유연성/확장성/간결성을 고려하여 추가/변경된 내용을 기술

매크로 함수를 이용해 wWinMain 해상도와 창이름등을 임의로 바꿀수 있고, 실행파일의 크기를 줄여주고 코드의 재사용성을 높여주었다.

기능 별로 세분화되어 재활용 할 수 있다.

상속을 통해 코드를 재활용함으로써 간소화된 클래스 구조를 갖고 있다.

참고: https://docs.microsoft.com/en-us/windows/desktop/direct3dgetstarted/understand-the-directx-11-2-graphics-pipeline