개발자 Portpolio

차승철

목차

- 프로젝트
  - Zelda-Like MMO
  - DirectX 12 Rendering Engine
  - 실외 CCTV 폭력 인식 시스템

Zelda-Like MMO Server

- 개발 목적
  - 평소에 즐겨하던 2D 게임을 멀티 버전으로 모방해보는 것이 목표
  - 유니티의 개발 방식을 공부해보는 것이 목표
  - 동시에 범용적인 게임 서버를 만들어 보기 위해서 만들어진 프로젝트입니다.
- 개발 기간 : 2022년 01월 ~ (개발중)
- 개발 인원 : 1명
- 사용 기술
  - C# / C++ - 개발을 위해 활용한 주 언어입니다. (github : C# / C++)
  - ProtoBuf - 서버와 클라이언트의 패킷 직렬화를 쉽게 하기 위해서 사용했습니다.
  - Python - ProtoBuf의 결과를 자동화하기 위해 사용했습니다.
- 시연 영상

Zelda-Like MMO Server

- 특징
  - 게임과 서버와 통신을 위해 필요한 모듈이 담긴 Core 라이브러리를 제공합니다.
  - ProtoBuf를 이용해서 Packet를 좀 더 편하게 개발할 수 있습니다.
  - 활용 예제를 위한 Game Server가 프로젝트가 존재합니다.
  - 서버의 패킷 송수신의 테스트를 할 수 있게 만든 Dummy Client 프로젝트가 프로젝트가 존재합니다.

프로젝트 개요

- ServerCore
  - 서버와 클라이언트(C++인 경우)가 공유하는 핵심 모듈들을 공유하는 라이브러리 클래스입니다.
  - 빌드시 정적 라이브러리 파일(.lib)로 반환됩니다.
  - Network 초기화 및 관리
  - Thread/Memory 관리
  - TLS 및 서버 관리에 필요한 전역 변수
  - Json 파싱
  - MySQL와 연결
  - 비동기 통신 시 패킷을 모아서 보내기 위한 JobQueue 지원

프로젝트 개요

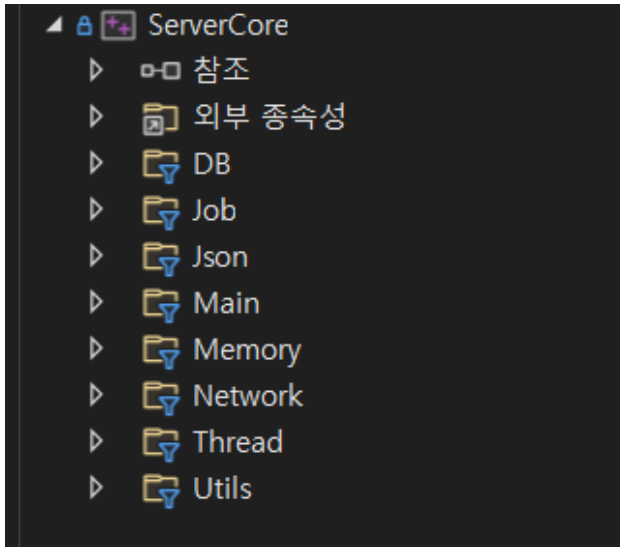
- GameServer
  - ServerCore를 이용해서 실제 컨텐츠를 제작하기 위해 만들어진 컨텐츠입니다.
  - 빌드 시 ProtoBuf 패킷 클래스가 같이 빌드가 됩니다.
  - 각종 패킷들은 PacketManager 클래스를 통해서 가공되어 전달됩니다.
  - 클라이언트가 접속하면 세션을 유지해주는 역할을 합니다.
  - 세션에 접속된 플레이어들에게 패킷들을 전달하는 역할을 진행합니다.

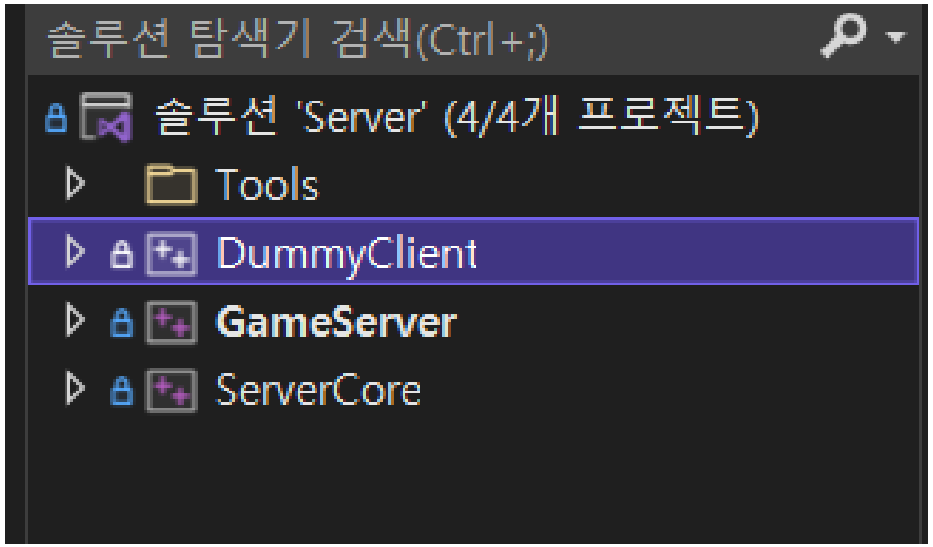
프로젝트 개요

- DummyClient
  - 실제 게임이 아니라 서버에 패킷을 전달하는 것을 테스트 하기 위한 프로젝트입니다.
  - 단순히 콘솔창으로 구성되어 있으며, PacketManager를 통해서 프로그래머가 테스트 하고 싶은 패킷을 보낼 수 있습니다.
- Tools
  - ProtoBuf 패킷을 빌드하는데 필요한 작업을 자동화하는 프로젝트입니다.

서버 코드 설명 (C++)

- 게임 서버에 개발에 있어서 필요한 부분들은 ServerCore라는 프로젝트에 모듈화.
- ServerCore에는 크게 DB, Job, Json Read, Memory, Network(IOCP), Thread를 관리하는 클래스들로 구성됨
- Main에는 개발에 필요한 Type이나 Header Include 등을 담당하는 코드로 구성





서버 코드 설명 (C++)

- ProtoBuf를 이용해서 유니티와 C++ 서버 간 데이터를 직렬화한 뒤 이를 패킷으로 전달
- JobQueue 방식을 이용해서 필요한 함수를 Job 단위로 등록해둔 뒤 데이터가 들어오면 Queue에 넣고 이를 한꺼번에 처리할 수 있도록 Async하게 처리할 수 있게 구현해서 기존의 Lock 방식의 단점을 어느 정도 해결함

서버 코드 설명 (C++)

- 데이터는 다음과 같이 Protobuf 를 통해 만든 클래스 양식을 이용.
- 이 때 보내기 위한 데이터를 다음과 같이 패킷에 저장을 한 뒤 현재 접속한 플레이어에게 전송

```
Protocol::S_ENTERGAME enterPacket;
auto info = enterPacket.mutable_player();
info->set_objectid(player->GetObjectInfo().objectid());
info->set_name(player->GetObjectInfo().name());
auto sendBuffer = ClientPacketHandler::MakeSendBuffer(enterPacket);
player->_session->Send(sendBuffer);
```

서버 코드 설명 (C++)

- 반대로 클라이언트가 패킷을 보낸다면, 패킷을 받아서 이에 대한 처리를 진행한 뒤 이를 그 플레이어가 들어있는 GameRoom에 있는 모든 플레이어에게 이러한 패킷의 정보를 Broadcast.

```
bool Handle_C_MOVE(PacketSessionRef& session, Protocol::C_MOVE& pkt)
{
    Protocol::C_MOVE movePacket = pkt;
    GameSession* game = dynamic_cast<GameSession*>(session.get());

    Player* player = game->GetPlayer();
    if (player == nullptr)
        return false;

    GameRoom* room = player->Room;
    if (room == nullptr)
        return false;

    room->DoAsync(&GameRoom::HandleMove, player, movePacket);

    return true;
}
```

Zelda-Like MMO Server
요약

- 이러한 방식을 이용해서 플레이어의 입장, 퇴장 시스템
- 몬스터 스폰 및 플레이어까지의 최단 경로 계산
- 오브젝트의 움직임, 스킬 사용
- 스킬 사용시 스킬 이펙트 생성
- 이러한 과정에서 생성되는 오브젝트 관리
- 클라이언트와 공유하는 맵 정보 불러오기
- 와 같은 컨텐츠를 구현했습니다.
- 이 외 MySQL을 통해 Player 정보 저장하는 기능을 구현했습니다.

- 현재는 추가적으로 C++ 서버에 로그인 -> 서버 -> 플레이 단계로 갈 수 있는 기능을 구현중입니다..

DirectX 12 Rendering Engine

- 개발 목적
  - 컴퓨터 그래픽스 복습 겸 Windows에서 제공하는 라이브러리를 공부해보는 것이 목표입니다.
  - 게임 엔진의 기반이 되는 API인 만큼 공부를 하고 넘어가는 편이 좋다는 취지 하에 학습 목표로 공부하기 시작했습니다.
- 개발 기간 : 2021년 07월 ~ 2022년 2월
- 개발 인원 : 1명
- Github / 구현 영상

DirectX 12 Rendering Engine

- 특징
  - 렌더링 엔진 구조는 게임 엔진인 Unity를 묘사하는 방향으로 개발을 진행했습니다.
  - 로직을 총괄하는 Core 부분을 정적 라이브러리화 하여 개발 편리성을 높였습니다.
  - 정적 라이브러리를 사용하는 Client 구조로 나뉘어져 있습니다.
  - 디버깅을 위한 멀티 렌더링 타겟을 지원합니다.

DirectX 12 Rendering Engine

- 사용 기술
  - C++ - DirectX 12가 지원하는 언어가 C++이기 때문에 사용했습니다.
  - DirectX 12 - Windows에서 제공하는 멀티미디어 API입니다.
  - FBX - 3D FBX 애니메이션을 실행하기 위해서 사용했습니다.

DirectX 12 Rendering Engine

- 클라이언트는 엔진 클래스를 초기화합니다.
- 제일 처음 씬을 생성한 뒤 이를 엔진이 관리하는 씬 매니저에게 전달합니다.
- 이후 엔진은 Update를 프로그램이 끝날 때까지 반복합니다.

```
void Game::Init(const WindowInfo& info)
{
    GEngine->Init(info);

    TestScene* testScene = new TestScene();
    GET_SINGLE(SceneManager)->LoadScene((Scene*)testScene);
}

void Game::Update()
{
    GEngine->Update();
}
```

DirectX 12 Rendering Engine

- 이후 엔진은 초기화를 실시합니다.
  - 디바이스
  - CPU에서 GPU에 전달하기 위한 Command를 저장하는 Queue
  - 스왑 체인
  - 루트 서명
  - 디스크립터 힙
  - 상수 버퍼
  - 각종 매니저 클래스

```
_device->Init();
_graphicsCmdQueue->Init(_device->GetDevice(), _swapChain);
_computeCmdQueue->Init(_device->GetDevice());
_swapChain->Init(_window, _device->GetDevice(), _device->GetDXGI(), _graphicsCmdQueue->GetCmdQueue());
_rootSignature->Init();
_graphicsDescHeap->Init(256);
_computeDescHeap->Init();
```

```
CreateConstantBuffer(CBV_REGISTER::b0, sizeof(LightParams), 1); // 여러
CreateConstantBuffer(CBV_REGISTER::b1, sizeof(TransformParams), 256);
CreateConstantBuffer(CBV_REGISTER::b2, sizeof(MaterialParams), 256);

CreateRenderTargetGroups();

ResizeWindow(info.width, info.height);

GET_SINGLE(Input)->Init(info.hwnd);
GET_SINGLE(Timer)->Init();
GET_SINGLE(Resources)->Init();
GET_SINGLE(ImGuiManager)->Init(info);
```

DirectX 12 Rendering Engine

- 초기화가 완료된 뒤에는 엔진이 계속 Update 됩니다.
- 먼저 입력을 받습니다.
- 이 입력에 대한 게임 매니저의 처리를 합니다.
- 그리고 타이머를 업데이트한 뒤
- 씬에 있는 모든 오브젝트를 업데이트합니다.
- 이후 인스턴스 매니저 안에 있는 인스턴스 버퍼를 비운 뒤
- 이러한 결과물을 컴퓨터에 그릴 수 있게 전달합니다.

```
void Engine::Update()
{
    GET_SINGLE(Input)->Update();
    GET_SINGLE(GameManager)->Update();
    GET_SINGLE(Timer)->Update();
    GET_SINGLE(SceneManager)->Update();
    GET_SINGLE(InstancingManager)->ClearBuffer();

    Render();

    ShowFps();
}
```

DirectX 12 Rendering Engine

- 씬 매니저에서는 씬 안에 있는 문맥에 대한 업데이트를 진행합니다.
- 이후 LateUpdate를 통해서 카메라에 대한 업데이트를 진행합니다. 이 방식은 Unity 엔진이 사용하고 있는 방식입니다.
- 이후 FinalUpdate를 통해서 충돌처리 등을 처리합니다.

```
void SceneManager::Update()
{
    if (_activeScene == nullptr)
        return;

    _activeScene->Update();
    _activeScene->LateUpdate();
    _activeScene->FinalUpdate();
}
```

DirectX 12 Rendering Engine

- 결론
  - 게임 엔진의 기초가 되는 렌더링 파이프라인에 대한 이해가 가능했습니다.
  - 동시에 게임 엔진의 기본 돌아가는 구조 등등을 파악할 수 있었습니다.
  - 혼자서 개발을 진행해서, 일정상 실질적인 컨텐츠 개발까지는 못할 것으로 판단해서, 개념 학습에서 멈춘 점이 아쉬운 점으로 남습니다.

CCTV 폭력 인식 시스템

- 개발 목적
  - 공공의 안전을 위해, 사람의 육안에 의존하는 현재 CCTV 관제 환경을 좀 더 사람이 편하게 개선해보자는 취지에서 시작
  - 그 외 사람들이 사용하기 편하게 하기 위해 웹 환경을 이용해서 조작할 수 있도록 하는 것이 목표
- 개발 기간 : 2021년 03월 ~ 2021년 06월
- 개발 인원 : 3명
- 깃허브 / 시연 영상

CCTV 폭력 인식 시스템

- 특징
  - 딥러닝 기술 중 시계열을 처리하는 LSTM(Long Short-Term Memory) 기술과 이미지 처리를 위한 CNN(Convolution Neural Network) 기술을 사용해서 범죄를 탐지해내는 모델을 개발
  - Node.JS 프레임워크를 이용해 서버를 만들고, 템플릿 엔진으로는 ejs, 데이터베이스는 mySql을 이용해서 연구를 위한 웹 페이지를 개발
  - 해당 서비스를 이용해 CCTV가 설치된 장소와 시간, 폭력일 확률과 비폭력일 확률, 그리고 인공지능 모델의 분석 결과가 폭력상황인지 비폭력 상황인지를 탐지 후 통보

CCTV 폭력 인식 시스템

- 사용 기술
  - NodeJS - 모델의 결과물을 웹 페이지로 전달하고, DB에 접근해서 저장합니다.
  - Tensorflow - OpenCV를 통해서 전달받은 신호를 버퍼에 저장하고 100장 모였을 시 범죄 상황을 탐지합니다.
  - OpenCV - 카메라의 신호를 가져와서 Tensorflow에 전달합니다.
  - MySQL - 녹화되고 있는 영상에 대한 정보, 관리자 및 사용자에 대한 정보, 그리고 범죄 상황에 대한 상세한 정보를 담기 위한 용도로 사용했습니다.

코드 설명 (Model)

```
if((total_no) > (total_fi)) :
    no_count += 1
    # 이 데이터는 클립 데이터입니다.
    json_data = {
        'filename' : filename, \
        "pred_type": 'Non-Violence', \
        "time" : datetime.now().strftime('%Y-%m-%d-%H-%M-%S'),\
        "Violence_percent" : (total_fi * 100) / pred_image_cnt, \
        "Non_Violence_percent" : (total_no * 100) / pred_image_cnt ,\
        "fi_count" : fi_count , \
        "no_count" : no_count
        }
    headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}
    sio.emit('pred_data', json_data)
    print('no')
else :
    fi_count += 1
    # 이 데이터는 클립 데이터입니다.
    json_data = {
        'filename' : filename, \
        "pred_type": 'Violence', \
        "time" : datetime.now().strftime('%Y-%m-%d-%H-%M-%S'),\
        "Violence_percent" : (total_fi * 100) / pred_image_cnt, \
        "Non_Violence_percent" : (total_no * 100) / pred_image_cnt , \
        "fi_count" : fi_count , \
        "no_count" : no_count
        }
    headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}
    sio.emit('pred_data', json_data)
    print('fi')
```

- 모델에서 현재 카메라의 신호를 100 프레임정도 받으면 이에 대한 예측을 진행
- 이러한 예측의 결과를 json으로 정리한 뒤 현재 시간과 폭력 예측이 최근 몇 번 일어났나를 계산해서 서버로 전송

```
# 100프레임마다 lstm 훈련을 실시한다.
if count == pred_image_cnt :
    # 2개의 값으로 구성된 배열이 나올 것
    # 0 : fight rate
    # 1 : non fight rate
    final_data = lstm_loaded_model.predict(np.array(pred_images))
```

코드 설명 (서버)

- 서버는 웹 소켓을 이용해서 데이터를 송수신
- 그리고 데이터를 서버 Log로 남겨두면서 동시에 웹 프론트로 보내고, 데이터베이스에 저장

```
io.emit('jsonData', data);
console.log(data);
async function change_video(){
  await video.update({
    starttime : data.start_time,
    endtime : data.end_time,
    fi_count : data.fi_count,
    non_count : data.no_count,
    videodate : data.start_time
  }
    ,{where: {locations : location}});
}
change_video();
```

```
io.on('connection', function (socket) {
  socket.join("videos");
  socket.on('image_data', function (data) {
      // base64 형식을 가진 데이터를 가져옵니다.
      var frame = Buffer.from(data, 'base64').toString();
      // 이 받은 데이터를 image라는 태그를 가진 데이터로써 웹 페이지에 뿌립니다.
      io.emit('image', frame);
      io.sockets.in("videos").emit('image_data', data);
  });
```

실외 CCTV 폭력 인식 시스템

- 과정
  - 캡스톤 팀장으로써, 실제 데이터를 제공하는 AIHub와 협력 및 그 분야에서 일하고 있는 지인과 인터뷰를 통해서 어떤 데이터가 평균적인지를 분석하고, 이를 정리하는 담당을 진행함
  - 개발 과정으로는 먼저 데이터를 폭력/비폭력으로 나눠서, 학습할 수 있게 데이터 전처리를 진행
  - 한 프레임 학습에 효과적인 CNN(VGG16)과 여러 장을 학습하는데 효과적인 LSTM 방식을 혼합해서 학습을 진행
  - 이렇게 학습된 모델의 결과와 OpenCV를 이용해서 받아온 영상을 프론트엔드(Nunjucks HTML)로 보낼 수 있도록, 프론트엔드 개발자와 WebSocket을 이용해서 구현

실외 CCTV 폭력 인식 시스템

- 결론
  - 최근 유행하는 NodeJS, WebSocket에 대한 개념 이해 및 실제 적용을 해봄
  - 직접 허가된 필드에 있는 데이터를 수집해보고, 인터뷰를 통해서 요구사항을 도출하는 소프트웨어공학적인 과정을 직접적으로 해봄

기타

- 개인 포트폴리오 사이트
  - 사용 기술 : Node.JS, AWS, AWS Code Deploy, AWS Route 53
  - 개발 기간 : 2021. 10 ~
  - 개발 목적
    - 개인 포트폴리오 관리 및 AWS 학습
  - 개발 과정
    - 최신 유행하는 백엔드 기술인 클라우드에 대한 감을 기르기 위해서 AWS에 인스턴스를 생성
    - 이 인스턴스에 NodeJS 서버를 올린 뒤 외부에서 쉽게 접속할 수 있게 Route 53에서 도메인 구매 및 적용
    - 본인이 작성한 코드 및 자원을 git에 올리면 바로 배포할 수 있게 Code Deploy를 활용
  - 사이트 주소 : Cha Seung Cheol's Portfolio (frokcreative.com)