**네트워크 게임 프로그래밍**

**PROJECT REPORT**

|  |
| --- |
| 게임공학과 2014180025 오한빛찬 |
| 게임공학과 2014182031 이영진 **( PM )** |
| 게임공학과 2014182034 이준호 |

1. 게임 설명

* 게임 장르

2D 탑 뷰 배틀 로얄

* 게임 내용

플레이어들이 각자의 배에 할당되어 자신의 무기와 이동 속도를 전략적으로 사용하여 최후까지 살아 남아야 하는 배틀 로얄 형식의 게임

* 게임 설정

최대 3인까지 플레이 가능

배들은 랜덤한 좌표에 배치 되어 서로 전투한다.

시간에 따라 바람의 방향과 세기가 변하며, 아이템이 랜덤하게 등장하며 바람 방향에 따라 선형적으로 이동한다.

전투 스테이지내엔 장애물 요소가 등장한다. 장애물 요소를 통해 포탄을 피할 수 있으며 장애물에 부딪히면 체력이 감소하고 속도가 0이 된다.

배끼리 부딪히면 각도에 맞춰 다른 데미지를 입고, 속도가 0이 된다.

상대방의 공격에 의해 피격 당할 시 체력이 감소하며, 체력이 0이 될 시 배는 침몰하며게임에서 패배한다.

배 한 척이 남을 때까지 진행된다.

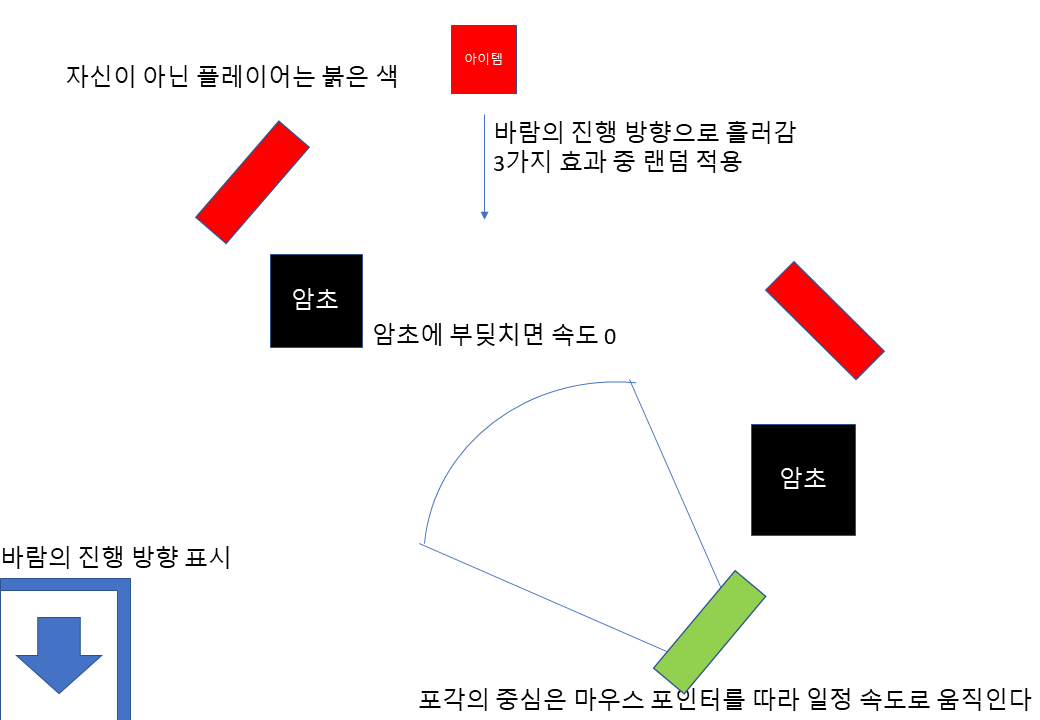
* 조작

W, S – 배의 속력 증감

A, D – 배의 이동 각도 증감

좌 클릭 – 마우스 포인터 위치로 포 발사

* 게임 진행 방식



2. 게임 구성 요소

\*1pixel 당 1m

윈도우

* 크기 : 1600x900 pixel

오브젝트

배

* 체력 : 100
* 최대 속도 : 60 m/s
* 크기 : 150 x 50 pixel
* 속도는 5단계로 구분 화살표 위아래로 바꿀 수 있고 속도는 유지가 됨

탄환

* 한 발 데미지 : 5
* 속도 : 300 m/s
* 크기 : 20 x 20 pixel

암초

* 크기 : (100 ~ 150) x (100 ~ 150) pixel
* 맵 곳곳에 박혀 있음

바람

* 속도 : 최소 0 m/s ~ 최대 30 m/s

아이템

* 시간에 따라 랜덤한 장소에 등장하여, 바람 방향에 따라 흘러감
* 체력 회복
* 속도 증가
* 데미지 증가

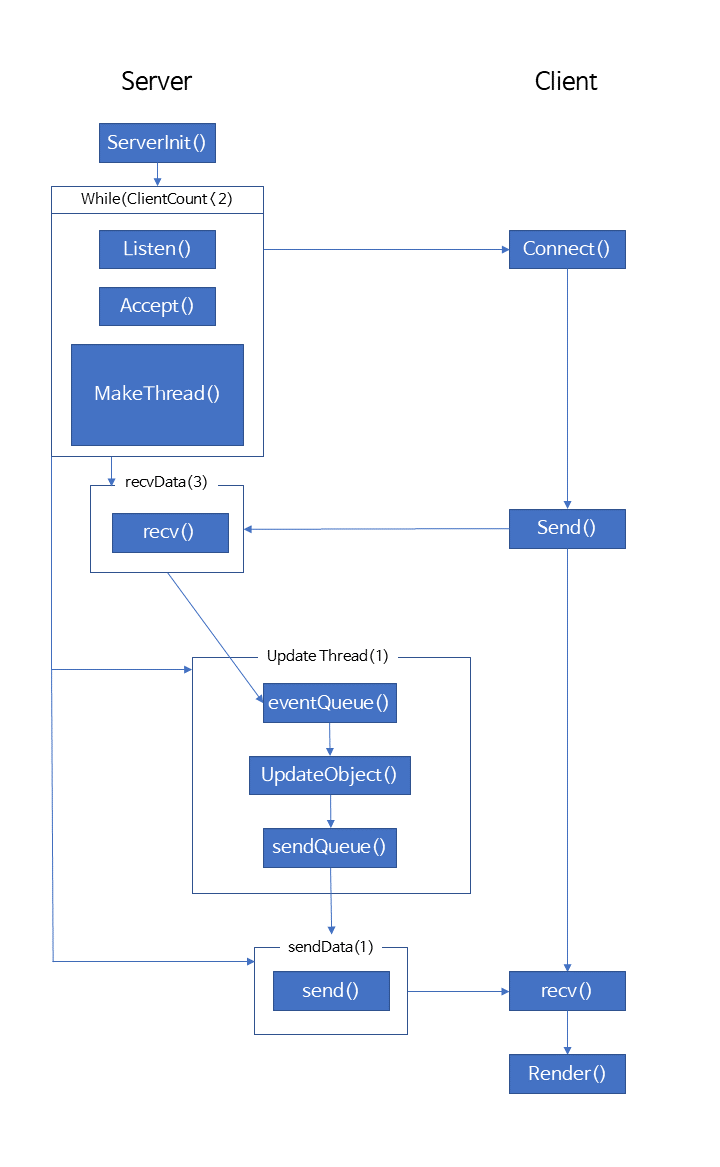
무기

* 마우스 클릭에 따라 발사 방식 전환
* 꾹 누를 때 – 전 탄 발사
* 단타 시 – 한 발 씩 발사
* 발사 각도에 따라 포탄의 사이즈가 변경됨

게임 진행

1. 최대 3인까지 플레이를 할 수 있다. 서버에 접속한 플레이어가 3명이 되면 게임 실행한다.
2. 게임 시작 시 각 플레이어들은 랜덤한 위치에 소환된다. 플레이어들은 랜덤으로 지정된 바람 중심으로 일정한 속도로 반시계 방향으로 회전한다.
3. 플레이어들은 배 좌 측면 지정된 범위의 포각을 가지고 있고 마우스 클릭으로 대포를 발사해 공격, 피격 시 체력 수치 하락, 체력이 0이 되면 패배 메시지 출력 후 게임 종료.
4. 마지막으로 살아남으면 게임 승리 이미지가 출력된다.

High Level



Server

1. Listen 상태에서 connect 요청이 오고 클라이언트가 모두 접속하면 각각의 recvData 함수를 만든다.
2. 접속한 클라이언트마다 ID를 부여하고 게임 시작 플래그를 보낸다.
3. 클라이언트에서 값이 변경되면 패킷 헤더를 전송하고 recvData 함수를 통해 서버가 받는다.
4. 서버는 클라이언트에게서 받은 패킷 헤드를 통해 패킷의 종류를 나누고 그에 따른 추가 데이터를 다시 받는다.
5. 추가 데이터를 eventQueue에 담는다.
6. updateThread 함수에서 eventQueue에 따른 연산을 실행한다
7. updateThread 함수에서 오브젝트들의 충돌 처리, 배의 속도와 진행 방향 변경 등의 연산을 한다.
8. updateThread 함수에서 모든 클라이언트에 전해줄 데이터가 있으면 sendQueue에 담는다.
9. sendQueue의 정보를 순서대로 sendData 함수에서 모든 클라이언트에게 전송한다..

Client

1. 서버에 connect 한다.
2. 서버에서 클라이언트 ID와 게임 시작 플래그와 다른 클라이언트의 ID를 받는다.
3. 클라이언트에서 값이 변경되면 recvData 함수에 데이터를 전송한다.
4. sendData 함수에서 데이터를 받고 그에 따라 값들을 수정한다.
5. 수정된 값들을 적용하여 게임 화면을 그린다.

Low Level

|  |
| --- |
| 패킷 데이터 구조체 |
| #pragma pack(1)  struct packetHead{ // 모든 주고받는 패킷들의 헤더  char id // 클라이언트 ID  char packetType // 패킷의 타입  }  struct simplePacket{ // 간단한 처리의 패킷용  char id // 클라이언트 ID  int value // 이벤트들의 값 ( 속도, 각도, 히트 등… )  char packetType // 패킷의 종류  }  struct shootPacket{ // 발사 이벤트의 패킷용  char id  int pos // 발사하는 클라이언트의 좌표  int targetPos // 발사 예정인 좌표  }  #pragma pack() |

|  |
| --- |
| 패킷 종류 |
| enum packetType : int {  E\_PACKET\_SPEED = 1, // 누군가의 속도 변경  E\_PACKET\_DEGREE, // 누군가의 이동 각도 변경  E\_PACKET\_SHOOT, // 누군가의 발사 요청  E\_PACKET\_HIT, // 누군가의 피격  E\_PACKET\_DIE, // 누군가의 사망  }; |

|  |
| --- |
| value 구조체 ( 클라이언트 내에서 사용, 간단한 x, y, z 를 표현하는데 사용 됨 ) |
| struct value {  value();  value(int);  value(float tx, float ty, float tz);  value operator\*(float scalar);  value operator+(const value& other);  bool operator<(float scalar);  float x, y, z;  }; |

|  |
| --- |
| Vector3D Class ( 클라이언트 내에서 사용, 물체의 x, y, z 와 방향을 나타내는데 사용 ) |
| class Vector3D {  Vector3D() = default;  Vector3D(float x, float y, float z);  Vector3D(value tv);  Vector3D& normalize();  Vector3D operator-(const Vector3D& rhs);  Vector3D& operator=(const Vector3D& rhs);  Vector3D operator+(const Vector3D& rhs);  Vector3D& operator+=(const Vector3D& rhs);  Vector3D operator\*(float \_scalar);  float getX() const;  float getY() const;  float getZ() const;  value getValue() const;  float distance(const Vector3D& rhs);  private:  float length();  value v;  }; |

|  |
| --- |
| object class ( 클라이언트 내에서 사용, 그려져야 할 모든 오브젝트는 이 객체형식 ) |
| class Object {  public:  Object() = default;  ~Object() = default;  void setPos(float x, float y, float z);  value getPos();  void setVolume(float x, float y, float z);  value getVolume();  void setMass(float m);  float getMass();  void update(float deltaTime);  void addForce(float x, float y, float z, float elapsedTime);  private:  value position;  value volume;  float mass;};  template <typename T>  void addObject(const value& pos, const value& volume, const value& velocity, float mass); |

|  |
| --- |
| Bullet Class |
| class bullet : public Object {  public:  bullet() = default;  void setVelocity(value v);  void getVelocity(value v);  private:  value velocity;  int damage;  }; |

|  |
| --- |
| ship class |
| class ship : public Object {  public:  ship() = default;  void setVelocity(value v);  void getVelocity(value v);  void shoot();  void recvDamage(int damage);  private:  value velocity;  int hp;  }; |

|  |
| --- |
| Event ( 모든 이벤트 패킷들의 집합체 ) |
| class Event {  public:  Event(simplePacket p) : var{ p } {}  Event(shootPacket p) : var{ p } {}  tuple<simplePacket\*, shootPacket\*> getPacket() {  return tuple(get\_if<simplePacket>(&var), get\_if<shootPacket>(&var));  }  private:  std::variant<simplePacket, shootPacket> var;  };  queue<Event> eventQueue;  queue<Event> sendQueue; |

|  |
| --- |
| PhysicsComponent ( 움직일 수 있는 오브젝트들이 갖는 컴포넌트) |
| class IPhysicsComponent : public IComponent {  public:  IPhysicsComponent() = default;  void process(Object\* o) override;  };  void IPhysicsComponent::process(Object\* o){  Vector3D velocity{ o->getVelocity() };  float time = D\_INPUT->getDeltaTime();  auto [x,y,z] = o->getPos() + velocity.getValue() \* time;  o->setPos(x, y, z);  } |

**서버**

* bool ServerInit()

WinSock 초기화, 소켓 생성, 소켓 주소를 할당, 만약 실패할 시 False 를 반환

* void sendStart()

모든 클라이언트가 접속 완료했을 시, 각각의 클라이언트에게 플레이어의 위치, 클라이언트의 ID와 시작 메시지를 보냄

* void makeThread()

클라이언트들에게 패킷을 전달받을 recvData 스레드를 만든다

오브젝트들을 업데이트할 updateThread 를 만든다.

데이터 전송이 필요할 경우, 전송을 위한 sendData 스레드를 만든다.

* void recvData(SOCKET s)

데이터 변경이 일어난 클라이언트에서 보낸 데이터를 받음

recvData 는 각 클라이언트의 packetHead를 대기하며, 받음과 동시에 head.packetType 을 통해 어떤 정보의 패킷인지를 확인한 후, 정보에 맞는 패킷을 다시 수신한 다음 이벤트 큐에 밀어넣음

void recvData(SOCKET s){

while (1) {

packetHead head;

int headSize = sizeof(head);

int retval = recvn(s, (char\*)&head, headSize, 0); // head 가 올때까지 기다림

switch (head.packetType) {

case E\_PACKET\_DEGREE: // 만약 각도 변경 요청일경우

simplePacket pack;

retval = recvn(s, (char\*)&pack, sizeof(pack), 0);

eventQueue.push(pack);

break;

}

}

* void sendData()

서버가 이벤트와 충돌처리를 업데이트하면서, 데이터를 전송해야 할 일이 생긴다면, sendQueue 에 전송할 메시지들이 들어가게 됨

sendData는 sendQueue가 비어있지 않다면, 모든 클라이언트들에게 전송을 계속 한다.

void sendData() {

if (!sendQueue.empty()) {

auto e = sendQueue.front();

auto[e1, e2] = e.getPacket();

if (e1 != nullptr)

// simplePacketEvent 라면 이곳에서 처리

else

// shootPacketEvent 라면 이곳에서 처리

sendQueue.pop();

}

* void updateThread()

클라이언트와의 동기화를 위해, 모든 연산을 처리하는 곳

배들의 이동, 충돌, 이벤트, 2초마다 클라이언트 정보 전송 등 모든 작업을 담당함

만약 보내야할 이벤트가 생긴다면, 이곳에서 sendQueue에 입력시킴

void updateThread() {

while (1) {

if (!eventQueue.empty()) {

auto e = eventQueue.front();

auto[e1, e2] = e.getPacket(); // 패킷을 열어봄

if (e1 != nullptr) {

//e1 에 대한 처리

}

else {

//e2에 대한 처리

}

eventQueue.pop(); // eventQueue에서는 이미 처리끝냈으므로 파괴

}

updateObject(); // 모든 오브젝트들에 대한 이동, 충돌 처리

// 만약 event Packet 에 대한 처리를 끝냈고, 오브젝트에 대한 업데이트를

다 끝냈는데 보내야할 데이터가 있다면 sendQueue에 밀어넣음

sendQueue.push(Event());

}

}

* Tuple<WORD, WORD> intToWord(int value)

int 타입의 변수를 WORD 두개로 나누어 Tuple에 넣는다.

* Int wordToInt(WORD d1, WORD d2)

WORD 타입 변수 두개를 int 타입 하나로 합친다.

**클라이언트**

* void recvStart()

서버에서 플레이어의 위치, 클라이언트의 ID와 시작 메시지를 받는다.

* void recvData()

서버에서 보낸 값들을 받고, 수정해야할 정보들을 수정함

* void sendData()

클라이언트에서 변경된 값을 서버에 보낸다.

* void Render()

연산처리 된 값들을 화면에 반영한다.

* Tuple<WORD, WORD> intToWord(int value)

int 타입의 변수를 WORD 두개로 나누어 Tuple에 넣는다.

* Int wordToInt(WORD d1,WORD d2)

WORD 타입 변수 두개를 int 타입 하나로 합친다.

* void addObject(value pos, value volume, value velocity, float mass)

각 각의 옵션들 ( 좌표, 부피, 속력, 질량 ) 들을 받아, 새로운 타입의 오브젝트를 생성함

생성한 오브젝트들은 Object 라는 class에서 파생된 ship, obstacle, item, … 이 될 것이며,

충돌처리가 필요한 오브젝트는 파생형식으로, 충돌처리가 필요 없는 ( 배경사진 등 ) 오브젝트는 Object 라는 class를 가지고 만들어 질 것이다.

template <typename T>

void addObject(const value& pos, const value& volume, const value& velocity, float mass){

Object\* o = new T();

o->setPos(pos.x, pos.y, pos.z);

o->setVolume(volume.x , volume.y, volume.z);

o->setVelocity(velocity.x, velocity.y, velocity.z);

o->setMass(mass);

// list<Object\*> v

v.push\_back(o);

} // 지금은 의사 코드이며 template 으로 역할을 하지만, 개발시엔 각 타입에 맞게 Factory 로 추상화 시킬 것

addObject<ship>(value{0,0,0}, value{1,1,0}, velocity{0,0,0}, 1.0f) // 이런식으로 사용

**스레드 함수**

* void makeThread()

recvData 함수

updateThread 함수

sendData 함수

를 만들어낸다.

* recvData 함수

서버가 모든 클라이언트들에게 정보 (이벤트)를 받는 부분

* sendData 함수

연결되어 있는 클라이언트 모두에게 데이터를 보낸다

* updateThread 함수

eventQueue에 따른 처리와 오브젝트들의 업데이트를 하고 sendQueue에 데이터를 넣는다.

**동기화 방식**

recvData 함수에서 eventQueue를 사용할 때 임계영역을 설정하여 동기화를 한다.

2초 마다 서버에서 클라이언트들에게 오브젝트들의 좌표 값을 다시 보내주어 클라이언트 간의 오차를 수정한다.

개발환경 및 세부계획

1. 개발환경

동작 OS : Windows 10

IDE : Visual Studio 2017

사용 라이브러리 및 헤더 : Window API / WinSock2.h / OpenGL

개발 언어 : C / C++

1. 역할분담

2014180025 오한빛찬

게임 개발(배 오브젝트)

서버 개발(sendData 함수 / 서버 내에서의 충돌체크)

클라이언트 / 서버 프레임 워크 제작

2014182031 이영진 **( PM )**

게임 개발(암초, 아이템 오브젝트 / 테스트 용 충돌체크)

서버 개발(recvData 함수 / 서버 내에서의 충돌체크)

2014182034 이준호

게임 개발(포탄, 바람 오브젝트 / 맵 제작)

서버 개발(updateThread 함수 / 서버 내에서의 충돌체크)

개발일정

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 오한빛찬 | | 이영진 | 이준호 |
| ~ 10 / 29 | 프로젝트 기획서 작성 / 수정 / 검토 | | | |
| 10 / 30 | 프로젝트 기획서 제출 및 피드백 | | | |
| 11 / 1 | 프로젝트 기획서 수정 작업 | | | |
| 11 / 2 |
| 11 / 3 |
| 11 / 4 | 프로젝트 기획서 재제출 | | | |
| 11 / 5 | 1주차(게임 개발) | | | |
| 11 / 6 | Object 클래스 정의  내부 메소드 정의 | |  | |
| 11 / 7 | 암초(Object 상속) 정의 및 구현 | 포탄(Object 상속) 정의 및 구현 |
| 11 / 8 | 배(Object 상속) 정의 | | 아이템(Object 상속) 정의 및 구현 | 포탄의 물리 공식 |
| 11 / 9 |  | | | |
| 11 / 10 | 2주차(게임 개발 / 서버 개발) | | | |
| 11 / 11 | 병합 작업 후 수정 | | | |
| 11 / 12 | 배 이동구현, 체력 바 생성 | | 테스트 용 충돌 검사 함수 정의 | 리소스 제작 |
| 11 / 13 | serverInit() | | 클라이언트 id,  게임시작 플래그 | 데이터 구조체 |
| 11 / 14 | sendQueue | | eventQueue | updateThread() |
| 11 / 15 | sendData() | | recvData() |
| 11 / 16 | 병합 후 테스트, 버그 수정 | | | |
| 11 / 17 | 3주차(클라이언트 개발) | | | |
| 11 / 18 | 서버와 클라이언트에서 데이터 송수신 테스트 | | | |
| 11 / 19 | Value 구조체 정의 및 구현 | | Vector3d 클래스 정의 및 구현 | Packet, Event |
| 11 / 20 |  | | | |
| 11 / 21 | Keyinput() | | 물리 함수 | sendObjectData() |
| 11 / 22 | 배의 이동을 서버에 보내기  sendMoveEvent() | | 서버에서 데이터를 받는다.  recvData() in client | 포탄의 발사를 서버에 보내기  sendShootEvent() |
| 11 / 23 |  | | | |
| 11 / 24 | 4주차(서버에서의 충돌 검사) | | | |
| 11 / 25 | collideItemAndObject() | | collidePlayerAndItem() | collideBulletAndObject() |
| 11 / 26 |  | | | |
| 11 / 27 | collidePlayerAndPlayer() | | collidePlayerAndObject() | collidePlayerAndBullet() |
| 11 / 28 |  | | | |
| 11 / 29 | 데이터 송수신에 의한 충돌 테스트 | | | |
| 11 / 30 |  | | | |
| 12 / 1 | 5주차(동기화 구현) | | | |
| 12 / 2 | 동기화 설계 | | | |
| 12 / 3 | eventQueue 사용시 임계영역 설정 | 좌표 값 재송신 | | |
| 12 / 4 | 동기화 테스트, 버그 수정 | | | |
| 12 / 5 |  | | | |
| 12 / 6 | 최적화  (서버와 클라이언트의 메모리와 속도) | | | |
| 12 / 7 |
| 12 / 8 | 6주차(최종점검) | | | |
| 12 / 9 | 최종 코드와 리포트 점검 및 마무리 | | | |
| 12 / 10 |
| 12 / 11 | 프로젝트 제출 | | | |