**게임개발자전문가과정**

**과 목 명 : DirectX 2D Game Project**

**능력단위 : DirectX 2D**

**제출일자 : 2022년 10월 06일**

**포트폴리오 : TopGun**

**작 성 자 : 신승빈**

**<제출내역>**

1. **OctTree를 이용한 충돌 검출 시스템**
2. **충돌 검출 시스템 구현 분석 및 세부 문서**

|  |
| --- |
|  |

**< Contents >**

1. **프로젝트 소개 및 개요**
2. 프로젝트 소개
3. 프로젝트 주요 기술
4. **프로젝트 설계 및 다이어그램**
5. 시퀀스 다이어그램
6. 클래스 다이어그램

2-1) 프로젝트에서 사용된 변수 역할

2-2) 프로젝트에서 사용된 함수 역할

1. **최종 결과 및 추가 내용**
2. 범용성/유연성/확장성/간결성 고려하여 추가된 내용
   1. 현재 프로젝트 설계(구현 내용)
   2. 향후 개발 내용
3. 최종 결과
4. **프로젝트 소개 및 개요**
5. **프로젝트 소개**

본 프로젝트는 객체지향적 설계를 중점으로 프로젝트 단위의 모듈 라이브러리와 템플릿을 이용한 충돌 검출 및 물리 연산 시스템을 구현하였다.

1. **프로젝트 주요 기술**

* **공간 분할**

인게임의 공간이 넓어질수록 그 안을 채우는 Object의 수 역시 기하급수적으로 늘어나게 된다. 이러한 Object들의 충돌을 하나하나 검사하는 것은 하드웨어의 성능을 많이 소모하기 때문에 가까운 거리에 있는 Object들만 충돌을 검사하는 것이 좋다. 공간분할의 종류는 매우 다양한데, 본 프로젝트에서는 다양한 공간분할 알고리즘을 수용할 수 있도록 알고리즘의 구현을 CollisionTree와 분리하였다. CollisionTree의 사용자는 임의의 공간 Volume을 받아온 후 사용자가 원하는 방식으로 공간을 분할하는 SpaceDivider에 알고리즘을 구현하여 공간 분할을 수행할 수 있다. 이번 프로젝트에서는 QuadTree와 OctTree를 직접 구현해보았다.

* **Template**

게임의 공간은 크게 2D와 3D로 나눠지며 이를 구현하는 방식은 다양하다. 물리적 세계관의 차원은 다를 수 있지만, 동일 차원 내부에서 작동하는 매커니즘은 기본적으로 차원에 따라 큰 차이가 없기 때문에 본 프로젝트에서는 코드의 재사용성을 높이기 위해 템플릿을 사용하여 구현해보았다. 또한 템플릿을 사용할 경우 코드의 작성이 번거로워질 수 있기 때문에, 코드 작성의 편의를 위해 using구문을 사용하여 사용자의 간편한 코드의 작성을 유도하였다.

* **팩토리 메서드 패턴**

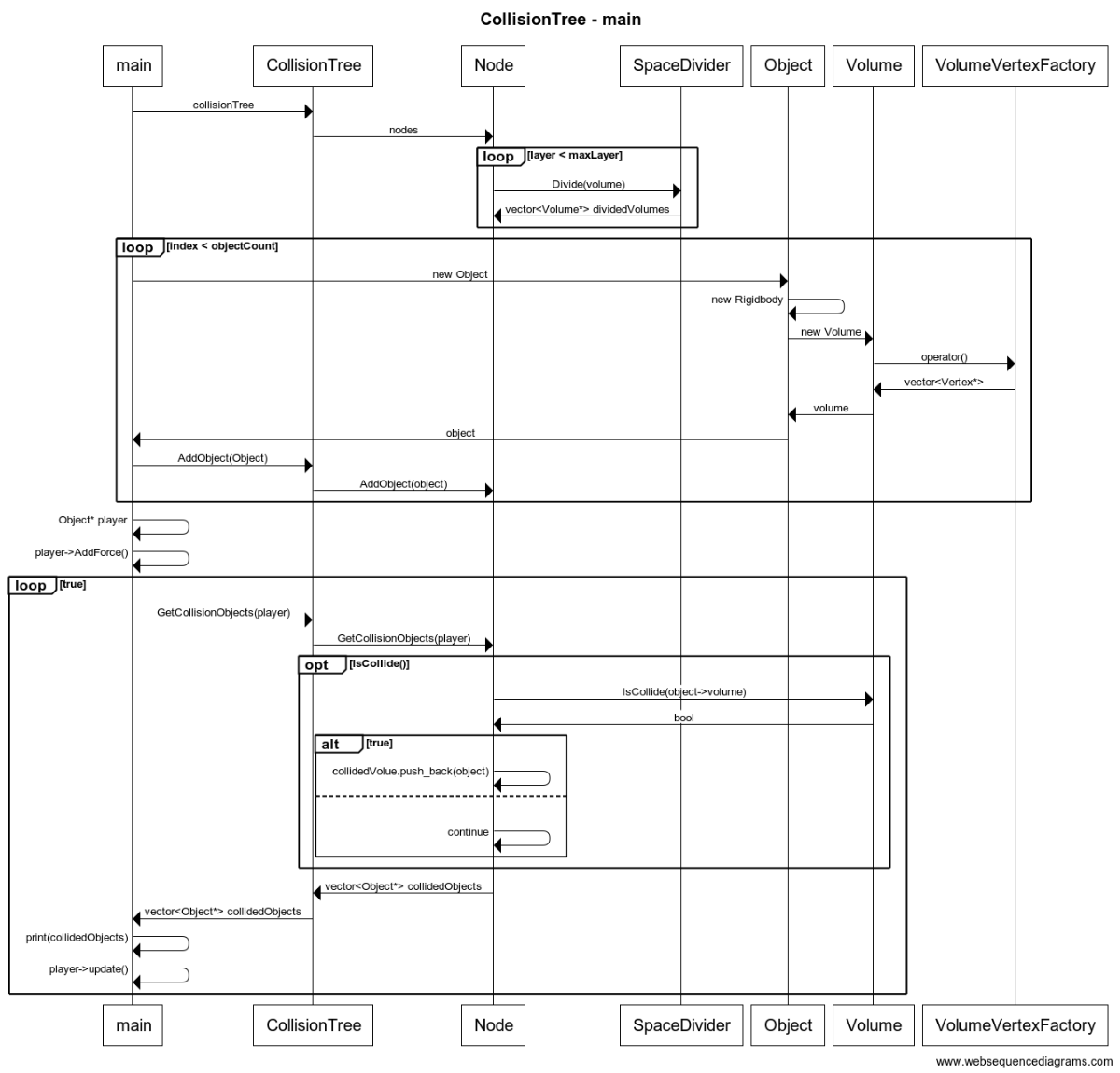
Volume의 경우 Vertex를 통해 자신이 점유하고 있는 공간을 나타낸다는 공통점이 있지만, 각 Volume의 SubClass들에 따라서 Vertex의 개수와 위치가 다 다르게 결정된다. Vertex집합인 vertexes를 protected로 선언하는 방법도 가능하지만, 객체지향과 캡슐화에 대한 관점에 집중하기 위해 VertexFactory를 통한 vertexes의 값을 가져오는 방식으로 구현하였다.

마찬가지로, CollisionTree의 경우 주어진 Volume을 알고리즘에 맞게 분할한 후, 분할된 Volume을 저장하는 과정이 필요한데, Volume을 분할하는 SpaceDivider class를 Factory 패턴으로 구현하여 분할된 Volume의 생성을 책임지도록 하였다.

* **모듈 단위 라이브러리 및 테스트**

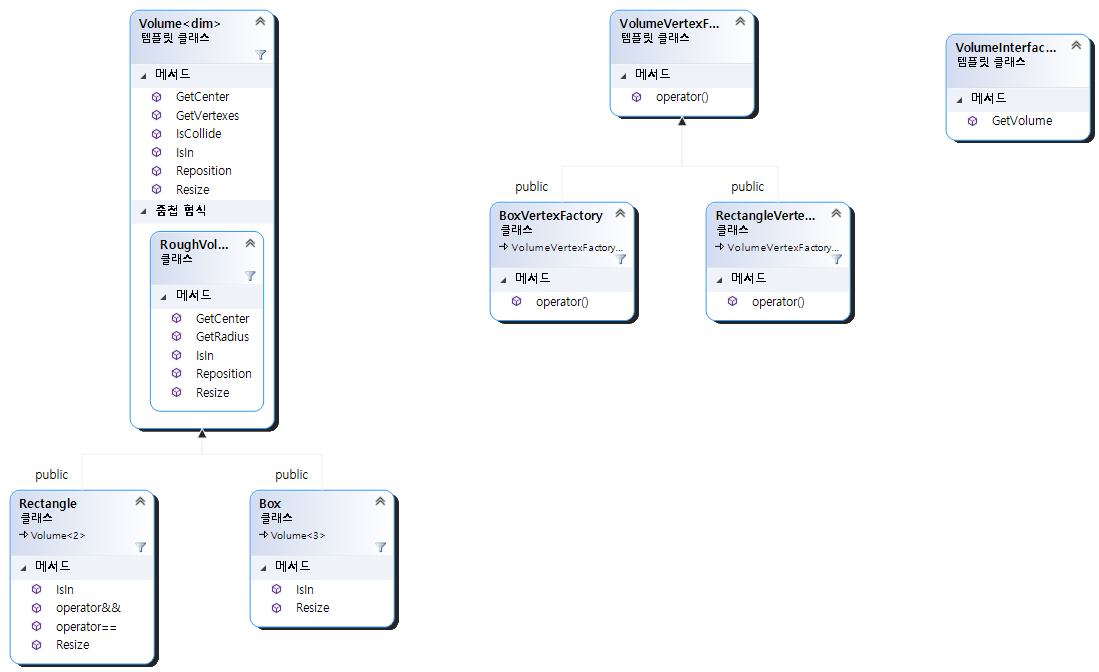
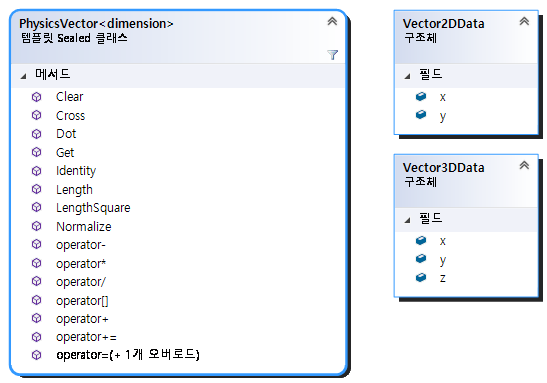
하나의 프로그램은 다양한 모듈로 이루어져 있다. 또한, 프로젝트를 이루는 모듈은 다양한 분야에서 재사용이 가능하기 때문에 각 모듈을 응집력 있게 설계하여 라이브러리로 만드는 것이 좋다. 이렇게 응집력 있게 작성된 라이브러리는 그 작동이 라이브러리 단위로 정상적으로 작동이 되어야 한다는 것을 보장해야 하는데, 이를 위해 Unit Test를 진행하여 이번 프로젝트에 사용될 라이브러리들이 Unit단위로 정상 작동함을 보장하였다.

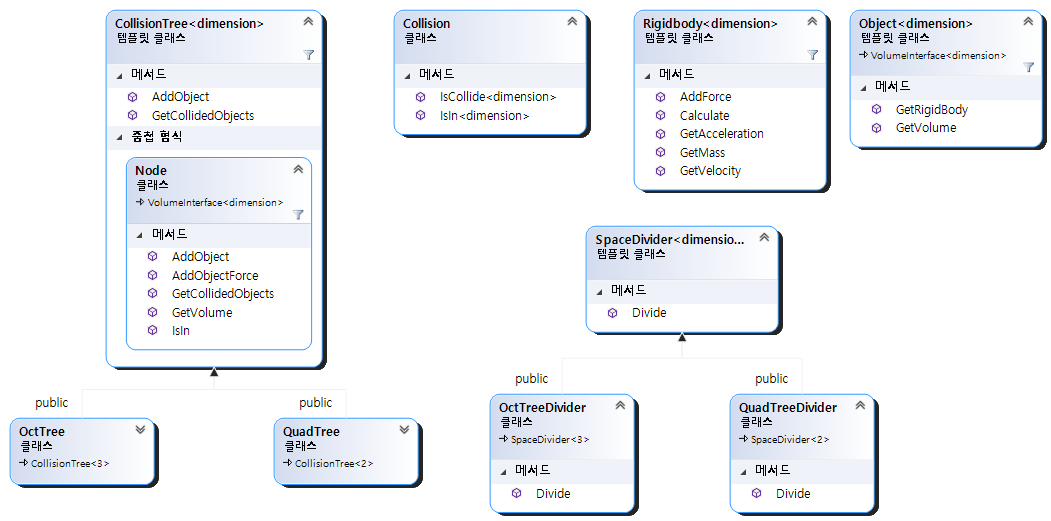
1. **프로젝트 설계 및 다이어그램**
2. **시퀀스 다이어그램(Sequence Diagram)**



[그림 1-1] 시퀀스 다이어그램(Sequence Diagram)

1. **클래스 다이어그램(Class Diagram)**





[그림 1-2] 클래스 다이어그램(Class Diagram)

**2-1) 프로젝트에서 사용된 변수 역할**

|  |  |  |
| --- | --- | --- |
| **클래스명** | **변수** | **내용** |
| **PhysicsVector** | float\* \_coordinate | 차원에 독립적인 물리벡터의 값을 저장하기 위한 변수 |
| **RoughVolume** | PhysicsVector<dim> \_center | Volume의 중심 위치 |
| float \_radius | Volume의 중심으로부터 가장 먼 곳에 위치한 vertex로부터의 길이 |
| **Volume** | RoughVolume \_roughVolume | 빠른 계산을 위한 RoughVolume |
| std::vector<PhysicsVector<dim>\*> \_vertexes | Volume을 이루는 정점들 |
| **Rectangle** | Vector2D \_lt  Vector2D \_rb | Left-top 위치의 정점  Right-bottom 위치의 정점 |
| float \_width  float \_height | Rectangle의 너비와 높이 |
| **Box** | Vector3D \_ltf  Vector3D \_rbb | Left-Top-Front 위치의 정점  Right-Bottom-Back 위치의 정점 |
| **Rigidbody** | float \_mass | 강체의 질량 |
| PhysicsVector<dimension> \_acc | 강체의 가속도 |
| PhysicsVector<dimension> \_vel | 강체의 속도 |
| **Object** | Volume<dimension>\* \_volume | Object의 충돌체 속성 |
| Rigidbody<dimension> \_rigidBody | Object의 강체 속성 |
| **Node** | const int \_divideCount | 노드를 분할하는 개수 |
| Node\*\* \_child | 노드의 분할된 자식 |
| Volume<dimension>\* \_space | 현 노드의 Volume |
| std::vector<Object<dimension>\*> \_staticObjects  std::vector<Object<dimension>\*> \_dynamicObjects | 현 노드 내부에 존재하는 Object의 개수 |
| **CollisionTree** | Node\* \_root | Tree의 root node |
| std::vector<Object<dimension>\*> \_dynamicObjectList | Tree에 저장된 동적 object들 |

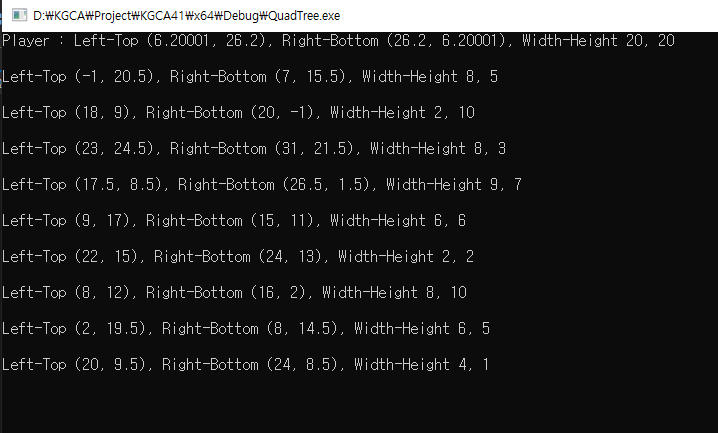
[표 1-1] 프로젝트의 클래스 변수

**2-2) 프로젝트에서 사용된 함수 역할**

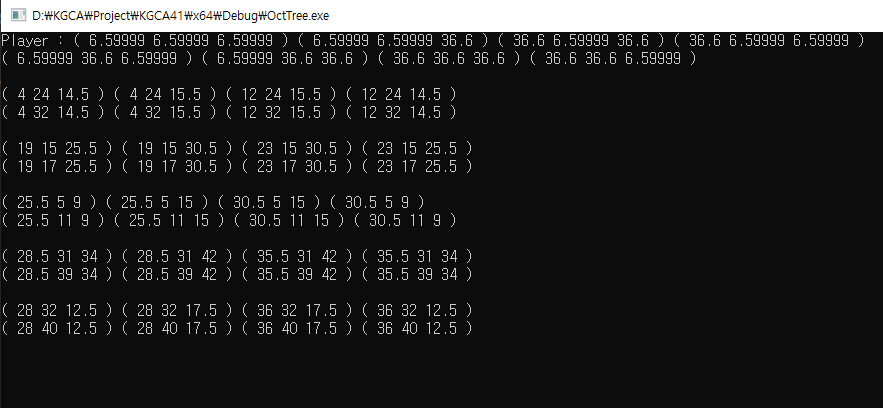
|  |  |  |
| --- | --- | --- |
| **클래스명** | **함수** | **내용** |
| **PhysicsVector** | void DimensionCheck(const PhysicsVector& vec) const | 벡터간의 연산 전 차원 검사  차원이 맞지 않을 경우 DimensionMatchException 생성 |
| template<typename UnaryFunction> void RepeatAction(UnaryFunction func) const | 벡터 내부를 순회하며 단항 연산 가상 함수를 이용하여 작동 |
| template<typename BinomialFunction> void RepeatAction(const PhysicsVector& vec, BinomialFunction func) const | 벡터 내부를 순회하며 이항 연산 가상 함수를 이용하여 작동 |
| float Length() const | 벡터의 길이 |
| float Dot(const PhysicsVector& vec) | 벡터의 내적 |
| PhysicsVector operator+(const PhysicsVector& vec) const  void operator+=(const PhysicsVector& vec)  PhysicsVector operator-(const PhysicsVector& vec) const  PhysicsVector operator\*(float scalar) const  PhysicsVector operator/(float scalar) const  void operator=(const PhysicsVector& copy)  void operator=(PhysicsVector&& move) | 벡터의 연산 |
| void Normalize() | 벡터의 정규화 |
| PhysicsVector Identity() | 정규 벡터 |
| float Get(int dimension) const | 벡터 element 값에 접근하는 함수 |
| **RoughVolume** | bool IsIn(const PhysicsVector<dim>& v) const | 충돌체 내부에 점이 존재하는지 점검하는 함수 |
| void Resize(float radius) | 충돌체의 크기를 조절하는 함수 |
| void Reposition(PhysicsVector<dim>& pos) | 충돌체의 중점을 바꾸는 함수 |
| const PhysicsVector<dim>& GetCenter() const | 충돌체의 중점을 가져오는 함수 |
| float GetRadius() const | 충돌체의 반지름을 가져오는 함수 |
| **VolumeVertexFactory** | virtual std::vector<PhysicsVector<dim>\*> operator()() | 충돌체를 이루는 정점을 생성하여 반환하는 함수 |
| **Volume** | bool IsCollide(const Volume& volume) const | 충돌체 간의 충돌 여부를 반환하는 함수 |
| virtual bool IsIn(const PhysicsVector<dim>& coordinate, const PhysicsVector<dim>& v) const | 충돌체 내부에 점이 존재하는지 점검하는 함수 |
| const std::vector<PhysicsVector<dim>\*>& GetVertexes() const | 충돌체를 이루는 정점을 반환하는 함수 |
| const PhysicsVector<dim>& GetCenter() const | 충돌체의 중점 |
| **Rectangle** | float GetLeft() const  float GetTop() const  float GetRight() const  float GetBottom() const  float GetWidth() const  float GetHeight() const | Rectangle을 이루는 속성 값을 가져오는 함수 |
| bool operator==(const Rectangle& rect) const  bool operator&&(const Rectangle& rect) const | Rectangle 계산 함수 |
| **Rigidbody** | void AddForce(const PhysicsVector<dimension>& newForce) | 강체에 힘을 더하는 함수 |
| void Calculate(float deltaTime) | 강체가 가지고 있는 가속도를 기반으로 속도를 계산하는 함수 |
| float GetMass()  PhysicsVector<dimension> GetAcceleration()  PhysicsVector<dimension> GetVelocity() | 강체의 속성을 가져오는 함수 |
| **Object** | Volume<dimension>\* GetVolume()  Rigidbody<dimension>\* GetRigidBody() | Object의 속성을 가져오는 함수 |
| **SpaceDivider** | virtual std::vector<Volume<dimension>\*> Divide(const Volume<dimension>& volume) | 임의의 Volume을 알고리즘에 맞게 분할하여 반환하는 함수 |
| **Node** | bool IsIn(Object<dimension>\* object) | Node 내부에 Object가 포함되는지 확인하는 함수 |
| void AddStaticObject(Object<dimension>\* object)  void AddDynamicObject(Object<dimension>\* object) | Node 내부에 Object가 포함되는지 점검하는 함수 |
| void AddStaticObjectForce(Object<dimension>\* object)  void AddDynamicObjectForce(Object<dimension>\* object) | Root Node같이 만약 Object가 Node에 완전히 포함되지 않더라도 강제로 추가하는 함수 |
| std::vector<Object<dimension>\*> GetCollidedObjects(Object<dimension>\* obj) | 자신의 내부에 있는 Object들 중 매개변수로 주어진 Object와 충돌한 Object들을 반환하는 함수 |
| void ClearDynamicObjects() | 이동 계산을 위해 dynamic object를 초기화하는 함수 |
| **CollisionTree** | void AddStaticObject(Object<dimension>\* object)  void AddDynamicObject(Object<dimension>\* object) | CollisionTree 내부에 Object를 추가하는 함수 |
| void UpdateDynamicObject(Object<dimension>\* object) | 동적 Object의 Node위치를 갱신하는 함수 |
| void Frame() | CollisionTree의 Frame을 갱신하는 함수 |
| std::vector<Object<dimension>\*> GetCollidedObjects(Object<dimension>\* obj) | CollisionTree 내부에 존재하는 Object들 중 매개변수로 주어진 Object와 충돌하는 Object들을 반환하는 함수 |

[표 1-2] 프로젝트의 클래스 함수

1. **최종 결과 및 추가 내용**
2. **범용성/유연성/확장성/간결성 고려하여 추가된 내용**
   1. **현재 프로젝트 설계(구현 내용)**
3. Vector를 표현하는 PhysicsVector는 template으로 구현되어 있으며, Vector에서 공통적으로 사용하는 연산들 중 외적을 제외한 대부분의 기능이 구현되어 있음. 사용자의 편의를 위해 흔히 사용하는 2D와 3D의 경우 using을 사용하여 Vector2D, Vector3D라는 형식으로도 사용이 가능함.
4. 충돌체를 의미하는 Volume의 경우 Rectangle, Circle, Box 등 Subclass의 Vertex를 지정하는 공통된 방법이 존재하지 않으므로 원하는 Subclass의 적절한 Vertex를 생산하는 VolumeVertexFactory를 이용.
5. Volume사이의 충돌의 경우 Volume과 다른 Volume을 이루는 Vertex사이의 충돌을 기반으로 계산하는데, 이 경우 연산 비용이 비싼 단점이 있음. 이를 완화하기 위해 RoughVolume이라는 원 기반 충돌 감지 체계를 사용함
6. CollisionTree의 공간분할을 위해 SpaceDivider라는 팩토리 메서드 패턴을 사용.
   1. **향후 개발 내용**
7. 2D와 3D를 통합할 수 있는 외적 메서드의 구현 방법에 대한 고민이 필요
8. Rectangle to Rectangle, Circle to Circle의 경우 빠르게 계산하는 방법이 존재하지만 Volume to Volume의 경우 직관적인 방법이 존재하지 않아서, 현재 Volume과 Volume을 이루는 각 Vertex들의 충돌 여부를 탐지하는 방향으로 구현. SAT나 GJK로 변경 필요
9. 생성자의 경우 매개변수로 lvalue reference와 rvalue reference를 받을 수 있는데, 변수의 개수가 많아질 경우 매 경우의 수를 모두 구현해야 함. 이를 쉽게 처리하는 방법이 없는지 연구 필요
10. 현재 2D Object는 QuadTree에 3D Object는 OctTree에 저장되어야만 하는데, 이 부분에 대해 서로 호환가능하도록 구현 필요
11. Dynamic Object의 경우 이동에 의해 소속된 Node의 위치가 변경될 수 있는데 학원의 경우 Tree 내부의 모든 Node에서 Dynamic Object를 삭제하는 방식으로 알려줬음. 현재 이 부분은 비효율적이다 생각하여 옵저버 패턴을 사용하여 이동한 Object만 점검하는 절차에 대해 고민 중
12. **최종 결과**



QuadTree



OctTree