**MOBA(Multiplayer Online Battle Arena) 게임 개발**

1. **Cocos2d-x framework을 이용한 MOBA(Multiplayer Online Battle Arena) 게임 개발**
2. **구성원과 역할**

구성원: 임인섭 양지석 유진선 주재철

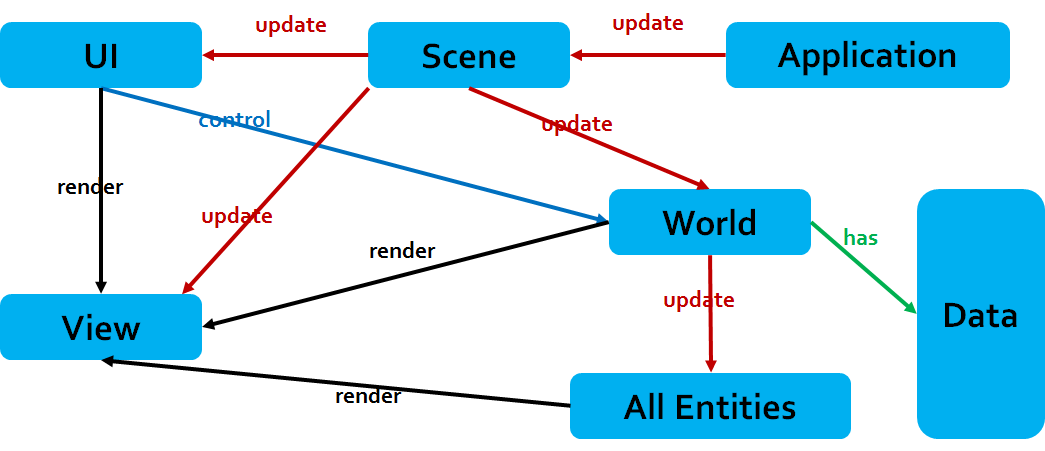
|  |  |
| --- | --- |
| 임인섭 | Geometry Library, Utility Library, Collision System, Goal System, Navigation System, Message System, Cell Space Partition System, Debug Renderer, Target System, Weapon System, Steering System, Animation Module, Sensory Memory, Timer Module, World, Map, Game Server, Packet 직렬화, Packet 정의, Packet 교환 Scenario 정의, Scenario 동기화 |
| 양지석 | User Interface, Control Module, Skill Entity(Penetrate, None Penetrate, Target, None Target, Continuous ..) 세분화 |
| 유진선 | Game Flow, Trigger System, Item Entity(Combination Item, Usable Item.. ), Structure Entity(Collision Area, Imaged Structure), Character Entity(User Character, Base Character, Creep Character, Boss Character..) |
| 주재철 | Tile Map 구현, Plist, Packed Image, Navigation Graph 만들기,  Particle System 구현, Skill Entitiy 세분화 |

1. **주제의 선정 배경**

실제로 많은 사람들이 즐기고 있는 장르인 MOBA(Multiplayer Online Battle Arena) 게임을 직접 개발해 봄으로써 학생 때 배운 다양한 지식을 활용해보는 기회를 갖는다. 또한 네트워크를 통해 여러 명의 플레이어가 동시에 즐길 수 있도록 개발해 본다.

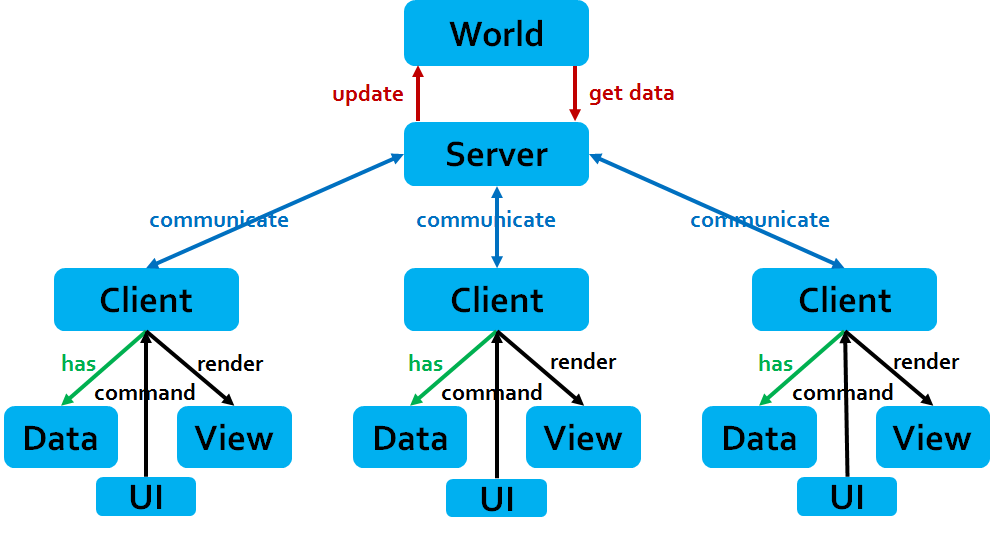
1. **시스템 구성도**

* 클라이언트만 고려한 1차 설계



게임이 시작되면 가장 먼저 Application이 실행되고 Application은 Scene을 실행시킨다.  
Application은 Scene을 [전환할](http://www.incruit.com/tools/spell/) 수 있으므로, Login 화면, Loading 화면, Matching 화면, 게임 화면 등으로 전환이 가능하다.  
기본적으로 게임의 모든 흐름이 돌기 위해서는 update와 render가 필요하다.  
Update를 통해 모든 entity는 다음 상태로 갱신된다. 그리고 render를 통해 모든 entity는 View에 그려지게 된다.  
Scene은 View, World, UI에 대해 [지속해서](http://www.incruit.com/tools/spell/) [update를](http://www.incruit.com/tools/spell/) 해준다.  
UI는 [update 됨으로써](http://www.incruit.com/tools/spell/) event를 다룰 수 있게 되고, View는 update를 통해 화면에 장면을 출력 하게 된다. 그리고 world의 update가 일어나면서 [모든 entity의](http://www.incruit.com/tools/spell/) 갱신이 이루어진다. [각 entity는](http://www.incruit.com/tools/spell/) 게임 진행 중에 생성과 소멸을 반복하는데 이에 대해 필요한 data는 world에 [적재](http://www.incruit.com/tools/spell/)된다.  
이러한 시스템의 구조는 사실 서버를 고려하지 않는 순수한 클라이언트의 설계 구조이다. 즉, 서버가 고려된다면 시스템은 이와 달라져야 한다. 하지만 개발을 [진행하면서](http://www.incruit.com/tools/spell/) world의 크기가 상당히 크기 때문에 먼저 클라이언트에 대한 설계로 [완성하고](http://www.incruit.com/tools/spell/) 다시 서버를 고려한 구조로 [나눌](http://www.incruit.com/tools/spell/) 계획이다.

* 서버와 클라이언트를 모두 고려한 2차 설계



1차로 클라이언트에 대한 [각 모듈이](http://www.incruit.com/tools/spell/) 완성되었다면, 이를 서버까지 고려한 더 큰 시스템으로 확장시켜야 한다. 서버가 고려된다면, 모든 [entity 간의](http://www.incruit.com/tools/spell/) 상호작용은 [서버 내에서](http://www.incruit.com/tools/spell/) 이루어져야 한다[. 왜냐하면, 각](http://www.incruit.com/tools/spell/) 플레이어에 대한 조작 정보는 서버에 모일 것이고, 모든 정보가 모였을 때 [비로소](http://www.incruit.com/tools/spell/) entity간 상호작용을 할 수 있기 때문이다.  
entity간 상호작용은 [update 과정에서](http://www.incruit.com/tools/spell/) 이루어지는 [모든 동작을](http://www.incruit.com/tools/spell/) 의미한다. [Entity 간](http://www.incruit.com/tools/spell/) 충돌처리, Graph 탐색을 통한 경로 획득, 복합 목적을 통한 game [AI](http://www.incruit.com/tools/spell/) 등등 모든 update를 통한 과정들이 서버 내부에서 이루어지고 서버는 완성된 data들을 만들어 각 [클라이언트들에](http://www.incruit.com/tools/spell/) 뿌려준다. 여기서 완성된 data는 rendering에 필요한 위치, 상태 값과 같은 [모든 data다](http://www.incruit.com/tools/spell/). 결과적으로 클라이언트는 이전 1차 설계 때와 달리 [서버로 와](http://www.incruit.com/tools/spell/) data를 [주고받으며](http://www.incruit.com/tools/spell/), 받은 정보를 토대로 화면에 출력만 해주도록 변경되어야 한다. 결국 2차 설계는 1차 설계를 update와 render로 완벽하게 분리한 설계가 될 것이다.

1. **개발 기능**

게임 내부 동작에 있어서 전반적으로 필요한 다양한 장치들이 있다. Library, Module, System으로 분류를 했는데, Library는 함수의 집합을 나타내고, Module은 원하는 특정한 기능을 담당하는 하나의 객체를 나타내고, System은 특정한 기능들을 위해 상호작용하는 객체들의 집합을 나타낸다. 먼저 이러한 장치들을 구현하고, 각 장치들을 조합하여 게임 내부에서 직접 상호작용하는 객체인 Entity들을 구현한다. 마지막으로 그러한 과정들이 하나의 클라이언트 내에서만 이루어지는 것이 아니라 여러 클라이언트들에게 동시에 보여지도록 하기 위해 Game Server를 구현한다.

|  |
| --- |
| **Geometry Library** |
| 게임 내부에 필요한 수많은 수학적 처리를 위해 Geometry Library를 개발한다. Geometry Library에서는 먼저 다양한 2d 도형에 대한 함수들을 정의한다. 나열식으로 정의된 [함수들뿐만](http://www.incruit.com/tools/spell/) 아니라, Geometry Library에서는 2d 도형들에 대해 정의하고 각 도형에 대한 [교차 여부](http://www.incruit.com/tools/spell/), [포함 여부](http://www.incruit.com/tools/spell/), 작용 힘, 교차점 등을 구하는 기능을 추상화하여, 다양한 게임 내 [entity들에](http://www.incruit.com/tools/spell/) 충돌을 위한 도형 선택을 자유롭게 한다.  그림에서 보이듯이 외적, 선분 간 교차, 선분 원 교차, 다각형 간 교차 등 다양한 수학 함수들이 구현된다.  **Implementation**  구현은 다음과 같은 형식으로 이루어진다. (문서에는 일부 함수만 나타나 있다.)  …  inline std::vector<cocos2d::Vec2> lineCircleIntersectPoints(  cocos2d::Vec2 A,  cocos2d::Vec2 B,  cocos2d::Vec2 P,  float R);  inline bool pointInRectangle(  cocos2d::Vec2 P,  cocos2d::Vec2 A,  cocos2d::Vec2 B);  inline bool rectanglesIntersect(  cocos2d::Vec2 bottom\_left1,  cocos2d::Vec2 top\_right1,  cocos2d::Vec2 bottom\_left2,  cocos2d::Vec2 top\_right2);  inline std::vector<cocos2d::Vec2> rectanglesIntersectPoints(  cocos2d::Vec2 bottom\_left1,  cocos2d::Vec2 top\_right1,  cocos2d::Vec2 bottom\_left2,  cocos2d::Vec2 top\_right2);  inline bool rectanglesOverlap(  cocos2d::Vec2 bottom\_left1,  cocos2d::Vec2 top\_right1,  cocos2d::Vec2 bottom\_left2,  cocos2d::Vec2 top\_right2);  inline bool rectangleInRectangle(  cocos2d::Vec2 bottom\_left1,  cocos2d::Vec2 top\_right1,  cocos2d::Vec2 bottom\_left2,  cocos2d::Vec2 top\_right2);  inline bool circleInRectangle(  cocos2d::Vec2 bottom\_left,  cocos2d::Vec2 top\_right,  cocos2d::Vec2 pos,  float radius);  inline bool rectangleCircleOverlap(  cocos2d::Vec2 bottom\_left,  cocos2d::Vec2 top\_right,  cocos2d::Vec2 pos,  float radius);  inline bool rectangleSegmentIntersect(  cocos2d::Vec2 bottom\_left,  cocos2d::Vec2 top\_right,  cocos2d::Vec2 A,  cocos2d::Vec2 B);  …  이러한 형태로 다양한 함수들이 구현된다. |
| **Utility Library** |
| 게임 개발을 위해서 필요한 수많은 기타 함수들의 집합이다. 방향벡터를 게임에서 정의하는 8방향으로 변환시켜주는 함수를 예로 들 수 있다.    첫 번째 그림처럼 단순한 parsing을 위한 함수들도 utility library에 포함될 수 있다. 또한 마지막 그림처럼, 의사 난수를 생성해주는 함수도 포함된다. |
| **Collision System** |
| 충동처리 System이다. 게임 내 다양한 entity들의 충돌 도형들이 서로 교차하는지, 교차점이 무엇인지, 어느 정도로 밀어내는지에 대해 정의된 template 함수들의 집합이다. 다양한 수학적 연산이 필요하기 때문에, Geometry Library를 적극적으로 이용한다. 또한 충돌처리에 대해 필요한 Shape들과 그에 대한 처리도 Geometry namespace 내에 포함되어 있다.  https://cdn.tutsplus.com/gamedev/uploads/legacy/060_gamedev_tutorial_roundup/xna_collision_detection.png  대부분의 간단한 충돌처리 방식으로 위와 같이 도형에 대한 AABB box를 만들고 AABB box간의 충돌로 충돌판단을 한다. 하지만, 좀 더 정교한 충돌을 위해서는 직사각형 말고 다른 도형들이 필요하다. 그래서 도형을 직선, 원, 직사각형, 다각형으로 확장하고, 4개의 도형에 대해, 다시 4개의 도형과 16가지 조합에 대해 충돌 점, 충돌여부 등을 추상화하여 사용한다.    위 그림에서 보이듯이 16가지 경우에 대한 충돌방법들이 정의된다.  **Implementation**  실제로 이를 구현해보자. 먼저 충돌처리 시스템에서 구현되어야 할 목록이다.   1. 4가지 도형을 정의하고, 모두 Shape이라는 class를 상속받도록 한다. 2. 16가지 경우에 대한 교차여부 판단 3. 16가지 경우에 대한 교차점 구하기 4. 16가지 경우에 대한 교차 or 포함 판단 5. 16가지 경우에 대한 겹침 거리 밀어내기   단순하게 교차여부를 판단하는 것과 교차점을 구하는 것은 기하 알고리즘에서 성능의 차이가 있다. 그러므로 분리해서 설계한다. 또한 교차에 추가로 포함되는 경우까지 판단해야 하는 상황이 올 수 있다. 이런 부분을 또 분리한다. 마지막으로 겹침 거리를 구해야 하는데, 그 이유를 구체적인 예제를 통해 알아보자.    먼저 두 물체가 겹치는 상황을 생각해보자. 게임 내부에서 이런 겹치는 상황이 생긴다면, 캐릭터가 벽을 뚫고 지나가는 상황이 올 수 있을 것이다. 그러므로 두 물체가 겹치지 않도록 해야 하고 그러기 위해서 서로 겹치는 거리를 지속적으로 밀어내야 한다.  지속적으로 겹치는 거리를 밀어낸다면, 두 물체는 겹치지 않게 될 것이다. 위의 예제는 원과 원에 대해서 설명했지만, 이러한 구현은 16가지 조합에 대해 모두 구현이 되어야 한다. 그리고 이런 실제적인 알고리즘은 모두 Geometry Library에 이미 되어있다. 그러므로 적절한 도형에 대해 적절한 함수를 사용하기만 하면 된다.  // 교차여부 판단  inline bool intersect(const Rect& rect1, const Rect& rect2);  inline bool intersect(const Rect& rect, const Circle& circle);  inline bool intersect(const Rect& rect, const Segment& segment);  inline bool intersect(const Rect& rect, const Polygon& polygon);  inline bool intersect(const Circle& circle, const Rect& rect);  inline bool intersect(const Circle& circle1, const Circle& circle2);  inline bool intersect(const Circle& circle, const Segment& segment);  inline bool intersect(const Circle& circle, const Polygon& polygon);  inline bool intersect(const Segment& segment, const Rect& rect);  inline bool intersect(const Segment& segment, const Circle& circle);  inline bool intersect(const Segment& segment1, const Segment& segment2);  inline bool intersect(const Segment& segment, const Polygon& polygon);  inline bool intersect(const Polygon& polygon, const Rect& rect);  inline bool intersect(const Polygon& polygon, const Circle& circle);  inline bool intersect(const Polygon& polygon, const Segment& segment);  inline bool intersect(const Polygon& polygon1, const Polygon& polygon2);  // 교차점 구하기  inline std::vector<cocos2d::Vec2> intersectPoints(  const Rect& rect1,  const Rect& rect2);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Rect& rect,  const Circle& circle);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Rect& rect,  const Segment& segment);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Rect& rect,  const Polygon& polygon);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Circle& circle,  const Rect& rect);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Circle& circle1,  const Circle& circle2);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Circle& circle,  const Segment& segment);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Circle& circle,  const Polygon& polygon);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Segment& segment,  const Rect& rect);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Segment& segment,  const Circle& circle);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Segment& segment1,  const Segment& segment2);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Segment& segment,  const Polygon& polygon);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Polygon& polygon,  const Rect& rect);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Polygon& polygon,  const Circle& circle);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Polygon& polygon,  const Segment& segment);  inline std::vector<cocos2d::Vec2> intersectPoints(  const Polygon& polygon1,  const Polygon& polygon2);  // 교차 or 포함 판단  inline bool overlap(const Rect& rect1, const Rect& rect2);  inline bool overlap(const Rect& rect, const Circle& circle);  inline bool overlap(const Rect& rect, const Segment& segment);  inline bool overlap(const Rect& rect, const Polygon& polygon);  inline bool overlap(const Circle& circle, const Rect& rect);  inline bool overlap(const Circle& circle1, const Circle& circle2);  inline bool overlap(const Circle& circle, const Segment& segment);  inline bool overlap(const Circle& circle, const Polygon& polygon);  inline bool overlap(const Segment& segment, const Rect& rect);  inline bool overlap(const Segment& segment, const Circle& circle);  inline bool overlap(const Segment& segment1, const Segment& segment2);  inline bool overlap(const Segment& segment, const Polygon& polygon);  inline bool overlap(const Polygon& polygon, const Rect& rect);  inline bool overlap(const Polygon& polygon, const Circle& circle);  inline bool overlap(const Polygon& polygon, const Segment& segment);  inline bool overlap(const Polygon& polygon1, const Polygon& polygon2);  // 겹치는 거리 밀어내기  inline cocos2d::Vec2 enforceFrom(const Rect& r1, const Rect& r2);  inline cocos2d::Vec2 enforceFrom(const Rect& r, const Circle& c);  inline cocos2d::Vec2 enforceFrom(const Rect& r, const Segment& s);  inline cocos2d::Vec2 enforceFrom(const Rect& r, const Polygon& p);  inline cocos2d::Vec2 enforceFrom(const Circle& c, const Rect& r);  inline cocos2d::Vec2 enforceFrom(const Circle& c1, const Circle& c2);  inline cocos2d::Vec2 enforceFrom(const Circle& c, const Segment& s);  inline cocos2d::Vec2 enforceFrom(const Circle& c, const Polygon& p);  inline cocos2d::Vec2 enforceFrom(const Segment& s, const Rect& r);  inline cocos2d::Vec2 enforceFrom(const Segment& s, const Circle& c);  inline cocos2d::Vec2 enforceFrom(const Segment& s, const Segment& s);  inline cocos2d::Vec2 enforceFrom(const Segment& s, const Polygon& p);  inline cocos2d::Vec2 enforceFrom(const Polygon& p, const Rect& r);  inline cocos2d::Vec2 enforceFrom(const Polygon& p, const Circle& c);  inline cocos2d::Vec2 enforceFrom(const Polygon& p, const Segment& s);  inline cocos2d::Vec2 enforceFrom(const Polygon& p1, const Polygon& p2);  하지만 어떤 두 타입에 대해 추상화를 하는 것은 쉽지 않은 문제이다. 일단 어떤 상황인지 먼저 알아보자.  어떤 구조물과 캐릭터가 충돌 영역을 정의하는 Shape\*을 하나씩 갖고 있다고 가정하자. 구조물은 충돌영역으로 Polygon을 갖고, 캐릭터는 충돌영역으로 Circle을 갖는다.  Shape\* bounding\_shape1 = new Polygon();  Shape\* bounding\_shape2 = new Circle();  하지만 기본적으로 두 자료형은 Shape이기 때문에 위에 정의해둔  intersect(const Circle& circle, const Polygon& polygon) 와 같은 함수는 사용할 수가 없다. 일단 다른 가상함수와 같이 각 Shape에 대해 다른 Shape들과의 intersect함수를 구현해 보자.  namespace geometry  {  class Shape;  class Circle;  class Polygon;  class Segment;  class Rect;  class Shape  {  public:  virtual bool intersect(const Shape\* const other) const = 0;  virtual bool intersect(const Rect\* const other) const = 0;  virtual bool intersect(const Segment\* const other) const = 0;  virtual bool intersect(const Circle\* const other) const = 0;  virtual bool intersect(const Polygon\* const other) const = 0;  };  }  이렇게 하면 이제 bounding\_shape1.intersect(bounding\_shape2)의 형태로 두 추상 Shape에 대한 intersect를 구해볼 수 있을 것이다. 가능할까??  가상함수의 도움으로 bounding\_shape1의 실제 인스턴스인 Polygon의 intersect함수를 호출하는데 까지는 성공하였다. 하지만, 여전히 bounding\_shape2는 Shape\*으로 남아있다. 이 문제를 해결하기 위해 dynamic\_cast를 이용할 수 있을 것이다.  {  if (Circle\* cir = dynamic\_cast<Circle\*>(bounding\_shape))  {  //..  }  else if (Polygon\* poly = dynamic\_cast<Polygon\*>(bounding\_shape))  {  //..  }  else if (Rect\* rect = dynamic\_cast<Rect\*>(bounding\_shape))  {  //..  }  else if (Segment\* seg = dynamic\_cast<Segment\*>(bounding\_shape))  {  //..  }  else  {  throw;  }  }  하지만 이 방법은 두 가지 문제가 있다. 일단 설계적인 관점에서 if else구조의 구현은 확장성, 의존성 문제에 취약하다. 그리고 현재 구현하고 있는 시스템은 충돌처리 시스템으로서, 게임의 성능에 가장 큰 영향을 미치는 부분이다. 여기에 큰 비용을 사용하는 dynamic\_cast를 중첩시키는 것은 결코 좋지 못한 선택이다. 그렇다면 어떻게 해야 할까?  (참고로 이 문제는 Double Dispatch라는 굉장히 유명한 문제 중 하나이다. <https://en.wikipedia.org/wiki/Double_dispatch> )  이 문제를 해결하는 여러 가지 방법들이 있다(모두 완벽하지는 않다.). 여기서는 virtual function만 이용하여 이 문제를 해결한다. Virtual function만 이용한 방법은 확장성이 결여되어 있다는 점에서 문제가 있지만, 성능이나, 안정성 면에서는 사용할 만 하다.  Shape에 대해 virtual bool intersect(Shape\* other) 함수를 만들어 준다면, bounding\_shape1.intersect(bounding\_shape2)에서 bounding\_shape1의 실제 인스턴스인 Polygon의 intersect(Shape\* other) 함수까지는 호출이 될 것이다.  namespace geometry  {  class Shape;  class Circle;  class Polygon;  class Segment;  class Rect;  class Shape  {  public:  virtual bool intersect(const Shape\* const other) const = 0;  virtual bool intersect(const Rect\* const other) const = 0;  virtual bool intersect(const Segment\* const other) const = 0;  virtual bool intersect(const Circle\* const other) const = 0;  virtual bool intersect(const Polygon\* const other) const = 0;  };  class Circle  {  public:  virtual bool intersect(const Shape\* const o) const override;  virtual bool intersect(const Rect\* const o) const override;  virtual bool intersect(const Segment\* const o) const override;  virtual bool intersect(const Circle\* const o) const override;  virtual bool intersect(const Polygon\* const o) const override;  };  class Polygon  {  public:  virtual bool intersect(const Shape\* const o) const override;  virtual bool intersect(const Rect\* const o) const override;  virtual bool intersect(const Segment\* const o) const override;  virtual bool intersect(const Circle\* const o) const override;  virtual bool intersect(const Polygon\* const o) const override;  };  }  여기서 Polygon의 intersect에서 o의 타입은 여전히 Shape\*으로 그 정체가 밝혀지지 않았지만, Polygon 자신은 타입이 결정되었다. 그러므로 뒤집어서 다시 호출을 한다.  bool Polygon::intersect(const Shape\* const other) const  {  return other->intersect(this);  }  다시 한번 other의 가상함수가 호출되면서 other의 타입도 결정이 된다. 실제 인스턴스가 Circle이었으므로 Circle class의 intersect(Polygon\*)이 호출되게 된다. 이 시점에서 이제 Circle과 Polygon 모두 타입이 결정되었기 때문에, overloading된 함수로 forwarding이 가능해진다.  bool Circle::intersect(const Polygon\* const other) const  {  return geometry::intersect(\*this, \*other);  }  그러면 위에서 정의하였던 16가지 함수들 중 하나의 intersect가 호출되면서 Circle과 Polygon의 충돌판단이 가능해진다. 결과적으로 Shape\*과 Shape\*에 어떠한 실제 인스턴스가 담겨져 있더라도 이중 virtual function을 통해 적절한 intersect 함수가 결정되어진다. 마찬가지의 방법으로, Segment class, Rect class들을 구현하고 intersectPoint, overlap method들을 모두 구현한다. 하지만 여기서 한번 더 문제가 생긴다. enforceFrom함수는 동작방식이 다르기 때문이다. 이전의 method들은 어떤 두 인자의 순서와 상관없이 같은 동작을 하기 때문에 이중 virtual을 통해 순서가 뒤집어지더라도 원하는 결과가 나왔던 것이지만, enforceFrom은 원하는 결과와 반대의 결과가 나오게 된다. 이를 방지하기 위해 enforceFrom을 위한 proxy method를 만들어 중간 중재역할을 맡기도록 한다.  Ex) Polygon이 Circle을 밀어낸다고 가정하자.  다른 method들과 달리 proxy method를 먼저 호출한다.  cocos2d::Vec2 Polygon::enforceFrom(const Shape\* const other) const  {  return other->enforceFromProxy(this);  }  Proxy method에서는 뒤집어질 연산을 바로잡기 위해 한 번 미리 뒤집는다.  cocos2d::Vec2 Circle::enforceFromProxy(const Polygon\* const other) const  {  return other->enforceFrom(this);  }  모든 타입이 결정되었고, 방향성도 보존되었으므로 overloading된 enforceFrom 함수를 호출한다.  cocos2d::Vec2 Polygon::enforceFrom(const Circle\* const other) const  {  return geometry::enforceFrom(\*this, \*other);  }  이렇게 임의의 두 도형에 대한 추상화가 구현되었다. 다음으로 구현되어야 할 것은, 임의의 container의 들어있는 entity들과 임의의 entity간의 충돌처리이다. 버전에 따라 다양한 종류들이 있다.  // 임의의 container와 임의의 entity에 대한 충돌처리  template <class Entity, class Container>  inline bool collisionToOthers(  const Entity& entity,  const Container& others)  {  for (auto it = std::begin(others); it != std::end(others); ++it)  {  if (!(\*it)->isAlive() || it->getTeam() == entity->getTeam())  continue;  if ((\*it)->getShape()->overlap(entity->getShape().get()))  {  return true;  }  }  return false;  }  // 임의의 분할공간과 임의의 entity에 대한 충돌처리  template <typename Entity, typename CellSpace>  inline bool collisionToOthersCellSpace(  const Entity& entity,  const CellSpace& others)  {  others->calculateNeighborsForSmall(  entity->getPos(), entity->getShape().get());  for (auto it = others->begin(); !others->end(); it = others->next())  {  if (!it->isAlive() || it->getTeam() == entity->getTeam())  continue;  if (it->getShape()->overlap(entity->getShape().get()))  {  return true;  }  }  return false;  }  // 임의의 entity와 충돌하는 임의의 분할공간 내 entity들 list  template <typename Entity, typename CellSpace>  inline std::list<typename CellSpace::entity\_type>  getListOfCollisionEntitiesCellSpace(  const Entity& entity,  CellSpace\* others)  {  std::list<typename CellSpace::entity\_type> hits;  others->calculateNeighborsForSmall(  entity->getPos(), entity->getShape().get());  for (auto it = others->begin(); !others->end(); it = others->next())  {  if (it->getId() == entity->getId()  || !it->isAlive() || it->getTeam() == entity->getTeam())  continue;  if (it->getShape()->overlap(entity->getShape().get()))  {  hits.push\_back(it);  }  }  return hits;  }  **Class Diagram**    **출력화면**    왼쪽 그림은 원(Circle)을 기본 도형으로 사용하는 Character와 다각형(Polygon)을 기본 도형으로 사용하는 CollisionArea class와 사이의 충돌을 보여준다. (CollisionArea는 충돌 영역에 대한 객체이다.) 오른쪽 그림은 원을 기본 도형으로 사용하는 Skill과 다각형을 기본 도형으로 사용하는 CollisionArea class 객체 사이의 충돌을 보여준다. |
| **Goal System** |
| 게임 내 AI를 구현하기 위한 복합 목적 시스템이다. 게임 내 Bot들은 어떤 목적을 갖고 자체적으로 행동하며, 그러한 목적들을 다시 어떤 부 목적들의 집합으로 정의될 수 있다. 그런 시스템을 구축하기 위해서는 Goal System이 완벽하게 구현되어야 하며, 이는 Design Pattern 중 Composite Pattern을 이용하여 효율적으로 구현할 수 있다.  Ex)    그림을 통해 예를 들어보면, “검을 산다” 라는 목적은 컴퓨터 내에서 처리되기 위해서는 부분 목적들로 정의할 수 있다. 돈이 있다면 바로 상인에게 가서 검을 사올 수 있겠지만, 돈이 없다면 먼저 돈을 구해야 할 것이다. 그러므로 “검을 산다” 라는 목적은 “[돈을 구한다.](http://www.incruit.com/tools/spell/)” + “상인에게서 검을 산다” 라는 두 가지 [부목 적의](http://www.incruit.com/tools/spell/) 복합체라고 표현할 수 있다. 마찬가지로 “[돈을 구한다.](http://www.incruit.com/tools/spell/)” 라는 목적도 “금덩어리로 갈 수 있는 [경로를 찾는다.” + “경로를](http://www.incruit.com/tools/spell/) [따라 걷는다.](http://www.incruit.com/tools/spell/)” + “금덩어리를 줍는다” 라는 [부목 적의](http://www.incruit.com/tools/spell/) 복합체라고 할 수 있다. 그리고 스스로 목적을 생성해내기 위해서 최상위 목적으로 Think라는 목적을 둔다. Think는 자신의 상태와 기억에 있는 데이터들을 종합하여 가장 적합한 의사를 결정한다. 그 후 가장 적합한 복합목적을 생성해내고 위 설명한 내용대로 처리한다. 이러한 형태로 [수많은 목적들을](http://www.incruit.com/tools/spell/) 표현할 수 있다. 그리고 게임 내부에는 다양한 목적들이 표현되어 있다.  **Implementation**  Goal System을 실제로 구현해보자. (쉬운 설명을 위해 실제 코드보다는 많이 생략되었다.)  먼저 Goal System에 필요한 요소들을 생각해보자. 기본적으로 목적 중에서는 다시 부목적의 집합을 가질 수 있는 복합목적과 더 이상 부목적을 갖지 않는 단일목적이 있을 것이다. 이러한 object tree 형태를 실제로 구현하기 위해 Composite Pattern을 이용한다. 더 이상 부목적을 갖지 않는 단일목적을 Goal class로 정의하고, 다시 부 목적들을 갖는 복합목적을 GoalComposite class로 정의한다. 여기서 중요한 점은 복합목적 또한 일종의 목적으로서 단일목적과 같은 형태로 동작한다는 것이다. 그러므로 단일목적을 상속받는다. 이런 형태로 기본적인 뼈대를 잡은 후, 각 class에 필요한 data와 method를 정의한다.   1. 먼저 모든 목적은 목적의 주체를 갖고 생성되어야 한다. 즉 모든 목적은 weak entity가 된다. 2. 그리고 목적은 자신의 상태를 나타낼 수 있어야 한다. 3. 모든 목적은 계획, 실행, 종료, 메시지 처리에 대한 방법을 갖는다.   목적의 주체는 어떤 타입이라도 가능하도록 template으로 정의한다.  class Telegram;  template <class Entity>  class Goal  {  public:  // 상태 열거자  enum { INACTIVE, ACTIVE, COMPLETED, FAILED };  Goal(Entity\* const owner)  :  \_status(INACTIVE),  \_owner(owner)  {}  // Goal의 계획  virtual void activate() = 0;  // Goal의 진행  // Goal이 성공적이었는지 반환  virtual int process() = 0;  // Goal의 종료 전 처리  virtual void terminate() = 0;  // Goal 중에는 Message 처리를 담당하는 종류도 있다.  virtual bool handleMessage(const Telegram& msg) { return false; }  private:  // 모든 목적은 상태를 갖는다.  int \_status;  // 모든 목적은 목적의 주체를 갖고 생성되어야 한다.  Entity\* const \_owner;  };   1. 복합목적 또한 목적이다. 2. 복합목적은 부목적 리스트를 갖는다. 3. 복합목적은 부목적에 대한 추가적인 처리 기능을 갖는다. 4. 복합목적은 부목적을 추가할 수 있다. 5. 복합목적은 모든 부목적을 제거할 수 있다. 6. 복합목적은 message 처리에 대한 책임을 부목적에게 전가할 수 있다.   #include <memory>  #include <list>  #include <algorithm>  using namespace std;  template <class Entity>  class GoalComposite : public Goal<Entity>  {  public:  GoalComposite(Entity\* owner)  :  Goal<Entity>(owner)  {}  // GoalComposite의 계획  virtual void activate() = 0;  // GoalComposite의 진행  // processSubgoals를 호출한다.  // GoalComposite이 성공적이었는지 반환  virtual int process() = 0;  // GoalComposite의 종료 전 처리  virtual void terminate() = 0;  // GoalComposite은 Message에 대한 처리 후 부목적으로 책임을 전가할 수 있다.  virtual bool handleMessage(const Telegram& msg);  // 새로운 Goal을 추가  void addSubGoal(Goal<Entity>\* goal);  // 모든 부목적 제거  void removeAllSubGoal();  private:  list<unique\_ptr<Goal<Entity> > > \_sub\_goals;  // 완료되었거나, 실패한 부목적을 리스트에서 제거하고,  // 가장 front에 있는 부목적을 실행한다.  int processSubgoals();  // 가장 front에 있는 부목적으로 Message 처리에 대한 책임을 전가한다.  bool forwardMessageToFrontMostSubgoal(const Telegram& msg);  };  이런 형태로 먼저 Goal System의 근간이 되는 Goal과 GoalComposite을 구현한다. 이후에는 다양한 단일목적과, 복합목적들을 위 class들을 이용하여 파생시키면 된다. 각 목적마다 동작방식이 다 다르고, 하위로 갖는 부목적 또한 다를 것이다. 하지만 위에 정의한 함수들을 이용하면, 필요한 모든 상황을 표현할 수 있다.  이제 가장 중요한 GoalThink를 구현해야 한다. GoalThink는 복합목적이면서 다른 목적들과 다른 특별한 성질을 갖는다. 목적의 주체로부터 주체가 소멸하기 전까지는 절대 소멸하지 않으며, 현재 상황에서 가장 적합하다고 판단되는 새로운 목적이나 복합목적을 주기적으로 생성해낸다. 그러므로 GoalThink는 다음 조건을 충족하여야 한다.   1. Think를 정의하는 적절한 평가자들을 가지고 있어야 한다. 2. 평가자들을 통해 의사결정을 하고, 그에 맞는 새로운 목적을 생성할 수 있어야 한다.   class AbstCharacter;  class GoalThink : public GoalComposite<AbstCharacter>  {  public:  explicit GoalThink(AbstCharacter\* const bot);  virtual ~GoalThink() override;  // 각 평가자들을 순회하며 가장 최고의 Desirability를 평가자를 선택하여  // setGoal을 실행시킨다.  // setGoal을 통해, 새로운 목적이 생성된다.  virtual void arbitrate();  // GoalThink도 다른 목적들과 마찬가지로 계획, 실행, 종료에 대한 처리를 갖는다.  virtual int process() override;  virtual void activate() override;  virtual void terminate() override  {}  protected:  // Think의 의사결정을 정의하는 평가자들  std::vector<std::unique\_ptr<GoalEvaluator> > \_evaluators;  };  마지막으로 평가자를 정의하여야 한다. 평가자는 다음 조건을 충족하여야 한다.   1. 각 평가자는 현재의 상태 값과, 정보를 바탕으로 하나의 목적에 대해 평가한다. 2. 의사결정은 개체마다 조금씩 다를 수 있으므로, 개성 값을 갖도록 한다. 3. 각 평가자는 평가 기준 목적을 생성할 수 있어야 한다.   class AbstCharacter;  class GoalEvaluator  {  public:  explicit GoalEvaluator(double character\_bias)  :  \_character\_bias(character\_bias)  {}  virtual ~GoalEvaluator()  {}  // 평가자는 목적 시스템의 주체의 데이터와 상태를 보고 적절한 의사결정을 수행한다.  // 생성하는 목적에 대한 0 ~ 1 사이의 Desirability 값을 반환한다.  virtual double calculateDesirability(AbstCharacter\* const bot) = 0;  // 평가자에 대한 목적을 생성한다.  virtual void setGoal(AbstCharacter\* const bot) = 0;  protected:  // 각 객체마다 의사결정 과정에 반영될 개성 값  double \_character\_bias;  };  **Class Diagram**    **출력 화면**    왼쪽 그림의 경우, 최상위 목적인 Think를 통해 Explore(탐험)라는 복합목적이 생성되었다. Explore는 임의의 위치로 움직이는 목적으로서 다시 [MoveToPosition 이라는](http://www.incruit.com/tools/spell/) [부목적을](http://www.incruit.com/tools/spell/) 만들고 그것은 다시 [SeekToPosition 이라는](http://www.incruit.com/tools/spell/) [부목적을](http://www.incruit.com/tools/spell/) 만들고 결국 원하는 목적을 달성해낸다. 오른쪽 그림의 경우, 주변에 적을 감지하였기 때문에 Think는 [Attack Target이라는](http://www.incruit.com/tools/spell/) 복합목적을 생성해낸다. 그리고 이 복합목적은 MeleeAttack이라는 [부목적을](http://www.incruit.com/tools/spell/) 생성해내고 agent로 하여금 공격에 대한 동작을 수행하도록 한다. |
| **Navigation System** |
| 게임 내 AI와 Player는 모두 Character를 통해 게임 속에서 이동한다. 하지만 게임 내에는 다양한 장애물들이 있기 때문에, 특별한 탐색 알고리즘과 항해 그래프가 정의되어 있지 않다면 원하는 목적지까지 이동할 방법이 없다. Navigation System에서는 Navigation Graph(Navigation Node, Navigation Edge, Sparse Graph)와 다양한 탐색에 필요한 알고리즘(DFS, BFS, A\*, Dijkstra, Minimum Spanning Tree, Time Sliced A\*, Time Sliced Dijkstra)을 구현하여 게임 내 Character들의 이동에 도움을 준다. 또한, 이러한 그래프와 알고리즘을 손쉽게 사용하기 위한 Path Planner를 구현한다. 이는 Design Pattern 중 Strategy Pattern을 이용하여 효율적으로 구현할 수 있다.  **Implementation**  실제로 이를 구현해보자. (source code는 실제 구현 내용에서 많이 생략한 형태이다.)  길 찾기 알고리즘을 정의하기 전에 먼저 그래프를 정의하여야 한다. 즉 게임 내부에 어떤 형태의 그래프를 사용할 것 인지가 결정되어야 한다. 그래프를 정의하기 위해서는 그 그래프가 어떤 Node 타입과 어떤 Edge 타입을 사용할 것인지를 결정해야 한다.  Node Type  기본적으로 그래프에서 사용하는 Node는 단순히 index만 갖는 형태로 구현된다. 하지만 여기서는 추가적으로 정점의 위치를 나타내는 Vec2를 갖도록 한다. 또한 각 정점에 대하여 어떤 이벤트를 발생시킬 수 있도록 하기 위해서 Trigger를 담을 수 있는 형태로 구현할 것이다. Trigger 자체가 어떤 타입으로 들어올 지 알 수 없기 때문에 임의의 타입을 수용할 수 있는 template으로 extra\_info를 갖도록 한다.  // 일반적인 그래프의 Node 타입  class GraphNode  {  public:  // default 생성의 경우 \_index를 -1로 초기화한다.  GraphNode();  explicit GraphNode(int idx);  virtual ~GraphNode();  // getters && setters  int getIndex() const { return \_index; }  void setIndex(int new\_index) { \_index = new\_index; }  protected:  // 정점의 index  int \_index;  };  // 2D 그래프의 정점, 위치의 대한 정보를 추가적으로 갖는다.  // ExtraInfo로 어떤 타입이라도 받을 수 있도록 한다.  // 이후에 Trigger 타입을 받아서 사용될 것이다.  template <class ExtraInfo = void\*>  class NavGraphNode : public GraphNode  {  protected:  // 정점의 위치  cocos2d::Vec2 \_position;  // 정점이 갖는 추가 정보  // Trigger 타입이 ExtraInfo로 들어오면,  // 해당 Trigger에 대한 이벤트를 처리할 수 있을 것이다.  ExtraInfo \_extra\_info;  public:  NavGraphNode();  NavGraphNode(int idx, cocos2d::Vec2 pos);  virtual ~NavGraphNode() override;  // getters && setters  cocos2d::Vec2 getPos() const { return \_position; }  void setPos(cocos2d::Vec2 new\_position) { \_position = new\_position; }  ExtraInfo getExtraInfo() const { return \_extra\_info; }  void setExtraInfo(ExtraInfo info) { \_extra\_info = info; }  };  정점이 정의되었으면, 그 다음으로 Edge를 정의한다.  Edge Type  Edge는 기본적으로 두 정점의 index인 from과 to와 그 간선에 대한 가중치인 cost를 갖는다. 여기에서는 추가로 해당 Edge가 어떤 속성을 갖는지에 대한 flag변수를 넣을 것이다.  // 기본 그래프의 간선  class GraphEdge  {  public:  // ctors && dtors  GraphEdge(int from, int to, double cost = 1.0);  GraphEdge();  virtual ~GraphEdge();  // getters && setters  int getFrom() const { return \_from; }  void setFrom(int new\_index) { \_from = new\_index; }  int getTo() const { return \_to; }  void setTo(int new\_index) { \_to = new\_index; }  double getCost() const { return \_cost; }  void setCost(double new\_index) { \_cost = new\_index; }  protected:  // 간선은 두 정점의 index를 갖는다.  int \_from;  int \_to;  // 간선의 가중치  double \_cost;  };  class NavGraphEdge : public GraphEdge  {  public:  // 간선의 속성 종류  enum  {  NORMAL = 0,  SWIM = 1 << 0,  CRAWL = 1 << 1,  CREEP = 1 << 3,  JUMP = 1 << 3,  FLY = 1 << 4,  GRAPPLE = 1 << 5,  GOES\_THROUGH\_DOOR = 1 << 6  };  protected:  // 간선의 속성  int \_flag;  public:  NavGraphEdge(  int from,  int to,  double cost,  int flags = 0,  int id = -1);  virtual ~NavGraphEdge() override;  int getFlags() const;  void setFlags(int flags);  };  정점과 간선이 정의되었으면, 이제 그래프를 정의해야 한다.  여기서 그래프를 만들 때, 어떤 정점과 어떤 간선을 사용할 지는 사용자가 결정할 수 있도록 해야 한다. 그러므로 template 인자로 빼도록 한다.  추가로 그래프에서는 정점의 빠른 참조를 위해 vector 자료구조에 정점을 저장하는데, 여기서 문제는, 정점의 추가와 삭제가 일어날 경우, 정점의 index가 꼬여버린다는 점이다. 그러므로 정점이 삭제되더라도 vector에서 실제로 erase할 것이 아니라, INVALID\_NODE\_INDEX로 설정하고 순회, 참조할 때 뛰어넘도록 한다. 그러므로 그래프는 그래프의 정점과 간선을 순회할 수 있는 순회자(iterator)와 유효한 정점을 참조할 수 있는 참조자(getter)를 제공할 필요가 있다.  template <class Node, class Edge>  class SparseGraph  {  public:  // 내부에서 Edge, Node를 typedef 함으로써,  // 외부에서 Edge의 타입을 가져올 수 있게 된다.  typedef Edge Edge;  typedef Node Node;  private:  // Node는 random access(빠른 참조)가 가능하도록 vector로 정의한다.  std::vector<Node> \_nodes;  // edge의 from to중에서 from을 이용하여,  // random access가 가능하도록 vector를 먼저 만들고,  // 각 vector에 from에 대한 to까지 포함하는 Edge class의 list를 갖도록 한다.  std::vector<std::list<Edge> > \_edges;  // 다음으로 유효한 node의 index를 저장한다.  int \_next\_node\_index;  // 유효하지 않는 node를 포함하는 edge들을 제거한다.  void cullInvalidEdges();  public:  // ctor  explicit SparseGraph(bool digraph);  // dtor  ~SparseGraph();  // Node를 추가한다.  int addNode(Node node);  // Node를 제거한다. (유효하지 않다고 표시한다.)  void removeNode(int node);  // Edge를 추가한다.  void addEdge(Edge edge);  // Edge를 제거한다.  void removeEdge(int from, int to);  // Edge의 가중치를 설정한다.  void setEdgeCost(int from, int to, double cost);  // getters && setters  const Node& getNode(int idx)const;  Node& getNode(int idx);  const Edge& getEdge(int from, int to)const;  Edge& getEdge(int from, int to);  int getNextFreeNodeIndex()const;  int numNodes() const;  int numActiveNodes() const;  int numEdges() const;    // Edge와 Node의 순회, 위치를 나타내기 위한 iterator를 제공한다.  // (const 버전, non-const 버전)  class EdgeIterator  {  //...  };  class NodeIterator  {  //...  };  };  이제 탐색 알고리즘들을 정의한다.  탐색 알고리즘들로는 DFS, BFS, Dijkstra, A\* 알고리즘 등이 있지만, 여기서는 가장 유용한 A\* 알고리즘에 대해서만 정의하도록 한다.  template <class Graph, class HeuristicPolicy>  class SearchAStar  {  private:  // template 인자로 받은 Graph의 Edge 타입을 Edge로 typedef 함으로써,  // 외부에서 이 알고리즘의 Edge 타입을 사용할 수 있도록 한다.  typedef typename Graph::Edge Edge;  private:  // template 인자로 얻은 Graph를 참조하도록 한다.  // 그러므로 이 class는 반드시 Graph를 참조하며 생성되는 weak entity가 된다.  const Graph& \_graph;  // 각 정점으로의 실제 누적 비용  std::vector<double> \_g\_costs;  // 각 정점으로의 휴리스틱 정책으로 얻은 비용  std::vector<double> \_f\_costs;  // 최소 경로 트리, 이후 이 트리를 이용하여 최단경로를 만들어 낼 수 있다.  std::vector<const Edge\*> \_shortest\_path\_tree;    // 최단경로를 만들기 위한 임시 저장공간  std::vector<const Edge\*> \_search\_frontier;  // 탐색 시작 위치과 끝 위치  int \_source;  int \_target;  // search 함수를 통해 A\* algorithm이 동작하고,  // \_shortest\_path\_tree에 최단경로를 얻을 수 있는 data가 만들어진다.  void search();  public:  SearchAStar(Graph &graph,  int source,  int target);  // 최단경로트리에 대한 getter  std::vector<const Edge\*> getSPT() const;  // 경로를 만들어서 반환한다.  // \_shortest\_path\_tree가 먼저 만들어져 있어야 한다.  std::list<int> getPathToTarget() const;  // 목표 위치로의 비용을 반환한다.  double getCostToTarget() const;  };  위와 같은 구조로 algorithm들의 대한 틀을 구현한다. 가장 중요한 것은 모든 search algorithm들은 공통적으로 getPathToTarget() 함수와 search() 함수를 제공해야 한다는 것이고, Graph를 template 인자로 받아야 한다는 것이다. 그렇게 하여 정적 다형성의 조건이 성립되면, 다양한 algorithm에 대해 선택적으로 원하는 path finder(경로 계획자)를 정의할 수 있도록 한다.  탐색 알고리즘들을 정의하였지만, 실제로 이 알고리즘들을 이용할 수는 없다. 왜냐하면, 어떤 한 entity가 꽤나 먼 위치까지의 경로를 탐색한다고 하면, 그 비용이 상당히 클 것이다. 사실은 그 모든 비용을 한 frame이 감당할 필요가 없는데도, 탐색 알고리즘이 완벽하게 끝날 때까지 기다려야만 한다. 그러므로 탐색 알고리즘을 이용할 때 그 알고리즘이 끝날 때까지 기다리는 것이 아니라, 시분할로 조금씩 탐색을 진행하고 탐색이 완료되면 event를 보내는 형태로 설계한다. Event가 왔다는 것은 경로가 완성되었다는 의미이고, 경로를 순회할 수 있다. 이런 형태로 설계하기 위해서는 먼저 탐색 알고리즘을 시분할 탐색으로 동작이 가능하도록 수정해야 한다. 먼저 시분할탐색으로 동작시킬 Dijkstra 알고리즘과 A\* 알고리즘을 포함하는 super class를 정의한다.  // 탐색 여부에 대한 열거값  enum { TARGET\_FOUND, TARGET\_NOT\_FOUND, SEARCH\_INCOMPLETE };  // 시분할 탐색 알고리즘을 위한 super class  template <class Edge>  class SearchTimeSliced  {  public:  enum SearchType { ASTAR, DIJKSTRA };  public:  explicit SearchTimeSliced(SearchType type);  virtual ~SearchTimeSliced();  // 호출될 때 이 method는 알고리즘을 한 탐색 cycle 만큼 동작시킨다.  // (TARGET\_FOUND, TARGET\_NOT\_FOUND, SEARCH\_INCOMPLETE)  // 중 맞는 열거값을 반환한다.  virtual int cycleOnce() = 0;  // 최단 경로 트리를 반환한다.  virtual std::vector<const Edge\*> getSPT() const = 0;  // 출발지부터 도착지까지의 최단 경로(node의 index 집합)를 반환한다.  virtual std::list<int> getPathToTarget() const = 0;  // 출발지부터 도착지까지의 최단 경로를 반환한다.  virtual std::list<PathEdge> getPathAsPathEdges() const = 0;  // type에 대한 getter  SearchType getType() const;  private:  SearchType \_search\_type;  };  Time Sliced Dijkstra Algorithm  // template 인자로 탐색 대상 그래프와, 종료 조건을 받는다.  // 또한 Dijkstra 알고리즘은 현재 시분할로 동작하도록 구현될 것이기 때문에,  // SearchTimeSliced class를 상속받도록 한다.  template <class Graph, class TerminationCondition>  class SearchDijkstrasTS  : public SearchTimeSliced<typename Graph::Edge>  {  private:  // 대상 그래프의 Node 타입과 Edge 타입을 제공한다.  typedef typename Graph::Edge Edge;  typedef typename Graph::Node Node;  public:  SearchDijkstrasTS(  const Graph& graph,  int source,  int target);  virtual ~SearchDijkstrasTS() override  {}  // 호출되면 이 method는 priority queue에서 한 node를 꺼내고,  // 그 node에 대한 모든 edge를 확인한다.  // 마찬가지로 결과를 (TARGET\_FOUND, TARGET\_NOT\_FOUND, SEARCH\_INCOMPLETE)  // 상태값으로 반환한다.  virtual int cycleOnce() override;  // 출발지로부터 도착지까지의 최단 경로를 반환한다. (node의 index)  virtual std::list<int> getPathToTarget() const override;  // 출발지로부터 도착지까지의 최단 경로를 반환한다.  virtual std::list<PathEdge> getPathAsPathEdges() const override;  // 최단 경로 트리를 반환한다.  virtual std::vector<const Edge\*> getSPT() const override;  private:  // 탐색 알고리즘은 반드시 graph를 갖고 생성되는 weak entity이다.  const Graph& \_graph;  // 각 node까지의 누적 cost  std::vector<double> \_cost\_to\_this\_node;  // 최소 비용 트리  std::vector<const Edge\*> \_shortest\_path\_tree;    // 최소 비용 트리를 만들기 위한 임시 공간  std::vector<const Edge\*> \_search\_frontier;  // 출발지 && 목적지  int \_source;  int \_target;  // 우선순위 큐  std::unique\_ptr<IndexedPriorityQLow<double> > \_pq;  };  Time Sliced A\* Algorithm  // template 인자로 탐색 대상 그래프와 휴리스틱 전략을 받는다.  // 마찬가지로 SearchTimeSliced class를 상속받는다.  template <class Graph, class HeuristicPolicy>  class SearchAStarTS : public SearchTimeSliced<typename Graph::Edge>  {  private:  // 대상 그래프의 Node 타입과 Edge 타입을 제공한다.  typedef typename Graph::Edge Edge;  typedef typename Graph::Node Node;  public:  SearchAStarTS(  const Graph& graph,  int source,  int target);  virtual ~SearchAStarTS() override;  // 호출될 때 이 method는 알고리즘을 한 탐색 cycle 만큼 동작시킨다.  // (TARGET\_FOUND, TARGET\_NOT\_FOUND, SEARCH\_INCOMPLETE)  // 중 맞는 열거값을 반환한다.  virtual int cycleOnce() override;  // 출발지로부터 도착지까지의 최단 경로를 반환한다. (node의 index)  virtual std::list<int> getPathToTarget() const override;  // 출발지로부터 도착지까지의 최단 경로를 반환한다.  virtual std::list<PathEdge> getPathAsPathEdges() const override;  // 최단 경로 트리를 반환한다.  virtual std::vector<const Edge\*> getSPT() const override;  private:  // 탐색 알고리즘은 반드시 graph를 갖고 생성되는 weak entity이다.  const Graph& \_graph;  // 누적 가중치  std::vector<double> \_g\_costs;  // 휴리스틱 가중치  std::vector<double> \_f\_costs;  // 최단 경로 트리  std::vector<const Edge\*> \_shortest\_path\_tree;  // 최단 경로 트리를 만들기 위한 임시 공간  std::vector<const Edge\*> \_search\_frontier;  // 출발지 && 도착지  int \_source;  int \_target;  // 우선순위 큐  std::unique\_ptr<IndexedPriorityQLow<double> > \_pq;  };  다음으로 경로 계획자를 정의한다.  경로 계획 객체가 필요한 이유는 다음과 같다.   1. 임의의 그래프에서, 임의의 탐색 알고리즘을 사용할 수 있도록 하기 위해서 2. 경로탐색을 원하는 entity가 경로를 얻는 동적인 방법을 제공하기 위해서   class AbstCharacter;  class GameMap;  class PathPlanner  {  public:  // 게임 맵이 갖고 있는 그래프의 Edge와 Node 타입을 제공한다.  typedef GameMap::NavGraph::Edge Edge;  typedef GameMap::NavGraph::Node Node;  public:  ~PathPlanner();  explicit PathPlanner(AbstCharacter\* const owner);  // 위치로부터 가장 근접한 Node의 index를 얻는다.  int getClosestNodeToPosition(cocos2d::Vec2 pos)const;  // 특정 위치로의 경로를 요청한다.  bool requestPathToPosition(cocos2d::Vec2 target\_pos);  // 경로가 얻어진 경우, 경로를 반환한다.  std::list<PathEdge> getPath();  // 시분할 탐색을 위한 경로 계획자의 method  // 탐색 알고리즘의 cycleOnce() 함수를 호출함으로써 탐색을 시분할로 진행시킨다.  int cycleOnce();  // getters && setters  cocos2d::Vec2 getDestination()const;  void setDestination(cocos2d::Vec2 new\_pos);  cocos2d::Vec2 getNodePosition(int idx) const;  LARGE\_INTEGER getTime();  private:  // 경로를 매끄럽게 해주는 함수들  void smoothPathEdgesQuick(std::list<PathEdge>& path);  void smoothPathEdgesPrecise(std::list<PathEdge>& path);  // 새로운 탐색을 위한 준비 과정  void getReadyForNewSearch();  // 경로 계획자는 반드시 경로를 얻을 수 있는 어떤 entity에 대한  // weak entity로서 생성된다.  AbstCharacter\* const \_owner;  // 경로 계획자는 반드시 게임 맵에 대한 참조를 갖으며 생성되는 weak entity이다.  const GameMap::NavGraph& \_graph;  // 현재의 탐색 알고리즘  std::unique\_ptr<SearchTimeSliced<Edge> > \_current\_algorithm;  // 목적지  cocos2d::Vec2 \_destination\_pos;  // 시분할을 위한 time 변수  LARGE\_INTEGER \_time;  };  **Class Diagram**    **출력 화면**  구현된 시스템을 이용하여 화면에서 보이듯이 장애물을 피해 Graph Search를 하고, 경로대로 움직인다. |
| **Message Module** |
| 게임 내 entity들을 서로 수많은 상호작용을 하기 때문에, 이를 위해서는 어떤 message를 전달해 줄 시스템이 필요하다. Message System을 구현하여 이를 가능하게 한다.  Messaging을 지속적으로 확인하지 않고, 송신자 측에서 직접 알려주는 형태로 구현되므로 Design Pattern 중 Observer Pattern을 이용하면 효율적이다.  **Implementation**  Message Module을 실제로 구현해보자. (쉬운 설명을 위해 실제 코드보다는 많이 생략되었다.)  먼저 Message를 보내는 작업을 어떤 방법으로 해결해야 하는지 알아보자. 가장 쉬운 방법으로는, Telegram(Message의 내용물을 담은 구조체)을 담은 queue를 모든 Entity가 갖도록 하고, World를 통해 각 Entity에게 보내도록 하는 것이다. 그리고 각 Entity들은 모두 update전에 자신의 message queue를 지속적으로 확인하는 것이다. 하지만 이렇게 구현되면, 메모리, 성능, 코드, 모든 부분에서 효율적이지 못하다.  그래서 더 좋은 방법으로는, Observer Pattern을 응용한 Notification Center를 구현하는 것이다.  일단 Observer Pattern의 아이디어를 살펴보면, 보내는 쪽에서 받는 쪽에 대한 참조를 갖고 있도록 하고, message를 받았을 때 받는 쪽에서 취해야 할 행동을 직접 호출해 주는 것이다. 이렇게 하면, 받는 쪽은 message가 오기를 지속적으로 기다릴 필요가 없어진다. 하지만, Observer Pattern으로의 구현은 message 교환이 1 : N인 관계에서는 효과적이지만, N : N의 관계에서는 그렇지 않다. N:N에서는 보내는 쪽에서 받는 쪽의 참조를 갖기 위해 너무 많은 메모리가 사용되기 때문이다. 이 문제를 해결하기 위해 EntityManager class를 구현한다. EntityManager는 모든 Entity의 id와 pointer를 map의 형태로 갖고 있다. 또한 Singleton 객체로 구현되어 있기 때문에 어디서든 참조가 가능하다. EntityManager가 구현되었다면, message를 보내기 위한 참조를 각 Entity가 갖고 있을 필요가 없어진다. Id만 있으면 언제든지 EntityManager를 통해 pointer를 얻을 수 있기 때문이다. 이것이 Notification Center의 핵심 아이디어이다.  MessageDispatcher에서는 dispatchMsg라는 method를 통해 message를 한 Entity로부터 다른 Entity로 보낸다. 이는 sender id와 receiver id를 받고, message type과 지연시간과 추가적인 정보를 담을 void\*를 받아서 수행된다. 내부적으로는, delay가 0일 때 EntityManager를 통해 id에 대한 pointer를 얻어서 discharge method를 호출한다. 만약 delay가 있는 message이면, priority queue에 넣는다.  MessageDispatcher에서는 dispatchDelayedMsg라는 method를 통해 모든 지연된 message를 보내도록 한다. 이 함수는 priority queue의 지연시간이 가장 작게 남은 message부터 보면서, 현재 시간과 비교하여(현재 시간은 FrameCounter class를 통해 얻는다.) 지속적으로 보내져야 하는지 판단하도록 한다. 그러므로 이 method는 world의 update안에 있어야 한다.  MessageDispatcher의 discharge method는 인자로 받은 Entity의 pointer를 통해, Entity의 handleMessage라는 함수를 직접 호출해 주어, 스스로 message에 대해 동작하도록 한다.  // Message 전달을 담당하는 class  // World의 모든 Message는 이 class가 담당하므로,  // 단 하나의 instance만 생성하도록 Singleton으로 구현한다.  class MessageDispatcher : public Singleton<MessageDispatcher>  {  public:  // 외부에서 Entity는 이 method를 통해 다른 Entity에게 message를 보낸다.  // message를 보내기 위해 EntityManager에서 id를 통해 Entity의 pointer를 얻고  // discharge 함수를 호출한다.  void dispatchMsg(  double delay,  unsigned int sender,  unsigned int receiver,  int msg,  void\* const extra\_info);  // Entity는 가끔 지연된 message를 보낸다.  // 이 method는 priority queue에서 지연시간이 다 된 모든 message를 보낸다.  void dispatchDelayedMsg();  private:  // Message를 보내는 method  // Observer pattern을 이용하여 구현되어 있다.  // receiver의 handleMessage를 직접 호출해 준다.  void discharge(BaseGameEntity\* const receiver, const Telegram& msg);  // 모든 message와 지연된 message를 시간을 기준으로 관리한다.  priority\_queue<Telegram, vector<Telegram>, greater<Telegram> > \_pque;  };  **Class Diagram**    **출력 화면**    한 entity가 다른 entity를 공격하고 있는데, 공격에 대한 데이터를 전달시키는 도구로써 Message Module을 이용한다. 왼쪽에 Console 창에 뜨는 것을 보면 메시지가 성공적으로 이동했음을 알 수 있다. |
| **Cell Space Partition System** |
| [수많은 entity들이](http://www.incruit.com/tools/spell/) 서로 충돌검사를 하고, 또한 world가 크기 때문에 이는 성능상에 심각한 문제를 줄 수 있다. 이 문제를 해결하기 위해 공간을 Cell 단위로 [나누고](http://www.incruit.com/tools/spell/) 공간 내 entity들 사이에서만 충돌검사를 하도록 개선하여야 한다.  Cell Space Partition System을 c++의 [template를](http://www.incruit.com/tools/spell/) 이용하여 구현한다. 그러면 여러 종류의 충돌에 대해서 일반화된 공간분할을 할 수 있다.    **Implementation**  Cell Space System을 실제로 구현해보자. (쉬운 설명을 위해 실제 코드보다는 많이 생략되었다.)  먼저 Entity들의 pointer를 저장할 Cell을 정의하여야 한다.  template <class Entity>  struct Cell  {  // Entity를 저장하는 임시 공간,  // Entity로는 AbstCharacter\*와 같은 ponter type이 들어오고,  // members는 단순히 대상을 참조할 뿐, life cycle을 책임지지 않는다.  std::list<Entity> members;  // Cell의 유효범위(직사각형)를 AABBox로 정의한다.  AABBox2D bounding\_box;  };  AABBox는 다음과 같이 정의된다.  class AABBox2D  {  private:  // 직사각형의 왼쪽 아래, 오른쪽 위 2D 위치  cocos2d::Vec2 \_bottom\_left;  cocos2d::Vec2 \_top\_right;  public:  // 겹침판단을 위한 함수  bool isOverlappedWith(const AABBox2D& other) const;  // gettor && setter  const cocos2d::Vec2& getTopRight() const { return \_top\_right; }  const cocos2d::Vec2& getBottomLeft() const { return \_bottom\_left; }  double getTop() const { return \_top\_right.y; }  double getLeft() const { return \_bottom\_left.x; }  double getBottom() const { return \_bottom\_left.y; }  double getRight() const { return \_top\_right.x; }  };  위와 같이 Cell이 정의되었으면, 이를 이용하여 CellSpace를 정의한다.  // CellSpacePartition은 공간분할 시스템의 중추적인 역할을 하는 자료구조이다.  // 어떠한 Entity에 대해서도 공간분할을 적용할 수 있도록 Entity를 template type으로 받는다.  template <class Entity>  class CellSpacePartition  {  public:  // Entity 추가  inline void addEntity(const Entity& ent);  // Entity 제거  inline void removeEntity(const Entity& ent);    // Entity가 움직일 수 있다면,  // 한 Cell에서 다른 Cell로 참조가 이동하여야 할 것이다.  // update method를 통해 이 문제를 해결한다.  inline void updateEntity(const Entity& ent, cocos2d::Vec2 old\_pos);  // 도형이 입력으로 들어오면,  // 그 도형 근처에 있는 Entity들을 \_neighbers vector에 세팅한다.  inline void queryNeighbors(geometry::Shape\* bounding\_shape);  // \_neighbers vector에 세팅된 Entity들을 순회하기 위한 장치들을 구현한다.  inline Entity& begin();  inline Entity& next();  inline bool end();  private:  // 위치가 입력으로 들어오면,  // 위치에 맞는 Cell의 index를 hashing하여 반환하는 함수  inline int positionToIndex(const cocos2d::Vec2& pos)const;  // CellSpacePartition class는 여러 Cell로 나누어져 있다.  std::vector<std::unique\_ptr<Cell<Entity> > > \_cells;  // 어떤 도형에 대해, 그 근처의 Entity들을 저장할 임시 공간  std::vector<Entity> \_neighbors;    // 어떤 도형에 대해, 그 근처의 Entity들을 순회하기 위한 iterator  typename std::vector<Entity>::iterator \_cur\_neighbor;  // 분할 공간의 전체 크기  double \_space\_width;  double \_space\_height;  // 분할 공간을 나누는 Cell의 개수  int \_num\_cellsX;  int \_num\_cellsY;  };  **Class Diagram**    **출력 화면**    분할된 공간(Cell)은 위와 같이 Character의 주소를 list형태로 보관한다. 왼쪽 조그만 숫자로 7이 화면에 나타나 있는데 이는 Character의 개수를 의미한다. 이 그림은 Character를 보관하는 공간분할 시스템이다. 하지만 더 성능상의 이득을 보기 위해서, CollisionArea를 보관하는 공간분할 시스템과, Graph Node를 보관하는 공간분할 시스템도 구현되어 있다.  (물론 c++의 template을 이용하여 코드상의 중복을 피한다.) |
| **Debug Renderer** |
| 게임을 개발하는 내내 다양한 문제들을 접하게 될 것이다. 하지만, 너무나도 큰 프로젝트이기 때문에, Console 창에 출력된 log만으로는 효율적인 디버깅이 불가능하다.  화면에 직접적으로 다양한 도형들을 그려서 더 효율적인 디버깅을 꾀한다.  어디서든 사용된다는 Debug의 특성에 부합시켜 Debug Renderer는 Design Pattern중 Singleton Pattern로 구현하면 효율적이다.  출력 화면    위 그림과 같이 다양한 도형들을 화면에 뿌려주어 디버깅에 많은 도움을 주도록 한다. |
| **Target System** |
| [Entity 간](http://www.incruit.com/tools/spell/) targeting을 하기 위해 c++에서의 포인터를 이용할 계획이다. 하지만 동적으로 생성되었다 [소멸하는](http://www.incruit.com/tools/spell/) [entity 간](http://www.incruit.com/tools/spell/) targeting에서는 dangling [pointer 문제가](http://www.incruit.com/tools/spell/) 빈번하게 발생한다[. 이런 형태로 targeting이](http://www.incruit.com/tools/spell/) 원활히 되지 않는다면 성능상에 치명적인 문제가 생길 수 있다. 또한 설계의 관점에서 Character의 종류마다 target을 설정하는 방법이 다를 수 있다. 동시에 이런 target 설정 방법은 선택할 수 있도록 구현되어야 하기 때문에 따로 class로 빼도록 설계하는 것이 유리하다. Targeting System을 구현하여 이런 문제들을 해결한다.  Class Diagram |
| **Weapon System** |
| 다양한 종류의 Character들이 있고 또 다양한 종류의 공격 방법들이 있다. 이런 다양한 방법들은 서로 조합되어 사용될 수 있기 때문에 Weapon System이라는 중간 매체를 통해 전략적으로 선택이 가능하게 한다. |
| **Steering Behavior** |
| 게임 내 움직일 수 있는 entity들을 사실적으로 표현하기 위해서는 실 세계에서의 움직임 방법과 흡사하여야 한다. 그러기 위해서는 물리적인 규칙들을 따라야 한다. 위치, 방향, 속도, 질량 등 다양한 물리적인 속성들과 그에 맞는 연산들이 있는데, 이러한 것들을 조종 행동(Steering Behavior)로 객체화 시켜 사용한다.    위 그림은 다양한 조종행동 중 하나인 leader following에 대한 그림이다. 어떤 대상을 한 agent가 따라간다고 했을 때, 그 움직임이 물리적인 연산을 거쳐서 자연스럽게 표현되기 위해서는 위처럼 여러 방향에 대한 벡터 값을 속도로써 누적시키고 완성된 하나의 벡터로 entity들을 움직이도록 해야 한다.  Class Diagram    Class Diagram 상으로는 특별할 것이 없지만, 내부 method를 굉장히 많이 갖는다. |
| **Animation Module** |
| 게임 내 entity들이 움직이고 있다는 느낌을 주기 위해서 여러 frame으로 구성된 이미지 파일을 준비하고, 이 frame들을 현재 상태에 맞게 [차례로](http://www.incruit.com/tools/spell/) 화면에 [rendering 해](http://www.incruit.com/tools/spell/) 주어야 한다. [Cocos2d](http://www.incruit.com/tools/spell/)-x에서 제공해주는 Animation class가 [있지만, 이는](http://www.incruit.com/tools/spell/) 우리에게 필요한 모든 표현 방법을 제공해주지 않기 때문에, 직접 Animation Module을 개발한다. |
| **Sensory Memory Module** |
| http://what-when-how.com/wp-content/uploads/2012/06/tmp613218_thumb.png  게임 내 Bot의 행동을 좀 더 사실적으로 표현하기 위해서는 Bot에게 “감각”과 “기억” 에 대한 처리를 할 수 있는 능력을 주어야 한다. Sensory Memory 모듈을 통해 Bot들은 주변 환경에 대해 감지하고, 일정 시간 동안 기억 할 수 있도록 한다.  Class Diagram    출력 화면    그림의 왼쪽부분에 label에 나온 데이터가 agent의 memory에 대한 data이다. Agent는 주변 환경에 대해 지각하고, 그에 대한 내용을 memory라는 객체로 기억한다. |
| **Timer Module** |
| 시간은 개임 내 다양한 상호작용에 효율적으로 사용될 수 있는 데이터이다. 시간을 측정하는 다양한 Timer들을 구현하여 상호작용에 도움을 준다.  Class Diagram |
| **Trigger System** |
| 게임 내 entity들 간의 event들을 생성, 처리하기 위해서 Trigger라는 개념을 도입한다.  영역, 방법, 조건을 정의하여 다양한 trigger들을 만들고 이를 이용하여 게임의 흐름을 구현한다.    위 그림과 같이, trigger를 이용하면 상당히 많은 게임의 흐름을 표현할 수 있다.  Class Diagram    출력 화면    위 그림은 SpawnArea라는 trigger에 대해 보여준다. SpawnArea는 자신만의 영역을 갖고 일정 시간이 지날 때마다 새로운 agent를 생성하는 trigger이다. 위 그림에서 agent가 새롭게 생성되는 모습을 확인할 수 있다. |
| **Control Module** |
| 게임을 실제로 플레이어가 조작하기 위하여, UI를 구현하고 그 각 UI에 control 기능들을 넣어준다. |
| **Particle System** |
| 게임 내 화려한 effect들을 구현하기 위해 particle system을 구현한다. 이를 좀 더 편하게 하기 위해서 Particle Editor를 이용하여 plist 포맷으로 data를 만들고 이를 parsing 하여 사용한다. |
| **User Interface** |
| 게임 조작을 위한 다양한 버튼과, 스크롤 바, 정보 창 등의 User Interface들을 구현한다. |
| **Game Flow** |
| 게임의 진행을 구현한다. 우측 진영과 좌측 진영에서 일정 주기로 7기의 Bot들이 출몰하고 이는 각각 상대 진영으로 공격을 간다. 상대 진영으로 갈 수 있는 2가지의 경로가 있고, 각 경로에서 마주친 Bot들을 [서로](http://www.incruit.com/tools/spell/) 공격한다.  일정 시간이 지날 때마다 [Player들에](http://www.incruit.com/tools/spell/) 일정량의 gold가 주어지고 이를 이용하여 좋은 아이템을 [살](http://www.incruit.com/tools/spell/) 수 있다. 또 각 경로 사이사이의 Natural Bot들이 존재하고 이것들을 사냥할 경우 일정량의 gold를 얻을 수 있다. 이러한 Game의 Flow를 조절하기 위해 Trigger를 [적극](http://www.incruit.com/tools/spell/) 이용한다. |
| **Skill Entity** |
| Skill Entity는 자신에 맞는 Particle을 갖고,  움직일 수 있으며, 적절한 방법으로 world에서 상호작용을 하는 entity이다.  종류가 상당히 많기 때문에 상속을 통하여 abstract class를 파생시키는 방법으로 구현한다.  출력 화면 |
| **Item Entity** |
| 게임의 아이템을 정의한다. 아이템에 대한 이미지와, 정보들을 갖고, 다른 entity들의 data를 변경시켜준다. |
| **Structure Entity** |
| 게임의 구조물, 지형을 정의한다. 구조물에 대한 이미지와 정보들을 갖는다.  출력 화면 |
| **Collision Area Entity** |
| 게임의 충돌영역을 정의한다. 구조물과 구별시킨 이유는, 성능상의 문제가 있을 수 있기 때문에 큰 영역을 한번에 영역으로 처리할 수 있도록 class를 분리하였다.  출력 화면 |
| **Character Entity** |
| 게임 내 Character를 정의한다. Character는 각 image에 대한 frame들을 출력하고 또 여러 행동들을 갖는다. 다양한 방법으로 다른 entity들과 상호작용한다.  출력 화면 |
| **Game Map** |
| 게임의 Map을 정의한다. Map의 기본 배경은 tile로 덮여 있고, 그 위에 여러 종류의 Structure와 CollisionArea들을 넣어서 지형지물을 구성한다. 그리고 graph를 갖고 있어서, 지형지물을 피할 수 있는 경로에 대한 정보를 제공한다. 또한 Trigger와 Tower의 위치 정보를 저장하고, 그 외에도 Agent의 경로를 돕기 위한 정보 등 다양한 정보를 저장한다.  이를 위해 Tiled라는 Game Map 개발 tool을 이용한다. 이 tool에서는 원하는 형태로 편하게 map을 제작할 수 있는 환경을 제공하고, 또한 사용자 지정 변수들을 등록할 수 있도록 도와준다. 그 후 해당 map을 xml중 하나인 .tmx 포맷으로 변환시켜준다. 게임 내부에서는 이 파일을 parsing하여 사용한다.  Tiled를 이용한 Game Map 제작 |
| **Game World** |
| 게임 내부의 모든 entity들과 game map과 다양한 장치들간의 상호작용을 담당한다.  Class Diagram |
| **Network** |
| Network를 통해 여러 [사용자의](http://www.incruit.com/tools/spell/) data가 [상호 작용할](http://www.incruit.com/tools/spell/) 수 [있게 하려고](http://www.incruit.com/tools/spell/), Game Server를 개발한다. 각 [사용자들은](http://www.incruit.com/tools/spell/) Game Client로서 서버와 통신하도록 한다. 게임이 single user에서 [multi](http://www.incruit.com/tools/spell/) user로 넘어가게 되면, 시스템의 구조가 판이하게 변경된다. [서버는](http://www.incruit.com/tools/spell/) 화면에 출력되는 기능이 필요가 없으므로, rendering과 관련된 함수들은 코드에서 제거된다. 그리고 [client는](http://www.incruit.com/tools/spell/) 충돌처리, 길 찾기, 메시지 교환 등등 다양한 상호작용이 있었지만, 서버가 생기게 되면, 이러한 작업이 [더는](http://www.incruit.com/tools/spell/) client에서 이루어져서는 안 된다. 왜냐하면, [모든 client의](http://www.incruit.com/tools/spell/) 정보가 종합된 상태에서 위와 같은 작업들이 처리되어야 하고, 모든 정보가 종합되는 위치는 Game Server가 되기 때문이다. 그러므로 world에서 있던 다양한 장치와 entity들의 작업들은 모두 Game Server가 감당하고, 모든 화면출력에 대한 작업은 client가 담당하게 된다.    그러므로 위 그림과 같은 구조로 구성된다. 서버는 World를 갖고 그 안에 다양한 장치들과, entity들의 상호작용을 통한 완성된 data를 만들어낸다. 그리고 주기적으로 그 데이터들을 각 클라이언트들에게 뿌려준다. 각 Client들은 사용자의 event를 처리하고, 주기적으로 서버로부터 받은 데이터들을 화면에 출력해 준다. 여기서 image와 같은 정적인 data들은 서버로부터 받는 것이 아니라 직접 갖고 있도록 한다. |
| **Packet** |
| 큰 구조는 위 Network에서의 설명과 같지만, 실제로 이를 구현하려고 하면 많은 어려움이 있다. 가장 먼저 해결해야 할 문제는, 서버와 클라이언트간의 교환하는 data의 종류가 상당히 많고, 그 중에는 복잡한 형태의 데이터도 있다는 것이다. 예를 들면, 마우스 클릭에 대해서 client -> server로는 위치를 나타내기 위해 float x, float y와 클릭한 마우스 버튼을 나타내기 위한 int button이 있을 것이다. 그러므로 { float x, float y, int button } 이라는 형태의 packet이 정의되어야 하고, 이를 stream으로 만드는 적절한 encoding 과정과 decoding 과정이 정의되어야 할 것이다. 하지만 데이터 교환 시나리오는 이 외에도 많기 때문에 위와 같은 packet의 종류도 훨씬 많아질 것이다. 또한 그 중에는 가변적인 형태의 packet도 있다. 예를 들어 100마리의 entity가 있을 때 그 각 entity의 rendering을 위한 data를 client에게 서버가 주어야 한다고 해보면, 각 entity에 대해 id, x, y, frame 을 주어야 할 것이다. 하지만 이것을 100번 하면 system call을 낭비하게 되므로 이를 감싸는 형태로 한번 더 packet을 만들 필요가 있다. 그러면 packet의 형태는 다음과 같을 것이다. { { int id, float x, float y, int frame }, { int id, float x, float y, int frame }, … ,{ int id, float x, float y, int frame } }  하지만 이런 가변적인 형태의 packet은, 직접 code로 구현하기가 쉽지가 않다. 그러므로, google의 직렬화 library인 flatbuffers library를 이용한다. Flatbuffers library는 먼 IDL로 packet의 필요한 데이터를 정의한다.  Ex) client의 chatting을 위한 packet을 IDL로 표현하면 다음과 같다.  namespace packets;  attribute "priority";  table PacketChatMessageToServer  {  id:uint;  chat\_message:string;  }  root\_type PacketChatMessageToServer;  이를 flatc 라는 컴파일러로 컴파일한다. 그러면, 위 packet에 대한 builder와 verifier와 decoder등의 code를 generate할 수 있다.  이 코드를 사용하면, 어떤 형태의 packet이라도 직렬화, 역직렬화 할 수 있다.  하지만, 아직 문제가 완전히 해결된 것은 아니다. Packet의 직렬화와, 역직렬화는 가능해졌지만, server와 client가 packet을 교환할 때 packet이 어떤 종류인지 보낸 곳에서는 알지만 받는 곳에서는 모른다. 받을 때는 stream으로 받기 때문에 이를 어떤 방식으로 decode 해야 할 지 알 길이 없다. 그러므로, flatbuffers를 이용하여 직렬화한 stream을 GameMessage라는 class로 한 번 더 wrapping 한다. GameMessage는 크게 header와 body로 나누어지고, 직렬화한 stream의 길이에 대한 정보와, stream의 종류에 대한 정보를 size, type의 형태로 header에 갖는다. Body에는 직렬화한 stream을 저장한다.  Body (stream을 저장)  Header (size와 type을 저장)  이제 서버에서 클라이언트로 packet을 보내면, client에서는 먼저 header를 해석하고 type을 알아낼 수 있게 되었다. Type을 파악하면, type에 맞는 flatbuffer의 decoder를 이용하여 packet을 역직렬화하여 사용한다. |
|  |

1. **이벤트 처리 시나리오**

게임에서 필요한 다양한 시나리오들을 정의하고 앞서 언급한 장치들을 이용하여 이런 시나리오들에 대한 해결책을 제시한다.

|  |
| --- |
| **Object picking by mouse** |
| Object를 클릭하여 object의 정보를 가져올 필요가 있다. 이를 위해 Cocos2d-x의 image bitmap을 가져와서 원하는 sprite를 갖고 있는 entity의 id를 가져오도록 한다. |
| **Mouse click on view** |
| 화면에 마우스 클릭을 함으로써 player의 character를 원하는 위치로 이동시킬 수 있다. 이를 위해 여러 가지 작업이 필요하다. 클릭 이벤트가 일어난다면 Player의 character가 [가진](http://www.incruit.com/tools/spell/) 복합목적 시스템에 움직임에 대한 목적을 넣어준다. 이 목적은 다시 부 목적들로 이루어지는데, 만약 출발 위치로부터 도착 위치까지 선분과 충돌영역들과의 [충돌 없다면](http://www.incruit.com/tools/spell/), 즉시 이동한다. 만약 충돌이 있다면, 그래프 탐색을 통해 이동한다. [이때](http://www.incruit.com/tools/spell/) 항해 시스템이 이용된다. |
| **Mouse click on buttons** |
| 버튼의 클릭에는 여러 가지 시나리오들이 있는데, 옵션 창을 여는 이벤트, 원하는 아이템(아이템에 대한 버튼)을 선택하는 이벤트 등등이 있다. 이를 해결하기 위해 조작 이벤트를 담당하는 scene과 실제 게임에 영향을 줄 수 있는 world 사이에 밀접한 관계가 필요하다. 각 시나리오에 대해 entity들에게 적용되는 영향들을 world를 통해 구현해 두고, scene에서 얻은 event를 world로 forwarding함으로써 문제를 해결할 수 있다. |
| **Press keys** |
| Mouse click on buttons와 마찬가지로 key event도 scene에서 일어난 event들을 world에 forwarding함으로써 해결할 수 있다. |
| **Respawn of monsters** |
| 게임이 진행되기 위해서 각 진영에서는 주기적으로 monster들이 재생성 되어야 한다. 이러한 동작을 구현하기 위해서 trigger들을 이용한다. Trigger는 world 내에 존재하는 일종에 entity이면서 동시에 world 내 [여러 event를](http://www.incruit.com/tools/spell/) 생성시켜주는 작업을 한다. Trigger는 영향 영역, 조건, 발생 event들을 갖기 때문에, [각 속성을](http://www.incruit.com/tools/spell/) 어떻게 개발하느냐에 따라 [수많은](http://www.incruit.com/tools/spell/) world의 event들을 구현할 수 있다. Monster의 재생성도 쉽게 해결할 수 있다. |
| **Activate buildings** |
| 게임의 진행 중에는, 중간에 활성화되는 건물들이 있다. 이런 활성화라는 동작을 만들기 위해 위와 마찬가지로 Trigger 시스템을 이용하여 쉽게 구현할 수 있다.  여기에서는 또 다른 신경 써야 할 부분이 있는데, 그것은 Trigger가 entity에게로 message를 보낸다는 점이다. [Respawn](http://www.incruit.com/tools/spell/) of [Monsters는](http://www.incruit.com/tools/spell/) Trigger가 world에 event를 주는 것이지만, 지금 경우는 Trigger가 다른 entity에게 event(message)를 주는 것이다. Trigger 또한 entity이기 때문에, 이런 동작을 구현하기 위해서는 [entity 간](http://www.incruit.com/tools/spell/) message 교환이 가능하도록 도와주는 장치가 필요하다. Message Module을 구현하여 이 문제를 해결할 수 있다. |
| **Agents** |
| 게임 내에는 스스로 목적을 생성하고 스스로 판단하여 행동하는 agent들이 존재한다. Agent들이 갖는 몸체는 character이고, 스스로 목적을 생성하기 위해서 복합목적 시스템을 갖고 있어야 한다. 복합목적 시스템의 가장 상위목적으로는 Think라는 목적이 있는데, 이 목적은 스스로 어떤 다음 행동(목적)을 선택할지에 대한 연산을 수행한다. 각 목적과 현재 상황을 기억하고, [그것에 맞게](http://www.incruit.com/tools/spell/) 욕구에 대한 가중치를 비교하여 가장 적합하다 판단되는 목적을 선택하도록 한다. |
| **Collision between entities** |
| Entity들 사이에는 충돌이 존재한다. Character가 Collision Area와 충돌할 때, Character가 Skill과 충돌할 때, Skill이 Collision Area와 충돌할 때 등등 충돌에 대한 여러 시나리오가 존대한다. 모든 시나리오에 대해 일반적인 해결책을 만들기 위해 [Template를](http://www.incruit.com/tools/spell/) 이용하여 [entity 간](http://www.incruit.com/tools/spell/) 충돌에 대한 함수들을 구현한다. 하지만 이러한 일반화가 갖는 문제는, entity가 갖는 경계 영역이 어떤 도형을 갖는지 모른다는 것이다. 예를 들어 원을 경계 영역으로 갖는 Skill과 다각형을 경계영역으로 갖는 Collision Area가 충돌할 때는 원과 다각형의 겹침을 판단하는 함수를 사용하여야 하고, 원을 경계 영역으로 갖는 Character와 원을 경계 영역으로 갖는 Skill과 충돌할 때는 원과 원의 겹침을 판단하는 함수를 사용하여야 한다. 하지만 entity가 어떤 도형을 경계영역으로 갖는지 모르기 때문에 이 문제를 해결하기 위해서는 도형의 겹침을 판단하는 함수를 다형성을 이용하여 구현해야 한다. 이를 위해 직접 geometry library를 구현하고 library에 일반적인 도형의 겹침 판단 함수들과 추상적으로 사용될 겹침 판단 함수들을 모두 구현한다. |
| **High performance in collision detection** |
| 게임에서 충돌처리라는 작업은 반드시 존재해야 한다. 하지만 충돌처리작업은 게임 내 성능을 결정하는 매우 중요한 요소이다. 그러므로 충돌검사를 어떤 형태로 [할지](http://www.incruit.com/tools/spell/) 좀 더 생각해볼 필요가 있다. 기하학적인 도형을 갖는 [entity 간](http://www.incruit.com/tools/spell/) 충돌을 검사하기 위해서 쉬운 방법으로 다른 [모든 entity에](http://www.incruit.com/tools/spell/) 대해 [loop을](http://www.incruit.com/tools/spell/) 돌며 자신의 겹침 영역과 상대의 겹침 영역이 겹치는지 확인할 수 있다. 하지만 성능적으로 봤을 때 이 방법은 문제가 있다. 어떤 entity가 map의 왼쪽 아래 [끝 부분에](http://www.incruit.com/tools/spell/) 있다고 가정하자. 그리고 map의 오른쪽 위 [끝 부분에](http://www.incruit.com/tools/spell/) [다수의 entity가](http://www.incruit.com/tools/spell/) 모여있다고 가정하자. 이 상황에서 왼쪽 아래에 있는 entity가 굳이 오른쪽 끝에 있는 [다수의 entity와](http://www.incruit.com/tools/spell/) 충돌검사를 할 필요가 있을까? 자신의 주변 영역에 대해서만 검사하면 되는데 저 멀리 있는 entity들과 검사하는 것은 큰 낭비이다. 이런 문제는 map이 커지고 entity가 많아질수록 O(n^2)의 복잡도로 더 큰 성능저하를 일으킨다. 이 문제를 해결하기 위해 공간 분할이라는 개념을 이용한다. 전체 map을 N x M의 [cell 단위로](http://www.incruit.com/tools/spell/) [나누고](http://www.incruit.com/tools/spell/), entity와 겹치는 cell 내부에 있는 entity들과만 충돌검사를 수행한다[. 또한, 자신에](http://www.incruit.com/tools/spell/) 대한 cell을 질의할 때 현재 위치를 [hashing하여](http://www.incruit.com/tools/spell/) 적절한 cell의 id를 찾아낼 수 있으므로 cell을 [탐색하는](http://www.incruit.com/tools/spell/) overhead도 [최소화시킬](http://www.incruit.com/tools/spell/) 수 있다. Map이 커지고 entity가 많아질수록 이 방법은 좋은 성능을 끌어낼 것이다. |
| **Path Finding** |
| 게임 내부에 있는 Character들은 사방으로 움직일 수 있다. 하지만 충돌영역들이 정의된 map에서 원하는 위치로 이동하지 못하는 상황은 충분히 발생할 수 있다. 어떤 Character의 목적지가 오른쪽 나무인데, Character의 위치와 나무 사이에 벽이 있다고 가정하자. 그렇다면 Character는 벽에 걸려 오른쪽 나무에 [도달하지 못할](http://www.incruit.com/tools/spell/) 것이다. 이 문제를 해결하기 위해서는 목표 위치로 도달하기 위한 길 찾기 과정이 필요하고, 이를 위해서는 map에 대한 적절한 navigation graph가 정의되어 있어야 한다[. 또한, 위치로](http://www.incruit.com/tools/spell/) 이동하기 위한 적절한 탐색 알고리즘도 정의되어 있어야 한다. 이러한 동작들을 일괄적으로 수행해주는 Navigation System을 구현한다. |
| **Interact between entities** |
| 모든 entity들은 서로 상호작용을 한다. 그러기 위해 서로 message를 교환하여야 하므로 Message Module을 구현한다. |

1. **구현 환경**

|  |  |
| --- | --- |
| Client | OS : Window  IDE : Visual Studio  Language : C++  Framework : cocos2d-x  (Rendering, Sound, Event, Scheduling)  Resource : Particle Editor, Texture Packer, Tile Editor |
| Server | OS : Window  IDE : Visual Studio  Language : C++  Library : Boost Asio Library  (고성능 Game Server를 구축하기 위해)  Google Flatbuffers Library  (게임 내부의 객체를 직렬화하기 위해) |

1. **소스 코드**

Server

<https://github.com/insooneelife/MOBAserver>

Client

<https://github.com/insooneelife/MOBAclient>

1. **참고 문헌**

게임 & 인터랙티브 애플리케이션을 위한 수학 – James M. Van Verth

Programming Game AI by Example – Mat Buckland

Effective C++ – Scott Meyers

Boost Asio를 이용한 네트워크 프로그래밍 – 최흥배