Quadtree for Seamless Loading

Homepage: https://sites.google.com/site/doc4code/
Email: goldpotion@outlook.com
2011/6/9

이 문서에서는 quadtree에 대해서 알아보고, 효과적인 구현 방법에 대하여 소개하겠다.
그 다음, quadtree를 심리스 로딩에 어떻게 이용하는 지에 대해서도 알아보겠다.

Original Quadtree

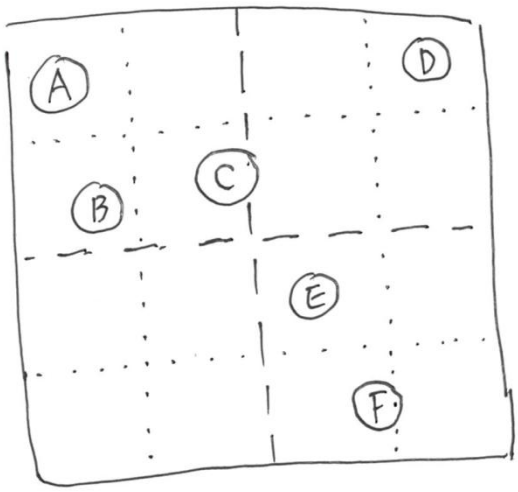


그림 1.

Quadtree는 4개의 지역으로 나누어서 관리한다.

그림 1을 Quadtree로 표현하면 아래와 같다.

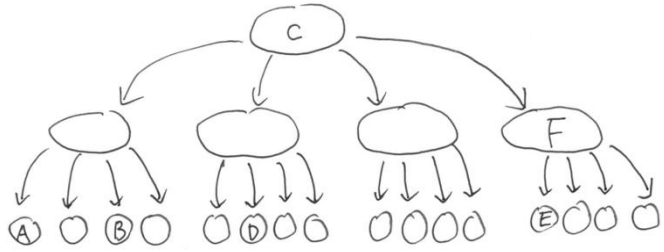


그림 2.

각 노드는 다음과 같은 구조를 가진다.

```
struct Node {
        Node *child[4];
        vector<Object *> obj;
};
```

최상위 노드는 전체 지역을 담당한다. 각 노드는 4개의 child노드를 가진다. (단, 최하위 노드는 child 노드를 가질 수 없다). 그리고, 그 child노드는 부모 노드의 1/4 지역을 담당한다. 만약, 오브젝트가 child 노드 간의 경계에 놓여 있으면 부모 노드에 포인터를 저장한다.

그런데, 이런 quadtree는 몇 가지 단점이 있다. 위 그림에서 오브젝트 D는 최하위 노드에 있는데, 최상위 노드에서 오브젝트 D를 검색하려면 여러 단계를 거쳐야 한다.
또, 오브젝트가 C처럼 지역간 경계에 있으면, quadtree의 성능이 떨어진다.

그래서, 단점을 보완하는 quad tree 구현 방법을 고민해 보았다.

new implementation

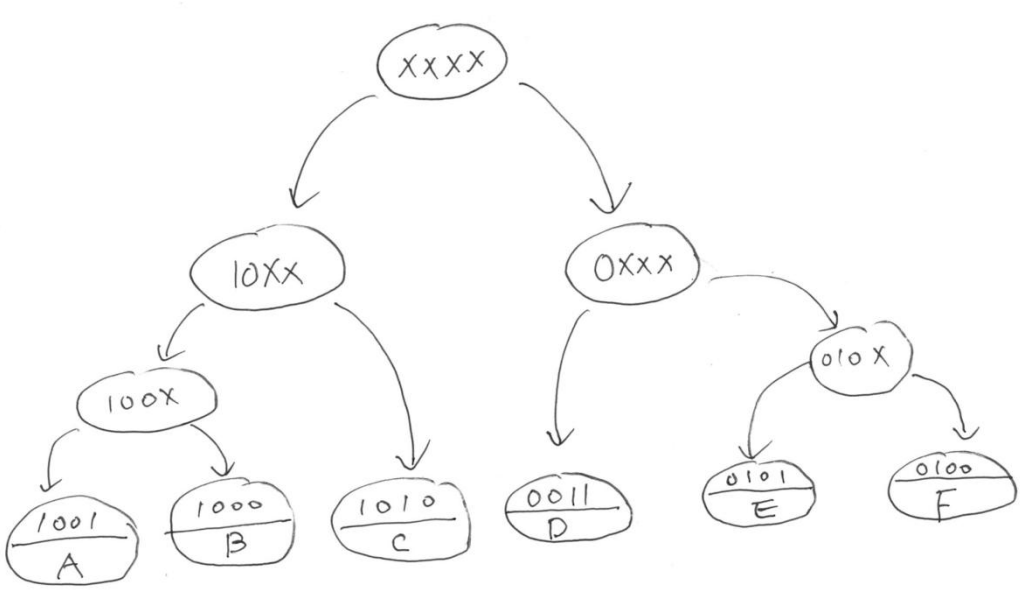


그림 3.

위 그림은 내가 구현한 방법의 자료 구조를 그림으로 표현한 것이다.
일종의 balanced binary tree형태로 구현하였다.

각 오브젝트는 최하위 노드의 역할도 동시에 한다.
최하위 노드를 제외한 중간 노드는 항상 2개의 자식 노드를 가진다.
우선 최하위 노드에 주소를 배정하는 방법부터 설명하겠다.
전체 지역을 x축으로 4부분으로 나누고, 각 지역의 좌표를 왼쪽부터 -2,-1,0,1로 정한다.

이 좌표 값을 이진수로 바꾼 후, 아래 2bit만 나타내면 10,11,00,01이 된다.
아래 그림은 각 지역의 좌표를 (x축 좌표),(y축 좌표) 형식으로 나타낸 것이다.

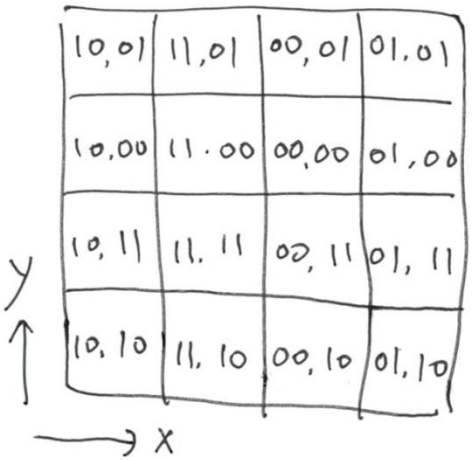


그림 4.

그 다음 x축과 y축의 좌표 값을 1bit씩 교대로 섞어서 노드의 주소값으로 정한다.

그림 5.

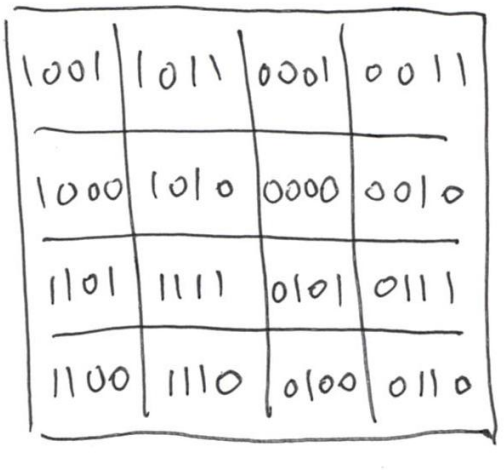


그림 6.

그림 6에서 Y축 방향으로 노드 2개씩 합쳐서 하나로 만들면, 그림 7처럼 된다.
이때 주소 끝부분의 X는 0또는 1어떤 값이어도 상관없다는 의미이다.

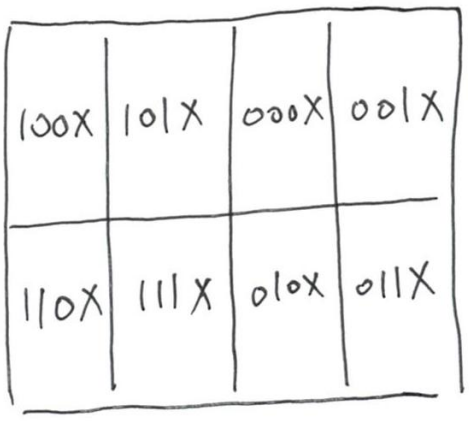


그림 7.

이번에는 X축방향으로 노드 2개를 합치면 그림 8과 같이 된다.

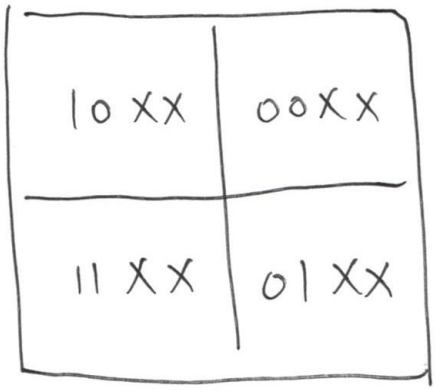


그림 8.

계속해서 2개씩 합치면, 그림 10처럼 전체 지역을 담당하는 노드의 주소는 XXXX가 된다.

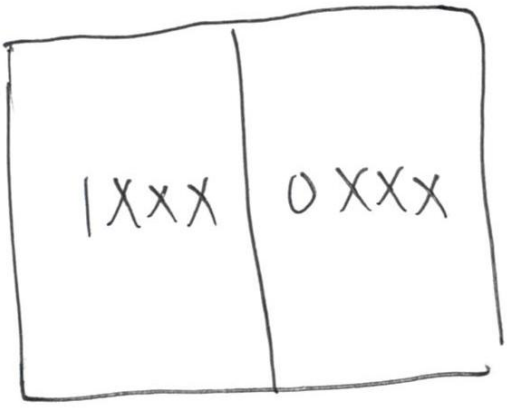


그림 9.

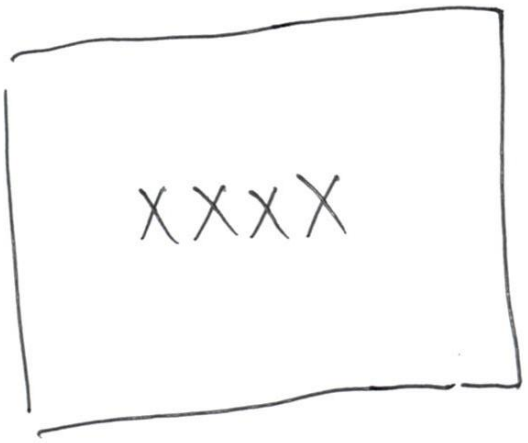
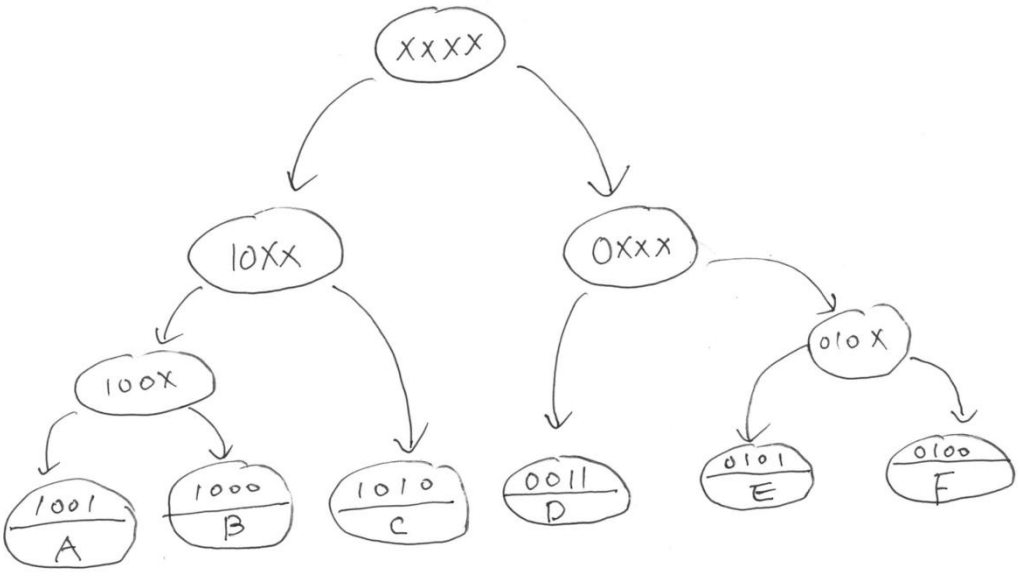


그림 10.



이제 그림 3과 비교해 보면, 대충 어떻게 구현했는지 알 수 있을 것이다.
기존 방법과 다른 점은, 노드 D의 경우에 노드 001X, 노드 00XX를 거치지 않고, 노드 0XXX의 자식 노드로 직접 연결되어 있다는 것이다.
그리고, 노드 C는 두 지역에 걸쳐 있는데, 중심 좌표가 있는 노드 1010으로 들어 가 있다는 것이다.
새로운 구현방법에서 노드 주소를 정할 때, 오브젝트의 크기는 중요하지 않고, 중심 위치만 보고 정한다. 오브젝트의 크기는 노드의 AABB에 별도로 저장된다.
그래서, 오브젝트 C는 두 지역에 걸쳐있지만, 노드 주소가 1010이 된다.

이제 노드의 자료 구조에 대해 설명하겠다.

```
class _Node
{
public:

        VECTOR  m_vCenter;
        VECTOR  m_vExtent;
        Node    *m_parent;
        Node    *m_child[2];
        DWORD   m_mask;
        DWORD   m_bit;
};
```

각 노드는 m_mask 와 m_bit 로 표현한다.
예를 들어 주소가 이진수 101X라면, m_mask 는 2진수 1110, m_bit 는 2진수 1010의 값을 가진다.
m_parent는 부모 노드의 포인터이고, m_child는 자식 노드의 주소이다.

자료 구조 자체가 Axis Aligned Bounding Box(AABB)를 포함하고 있어서, 그 노드에 속해 있는 하위 오브젝트를 포함하는 직육면체의 데이터를 저장한다. m_vCenter는 그 직육면체의 중심 위치이고, m_vExtent는 직육면체 크기의 반이다. 이렇게 AABB 데이터를 별도로 저장함으로써, 오브젝트가 두 지역에 걸쳐있을 때 처리가 간단해졌다.

실제 구현된 코드를 살펴 보자. 여러 부분에서 사용하기 쉽게 하기 위해 template으로 구현하였다.
첨부된 Quadtree.h와 quadtree.inl 파일에서 QuadTreeA를 참고하기 바란다.

https://sites.google.com/site/doc4code/source/quadtree.h
https://sites.google.com/site/doc4code/source/quadtree.inl

이 알고리즘의 또 한가지 장점은 QuadTreeA::MoveObject() 의 처리시간이 빠르다는 것이다.
n 개의 오브젝트가 있을 때, 이 알고리즘의 처리시간은 $O(1)$에 가깝다. 루트 (root)노드부터 검색하지 않고, 이동된 오브젝트의 노드와 그 부모 노드 몇 개만 처리하면 되기 때문이다.
일반적인 QuadTree 알고리즘에서의 처리 시간이 $O(\log n)$인데 비하면, 이 알고리즘이 더 빠르다.
이것은 물리엔진에서 많이 사용되는 Sweep and Prune 알고리즘보다 더 좋은 성능이므로, Sweep and Prune 알고리즘 대신에 사용할 수 있다.

Seamless Loading

심리스 로딩 방식에서는 캐릭터가 이동할 때 마다 행동을 예측해 미리 로딩 하는 방식으로 맵의 이동시 로딩 없이 이어지는 방식을 말한다. 엄격하게 말하면, 로딩이 없는 것이 아니라, 로딩으로 인한 중단 시간이 짧아서, 로딩이 없는 것처럼 느껴지게 하는 것이다.

여기서 사용하는 방법은 프로그램 시작 시 오브젝트의 크기와 관련된 데이터만 미리 읽어 둔다. 그러다가 주인공 캐릭터가 움직일 때마다, 그 위치에서 가까운 오브젝트를 조금씩 로딩하는 것이다.
이 때, 얼마나 가까워졌을 때 로딩해야 되는지는 그 오브젝트의 크기와 관련이 있다. 그래서, 큰 물체일수록 멀리서부터 로딩될 수 있도록 해야 한다.
그리고, 그 오브젝트에서 멀리 떨어지면 그 오브젝트를 언로딩(Unloading)하게 하면 된다.

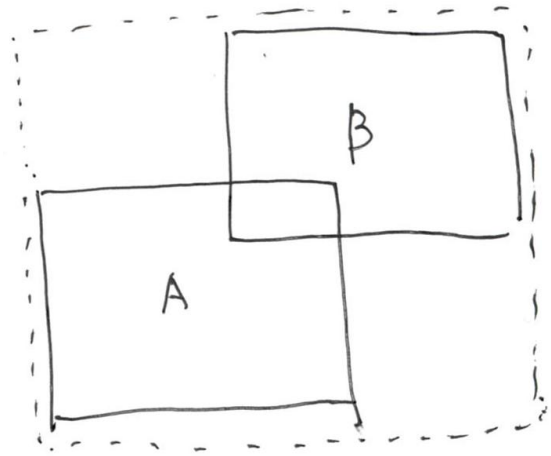


그림 11.

QuadTreeA에서 _node 구조체 자체가 AABB를 가지고 있기 때문에 로딩하는 것을 쉽게 구현할 수 있다. 캐릭터가 사각형 A안으로 들어가면, 오브젝트 A를 로딩하면 된다. 이 때, 노드 A와 노드 B를 자식으로 가지는 부모 노드의 AABB는 점선으로 된 사각형이다. 그래서, 부모 노드의 AABB를 검사해서 검사에 성공했을 때에만, 자식 노드를 검사하게 하면 된다.

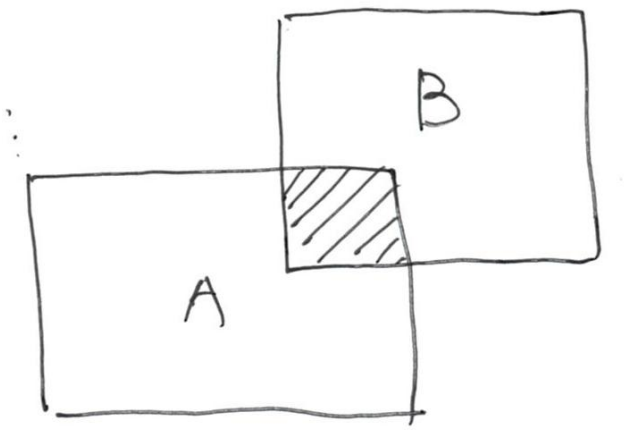


그림 12.

심리스 언로딩(seamless unloading)을 효율적으로 검사하기 위해, 새로운 타입의 Quadtree를 구현하였다.

QuadTreeB에서는 부모 노드의 AABB가 자식 노드의 교집합으로 되어 있다. 노드 A와 노드 B를 자식으로 가지는 부모 노드의 AABB는 빗금 친 사각형이다. 그래서, 캐릭터가 부모 노드의 AABB(빗금 친 영역) 바깥으로 나갔을 때만, 자식 노드 A와 B를 검사해서, 언로드 할 지 결정한다.