Object and Scene Management

Table of Contents

[2 Scene Overview 1](#_Toc382851604)

[3 Architecture 1](#_Toc382851605)

[3.1 Game Object requirements 1](#_Toc382851606)

[3.2 Scene and SceneNode 2](#_Toc382851607)

# Scene Overview

The various game objects in the world need to be stored somewhere. We also need to be able to query them in interesting ways (eg. visible objects, overlapping objects, shadow casting lights, etc…). To solve this, we make use of a Scene object which has the following goals:

* Container for all the game objects in the world
* Supports various queries required by the game

The queries we design will be tailored exactly to the needs of the game, nothing more. Here are the known queries we have (we’ll add to this list as we encounter new requirements):

* GetVisibleObjectsWithAllFlags(Camera, Flags, out Objects)
  + Used to get visible objects to render
  + Used to get visible lights
* GetOverlappingPairsWithAnyFlag(Flags, out Pairs)
  + Can be used to get preliminary list for collision detection
* GetAllObjects(out Objects)

The methods take in a flags argument, which is used to filter which types of objects are returned. All objects exist in the same Scene, so a filter is provided to select a set of objects.

For instance, there are flags for “Visible”, “Light”, “Renderable”, “Sensor”, “Solid”, “Casts Shadows”, etc…

We may want to experiment and see if it’s better to get the list of all visible lights back first, then prune it based on which objects are being rendered, or if it’s better to make a query with each object and get the set of affecting lights. Both are easy to do when the fundamentals are in place.

# Architecture

## Game Object requirements

The Game Object type will need to support a few things to work in this scene architecture. These are summarized with the pseudo code below:

enum class GameObjectFlag

{

None = 0,

Visible = 0x0001,

Renderable = 0x0002,

Light = 0x0004,

Solid = 0x0008,

Sensor = 0x0010,

// etc...

Max

};

class GameObject

{

// ...

private:

// ...

GameObjectFlag Flags;

AABB Bounds;

Scene\* Scene;

SceneNode\* Node;

};

With these pieces in place, we can build a Scene object that properly stores and manages the objects and supports the various queries we need.

## Scene and SceneNode

The Scene is implemented as a dynamic AABB tree. This gives the most flexibility while still being relatively simple and efficient. Initially the tree will not rebalance, until proven by a profiler that it’s necessary.

The architecture is as follows:

class Scene

{

// ...

private:

// ...

SceneNode\* Root;

};

struct SceneNode

{

SceneNode\* Parent;

union

{

SceneNode\* Children[2];

GameObject\* Object;

};

AABB Bounds;

bool Dirty;

};

Each node points up to it’s parent, and each node has either 2 children (inner node), or a game object (leaf node). If a node drops to a single child, the inner node is deleted and replaced by the leaf. The Dirty bit is set (and immediately propagated up the tree) as soon as any bounds-affecting attribute on the game object changes. Instead of recomputing the AABBs every time a position or orientation, etc… changes, we just track dirtiness of portions of the tree, and recomputed on demand the next time any of those nodes are touched. This defers the work quite a bit and allows “free” coalescing of multiple state changes without having to waste effort.

All the queries start at the top of the tree and recurse until all leaves satisfying the query are visited.