# Program a Game Engine from Scratch

Mark Claypool

## Development Checkpoint #14

## Scene Graph

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 9.0). Information online at: `http://dragonfly.wpi.edu/book/`

## 4.17 End Game (optional)

Up to this point, Dragonfly is a fairly full-featured, completely functional game engine. A few potential enhancements remain, however, that bring in elements common to many game engines and improve performance, appearance and functionality.

### 4.17.1 Scene Graphs (optional)

Scene graphs are data structures that arrange elements of a graphics scene in order to provide more efficient rendering. For example, when drawing objects in a 3d scene, a scene graph might arrange the objects based on distance from the camera. Rendering the frame then draws the objects that are farthest away from the camera first, proceeding to the objects that are closest to the camera since the closer objects may occlude those behind.

Consider Dragonfly, where Objects have altitude (see Section 4.8.5 on page 121). Objects that are at lower altitude are drawn first before Objects at higher altitude, allowing the higher altitude to be layered "on top" of the lower ones, as necessary. Without a scene graph, Dragonfly implements altitude by iterating through all the Objects for each altitude implementation, as in Listing 4.84 on page 122 – effectively, doing $n\times$ `MAX_ALTITUDE` comparisons, where $n$ is the number of Objects in the game world. With a scene graph, the objects can be arranged by altitude, making the WorldManager `draw()` method only go through the list of Objects once, so only doing $n$ comparisons.

For a game engine, a scene graph often arranges objects for more efficient queries, also. Objects that are not solid do not cause collisions. Without any other organization, detecting whether a moving object collides with any other object must look through all objects, regardless of whether they are solid or not. Thus far, Dragonfly is implemented this way, too, as in Listing 4.106 (page 145), iterating through all Objects, checking for collision with every Object, even the non-solid ones. Other common organizations group Objects by location in the game world, allowing selection and iteration over only those Objects at or near a specific location.

Since a scene graph organizes Objects, whether for drawing or query efficiency, it is naturally part of the WorldManager. In fact, an easy way of viewing a scene graph is that it replaces a simple list of game Objects with a more complex data structure where the Objects are organized and indexed in different ways. In Dragonfly, this means replacing `ObjectList m_updates` on line 10 of Listing 4.55 (page 99) with `SceneGraph scene_graph`.

The header file for `SceneGraph` is shown in Listing 4.206. SceneGraph needs to `#include` both `Object.h` and `ObjectList.h`. The definition of `MAX_ALTITUDE` on line 3 has been moved from `WorldManager.h` to `SceneGraph.h`.

To support efficient queries by the WorldManager (e.g., to provide a list of all the solid objects), starting at line 8, the SceneGraph defines three lists of Objects. The first, `objects`, is a list of all the Objects in the game – formerly, this was `m_updates` in the WorldManager. The second, `solid_objects`, is a list of just the solid Objects in the game. The third, `visible_objects`, is an array of ObjectLists, with each element being a list of Objects at that altitude. Methods to add and remove objects to the scene graph are provided by `insertObject()` and `removeObject()`, respectively.

To support queries that may be made by the WorldManager (or even the game program-

mer), SceneGraph includes methods: `activeObjects()`, which returns all active Objects; `inactiveObjects()`, which returns all inactive Objects; `solidObjects()`, which returns all solid Objects; and `visibleObjects()`, which returns all visible Objects at a given altitude. The methods all return an empty ObjectList if there are no Objects matching the query. The method `updateAltitude()` is invoked when an Object re-positions itself to a new altitude and the method `updateSolidness()` is invoked when an Object updates its solidness.

Listing 4.206: SceneGraph.h

```
#include "Object.h"
#include "ObjectList.h"

const int MAX_ALTITUDE = 4;

class SceneGraph {

 private:
  ObjectList m_objects;                          // All Objects
  ObjectList m_solid_objects;                    // Solid objects.
  ObjectList m_visible_objects[MAX_ALTITUDE+1];  // Visible objects.

 public:
  SceneGraph();
  ~SceneGraph();

  // Insert Object into SceneGraph.
  int insertObject(Object *p_o);

  // Remove Object from SceneGraph.
  int removeObject(Object *p_o);

  // Return all active Objects. Empty list if none.
  ObjectList activeObjects() const;

  // Return all active, solid Objects. Empty list if none.
  ObjectList solidObjects() const;

  // Return all active, visible Objects at altitude. Empty list if none.
  ObjectList visibleObjects(int altitude) const;

  // Return all inactive Objects. Empty list if none.
  ObjectList inactiveObjects() const;

  // Re-position Object in SceneGraph to new altitude.
  // Return 0 if ok, else -1.
  int updateAltitude(Object *p_o, int new_alt);

  // Re-position Object in SceneGraph to new solidness.
  // Return 0 if ok, else -1.
  int updateSolidness(Object *p_o, Solidness new_solidness);

  // Re-position Object in SceneGraph to new visibility.
  // Return 0 if ok, else -1.
  int updateVisible(Object *p_vo, bool new_visible);
```

```
45
46    // Re-position Object in SceneGraph to new activeness.
47    // Return 0 if ok, else -1.
48    int updateActive(Object *p_o, bool new_active);
49  };
```

Implementation of SceneGraph `insertObject()` is shown in Listing 4.207. The method first inserts the Object into the `objects` list, since that is the "master" list that contains all Objects. Then, if the Object is solid, it is added to the `solid_objects` list. Next, the Object's altitude is checked – if it is not in range (calling `valueInRange(altitude, 0, MAX_ALTITUDE)`, see line 1 in Listing 4.152 on page 182), it returns an error (-1). Otherwise, the object is inserted into the `visible_objects` list at the correct altitude. Note, the calls to `ObjectList::insert()` need to be error checked. If they encounter an error, an appropriate message should be written to the logfile and `insertObject()` should return -1.

While it may seem that keeping 3 object lists is inefficient, remember that game objects are stored as pointers to Objects, thus manipulating and copying such lists is not actually doing the much more expensive operation of copying the memory space for each Object. As a refresher, see Section 4.5.2 on page 4.5.2 for details on the ObjectList implementation.

Listing 4.207: SceneGraph insertObject()

```
0  // Insert Object into SceneGraph.
1  int SceneGraph::insertObject(Object *p_o)
2
3    // Add object to list.
4    insert p_o into objects list
5
6    // If solid, add to solid objects list.
7    if p_o -> isSolid() then
8      insert p_o into solid_objects list
9    end if
10
11   // Check altitude.
12   if not valueInRange(p_o->getAltitude(), 0, MAX_ALTITUDE) then
13     return error
14   end if
15
16   // Add to visible objects at right altitude.
17   insert p_o into visible_objects[p_o->getAltitude()] list
```

Implementation of the SceneGraph `removeObject()` is basically the "undo" of the `insertObject()` method, as shown in Listing 4.208. The indicated Object (`p_o`) is removed from the `objects`, `solid_objects` and `visible_objects` lists. As always, the calls to `ObjectList::remove()` need to be error checked, writing an appropriate message to the logfile and returning -1 on encountering an error.

Listing 4.208: SceneGraph removeObject()

```
0  // Remove Object from SceneGraph.
1  int SceneGraph::removeObject(Object *p_o)
2
3    remove p_o from objects list
4
```

```
5   if p_o is solid then
6     remove p_o from solid_objects list
7   end if
8
9   remove p_o from visible_objects[p_o->getAltitude()] list
10
11  if no errors then // means no errors in any of the above
12    return ok
13  else
14    return error
15  end if
```

The methods `allObjects()` and `solidObjects()` just return `objects` and `solid_objects`, respectively. `visibleObjects()` first checks that the parameter `altitude` is in range (calling `valueInRange(altitude, 0, MAX_ALTITUDE)`, see line 1 in Listing 4.152 on page 182), then returns `visible_objects[altitude]`.

Objects may change their attributes, such as a `SPECTRAL` Object becoming `SOFT` or an Object changing altitude from 3 to 4. All such changes need to modify the contents of the SceneGraph lists, `solid_objects` and `visible_objects[]`, respectively. Listing 4.209 shows the implementation for updating the solidness of an Object. The first block of code checks if the Object is solid and, if so, removes it from the `solid_objects` list. The second block of code checks if the new solidness value for the Object is solid ( `HARD` or `SOFT`) and, if so, inserts it into the `solid_objects` list. Error checking on the `ObjectList::insert()` calls is needed, as usual. Note, the solidness attribute for the Object is *not* changed – that is a private value for Object and is done in the Object `setSolidness()` method.

Listing 4.209: SceneGraph updateSolidness()

```
0 // Re-position Object in SceneGraph to new solidness.
1 // Return 0 if ok, else -1.
2 int SceneGraph::updateSolidness(Object *p_o, Solidness new_solidness)
3
4   // If was solid, remove from solid objects list.
5   if p_o->isSolid() then
6     remove p_o from solid_objects list
7   end if
8
9   // If is solid, add to list.
10  if new_solidness is HARD or new_solidness is SOFT then
11    insert p_o into solid_objects list
12  end if
13
14  // All is well.
15  return ok
```

Listing 4.210 shows the implementation for updating the altitude of an Object. First, the altitude values for both the new and old altitudes are checked for validity. It may seem odd to check the old value, since it seems it must be right, but it could have been corrupted someplace – if it was, trying to remove the Object from the `visible_objects[]` list at the altitude may result in a crash. If both old and new are in the valid range, the Object is first removed from `visible_objects[]` at the old altitude, then added to `visible_objects[]` at the new altitude. Error checking on the `ObjectList::insert()` calls are needed, as

usual.

Listing 4.210: SceneGraph updateAltitude()

```
// Re-position Object in scene graph to new altitude.
// Return 0 if ok, else -1.
int SceneGraph::updateAltitude(Object *p_o, int new_alt)

  // Check if new altitude in valid range.
  if not valueInRange(new_alt, 0, MAX_ALTITUDE) then
    return error
  end if

  // Make sure old altitude in valid range.
  if not valueInRange(p_o->getAltitude(), 0, MAX_ALTITUDE)) then
    return error
  end if

  // Remove from old altitude.
  remove p_o from visible_objects[p_o->getAltitude()]

  // Add to new altitude.
  insert p_o into visible_objects[new_alt]

  // All is well.
  return ok
```

Calls to `updateSolidness()` and `UpdateAltitude()` are made from Object, specifically Object `setSolidness()` and Object `setAltitude()`, respectively. The needed extension to Object `setSolidness()` to support SceneGraph is shown in Listing 4.211. The first block of code checks if the new solidness is valid (`HARD`, `SOFT` or `SPECTRAL`). If not, an error is returned. Otherwise, the `updateSolidness()` method of the SceneGraph is called and `solidness` is set in the Object.

Listing 4.211: Object class extension to setSolidness() to support SceneGraph

```
// Set object solidness, with checks for consistency.
// Return 0 if ok, else -1.
int Object::setSolidness(Solidness new_solidness)

  // If solidness not valid, then ignore.
  if new_solidness not (HARD or SOFT or SPECTRAL) then
    return error
  end if

  // Update scene graph and solidness.
  scene_graph.updateSolidness(this, new_solidness)
  solidness = new_solidness

  // All is well.
  return ok
```

Extension to Object `setAltitude()` to support SceneGraphs is shown in Listing 4.212. The first block of code checks if the new altitude is in a valid range. If not, an error is

returned. Otherwise, the SceneGraph `updateAltitude()` method is called and `altitude` is set in the Object.

Listing 4.212: Object class extension to setAltitude() support SceneGraphs

```
// Set Object altitude.
// Checks for in range [0, MAX_ALTITUDE].
// Return 0 if ok, else -1.
int Object::setAltitude(int new_altitude)

  // If altitude outside of range, then ignore.
  if not valueInRange(new_altitude, 0, MAX_ALTITUDE) then
    return error
  end if

  // Update scene graph and altitude.
  scene_graph.updateAltitude(this, new_altitude)
  altitude = new_altitude

  // All is well.
  return ok
```

With the SceneGraph in place, the Dragonfly WorldManager needs to be refactored to use the SceneGraph to manage game world Objects instead of storing the ObjectLists directly. Listing 4.213 shows the change in the WorldManager needed to use a SceneGraph. Basically, the attribute `ObjectList m_updates` is replaced with `SceneGraph scene_graph`. The method `getSceneGraph()` returns a reference to `scene_graph`.

Listing 4.213: WorldManager extensions to support SceneGraph

```
 private:
  SceneGraph scene_graph; // Storage for all Objects.

 public:
  // Return reference to the SceneGraph.
  SceneGraph &getSceneGraph() const;
```

Then, internally, each of the WorldManager methods in Listing 4.214 needs to be refactored to support the SceneGraph. The methods `insertObject()` and `removeObject()` call and return `scene_graph.insertObject()` and `scene_graph.removeObject()`, respectively. The methods `update()` and `setViewFollowing()` call `scene_graph.allObjects()` to iterate through all the world Objects. The method `draw()` iterates through each altitude, calling `scene_graph.visibleObjects()` for each altitude. The method `getCollisions()` checks for collisions only with Objects in the ObjectList returned from `scene_graph.solidObjects()`.

Listing 4.214: WorldManager methods to refactor to support SceneGraph

```
ObjectList getAllObjects()
int insertObject(Object *p_o)
int removeObject(Object *p_o)
int setViewFollowing(Object *p_new_view_following)
void update()
void draw()
ObjectList getCollisions(const Object *p_o, Vector where) const
```

#### 4.17.1.1 Inactive Objects (optional)

For many games, it is useful for the game program to have some game objects be ignored by the engine for some time, but without removing the objects from the game world altogether. For example, the Saucer Shoot tutorial game (Section 3.3.11 on page 39) has the main menu become inactive when the game is being played, becoming active again after the player's ship has been destroyed. Such inactive objects are not drawn by the engine, are neither moved nor considered in collisions, nor do they receive any events.

In order to support inactive Objects in Dragonfly, the Object class is extended with an attribute and methods to support whether an Object is active or inactive, shown in Listing 4.215. The boolean attribute `is_active` is `true` when the Object is active (note, all the Objects that have been dealt with to this point are active) and `false` when the Object is inactive and not acted upon by the engine. This value can be set via the `setActive()` method and queried via the `isActive()` method.

Listing 4.215: Object class extensions to support inactive objects

```
private:
 bool is_active;     // If false, Object not acted upon.

public:
 // Set activeness of Object.  Objects not active are not acted upon
 // by engine.
 // Return 0 if ok, else -1.
 int setActive(bool active=true);

 // Return activeness of Object.  Objects not active are not acted upon
 // by engine.
 bool isActive() const;
```

As shown in Listing 4.216, the method `setActive()` allows the game programmer to set the Object activeness, changing `is_active` as appropriate. Objects are active (`is_active` is `true`) by default, set in the constructor. In addition, the SceneGraph is obtained from the WorldManager and the SceneGraph `updateActive()` method is called.

Listing 4.216: Object setActive()

```
// Set activeness of Object.  Objects not active are not acted upon
// by engine.
// Return 0 if ok, else -1.
int Object::setActive(bool active)

  // Update scene graph.
  scene_graph = WorldManager getSceneGraph()
  scene_graph.updateActive(this, active)

  // Set active value.
  is_active = active
```

The SceneGraph is refactored to have an additional ObjectList, one that holds only inactive Objects while the main object list will hold active Objects. Listing 4.217 shows the changes to the SceneGraph attributes for this. The `objects` ObjectList has been renamed

to active_objects to differentiate it from the ObjectList holding the inactive objects, inactive_objects.

Listing 4.217: SceneGraph extensions to support inactive Objects

```
private:
  ObjectList active_objects;      // All active Objects.
  ObjectList inactive_objects;    // All inactive Objects.

public:
  // Return all active Objects. Empty list if none.
  ObjectList activeObjects() const;

  // Return all inactive Objects. Empty list if none.
  ObjectList inactiveObjects() const;

  // Re-position Object in SceneGraph to new activeness.
  // Return 0 if ok, else -1.
  int updateActive(Object *p_o, bool new_active);
```

The methods activeObjects() and inactiveObjects() return active_objects and inactive_objects, respectively.

Listing 4.218 shows the SceneGraph updateActive() method. The first block of code checks if the activeness is being changed. If not, there is nothing more to do and an "ok" (0) is returned. The second block of code does the actual work. If the Object was active and became inactive, remove() is called on active_objects, visible_objects[] and, if solid, solid_objects and the Object is inserted into the inactive_objects list. Otherwise, if the Object was inactive and became inactive, insert() is called on active_objects, visible_objects[] and, if solid, solid_objects and the Object is removed from the inactive_objects list. All method calls should be error checked and an error (-1) returned, as appropriate. Otherwise, "ok" is returned at the end.

Listing 4.218: SceneGraph updateActive()

```
// Re-position Object in SceneGraph to new activeness.
// Return 0 if ok, else -1.
int SceneGraph::updateActive(Object *p_o, bool new_active)

  // If activeness unchanged, nothing to do (but ok).
  if p_o->isActive() is new_active then
    return ok
  end if

  // If was active then now inactive, so remove from lists.
  if p_o->isActive() then

    active_objects.remove(p_o)

    visible_objects[p_o->getAltitude()].remove(p_o)

    if p_o->isSolid() then
      solid_objects.remove(p_o)
    end if

```

```
20      // Add to inactive list
21      inactive_objects.insert(p_o)
22
23    else // Was active, so add to lists.
24
25      active_objects.insert(p_o)
26
27      visible_objects[p_o->getAltitude()].insert(p_o)
28
29      if p_o->isSolid() then
30        solid_objects.insert(p_o)
31      end if
32
33      // Remove from inactive list
34      inactive_objects.remove(p_o)
35
36    end if
37
38    // All is well.
39    return ok
```

The WorldManager `getAllObjects()` method is refactored, as in Listing 4.219. A boolean parameter `inactive` is provided to indicate whether the method should return only active Objects (`inactive` is `false`, the default) or both active and inactive Objects (`inactive` is `true`).

Listing 4.219: WorldManager extensions to support inactive Objects

```
0   // Return list of all Objects in world.
1   // If inactive is true, include inactive Objects.
2   // Return NULL if list is empty.
3   ObjectList getAllObjects(bool inactive=false);
```

The revised `getAllObjects()` is shown in Listing 4.220. The inactive case can use the overloaded '+' operator from Section 4.5.2.2 (page 85).

Listing 4.220: WorldManager extensions to getAllObjects() to support inactive Objects

```
0 // Return list of all Objects in world.
1 // If inactive is true, include inactive Objects.
2 // Return NULL if list is empty.
3 ObjectList WorldManager::getAllObjects(bool inactive) const
4
5   if inactive then
6     return scene_graph.activeObjects() + scene_graph.inactiveObjects()
7   else
8     return scene_graph.activeObjects()
9   end if
```

The Manager `onEvent()` method needs to be modified to check if an interested Object is actually active before sending it an event. This is shown on line 3 of Listing 4.221.

Listing 4.221: Manager extension to onEvent() to support inactive Objects

```
0   ...
1   create ObjectListIterator li on obj_list[i]
```

```
  while not li.isDone() do
    if li.currentObject -> isActive() then
      call i.currentObject() -> eventHandler() with p_event
    end if
    li.next()
  end while
  ...
```

Lastly, WorldManager `shutDown()` should be revised to call `getAllObjects(true)` to delete both active and inactive Objects when the engine is shut down.

#### 4.17.1.2 Invisible Objects (optional)

Another useful property for many game objects is to become invisible. For a game object, invisibility could be a special power, say, for the hero or a bad guy to vanish from sight – but as such, it is rather rare. However, invisibility is commonly used to limit the player's ability to see objects that may be on the window, but should not yet be shown to the player because of the player's avatar's orientation, or because of terrain or other "fog of war" type of effect. From a game engine perspective, an invisible game object is not drawn, but is still updated each game loop and can be collided with, if solid, as appropriate.

To support invisibility, a new attribute is added to Object with methods for getting and setting it, shown in Listing 4.222. The method `isVisible()` returns the value of `is_visible`.

Listing 4.222: Object class extensions to support invisibility

```
private:
 bool is_visible;      // If true, object gets drawn.

public:
 // Set visibility of Object. Objects not visible are not drawn.
 // Return 0 if ok, else -1.
 int setVisible(bool visible=true);

 // Return visibility of Object. Objects not visible are not drawn.
 bool isVisible() const;
```

As shown in Listing 4.223, the method `setVisible()` allows the game programmer to set the Object visibility, changing `is_visible` as appropriate. Objects are visible (`is_visible` is `true`) by default. In addition, the SceneGraph is obtained from the WorldManager and the SceneGraph `updateVisible()` method is called.

Listing 4.223: Object setVisible()

```
// Set visibility of Object. Objects not visible are not drawn.
// Return 0 if ok, else -1.
int Object::setVisible(bool visible)

  // Update scene graph.
  scene_graph = WorldManager getSceneGraph()
  scene_graph.updateVisible(this, visible)

  // Set visibility value.
```

```
9   is_visible = visible
```

Listing 4.224 shows the SceneGraph `updateVisible()` method. The first block of code checks if the visibility is being changed. If not, there is nothing more to do and an "ok" (0) is returned. The second block of code does the actual work. If the Object was visible and went invisible, `remove()` is called on the ObjectList, otherwise `insert()` is called, at the right altitude (`p_o->getAltitude()`). All method calls should be error checked and an error (-1) returned, as appropriate. Otherwise, "ok" is returned at the end.

Listing 4.224: SceneGraph updateVisible()

```
0  // Re-position Object in scene graph based on visibility.
1  // Return 0 if ok, else -1.
2  int SceneGraph::updateVisible(Object *p_o, bool new_visible)
3
4    // If visibility unchanged, nothing to do (but ok).
5    if p_o->isVisible() is new_visible then
6      return ok
7    end if
8
9    // If was visible then now invisible, so remove from list.
10   if p_o->isVisible() then
11     visible_objects[p_o->getAltitude()].remove(p_o)
12   else // Was invisible, so add to list.
13     visible_objects[p_o->getAltitude()].insert(p_o)
14   end if
15
16   // All is well.
17   return ok
```

### 4.17.2 Development Checkpoint #14!

To develop the SceneGraph for Dragonfly, use the following steps:

1. Create the SceneGraph class, referring to Listing 4.206 as needed. Add `SceneGraph.cpp` to the project and stub out each method so the SceneGraph compiles.

2. Implement the SceneGraph `insertObject()` and `removeObject()` methods based on Listing 4.207 and Listing 4.208, respectively. Test outside of the game engine by adding and removing Objects.

3. Implement the SceneGraph `updateSolidness()` method, based on Listing 4.209 and `updateAltitude()`, based on Listing 4.210.

4. Extend the Object class `setSolidness()` to support a SceneGraph, referring to Listing 4.211. Do the same for `setAltitude()`, referring to Listing 4.212.

5. Extend the WorldManager to support a SceneGraph, as in Listing 4.213. Refactor the methods shown in Listing 4.214, as appropriate.

6. Test by verifying that previous code that worked without SceneGraphs *still* works, such as test code from the last development checkpoint (on page 226).

If support for inactive Objects is desired (optional), continue development:

1. Extend Object to support activeness based on Listing 4.215, implementing `setActive()` based on Listing 4.216.
2. Refactor the SceneGraph based on Listing 4.217, implementing `updateActive()` based on Listing 4.218.
3. Refactor the WorldManager based on Listing 4.219, extending `getAllObjects()()` to return inactive Objects, too, based on Listing 4.220.
4. Test with game code that sets another game object to inactive and back, say, based upon key presses.

If support for invisibility is desired (optional), continue development:

1. Extend Object to support invisibility based on Listing 4.222.
2. Implement Object `setVisible()` based on Listing 4.223 and SceneGraph `updateVisible()`, based on Listing 4.224.
3. Test with game code that has game objects set themselves to invisible and back, say, depending upon key presses or positions on the screen.