Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #10

Collisions & Views

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/`

4.13 Boxes

Boxes (also known as rectangles) are useful for providing a variety of 2d or 3d game features. Boxes can be used to determine the bounds of an object for collisions, as discussed in Section 4.10.1. Boxes can also be used to determine the boundary of the game world, helping detect when a game object goes out of bounds and/or off the boundary of the visual window. This latter feature is useful for when the game world is larger than what can be seen on the window, such as is typically the case for adventure-type games that feature exploring a large work, or for side-scrolling platformers. In this case, the player's view of the world is through a view window that moves with, say, the player's avatar. In order to support these features, Dragonfly has a Box class.

4.13.1 The Box Class

The definition for the Box class is provided in Listing 4.146. The Box uses a Vector attribute (`corner`) for the upper left corner, with horizontal and vertical attributes stored as integers. The default constructor creates an empty (zero width, zero height) box at (0,0). It is often useful to create a Box with given attributes, so the other constructor allows specification of position, horizontal and vertical attributes upon instantiation. Since the Box is just a container, the rest of the methods just get and set the attribute values.

Listing 4.146: Box.h

```
0  #include "Vector.h"
1
2  class Box {
3
4   private:
5    Vector m_corner;      // Upper left corner of box.
6    float m_horizontal;   // Horizontal dimension.
7    float m_vertical;     // Vertical dimension.
8
9   public:
10   // Create box with (0,0) for the corner, and 0 for horiz and vert.
11   Box();
12
13   // Create box with an upper-left corner, horiz and vert sizes.
14   Box(Vector init_corner, float init_horizontal, float init_vertical);
15
16   // Set upper left corner of box.
17   void setCorner(Vector new_corner);
18
19   // Get upper left corner of box.
20   Vector getCorner() const;
21
22   // Set horizontal size of box.
23   void setHorizontal(float new_horizontal);
24
25   // Get horizontal size of box.
26   float getHorizontal() const;
27
28   // Set vertical size of box.
```

```
29   void setVertical(float new_vertical);
30
31   // Get vertical size of box.
32   float getVertical() const;
33 };
```

4.13.2 Bounding Boxes

Boxes are used for the "size" of an Object, also known as a *bounding box* since it bounds the borders of a game object. The bounding box determines the region an Object occupies and is used in the computation to figure out if an Object collides with another Object.

Extensions to the Object class to support bounding boxes are shown in Listing 4.147. The bounding box is stored in the `private` attribute `m_box`, with methods provided to get and set it.

Listing 4.147: Object class extensions to support bounding boxes

```
0  private:
1   Box m_box;  // Box for sprite boundary & collisions.
2
3  public:
4   // Set Object's bounding box.
5   void setBox(Box new_box);
6
7   // Get Object's bounding box.
8   Box getBox() const;
```

The Object bounding box is initialized to a unit square (a Box with horizontal and vertical of 1), but typically the game programmer wants the bounding box to be the size of the Object as drawn. Thus, by default, the `setSprite()` method from Listing 4.143 (page 175) sets `m_box` to the width and height of the indicated sprite (as computed by the Animation object, `m_animation`). This can be done by adding the following line:

```
0   setBox(m_animation.getBox())
```

to the end of the method in Listing 4.145, right before the return. Animation should be extended with a `getBox()` method shown in Listing 4.148.

Listing 4.148: Animation class extensions to support bounding boxes

```
0  // Get bounding box of associated Sprite.
1  Box Animation::getBox() const
2
3    // If no Sprite, return unit Box centered at (0,0).
4    if not m_p_sprite then
5      Box box(Vector(-0.5,-0.5), 0.99, 0.99)
6      return box
7    end if
8
9    // Create Box around centered Sprite.
10   Vector corner(-1 * m_p_sprite->getWidth()/2.0,
11                 -1 * m_p_sprite->getHeight()/2.0)
12   Box box(corner, m_p_sprite->getWidth(),
```

```
                    m_p_sprite->getHeight())

  // Return box.
  return box
```

A major change needed to support bounding boxes regards collisions. Instead of Objects only colliding if their positions overlap, with boxes, Objects collide if their bounding boxes overlap. The idea is to replace the call to `positionsIntersect()` in the WorldManager `moveObject()` method (Listing 4.107 on page 146) with `boxIntersectsBox()`.

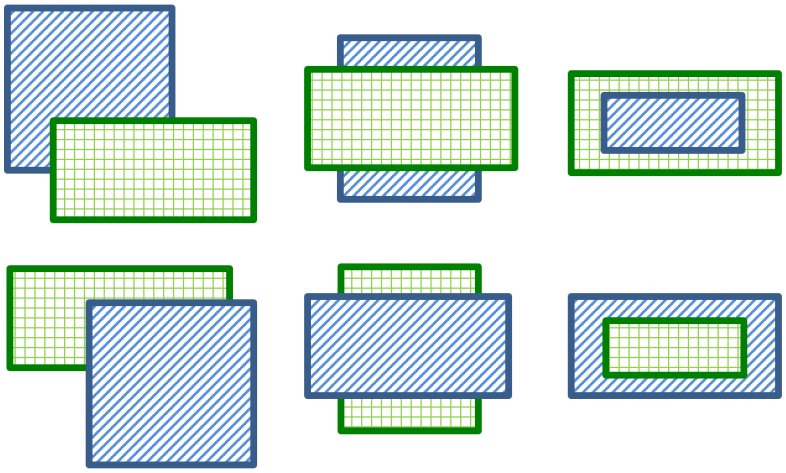

Figure 4.5: Positional possibilities for two overlapping boxes

There are several positional possibilities that must be considered when devising an algorithm to detect box overlap, as depicted by Figure 4.5. Any algorithm designed to detect overlap between two boxes in general should be carefully checked against these cases, both by hand and by coding up specific examples.

For the actual algorithm to test if two boxes overlap, while there are numerous possibilities, an intuitive and fairly fast method is as follows: consider Figure 4.6, where the upper left corner of a box is (`x1`, `y1`) and the bottom right corner is (`x2`, `y2`). An overlap of box A and box B only occurs if the left edge of B is contained within the width of A *and* the top edge of box B is contained within the height of box A. If both of those are true, then the two boxes overlap, otherwise they do not. And vice versa for A within B.

Using this idea, a new function `boxIntersectsBox()` is created in `utility.cpp`, with pseudo code shown in Listing 4.149.

Listing 4.149: boxIntersectsBox()

```
// Return true if boxes intersect, else false.
bool boxIntersectsBox(Box A, Box B)

  // Test horizontal overlap (x_overlap).
  Bx1 <= Ax1 && Ax1 <= Bx2     // Either left side of A in B?
  Ax1 <= Bx1 && Bx1 <= Ax2     // Or left side of B in A?

```

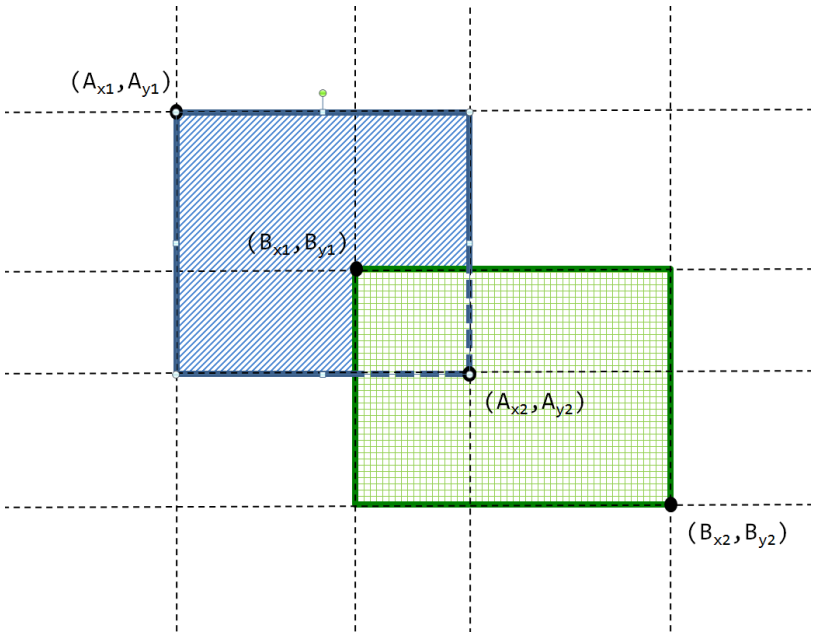

Figure 4.6: Corner notation used to determine if boxes overlap

```
  // Test vertical overlap (y_overlap).
  By1 <= Ay1 && Ay1 <= By2      // Either top side of A in B?
  Ay1 <= By1 && By1 <= Ay2      // Or top side of B in A?

  if (x_overlap) and (y_overlap) then
    return true          // Boxes do intersect.
  else
    return false         // Boxes do not intersect.
  end if
```

In replacing the call to `positionsIntersect()` in the WorldManager `getCollisions()` method (Listing 4.106 on page 145) with `boxIntersectsBox()`, it is important to remember that the bounding boxes for Objects are relative to the Objects themselves. For example, the top left corner of a bounding box for an Object with a 1-character Sprite is (0,0) and the top left corner of a bounding box for a 3x3 character Sprite (centered on the Object) is (-1.5,-1.5). Neither of these boxes are in terms of the game world coordinates.

In order to compute collisions, the bounding box position needs to be converted to a game world position. A useful utility (in `utility.cpp`) for this conversion is `getWorldBox()` that converts the bounding box positioned relative to an Object to a bounding box positioned relative to the game world.

Listing 4.150: getWorldBox()

```
// Convert relative bounding Box for Object to absolute world Box.
Box getWorldBox(const Object *p_o)

  Box box = p_o -> getBox()
  Vector corner = box.getCorner()
```

```
  corner.setX(corner.getX() + p_o -> getPosition().getX())
  corner.setY(corner.getY() + p_o -> getPosition().getY())
  box.setCorner(corner)

  return box
```

In addition, a similar version can be made that converts a relative bounding box for an Object to an absolute world Box at position `where`.

```
// Convert relative bounding Box for Object to absolute world Box.
Box getWorldBox(const Object *p_o, Vector where)
```

For ease of implementation, the first `getWorldBox()` can call the second, providing `p_o->getPosition()` as the argument to `where`.

Once created, collision detection in the WorldManager `getCollisions()` method is modified as in Listing 4.151.

Listing 4.151: WorldManager getCollisions() with bounding boxes

```
// Return list of Objects collided with at position 'where'.
// Collisions only with solid Objects.
// Does not consider if p_o is solid or not.
ObjectList getCollisions(const Object *p_o, Vector where) const
   ...

   // World position bounding box for object at where
   Box b = getWorldBox(p_o, where)

   // World position bounding box for other object
   Box b_temp = getWorldBox(p_temp_o)

   if boxIntersectsBox(b, b_temp) and p_temp_o->isSolid() then
      ...
```

Tip 19! Visually Debugging Bounding Boxes. While debugging bounding boxes can be done using `writeLog()` messages via the LogManager, sometimes it is easier to see the problems with a bounding box rather than figure it out through print messages. A fairly easy way to do this is to display part of the bounding box on the screen, above any sprite that is drawn. Specifically, after in Object `draw()`, after drawing a Sprite frame, place a symbol (e.g., a '+') for each corner of the bounding box, with an additional symbol for the Object position. Listing **??** provides code to do just this. The visual bounding box can be compiled in and out of the engine using conditional compilation (see Section 4.3.4 on page 58).

4.13.3 Utility Functions (optional)

The function `boxIntersectsBox()` is not only helpful for Dragonfly in detecting collisions, it can be a generally useful utility for a game programmer. The name suggests other utilities

shown in Listing 4.152 that are not necessarily used by the game engine but can be used by game programmers. Line 1 has a simple function that tests whether a value lies between the other two, useful in computing whether or not two Boxes intersect (see Listing 4.149). Line 4 has a function that converts the relative bounding box of an Object along with its position into a Box placed in the world (see Listing 4.150).

Listing 4.152: Utility functions

```
0  // Return true if value is between min and max (inclusive).
1  bool valueInRange(float value, float min, float max);
2
3  // Convert relative bounding Box for Object to absolute world Box.
4  Box getWorldBox(const Object *p_o);
5  Box getWorldBox(const Object *p_o, Vector where);
6
7  // Return true if Box contains Position.
8  bool boxContainsPosition(Box b, Vector p);
9
10 // Return true if Box 1 completely contains Box 2.
11 bool boxContainsBox(Box b1, Box b2);
12
13 // Return true if Line segments intersect.
14 // (Parallel line segments don't intersect).
15 bool lineIntersectsLine(Line line1, Line line2);
16
17 // Return true if Line intersects Box.
18 bool lineIntersectsBox(Line line, Box b);
19
20 // Return true if Circle intersects or contains Box.
21 bool circleIntersectsBox(Circle circle, Box b);
22
23 // Return distance between any two positions.
24 float distance(Vector p1, Vector p2);
```

The rest of the utility function prototypes shown in Listing 4.152 are not needed for Dragonfly, but may be useful to game programmers. Lines 7 through 21 are variations of the `boxIntersectsBox()` function, but with different shapes.[19] Line 24 has a function that returns the distance between any two positions.

[19] Classes for Line and Circle are not provided in this book, but are left as exercises for the aspiring programmer.

Tip 20! Line of sight. Many games employ the use of line of sight to determine if one object can "see" another. For example, can the hero see the treasure chest behind a wall? Can the bad guy see the hero sneaking up? Can the mummy see an intruder in the halls of its tomb? In order to check if a first object can "see" a second object, a common technique is to draw a line from the first to the second. If the line does not intersect any other objects, the second object is spotted. In Dragonfly, the function `lineIntersectsBox()` can be used for this, checking each Object to see if a line from the position of the first Object to the position of the second Object for intersection with any other Object. If so, the intersected Object occludes the vision. If not, there is a clear line of sight.

4.13.4 Views

In a game like *Pac-Man*, the entire game board is visible on the computer screen. However, there are many games where this is not the case, games in which the game world is larger than what can be seen on the screen. Think of a game where the player explores a game-world, too vast to be contained to one, single computer screen. In such a case, the game shows a "window" that acts as a "viewport" over the world, with the camera moving to show different world *views* in response to player actions. Sometimes, the camera will move with an avatar, say, keeping the avatar in the window as the world behind it moves. This is what happens in a platformer such as *Super Mario*, where the player controls the main avatar (Mario), jumping and falling vertically and running left and right in a large game world while the camera follows the avatar. Similarly, in the case of an adventure game such as *The Legend of Zelda*, the player controls the main avatar (Link), exploring a very large world in the course of rescuing the princess. Omnipresent games, where the player has a top-down view of part of the game world, such as is the case of many real time strategy games, have the camera show part of the game world on the window while the entire game world is much larger. For Dragonfly, since it uses text-based cells, the view afforded by the camera is limited to the size of the initial window. When the game world is larger than this window, the game engine needs to map the world coordinates to the window coordinates.

Extensions to the WorldManager to support views are shown in Listing 4.153. The limit provided by the terminal window is called the *view boundary* and the limit provided by the game world is called the *world boundary*. Both boundaries are stored as Box attributes, privately kept in the WorldManager. Methods to get and set the boundaries are provided. By default, in the WorldManager constructor, the size of both boundaries, length and width, should be set to 0.

Listing 4.153: WorldManager extensions to support views

```
0 private:
1  Box boundary;  // World boundary.
2  Box view;      // Player view of game world.
3
4 public:
5  // Set game world boundary.
6  void setBoundary(Box new_boundary);
```

```
7
8   // Get game world boundary.
9   Box getBoundary() const;
10
11  // Set player view of game world.
12  void setView(Box new_view);
13
14  // Get player view of game world.
15  Box getView() const;
```

The GameManager sets the default world boundary and the view boundary to be the size of the initial window, obtained from the DisplayManager via `getHorizontal()` and `getVertical()` (see Section 4.8.2 on page 112).* The GameManager does this in its own `startUp()` method, after both the DisplayManager and WorldManager have been successfully started.

The world boundary as a Box provides an immediate opportunity to refactor the "out of bounds" event from Section 4.10.1.4 (page 148). Listing 4.154 depicts the new pseudo-code.

Listing 4.154: WorldManager moveObject() refactored for EventOut

```
0  // Move Object.
1  // ...
2  // If moved from inside world boundary to outside, generate EventOut.
3  int WorldManager::moveObject(Object *p_o, Vector where)
4
5     ...
6
7     // Do move.
8     Box orig_box = getWorldBox(p_o)  // original bounding box
9     p_o -> setPosition(where)        // move object
10    Box new_box = getWorldBox(p_o)   // new bounding box
11
12    // If object moved from inside to outside world, generate
13    // "out of bounds" event.
14    if boxIntersectsBox(orig_box, boundary) and // Was in bounds?
15       not boxIntersectsBox(new_box, boundary)  // Now out of bounds?
16       EventOut ov                              // Create "out" event
17      p_o -> eventHandler(&ov)                  // Send to Object
18    end if
19
20    ...
```

Game objects' positions are specified in relation to the world coordinates. In other words, the position attribute in an Object that provides an (x,y) location means that object should be at (x,y) in the world, but not necessarily at (x,y) on the window. With views, the world (x,y) position need to be mapped to the view/window (x,y) position.

Consider an example in Figure 4.7. The game world is 35x25 and the origin (0,0) is in the upper left corner. There are three Objects in the world: A is at (15,10), B is at

* **Did you know (#10)?** The *Globe Skimmer* dragonfly has the longest migration of any insect, back and forth across the Indian Ocean, about 11,000 miles. – "14 Fun Facts About Dragonflies", *Smithsonian.com*, October 5, 2011.

(8,5) and C is at (25,2). The coordinates depicted for all the Objects are relative to the game world. The view, 10x10, what the player sees on the window, is smaller than the game world, 35x25. The view origin, position (0,0) on the window, is at position (10,3) in game world coordinates. To correctly display Objects on the window, the view origin position is subtracted from each Object's position before drawing. For example, for Object A, subtracting (10,3) from (15,10) puts A at location (5,7) on the window. For Object B, subtracting (10,3) from (8,5) puts B at (-2,2). Since the x coordinate is negative, B is not drawn. For Object C, subtracting (10,3) from (25,2) puts C at (15,-1). Since the y coordinate is negative and 15 is greater than the window width, C is not drawn.

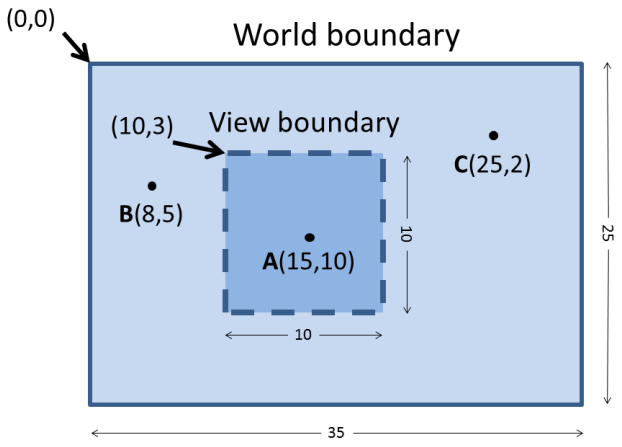

Figure 4.7: View boundary in relation to world boundary

This world-to-view translation is most easily done in the DisplayManager right before drawing a character on the window. The `utility.cpp` method `worldToView()`, shown in Listing 4.155, converts a world (x,y) position to a view (x,y) position on the window based on the current view.

Listing 4.155: worldToView()

```
0 // Convert world position to view position.
1 Vector worldToView(Vector world_pos)
2   view_origin = WorldManager getView().getCorner()
3   view_x = view_origin.getX()
4   view_y = view_origin.getY()
5   Vector view_pos(world_pos.getX()-view_x, world_pos.getY()-view_y)
6   return view_pos
```

The DisplayManager `drawCh()` is re-factored to call `worldToView()` right before each character is drawn, shown in Listing 4.156.

Listing 4.156: DisplayManager extensions to drawCh() to support views

```
0 int DisplayManager::drawCh(Vector world_pos, char ch, Color color) const
1   Vector view_pos = worldToView(world_pos)
```

```
  ...
```

Then, the subsequent calls to draw (e.g., the rectangle and the character) use `view_pos` instead of `world_pos`.

With views, not all Objects need to be drawn every loop. For example, in Figure 4.7, objects B and C are off the visible window so while calling `drawCh()` would not cause errors, it is a waste of time. In the WorldManager `draw()` method, instead of automatically drawing all objects, first the bounding box for each object is checked for intersection with the current view. If there is intersection, the Object is drawn. If there is not intersection, the Object is not drawn. This logic, done inside the "altitude" loop, is shown in Listing 4.157.

Listing 4.157: WorldManager extensions to draw() to support views

```
  ...
  // Bounding box coordinates are relative to Object,
  // so convert to world coordinates.
  temp_box = getWorldBox(p_temp_o)

  // Only draw if Object would be visible on window (intersects view).
  if boxIntersectsBox(temp_box, view) then
    p_temp_o -> draw()
  end if
  ...
```

In order to give the game programmer control over the view, the WorldManager is extended as indicated in Listing 4.158. The method `setViewPosition()` positions the view at a specific (x,y) world coordinate, and `setViewFollowing()` automatically moves the view to keep the indicated game object, stored in the private attribute `p_view_following`, in the center of the window. The latter is useful when the game programmer wants the camera to follow an avatar as it moves around the world, such as is typical in a platformer game.

Listing 4.158: WorldManager extensions to support view following Object

```
private:
 Object *p_view_following;  // Object view is following.

public:
 // Set view to center window on position view_pos.
 // View edge will not go beyond world boundary.
 void setViewPosition(Vector view_pos);

 // Set view to center window on Object.
 // Set to NULL to stop following.
 // If p_new_view_following not legit, return -1 else return 0.
 int setViewFollowing(Object *p_new_view_following);
```

Pseudo code for the method `setViewPosition()` is shown in Listing 4.159. The method takes in a position in the game world and sets the view to be centered on that position. In the first two blocks of code, the method makes sure that the edges of the view do not go outside of the edges of the game world, horizontally and then vertically. If it does, then the boundary is moved to flush with the edge.

Listing 4.159: WorldManager setViewPosition()

```
// Set view to center window on position view_pos.
// View edge will not go beyond world boundary.
void WorldManager::setViewPosition(Vector view_pos)

  // Make sure horizontal not out of world boundary.
  x = view_pos.getX() - view.getHorizontal()/2
  if x + view.getHorizontal() > boundary.getHorizontal() then
    x = boundary.getHorizontal() - view.getHorizontal()
  end if
  if x < 0 then
    x = 0
  end if

  // Make sure vertical not out of world boundary.
  y = view_pos.getY() - view.getVertical()/2
  if y + view.getVertical() > boundary.getVertical() then
    y = boundary.getVertical() - view.getVertical()
  end if
  if y < 0 then
    y = 0
  end if

  // Set view.
  Vector new_corner(x, y)
  view.setCorner(new_corner)
```

Pseudo code for the method `setViewFollowing()` is shown in Listing 4.160. The first block of code starting on line 6 checks if `p_new_view_following` is `NULL` – if so, this indicates the game programmer intends to stop having the view follow any particular Object.

The second block of code starting on line 12 iterates over all the Objects in the world. Each Object is compared to the `p_new_view_following` to make sure the game programmer has requested to follow a legitimate object. The boolean variable `found` is set to true if the Object is matched with one of the known game objects.

The third block of code starting on line 17 sets the `p_view_following` variable if the Object has been found and, if so, sets the view position to be centered on that Object.

If the Object is not found, the method returns -1 (an error).

Listing 4.160: WorldManager setViewFollowing()

```
// Set view to follow Object.
// Set to NULL to stop following.
// If p_new_view_following not legit, return -1 else return 0.
int WorldManager::setViewFollowing(Object *p_new_view_following)

  // Set to NULL to turn 'off' following.
  if p_new_view_following is NULL then
    p_view_following = NULL
    return ok
  end if

  // ...
  // Iterate over all Objects.  Make sure p_new_view_following
  // is one of the Objects, then set found to true.
```

```
14    // ...
15
16    // If found, adjust attribute accordingly and set view position.
17    if found then
18      p_view_following = p_new_view_following
19      setViewPosition(p_view_following -> getPosition)
20      return ok
21    end if
22
23    // If we get here, was not legit.  Don't change current view.
24    return error
```

The last adjustment is to the WorldManager `moveObject()` method. Here, at the very end of the method, if the Object has been successfully moved and the Object being moved is the same as the Object being followed, then the view is adjusted to the new position of the Object. This logic is shown in Listing 4.161.

Listing 4.161: WorldManager extensions to moveObject() to support views

```
0   ...
1   // If view is following this object, adjust view.
2   if p_view_following is p_o then
3     setViewPosition(p_o -> getPosition())
4   end if
5   ...
```

4.13.4.1 Advanced View Control (optional)

As implemented, when following an Object with the view, the camera keeps the Object dead-center on the screen at all times (subject to the world boundaries, of course). For the player, a camera locked in this mode can be tedious for a game where the player is moving the view a lot. There are a variety of advanced camera control techniques that could be incorporated into Dragonfly to provide for more advanced camer control techniques, including *zoning*, *blending* and *rails*. The interested developer is encouraged to see Phil Wilkins excellent talk on dynamic camera systems [9] with many more details in Mark Haigh-Hutchinson's book on real-time cameras [5].

An additional camera technique shown here is *dynamics*. With dynamics, the camera still follows an Object, but does not require the Object to be strictly in the middle of the screen. Instead, the Object can stay within a smaller rectangle inside the screen without the camera moving. This provides some "slack" that allows the Object to be within a center area before the camera needs to move.

To support these dynamics, the WorldManager is extended with a `view_slack` attribute which is a Vector representing the (x, y) dimensions of the inner rectangle. The default value for `view_slack`, set in the WorldManager constructor, should be (0, 0). Attributes to get and set `view_slack`, (`getViewSlack()` and `setViewSlack()`, respectively) should also be created.

Then, WorldManager `moveObject()` is re-factored to support dynamics, as shown in Listing 4.162.

Listing 4.162: WorldManager extensions to moveObject() to support view dynamics

```
0  ...
1  // If view is following this object, adjust view as appropriate.
2  if p_view_following is p_o then
3
4    // Get center of view.
5    view_center_x = view.getCorner().getX() + view.getHorizontal()/2
6    view_center_y = view.getCorner().getY() + view.getVertical()/2
7
8    // Compute inner "slack" Box edges.
9    left = view_center_x - view.getHorizontal() * view_slack.getX()/2
10   right = view_center_x + view.getHorizontal() * view_slack.getX()/2
11   top = view_center_y - view.getVertical() * view_slack.getY()/2
12   bottom = view_center_y + view.getVertical() * view_slack.getY()/2
13
14   new_pos = p_o -> getPosition()
15
16   // Move view right/left?
17   if (new_pos.getX() < left)
18     view_center_x -= left - new_pos.getX()
19   else if (new_pos.getX() > right)
20     view_center_x += new_pos.getX() - right
21
22   // Move up/down?
23   if (new_pos.getY() < top)
24     view_center_y -= top - new_pos.getY()
25   else if (new_pos.getY() > bottom)
26     view_center_y += new_pos.getY() - bottom
27
28   // Set new view position.
29   setViewPosition(Vector(view_center_x, view_center_y))
30 end if  // following p_o
31 ...
```

The first block of code, lines 4 to 12, compute the edges of the inner Box (the "slack" in the camera dynamics). The next block of code, lines 16 to 26, compare the position of the Object the camera is following to these edges, moving the edge as appropriate to keep the Object within the inner Box. The last bit of code, line 29, actually adjusts the view to the new location.

4.13.4.2 Using Views

The view support added to Dragonfly can be used by the game programmer to provide the player with a game world larger than the window. Assume, for example, that in Saucer Shoot (see Section 3.3 on page 15), the game programmer wants the game world to be about twice as large vertically as a window. This can be done by explicitly setting the world boundary in `game.cpp`, as shown in Listing 4.163, setting the view boundary to be the typical size of 80x24.

Listing 4.163: Explicitly setting game world boundaries

```
0 // Set world boundaries to 80 horizontal, 50 vertical.
1 Vector corner(0,0)
```

```
Box world_boundary(corner, 80, 50)
WorldManager setBoundary(world_boundary)

// Set view to 80 horizontal, 24 vertical.
Box view(corner, 80, 24)
WorldManager setView(view)
```

With the WorldManager controlling the world boundaries, code that generates the out event (see Section 4.10.1.4 on page 148) should be refactored to use the WorldManager `getBoundary()` instead of the DisplayManager window limits.

With the world larger than the window, the intent is probably to always keep the Hero centered vertically in the window. This could be done by extending the original `move()` method (Listing 3.11 on page 46) with the code defined in Listing 4.164. Basically, when the Hero moves vertically, the new code adjusts the view by the same vertical amount.

Listing 4.164: Example Hero extension to move() to support views

```
// Always keep Hero centered in window.
void Hero::move(float dy)
  // Move as before...

  // Adjust view.
  Box new_view = WorldManager getView();
  Vector corner = new_view.getCorner();
  corner.setY(corner.getY() + dy);
  new_view.setCorner(corner);
  WorldManager setView(new_view);
```

Alternatively, the WorldManager can just be told to follow the Hero by calling @@ `setViewFollowing()` in the Hero's constructor, as in Listing 4.165.

Listing 4.165: Example Hero extension to support views

```
// Always keep Hero centered in window.
void Hero::Hero()
  // ...
  WorldManager setViewFollowing(this);
```

Tip 21! Player camera control. Game programmers can give the player camera control without having an avatar with the following trick: a `SPECTRAL` game object is created without a sprite. The Object is programmed to respond to mouse or arrow keys by changing position – left, right, up, down. Then, Dragonfly is told to follow the game object via `setViewFollowing()`. To the player, it appears as if the controls are changing the camera!

4.13.5 Development Checkpoint #10!

Continue Dragonfly development, using Boxes to first provide bounding boxes for Objects and next to provide view and world boundaries. Steps:

1. Add the Box class, referring to `Box.h` in Listing 4.146. Add `Box.cpp` to the project and stub out each method so it compiles. The Box is really just a container, but test the Box class thoroughly, anyway. Do this outside of the engine, making sure that the attributes can all be get and set properly and that the constructor with corner, horizontal and vertical dimensions specified works.

2. Extend the Object class support bounding Boxes, referring to Listing 4.147. Test that Object bounding boxes can be set and retrieved properly. Be sure to extend `setSprite()` to set the Object Box to the dimensions of the associated Sprite, linked to the Animation attribute (`m_animation`).

3. Write the utility function `boxIntersectsBox()` (Listing 4.149) that determines if two Boxes overlap. Test this with a program that uses Boxes of a variety of dimensions and locations with all sorts of intersection combinations (including containment). Verify all test cases work before proceeding – this function gets called a *lot* in a typical game.

4. Replace `positionsIntersect()` with `boxIntersectsBox()` in the WorldManager. First, verify using former test code that the engine still works with single character Objects. Then, create test code with multi-character Sprites, testing a variety of Objects and collisions. Use one non-moving Object with one moving Object at first to make debugging simpler.

5. Add views, starting by extending the WorldManager to support views as in Listing 4.153. Test the get and set methods for the `view` and `boundary` attributes.

6. Write and test the DisplayManager `worldToView()`, referring to Listing 4.155 as needed. Put calls to `worldToView` in the DisplayManager `drawCh()`, as per Listing 4.156. Test that previous code without views still works, then test that having a view that is not positioned at the world's origin works as expected. At this point, use just a hard-coded view in the WorldManager.

7. Extend the WorldManager `draw()` method to only draw Objects that are in the view, as shown in Listing 4.157. Test with a variety of Objects that are in the view, completely out of the view, and partially in/out of the view.

8. Add settings to the WorldManager that enable setting the view, including attributes and methods from Listing 4.158. Refer to details from Listing 4.159 and Listing 4.160, as needed. Extend WorldManager `moveObject()` as in Listing 4.161.

9. For an integrated test, modify the Saucer Shoot tutorial to use views. First, set the game world boundaries as in Listing 4.163, then modify the Hero `move()` method as in Listing 4.164. Test thoroughly, making sure the window is smaller than the game world settings. Once convinced all works, revert back to the former `move()` method and have the view follow the `Hero()` as in Listing 4.165. Test this thoroughly, too.