Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #7

Dragonfly Naiad

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 11.0). Information online at: `http://dragonfly.wpi.edu/book/`

Copyright ©2012–2025 Mark Claypool and WPI. All rights reserved.

4.10 Kinematics

Kinematics is the physics that describes the motion of objects without taking into account forces or masses. In relation to our engine, up until now, moving an object has to be done in game code with the game programmer writing code to move an game object each step event. For example, if a Saucer needs to move, say 1 space to the left every 4 steps (0.25 spaces per step), when the Saucer receives the step event in the `eventHandler()`, it moves, as in Listing 4.98.

Listing 4.98: Saucer movement without velocity support

```
// Move 0.25 spaces per step.
int Saucer::eventHandler(const df::Event *p_e) override {
  if (p_e->getType() == df::STEP_EVENT) {
    df::Vector pos = getPosition();
    pos.setX(pos.getX() - 0.25);
    setPosition(pos);
    return 1; // Event handled.
  }
  return 0; // Event not handled.
}
```

While this can work, all of it can be tedious for a game programmer and prone to errors. Fortunately, a significant service provided by most game engines, including Dragonfly, is moving game objects automatically, in the right direction at the right speed. This allows a game programmer to provide a game object with a velocity and have the object move an appropriate amount in an appropriate direction each game step.

To support this, Object is extended with additional attributes for velocity, shown in Listing 4.99 – namely `m_direction` as a vector and `m_speed` as a float, initialized to `(0,0)` and `0`, respectively, in the Object constructor. They can be set by normal getters and setters.

Listing 4.99: Object extensions to support kinematics

```
 private:
  Vector m_direction;  // Direction vector.
  float m_speed;       // Object speed in direction.

 public:
  // Set speed of Object.
  void setSpeed(float speed);

  // Get speed of Object.
  float getSpeed() const;

  // Set direction of Object.
  void setDirection(Vector new_direction);

  // Get direction of Object.
  Vector getDirection() const;

  // Set direction and speed of Object.
  void setVelocity(Vector new_velocity);
```

```

  // Get velocity of Object based on direction and speed.
  Vector getVelocity() const;

  // Predict Object position based on speed and direction.
  // Return predicted position.
  Vector predictPosition()
```

In addition, getting and setting the Object velocity is done via `getVelocity()` and `setVelocity()`, respectively, which use Vector operations `scale()` and `normalize()` (see Listing 4.28 on page 77 and Listing 4.29 on page 78), as appropriate. Specifically: a) speed is just the magnitude of the velocity, a float, b) direction is the normalized velocity, a vector, and c) velocity is the direction scaled by the speed. So, with the developed Vector methods:

```
  // Set speed and direction based on velocity.
  speed = velocity.getMagnitude()
  direction = velocity.normalize()

  // Set velocity based on speed and direction.
  velocity = direction
  velocity.scale(speed)
```

The real magic happens in an Object support method – one that computes where an Object will be after a game loop's worth of velocity is added, but without actually moving it. This method, called `predictPosition()`, is shown in Listing 4.100. Basically, it indicates where an Object will be after it moves given the speed and direction attributes (i.e., the velocity)*.

Listing 4.100: Object predictPosition()

```
// Predict Object position based on speed and direction.
// Return predicted position.
Vector Object::predictPosition()

  // Add velocity to position.
  Vector new_pos = m_position + getVelocity()

  // Return new position.
  return new_pos
```

Once support for velocities is in the Object class, the WorldManager needs to be updated to use it. This is done by adding functionality to the `update()` method (see Section 4.5.6.2 on page 104). Listing 4.101 shows the necessary pseudo code.

Listing 4.101: WorldManager extensions to update() to support kinematics

```
  // Update Object positions based on their velocities.

  // Iterate through all Objects.
  for i = 0 to m_updates count

```

* **Did you know (#7)?** Much like humans, dragonflies use predictive models to intercept prey. – David Shultz. "Watch: A Dragonfly Predicts the Movements of Its Prey", *Science Magazine*, December 10, 2014.

```
    // Add velocity to position.
    Vector new_pos = m_updates[i] -> predictPosition()

    // If Object should change position, then move.
    if new_pos != getPosition() then
      moveObject() to new_pos
    end if

  end for // End iterate.
  ...
```

Basically, `update()` iterates through all Objects. For each, it calls `predictPosition()` to check if the Object should move or not. If so, it moves the object by calling `moveObject()`. The functionality inside `moveObject()` is discussed in the next section, Section 4.10.1.

Using velocity support in Dragonfly for the game programmer is easy. Instead of a game object having to move itself by handling every step event in the `eventHandler()` and determining when and where to move, the game programmer can just set the speed and direction for game objects (derived from the Object class, of course). For example, for a Saucer traveling right to left across the screen, the programmer sets the speed and direction (or setting the velocity) of the Object, as in Listing 4.102. There is no longer a need for a game object to handle the step event for moving.

Listing 4.102: Saucer movement with velocity support

```
// Set movement left 1 space left every 4 frames.
setSpeed(0.25);
setDirection(df::Vector(-1.0, 0));

// Note: the above two lines are equivalent to:
setVelocity(df::Vector(-0.25, 0));
```

Newtonian Mechanics Physics (optional) Velocity support is complete, but could be enhanced. Possible options could extend the velocity support to include a acceleration component (another `Vector`) for each game object. With acceleration, a game object's velocity would change slightly (it would accelerate or decelerate according to the direction) each step. This could be especially useful for providing, say, gravity pulling an avatar down (a fixed acceleration in the vertical direction) for a platformer game.

While the kinematics in Dragonfly provides support for a lot of games, additional classical Newtonian physics mechanics include forces and masses. In fact, while acceleration can simply be added to objects as mentioned in the previous paragraph, in the real world it is achieved through application of a force on an object through the equation:

$$F = m \cdot a$$

where F is the applied force, m is the object mass and a is the acceleration. Adding a mass attribute to game objects would be a first step to using forces as a precursor so acceleration and velocity.

And that is really just the tip of the iceberg. More advanced physics techniques that could be incorporated into Dragonfly include elastic and non-elastic collisions, rigid body,

soft body and ragdoll simulation, joints as constraints, and more. The aspiring programmer is encouraged to check out one of the many books on physics for game engines, such as the *Game Physics Engine Development* by Ian Millington [6].

4.10.1 Collisions

A closely related service to moving a game object is detecting when two or more game objects collide and, when they do, triggering object responses as appropriate. However, determining when objects collide is not as easy as it may seem:

- Geometries of objects can be complex – square or circular objects are efficient in terms of computing area (or volume, if 3d) and locations in the game world, but more complex objects, for example humanoids and trees, take many more computations to compute exact locations in the game world.
- Objects can be moving fast – objects that move great distances mean there is a larger area (or volume, if 3d) from the old position to the new position that may result in potential collisions.
- There can be many objects – the more objects there are, the more opportunities there are for collisions and the more computations the game engine needs to do each step to determine if objects collide. A naïve solution could take $O(n^2)$, meaning every object (all n of them) is checked for a collision with every other object, so $n \times n$ comparisons are made each step.

There are many possibilities when it comes how to implement to collision detection. Dragonfly uses what is perhaps the most commonly used technique for games – *overlap testing*. Overlap testing is relatively easy, both conceptually (for the game programmer as well as the game engine programmer) and in terms of implementation. However, it may exhibit more errors than some other forms of collision. Basically, with overlap testing, when an game object moves, the engine checks whether or not its area (or volume, if 3d) overlaps that of any other object. If so, there has been a collision. If not, then the object can move unimpeded. When a collision has occurred, both game objects get notified of the event.

Despite the advantages (simplicity and speed of execution), overlap testing can fail when game objects move too fast relative to their size. Consider the game example below where an arrow is fired at a window. The arrow has a velocity of 15, meaning it moves 15 spaces horizontally each step.

```
                                |
>---            >---            |        >---
 t1              t2             |         t3
```

At time $t1$, the arrow (`>---`), moving left to right, approaches the window (`|`). One step later, at time $t2$, the arrow's velocity moves it 15 spaces to the right and a collision seems imminent. However, there is not yet an overlap between the arrow and the glass, so no collision is detected. At time $t3$, the arrow's velocity carries it another 15 spaces to the

right, past the window and as there is still no overlap, no collision is detected. Effectively, the arrow appears to have gone right through the glass window with breaking it (or even hitting it)!

There are several possible solutions to this problem. A game programmer, knowing that the engine is using overlap testing, can put a constraint on game object sizes and speeds such that the fastest object moves slow enough (per step) to overlap with the thinnest object. For the example above, the window would need to be 13 spaces thick, the arrow would need to be 13 spaces long, or the arrow would need to only move 3 spaces per step, or some combination of the above. While this would ensure the arrow would always hit the window, it might not be practical for all games (e.g., the arrow speed might be too slow for other uses and/or the window too thick for the environment).

Another solution is to reduce the step size. The step size dictates how often the game world is updated. Game programmers want aspects such as the speed of an object to be relative to real time in terms of how fast the player sees the arrow move across the window, and not relative to the game step. This means, for example, that a velocity of 1 with a step frequency of 30 f/s moves 30 spaces per second. If the frequency was doubled, to 60 f/s, then the game programmer would want to adjust the velocity to be 0.5 so the player still sees the object move 30 spaces per second. What this will do, however, is make the object move fewer spaces each step, making it less likely for overlap testing to fail. However, this comes at a cost – namely, increased computation as the game loop runs more often (in this example, twice as often), possibly leading to the engine not being able to keep up with the update rate. *

A more complex solution (incidentally, *not* supported by Dragonfly) is to have a different step size for different objects. Thus, fast objects and/or small objects would be updated more frequently than slow objects. This adds complexity and computation overhead in the update step, as well, but could be applied selectively.

The overlap test itself is fairly easy to compute with simple volumes, like circles (or spheres, if 3d). More complex volumes can require more computation. For example, a humanoid-shaped object and a tree-shaped object may be made up of hundreds or even thousands of smaller polygons. If every polygon in an object needs to be tested for overlap with the polygons of every other object to determine if the objects collide, this scales $O(n^2)$ for n polygons per object.

To overcome this, complex geometries are often simplified in order to reduce the number of comparisons needed. Rather than doing this simplification in the model itself (which can compromise how it looks), each object can instead be given a bounding volume that approximates the object. For example, the humanoid-shaped object or a tree can be approximated by an ellipsoid. For a depiction of this last case, consider the brown, ASCII tree in Figure 4.3. Computing whether a single character collides with this tree would involve checking all the nooks and crannies of the branches – effectively examining every character in the tree for a collision as if each character was a separate object. Instead, the entire tree can be bound by a simpler shape for computing collisions – in this case, an ellipse, shown by the dashed oval surrounding the tree. Determining if another object collides with this

* **Did you know (#8)?** Adult dragonflies have 6 legs but cannot walk. – "30,000 Facets Give Dragonflies a Different Perspective: The Big Compound Eye in the Sky", *ScienceBlogs*, July 8, 2009.

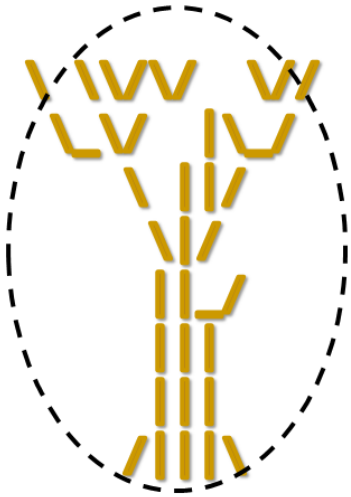

Figure 4.3: Object with bounding ellipse

tree is now much simpler, merely computing if it intersects the ellipse. It might be noted that the oval itself does not fully encompass the tree – some of the branches are poking out. This would mean that an object could hit those external branches and not collide with the tree. For many games, this might be just fine and could be preferred over drawing a bigger ellipse.

Commonly used bounding volumes are circles (or spheres, if 3d) where an overlap test compares the distances between the centers with the sum of the radii. Another common bounding volume is a rectangular box, or a *bounding box*.[17] In Dragonfly, each Object has a bounding box encompassing the game object, representing the Object with a simplified geometry.

In Dragonfly, if the edges of any two boxes overlap, there is a collision. A optional refinement could be to provide an game programmer option for precise collision testing. In this case, the bounding boxes would be tested for overlap normally. If there was an overlap, then a more refined test could be done to see if the individual characters in the object sprite overlapped, indicating a collision. Depicting this, consider the two objects in Figure 4.4 on the left. The Saucer and the Hero ship are clearly not touching – boxes around each do not intersect. However, a short time later, the game world state may be as on the right. Here, the boxes do overlap. Under normal Dragonfly operation, this overlap itself would indicate a collision. However, looking more closely, the characters themselves do not overlap. So, an alternative option could be that if the boxes overlap, then more precise consideration of each character is done to see if there is, in fact, a collision.

Once a collision is detected, action is taken to resolve the collision. What the actions are, exactly, often depends upon what the game objects are. For example, if two billiard balls collide, then computation as to where, exactly they hit is done first, followed by computation of new velocities for both balls, accompanied by playing a "clinking" sound.

[17] Bounding boxes are also known as *hitboxes* since they are used to determine if an object is "hit".

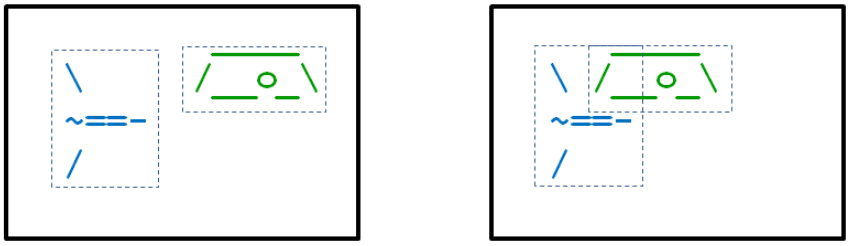

Figure 4.4: Refined collision detection with boxes

As another example, if a rocket slams into a rock wall, the rocket is destroyed, the wall takes on a charred texture and an explosion animation object is created. As a third example, a character walks through an invisible wall, a magic, propagating ripple effect is triggered, but no velocities or other impacts are recorded.

In Dragonfly, the engine detects collisions through overlap testing of bounding boxes, then provides a collision event to both Objects involved in the collision. Resolution involves sending the collision event to both Objects. The Objects themselves receive the event in their `eventHandler()` methods and can react appropriately.

4.10.1.1 Collidable Entities

However, not all Objects are collidable entities. Consider display elements on the screen, such as player score or indication of time left. In most cases, these objects do not collide with, say, the player avatar. Similarly, background objects that provide scenery, such as the Stars in Saucer Shoot (Section 3.3), do not collide with other game elements. To support this, Dragonfly has a notion of "solidness", with three states defined via an `enum`, as in Listing 4.103. This `enum` is defined in `Object.h`.

Listing 4.103: Solidness states

```
enum Solidness {
  HARD,      // Object causes collisions and impedes.
  SOFT,      // Object causes collisions, but doesn't impede.
  SPECTRAL,  // Object doesn't cause collisions.
};
```

Collisions only happen between solid Objects – Objects that are `SPECTRAL` cannot collide. So, for example, Saucer Shoot's Stars are `SPECTRAL` and not solid. Solid Objects are either `HARD` or `SOFT`. A `HARD` object cannot occupy the same space as another other `HARD` Object, but any number of `SOFT` Objects can occupy a space, with our without at most one `HARD` Object.

In addition to notifying colliding solid (`HARD` or `SOFT`) Objects of the event, resolution then disallows two `HARD` objects to occupy the same space. Basically, if a first solid Object moves onto (collides with) a second `HARD` Object, the first object is moved back to its original location – the collision still happens, but the movement does not take place.

The Object class extensions needed to support solidness are shown in Listing 4.104. The attribute `solidness` determines if the Object is treated as solid (`HARD` or `SOFT`) or non-

solid (SPECTRAL). By default, Objects should be HARD (set in the Object constructor). The method isSolid() provides a convenient boolean test for whether or not the Object is solid. Methods to get and set the solidness are provided in getSolidness() and setSolidness(), respectively. For setSolidness(), if the new value passed in is one of HARD, SOFT or SPECTRAL then 0 is returned and the solidness is changed, otherwise -1 is returned and the solidness is unchanged.

Listing 4.104: Object class extensions to support solidness

```
private:
  Solidness m_solidness; // Solidness of object.

public:
  bool isSolid() const; // True if HARD or SOFT, else false.

  // Set object solidness, with checks for consistency.
  // Return 0 if ok, else -1.
  int setSolidness(Solidness new_solid);

  // Return object solidness.
  Solidness getSolidness() const;
```

4.10.1.2 Collision Event

When Dragonfly detects a collision, it sends a collision event to each Object involved. Listing 4.105 provides the header file for the EventCollision class, derived from the Event class (Listing 4.51 on page 96). Like other event classes, the EventCollision is a "container" and does not have any significant functionality itself. The attributes store information about the collision: m_pos stores the position where the collision occurred; m_p_obj1 is a pointer to the Object moving, the one causing collision; and m_p_obj2 is a pointer to the Object being collided with. Methods are provided to get and set each of the attributes.

Listing 4.105: EventCollision.h

```
#include "Event.h"
#include "Object.h"

const std::string COLLISION_EVENT = "df::collision";

class EventCollision : public Event {

 private:
  Vector m_pos;      // Where collision occurred.
  Object *m_p_obj1; // Object moving, causing collision.
  Object *m_p_obj2; // Object being collided with.

 public:
  // Create collision event at (0,0) with o1 and o2 NULL.
  EventCollision();

  // Create collision event between o1 and o2 at position p.
  // Object o1 'caused' collision by moving into object o2.
  EventCollision(Object *p_o1, Object *p_o2, Vector p);
```

```

  // Set object that caused collision.
  void setObject1(Object *p_new_o1);

  // Return object that caused collision.
  Object *getObject1() const;

  // Set object that was collided with.
  void setObject2(Object *p_new_o2);

  // Return object that was collided with.
  Object *getObject2() const;

  // Set position of collision.
  void setPosition(Vector new_pos);

  // Return position of collision.
  Vector getPosition() const;
};
```

4.10.1.3 Collisions in the WorldMananger

Dragonfly handles collisions inside the WorldManager. In order to do this, the WorldManager needs to be extended with several methods.

The first method is `getCollisions()` which tests whether or not an Object moving to a given location has a collision there. If so, all Objects at that location are returned in an ObjectList (there can be more than one Object at a location since multiple `SOFT` Objects can occupy the same location, with up to one `HARD` Object).

The second method is `moveObject()` that moves an Object to another location, as long as there is not a collision between two `HARD` Objects at this location.

Listing 4.106: WorldManager extensions for collision support

```
 public:
  // 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);

  // Move Object.
  // If collision with solid, send collision events.
  // If no collision with solid, move ok else don't move Object.
  // If Object is Spectral, move ok.
  // Return 0 if move ok, else -1 if collision with solid.
   int moveObject(Object *p_o, Vector where);
```

An additional utility function (in `utility.h` and `utility.cpp`) called `positionsIntersect()` is helpful for the WorldManager `getCollisions()` method, as shown in Listing 4.107. Basically, if two Positions are the same (their x coordinates are within 1 space of each other *and* their y coordinates are within 1 space of each other) then the method returns `true`, otherwise it returns `false`. This method may seem somewhat trivial, but it serves as

a placeholder for a method introduced in Section 4.153 on page 183, boxIntersectsBox(), that will test whether or not two bounding boxes overlap.

Listing 4.107: Utility positionsIntersect()

```
// Return true if two positions intersect, else false.
bool positionsIntersect(Vector p1, Vector p2)
  if abs(p1.getX() - p2.getX()) <= 1 and
     abs(p1.getY() - p2.getY()) <= 1 then
    return true
  else
    return false
  end if
```

The WorldManager getCollisions() method is provided in Listing 4.108. Since the method returns a list of all Objects that are collided with, line 6 creates an empty list (collision_list) to start. The next block of code iterates through all the Objects in the game. Each Object is first checked if it is the same Object passed into getCollisions() – if it is, it is ignored since an Object cannot collide with itself. If it is not, then the Object is checked to see if it is at the same location, and that it is solid. If both of these are true, the Object is added to the collision_list. When the loop is finished, the list of all Objects collided with are returned in collision_list. Note, if no Objects have been collided with, collision_list is an empty list.

Listing 4.108: WorldManager getCollisions()

```
0  // Return list of Objects collided with at position 'where'.
1  // Collisions only with solid Objects.
2  // Does not consider if p_o is solid or not.
3  ObjectList getCollisions(const Object *p_o, Vector where)
4
5    // Make empty list.
6    ObjectList collision_list
7
8    // Iterate through all Objects.
9
10   for i = 0 to m_updates count
11
12     Object *p_temp_o = m_updates[i]
13
14     if p_temp_o is not p_o then  // Do not consider self.
15
16       // Same location and both solid?
17       if positionsIntersect(p_temp_o->getPosition(), where)
18          and p_temp_o -> isSolid() then
19
20         add p_temp_o to collision_list
21
22       end if // No solid collision.
23
24     end if // Not self.
25
26   end for // End iterate.
27
28   return collision_list
```

The getCollisions() method is used in the moveObject() method, shown in Listing 4.109. The moveObject() method first checks if the calling Object (the one that wants to be moved) is solid (calling isSolid()) – if not, there are no further checks needed and the Object can move. If the Object is solid, getCollisions() is called to produce a list of solid Objects collided with. That collision list is iterated through,[18] and each Object is sent a collision event starting at line 23. If both Objects are HARD (line 31), then the move will not be allowed by setting do_move to false in line 32. If a move is allowed (no HARD collisions), then the actual move happens at the end of the method, on line 47.

Listing 4.109: WorldManager moveObject()

```
0  // Move Object.
1  // If collision with solid, send collision events.
2  // If no collision with solid, move ok.
3  // If all collided objects soft, move ok.
4  // If Object is spectral, move ok.
5  // If move ok, move.
6  // Return 0 if moved, else -1 if collision with solid.
7  int WorldManager::moveObject(Object *p_o, Vector where)
8
9    if p_o -> isSolid() then // Need to be solid for collisions.
10
11     // Get collisions.
12     ObjectList list = getCollisions(p_o, where)
13
14     if not list.isEmpty() then
15
16       boolean do_move = true  // Assume can move.
17
18       // Iterate over list.
19       for i = 0 to list count
20
21         Object *p_temp_o = list[i]
22
23         // Create collision event.
24         EventCollision c(p_o, p_temp_o, where)
25
26         // Send to both objects.
27         p_o -> eventHandler(&c)
28         p_temp_o -> eventHandler(&c)
29
30         // If both HARD, then cannot move.
31         if p_o is HARD and p_temp_o is HARD then
32           do_move = false  // Can't move.
33
34         end if
35
36       end for // End iterate.
37
38       if do_move is false then
39         return -1 // Move not allowed.
40       end if
```

[18] Note, if the collision list is empty, the iteration loop immediately ends since there is nothing to collide with.

```
41
42      end if // No collision.
43
44    end if // Object not solid.
45
46    // If here, no collision between two HARD objects so allow move.
47    p_o -> setPosition(where)
48
49    // All is well.
50    return ok // Move was ok.
```

Disallow Movement onto Soft Objects (optional) The construct of soft Objects allows many solid objects to reside in one location (if they are `SOFT`). This allows a solid Object to move onto a group of soft objects, generating collisions for each one in the process.

However, a game programmer may not want to allow some Objects to move onto the soft Objects. Dragonfly can be extended to support this functionality, first by adding an additional attribute to the Object class, shown in Listing 4.110, along with basic functions to get and set the attribute. If the attribute `m_no_soft` is `true` (it should default to `false`), the Object is not allowed to move onto soft game objects (but will still generate collisions – effectively, soft game objects are treated as hard game objects in this case).

Listing 4.110: Object class extensions to support no soft

```
0 private:
1   bool m_no_soft;    // True if won't move onto soft objects.
2
3 public:
4   // Set 'no soft' setting (true - cannot move onto SOFT Objects).
5   void setNoSoft(bool new_no_soft=true);
6
7   // Get 'no soft' setting (true - cannot move onto SOFT Objects).
8   bool getNoSoft() const;
```

Then, the WorldManager `moveObject()` method is refactored. In Listing 32 on page 148, similar to the block of code around line 32 that indicates the Object should not move if both Objects are solid, an additional check is made if the Object being moved is soft and the Objects are soft. This is shown in Listing 4.111.

Listing 4.111: WorldManager extensions to moveObject() to support no soft

```
0   ...
1   // If object does not want to move onto soft objects, don't move.
2   if p_o->getNoSoft() and p_temp_o->getSolidness() is SOFT then
3     do_move = false
4   end if
5   ...
```

4.10.1.4 World Boundaries – Out of Bounds Event

Generally, game objects expect to stay inside the game world. This is not to say that all game objects are on the *screen* at the same time – for example, think of a side scroller

where a lot of the level is off the screen – but game objects stay in the known game world. As such, when a game object moves outside of the game world it is helpful to tell indicate this via an event. The out of bounds object still has the option to ignore the event and stay outside the game world, but in many cases, it will want to take action, such as moving back inside the game world or, in the case of a bullet that would otherwise fly off forever, destroy itself.

Specifically for Dragonfly, when an Object inside the game world moves outside the game world, the WorldManager generates an out of bounds event, an EventOut. The move is still allowed, giving the Object freedom to stay outside of the game world should it so choose, but providing the information in the form of the event in case the Object wants to act.

Listing 4.112 provides the header file for the EventOut class. As for all Dragonfly events, EventOut is derived from the base Event class. Unlike some other events, EventOut only has information that the event occurred – i.e., that the object moved from inside to outside the game world. The Object itself already knows where (it has its (x,y) position) so that does not need to be indicated. The constructor sets `event_type` to `OUT_EVENT`. There are no other attributes and no additional methods – the game programmer merely needs to consult the Event type (defined in the parent class, `Event::getType()`) to determine what happened.

Listing 4.112: EventOut.h

```
#include "Event.h"

const std::string OUT_EVENT = "df::out";

class EventOut : public Event {

 public:
  EventOut();
};
```

As suggested above, game world boundaries can be different than window boundaries – this functionality is supported by Dragonfly in subsequent development (see Section 4.13.4 on page 187). However, at this point in Dragonfly, the game world boundaries are determined by the window boundaries. In order to determine the boundaries of the world, the DisplayManager routines `getHorizontal()` and `getVertical()` can be used.

The WorldManager `moveObject()` is extended slightly. After a move is allowed (line 47 in Listing 4.109), the Object's position is checked against the horizontal and vertical limits obtained from the DisplayManager. If the Object is out of bounds, an EventOut object is generated and sent to the Object's event handler as in Listing 4.113.

Listing 4.113: Generate and send EventOut

```
  ...
  // Generate out of bounds event and send to Object.
  EventOut ov
  p_o -> eventHandler(&ov)
  ...
```

Note, the WorldManager only sends an EventOut once, when the Object first moves from inside to outside the game world. If the Object moves outside then stays outside, no additional events are generated. Presumably, the Object already knows it is outside the game world upon receiving the first out event, so if it stays outside it does not need to be reminded.

4.10.2 Program Flow for Moving Objects

We can step back and summarize from a high level the program flow that goes into moving game Objects, depicted in Listing 4.114.

Listing 4.114: Program Flow for Moving Objects

```
GM.run()
  ...
  WM update()
    Object predictPosition()
      new_pos += velocity
    moveObject()
    getCollisions()
    // Send any needed collision events.
    if can_move then
      Object setPosition(new_pos)
    end
    // Send any needed out of bounds events.
  ...
  ...
```

Inside the game loop, the game manager calls WorldManager `update()`. For each Object, the world manager asks each Object to predict its position by calling Object `predictPosition()`, which computes the predicted position by adding the Object's velocity to the current position. The world manager then calls `moveObject()` calls `getCollisions()` to get a list of any collisions that would result if the Object moved to the predicted position. The world manager then sends a collision event (EVentCollision) to any and all Objects in that list. If the Object can actually move to the predicted position (i.e., there are not two `HARD` Objects in the same location), the world manager actually changes the Object's position by calling `setPosition()`. Lastly, the world manager computes if the Object was inside the game world and then went out and, if so, sends it an out of bounds event (EventOut).

4.11 Development Checkpoint #7 – Dragonfly Naiad!

Continue with your Dragonfly development by adding to your code base to create a *Dragonfly Naiad*.* Steps:

1. Extend the Object class to support kinematics, as in Listing 4.99. Add code to Object to predict the position if it applies a step of velocity, as in Listing 4.100. Add extensions to `update()` to do the velocity step for all Objects, as in Listing 4.101.

* **Did you know (#9)?** A *naiad* is a dragonfly in larval stage.

2. Test the velocity code thoroughly by making Objects with different starting locations and different velocities. Verify visually and via logfile messages that Objects move an appropriate amount each step. Test with different velocity values (x and y), positive and negative and greater than 1 and less than 1.

3. Extend the WorldManager to support collisions, as in Listing 4.106. Implement support function `positionsIntersect()` in `utility.cpp`, as per Listing 4.107. Implement `getCollisions()` in the WorldManager based on Listing 4.108. Support for solidness needs to be added to the Object class, based on Listing 4.104 and Listing 4.103. Lastly, write `moveObject()` based on Listing 4.109. EventCollision needs to be created for this, following Listing 4.105, and can be tested outside of the game engine before using it in `moveObject()`. Since the code for all the above is fairly extensive, add liberal `writeLog()` statements to provide meaningful output to verify it is working.

4. Write numerous test cases to verify that collisions work properly. Start first with a solid Object that attempts to move onto another solid Object. Verify the move is not allowed (visually and in the logfile) and make sure both Objects get an event with appropriate pointers. Next, test that soft Objects can move on each other, but that all soft Objects colliding get collision events. Lastly, check with test examples that spectral Objects do not generate collisions.

5. Create an EventOut class based on Listing 4.112. Add `EventOut.cpp` to the project and stub out each method so it compiles. Add code to the WorldManager `moveObject()` method that sends the out of bounds event to objects that move out of the game world. Refer to Listing 4.113, as needed.

6. Test the out of bounds additions by making a game object that starts inside the game world, but soon moves out. Output messages should be seen in the logfile, but events should checked and handled by the Object `eventHandler()` method. Make different Objects that move in and out, multiple times and verify only one EventOut message is generated each time an Object moves from inside to outside.

After completing the above steps (and all the previous Development Checkpoints), you will have a fully functional game engine!

Features include:

- Objects can draw themselves in 2d, as colored text characters.
- Objects can appear above (foreground) or behind (background) when drawing.
- Objects can get input from the keyboard and mouse.
- Objects are moved automatically based on their velocities.
- Objects that move out of the game world get an "out of bounds" event.
- Objects have solidness – soft, hard, or spectral – affecting movement and collisions.

- Solid Objects that collide get a collision event, providing information on both Objects enabling them to react appropriately.

The above functionality to support objects, graphics, input, and interaction with collisions allows creation of a wide variety of games. Consider, for example, making the game Saucer Shoot from Section 3.3. The core gameplay for Saucer Shoot can be made (aside from the Points and Nuke ViewObjects), with the main exception that Dragonfly Naiad only supports single character game objects without animation, rather than animated, multi-character sprites.