Adding some simple physics to a game can make it much more realistic. If you go to amazon you will find numerous books on game physics. If you want to learn more go explore. Some web starting places:

http://www.rit.edu/~ndr0470/game_physics/

http://www.krazydad.com/bestiary/bestiary_superball.html (and following pages)

Let's start with the concept of acceleration. When you jump into a car and hit the gas, intending to drive at legal and prudent speed of 55 miles per hour, the car does NOT instantly go from 0 to 55. Instead, you accelerate from 0 up to 55. The velocity of an object, usually represented mathematically as a vector, is the speed and direction of an object. Velocity is measured in (distance / time), usually (meters / second) or (miles / hours) or (Kilometers / hour). Acceleration is the rate at which velocity changes and hence has units of (distance / time / time).

There is no reason to model acceleration “exactly”, all we need to do is achieve the same appearance of acceleration so it is convincing.

Run each of the following three animation in order:

(a_physics_1a.fla)
(a_physics_1b.fla)
(a_physics_1c.fla)

The first example just uses velocity, i.e. 0 to 55 instantly. Not believable. The next two files give the acceleration look.

The common way to get the acceleration effect is that when you update the location of an object by incrementing with it’s velocity, you also increment the velocity so that the next frame the increase is greater and so on. The pertinent code in a_physics_1b.fla is:

```javascript
var xv:Number = 1.0 ;
var acceleration:Number = 0.5 ;
onEnterFrame = function() {
   theBall._x += xv ;
   if (xv < 15) xv += acceleration ; // once > 15 stop accelerating
}
```

In this code you can see that not only do we increment the _x location of the ball on each frame by velocity “xv”, but, we also increase velocity “xv” on each frame so the ball goes
faster and faster. Now we want a top speed (velocity), so we stop increment “xv” once it is greater than the top velocity.

A second and very similar way to achieve this effect is to increase the velocity multiplicatively instead of additively. That is was is done in a_physics_1c.fla:

```actionscript
var xv:Number = 1.0;
var acceleration:Number = 1.08;

onEnterFrame = function() {
   theBall._x += xv;
   if (xv < 15) xv *= acceleration; // once > 15 stop accelerating
}
```

Here we just keep multiplying “xv” by some number greater than 1.0 until the velocity exceeds our top speed.

Acceleration can be either positive (increasing speeds) or negative (decreasing speeds). Just as a car does not go instantly from 0 to 55, it also does not go instantly from 55 to 0, unless it hits a wall! Check out the code in:

(a_physics_2.fla)

Here you see the use of negative acceleration to slow down the ball. It is done multiplicatively in this example, but could be done additively. For multiplication the number needs to be less than 1.0. For addition the number needs to be the opposite of the speeding up number (which depends on the direction). In this example, once the ball velocity reaches a sufficiently small number the velocity is just set to zero.

Often games need the concept of friction. Lets say you are creating a game on curling. You know, the funny game where they throw big weights on ice and one person sweeps in front of the path of the weight. No doubt sure to be the most popular flash game on the web! Anyways, you need the weights to slow down. This slowing down is caused by friction. Ice has low friction, but if you broom snow in the path the weight is slowed down more quickly. You can use the concept of negative acceleration to mimic friction.

Another special case of acceleration is gravity. When objects fall to the earth there speed is accelerated. Watch someone drop an object. It start out falling slowly than speeds up. Gravity has an acceleration of 9.8 meters / second / second. Again, there is no reason to model this exactly. Just mimic it. If you look at the code in the following to examples you will see the effect of velocity alone versus acceleration:

(a_physics_3a.fla)
(a_physics_3b.fla)

The code is really just the same as before, but I think you agree the effect becomes even more important once we add images in a “known” scenario where we have expectations on how objects should behave.
Another common use for physics is in trajectories of objects such as bullets, missiles, or flying frogs. Check out the following file:

(a_physics_4.fla)

The ball has an x-velocity and a y-velocity. The x-velocity is not changed (although actually air particle friction would slow it down a tad so small it can be ignored, unless we are talking about the trajectory of a feather or parachute) but the y-velocity is changed. When an object is moving up away from the earth gravity pulls it back to earth by causing an acceleration in the direction of the earth. The code below, from a_physics_4, is the essence of one way to achieve this affect.

```actionscript
var redBall:MovieClip = attachMovie("redBall",rb,1) ;
redBall._y = 200 ;
redBall._x = 1 ;
var yv:Number = -5.0 ;
var xv:Number = 5.0 ;
var gravity:Number = 0.2 ;

onEnterFrame = function() {
    redBall._y += yv ;
    redBall._x += xv ;
    yv += gravity ;
}
```

Notes 24: Game AI

The whole goal of AI (artificial intelligence) in games is to make the game more fun to play by making it a more challenging and realistic opponent. Say you are playing a game of chess or checkers, and the computer simply moved a piece at random every turn. Very easy to program, but who other than a 5 year old would want to play such a stupid game? Incidentally, you would be doing the 5 year old a disservice by letting them play such a game. The algorithms that the game uses to decided next moves are called AI. We do not have time to go into all different types of AI and will limit ourselves to path finding on tile mazes. For a good treatment of AI, consider the following two sources:

http://www.gameai.com/ai.html


http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
Path Finding

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

Lets assume the above 6x6 set of tiles and that we want to find a path for our sprite from cell 28 to cell 11. Lets further assume that we can not move on diagonals and that we can not move into black cells. Above are two examples, the left has not obstacles, whereas the right example has obstacles in tiles 10, 16, 17, and 18. This problem is called the “path finding” problem. There are many AI techniques for path finding. Here we will present only two simple methods and a third more sophisticated. The three methods are:

a) Breadth first search  
b) Directed Depth first search  
c) A star

We will use a similar framework and notation for all three to make it easier to understand the differences between the three. For each algorithm there are two key cells:

- **Start**: The cell we are starting at, represented by its coordinates: (i,j)  
- **Goal**: The cell we want to reach, represented by its coordinates: (i,j)

In the algorithms we will refer to the members of the cells. These are not necessarily the members found in the GameBoardTiles, but rather the members as needed for the algorithms. The cell structure for the first two algorithms is as follows, for the A-star algorithm some additional cell members are needed:

Cell Structure:

- (i,j): The coordinates of the cell  
- parent: The cell (i,j) from which this nodes was reached

The algorithms need a queue to hold the cells. If you remember, a queue is a data structure that is FIFO (first in first out). You add items at the end of the queue and you take them off from the front of the queue. In the first two algorithms we will call this queue the **ToVisitQueue**. In addition we need a second data structure (lets say an array) to keep track of which cells we have visited. Lets call this the **VisitedArray**.
**Breadth First Search**

a) Add the starting cell to the **ToVisitQueue**

b) Repeat until goal reached or the **ToVisitQueue** is empty:

   A. $N = \text{First cell from ToVisitQueue}$. If $N$ is the goal, exit the loop.
   
   B. Add $N$ to the **VisitedArray**

   C. For each of the $N$'s 4 adjacent squares, denoted $\{a_1 \ldots a_4\}$:

      I. If $a_i$ is the goal, exit the repeat loop
      
      II. If $a_i$ is not in the **ToVisitQueue** and also not in the **VisitedArray**, add $a_i$ to the end of the **ToVisitQueue** and set $a_i.parent$ to $N$

If you wanted to allow for diagonal movement then you would need to consider each of the 8 adjacent square instead of only 4.

c) If the **ToVisitQueue** is empty and the goal is not found, there is no path. Otherwise, create the path by following the chain of parent pointers backwards from the goal node just removed.

To really understand how this works trace through the state of the data structures step-by-step. Let's consider the example at the top of this section, wanting to move from cell 28 to cell 11. Let's consider the no obstacle example. Cell 28 would be put in the **ToVisitQueue**. Then it is removed and cells $\{27, 22, 29, 34\}$ are added. Each cell has a ptr back to 34, thus, if we represent one cell as (cell, parent) the state of the **ToVisitQueue** would be:

T: (27,28) (22,28) (29,28) (34,28)  
V: (28,28)

Where “T” above means the entries in the ToVisitQueue and “V” is the entries in the VisitedArray. Next we remove the first element from the ToVisitQueue add its neighbors (they can be added in any order, I am assuming they are being added W,N,E,S) resulting in:

T: (22,28) (29,28) (34,28) (26,27) (21,27) (33,27)  
V: (28,28) (27,28)

We keep doing this until we get to the goal cell. After each iteration we have:

T: (29,28) (34,28) (26,27) (21,27) (33,27) (16,22) (23,22)  
V: (28,28) (27,28) (22,28)

T: (34,28) (26,27) (21,27) (33,27) (16,22) (23,22) (30,29) (35,29)  
V: (28,28) (27,28) (22,28) (29,28)

T: (26,27) (21,27) (33,27) (16,22) (23,22) (30,29) (35,29)  
V: (28,28) (27,28) (22,28) (29,28) (34,28)
One of the neighbors of 10 is 11 and hence we exit the loop. To reconstruct the path we just follow the parent pointers found in the VisitedArray. 11 -> 10 -> 16 -> 22 -> 28
Did that seem like a lot of tracing? It was, and that is how much computation this search algorithm had to do! The algorithm visited every cell except for cells \{1, 2, 3, 5, 6, 7, 8, 12, 13, and 18\}. If you go through the example with the obstacles you will see the algorithm visits every cell except for cell 12!

This means the algorithm will be SLOW. If you want to run this algorithm on every frame entry your game would likely die.

Given that we wanted to go from cell 28 to 11, it really made no sense to look at the cells to the left and below 28. The next algorithm improves on breadth first search by first visiting cells in the direction of the goal.

**Directed Depth First Search**

This algorithm is identical to above except the cells are added to a PRIORITY queue instead of a regular queue. In a priority queue elements no longer are added to the end. Instead, the elements are added in the correct sorted position. In this case we give highest priority (closest to the front of the queue) to cells that are closest to the goal. Our metric for closest is the Euclidean distance from a cell to the goal, thus, the closest is the one with the shortest distance and the priority queue is ordered from least distance to greatest distance.

**Algorithm:**

1. Add the starting cell to the `ToVisitPriorityQueue`
2. Repeat until goal reached or the `ToVisitPriorityQueue` is empty:
   a. \( N = \) First cell from `ToVisitPriorityQueue`. If \( N \) is the goal, exit the loop.
   b. Add \( N \) to the `VisitedArray`
   c. For each of the \( N \)'s 4 adjacent squares, denoted \( \{ a_1, \ldots, a_4 \} \):
      i. if \( a_i \) is the goal, exit the repeat loop
      ii. If \( a_i \) is not in the `ToVisitPriorityQueue` and also not in the `VisitedArray`, add \( a_i \) to the `ToVisitPriorityQueue` and set \( a_i.parent \) to \( N \)

If you wanted to allow for diagonal movement then you would need to consider each of the 8 adjacent square instead of only 4.

3. If the `ToVisitQueue` is empty and the goal is not found, there is no path. Otherwise, create the path by following the chain of parent pointers backwards from the goal node just removed.

Lets trace through this algorithm. First (28,28) is added. Then (28,28) is removed and the four neighbors are added to the ToBeVisitedPriorityQueue as follows (assuming the frontmost element is the highest priority):
Next we remove 22 and add in priority order each of it’s neighbors. Note, that each new neighbor is added in sorted order among the whole queue, not just from the 4 neighbors:

T: (16,22) (23,22) (29,28) (21,22) (27,28) (34,28) 
V: (28,28) (22,28)

Next we pull off (16,22) and add neighbors to get:

T: (10,16) (17,16) (15,16) (23,22) (29,28) (21,22) (27,28) (34,28) 
V: (28,28) (22,28) (16,22)

One the next iteration we pull off (10,16) and find the goal! Compare this amount of work to the previous example.

It might be tempting to add JUST the closest neighbor and forget the rest. The problem with that approach is what happens if you include obstacles. In the example above, we would visit in this order: 28, 22, 23, and then we are stuck! None of cell 23’s unvisited neighbors are closer to 11 then itself and we do not find a path, but a path does exist. If you go through the example with these obstacles, you will see that by putting all of the neighbors into the priority queue we have the ability to move back to a less optimal choice based on distance that infact is the best choice once the obstacles are taken into consideration.

Go ahead and run through the example above. The algorithm should visit 28, 22, 23,21,15,9,3,4,5,11. Again, this is MUCH better than the breadth first search.

A Star Algorithm

The directed depth first search algorithm above is a huge improvement but still has some shortcomings. The main one being that it is not easy to generalize to tiles that have different costs of movement. For example, lets say you have water, grass, pavement, forests, and mountain tile types. Moving through these different types has different costs and hence the path that follows the shortest Euclidean distance may not be the fastest. The A star algorithm allows this to be taken into consideration. Assume the following:

Cell Structure:

- (i,j): The coordinates of the cell
- g: The actual cost to get from the start to this cell
• h: An estimate of the cost from this cell to the goal. Note, for the algorithm to guarantee finding the optimal solution this estimate cost must be less or equal to the actual cost.
• f: The sum of g and h (i.e. g+h)
• parent: The cell (i,j) from which this node was reached

Data Structures:

• **Open**: A priority queue of cells with highest priority given to the cell with the lowest F cost. This is the list of cells that could potentially be on the path
• **Closed**: An array of cells that have already been “expanded”, meaning all of the cell’s neighbors have been previously added to the open list

Functions:

• **MC(A,B)**: The cost of moving from cell A to cell B. This function is only used when A,B are adjacent cells. We will assume the cost is 1 for left/right up/down neighbors. If you wanted to allow for diagonal movement then we would assume 1.41 for diagonal neighbors. This function can be modified for different tile types (water, rock, etc)
• **EST(A,G)**: The estimated cost of moving from cell A to the goal G. We will assume this function is the euclidian distance between the cells calculated by treating their (i,j) coordinates as integer locations of the cells.

Algorithm:

1) Add the starting cell to the open list
2) Repeat until goal reached or the open list is empty:
   A. N = First cell from open (the one with the lowest f-value). If N is the goal, exit the loop.
   B. Add N to the closed list
   C. For each of the N’s 4 adjacent squares, denoted {a1 … a4}:
       III. If entry into a_i is not allowed ignore a_i
       IV. If a_i is in the closed list ignore a_i
       V. If neither (2.a or 2.b) and a_i is not in the open list, then :
           • Set a_i.g = N.g plus the cost of moving from N to a_i
           • Set a_i.h to EST(a_i,G)
           • Set a_i.f to (a_i.g + a_i.h)
           • Set a_i.parent to N
       VI. If neither (2.a or 2.b) and a_i is in the open list, then :
           • If a_i.g (from the cell in the open list) > (N.g + MC(N,a_i) ) then change a_i.g to (N.g + MC(N,a_i)), recalculate a_i.f and resort the open list

3) If the open list is empty and the goal is not found, there is no path. Otherwise, create the path by following the chain of parent pointers backwards from the goal node just removed.

If you look at the line 2.C.IV you will see a cool part of this algorithm. As you saw when we traced through the breadth first search and depth first algorithms tiles are sometimes visited more than once. It is possible that on the second visit the path that lead
to that tile the second time is cheaper than the first path. The A-star algorithm allows you to then replace getting to this tile with the cheaper path.

Exercises:

1. Come up with an example where A-star yields a better result than directed depth first search.
2. Implement the three search algorithms in actionscript and compare the performance.