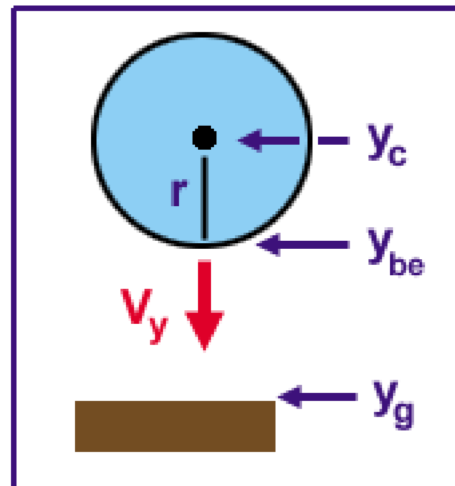
Bouncing Ball Part II
(The Ground)

Prepared by: Teton Multimedia (www.TetonMultimedia.com)

A ball falls toward the ground. It hits the ground and bounces back into the air. Question: "How can the bounce be represented mathematically, and how can we use this math within a computer program?"

As we noted in "Bouncing Ball Part I (Gravity)," gravity causes a falling ball to accelerate toward the ground. The acceleration is orderly and predictable, and can be modeled mathematically with relative ease. However, at some point, the falling ball must strike the ground and bounce upward. Assuming that the bounce is completely elastic, then the math of the bounce *per se* is not relevant. Instead, the *timing of impact* is our primary concern.

The picture to the right describes the situation. The ball's center is at an elevation of y_c , it's radius (distance from center to edge) is r, the elevation at the bottom edge is y_{be} , the elevation of the ground is y_g , and the ball's velocity is v_y. Note also that the velocity is *not* constant (it is increasing).



Note: using Director's screen orientation, y_g is actually *greater than* y_{be}. This is because Director's zero is at the top of the screen. So as the ball falls, it's mathematical elevation will actually *increase*. (This may seem odd… but so it goes.)

Impact occurs when the bottom surface of the ball (y_{be}) strikes the ground surface (y_g). Restated in equation form, impact occurs when $y_{be} = y_g$.The better we are at predicting *when* this occurs, the better our results will be. However, it should be noted that we will *never* predict the timing exactly due to one or more of

1. inaccuracies in our prediction
2. round-off error
3. intervention

Round-off error results from working with a finite number of decimal places. So, for example, we may know that $y_g = 3.000000000000$ feet exactly. Meanwhile, if $y_{be} = 3.000000000001$ feet, has a collision occurred? Answer: "Yes, or at least close enough." In fact, the ball's bottom surface is slightly *below* the ground. This implies that our collision prediction isn't looking for an exact equality. Instead, it is looking for a range of acceptable proximity.

By "intervention" we mean simply that another object, such as a baseball bat, may strike the ball an instant before it hits the ground, lofting it upward. For a moment we *thought*

we knew when the ball would hit the ground, but then something intervened to upset our prediction.

For these reasons, it is easiest to treat collision prediction instead as “overlap detection.” When two surfaces overlap, then a collision most certainly has occurred. Provided that they don’t overlap very far, then we can “pretend” that they only bumped into each other. The human eye won’t notice the overlap, and instead will simply see the ball bounce off the ground.

In this example, overlap occurs when

1.1

$$| y_g - y_c | \leq r$$

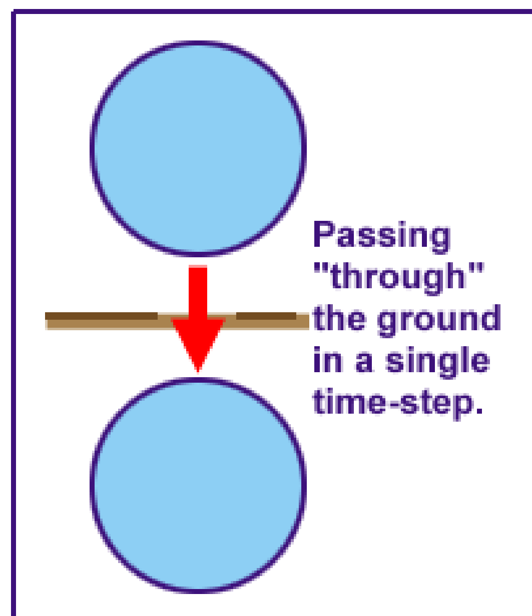
By introducing the absolute value operator, the “up Vs. down” orientation becomes irrelevant. Whenever the distance from the ball’s center to the ground is less than the ball’s radius, then the ball *must be* contacting the ground. This fact is generally useful when detecting overlap between a round ball and virtually any surface (such as a ceiling or a wall).

So what happens when the collision occurs? In this example, with a perfectly elastic collision, we simply change the sign of the velocity. If the ball was moving at 30 ft/sec before the collision, then its velocity after the collision is –30 ft/sec.

1.2

$$v_a = -1 * v_b$$

Wow! This is easy! But wait, now the hard part. If the ball travels too far during one time-step, then it may pass completely through the ground’s surface in a single time-step, and equation 1.1 will never be satisfied. This picture shows what can happen. One time-step the ball is completely above the surface, and the next it is completely below. It never actually overlaps the ground surface, and so a collision is not detected. Consequently, if we used equation 1.1 blindly in this situation, the ball would simply pass through the ground and keep right on going.



To resolve this problem, we must reduce the time-step size so that the ball cannot travel too far during a single time-step. A “good” time-step size is one that prevents the ball from traveling more than ½ radius during the time-step. Each time-step, the ball’s motion is approximately $\Delta y = v * \Delta t$. We want to limit Δy to $r / 2$. This gives us the inequality

1.3a

$$r/2 > v * \Delta t$$

or

1.3b

$$\Delta t < 2 * v / r$$

Believe it or not, the equations above are sufficient to represent the bouncing ball. However, there are some programming “gotcha’s” to be aware of. For example, because Δt must be *less than* a particular value, the FrameTempo must be *greater than* the value’s reciprocal. This gives us the equation

1.4

$$\text{FrameTempo} > r / (2 * v)$$

Notice that in this equation, we are dividing by v. Because v *can be* zero, our programming must skip this particular calculation whenever v is zero (or near-zero).

Another point to consider is that we want our bouncing ball to animate smoothly, which itself implies a minimum FrameRate of around 30 frames per second (fps). Consequently, the tempo calculated via equation 1.4 must be compared against 30, and the *maximum* value must be used.

Lastly, although it is implied that Director doesn’t let us calculate the FrameTempo on the fly, this can actually be accomplished via PuppetTempo. For simplicity, we can instead select a single value to use throughout. This implies that we must guess, calculate, or impose the maximum velocity expected during our animation, and the time-step size must be selected as appropriate for this velocity. Equation 1.4 then becomes

1.5

$$\text{FrameTempo} > r / (2 * v_{max})$$

Lingo!

The above collision logic can be implemented within the On ExitFrame handler of the behavior script shown in “Bouncing Ball Part I (Gravity).” Note that the FrameTempo must be calculated manually and specified in the tempo channel. The programming is as follows.

```
1.    r = 0.5 * MyS.Height          -- get r
2.    yg = 400                      -- define the ground elevation
3.    if abs(yg – ye) <= r then     -- equation 1.1
4.       vi = -1.0 * vi             -- equation 1.2
5.    end if
```

Line 1 assumes that the round ball fills the sprite exactly; consequently, the radius is half of the sprite height. Note that line 2 defines the ground elevation for us. Alternatively, the ground elevation can be taken from a sprite that represents the ground.

The accompanying Director movie on the Teton Multimedia website shows the programming in action. You'll note that the FrameRate is set to 120 fps. This produces a very smooth animation. We suggest that you try reducing the FrameRate to observe the result. You'll see noticeable overlap at a frame rate of 40, and when the frame rate is 10, the ball falls through the ground and keeps right on going.